

Licenciatura en Sistemas

Trabajo Práctico

Algoritmos de Ordenamiento

Introducción a la Programación
segundo semestre 2025

Resumen: En este trabajo se desarrollarán distintos tipos de algoritmos de ordenamiento, aplicando los conocimientos adquiridos a lo largo de la materia. Veremos el funcionamiento de nuestro algoritmo a través de un visualizador ya programado.

Integrantes: Jaimes Valentina *valentinajaimes003@gmail.com*
 Pereyra Rios Marco Leon *marcoleonpereyrarios@gmail.com*

1. Introducción

En el presente trabajo se llevará a cabo el desarrollo e implementación de diversos algoritmos de ordenamiento a partir de los contenidos trabajados en la materia. Nuestro objetivo principal es comprender el funcionamiento de cada algoritmo a partir de una investigación previa, para luego poder aplicar la estrategia de ordenamiento correspondiente y observar, a través del visualizador ya desarrollado, cómo se comparan los elementos, cómo cambian de lugar y cómo avanza el proceso hasta llegar al resultado final.

2. Desarrollo

Bubble sort:

El objetivo fue diseñar una función "step" que permite inicializar la estructura de datos y avanzar un único paso del algoritmo en cada llamada, devolviendo información útil para visualizar o depurar el proceso de ordenamiento.

El sistema mantiene un estado interno compuesto por:

- ítems: la lista que se está ordenando
- n: su longitud
- i: la pasada actual del algoritmo
- j: el índice del par que se está comparando en este paso

La función "init" configura ese estado inicial, y durante la función "Step" la idea general es ejecutar un único paso del Bubble Sort. En cada llamada compara un par de elementos consecutivos, realiza el intercambio si corresponde y actualiza los índices, entonces devuelve información estructurada para saber qué ocurrió en ese paso.

Esto permite, visualizar el proceso en tiempo real, ejecutar el algoritmo de manera controlada, integrar el ordenamiento a sistemas que requieren paso a paso (animaciones, depuración, etc.).

"Código comentado"

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

    # Si no hay elementos suficientes o todas las pasadas ya se hicieron
    if n < 2 or i >= n - 1:
        return {"done": True}

    a = j # Primer índice del par a comparar
    b = j + 1 # Segundo índice del par
    swap = False

    # Si el par está desordenado, se intercambia
    if items[a] > items[b]:
        items[a], items[b] = items[b], items[a]
        swap = True

    j += 1 # Avanza al próximo par

    # Si llegamos al final de la pasada
    if j >= n - i - 1:
        j = 0 # Reinicio del índice
        i += 1 # Nueva pasada

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

Este código devuelve un diccionario con:

- a: índice del primer elemento comparado
- b: índice del segundo
- swap: True si hubo intercambio
- done: True si el algoritmo terminó

Selection sort:

Para este algoritmo cada llamada a “step()” avanza un único paso, este paso realiza una comparación o intercambio y devuelve información sobre lo ocurrido.

El algoritmo se divide en dos fases internas, durante la primera, se recorre la parte no ordenada de la lista buscando el mínimo. Y en la segunda, se intercambia ese mínimo con el elemento en la posición actual “i”.

Estas fases se controlan mediante la variable “fase”, lo que permite que el algoritmo pueda ser visualizado o depurado paso a paso sin perder coherencia entre llamadas.

