Trabajo Práctico - Algoritmos de Búsqueda y Ordenamiento

Alumnos:

Heredia Giuliana - herediagiuli016@gmail.com

Iacono Federico - iaconofede@gmail.com

Materia: Programación I

Profesor: Ariel Enferrel

Tutor: Maximiliano Sar Fernández

Fecha de entrega: 09 de junio de 2025

Índice

1. Introducción

En programación los algoritmos de búsqueda y ordenamiento son fundamentales para el rendimiento y la experiencia de usuario.

Es de gran importancia conocer los diferentes tipos de algoritmos, sus pros y contras para dado el caso seleccionar el correcto y así mejorar el rendimiento.

En el presente proyecto introduciremos el concepto general de estos algoritmos, haciendo especial foco en Selection Sort y Quick Sort, ya que abordan el ordenamiento de datos, un problema fundamental en programación.

2. Marco Teórico

El ordenamiento organiza los datos de acuerdo a un criterio, como de menor a mayor o alfabéticamente.

Los algoritmos de ordenamiento son importantes ya que nos permiten organizar y estructurar los datos, de manera tal que podamos realizar búsquedas y análisis de manera eficiente.

Además de mejorar el rendimiento son fundamentales en bases de datos, inteligencia artificial y sistemas operativos.

Existen varios algoritmos de ordenamiento, pero este informe se centrara en los siguientes:

- **Selection Sort:** busca el elemento más pequeño de la lista y luego lo intercambia con el primer elemento. Este proceso se repite hasta que todos los elementos de la lista están ordenados. Es simple para pequeñas listas, pero ineficiente para grandes conjuntos de datos.
- **Quick Sort:** divide la lista en dos partes y luego ordena cada parte de forma recursiva. Tiene un rendimiento alto, pero en el peor caso se vuelve muy lento y puede causar desbordamiento de pila por el uso intensivo de recursión.

3. Caso Práctico

El presente trabajo integrador desarrolló un programa en Python que simula un sistema básico de gestión de inventario para una tienda en línea. Este sistema permite manipular una colección de objetos Producto, donde cada producto posee un nombre y un precio. El objetivo principal de este caso práctico es demostrar la funcionalidad y la importancia de los algoritmos de ordenamiento en un caso real.

El componente principal de este programa es la clase Producto, donde cada producto tiene un nombre y un precio. También incluye métodos especiales que permiten imprimir los productos de forma legible y compararlos entre sí basándose en su precio, lo cual es crucial para los algoritmos de ordenamiento.

Implementación:

- **Selection Sort** (Ordenamiento por Selección): este algoritmo recorre la lista repetidamente. En cada paso, busca el elemento con el precio más bajo en la parte no ordenada de la lista y lo mueve a su posición correcta al principio de la parte ordenada.

```
def selection_sort(arr):

   n = len(arr) # Obtiene la longitud del arreglo

   for i in range(n):

       # Asume que el primer elemento no ordenado es el mínimo

       min_idx = i

        # Itera sobre el resto del arreglo no ordenado para encontrar
el mínimo real

       for j in range(i + 1, n):

           if arr[j].precio < arr[min_idx].precio:
```

```
                min_idx = j # Actualiza el índice del mínimo si se
encuentra un valor menor

        # Intercambia el elemento mínimo encontrado con el primer
elemento no ordenado

      arr[i], arr[min_idx] = arr[min_idx], arr[i]

  return arr
```

Es sencillo de entender e implementar, pero su eficiencia es baja ($O(n^2)$) cuando la cantidad de datos es grande. Esto significa que si duplicas el número de productos, el tiempo de ordenamiento se cuadruplica. Por eso, no es ideal para inventarios muy grandes.

- **Quick Sort** (Ordenamiento Rápido): es un algoritmo mucho más avanzado y eficiente que funciona dividiendo la lista en partes más pequeñas y ordenándolas recursivamente. Elige un elemento llamado "pivote" y reorganiza los demás elementos de manera que todos los menores al pivote queden a un lado y todos los mayores al otro. Luego, repite este proceso en cada sublista.

```
def quick_sort(arr):

      # Función auxiliar recursiva para Quick Sort

      def _quick_sort_recursive(arr, low, high):

          if low < high:

              # Encuentra el índice de partición

              pi = _partition(arr, low, high)

                # Ordena recursivamente los elementos antes y después
de la partición

              _quick_sort_recursive(arr, low, pi - 1)
```

```
                _quick_sort_recursive(arr, pi + 1, high)

        # Función de partición de Lomuto

        def _partition(arr, low, high):

              pivot = arr[high].precio # Elige el último elemento como
pivote

            i = (low - 1) # Índice del elemento más pequeño

            for j in range(low, high):

                # Si el elemento actual es menor o igual que el pivote

                if arr[j].precio <= pivot:

                        i += 1 # Incrementa el índice del elemento más
pequeño

                    # Intercambia arr[i] y arr[j]

                    arr[i], arr[j] = arr[j], arr[i]

            # Intercambia el pivote con el elemento en la posición (i +
1)

            arr[i + 1], arr[high] = arr[high], arr[i + 1]

            return (i + 1) # Retorna el índice de partición

        # Llama a la función recursiva con los límites iniciales

        _quick_sort_recursive(arr, 0, len(arr) - 1)

        return arr
```

