Universidad Nacional de General Sarmiento

*Licenciatura en Sistemas*

# Trabajo Práctico

# Visualización de algoritmos de ordenamiento

Introducción a la Programación

(Segundo semestre 2025)

**Resumen:** Este trabajo consiste en, a través de un visualizador HTML, poder observar cómo es que se llevan a cabo los distintos métodos de ordenamiento propuestos y desarrollados (con ayuda de IA) en VSCode. Lo realizado por nuestro grupo nos permite tener una gran noción de cómo es que funcionan todos estos algoritmos, los cuales son *"Bubble Sort"*, *"Selection Sort"*, *"Insertion Sort"*, *"Quick Sort"*, *"Merge Sort"* y *"Shell Sort"*, incluyendo también un apartado para la implementación de métricas que nos ayudan a contabilizar el tiempo, los pasos, los swaps y las comparaciones.

Integrantes:

- Fournier, Gabriel Fernando / gabrielferfour@gmail.com
- González Ferrere, Bautista / gonzalezferrerebautista@gmail.com
- Villalva, Juan Pablo / juanpablovillalva2@gmail.com

## 1. *Introducción*

Este trabajo tiene como finalidad implementar varios algoritmos de ordenamiento y visualizar su ejecución paso a paso mediante una interfaz web. Cada algoritmo debía respetar el contrato “init(vals) + step()”, donde cada llamada a “step()” representa un micro paso del proceso del ordenamiento.

El objetivo principal es comprender de manera práctica cómo trabajan internamente distintos métodos de ordenamiento, desde los clásicos de complejidad básica (Bubble Sort, Selection Sort e Insertion Sort) hasta otros más avanzados, como Merge Sort, Quick Sort y Shell Sort, que se decidió implementar como extensión. Además, se realizaron modificaciones visuales en la interfaz y se incorporó un sistema de métricas que muestra tiempo de ejecución, pasos, swaps y comparaciones.

## 2. *Desarrollo*

Para resolver el problema fue necesario adaptar los algoritmos de ordenamiento a un modelo de ejecución basado en *“*micro-pasos*”*. Cada llamado a step() debía realizar una única acción simple, como comparar dos valores, mover un puntero o efectuar un intercambio adyacente. Esto requirió descomponer algoritmos completos, que tradicionalmente operan con ciclos grandes, en unidades mínimas.

Una de las principales dificultades fue que el visualizador sólo permite swaps entre elementos consecutivos. Esto implicó modificar algoritmos como Merge Sort, aplicando un mecanismo de “shifting” mediante swaps adyacentes para desplazar elementos hacia su posición correcta. También fue necesario reescribir Quick Sort en una versión iterativa usando una pila, ya que la recursión no se adecúa a la ejecución fragmentada requerida.

Otro desafío fue mantener el estado interno entre llamadas a step(). Para ello se utilizaron punteros auxiliares y pequeñas máquinas de estado que permiten reanudar el algoritmo exactamente en el punto en que había quedado.

Además de cumplir los requisitos, se incorporaron mejoras visuales y un sistema de métricas que contabiliza comparaciones, swaps y tiempo de ejecución, lo que permitió observar diferencias concretas entre métodos.

## *Bubble Sort*

El algoritmo del Bubble Sort lo que hace en cada paso es comparar un par de elementos adyacentes, representados por los índices **a** y **b**, y si están en el orden incorrecto los intercambia, viéndose como un efecto donde el mayor elemento de la pasada es llevado al final y queda “separado”. El control del proceso lo realizan dos punteros, **i** que indica cuantas vueltas hizo el algoritmo, y **j** que recorre los elementos en cada vuelta.

