Trabajo Práctico – Ordenamiento y Búsqueda

Estudiantes:

Chiaravalloti Agustín– achiaravalloti54@gmail.com
Etchevest Jorgelina Carla – etchevestjor@gmail.com

Materia: Programación I. Comisión 2.
Profesora: Trapé Julieta
Tutor: Barrera Oltra Miguel
Fecha de Entrega: 9 de Junio 2025

Índice

1. Introducción

En el análisis y procesamiento de datos, contar con estructuras ordenadas y accesibles de forma eficiente resulta fundamental. Los algoritmos de búsqueda y ordenamiento permiten mejorar tanto la organización como el rendimiento en tareas donde se requiere acceder o modificar grandes volúmenes de información. En este contexto, el lenguaje Python ofrece herramientas claras y versátiles para implementar estos algoritmos de manera sencilla, permitiendo explorar su funcionamiento tanto a nivel teórico como práctico.

El presente trabajo se propone estudiar distintos algoritmos de ordenamiento y búsqueda de datos, evaluando sus ventajas, limitaciones y aplicabilidad según el contexto.

UTN
TECNICATURA UNIVERSITARIA
EN PROGRAMACIÓN
A DISTANCIA

Objetivo general

- Analizar las ventajas, desventajas y posibles usos de distintos algoritmos de búsqueda y ordenamiento en Python, comprendiendo en qué contextos resulta más apropiado aplicar cada uno.

Objetivos específicos

- Implementar un caso práctico con el algoritmo QuickSort, observando cómo se comporta frente a una mala elección del pivote y proponiendo una solución para evitar el desbalanceo.

2. Marco Teórico

ALGORITMOS DE BÚSQUEDA

Lineal

La búsqueda lineal es un algoritmo destacado por su sencillez y facilidad de implementación. Su funcionamiento consiste en recorrer secuencialmente cada elemento de una lista hasta hallar el valor deseado, lo que lo convierte en una solución flexible y aplicable a cualquier tipo de lista, sin importar si está ordenada o no. Esta característica la hace especialmente útil en situaciones en las que no se dispone de estructuras de datos más complejas o cuando se trabaja con listas pequeñas.

Sin embargo, su principal desventaja radica en su ineficiencia cuando se trata de listas extensas. Al analizar cada elemento uno a uno, el tiempo de búsqueda aumenta proporcionalmente con el tamaño de la lista, lo que puede resultar poco práctico en contextos de gran volumen de datos. Además, en listas ordenadas, este método no aprovecha esa organización, por lo que algoritmos más eficientes como la búsqueda binaria resultan mucho más adecuados.

Por ejemplo, en la siguiente lista

A = [1, 3, 8, 4, 9, 0]

El algoritmo recorrería elemento por elemento buscando el valor que hayamos elegido como objetivo. Supongamos que queremos encontrar el 8. El algoritmo comenzará evaluando el primer valor, es decir la posición 0

A [0] == 8 No lo es. Sigue a evaluar el elemento en la posición siguiente.

A [1]== 8 No lo es. Sigue a evaluar el elemento en la posición siguiente

A [2]==8 Si, es verdad. Por lo tanto, encontró el valor buscado.

Método que funciona bien, independientemente que la lista esté ordenada o no.

UTN
TECNICATURA UNIVERSITARIA
EN PROGRAMACIÓN
A DISTANCIA

Binaria

La búsqueda binaria se destaca principalmente por su alta eficiencia cuando se aplica sobre listas ordenadas. Gracias a que divide el espacio de búsqueda a la mitad en cada paso, su tiempo de ejecución crece logarítmicamente con respecto al tamaño de la lista, lo que permite localizar un elemento con muy pocas comparaciones, especialmente en estructuras de datos grandes. Esta ventaja la convierte en una opción mucho más rápida que la búsqueda lineal en escenarios adecuados.

No obstante, presenta ciertas limitaciones. Una de ellas es que solo puede utilizarse en listas previamente ordenadas, por lo que, si los datos no están organizados, es necesario realizar un paso previo de ordenamiento. Además, su implementación es algo más elaborada en comparación con la búsqueda lineal, ya que generalmente requiere aplicar técnicas recursivas o manejar con cuidado los índices de inicio y fin de la lista durante el proceso.

