Trabajo Practico: Algoritmos de Ordenamiento

Universidad Nacional de General Sarmiento

Integrantes:
Alejandro Peralta
Aylen Melgarejo
Lautaro Lang

Materia: Introducción a la programación

Comisión: 8

Profesoras:
Viviana Varela
Jorgelina Rial

Introducción

El objetivo de este trabajo práctico es comprender distintas formas en las que se puede organizar una lista de datos. Para eso se implementaron tres métodos clásicos de ordenamiento: Bubble Sort, Inserción Sort y Selection Sort. Cada uno sigue una lógica diferente para comparar y mover los elementos, lo que permite observar cómo varía su comportamiento y sus resultados. Además, se integraron estos algoritmos con un visualizador paso a paso, que muestra cada comparación y posible intercambio. De esta manera, el enfoque del trabajo no es solamente obtener el listado ordenado, sino entender de forma clara y visual cómo trabaja cada técnica.

1. Bubble Sort

Código

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

    if i < n - 1:
        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 - 1 - i:
            j = 0
            i += 1

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

    return {"done": True}
```

Explicación, dificultades y decisiones

El código implementa el algoritmo Bubble Sort de manera paso a paso mediante la función step(). A diferencia de la versión tradicional del algoritmo, que utiliza bucles completos para procesar todas las comparaciones, esta implementación está pensada para que cada llamada a la función realice únicamente una comparación entre dos elementos de la lista y, si corresponde, un intercambio. Esto permite visualizar y analizar el proceso de ordenamiento en tiempo real, útil para fines educativos o para depuración.

El funcionamiento se basa en dos variables principales: i, que indica la pasada actual del algoritmo, y j, que indica la posición del par de elementos que se está comparando. Durante cada llamada a step(), los elementos en las posiciones j y j+1 se comparan, y si están en el orden incorrecto, se intercambian. Luego, j avanza a la siguiente posición. Cuando j alcanza el límite de comparaciones permitido para la pasada actual, se reinicia a 0 y se incrementa i, avanzando así a la siguiente etapa del algoritmo. El proceso continúa hasta que i alcanza n-1, momento en el que ya no quedan comparaciones por hacer y la función devuelve que el proceso ha terminado.

Las dificultades que tuve al hacerlo fue que no supe acomodar el código, algunas cosas las ponía dentro de un if como j+=1 o sacaba un if fuera del if principal sin darme cuenta, básicamente fue eso, el código ya lo había hecho, pero de la forma tradicional así que tenía una idea de cómo hacerlo.

2. Insertion Sort

Código

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

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

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

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

Explicación, dificultades y decisiones

En este código se implementa el algoritmo Insertion Sort usando la función step() con el objetivo de que se haga paso a paso. En esta implementación con step se divide el proceso en pasos para tener una visualización del funcionamiento del ordenamiento y como las barras se van moviendo o intercambiando en el lugar correcto.

El funcionamiento depende de tres variables y una lista, las principales son j y i, donde i toma la posición del elemento en la lista que se trata de ordenar, mientras que j funciona como un puntero que se mueve

hacia la izquierda y compara el elemento actual y el que esta antes de el (en la izquierda) si se hace un swap o no, también se usa una variable adicional n, la que representa la longitud de la lista que permite saber cuándo se recorrió toda la lista.

Para cada llamado que se hace en el step(), se cumple este orden de pasos. En el primer paso, i toma el valor de 1 para comenzar en el segundo elemento de la lista (ya que el primero se considera ordenado) y j toma el mismo valor de i, con el objetivo de comparar el elemento de la izquierda. En el segundo paso, si se cumple que j sea mayor que 0 y que el elemento de la izquierda sea mayor al de la derecha se hace un swap entro los elementos, después de ese intercambio, j se le resta un valor para que retroceda de posición para comparar con los elementos anteriores, por lo que se va a ir comparando y hacer swap con el elemento actual y los de la izquierda hasta que se deje de cumplir que el elemento actual sea mayor al que se compara o siquiera hay elemento restante para comparar devolviendo que se cumplió el swap. En el tercer paso, a partir del cumplimiento del swap, se le suma un valor a i para que en la segunda vuelta se compare desde el siguiente elemento de la lista. En el último paso, después de repetir el ciclo, se frena hasta que i (el elemento que se compara) alcance n (el valor de la longitud de la lista), cumpliendo con el ordenamiento.

Una dificultad que tuve fue que no reconocía como modificar las variables porque no tenía un buen conocimiento de cómo funcionaba el algoritmo en sí, eso me complicaba a la hora de hacer los returns, ya que no sabía que devolver, por lo que investigué para entender como era el funcionamiento y como de verdad tenía que comparar cada variable. Otras complicaciones fueron errores de notación y de ordenamiento, como poner un else: cuando no iba, rompiendo mi codigo, eso lo pude resolver después de encontrarlos o hablando con mis compañeros de grupo. También tuve dificultad al principio de que no sabía que representaba cada variable en el codigo dado, cosa que pude entender prestando atención a lo que estaba anotado, como n = len(items), que representaba la longitud de la lista, o a las consignas que decían “comenzar desplazamiento para item[i]”, comprendí que i era la encargada a asignar los elementos de la lista, y j las que recorra la lista a partir de i.

3. Selection Sort

Código

