Universidad Nacional de General Sarmiento

Visualización de algoritmos de ordenamiento

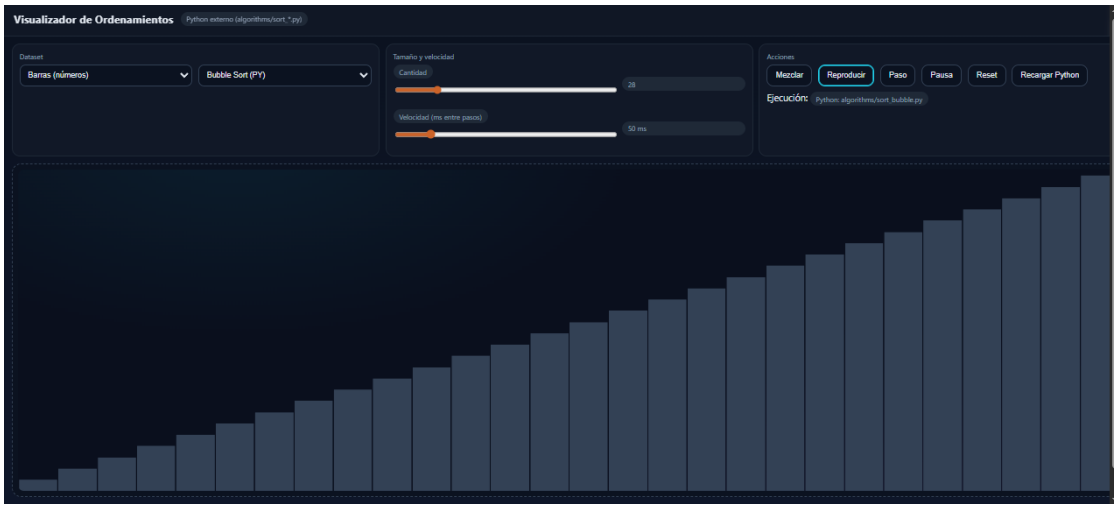
Introducción a la programación(2c-2025)

Integrantes:
Daniel Vanney – danielvanney04@gmail.com
Florencia Campos - flor.v.campos@gmail.com
Leandro Villalba - lean9villalba@gmail.com

Resumen: El trabajo consiste en implementar algoritmos de ordenamiento en Python y ver su ejecución animada en el navegador. Cumpliendo el contrato init(vals) + step () que usa el visualizador.

1. Introducción:
El objetivo de este trabajo es el de solucionar la necesidad de realizar un ordenamiento de diferentes conjuntos de datos mediante la utilización de algoritmos de ordenamientos diseñados a partir del lenguaje de programación Python. Las ejecuciones de estas implementaciones son controladas y visualizadas mediante una interfaz gráfica provista por la Universidad.

A continuación, se presenta imágenes de la interfaz gráfica utilizada para ejecución de los algoritmos y la visualización de uno de los algoritmos implementados:



2. Desarrollo

Ante la necesidad de una solución a esta problemática presentada, se diseñó una solución compuesta por un conjunto de algoritmos que cumplen con los requerimientos de las tareas de ordenamiento de datos y cuya ejecución es compatible con la del visualizador de algoritmos.

Los algoritmos implementados para esta solución son:

- Algoritmo Bubble Sort: Recorre repetidamente la lista comparando pares de elementos adyacentes y si están desordenados, intercambiarlos en cada pasada, el mayor elemento “flota” hacia el final de la lista.
- Algoritmo Insertion: Desplazamiento hacia la izquierda del elemento actual mediante swaps sucesivos.
- Algoritmo Quick: Particionado incremental basado en una pila de subrangos pendientes.
- Algoritmo Selection: Búsqueda del mínimo en la parte no ordenada y colocación en su

posición final.

- Algoritmo Shell: Inserción con saltos decrecientes definidos por una secuencia de espacios

Principales dificultades encontradas:
Durante el desarrollo del proyecto, se encontraron dificultades relacionadas a la forma más adecuada de estructurar los códigos que deben tener los algoritmos, dificultad para solucionar errores inesperados de ejecución, dificultades para comprender cómo ejecutar y optimizar las acciones necesarias para la solución del problema y dificultades para comprobar la compatibilidad de las soluciones diseñadas en el entorno de la interfaz gráfica para realizar las pruebas de necesarias.

Soluciones implementadas:
Se procedió a la revisión e integración de técnicas y estructuras de programación utilizadas previamente en otros proyectos y correspondientes a profesores supervisores del proyecto para solucionar los problemas referentes a la estructuración, errores correcta ejecución y optimización de algoritmos. Se realizó una revisión del material pedagógico alcanzado por la institución para solucionar el problema de la comprobación de la compatibilidad entre las implementaciones desarrolladas y la herramienta de visualización de ejecución de algoritmos.

