Trabajo Práctico Integrador Programación

Búsqueda y Ordenamiento en programación

Alumnos: Giuliano Scaglioni, Mariano Rossi.

Comisión 21

Tecnicatura Universitaria en Programación - Universidad Tecnológica Nacional.

Programación 1

Docente Titular

Prof. Nicolás Quirós

Docente Tutor

Neyén Bianchi

06 de junio de 2025

Tabla de contenido

Introducción

Un algoritmo de ordenamiento es un algoritmo que organiza los elementos de una lista en un cierto orden. Los órdenes más usados son el numérico y el alfabético, y generalmente en orden creciente o ascendente. Es importante que el ordenamiento sea eficiente para optimizar el uso de otros algoritmos (como, por ejemplo, el de búsqueda) que requieren que los datos de entrada estén ordenados; o para producir resultados ordenados para las personas. Formalmente, la salida debe satisfacer dos condiciones:

1. La salida está en orden no decreciente (cada elemento no es menor que los elementos previos de acuerdo al orden total deseado).

2. La salida es una permutación (reordenamiento) de la entrada.

Este es un problema que, desde los principios de la computación, ha atraído una gran cantidad de interés, tal vez debido a la complejidad de resolverlo eficientemente aunque sea simple y familiar. Si bien muchos consideran que el problema de ordenamiento está resuelto ,el método de ordenamiento por mezcla fue desarrollado en 1945, el de la burbuja fue analizado en 1956, todavía se crean algoritmos útiles, como Timsort (2002) y el método de la Biblioteca (publicado por primera vez en 2006).

Marco teórico

El presente marco teórico tiene como objetivo situar los procesos de búsqueda y ordenamiento de datos en la literatura sobre algoritmos de gestión de información, defendiendo conceptos claves como los métodos de ordenamientos Bubble Sort como Quick Sort. Además, realizamos estudios previos que fundamentan la eficiencia y aplicabilidad en los diferentes entornos.

Los algoritmos de búsqueda Lineal y binaria resuelven problemas donde se busca identificar un valor especifico dentro de una estructura de datos. Ejemplo (búsqueda de X DNI en una agenda, búsqueda de una persona, entre otros), tienen diferentes enfoques y eficiencias.

Búsqueda lineal: Se realiza la misma buscando elemento por elemento, hasta encontrar el valor deseado. No requiere que los datos estén ordenados, es ideal para listas cortas o no ordenadas. Su complejidad de los algoritmos es de O(n), en listas grandes demoran más, ya que son directamente proporcionales con la cantidad de elementos que contienen.

Ejemplo de código Python:

```
for i in range(len(lista_dni)): #recorro cada posición de la lista de dni
        if lista_dni[i] == dni_buscado: # si el valor de la posición es igual al buscado retorna la posición
            return i  # Retorna la posición si lo encuentra
    return False  # Si no lo encuentra, retorna False
```

Búsqueda Binaria:

Utiliza la estrategia de división en mitades (divide y vencerás). Cada paso lo realiza comparando el elemento medio con el valor buscado, es ideal para listas grandes, pero requiere que los datos estén previamente ordenados. Su complejidad es de O (log n).

