UTN
TECNICATURA UNIVERSITARIA EN PROGRAMACIÓN A DISTANCIA

Trabajo Integrador Programación 1

Algoritmos de búsqueda y ordenamiento

Alumnas:

- Demarchi, Eugenia – Comisión 2
- Rege, María Soledad – Comisión 20

Materia: Programación 1

Docente: Quirós, Nicolás

Año: 2025

Índice

Trabajo integrador Programación 1

Algoritmos de Búsqueda y ordenamiento

1. Introducción

El presente trabajo se desarrolla en torno a los algoritmos de búsqueda y ordenamiento, pilares fundamentales en la programación y el desarrollo de software eficiente. La elección de este tema responde a la necesidad de comprender cómo estas estructuras afectan directamente el rendimiento de las aplicaciones y, en consecuencia, la experiencia del usuario.

En un entorno digital cada vez más competitivo, los usuarios esperan que las aplicaciones respondan de manera casi instantánea. Según un informe de *Google* (2023), el 53% de los usuarios abandonan un sitio web si éste tarda más de 3 segundos en cargar. Esta estadística evidencia que la eficiencia algorítmica no es solo una cuestión técnica, sino también una condición clave para la usabilidad y la permanencia en el mercado. Por ello, la implementación de algoritmos adecuados de búsqueda y ordenamiento tiene un impacto directo en la calidad del software.

Además, la correcta selección de estos algoritmos en función del tamaño de los datos y de la estructura empleada es un indicador de buenas prácticas profesionales. Tomar decisiones informadas sobre qué técnica aplicar en cada contexto permite optimizar recursos, reducir tiempos de ejecución y evitar cuellos de botella computacionales.

Comprender el funcionamiento interno de los principales algoritmos de búsqueda y ordenamiento, analizar sus distintos rendimientos en relación al ordenamiento y el consumo de memoria y aplicar estos conocimientos al lenguaje de programación Python, abordando sus implementaciones más comunes son algunos de los objetivos a llevar a cabo en este trabajo.

2. Marco teórico

Búsqueda

La búsqueda algorítmica de elementos es el proceso mediante el cual un algoritmo recorre una estructura de datos con el objetivo de localizar un valor específico o determinar su ausencia. Este proceso puede aplicarse a distintos tipos de estructuras, como listas, árboles, grafos o tablas hash, y su eficiencia depende tanto del algoritmo utilizado como de la organización de los datos. Los algoritmos de búsqueda pueden clasificarse según el método empleado -por comparación, basada en hashing, heurística- como por el tamaño de la entrada. Este último factor es fundamental al momento de implementar un algoritmo, ya que influye directamente en su rendimiento.

El tiempo que tarda un algoritmo en ejecutarse se conoce como complejidad temporal, y está directamente relacionado con la eficiencia del algoritmo al resolver un problema en función del tamaño de los datos de entrada. Para analizar esta complejidad, se utiliza la notación Big O (O-grande), que permite estimar cómo crece el tiempo de ejecución a medida que aumenta el tamaño de la entrada. Esta herramienta es fundamental para comparar distintos algoritmos y elegir el más adecuado según la tarea a resolver.

El análisis de la complejidad puede abordarse desde tres perspectivas, el peor caso es el escenario más desfavorable, donde el algoritmo tarda el máximo posible, y el mejor caso es aquel donde el algoritmo encuentra la solución inmediatamente y caso promedio: el comportamiento esperado en la mayoría de las ejecuciones.

En los algoritmos de búsqueda y ordenamiento es común encontrar complejidades cuadráticas –$O(n^2)$– y logarítmicas –$O(\log n)$–.

La complejidad cuadrática implica que el tiempo de ejecución crece proporcional al cuadrado del tamaño de la entrada, lo cual la hace poco eficiente para grandes volúmenes de datos. En cambio, la complejidad logarítmica crece muy lentamente, incluso con conjuntos de datos grandes, ya que el algoritmo divide el problema en partes más pequeñas en cada paso, como sucede, por ejemplo, en la búsqueda binaria.

A continuación, se desarrollan los algoritmos de búsqueda más usados según su método. Asimismo, se caracteriza cada uno con su complejidad temporal, ventajas y desventajas.