Pongamos de ejemplo la misma lista A, tomada en el método anterior, para demostrar cómo opera la búsqueda binaria. El primer requisito para aplicar el método es que la lista esté ordenada.

A = [1, 3, 8, 4, 9, 0]

Ordenada

A = [0, 1, 3, 4, 8, 9]

Para estimar el valor del medio utiliza los índices, en este caso va de 0 a 5, la posición media seria 2,5 pero se toma el valor entero, 2. El elemento en la posición 2

A [2] = 3 No es nuestro objetivo. Pero permite descartar la parte izquierda de la lista ordenada, al ser 3 menor que el valor objetivo (8).

La lista que queda es [4, 8, 9]

Al buscar el valor central es el valor buscado: 8.

En este sencillo ejemplo es posible comparar que, entre ambos métodos de búsqueda, la búsqueda binaria llevó menos comparaciones para encontrar el valor objetivo, comparada con la búsqueda lineal. Con el único requisito previo, que la lista estuviese ordenada antes de comenzar la búsqueda.

ALGORITMOS DE ORDENAMIENTO

Los algoritmos de ordenamiento son procedimientos fundamentales en la informática y el análisis de datos, ya que permiten reorganizar los elementos de una lista o estructura según un criterio determinado, generalmente de menor a mayor o viceversa.

UTN
TECNICATURA UNIVERSITARIA
EN PROGRAMACIÓN
A DISTANCIA

Existen diversos métodos de ordenamiento, cada uno con sus propias ventajas, desventajas y comportamientos según el tamaño y la naturaleza de los datos. Algunos, como el método burbuja, son intuitivos y fáciles de implementar, aunque poco eficientes para grandes volúmenes. Otros, como MergeSort o QuickSort, están diseñados para ofrecer un mejor rendimiento mediante estrategias más complejas como la recursión o la división y conquista.

Burbuja

El algoritmo de ordenamiento por burbuja funciona comparando pares de elementos e intercambiándolos si están en el orden incorrecto, este proceso se hace una y otra vez hasta que la lista esté ordenada de forma correcta.

Por ejemplo, tomando una lista con 6 elementos:

A = [1, 3, 8, 4, 9, 0]

Primera vuelta:

Compara 1 y 3: No cambia, dado que 1 es menor que 3.

Compara 3 y 8: No cambia, dado que 3 es menor que 8.

Compara 8 y 4: Intercambia. Quedando la nueva lista [1, 3, 4, 8, 9, 0]

Compara 8 y 9: No cambia. 8 es menor que 9.

Compara 9 y 0: Intercambia. Quedando la nueva lista [1, 3, 4, 8, 0, 9]

Segunda Vuelta:

Compara 1 y 3: No cambia.

Compara 3 y 4: No cambia.

Compara 4 y 8: No cambia.

Compara 8 y 0: Intercambia. [1, 3, 4, 0, 8, 9]

