Universidad Nacional de General Sarmiento

TP Visualización de algoritmos de ordenamiento

Profesores:

Nora Martinez, Flavia Bottino

Cátedra:

Introducción a la programación

Comisión:

12

Integrantes:

Álvarez Ezequiel

De Nevares Jaime

Oberto Danilo

Romero Melanie

Introducción

En este informe se presentarán 5 algoritmos de ordenamiento en la programación, usando el lenguaje de python. Se describe el funcionamiento de cada método, qué problemas nos encontramos a la hora de implementarlos, y las soluciones que tomamos para resolver las complicaciones encontradas.
Además, se explicará también las funciones y todo aquello que agregamos al proyecto base para hacer un trabajo mucho más completo en cuanto a una herramienta educativa para aprender sobre algoritmos de ordenamiento.
Finalmente, se realizará una comparación de su eficacia con el objetivo de ver sus diferencias.

Código de Algoritmos de ordenamiento

1. Metodo burbuja (bubble sort):

```
# ============================================================
# BUBBLE SORT - IMPLEMENTACIÓN ROBUSTA CON CONTADORES
# Contrato de retorno extendido:
# {"a": int, "b": int, "swap": bool, "done": bool, "comp_count": int,
"swap_count": int}
# ============================================================

# Variables de estado del algoritmo
items = []
n = 0
i = 0  # Índice de pasadas (elementos ya ordenados al final)
j = 0  # Índice de comparación actual (la "burbuja")

# Variables de rendimiento
comparison_count = 0
swap_count = 0

def init(vals):
    # Declaramos globales las variables de estado y rendimiento
    global items, n, i, j, comparison_count, swap_count

    items = list(vals)
    n = len(items)
    i = 0  # Reiniciar pasadas
    j = 0  # Reiniciar puntero de burbuja
```

```
    # Reinicialización de contadores
    comparison_count = 0
    swap_count = 0

def step():
    # Declaramos globales las variables que vamos a modificar
    global items, n, i, j, comparison_count, swap_count

    # 1. Revisar si terminamos todas las pasadas (n-1 pasadas son
necesarias)
    if i >= n - 1:
        # Fin del algoritmo
        return {
            "done": True,
            "a": None, "b": None,
            "swap": False,
            "comp_count": comparison_count,
            "swap_count": swap_count
        }

    swap = False

    # Los índices a comparar son j y j+1
    a = j
    b = j + 1

    # 2. Lógica de Bubble Sort

    # INCREMENTO 1: Cada vez que entramos aquí es una comparación de
elementos.
    comparison_count += 1

    if items[a] > items[b]:
        items[a], items[b] = items[b], items[a]  # Intercambiar en el
array interno
        swap = True

        # INCREMENTO 2: Solo incrementamos si realmente hubo un
intercambio.
        swap_count += 1
```

```
    # 3. Avanzar los punteros (índices) para el próximo paso
    j += 1

    # 4. Revisar si la burbuja (j) llegó al final de la parte no
ordenada
    # El final de la pasada es (n - 1 - i)
    if j >= n - 1 - i:
        j = 0     # Reiniciamos la burbuja al inicio del array
        i += 1    # Avanzamos a la siguiente pasada (la última posición
ya está ordenada)

    # 5. Devolver los resultados del paso
    return {"a": a,"b": b, "swap": swap, "done": False, "comp_count":
comparison_count, "swap_count": swap_count}
```

Explicación:

El método Burbuja es un algoritmo de ordenamiento que funciona comparando elementos vecinos de una lista y, si están en el orden incorrecto, los intercambia. Para usarlo, primero se inicializa la lista y los índices que señalan qué dos elementos se comparan. Luego, en cada paso del proceso, se toman dos posiciones consecutivas y se decide si se deben intercambiar. Después de cada comparación, los índices avanzan y cuando llegan al final de la parte no ordenada, se reinician y se reduce la cantidad de elementos restantes, ya que el valor más grande va quedando al final. Este proceso se repite hasta que ya no queda nada por ordenar. En otras palabras, el algoritmo “hace burbujear” los valores más grandes hacia el final mediante comparaciones sucesivas.

