Informe de tp de algoritmos

Comisión -04

Profesores: Omar Argañaras y Nancy Gómez

Alumnos: Martin Saavedra y Torres Bautista

En este informe voy a hablar sobre tres algoritmos de ordenamiento: Bubble Sort, Insertion Sort y Selection Sort. La idea es mostrar cómo funcionan, cómo los fui armando paso por paso y qué cosas me complicaron mientras los hacía. Estos algoritmos son como la base para entender otras estructuras más complejas que después se usan en programas reales, tanto en la escuela como en trabajos. Aunque hoy existan métodos mucho más rápidos, estos son los que te ayudan a entender de verdad cómo se comparan datos, cómo se mueven y cómo van quedando ordenados.

Para hacer este trabajo práctico, cada algoritmo usamos una función step() que ayuda un montón porque podés ver exactamente qué está pasando adentro del algoritmo: qué compara, si cambia elementos de lugar o no, y cómo va avanzando hasta terminar. Esta forma de hacerlo también ayuda a encontrar errores, entender los casos especiales y evitar problemas si los índices se van de rango.

Tambien en el informe se redactó qué problemas tuve al hacerlo paso a paso y qué decisiones tomé para que quede bien. La idea no es explicar cómo funciona Python ni nada del lenguaje, sino mostrar claramente cómo está pensado cada algoritmo por dentro y que se entienda bien el proceso de ordenamiento.

Bubble Sort:

El Bubble Sort básicamente funciona recorriendo la lista una y otra vez, comparando elementos que están al lado y cambiándolos de lugar si están desordenados. Para poder hacerlo pasó por paso, tuve que dividir todo en dos índices: **i**, que marca cuántas pasadas completas ya se hicieron, y **j**, que es la posición dentro de la pasada actual.

Esto trajo varios problemas, sobre todo con los límites del recorrido, porque en cada pasada nueva el algoritmo ya no tiene que comparar los elementos que ya quedaron en su lugar. Otra complicación fue lograr que cada llamada a la función step() hiciera *solo* una comparación (y el intercambio si hacía falta), sin que el algoritmo se adelantara dos pasos o hiciera comparaciones demás.

También tuve que decidir exactamente cuándo reiniciar **j** y cuándo aumentar **i**, porque eso es clave para que el algoritmo funcione igual que el original. Al final, usé un sistema

simple: cuando **j** llega al límite de la pasada, vuelve a cero y **i** avanza. Así el algoritmo va armando el ordenamiento de manera lenta pero segura, paso a paso

Código:

```
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 n <= 1 or 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
    if j >= n - i - 1:
        j = 0
        i += 1
    if i >= n - 1:
        return {"done": True}
    return {"a": a, "b": b, "swap": swap, "done": False}
```

Insertion Sort:

El Insertion Sort funciona como si fueras armando una parte ordenada de la lista de a poco. Cada vez agarra un elemento y lo va moviendo hacia la izquierda hasta que queda en el lugar donde tiene que estar. Para poder hacerlo paso por paso usé dos índices: **i**, que marca qué elemento estoy intentando insertar, y **j**, que es el que se va moviendo hacia la izquierda para acomodarlo.

Una de las cosas que más complicó fue manejar bien el estado de **j**, porque solo tenía que empezar cuando arrancaba una nueva inserción. Otro desafío fue lograr que cada vez que

llamara a step() el algoritmo hiciera *solo* una comparación o un movimiento, y no toda la inserción de una.

También decidí que cuando empieza una nueva inserción, la función devuelva una acción sin intercambio, así se nota bien cuándo arranca ese proceso. Gracias a eso se puede ver clarito cómo la parte ordenada va creciendo paso a paso y cómo cada elemento se va ubicando en su lugar de forma controlada.

Código:

```
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== None:
         j = i
         return {"a":j,"b":j-1, "swap":False,"done":False}
    if j > 0 and  items[j-1] > items[j]:
        items[j],items[j-1]=items[j-1],items[j]
        j-= 1
        return {"a":j+1,"b":j,"swap":True,"done":False}
    i+= 1
    j = None
    return {"a":i+1,"b":i,"swap":False,"done":False}
```

Selection Sort:

El Selection Sort funciona buscando, en cada pasada, el elemento más chico de la parte que todavía no está ordenada y poniéndolo en el lugar donde corresponde. Para poder hacerlo paso por paso tuve que separar todo en dos fases: una donde busco el mínimo y otra donde hago el intercambio. Dividirlo así ayuda un montón porque evita confusiones y hace que cada vez que llamo a step() el algoritmo haga solo una acción lógica.

Una de las partes más complicadas fue manejar el momento en el que el algoritmo tenía que pasar de la fase de “buscar” a la fase de “swap”, sobre todo cuando la búsqueda del mínimo llegaba al final del segmento que tocaba revisar. Tuve que definir bien ese cambio para que todo funcione ordenado.

También fue importante actualizar siempre el índice **min_idx**, que es el que marca dónde está el mínimo encontrado. Si ese valor quedaba viejo o se cambiaba en un momento equivocado, todo el algoritmo empezaba a fallar.

