ALGORITMOS Y ESTRUCTURAS DE DATOS

Análisis de eficiencia de los algoritmos Merge Sort y Búsqueda Secuencial

---

Integrantes:

Enrique Miguel Solís Sandoval

Joshua Salvador Vílchez Gutiérrez

Camilo Darío Galeano Vargas

Elián Gabriel Olivares Traña

Docente:

Silvia Gigdalia Ticay López

7 de julio del 2025

Índice

I. Introducción

Este trabajo de investigación, realizado para la asignatura *Algoritmos y Estructuras de Datos*, evalúa la eficiencia de los algoritmos Mergesort y búsqueda secuencial en arreglos desordenados, midiendo su tiempo de ejecución y consumo de memoria. La búsqueda secuencial es un método simple que examina cada elemento de un arreglo de forma lineal hasta encontrar el valor deseado o determinar su ausencia, siendo intuitivo pero ineficiente para grandes conjuntos de datos debido a su complejidad temporal O(n) (Knuth, 1998). Por su parte, Merge Sort es un algoritmo de ordenamiento basado en la estrategia de "divide y conquista", que divide el arreglo en subarreglos, los ordena y los combina, logrando una complejidad temporal de O(n log n) incluso en el peor caso (Cormen et al., 2009).

El estudio compara el desempeño de la búsqueda secuencial directa frente a la estrategia de ordenar el arreglo con Mergesort para luego aplicar una búsqueda más eficiente, como la binaria. Implementaremos ambos algoritmos en Python, ejecutaremos pruebas con arreglos de diferentes tamaños y características, y presentaremos los resultados. Respaldados por fuentes académicas como Cormen et al. (2009) y Sedgewick y Wayne (2011), este análisis busca identificar las condiciones óptimas para el uso de estos algoritmos en aplicaciones prácticas.

1. Planteamiento del problema

En el desarrollo de aplicaciones que manejan grandes volúmenes de datos, la eficiencia en la búsqueda y ordenamiento de información es fundamental. El tiempo de ejecución y el uso de recursos computacionales se ven directamente afectados por la elección del algoritmo adecuado para cada tarea. Por ejemplo, cuando se requiere buscar un dato específico en un conjunto de elementos, el tipo de algoritmo utilizado puede hacer la diferencia entre una operación rápida y una que ralentiza el sistema.

Uno de los algoritmos de búsqueda más simples es la búsqueda secuencial, que consiste en recorrer uno a uno los elementos de un arreglo hasta encontrar el objetivo. Aunque

es fácil de implementar y funciona bien con pequeños conjuntos de datos desordenados, su rendimiento disminuye significativamente a medida que el tamaño del arreglo crece.

Por otro lado, para ordenar datos, el algoritmo Merge Sort es reconocido por su eficiencia, ya que mantiene una complejidad de tiempo O(n log n) incluso en el peor de los casos. A diferencia de métodos más simples como el bubble sort, merge sort divide y conquista los datos, permitiendo un ordenamiento más rápido y estable.

Este proyecto busca evaluar y comparar el desempeño de ambos algoritmos en contextos donde se trabaja con arreglos desordenados. Se analizará si ordenar primero el arreglo con Merge Sort antes de aplicar una búsqueda más eficiente (como la binaria) supera en rendimiento a una búsqueda secuencial en el arreglo desordenado. Este análisis permitirá tomar decisiones informadas sobre qué estrategia utilizar en distintas situaciones prácticas, optimizando así el rendimiento de los sistemas.

2. Objetivo de la investigación

Evaluar la eficiencia computacional de los algoritmos Merge Sort y la búsqueda secuencial en arreglos desordenados, determinando su impacto en el tiempo de ejecución y el uso de recursos en aplicaciones que requieran análisis de datos.

3. Objetivos específicos

1. Analizar el funcionamiento teórico de los algoritmos Merge Sort y Búsqueda Secuencial con el fin de contrastar sus resultados esperados con los obtenidos empíricamente.

2. Comparar el consumo de tiempo y memoria de los algoritmos Merge Sort y Búsqueda Secuencial en distintos contextos de aplicación y tamaños de datos.

3. Determinar los contextos más adecuados para la aplicación de los algoritmos Merge Sort y Búsqueda Secuencial, según el tipo de problema, volumen de datos y requerimientos de eficiencia.

II. Metodología

1. Diseño de la investigación

El diseño es de tipo experimental, definido como un método de investigación que manipula variables independientes para observar su efecto sobre variables dependientes en un entorno controlado, permitiendo establecer relaciones causales (Campbell & Stanley, 1966), ya que se manipulan estructuras de datos y se implementan los algoritmos Mergesort y Búsqueda Secuencial con el propósito de evaluar su rendimiento en condiciones controladas. Se busca observar cómo varía el comportamiento de cada algoritmo ante diferentes volúmenes de datos, permitiendo extraer conclusiones objetivas sobre su eficiencia.

2. Enfoque de la investigación

El enfoque adoptado es cuantitativo, dado que se recopilan datos numéricos específicos como el tiempo de ejecución y el consumo de memoria. Estos datos permiten realizar comparaciones objetivas y medibles entre los algoritmos evaluados.

3. Alcance de la investigación

La investigación tiene un alcance descriptivo y explicativo. Es descriptivo porque se detallan las características y funcionamiento de los algoritmos Merge Sort y búsqueda secuencial. Es explicativo porque analiza y justifica el comportamiento de cada algoritmo en relación con los datos de entrada y los resultados obtenidos.

4. Procedimiento

El desarrollo de la investigación seguirá los siguientes pasos:

1. Selección de los algoritmos a evaluar: Mergesort para ordenamiento y búsqueda secuencial en arreglos desordenados. Estos algoritmos se relacionan porque la búsqueda secuencial opera directamente sobre arreglos desordenados con complejidad

O(n), mientras que Mergesort, con complejidad O(n log n), permite ordenar el arreglo para habilitar búsquedas más eficientes, como la binaria (O(log n)), lo que plantea un trade-off entre el costo del ordenamiento y la ganancia en eficiencia de búsqueda (Sedgewick & Wayne, 2011).

2. Codificación e implementación de ambos algoritmos en Python, un lenguaje adecuado por su sintaxis clara, facilidad de implementación de estructuras de datos y bibliotecas como time numpy para generar arreglos de prueba, y tracemalloc, que permiten medir con precisión el tiempo de ejecución y el consumo de memoria (Lutz, 2013).
3. Ejecución de pruebas con arreglos de diferentes tamaños y características (aleatorios, ordenados, parcialmente ordenados).
4. Medición de la eficiencia de cada algoritmo en términos de tiempo de ejecución y uso de memoria.
5. Análisis comparativo de los resultados obtenidos, determinando ventajas, desventajas y condiciones óptimas de uso para cada algoritmo.

