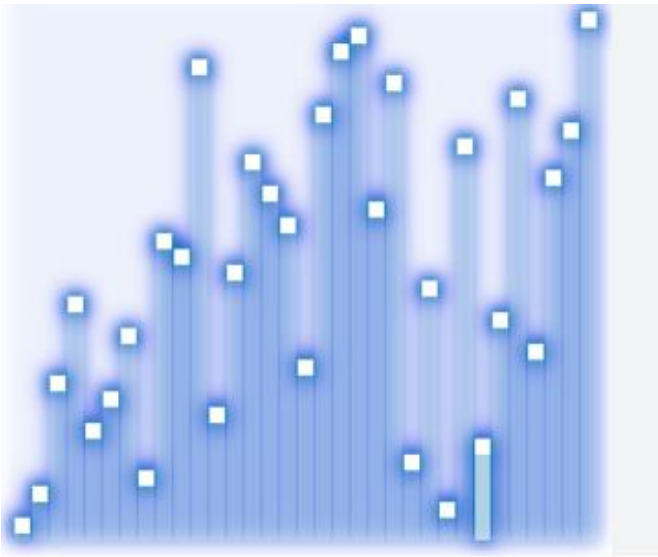
Universidad Nacional de General Sarmiento

Introducción a la Programación

“Informe de Implementación de Algoritmos de Ordenamiento”



Profesores:

- Sergio Santa Cruz
- Miguel Rodriguez

Alumnos:

- Federico Pagano
- Santiago Julio
- Ana Rebeco

Fecha de entrega: 26 de noviembre de 2025

Introducción

El ordenamiento es una actividad presente en la vida cotidiana: organizar permite administrar mejor el tiempo, reducir la complejidad de las tareas y acceder más fácilmente a la información necesaria. En informática, esta idea se vuelve esencial. A medida que la cantidad de datos crece de forma exponencial, se requieren estrategias que permitan organizarlos de manera eficiente para poder procesarlos, buscarlos y visualizarlos sin dificultad.

Cuando realizamos una búsqueda en internet, podría parecer que encontrar información entre millones de resultados debería ser extremadamente complejo. Sin embargo, gracias a los algoritmos de ordenamiento y a las estructuras de datos utilizadas por los sistemas, esto ocurre de manera eficiente. Incluso en la comunicación digital, los paquetes de información deben llegar ordenados para reconstruirse correctamente, lo que evidencia la importancia del ordenamiento en la transmisión de datos.

A lo largo de la historia de la computación surgieron numerosos algoritmos con distintos propósitos. Entre ellos, los algoritmos de ordenamiento ocupan un lugar central. Permiten organizar datos según un criterio determinado (por ejemplo, de menor a mayor, de mayor a menor o alfabéticamente), optimizando búsquedas y mejorando tiempos de respuesta.

En síntesis, el ordenamiento cumple un rol fundamental en la informática moderna: permite que la información sea accesible, utilizable y eficiente, y constituye la base de muchos otros algoritmos actuales.

Universidad Nacional de General Sarmiento

Objetivo del Trabajo

El propósito de este trabajo es presentar, analizar e implementar distintos algoritmos de ordenamiento de datos, específicamente:

- Bubble Sort
- Selection Sort
- Insertion Sort

Para cada algoritmo se incluye:

- Una descripción conceptual.
- El código implementado (con los contratos requeridos).
- El funcionamiento paso a paso mediante la función `step()`.
- Las decisiones tomadas durante la implementación.
- Las dificultades encontradas.
- Una breve reflexión final.

Metodología

En este TP, todos los algoritmos se implementaron utilizando dos funciones:

- `init(vals)`: inicializa las variables del algoritmo, se ejecuta una sola vez al comenzar o al mezclar la lista.
- `step()`: ejecuta un único micro-paso del algoritmo (comparación, avance o intercambio).

✓ No se utilizaron bucles `for` ni `while`.

✓ Cada llamada a `step()` avanza exactamente un paso del ordenamiento.

Este enfoque permite observar cada comparación e intercambio, lo que es ideal para el visualizador.