Algoritmos de búsqueda

Búsqueda por comparación

La búsqueda lineal (secuencial) es la más simple. Recorre cada elemento de forma secuencial hasta encontrar el elemento deseado o llegar hasta el final. Es un algoritmo fácil de implementar y no requiere que la lista esté ordenada, pero es poco eficiente en conjuntos grandes de elementos. Suele ser utilizada en arrays, listas o cualquier colección lineal.

En relación a su complejidad temporal, su caso promedio es O(n), es decir a medida que la entrada de datos es más grande, también crece su tiempo de ejecución.

La búsqueda binaria es un algoritmo que requiere que los elementos estén previamente ordenados. Divide el conjunto en dos mitades y busca el elemento deseado en la mitad correspondiente. Así, repite el proceso hasta encontrar el elemento o determinar que no está en el conjunto. Este algoritmo es muy eficiente en listas grandes ya ordenadas. Asimismo, reduce el problema a la mitad en cada paso. Es típicamente utilizada en arrays o listas ordenadas. Su complejidad temporal es O(log n), lo que significa que, aunque el tamaño del problema crezca, el número de pasos necesarios aumenta lentamente.

La búsqueda por interpolación es un algoritmo que mejora la búsqueda binaria al estimar la posición del elemento deseado en función de su valor. Para ser eficiente requiere una distribución uniforme de valores. Desde ya, esta búsqueda es más rápida que la búsqueda binaria, pero solo funciona bien si los datos están ordenados y bien distribuidos. Si los datos no están uniformemente distribuidos es tan lenta como la lineal. Además, es más compleja de implementar que la búsqueda binaria. Se utiliza comúnmente en arreglos ordenados y distribuidos uniformemente Tiene una complejidad temporal promedio de O (log log n), lo que implica que es aún más rápida que la búsqueda binaria y que los pasos necesarios crecen aún más lentamente.

Búsqueda basada en hashing

La búsqueda de hash es un método que asigna a cada elemento una posición específica dentro de una tabla, utilizando una función hash. Esta función transforma cada dato en un índice, permitiendo almacenarlo y recuperarlo rápidamente. Gracias a este mecanismo, la búsqueda se realiza en tiempo constante, incluso en conjuntos de datos grandes, lo que la hace muy eficiente.

Este algoritmo permite un rápido acceso a datos. Además, es escalable a grandes volúmenes de datos y es fácil de implementar para tipos de datos simples. Asimismo, no es necesario que los datos estén ordenados.

Entre sus desventajas se pueden mencionar las colisiones: cuando dos claves distintas producen el mismo índice se generan 'colisiones', que deben resolverse mediante técnicas adicionales. Si la función hash está mal diseñada se generan muchas colisiones, degradando el rendimiento. Además, puede requerir mucha memoria si la tabla se sobredimensiona (con la intención de evitar colisiones). Asimismo, no mantiene orden. No es posible recorrer los elementos de forma ordenada.

Las estructuras típicas donde se utiliza este algoritmo son las tablas hash, con técnicas como encadenamiento o direccionamiento abierto para resolver colisiones. Tiene una complejidad temporal promedio de O(1) para inserción, búsqueda y eliminación, siempre que la función hash sea adecuada y las colisiones estén bien gestionadas

Dado el tiempo limitado para desarrollar este trabajo, mencionaremos brevemente algunos otros casos de búsqueda que emplean métodos distintos.

Búsqueda heurística

Las heurísticas se basan en el conocimiento o experiencia acumulada sobre un determinado problema, lo que permite tomar decisiones más rápidas y eficientes. Este tipo de búsqueda guía la exploración hacia las soluciones más prometedoras, reduciendo el espacio de búsqueda sin garantizar siempre la solución óptima.

Búsqueda en estructuras (árboles y grafos)

Este método se basa en recorrer nodos dentro de estructuras jerárquicas o conexas, como árboles y grafos, siguiendo sus ramificaciones. Dependiendo de la estrategia utilizada, se puede priorizar la exploración por niveles (en anchura) o por ramas (en profundidad).

Búsqueda probabilística y aleatoria

Utiliza el azar o mecanismos inspirados en procesos naturales para explorar soluciones en espacios amplios o complejos. Estas técnicas son valiosas en problemas donde los métodos predecibles pueden quedar atrapados en soluciones poco eficientes.