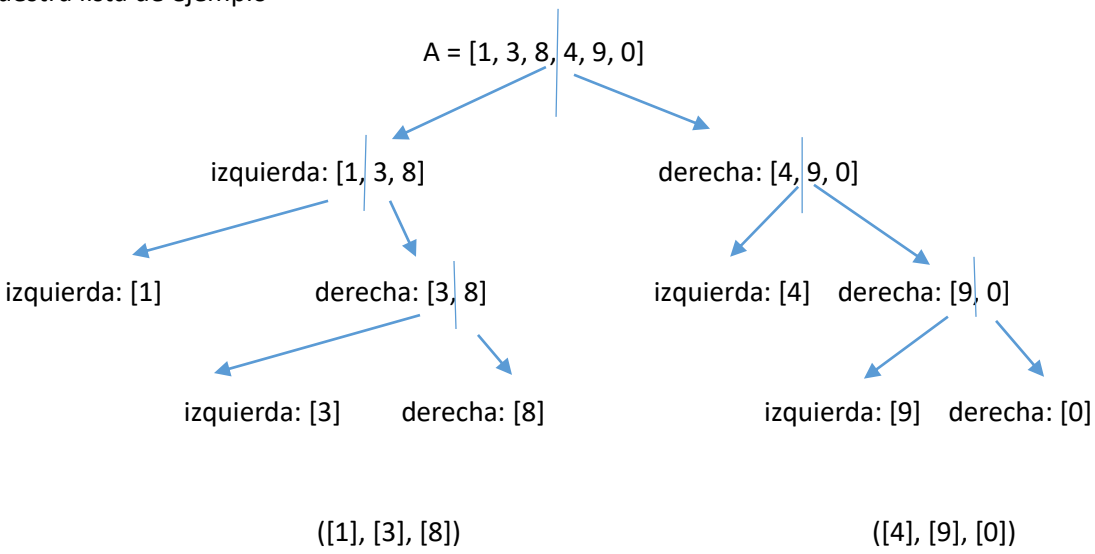
Seguirá con el mismo proceso hasta que el cero haya quedado en la primera posición. En este caso el valor más bajo estaba en el último lugar. El tiempo no es una consideración menor en este tipo de algoritmos por la cantidad de pasos requeridos, en consecuencia, funciona bien en listas pequeñas, pero es ineficiente para listas más grandes.

MergeSort

El algoritmo divide la lista en mitades hasta que cada sub-lista tenga un solo elemento. Luego fusiona esas sub-listas ordenadamente. El resultado final es una lista completamente ordenada.

UTN
TECNICATURA UNIVERSITARIA
EN PROGRAMACIÓN
A DISTANCIA

En nuestra lista de ejemplo



([1], [3], [8])

([4], [9], [0])

Ahora que tenemos todo dividido, empezamos a fusionar ordenadamente desde las partes más pequeñas, hasta llegar al resultado deseado.

[0, 1, 3, 4, 8, 9]

QuickSort

Elige un elemento como pivote, separar menores a la izquierda y mayores a la derecha, y aplica el mismo procedimiento recursivamente.

En nuestra lista de ejemplo, tomamos el último valor como pivote:

A = [1, 3, 8, 4, 9, 0]

Pivote: 0
Menores a la izquierda: [ ]
Mayores a la derecha: [1, 3, 8, 4, 9]
Aplico recursivamente.

Pivote: 9
Menores a la izquierda: [1, 3, 8, 4]
Mayores a la derecha: [ ]
Aplico recursivamente.

UTN
TECNICATURA UNIVERSITARIA
EN PROGRAMACIÓN
A DISTANCIA

Pivote: 4
Menores a la izquierda: [1,3]
Mayores a la derecha: [8]
Aplico recursivamente.

Pivote: 3
Menores a la izquierda: [1]
Mayores a la derecha: [ ]

Juntado todo:

[1] + [3]+ [4] + [8] → [1, 3, 4, 8]

Luego [1, 3, 4, 8] + [9] → [1, 3, 4, 8, 9]

Finalmente: [0] + [1, 3, 4, 8, 9]

Resultado final: [0, 1, 3, 4, 8, 9]

Claramente el funcionamiento del algoritmo depende de la ubicación del número es elegido como pivote.

3. Caso Práctico

QuickSort es uno de los algoritmos de ordenamiento más utilizados por su eficiencia en conjuntos de datos medianos o grandes. Una de sus principales ventajas es que opera directamente sobre la lista original, sin necesidad de crear estructuras auxiliares, a diferencia de MergeSort, que requiere espacio adicional para fusionar sublistas. Esta característica convierte a QuickSort en una opción muy rápida y liviana en la mayoría de los casos prácticos.

Sin embargo, su rendimiento puede variar significativamente según cómo se seleccione el pivote. Python —al igual que muchos lenguajes— permite implementar QuickSort con diferentes estrategias de selección de pivote: puede elegirse el primer elemento, el último, uno aleatorio o incluso aplicar la técnica de "mediana de tres", que consiste en comparar el primero, el último y el del medio para obtener un pivote más representativo. La elección incorrecta del pivote puede provocar un desbalanceo en la división de la lista, especialmente cuando los datos están ordenados o casi ordenados. En estos casos, el algoritmo pierde eficiencia al generar particiones muy dispares, y el número de llamadas recursivas crece significativamente.

