Universidad Nacional de General Sarmiento

VISUALIZACION DE Algoritmos de Ordenamiento

INFORME

MATERIA:
Introduccion a la Programacion (COM 11)

DOCENTES:
Omar Argañaras - Luca Velazquez

PRESENTADO POR:

Sanchez, Rocio Micaela

FECHA DE ENTREGA
17- Noviembre- 2025

Introducción

El presente Trabajo Práctico tiene como objetivo principal la implementación de, al menos, tres algoritmos de ordenamiento fundamentales basados en comparación (como Bubble Sort, Selection Sort e Insertion Sort, entre otras estrategias). El desarrollo se realiza en el lenguaje de programación Python con el fin de asimilar la lógica algorítmica y observar su ejecución en un entorno visual web.

La consigna central de este trabajo consistió en adaptar la lógica iterativa de estos algoritmos para que funcionen bajo un sistema de micro-pasos. Esto implica que, en lugar de ordenar la lista completa en una única operación, cada llamada a la función step() solo realiza la comparación y/o el intercambio de un único par de elementos. Esta aproximación permite la visualización animada del proceso de ordenamiento en un entorno web interactivo.

Tecnologías utilizadas

Para la realización del proyecto se utilizaron varias tecnologías clave. Por un lado, se empleó la plataforma **GitHub** para la gestión y control de versiones del repositorio, lo que facilitó la clonación del proyecto original y la posterior contribución de los cambios realizados. En sintonía con ello, se utilizó **Git** con el afán de registrar cada cambio, asegurando un historial de desarrollo ordenado y permitiendo revertir errores. A su vez, se utilizó el entorno de desarrollo **Visual Studio Code (VS Code),** específicamente con la extensión **Live Server** para gestionar la visualización de manera ágil y eficiente. Finalmente, el resultado se visualiza en un **visualizador web externo** que interactúa con el código Python a través de un contrato de funciones específico (init y step), cumpliendo con el objetivo de presentar el proceso de ordenamiento de forma animada.

Desarrollo

Implementación y documentación de algoritmos

El núcleo de la implementación reside en replicar la lógica secuencial de los bucles de ordenamiento utilizando variables globales (o "punteros") para mantener el estado entre las llamadas a la función step(). El contrato (init/step) exige que step() devuelva un diccionario con los índices resaltados (a, b), si se realizó un intercambio (swap) y si el algoritmo finalizó (done).

A continuación, se detalla el funcionamiento, la implementación y las decisiones tomadas para cada algoritmo.

1) Algortimo Bubble Sort (sort_bubble.py)

Este algoritmo recorre repetidamente la lista, comparando elementos adyacentes y cambiándolos de lugar si están en el orden incorrecto. El elemento más grande "sube" (hace burbuja) al final de la lista en cada pasada completa. El algoritmo se optimiza al reducir el rango de comparación en cada nueva pasada, ya que los últimos elementos ya están en su posición final.

El Bubble Sort se implementó basándose en dos índices principales (i y j) que deben ser accedidos y modificados globalmente. El puntero i simula el bucle externo, marcando la región de la lista que ya está ordenada (desde la derecha hacia el centro). El puntero j simula el bucle interno, que avanza a través de la sublista no ordenada en cada pasada.

```python
items = []
n = 0
i = 0   # número de pasadas completas
j = 0   # índice dentro de la pasada

def init(vals):
    global items, n, i, j
    items = list(vals)
    n = len(items)
    i = 0
    j = 0
```

En cada llamada a step(), se ejecuta la comparación entre items[j] y items[j+1].
Se decidió que el avance se gestionaría por el incremento continuo de j. La transición de pasada ocurre cuando j alcanza el límite dictado por i (el fin de la sublista no ordenada): en ese momento, j se reinicia a cero e i avanza a la siguiente posición, marcando un nuevo elemento como ya ordenado.

```python
def step():
    global items, n, i, j
    # Si ya terminó (pasadas ≥ n-1)
    if i ≥ n-1:
        return {"done": True}

    a = j
    b = j+1

    # comparar y swap si corresponde
    if items[a] > items[b]:
        items[a], items[b] = items[b], items[a]
        swapped = True
    else:
        swapped = False

    # avanzar j; si llegamos al final de la pasada, reiniciar j y aumentar i
    j += 1
    if j+1 > n - i - 1:
        j = 0
        i += 1

    return {"a": a, "b": b, "swap": swapped, "done": False}
```