III. Marco conceptual / referencial

Marco Conceptual

El presente marco conceptual define los conceptos fundamentales relacionados con la búsqueda secuencial en arreglos desordenados, proporcionando una base teórica para la investigación en el contexto de la asignatura Algoritmos y Estructuras de Datos. A través de un enfoque de vocabulario, se presentan términos clave, sus definiciones, su relevancia en el estudio de algoritmos y su conexión con la eficiencia computacional, respaldados por fuentes académicas reconocidas.

Algoritmo: Conjunto finito de instrucciones bien definidas que, al ejecutarse, resuelve un problema específico en un tiempo finito (Cormen, Leiserson, Rivest, & Stein, 2009). La búsqueda secuencial es un ejemplo de algoritmo que localiza un elemento en un arreglo desordenado, destacando por su simplicidad y utilidad en la enseñanza de los fundamentos de diseño algorítmico.

Búsqueda: Proceso de localizar un elemento específico (o determinar su ausencia) dentro de una colección de datos, como un arreglo, mediante la aplicación de un algoritmo que compara elementos según un criterio definido (Sedgewick & Wayne, 2011). En el contexto de arreglos desordenados, la búsqueda es un problema fundamental que requiere algoritmos como la búsqueda secuencial, cuya eficiencia depende del tamaño y organización de los datos, siendo un componente crítico en aplicaciones de procesamiento de información.

Búsqueda secuencial: Método que examina cada elemento de un arreglo de manera secuencial hasta encontrar el elemento objetivo o determinar su ausencia (Sedgewick & Wayne, 2011). Su aplicabilidad en arreglos desordenados radica en que no requiere preprocesamiento, aunque su complejidad temporal lineal limita su eficiencia en conjuntos grandes.

Arreglo: Estructura de datos lineal que almacena elementos accesibles mediante índices (Cormen et al., 2009). En arreglos desordenados, la falta de un patrón organizativo obliga a algoritmos como la búsqueda secuencial a recorrer todos los elementos, lo que afecta su rendimiento.

Arreglo desordenado: Arreglo cuyos elementos no están organizados según un criterio específico, como orden ascendente o descendente (Sedgewick & Wayne, 2011). Esta característica hace que la búsqueda secuencial sea una técnica adecuada, pero ineficiente para grandes volúmenes de datos.

Orden: Proceso de organizar los elementos de un arreglo según un criterio definido, como en algoritmos de ordenamiento tipo Mergesort con complejidad O(n log n) (Sedgewick & Wayne, 2011). La búsqueda secuencial no requiere orden, a diferencia de la búsqueda binaria, lo que resalta la importancia de evaluar estrategias de preprocesamiento.

Complejidad temporal: Medida del tiempo de ejecución de un algoritmo en función del tamaño de la entrada, expresada en notación Big O (Cormen et al., 2009). La búsqueda secuencial tiene una complejidad temporal de O(n), reflejando su dependencia lineal del tamaño del arreglo.

Complejidad espacial: Cantidad de memoria adicional requerida por un algoritmo (Jain, 2023). La búsqueda secuencial es eficiente con una complejidad espacial de O(1),

utilizando solo memoria constante, lo que la hace atractiva en sistemas con recursos limitados.

Análisis a priori: Estimación teórica del rendimiento de un algoritmo antes de su implementación, basada en el cálculo de operaciones y recursos necesarios (Sedgewick & Wayne, 2011). Para la búsqueda secuencial, este análisis predice un comportamiento lineal en arreglos desordenados.

Análisis a posteriori: Evaluación empírica del rendimiento de un algoritmo mediante pruebas con datos reales (Cormen et al., 2009). Este análisis valida las estimaciones teóricas de la búsqueda secuencial, comparándolas con otras técnicas en diferentes escenarios.

Comparativa de algoritmos: Proceso de evaluar y contrastar el rendimiento de diferentes algoritmos en términos de eficiencia temporal, espacial y aplicabilidad (Jain, 2023). En esta investigación, se compara la búsqueda secuencial con enfoques que involucran ordenamiento previo.

La relevancia de estos conceptos radica en su capacidad para sustentar el estudio de la búsqueda secuencial en arreglos desordenados, un tema fundamental en la educación sobre algoritmos. Su simplicidad permite introducir nociones de eficiencia y diseño algorítmico, mientras que su comparación con métodos más avanzados resalta las compensaciones entre simplicidad y rendimiento. Este marco, respaldado por fuentes como *Introduction to Algorithms* de Cormen et al. (2009), *Algorithms* de Sedgewick y Wayne (2011), y el artículo de Jain (2023), proporciona una base sólida para analizar estrategias de búsqueda en contextos académicos y prácticos.

Este marco conceptual establece los fundamentos teóricos para investigar la búsqueda secuencial, destacando su simplicidad, limitaciones y aplicaciones. Al conectar estos conceptos con el análisis de eficiencia, se sientan las bases para evaluar estrategias de búsqueda en arreglos desordenados, contribuyendo al diseño de sistemas computacionales optimizados.

IV. Implementación del algoritmo

En esta sección se presentan los códigos fuente de los algoritmos búsqueda secuencial y Merge Sort, implementados en Python, junto con una explicación de su funcionamiento y estructura. Estos códigos representan las implementaciones clásicas de los algoritmos, diseñados para operar sobre arreglos desordenados (búsqueda secuencial) o para ordenar arreglos (Mergesort), y sirven como base para su evaluación en términos de eficiencia, la cual se realiza en otras secciones de la investigación (Sedgewick & Wayne, 2011).

Algoritmo Búsqueda Secuencial

```
def busqueda_secuencial(arreglo, valor):
    for i in range(len(arreglo)):
        if arreglo[i] == valor:
            return i
    return -1
```

Funcionamiento y estructura:

La búsqueda secuencial es un algoritmo simple que recorre un arreglo de manera lineal, comparando cada elemento con el valor objetivo hasta encontrarlo o determinar su ausencia. La función busqueda_secuencial recibe un arreglo y un valor a buscar, devolviendo el índice del valor si se encuentra o -1 si no está presente. Su estructura utiliza un bucle for que itera sobre todos los índices del arreglo, lo que resulta en una complejidad temporal de O(n) en el peor caso, donde n es el tamaño del arreglo. Este algoritmo es adecuado para arreglos desordenados debido a su simplicidad y falta de requisitos de preprocesamiento, aunque su eficiencia disminuye en conjuntos de datos grandes (Cormen et al., 2009).

