

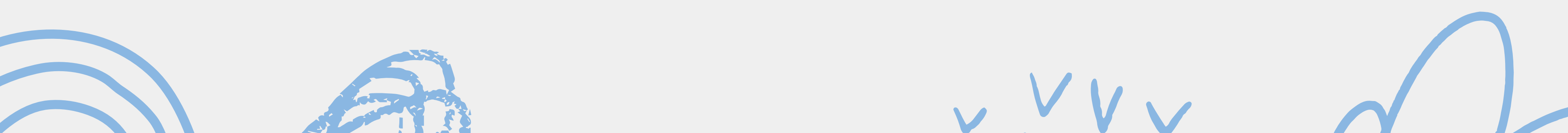


# Ayudantia 1

**Insertion y Selection Sort**



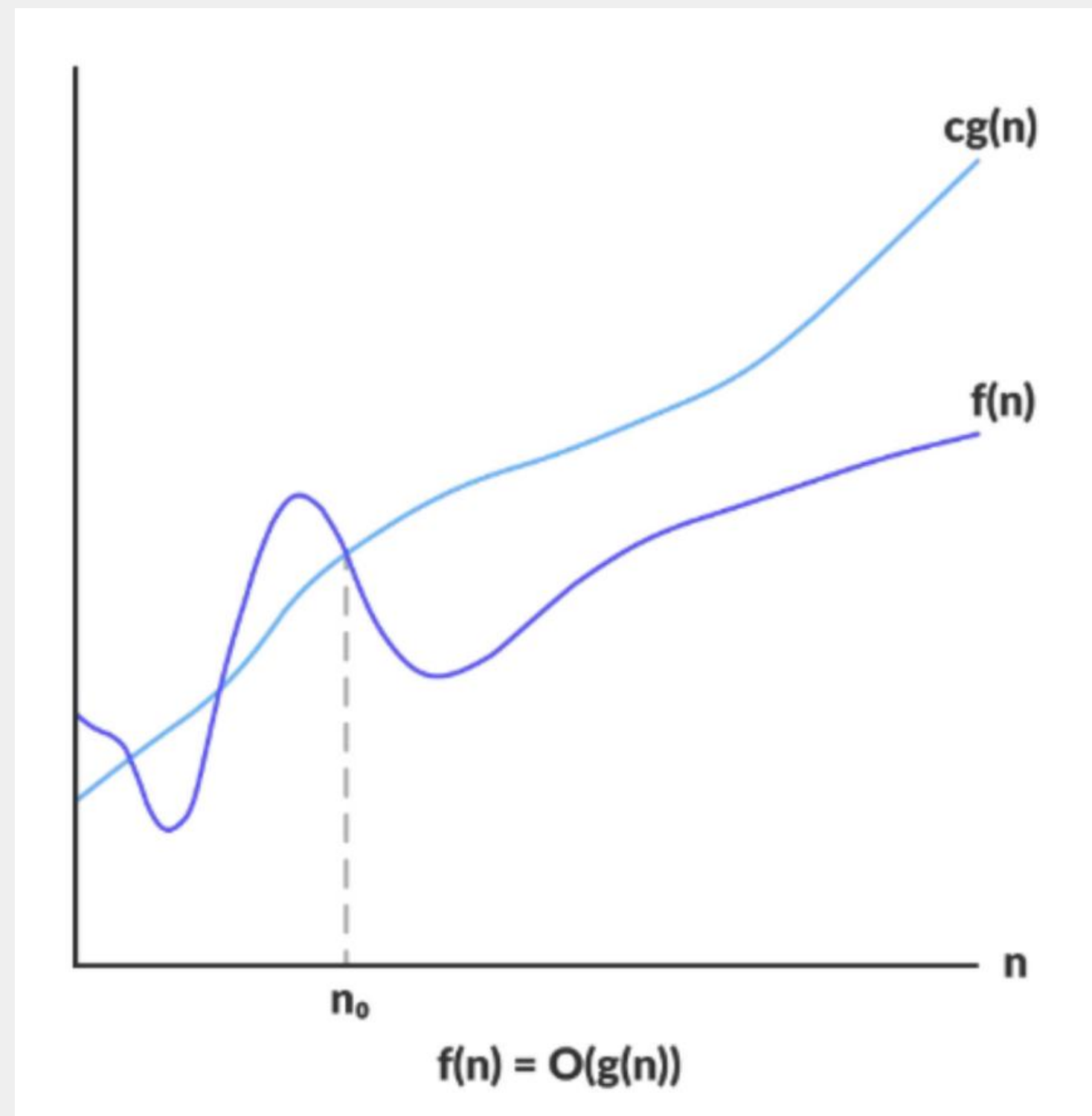
# Notacion asintotica

1. Determinar tiempos de respuesta de nuestro algoritmo
  2. Determinar recursos computacionales
  3. Ver escalabilidad de nuestra función
  4. Elegir algoritmos de forma eficiente
- 

# Notacion O

Si un tiempo de ejecución es de  $O(g(n))$ , entonces para  $n$  suficientemente grande, el tiempo de ejecución es a lo más  $c \cdot g(n)$  para alguna constante  $c$

Es la que mas vamos a utilizar :D






# Correctitud

Un algoritmo es correcto si siempre se cumple que:

1. El algoritmo termina en tiempo finito