Para ilustrar este fenómeno, trabajaremos con una lista específica y aplicaremos QuickSort utilizando diferentes criterios de pivote. Analizaremos cómo se produce el desbalanceo y cómo puede corregirse mediante una mejor elección del pivote.

Para comenzar, probamos en Google Colab con una lista ordenada de 1000 números, utilizando como pivote el primer elemento. Esta elección provoca un desbalance severo en el algoritmo QuickSort, ya que una de las particiones queda vacía y la otra contiene casi todos los elementos, generando así el peor caso posible en cuanto a rendimiento. Luego, realizamos la misma prueba utilizando como pivote la mediana de tres (calculada a partir del primer, el último y el valor central de la lista), lo que permite obtener particiones más equilibradas.

Al trasladar este código desde Colab a un entorno local de Python, se produjo un error de desbordamiento de pila (RecursionError) debido a la profundidad excesiva de llamadas recursivas generadas por el desbalance intencional del primer caso.

Para solucionarlo, importamos el módulo SYS y aplicamos sys.setrecursionlimit(20000) para aumentar el límite máximo de recursión permitido por Python, lo que nos permitió ejecutar correctamente el algoritmo QuickSort incluso en los peores casos, donde la recursividad alcanza profundidades muy altas. Sin embargo, es importante tener en cuenta que incrementar este límite puede implicar riesgos de consumo excesivo de memoria y posibles bloqueos del sistema si no se controla adecuadamente la estructura del algoritmo.

main.py > ...
You, 4 minutes ago | 1 author (You)

```python
import time
import timeit
import sys

sys.setrecursionlimit(20000)

#Método Quicksort con lista ordenada y el primer valor como pivote

contador_primero = 0

def quicksort(arr):
    global contador_primero
    contador_primero += 1

    if len(arr) <= 1:
        return arr
    else:
        pivot = arr[0]
        less = [x for x in arr[1:] if x <= pivot]
        greater = [x for x in arr[1:] if x > pivot]
        return quicksort(less) + [pivot] + quicksort(greater)
```

UTN
TECNICATURA UNIVERSITARIA
EN PROGRAMACIÓN
A DISTANCIA

```python
def mediana_de_tres(a, b, c):
    return sorted([a, b, c])[1]

contador_media = 0

def quicksort_media(arr):
    global contador_media
    contador_media += 1

    if len(arr) <= 1:
        return arr
    else:
        primero = arr[0]
        medio = arr[len(arr) // 2]
        ultimo = arr[-1]
        pivot = mediana_de_tres(primero, medio, ultimo)

        less = [x for x in arr if x < pivot]
        equal = [x for x in arr if x == pivot]
        greater = [x for x in arr if x > pivot]

        return quicksort_media(less) + equal + quicksort_media(greater)
```

```python
# Medir tiempo y llamadas recursivas para cada pivote:
start_time = timeit.default_timer()
quicksort(A.copy())
end_time = timeit.default_timer()
print("Tiempo para una lista con pivote en primer elemento:", end_time - start_time)
print("Llamadas recursivas (pivote primero):", contador_primero)


start_time_media = timeit.default_timer()      You, 55 seconds ago • Uncommitted changes
quicksort_media(B.copy())
end_time_media = timeit.default_timer()
print("Tiempo para una lista con pivote en la media de 3:", end_time_media - start_time_media)
print("Llamadas recursivas (pivote media de 3):", contador_media)
```

```
Tiempo para una lista con pivote en primer elemento: 0.053208400000585
Llamadas recursivas (pivote primero): 3998
Tiempo para una lista con pivote en la media de 3: 0.00175710000621620
Llamadas recursivas (pivote media de 3): 2046
```

UTN
TECNICATURA UNIVERSITARIA
EN PROGRAMACIÓN
A DISTANCIA

4. Metodología Utilizada

4.1. Entorno y herramientas de desarrollo

Para implementar y probar los algoritmos se utilizaron dos entornos complementarios: Google Colab y un editor local de Python. Google Colab permitió trabajar con listas extensas gracias a su mayor tolerancia a la recursividad, mientras que el entorno local fue útil para validar el comportamiento en escenarios con menos recursos, identificando los límites del lenguaje en situaciones específicas.