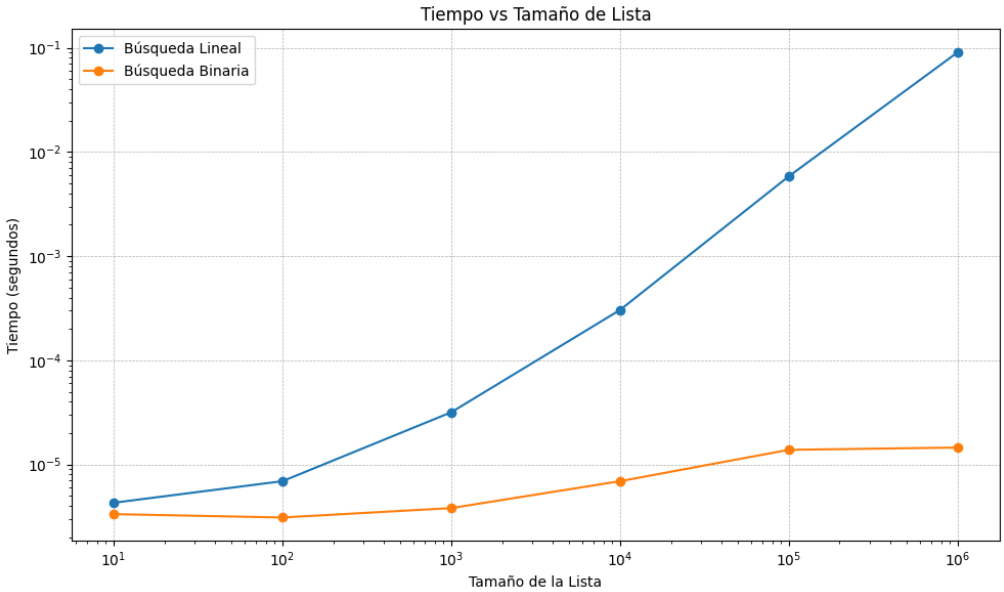
Ejemplo de código Python

```
inicio = 0  # Inicializamos la búsqueda desde el primer índice (inicio de la lista)
fin = len(lista_dni) - 1  # Cargamos en la variable fin la cantidad de elementos de la lista.(final de la lista)

while inicio <= fin:  # Mientras el rango de búsqueda sea válido es atrapado por el bucle(inicio no haya pasado a fin)
    medio = (inicio + fin) // 2  # obtenemos la posición del medio de la lista
    print(f"Comparando con índice {medio}: {lista_dni[medio]}")  # Mostramos el valor que estamos comparando

    if lista_dni[medio] == dni_buscado:  # abrimos un condicional para comparar el elemnto con el valor buscado
        return medio  # Devolvemos el índice donde fue encontrado
    elif lista_dni[medio] < dni_buscado:  # Si el valor en medio es menor que el que buscamos
        inicio = medio + 1  # Acotamos la búsqueda al tramo derecho (valores mayores)
    else:
        fin = medio - 1  # Si el valor en medio es mayor, buscamos en el tramo izquierdo

return False  # Si se termina el bucle sin encontrar el DNI, devolvemos False indicando que no está
```



En este grafico se muestra como a mayor cantidad de datos las búsquedas lineales demoran más tiempo versus las búsquedas binarias. Grafico de la documentación oficial de la UTN

Ordenamientos Bubble Sort y Cocktail Sort. (nos focalizamos en estos métodos)

Los algoritmos de ordenamiento de datos nos ayudan a realizar búsquedas más rápidas, realizar análisis y operaciones más eficientes, al ordenarlos bajo un criterio (numéricos, alfabetos o algún dato especifico).

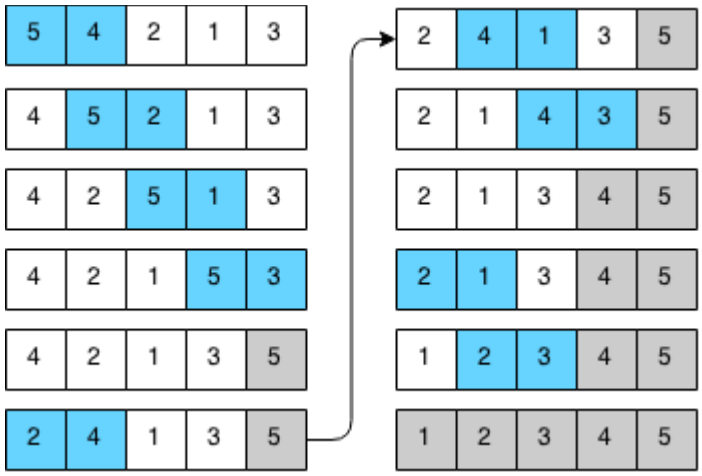
Existen varios métodos de ordenamientos (Burbuja, selección, inserción, ordenamiento rápido).

Ordenamiento Bubble sort:

Funciona recorriendo la lista repetidamente comparando con el elemento siguiente, si este elemento es mayor o menor, lo intercambiará. Logrando así el ordenamiento deseado por nosotros. (dependiendo la condición que hayamos usado para la búsqueda).

Este ordenamiento es fácil de entender e implementar, la desventaja es que es muy lento para listas grandes e ineficiente si la lista esta casi ordenada y su complejidad es siempre $O(n^2)$.

Imagen explicativa del ordenamiento Bubble Sort en cada pasada.



Cocktail Sort (Ordenamiento Bidireccional o Shaker Sort)

Este ordenamiento es la mejora del Buble Sort, ya que recorre la lista en ambas direcciones (izquierda a derecha y luego de derecha a izquierda). De esta forma evita el problema donde los elementos pequeños tardan muchas pasadas en llegar al inicio de la lista, cosa que ocurre en el ordenamiento Bubble Sort. El mismo funciona mejor con listas casi ordenadas y es más complejo de implementar vs el anterior. Su complejidad es $O(n^2)$ pero puede ser más rápido en algunos casos.

