Universidad Nacional de General Sarmiento

Licenciatura en Sistemas

Trabajo Práctico
Métodos de ordenamiento

Introducción a la Programación

(2°semestre-2025)

Resumen: En este informe desarrollaremos los diferentes métodos de ordenamiento implementados en Python, que se verán en una ejecución animada desde el navegador.

Integrantes: López Candela, candela.aylen.l07@gmail.com
Messina Valentina, messina.valu@gmail.com
Perez Victoria, cataperez953@gmail.com
Vásquez Ludmila, ludmilavesquez08@gmail.com

En el siguiente informe, hablaremos sobre la implementación y análisis de distintos métodos de ordenamiento. Nuestro problema es que debemos ordenar distintos elementos (barras/imagen), utilizando cuatro algoritmos clásicos de ordenamiento: bubble sort, insertion sort, selection sort y quick sort.

Códigos:

Bubble Sort:

```
# Estado global
items = [] #lista de lo que vamos a ordenar
n = 0 #longitud de la lista
i = 0 #pasadas completadas
j = 0 #compara items[j] con items[j+1]
finished = False #indica si el algoritmo ha terminado
def init(vals):
    global items, n, i, j, finished
    items = list(vals) #copia de entrada
    n = len(items) #tamaño de la lista
    i = 0
    j = 0
    finished = False
def step():
    global items, n, i, j, finished
#si el algoritmo ha terminado, devolver done True
    if finished:
        return {"done": True}
    if i >= n - 1:
        finished = True
        return {"done": True}
    a = j
    b = j + 1
    swapped = False
#comparar y posiblemente intercambiar
```

```
    if items[a] > items[b]:
        items[a], items[b] = items[b], items[a]
        swapped = True
#avanzar cursores
    j += 1
    if j >= n - i - 1:
        j = 0
        i += 1
#devolver resultado del paso
    return {
        "a": a,
        "b": b,
        "swap": swapped,
        "done": False
    }
```

Insertion Sort:

```
#Insertion sort
items = []
n = 0
i = 0      # elemento que queremos insertar
j = None   # cursor de desplazamiento hacia la izquierda (None = empezar)
def init(vals):
    global items, n, i, j
    items = list(vals)
    n = len(items)
    i = 1      # se empieza desde el segundo elemento
    j = None
def step():
    global items, n, i, j
```

```
    # 1) Si terminamos
    if i >= n:
        return {"done": True}
    # 2) Si j es None: recién vamos a empezar con este i
    if j is None:
        j = i
        return {"a": j, "b": j-1 if j-1 >= 0 else j, "swap": False, "done": False}
    # 3) Si debemos seguir desplazando hacia la izquierda
    #    Mientras j > 0 y items[j-1] > items[j]
    if j > 0 and items[j - 1] > items[j]:
        # swap adyacente (j-1 con j)
        a = j - 1
        b = j
        items[a], items[b] = items[b], items[a]
        j -= 1
        return {"a": a, "b": b, "swap": True, "done": False}
    # 4) Ya no hay que desplazar → avanzar al siguiente i
    i += 1
    j = None
    # Highlight mínimo para mostrar avance
    return {"a": i-1, "b": i-1, "swap": False, "done": False}
```

Selection Sort:

```
# sort_selection.py
# Algoritmo de ordenamiento: SELECTION SORT (versión paso a paso para el visualizador)
items = []
n = 0
# Punteros / estado del algoritmo
i = 0
j = 0
min_idx = 0
```

```
fase = "buscar"  # "buscar" = buscando mínimo, "swap" = realizar intercambio
def init(vals):
    global items, n, i, j, min_idx, fase
    items = list(vals)
    n = len(items)
    # Inicializar punteros
    i = 0
    j = i + 1
    min_idx = i
    fase = "buscar"
def step():
    global items, n, i, j, min_idx, fase
    # Caso final
    if i >= n - 1:
        return {"done": True}
    # ==========================
    # FASE 1: BÚSQUEDA DEL MÍNIMO
    # ==========================
    if fase == "buscar":
        # Comparar j con min_idx
        if items[j] < items[min_idx]:
            min_idx = j
        # Devolver comparación (a=j, b=min_idx)
        a, b = j, min_idx
        swap = False
        # Avanzar j
        j += 1
        # Si terminó la búsqueda, pasar a swap
        if j >= n:
            fase = "swap”
```

```
        return {"a": a, "b": b, "swap": swap, "done": False}
    # ========================
    # FASE 2: HACER EL SWAP
    # ========================
    else:  # fase == "swap"
        a, b = i, min_idx
        # Hacer el swap real
        items[a], items[b] = items[b], items[a]
        # Avanzar a la próxima pasada
        i += 1
        j = i + 1
        min_idx = i
        fase = "buscar"
        return {"a": a, "b": b, "swap": True, "done": False}
```