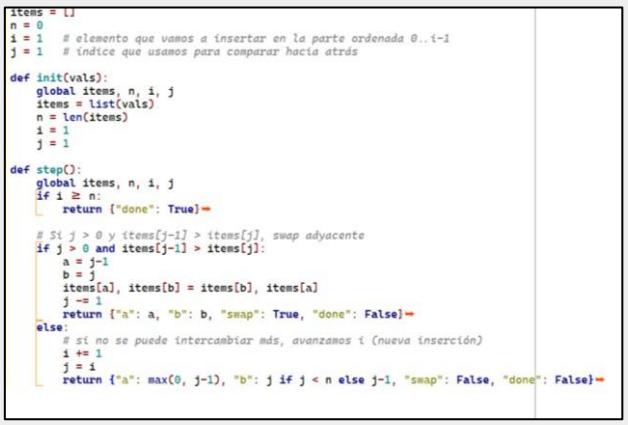
2) Algoritmo Insertion Sort (sort_insertion.py)

El Insertion Sort divide la lista en una porción ordenada (izquierda) y no ordenada (derecha). En cada iteración, toma el primer elemento de la porción no ordenada y lo "inserta" en su lugar correcto dentro de la porción ordenada, desplazando los elementos mayores hacia la derecha.

La clave es que cada "pasada" (determinada por i) puede requerir múltiples micropasos de swap hacia atrás (determinados por j).

- **i:** Posición del elemento actual que se intenta insertar.
- **j:** Índice de desplazamiento hacia la izquierda dentro de la porción ordenada.

def step():



```
items = []
n = 0
i = 1   # elemento que vamos a insertar en la parte ordenada 0..i-1
j = 1   # índice que usamos para comparar hacia atrás

def init(vals):
    global items, n, i, j
    items = list(vals)
    n = len(items)
    i = 1
    j = 1

def step():
    global items, n, i, j
    if i ≥ n:
        return {"done": True}

    # Si j > 0 y items[j-1] > items[j], swap adyacente
    if j > 0 and items[j-1] > items[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}
    else:
        # si no se puede intercambiar más, avanzamos i (nueva inserción)
        i += 1
        j = i
        return {"a": max(0, j-1), "b": j if j < n else j-1, "swap": False, "done": False}
```

- **Condición de fin:** Si i (la cabeza de la porción no ordenada) es mayor o igual a n, finaliza.

- **Caso base de avance (j es None o 0):** Cuando j es None o ha llegado al inicio de la lista (o sea, el elemento i ya está en su lugar), se avanza i y se reinicia el proceso de inserción para el siguiente elemento.

- **Desplazamiento (j > 0):** Compara el elemento en j con el elemento en j-1. Si items[j] < items[j-1], realiza el swap de items[j] y items[j-1]. Decrementa j (j = j - 1) para seguir comparando hacia la izquierda. Devuelve swap=True.

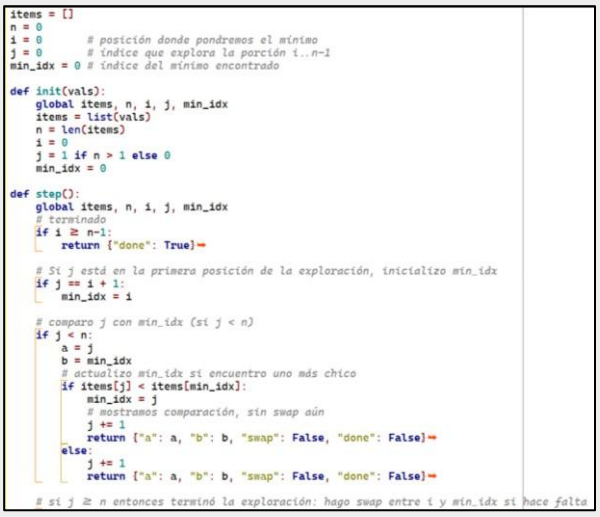
- **Elemento en posición:** Si items[j] >= items[j-1], el elemento i ya está en su posición correcta dentro de la parte ordenada. Se detiene el desplazamiento y se prepara para el Caso Base de Avance (paso 2).

