TRABAJO PRÁCTICO FINAL

ALGORITMOS DE ORDENAMIENTO

Farias Nicolás
COM-04

MATERIA:

Introducción a la Programación

DOCENTES:

- Argañaras, Omar
- Nores, Nancy

FECHA DE ENTREGA:

26/11/2025

Introducción

En este trabajo práctico se realizaron tres algoritmos de ordenamiento: el ordenamiento de burbuja, por selección y por inserción, los cuales fueron programados de manera que puedan ser ejecutados por un visualizador, el cual muestra una animación del funcionamiento de cada uno de ellos paso a paso mediante barras verticales de distintos tamaños que se van ordenando. A continuación, se explicará detalladamente cada algoritmo, el funcionamiento del visualizador, los cambios que se realizaron, y las dificultades encontradas.

Desarrollo

Visualizador:

La web del visualizador a utilizar requiere la programación de dos funciones:
• `def init(vals)`, la cual toma los valores aleatorios y mezclados proporcionados por el visualizador, y llama a las variables globales (`items, n, i, j`). Se ejecuta una vez al comenzar o tras mezclar y guarda una copia de los valores desordenados mencionados anteriormente en la variable *ítems*, guarda la cantidad de valores en la variable *n*, e inicializa los punteros o estado interno (i, j, min_idx, pila, etc.).
• `def step()`, función que realiza un solo micro-paso, se llama muchas veces y devuelve un diccionario específico para el correcto funcionamiento de la animación. Esta utiliza, además de las variables globales (`items, n, i, j`), los valores `"a"` y `"b"` (valores a comparar en el micro-paso), y las variables `"swap"`, la cual determina si los valores (barras) son reordenados y cambiados de posición, y `"done"`, que notifica al visualizador si el algoritmo terminó o no.

Bubble Sort:

Este algoritmo funciona comparando pares de elementos adyacentes e intercambiándolos si están en el orden incorrecto. Este proceso se repite en múltiples pasadas hasta que la lista está completamente ordenada, lo que hace que los elementos más grandes "burbujeen" hacia el final de la lista. Es útil para listas pequeñas y para aprender los conceptos básicos de los algoritmos de ordenamiento, pero es ineficiente para conjuntos de datos grandes.

Se modificó, del esqueleto del repositorio, la función *step*, la cual representa un solo paso del ciclo, siendo el código el siguiente:

```
def step():
    global items, n, i, j
    a=j
    b=j+1
    if items[a]>items[b]:
        aux=items[a]
        items[a]=items[b]
        items[b]=aux
        swap=True
    else:
        swap=False
```

```
    j+=1
    if j>=n-1-i:
        j=0
        i+=1
    if i==n-1:
        done=True
    else:
        done=False
    return {"a": a, "b": b, "swap": swap, "done": done}
```

A continuación, se enumerarán todos los cambios realizados:

1- Dentro de la función *step* se llama a las variables globales (*ítems, j, n, i*).

2- Se le asigna a la variable *"a"* un valor a comparar (*j*) y a *"b"* el otro valor a comparar (el siguiente de *"a"*, es decir, *j+1*).

3- Se evalúa si re realiza el *swap* o no mediante la lógica de *bubble sort* y se pasa al siguiente valor a comparar (*j+=1*).

4- Se evalúa si es el último valor a comparar de la vuelta actual.

5- Se determina el booleano de *done* evaluando si es el último valor de la última vuelta.

6- Se retorna el diccionario con los valores *a*, *b*, *swap*, *done* para que lo pueda leer el visualizador.

Dificultades:

Hubo problemas con cómo formular el condicional para que devuelva *done=True* o *done=False* según corresponda en las siguientes líneas:

```
if j>=n-1-i:
    j=0
    i+=1
if i==n-1:
    done=True
else:
    done=False
```

Selection Sort:

Este es un algoritmo de ordenamiento que funciona buscando repetidamente el elemento más pequeño en una lista y colocándolo en la posición correcta. Este proceso se repite para el resto de la lista desordenada hasta que esté completamente ordenada. El algoritmo es simple de entender y fácil de implementar, pero su eficiencia no es la mejor para conjuntos de datos grandes.
Al igual que en el algoritmo anterior, se modificó únicamente la función *step*, siendo su código el siguiente:

```
def step():
    global items, n, i, j, min_idx, fase
    swap=False
    done=False
    a = min_idx
```

```
    b = i
    if fase=="buscar":
        if j<n:
            if items[j]<items[min_idx]:
                min_idx =j
            j+=1
        if j>=n:
            fase="swap"
            a = min_idx
            b = i
    if fase=="swap":
        if i<n:
            if min_idx!=i:
                aux=items[a]
                items[a]=items[b]
                items[b]=aux
                swap=True
            i+=1
            j=i+1
            min_idx=i
        if i==n:
            done=True
        else:
            fase = "buscar"
    return {"a": a, "b": b, "swap": swap, "done": done}
```

Los cambios realizados fueron los siguientes:

1- Se declaran las variables globales dentro de la función *step* (*items, n, i, j, min_idx, fase*)

2- Se inicializan *a*, *b*, *swap* y *done* al inicio del *step* para evitar errores del visualizador, que requiere siempre esos valores en el diccionario del retorno.

3- Se divide la función *step* en las dos fases del algoritmo ("*buscar*" y "*swap*").

4- En la fase "*buscar*", el índice *j* inicializado en 0 recorre la parte desordenada de la lista buscando el índice del valor mínimo (*min_idx*).