Quick Sort:

```
# Algoritmo de ordenamiento QuickSort:
items = []
n = 0
# Los punteros/estado:
puntero_i = 0      # Índice para el rastreo del último elemento <= pivote
puntero_j = 0          # Puntero que recorre el sub-array (elemento actual a comparar)
indice_pivote = 0  # Índice del elemento pivote (generalmente el final del rango)
limite_inferior = 0# Límite bajo (low) del sub-array actual
limite_superior = 0# Límite alto (high) del sub-array actual
fase_actual = "INIT" # Controla la máquina de estados: "INIT", "PARTICION", "INTERCAMBIO_PIVOTE", "TERMINADO"
pila_rangos = []     # Pila para almacenar los límites (low, high) de los sub-arrays pendientes
def init(vals):
    """Inicializa la lista y el estado del algoritmo."""
```

```
    global items, n, puntero_i, puntero_j, indice_pivote, limite_inferior, limite_superior, fase_actual, pila_rangos

    items = list(vals)

    n = len(items)

    # Inicialización de punteros/estado

    puntero_i = 0

    puntero_j = 0

    indice_pivote = 0

    limite_inferior = 0

    limite_superior = n - 1

    fase_actual = "INIT"

    pila_rangos = []

    if n > 0:

        # Agrega el rango inicial a la pila para comenzar

        pila_rangos.append((limite_inferior, limite_superior))

def _configurar_particion():

    """Configura el estado para el inicio de una partición."""

    global puntero_i, puntero_j, indice_pivote, limite_inferior, limite_superior, fase_actual

    # El pivote es el último elemento del rango

    indice_pivote = limite_superior

    # El puntero i se inicializa justo antes del inicio del rango

    puntero_i = limite_inferior - 1

    # El puntero j comienza en el primer elemento del rango

    puntero_j = limite_inferior

    fase_actual = "PARTICION"

def step():

    """Implementa UN micro-paso del Quicksort iterativo y devuelve el dict."""

    global items, n, puntero_i, puntero_j, indice_pivote, limite_inferior, limite_superior, fase_actual, pila_rangos

    if fase_actual == "TERMINADO":
```

```
        return {"done": True}
    if fase_actual == "INIT":
        if not pila_rangos:
            fase_actual = "TERMINADO"
            return {"done": True}
        # Saca un rango de la pila y lo configura
        limite_inferior, limite_superior = pila_rangos.pop()
        # Si el sub-array tiene 1 o 0 elementos, avanza al siguiente
        if limite_inferior >= limite_superior:
            return step()
        _configurar_particion()
    if fase_actual == "PARTICION":
        # Recorrido de puntero_j
        if puntero_j < indice_pivote:
            # Compara el elemento actual con el pivote
            if items[puntero_j] <= items[indice_pivote]:
                # Si es menor o igual, incrementa i y prepara el posible intercambio
con j
                puntero_i += 1
                if puntero_i != puntero_j:
                    # Devuelve el paso de intercambio y pasa a la fase de transición
                    fase_actual = "TRANSICION_SWAP"
                    return {"a": puntero_i, "b": puntero_j, "swap": True, "done": False}
            # Avanza puntero_j para el siguiente elemento
            puntero_j += 1
            # Devuelve el paso de comparación (sin swap)
            indice_a = puntero_i if puntero_i != puntero_j else puntero_j - 1
            return {"a": indice_a, "b": puntero_j - 1, "swap": False, "done": False}
        # Si puntero_j ha llegado al pivote
        elif puntero_j == indice_pivote:
```

```
            # Prepara el intercambio final del pivote a su posición correcta (puntero_i + 1)
            puntero_i += 1
            fase_actual = "INTERCAMBIO_PIVOTE"
            return {"a": puntero_i, "b": indice_pivote, "swap": True, "done": False}
    if fase_actual == "TRANSICION_SWAP":
        # Después de que se realiza el swap entre i y j, avanzamos j y volvemos a la PARTICION
        puntero_j += 1 # Ya se avanzó en la llamada anterior, pero por la estructura lo hacemos aquí (o en PARTITION). Lo quitamos de la anterior.
        fase_actual = "PARTICION"
        # Llamamos a step para avanzar la lógica inmediatamente
        return step()
    if fase_actual == "INTERCAMBIO_PIVOTE":
        # El pivote ha sido colocado en su posición final: items[puntero_i]
        indice_pivote_final = puntero_i

        # 1. Agrega el sub-array izquierdo a la pila
        if indice_pivote_final - 1 > limite_inferior:
            pila_rangos.append((limite_inferior, indice_pivote_final - 1))
        # 2. Agrega el sub-array derecho a la pila
        if indice_pivote_final + 1 < limite_superior:
            pila_rangos.append((indice_pivote_final + 1, limite_superior))
        # Pasamos a la siguiente partición
        fase_actual = "INIT"
        # Devolvemos el estado del pivote finalizado (sin swap en este paso)
        return {"a": indice_pivote_final, "b": indice_pivote_final, "swap": False, "done": False}
    return {"done": False} # Estado por defecto
```

