Analisis de Algoritmos

J.F. Cázares Marroquin

Tabla de contenidos

1 Prefacio

Introducción

El análisis de algoritmos es una herramienta esencial para comprender cómo se comportan distintas soluciones computacionales frente a diferentes tipos de entrada. Este conocimiento permite detectar el problema algorítmico detrás de una situación concreta y, con ello, elegir o construir una solución que sea eficiente en cuanto a uso de tiempo y recursos. Implementar una solución adecuada puede traducirse en menores costos operativos o en la posibilidad de trabajar con grandes volúmenes de datos de forma ágil.

Dominar el diseño, la implementación y la evaluación de algoritmos es clave para desarrollar un buen juicio como científico de datos. A lo largo del curso se han trabajado fundamentos que fortalecen la capacidad de resolver problemas reales con un enfoque en eficiencia y escalabilidad, partiendo de modelos de cómputo y estructuras bien definidas.

En estos reportes se documentan ejercicios tanto teóricos como prácticos sobre diversos algoritmos y estructuras de datos, con énfasis en la experimentación, implementación propia y análisis crítico de los resultados. Al finalizar, se espera que quien lea este material tenga una visión clara para seleccionar, adaptar e implementar algoritmos que optimicen recursos como memoria y tiempo de procesamiento, incluso en sistemas complejos a partir de bloques básicos bien elegidos.

1.1 Contenido del sitio

Este sitio reúne los reportes desarrollados durante el curso "Análisis de Algoritmos" de la Maestría en Ciencia de Datos e Información (INFOTEC, 2025), los cuales están organizados por tema. Cada reporte aborda un conjunto específico de algoritmos, desde estructuras lineales y métodos de ordenamiento, hasta algoritmos avanzados para búsqueda e intersección de conjuntos. En particular:

- El **Reporte 1** introduce el enfoque experimental y el análisis de tiempo sobre estructuras simples.
- El **Reporte 2** aborda las estructuras de datos y su representación computacional.
- En el **Reporte 3** se analizan distintos algoritmos de ordenamiento en el modelo de comparación.
- El **Reporte 4** trata algoritmos de búsqueda secuencial y binaria, comparando su desempeño práctico.
- Finalmente, el **Reporte 5** se enfoca en algoritmos de intersección de listas y conjuntos, relevantes en sistemas de recuperación de información y bases de datos a gran escala.

2 1A Reporte escrito. Experimentos y análisis

3 Introducción

El análisis de algoritmos constituye un área fundamental en la informática, ya que permite comprender y predecir el rendimiento de soluciones computacionales en diferentes contextos. En este trabajo se estudiarán los principales órdenes de crecimiento que determinan la eficiencia de los algoritmos, contrastando tanto su impacto teórico como sus implicaciones prácticas a través de simulaciones y análisis visual. Este enfoque proporciona las herramientas necesarias para evaluar el costo computacional de las operaciones en función del tamaño de la entrada, un aspecto clave en el diseño y optimización de soluciones informáticas.

El crecimiento asintótico, como se detalla en Introduction to Algorithms [@cormen2022], permite clasificar los algoritmos en términos de su rendimiento al trabajar con entradas de gran escala. La notación asintótica—incluyendo O, Θ y Ω —es crucial para modelar estos comportamientos, ya que proporciona una descripción simplificada pero robusta del tiempo de ejecución y uso de recursos, independientemente de factores secundarios como la arquitectura de hardware o el lenguaje de programación utilizado.

Además, la relevancia de este análisis se extiende al impacto práctico que los algoritmos tienen en sistemas modernos. Según [@sedgewick2011], los algoritmos son fundamentales en diversas disciplinas, desde la inteligencia artificial hasta la computación gráfica, y su eficiencia puede ser el factor decisivo en el éxito o fracaso de una aplicación. Por su parte,[@kleinberg2006] enfatizan que el análisis de algoritmos no solo implica evaluar su comportamiento en el peor caso, sino también en promedio y en el mejor caso, lo

que proporciona una perspectiva más completa sobre su rendimiento en diferentes escenarios.

A lo largo de este documento se abordan las siguientes comparaciones de los órdenes de crecimiento:

a. $O(1)$ vs $O(logn)$
b. $O(n)$ vs $O(n \log n)$
c. $O(n^2)$ vs $O(n^3)$
d. $O(a^n)$ vs $O(n!)$
e. $O(n!)$ vs $O(n^n)$

Estos casos permiten identificar, tanto en la teoría como en la práctica, cómo diferentes algoritmos escalan con el tamaño de los datos de entrada. Según Cormen et al. (2022), este tipo de análisis no solo facilita una comparación objetiva entre alternativas algorítmicas, sino que también guía el diseño de soluciones eficientes en aplicaciones reales.

En términos aplicados, el análisis experimental complementa el teórico al proporcionar una validación práctica del comportamiento de los algoritmos bajo condiciones específicas. Las simulaciones incluidas en este trabajo se realizaron utilizando herramientas computacionales modernas que permiten graficar el rendimiento de los órdenes de crecimiento y calcular tiempos de ejecución simulados. Este enfoque, además de ser didáctico, pone en evidencia los límites prácticos de ciertos algoritmos, especialmente aquellos con un crecimiento exponencial o factorial, donde las operaciones se vuelven computacionalmente inviables incluso para tamaños moderados de entrada.

En síntesis, este trabajo no solo contextualiza los fundamentos teóricos del análisis de algoritmos, sino que también los aplica y analiza mediante experimentación computacional, logrando sintetizar las características más relevantes de cada orden de crecimiento. Este enfoque integral permite al lector comprender cómo los algoritmos pueden ser diseñados, seleccionados y evaluados para resolver problemas reales con una gestión óptima de los recursos computacionales.

4 Código utilizado

4.1 Importación de librerías

Se utilizará el lenguaje de programación Python para la demostración de los órdenes de crecimiento, para los requerimientos se importarán las librerías matplotlib para graficar los resultados, numpy para las operaciones matemáticas y pandas para el manejo de los datos.

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import math

plt.rcParams.update({
    "text.usetex": False,  # No usar LaTeX
    "font.family": "serif",  # Usa una fuente serif general
    "font.serif": "Times New Roman",  # Alternativa compatible
    "font.size": 12,
    "axes.titlesize": 14,
    "axes.labelsize": 12,
    "xtick.labelsize": 10,
    "ytick.labelsize": 10,
    "legend.fontsize": 10,
    "figure.figsize": (6, 4),
    "grid.alpha": 0.6,
    "grid.linestyle": "--",
})
```

4.2 Funciones de órdenes de crecimiento

Para mejor manejo del código se definirá una funcion para cada orden de crecimiento, como se muestra en el siguiente código:

```
def o1(n):
    return np.ones_like(n)

def ologn(n):
    return np.log(n)

def on(n):
    return n

def onlogn(n):
    return n * np.log(n)

def on2(n):
    return n ** 2

def on3(n):
    return n ** 3

def a_n(n, a=2):
    return a ** n

def nfact(n):
    return np.array([math.factorial(int(x)) for x in n])

def nn(n):
    return n ** n  # O(n^n)
```

4.3 Eleccion del rango de valores de n

La teoría nos indica que los órdenes de crecimiento constante ($O(1)$) o logarítmico ($O(logn)$) son simples y de bajo costo, por lo que se crea un diccionario donde para el rango seleccionado para los órdenes de crecimiento simples es un rango aproximado de $n = 1000$ y conforme el orden de crecimiento se vuelva mas complejo el rango de n disminuye hasta $n = 10$.

```
ranges = {
    "O(1) vs O(log n)": np.arange(1, 1001),
    "O(n) vs O(n log n)": np.arange(1, 1001),
    "O(n^2) vs O(n^3)": np.arange(1, 101),
    "O(a^n) vs O(n!)": np.arange(1, 16),
    "O(n!) vs O(n^n)": np.arange(1, 11)
}
comparisons = [
    ("O(1)", o1, "O(log n)", ologn),
    ("O(n)", on, "O(n log n)", onlogn),
    ("O(n^2)", on2, "O(n^3)", on3),
    ("O(2^n)", lambda n: a_n(n, 2), "O(n!)", nfact),
    ("O(n!)", nfact, "O(n^n)", nn)
]
```

4.4 Generación de gráficas

Para el caso de las gráficas se utilizará la librería de matplotlib, para generar las gráficas de cada comparación de orden de crecimiento y su respectivo análisis.

```
import os
os.makedirs("images", exist_ok=True)
```

```
for (label1, func1, label2, func2), range_key in zip(comparisons, ranges.keys
    n_values = ranges[range_key]
    y1 = func1(n_values)
    y2 = func2(n_values)

    safe_name = range_key.replace(' ', '_').replace('/', '_').replace('^', '

    plt.figure(figsize=(10, 6))
    plt.plot(n_values, y1, label=label1, linewidth=2)
    plt.plot(n_values, y2, label=label2, linewidth=2)
    plt.xlabel("n")
    plt.ylabel("Orden de crecimiento")
    plt.title(range_key)
    plt.legend()
    plt.grid(True)
    plt.savefig(f"images/{safe_name}.png")
    plt.close()

nanosecond = 1e-9


tamanos = [10,20,30,100, 1000, 10000, 100000]

tiempo_simulado = pd.DataFrame({
    "Tamano": pd.Series(dtype=int),
    "O(1)": pd.Series(dtype=float),
    "O(logn)": pd.Series(dtype=float),
    "O(n)": pd.Series(dtype=float),
    "O(nlogn)": pd.Series(dtype=float),
    "O(n^2)": pd.Series(dtype=float),
```

```
    "O(n^3)": pd.Series(dtype=float),
    "O(a^n)": pd.Series(dtype=object),
    "O(n!)": pd.Series(dtype=object),
    "O(n^n)": pd.Series(dtype=object)
})

for n in tamanos:
    tiempo_simulado = pd.concat([
        tiempo_simulado,
        pd.DataFrame({
            "Tamano": [n],
            "O(1)": [nanosecond],
            "O(logn)": [nanosecond * np.log(n)],
            "O(n)": [nanosecond * n],
            "O(nlogn)": [nanosecond * n * np.log(n)],
            "O(n^2)": [nanosecond * n**2],
            "O(n^3)": [nanosecond * n**3],
            "O(a^n)": [nanosecond * 2**n  if n<1000 else "Overflow" ],
            "O(n!)": [nanosecond * math.factorial(n) if n<31 else "Overflow"],
            "O(n^n)": [nanosecond * n**n if n<31 else "Overflow"],
        })
    ], ignore_index=True)


pd.set_option("display.max_rows", None)
pd.set_option("display.max_columns", None)
tiempo_simulado.columns = [
    col.replace("^", r"\^").replace("_", r"\_") for col in tiempo_simulado.columns
]
latex_code = r"""
\begin{table}[H]
\centering
\resizebox{0.85\textwidth}{!}{%
""" + tiempo_simulado.to_latex(index=False, escape=False) + r"""
```