Ordenamiento

El ordenamiento es una operación fundamental en la programación que consiste en reorganizar los elementos de una lista u otra colección de datos según un criterio determinado. Este criterio puede ser una secuencia ascendente, descendente, alfabética, numérica, cronológica, entre otros.

La importancia del ordenamiento radica en varios aspectos. En primer lugar, facilita las operaciones de búsqueda. Por ejemplo, la búsqueda binaria, ampliamente utilizada por su eficiencia, sólo es posible si la lista se encuentra previamente ordenada. Además, mejora la presentación y comprensión de los datos, ya que una estructura ordenada resulta más legible. Por último, ordenar los datos puede reducir el costo computacional de otras tareas, ya que muchos algoritmos trabajan más rápido sobre listas ordenadas.

Los algoritmos de ordenamiento se diferencian tanto por su lógica de funcionamiento como por su complejidad temporal, es decir, la cantidad de pasos que requieren para completarse en función del tamaño de la entrada. Algunos presentan una complejidad cuadrática ($O(n^2)$), lo cual los vuelve poco eficientes para grandes volúmenes de datos; otros, en cambio, logran una eficiencia mucho mayor, con una complejidad logarítmica ($O(n \log n)$).

Algoritmos de ordenamiento

El *Bubble Sort* u ordenamiento por burbuja es uno de los métodos más sencillos. Consiste en recorrer repetidamente la lista comparando pares de elementos adyacentes, e intercambiándolos si están en el orden incorrecto. Aunque su implementación es muy simple, su rendimiento es pobre, especialmente en listas largas, ya que requiere una gran cantidad de comparaciones incluso cuando la lista está parcialmente ordenada.

El *Selection Sort* u ordenamiento por selección, separa la lista en una parte ordenada y otra desordenada. Continuamente selecciona el valor mínimo de la parte desordenada y lo coloca al final de la parte ordenada. Aunque mejora un poco el comportamiento frente al Bubble Sort, su eficiencia sigue siendo limitada en términos de complejidad temporal, ya que también opera en $O(n^2)$.

Por otro lado, el *Insertion Sort* funciona construyendo la lista ordenada de a un elemento a la vez. A medida que recorre los datos, va insertando cada nuevo valor en su lugar correcto dentro de la lista ordenada. Este método es particularmente eficiente cuando los datos ya se encuentran parcialmente ordenados o cuando se trabaja con listas pequeñas, aunque su rendimiento decrece con volúmenes grandes.

En un nivel superior de eficiencia se encuentra el *Merge Sort*, un algoritmo basado en la estrategia de “divide y vencerás”. Su funcionamiento se basa en dividir recursivamente la lista en mitades hasta que cada sublista contenga un único elemento, y luego fusionar esas sublistas ordenadas hasta reconstruir la lista completa. Este algoritmo garantiza un rendimiento consistente con complejidad $O(n \log n)$ en todos los casos, aunque requiere memoria adicional para crear sublistas temporales. Es particularmente útil en estructuras como listas enlazadas, aunque en Python, que utiliza listas como arreglos dinámicos, el impacto de la memoria es un punto a considerar.

También dentro de los algoritmos de tipo “divide y vencerás”, el *Quick Sort* destaca por su gran velocidad promedio. En este caso, se selecciona un pivote y se divide la lista en dos partes. Una parte contiene los elementos menores al pivote y la otra, los mayores. A partir de allí, se aplica el mismo procedimiento recursivamente a cada sublista. Su rendimiento promedio es también O(n log n).

Algoritmos de ordenamiento en Python

En el caso de Python, el algoritmo de ordenamiento por defecto es *Timsort*, una versión híbrida y optimizada que combina *Merge Sort* e *Insertion Sort*. Es un algoritmo altamente eficiente, con una complejidad O(n log n) en la mayoría de los casos. Aunque fue desarrollado para Python, hoy en día también se emplea en otros lenguajes como Java.

Python ofrece dos formas principales de acceder a este algoritmo: el método list.sort() y la función sorted(). La primera se aplica directamente sobre listas y modifica el objeto original, mientras que la segunda devuelve una nueva lista ordenada sin alterar la original. Ambos métodos aceptan un parámetro opcional llamado reverse que, al establecerse en True, permite ordenar los datos de forma descendente.