La solución desarrollada consiste en implementar los cuatro métodos de ordenamiento. Para esto, se trabajó bajo una estructura común definida por dos funciones principales: init(vals), encargada de recibir e inicializar la lista de valores a ordenar, y step (), responsable de ejecutar de manera incremental cada paso del algoritmo hasta completar el ordenamiento.

El enfoque elegido permite visualizar el comportamiento interno de cada método, ya que la ejecución paso a paso muestra cómo los elementos se intercambian o reposicionan con cada iteración. Durante el desarrollo surgieron desafíos, como adaptar cada algoritmo para que pudiera ejecutarse de forma pausada y no en un solo ciclo completo y la utilización de Git/GitHub.

A continuación, desarrollaremos las funcionalidades de los distintos métodos de ordenamientos:

-Bubble Sort: La funcionalidad de este método de ordenamiento consiste en hacer múltiples pasadas por la lista, empujando el mayor elemento de cada pasada hacia el final. En lugar de ejecutar el algoritmo completo de una vez, lo divide en micro pasos. Cada llamada a step () realiza solo una comparación (y, si corresponde, un intercambio) y devuelve un diccionario indicando qué sucedió en ese micro paso. Para que esto funcione, el programa necesita recordar su estado entre llamadas; por eso usa variables globales.

El programa guarda:

- items: la lista que se está ordenando
- n: el tamaño de la lista
- i: el número de pasadas completas que ya se realizaron
- j: el índice de comparación actual dentro de la pasada
- finished: indica si el algoritmo ya termino.

Esto permite que cada llamada a step () continue en donde quedo la llamada anterior.

La función init(vals) se llama solo una vez al comienzo, y su función es preparar el estado inicial, para luego llamar a step (). Lo que hace es una copia de la lista original(ítems), calcula el tamaño de la lista, inicia a i y j en cero y marca finished=False.

La función step () lo que hace es:

- Si finished es True, devuelve únicamente {"done": True}. O si i >= n-1, significa que ya se hicieron todas las pasadas necesarias, así que marca finished = True y devuelve {"done": True}.
- Compara dos elementos: el micro paso consiste en comparar: items[j] con items[j+1], por lo que guarda: a = j, b = j+1 y una variable swapped que inicialmente es False. Si items[a] > items[b], significa que están desordenados, así que se intercambian y swapped pasa a ser True.

- Luego se incrementa j (j+1) y se compara j y j+1. Si j llega al limite de pasada (j>= n-i-1) significa que se termino la pasada completa, j vuelve a cero e i aumenta en uno.
- Finalmente, lo que devuelve step () es un diccionario con los índices comparados (a y b), el swap: swapped (True si hubo un intercambio) y el done: False (todavía no termino)

Cada micro paso informa exactamente que comparación se hizo y si hubo o no un swap. Cuando todo termina devuelve {“done”: True}.

-Insertion Sort: La funcionalidad implementada en el algoritmo de ordenamiento es una simulación paso a paso del mismo. Su objetivo es mostrar visualmente cómo se desplaza cada elemento hacia su posición correcta comparándose con los elementos anteriores al mismo. Para lograrlo, el programa utiliza tres variables principales, siendo "i" el índice del elemento que se desea insertar en la parte ya ordenada; "j" el cursor que se mueve hacia la izquierda comparando y haciendo intercambios; y los "ítems" siendo la lista de valores que se están ordenando.

La idea general es que la función step () realiza un único paso del proceso de Insertion cada vez que se la llama, lo que permite ver el ordenamiento progresivamente. El “swap” se realiza cuando el elemento actual debe desplazarse hacia la izquierda del elemento anterior y cuando ya no necesita moverse más, el algoritmo avanza al siguiente índice.

El código está pensado para un visualizador de algoritmos, donde se necesita que cada llamada represente un movimiento sencillo y visible, siendo este el motivo de que no ordene todo de una vez, sino que va devolviendo información útil para la animación.

Dentro de otras funciones se incluyen: init(vals) que carga la lista inicial y reinicia los índices y step () que controla el avance del algoritmo.