“Código comentado”

```
def step():
    global i, j, min_idx, fase
    # Si ya colocamos todos los elementos, terminamos
    if i >= n - 1:
        return {"done": True}
    # -----
    # FASE 1: BUSCAR EL MÍNIMO
    # -----
    if fase == "buscar":
        # Si j ya recorrió todo, pasamos a la fase de swap
        if j >= n:
            fase = "swap"
            return step() # Ejecutamos inmediatamente la fase swap
        # Si encontramos un valor menor, actualizamos min_idx
        if items[j] < items[min_idx]:
            min_idx = j
        # Guardamos el índice actual para reportarlo
        current_j = j
        # Avanzamos j al siguiente elemento
        j += 1
        # Devolvemos la comparación
        return {"a": min_idx, "b": current_j, "swap": False, "done": False}
    # -----
    # FASE 2: SWAP FINAL
    # -----
    if fase == "swap":
        # Si el mínimo no está en posición i, hacemos el intercambio
        if min_idx != i:
            items[i], items[min_idx] = items[min_idx], items[i]
            result = {"a": i, "b": min_idx, "swap": True, "done": False}
        else:
            # Si ya estaba en su lugar, no hay swap
            result = {"a": i, "b": min_idx, "swap": False, "done": False}
        # Avanzamos a la siguiente pasada
        i += 1
        # Si llegamos al final, terminamos
        if i >= n:
            return {"a": 0, "b": 0, "swap": False, "done": True}
        # Reiniciamos valores para la nueva pasada
        j = i + 1 # Cursor empieza justo después de i
        min_idx = i # El mínimo inicial es el de la nueva posición i
        fase = "buscar" # Volvemos a fase de búsqueda
    return result
```

Este código devuelve un diccionario con:

- a: índice del elemento considerado mínimo o comparado
- b: índice del elemento comparado en ese paso
- swap: True si se realizó un intercambio
- done: True si el algoritmo terminó

Insertion sort:

El algoritmo INSERTION recorre la lista de izquierda a derecha, en cada paso que realiza toma el elemento actual y lo compara con el anterior. Para ello, parte desde el segundo elemento de la lista (ya que el primero lo considera como ordenado), toma el elemento actual (en nuestro caso el puntero *i*) y lo inserta en la posición correcta, a la izquierda, para luego desplazar a los elementos que sean mayores a su derecha.

Funciones implementadas:

Para adaptarlo a nuestro visualizador previsto, tenemos en cuenta que el algoritmo se divide en micro-pasos, de modo tal que cada llamada a step() realiza solo una acción pequeña: una comparación o un swap.

Estado del algoritmo:

- **items:** la lista que se está ordenando
- **n:** la longitud total de la lista
- **i:** el índice del elemento que se está intentando insertar en la parte ordenada
- **j:** el índice que se mueve hacia la izquierda comparando e intercambiando elementos adyacentes (elementos cercanos)
- **steps:** cantidad de pasos ejecutados.
- **comparaciones:** cantidad de comparaciones realizadas.

En la función init() configuramos el estado inicial, declarando *i* = 1 ya que consideraremos al primer elemento como ordenado y *j* = None para indicar que la inserción aún no se ha iniciado.

Ya para la función step(), se ejecutará un único micro-paso, arrancando con la activación de *j* para iniciar la comparación. Seguido a esto, se compara los elementos adyacentes (*items[j]* y *items[j-1]*) y en caso de que estén en el orden incorrecto se realiza el intercambio y *j* retrocede para seguir insertando dentro de la parte ordenada (izq). En caso de no haber intercambio, avanza *i* y reinicia *j*.

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

```

    steps += 1    # se contó un micro-paso
    print(f"DEBUG  {steps}:  Comparaciones={comparaciones},  Steps={steps}")
#Muestra en DevTools (f12)

    if i >= n:
        # Si i >= n: devolver {"done": True}.
        print(f"***  ALGORITMO  FINALIZADO.  Comparaciones  totales:
{comparaciones}, Steps totales: {steps} ***")
        return {"done": True}

    # - Si j es None: empezar desplazamiento para el items[i] (p.ej., j = i) y
devolver un highlight sin swap.
    if j is None:
        j = i
        return {"a": j-1, "b": j, "swap":False, "done": False}

    comparaciones += 1

    # - 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]:
        items[j-1], items[j] = items[j], items[j-1]
        j -= 1
        return {"a": j, "b": j+1, "swap":True, "done": False}

    # - Si ya no hay que desplazar: avanzar i y setear j=None.
    i += 1
    j = None
    return {"a": i-1, "b": i, "swap":False, "done": False}

```

Finalmente, en cada llamada se devuelve qué elementos fueron comparados, si hubo algún swap y si el algoritmo ya terminó de comparar.

