Informe

COMISIÓN 4

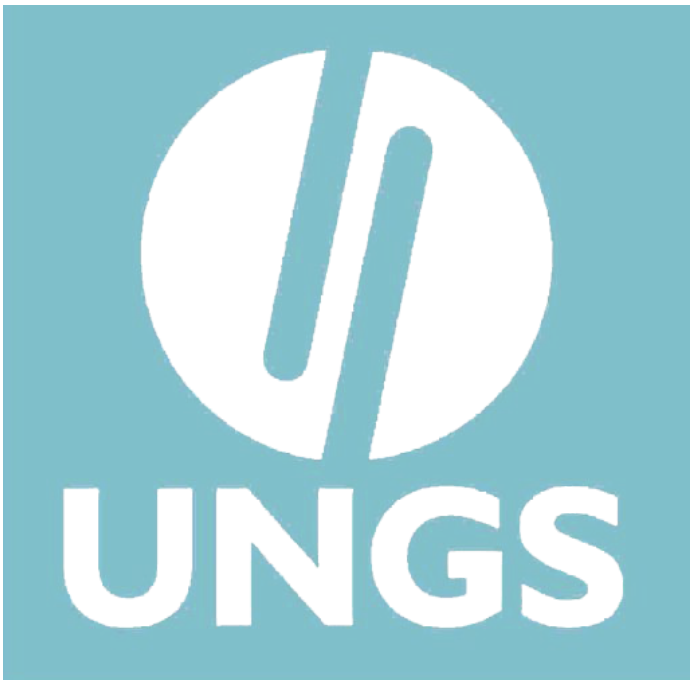
Profesores:

Omar Argañaraz

Nancy Nores

Integrantes:

Silva Fabricio



Introducción:

El trabajo Práctico consiste en programar distintos sistemas de ordenamiento e implementarlos a la aplicación, los sistemas integrados son *Bubble*,*Insertion*,*Selection*,*Shell*:

Sort_Bubble:

```
def init(vals):
    global items, n, i, j
    items = list(vals)
    n = len(items)
    i = n-1
    j = 0

def step():
    global items, n, i, j
    # Condición para que el algoritmo devuelva "done": True. (termine),si i==0
termine de ordenar
    if i<1:
        return {"done":True}
    a=j
    b=j+1
    swap=False
    #comparo desde items[0] con el siguiente en la lista y hago swap si
corresponde
    if items[a]>items[b]:
        items[a],items[b]=items[b],items[a]
        swap=True
    #avanzo j
    j+=1
    #si j llega al final de la lista, vuelvo a empezar desde 0 y disminuyo i
    if j>=i:
        i-=1
        j=0
    return {"a": a, "b": b, "swap": swap, "done": False}
```

El primer sistema en ser implementado fue el Bubble,primero se setea la cantidad de veces que se recorre la lista con el valor i=len(items)-1,después de cada pasada se resta 1 y con un if se pone de condición que cuando i<1,termine el ordenamiento (“done”:True).para empezar el ordenamiento,tomo un valor j dentro de la lista Items.J es comparado con J+1 en cada pasada y, si corresponde son intercambiados.Cuando J llegue al final de la lista(i), resto i-1 y reseteo J.

la mayor dificultad que tuve para este fue entender el return del diccionario, necesario para que el sistema entienda que variables está utilizando para comparar y resaltar.