```
}
\caption{Tiempos simulados para diferentes ordenes de crecimiento.}
\label{tab:tiempos_simulados}
\end{table}
"""


with open("tabla_tiempos.tex", "w") as f:
    f.write(latex_code)
```

5 Resultados y Discusión

A continuación se presentan las gráficas obtenidas para cada comparación de órdenes de crecimiento, acompañadas de un análisis correspondiente.

5.1 $O(1)$ vs $O(\log n)$

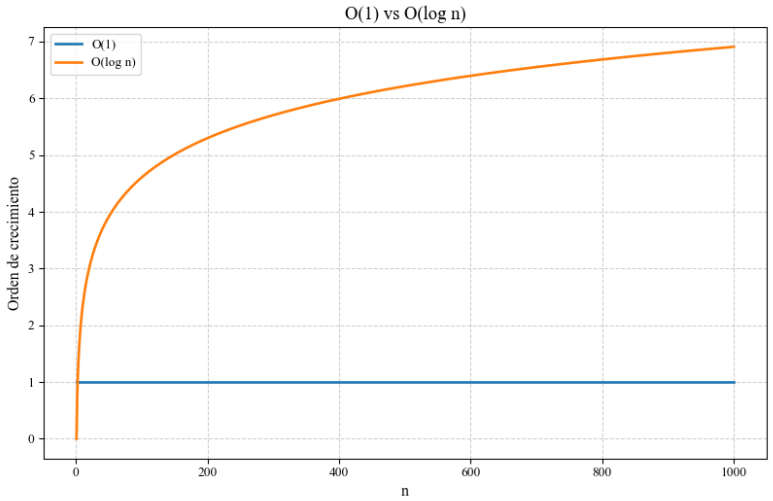

Figura 5.1: Comparación $O(1)$ vs $O(\log n)$

En la Figura 1, se observa cómo el orden constante $O(1)$ se mantiene invariable independientemente del tamaño de entrada, mientras que ($O(\log n)$) crece lentamente a medida que aumenta el tamaño de la entrada, un ejemplo de una función que tenga un orden de crecimiento logarítmico podría ser la búsqueda binaria en una lista ordenada. El orden de crecimiento constante denota que no importa el tamaño de la entrada; el tiempo de ejecución es el mismo, un ejemplo de esto podria ser el acceder al primer elemento de una lista.

5.2 $O(n)$ vs $O(n \log n)$

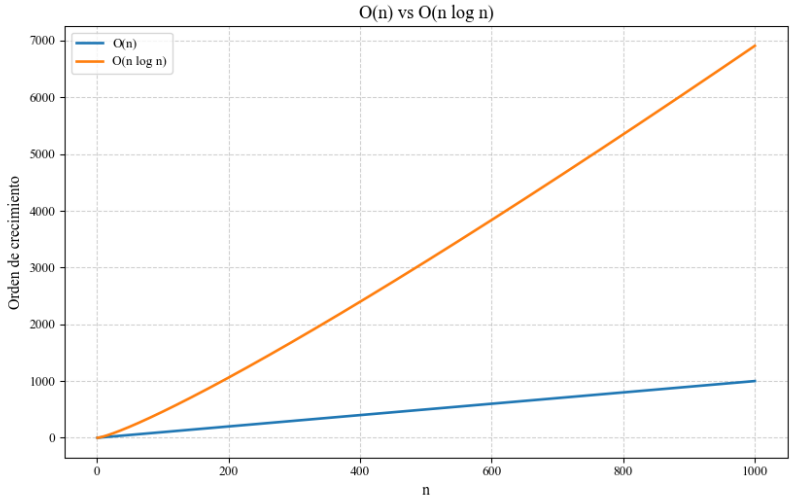

Figura 5.2: Comparación $O(n)$ vs $O(n \log n)$

En la Figura 2 se compara el crecimiento lineal ($O(n)$) con el crecimiento lineal logarítmico ($O(n \log n)$), en el primero el tiempo de ejecución crece

proporcionalmente al tamaño de la entrada, mientras que el crecimiento lineal logarítmico a entradas más grandes el tiempo de ejecución aumenta varias veces en comparación con el crecimiento lineal. Por dar ejemplos de cada uno un algoritmo de crecimiento lineal puede ser un algoritmo de búsqueda que recorre una lista para encontrar un elemento, mientras que un algoritmo de crecimiento lineal logarítmico es común en algoritmos de ordenamiento, como Merge Sort.

5.3 $O(n^2)$ vs $O(n^3)$

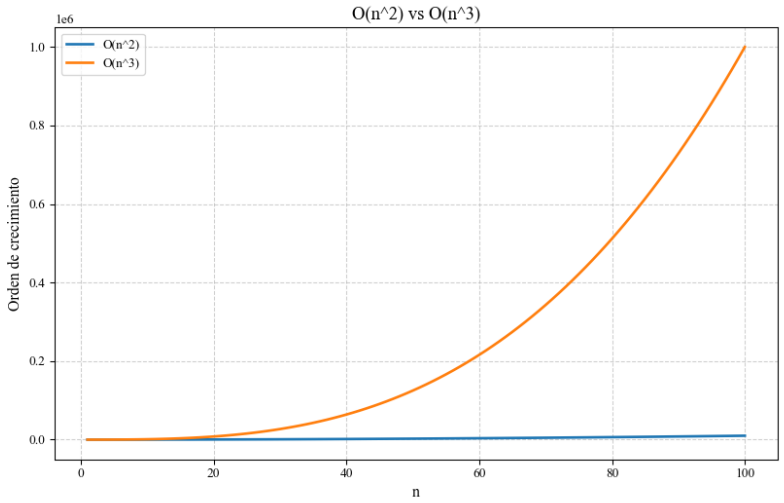

Figura 5.3: Comparación $O(n^2)$ vs $O(n^3)$

La comparación del crecimiento cuadrático y el crecimiento cúbico se observan en la figura 3. Según sea el caso, el tiempo de ejecucion crece con

el cuadrado o el cubo del tamaño de la entrada. Siendo que el crecimiento cúbico alcanza mayores tiempos de ejecución con menores tamaños de entrada. El crecimiento cuadrático se puede observar en algoritmos de ordenamiento simples como Bubble Sort, mientras que el crecimiento cúbico es común en algoritmos que involucran múltiples bucles anidados procesando una matriz tridimensional.

5.4 $O(a^n)$ vs $O(n!)$

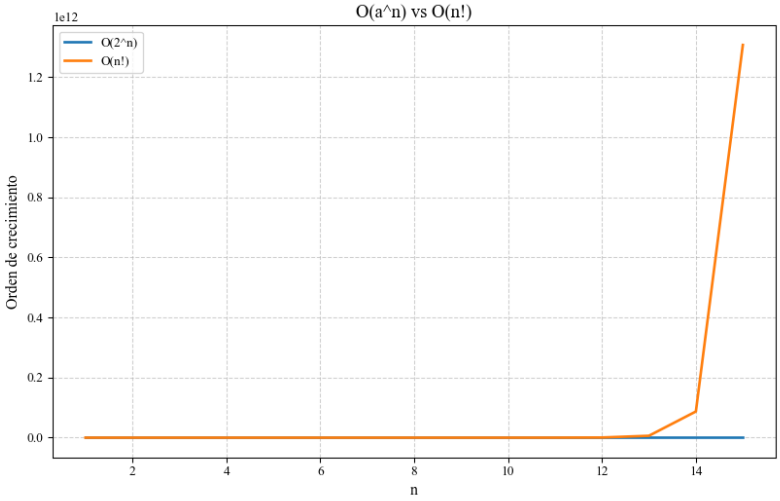

Figura 5.4: Comparación $O(a^n)$ vs $O(n!)$

En el caso de la comparación del crecimiento exponencial con base a y el crecimiento factorial. Los algoritmos de crecimiento factorial son más rapidos que los de crecimiento exponencial ya que, ambos crecen rápidamente aunque con pequeños tamaños de de entrada se ve una clara aceleración de

crecimiento de parte del algoritmo factorial. Ambos crecen rápidamente, y llegan al punto de que el tiempo de ejecución aumenta tanto que los algoritmos se vuelven imprácticos por su alto tiempo de ejecución, para el crecimiento exponencial con base a se puede decir que si $n > 20$ el tiempo de ejecución se vuelve impráctico para muchas aplicaciones, mientras que para el crecimiento factorial que crece más rápido despúes del punto que n>10 se vuelve impráctico, tal como se aprecia en la gráfica que con un tamaño de entrada $n > 13$ el tiempo de ejecución esta en el orden de $x10^{12}$. Un ejemplo de crecimiento exponencial con base a son los algoritmos recursivos que exploran árboles de decisión, los cuales tienen un patrón que generan todas las combinaciones posibles de elementos, mientras que los algoritmos con crecimiento factorial es común en problemas de permutación donde se considera cada orden posible de los elementos.

5.5 $O(n!)$ vs $O(n^n)$

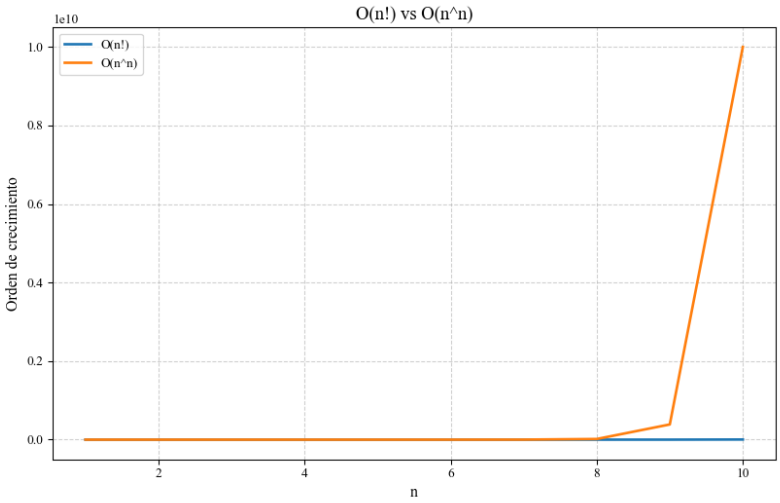

Figura 5.5: Comparación $O(n!)$ vs $O(n^n)$

Finalmente, el crecimiento exponencial aumenta extremadamente rápido, y de igual manera se vuelve impráctico para muchas aplicaciónes con tamaños de entrada pequeños siendo para $n > 8$ que el tiempo de ejecución se vuelve muy grande, esto es más rápido que el crecimiento factorial analizado en la Figura 5. Dentro de lo que cabe es un patrón de crecimiento raro que ocurre cuando cada elemento de una entrada puede tomar n posibles valores y las combinaciones de todos los elementos deben evaluarse

5.6 Tabla de tiempos simulados

La tabla presentada muestra los tiempos simulados para algoritmos con diferentes órdenes de crecimiento ($O(1)$, $O(logn)$, $O(n)$, $O(log(n))$, $O(n^2)$, $O(n^3)$, $O(a^n)$, $O(n!)$, $O(n^n)$) en función del tamaño de entrada (n). Los valores se expresan en nanosegundos, simulando el costo computacional con una operación básica que toma 1 ns (10^-9 s).

Tamano	O(1)	O(logn)	O(n)	O(nlogn)	O(n2)	O(n3)	O(añ)	O(n!)	O(nñ)
10	0.000000	0.000000	0.000000	0.000000	0.000000	0.000001	0.000001	0.003629	10.000000
20	0.000000	0.000000	0.000000	0.000000	0.000000	0.000008	0.001049	2432902008.176640	104857600000000000.000000
30	0.000000	0.000000	0.000000	0.000000	0.000001	0.000027	1.073742	265252859812191097847808.000000	205891132094649888071971638707486720.000000
100	0.000000	0.000000	0.000000	0.000000	0.000010	0.001000	1267650600228229480448.000000	Overflow	Overflow
1000	0.000000	0.000000	0.000001	0.000007	0.001000	1.000000	Overflow	Overflow	Overflow
10000	0.000000	0.000000	0.000010	0.000092	0.100000	1000.000000	Overflow	Overflow	Overflow
100000	0.000000	0.000000	0.000100	0.001151	10.000000	1000000.000000	Overflow	Overflow	Overflow

Tabla 5.1: Tiempos simulados para diferentes ordenes de crecimiento.

De la tabla 1 se puede concluir lo siguiente:

1. $O(1)$ y $O(\log\ n)$ son los órdenes de crecimiento más eficientes, ya que el tiempo de ejecución es constante para el caso de $O(1)$ o crece muy lentamente a medida que n aumenta. Esto hace que ambos sean ideales para problemas que requieren alta eficiencia y escalabilidad, incluso con grandes volúmenes de datos.

2. $O(n)$ y $O(n\ \log\ n)$ aunque son menos eficientes que los órdenes de crecimiento anteriores, estos siguen siendo adecuados para problemas prácticos en aplicaciones de procesamiento de datos algoritmos de búsqueda ordenamiento.

3. Los órdenes de crecimiento polinomiales $O(n^2)$ y $O(n^3)$ siguen siendo manejables pero limitados a tamaños de entrada pequeños o moderados, y su uso en las aplicaciones prácticas debe evaluarse con cuidado.

4. Los órdenes de crecimiento exponenciales y factoriales ($O(a^n)$, $O(n!)$, $O(n^n)$) son inaplicables para problemas con tamaños de entrada moderados a grandes, y su uso se limita a casos extremadamente pequeños o análisis teóricos. Ya que el orden de crecimiento $O(a^n)$ para un tamaño de

100 se obtiene un numero del orden de $1x10^{22}$ ns ($1x10^{13}$ s lo equivalente a 317,000 años). Para el orden de crecimiento $O(n!)$, desde el tamaño de $n = 30$ se obtiene un tiempo de ejecución en el orden de $1X10^{15}$ segundos (equivalente a 31 millones de años), y finalmente para el orden de crecimiento de $O(n^n)$ el tiempo de ejecución con un tamaño de n=30 es del orden de los $1X10^{25}$ segundos, lo que equivale a 317,000,000 de eones.

6 Conclusiones

1. Órdenes altamente eficientes ($O(1)$ y $O(\log\ n)$): Estos órdenes destacan por su eficiencia y bajo impacto en el costo computacional, incluso con entradas grandes. $O(1)$ permite tiempos constantes, como en el acceso a estructuras indexadas, mientras que $O(\log\ n)$), característico de búsquedas binarias, crece lentamente a medida que aumenta el tamaño de entrada, como señalan [@cormen2022a] y [@goodrich2014]. Su escalabilidad los convierte en las mejores opciones para sistemas que manejan grandes volúmenes de datos, como bases de datos o algoritmos de búsqueda en tiempo real.
2. Órdenes prácticos ($O(n)$ y $O(\text{n log } n)$): Estos órdenes muestran un crecimiento significativo en el tiempo de ejecución conforme aumenta n. $O(n^2)$ es típico en algoritmos simples como Bubble Sort, mientras que $O(n^3)$ se encuentra en problemas como la multiplicación de matrices estándar. Como destacan [@kleinberg2006a] y [@dasgupta2006], estos algoritmos son útiles para problemas educativos o de pequeña escala, pero su aplicabilidad en sistemas con grandes volúmenes de datos es limitada.
3. Órdenes polinomiales ($O(n^2)$ y $O(n^3)$): Aunque estos órdenes son significativamente más lentos, pueden ser útiles en problemas de escala moderada o cuando no existen alternativas más eficientes. Sin embargo, su crecimiento rápido limita su aplicabilidad en escenarios con datos masivos.
4. Órdenes exponenciales y factoriales ($O(a^n)$, $O(n!)$ y $O(n^n)$): Estos órdenes tienen un crecimiento extremadamente rápido que los hace imprácticos incluso para tamaños de entrada pequeños. En aplicaciones reales, no son fáctibles para volúmenes de datos moderados

a grandes y se limitan a análisis teóricos o casos donde n es muy reducido.

5. Casos extremos y variables asintóticamente grandes: Los algoritmos con órdenes como $O(n!)$ y $O(n^n)$ enfrentan barreras físicas y computacionales debido al tiempo y recursos requeridos. Estas complejidades resaltan la importancia de recurrir a enfoques alternativos, como heurísticas o aproximaciones, en problemas complejos donde el tamaño de la entrada crece exponencialmente.
6. Simulación y análisis práctico: Las simulaciones realizadas muestran que, si bien los órdenes bajos y medios son útiles y escalables, los órdenes altos y extremos (exponenciales y factoriales) no son prácticos para aplicaciones computacionales. Esto subraya la necesidad de elegir algoritmos con un análisis previo de su comportamiento teórico y práctico.
7. Conclusión general: El análisis y las simulaciones confirman que la elección del algoritmo depende directamente de su orden de crecimiento, especialmente en escenarios con grandes volúmenes de datos. Como destacan [@dasgupta2006], diseñar algoritmos eficientes implica priorizar órdenes bajos ($O(1)$, $O(\log n)$) y evitar órdenes altos en problemas de escala, donde los recursos computacionales son limitados.

6.1 Lista de Cambios

- Se corrigieron fallas errores ortográficos
- Se corrifieron errores de palabras repetidas o redundantes
- Se corrigieron referencias a figuras y tablas dentro del documento.

7 Bibliografía

8 2A Reporte escrito. Experimentos y análisis de estructuras de datos

9 Introducción

El análisis experimental de algoritmos en el manejo de matrices es fundamental en la ciencia de datos, la ingeniería y la computación científica. Las matrices son estructuras de datos esenciales utilizadas en numerosos campos como el modelado de sistemas físicos, el procesamiento de imágenes, la inteligencia artificial y la computación gráfica. En este trabajo, se analizan dos algoritmos ampliamente utilizados en operaciones matriciales: la multiplicación de matrices y la eliminación gaussiana (incluyendo Gauss-Jordan). Estos algoritmos son cruciales para la solución de sistemas de ecuaciones lineales, la optimización de cálculos y el procesamiento de grandes volúmenes de datos en entornos computacionales avanzados [@golub2013].

La multiplicación de matrices es una operación matemática fundamental con aplicaciones en diversas áreas, desde la estadística multivariada hasta la simulación numérica. Su eficiencia es clave, ya que los cálculos matriciales pueden volverse computacionalmente costosos a medida que aumenta el tamaño de las matrices involucradas [@strang2016]. Por su parte, la eliminación de Gauss-Jordan es un método empleado para resolver sistemas de ecuaciones lineales y encontrar la inversa de matrices, lo que lo hace útil en cálculos de dinámica estructural, redes eléctricas y análisis de datos en ingeniería [@trefethen1997].

Uno de los principales desafíos en el uso de estos algoritmos es su complejidad computacional, ya que la cantidad de operaciones requeridas crece significativamente con el tamaño de las matrices. Para evaluar el rendimiento de estos algoritmos, se debe considerar tanto el tiempo de ejecución como el número de operaciones realizadas, incluyendo sumas y multiplicaciones

escalares. La eficiencia de estos cálculos no solo depende del algoritmo en sí, sino también del acceso a los datos en memoria. El acceso secuencial a elementos contiguos mejora significativamente la velocidad de ejecución al aprovechar la localidad espacial en caché del procesador, optimizando el uso de los niveles de memoria [@patterson2020].

Además, este trabajo analiza las implicaciones de utilizar matrices dispersas en lugar de matrices densas. En problemas reales, muchas matrices contienen una gran cantidad de ceros, lo que motiva el uso de representaciones optimizadas que almacenan solo los valores no nulos. Esto puede reducir significativamente el consumo de memoria y mejorar la eficiencia computacional. Sin embargo, el uso de matrices dispersas introduce ciertos desafíos, como una mayor complejidad en el acceso a elementos individuales y posibles aumentos en el tiempo de ejecución debido a estructuras de datos más elaboradas [@davis2006].

El objetivo principal de este estudio es evaluar el rendimiento de la multiplicación de matrices y la eliminación de Gauss-Jordan en términos de tiempo de ejecución y cantidad de operaciones requeridas, utilizando matrices aleatorias de tamaños n=100,300,1000. Se realizarán comparaciones entre estos algoritmos y se analizará el impacto del acceso a elementos contiguos en memoria, así como los costos y beneficios de utilizar matrices dispersas en diferentes escenarios computacionales [@higham2002].

En los próximos apartados, se presentarán los fundamentos teóricos de estos algoritmos, la metodología empleada para su análisis, los experimentos realizados y los resultados obtenidos, seguidos de una discusión sobre su eficiencia en la práctica y posibles optimizaciones futuras.

10 Metodología

Las operaciones analizadas en este reporte son:

1. **Multiplicación de matrices**: Se implementa un algoritmo de multiplicación de matrices basado en la definición matemática tradicional.
2. **Eliminación de Gauss-Jordan**: Algoritmo utilizado para resolver sistemas de ecuaciones y encontrar inversas de matrices.

Los algoritmos se evaluarán sobre matrices de tamaño n x n con valores aleatorios para (n = 100, 300, 1000). Se medirán separadamente el tiempo de ejecución y el número de operaciones realizadas.

11 Código utilizado