2. Se obtiene el resultado esperado (ej: ordenar)

La correctitud se suele demostrar mediante inducción porque es compatible con problemas de tamaño crecientes



# Intro a Sorting

# Selection Sort

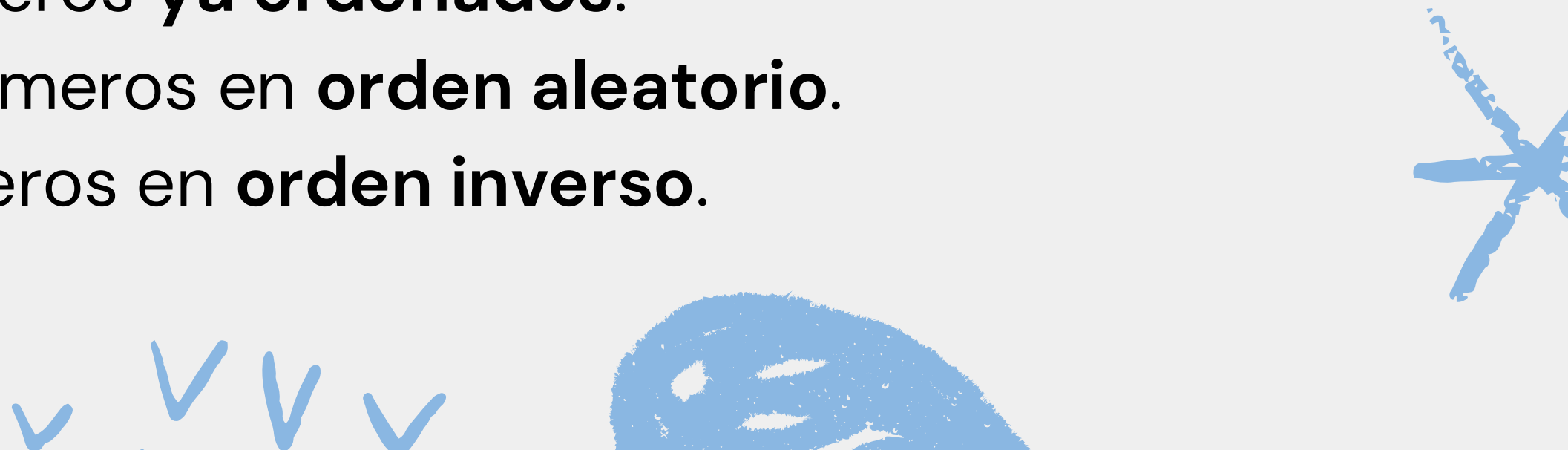
1. Tenemos una secuencia desordenada
2. Iniciar en posición 0
3. Buscar el menor dato 'x' en la secuencia
4. Intercambiar ese elemento 'x' con el elemento actual de la secuencia
5. Avanzar uno en la secuencia
6. Si aún queda secuencia, volver al paso 2



# Problema

Tienes a tu disposición 2 ejecutables compilados **E1** y **E2**, correspondientes a 2 algoritmos de ordenación: InsertionSort SelectionSort. El problema es que no sabes cuál archivo corresponde a qué algoritmo.