Sort_Selection:

```
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"

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

    # Condición para que el algoritmo devuelva "done": True. (termine)
    if n <= 1 or i >= n - 1:
        return {"done": True}

    # Fase de búsqueda: comparar j con min_idx
    if fase == "buscar":
        if j < n:
            a = min_idx
            b = j
            # actualizo min_idx cuando encuentro un valor menor
            if items[j] < items[min_idx]:
                min_idx = j
            j += 1
            return {"a": a, "b": b, "swap": False, "done": False}
        else:
            # en else, j>n, osea recorrí la lista entera, paso a swap
            fase = "swap"

    # Fase de swap: realizar (si corresponde) el intercambio entre i y min_idx
    if fase == "swap":
        a = i
        b = min_idx
        swap = False
        # condición que min_idx sea diferente de i para hacer el swap,sino no
hace nada.
        if min_idx != i:
            items[i], items[min_idx] = items[min_idx], items[i]
            swap = True
        # avanzo i para la siguiente iteración y reinicio j y min_idx
        i += 1
        min_idx = i
        j = i + 1
        fase = "buscar"
        # si ya estamos en el ultimo i, marcar "done" True.
        if i >= n - 1:
            return {"a": a, "b": b, "swap": swap, "done": True}
        return {"a": a, "b": b, "swap": swap, "done": False}
```

El segundo sistema implementado fue el Sort_selection.Primero se divide el código en dos fases,"buscar" y "swap".En la fase "buscar" recorro la lista buscando el valor más chico y lo guardo en min_idx,este valor es comparado con toda la lista y cuando encuentre uno menor se actualiza, una vez terminada se pasa a la fase "swap" utilizando el valor i como índice de ítems hago el swap entre i y min_idx siempre que estos no sean el mismo elemento. Finalmente sumo i+1 para ir al siguiente elemento de la lista y reinicio j y min_idx.Cuando i>=n,termino el ordenamiento.

No tuve mayores dificultades para este sistema,logré terminarlo a base de prueba y error.

Sort_Insertion:

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

```
def step():
    global items, n, i, j
    #condición para retornar "done":True, cuando haya recorrido la lista entera.
    if i >= n:
        return {"a": 0, "b": 0, "swap": False, "done": True}

    # comenzar la inserción para items[i]
    if j is None:
        j = i
        return {"a": j-1, "b": j, "swap": False, "done": False}

    #pongo las condiciones para el swap y voy disminuyendo j hasta llegar al
inicio de la lista, swapeando cuando corresponda.
    a = j-1
    b = j
    if j > 0 and items[a] > items[b]:
        items[a], items[b] = items[b], items[a]
        swap=True
        j -= 1
        return {"a": a, "b": b, "swap": True, "done": False}

    # ya no hay más desplazamientos para este i
    i += 1
    j = None
    return {"a": 0, "b": 0, "swap": False, "done": False}
```

Sort_insertion fue el tercer sistema implementado.Este sistema va avanzando en la lista con el elemento i,y compara todos los elementos a la izquierda tomando j y j-1,swapeando cada vez que j-1 sea mayor a j siendo que en cada pasada j disminuye en 1 hasta así llegar al inicio de la lista.Una vez todos los ítems estén en su posición correcta, se avanza i+1 y se

resetea j,una vez se alcanza i>=n, todos los elementos están en el orden correcto,y se devuelve “done”:True.
Este sistema fue con diferencia el que más me costó de los primeros tres porque no lograba visualizar cómo hacer las comparaciones.Recurrí a ver videos en youtube y consultar con compañeros para hacerme la idea y finalmente después de mucho prueba y error terminar con este código.
Sort_Shell:

```
def init(vals):
    global elementos, n, brechas, brecha, i, j, fase
    elementos = list(vals)
    n = len(elementos)
    # Secuencia simple de gaps: n//2, //2, ...
    brechas = []
    g = n // 2
    while g > 0:
        brechas.append(g)
        g //= 2
    brecha = 0
    i = 0
    j = None
    fase = "cambio"

def step():
    global elementos, n, brechas, brecha, i, j, fase

    #para listas ya ordenadas o de tamaño 0 o 1
    if n <= 1:
```

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

    # Si no hay brechas pendientes, terminamos
    if fase == "cambio":
        if not brechas:
            return {"a": 0, "b": 0, "swap": False, "done": True}
        # tomar siguiente gap, con brechas.pop(0),tomo el valor y lo quito de la
lista en un paso
        brecha = brechas.pop(0)
        i = brecha
        j = None
        fase = "procesando"
        # muestro la primera comparación para la nueva brecha
        return {"a": i - brecha, "b": i, "swap": False, "done": False}