```
import numpy as np
import time
import pandas as pd
import matplotlib.pyplot as plt


def multiplicar_matrices(A, B):
    filas_A, columnas_A = A.shape
    filas_B, columnas_B = B.shape
    if columnas_A != filas_B:
        raise ValueError("Las matrices no se pueden multiplicar")

    C = np.zeros((filas_A, columnas_B))
    operaciones = 0

    for i in range(filas_A):
        for j in range(columnas_B):
            for k in range(filas_B):
                C[i, j] += A[i, k] * B[k, j]
                operaciones += 2

    return C, operaciones

def eliminacion_gaussiana(A):
    n = A.shape[0]
    U = A.copy()
```

```
    operaciones = 0

    for i in range(n):
        if U[i, i] == 0:
            for j in range(i+1, n):
                if U[j, i] != 0:
                    U[[i, j]] = U[[j, i]]
                    break

        pivote = U[i, i]
        if pivote == 0:
            continue

        for j in range(i+1, n):
            if U[j, i] != 0:
                factor = U[j, i] // pivote
                for k in range(i, n):
                    U[j, k] -= factor * U[i, k]
                    operaciones += 2

    return U, operaciones

def evaluar_algoritmos():
    tamanos = [100, 300, 1000]
    resultados = []

    for n in tamanos:

      A = np.random.randint(-10, 10, size=(n, n))
      B = np.random.randint(-10, 10, size=(n, n))

      inicio = time.time()
      _, ops_mult = multiplicar_matrices(A, B)
      tiempo_mult = time.time() - inicio
```

```
        inicio = time.time()
        _, ops_gauss = eliminacion_gaussiana(A)
        tiempo_gauss = time.time() - inicio

        resultados.append((n, tiempo_mult, ops_mult, tiempo_gauss, ops_gauss))


    return resultados

resultados = evaluar_algoritmos()

df = pd.DataFrame(resultados, columns=["Tamano", "Tiempo Multiplicacion",
"Operaciones Multiplicacion","Tiempo Gauss", "Operaciones Gauss"])

latex_table = df.to_latex()

latex_table = """
\\begin{table}[ht]
\\caption{Resultados de tiempos de ejecucion y operaciones}
\\label{tab:results}
\\centering
""" + latex_table + """

\\end{table}
"""
with open('tabla_resultados.tex', 'w') as f:  # Guardar en un archivo .tex
    f.write(latex_table)
plt.figure(figsize=(12, 6))


plt.plot(df["Tamano"], df["Tiempo Multiplicacion"], 'o-',
label="Multiplicacion")
plt.plot(df["Tamano"], df["Tiempo Gauss"], 's-', label="Gauss")
```

```
plt.xlabel("Tamaño de matriz")
plt.ylabel("Tiempo (s)")
plt.legend()
plt.grid(True)
plt.savefig("comparacion_algoritmos_tiempo.png")