Dificultades:

- Fallos con el return
- Manejo de los índices, especialmente a la hora de desplazar j.
- A la hora de sumar métricas

Gnome sort (extra):

El objetivo del Gnome Sort es ordenar una lista verificando un par de elementos a la vez y avanzando o retrocediendo según corresponda. Si el par está en orden, el

algoritmo avanza al siguiente elemento; en caso contrario, intercambia los elementos y retrocede una posición para verificar nuevamente. Este proceso se repite hasta que todos los elementos se encuentran ordenados.

En el caso de Gnome Sort, se puede considerar como una variante del Insertion Sort, ya que ambos buscan insertar elementos en su posición correcta, pero difieren en la forma de moverlos: en Insertion se da mediante desplazamientos y en Gnome se dan retrocesos e intercambios.

Estado del algoritmo:

- **items:** la lista de números que se está ordenando.
- **n:** la cantidad de elementos de la lista.
- **i:** el índice actual del elemento, que se mueve hacia adelante o hacia atrás según el caso.
- **steps:** cantidad de pasos ejecutados.
- **comparaciones:** cantidad de comparaciones realizadas.

Funciones implementadas:

En la función `step()` el algoritmo también avanza en un único paso, en ella definimos que:

El algoritmo empieza en $i = 1$, ya que el primer par válido para comparar es $(0, 1)$.

Luego compara el elemento actual `items[i]` con el anterior `items[i-1]`. En caso de que estén en orden (el de la izquierda es menor o igual), i avanza. Si llega a estar fuera de orden, i los intercambia y luego retrocede para seguir corrigiendo. Si i llegase a retroceder hasta la posición 0, entonces se lo envía de nuevo al índice 1 para que continúe ordenando hasta que $i \geq n$ (donde está todo ordenado y corta).

```
def step():
    global items, n, i
    global steps, comparaciones

    steps += 1    # Se cuenta el paso
    print(f"DEBUG  {steps}:  Comparaciones={comparaciones},  Steps={steps},
i={i}")    #Muestra en DevTools (f12)

    # Si i >= n: devolver {"done": True}. // Fin del programa
    if i >= n:
        print(f"***  ALGORITMO  FINALIZADO.  Comparaciones  totales:
```

```

{comparaciones}, Steps totales: {steps} ***)
    return {"done": True}

    # En caso de que estemos en el inicio, el programa debe avanzar al
siguiente
    if i == 0:
        i = 1
        return {"a": 0, "b": 0, "swap": False, "done": False}

    comparaciones += 1

    # Hago la comparación entre los items
    if items[i] >= items[i - 1]:          # si la comparacion es correcta
entonces avanzo
        i += 1
        return {"a": i - 1, "b": i - 2, "swap": False, "done": False}
    else:
        # En caso de que sea incorrecta, retrocedo e intercambio para seguir
avanzando
        items[i], items[i - 1] = items[i - 1], items[i]
        i -= 1
        return {"a": i, "b": i + 1, "swap": True, "done": False}

```

Dificultades:

- Si bien el algoritmo fue fácil de implementar ya que se parece mucho al insertion, al principio fue un poco complicado aplicar lo investigado según lo que requiere las condiciones del trabajo (el ejemplo investigado estaba realizado en otro lenguaje).

Shell sort (extra):

Con el algoritmo de Shell Sort encontramos que es una mejora del Insertion. Tiene como objetivo principal acelerar el proceso de ordenamiento reduciendo la cantidad de desplazamientos necesarios. Para ello, divide la lista en “sublistas” (mediante un valor llamado gap) que indica cada cuántas posiciones se comparan los elementos.