Universidad Nacional de General Sarmiento

1. Bubble Sort

Descripción conceptual

Bubble Sort compara elementos adyacentes y los intercambia si están en el orden incorrecto. Después de cada pasada completa, el elemento más grande "sube" a su posición final, como una burbuja.

Variables utilizadas

- `i`: número de la pasada.
- `j`: posición actual dentro de la pasada.
- `items`: lista interna que se va ordenando.

Decisiones y dificultades

- La mayor dificultad fue controlar manualmente los índices sin usar bucles.
- Hubo que separar claramente las condiciones: fin de pasada, comparación y swap.
- `step()` está diseñado para hacer una sola comparación por llamada.

Código implementado

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

items = []
n = 0
i = 0
j = 0

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

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

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

if j >= n - i - 1:
    j = 0
    i += 1
    if i >= n - 1:
        return {"done": True}

a = j
b = j + 1
swap = False

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

j += 1

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

2. Selection Sort

Descripción conceptual

Selection Sort busca el mínimo de la parte no ordenada de la lista y lo coloca en su posición final. En cada pasada:

1. Se busca el mínimo.
2. Se intercambia con el elemento en la posición correspondiente.

Variables utilizadas

- `i`: marca dónde empieza la parte **no ordenada**, es decir, los elementos que aún hay que colocar en su posición correcta.
- `j`: cursor que busca el mínimo.
- `min_idx`: índice del menor elemento encontrado.
- `fase`: `"buscar"` o `"swap"`.

Decisiones y dificultades

- Fue necesario dividir el algoritmo en dos fases para que el visualizador pudiera mostrar cada comparación.
- `step()` nunca hace más de una acción: o compara, o hace el swap.
- Manejar el fin de la fase `"buscar"` sin bucles es lo más delicado.

Código implementado

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

items = []
n = 0
i = 0
j = 0
min_idx = 0
fase = "buscar"

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

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

    if fase == "buscar":
        if j < n:
            j_actual = j
            if items[j] < items[min_idx]:
                min_idx = j
            j += 1
            return {"a": min_idx, "b": j_actual, "swap": False, "done":
False}
        else:
            fase = "swap"

    if fase == "swap":
        if min_idx != i:
            a = i
            b = min_idx
```

```
            items[a], items[b] = items[b], items[a]
            fase = "buscar"
            i += 1
            j = i + 1
            min_idx = i
            return {"a": a, "b": b, "swap": True, "done": False}

        fase = "buscar"
        i += 1
        j = i + 1
        min_idx = i
        return {"a": i - 1, "b": i - 1, "swap": False, "done": False}
```

3. Insertion Sort

Descripción conceptual

Insertion Sort recorre la lista de izquierda a derecha. En cada paso toma un elemento y lo "inserta" en la posición correcta dentro de la parte ya ordenada.

Variables utilizadas

- `i`: índice del elemento actual a insertar.
- `j`: cursor que se mueve hacia la izquierda.
- `items`: lista interna.

Decisiones y dificultades

- Controlar correctamente el avance y reinicio de `j` sin bucles.
- Asegurar que cada `step()` haga solo un swap o una comparación, y no más.
- Resetear `j = None` para marcar el inicio de cada inserción fue clave.

Código implementado

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

items = []
```

```
n = 0
i = 0
j = None

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

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

    if j is None:
        j = i
        return {"a": j-1, "b": j, "swap": False, "done": False}

    if j > 0 and items[j-1] > items[j]:
        a = j-1
        b = j
        items[a], items[b] = items[b], items[a]
        j -= 1
        return {"a": a, "b": b, "swap": True, "done": False}

    i += 1
    j = None
    return {"done": False}
```

Reflexión final

Los tres algoritmos implementados (Bubble, Selection e Insertion Sort) permiten comprender distintas estrategias de ordenamiento:

- Bubble Sort es simple pero ineficiente; se destaca por su claridad visual.
- Selection Sort minimiza la cantidad de swaps, aunque no reduce las comparaciones. Esto significa que, aunque **Selection Sort hace pocos intercambios (swaps)** porque solo mueve cada mínimo una vez por pasada, **sigue comparando muchos elementos**: en cada pasada recorre toda la parte no ordenada para encontrar el mínimo.