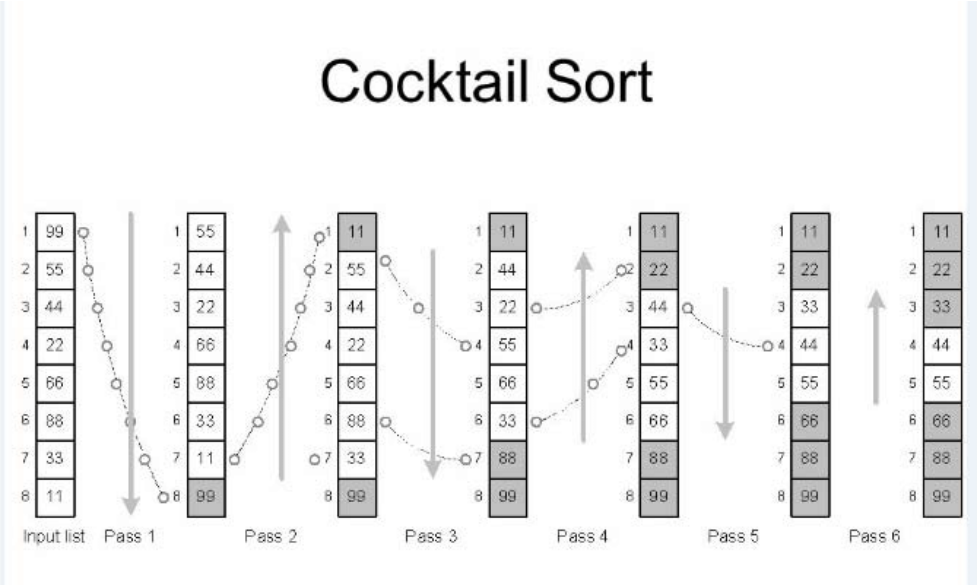


Imagen de: https://www.researchgate.net/figure/cocktail-sort_fig1_314753240

Caso practico

En este proyecto, se presenta un caso práctico en el que necesitamos ordenar listas de números, como los números de DNI, para hacer más fácil su búsqueda o clasificación. Para lograr esto, implementamos dos algoritmos clásicos de ordenamiento: Bubble Sort (en orden descendente) y Cocktail Sort (en orden ascendente).

El usuario tiene la opción de seleccionar listas de diferentes tamaños, desde pequeños arrays hasta listas bastante grandes con 1000 elementos, o incluso puede crear su propio conjunto personalizado. Después se aplica el algoritmo que elija y se mide el tiempo que tarda en ejecutarse, lo que nos permite comparar cuál es más eficiente.

Este enfoque nos da la oportunidad de observar cómo se comportan los algoritmos bajo diferentes condiciones, como el tamaño del array o el orden inicial de los datos, lo cual es fundamental para evaluar la eficiencia en situaciones reales de procesamiento de datos.También hicimos uso de un try y except, si bien no es algo que se llegó a ver en esta cursada, ambos tenemos conocimientos en el uso de manejo de errores.

Función Cocktail Sort en nuestro proyecto

cocktail.py

cocktail.py > ord_cocktail

Giuliano Scaglioni, yesterday | 1 author (Giuliano Scaglioni)

```python
def imprimir_array(vector):
    print("[", end="")
    for elemento in vector:
        print(f" {elemento} ", end="")
    print(" ]")

def ord_cocktail(vector):
    """
    Algoritmo Cocktail Sort (Shaker Sort)
    Ordena el array de forma ascendente
    """
    n = len(vector)
    izq = 0        Giuliano Scaglioni, last week • Metodos de ordenamiento
    der = n - 2
    cambios = True

    while izq < der and cambios:
        cambios = False

        # Pasada de izquierda a derecha
        for i in range(izq, der + 1):
            if vector[i] > vector[i + 1]:
                # Intercambio
                vector[i], vector[i + 1] = vector[i + 1], vector[i]
                cambios = True

        izq += 1

        # Pasada de derecha a izquierda
        for i in range(der, izq - 1, -1):
            if vector[i] < vector[i - 1]:
                # Intercambio
                vector[i], vector[i - 1] = vector[i - 1], vector[i]
                cambios = True

        der -= 1
    print(f"Finalizo el ordenamiento luego de {izq} pasadas")
    print()  # Nueva línea al final
```