plt.figure(figsize=(12, 6))
plt.plot(df["Tamano"], df["Operaciones Multiplicacion"], 'o-',
label="Multiplicación")
plt.plot(df["Tamano"], df["Operaciones Gauss"], 's-', label="Gauss")
plt.xlabel("Tamaño de matriz")
plt.ylabel("Operaciones")
plt.yscale('log')
plt.legend()
plt.grid(True)
plt.savefig("comparacion_algoritmos_ops.png")
```

12 Analisis de resultados

En la tabla 19.2 se puede apreciar como conforme el tamaño de la matriz aumenta el tiempo de ejecución y las operaciones van creciendo exponencialmente.

Tabla 12.1: Resultados de tiempos de ejecucion y operaciones

	Tamano	Tiempo Multiplicacion	Operaciones Multiplicacion	Tiempo Gauss	Operaciones Gauss
0	100	1.243188	2000000	0.287769	662052
1	300	37.067283	54000000	7.862818	17948116
2	1000	1378.784946	2000000000	349.595479	665973536

En la siguiente figura se muestra la comparación del tiempo de ejecución entre la eliminación gaussiana y la multiplicación de matrices. El tiempo de ejecución de la multiplicación de matrices aumenta exponencialmente con una matriz de tamaño n=1000. Este ultimo tiempo de ejecución fue de alrededor de 15 minutos para la matriz de tamaño n=1000.

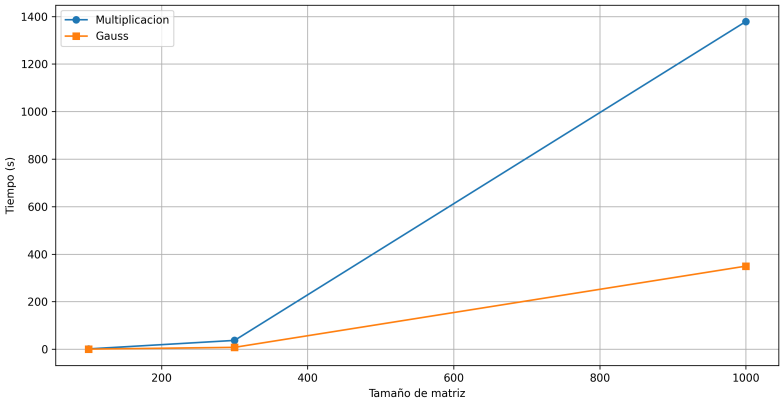

Figura 12.1: Tiempos de Ejecución de Algoritmos

En la siguiente figura se muestra la comparación del Número de operaciones entre la eliminación gaussiana y la multiplicación de matrices. La multiplicación de matrices tiene un numero mayor de operaciones, aunque la tendencia de operaciones es similar entre los dos algoritmos.

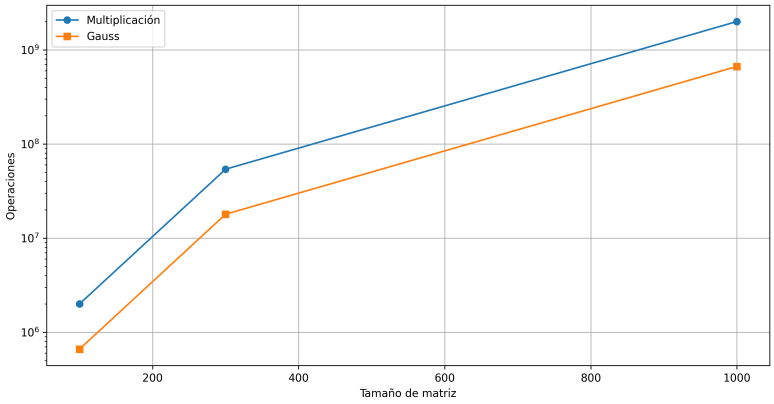

Figura 12.2: Número de operaciones de Algoritmos

13 Conclusión

Los experimentos realizados demuestran que la eficiencia de los algoritmos para operar con matrices depende tanto de la operación específica como del tamaño de las matrices involucradas. En el caso de la multiplicación de matrices y la eliminación de Gauss-Jordan, se observó un comportamiento diferenciado al variar el tamaño de la matriz.

Para tamaños pequeños (n=100), la eliminación de Gauss-Jordan presentó un tiempo de ejecución menor y un número de operaciones inferior en comparación con la multiplicación de matrices. Sin embargo, al aumentar n a 1000, ambos algoritmos mostraron un incremento considerable en tiempo y operaciones, aunque la eliminación de Gauss-Jordan se mantuvo más eficiente. Esto se debe a que su estructura evita las redundancias de cálculo que ocurren en la multiplicación de matrices estándar, reduciendo la cantidad de multiplicaciones y sumas necesarias [@golub2013a].

En términos de complejidad computacional, la multiplicación de matrices tiene una complejidad teórica de $O(n^3)$ cuando se implementa en su forma más básica, lo que la hace poco eficiente para tamaños grandes. Alternativamente, existen métodos más avanzados como la multiplicación de Strassen $O(n^2.81)$ o algoritmos basados en el uso de GPUs que pueden reducir significativamente el tiempo de ejecución [@cormen2009]. Por otro lado, la eliminación de Gauss-Jordan sigue una estructura que también es $O(n^3)$, pero en la práctica suele ser más eficiente que la multiplicación directa de matrices cuando se trata de resolver sistemas de ecuaciones lineales en una única matriz aumentada.

13.1 Impacto del acceso contiguo en memoria

El acceso a elementos contiguos en una matriz mejora el uso de la memoria caché y reduce la latencia de acceso a datos, optimizando el rendimiento del algoritmo. En arquitecturas modernas, el tiempo de acceso a memoria es un factor crítico en la eficiencia de los cálculos matriciales. Cuando un algoritmo accede a datos de manera secuencial y alineada con la memoria caché, el procesador puede predecir y optimizar las lecturas, reduciendo los tiempos de espera y acelerando la ejecución [@patterson2020].

En el caso de la multiplicación de matrices, el acceso a los elementos de forma no contigua en la memoria puede generar una penalización de rendimiento debido a los *cache misses*, especialmente en implementaciones ingenuas. Técnicas como el *loop tiling* o el uso de *column-major ordering* en ciertos lenguajes pueden ayudar a optimizar este comportamiento, permitiendo que los algoritmos se beneficien del uso eficiente de la caché [@strang2016].

13.2 Consideraciones sobre matrices dispersas

El uso de matrices dispersas podría reducir significativamente el almacenamiento y mejorar la eficiencia computacional en ciertos casos. Sin embargo, esto introduce desafíos como:

- **Mayor complejidad de implementación**: Se requiere una representación especial para matrices dispersas, como el formato Compressed Sparse Row (CSR) o Compressed Sparse Column (CSC), lo que puede complicar su manipulación en comparación con matrices densas [@davis2006].
- **Acceso más costoso a elementos individuales**: Dependiendo de la estructura de almacenamiento, el acceso a elementos en una matriz dispersa puede ser más lento que en una matriz densa, ya

que se requiere recorrer índices y punteros en lugar de realizar un acceso directo en memoria. Esto afecta el rendimiento en aplicaciones que requieren accesos aleatorios frecuentes [@higham2002].

- **Incremento en la densidad debido a operaciones**: Algunas transformaciones, como la eliminación gaussiana, pueden llenar la matriz con valores no nulos que antes eran ceros, reduciendo la eficiencia esperada. Este fenómeno, conocido como *fill-in*, puede aumentar significativamente la memoria utilizada y el tiempo de cómputo, especialmente en problemas de álgebra lineal aplicada [@trefethen1997].

A pesar de estos desafíos, el uso de matrices dispersas es clave en aplicaciones como la solución de sistemas de ecuaciones diferenciales parciales, simulaciones físicas y análisis de redes, donde la estructura de los datos hace que el almacenamiento en forma dispersa sea más eficiente que en una matriz densa [@saad2003].

La elección del algoritmo adecuado y la representación de la matriz dependen de la naturaleza del problema y los recursos computacionales disponibles. Para tamaños de matrices moderados, la eliminación de Gauss-Jordan es más eficiente que la multiplicación de matrices tradicional, mientras que para tamaños grandes es crucial emplear optimizaciones como métodos iterativos o paralelización en hardware especializado.

El impacto del acceso contiguo en memoria es significativo y puede mejorar el rendimiento de los algoritmos si se optimiza adecuadamente el acceso a datos en caché. En escenarios donde se emplean matrices dispersas, se debe evaluar el compromiso entre el ahorro de memoria y el costo adicional en operaciones de acceso y manipulación de los datos.

En aplicaciones del mundo real, la elección entre matrices densas y dispersas, así como entre distintos métodos de factorización, depende en gran medida del problema que se está resolviendo. La investigación futura podría enfocarse en el uso de hardware especializado, como GPUs o arquitec-

turas de cómputo de alto rendimiento, para mejorar la eficiencia de estos cálculos en escenarios de gran escala [@patterson2020].

13.3 Lista de Cambios}

- Sin modificaciones

14 Bibliografía

15 3A. Reporte escrito. Experimentos y análisis de algoritmos de ordenamiento.

16 Introducción

El análisis experimental de algoritmos de ordenamiento es fundamental en la computación y la ciencia de datos. Estos algoritmos son utilizados en diversas aplicaciones, como bases de datos, análisis de grandes volúmenes de datos y sistemas operativos. En este trabajo, se analizan cinco algoritmos de ordenamiento basados en comparación: Heapsort, Mergesort, Quicksort, Bubblesort y Skiplist. Estos algoritmos presentan diferentes complejidades computacionales y eficiencia en distintos escenarios [@cormen2009].

El ordenamiento es una operación esencial en la manipulación de datos, ya que permite la búsqueda eficiente, la optimización de estructuras de datos y la mejora en el rendimiento de consultas en bases de datos. La eficiencia de estos algoritmos varía dependiendo de factores como la distribución inicial de los datos y el acceso a memoria. Mientras que algoritmos como Mergesort son eficientes y estables, requieren memoria adicional, lo que los hace menos adecuados para sistemas con restricciones de memoria. Por otro lado, Quicksort es rápido en la práctica, pero su rendimiento puede degradarse en ciertos casos [@knuth1998].

Uno de los principales desafíos en el uso de estos algoritmos es su complejidad computacional, ya que el número de comparaciones y movimientos de elementos varía considerablemente entre ellos. Evaluar su rendimiento implica medir tanto el tiempo de ejecución como la cantidad de comparaciones realizadas. Además, el uso de estructuras de datos como Skiplist introduce una perspectiva diferente sobre la eficiencia en el ordenamiento, ya que utiliza niveles de listas enlazadas para mejorar la velocidad de búsqueda y manipulación de datos [@pugh1990].

El objetivo principal de este estudio es evaluar el rendimiento de estos algoritmos de ordenamiento en términos de tiempo de ejecución y número de comparaciones, utilizando conjuntos de datos con diferentes niveles de desorden. Se analizarán las ventajas y desventajas de cada algoritmo en función de su eficiencia en distintos escenarios computacionales [@sedgewick2011].

17 Metodología

Los algoritmos analizados en este reporte son:

1. **Heapsort**: Algoritmo basado en estructuras de montículos (heaps) que ordena extrayendo elementos del heap máximo, el orden de crecimiento de Heapsort es de O(n log n)
2. **Mergesort**: Algoritmo de ordenamiento basado en dividir y conquistar, que divide el conjunto en mitades y luego las combina ordenadamente. Igual que Heapsort tiene un nivel de complejidad de O (n log n).
3. **Quicksort**: Algoritmo recursivo que selecciona un pivote y divide el conjunto en dos partes, ordenándolas de manera independiente. Es uno de los agloritmos más rapidos en la práctica y su orden de crecimiento en promedio tambien es de O(n log n)
4. **Bubblesort**: Algoritmo simple pero ineficiente que intercambia elementos adyacentes repetidamente hasta que la lista está ordenada. Su orden de crecimiento es de O(n^2)
5. **Skiplist Sort**: Algoritmo basado en la estructura de Skiplist, que permite ordenamiento eficiente aprovechando múltiples niveles de listas enlazadas.

Los algoritmos se evaluarán utilizando diferentes conjuntos de datos provenientes de archivos JSON, los cuales contienen listas con distintos niveles de desorden. Para la evaluación de los algoritmos se medirán separadamente el tiempo de ejecución y el número de comparaciones realizadas para cada lista con su distinto nivel de desorden.

18 Código utilizado

```
import time
import json
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import random



def heapsort(arr):
    comparisons = [0]
    def heapify(arr, n, i):
        largest = i
        l = 2 * i + 1
        r = 2 * i + 2
        if l < n and arr[l] > arr[largest]:
            largest = l
            comparisons[0]+=1
        if r < n and arr[r] > arr[largest]:
            largest = r
            comparisons[0]+=1
        if largest != i:
            arr[i], arr[largest] = arr[largest], arr[i]
            heapify(arr, n, largest)
    n = len(arr)
    for i in range(n // 2 - 1, -1, -1):
```

```
        heapify(arr, n, i)
    for i in range(n - 1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i]
        heapify(arr, i, 0)
    return comparisons[0]

def mergesort(arr):
    comparisons = [0]

    def merge(l, m, r):
        L = arr[l:m+1]
        R = arr[m+1:r+1]
        i = j = 0
        k = l
        while i < len(L) and j < len(R):
            comparisons[0] += 1
            if L[i] <= R[j]:
                arr[k] = L[i]
                i += 1
            else:
                arr[k] = R[j]
                j += 1
            k += 1
        while i < len(L):
            arr[k] = L[i]
            i += 1
            k += 1
        while j < len(R):
            arr[k] = R[j]
            j += 1
            k += 1

    def mergesort_rec(l, r):
        if l < r:
```

```
            m = (l + r) // 2
            mergesort_rec(l, m)
            mergesort_rec(m + 1, r)
            merge(l, m, r)

    mergesort_rec(0, len(arr) - 1)
    return comparisons[0]

def quicksort(arr):
    comparisons = [0]

    def partition(low, high):
        pivot = arr[high]  # último elemento como pivote
        i = low
        for j in range(low, high):
            comparisons[0] += 1
            if arr[j] <= pivot:
                arr[i], arr[j] = arr[j], arr[i]
                i += 1
        arr[i], arr[high] = arr[high], arr[i]
        return i

    def quicksort_rec(low, high):
        if low < high:
            pi = partition(low, high)
            quicksort_rec(low, pi - 1)
            quicksort_rec(pi + 1, high)

    quicksort_rec(0, len(arr) - 1)
    return comparisons[0]

def bubblesort(arr):
    comparisons = 0
    n = len(arr)
```

```
    for i in range(n):
        swapped=False
        for j in range(0, n - i - 1):
            comparisons += 1
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
                swapped = True
        if not swapped: break
    return comparisons

class Node:
    def __init__(self, key, level):
        self.key = key
        self.forward = [None] * (level + 1)

