Universidad Nacional de General Sarmiento

Introducción a la programación-COM 07

2do Cuatrimestre del 2025

Trabajo Práctico: Algoritmos de Ordenamiento - Informe

Alumnos:

- Nahuel Cordero
- Demian Valentín Enríquez Fernández
- Axel Alvares
- Santiago Velázquez

Introducción:

Este trabajo práctico tiene como objetivo implementar los siguientes algoritmos de ordenamiento: Bubble Sort, “Selection Sort”, “Insertion Sort”, “Shell Sort”. Es importante estudiar este tipo de algoritmos ya que ayuda a entender bucles anidados y a conocer diferentes formas de organizar información y entender el rendimiento de cada uno y cuando es conveniente utilizarlos.

Cada uno de los algoritmos mencionados se implementó en un código base previamente brindado por los profesores dentro de la carpeta algoritmos. Nosotros nos encargamos de aprender cómo funcionan y desarrollar el código correspondiente.

La implementación se realizó bajo un modelo de ejecución paso a paso, simulando el avance de un algoritmo tradicional mediante las funciones `init()` y `step()`. `init()` se encarga de brindar los valores iniciales y `step()` se encarga de dar los pasos y hacer las comparaciones de valores.

Código:

Algoritmo Bubble Sort:

```python
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:
        return {"done": True}

    if j < n - 1 - i:
        a = j
        b = j + 1
        swap = False

        if items[a] > items[b]:
            temp = items[a]
            items[a] = items[b]
            items[b] = temp
            swap = True

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

    else:

        j = 0
        i += 1

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

```

En el bubble sort tradicional se utilizan dos bucles anidados para ordenar la lista. La principal dificultad en esta implementación fue reemplazar esta estructura de bucles anidados por una lógica de estados que avanza con cada llamada a la función step.

Para la ejecución, se definieron variables de entornos globales:

- **N** (Longitud): inicializado en init()**,** representa el tamaño total de la lista.
- **I** (El contador de pasadas): Actúa como un acumulador de la cantidad de elementos ordenados y fijos al final de la lista, solo se incrementa en el else (al final de una pasada). Simula el bucle externo.
- **J** (Comparador de índice): Se utiliza como índice de la posición actual y avanza (j +=1) en cada llamada a step(). Simula el bucle interno.

Lógica de step():

1. Condición de finalización: la verificación if i >= n nos garantiza que el algoritmo finalice cuando el contador de pasadas (i) haya llegado al limite necesario para ordenar la lista.
2. Ejecución de pasos (if j < n- 1 -i): Esta es la condición que se encarga de simular el bucle interno. La expresión n – 1- i define el limite de la parte no ordenada de la lista. Siempre que j esté debajo de este límite, se realiza la comparación y si items[j] > items [j +1] se efectúa el swap o cambio.
3. Pasada (else): Cuando j alcanza el límite, el bloque else se encarga de simular el fin del bucle interno. Esto hace lo siguiente:
    - Reinicia el índice de comparación (j = 0)
    - Se incrementa el contador de pasadas (i += 1)

Selection Sort

Autor: Santiago Velázquez

El algoritmo Selection Sort posee originalmente una estructura de bucles anidados tomando el primer valor de una lista de elementos como el menor y con un bucle interno recorre la lista buscando el menor o el mayor de los elementos (depende la forma en la que se quiera ordenar la lista) para guardar su indice y así realizar un cambio de posiciones cuando sea necesario.

Esta estructura tal como lo pide la consigna fue cambiada por una lógica que realiza una comparación un paso a la vez con una función llamada step.

Variables globales establecidas

- **n (Longitud):** Representa la longitud total de la lista.
- **i (Cabeza de la parte no ordenada):** Esta variable es la que se toma como cabeza para comparar con otros valores (siempre se toma como el valor más pequeño para luego compararlo), aumenta en i+1 por cada llamada a step().
- **j (Cursor que recorre la lista):** Es la variable que recorre la lista en busca de un valor más pequeño que i.
- **min_idx:** Es el índice del valor mínimo durante la llamada a step().

- **fase:** Esta variable divide la función en dos partes, la fase **"buscar"** en la cual se busca el índice del valor más pequeño y la fase **"swap",** la cual se encarga de realizar cambio de posiciones de ser necesario

Lógica de step():