Funciones implementadas:

El proceso dentro de step() inicia eligiendo un gap grande (como $n/2$) para dividir la lista en subgrupos. Dentro de cada subgrupo, se aplica un ordenamiento por inserción donde luego el gap se reduce progresivamente (siempre a la mitad),

repitiendo el proceso en los distintos subgrupos que se van formando. El algoritmo finaliza cuando el gap se reduce a uno, momento en el cual se ejecuta un Insertion Sort sobre una lista que ya está prácticamente se encuentra semi-ordenada.

Estado del algoritmo:

- **items:** la lista a ordenar.
- **n:** la longitud de la lista.
- **gap:** la distancia entre elementos que se comparan (inicialmente `n // 2`).
- **i:** el índice del elemento actual dentro del grupo definido por el gap.
- **j:** el cursor que permite desplazar un elemento hacia la izquierda usando saltos de tamaño gap.
- **steps:** cantidad de pasos ejecutados.
- **comparaciones:** cantidad de comparaciones realizadas.

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

    steps += 1
    print(f"DEBUG    {steps}:    Comparaciones={comparaciones},
Steps={steps}") #Muestra en DevTools (f12)

    # Si gap llega a 0, fin
    if gap == 0:
        print(f"*** ALGORITMO FINALIZADO. Comparaciones totales:
{comparaciones}, Steps totales: {steps} ***")
        return {"done": True}

    # si j es None, comenzamos nueva insercion con gap
    if j is None:
        if i >= n:
            # reducir gap
            gap //= 2
            if gap == 0:
                print(f"*** ALGORITMO FINALIZADO. Comparaciones totales:
{comparaciones} ***")
                return {"done": True}
            i = gap
            j = None
        return {"a": 0, "b": 0, "swap": False, "done": False}
    j = i
```

```

        return {"a": j - gap, "b": j, "swap": False, "done": False}

    comparaciones += 1
    # comparar con separacion gap
    if j - gap >= 0 and items[j] < items[j - gap]:
        a = j
        b = j - gap
        items[a], items[b] = items[b], items[a]
        j -= gap
        return {"a": a, "b": b, "swap": True, "done": False}

    # - Si ya no hay que desplazar: avanzar i y setear j=None.
    i += 1
    j = None
    return {"a": 0, "b": 0, "swap": False, "done": False}

```

Dificultades:

- manejo y aplicación del gap
- adaptar la lógica al modelo paso a paso (ya que en la investigación la lógica original del algoritmo está basada en bucles anidados)

Funciones extras agregadas:

1. Implementación de métricas en 3 algoritmos similares para la comparación de eficacia:

El objetivo principal en este caso fue comparar la eficiencia y el rendimiento de 3 de los algoritmos implementados: Insertion sort, Gnome sort y Shell sort. Si bien estos algoritmos cumplen una función parecida, podemos decir que cada uno desarrolla una estrategia interna diferente. Por esta misma razón, se ha tomado la decisión de aplicarles ciertas métricas para medir su eficacia y finalmente poder decidir cuál de los 3 algoritmos maneja mejor la complejidad dada a la hora del ordenamiento.

En el siguiente cuadro comparativo presentamos uno de los tantos resultados obtenidos al ejecutar cada algoritmo. Tomaremos como pie la velocidad predeterminada del visualizador y la cantidad de ítems serán 50.

Algoritmo	Cant. Items	Velocidad	Steps totales	Comparaciones totales	Tiempo de duración
Insertion sort	50	50 ms	679	629	43 seg
Gnome sort	50	50 ms	1100	1096	1min 10seg
Shell sort	50	50 ms	556	348	36 seg

Con este recuadro podemos concluir que:

Shell sort tiene el menor número de comparaciones y el menor tiempo de duración, siendo entonces el algoritmo más eficiente gracias a su estrategia de gaps que le permite reducir grandes cantidades de steps.