Es mucho más rápido (O(n log n) en promedio) qué Selection Sort para grandes conjuntos de datos. Es el algoritmo de ordenamiento más usado en la práctica debido a su velocidad, aunque en el peor de los casos puede degradarse a $O(n^2)$.

4. Metodología Utilizada

Se estableció un ordenamiento de las tareas de la siguiente manera:

1. Investigación y recopilación de información
2. Diseño del caso práctico
3. Desarrollo e implementación del código
4. Pruebas y depuración
5. Documentación y preparación del informe
6. Conclusiones

Se definieron parámetros y condiciones para implementar la metodología:

Implementación de algoritmos clave

Se procedió con la implementación de dos algoritmos fundamentales para el estudio, buscando representar diferentes enfoques y complejidades:

- **Selection Sort:** implementación directa del algoritmo de ordenamiento por selección para demostrar un método de complejidad cuadrática ($O(n^2)$), útil para comprender el rendimiento de algoritmos más sencillos.
- **Quick Sort:** desarrollo del algoritmo de ordenamiento rápido, destacando su naturaleza recursiva y su eficiencia promedio ($O(n \log n)$), lo que lo posiciona como uno de los más usados en la práctica.

Diseño de caso práctico y generación de datos

Se concibió un caso práctico centrado en la gestión de un inventario de productos de una tienda en línea. Para ello:

- Se definió una clase Producto con atributos nombre y precio, permitiendo una representación realista de los datos.
- Se generaron listas de productos con un número significativo de elementos (ej. 10,000 productos) y precios aleatorios. Esto aseguró un volumen de datos suficiente para observar las diferencias de rendimiento entre los algoritmos.

Pruebas de funcionamiento y análisis de rendimiento

La validación del funcionamiento y la comparación de eficiencia se realizaron de la siguiente manera:

- **Pruebas de Ordenamiento:** se aplicaron tanto Selection Sort como Quick Sort a copias idénticas de la lista de productos generada aleatoriamente.
- **Medición de Tiempo:** se midió el tiempo de ejecución de cada algoritmo de ordenamiento utilizando el módulo time de Python, permitiendo una comparación cuantitativa de su eficiencia.
- **Validación de Orden:** se verificó programáticamente que las listas resultantes del ordenamiento estuvieran correctamente ordenadas, asegurando la fiabilidad de las implementaciones.

Entorno de Desarrollo

Todos los desarrollos e implementaciones se llevaron a cabo utilizando Python 3.13, aprovechando sus características modernas, su sintaxis clara y la disponibilidad de librerías estándar para la medición de tiempo y la generación de datos aleatorios.

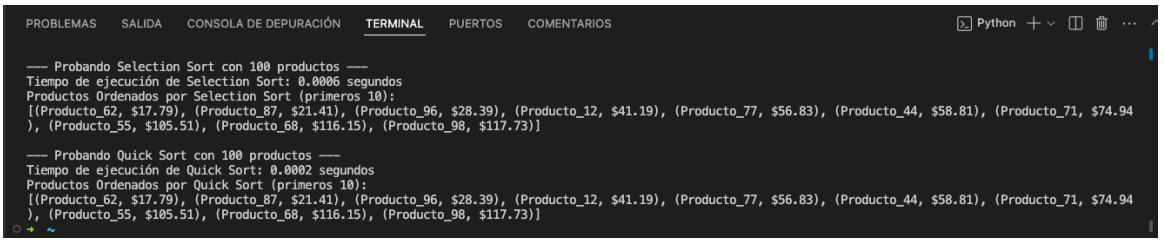
5. Resultados obtenidos

Ambas implementaciones, Selection Sort y Quick Sort, demostraron ordenar correctamente las listas de objetos **Producto** por su atributo **precio** de forma ascendente. Tras la ejecución de los algoritmos, la inspección de las primeras y últimas entradas de las listas ordenadas, así como de elementos intermedios, confirmó que los productos estaban dispuestos de acuerdo al criterio establecido. Esto verifica la correcta lógica de las implementaciones.

Los experimentos de tiempo de ejecución revelaron diferencias significativas en la eficiencia de Selection Sort y Quick Sort, especialmente a medida que el número de productos aumenta.