- **Condición de finalización:** la verificación if i >= n es fundamental ya que si esto se cumple significa que i (cabeza de la parte no ordenada) es mayor a n (longitud de la lista) significa que ya terminó de recorrerla en su totalidad, por lo tanto, ya está totalmente ordenada.
- **Fase "buscar":** la función verifica que la fase se encuentre en "buscar" para comparar **j** (cursor que recorre la lista) con **min_idx** (índice del valor más pequeño actual), si la verificación items[j] < items[min_idx] se cumple, **min_idx** toma como nuevo valor **j**.
- **Fase "swap":** Durante esta fase, el algoritmo realiza una verificación esencial que es if min_idx != i:, esta condición es necesaria para saber si es hay que realizar un cambio de posiciones, si esta condición no se cumple significa que i es el índice con el valor más pequeño por lo tanto no hay que realizar un swap. Por otro lado, si esta condición si se cumple se realiza el cambio de posiciones.
- **Reinicio y avance de valores:** Al terminar las dos fases, las variables cambian sus valores para prepararse a la próxima llamada de la función.
    - La variable **i** aumenta en uno para cambiar de índice al valor siguiente.
    - La variable **j** cambia de valor a **i+1** para empezar a recorrer la lista un índice después de **i.**
    - La variable **min_idx** toma el valor de **i** para utilizarlo como el índice que guarda el valor más pequeño.
    - La variable **fase** reinicia su valor a **"buscar"** para alistarse a la próxima llamada de la función.

Dificultades durante el proceso de desarrollo

Durante el proceso de desarrollo de este algoritmo se me presentaron distintos inconvenientes que frenaron un poco el avance. Uno de los inconvenientes fue el cómo remplazar la lógica de bucles anidados por una función que compare los valores un paso a la vez y se divida en distintas fases, que además de esto retornaba diccionarios, los cuales tuve que hacer una pequeña investigación para entender bien su funcionamiento. Otro inconveniente apareció al momento de hacer el push de mi código actualizado hacia el repositorio remoto ya que no terminaba de entender si tenía que armar un fork del repositorio o desde que entorno había que realizar ese push.

Los inconvenientes nombrados pudieron resolverse mediante pequeñas investigaciones que no presentaron mayor problema pero fueron necesarias para entender el funcionamiento de las distintas herramientas que utilizamos.

Algoritmo Insertion Sort:

Autor= Demian Valentin Enríquez Fernández

El algoritmo **insertion sort** construye progresivamente una lista que queda organizada, insertando cada elemento en la posición correcta, comparando cada nuevo elemento con los anteriores y desplazándose a la izquierda hasta encontrar su posición correcta dentro de sección ya ordenada, para este algoritmo utilizamos un modelo de ejecución paso a paso haciendo uso de la función **step()**.

Se utilizaron las siguientes variables globales*:*

- **n** = Se obtiene al inicio y determina cuando finaliza el proceso.
- **i** = Indica el valor a ser posicionado correctamente dentro de la sección ordenada actual.

- **j** = Mueve el elemento actual hacia su posición final, comparándolo con los valores anteriores, cuando **j is None** significa que comienza una nueva **inserción**.

Comportamiento de Step (En orden):

1. **Condición para concluir:** Verifica si **i** llegó al final de la lista, en caso de ser así, el algoritmo ya dejó cada elemento en su posición correcta y concluye.
2. **Inicio de inserción:** Cuando **j == None (j is None)** comienza la inserción del valor **items[i]**, no se realiza ningún intercambio, solo se prepara los elementos que se van a comparar.
3. **Desplazamiento a su posición correcta:** Mientras el elemento actual sea menor al elemento a su izquierda, se realiza un intercambio con el elemento anterior, realizando un único intercambio por cada step permitiendo así visualizar de forma gradual.
4. **Final del desplazamiento:** Si el elemento ya llegó a su posición definitiva, se avanza en **i** *(i+=1)* para comenzar la inserción del valor próximo y **j** vuelve a **None**, lo que indica que la siguiente llamada dará inicio a una nueva inserción.

Dificultades presentadas:

En primera instancia comencé a programar el código en **PyScripter**, debido a unos errores al introducir los bool el código no funcionaba y tampoco me permitía reiniciarlo en el visualizador proporcionado, lo cual me dificultó comprender el comportamiento de **step()**, gracias una revisión dada en clase de los profesores a mi código y la recomendación de implementar el uso de **Visual Studio Code**.

Continúe el desarrollo en **VSC** con el cual conseguí solucionarlo y corregir algunos pequeños errores que pase por alto, una vez dando respuesta el visualizador pude comprender mejor cómo debía ser el código, tomando como ejemplo una presentación gráfica del *insertion sort* que busque en línea, a partir de esa etapa no tuve más complicaciones con el código y este funciono sin complicaciones.

Luego tuve un inconveniente con el uso de **git bash**, siendo más específicos con el comando **push** por un conflicto generado desde mi entorno local *(que anteriormente me había funcionado sin más)* debido a que otro compañero había actualizado el fork con su código, por otro lado deje una operación inconclusa en git bash lo que me impedía continuar. Ambos problemas fueron solucionados investigando los comandos necesarios en la web y sincronizando correctamente el repositorio.