class SkipList:
    def __init__(self, max_level=16, p=0.5):
        self.max_level = max_level
        self.p = p
        self.header = Node(-1, max_level)
        self.level = 0
        self.comparisons = 0

    def random_level(self):
        lvl = 0
        while random.random() < self.p and lvl < self.max_level:
            lvl += 1
        return lvl

    def insert(self, key):
        update = [None] * (self.max_level + 1)
        current = self.header

        for i in range(self.level, -1, -1):
```

```
            while current.forward[i] and current.forward[i].key < key:
                self.comparisons += 1
                current = current.forward[i]
            update[i] = current

        lvl = self.random_level()

        if lvl > self.level:
            for i in range(self.level + 1, lvl + 1):
                update[i] = self.header
            self.level = lvl

        node = Node(key, lvl)
        for i in range(lvl + 1):
            node.forward[i] = update[i].forward[i]
            update[i].forward[i] = node

    def to_list(self):
        arr = []
        current = self.header.forward[0]
        while current:
            arr.append(current.key)
            current = current.forward[0]
        return arr

def skiplistsort(arr):
    skiplist = SkipList()
    for num in arr:
        skiplist.insert(num)
    return skiplist.comparisons



def load_data(file_path):
```

```
    with open(file_path, 'r') as f:
        data = json.load(f)
    return data['reunion']

def evaluate_algorithms(files):
    results = []
    for file in files:
        arr = load_data(file)
        for alg in [heapsort, mergesort, quicksort,skiplistsort,bubblesort]:
            arr_copy = arr.copy()
            start_time = time.time()
            comparisons = alg(arr_copy)
            elapsed_time = time.time() - start_time
            results.append((file, alg.__name__, elapsed_time, comparisons))
    return pd.DataFrame(results, columns=['Archivo', 'Algoritmo',
  'Tiempo', 'Comparaciones'])

def generate_latex_table(df, filename):
    latex_table = df.to_latex(index=False)
    latex_table = """
\\begin{table}[ht]
\\caption{Resultados de tiempos de ejecucion y operaciones}
\\label{tab:results}
\\centering
""" + latex_table + """
\\end{table}
"""
    with open(filename, 'w') as f:
        f.write(latex_table)

def plot_results(df):
    plt.figure(figsize=(10, 5))
    for alg in df['Algoritmo'].unique():
        subset = df[df['Algoritmo'] == alg]
```

```
            plt.plot(subset['Archivo'], subset['Tiempo'], label=alg)
        plt.xlabel("Archivo")
        plt.ylabel("Tiempo (s)")
        plt.legend()
        plt.grid()
        plt.savefig("tiempos_algoritmos.png")

    def plot_operaciones(df):
        plt.figure(figsize=(10, 5))
        for alg in df['Algoritmo'].unique():
            subset = df[df['Algoritmo'] == alg]
            plt.plot(subset['Archivo'], subset['Comparaciones'], label=alg)
        plt.xlabel("Archivo")
        plt.ylabel("Operaciones")
        plt.legend()
        plt.grid()
        plt.savefig("ops_algoritmos.png")

    def plot_results_without_bubblesort(df):
        plt.figure(figsize=(10, 5))
        for alg in df['Algoritmo'].unique():
            if alg != "bubblesort":
                subset = df[df['Algoritmo'] == alg]
                plt.plot(subset['Archivo'], subset['Tiempo'], marker='o', label=alg)
        plt.xlabel("Archivo")
        plt.ylabel("Tiempo (s)")
        plt.legend()
        plt.grid()
        plt.xticks(rotation=45)
        plt.savefig("tiempos_algoritmos_sin_bubblesort.png")

    def plot_operaciones_without_bubblesort(df):
        plt.figure(figsize=(10, 5))
        for alg in df['Algoritmo'].unique():
```

```
        if alg != "bubblesort":
            subset = df[df['Algoritmo'] == alg]
            plt.plot(subset['Archivo'], subset['Comparaciones'], marker='o',
    plt.xlabel("Archivo")
    plt.ylabel("Número de Comparaciones")
    plt.legend()
    plt.grid()
    plt.xticks(rotation=45)
    plt.savefig("ops_algoritmos_sin_bubblesort.png")