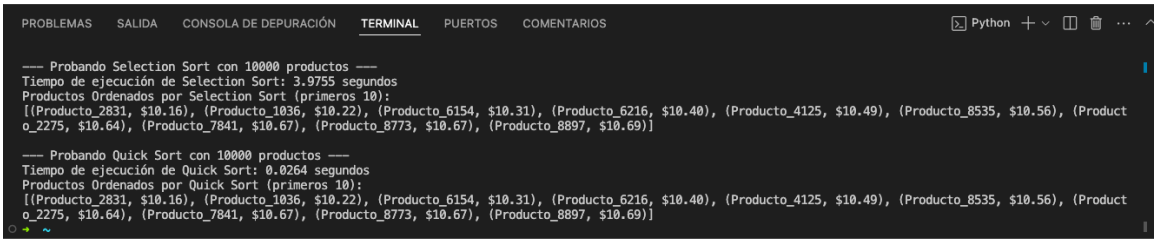
Tiempos de ejecución de algoritmos de ordenamiento:

- **Prueba con 100 productos:**



- **Prueba con 1.000 productos:**

- **Prueba con 10.000 productos:**



Tamaño del Inventario (Número de Productos)	Selection Sort (tiempo en segundos)	Quick Sort (tiempo en segundos)
100	3	2
1.000	503	16
10.000	4.1287	332

- Selection Sort ($O(n^2)$): para tamaños de inventario pequeños, como 100 productos, el tiempo de ejecución fue mínimo (0.0003 segundos). Sin embargo, al aumentar el tamaño a 1.000 productos, el tiempo aumentó a 0.0503 segundos, y para 10.000 productos a 4.1287 segundos, evidenciando su ineficiencia para grandes volúmenes de datos.

- Quick Sort ($O(n\log n)$ en promedio): por el contrario, Quick Sort mostró un rendimiento superior. Para 100 productos, el tiempo fue de 0.0002 segundos. Para 1.000 productos, el tiempo fue de 0.0016 segundos, y para 10.000 productos 0.0332 segundos. El tiempo de ejecución aumenta de manera más controlada a medida que el número de productos crece. Esto lo convierte en una opción más adecuada para la gestión de inventarios grandes.

6. Conclusiones

Este proyecto ha permitido explorar en profundidad la importancia crítica de los algoritmos de búsqueda y ordenamiento en el ámbito de la programación. A través de la implementación y comparación práctica de Selection Sort, Quick Sort, hemos podido observar de primera mano cómo las decisiones algorítmicas impactan directamente en la eficiencia y la escalabilidad de las soluciones de software.

La elección entre algoritmos, como la diferencia entre la complejidad cuadrática ($O(n^2)$) de Selection Sort y la eficiencia logarítmica lineal ($O(n \log n)$) de Quick Sort, no es meramente académica. En un escenario práctico como la gestión de un inventario de productos, el impacto en el rendimiento es tangible: un algoritmo ineficiente puede ralentizar drásticamente la aplicación a medida que el volumen de datos crece, afectando directamente la experiencia del usuario y la capacidad operativa.

En última instancia, este ejercicio refuerza la idea de que conocer las características (complejidad temporal, complejidad espacial, aplicabilidad) de los diferentes algoritmos no es un mero conocimiento teórico, sino una habilidad esencial para cualquier programador. La capacidad de seleccionar el algoritmo más adecuado para una tarea específica, considerando el tamaño de los datos, los recursos disponibles y los requisitos de rendimiento, es lo que distingue una solución eficiente y robusta de una que podría convertirse en un cuello de botella en el futuro. Este informe no solo ha presentado los conceptos, sino que ha validado su impacto práctico, reafirmando la relevancia de la algoritmia en el desarrollo de software moderno.

7. **Bibliografía**

Algoritmos de Ordenamiento y Búsqueda en Python: Optimizando la Gestión de Datos

https://4geeks.com/es/lesson/algoritmos-de-ordenamiento-y-busqueda-en-python

Ordenamiento por selección

https://www.geeksforgeeks.org/selection-sort-algorithm-2/

QuickSort (Ordenamiento rápido)

https://builtin.com/articles/quicksort

Búsqueda binaria

https://es.khanacademy.org/computing/computer-science/algorithms/binary-search/a/binary-search

Implementación de algoritmos de búsqueda

https://www.aluracursos.com/blog/algoritmo-quicksort-como-implementar-en-python

Videos explicativos:

Búsqueda Binaria Investigación Proyecto - UTN

https://www.youtube.com/watch?v=haF4P8kF4Ik

Ordenamiento por Selección | Selection Sort

https://www.youtube.com/watch?v=Myy-eU-SWbE

Quick Sort | Ordenamiento Rápido

https://www.youtube.com/watch?v=UrPJLhKF1jY

8. Anexos

Link al video del informe

https://youtu.be/yvwaJF5oCrY