3.Caso práctico

El objetivo de este caso es simular un sistema de catálogo de libros y aplicar distintos algoritmos de búsqueda y ordenamiento para gestionar eficientemente los datos. Este ejercicio permite observar en la práctica cómo distintos algoritmos resuelven el mismo problema con distintos niveles de eficiencia, especialmente cuando se trabaja con grandes volúmenes de información.

Para comenzar, fue necesario simular el inventario con una cantidad de elementos suficientes para demostrar nuestro caso. Para esto se utilizó un dataset en formato .csv proporcionado por el sitio kaggle con información real de libros, incluyendo campos como título, autor, número de páginas o valoración promedio. Este dataset cuenta con más de diez mil elementos.

En cuanto a los algoritmos de búsqueda, se implementaron dos enfoques: búsqueda lineal y búsqueda binaria, con el objetivo de comparar su rendimiento y eficacia. No se incluyó la búsqueda por interpolación, ya que los datos están ordenados alfabéticamente, lo que impide una distribución uniforme, condición necesaria para el correcto funcionamiento de dicho método. Estas estrategias representan casos opuestos en cuanto a eficiencia algorítmica: la búsqueda lineal, con complejidad O(n), y la búsqueda binaria, con complejidad O(log n), cuyos tiempos de ejecución reflejan de manera concreta las diferencias teóricas entre ambas.

Búsqueda lineal

```python
#Función para elegir autor aleatorio
def elegir_autor_aleatorio(dataset):
  return random.choice(dataset)['authors']

# Función de búsqueda lineal:
# Recorre la lista comparando el autor objetivo con el autor de cada elemento del dataset.
# La comparación ignora mayúsculas y minúsculas (convierte ambos nombres a minúsculas).
def busqueda_lineal(lista, autor_objetivo):
  for i in range(len(lista)):
    if lista[i]['authors'].lower() == autor_objetivo.lower():
      return i
  return -1

#Elegir autor aleatorio
autor_buscado = elegir_autor_aleatorio(libros)

#Medir tiempo de busqueda lineal:
inicio=time.time()
resultado=busqueda_lineal(libros, autor_buscado)
fin=time.time()
tiempo_busqueda_lineal = fin - inicio
```

Búsqueda binaria

```python
# Ordenar la lista alfabéticamente por autor (ignorando mayúsculas/minúsculas)
libros_ordenados= sorted(libros, key=lambda x: x['authors'].lower())

# Función de búsqueda binaria:
# Requiere una lista ordenada alfabéticamente por autor (en minúsculas para
# comparación insensible a mayúsculas).
# Utiliza un bucle while para dividir la lista en mitades hasta encontrar
# el autor buscado o agotar las posibilidades.
def busqueda_binaria(lista, autor_objetivo):
  izquierda, derecha= 0, len(lista) -1
  objetivo = autor_objetivo.lower()

  while izquierda <= derecha:
    mitad = (izquierda + derecha) // 2
    autor_mitad= lista[mitad]['authors'].lower()

    if autor_mitad == objetivo:
      return mitad #retorna el valor del medio (mejor caso)
    elif autor_mitad < objetivo:
      izquierda = mitad +1 # El autor está en la mitad derecha
    else:
      derecha= mitad -1 # El autor está en la mitad izquierda
  return -1       # Autor no encontrado

# Medir tiempo de búsqueda binaria
inicio_binaria = time.time()
resultado_binaria = busqueda_binaria(libros_ordenados, autor_buscado)
fin_binaria = time.time()
tiempo_busqueda_binaria=fin_binaria - inicio_binaria
```

En cuanto a los algoritmos de ordenamiento, se implementaron tres casos diferentes: *Bubble sort, Merge sort y Quick Sort*. Cada uno fue evaluado en términos de eficiencia y tiempos de ejecución, permitiendo así comprender sus diferencias en complejidad algorítmica y aplicabilidad según el caso.

La elección de estos métodos tiene varias razones. *Bubble Sort* se incluyó como ejemplo clásico de algoritmo ineficiente para listas extensas, dado que presenta un tiempo de ejecución de orden $O(n^2)$. Su implementación resultó útil como punto de comparación frente a otros métodos más eficientes.