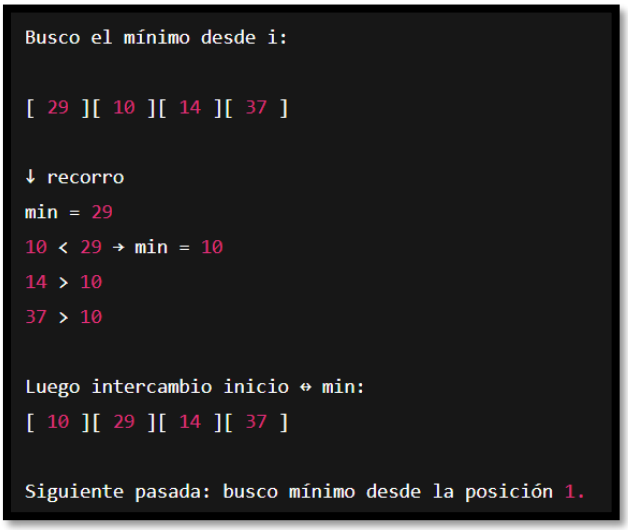
```
Primera pasada:
 [ 5 ][ 3 ][ 8 ][ 2 ]

Comparo 5 y 3 → swap
 [ 3 ][ 5 ][ 8 ][ 2 ]

Comparo 5 y 8 → no swap
 [ 3 ][ 5 ][ 8 ][ 2 ]

Comparo 8 y 2 → swap
 [ 3 ][ 5 ][ 2 ][ 8 ]
```

Cuando **j** llega al final de la pasada se reinicia a 0 e **i** aumenta una unidad. Entonces cuando **i** alcanza el largo de la lista menos 1, quiere decir que ya no hay más elementos para ordenar, por ende, la lista está ordenada y el código devuelve {“done”: True}.

```python
i = 0
j = 1
min_idx = 0
fase = "buscar"

def step():
    global i, j, min_idx, fase, items

    if i >= n - 1:
        return {"done": True}

    if fase == "buscar":
        if j < n:
            if items[j] < items[min_idx]:
                min_idx = j
            j += 1
            return {"a": min_idx, "b": j-1, "swap": False, "done": False}
        fase = "swap"

    if fase == "swap":
        if min_idx != i:
            items[i], items[min_idx] = items[min_idx], items[i]
        i += 1
        j = i + 1
        min_idx = i
        fase = "buscar"
        return {"a": i-1, "b": min_idx, "swap": True, "done": False}
```

## *Selection Sort*

El método Selection Sort está dividido en dos fases: ‘buscar’ y ‘swap’. En un inicio, la fase es la primera, y en ella el algoritmo busca el elemento más pequeño de la lista, mediante una variable **j** que recorre los elementos desde i+1 (inicialmente **i** = 0 y en cada vuelta se lo toma como referencia de mínimo) hasta el final comparándolos con el índice del mínimo actual (min_idx) y en caso de que sea menor este último se actualiza.

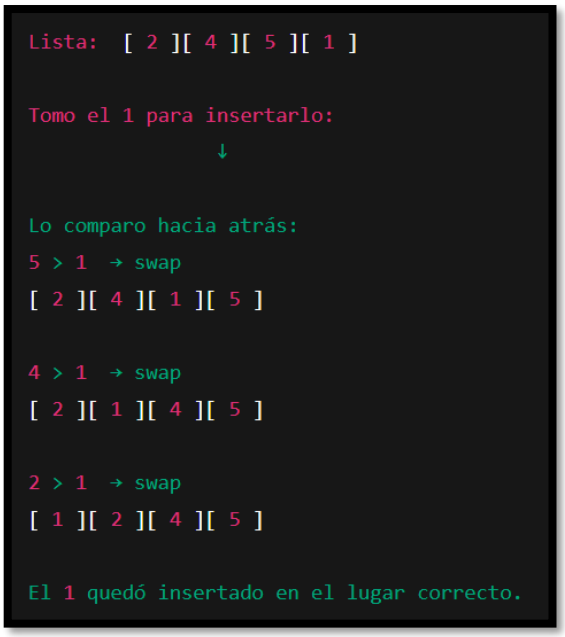



Cuando **j** llega al final de la lista, el algoritmo cambia de fase a “swap”. En esta, si el mínimo encontrado no está en la posición **i** (o sea, si no es el más chico de los recorridos), se realiza un swap entre el elemento de la posición **i** y el más pequeño.

```
i = 0
j = 1
min_idx = 0
fase = "buscar"

def step():
    global i, j, min_idx, fase, items

    if i >= n - 1:
        return {"done": True}

    if fase == "buscar":
        if j < n:
            if items[j] < items[min_idx]:
                min_idx = j
            j += 1
            return {"a": min_idx, "b": j-1, "swap": False, "done": False}
        fase = "swap"

    if fase == "swap":
        if min_idx != i:
            items[i], items[min_idx] = items[min_idx], items[i]
        i += 1
        j = i + 1
        min_idx = i
        fase = "buscar"
        return {"a": i-1, "b": min_idx, "swap": True, "done": False}
```

Luego de eso, se inicia una nueva vuelta por lo que **i** suma 1 unidad y se reposiciona j. El proceso termina cuando **i** alcanza el largo de la lista menos 1, que quiere decir que no hay más elementos para ordenar y por ende la lista está ordenada.