Decisiones tomadas:
Se decidió organizar los archivos del trabajo desarrollado mediante carpetas presentes dentro del archivo del proyecto. En la carpeta con el nombre “visualizador”, se decidió organizar la carpeta “algorithms”, la cual contiene los archivos correspondientes a los algoritmos desarrollados. En la carpeta algorithms, de encuentran los archivos sort_bubble.py, sort_insertion.py, sort_quick.py, sort_selection.py y sort_shell.py.
Se decidió crear una carpeta bajo el nombre “Documentacion” en los archivos del trabajo desarrollado para guardar los archivos de documentación referentes al trabajo.

Se decidió organizar la codificación de los algoritmos en función de las funciones init() y step().

- Función init(): Función que setea las variables globales que se utilizarán en el algoritmo.
- Función step(): Función que contiene y organiza las acciones del algoritmo, utilizando las variables seteadas de la función init() y retornando un output definido por cada ejecución.

Se decidió utilizar las variables n, i, j, swaps y la lista items como variables globales.
Se decidió que los inputs retornados por la función step() sea un objeto en formato JSON, el cual es utilizado para manipular los datos de las listas a ordenar.
Se decidió utilizar la plataforma Github para realizar un control de versiones, implementar un repositorio y realizar las implementaciones del desarrollo de manera efectiva.

2.1 Descripción general:

La solución completa de este trabajo consiste en un conjunto de implementaciones funcionales que permiten visualizar con un enfoque pedagógico y dinámico el comportamiento de cada uno de estos algoritmos. De esta forma, las implementaciones logran satisfacer los requerimientos impuestos.

2.2 Funcionalidades principales:

Bubble Sort (sort_bubble.py): La idea general es recorrer la lista en múltiples pasadas, comparando elementos adyacentes y moviendo el mayor hacia el final. Se decidió utilizar los punteros i para contar las pasadas y j para recorrer la lista dentro de cada pasada.

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

    if i>=n-1: #si el algoritmo terminó
        return {"done": True}#devuelve que ya no hay mas pasos

    # 1) Elegir índices a y b a comparar en este micro-paso (según tu Bubble).
    a=j
    b=j+1
    swap=False
    # 2) Si corresponde, hacer el intercambio real en items[a], items[b] y marcar swap=True.
    if items[a]>items[b]:
        items[a],items[b]=items[b],items[a]
        swap=True

    # 3) Avanzar punteros (preparar el próximo paso).
    if j+1<n-i-1: #si todavia quedan comparaciones
        j+=1
    else:
        j=0     #reinicia j y avanza i
        i+=1
    return {"a": a, "b": b, "swap": swap, "done": False}
    # 4) Devolver {"a": a, "b": b, "swap": swap, "done": False}.
    #
    # Cuando no queden pasos, devolvé {"done": True}.
```

La función step () no recibe parámetros, pero usa las variables globales (ítems, n, i, j) y devuelve un diccionario con “a” y “b” que son los índices comparados, "swap” que indica si hubo intercambio y “done”, indica si el algoritmo termino.

Dificultades: controlar los límites de los índices para evitar errores fuera de rango.

Conclusión: Bubble sort tiene un buen punto de partida para introducirse en el mundo de los algoritmos de ordenamientos, preparándonos para implementar métodos más complejos.Resulta muy didáctico para comprender los fundamentos de los algoritmos de ordenamiento y el manejo de lista, índices y condicionales en programación.
La visualización paso a paso fue fundamental para detectar errores y entender mejor el flujo interno del algoritmo.

Sort_Insertion.py:

Descripción del algoritmo:

Se implementa el algoritmo Insertion_Sort.py para poder integrarlo en el visualizador de algoritmos. La idea general de este algoritmo es realizar un ordenamiento de un conjunto de valores desordenados.
Este algoritmo de ordenamiento realiza sus funciones tomando un elemento perteneciente a la lista cada vez que se ejecuta y lo inserta en la posición correcta entre los elementos que ya fueron procesados y ordenados. En cada

ejecución del algoritmo, se compara el elemento del índice actual con los elementos de los índices anteriores e inserta el valor del elemento de la lista tomado en el índice de lista que le corresponde, mediante un swap de valores.

Funcionalidad:
Durante el desarrollo del algoritmo, se decidió utilizar la variable i para indicar el valor del elemento actual que será insertado y ordenado. Se decidió utilizar la variable j como cursor de desplazamiento hacia los primeros índices.
A raíz de que el visualizador necesita realizar solo una ejecución de la función step() para realizar una ejecución del algoritmo, se decidieron realizar cambios en el código del algoritmo. Estos cambios son los siguientes: Setear j=i para la primera ejecución en la que se realice un desplazamiento por el algoritmo; Que cada ejecución realice un único swap entre valores; Realizar un incremento en el valor de i al finalizar un desplazamiento de forma efectiva. Con el fin de mantener una coherencia entre las ejecuciones de la función step(), se utilizan las variables ítems, i, j y n para cumplir con las exigencias y compatibilidades con el visualizador.

