

Licenciatura en Sistemas

# Trabajo Práctico

## Algoritmos de ordenamiento

Introducción a la Programación

2°. Semestre 2025

Resumen: Implementación y ejecución visual de algoritmos de ordenamiento en Python.

Integrantes: Fariás, Celeste, cele2948@gmail.com  
Jara, Santiago, santiagojara981@gmail.com

## Introducción:

En este trabajo nos han solicitado implementar y ejecutar visualmente diferentes tipos de algoritmos de ordenamiento. Nosotros nos centramos en los tres siguientes:

- Bubble
- Selection
- Insertion

Cabe aclarar que todos ellos cumplen con un contrato `init(vals)` + `step()`, dado que el visualizador los necesita para poder operar adecuadamente.

## Desarrollo

Con el fin de llevar adelante la visualización de los algoritmos de ordenamiento (Bubble, Selection e Insertion), se trabajó implementando cada uno de ellos bajo el contrato propuesto, determinado por las funciones previamente mencionadas `init(vals)` y `step()`.

La consigna del trabajo propone que cada uno de los algoritmos trabaje con un micro-paso, de manera que el visualizador logre mostrar cuales son los elementos a comparar, cuales necesitan intercambiarse y en qué momento finaliza el algoritmo. De esta manera conseguimos una animación clara, comprensible y fiel al funcionamiento real de cada algoritmo.

Entre las principales dificultades encontradas, concluimos en que dos fueron las más relevantes. La primera fue lograr adaptar los algoritmos con las funciones ya definidas que nos habían otorgado, por lo que comprender el funcionamiento, que cambiaba según cada algoritmo, implicó gran complejidad. La segunda dificultad fue entender la lógica detrás de cada algoritmo de manera que el código final de cada uno de ellos sea funcional y acorde al contrato indicado en el trabajo.

Para resolver las dificultades encontradas, decidimos investigar el funcionamiento de cada algoritmo de manera independiente, por lo que consultar con distintas fuentes de información resultó de gran ayuda. Otra manera fue realizando muchas pruebas del código hasta obtener un resultado funcional.

## Estructura general del funcionamiento

Los algoritmos se implementaron siguiendo la siguiente estructura:

**init(vals)**: Se llama una única vez en el inicio o al remezclar, e inicializa las variables globales necesarias para controlar el estado del algoritmo.

**step()**: ejecuta un micro-paso del algoritmo y devuelve un diccionario con la forma:

```
{"a": int, "b": int, "swap": bool, "done": bool }
```

Esta estructura es la que permite la ejecución de los cambios y/o ordenamientos que se ven reflejados en el visualizador y se actualice la animación.

### Bubble Sort

Este algoritmo compara pares de elementos con sus elementos siguientes y, si están en un orden incorrecto, se intercambian. En cada pasada, el elemento más grande “burbuja” hasta el final de la lista.

Para adaptar el algoritmo al formato de micro-paso, se utilizaron los índices i y j como punteros del bucle externo e interno. Cada vez que se llama a la función step(), se realiza una comparación y si corresponde, un intercambio

Se devuelve el par de índices comparados y si ocurrió o no un swap. Cuando i alcanza el límite y por ende el final de la lista, se retorna {"done": True}.

---

### Selection Sort

El algoritmo de selección o Selection Sort ordena una lista buscando el valor mínimo de la parte no ordenada de la lista y lo intercambia con la posición inicial de esa sección.

Para representar este algoritmo visualmente de manera correcta, se dividió el proceso en dos fases:

**Fase “buscar”**: compara el elemento actual con el mínimo encontrado hasta el momento.

**Fase “swap”**: cuando finaliza el recorrido, se realiza un único intercambio entre la posición de inicio (**i**) y el índice del mínimo (**min\_idx**).

La división en fases permite que el visualizador muestre correctamente la búsqueda del valor mínimo y el momento del intercambio con la posición inicial.

Cada paso devuelve los índices comparados y, en la fase de swap, la información del intercambio.

---

## Insertion Sort

Este algoritmo lo que hace es tomar un par de elementos y los compara; si el primer elemento es mayor, los intercambia y pasa a la siguiente comparación de elementos. Es decir, desplaza los mayores elementos hacia la derecha y los menores hacia la derecha.