Algoritmo Merge Sort

```
def merge(arr, left, mid, right):
```

```
    n1 = mid - left + 1

    n2 = right - mid

    L = [0] * n1

    R = [0] * n2

    for i in range(n1):

        L[i] = arr[left + i]

    for j in range(n2):

        R[j] = arr[mid + 1 + j]

    i = j = 0

    k = left

    while i < n1 and j < n2:

        if L[i] <= R[j]:

            arr[k] = L[i]

            i += 1

        else:

            arr[k] = R[j]

            j += 1

        k += 1

    while i < n1:

        arr[k] = L[i]

        i += 1
```

```
        k += 1
    while j < n2:
        arr[k] = R[j]
        j += 1
        k += 1

def merge_sort(arr, left, right):
    if left < right:
        mid = (left + right) // 2
        merge_sort(arr, left, mid)
        merge_sort(arr, mid + 1, right)
        merge(arr, left, mid, right)
```

Funcionamiento y estructura:

Merge Sort es un algoritmo de ordenamiento basado en la estrategia de "divide y conquista". La función merge_sort divide recursivamente el arreglo en subarreglos hasta que cada uno contiene un solo elemento, y luego los combina en orden ascendente usando la función merge. Esta última crea dos arreglos temporales (L y R) para almacenar las mitades izquierda y derecha del segmento del arreglo, compara sus elementos y los fusiona en el arreglo original en orden ascendente. La complejidad temporal es O(n log n) en todos los casos, debido a la división logarítmica y la fusión lineal, mientras que la complejidad espacial es O(n) por los arreglos temporales. En esta investigación, Merge Sort se utiliza para ordenar arreglos desordenados, permitiendo la aplicación de búsquedas más eficientes, como la binaria, en lugar de la búsqueda secuencial (Sedgewick & Wayne, 2011).

V. Análisis a Priori

1. Eficiencia espacial

Análisis de la cantidad de memoria que necesita el algoritmo para funcionar, considerando estructuras auxiliares y variables temporales.

Criterio	Merge Sort	Búsqueda Secuencial
Memoria Adicional	O(n) por arreglos temporales en la fusión, más O(log n) por recursión.	O(1). Solo usa variables locales (índice, valor buscado).
Estructuras Auxiliares	Utiliza dos arreglos temporales (izquierda y derecha) en cada nivel de la recursión	No utiliza estructuras auxiliares
Uso de pila	Consume O(log n) en la pila debido a las llamadas recursivas en la división del arreglo.	No usa recursión ni llamadas anidadas por lo que no consume memoria en la pila.

2. Eficiencia temporal

Estimación del número de operaciones o pasos necesarios para completar el algoritmo.

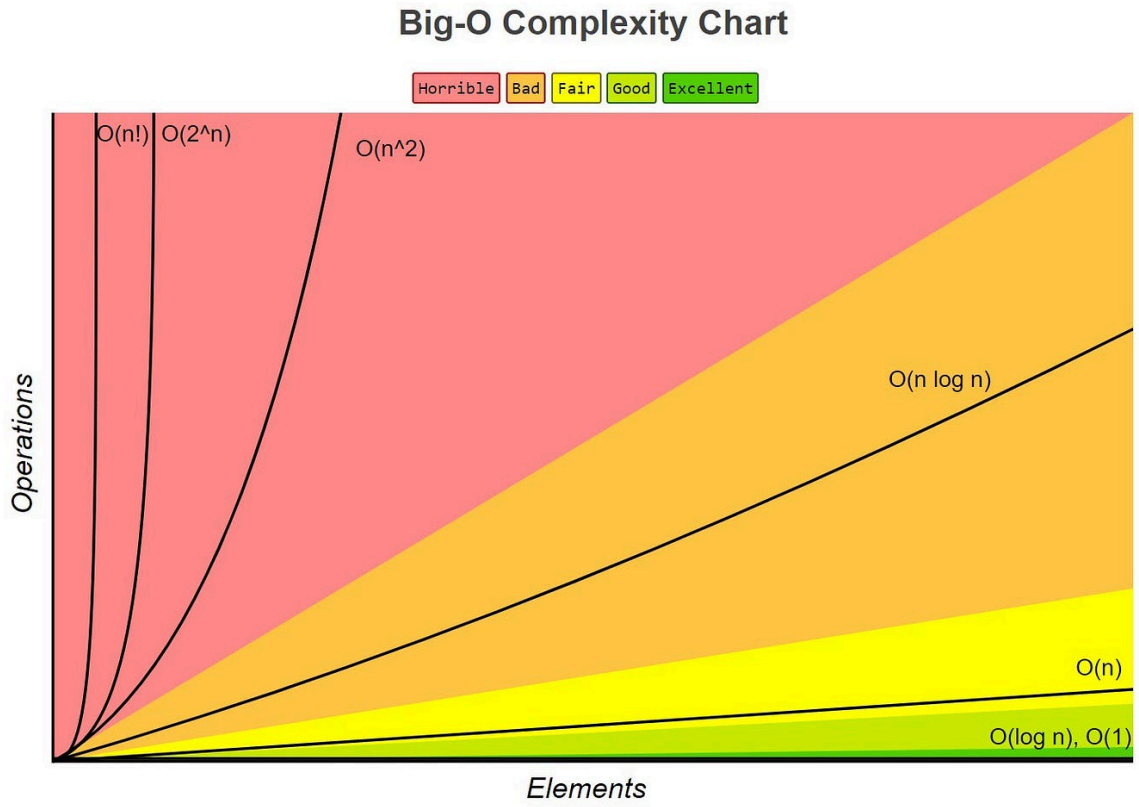
Caso	Merge Sort	Búsqueda Secuencial
Mejor	O(n log n). Realiza ~2n log n comparaciones, independientemente de si el arreglo está ordenado.	O(1) El mejor caso de la búsqueda secuencial es cuando el valor buscado se encuentra en la primera posición del arreglo. Es decir, solo hace 1 comparación.
Promedio	O(n log n) El caso promedio del Merge Sort es cuando el arreglo está ordenado aleatoriamente.Realiza ~2n log n.	El caso promedio de la búsqueda secuencial es O(n), cuando el valor buscado está en una posición aleatoria (uniformemente distribuida) o no está. El número estimado de comparaciones si el valor está es de $(n + 1)/2$, y si no está es de n.
Peor	El peor caso del Merge Sort es O(n log n). Realiza ~2n log n cuando el arreglo está ordenado en orden inverso.	El peor caso de la búsqueda secuencial es O(n), cuando el valor buscado está en la última posición del arreglo o no está en este. El número estimado de comparaciones es de n.