Se decidió que la variable j controlara los swaps hacia atrás. Cada llamada a step() realiza una comparación y, si es necesario, un swap en j y j-1, y luego j retrocede. Esto simula el "desplazamiento" de un solo elemento por vez.

El algoritmo debe detener el desplazamiento cuando el elemento ya no necesita moverse hacia la izquierda (items[j] >= items[j-1]). Esto se maneja en el else del código de la función.

3) Algoritmo Selection Sort (sort_selection.py)

El Selection Sort divide la lista en una porción ordenada (a la izquierda) y una no ordenada (a la derecha). En cada pasada, recorre la porción no ordenada, encuentra el elemento más pequeño, y lo intercambia con el primer elemento de la porción no ordenada. El índice de la porción ordenada avanza en uno con cada intercambio.



```
items = []
n = 0
i = 0       # posición donde pondremos el mínimo
j = 0       # índice que explora la porción i..n-1
min_idx = 0 # índice del mínimo encontrado

def init(vals):
    global items, n, i, j, min_idx
    items = list(vals)
    n = len(items)
    i = 0
    j = 1 if n > 1 else 0
    min_idx = 0

def step():
    global items, n, i, j, min_idx
    # terminado
    if i ≥ n-1:
        return {"done": True}

    # Si j está en la primera posición de la exploración, inicializo min_idx
    if j == i + 1:
        min_idx = i

    # comparo j con min_idx (si j < n)
    if j < n:
        a = j
        b = min_idx
        # actualizo min_idx si encuentro uno más chico
        if items[j] < items[min_idx]:
            min_idx = j
            # mostramos comparación, sin swap aún
            j += 1
            return {"a": a, "b": b, "swap": False, "done": False}
        else:
            j += 1
            return {"a": a, "b": b, "swap": False, "done": False}

    # si j ≥ n entonces terminó la exploración: hago swap entre i y min_idx si hace falta
```

```
    # si j ≥ n entonces terminó la exploración: hago swap entre i y min_idx si hace falta
    if min_idx ≠ i:
        items[i], items[min_idx] = items[min_idx], items[i]
        did_swap = True
    else:
        did_swap = False

    # preparo la siguiente pasada
    i += 1
    j = i + 1 if i + 1 < n else i
    return {"a": i-1, "b": min_idx, "swap": did_swap, "done": False}
```

La implementación requiere un manejo de estado más complejo, especialmente para guardar el índice del elemento mínimo encontrado durante la búsqueda:

- **i:** Cabeza de la porción no ordenada (el índice donde se colocará el elemento mínimo).
- **j:** Cursor que recorre la porción no ordenada (i hasta n-1) buscando el mínimo.
- **min_idx:** Índice del elemento mínimo encontrado hasta ahora en la pasada actual.
- **phase (Decisión de Diseño):** Variable de estado clave para alternar entre la fase de BÚSQUEDA y la fase de SWAP (intercambio) con el fin de cumplir con el contrato de micro-pasos.

def step():

- **Condición de Fin:** Si i (cabeza de la porción no ordenada) llega al final de la lista, devuelve {"done": True}.

- **Fase de Búsqueda:** Si j está dentro de los límites y items[j] < items[min_idx], actualiza min_idx = j.Avanza j. Devuelve la comparación resaltando j y min_idx.

- **Fin de Pasada / Fase de Intercambio:**Cuando j llega al final de la lista, la búsqueda ha terminado.Si i no es igual a min_idx, se realiza el swap de items[i] con items[min_idx] en la lista. Se devuelve swap=True.