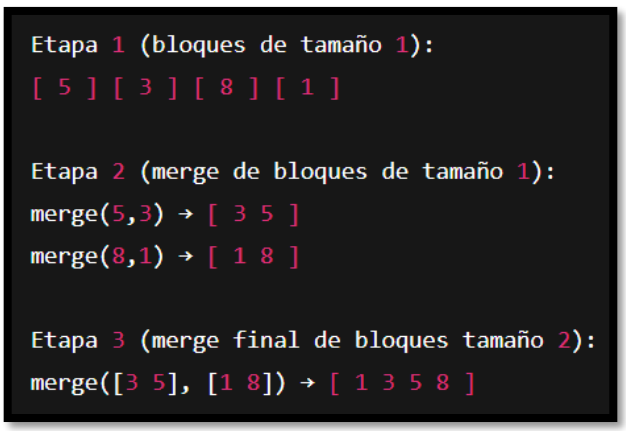
## *Insertion Sort*




El algoritmo del Insertion Sort recorre la lista a partir del segundo elemento e intenta "insertar" cada valor seleccionado en la posición correcta dentro de una parte ya ordenada a la izquierda. En este caso, la variable **i** es la que lleva el registro de en cual índice se inicia la vuelta (en un comienzo es 1) y la variable **j** funciona como un cursor que se mueve a la izquierda si el elemento actual es menor al anterior.

Entonces, cuando **j** no tiene al principio del código o luego de una pasada, toma el valor de **i**, y el código resalta el índice. Posterior a eso, compara al elemento con su anterior y si están desordenados realiza un swap, además de que **j** retrocede uno. Entonces el próximo paso es comparar nuevamente el actual **j** con su anterior así hasta que no haya swap, momento en el cual la variable **i** suma 1 y **j** se reinicia en **"None"**. El proceso termina cuando **i** alcanza el largo de la lista.

## *Marge Sort*

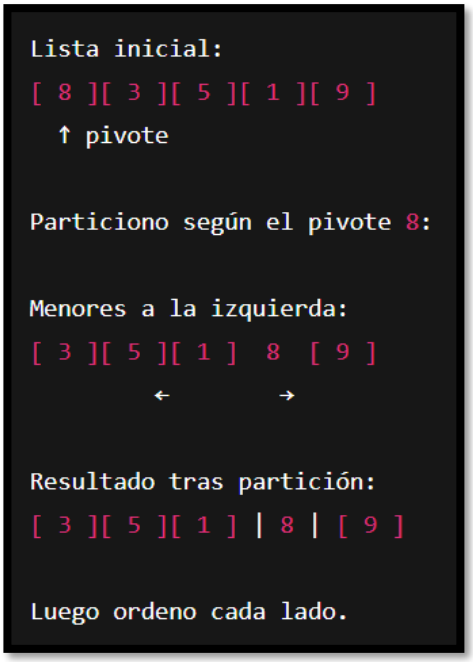



El método Merge Sort implementado es una adaptación realizada con ayuda de la IA para que funcione en el visualizador de forma que se vea correcta y muestre swaps uno por uno. El algoritmo trabaja de forma "bottom-up", o sea, considera bloques de tamaño 1, luego 2, luego 4… Y así hasta llegar al largo de la lista. Para cada bloque, define una zona izquierda, una media y una derecha. Dentro de cada zona, compara *ítems* [ **i** ] del bloque izquierdo e *ítems* [ **j** ] del derecho: Si el de la izquierda es menor, avanza i; si el del lado derecho es menor,

se pasa a la fase shifting.

