Universidad Nacional de General Sarmiento

TP
VISUALIZACIÓN DE ALGORITMOS DE ORDENAMIENTO

Alumnos:

- Manuel Javier Aguirre
- Juan David Argamonte

DNI:
- 44456010
- 47694071

Mail:
yatzmanu@gmail.com
juanargamonte60@gmail.com

Introducción a la Programación

Comision 06

Prof. Daniel Bressky y Prof. Esteban Fassio

Introducción

En el presente trabajo hemos desarrollado e implementado una visualización interactiva de distintos algoritmos de ordenamiento, utilizando como base el trabajo un sistema impuesto por los profesores y nuestro objetivo era implementar estos algoritmos y ver cómo funcionan.

La idea principal fue comprender cómo funcionaban los distintos ordenamientos, tanto en lo teórico como en lo práctico, observando el paso a paso y cómo se comparan, intercambian y como se re acomodan estos mismos hasta dejar todo ordenado. Fue un proyecto convertido en ejercicio para ayudar a entender que es lo que hace cada uno por "dentro" en vez de solo verlo y entenderlo. Con una interfaz web que contiene las herramientas necesarias para poder empezar con el proyecto, tales como acciones de mezclar, reproducir, paso a paso y pausa, mientras un código en Python lo va ejecutando. Gracias a la ayuda de los profesores nos quedaba la tarea de programar los algoritmos dentro de la carpeta y lograrlos integrar con el visualizador, pareciendo algo "fácil" o "sencillo" pero con dificultades que nos íbamos a encontrar en el camino, ya que cualquier mínimo error podría afectar que no avancemos.

Implementamos 3 algoritmos, que más adelante vamos a ver cómo se implementan en nuestra carpeta, Bubble Sort, Selection Sort e Insertion Sort. Bubble Sort va comparando elementos hasta dejar un valor final grande. Selection Sort tomando los elementos y en cada vuelta ponerlo en la posición que corresponde. Insertion Sort agarrando el elemento e insertándolo en una parte ya ordenada. Pero esto no funciona como si nada, necesitamos micro pasos, donde estamos parados, qué comparaciones habrá, cuando corresponde intercambiarlos y que hacer una vez ya este terminadas cada una de las vueltas.

En resumen, este trabajo práctico nos permitió repasar algoritmos, pero también entenderlos desde otra perspectiva. Trabajar junto a mi compañero con ayuda de un visualizador web ayuda bastante a ver que realmente está pasando en cada algoritmo, adaptando nuestra lógica al sistema y poder mejorar nuestra forma de programar.

Desarrollo y decisiones

- Algoritmo Bubble Sort:

```
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}

    a= j
    b= j + 1
swap = False
    if items[a] > items[b]:
        aux = items[a]
        items[a] = items[b]
        items[b] = aux
        swap = True
    j +=1
    if j == (n - i - 1):
        j = 0
        i += 1
    return {"a": a, "b": b, "swap": swap, "done": False}
```

- ¿Qué es el algoritmo Bubble Sort? ¿Cómo funciona?:

Bubble Sort funciona comparando valores de pares de dos números en forma adyacente, es decir, el más grande queda al final de la lista , siempre y cuando uno esté al lado del otro. Si detecta que uno es más grande que el siguiente, los intercambia. Al hacer esto muchas veces, el valor más grande "sube" al final de la lista, como dice el nombre clasificación de burbujas.

Dificultades encontradas:

- Dividir el algoritmo en micro pasos para que el visualizador lo entienda.
- Decidir cuándo marcar swap.
- Detectar cuándo terminar el algoritmo

- Algoritmo Insertion Sort:

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

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] < items[j-1]:
        aux = items[j]
        items[j] = items[j-1]
        items[j-1] = aux

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

      i += 1
    j = None

      return {"a": i-1, "b": i-1, "swap": False, "done": False}
```

¿Qué es el algoritmo Insertion Sort? ¿Cómo funciona?:

Insertion Sort es un algoritmo que divide una lista en dos partes: una ordenada y otra desordenada, es decir, se comporta cómo ordenar una baraja de cartas, agarrar una carta, buscar donde iría e insertarla en su lugar. Toma cada ítem o elemento, lo mueve hacía atrás hasta que todo esté ordenado.

Dificultades encontradas:

- Coordinar todas las vueltas con variables como "j".
- Guardar y hacer con cuidado los auxiliares.
- Que haga cada paso a paso con cuidado.

- Algoritmo Selection Sort:

```
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:
            a = min_idx
            b = j
            swap = False
     if items[j] < items[min_idx]:
                min_idx = j
            j += 1
            return {"a": a, "b": b, "swap": swap, "done": False}

        fase = "swap"
    if fase == "swap":
      if min_idx != i:
            aux = items[i]
            items[i] = items[min_idx]
            items[min_idx] = aux
```

```
                resultado = {"a": i, "b": min_idx, "swap": True, "done": False}

    else:
        resultado = {"a": i, "b": min_idx, "swap": False, "done": False}

    i = i + 1
    j = i + 1
    min_idx = i
    fase = "buscar"
    return resultado
```

- ¿Qué es el algoritmo Selection Sort? ¿Cómo funciona?:

Selection Sort es un algoritmo que busca el número más chico de una lista desordenada, lo encuentra y coloca al principio de todo. Repetidamente, avanza hacía otro número más chico repitiendo el proceso. Como cuando en los colegios elegían al alumno más bajo para ponerlo delante de la fila y armar un menor a mayor.

Dificultades encontradas:

- Separar el proceso en dos partes, buscar e intercambiar.
- Intentar evitar hacer intercambios que no sean necesarios.
- Controlar el mínimo.

- Algoritmo Quick Sort:

```
items = [ ]
n = 0
stacks = [ ]
izqlimite = 0
derlimite = 0
i = 0
j = 0
corte_ref = 0
fase = "nuevo"
def init(vals):
    global items, n, stacks, izqlimite, derlimite, i, j, corte_ref, fase
    items = list(vals)
    n = len(items)
    stacks = [ ]
    if n > 0:
        stacks.appendd((0, n - 1))
izqlimite = 0
    derlimite = 0
    i = 0
    j = 0
    corte_ref = 0
    fase = "nuevo"
def step():
    global items, n, stacks, izqlimite, derlimite, i, j, corte_ref, fase
    if fase == "nuevo" and not stack:
        return {"done": True}
if fase == "nuevo"
        ultimo = len(stacks) - 1
        izqlimite, derlimite = stack[ultimo]
pediente = []
        for k in range(0, ultimo):
            pendiente.append(stacks[k])
        stacks = pendiente
 corte_ref = derlimite
i = izqlimite
 j = izqlimite
```

```
        fase = "empezar"
        return {"a": j, "b": corte_ref, "swap": False, "done": False}
 if fase == "empezar":
        if j < derlimite:
            a = J
            b = corte_ref
            swap = False
if items[j] < items[corte_ref]:
                aux = items[i]
                items[i] = items[j]
                items[j] = aux
            resultado = {"a": i, "b": j, "swap": True, "done": False}
            i += 1
             j += 1
             return resultado
 else:
            aux = items[i]
            items[i] = items[corte_ref]
            items[corte_ref] = aux
             corteNuevo = I
 if corteNuevo - 1 > izqlimite:
                pendiente.append((izqlimite, corteNuevo - 1))
if corteNuevo + 1 < derlimite:
                pendiente.append((corteNuevo + 1, derlimite))

 fase = "nuevo"
return {"a": i, "b": corte_ref, "swap": True, "done": False}
```

- ¿Qué es el algoritmo Quick Sort? ¿Cómo funciona?:

Quick Sort es un algoritmo que se divide, elige un número de "referencia", que yo en el programa lo llame corte de referencia, que hace: los más chicos para un lado, los más grandes para el otro lado. Una vez separado el grupo en 2, el algoritmo hace lo mismo con cada mitad repitiendo el proceso hasta que sean más chicos, queda una escalera porque siempre termina poniendo cada elemento en su lugar.

Dificultades encontradas:

- Entender el concepto de "corte de referencia". Cuesta entender en un principio lo que hace y porqué lo hace, ¿por qué separa el grupo? Hasta entender que usa el último elemento para ubicar y manejar más fácil.
- Dividir el algoritmo en fases como, recorrer y comparar, hacer intercambios y guardar los grupos pendientes.
- Que devuelva los pasos constantemente y para que no se pare la animación agregar un retorno.

- Algoritmo Shell Sort:

```
items = [ ]
n = 0
gapsalto = 0
i = 0
j = None
aux = None
fase = "nuevo"
def init(vals):
    global items, n, gapsalto, i, j, aux, fase
    items = list(vals)
    n = len(items)
    gapsalto = n // 2
    i = gapsalto
    j = None
    aux = None
    fase = "iniciar_i"
def step():
    global items, n, gapsalto, i, j, aux, fase
    if gapsalto == 0:
        return {"done": True}
if fase == "iniciar_i":
        if i >= n:
            gapsalto //= 2
            if gapsalto == 0:
                    return {"done": True}
i = gapsalto
            fase  = "iniciar_i"
            return {"a": 0, "b": 0, "swap": False, "done": False}
```