Tamaño del Arreglo (n)	Merge Sort	Búsqueda Secuencial
100	0.05ms	0.01ms
1,000	0.6ms	0.1ms
10,000	8ms	1ms

100,000	100ms	10ms
1,000,000	1,300ms (1.3s)	100ms
10,000,000	16,000ms (16s)	1,000ms (1s)

3. Análisis de Orden

Clasificación en notación Big O del algoritmo: por ejemplo, O(n), O(log n), O(n²), etc.



Algoritmo	Mejor Caso	Caso Promedio	Peor Caso	Orden Final
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Búsqueda Secuencial	$O(1)$	$O(n)$	$O(n)$	$O(n)$

La notación Big O refleja la complejidad en el peor caso, que es el estándar para comparar algoritmos. Merge Sort mantiene O(n log n) debido a su estrategia de división y conquista,

mientras que la búsqueda secuencial es O(n) en el peor caso por su naturaleza lineal. Estas complejidades predicen que Mergesort es más eficiente para ordenar grandes arreglos, pero la búsqueda secuencial es más simple para búsquedas directas en arreglos desordenados (Cormen et al., 2009).

VI. Análisis a Posteriori

Utilizando el código implementado en Python, disponible en el repositorio de GitHub, se evaluó el rendimiento de los algoritmos de búsqueda secuencial y Merge Sort en términos de tiempo de ejecución y consumo de memoria. Las pruebas se realizaron con arreglos desordenados de tamaños 100, 1,000, 10,000 y 100,000 elementos, considerando los casos mejor, promedio y peor para ambos algoritmos. Los resultados se presentan en tablas organizadas por tamaño de arreglo y caso, detallando los tiempos en milisegundos (ms) y el uso de memoria en kilobytes (KB).

1. Análisis del mejor caso

Interpretación:

En el mejor caso, la búsqueda secuencial es extremadamente rápida (0.0000 ms) para todos los tamaños, ya que encuentra el elemento en la primera posición (O(1)), con memoria constante (0.1-0.16 KB). Merge Sort, incluso en su mejor caso (arreglo ordenado), muestra tiempos crecientes (0.9973 ms para n=100, 5385.0074 ms para n=100,000) debido a su complejidad O(n log n), y su memoria crece linealmente (0.16 KB a 781.53 KB). Esto confirma que la búsqueda secuencial es más eficiente para el mejor caso en todos los tamaños.

2. Análisis del caso promedio

Interpretación:

En el caso promedio, la búsqueda secuencial muestra tiempos crecientes (0.0000 ms para $n \leq 1{,}000$, 34.9803 ms para n=100,000), reflejando su complejidad O(n), con memoria constante (0.08-0.14 KB). Merge Sort tiene tiempos mucho mayores (1.0014 ms para n=100,

5029.9959 ms para n=100,000) debido a O(n log n), y su memoria crece linealmente (0.16 KB a 781.53 KB). La búsqueda secuencial es más rápida para tamaños pequeños (n≤1,000).

3. Análisis del peor caso

Interpretación:

En el peor caso, la búsqueda secuencial tiene tiempos crecientes (1.0071 ms para n=1,000, 45.0184 ms para n=100,000) debido a O(n), ya que recorre todo el arreglo, con memoria constante (0.08-0.14 KB). Merge Sort mantiene tiempos altos (22.0089 ms para n=1,000, 5359.5214 ms para n=100,000) por su complejidad O(n log n), con memoria lineal (0.16 KB a 781.53 KB). La búsqueda secuencial es más lenta en el peor caso para tamaños grandes, mientras que Mergesort es consistente pero costoso.

Comparación con la teoría: Los tiempos de búsqueda secuencial (0.0000-45.0184 ms) crecen linealmente con n, validando O(n) en los casos promedio y peor, y O(1) en el mejor caso. Merge Sort, con tiempos de 0.0000-5385.0074 ms, confirma O(n log n) en todos los casos, ya que su rendimiento es independiente del orden inicial. El consumo de memoria constante de búsqueda secuencial (O(1)) y lineal de Merge Sort (O(n)) coincide con las predicciones teóricas.

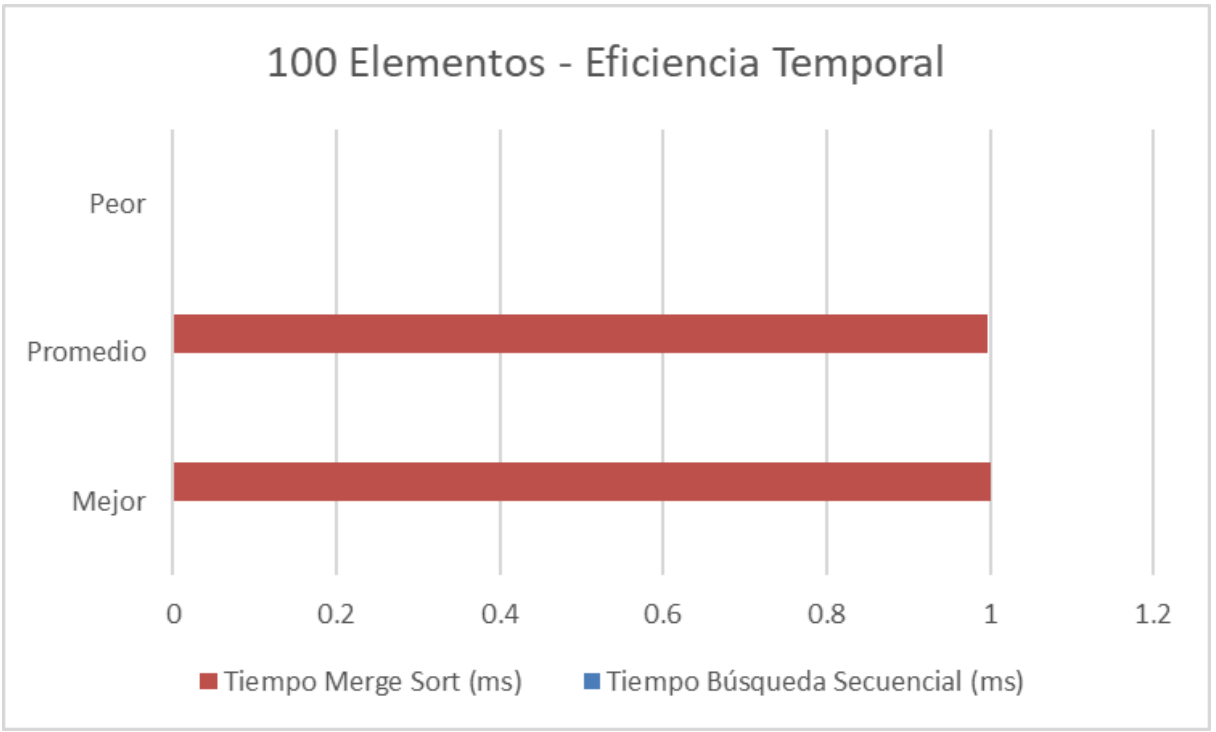
Conclusión práctica: La búsqueda secuencial es ideal para arreglos pequeños (n≤1,000) debido a su rapidez y bajo uso de memoria, especialmente en el mejor caso. Merge Sort, aunque más lento y con mayor consumo de memoria, es adecuado para ordenar arreglos grandes (n≥10,000) cuando el ordenamiento es un paso previo necesario en aplicaciones específicas.

