INTRODUCCIÓN A LA PROGRAMACIÓN – COM 04

Profesores:

- Omar Argañaras
- Nancy Nores

Alumnos:

- Mateo Flores
- Agustín Heizenreder

Fecha de entrega:

- 26/11/2025

INTRODUCCIÓN

El trabajo pide en implementar al menos tres estrategias de algoritmos de ordenamiento en un repositorio en GitHub que contiene los algoritmos en Python, que son: short_bubble.py – short_selection.py – short_insertion.py – short_template.py (este último es opcional)

El objetivo de cada uno de estos es poder implementar y lograr el funcionamiento de cada algoritmo para su objetivo, que es poder ver el algoritmo funcionando y animado en paso a paso en cuanto a su función. Todo este trabajo se trata de realizar una visualización de algoritmos de ordenamiento.

Las tecnologías utilizadas para realizar el funcionamiento de este trabajo práctico fue el programa Visual Studio Code v.1.105 con Python v.3.12.0, y las IA's Copilot y ChatGPT.

DESARROLLO

Cada algoritmo tiene su propia función **init(vals)**. Esta función se encarga de inicializar las variables que se utilizan a lo largo del algoritmo. Cada algoritmo tiene sus propias variables inicializadas en la función **init()**, ya que cada una tiene distintas variables y funcionalidades.

Bubble sort () es de una versión de un algoritmo de ordenamiento que se enfoca en ordenar una lista comparando el primer elemento con todos los de la lista; si alguno es mayor que el elemento que le sigue en la lista, estos se intercambian de posición, dejando a los mayores en la lista hacia el final ordenándolos de una manera ascendente.

Comienza desde el primer elemento de la lista; compara ese elemento con el siguiente, y si el primer elemento es mayor que el segundo, se intercambian, luego se vuelve a comparar con los siguientes elementos que siguen en la lista hasta llegar al final, entonces en la primera pasada el elemento mayor ya esta ordenado al final, así que cuando el proceso se repite, ya no se incluye al último elemento mayor.

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

➢ La función **init()** de Bubble inicializa las variables que vamos a utilizar, la lista **ítems** con los elementos a ordenar que recibe **vals**, y la cantidad de elementos de la lista **n**.

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

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

➢ En la función step(), esta condición termina el algoritmo cuando ya se hicieron las suficientes pasadas y se ordenaron todos los elementos. Devuelve un diccionario que indica que el programa termino.

```
    a = i
    b = i + 1
    swap = False

    if items[a] > items[b]:
        items[a], items[b] = items[b], items[a]
        swap = True

    i += 1
```

➢ Se les asigna a las variables **a** y **b** el valor **i**, y se inicializa una variable que indica si ubo cambio **swap.** Luego hay una comparación de los dos elementos en su respectiva posición. Si el elemento de la izquierda es mayor al de su derecha, entonces se cambian, ordenado de esa manera cada elemento en la lista, y una vez terminada la condición, el valor **i** va incrementando.

```
    if i>=n-1-j:
        i = 0
        j += 1

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

➢ La condición indica que, si la pasada **j** llego al final, el valor **i** se reinicia y **j** incrementa, lo que significa que el elemento más grande ya está al final de la lista. Por último, retorna un diccionario con la información del paso **step()**.

Insertion sort() es otro algoritmo de ordenamiento que comienza en la segunda posición de la lista y se compara con los elementos anteriores. Si encuentra un elemento mayor lo mueve a su posición derecha, y repite ese proceso hasta encontrar la posición correcta de su elemento, dejando el valor en su respectivo lugar. Repite este intercambio de manera continua hasta que no haya más elementos a ordenar.

```
def init(vals):
    global items, n, i, j
    items = list(vals)
    n = len(items)
    i = 1
    j = None
```

➢ Se declaran las listas, la cantidad de elementos de la lista **n**, y las variables globales. **J** empieza como tipo *none* lo que indica que aún no empezó a comparar, e **i** comienza en la segunda posición de la lista.

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

    if j is None:
        j = i
        return {"highlight": True}

    a = j
    b = j-1
    swap = False
```

➢ Se definen las variables globales en el step(). Si **i** supera o iguala la lista **n**, entonces ya habrá recorrido la lista y el algoritmo termina retornando un diccionario **done: True**.