Bubble sort

```python
# Itera cada elemento y lo va comparando con su adyacente.
# Si es mayor, lo mueve un lugar en la lista y sigue comparando
# Parametro key para definir atributo a ordenar
def bubble_sort(lista, key=lambda x: x):  2 usages  Sol Rege
    n = len(lista)

    for i in range(n):
        for j in range(0, n - i - 1):
            if key(lista[j]) > key(lista[j + 1]):
                lista[j], lista[j + 1] = lista[j + 1], lista[j]
```

Por otro lado, se eligieron *Merge Sort* y *Quick Sort* por su mejor rendimiento, especialmente sobre grandes volúmenes de datos. *Merge Sort* destaca por su estabilidad y por ser la base del algoritmo de ordenamiento utilizado internamente por Python en su método sort().

Merge sort

```python
# Divide la lista en mitades hasta que cada lista tenga un solo elemento,
# se utiliza la recursividad y un condicional que verifica cuando exista un elemento en la lista
# y da por finalizada la recursión
# Parametro key para definir atributo a ordenar
def merge_sort(lista, key=lambda x: x):  4 usages  Sol Rege
    # Condición para detener la recursión
    if len(lista) == 1:
        return lista

    # División de la lista utilizando slicing
    mitad = len(lista) // 2
    izquierda = lista[:mitad]
    derecha = lista[mitad:]

    # Recursividad
    izquierda_ordenada = merge_sort(izquierda, key)
    derecha_ordenada = merge_sort(derecha, key)

    # Retorna las listas ordenadas
    return merge(izquierda_ordenada, derecha_ordenada, key)

# Función para unificar los valores en una lista nueva
def merge(izquierda, derecha, key):  1 usage  Sol Rege
    lista_ordenada = []
    i = 0
    j = 0

    # Bucle comparativo entre ambas listas
    while i < len(izquierda) and j < len(derecha):
        if key(izquierda[i]) < key(derecha[j]):
            lista_ordenada.append(izquierda[i])
            i += 1
        else:
            lista_ordenada.append(derecha[j])
            j += 1

    # El while finaliza cuando una de las listas se ha recorrido completamente
    # La otra lista puede aun contener elementos, que serán agregados a continuación en la lista nueva
    lista_ordenada.extend(izquierda[i:])
    lista_ordenada.extend(derecha[j:])

    return lista_ordenada
```

En cambio, *Quick Sort* es conocido por su eficiencia en el uso de memoria y por ofrecer muy buen desempeño en la mayoría de los casos prácticos, especialmente cuando el pivote se elige adecuadamente.

Quick sort

```python
# Funcion de ordenamiento que utiliza un pivote y divida la lista en partes
# Parametro key para definir atributo a ordenar
def quick_sort(lista, key=lambda x: x):  4 usages  Sol Rege
    # Condición para detener la recursión
    if len(lista) <= 1:
        return lista

    # Elección del Pivote
    # Se elige el del medio ya que se considera el mejor caso
    mitad = len(lista) // 2
    pivote = lista[mitad]

    menores = []
    iguales = []
    mayores = []

    for elemento in lista:
        if key(elemento) < key(pivote):
            menores.append(elemento)
        elif key(elemento) == key(pivote):
            iguales.append(elemento)
        else: # key(elemento) > key(pivote)
            mayores.append(elemento)

    # Recursividad y combinación
    return quick_sort(menores, key) + iguales + quick_sort(mayores, key)
```

Funcionamiento del caso práctico

```
--- PRUEBAS DE BÚSQUEDA ---
Autor aleatorio buscado: 'Lisa Whelchel/Stormie Omartian'

--- Búsqueda Lineal ---
Tiempo de ejecución con una busqueda lineal
Autor: Lisa Whelchel/Stormie Omartian. Posición: 1130
Tiempo de búsqueda lineal: 0.000135 segundos


Tiempo de ejecución con una busqueda binaria
Autor: Lisa Whelchel/Stormie Omartian. Posición: 6398
Tiempo de búsqueda binaria: 0.000007 segundos


COMPARACION:
Tiempo de búsqueda lineal: 0.000135 segundos
Tiempo de búsqueda binaria: 0.000007 segundos

--- PRUEBAS DE ORDENAMIENTO ---
Bubble Sort: 14.272626 segundos
Merge Sort : 0.039099 segundos
Quick Sort : 0.045930 segundos

Process finished with exit code 0
```

4. Metodología Utilizada

La primera etapa en la realización del trabajo consistió en una investigación teórica sobre el tema elegido: algoritmos de búsqueda y ordenamiento. Comenzamos consultando la bibliografía proporcionada por la universidad, para luego ampliar la información mediante fuentes lo más confiables posible, como la documentación oficial de Python, artículos técnicos y sitios de referencia como w3schools.com.