```
DEBUG 556: Comparaciones=348, Steps=556
*** ALGORITMO FINALIZADO. Comparaciones totales: 348
```

Por otro lado, comparando los dos algoritmos restantes podemos decir que Gnome, a diferencia de Insertion, tiene mayor número de steps dada por su lógica del retroceso (vuelve atrás para realizar el intercambio) lo que requiere de más micro-pasos.

```
*** ALGORITMO FINALIZADO. Comparaciones totales: 629,
Steps totales: 679 ***
```

Insertion

```
*** ALGORITMO FINALIZADO. Comparaciones totales: 1096,
Steps totales: 1100 ***
```

Gnome

2. Cambios en la apariencia del visualizador:

Para cambiar la apariencia del Visualizador, se realizaron cambios el código "index"

Estos cambios se realizaron los apartados "Style" y las funciones "makeImageColumns" y "draw"

En el apartado "Style" se cambio sus codigos de color "Hex" y "RGB"

```
/style {
root { --bg:#eb0000; --panel:#000000; --muted:#ffffff; --text:#ffffff; --accent:#ff0000; }
@keyframes loading{from{width:0%;}to{width:100%;}}
body{margin:0;font-family:sans-serif;-apple-system,Segoe UI,Roboto,Ubuntu,Cantarell,Arial;background:linear-gradient(10deg,#000000,var(--bg));color:var(--text);min-height:100vh;display:grid;grid-template-rows:auto 1fr;
font-size:16px;letter-spacing:-0.5px;line-height:1.2;position:relative;z-index:0;}
h1{font-size:18px;margin:0;letter-spacing:-1px;}
p{padding:4px 10px;border-radius:999px;background:#000000;color:#ffffff;font-size:12px;}
main{display:12px 16px 20px;display:grid;grid-template-rows:auto 1fr auto;gap:12px;}
controls{display:grid;grid-template-columns:repeat(auto-fit,minmax(200px,1fr));gap:10px;width:100%}
card{background:var(--panel);border:1px solid #ffffff;border-radius:12px;padding:12px}
label{font-size:12px;color:var(--muted);display:block;margin-bottom:6px}
input[type=range],input[type=file]{width:100%;padding:10px;border:1px solid #ffffff;background:#000000;color:var(--text)}
.row{display:flex;align-items:center;gap:10px}
.row>*{flex:1}
.btns{display:flex;flex-wrap:wrap;gap:10px}
button{background:#000000;color:var(--text);border:1px solid #ffffff;padding:10px 14px;border-radius:12px;cursor:pointer;transition:transform .06s ease,border-color .2s ease}
button:hover{border-color:var(--accent)}
button.active{transform:translateY(-1px)}
button.primary{background:var(--accent);color:#000;text-align:center}
canvas{width:100%;height:64vh;display:block;border-radius:16px;padding:10px;position:relative}
#visu{width:100%;height:64vh;display:block;border-radius:16px;padding:10px;background:radial-gradient(100px 50px at 20% -10%,rgba(60,0,0),0),transparent 0%,radial-gradient(900px 500px at 80% 110%,rgba(60,0,0),0),transparent 70%,#000000}
.footer{color:var(--muted);font-size:12px;text-align:center;padding:8px 0}
kbd{font-family:ui-monospace,SFMono-Regular,Menlo,Consolas,monospace;background:#000000;border:1px solid #ffffff;padding:2px 6px;border-radius:6px}
}
```

En la función “makelmImageColumns” se cambio el codigo de color hex para cambiar el color de la "imagen" que se puede ver cuando en “Dataset” se selecciona “Imagen por columnas”