Si **j** es igual a *none*, es que indica un comienzo para una nueva iteración y se establece al elemento **j** en **i**. Luego devuelve **highlight:True** para que se pueda resaltar el elemento actual que este posicionado. Fuera de los condicionales, se definen dos variables a las que se les asigna **j** y **j-1**, y se inicializa la variable **swap** como **False**.

```
    if j > 0 and items[b] > items[a]:
        items[b], items[a] = items[a], items[b]
        swap = True
        j -= 1
        return {"a": a, "b": b, "swap": swap,  "done": False}
    else:
        i += 1
        j = None
        return {"a": a, "b": b, "swap": swap,  "done": False}
```

➢ Si se cumple la condición, los elementos se intercambian y si **j>0** para no salir del rango de la lista, y se retrocede para que vaya comparando hacia la izquierda en el próximo paso **j-=1**. Luego se retorna el diccionario con la información de los índices comparados.

Si la condición no se cumple, significa que el elemento ya esta en su correcto lugar, entonces se avanza a la siguiente posición **i+=1**, y se reinicia **j**. Por ultimo se vuelve a retornar la información de el diccionario.

Selection sort() es un algoritmo que ordena una lista con un mínimo en la lista, y si encuentra un elemento menor, lo intercambia con el primer elemento de la lista, y ese nuevo elemento será el mínimo en la lista.

```
def init(vals):
    global items, n, i, j, min_idx, fase
    items = list(vals)
    n = len(items)
    i = 0
    j = i + 1
    min_idx = i
    fase = "buscar"
```

➢Se inicializan las variables globales en la función init() de **selection**, y también las listas. Pero se utiliza una variable nueva **min_idx** que es igual al índice con menor valor entre los elementos de la lista.

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

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

➢ Se declaran las variables globales y hay una condición de que si **i** es mayor o igual a la longitud de la lista, quiere decir que la ha recorrido, el algoritmo termina y devuelve **done: True**.

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

    a = i
    b = min_idx
```

➢ Compara el elemento actual con la posición **j item[b]** con el elemento mínimo actual en la posición **min_idx item[a]**. Si el elemento es menor, **j** se vuelve **min_idx**. **J** se incrementa en 1 para seguir buscando.

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

➢ Si el índice mínimo **min_idx** es diferente y no está en la posición actual **i**, se intercambian el elemento en la posición **i** con **min_idx**. Por último, **i** se incrementa para seguir avanzando en los siguientes elementos no ordenados, y retorna el diccionario con el estado de los índices actuales.

Algunos problemas y dificultades que nos encontramos durante el proceso de realización del trabajo que pudieron solucionarse están relacionados con la

comprensión y entendimiento de algunas funciones de los algoritmos con sus respectivos *returns* con los diccionarios. Pero al final pudimos solucionarlo con ayuda de herramientas externas como el internet o ChatGPT para la comprensión de lo que se pedía hacer en cada programa y en como realizarlo, utilizando las variables, listas y funciones ya definidas en el algoritmo.

También fue difícil comprender el uso de algunas variables, como el **min_idx** de selection sort(), o el uso de valores booleanos en algunos casos de insertion sort(). Pero finalmente con explicaciones externas y relacionadas con le tema, y pruebas constantes del funcionamiento del programa, logramos entender el uso y funcionamiento de las variables con esa ayuda.

Lo que nos hubiera gustado haber hecho en el trabajo es poder haber agregado un algoritmo de ordenamiento más, como el **sort template()**, pero a causa de la falta de tiempo no pudimos realizar otro algoritmo de ordenamiento para el trabajo.

CONCLUSIÓN

A lo largo de este trabajo practico, aprendimos que son los algoritmos de ordenamiento y como funcionan tres de ellos, siendo **bubble sort, insertion sort** y **selection sort** los algoritmos aprendidos en este caso. También aprendimos las funcionalidades de cada función de los algoritmos, como el **init()** y el **step()**, y como cada uno si bien tienen el mismo objetivo de ordenamiento de elementos, los algoritmos lo hacen a su manera siendo diferentes la una con la otra en cuanto al método de para ordenar.

Aun así, el proceso de realización y comprensión del trabajo y sus algoritmos nos resultó dificultoso, ya que esa era nuestra principal dificultad en la comprensión del trabajo. Se nos dificultaba poder entender el uso de las variables y funciones en el programa, pero a medida que fuimos analizando el código línea por línea, pidiendo y consultando ayudas para la mejor comprensión y realización de los algoritmos, en nuestra investigación, pudimos resolver y entender las dudas que teníamos acerca de este trabajo.