La fase shifting toma el elemento en **j** y lo mueve hacia la izquierda swap por swap hasta llegar a la posición correcta **i**. Luego, el código actualiza los limites **(mid, j, i)** porque el primer sub-bloque se expandió en uno. Cuando ya se juntaron los dos bloques, avanza al siguiente, y cuando no hay más, duplica el tamaño de bloques **(width)** y repite todo hasta finalizar. De esta manera, simula el Merge clásico en el que se divide la lista muchas veces y se ordenan los elementos para luego unirse nuevamente ordenadas.

### *Quick Sort*



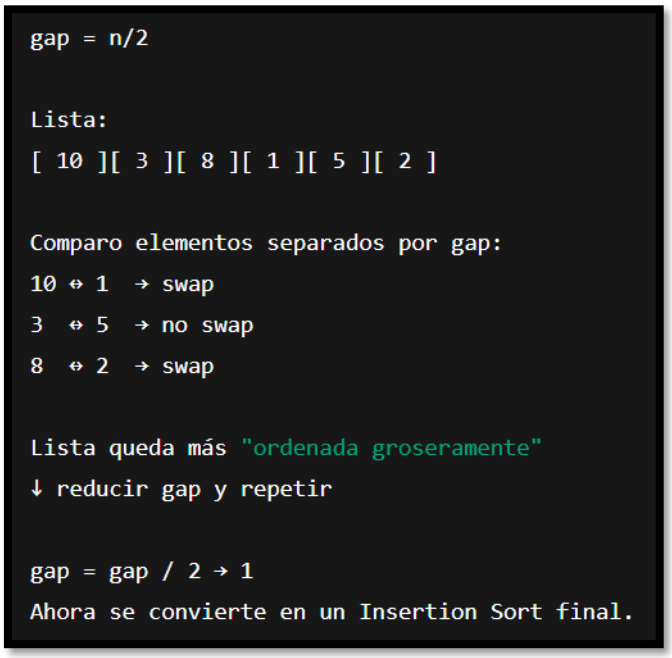

El método Quick Sort se implementó de forma iterativa con ayuda de la IA. El código utiliza una pila manual **(stack)** y está dividido en **"micro-fases"** para que el visualizador lo anime paso a paso. Este código no usa recursión como lo hace el Quick, si no que cada rango pendiente de ordenar se guarda como un par **(left, right)** en la pila. En cada paso, el algoritmo toma un rango, elige un pivote (el último elemento del rango) y arranca una partición, en donde dos cursores (i y j) buscan elementos mal ubicados respeto al pivote. Si **j** es más chico que el pivote y **i** más grande, se intercambian, dejando de un lado del pivote elementos más chicos que el mismo y de otro lado elementos más grandes. Todo eso se hace en distintas fases para que se vea en el visualizador.

Primero, la fase scan_left, el cursor **i** avanza mientras el elemento sea menor que el pivote, y en scan_right el cursor **j** retrocede mientras el elemento sea mayor o igual al pivote. Si ambos detectan elementos incorrectos, se hace un swap entre ellos. Cuando los cursores se encuentran, se pasa a la fase finish_partition donde se toma el pivote y lo coloca en su posición final intercambiándolo con ítems**[ i ]**. Una vez que queda el pivote ubicado, lo que quedo a su izquierda pasa a ser un subproblema, y lo que quedo a la derecha otro, que se apilan para procesarse después. El ciclo continua hasta que la pila queda vacía lo que significa que no hay más pasos para hacer y por

ende la lista ya quedaría ordenada.

### *Shell Sort:*

El método *“Shell Sort”* es una mejora del *“Insert Sort”* que compara y ordena elementos que están separados por una cierta distancia llamada **“gap”** que en este código es la variable **g**. Al inicio del procedimiento, g se fija como la mitad del tamaño de la lista de elementos, y en cada ciclo se va reduciendo a la mitad. Con cada valor de gap, el algoritmo toma un índice i, compara el elemento actual j con otro que está a g posiciones atrás. En otras palabras, toma dos elementos que están a g elementos de distancia, y si están desordenados los intercambia.

