TRABAJO PRÁCTICO FINAL

MATERIA: Introducción a la programación

PROFESORES:

- Sergio Santa Cruz
- Nahuel Sauma
- Gonzalo Godoy

GRUPO 5

INTEGRANTES:

-Franco Nicolas Parisi
-Nicolas Tschering

Algoritmos de ordenamiento:

Los algoritmos de ordenamiento son procedimientos que permiten organizar una lista de elementos, como números o palabras, para que queden ordenados de manera clara y útil. Entre los algoritmos más conocidos se encuentran Bubble Sort, Insertion Sort y Selection Sort. Cada uno utiliza una estrategia distinta: Bubble Sort intercambia elementos desordenados, Insertion Sort coloca cada elemento en su posición correcta dentro de un grupo ya ordenado, y Selection Sort selecciona el valor más pequeño o más grande para ubicarlo primero.

- BUBBLE SORT:

```
items = []
n = 0
i = 0 #pasada actual
j = 0 #posicion dentro de la pasada

def init(vals):
    global items, n, i, j
    items = list(vals) #guarda la lista
    n = len(items)
    i = 0 #numero de pasada
    j = 0 #posicion de comparacion

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

    if i >= n - 1:
        return{"done": True}    # Cuando no queden pasos, devolvé {"done":
True}.
    #NO SE PUEDE PONER AL FINAL ESTE CONDICIONAL, PORQUE LOS DEMAS RETURN
LO    ANULARIAN ANTES.

    # 1) Elegir índices a y b a comparar en este micro-paso (según tu
Bubble).

    a = j
    b = j + 1
```

```
    # 2) Si corresponde, hacer el intercambio real en items[a], items[b] y
marcar swap=True.

    swap = False

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

    # 3) Avanzar punteros (preparar el próximo paso).

    j = j + 1

    # Si llegamos al final de la pasada, reiniciar j y avanzar la pasada i

    if j >= n - 1 - i: #Le resta las posiciones que ya están ordenadas al
final, i se usa para ignorar los valores que quedaron al final de la lista
en la actual pasada.
        j = 0
        i = i + 1

    # 4) Devolver {"a": a, "b": b, "swap": swap, "done": False}.

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

Este algoritmo realiza comparaciones de izquierda a derecha e interpreta cuál de los dos elementos es mayor. Si el elemento de la izquierda es mayor, debe intercambiarlo por el de la derecha, lo que hace que se realice un swap. Luego, sigue comparando hasta el final de la lista, empujando el elemento de mayor valor hacia el final de la misma. Una vez terminada dicha pasada, ese último valor no será tenido en cuenta en la próxima pasada de comparaciones.

En otras palabras, Bubble Sort va ordenando la parte del final de la lista. Cada pasada “empuja” el número más grande hacia el final. Por eso, después de la primera pasada, el último ya está en su lugar;
después de la segunda, los dos últimos, y así sucesivamente.

En cada paso de comparación la variable “J”, que funciona para determinar las posiciones a comparar, aumenta su valor en 1. Luego de haber comparado todos los elementos, “j” vuelve a su valor 0 y la variable “i” aumenta en 1, ya que esta última se utiliza como contador de pasadas completas (cada vez que llegás al final y volvés al principio de la lista).

n - 1 - i Esta línea es muy importante porque indica hasta dónde comparar en esta vuelta:
n(cantidad total de elementos)
i (cantidad de pasadas)

Y el siguiente condicional indica que si “j” ya llegó al límite de comparación de elementos, termine la pasada.

if j >= n - 1 - i

Una decisión importante que tomamos desarrollando este algoritmo (y que luego decidimos mantenerla en el resto de los algoritmos) fue colocar el siguiente bloque de código al principio del step, ya que no pudimos hacer que funcione ningún algoritmo colocándolo al final.

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

Al ser el primer algoritmo en desarrollar fue bastante complicado entender cómo funcionaba por lo que nos tuvimos que apoyar mucho en videos tutoriales, páginas dedicadas a este tipo de algoritmos e inteligencia artificial.

- INSERTION SORT

```
items = []
n = 0
i = 0      # elemento que queremos insertar
j = None   # cursor de desplazamiento hacia la izquierda (None = empezar)
# arranca en NONE Porque al principio del paso, todavía NO estamos
desplazando nada.

def init(vals):
    global items, n, i, j
    items = list(vals)
    n = len(items)
    i = 1      # común: arrancar en el segundo elemento (la primer
posicion ya arranca ordenada). I es el elemento que quiero ordenar
```

```
    j = None

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

    # - Si i >= n: devolver {"done": True}.
    if i >= n:
        return{'done': True} #NO SE PUEDE PONER AL FINAL ESTE CONDICIONAL, PORQUE LOS DEMAS RETURN LO ANULARIAN ANTES.