Función del Bubble Sort en nuestro proyecto

bubblesort.py > ord_burbuja

You, 19 hours ago | 2 authors (You and one other)

```python
# Función para mostrar el array con un formato ordenado.
def imprimir_array(vector):
    print("[", end="")  # Imprime el corchete inicial sin salto de línea.
    for elemento in vector:
        print(f" {elemento} ", end="")  # Muestra cada elemento con espacios, sin salto.
    print(" ]")  # Imprime el corchete de cierre con salto de línea.

    """
    Algoritmo Bubble Sort u Ordenamiento Burbuja en nuestro caso en forma descendente.
    """
def ord_burbuja(vector):

    n = len(vector)  # Obtiene la cantidad total de elementos del array.
    i = 0  # Contador de pasadas.
    cambio = True  # Bandera que nos dice si hubo algún intercambio.

    while i <= n - 1 and cambio:  # Mientras no hayamos terminado todas las pasadas y todavía haya cambios.
        cambio = False  # Por ahora asumimos que no habrá cambios en esta pasada.        You, 19 hours ago • Uncommitted chang
        for j in range(n - i - 1):# Recorremos el array desde el inicio hasta el último elemento no ordenado.
            # Si el número actual es menor que el siguiente, los intercambiamos (para orden descendente).
            if vector[j] < vector[j + 1]:
                vector[j], vector[j + 1] = vector[j + 1], vector[j]  # Intercambio de posiciones.
                cambio = True  # Como hubo un cambio la lista no está ordenada todavía.

        i += 1  # Aumentamos el contador de pasadas.

        if len(vector) <= 20: # Si el array no es muy grande, mostramos los primeros 15 elementos de cómo va quedando.
            print(f"Pasada {i}: primeros 15 elementos: ", vector[:15])

    print(f"Finalizo el ordenamiento luego de {i} pasadas") # Al finalizar, mostramos cuántas pasadas se hicieron.

```

Metodología Utilizada

Para el desarrollo de este trabajo practico, seguimos los pasos que nos describieron en la plantilla de requisitos, los mismos no permitieron implementar y comparar dos algoritmos de ordenamientos: Bubble Sort y Cocktail Sort. A continuación, detallamos las etapas llevadas a cabo:

- Investigación previa:

  Realizamos una revisión bibliográfica de los principales algoritmos de ordenamientos y búsquedas. Entre las fuentes consultadas se encuentran:

  - The Algorithm Design Manual – Skiena
  - Algorithms in a Nutshell – Heineman, Pollice y Selkow
  - Essential Algorithms – Rod Stephens
  - he Art of Computer Programming – Donald Knuth

  También consultamos documentación official y artículos del sitio académico de la UTN:

  https://tup.sied.utn.edu.ar/course/view.php?id=12&section=62

- Etapas de diseño y prueba del código:

  Utilizamos un diseño modular, separando el proyecto en 4 archivos .py:

  bubblesort.py: Contiene la lógica del método Bubble Sort en orden descendente.

  cocktail.py: Implementa Cocktail Sort (Shaker Sort) en orden ascendente.

  pruebas.py: Genera arrays de distintos tamaños para las pruebas.

  index.py: Menú principal e interacción con el usuario.

- Implementación progresiva:

  Primero codificamos cada algoritmo y realizamos pruebas con arrays simples. Luego, adaptamos los métodos para trabajar con listas grandes (hasta 1000 elementos) y permitimos que el usuario cree su propio conjunto personalizado (Ejemplo una lista de DNIs).

- Pruebas de rendimiento:

  Añadimos funciones para medir el tiempo de ejecución en cada algoritmo con la librería time, y comparamos los resultados en distintas condiciones (tamaño y tipo de datos).

- Herramientas y recursos utilizados:
    - Lenguaje: Python 3.10.
    - IDE: Visual Studio Code.
    - Librerías: random y time para generar datos aleatorios y medir tiempos.
    - Control de versiones: Git (uso local para control de cambios).
    - Formato de trabajo: Código modularizado y comentado para facilitar el entendimiento y la colaboración.

Resultados obtenidos:

Casos de prueba realizados:

- Se probaron los algoritmos con los siguientes conjuntos de datos:
- Array pequeño: lista fija de 7 elementos.
- Array mediano: 20 números aleatorios entre 0 y 1000.
- Array grande: 100 elementos aleatorios.
- Array muy grande: 1000 elementos aleatorios.
- Array personalizado: el usuario puede definir el tamaño y el rango, útil para simular DNIs u otras aplicaciones reales.

