



**Universidad
Nacional de
General
Sarmiento**

TRABAJO PRÁCTICO

“Algoritmos de ordenamiento”

COMISIÓN 2 - 2do semestre 2025

DOCENTES : Flavia Bottino y Lucas Bidart

MATERIA : Introducción a la Programación

GRUPO 10:

Erica Tinuco

Lourdes Flores

Juliana Muñoz

Debora Narvaez

Introducción:

El trabajo práctico consiste en la implementación de “**algoritmos de ordenamiento**” y la visualización de los mismos desde el navegador. Los **algoritmos de ordenamiento** son un conjunto de instrucciones que recibe como entrada una lista para organizar sus elementos en una secuencia específica, como orden ascendente o descendente, numérico o alfabético.

En este trabajo se completó el código del visualizador con **7** algoritmos: **Bubble sort**, **Selection sort**, **Insertion sort**, **Quick sort**, **Merge sort**, **Shell sort** y **Cocktail sort**, cumpliendo con el contrato que utiliza el visualizador. Bajo un modelo de ejecución paso a paso. Cada llamada a la función `step()` debe realizar la unidad mínima del trabajo (una comparación o intercambio). Esto requiere utilizar variables globales para continuar el estado del algoritmo entre llamados, simulando los bucles `for` tradicionales.

El contrato de la función `step()`:

La función `step()` debe retornar un diccionario con la siguiente información:

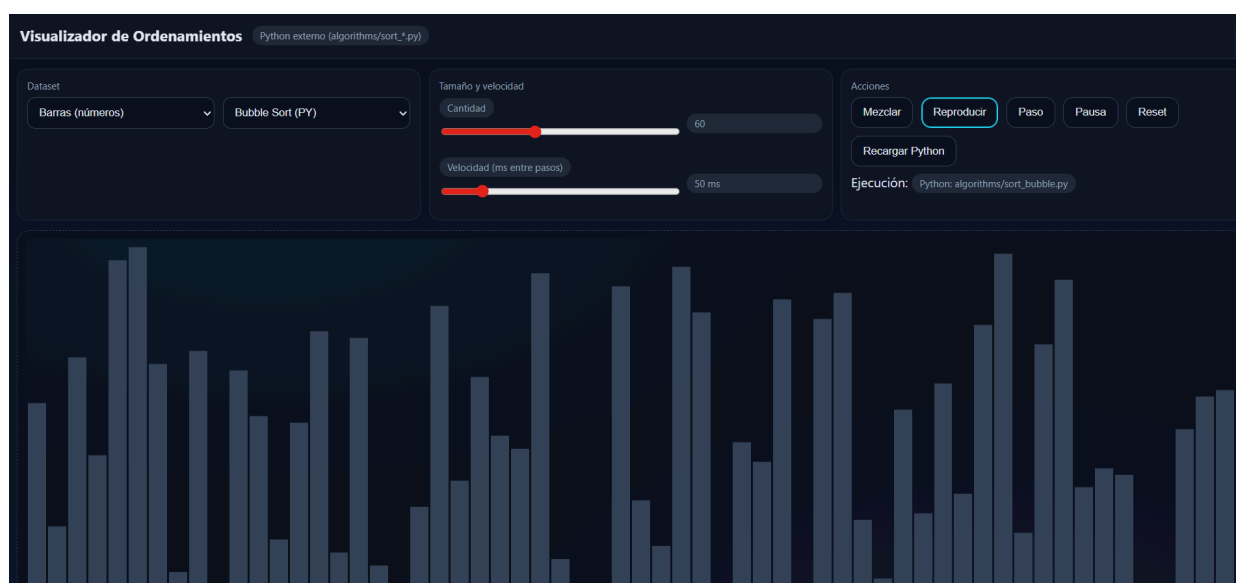
a (int): *Índice del primer elemento involucrado* (comparado o intercambiado).

b (int): *Índice del segundo elemento involucrado* (comparado o intercambiado).

swap (bool): *Es **True** si se realizó un intercambio de datos. Es **False** si solo fue una comparación.*

done (bool): *Es **True** si el algoritmo ha finalizado el ordenamiento.*

El visualizador:



¿Qué función realiza cada “botón”?

El visualizador contiene, un panel de controles que generalmente tiene diferentes acciones:

Mezclar: Desordena la lista para volver a ordenarla utilizando “Reproducir”, o “Paso”.

Reproducir: Llama a `step()` en bucles con pausa, permitiendo así que las barras del visualizador se intercambian automáticamente.

Paso: Llama a `step()` una sola vez, permitiendo observar cómo se realiza el intercambio de menor a mayor paso a paso.

Pausa: Lo detiene.