Se utiliza a la variable *j* para indicar la posición del elemento que se encuentra en proceso de inserción. A este elemento se lo compara y si debe de haber un intercambio se lo hace de manera adyacente:

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

En el caso de que no haya más intercambios por realizar, se pasa al siguiente índice.

Al finalizar cada *step()*, el algoritmo devuelve los índices de los elementos comparados y un “*swap*”: *True* que le indica al visualizador que cambie las columnas de lugar.

## Conclusiones

Para finalizar con el informe presentado, nos gustaría comentar que nos resultó un tanto confuso y complejo comprender el trabajo, pues trabajamos con herramientas y algoritmos que desconocíamos, sumado al uso de GitHub. Asimismo, la organización y la conformación de los grupos se vio afectada debido a la modalidad de cursada virtual.

**Anexo:** Enunciado (Enunciado del trabajo copiado textual)

**Código Bubble Sort:**

```
def step():
    global items, n, i, j

    # Comparar los indices j + j+1
    a = j
    b = j+1
    swap = False

    # Intercambiar si corresponde
    if items[a]>items[b]:
        items[a], items[b] = items[b], items[a]
        swap = True

    # Avanzar punteros |
    j += 1
    if j >= n-1-i:
        j = 0
        i += 1

    # Si i llego al limite (final de la lista) devuelve "done": True para terminar
    if i >= n-1:
        return {"done": True}

    # Devolver la informacion al visualizador
    return {"a": a, "b": b, "swap": swap, "done": False}
```

## Código comentado Selection Sort:

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

    # Si ya terminó
    if i >= n - 1:
        return {"done": True}

    # FASE BUSCAR
    if fase == "buscar":

        a = min_idx
        b = j

        # Comparar indices j con min_idx
        if items[j] < items[min_idx]:
            min_idx = j

        j += 1 # Pasamos al siguiente elemento

        # Si ya termino la parte no ordenada, pasa a fase swap
        if j >= n:
            fase = "swap"

        # Devolver la comparacion sin swap
        return {"a": a, "b": b, "swap": False, "done": False }

    # FASE SWAP
    if fase == "swap":

        a = i
        b = min_idx

        hubo_swap = (min_idx != i)

        # Intercambiar si corresponde
        if hubo_swap:
            items[i], items[min_idx] = items[min_idx], items[i]

        # Preparar la siguiente pasada
        i += 1
        min_idx = i
        j = i + 1
        fase = "buscar" # Vuelve a la fase buscar

        # Devuelve al visualizador
        return {"a": a, "b": b, "swap": hubo_swap, "done": False }
```

### Código comentado Insertion Sort:

```
def step():
    global items, n, i, j

    # Si ya termino
    if i >= n:
        return {"done": True}

    # Si empieza a desplazarse
    if j is None:
        j = i
        # Highlight del elemento sin swap
        if j > 0:
            return {"a": j-1, "b": j, "swap": False, "done": False}
        else:
            return {"a": j, "b": j, "swap": False, "done": False}

    # Mientras haya que desplazar hacia atrás
    if j > 0 and items[j-1] > items[j]:
        a, b = j-1, j
        items[a], items[b] = items[b], items[a]
        j -= 1
        return {"a": a, "b": b, "swap": True, "done": False}

    # Si no hay desplazamiento avanzar al siguiente i
    i += 1
    j = None
    if i>0:
        return {"a": i-1, "b": i, "swap": False, "done": False }
    else:
        return {"a": 0, "b": i, "swap": False, "done": False }
```

## TP – Visualización de algoritmos de ordenamiento

### Objetivos

Implementar Bubble, Selection e  
Insertion cumpliendo el contrato init(vals) + step() que  
usa la UI.

Ver el algoritmo animado y paso a paso

(una operación por llamada a step).

(Opcional) Agregar algoritmos extra y/o  
métricas (comparaciones, swaps, tiempo), y documentar un análisis breve.

---

¿Qué es un algoritmo de ordenamiento?

Un algoritmo de ordenamiento es un  
procedimiento que re-acomoda una colección según un criterio (números,  
palabras, objetos por propiedad, etc.).

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

---

### Estructura del repositorio

/visualizador/

index.html

#

visualizador web (provisto)

/algorithms/

sort\_bubble.py

sort\_selection.py

sort\_insertion.py

sort\_template.py