- **Preparación para la siguiente pasada:** Incrementa i y reinicia j = i y min_idx = i para la siguiente pasada, volviendo a la fase de BÚSQUEDA

Implementación de algoritmos extras (Quick y Shell)

En esta sección se detallará de manera general la lógica de implementación para los algoritmos Quick Sort y Shell Sort.

4)Algoritmo Quick Sort (sort_quick.py)

Quick Sort es un algoritmo de división y conquista que selecciona un pivote y particiona el arreglo en dos subarreglos: elementos menores al pivote y elementos mayores al pivote.

```
# Variables Globales de Estado
items = []
n = 0
# Stack: Almacena las tareas pendientes de QuickSort como tuplas (low, high)
stack = []
# Punteros de la Partición Actual (La que se está trabajando en step())
low = 0
high = 0
# i es el puntero que rastrea la posición para el siguiente elemento menor al pivote
i = 0
# j es el cursor que explora el sub-array
j = 0
# pivot_val es el valor del pivote (tomado de items[high])
pivot_val = 0
# fase: Controla qué parte del algoritmo QuickSort se ejecuta en este step()
# 'partition': Fase de barrido (j avanza comparando con el pivote)
# 'swap_pivot': Fase donde se coloca el pivote en su posición final
# 'check': Fase de chequeo, donde se finaliza la tarea y se añaden nuevas subtareas a la pila
fase = 'check'

def init(vals):
    """
    Inicializa el estado del algoritmo Quick Sort.
    """
    global items, n, low, high, i, j, stack, fase
    items = list(vals)
    n = len(items)

    # Inicializa la pila con la tarea de ordenar el array completo
    stack = [(0, n - 1)]
    # La fase inicial debe ser 'check' para que el primer step() saque la primera tarea de la pila
    fase = 'check'
    low = 0
    high = n - 1
    i = -1
    j = 0
```

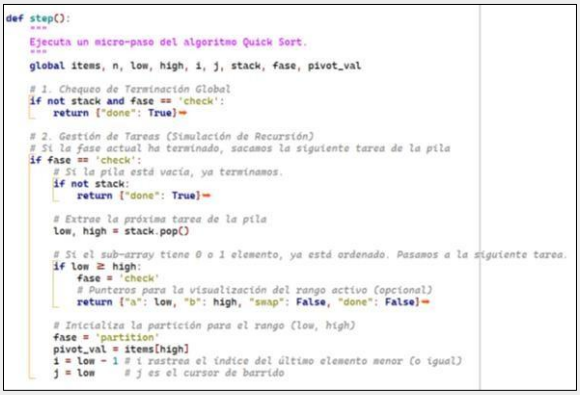
- **Adaptación de la Recursividad (Uso de Stack Explícito)**

  La recursividad natural de Quick Sort se elimina y se reemplaza con una pila de control (stack).
  Esta pila almacena las subtareas pendientes, es decir, los rangos (low, high) que aún necesitan ser particionados.

- La ejecución se divide en tres fases controladas por la variable fase: 'check', 'partition' y 'swap_pivot'.

- **Lógica del step() y las Fases**

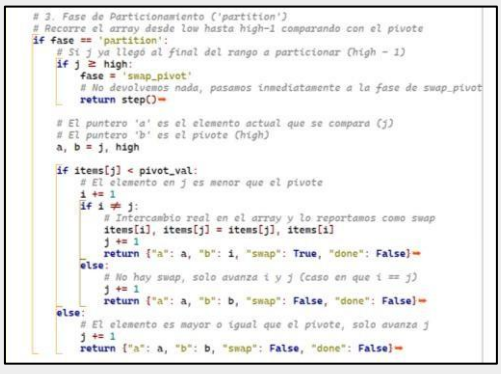
  - *Fase 'check' (Gestor de Tareas):* Cada vez que se llama a step() en esta fase, se saca un rango (low, high) de la pila. Si el rango es válido, se inicializa la Fase 'partition'.



  - *Fase 'partition' (Barrido):* Los punteros i y j recorren el sub-arreglo [low, high-1]. Si items[j] es menor al pivote, se incrementa i y se realiza un intercambio (swap: True) con items[i], moviendo el elemento pequeño a la izquierda.