En cuanto al trabajo colaborativo, se realizó una videollamada inicial para acordar los lineamientos generales del proyecto y dividir las tareas de forma equitativa. Como complemento a esta dinámica, se creó un documento compartido para trabajar el marco teórico, y un repositorio en GitHub para centralizar el desarrollo del código y mantener el control de versiones.

A lo largo del proceso, a medida que surgieron dificultades o nuevos aspectos a desarrollar, se buscó abordar las tareas de manera equitativa. Si bien no se siguió una planificación estricta, se mantuvo una organización fluida y cooperativa, en donde existió la colaboración activa de ambas partes.

Entre las herramientas empleadas, se destaca el sitio kaggle.com, del cual se descargó el archivo .csv con datos reales de libros, utilizado para simular el inventario. La librería Pandas fue clave para la manipulación de este archivo y la conversión a estructuras de datos adecuadas, como listas de diccionarios.

5.Resultados obtenidos:

El caso práctico sirvió para implementar los algoritmos de búsqueda y ordenamiento más comunes experimentando por primera vez con el uso de un dataset de tamaño mediano extraído de la plataforma keggle.com.

En cuanto a los *algoritmos de búsqueda*, en general no presentaron mayores dificultades, ya que su lógica es comprensible y forma parte de las estructuras trabajadas previamente durante la cursada. Un aspecto nuevo fue la extracción de un dato específico desde un diccionario. Si bien esto no representó un obstáculo, sí se percibe como una complejidad adicional dentro de los algoritmos.

Respecto a las dificultades que surgieron, hubo una confusión al momento de plantear la búsqueda binaria: se asumió erróneamente que los datos ya estaban ordenados, al observar que los ID aparecían numerados de forma ascendente. Se interpretó que eso era suficiente para ordenar los autores, lo cual es incorrecto, ya que la búsqueda se realiza sobre cadenas de texto y, por lo tanto, requiere un ordenamiento alfabético.

Por lo demás, ni la construcción ni la validación presentaron mayores inconvenientes. Fue necesario aplicar el entrecruzamiento de datos, común cuando se trabaja con distintas funciones, para implementar una búsqueda aleatoria que pudiera utilizarse tanto con el método lineal como con el binario. En este último caso, fue importante tener en cuenta el ordenamiento desde el inicio.

En cuanto a la evaluación del rendimiento, los resultados obtenidos fueron coherentes con la complejidad temporal esperada para cada algoritmo. El tiempo de búsqueda lineal fue de 0.001415 segundos, mientras que la búsqueda binaria arrojó un tiempo de 0.000007 segundos. Esta diferencia evidencia la mayor eficiencia de la búsqueda binaria en conjuntos de datos ordenados, ya que reduce significativamente el número de comparaciones necesarias en relación con la búsqueda lineal.

De todas formas es importante mencionar que ambos algoritmos presentan un consumo de memoria bajo —con espacio constante O(1) en sus versiones iterativas—, dado que funcionan mediante comparación y descarte sin requerir estructuras auxiliares.

En relación con los algoritmos de ordenamiento, no se presentaron mayores dificultades, salvo al implementar Merge Sort, cuyo enfoque recursivo implica una estructura menos intuitiva y demanda un mayor esfuerzo cognitivo.

Como era de esperarse, Bubble Sort resultó ser el menos eficiente, con un tiempo de ejecución significativamente mayor que el de los otros dos algoritmos: 9.069011 segundos.

Entre Merge Sort y Quick Sort, las diferencias de rendimiento fueron mínimas. Merge Sort mostró un tiempo de ejecución de 0.024305 segundos, mientras que Quick Sort registró 0.027782 segundos, lo que posiciona a Merge Sort como el de mejor desempeño en este caso.