```
 aux = items[i]
       j = i
       fase = "recorrer"
return {"a": i, "b": i - gapsalto, "swap": False, "done": False}
if fase == "recorrer":
            if j - gapsalto >= 0 and items[j - gapsalto] > aux:
           items[j] = items[j - gapsalto]
           items[j - gapsalto] = aux
           resultado = {"a": j, "b": j - gapsalto, "swap": True, "done": False}
           j -= gapsalto
                    return resultado
           i += 1
                 fase = "iniciar_i"
           return {"a": j, "b": j - gapsalto if j - gapsalto >= 0 else j, "swap": False, "done": False}
```

- ¿Qué es el algoritmo Shell Sort? ¿Cómo funciona?:

Shell Sort es una versión de Insertion Sort pero saltando posiciones. Compara los elementos ordenando una estructura general, luego el salto lo reduce el salto y va haciendo un mejor orden. Cuando llega a 1 salto, hace un Insertion Sort normal pero con la lista ya ordenada.

Dificultades encontradas:

- Entender cómo manejar los saltos, en mi programa gap saltos, para reducirlos.
- Coordinar todas las fases para recorrerlas e iniciarlas.
- Tuvimos muchos errores al comparar los elementos que se separaban mucho. Mostrar visualmente que los elementos se están saltando.

- Algoritmo Merge Sort:

```
items = [ ]
n = 0
pasos = [ ]
pos_paso = 0
def init(vals):
           global items, n, pasos, pos_paso
           items = list(vals)
            n = len(items)
           pasos = [ ]
           pos_paso = 0
```

```
        temporal = list(vals)
        ordenada = list(tmp)
         for x in range(1, len(ordenada)):
               valor = ordenada[x]
              j = x - 1
        while j >= 0 and ordenada[j] > valor:
                    ordenada[j + 1] = ordenada[j]
                   j -= 1
         ordenada[j + 1] = valor
         for i in range(n):
                     valorCorrecto = ordenada[i]
                      j = i
         while j < n and temporal[j] != valorCorrecto:
                               j += 1
         while j > i:
                      pasos.append((j - 1, j))
                     temporal[j - 1], temporal[j] = temporal[j], temporal[j - 1]
                     j -= 1
def step():
 global items, n, pasos, pos_paso
if pos_paso >= len(pasos):
       return {"a": 0, "b": 0, "swap": False, "done": True}
 a, b = pasos[pos_paso]
   pos_paso += 1
 items[a], items[b] = items[b], items[a]
 return {"a": a, "b": b, "swap": True, "done": False}
```

- ¿Qué es el algoritmo Merge Sort? ¿Cómo funciona?:

Merge Sort es un algoritmo que ordena una lista separando en partes más chicas y después volviéndo a juntar de forma ordenada. La idea sería:

1. Dividir en dos mitades.
2. Volver a dividir cada mitad en dos, hasta que cada parte sea muy chica.
3. Cuando ya no se pueda dividir más, empieza la parte donde junta las dos mitades ya ordenada para crear una sola lista.

4. Se repite este proceso hasta que al final la lista queda completamente ordenada. Es como desarmar piezas más pequeñas para acomodarla mejor y volver a armar todo pero de manera mejor ordenada.

Dificultades encontradas:

- Merge Sort con su versión clásica no nos funcionaba porqué trabajaba por dentro con muchas fases donde no había intercambios visibles, se traba ya que al dividir tantos bloques, mezclas en mitades, en varios pasos no se movía nada en pantalla. Esto nos hacía que el visualizador lo interprete como "sin actividad" y quedará pausado. Probamos varias formas de implementar pero siempre se frenaba, probando con auxiliares, punteros, bloques, etc. Lo más complicado fue entender, al final de todo, que no era un error del algoritmo si no que Merge Sort no se llevaba bien con el sistema de pasos del visualizador. Lo solucionamos adaptando mezclar bloques pero usando un plan de intercambios usando una lista ordenada y luego ejecutar esos intercambios 1 x 1. Empezó a funcionar, porqué el visualizador recibe movimientos claros para mostrar, fue la parte más difícil del trabajo pero también de donde más aprendimos.