```
def step():
global i, j, min_idx, fase,n

  if i >= n - 1:

     return {"done": True}

  if fase == "buscar":

       if items[j] < items[min_idx]:

          min_idx = j

       salida={"a":min_idx, "b": j,"swap": False, "done": False}
```

```
        j+=1
        if j>=n:
            fase = "swap"
        return salida
    elif fase == "swap":
        swapped = min_idx != i
        if swapped:
            items[i], items[min_idx] = items[min_idx], items[i]
            salida= {"a”: i, "b": min_idx,"swap": True, "done": False}
        else:
            salida= {"a”: i, "b": min_idx,"swap": False, "done": False}
        i += 1
        j = i + 1
        min_idx = i
        fase = "buscar"
        if i>=n-1:
            return {"done":True}
        return salida
```

Explicación, dificultades y decisiones

Este código implementa el algoritmo Selection Sort de manera paso a paso, a través de la función step (). Esta implementación se enfoca en mostrar el proceso de ordenamiento en tiempo real, en lugar de realizar un ordenamiento completo en una sola llamada. La función step() realiza una única operación en cada llamada: busca el elemento más pequeño en la parte no ordenada de la lista y lo intercambia con el elemento que está al principio de esa parte. El funcionamiento se basa en las variables i, j, min_idx, y fase. i indica la posición del elemento que ya está ordenado en la parte izquierda de la lista. j es un cursor que recorre la parte no ordenada para encontrar el elemento más pequeño. min_idx almacena el índice del elemento más pequeño encontrado en la parte no ordenada. fase controla si estamos en la fase de "buscar" (encontrar el mínimo) o "swap" (intercambiar el mínimo con el elemento en la posición i).

Durante cada llamada a step(), si estamos en la fase "buscar", se compara el elemento en la posición j con el elemento actualmente considerado como el mínimo (items[min_idx]). Si el elemento en j es menor, min_idx se actualiza. Luego, j avanza a la siguiente posición. Cuando j llega al final de la parte no ordenada, la fase cambia a "swap". En la fase "swap", si min_idx es diferente de i, se realiza el intercambio entre los elementos en esas posiciones. Después del intercambio (o si no es necesario), i se incrementa, j se reinicia a i+1, min_idx se establece a i, y la fase vuelve a "buscar", comenzando un nuevo ciclo. El proceso continúa hasta que i alcanza n-1, indicando que todos los elementos están ordenados.

Algunas dificultades que tuve fue relacionado momento de acomodar la devolución que luego de la comparación actualiza min_idx para luego avanzar en j, para ello el resultado “{"a":i, "b": min_idx,"swap": False, "done": False} “ lo devuelvo mediante una variable por ejemplo “salida”(return salida). Lo mismo con el caso de swap el resultado de si es verdadero o falso se guarda dentro de la variable anterior y luego al preparar la siguiente iteración, si i >=n-1 de pasadas termina, sino muestra los intercambios de verdadero o falso hasta que termina.