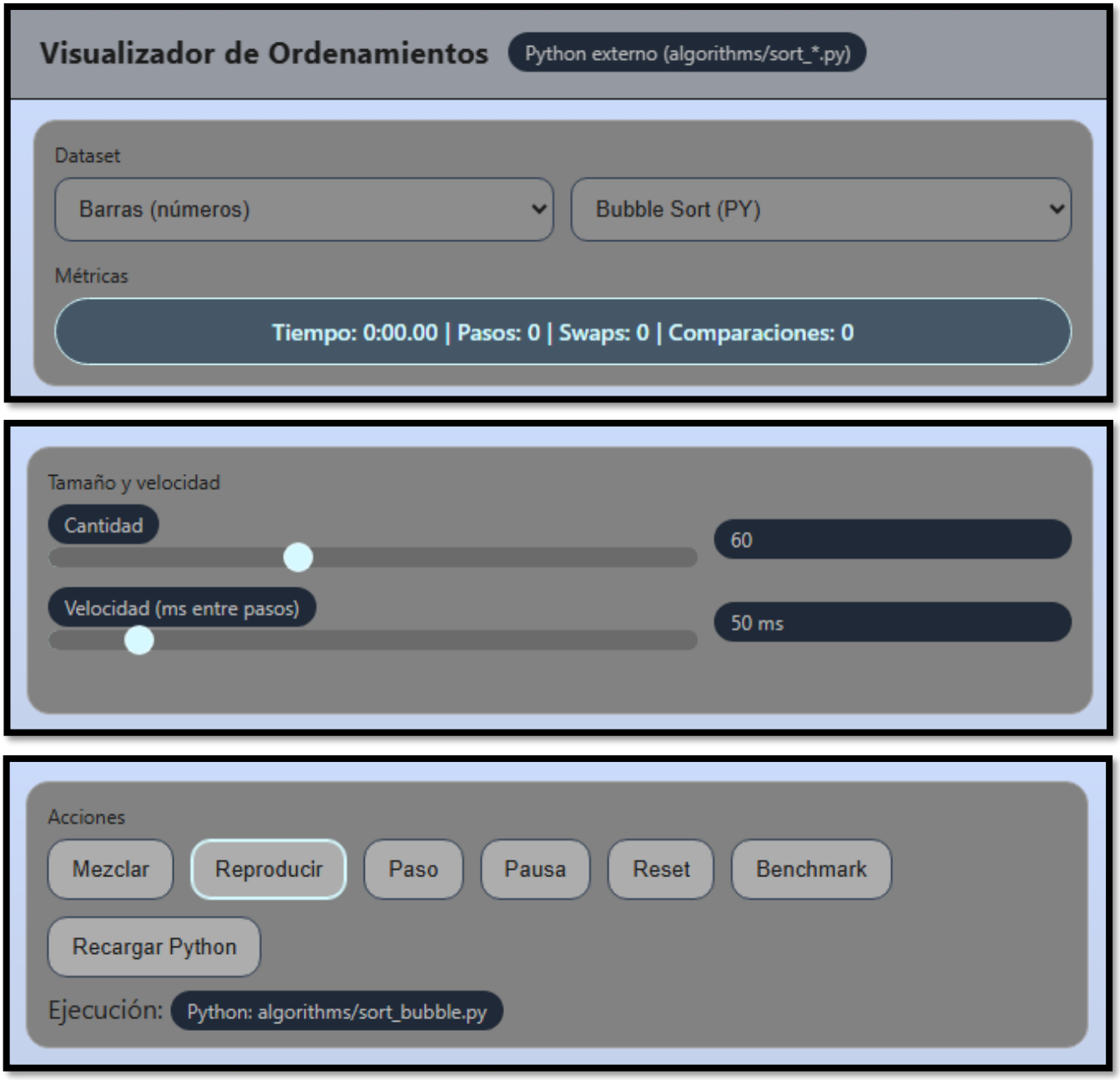



En cada paso, el algoritmo muestra los índices comparados a y b, y si hubo intercambio. Cuando ya ha recorrido todo el arreglo para un gap, el gap se reduce como se mencionó antes, y se reinicia el proceso. Cuando el gap llega a 0, significa que no hay distancias para comparar y eso quiere decir que la lista quedó ordenada.

### *Visualizador*

Con respecto al visualizador, se realizaron cambios de interfaz mediante nuestro básico conocimiento en HTML y ayuda de la IA. Por ejemplo, el visualizador es de color oscuro, con una gamma de grises y celestes que otorgan originalidad al proyecto. Se logro, sumado a ello, mostrar de distintos colores las barras del visualizador para que se distingan las variables a y b.