4.2. Estrategia de evaluación

Se implementaron dos versiones del algoritmo QuickSort, variando la estrategia de selección del pivote: una utilizando el primer elemento de la lista y otra basada en la técnica de la mediana de tres. Para simular un caso desfavorable, se utilizó una lista ordenada de 1000 elementos, lo que permite observar el impacto de una mala elección del pivote. Se evaluaron tanto la cantidad de llamadas recursivas como el tiempo de ejecución en cada caso.

En un principio, en el entorno local de Python se optó por reducir la lista a 100 elementos para evitar errores de desbordamiento de pila (RecursionError). Sin embargo, para poder trabajar con los 1000 datos en este entorno, fue necesario importar el módulo sys y aumentar el límite de recursión mediante sys.setrecursionlimit(20000), lo que permitió ejecutar el algoritmo sin interrupciones en los casos más exigentes.

4.3 Evaluación del rendimiento

Se observaron y compararon las diferencias en el número de llamadas recursivas, el tiempo de ejecución y el balance de particiones generadas por cada implementación. En casos donde la recursividad fue excesiva, se generaron errores de desbordamiento de pila, los cuales fueron documentados como parte del análisis.

5. Resultados Obtenidos y Conclusiones

Los algoritmos de búsqueda y ordenamiento cumplen un rol central en el análisis y procesamiento eficiente de datos. Su correcta implementación permite optimizar tiempos de respuesta y reducir el uso de recursos, especialmente cuando se trabaja con grandes volúmenes de información. En contextos de análisis estadístico, la inteligencia artificial o la simulación numérica, elegir el algoritmo adecuado puede determinar la viabilidad de un proyecto. Por eso, comprender las ventajas y

limitaciones de métodos como la búsqueda lineal o binaria, y de algoritmos de ordenamiento como Burbuja, MergeSort o QuickSort, es clave para tomar decisiones informadas en el diseño de soluciones computacionales.

En el caso práctico desarrollado, los resultados permitieron evidenciar con claridad cómo la elección del pivote influye directamente en el rendimiento y la eficiencia del algoritmo QuickSort. Cuando se seleccionó el primer elemento de una lista ordenada como pivote, el algoritmo presentó un fuerte desbalanceo, generando una recursividad excesiva y alcanzando rápidamente los límites del sistema. En cambio, al utilizar la mediana de tres como estrategia de pivote, las particiones fueron más equilibradas, lo que redujo significativamente la profundidad de las llamadas recursivas y mejoró el rendimiento general.

Estos resultados refuerzan la importancia de considerar el tipo de datos y su distribución al momento de aplicar algoritmos de ordenamiento. QuickSort es extremadamente eficiente en condiciones adecuadas, pero puede degradarse rápidamente si no se elige correctamente el pivote. Este caso práctico evidencia cómo decisiones de implementación aparentemente simples pueden tener un gran impacto en la eficiencia del código, y pone de relieve el valor del análisis previo del conjunto de datos antes de aplicar técnicas de procesamiento.

6. Anexos

Link al repositorio de Github: https://github.com/Jorgelina-Etchevest/tp_programacion

Link al repositorio de Github: https://github.com/AgustinChiaravalloti/tp_integradorP1.git

7. Bibliografía

- Bahit, E. (s.f.). *Introducción a la ciencia de datos con Python*. Escuela de Informática Eugenia Bahit. http://escuela.eugeniabahit.com
- McKinney, W. (2018). *Python for data analysis: Data wrangling with pandas, NumPy, and IPython* (2nd ed.). O'Reilly Media.
- Sánchez Alberca, A. (2020). *Manual de Python*. Creative Commons BY-NC-SA 4.0.