    # Fase de procesado para la brecha actual
    if fase == "procesando":
        # Si ya recorrí todos los i para esta brecha, paso a cambio para tomar
la siguiente
        if i >= n:
            fase = "cambio"
            return {"a": 0, "b": 0, "swap": False, "done": False}

        # Si todavía no recorrí con el i actual, inicializo j
        if j is None:
            j = i
            # mostrar comparación inicial entre j-brecha y j
            return {"a": j - brecha, "b": j, "swap": False, "done": False}

        # Comparar y, si corresponde, hacer swap adyacente a distancia 'brecha'
        a = j - brecha
        b = j
        if a >= 0 and elementos[a] > elementos[b]:
            elementos[a], elementos[b] = elementos[b], elementos[a]
            # mover cursor hacia la izquierda para seguir insertando este
elemento
            j -= brecha
            # si j queda menor que brecha, esto indica que la inserción terminó
en el borde;
            # la próxima llamada detectará j<brecha y avanzará i.
            return {"a": a, "b": b, "swap": True, "done": False}
        else:
```

```
            # no se requiere más desplazamiento para este i; pasar al siguiente
i
            i += 1
            j = None
            return {"a": 0, "b": 0, "swap": False, "done": False}
```

El cuarto sistema implementado es el Sort_Shell.Este sistema funciona parecido al Insertion, solo que compara un elemento con otro a una distancia que está definida en la función Init(vals). Estas distancias que se utilizan son agregadas a una lista brechas,los valores posibles son definidos por g=len(items)//2 dentro de un loop while con la condición de que g>0.El código está dividido en dos fases,”procesando” y “cambio”,se empieza en la fase cambio para elegir la primer brecha que se utilizará para comparar elementos j y j-brecha, siendo brecha=brechas[0], si j es menor a j-brecha se swapea, y j cambia a j-brecha para comparar con todos los elementos a la izquierda hasta que j>brecha, cuando esto pase se cambia a fase “cambio” y se

utiliza el siguiente valor de la lista brechas y se repite el ciclo.Para tomar el valor de brechas, se utiliza brecha=brechas.pop(0), lo cual setea en brecha el primer valor y al mismo tiempo lo saca de la lista.Esto eventualmente hace que la lista quede vacía y dentro de la fase “cambio” se pone la condición para terminar el ordenamiento, si la lista está vacía, se devuelve “done”:True.
Este sistema me fue imposible de programar por lo que recurrí a la IA Copilot para que me generara un código y me lo explicara hasta entenderlo. Finalmente terminé escribiendo el código utilizando mucho de lo que la IA me proporcionó.

Implementación de página inicial:
con ayuda de Copilot implemente una pagina principal anterior al programa de algoritmos, donde se muestra el objetivo del trabajo y más datos, incluyendo un

botón que redirecciona al visualizador y otro al informe del trabajo.

Conclusión:
El trabajo fue mucho más difícil que cualquier otra cosa que hayamos hecho durante la cursada.Tuve problemas para entender todo lo relacionado a las funciones globales y como implementarlas en cada algoritmo,nunca pude hacer “levantar” la app del visualizador para utilizarlo con las instrucciones dadas en el trabajo con el comando ***python -m http.server*** y con la implementación de la página principal, pues durante la cursada no se vio nada relacionado a HTML.
Para solucionar algunos de estos problemas fue cuestión de practicar codificando, pero para otros recurrí a la ayuda de compañeros e IA de manera orientativa.
No logre implementar los algoritmos Quick ni Merge, pues resultaron muy complicados para mi y lamentablemente mi compañero de

grupo desaprobó ambos parcial y recuperatorio, por lo que tuve que hacer el trabajo solo.
Más allá de todo esto logré implementar 4/6 algoritmos y una página funcional, el trabajo me pareció muy difícil es una primer instancia pero con el tiempo se logró avanzar.