Licenciatura en Sistemas

Trabajo Práctico:

Algoritmos de ordenamiento

Introducción a la Programación

(segundo semestre, 2025)

Resumen: *El trabajo consiste en realizar tres (o más) diferentes tipos de algoritmos de ordenamiento que se deben ejecutar de manera correcta en un visualizador.*

Integrantes:

- Galeano Liz, lizzgaleano54@gmail.com
- Moreno Giuliana, morenogiuliana015@gmail.com
- Poisa Ariana, arianacandela2007@gmail.com

Docentes:

- Montiel Santiago
- Velazquez Luca
- Martinez Nora

Nota:

ÍNDICE:

1. Introducción

En el siguiente trabajo se realizaron una serie de algoritmos de ordenamiento en Python, con el objetivo de ejecutarlos a través de un visualizador que fue provisto por los docentes. El visualizador permite observar paso a paso cómo funciona cada algoritmo, mostrando comparaciones, desplazamientos e intercambios entre los

elementos de la lista.

La consigna principal consistió en implementar correctamente el funcionamiento interno de cada algoritmo, respetando un contrato basado en dos funciones: init(vals) y step().

Cada integrante del grupo trabajó en la codificación y comprensión de distintos algoritmos, con el fin de lograr un funcionamiento correcto y una visualización clara del proceso de ordenamiento. El trabajo incluyó dificultades de interpretación del algoritmo paso a paso, como también del manejo de variables globales, estructuras de control y casos límite.

A continuación se detalla el desarrollo del trabajo, las decisiones tomadas y las soluciones implementadas.

2. Desarrollo

En esta sección se explicará cómo fue resuelto el trabajo práctico, qué pasos seguimos y qué fue lo que aprendimos de ello. El objetivo del desarrollo es mostrar el proceso completo que realizamos para poder implementar los algoritmos de ordenamiento y su correcta ejecución.

2.1 Descripción general:

El trabajo consistió en implementar varios algoritmos de ordenamiento dentro del marco del visualizador. Cada algoritmo se encuentra en un archivo independiente y contiene dos funciones principales:

- `init(vals)`: inicializa las variables necesarias según el algoritmo.
- `step()`: ejecuta un solo paso del algoritmo y devuelve un diccionario con información sobre comparaciones, intercambios o el estado final del proceso.

Este enfoque permitió que los algoritmos se ejecuten de manera incremental, paso a paso, facilitando la visualización y el análisis de su comportamiento interno.

Para cada algoritmo (Bubble, Selection, Insertion y Quick) trabajamos en: definir variables que avancen paso a paso, controlar los límites de la lista para evitar errores, realizar los intercambios y devolver siempre la información al visualizador. Para lograrlo, tuvimos que pensar bien qué variables usa cada algoritmo, cómo avanzan las variables y en qué momento debe hacerse cada comparación o intercambio.

2.2 Funcionalidades principales:

ALGORITMO BUBBLE:

El algoritmo Bubble Sort recorre la lista comparando el número que está a la izquierda con el que está a su derecha. Si se encuentran en el orden incorrecto, se los intercambia de lugar. De esta manera, en cada pasada el número más grande va quedando al final de la lista.

Decisiones tomadas:

- Se utilizó la variable `i` para contar cuántas pasadas ya se completaron.
- La variable `j` avanza elemento por elemento dentro de cada pasada.
- Cuando `j` llega al final (`n - i - 1`), se reinicia y aumenta `i`.

CÓDIGO:

```
def step():
    global items, n, i, j
    #Cuando no queden pasos, devuelve {"done": True}
    if i >= n - 1 or n <= 1:
       return {"done": True}
    a = j
    b = j + 1
    swap = False     #Inicializa la bandera de intercambio de False

    if b < n - i:
        #Comparación: ¿el elemento 'a' es mayor que el elemento 'b'?
        if items[a] > items[b]:
            items[a], items[b] = items[b], items[a]
            swap = True     #Avisa que hubo un intercambio

        # Se incrementa j para pasar al siguiente par de elementos
```

```
        en la próxima llamada a step()
        j += 1

        #Pregunta si j llegó al final de la parte no ordenada de la
        lista.
        if j >= n - 1 - i:
            i += 1
            j = 0

    #Devuelve al visualizador qué elementos se miraron, si se
    intercambiaron y si termino el recorrido
    return {
        "a": a,
        "b": b,
        "swap": swap,
        "done": False
    }
```

Parámetros y valores devueltos:

- No recibe parámetros en `step()`, sigue trabajando con los valores que el algoritmo guardó anteriormente y no necesita que se los pasen cada vez.
- `step()` compara `items[j]` con `items[j+1]`.
- Si `items[j]` es mayor, se hace el swap.
- Si `j` ya terminó la pasada, se reinicia y avanza `i`
- Devuelve los índices comparados, si hubo swap y si terminó el recorrido.