Reset: Con *Reset* se vuelve a su estado original, una vez que “Reproducir” o “Paso”, haya sido utilizado.

La consigna:

Este trabajo práctico consiste en utilizar algoritmos de ordenamiento donde se debe completar los códigos cumpliendo con el contrato `init(vals) + step()`, los mismo deben estar en el repositorio interno del grupo(`fork`).

Bubble Sort (Ordenamiento de Burbujas):

```
def step():
    # TODO:
    global items, n, i, j
    if i >= n - 1:
        return {"done": True}
    # Elegir indices a y b a comparar en este micro-paso (segun tu Bubble).
    a = j
    b = j + 1 #Adyacente del indice, elemento de al lado.
    # Si corresponde, hacer el intercambio real en items[a], items[b] y marcar swap=True.
    if i < n-1:
        if items[a] > items[b]: #Sin ciclos for porque funciona "paso a paso"
            items[a], items[b] = items[b], items[a] #Intercambio de menor a mayor.
            swap = True
        else:
            swap = False
    j += 1 #Aumento el indice

    if j >= n - i - 1: #Si hay que reiniciar el indice
        j = 0
        i += 1
    # Devolver {"a": a, "b": b, "swap": swap, "done": False}.
    return {"a": a, "b": b, "swap": swap, "done": False}
    # Cuando no queden pasos, devolver {"done": True}.
    if i >= n-1:
        return {"done": True}
```

El **Bubble Sort** es un algoritmo de ordenamiento simple que sirve para ordenar listas(por ejemplo, números) de menor a mayor o al revés.

Variables de Estado Globales:

i: Contador del bucle externo. Rastrea cuántos elementos ya están ordenados al final de la lista.

j: Cursor del bucle interno. Recorre la porción no ordenada comparando adyacentes ($a=j$ y $b=j+1$).

swap: Bandera que indica si hubo algún intercambio en la pasada actual. Se usa para la optimización, si no hay swap, el algoritmo termina.

Insertion Sort (Ordenamiento por Inserción)

```
15 def step():
16     # TODO:
17     global items, n, i, j
18     if i >= n:
19         return {"done": True}
20     if j is None:
21         j = i
22         return {"a": j-1 if j > 0 else 0, "b": j, "swap": False, "done": False}
23     if j > 0 and items[j-1] > items[j]:
24         a = j - 1
25         b = j
26         items[a], items[b] = items[b], items[a]
27         j -= 1
28         return {"a": a, "b": b, "swap": True, "done": False}
29     i += 1
30     j = None
31     return {"a": 0, "b": 0, "swap": False, "done": False}
```

Algoritmo de ordenamiento que organiza una lista o arreglo insertando cada elemento en su posición correcta dentro de una parte ya ordenada de la lista.

Variables de Estado Globales:

items : es la lista que vamos a ordenar.

n: el tamaño de la lista.

i: marca la posición del elemento que estamos “insertando” dentro de la parte ya ordenada. Cada vez que finaliza un ciclo de inserción, **i** avanza.

j: compara hacia la izquierda y se va reduciendo mientras el elemento de la izquierda sea mayor.

is none: indica si todavía no se indicó el valor para comparar.

True → no estamos moviendo elementos (inicio del paso).

False → ya estamos moviendo elementos con **j**.

(esto sirve para controlar si debemos inicializar **j** al inicio de cada iteración de **i**).

swap: indica si el algoritmo debe: mover el elemento en **j** hacia la derecha, o pasar a la siguiente comparación.

Selection Sort (Ordenamiento por Selección):

```
19 def step():
20     global items, n, i, j, min_idx, fase
21     if i >= n:
22         return {"done": True}
23     if fase == "buscar":
24         if j < n:
25             # Todavía buscando el mínimo en la parte no ordenada
26             curr_j = j # Guardamos el valor actual antes de incrementar j
27             if items[j] < items[min_idx]:
28                 min_idx = j
29             j += 1
30             # Devolvemos el estado actual para visualización (comparación de j y min_idx)
31             return {"a": min_idx, "b": curr_j, "swap": False, "done": False}
32         else:
33             # Se completó el barrido de búsqueda, pasar a fase "swap"
34             fase = "swap"
35             # La próxima llamada a step ejecutará la fase "swap"
36             return step() # Llamada recursiva para ejecutar el swap inmediatamente
37     elif fase == "swap":
38         if min_idx != i:
39             # Realizar el único swap
40             items[i], items[min_idx] = items[min_idx], items[i]
41             # Devolver la acción de swap para visualización
42             result = {"a": i, "b": min_idx, "swap": True, "done": False}
43         else:
44             # No se necesita swap (el elemento ya está en su lugar)
45             result = {"a": i, "b": i, "swap": False, "done": False}
46             # Prepararse para la siguiente iteración
47             i += 1
48             j = i + 1
49             min_idx = i
50             fase = "buscar"
51     return result
52 # Caso por defecto, aunque no debería ser alcanzado
53 return {"done": True}
```

Es un algoritmo de ordenamiento que funciona seleccionando el elemento más pequeño de la parte desordenada de la lista colocándolo en su posición correcta.