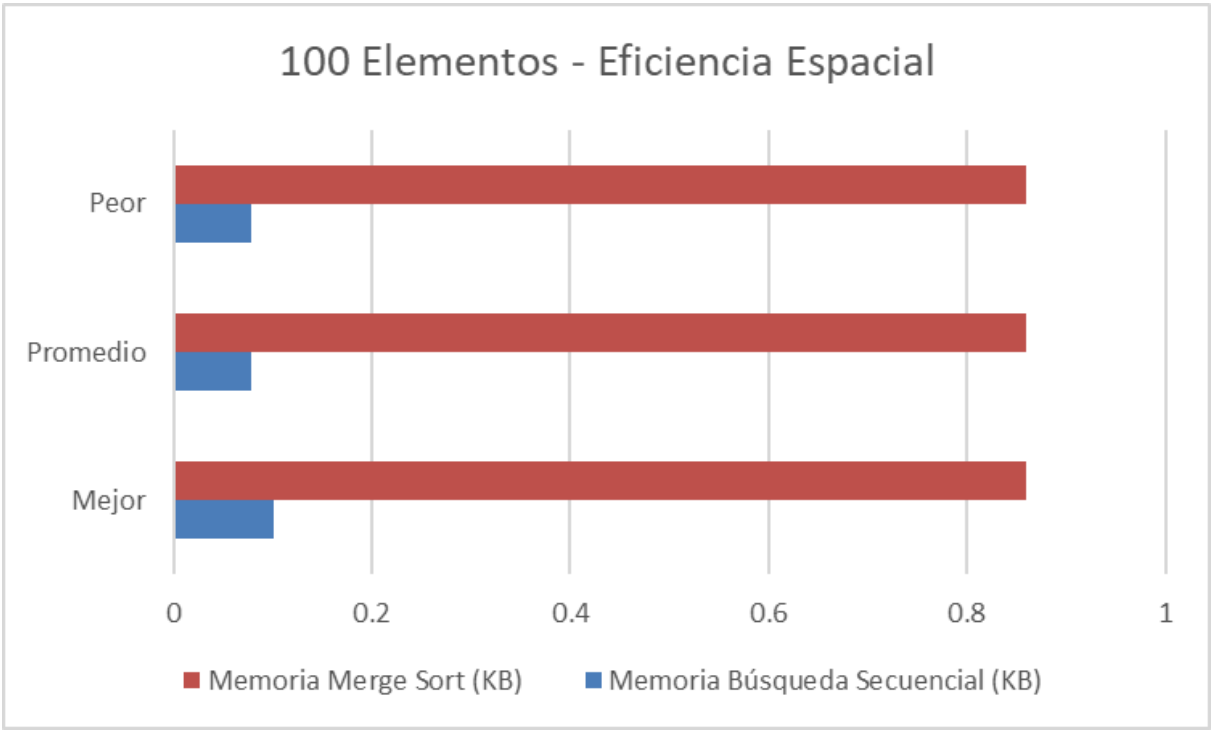
VII. Resultados

Presentación de tablas, gráficas y análisis cuantitativo con datos obtenidos en la experimentación. Utilizando el código anexo en el repositorio de github, el cual evalúa el tiempo y uso de memoria de los algoritmos mergesort y búsqueda secuencial, obtuvimos los siguientes resultados presentados en tablas por cantidad de elementos en el arreglo.

Arreglo de 100 elementos:

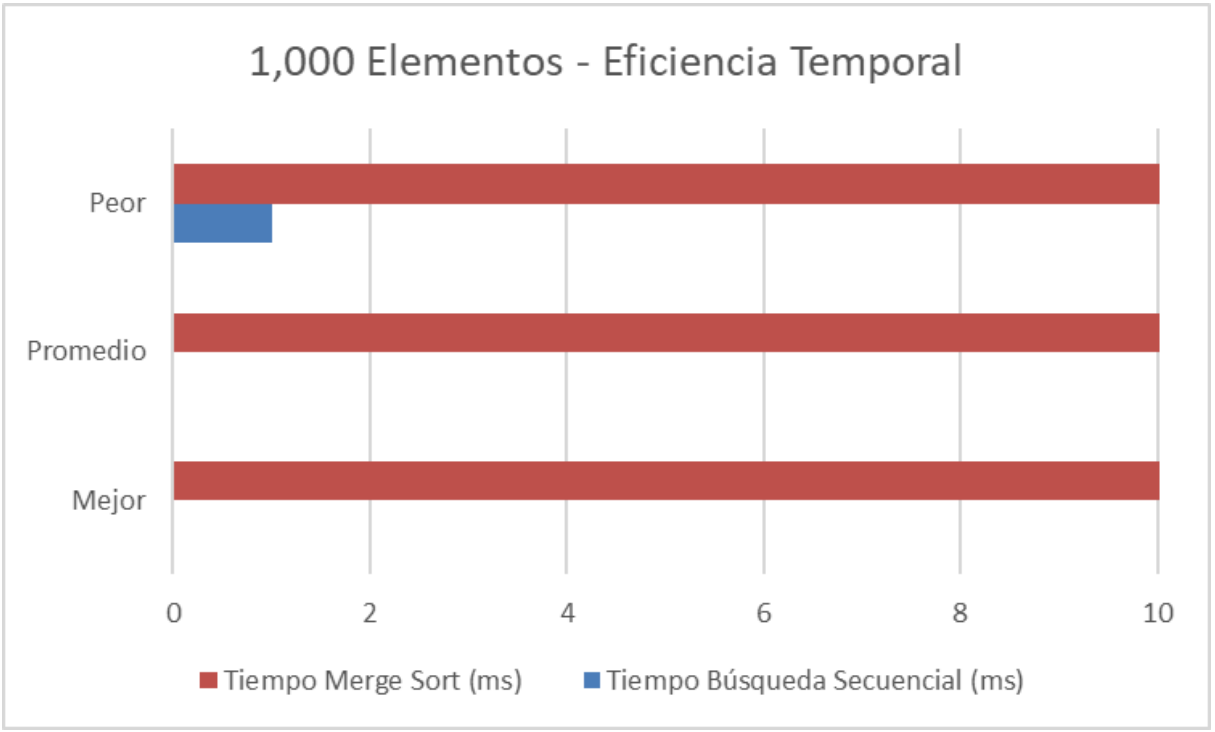
Caso	Tiempo B. S.(ms)	Memoria B. S. (KB)	Tiempo Merge Sort (ms)	Memoria Merge Sort (KB)
Mejor	0.0000	0.1016	0.9973	0.8594
Promedio	0.0000	0.0781	1.0014	0.8594
Peor	0.0000	0.0781	0.0000	0.8594

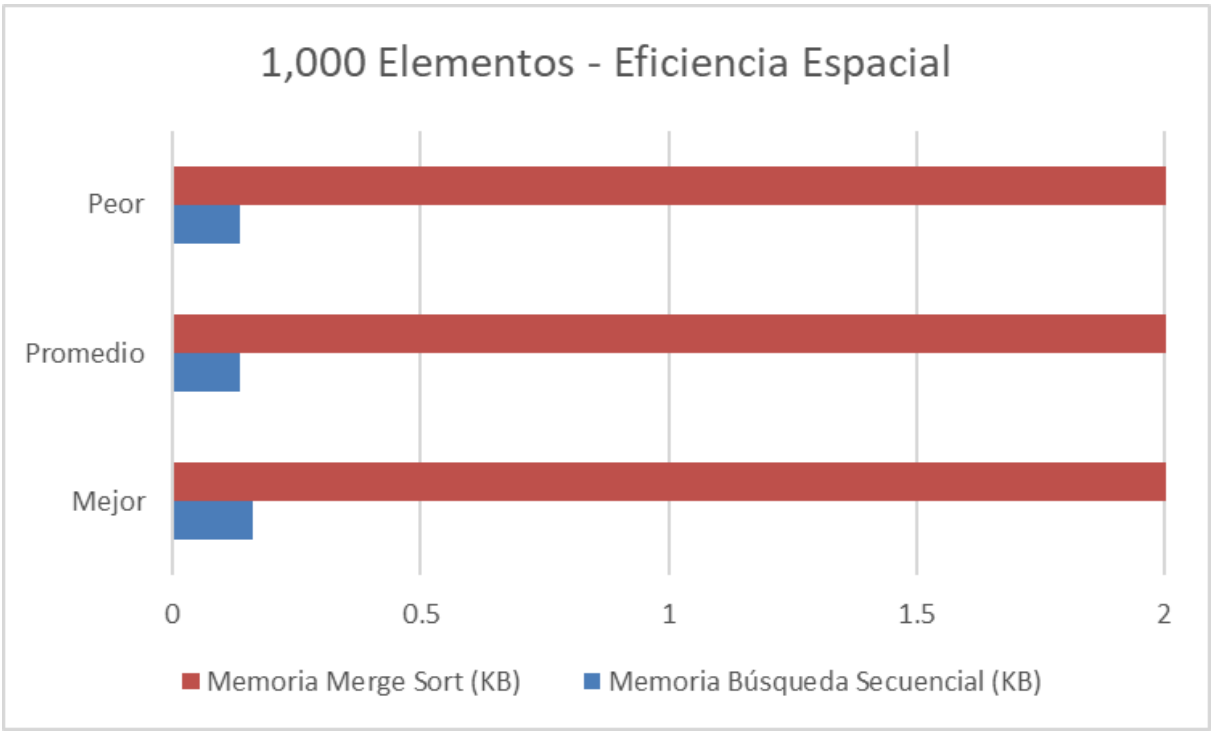




Arreglo de 1,000 elementos:

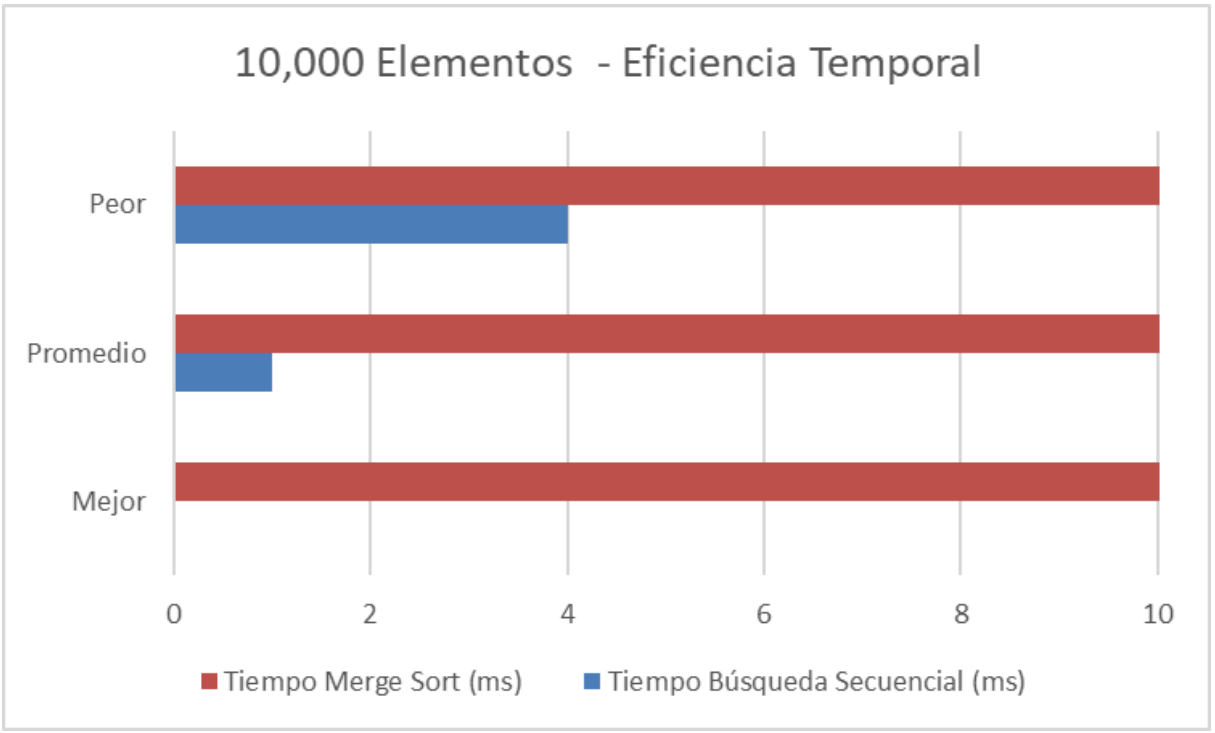
Caso	Tiempo B. S.	Memoria B. S.	Tiempo Merge Sort	Memoria Merge Sort
Mejor	0.00ms	0.1602kb	25.001ms	8.0938kb
Promedio	0.00ms	0.1367kb	24.019ms	8.0938kb
Peor	1.0071ms	0.1367kb	22.008ms	8.0938kb