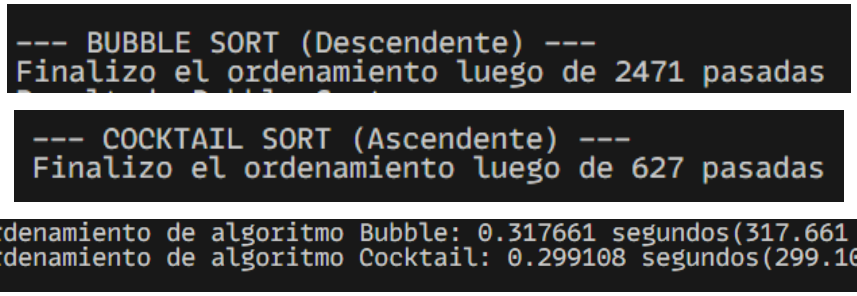
Errores corregidos:

Durante la fase de pruebas se identificaron y solucionaron los siguientes errores:

- Error al mostrar arrays grandes: se optimizó la impresión limitando la visualización a los primeros 15 elementos para evitar saturación en consola.
- Validación de entrada de datos: se añadieron controles para evitar fallos si el usuario ingresaba valores no válidos al crear arrays personalizados.
- Detección de bucles innecesarios en el algoritmo Bubble Sort: se utilizó una bandera cambio para detener el proceso cuando el array ya está ordenado.

Evaluación de rendimientos:

Se realiza la comparación con un array muy grande de 2500 elementos.



En este ejemplo, aunque el Cocktail sort utilizo 627 pasadas demoro 0.299 mseg versus los 0.317 mseg que utilizo el Bubble sort en 2471 pasadas. Este es un claro ejemplo de eficiencia del ordenamiento cocktail sort.

Conclusión

A través de este proyecto, logramos entender a fondo el funcionamiento de los algoritmos de ordenamiento, abarcando tanto su razonamiento como su implementación real en el código. Aprendimos a desarrollar y contrastar dos métodos tradicionales (Bubble Sort y Cocktail Sort), lo que nos ayudó a consolidar conceptos fundamentales, la eficiencia en diferentes situaciones y el uso de estructuras iterativas.

También llegamos a la conclusión de que la selección del algoritmo adecuado no se basa únicamente en su lógica interna, sino también en el tipo de datos, el tamaño del conjunto de datos y las condiciones bajo las cuales se ejecuta. Esto es importante en campos como las bases de datos, el análisis de datos y cualquier sistema donde el orden afecte el desempeño.

Entre las posibles mejoras, evaluamos la posibilidad de incorporar más algoritmos para ampliar la comparación (como Quick Sort o Merge Sort), emplear gráficos para visualizar los resultados y mejorar la interfaz del programa, haciéndola más accesible para usuarios sin conocimientos técnicos.

Nos resultó un buen proyecto para terminar de entender la teoría y la práctica.

Referencias

- *Skiena, S. (2008) The Algorithm Design Manual. 2nd Edition, Springer-Verlag London. pp 121-133*
- *Heineman, G., Pollice, G. y Selkow, S. (2008) Algorithms in a Nutshell , A Desktop Quick Reference.*
- *O'Reilly Media, Inc. Capítulos 4 y 5*
- *Stephens, Rod. Essential Algorithms. A Practical Approach to computer Algorithms. Wiley*
- *Essentials. Cap 6.*
- *Knuth, Donald. (1998) The Art of Computer Programming, Volume 3: Sorting and Searching, 2nd*
- *Edition, Indianapolis, IN:Addison-Wesley Professional(Pearson Education/InformIT)*
- *https://www.researchgate.net/figure/cocktail-sort_fig1_314753240*
- *https://tup.sied.utn.edu.ar/course/view.php?id=12&section=62*

Videos Explicativos

- Link al repositorio: https://github.com/Marianoarossi/TPI_Programacion1_UTN.git
- Drive con video para descargar: https://drive.google.com/file/d/1v0EIrgZOi8LLT3j3btX_xC0I8f2dBTt4/view?usp=sharing
- Link de YouTube: https://youtu.be/xxEef6YQc4c

Tecnicatura Universitaria en Programación. Programación 1. Unidad Búsquedas y Ordenamiento en programación.: 06 de junio (2025). (1° ed.). Universidad Tecnológica Nacional.