Para determinarlo, se crean tres archivos con 1 millón de números para ordenar:

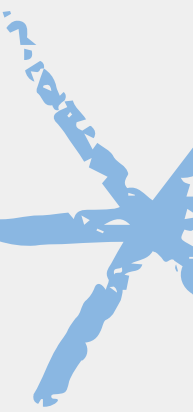
1. El primer archivo contiene números **ya ordenados**.
  2. El segundo archivo contiene números en **orden aleatorio**.
  3. El tercer archivo contiene números en **orden inverso**.
- 



# Problema

Al ejecutar cada programa con cada conjunto de números, se obtiene la siguiente tabla de tiempos:

Archivo	Datos ordenados	Datos aleatorios	Datos al revés
E1	7.35s	25.47s	46.11s
E2	44.88s	48.72s	43.35s



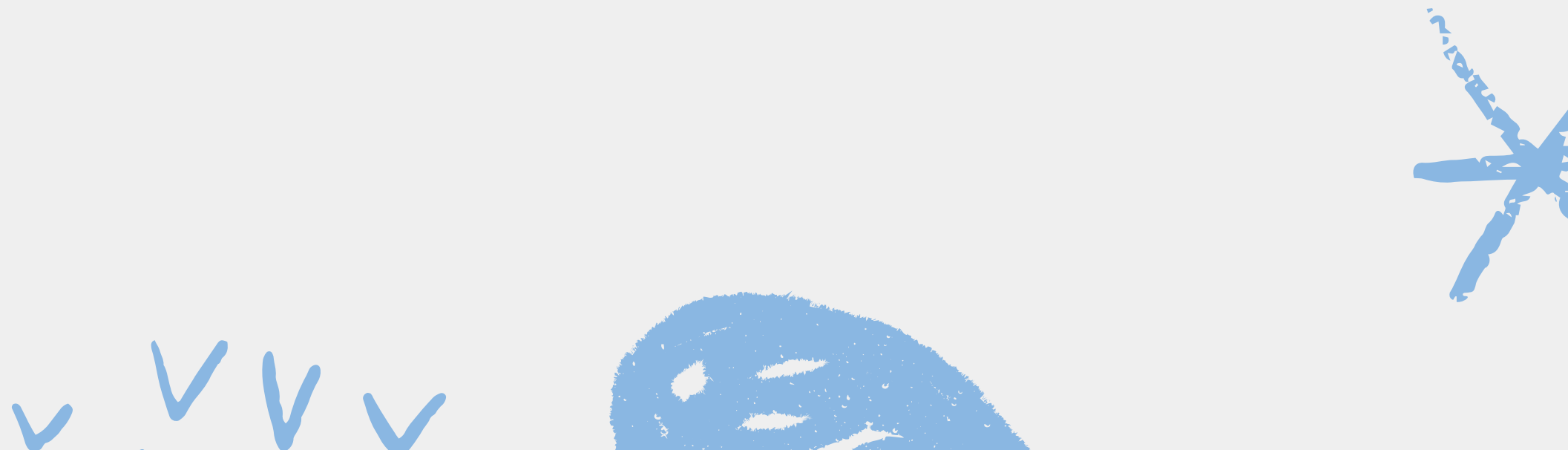




# Solucion

Recordemos que:

Algoritmo	Mejor caso	Caso promedio	Peor caso	Memoria adicional
Selection Sort	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$
Insertion Sort	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$





# Solucion

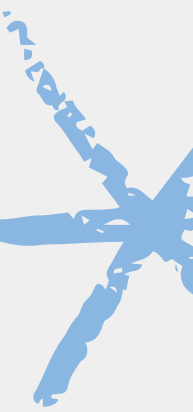
Algoritmo	Mejor caso	Caso promedio	Peor caso	Memoria adicional
Selection Sort	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$
Insertion Sort	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$

Sabemos que:

**Selection sort** tiene complejidad  $\mathcal{O}(n^2)$  independiente de como esten ordenados los datos.

Por otro lado **Insertion sort**:

- Mejor caso  $\mathcal{O}(n)$ : este ocurre cuando los datos ya estan ordenados.
- Peor caso  $\mathcal{O}(n^2)$ : este ocurre cuando los datos estan en orden inverso



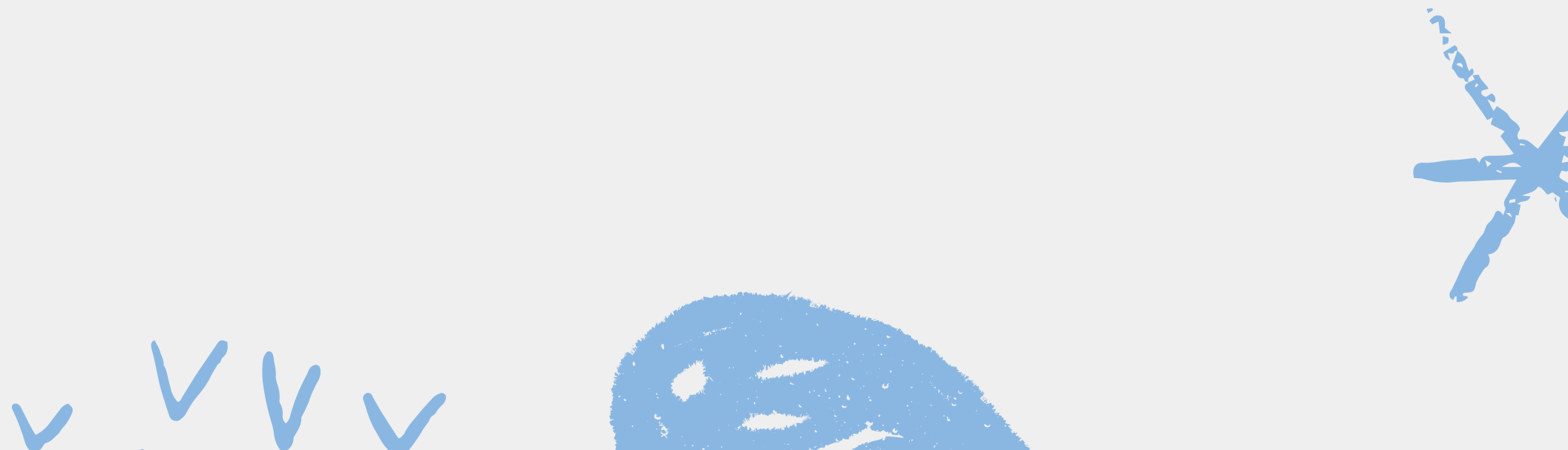


# Solucion

Finalmente,

**E1 -> Insertion Sort:** Muestra tiempos bajos con datos ordenados, pero se deteriora con datos aleatorios y en orden inverso, confirmando su comportamiento característico.

**E2 -> Selection Sort:** Mantiene tiempos consistentemente altos, independiente del orden de los datos, lo que corresponde a su complejidad  $O(n^2)$ .



# Problema

Bob ha cocinado pancakes y los dejó apilados sobre la mesa, formando una pila  $P[0..n-1]$  tal que  $P[0]$  almacena el pancake que está en la cima de la pila, mientras que  $P[n-1]$  representa el que está en la base. Los pancakes tienen un diámetro representado por un número real. Alice ve la pila y decide ordenarla de tal manera que el diámetro de los pancakes sea descendente desde la base de la pila hacia la cima. Para realizar la tarea, suponga que cuenta con las siguientes funciones:

- **Diámetro( $P[i]$ ):** devuelve el diámetro del pancake  $P[i]$ , para  $0 \leq i \leq n-1$ . Asuma que todos los diámetros son distintos entre sí.
- **Invertir( $i$ ):** invierte  $P[0..i]$ , para  $0 \leq i \leq n-1$ . Esta función simula la acción de colocar una espátula debajo del pancake  $P[i]$ , para luego invertir el orden de los elementos de la pila  $P[1..i]$  que queda por encima de la espátula. Asuma que este proceso es in-place. Por ejemplo, para la pila  $P[0..5] = \langle 3.2, 4.1, 2.8, 7.3, 6.1, 1.3 \rangle$ , **Invertir(3)** produce como resultado la pila  $\langle 7.3, 2.8, 4.1, 3.2, 6.1, 1.3 \rangle$ .