Complicaciones:

En lo particular, el algoritmo de burbuja ya lo conocía, conocía la estructura básica y su funcionamiento en el lenguaje C, y al ser uno de los algoritmos más fáciles para aprender no hubo mayor complicación, más que la de entender al inicio que el código debía tener la lógica de las comparaciones como micro pasos, y no el código entero con los bucles for, debido a que es una función que ya se repite al llamarse consecutivamente.

2. Metodo Insertion :

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

items = []
n = 0
i = 0      # elemento que queremos insertar
j = None   # cursor de desplazamiento hacia la izquierda (None = empezar)

def init(vals):
    global items, n, i, j
    items = list(vals)
    n = len(items)
    i = 1      # común: arrancar en el segundo elemento
    j = None
```

```
def step():
    # TODO:
    global items, n, i, j
    # - Si i >= n: devolver {"done": True}.
    if i >= n:
        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": i, "b": j, "swap": False, "done": False}
```

```
    # - Mientras j > 0 y items[j-1] > items[j]: hacer UN swap adyacente (j-1, j) y devolverlo con swap=True.
    if j > 0:
        while items[j-1] > items[j]:
            items[j], items[j-1] = items[j-1], items[j]
            return {"a": j, "b": j-1, "swap": True, "done": False}
    j = j-1
```

```
# - Si ya no hay que desplazar: avanzar i y setear j=None.
if items[j] > items[j-1]:
    i = i + 1
    j = None
return {"done": False, "swap": False}
```

Explicación:

El algoritmo agarra el segundo elemento de la lista (porque el primero ya lo considera automáticamente como ordenado), y lo compara con el elemento anterior, si el elemento que queremos insertar es menor al elemento que está en la lista ordenada, se intercambian de lugar, y se sigue repitiendo ese mismo ciclo mientras la lista se va ordenando. Si la lista ordenada ya tiene varios elementos ordenados, va a tomar el siguiente que no esté ordenado y recorrer toda la lista ordenada de derecha a izquierda hasta posicionarlo en su lugar correspondiente.

Complicaciones:

En lo particular me costó un poco adaptarme a Github y a cómo commitear y actualizar los datos, y en cuanto al código me resultó un poco complicado hacer que lo que estaba haciendo aparezca en pantalla, no había puesto bien los return que hacían falta, y también me tardó bastante hacer bien la parte en la que se intercambian los elementos, porque empezaba pero se detenía apenas empezaba.

3. Metodo selection

Este método se basa primeramente en recorrer toda la lista buscando el elemento más pequeño y colocarlo en la primera posición. Para esto, se busca el mínimo entre

todos los elementos y, si no está en la primera posición,lo cambia de lugar con el primer elemento, de lo contrario como ya es el primer elemento no lo mueve.
En el siguiente paso, el algoritmo busca nuevamente al más pequeño, pero ahora entre los elementos restantes(del segundo al último) y lo coloca en la segunda posición.
Este procedimiento se repite sucesivamente, hasta que llega al último elemento por lo que todos los elementos se encuentran ordenados.

```
items = []
n = 0
i = 0           # cabeza de la parte no ordenada
j = 0           # cursor que recorre y busca el mínimo
min_idx = 0     # índice del mínimo de la pasada actual
fase = "buscar"  # "buscar" | "swap"
```

```
def step():
    global items, n, i, j, min_idx, fase
    if fase == "buscar":
        if  j < n:
            if items[j] < items[min_idx]:
                min_idx = j
            j = j+1
            return {"a": min_idx, "b": j, "swap": False, "done": False}
```

```
    fase = "swap"

    if fase == "swap":

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

    return {"done": True}
```

Inconvenientes:
El principal inconveniente que tuve con este método fue más que nada entender cómo funcionaban los return y entender cómo hacer que se repita el ciclo, pero luego de muchos intentos pude entender mejor la lógica detrás de los return y pude terminarlo.

4. Metodo Quick Sort

```
items = []
n = 0
inicio = None
fin = None
i = None
j = None
stack = []