5- Cuando *j* llega al final de la lista, se cambia a la fase "*swap*" y se declaran de nuevo las variables *a* y *b* con los nuevos valores a evaluar en fase *swap*.

6- En la fase *swap*, si en índice del mínimo elemento no es *i* (inicializado en 0 para el primer *step*), se intercambian los elementos ubicados en esos índices.

7- Se determina *done=True* cuando termina la fase *swap*, es decir, cuando *i == n*.

8- Se reinicia el valor de los índices *j* y *min_idx* para volver a declarar la fase "*buscar*".

9- Se retorna el diccionario con los valores *a*, *b*, *swap*, *done* para que lo pueda leer el visualizador.

Dificultades:

• Hubo confusiones con respecto al uso de la variable *fase*, hasta que se vio conveniente utilizarla para organizar la función *step*, alternando la fase cada vez que la fase actual terminaba (de "*buscar*" a "*swap*" o viceversa).

• Al probar el visualizador detectamos que no se realizaba la animación debido a que el código no declaraba las variables a y b, las cuales eran requeridas por el diccionario del *return* (inicialmente el *return* era *"return {"a": min_idx, "b": i, "swap": swap, "done": done}"* lo cual no podía ser leído por el visualizador).

• Inicialmente se utilizó el condicional *"if i == n – 1"* (ultima pasada) para determinar *done=True*, pero esto iba a dar error debido a que *i* se incrementaba al final del condicional anterior, asi que se solucionó cambiando al condicional *"if i == n"*.

• Hubo un error en el *return*, ya que inicialmente no estaban definidos siempre los booleanos de *swap* y *done*, asi que se solucionó definiéndolas al principio de la función *step* para evitar errores.

Insertion Sort:

Este último algoritmo divide la lista en una parte ordenada y otra desordenada, iterando a través de la parte desordenada e insertando cada elemento en la posición correcta de la parte ordenada, desplazándolo hacia la izquierda mientras sea necesario. A medida que se insertan los elementos, la parte ordenada crece de izquierda a derecha. Este algoritmo es similar al *Selection Sort*, pero se diferencia es que, el *Insertion Sort* ubica el elemento seleccionado exactamente en la posición correcta, moviéndolo dentro de la parte ordenada si es necesario.

En este caso tambien se modificó únicamente la función *step* del esqueleto del repositorio. Se muestra su código a continuación:

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

Se realizaron los siguientes cambios:

1- Se llamó a las variables globales (*items, n, i, j*) dentro de la función *step*.

2- Se declararon al principio las variables *done* y *swap* con *False* predeterminadamente (excepto que luego se modifiquen esos valores si corresponden) para evitar errores con el *return*, y por lo tanto con el visualizador.

3- Inicialmente se declara un condicional para que cuando se llegue al final del recorrido, es decir, cuando el índice del valor a acomodar sea el último (*i>=n*), se defina *done=True* y se retorne únicamente el valor de *done* para que el visualizador lo pueda leer y automáticamente terminar la animación.

4- Si todavía no se empezó a acomodar el valor seleccionado, es decir, si *j* es *None*, se crea una copia de ese índice (*j=i*) porque se va a ir disminuyendo el valor de esa copia, pero no se busca todavía que se cambie el valor del índice que estamos comparando (*i*).

5- Luego se define *a* y *b*, valores que necesita el diccionario del *return*, con los índices de los valores a comparar, es decir, el que esta seleccionado (*j*) se guarda en "*b*", y el de su izquierda (*j-1*) se guarda en "*a*".

6- Si *j >0* e *items[a]>items[b]*, se realiza el intercambio y se define *swap=True*. Si no se cumple alguna de estas dos condiciones significa que el valor ya está completamente acomodado en la parte ordenada, asi que se pasa a comparar el siguiente valor, es decir, se incrementa *i*, además de resetearse *j* con el valor *None* para volver a hacer la copia.

7- Se retorna el diccionario con los valores *a*, *b*, *swap*, *done* para que lo pueda leer el visualizador.

Dificultades:

Hubo un problema al probar el visualizador ya que no funcionaba correctamente la animación. Se llegó a la conclusión de que el problema estaba en que nunca avanzaba al siguiente valor de *i* ni reseteaba *j*, es decir, estaba mal el condicional. Inicialmente, este era el siguiente:

```
if j<=0:
    i+=1
    j=None
```

Lo incorrecto era que, esto hacía que solo avanzara cuando *j* llegaba exactamente a *0*, pero no cuando el elemento ya estaba acomodado sin llegar al inicio. Se descubrió que también era necesario avanzar cuando *items[a] <= items[b]*, porque eso indica que el valor ya está en su posición correcta dentro de la parte ordenada. Por lo tanto, se modificó el código de la siguiente manera:

```
else:
    i+=1
    j=None
```

Siendo así directamente el *else* del condicional anterior (*if j>0 and items[a]>items[b]:*)

Conclusión

A partir de la resolución de este trabajo practico, se concluye que éste resultó sencillo en cuanto a la elaboración del código. Sin embargo, tomó más tiempo comprender la funcionalidad específica del visualizador utilizado, y como esta afectó directamente a la elaboración del código. Además, se considera que, a pesar de ser algo nuevo y agobiante inicialmente, la implementación de la herramienta *GIT* resultó muy positiva e importante de adquirir como nuevo conocimiento.