- (a) (3 puntos) Escriba el algoritmo **PancakeSort( $P[0..n-1]$ )**, que permita ordenar la pila de pancakes  $P[0..n-1]$  como se indica. La única manera permitida de manipular los pancakes de la pila es a través de las funciones **Diámetro** e **Invertir**. Asuma que dichas funciones ya están implementadas. Si necesita emplear otra función distinta a esas, debe implementarla. Hay muy poco espacio disponible en la mesa, por lo que su algoritmo debe ser *in-place*.



# Solucion

- (a) El algoritmo es similar a Selectionsort, salvo por las restricciones que impone este ejercicio. En cada iteración se debe buscar el siguiente pancake de mayor diámetro, para trasladarlo hacia la parte inferior de la pila usando la función Invertir. El pseudocódigo de una posible solución es el siguiente:

**input** : Una pila de pancakes  $P[0..n - 1]$   
**output**: La pila de pancakes  $P$ , ordenada

```
PancakeSort( $P[0..n - 1]$ )  
for  $i \leftarrow n - 1$  downto 1 :  
    |  $m \leftarrow \text{MÁXIMO}(P[0..i])$   
    | Invertir( $m$ )  
    | Invertir( $i$ )  
return  $P$ 
```



# Problema

(b) (2 puntos) Demuestre formalmente la correctitud de su algoritmo.



# Solucion

(b) La demostración de correctitud del algoritmo debe mostrar que:

- PancakeSort termina en una cantidad finita de pasos (**0.5 pts.**).
- PancakeSort cumple su propósito (**1.5 pts.**).

Para demostrar la finitud del algoritmo, note que realiza siempre  $n - 1$  iteraciones. En cada iteración, busca el pancake de mayor diámetro, lo cual requiere la invocación al algoritmo MÁXIMO( $P[0..i]$ ) (que finaliza en  $i - 1$  iteraciones), para luego invocar dos veces a la operación Invertir. En otras palabras, cada una de las operaciones que se realizan en cada iteración de PancakeSort son finitas, y por lo tanto el algoritmo finaliza en una cantidad finita de pasos.

# Solucion

La demostración de que PancakeSort cumple su propósito es por inducción, similar a la demostración respectiva hecha en clases para SelectionSort. La propiedad que demostraremos es la siguiente:

- $\mathbb{P}(i)$ : al finalizar la iteración  $i$  (para  $1 \leq i \leq n$ ) de PancakeSort,  $P[n - i..n - 1]$  está ordenada y contiene los  $i$  pancakes de mayor diámetro.

**Caso Base:** el caso base es para  $i = 1$  (es decir, la primera iteración). Al finalizar la primera iteración,  $\mathbb{P}(1)$  se cumple ya que el elemento de mayor diámetro es movido a la base  $P[n - 1]$  de la pila.

**Hipótesis Inductiva:** Al finalizar la iteración  $i$ , se cumple  $\mathbb{P}(i)$ .

**Paso Inductivo:** Demostramos ahora que  $\mathbb{P}(i + 1)$  se cumple, dado que  $\mathbb{P}(i)$  (la H.I.) se cumple. Esto último significa que  $P[n - i..n - 1]$  está ordenada y contiene los  $i$  pancakes de mayor diámetro, como hemos dicho, y por lo tanto  $P[0..n - i - 1]$  contiene los  $n - i$  pancakes de menor diámetro. De entre esos pancakes de menor diámetro se selecciona aquel que tiene mayor diámetro, el cual es movido a la posición  $P[n - (i + 1)]$  de la pila. Eso significa que ahora  $P[n - (i + 1)..n - 1]$  está ordenada y contiene los  $i + 1$  pancakes de mayor diámetro, por lo que  $\mathbb{P}(i + 1)$  se cumple.