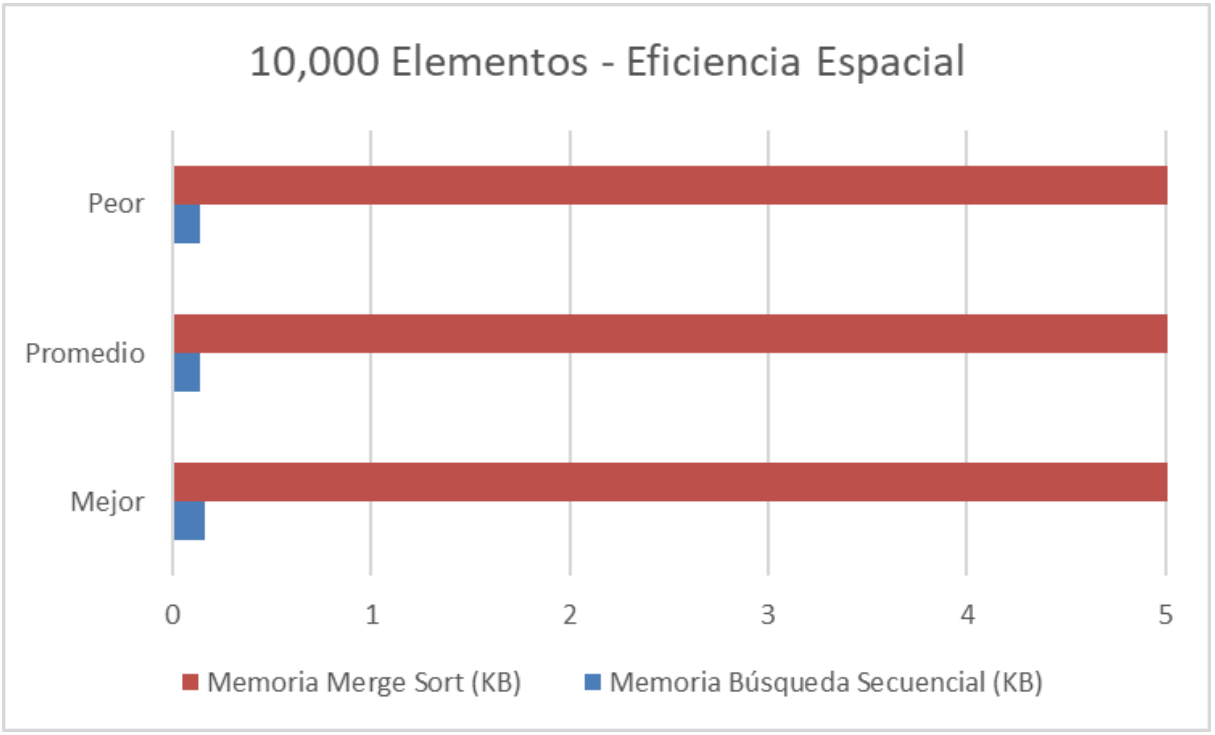

1,000 Elementos - Eficiencia Temporal
Peor
Promedio
Mejor
0
2
4
6
8
10
Tiempo Merge Sort (ms)
Tiempo Búsqueda Secuencial (ms)


1,000 Elementos - Eficiencia Espacial
Peor
Promedio
Mejor
0
0.5
1
1.5
2
Memoria Merge Sort (KB)
Memoria Búsqueda Secuencial (KB)

Arreglo de 10,000 elementos:

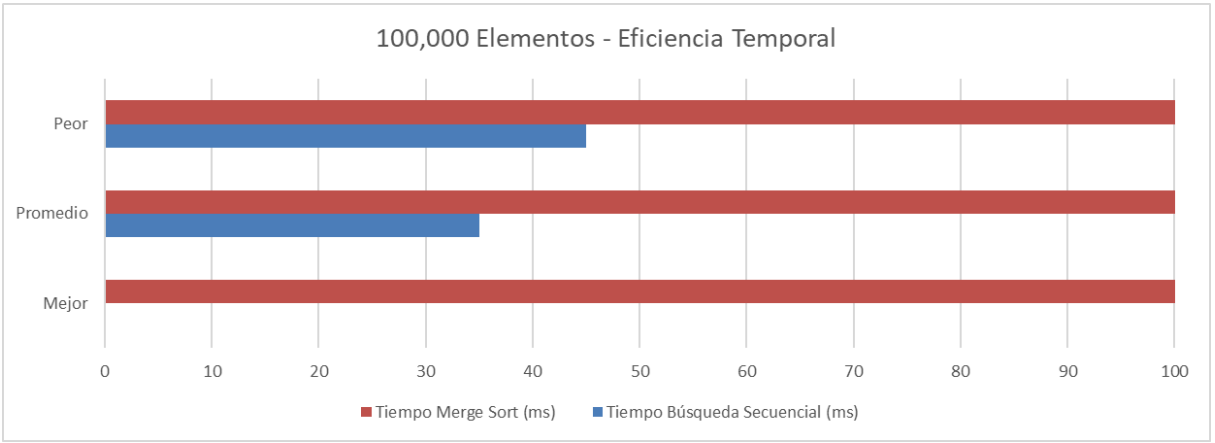
Caso	Tiempo BS	Memoria BS	Tiempo Merge Sort	Memoria Merge Sort
Mejor	0.00ms	0.1602kb	376.0021ms	78.4062kb
Promedio	0.9944ms	0.1367kb	390.9979ms	78.4062kb
Peor	4.0042ms	0.1367kb	384.0227ms	78.4062kb