```
# plantilla para nuevos algoritmos
```

podés agregar:

sort\_quick.py, sort\_merge.py, sort\_shell.py, ...

Nota: la extensión “imagen por columnas” ya está implementada.

Tus algoritmos trabajan sobre una lista de enteros; la UI se encarga de mostrar y mover las columnas.

---

### Cómo

ejecutar el visualizador

En una terminal, ubicarse en la carpeta /visualizador

Ejecutar:

python -m http.server

Abrir <http://localhost:8000> (recomendado en modo incógnito).

Elegir dataset y algoritmo.

Usar los botones: Mezclar, Reproducir,

Paso, Pausa, Reset.

El selector de algoritmo carga

automáticamente el archivo algorithms/sort\_<valor>.py.

---

### Contrato de los archivos sort\_<algo>.py

Cada archivo debe exponer dos funciones

globales:

init(vals:

list[int]) -> None

step()

-> dict

init(vals)

Se ejecuta una vez al comenzar (o tras mezclar).

Debe:

Guardar copia: items

= list(vals)

Guardar n

= len(items)

Inicializar los punteros o estado interno (i, j, min\_idx,

pila, etc.)

step()

Se llama muchas veces. Cada llamada

realiza un solo micro-paso y devuelve un diccionario:

{

"a":

int, # índice A (0..n-1)

"b": int, # índice B (0..n-1)

"swap":

bool, # True si hiciste items[a] <-> items[b]

"done":

bool # True si el algoritmo terminó

}

Reglas:

0 <= a,b < n

Si swap=True, el

intercambio ya debe haberse realizado:

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

Al  
finalizar: return

```
{"done": True}
```

Actualizá correctamente los  
punteros/estado en cada paso.

---

Nuevos algoritmos

Archivo: algorithms/sort\_<algo>.py

Agregar al <select>

`id="algorithm">` de index.html con value="`<algo>`"

No hace falta modificar index.html para

Bubble, Selection e Insertion

---

Guía de implementación

Bubble Sort

Estado: i, j, n

Comparar items[j] con items[j+1],

hacer swap si corresponde

Cuando j+1

`== n-i-1`, reiniciar j=0 y i+=1

Selection Sort

Estado: i, j, min\_idx

Buscar el mínimo en  $i..n-1$ , swap

con  $i$  al final de la pasada

Insertion Sort

Estado:  $i, j$

Insertar el elemento  $i$  en  
la porción ordenada  $0..i-1$ ,

intercambiando adyacentes

---

### Entregables

Obligatorio:

Carpeta /algorithms/ con

al menos 3 algoritmos (bubble, selection, insertion)

Informe detallado con decisiones y

dificultades

README del equipo con integrantes y

notas de implementación

Opcional:

Nuevos algoritmos (quick, merge, shell,

etc.)

Métricas, benchmarks o mejoras visuales

---

### Formato de entrega

La entrega se divide en 2 partes: código

e informe.

Parte 1: Código

Todo el desarrollo debe estar en un

repositorio interno del grupo (fork del repo base del TP).

Agregar a los docentes de la comisión

para revisión y seguimiento.

Los alumnos deben notificar a los  
docentes para pre-entregas o bloqueos.

Sugerencias:

Cada integrante debe tener su propia  
cuenta de GitHub.

Cada integrante debe commitear su parte  
del código, mostrando aportes individuales.

Parte 2: Informe

Debe incluir:

Una introducción general no técnica.

El código de los algoritmos  
implementadas.

Una breve explicación de cada algoritmo  
con dificultades y decisiones justificadas.

No incluir explicaciones de  
funcionalidades de Python u otros frameworks.

El informe debe estar en PDF dentro de  
la carpeta del TP.

🔥 Ambas partes (código + informe) son obligatorias para  
aprobar.

---

 Documentación adicional

[Documentación oficial de Django](#)

Sección GIT:

Introducción a GIT: [clic acá](#).

Manejo de ramas/branches: [clic acá](#).

Merge & resolución de conflictos: [clic acá](#).

---

 Checklist antes de entregar

Los 3 algoritmos base están  
implementados y finalizan correctamente

init

resetea el estado

step

realiza un micro-paso

Swaps hechos antes de devolver swap=True

Probado con listas vacías, cortas,  
ordenadas e inversas

Informe y README listos