# Defino las variables
def init(vals):
    global items,n,inicio,fin,i,j,stack
    items = list(vals) #la lista
    n = len(items) # la longitud de la lista
    inicio = 0 #
    fin = n-1 #
    i = None #
    j = inicio #
    # TODO: inicializar punteros/estado
    stack.append([inicio, fin])
```

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

    if len(stack) == 0:
        return {"done": True}
    inicioAFin = stack.pop()


    current = inicioAFin[0] + j
    pivot = inicioAFin[1]

    if current < pivot:
        if items[current] > items[pivot]:
            if i == None:
                i = current
            else:
                j=j+1
            stack.append(inicioAFin) #
            print(pivot, current)
            return {"a": pivot, "b": current, "swap": False, "done": False}

        if items[current] < items[pivot]:
            if i == None:
                j=j+1
            else:
                items[i],items[current] = items[current], items[i]
                temp = i
                i = i+1

                stack.append(inicioAFin)
                return {"a": temp, "b": current, "swap": True, "done": False}

        stack.append(inicioAFin)
```

```
else:
    if i != None:
    #Particion de mitades y ordenar pivot
        items[pivot], items[i] = items[i], items[pivot]
        if (inicioAFin[0] < i-1):
            stack.append([inicioAFin[0], i-1])

        if (inicioAFin[1] > i+1):
            stack.append([i+1, inicioAFin[1]])

        temp = i
        #reinicio las variables
        j = 0
        i = None
        return {"a": temp, "b": current, "swap": True, "done": False}
    else:
        if (inicioAFin[0] < inicioAFin[1]-1):
            stack.append([inicioAFin[0], inicioAFin[1]-1])
        j = 0
return {"done": False}
```

Explicacion

Lo primero que hace el método es elegir un pívot, en el caso de mi algoritmo elijo como el pivot el último elemento de la lista, luego hago que recorra toda la lista, poniendo a los más pequeños que el pívot del lado izquierdo y los más grandes del lado derecho, luego con una pila, divido en dos casos, el de los más pequeños y el de los más grandes y los ordeno de la misma manera, elijo último número y los divido.

Ordenamiento

La manera en la cual los ordeno sería la siguiente, uso la variable “i” que me sirve como índice para seleccionar la primera pieza del conjunto que tengo, la variable ”j“ va a recorrer y cuando encuentre un elemento más pequeño que el pívot los intercambia.

Inconvenientes

El mayor inconveniente que tuve con este método, sería el uso del stack/pila para poder ordenar de manera “recursiva” sin usar recursión directamente

5. Merge sort:

```
# Variables de estado
items = []
n = 0

# Pila de tareas: (start, mid, end)
pending_merges = []

# Estado de la fusión actual:
current_merge_state = None
```

```
# Variables de rendimiento
comparison_count = 0
swap_count = 0

def init(vals):
    global items, n, pending_merges, current_merge_state,
comparison_count, swap_count

    items = list(vals)
    n = len(items)
    comparison_count = 0
    swap_count = 0
    current_merge_state = None

    # Generación de tareas iterativa (Bottom-Up)
    pending_merges = []
    size = 1
    while size < n:
        for start in range(0, n - size, size * 2):
            mid = start + size - 1
            end = min(start + size * 2 - 1, n - 1)
            pending_merges.append((start, mid, end))
        size *= 2

def step():
    global items, n, pending_merges, current_merge_state,
comparison_count, swap_count

    # 1. Si no hay tareas ni estado activo, terminamos
    if not pending_merges and current_merge_state is None:
        return {
            "done": True, "a": None, "b": None, "swap": False,
            "comp_count": comparison_count, "swap_count": swap_count
        }

    # 2. Iniciar nueva tarea si es necesario
    if current_merge_state is None:
        start, mid, end = pending_merges.pop(0)
        current_merge_state = {
            "start": start, "mid": mid, "end": end,
            "i": start,      # Inicio del sub-array izquierdo
```

```
            "j": mid + 1,    # Inicio del sub-array derecho
            "mode": 'compare'
        }
        # Retorno "dummy" para resaltar el inicio
        return {
            "a": start, "b": end, "swap": False, "done": False,
            "comp_count": comparison_count, "swap_count": swap_count
        }

    # 3. Procesar estado actual
    st = current_merge_state

    # Verificar si terminamos esta fusión (cuando i o j salen de sus
rangos)
    if st['i'] > st['mid'] or st['j'] > st['end']:
        current_merge_state = None
        return {
            "a": st['end'], "b": st['end'], "swap": False, "done":
False,
            "comp_count": comparison_count, "swap_count": swap_count
        }

    # MODO COMPARACIÓN
    if st['mode'] == 'compare':
        comparison_count += 1
        a_idx, b_idx = st['i'], st['j']

        # Si el elemento izquierdo es menor o igual, ya está en su
sitio
        if items[a_idx] <= items[b_idx]:
            st['i'] += 1
            return {
                "a": a_idx, "b": b_idx, "swap": False, "done": False,
                "comp_count": comparison_count, "swap_count":
swap_count
            }
        else:
            # El elemento derecho (j) es menor. Debe moverse a (i).
            # Cambiamos a modo 'shift' para moverlo burbujeando hacia
la izquierda.
            st['mode'] = 'shift'
            st['shift_k'] = st['j']
```

```
            return {
                "a": a_idx, "b": b_idx, "swap": False, "done": False,
                "comp_count": comparison_count, "swap_count":
swap_count
            }

    # MODO DESPLAZAMIENTO (Simula inserción con swaps)
    elif st['mode'] == 'shift':
        # Intercambiamos shift_k con el anterior (shift_k - 1)
        idx = st['shift_k']
        prev = idx - 1

        items[idx], items[prev] = items[prev], items[idx]
        swap_count += 1

        st['shift_k'] -= 1

        # Si el elemento llegó a la posición 'i', el desplazamiento
terminó
        if st['shift_k'] == st['i']:
            # Ajustamos punteros:
            st['i'] += 1
            st['mid'] += 1
            st['j'] += 1
            st['mode'] = 'compare'

        return {"a": idx, "b": prev, "swap": True, "done": False,
"comp_count": comparison_count, "swap_count": swap_count}

    return {"done": True, "comp_count": comparison_count, "swap_count":
swap_count} # Fallback
```