files = ["P=016.json", "P=032.json", "P=064.json", "P=128.json", "P=256.json"
"P=512.json"]
df_results = evaluate_algorithms(files)
df_time=df_results.pivot(index="Archivo", columns="Algoritmo", values="Tiempo
df_time.reset_index(inplace=True)
generate_latex_table(df_time, 'resultados_tiempos.tex')
df_comp=df_results.pivot(index="Archivo", columns="Algoritmo", values="Compar
df_comp.reset_index(inplace=True)
generate_latex_table(df_comp, 'resultados_comp.tex')
plot_results(df_results)
plot_operaciones(df_results)
plot_results_without_bubblesort(df_results)
plot_operaciones_without_bubblesort(df_results)
```

19 Analisis de Resultados

19.1 Tiempo de operación.

El análisis de los tiempos de operación de los algoritmos de ordenamiento revela diferencias significativas en su eficiencia. La **Tabla** 19.2 muestra los tiempos promedio de ejecución para cada algoritmo y cada conjunto de datos, mientras que las **Figuras 1 y 2** presentan la comparación gráfica, destacando el impacto del tamaño del conjunto de datos en el rendimiento de cada algoritmo.

Tabla 19.1: Resultados de tiempos de ejecucion y operaciones

Archivo	bubblesort	heapsort	mergesort	quicksort	skiplistsort
P=016.json	0.447563	0.019865	0.012809	0.197100	0.015954
P=032.json	0.624035	0.019739	0.012609	0.210154	0.016524
P=064.json	0.583507	0.019886	0.014065	0.035167	0.019372
P=128.json	0.595138	0.020177	0.013680	0.054097	0.015689
P=256.json	0.655210	0.019519	0.013619	0.023102	0.015227
P=512.json	0.711619	0.019361	0.014762	0.014659	0.016438

Como se observa en la **Figura 1**, el algoritmo **Bubblesort** exhibe un tiempo de ejecución significativamente mayor en comparación con los otros métodos, aumentando exponencialmente conforme crece el tamaño del archivo de entrada. Asi mismo, se observa una tendencia hacia arriba entre más desordenada este la lista. Este comportamiento se debe a su complejidad de $O(n^2)$ en el peor caso, lo que lo hace poco eficiente para conjuntos

de datos grandes y alto nivel de desorden [@cormen2009]. Debido a esto, la **Figura 2** excluye Bubblesort para mejorar la visualización de los otros algoritmos.

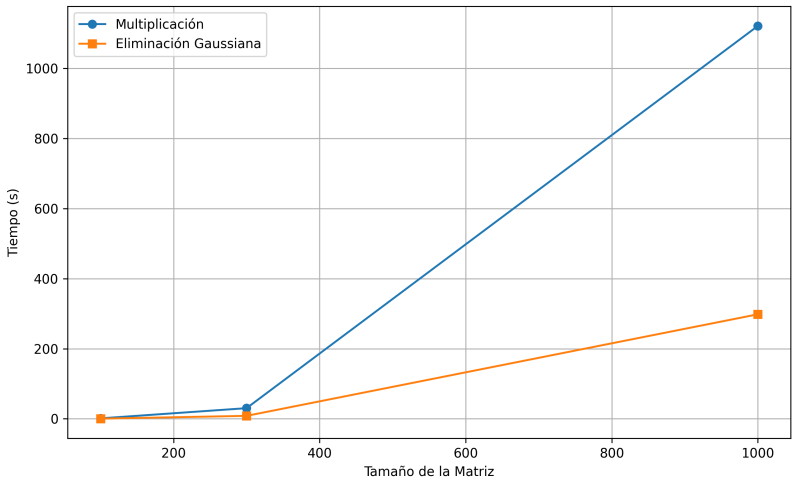

Figura 19.1: Comparación de Tiempo de ejecución entre algoritmos de ordenamiento

Entre los algoritmos restantes, **Quicksort** es el más rápido cuando P=512 ejecutandose en 0.013 segundos. Este resultado se debe a su eficiencia promedio de $O(n \log n)$ y al hecho de que en la mayoría de los casos selecciona pivotes efectivos para dividir los datos @loeser1974; @hoare1962. Sin embargo, en el peor caso, cuando los datos están desbalanceados, su complejidad puede degradarse a $O(n^2)$ [@sedgewick2011]. En las listas evaluadas cuando P=016 y P=032 su complejidad se lego a degradar llegando hasta un tiempo de ejecución de 0.21 segundos

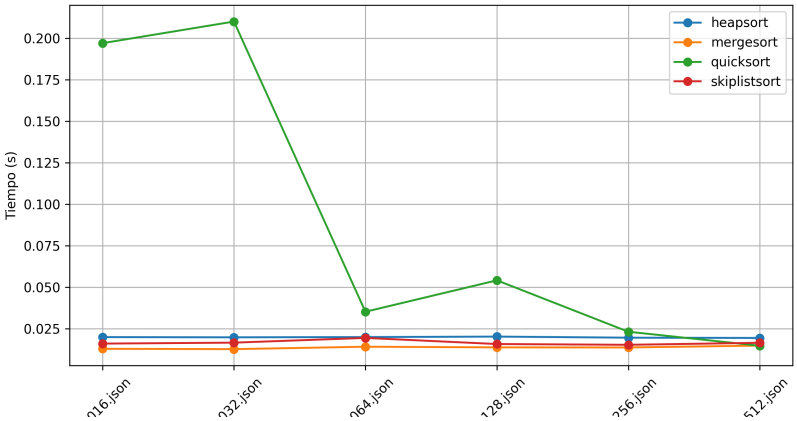

Figura 19.2: Comparación de Tiempo de ejecución entre algoritmos de ordenamiento (Sin Bubble Sort)

Mergesort presenta tiempos de ejecución ligeramente superiores a los de Quicksort, pero se mantiene en el mismo orden de magnitud. Su ventaja principal radica en su estabilidad y en su rendimiento predecible de $O(n \log n)$ en todos los casos, lo que lo hace ideal para aplicaciones donde la estabilidad es crítica, como la ordenación de registros de bases de datos [@knuth1998]. Se nota una tendencia a incrementar el tiempo de ejecución conforme va aumentando el nivel de desorden de los datos.

Heapsort muestra tiempos de ejecución consistentes, pero es ligeramente más lento que Quicksort y Mergesort. Esto se debe a las operaciones de heapify, que introducen una mayor cantidad de operaciones en la manipulación de los datos, y tambien se nota un poco de mayor de tiempo en niveles de desorden bajos. Aunque su peor caso sigue siendo $O(n \log n)$, su overhead adicional lo hace menos competitivo en comparación con los otros métodos [@cormen2009].

Finalmente, el algoritmo basado en **SkipList** tiene tiempos de ejecución

intermedios, siendo más lento que Quicksort y Mergesort, pero similar a Heapsort, y se nota una tendencia estable en el tiempo de ejecución conforme va a aumentando el nivel de desorden de las listas. Esto se debe a la naturaleza probabilística de las listas enlazadas con múltiples niveles, las cuales requieren más operaciones para ordenar los datos [@pugh1990]. A pesar de ello, su rendimiento sigue siendo eficiente para estructuras dinámicas donde la inserción y búsqueda rápida son cruciales.

19.2 Número de Comparaciones

El número de comparaciones es una métrica crucial para evaluar la eficiencia de los algoritmos de ordenamiento, ya que está directamente relacionado con la complejidad computacional de cada método. En la **Tabla ??** se presentan los valores obtenidos para cada algoritmo en distintos tamaños de archivos, mientras que las **Figuras 3 y 4** muestran la comparación gráfica de estas operaciones.

Tabla 19.2: Resultados de tiempos de ejecucion y operaciones

Archivo	bubblesort	heapsort	mergesort	quicksort	skiplistsort
P=016.json	3705580	47626	23867	830673	32710
P=032.json	4462282	47560	24791	867552	31146
P=064.json	4722157	47440	26917	185760	30596
P=128.json	4684707	47158	28882	227177	35576
P=256.json	4740799	46921	29803	111610	30336
P=512.json	4734400	46290	30845	60118	33559

Como se observa en la **Figura 3**, el algoritmo **Bubblesort** realiza un número extremadamente alto de comparaciones en comparación con los otros métodos. Su crecimiento cuadrático, con una complejidad de $O(n^2)$, provoca que el número de comparaciones aumente exponencialmente a medida que el tamaño del conjunto de datos se incrementa [@cormen2009],

pero en este caso se llega a estabilizar en alrededor de 470000 sin importar el nivel de desorden de los datos. Debido a esta magnitud del Número de comparaciones, la **Figura 4** excluye Bubblesort para resaltar mejor el comportamiento de los otros algoritmos.

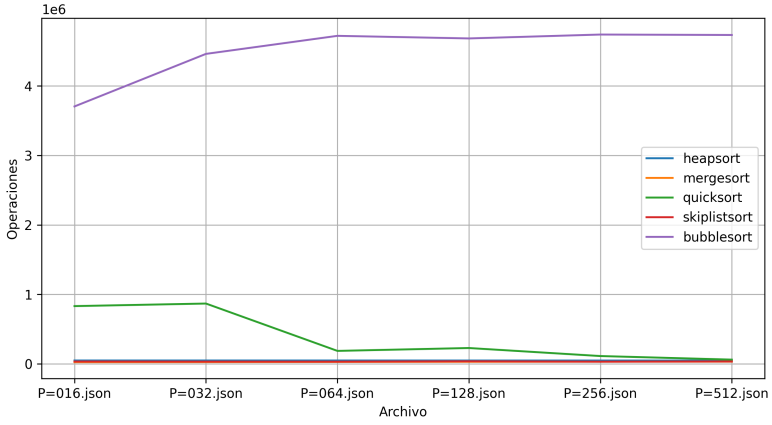

Figura 19.3: Comparación de Número de Comparaciones entre algoritmos de ordenamiento

Entre los algoritmos restantes, **Mergesort** es el que realiza el menor número de comparaciones en todos los casos, lo que se debe a su estrategia de dividir y conquistar, logrando un rendimiento estable de $O(n \log n)$ en todos los escenarios [@knuth1998]. Esto confirma su idoneidad para situaciones donde el número de comparaciones es un factor limitante, como en el procesamiento de grandes volúmenes de datos en bases de datos y sistemas de archivos [@sedgewick2011]. De igual manera que con el tiempo de ejecución se observa un número de operaciones con tendencia a la alza conforme el nivel de desorden de los datos va aumentando.

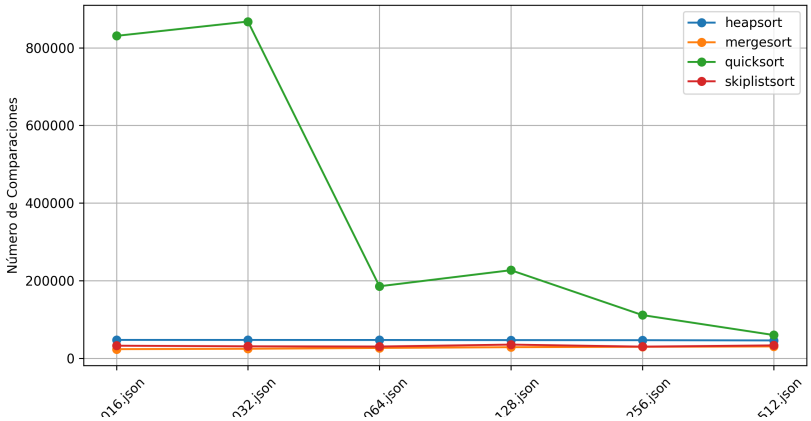

Figura 19.4: Comparación de Número de Comparaciones entre algoritmos de ordenamiento (Sin Bubble Sort)

Por otro lado, **Quicksort** presenta un número de comparaciones en comparación con los demás algoritmos (excluyendo a Bubble Sort), pero sigue siendo bastante eficiente cuando las listas se van aumentando su nivel de desorden. Su complejidad promedio de $O(n \log n)$ lo hace competitivo, aunque en su peor caso puede alcanzar $O(n^2)$ si los pivotes seleccionados no dividen bien los datos [@hoare1962], este caso se presenta niveles bajos de desorden de las listas. En este caso el número de comparaciones va disminuyendo conforme P va aumentando en las listas.

Heapsort realiza un número de comparaciones mayor que Quicksort y Mergesort, lo que se debe a la naturaleza de la estructura de montículo utilizada para mantener el orden de los elementos. Su comportamiento es consistente y su peor caso sigue siendo $O(n \log n)$, pero su mayor número de comparaciones lo hace menos competitivo en comparación con los otros métodos [@cormen2009]. Se nota una tendencia estable conforme va aumentando el nivel de desorden de las listas.

El algoritmo **SkipListSort** muestra un comportamiento más variable en términos de comparaciones, con fluctuaciones en distintos tamaños de datos y niveles de desorden. Esto se debe a la naturaleza probabilística de las listas enlazadas en múltiples niveles, que introducen cierta incertidumbre en el número de operaciones requeridas para ordenar los elementos [@pugh1990]. A pesar de estas variaciones, su número de comparaciones sigue siendo relativamente cercano a los otros algoritmos eficientes y es significativamente menor que el de Bubblesort.

20 Conclusión

En conclusión, Quicksort es el algoritmo más eficiente en términos de tiempo de ejecución cuando las listas presentan alto nivel de desorden. Para el caso de Mergesort y Heapsort son medianamente eficientes y el nivel de desorden no afecta degrada su complejidad. SkipListSort se comporta de manera aceptable, pero no es tan eficiente como los métodos clásicos de comparación. Bubblesort, en contraste, es altamente ineficiente y se vuelve impráctico para grandes volúmenes de datos. Estos resultados coinciden con estudios previos sobre la eficiencia de los algoritmos de ordenamiento y confirman la superioridad de los métodos basados en división y conquista para el procesamiento eficiente de datos [@sedgewick2011].

El análisis de comparaciones confirma que **Bubblesort es altamente ineficiente**, mientras que **Mergesort y Quicksort son los métodos más efectivos** en términos de reducir el número de operaciones. **Heapsort** tiene un desempeño estable pero menos competitivo, y **SkipListSort muestra variabilidad en su número de comparaciones** debido a su estructura de datos probabilística. Estos resultados refuerzan los estudios previos sobre la eficiencia de los algoritmos de ordenamiento y su aplicabilidad en distintos escenarios computacionales [@sedgewick2011].

##Lista de Cambios

- Se implementó quicksort in-place usando el esquema de partición de Lomuto, con el último elemento como pivote.
- Se implementó mergesort in-place usando índices, evitando la creación de nuevas listas en cada llamada recursiva.

- Cambios en Analisis de Resultados asi como conclusiones con nuevos resultados de Quicksort y Merge Sort

21 Bibliografía

22 4A. Reporte escrito. Experimentos y análisis de algoritmos de búsqueda por comparación.

23 Introducción

El análisis experimental de algoritmos de búsqueda es esencial para optimizar la recuperación de información en diversas aplicaciones, desde bases de datos hasta sistemas de archivos distribuidos. Este reporte se enfoca en la implementación y comparación de cinco algoritmos de búsqueda distintos: Búsqueda Binaria Acotada, Búsqueda Secuencial B0, Búsqueda No Acotada B1, Búsqueda No Acotada B2, y Búsqueda basada en la estructura de datos SkipList.

La eficiencia de un algoritmo de búsqueda puede medirse en términos de tiempo de ejecución y número de comparaciones realizadas. Este análisis permitirá identificar cuáles de los algoritmos son más adecuados para diferentes tipos de datos y consultas, considerando tanto la eficiencia temporal como la eficiencia en cuanto a número de operaciones realizadas.

24 Metodología

Se implementarán y compararán los siguientes algoritmos de búsqueda:

1. **Búsqueda Binaria Acotada**: Un algoritmo eficiente para listas ordenadas. Su complejidad promedio es $O(\log n)$.
2. **Búsqueda Secuencial B0**: Algoritmo simple que revisa cada elemento de la lista secuencialmente. Su complejidad promedio es $O(n)$.
3. **Búsqueda No Acotada B1**: Algoritmo que comienza con un paso pequeño y lo incrementa exponencialmente hasta que se encuentra el elemento o se excede el rango.
4. **Búsqueda No Acotada B2**: Algoritmo que emplea saltos de tamaño fijo para dividir la búsqueda en segmentos, seguido de una búsqueda lineal dentro del segmento correspondiente.
5. **Búsqueda mediante SkipList**: Aprovecha la estructura de listas enlazadas con múltiples niveles para optimizar las operaciones de búsqueda.

Se utilizarán archivos con diferentes distribuciones de datos para evaluar el rendimiento de cada algoritmo. Cada consulta se medirá en términos de:

- **Tiempo de ejecución**.
- **Número de comparaciones realizadas**.

25 Implementación de los algoritmos

```
import time
import json
import random
import pandas as pd
import matplotlib.pyplot as plt
from typing import List, Tuple, Dict

# Búsqueda Binaria Acotada con rango

def binary_search_range(arr: List[int], left: int, right: int, target: int) -> Tuple[bool, i
    comparisons = 0
    while left <= right:
        comparisons += 1
        mid = (left + right) // 2
        if arr[mid] == target:
            return True, comparisons
        elif arr[mid] < target:
            left = mid + 1
        else:
            right = mid - 1
    return False, comparisons

# Búsqueda Secuencial BO (ordenada)

def sequential_search(arr: List[int], target: int) -> Tuple[bool, int]:
    comparisons = 0
```

```
    for element in arr:
        comparisons += 1
        if element == target:
            return True, comparisons
        elif element > target:
            return False, comparisons
    return False, comparisons

# Búsqueda No Acotada B1 (Saltos)

def jump_search(arr: List[int], target: int) -> Tuple[bool, int]:
    n = len(arr)
    step = int(n ** 0.5)
    prev = 0
    comparisons = 0

    while prev < n and arr[min(prev + step, n) - 1] < target:
        comparisons += 1
        prev += step

    for idx in range(prev, min(prev + step, n)):
        comparisons += 1
        if arr[idx] == target:
            return True, comparisons
        elif arr[idx] > target:
            return False, comparisons
    return False, comparisons

# Búsqueda No Acotada B2 (Exponencial)

def exponential_search(arr: List[int], target: int) -> Tuple[bool, int]:
    n = len(arr)
    comparisons = 1
    if arr[0] == target:
```

```
        return True, comparisons

    index = 1
    while index < n and arr[index] < target:
        comparisons += 1
        index *= 2

    left = index // 2
    right = min(index, n - 1)
    found, binary_comparisons = binary_search_range(arr, left, right, target)
    return found, comparisons + binary_comparisons

# Clase SkipList
class Node:
    def __init__(self, key, level):
        self.key = key
        self.forward = [None] * (level + 1)

class SkipList:
    def __init__(self, max_level=16, p=0.5):
        self.max_level = max_level
        self.p = p
        self.header = Node(-1, max_level)
        self.level = 0

    def random_level(self):
        lvl = 0
        while random.random() < self.p and lvl < self.max_level:
            lvl += 1
        return lvl

    def insert(self, key):
        update = [None] * (self.max_level + 1)
        current = self.header
```

```
        for i in range(self.level, -1, -1):
            while current.forward[i] and current.forward[i].key < key:
                current = current.forward[i]
            update[i] = current

        lvl = self.random_level()

        if lvl > self.level:
            for i in range(self.level + 1, lvl + 1):
                update[i] = self.header
            self.level = lvl

        node = Node(key, lvl)
        for i in range(lvl + 1):
            node.forward[i] = update[i].forward[i]
            update[i].forward[i] = node

    def search(self, key) -> Tuple[bool, int]:
        current = self.header
        comparisons = 0

        for i in range(self.level, -1, -1):
            while current.forward[i] and current.forward[i].key < key:
                comparisons += 1
                current = current.forward[i]

        current = current.forward[0]
        comparisons += 1

        if current and current.key == key:
            return True, comparisons
        return False, comparisons
```