Complicaciones:

- Calcular mal cuando termina una pasada, provoca que `i` no aumente cuando corresponde y se repitan pasos innecesarios.
- Detección mal del `return {“done”:True}` lo que hace que el algoritmo termine antes de completar el ordenamiento o que nunca finalice por la condición mal planteada.
- Olvidar reiniciar j (`j=0`) cuando aumenta i (`i+=1`), esto provoca que la siguiente pasada no empiece desde el principio, impidiendo que se ordene la lista.
- Confundir las funciones de `j` e `i`, hace que el algoritmo compare elementos incorrectos o no avance correctamente la lista.

ALGORITMO INSERTION:

El algoritmo recorre la lista de izquierda a derecha y, para cada elemento, lo compara con los anteriores hasta encontrar su lugar correcto. A medida que avanza, va formando una parte inicial que siempre queda ordenada: toma un elemento, lo "inserta¨ donde corresponde dentro de lo ya ordenado y luego continúa con el siguiente. Repite este proceso hasta que todos los elementos estén acomodados en orden.

Decisiones tomadas:

- **Items**: Guarda la lista de valores que se va a ordenar.
- **n**: Almacena la cantidad total de elementos de la lista.
- **i**: Marca el índice del elemento actual que queremos insertar en su posición correcta.
- **j**: Funciona como cursor que se desplaza hacia la izquierda comparando y moviendo elementos (y empieza en `None` para indicar que inicia una nueva inserción).

CÓDIGO:

```
def step():
    global items, n, i, j
    # Si ya terminamos, para saber si no nos excedimos del largo de nuestra lista [items]
    if i >= n:
        return {"done": True}

      # Al comenzar con un nuevo elemento, comprobar que todavía estamos dentro de lo permitido
    if j is None:
        j = i
        return {"a": j-1, "b": j, "swap": False, "done": False}

    # Comparar e intercambiar si hace falta
    if j > 0 and items[j-1] > items[j]:
        items[j-1], items[j] = items[j], items[j-1]
```

```
        j -= 1
        return {"a": j, "b": j+1, "swap": True, "done": False}

    # Si no hay más intercambios, avanzar al siguiente elemento
    i += 1
    j = None
    return {"a": i-1, "b": i, "swap": False, "done": False}
```

Parámetros y valores devueltos

- No recibe parámetros: usa las variables globales inicializadas en `init()`.
- Si `i` llegó al final de la lista, devuelve `done=True`.
- Cuando se empieza una nueva inserción, coloca `j = i` y devuelve los índices a comparar.
- Compara `items[j]` con `items[j-1]`: si el de la izquierda es mayor, hace el swap y mueve `j` hacia atrás.
- Cuando ya no corresponde intercambiar, avanza `i` y reinicia `j` para el próximo elemento.
- Siempre devuelve los índices comparados (`a`, `b`), si hubo swap y si terminó.

Complicaciones

- Equivocaciones al reiniciar los valores de las variables, haciendo que el código no pueda avanzar.
- Mal posicionamiento del **return,** obstaculizando el recorrido del código, ya que hacía que se corte antes de siquiera comenzar.
- Exceder el límite de la lista (osea su largo) haciendo que el código no haga el ordenamiento.
- Incrementar mal ¨ **i** ¨ causando que el algoritmo se saltee elementos.

ALGORITMO SELECTION:

El algoritmo Selection Sort ordena una lista, buscando en cada pasada el valor mínimo dentro de la parte no ordenada y colocándolo en la posición correcta.
En la primera pasada se busca el número más chico de toda la lista y se lo coloca en el índice 0.