Variables de Estado Globales:

items lista que vamos a ordenar

i: posición donde vamos a colocar el siguiente mínimo.

j: recorre la parte no ordenada para buscar el mínimo .

min_idx: guarda el índice del mínimo encontrado.

fase: dice que en fase del algoritmo ejecuta "busca" (halla el mínimo) o

swap: intercambia y se reinicia con la posición mínimo en i

Quick Sort (Ordenación Rápida):

```
48 def step() -> dict:
49     global items, n, stack, low, high, pivot_index, i, j, partición_terminada
50     if not stack and partición_terminada:
51         return {"done": True}
52     if partición_terminada:
53         if not stack:
54             return {"done": True} # Doble chequeo, por si acaso
55         low, high = stack.pop()
56         if low >= high:
57             return step() # Llamada recursiva para procesar el siguiente subarreglo
58         pivot_index = high
59         i = low - 1 # 'i' comienza un paso antes del subarreglo
60         j = low     # 'j' comienza en el primer elemento del subarreglo
61         partición_terminada = False # Estamos dentro de un proceso de partición
62         return {"a": low, "b": pivot_index, "swap": False, "done": False}
63     if j < pivot_index: # Mientras 'j' no haya alcanzado el pivote (high)
64         a = j
65         b = pivot_index # El pivote siempre es high
66         if items[j] <= items[pivot_index]:
67             i += 1
68             _swap(i, j)
69             result = {"a": i, "b": j, "swap": True, "done": False}
70         else:
71             result = {"a": i if i >= low else low, "b": j, "swap": False, "done": False}
72         j += 1 # Mover 'j' al siguiente elemento
73     return result
```

```
74     if j == pivot_index:
75         pivot_pos = i + 1
76         a = pivot_pos
77         b = pivot_index
78         _swap(pivot_pos, pivot_index)
79         partición_terminada = True
80         if low < pivot_pos - 1:
81             stack.append((low, pivot_pos - 1))
82         if pivot_pos + 1 < high:
83             stack.append((pivot_pos + 1, high))
84     return {"a": a, "b": b, "swap": True, "done": False}
85     return {"done": True}
```

Es un algoritmo de ordenamiento rápido y muy usado. Funciona dividiendo la lista en partes más pequeñas usando un “pivote”.

Variables de Estado Globales:

ítems : El arreglo que se está ordenando.

n : La longitud total del arreglo.

stack : Pila que almacena las tuplas (low, high) de los subarreglos pendientes por ordenar (la parte recursiva de quick sort).

low, high : Los índices de inicio y fin del subarreglo actual que se está particionando.

pivot_index : El índice del elemento pivote.

pivot_pos : Determinar la posición final y correcta del elemento pivote (originalmente en high) después de que la partición de Lomuto ha concluido.

i : Índice que rastrea el límite entre los elementos menores o iguales al pivote y los elementos mayores (comienza en low -1).

j : Índice que se mide a través de los elementos del subarreglo comparando cada elemento con el pivote.

particion_terminada : Un indicador booleano que señala si la partición actual ha finalizado y se debe empezar la siguiente (o terminar el algoritmo).

Pivote: Es un valor que se usa para dividir la lista en dos partes.

Merge Sort (Ordenamiento por combinación):

```
1 def merge(left_list, right_list):
2     sorted_list = []
3     left_list_index = right_list_index = 0
4     left_list_length, right_list_length = len(left_list), len(right_list)
5
6     for _ in range(left_list_length + right_list_length):
7         if left_list_index < left_list_length and right_list_index < right_list_length:
8             if left_list[left_list_index] <= right_list[right_list_index]:
9                 sorted_list.append(left_list[left_list_index])
10                left_list_index += 1
11            else:
12                sorted_list.append(right_list[right_list_index])
13                right_list_index += 1
14        elif left_list_index == left_list_length:
15            sorted_list.append(right_list[right_list_index])
16            right_list_index += 1
17        elif right_list_index == right_list_length:
18            sorted_list.append(left_list[left_list_index])
19            left_list_index += 1
20    return sorted_list
```

```
22 def mergeSort(nums):
23     if len(nums):
24         return nums
25     mid = len(nums) // 2
26     left_list = mergeSort(nums[:mid])
27     right_list = mergeSort(nums[mid:])
28     return merge(left_list, right_list)
29
30 #Comprobamos el funcionamiento
31 listaNumerosAleatorios = [5, 2, 1, 8 , 4]
32 print("Lista sin ordenar: " + str(listaNumerosAleatorios))
33 listaNumerosAleatorios = mergeSort(listaNumerosAleatorios)
34 print("Lista ordenada; " + str(listaNumerosAleatorios))
```

Es un algoritmo de ordenamiento que funciona dividiendo la lista en partes pequeñas y luego uniendo las (merge) ya ordenadas. Es el método más elegante y eficiente.

