Ordenamiento por selección (Selection Sort) en Python — explicación y ejemplo línea por línea

El **algoritmo de ordenamiento por selección** ordena una lista encontrando repetidamente el elemento mínimo (o máximo) de la porción no ordenada y colocándolo al principio de esa porción. Es simple, in-place (no usa memoria extra significativa) y tiene complejidad O(n2)O(n^2)O(n2). No es estable (puede cambiar el orden relativo de elementos iguales) y se usa principalmente por su simplicidad pedagógica.

Aquí tienes un ejemplo claro en Python, seguido de la explicación línea por línea.

```
def selection_sort(arr):
    """
    Ordena la lista arr in-place usando el algoritmo de selección.
    Devuelve la misma lista ordenada (aunque la lista original se modifica).
    """
    n = len(arr)                        # 1
    for i in range(n - 1):              # 2
        min_index = i                   # 3
        for j in range(i + 1, n):       # 4
            if arr[j] < arr[min_index]:# 5
                min_index = j           # 6
        # si min_index cambió, intercambiamos arr[i] y arr[min_index]
        if min_index != i:              # 7
            arr[i], arr[min_index] = arr[min_index], arr[i]  # 8
    return arr                          # 9

# Ejemplo de uso:
lista = [64, 25, 12, 22, 11]
print("Antes:", lista)
print("Después:", selection_sort(lista))
```

Explicación línea por línea

1. `n = len(arr)`
   Calcula la longitud de la lista y la guarda en `n`. La usaremos para saber hasta dónde iterar.
2. `for i in range(n - 1):`
   Recorremos índices `i` desde `0` hasta `n-2`. En cada iteración fijamos la posición `i` donde colocaremos el siguiente elemento más pequeño. No hace falta ir hasta `n-1` porque al llegar al penúltimo elemento el último ya estará automáticamente en su lugar.
3. `min_index = i`
   Suponemos inicialmente que el mínimo de la porción no ordenada (desde `i` hasta `n-1`) está en la posición `i`.
4. `for j in range(i + 1, n):`
   Recorremos las posiciones posteriores a `i` para buscar si existe un elemento menor que el actual supuesto mínimo. `j` va de `i+1` hasta `n-1`.
5. `if arr[j] < arr[min_index]:`
   Comparamos el elemento en `j` con el elemento en la posición `min_index`. Si `arr[j]` es menor, encontramos un nuevo mínimo.
6. `min_index = j`
   Actualizamos `min_index` para apuntar al índice del nuevo menor encontrado.
7. `if min_index != i:`
   Al terminar el bucle interno, si `min_index` cambió (es decir, encontramos un elemento menor que el de `i`), entonces procedemos a intercambiar. Si no cambió, la posición `i` ya tenía el mínimo y no necesitamos intercambio.

8. `arr[i], arr[min_index] = arr[min_index], arr[i]`
   Intercambiamos los valores: colocamos el mínimo encontrado en la posición `i` y movemos el valor antiguo de `i` al lugar del mínimo. Este intercambio es in-place (no crea una nueva lista).
9. `return arr`
   Devolvemos la lista ya ordenada. (Nota: la lista original `arr` fue modificada en el proceso.)

Ejecución del ejemplo

Con `lista = [64, 25, 12, 22, 11]`:

- Iteración i=0: encuentra 11 como mínimo → intercambia posición 0 y la del 11 → lista pasa a `[11, 25, 12, 22, 64]`.
- i=1: encuentra 12 como mínimo entre los restantes → intercambia → `[11, 12, 25, 22, 64]`.
- i=2: encuentra 22 como mínimo → intercambia → `[11, 12, 22, 25, 64]`.
- i=3: ya el menor es 25, intercambia si hace falta → lista ordenada `[11, 12, 22, 25, 64]`.

Complejidad y notas rápidas

- Tiempo: peor/medio/mejor caso O(n2)O(n^2)O(n2) (dos bucles anidados).
- Espacio: O(1)O(1)O(1) adicional (in-place).
- Estabilidad: **no estable** en la implementación básica (porque al intercambiar se puede cambiar el orden relativo de elementos iguales).
- Uso típico: educativo o cuando `n` es pequeño y la simplicidad importa.