Dentro de los parámetros que toma se encuentran: init(vals) que recibe una lista de valores a ordenar y no devuelve nada; step () que no recibe parámetros y devuelve un diccionario con: a y b (índices afectados en ese paso); swap (si hubo intercambio); y done (si el algoritmo terminó).

-Selection Sort: La funcionalidad implementada de este algoritmo de ordenamiento es una simulación paso a paso del proceso de selección del mínimo. Su objetivo es mostrar visualmente cómo, en cada posición de la lista, el algoritmo busca el elemento más pequeño del resto del arreglo y luego lo intercambia con la posición actual.

Para lograr esto, el programa utiliza tres variables principales:

- i: indica la posición donde se colocará el mínimo encontrado.
- j: recorre el resto de la lista buscando el valor más pequeño.
- min_idx: guarda el índice del elemento mínimo durante la búsqueda.

La idea general es que la función step () realiza un único paso del Selection Sort cada vez que es llamada, permitiendo ver el proceso de forma gradual. Cuando j termina de recorrer la lista, se realiza el “swap” entre el elemento en i y el mínimo encontrado en min_idx. Después de eso, el algoritmo avanza al siguiente índice.

El código está pensado para un visualizador de algoritmos, por lo que cada paso debe representar una acción simple y animable (una comparación o un intercambio), en lugar de ejecutar todo el ordenamiento de una sola vez.

Las funciones principales son:

- init(vals): carga la lista a ordenar, reinicia los índices y establece la fase inicial del algoritmo.
- step (): controla el avance del ordenamiento, mostrando las comparaciones, actualizando el mínimo y realizando los intercambios cuando corresponde.

Los parámetros utilizados son:

- init(vals) recibe una lista de valores y no devuelve nada.
- step () no recibe parámetros y devuelve un diccionario con la información necesaria para la animación:
- a y b: índices comparados en ese paso,
- swap: indica si hubo intercambio,
- done: señala si el algoritmo finalizó.

-Quick Sort: El código implementa el algoritmo de ordenamiento de Quicksort. Su objetivo principal conta en ordenar elementos mediante la iteración con pila (pila_rangos), separándose de su forma tradicional dada mediante la recursión. Esta pila sirve como una lista de tareas pendientes, que almacena los índices de los segmentos divididos de una lista que deben ser ordenados, por lo cual el ordenamiento finaliza cuando la pila se encuentra vacía. Para lograr esto, la funcionalidad central se basa en la máquina de estados (fase_actual), ya que esta dirige la lógica del programa paso a paso.

El funcionamiento se organiza en tres funciones principales: Primero, se encuentra la función init(vals), la cual sirve como el punto de entrada y se encarga de la preparación. Su objetivo es copiar la lista original de números a la variable de trabajo (items), reiniciar todos los punteros y variables de estado a cero e inicializar la pila (pila_rangos) guardando el rango completo de la lista como la primera tarea a ejecutar. Segundo, está la función auxiliar _configurar_particion (), la cual se llama para ajustar los punteros cada vez que se inicia un nuevo trabajo de la pila. Establece el pivote (indice_pivote) como el número final del rango y ajusta los punteros de rastreo (puntero_i) y de recorrido (puntero_j) para que empiecen con la partición correctamente. Asimismo, cambia el estado (fase_actual) a "PARTICION". Finalmente, tenemos a la función step (), la cual ejecuta paso por paso, basándose en lo que indica la variable fase_actual. Si el estado es "INIT", saca la siguiente tarea de la pila.

Si es "PARTICION", compara un elemento con el pivote, avanza el puntero puntero_j y de ser necesario notifica que debe realizarse un intercambio (cambiando el estado a "TRANSICION_SWAP"). Si el estado es "INTERCAMBIO_PIVOTE", el proceso ha llegado a la etapa del desglosamiento, en el cual se coloca el pivote en su lugar definitivo y el algoritmo calcula los límites de los dos nuevos sub-arrays generados (los más pequeños y los más grandes que el pivote), guardándolos en la pila como nuevas tareas antes de volver al estado "INIT" para buscar la siguiente tarea pendiente. Al terminar cada paso, la función devuelve un diccionario que informa al sistema externo cuáles son los índices que se modificaron y/o miraron.

En conclusión, la realización de este trabajo permitió comprender en profundidad el funcionamiento de los cuatro algoritmos de ordenamiento implementados (bubble sort, insertion sort, selection sort y quick sort) adaptándolos al esquema init(vals) y step () para ejecutarlos de manera incremental y visualizable. Si bien surgieron desafíos, especialmente al descomponer cada algoritmo en micro pasos y al manejar correctamente los estados internos (en especial en quick sort), estas dificultades permitieron afianzar el entendimiento de su lógica interna. El uso de Git/GitHub también facilitó la organización y el control de versiones durante el desarrollo.

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**.