Variables de Estado Globales:

right_list : mitad derecha de la lista

left_list : mitad izquierda de la lista

mid : punto medio donde se divide la lista

sorted_list.append : agrega el elemento menor a la lista final

sorted_list : lista resultante ya ordenada

left_list_index : puntero que recorre la lista izquierda

right_list_index : puntero que recorre la lista derecha

Cocktail Sort (Clasificación de cócteles)

```
23 def step():
24     global items, n, start, end, i, direction, active
25     if not active or n <= 1:
26         active = False
27         return {"a": 0, "b": 0, "swap": False, "done": True}
28     if direction == "forward":
29         if i < end:
30             a = i
31             b = i + 1
32             if items[a] > items[b]:
33                 items[a], items[b] = items[b], items[a]
34                 out = {"a": a, "b": b, "swap": True, "done": False}
35             else:
36                 out = {"a": a, "b": b, "swap": False, "done": False}
37             i += 1
38             return out
39         end -= 1
40         direction = "backward"
41         i = end # empezamos a comparar hacia atrás
42         return {"a": i, "b": i - 1 if i > start else start, "swap": False, "done": False}

43     else: # direction == "backward"
44         if i > start:
45             a = i - 1
46             b = i
47             if items[a] > items[b]:
48                 items[a], items[b] = items[b], items[a]
49                 out = {"a": a, "b": b, "swap": True, "done": False}
50             else:
51                 out = {"a": a, "b": b, "swap": False, "done": False}
52             i -= 1
53             return out
54         # Fin del barrido backward
55         start += 1
56         if start >= end:
57             active = False
58             return {"a": 0, "b": 0, "swap": False, "done": True}
59         direction = "forward"
60         i = start
61         return {"a": i, "b": i + 1 if i < end else end, "swap": False, "done": False}
```

Es una versión mejorada de Bubl  Sort, la diferencia principal es que ordena en las dos direcciones: primero avanza hacia la derecha y luego vuelve hacia la izquierda, como un c ctel que se sacude.

Variables de Estado Globales:

Items: Lista que será ordenada.

n: Tamaño de la lista.

start(comenzar) : Límite inferior del barrido. Aumenta cuando termina una pasada hacia atrás.

end(fin) : Límite superior del barrido. Disminuye cuando termina una pasada hacia adelante.

i: Índice que se mueve paso a paso comparando elementos.

Direction: Controla el sentido del barrido.

Active: Controla si la clasificación sigue llamando a step().

Shell Sort (Ordenamiento de conchas):

```
15 def step() -> dict:
16     global items, n, gap, i, j
17     if gap == 0:
18         return{"done": True}
19     if i < n:
20         if j >= gap:
21             a = j
22             b = j - gap
23             if items[a] < items[b]:
24                 items[a], items[b] = items[b], items[a]
25                 j -= gap
26                 return{"a": a, "b": b, "swap": True, "done": False}
27             else:
28                 j = -1
29                 return{"a": a, "b": b, "swap": False, "done": False}
30         i += 1
31         j = i
32         return{"a": 0, "b": 0, "swap": False, "done": False}
33     gap //= 2
34     if gap > 0:
35         i = gap
36         j = i
37         return{"a": 0, "b": 0, "swap": False}
38     return{"done": True}
```

Es una versión mejorada del Insertion Sort. Ordena más rápido porque primero compara elementos que están lejos entre sí, y después, cuando todo ya está más , “aproximadamente ordenado”, usa un insertion Sort normal para acomodar los detalles.

Variables globales:

Gap: Significa salto o distancia

Swap: Significa intercambiar dos valores entre sí.

Conclusión:

En este trabajo práctico conocimos nuevos algoritmos ,que al principio no sabíamos cómo utilizarlos pero luego de investigarlos pudimos volcar nuestro conocimiento y logramos llegar a ver las imágenes.

Tuvimos inconvenientes con el algoritmo de merge, que no hemos logrado llegar a su funcionamiento.

Nos interesó mucho que diferentes algoritmos puedan resolver los mismos problemas pero con formas y tiempos distintos.