- **Insertion Sort**: eficiente cuando la lista ya está casi ordenada; mueve pocos elementos y es rápido en la práctica, aunque en listas muy desordenadas tiene un comportamiento similar a Bubble o Selection.

La principal dificultad del TP fue adaptar algoritmos que normalmente se escriben con bucles a una forma paso a paso usando step(). Esto nos obligó a pensar cada acción de manera muy detallada y a organizar todo el código de forma lógica. Además, tuvimos que entender la necesidad de mostrar el diccionario, aunque no se haga un swap, porque el visualizador lo necesitaba para poder mostrarlo. Para quienes no habíamos visto algunos de los conceptos necesarios para la programación del algoritmo, ya sea en la materia de introducción a la programación o fuera de clase, fue difícil de entender al principio, y eso hizo que nos llevara más tiempo terminar el trabajo. Al final, recién cuando todo funcionaba en el visualizador, entendimos cuál era el objetivo del TP: programar algo que funcione como una "máquina", en nuestro caso, mostrar lo que hace el visualizador paso a paso.

Métricas y observaciones

Para complementar la visualización, se midieron los tiempos de ejecución reales de cada algoritmo utilizando listas de 60 elementos con valores aleatorios y comparando swaps:

Algoritmo	Swaps	Tiempo en visualizador (seg)	Tiempo real Python (seg)	Observaciones
Bubble Sort	819	1.40	0.000749	Es el más lento en la animación, refleja su complejidad $O(n^2)$.
Insertion Sort	819	0.53	0.000380	Muy eficiente en listas parcialmente ordenadas.

Selection Sort	56	1.13	0.000675	Menos swaps que Bubble, pero aún $O(n^2)$ en comparaciones.

Nota sobre el visualizador:

- Cada micro-paso se ejecuta con un **delay de 50 ms por barra**, controlable en la UI. Es decir, el tiempo fijo que tarda antes de mostrar el siguiente paso.
- El tiempo que se ve en pantalla depende de esta velocidad y **no refleja el tiempo real de ejecución de la CPU**. Si se aumenta o disminuye ese delay, la animación se verá más lenta o más rápida, pero **el algoritmo en sí no cambió su velocidad de ejecución real**.

Medición del tiempo con código Python:

- Se utilizó un script que ejecuta los algoritmos y mide el tiempo real con `time.perf_counter()`.
- El código hace lo siguiente:
    1. Inicializa la lista de datos y el algoritmo (`init()`).
    2. Ejecuta cada micro-paso del algoritmo con `step()` hasta completarlo.
    3. Cuenta la cantidad de swaps realizados.
    4. Mide el tiempo transcurrido desde el primer step hasta que termina.
- Este tiempo es mucho más rápido que el que se ve en el visualizador porque **no incluye la animación ni los delays**, solo las operaciones internas del algoritmo.

Complejidad y número de operaciones:

- Bubble Sort e Insertion Sort tienen complejidad $O(n^2)$, por lo que el número de operaciones crece aproximadamente con el cuadrado del tamaño de la lista.

- Selection Sort hace menos swaps, pero realiza muchas comparaciones, por eso también se mantiene en $O(n^2)$.
- Esta diferencia explica por qué, aunque los swaps de Bubble e Insertion sean iguales, el tiempo que se ve en pantalla para Bubble Sort es mayor: hace muchas más comparaciones.
- Entender $O(n^2)$ ayuda a dimensionar el problema: si duplicamos la cantidad de elementos, el número de operaciones se cuadruplica, lo que impacta directamente en el tiempo de ejecución. Por ejemplo, si se tiene 10 elementos, puede haber ~100 comparaciones. Eso es lo que impacta en el tiempo de ejecución: **más elementos → más operaciones → más tiempo**.

Bibliografía

- Wikipedia. *Selection Sort*.
- Wikipedia. *Insertion Sort*.
- Wikipedia. *Bubble Sort*.
- Material provisto en el aula virtual.
- Christian A. Morales. *Ordenamiento por Inserción | Insertion Sort* [Video]. YouTube. https://www.youtube.com/watch?v=Hd5jp935ays&t=201s
- Chio Code. *Ordenamiento por Selección | Selection Sort* [Video]. YouTube. https://www.youtube.com/watch?v=Myy-eU-SWbE