```
items = []
n = 0        # Cantidad de elementos que tendrá la lista y que serán ordenados
i = 0      # índice del elemento que queremos insertar
j = None   # cursor de desplazamiento hacia la izquierda (None = empezar)
swaps = 0

def init(vals):
    global items, n, i, j, swaps
    items = list(vals)
    n = len(items)
    i = 1      # insertion sort empieza en el segundo elemento
    j = None
    swaps = 0

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

    #Todo
    # - Si i >= n: devolver {"done": True}.
    if i >= n:
        print("Insertion Sort – swaps totales:", swaps)
        return {"a": i, "b": j, "swap": False, "done": True}

    # - Si j es None: empezar desplazamiento para el items[i] (p.ej., j = i) y
devolver un highlight sin swap.
    if j is None:
        j = i
        if j > 0:
            return {"a": j-1, "b": j, "swap": False, "done": False}
        else:
```

```
            return {"a": 0, "b": 0, "swap": False, "done": False}

    # - Mientras j > 0 y items[j-1] > items[j]: hacer UN swap adyacente (j-1, j) y
devolverlo con swap=True.
    if j > 0 and items[j-1] > items[j]:
        items[j-1], items[j] = items[j], items[j-1]
        swaps += 1
        j -= 1
        return {"a": j, "b": j+1, "swap": True, "done": False}

    # - Si ya no hay que desplazar: avanzar i y setear j=None.
    i += 1
    j = None
    return {"a": i, "b": j, "swap": False, "done": False}
```

En el código se compone de la siguiente manera:

- Def init(vals): Función que devuelve las variables globales de seteadas a sus valores iniciales. Se implementa debido a que es necesario el seteo de cada una de las variables del algoritmo antes de la ejecución de la función step(). Estas variables son:
    - n: Cantidad de elementos almacenados en la lista a ordenar.
    - j: Cursos de desplazamiento de hacia la izquierda, el cual permite determinar la posición en la cual va a ser ubicado el elemento a ordenar.
    - i: Valor del elemento de la lista que será ordenado.
    - items: Lista que se utiliza para almacenar las variables anteriormente mencionadas y utilizarlas en otras funciones del algoritmo.

- Def step(): Función de ejecución del algoritmo. Se integra porque es necesaria la integración de una función que contenga las instrucciones lógicas del algoritmo. En esta función se aplican las condiciones lógicas mediante if y else. Sesta función se ejecuta una vez por cada run del algoritmo. Devuelve un objeto JSON con las claves “a”, “b”, “swap” y “done”, cuyos valores serán evaluados en el visualizador y varían dependiendo de las condiciones lógicas implementadas.
    - a: Clave que almacena valores int. Se usa para almacenar el índice del cursor de desplazamiento que mueve el elemento hacia la izquierda.
    - B: Clave que almacena valores int. Se usa para almacenar el valor del elemento que será ordenado.
    - Swap: Clave que almacena valores tipo booleano. Se usa para determinar si el valor de las claves a y b fueron intercambiados entre sí. Si toma el valor True, significa que los valores fueron cambiados. Si toma el valor False, significa que los valores no fueron cambiados.
    - Done: Clave que almacena valores tipo booleano. Se usa para determinar si la ejecución el algoritmo debe continuar o finalizar. Si el valor es True, el algoritmo ha finalizado. Si el valor es False, el algoritmo ha terminado.

SELECTION SORT

Este algoritmo consiste en recorrer la lista iterativamente para identificar el elemento mínimo de la porción no ordenada y colocarlo en su posición correcta mediante un intercambio. Mediante la función `init(vals)` inicializamos las variables necesarias para comenzar el algoritmo: índices `i, j` y `min_idx`, el estado de la fase actual (“`buscar`” o “`swap`”) y el contador de intercambios. Esto con el fin de que todas las variables que usa el algoritmo estén correctamente reiniciadas y en un estado válido antes de comenzar a ordenar.
En la función `step ()` ejecuta un único paso del algoritmo alternando entre dos fases:
Fase “`buscar`” : recorre con j para encontrar el indice mínimo, en cada iteración se compara el elemento actual con el mínimo encontrado hasta ese momento