También, se logró añadir mediante la ayuda de la inteligencia artificial la reproducción de métricas de tiempo, y contador de pasos, swaps y comparaciones.




También, en el repositorio, se incluyen algunas imágenes divertidas para utilizar y probar los algoritmos de ordenamiento con ellas.

## 3. *Conclusión*

La realización del trabajo nos permitió comprender de manera más profunda el funcionamiento interno de los algoritmos de ordenamiento. El hecho de tener que adaptarlos a un modelo de ejecución basado en micro-pasos obligó a analizar con detalle cómo avanzan los punteros, cómo se comparan los elementos y qué implica cada intercambio dentro del proceso de ordenamiento.

Lo más difícil del trabajo fue convertir algoritmos completos, como Merge Sort y Quick Sort, en versiones compatibles con la visualización paso a paso. Esto nos llevó a estudiar su lógica interna, consultar en fuentes e inteligencia artificial, y a tomar decisiones de implementación que no suelen verse en un uso tradicional, como utilizar swaps adyacentes o reemplazar recursión por una pila.

También resultó muy útil observar cómo cambian el rendimiento y la estructura de los algoritmos al comparar métodos simples (Bubble, Selection, Insertion) con otros más eficientes (Merge, Quick y Shell). Las métricas incorporadas en el visualizador nos permitieron dimensionar esas diferencias de manera concreta. Algo que se notó muy fuerte es la diferencia de calidad entre las primeras con estas últimas. Decidimos implementarlas por querer hacer un trabajo más detallado y completo, sin embargo, se tornó un desafió grande.

En conjunto, el trabajo nos ayudó a reforzar la comprensión conceptual y práctica de los algoritmos, a enfrentar problemas de implementación reales y a trabajar en equipo para integrar las soluciones. Consideramos que el proyecto cumplió plenamente su propósito formativo.

**ANEXO: CONSIGNA**

Universidad Nacional de General Sarmiento

Introducción a la Programación

# TP – Visualización de algoritmos de ordenamiento

## Introducción (¿qué es un algoritmo de ordenamiento?)

Un **algoritmo de ordenamiento** es un procedimiento que re-acomoda una colección de elementos según un criterio, por ejemplo:

- ordenar números de menor a mayor,
- ordenar palabras alfabéticamente,
- ordenar objetos por alguna propiedad (precio, fecha, etc.).

Existen muchas estrategias (Bubble, Selection, Insertion, Quick, Merge…), cada una con una idea distinta para comparar e intercambiar elementos.

## ¿Qué haremos y cómo se conectará con Python?

Vas a implementar algoritmos de ordenamiento **en Python**, y ver su ejecución **animada en el navegador**. Esto se alinea con lo visto en la materia:

- **Variables** para guardar estado.
- **Listas** e **índices** para acceder e intercambiar elementos.
- **Condicionales** para decidir intercambios.
- **Bucles** modelados como **punteros** que avanzan entre llamadas (en lugar de `for` grandes, el TP usa la idea de "un paso por vez").
- **Booleanos** para indicar acciones (por ejemplo, si hubo intercambio).

## Objetivo

Implementar **al menos 3 algoritmos distintos** de ordenamiento cumpliendo el contrato `init(vals)` + `step()` que usa el visualizador.

A implementar:

- **Bubble**
- **Selection**
- **Insertion**.

Opcionales / extra (para quienes quieran sumar puntos):

- **Quick** (iterativo con pila), **Merge** (bottom-up), **Shell**, **Heap**, etc.
- **Métricas** (contadores de comparaciones/intercambios) y análisis breve.

## Estructura del proyecto y ejecución

```
/visualizador/
  index.html              # visualizador web (provisto)
  /algorithms/
    sort_bubble.py
    sort_selection.py
    sort_insertion.py
    sort_template.py      # plantilla para agregar nuevos
```

*podés sumar sort_quick.py, sort_merge.py, sort_shell.py, ...*

**Cómo correr:**

En `/visualizador` ejecutar:

1. **python -m http.server**
2. Abrir `http://localhost:8000` (ideal en incógnito).
3. Elegir **dataset** y **algoritmo** en los selectores.
4. Botones: **Mezclar**, **Reproducir**, **Paso**, **Pausa**, **Reset**.