Una vez abordados estos inconvenientes me permitió comprender mis errores, aclaró mi visión de la lógica del algoritmo y pude conseguir un resultado funcional.

Algoritmo Shell Sort:

El algoritmo Shell Sort es una mejora del Insertion Sort que introduce un concepto fundamental: **ordenar elementos distantes entre sí antes de ordenar elementos contiguos(un elemento al lado de otro)**. Para lograrlo utiliza saltos o *gaps*, que comienzan siendo grandes y se van reduciendo progresivamente hasta llegar a 1, donde finalmente se comporta como un Insertion Sort tradicional.

La dificultad principal en esta implementación fue **adaptar la lógica original basada en bucles anidados**(dos bucles: uno para los saltos y otro para cada "sublista") a un modelo **paso a paso**, donde cada llamada a la función step() debe realizar exactamente **una operación**.
Esto implicó transformar el algoritmo en una **máquina de estados**, controlando cuidadosamente en qué parte del proceso se encuentra.

Variable establecidas:

Items = La lista de números que se está ordenando.

n = Cantidad de elementos de items.

gap = El “salto” del Shell Sort. Arranca en n//2 y se va reduciendo.

i = el elemento seleccionado para acomodar.

j = el lugar donde estoy comparando y moviendo otros elemento para abrir espacio.

temp = el elemento que quiero acomodar y todavía no sé dónde va.

Stage = La etapa actual del algoritmo. Puede ser: "gap", "i_loop", "shift", "insert".

Done = Indica si el algoritmo ya terminó.

Lógica de step():

Condición de finalización:
La primera verificación importante es if done: o cuando el algoritmo llega a un gap == 0. Si esto sucede, significa que ya no quedan saltos por utilizar y, por lo tanto, la lista está completamente ordenada. Este control es fundamental para que el algoritmo sepa en qué momento debe detenerse y evitar seguir ejecutando pasos innecesarios.

- **Fase “gap”:**

En esta fase, el algoritmo trabaja con el valor de *gap*, que representa la distancia entre los elementos que se comparan. Si gap es mayor que 0, entonces se prepara la siguiente etapa ubicando a i en el primer índice válido para trabajar (i = gap). De esta manera, el algoritmo simula el bucle externo tradicional del Shell Sort, avanzando con distintos tamaños de salto que se irán reduciendo progresivamente. Una vez seteado el índice inicial, la fase cambia a "i_loop".

Fase “i_loop”:
Dentro de esta fase se decide si el índice i ya superó el tamaño de la lista. Si la condición i >= n se cumple, significa que se completó una pasada usando el salto actual, por lo que el algoritmo reduce el tamaño del salto a la mitad (gap //= 2) y vuelve nuevamente a la fase "gap".
Si todavía quedan elementos por procesar, el algoritmo guarda el valor items[i] en la variable temporal temp, asigna j = i y avanza hacia la fase "shift", donde se desplazará el elemento hasta encontrar su posición correcta.

Fase “shift”:
Esta es una fase clave del Shell Sort. Aquí el algoritmo realiza corrimientos hacia atrás utilizando la distancia definida por gap.
La condición if j >= gap and items[j - gap] > temp: determina si el elemento que está *gap* posiciones atrás es mayor que el que queremos insertar (temp).
Si la condición se cumple, se mueve ese elemento hacia adelante (items[j] = items[j - gap]) y se retrocede j para seguir comparando. Cada desplazamiento se realiza paso a paso, respetando la idea del visualizador.
Si la condición no se cumple, significa que el lugar indicado para temp ya fue encontrado y el algoritmo pasa a la fase "insert".

● **Fase “insert”:**
En esta fase el algoritmo coloca finalmente el valor guardado en temp dentro de su posición correcta (items[j] = temp).
Después de realizar esta inserción, se avanza en el índice principal (i += 1) para continuar con el siguiente elemento de la lista usando el mismo salto. Finalmente, la fase vuelve a "i_loop", preparando la próxima llamada de la función.

Dificultades presentadas:

Una de las principales dificultades fue que Shell Sort tenía que funcionar paso a paso, y no como un algoritmo normal que corre todo de una vez. Esto me obligó a dividirlo en **fases** (calcular el salto, comparar, mover, etc.) para que no se mezclen operaciones y no haya errores. Otra complicación importante fue que, para que el algoritmo pudiera mostrarse correctamente en la UI, hubo que modificar partes del código HTML del proyecto.

También fue fácil confundirse con los índices i y j, porque si se actualizaban en el momento equivocado, el algoritmo se rompía o quedaba atrapado en un bucle. Otra dificultad fue guardar el valor temporal (temp) sin perderlo mientras movíamos otros elementos hacia atrás. En general, lo más complicado fue organizar bien el estado del algoritmo para que cada llamada a step() haga una sola acción y se pueda ver animado sin que se salte pasos importantes.