  - Si items[j] es mayor o igual, solo avanza el cursor de barrido j (swap: False).

- **Fase 'swap_pivot' (Finalización):** Una vez que j ha barrido todo, esta fase se ejecuta una sola vez. Coloca el pivote (items[high]) en su posición final (i + 1) mediante un intercambio (swap: True) y luego añade las dos nuevas subtareas a la pila: (low, i) y (i + 2, high). Vuelve a la fase 'check'.



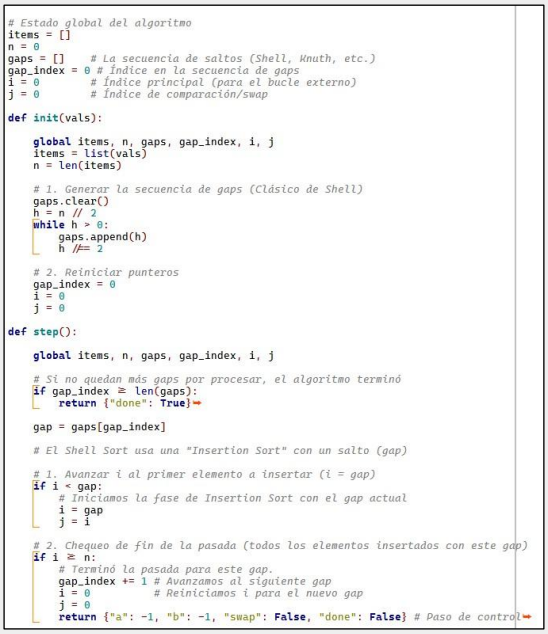
5)Algoritmo Shell Sort (sort_quick.py)

El algoritmo de ordenamiento Shell Sort es una mera optimización del algoritmo Insertion Sort. Se basa en una secuencia de gaps (saltos) para ordenar elementos distantes, reduciendo el número de desplazamientos de elementos.

1. Implementación por Micro-Pasos

La clave para adaptar Shell Sort a la función step() es gestionar los tres bucles anidados de forma iterativa:

- **Bucle externo:** Controlado por el gap_index que recorre la lista de saltos (gaps).
- **Bucle intermedio (i):** Recorre la lista de elementos que deben ser insertados para el gap actual.
- **Bucle interno (j):** Simula el desplazamiento hacia atrás (Insertion Sort) mientras el elemento actual sea menor que el elemento con el salto anterior.

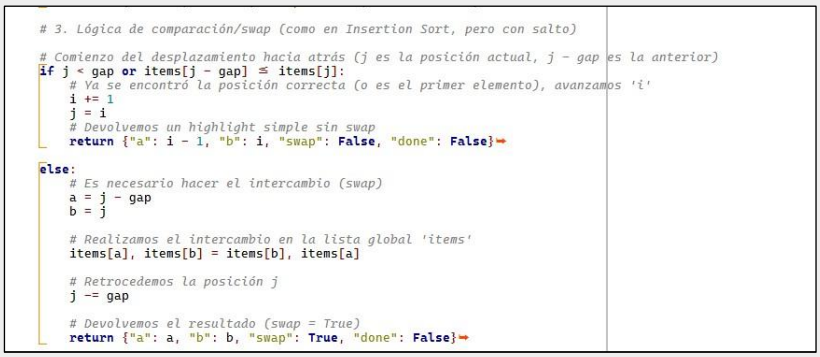


```python
# Estado global del algoritmo
items = []
n = 0
gaps = []    # La secuencia de saltos (Shell, Knuth, etc.)
gap_index = 0 # Índice en la secuencia de gaps
i = 0        # Índice principal (para el bucle externo)
j = 0        # Índice de comparación/swap

def init(vals):

    global items, n, gaps, gap_index, i, j
    items = list(vals)
    n = len(items)

    # 1. Generar la secuencia de gaps (Clásico de Shell)
    gaps.clear()
    h = n // 2
    while h > 0:
        gaps.append(h)
        h //= 2

    # 2. Reiniciar punteros
    gap_index = 0
    i = 0
    j = 0

def step():

    global items, n, gaps, gap_index, i, j

    # Si no quedan más gaps por procesar, el algoritmo terminó
    if gap_index >= len(gaps):
        return {"done": True}

    gap = gaps[gap_index]

    # El Shell Sort usa una "Insertion Sort" con un salto (gap)

    # 1. Avanzar i al primer elemento a insertar (i = gap)
    if i < gap:
        # Iniciamos la fase de Insertion Sort con el gap actual
        i = gap
        j = i

    # 2. Chequeo de fin de la pasada (todos los elementos insertados con este gap)
    if i >= n:
        # Terminó la pasada para este gap.
        gap_index += 1 # Avanzamos al siguiente gap
        i = 0          # Reiniciamos i para el nuevo gap
        j = 0
        return {"a": -1, "b": -1, "swap": False, "done": False} # Paso de control
```