Al final, la implementación hace que cada step() muestre claramente si se hizo una comparación o un intercambio, y gracias a eso se puede ver sin problemas cómo se elige el mínimo, cómo se lo cambia y cómo se va ordenando toda la lista paso a paso.

Código:

```python
j = 0
min_idx = 0
fase = ""
items = []
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":
        j_actual = j if j < n else n - 1
        if j < n:
            if items[j] < items[min_idx]:
                min_idx = j
            j += 1
            return {"a": min_idx,"b": j_actual,"swap": False,"done": False }
        fase = "swap"
        return { "a": min_idx,"b": j_actual,"swap": False, "done": False }
    if fase == "swap":
        hizo_swap = False
        i_actual = i
        min_actual = min_idx
        if min_idx != i:

            items[i], items[min_idx] = items[min_idx], items[i]
            hizo_swap = True
        i += 1
        j = i + 1
        min_idx = i
        fase = "buscar"
```

```
fase = ""
items = []
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":
        j_actual = j if j < n else n - 1
        if j < n:
            if items[j] < items[min_idx]:
                min_idx = j
            j += 1
            return {"a": min_idx,"b": j_actual,"swap": False,"done": False }
        fase = "swap"
        return { "a": min_idx,"b": j_actual,"swap": False, "done": False }
    if fase == "swap":
        hizo_swap = False
        i_actual = i
        min_actual = min_idx
        if min_idx != i:

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

También agregamos otro index.html que nos da información general de los algoritmos y al presionar el nombre del algoritmo en el cuerpo del texto nos re direcciona directo a la página para comprobar el funcionamiento de los algoritmos de forma visual.

Un ejemplo de una línea de código que nos permite re direccionarnos al otro índex es <h2>Insertion sort</h2>

```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Programming - Algoritmos</title>
    <link rel="stylesheet" href="Styles.css">
    <link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/font-awesome/6.0.0-beta3/css/all.min.css">
</head>
<body>

    <header class="navbar">
        <div class="logo-container">
            <i class="fas fa-gamepad logo-icon"></i>
            <span class="site-name">Programming</span>
        </div>
    </header>

    <div class="main-content">
        <aside class="sidebar">
            <nav>
                <h3>Algoritmos</h3>
                <ul>
                    <li><a href="#bubble-sort">Bubble Sort</a></li>
                    <li><a href="#insertion-sort">Insertion Sort</a></li>
                    <li><a href="#selection-sort">Selection Sort</a></li>
                </ul>
            </nav>
        </aside>

        <main class="content-body">
            <h1>Conceptos Generales sobre Algoritmos en Python</h1>
            <p>
                Los algoritmos son el **corazón de la programación**, definiendo una serie de pasos finitos y bien definidos para resolver un problema. En el contexto de **Python**, un lenguaje de programación de alto nivel conocido por su legibilidad, la imple
            </p>

            <p>
                La **eficiencia** de un algoritmo se mide generalmente por su **complejidad temporal** y **espacial**, lo que se describe utilizando la **notación de la Gran O** ($O(n)$). Elegir el algoritmo adecuado puede hacer una diferencia significativa en 
            </p>

            <section id="bubble-sort">
                <h2> <a href="index.html" >Bubble Sort</a></h2>
                <p>
                    El **Bubble Sort** (Ordenamiento de Burbuja) es un algoritmo de ordenamiento simple que funciona revisando repetidamente la lista a ordenar e intercambiando los elementos adyacentes si están en el orden equivocado. Se caracteriza por su simp
                </p>
                <div class="example-code">
                    <h4>Ejemplo básico en Python:</h4>
                    <pre><code>
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        for j in range(0, n-i-1):
            if arr[j] > arr[j+1]:
                arr[j], arr[j+1] = arr[j+1], arr[j]
    return arr
                    </code></pre>
                </div>
            </section>
```

Conclusión:

Para terminar, este trabajo nos sirvió un montón para entender mejor cómo funcionan los algoritmos de ordenamiento por dentro. Al principio parecía fácil, pero cuando empezamos a hacerlos paso por paso con la función step() se nos complicó bastante, porque había que controlar bien los pasos, los índices y que no se rompa todo. Igual, después de varios intentos y errores, logramos que Bubble Sort, Insertion Sort y Selection Sort funcionen como queríamos y de forma visual, que ayuda mucho a entenderlos.

Nos quedamos con ganas de agregarle más cosas al HTML, como más información, mejoras visuales o incluso más algoritmos, pero el código ya era bastante largo y se nos hacía difícil encontrar todo sin perdernos. También intentamos agregar el Shell Sort, pero se me hizo muy complicado meter un nuevo algoritmo porque se empezaba a buguear todo y en el localhost a veces ni se veía bien el código, así que decidimos dejarlo así para que funcione bien lo que ya teníamos.

En general, fue un trabajo que nos hizo pensar bastante, equivocarnos, probar muchas veces y aprender cómo funcionan de verdad estos algoritmos. Más allá de las complicaciones, creemos que el resultado final está bueno y cumple con lo que nos pidieron