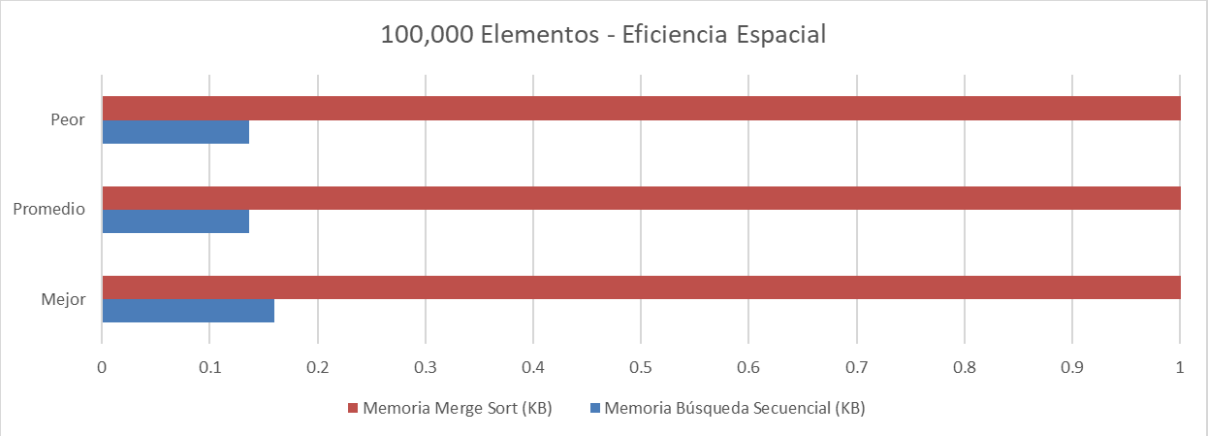




Arreglo de 100,000 elementos:

Caso	Tiempo BS	Memoria BS	Tiempo Merge Sort	Memoria Merge Sort
Mejor	0.00ms	0.1602kb	5029.9959ms	781.5312kb
Promedio	34.9803ms	0.1367kb	5385.0074ms	781.5312kb
Peor	45.0184ms	0.1367kb	5359.5214ms	781.5312kb





VIII. Conclusiones

Reflexiones sobre cuál o cuáles algoritmos son más eficientes y en qué contexto. Se destacan los hallazgos más importantes y posibles recomendaciones de uso.

Tras analizar el rendimiento de los algoritmos de búsqueda secuencial y Merge Sort en arreglos de 100, 1,000, 10,000 y 100,000 elementos, se derivan conclusiones sobre su eficiencia y aplicabilidad. La búsqueda secuencial destaca por su bajo consumo de memoria (0.0781-0.1602 KB en todos los casos) y rapidez en el mejor caso (0.0000 ms para todos los tamaños), pero su tiempo en el peor caso crece linealmente (hasta 45.0184 ms para n=100,000), confirmando su complejidad O(n). Esto la hace ideal para arreglos pequeños ($n \leq 1{,}000$) o consultas únicas donde la simplicidad es prioritaria.

Merge Sort, con una complejidad O(n log n), muestra tiempos consistentes (0.0000-5385.0074 ms) en todos los casos, pero requiere memoria auxiliar significativa (0.8594-781.5312 KB para n=100,000). Su estabilidad y escalabilidad lo hacen adecuado para ordenar arreglos grandes ($n \geq 10{,}000$) cuando el ordenamiento es un paso previo necesario.

Los resultados obtenidos cumplen el objetivo de evaluar la eficiencia de la búsqueda secuencial en arreglos desordenados frente a Merge Sort. La búsqueda secuencial es más eficiente en tiempo y memoria para arreglos pequeños ($n \leq 1{,}000$), mientras que Merge Sort sobresale en ordenar arreglos grandes, validando que el tamaño del arreglo determina la elección del algoritmo.

La búsqueda secuencial es óptima para tamaños pequeños, mientras que Merge Sort es superior para ordenar grandes volúmenes de datos. Es debido a esto, que la búsqueda

secuencial se debería utilizar para arreglos pequeños o sistemas con recursos limitados, y Merge Sort para aplicaciones que requieren ordenamiento eficiente en arreglos grandes.Sin embargo, ambos algoritmos tienen fortalezas bien definidas, la búsqueda secuencial destaca por su sencillez y bajo uso de recursos, mientras que Merge Sort sobresale en rendimiento escalable y consistencia. La elección entre uno u otro debe basarse en el tamaño del problema, el contexto de uso y los requisitos de eficiencia.

IX. Referencias bibliográficas

Jain, S. (2023). *Sequential Search vs Binary Search: Implementation and Complexity*. SSRN Electronic Journal. https://doi.org/10.2139/ssrn.4451861

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (3rd ed.). MIT Press. https://archive.org/details/introductiontoal00corm

Addison-Wesley. Sedgewick, R., & Wayne, K. (2011). Algorithms (4th ed.).

Sedgewick, R., & Wayne, K. (2011). *Algorithms* (4th ed.). Addison-Wesley. https://books.google.com/books/about/Algorithms.html?id=NLngYyWFl_YC

Addison-Wesley. Jain, S. (2023). Sequential Search vs Binary Search: Implementation and Complexity. SSRN Electronic Journal. https://doi.org/10.2139/ssrn.4451861

Campbell, D. T., & Stanley, J. C. (1966). *Experimental and quasi-experimental designs for research*. Rand McNally. (Original work published 1963). https://archive.org/details/experimentalquas00camp

Lutz, M. (2013). *Learning Python* (5th ed.). O'Reilly Media. https://archive.org/details/learningpython0000lutz

Sedgewick, R., & Wayne, K. (2011). *Algorithms* (4th ed.). Addison-Wesley. https://books.google.com/books/about/Algorithms.html?id=NLngYyWFl_YC