```
// ===== Imagen por columnas =====
function makeImageColumns(n){
  return new Promise(function(resolve){
    if(!state.imageBitmap){
      var off = document.createElement('canvas');
      off.width = 800; off.height = 500;
      var octx = off.getContext('2d');
      octx.imageSmoothingEnabled = false;
      var grad = octx.createLinearGradient(0,0,off.width,off.height);
      grad.addColorStop(0,'#000000'); grad.addColorStop(1,'#eb0000');
      octx.fillStyle = grad; octx.fillRect(0,0,off.width,off.height);
      state.imageBitmap = off; // canvas fuente
    }
  });
}
```

En la función “draw” se realizaron cambios en el código de color hex, para alterar la apariencia de las barras que se puede ver cuando en “Dataset” se selecciona “Barras”

```
// ===== Dibujo =====
function draw(){
  if(!ctx) return;
  clearCanvas();

  var W = canvas.width, H = canvas.height;

  if(state.mode === 'bars'){
    var n = state.items.length; if(!n) return;
    var gap = Math.round(2 * DPR);
    var usable = W - gap*(n+1);
    if (usable < 1) return;

    var widths = columnWidths(n, usable);
    var maxVal = 1;
    for(var i=0;i<n;i++) if(state.items[i].value > maxVal) maxVal = state.items[i].value;

    var x = gap;
    for(var k=0;k<n;k++){
      var it = state.items[k];
      var w = widths[k] | 0;
      var h = Math.max(2*DPR, (it.value/maxVal) * (H - 10*DPR)) | 0;
      var y = (H - h) | 0;
      var hi = (state.highlight.a===k || state.highlight.b===k);
      ctx.fillStyle = hi ? '#eb0000' : '#ffffff';
      ctx.fillRect(x|0, y|0, w|0, h|0);
      x += w + gap;
    }
  }
}
```

3. Conclusiones

A lo largo del desarrollo hemos podido ver cómo finalmente han funcionado nuestros algoritmos de ordenamiento, Bubble Sort, Selection Sort, Insertion Sort, Gnome Sort y Shell Sort, donde todos han podido ser adaptados al modelo paso a paso mediante la función `step()`. Gracias a este enfoque fue posible visualizar y depurar cada etapa del ordenamiento y comprender el comportamiento real de cada algoritmo.

Por otra parte, apoyándonos sobre el contenido teórico de la materia, hemos podido aplicar métricas de rendimiento (swaps, comparaciones y tiempo de duración) a tres de nuestros algoritmos, lo cual nos ha permitido observar la eficacia de los mismos. Sumado a esto, se ha podido realizar cambios en la apariencia del visualizador, siendo esto posible gracias a una investigación previa.

Si bien el trabajo ha transcurrido a partir de pruebas y errores, este proyecto nos ha permitido poder aplicar nuestros conocimientos, hemos comprendido la lógica de los distintos tipos de algoritmos, nos hemos afianzado un poco más a la hora de romper un código ya desarrollado y finalmente pudimos satisfacer nuestros objetivos logrando que todos nuestros algoritmos funcionaran correctamente.

Anexo:

TP - Visualización de algoritmos de ordenamiento

Objetivos:

Implementar al menos 3 algoritmos distintos de ordenamiento cumpliendo el contrato `init(vals) + step()` que usa el visualizador.

A implementar:

- **Bubble**
- **Selection**
- **Insertion.**

Entregable:

- **(Obligatorio)** Carpeta `/algorithms/` con al menos 3 archivos funcionando (ej.: `sort_bubble.py`, `sort_selection.py`, `sort_insertion.py`).
- **(Obligatorio)** Informe detallado documentando los algoritmos implementados junto con las decisiones tomadas, implementaciones aplicadas y dificultades encontradas.

- **(Obligatorio)** Un README.md breve con: integrantes, algoritmos implementados y una nota corta sobre decisiones de implementación.
- **(Opcional)** Algoritmos extra (sort_quick.py, sort_merge.py, sort_shell.py, ...)
- **(Opcional)** Agregar métricas sobre cantidad swaps, tiempo de duración, etc
- **(Opcional)** Quien quiera explorar el código JS para cambiar cómo se corta/mezcla la imagen puede hacerlo y documentarlo como anexo.