En la segunda pasada se busca el número más chico del resto de la lista y se lo coloca en el índice 1.
Así sucesivamente hasta completar todas las posiciones.
De esta manera, en cada pasada se fija un valor en la posición correcta, mientras la parte no ordenada se hace cada vez más pequeña.

Decisiones tomadas:

- Se utiliza la variable `i` para indicar la posición donde se colocará el próximo mínimo (cantidad de pasadas completadas).
- Se utiliza la variable `j` para recorrer el resto de la lista buscando el elemento más chico.
- Se usa `min_index` para guardar el índice donde se encontró el valor mínimo de la pasada.
- Cuando `j` llega al final de la parte no ordenada (n – 1), se realiza el intercambio entre `i` y `min_index`, se cierra esa pasada y avanza i.

CÓDIGO:

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

    # Cuando ya no quedan pasadas por realizar → devuelve done:
True
    if i >= n - 1:
        return {"done": True}

    # Si j == i significa que recién empieza una nueva pasada
    if j == i:
        min_index = i   # Inicializamos el mínimo como el primer
elemento de la parte no ordenada

    # Comparación: si el elemento en j es menor que el mínimo
actual, se actualiza min_index
    if items[j] < items[min_index]:
        min_index = j

    # Avanzamos j para seguir buscando el mínimo en el resto de la
lista
    j += 1
```

```
    # Si j llegó al final de la lista → terminó la búsqueda del
mínimo en esta pasada
    if j == n:
        # Intercambio: colocamos el mínimo en la posición i
        items[i], items[min_index] = items[min_index], items[i]

        # Pasada finalizada → avanzamos i y reiniciamos j
        i += 1
        j = i

    # Devuelve al visualizador qué índice es el actual, cuál es el
mínimo encontrado y si se terminó la pasada
    return {
        "i": i,
        "j": j,
        "min_index": min_index,
        "done": False   }
```

Parámetros y valores que Devuelve:

- El índice `i` (pasada actual).
- El índice `j` (posición actual dentro de la pasada).
- `min_index` (posición del mínimo encontrado).
- `done` (True cuando se terminó todo el algoritmo).

Complicaciones:

-

ALGORITMO QUICK:

El algoritmo funciona dividiendo la lista en partes más pequeñas. Primero elige un elemento llamado pivote, luego separa los demás en dos grupos: los que son menores que el pivote y los que son mayores.
Después coloca al pivote en su posición correcta dentro de la lista. Una vez hecho esto, el algoritmo repite el mismo proceso de forma recursiva con la parte izquierda y la parte derecha de la lista hasta que todo queda ordenado.

Decisiones tomadas:

- Guardar las variables internas como globales (`i`, `j`, `pivot`, etc.) para que el algoritmo recuerde su progreso entre pasos.
- Elegir el pivote como el último elemento del segmento, por simplicidad
- Implementar partición de manera que: **i** marca el límite de “menores”, **j** recorre y compara).
- Usar un `estado` interno ("partition" / None) para saber si se está empezando un segmento o procesándolo.
- Dividir los subsegmentos a ordenar después de fijar el pivote, agregándolos a la pila para procesarlos más adelante.

CÓDIGO:

```
def step():

    global items, rangos, i, j, pivot, estado, done

    global current_izquierda, current_derecha

    if done:

        return {"done": True}

    # Si no quedan más partes por ordenar y no estamos a mitad de
proceso terminó de manera definitiva

    if not rangos and estado is None:

        done = True

        return {"done": True}

    # Si no estamos en medio de una partición, arrancamos un nuevo
segmento de trabajo (se toma uno de la pila)

    if estado is None:

        current_izquierda, current_derecha = rangos.pop()

# Sacamos un pedazo de la lista para ordenar y elegimos pivot

        pivot = items[current_derecha]

        i = current_izquierda - 1

# i marca el "límite" de la zona de elementos menores

        j = current_izquierda
```

```
# j recorre el segmento comparando cada elemento

        estado = "partition"

# Cambiamos al modo de partición

        return {"a": current_derecha, "b": j, "swap": False, "done":
False}

    # Si estamos en modo partición, seguimos procesando el segmento
actual

    if estado == "partition":

        # Mientras j no llegue al pivote, seguimos comparando
elementos

        if j < current_derecha:

            # Si el elemento es menor (<= pivote), lo movemos al
sector de elementos menores

            if items[j] <= pivot:

                i += 1

# se realiza un intercambio para poner el elemento en su lugar
correspondiente

                items[i], items[j] = items[j], items[i]

                j += 1

                return {"a": i, "b": j-1, "swap": True, "done":
False}

            # Si el elemento es más grande que el pivote, se deja
donde está y seguimos recorriendo

            else:

                j += 1

                return {"a": j-1, "b": None, "swap": False, "done":
False}

        # Si j ya llegó hasta el final, se coloca el pivote en su
posición fija
```

```
        items[i+1], items[current_derecha] = items[current_derecha],
items[i+1]

        pos_fija = i + 1      # Donde quedó finalmente el pivote

        # Si hay una parte a la izquierda del pivote que aún no está
ordenada, la agregamos a la pila de pendientes y se repite lo mismo
para la derecha

        if pos_fija - 1 > current_izquierda:

            rangos.append((current_izquierda, pos_fija - 1))

        if pos_fija + 1 < current_derecha:

            rangos.append((pos_fija + 1, current_derecha))

        # Terminamos con este segmento, así que en la próxima
llamada se inicia uno nuevo

        estado = None

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

Parámetros y valores devueltos

- Se aplica **current_izquierda y current_derecha** para saber el rango exacto del segmento activo.
- Se usa **pivot, i y j** para manejar la lógica de partición (comparaciones, swaps y posición final del pivote).
- Se utiliza **estado** para decidir si debe iniciar un nuevo segmento o continuar particionando.
- Devuelve **done = True** cuando ya no quedan segmentos pendientes.
- Devuelve **a y b** indicando qué índices fueron comparados o usados en el paso actual.
- Devuelve **swap = True/False** según si ocurrió un intercambio en ese paso.
- Cada devolución refleja exactamente en qué parte del proceso está: inicio de segmento, comparación, swap o fijación del pivote.

Complicaciones

- Pérdida del pívot, **j** e **i** comenzaron a avanzar en destiempo perdiendo así su posición correcta y haciendo que deje de funcionar el código.
- El mal nombramiento de variables dificulto la realización del algoritmo, ya que, había nombres que se querían utilizar que no eran reconocidos por Python
- Resultó difícil entender paso a paso cómo se identificaba el índice del valor mínimo en cada recorrido.
- Hubo dificultades para visualizar qué comparación realizaba el código en cada iteración.
- Al principio no se comprendía con claridad cómo iba cambiando la lista después de cada intercambio.

3. Conclusión

A lo largo del trabajo se puede evidenciar que el código presentó un desafío para las integrantes del grupo. El entendimiento del código, la realización del mismo y las dificultades que se presentaron a lo largo del proceso se tomaron como maneras de aprendizaje que en un futuro serán de utilidad. El trabajo nos enseño a utilizar herramientas como GitHub y el manejo de los repositorios, que si bien fueron difíciles de entender, son importantes de saber manejar por la utilidad que representan en el ámbito laboral del programador. Otras herramientas útiles presentadas por el trabajo práctico fueron los distintos algoritmos de ordenamiento que se fueron implementando (Bubble Sort, Insertion Sort, Selection Sort y Quick Sort); estos son fundamentales al trabajar con listas, ya que, sus métodos garantizan la eficacia en su ordenamiento y optimizan el trabajo del programador.

En conclusión, el trabajo que fue presentado por los docentes es rico en conocimientos para implementar en un futuro, representa desafíos que ayudan en el crecimiento estudiantil e implementa los conocimientos que se fueron adquiriendo a lo largo del cuatrimestre.

Anexo:

El trabajo es entregable cuando cumple con lo siguiente:

● (Obligatorio) Carpeta /algorithms/ con al menos 3 archivos funcionando (ej: sort_bubble.py, sort_selection.py, sort_insertion.py).
● (Obligatorio) Informe detallado documentando los algoritmos implementados junto con las decisiones tomadas, implementaciones aplicadas y dificultades encontradas.

● (Obligatorio) Un README.md breve con: integrantes, algoritmos implementados y una nota corta sobre decisiones de implementación.
● (Opcional) Algoritmos extra (sort_quick.py, sort_merge.py, sort_shell.py, etc)