> El selector de algoritmo carga automáticamente tu archivo `algorithms/sort_<valor>.py`.

## Dónde programar y qué tocar

- Todo tu código va en **`/algorithms/`**.
- Para crear un algoritmo nuevo, copiá `sort_template.py` → `sort_mialgo.py` y **agregá** la opción en el `<select id="algorithm">` del `index.html` con `value="mialgo"`.
- **No es necesario modificar `index.html`** para los algoritmos provistos.

## Contrato que debe cumplir tu código

Cada archivo `sort_<algo>.py` debe exponer **dos funciones** globales:

## `init(vals: list[int]) -> None`

Se llama **una vez** al comenzar (o cuando se remezcla). Debés:

- Guardar una **copia**: `items = list(vals)`.
- Guardar `n = len(items)`.
- Inicializar el **estado/punteros** de tu algoritmo (p. ej. `i`, `j`, `min_idx`, una pila, etc.).

## `step() -> dict`

Se llama **muchas veces**. Cada llamada realiza **UN solo micro-paso** y devuelve:

```
{
  "a": int,    # índice A para resaltar (0..n-1)
  "b": int,    # índice B para resaltar (0..n-1)
  "swap": bool,  # True si en ESTE paso intercambiaste items[a] <-> items[b]
  "done": bool   # True cuando el algoritmo terminó
}
```

**Sobre `swap`:**

- `swap=True` le indica al visualizador que **también** mueva las columnas A y B.

Vos **debés realizar el mismo swap** en tu lista Python **antes** de devolver el dict:

```
 items[a], items[b] = items[b], items[a]
return {"a": a, "b": b, "swap": True, "done": False}
```

- 

**Siempre debes cumplir:**

- `0 <= a,b < n` siempre que devuelvas índices.
- Si `swap=True`, ya hiciste ese intercambio en `items`.
- Al finalizar, devolves `{"done": True}` y detener la secuencia de pasos.
- Actualizar correctamente tus punteros/estado en cada paso.

## Cómo usa la UI tu `step()`

- **Reproducir**: llama a `step()` en bucle con pausas.
- **Paso**: llama a `step()` **una sola vez**.
- Si `step()` devuelve `{"done": True}`, la ejecución se detiene y se limpia el resaltado.
- Si `swap=True`, se destaca `(a,b)` y luego el visualizador intercambia esos dos elementos en pantalla para sincronizar con tu lista.

## Extensión de imagen por columnas (¿deben implementarla?)

- **No.** La extensión de "Imagen por columnas" ya está **provista** en el visualizador y **no requiere cambios de tu parte**. Tus algoritmos trabajan sobre una lista de enteros y la UI se encarga de mostrar/mover las columnas.
- Implementar desde cero el "corte de imagen" implicaría manipulación de imágenes en el navegador (JavaScript/Canvas) y **queda fuera de alcance** de esta materia..

## Entregable

- **(Obligatorio)** Carpeta `/algorithms/` con **al menos 3** archivos funcionando (ej.: `sort_bubble.py`, `sort_selection.py`, `sort_insertion.py`).
- **(Obligatorio)** Informe detallado documentando los algoritmos implementados junto con las decisiones tomadas, implementaciones aplicadas y dificultades encontradas.
- **(Obligatorio)** Un **README.md** breve con: integrantes, algoritmos implementados y una nota corta sobre decisiones de implementación.
- **(Opcional)** Algoritmos extra (`sort_quick.py`, `sort_merge.py`, `sort_shell.py`, ...)
- **(Opcional)** Agregar métricas sobre cantidad swaps, tiempo de duración, etc
- **(opcional)** Quien quiera explorar el código JS para cambiar cómo se corta/mezcla la imagen puede hacerlo y documentarlo como anexo.

## Errores comunes (y cómo evitarlos)

- `swap=True` **sin swap real**: la animación se desincroniza. Hacé primero el intercambio en `items` y luego devolvé `swap=True` con esos índices.

- **Índices fuera de rango**: cuidá límites (ej. en Bubble, `j+1 < n-i`).

- **Nunca llega `done`**: asegurá los avances de fase/anchura/pila al completar una pasada/bloque.

- **No reseteás estado en `init`**: remezclás y queda estado viejo; siempre reinicializá.