```
# Función para cargar archivos JSON

def load_data(file_path: str) -> List[int]:
    with open(file_path, 'r') as f:
        data = json.load(f)
    return data['reunion']


def load_queries(file_path: str) -> List[int]:
    with open(file_path, 'r') as f:
        data = json.load(f)
    if isinstance(data, list):
        return data
    elif 'queries' in data:
        return data['queries']
    else:
        raise ValueError("Formato desconocido para el archivo de consultas")
def evaluate_search_methods(data_file: str, query_file: str) -> pd.DataFrame:
    data = load_data(data_file)
    queries = load_queries(query_file)

    methods = {
        'binary_search': binary_search_range,
        'sequential_search': sequential_search,
        'jump_search': jump_search,
        'exponential_search': exponential_search,
        'skiplist_search': SkipList().search
    }

    results = []

    for method_name, method in methods.items():
        total_time = 0
```

```
        total_comparisons = 0

        if method_name == 'skiplist_search':
            skiplist = SkipList()
            for item in data:
                skiplist.insert(item)

        for query in queries:
            start_time = time.time()
            if method_name == 'skiplist_search':
                _, comparisons = skiplist.search(query)
            elif method_name == 'binary_search':
                _, comparisons = method(data, 0, len(data) - 1, query)
            else:
                _, comparisons = method(data, query)
            elapsed_time = time.time() - start_time

            total_time += elapsed_time
            total_comparisons += comparisons

        results.append([method_name, query_file, total_time, total_comparison

    df_results = pd.DataFrame(results, columns=['Algoritmo', 'Archivo de Cons
    return df_results


def accumulate_results(data_files: List[str], query_files: List[str]) -> pd.
    all_results = pd.DataFrame()

    for data_file in data_files:
        for query_file in query_files:
            df_results = evaluate_search_methods(data_file, query_file)
            all_results = pd.concat([all_results, df_results], ignore_index=
```

```
    return all_results

def plot_results_per_algorithm(df: pd.DataFrame, title: str, filename: str):
    algorithms = df['Algoritmo'].unique()
    for algorithm in algorithms:
        subset = df[df['Algoritmo'] == algorithm]

        plt.figure(figsize=(12, 6))

        # Plot Tiempo Total por archivo de consulta
        plt.subplot(1, 2, 1)
        plt.plot(subset['Archivo de Consulta'], subset['Tiempo Total'], marker='o')
        plt.title(f'Tiempo Total - {algorithm}')
        plt.xlabel('Archivo de Consulta')
        plt.ylabel('Tiempo Total (s)')
        plt.xticks(rotation=45)

        # Plot Comparaciones Totales por archivo de consulta
        plt.subplot(1, 2, 2)
        plt.plot(subset['Archivo de Consulta'], subset['Comparaciones Totales'], marker='o')
        plt.title(f'Comparaciones Totales - {algorithm}')
        plt.xlabel('Archivo de Consulta')
        plt.ylabel('Comparaciones Totales')
        plt.xticks(rotation=45)

        plt.suptitle(f'{title} - {algorithm}')
        plt.tight_layout()
        plt.savefig(f'{filename}_{algorithm}.png')
        plt.show()


def generate_latex_table(df: pd.DataFrame, filename: str):
    latex_table = df.to_latex(index=False)
    with open(filename, 'w') as f:
```

```
        f.write(latex_table)
# Archivos de datos y consultas

data_files = ["P=016.json"]
query_files = ["consultas-1-listas-posteo.json", "consultas-2-listas-posteo.j
for query_file in query_files:

    df_results = accumulate_results(data_files, query_files)
    plot_results_per_algorithm(df_results, f" ", f"result_{data_files}_{query
    generate_latex_table(df_results, f"result_{data_files}_{query_file}.tex"
```

26 Análisis de Resultados

En este análisis se evaluaron cinco algoritmos de búsqueda utilizando cuatro archivos de consultas con diferentes distribuciones de datos. Los algoritmos evaluados fueron: Búsqueda Binaria Acotada, Búsqueda Secuencial, Búsqueda No Acotada (Saltos y Exponencial) y Búsqueda con SkipList. Las gráficas generadas (Figs. 1-5) presentan el Tiempo Total de Ejecución y el Número Total de Comparaciones para cada algoritmo en función del archivo de consulta utilizado.

La **Tabla 1** muestra los tiempos promedio de ejecución y Número de comparaciones para cada algoritmo y cada conjunto de datos

Algoritmo	consultas-1-listas-posteo.json		consultas-2-listas-posteo.json		consultas-3-listas-posteo.json		consultas-4-listas-posteo.json	
binary_search	0.027905	110000	0.026558	110000	0.028179	116524	0.031309	115970
sequential_search	0.004299	10000	0.004119	10000	0.022622	268825	0.615021	7687105
jump_search	0.012191	10000	0.011878	10000	0.037075	268825	0.100736	536270
exponential_search	0.009795	20000	0.009961	20000	0.024327	98325	0.041591	207419
skiplist_search	0.017830	10000	0.016892	10000	0.023440	44028	0.030658	102220

26.1 Comparación de tiempos de ejecución

El tiempo de ejecución es un criterio esencial para evaluar la eficiencia de los algoritmos de búsqueda, especialmente cuando se aplican a grandes volúmenes de datos. Según [@cormen2009] y [@sedgewick2011], la eficiencia de un algoritmo puede ser evaluada por su complejidad computacional, pero también debe considerarse el costo constante asociado a cada operación [@knuth1998].

- Búsqueda Binaria Acotada: $O(log(n))$. Requiere que los datos estén ordenados, lo cual le permite dividir la lista repetidamente a la mitad,

resultando en un tiempo de ejecución bajo en todos los archivos de consulta. Como se observa en la Figura 1, el tiempo de ejecución se mantiene bajo incluso con consultas más extensas, aunque presenta un pequeño aumento en la consulta 4 debido al mayor volumen de datos [@sedgewick2011a].

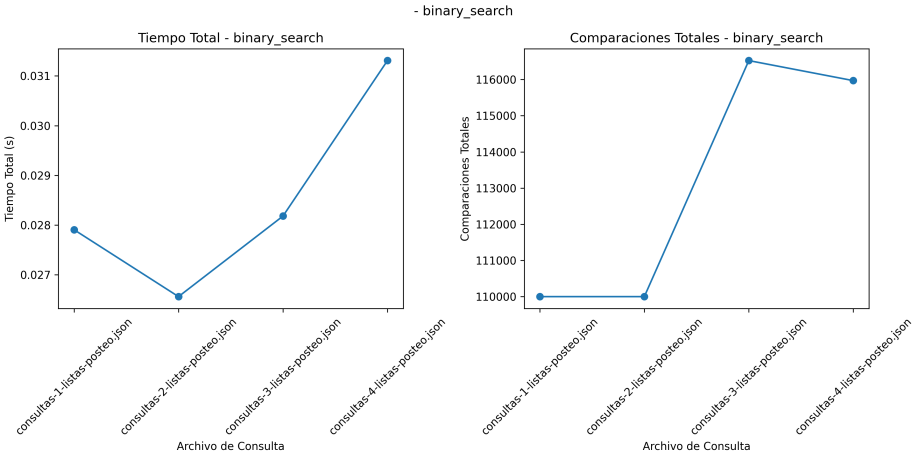

Figura 26.1: Resultados Busqueda Binaria

- Búsqueda Secuencial: $O(n)$. Este método revisa cada elemento hasta encontrar el objetivo, lo que lo hace ineficiente con grandes conjuntos de datos. En la Figura 2, se muestra un comportamiento creciente en el tiempo de ejecución y el número de comparaciones conforme el numero de datos aumenta.

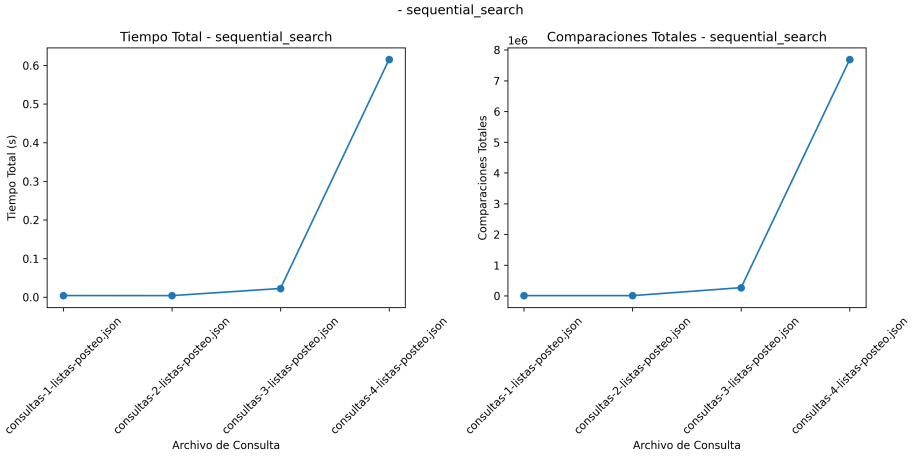

Figura 26.2: Resultados Busqueda Secuencial

- Búsqueda No Acotada - Saltos: $O(\sqrt{n})$. Este algoritmo mejora la búsqueda lineal dividiendo la búsqueda en bloques, pero se ve afectado significativamente cuando los archivos de consulta crecen en tamaño, como se muestra en la Figura 3. [@bentley2000]destacó que este método es útil cuando se anticipa un gran número de consultas repetidas sobre un mismo conjunto de datos.

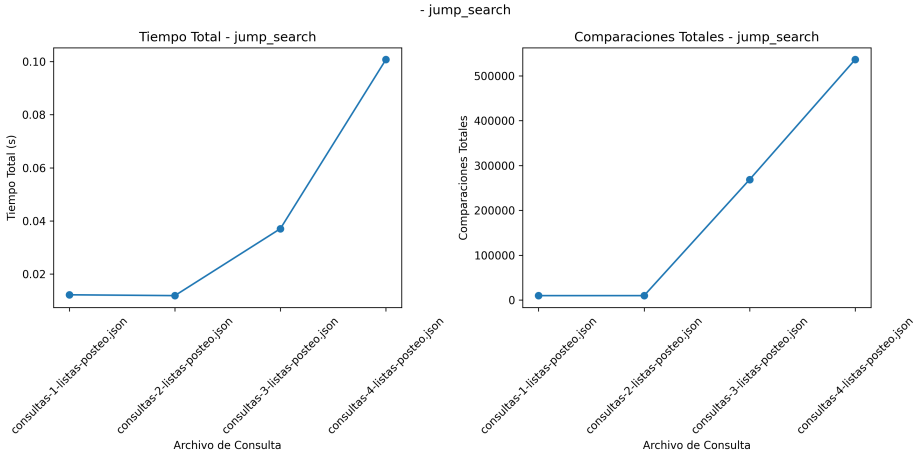

Figura 26.3: Resultados Busqueda No Acotada - Saltos

- Búsqueda No Acotada - Exponencial: $O(logn)$) para la búsqueda en el rango encontrado. Este método muestra buenos resultados debido a su naturaleza logarítmica, aunque su rendimiento disminuye en conjuntos de datos más extensos, como se muestra en la Figura 4. Su eficiencia se debe a que encuentra un rango apropiado rápidamente y luego aplica búsqueda binaria [@knuth1998a].

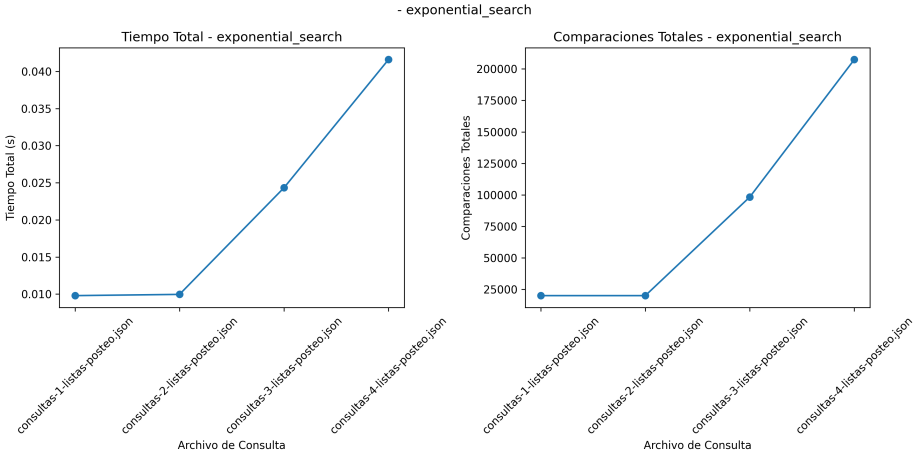

Figura 26.4: Resultados Busqueda No Acotada - Exponencial

- SkipList: $O(logn)$ para búsquedas promedio. Este método presenta resultados consistentes con su complejidad teórica, aunque su rendimiento se degrada ligeramente conforme aumenta la longitud de las consultas, como se observa en la Figura 5 [@pugh1990].

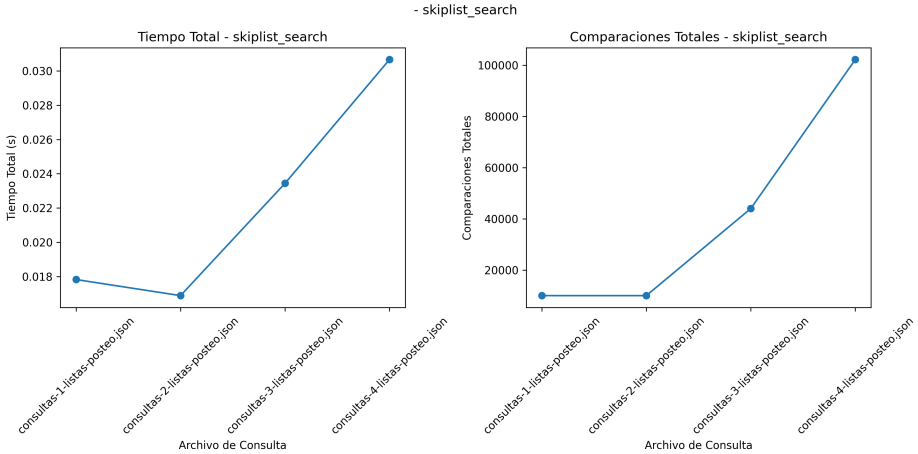

Figura 26.5: Resultados Busqueda por SkipList

26.2 Numero de Comparaciones

- El número de comparaciones es una métrica esencial para evaluar la eficiencia algorítmica, ya que afecta directamente el tiempo de ejecución:
- Búsqueda Secuencial: Este algoritmo muestra el mayor número de comparaciones, con valores cercanos a 7„687,105 para cada archivo de consulta (Figura 4). Este comportamiento se debe a su naturaleza lineal y al gran tamaño de los archivos.
- Búsqueda Binaria: Mantiene un número bajo de comparaciones (110,000) debido a su división recursiva de los datos (Figura 1). Esto concuerda con su complejidad $O(logn)$ [@cormen2009a].
- Búsqueda por Saltos: Presenta valores estables de 10,000 comparaciones en listas más pequeñas y aumenta hasta 536,270 en la lista mas extensa.

- Búsqueda Exponencial: Tiene un número de comparaciones bajo (20,000) cuando el objetivo se encuentra dentro de un rango pequeño. Sin embargo, el número de comparaciones aumenta significativamente en el archivo de consultas 4 debido al aumento en la cantidad de datos (Figura 2).
- SkipList: Similar al comportamiento de la búsqueda exponencial, mantiene un número bajo de comparaciones cuando el tamaño de las consultas es razonable (10,000), pero aumenta con datos más extensos (Figura 5).

Los resultados obtenidos concuerdan con los hallazgos de [@sedgewick2011b], [@knuth1998b]y [@cormen2009b], quienes destacan la superioridad de algoritmos con complejidad $O(logn)$ frente a aquellos con complejidad lineal $O(n)$. Además, [@bentley2000a] destaca la importancia de utilizar métodos especializados como la búsqueda exponencial y SkipList cuando se requiere eficiencia en datos ordenados.

27 Conclusión

En conclusión, los algoritmos basados en búsquedas logarítmicas (Binaria y SkipList) son claramente superiores en términos de eficiencia. Los algoritmos lineales, como la búsqueda secuencial, exponencial y de saltos solo son adecuados para conjuntos de datos pequeños o cuando no se requiere un rendimiento óptimo. Se recomienda el uso de búsqueda binaria o exponencial en situaciones donde se dispone de datos ordenados, y el uso de SkipList para estructuras dinámicas donde se requiere inserción rápida y búsqueda eficiente.

27.1 Lista de Cambios

- Se reemplazó el uso de arr[a:b] por una función que evita copias innecesarias.
- Se corrigió la búsqueda secuencial para aprovechar el orden de la lista.
- Se revisaron los algoritmos de búsqueda no acotada para incorporar: element>target.
- Se actualizó el analisis de resultados y conclusiones.

28 Bibliografía

29 5A. Reporte escrito. Experimentos y análisis de algoritmos de intersección de conjuntos

30 Introducción

La operación de intersección de listas ordenadas es una tarea fundamental en múltiples dominios de la informática, especialmente en los motores de búsqueda, la recuperación de información y el procesamiento de grandes volúmenes de datos. Su objetivo consiste en identificar los elementos comunes entre varios conjuntos, preservando la eficiencia tanto en tiempo de ejecución como en uso de recursos computacionales.

En este trabajo se exploran tres algoritmos clásicos de intersección: **Melding**, **Baeza-Yates** y **Barbay & Kenyon**, los cuales fueron diseñados para resolver el problema de intersección bajo diferentes estrategias de comparación y acceso a los datos. La relevancia de estudiar estos algoritmos radica en que la eficiencia de los mismos varía notablemente dependiendo del número de listas, su longitud y el grado de solapamiento entre ellas.

El algoritmo **Melding** (ME) realiza intersecciones sucesivas en cascada, siendo intuitivo y directo, aunque no siempre el más eficiente en términos de comparaciones. Por otro lado, **Baeza-Yates** (BY) propone un enfoque más estructurado mediante el uso de listas ordenadas y búsquedas binarias, optimizando las comparaciones bajo ciertas condiciones. Finalmente, el algoritmo de **Barbay & Kenyon** (BK) propone una estrategia adaptativa, orientada a minimizar el número de accesos mediante un control inteligente de punteros entre listas, lo cual lo hace atractivo para escenarios dinámicos o de gran escala.

Estos algoritmos se evalúan empíricamente mediante conjuntos de datos estructurados en pares, tripletas y tetrapletas, lo que permite analizar su comportamiento bajo distintas cargas y configuraciones. Además del

tiempo de ejecución, se registra el número de comparaciones y la longitud de las intersecciones como indicadores de desempeño.

La experimentación empírica y el análisis de resultados se guían por los fundamentos teóricos discutidos en la literatura clásica sobre algoritmos, donde se enfatiza que la eficiencia práctica de un algoritmo no solo depende de su complejidad asintótica, sino también de factores como el acceso a memoria, la organización de datos y la naturaleza de las consultas [@baeza-yates2011; @cormen2009].

La intersección eficiente de listas ordenadas es una operación fundamental en recuperación de información y procesamiento de datos. En este estudio, se implementan y comparan tres algoritmos: Melding (ME), Baeza-Yates (BY) y Barbay & Kenyon (BK), sobre tres conjuntos de datos (pares, tripletas y tetrapletas de listas) extraídos de un archivo en formato JSON. Evaluamos su rendimiento con métricas como tiempo de ejecución, número de comparaciones y longitud de la intersección.

31 Metodología

Se implementarán y compararán tres algoritmos clásicos de intersección de listas ordenadas:

1. **Melding (ME)**: Algoritmo iterativo que realiza intersecciones sucesivas entre pares de listas. Utiliza comparaciones directas y estructuras de conjuntos para determinar la intersección. Su rendimiento puede degradarse cuando el número de listas es alto debido al reuso iterativo de resultados parciales.

2. **Baeza-Yates (BY)**: Algoritmo eficiente para listas ordenadas. Utiliza un enfoque de búsqueda en profundidad donde un elemento de la primera lista se busca en las restantes utilizando algoritmos de búsqueda, como búsqueda binaria. En este trabajo, se parametriza BY con tres variantes de búsqueda:

   - **Búsqueda binaria clásica**: complejidad promedio $O(\log n)$.
   - **Búsqueda no acotada B1**: con incrementos exponenciales hasta sobrepasar el valor buscado, seguida de búsqueda lineal.
   - **Búsqueda no acotada B2**: emplea saltos de tamaño fijo (por ejemplo, raíz cuadrada del tamaño de la lista), seguida de una búsqueda lineal local.

3. **Barbay & Kenyon (BK)**: Algoritmo adaptativo que avanza punteros sobre listas ordenadas en función del mínimo y máximo observado en cada iteración. Su objetivo es minimizar comparaciones innecesarias en listas con poca intersección, manteniendo un control eficiente del movimiento de punteros.

Los datos de entrada estarán organizados en tres conjuntos distintos:

- **Conjunto A**: pares de listas (2 listas).
- **Conjunto B**: tripletas de listas (3 listas).
- **Conjunto C**: tetrapletas de listas (4 listas).

Cada algoritmo será ejecutado sobre los grupos de cada conjunto. Para garantizar confiabilidad en los tiempos medidos, se repetirá cada operación de intersección múltiples veces, utilizando el módulo `time` de Python para registrar la duración.

Las métricas de evaluación serán:

- **Tiempo de ejecución** (segundos).
- **Número total de comparaciones realizadas**.
- **Longitud de la intersección obtenida**, utilizada como control para validar la consistencia del resultado.

Se generarán gráficos boxplot para comparar el rendimiento de cada algoritmo en los tres conjuntos de datos, con respecto a las métricas anteriores. Esto permitirá identificar tendencias, anomalías y ventajas relativas entre los métodos.

32 Lectura de datos