Según nuestra investigación, esta diferencia podría deberse al mayor consumo de memoria en la implementación de Quick Sort. Por un lado, la elección del pivote no siempre garantiza una partición perfectamente equilibrada. Además, en nuestro código, Quick Sort genera nuevas listas (menores, iguales, mayores) en cada llamada recursiva, lo que implica un mayor gasto de memoria y tiempo de procesamiento. En Python, crear y concatenar listas tiene un costo considerable, por lo que, aunque teóricamente Quick Sort tiene una complejidad promedio similar a la de Merge Sort (O(n log n)), en la práctica puede resultar un poco más lento si no se optimiza adecuadamente.

Por su parte Merge Sort divide la lista en mitades iguales y luego las fusiona ordenadamente. Aunque realiza más operaciones, tiene un comportamiento estable y predecible, y su implementación aprovecha bien la estructura de datos, lo que permite realizar fusiones de manera eficiente.

6. Conclusiones

Este trabajo fue una excelente oportunidad para comprender que el rendimiento de un algoritmo no sólo está determinado por su complejidad temporal promedio, sino también por el uso de recursos como la memoria, la calidad de la implementación y las características del entorno de ejecución. Se pudieron aplicar de forma concreta estos conceptos y comprender cómo distintos factores interactúan en el desempeño de las aplicaciones.

En cuanto a la utilidad de los algoritmos implementados, como se mencionó en la introducción, los tiempos de ejecución son un factor clave en la experiencia del usuario y, por ende, en la captación y retención de potenciales clientes. De allí la importancia de elegir y aplicar el algoritmo adecuado según el contexto.

Una posible extensión futura del proyecto podría consistir en su desarrollo como una aplicación fullstack, donde ambos desarrolladores trabajen de manera integrada para optimizar tanto el backend como el frontend. Esto incluiría mejoras como la compresión de imágenes, la implementación de carga diferida (lazy loading), la eficiencia en funciones, el diseño responsive, y una mejor gestión del rendimiento general de la aplicación.

7. Bibliografía

- Python Software Foundation. (s.f.). *pandas.read_csv — pandas documentation*. pandas. Recuperado de: https://pandas.pydata.org/docs/reference/api/pandas.read_csv.html

- Python Software Foundation. (s.f.). *How to sort*. Python Docs. Recuperado de: https://docs.python.org/es/3.13/howto/sorting.html

- W3Schools. (s.f.). *Bubble Sort Algorithm*. Recuperado de: https://www.w3schools.com/dsa/dsa_algo_bubblesort.php

- W3Schools. (s.f.). *Merge Sort Algorithm*. Recuperado de: https://www.w3schools.com/dsa/dsa_algo_mergesort.php

- W3Schools. (s.f.). *Quick Sort Algorithm*. Recuperado de: https://www.w3schools.com/dsa/dsa_algo_quicksort.php

- Think with Google. (s.f.). *Ya están aquí los rankings de velocidad de los sitios web móviles europeos: ¿Estás a la altura?*. Recuperado de https://www.thinkwithgoogle.com/intl/es-es/estrategias-de-marketing/aplicaciones-y-moviles/ya-est%C3%A1n-aqu%C3%AD-los-rankings-de-velocidad-de-los-sitios-web-m%C3%B3viles-europeos-est%C3%A1s-la-altura/

- Ecommerce News. (s.f.). *El 40% de los usuarios abandonan un sitio web si tarda más de tres segundos en cargar*. Recuperado de: https://ecommerce-news.es/el-40-de-los-usuarios-abandonan-un-sitio-web-si-tarda-mas-de-tres-segundos-en-cargar/

- Shopify. (s.f.). *Website Load Time Statistics*. Recuperado de: https://www.shopify.com/ph/blog/website-load-time-statistics

- *Implementación de algoritmos de búsqueda binaria en Python* [Presentación de clase]. (s.f.). Tecnicatura Universitaria en Programación, Universidad.

8. Anexos

Link al repositorio:

https://github.com/EugeniaDemarchi/trabajo_integrador_programacion_1.git

Link al video en Youtube:

https://youtu.be/p0fY18V4uUg?si=aYKAQm3W0-0VaM7r