Explicación:

El algoritmo de Merge Sort sigue un refrán muy conocido, “divide y vencerás”, quiere decir que su método es dividir la lista a la mitad una y otra vez hasta que quedan listas muy pequeñas o de un solo elemento, que por definición una lista de un elemento ya está ordenada, por lo que una vez que tiene ordenadas las listas pequeñas, las fusiona hasta tener todos los elementos ordenados. Es un algoritmo muy eficiente cuando se trabaja con muchos datos, ya que al dividirlos trabaja sobre un menor rango de datos para ordenar.

Complicaciones:

Este algoritmo en realidad utiliza recursividad para la división de la lista en muchas partes, y utiliza un array auxiliar para las listas creadas a partir de la división, por lo que no se podía mostrar en el visualizador “paso a paso” mostrando los swaps uno a uno, porque su funcionamiento es distinto. La solución fue crear una variante "híbrida" que simula la lógica del Merge Sort, pero que realiza los movimientos de datos con la mecánica de un Insertion Sort (empujando los elementos con muchos swaps) para cumplir con el contrato de visualización de JavaScript.

Funciones extras:

A continuación, se detallan y explican de forma concisa las funcionalidades incorporadas al proyecto original de visualización de algoritmos de ordenamiento.

1. Un menú atractivo visualmente que contiene 3 páginas para acceder.
    a. **Visualizador en acción:** esta página es ahora el index original con la interfaz de ordenamiento, con características nuevas como métricas de tiempo en segundos, cantidad de comparaciones por algoritmo, cantidad de intercambios (swaps) por algoritmo, y música de fondo al reproducir.
    b. **Ranking y eficiencia:** esta página está dedicada a listar en orden a los algoritmos presentados en cuanto a la eficiencia que tiene cada uno.
    c. **Análisis de complejidad:** con esta sección queremos añadir un espacio educativo que explique de forma simple y con analogías cómo funcionan los algoritmos que se pueden probar en el visualizador, qué es la notación Big O (forma de medir el rendimiento del algoritmo), etc.
2. Archivos modificados para esto:
    a. La estructura de archivos se modificó para esta aplicación web:
    b. Se modificaron distintas funciones de javascripts que ya estaban definidas en el proyecto, con el fin de adaptarlo a las mejoras incluidas, como los contadores/métricas y tiempo para el control (variables de estado añadidas), se añadieron funciones nuevas que controlan el cronómetro y contadores (resetCounters(), updateCounters(time, comp, swap), y startTimer()), la función Pystep fue crítico modificarla ya que ahora enviamos dos parámetros nuevos (comparaciones y swaps). Añadimos también en la función **draw()** la visualización de los valores de las barras. Por último la función “run” y otras de control como pausar, reset, etc se sincronizan con el uso de música, para que esta solo suene cuando está en “reproducir”.