2. Lógica del step()

Cada llamada a step() realiza una de las siguientes acciones y actualiza el estado (i, j, gap_index):

- **Avance de Gap:** Si el puntero i llega al final del arreglo, se avanza gap_index al siguiente salto, y se reinician i y j.
- **Comparación/Fin de Inserción:** Si el elemento actual (items[j]) está en su posición correcta respecto al elemento items[j - gap], el algoritmo avanza i al siguiente elemento a insertar (swap: False).
- **Intercambio (Swap):** Si items[j] es menor, se realiza un intercambio (swap: True), y el puntero j retrocede (j -= gap) para continuar la Inserción hacia atrás.



```
# 3. Lógica de comparación/swap (como en Insertion Sort, pero con salto)

# Comienzo del desplazamiento hacia atrás (j es la posición actual, j - gap es la anterior)
if j < gap or items[j - gap] <= items[j]:
    # Ya se encontró la posición correcta (o es el primer elemento), avanzamos 'i'
    i += 1
    j = i
    # Devolvemos un highlight simple sin swap
    return {"a": i - 1, "b": i, "swap": False, "done": False}

else:
    # Es necesario hacer el intercambio (swap)
    a = j - gap
    b = j

    # Realizamos el intercambio en la lista global 'items'
    items[a], items[b] = items[b], items[a]

    # Retrocedemos la posición j
    j -= gap

    # Devolvemos el resultado (swap = True)
    return {"a": a, "b": b, "swap": True, "done": False}
```

Conclusión

A nivel técnico, el principal logro fue la capacidad de transformar algoritmos iterativos complejos, con bucles anidados (for y while), en una secuencia de micropasos que respetan el contrato de las funciones init() y step(). Esto exigió la adaptación a un modelo de programación basado en el estado persistente, donde se utilizan variables globales (i, j, min_idx) para simular la memoria del algoritmo entre llamadas sucesivas. El reto fue significativo, especialmente en el Selection Sort y el Insertion Sort, donde la lógica de la búsqueda y el desplazamiento interno tuvo que ser segmentada en pasos atómicos y condicionales.

Por otro lado, resulta importante remarcar que el proyecto no estuvo exento de desafíos operativos. Inicialmente, la complejidad radicó en la falta de familiaridad en relación a las herramientas de control de versiones y el entorno de desarrollo. Conceptos como repositorios de GitHub, la mecánica de clone y commit, y la configuración inicial de Visual Studio Code (VS Code) para interactuar con la terminal y el servidor Python, han representado una curva de aprendizaje considerable. En concordancia con lo antes mencionado, en un contexto de complejidad técnica y operativa, las herramientas de Inteligencia Artificial (IA) han sido un recurso eficiente para la totalidad del desarrollo del proyecto y para desglosar la lógica de estado de los algoritmos y simplificar la redacción de las funciones de reporte. La IA actuó como un tutor, clarificando los errores en la sintaxis de Python y ofreciendo explicaciones concisas sobre cómo manejar la persistencia de las variables globales, acelerando significativamente el proceso de depuración y permitiendo centrar la atención en la comprensión conceptual de cada algoritmo implementado.