```
if j < n:
    j_actual = j
    if items[j] < items[min_idx]:
      min_idx = j
    j+=1
```

El recorrido sigue avanzando a menos que j alcance a n, si esto ocurre significa que la búsqueda terminó y se encontró al mínimo de todo el recorrido, una vez completa esta fase, el algoritmo cambia automáticamente a la fase “swap”

Fase “swap” : intercambia el mínimo encontrando con la posición i si corresponde, y avanza a la siguiente pasada. Esto ocurre únicamente cuando el índice del mínimo encontrado es distinto al índice actual i y se realiza de esta manera:

```
if min_idx != i:
          items[a], items[b] = items[b], items[a]
```

Esta línea toma los valores en las posiciones a (que es `i`) y `b` (que es `min_idx`) y los intercambia.
Todo este proceso se va a repetir hasta que la lista quede completamente ordenada.

QUICK SORT

Consiste en ordenar una lista, dividiéndola en segmentos y subarreglos alrededor de un separador (último elemento de cada segmento) y colocando los elementos menores a su izquierda y los mayores a su derecha.
Los segmentos pendientes se almacenan en una pila como tuplas `(inicio, fin)`, y la partición se realiza con dos punteros: i, que marca la frontera de los elementos menores que el separador y `j`, que recorre el segmento actual comparando cada elemento con el separador. Se cuenta con dos fases:
Fase vacía “” : indica que se toma un nuevo segmento de la pila para comenzar a ordenar; si el segmento tiene un solo elemento se pasa inmediatamente al siguiente.

```
separador = items[fin]        se elige el separador/pivote
    i = inicio - 1                frontera de elementos menores
    j = inicio                    índice que recorre el segmento
```

Fase “`particionar`”: controla el proceso de comparación y swap dentro del segmento. En

esta fase cada elemento en `j` se compara con el separador y si corresponde, se intercambia con `i`.

```
items[i], items[j] = items[j], items[i]    swap normal si i ≠ j
```

Cuando `j` alcanza el separador, se realiza un swap final para colocar el separador en su posición correcta, asegurando que todos los elementos menores queden a la izquierda y los mayores a la derecha.

```
items[i], items[fin] = items[fin], items[i]    swap final si necesario
```

Luego se generan los subsegmentos izquierdo y derecho, que se agregan a la pila solo si contienen más de un elemento, repitiendo el proceso hasta que la lista quede ordenada.

```
pila.append((inicio, i - 1))      subsegmento izquierdo
pila.append((i + 1, fin))         subsegmento derecho
```

SHELL SORT

Shell Sort ordena la lista mediante un método de inserción con saltos decrecientes, comenzando con un salto grande y reduciéndolo a la mitad hasta llegar a 1. La implementación está adaptada al visualizador mediante la función `step()`, de manera que cada llamada ejecuta un único micro-paso y permite observar cada movimiento de los elementos.
El algoritmo opera en tres fases principales.
Fase `"nuevo_salto"`: Determina el valor inicial del salto y reinicia el índice de recorrido; si el salto llega a cero, se pasa a la fase `"fin"` que indica la finalización del algoritmo:

```
indice = salto           inicializa el índice para el nuevo salto
```

Fase `"nuevo_indice"`: Se selecciona el valor actual (guardado) y se prepara el índice interno para desplazar elementos dentro del subarreglo correspondiente al salto:

```
guardado = items[indice]       valor que se quiere insertar
     indice_interno = indice        índice interno para desplazamiento
```

Fase `"bucle_interno"`: Realiza los movimientos dentro del subarreglo: compara guardado con los elementos separados por el salto y los desplaza hacia adelante si son mayores, contando cada swap:

```
     items[indice_interno] = items[indice_interno - salto]
#desplazamiento
     conteo_swaps += 1
```

Cuando no quedan más elementos que mover, guardado se coloca en su posición correcta y se avanza al siguiente índice, reiniciando el ciclo para el subarreglo actual o reduciendo el salto:

```
     items[indice_interno] = guardado   #colocar guardado en posición
correcta
```

3. Conclusiones:

La realización del trabajo ayudó a comprender en profundidad el funcionamiento técnico de diferentes algoritmos de ordenamiento, teniendo en cuenta la lógica, la estructura y complejidad que estos presentan. El requerimiento de construir el código de los algoritmos en base a las funciones init() y step() ayudó a lograr un óptimo desarrollo y estructuración de estos. La implementación de la interfaz gráfica posibilita la visualización y comprensión de las ejecuciones de las implementaciones desarrolladas, logrando una comprensión más dinámica por parte del usuario del programa. El trabajo potenció la práctica de lógica, variables, listas e índices, condicionales, bucles, booleanos, manejo de objetos JSON y la utilización de herramientas de repositorio y control de versiones como Github.