# Insertion Sort

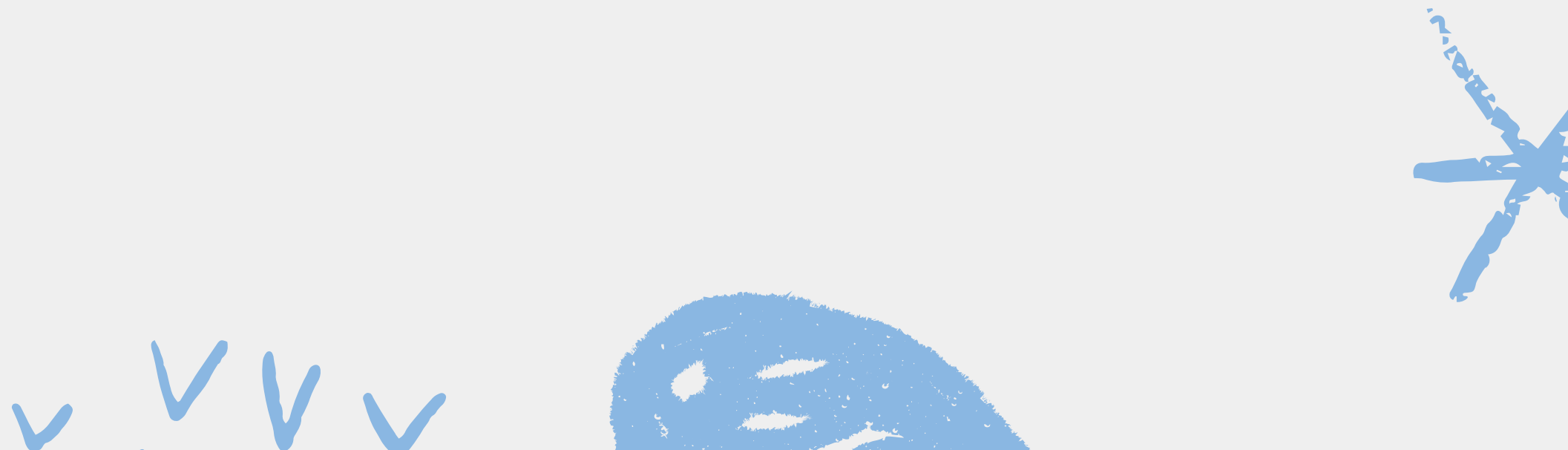
1. Tenemos una secuencia desordenada
2. Tomar el primer dato 'x' de la secuencia
3. Insertar 'x' en los elementos anteriores de manera que quede ordenado
4. Avanzar en la secuencia
5. Si aún queda secuencia, volver al paso 2



# Problema

Considera el concepto de inversión que vimos cuando estudiamos el algoritmo **insertionSort**: Los índices  $i$  y  $j$  son una inversión en el arreglo  $r$  si  $i < j$  pero  $r[i] > r[j]$ .

a) Escribe un algoritmo basado en (o similar a) **insertionSort** que cuente una a una el número de inversiones de un arreglo  $r$



# Solucion

insertionSort corrige una a una cada una de las inversiones de  $r$ , esto es, cada vez que ejecuta un intercambio corrige una inversión.

Por lo tanto, para contar una a una el número de inversiones, basándonos en insertionSort, basta con contar uno a uno los intercambios que realiza insertionSort.

```
int inversionCount( $r, n$ ):  
    int count = 0  
    for  $i = 1 \dots n - 1$  :  
         $j = i$   
        while ( $j > 0$ )  $\wedge$  ( $r[j] < r[j - 1]$ ) :  
            intercambiar  $r[j]$  con  $r[j - 1]$   
            count = count + 1  
             $j = j - 1$   
    return count
```



# Problema

b) Determina la complejidad de tiempo —en notación  $O()$ — del peor caso de tu algoritmo de a)



# Solucion

```
int inversionCount(r, n):  
    int count = 0  
    for i = 1 . . . n - 1 :  
        j = i  
        while (j > 0)  $\wedge$  (r[j] < r[j - 1]) :  
            intercambiar r[j] con r[j - 1]  
            count = count + 1  
            j = j - 1  
    return count
```

*inversionCount* realiza básicamente las mismas operaciones que *insertionSort*, excepto que incrementa un contador *count* cada vez que ejecuta un intercambio. Por lo tanto, tiene la misma complejidad que *insertionSort*:  $O(n^2)$  en el peor caso

# Feedback