    # - Si j es None: empezar desplazamiento para el items[i] (p.ej., j = i) y devolver un highlight sin swap.
    if j == None:
        j = i #(porque si j fuera 0, no hay nada a la izquierda para comparar)
        return{"a": j-1, "b": j, "swap": False, "done": False} #estoy viendo estos valores, pero que aun no hice ningun intercambio”

    # - 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]: #Si j es mayor a 0, es porque hay un elemento a la izquierda para poder comparar.
        aux = items[ j - 1]
        items[ j - 1] = items[j]
        items[j] = aux
        j = j - 1 #Movemos j a una posicion mas a la izquierda
        return{'a': j, 'b': j + 1, 'swap': True, 'done': False} #a y b contiene el valor de las posiciones en ese momento. Le aviso al visualiador que cambiaron.
        # COMO A J SE LE RESTA UNO ANTES DEL RETURN, ES NECESARIO SUMARLE +1 CUANDO SE LE DICE AL VISUALIZADOR QUE VALORES ACABAMOS DE COMPARAR PARA QUE VEA AL CORRECTO.

    # - Si ya no hay que desplazar: avanzar i y setear j=None.

    i = i + 1
    j = None
```

```
    return {"a": 0, "b": 0, "swap": False, "done": False} #NO se compara
nada, por eso las variables estan con valor 0. Y aun debe seguir el
algoritmo, por ello dice done False.
    #Es necesario retornar algo porque mas que este vacio, sino el
visualizador no sirve.
```

Este algoritmo también empieza las comparaciones desde el principio de la lista. Arranca comparando de derecha a izquierda, en el bloque ya ordenado de la lista, y se va a desplazando hacia la izquierda hasta que el valor a comparar encuentre su lugar correspondiente en la lista en esa pasada. En el código su forma de escribirla es bastante similar, pero la variable “J” arranca con el mismo valor de la variable “I” en cada pasada y esa variable “I” es la que determina el índice que se va a arrancar comparando.

Algo a tener en cuenta es que la variable “J” arranca con valor none para luego tomar el valor de “I” una vez que comience a realizar la pasada de comparaciones. En este caso, “J” funciona como cursor de desplazamiento.

Este algoritmo fue un poco más sencillo de desarrollar, porque es bastante similar a Bubble Sort, pero tiene algunos detalles que lo distinguen: desplazamiento, utilización de variables, etc.

La dificultad que tuvimos al desarrollarlo fue en esta línea:

.

```
 j = j - 1 #Movemos j a una posicion mas a la izquierda
        return{'a': j, 'b': j + 1, 'swap': True, 'done': False}
```

Ya que no podíamos entender cómo realizar el return correctamente del término “b” porque en la línea de arriba habíamos editado el valor de “J”. Luego entendimos que si le sumamos uno, comenzábamos dicha edición previa y el algoritmo funcionaba bien.

- SELECTION SORT

```
# Contrato: init(vals), step() -> {"a": int, "b": int, "swap": bool,
"done": bool}

items = []
n = 0
i = 0          # donde inicia la parte desordenada de la lista
j = 0          # cursor que recorre y busca el mínimo
min_idx = 0    # índice del mínimo de la pasada actual
fase = "buscar"  # "buscar" | "swap"
```

```
def init(vals):
    global items, n, i, j, min_idx, fase
    items = list(vals)
    n = len(items)
    i = 0 #se inicia al principio # es la posición donde quiero colocar el número más chico de esta pasada.
    j = i + 1 # arranca desde el siguiente
    min_idx = i #se supone que el primer valor es el mas chico
    fase = "buscar" #para arrancar buscando el minimo y la cambiamos a swap cuando sea necesaro intercambiar

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

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

    # - Fase "buscar":
    if fase == 'buscar':
        if j < n:
            if items[j] < items[min_idx]: #comparar j con min_idx,
                min_idx = j #actualizar min_idx,
            j = j + 1 #avanzar j.

    #   Devolver {"a": min_idx, "b": j_actual, "swap": False, "done": False}.
            return{"a": min_idx, "b": j-1, "swap": False, "done": False}

        else:
            fase = 'swap'
            return {"a": i, "b": min_idx, "swap": False, "done": False}

    #   Al terminar el barrido, pasar a fase "swap".

    if fase == 'swap':  # - Fase "swap": si min_idx != i, hacer ese único swap y devolverlo.
        a = i
        b = min_idx # Tuve que agregar estas variables porque no se me ocurrio como retornar los valores necesario de i e min_idx en el return porque antes los modificaba en el if y luego fuera de el.
```

```
        swap = False
        if min_idx != i:
            aux = items[i]
            items[i] = items[min_idx]
            items[min_idx] = aux
            swap = True

        #   Luego avanzar i, reiniciar j=i+1 y min_idx=i, volver a
"buscar".

        i = i + 1 #sumamos para seguir con la siguiente pasada
        j = i + 1
        min_idx = i
        fase = 'buscar'

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

Selection Sort busca siempre el menor elemento y, una vez que lo localiza, lo ordena al principio de la lista. Funciona buscando, en cada pasada, el elemento más pequeño de la lista y colocándolo en su posición correcta. Primero encuentra el menor de todos y lo pone primero; luego busca el segundo menor y lo pone en el segundo lugar, y así sucesivamente hasta que toda la lista queda ordenada.

En otras palabras. Divide la lista en dos, una parte ordenada (que evoluciona mediante ocurren las iteraciones) y otra desordenada. El algoritmo selecciona el elemento más chico de la parte desordenada de la lista y la coloca al final de la parte ordenada de la misma.

Funciona a través de dos fases: la primera es “buscar”

Este algoritmo fue el más complicado. Tuvimos que apoyarnos bastante en la IA y fue necesario entender línea por línea que hacía el programa y entender el porqué de cada bloque.