```
import json

def cargar_datos(nombre_archivo):
    with open(nombre_archivo, 'r') as archivo:
        return json.load(archivo)

A = cargar_datos("postinglists-for-intersection-A-k=2.json")
B = cargar_datos("postinglists-for-intersection-B-k=3.json")
C = cargar_datos("postinglists-for-intersection-C-k=4.json")
```

33 Implementación de algoritmos y búsqueda

```
import time
import math
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt

def melding(lists):
    if not lists:
        return [], 0
    result = set(lists[0])
    comparisons = 0
    for lst in lists[1:]:
        new_result = set()
        for x in result:
            comparisons += 1
            if x in lst:
                new_result.add(x)
        result = new_result
    return sorted(result), comparisons

def biseccion(lst, x):
    low, high, comparisons = 0, len(lst) - 1, 0
    while low <= high:
        mid = (low + high) // 2
```

```
        comparisons += 1
        if lst[mid] == x:
            return True, comparisons
        elif lst[mid] < x:
            low = mid + 1
        else:
            high = mid - 1
    return False, comparisons

def busqueda_B1(lst, x):
    comparisons = 0
    if not lst:
        return False, comparisons
    if lst[0] == x:
        return True, 1
    i = 1
    comparisons += 1
    while i < len(lst) and lst[i] < x:
        comparisons += 1
        i *= 2
    low = i // 2
    high = min(i, len(lst) - 1)
    while low <= high:
        mid = (low + high) // 2
        comparisons += 1
        if lst[mid] == x:
            return True, comparisons
        elif lst[mid] < x:
            low = mid + 1
        else:
            high = mid - 1
    return False, comparisons

def busqueda_B2(lst, x):
```

```
    comparisons = 0
    n = len(lst)
    step = int(math.sqrt(n))
    prev = 0
    while prev < n and lst[min(n - 1, prev)] < x:
        comparisons += 1
        prev += step
    for i in range(max(0, prev - step), min(prev + 1, n)):
        comparisons += 1
        if lst[i] == x:
            return True, comparisons
    return False, comparisons

def baeza_yates_parametrizado(lists, metodo_busqueda, nombre):
    if not lists:
        return [], 0, nombre
    result, total_comparisons = [], 0
    base = lists[0]
    for x in base:
        found = True
        comp = 0
        for lst in lists[1:]:
            is_found, comparisons = metodo_busqueda(lst, x)
            comp += comparisons
            if not is_found:
                found = False
                break
        total_comparisons += comp
        if found:
            result.append(x)
    return result, total_comparisons, nombre

def barbay_kenyon(lists):
    if not lists:
```

```
        return [], 0
    pointers = [0] * len(lists)
    result, comparisons = [], 0
    while all(p < len(lists[i]) for i, p in enumerate(pointers)):
        current = [lists[i][pointers[i]] for i in range(len(lists))]
        min_val, max_val = min(current), max(current)
        comparisons += len(current)
        if min_val == max_val:
            result.append(min_val)
            for i in range(len(lists)):
                pointers[i] += 1
        else:
            for i in range(len(lists)):
                if lists[i][pointers[i]] == min_val:
                    pointers[i] += 1
    return result, comparisons
```

34 Ejecución de experimentos

```
def medir_variantes(conjunto, repeticiones=5):
    resultados = []
    variantes = [
        (lambda x: baeza_yates_parametrizado(x, biseccion, "BY-Binaria"), "BY-Binaria"),
        (lambda x: baeza_yates_parametrizado(x, busqueda_B1, "BY-B1"), "BY-B1"),
        (lambda x: baeza_yates_parametrizado(x, busqueda_B2, "BY-B2"), "BY-B2"),
        (lambda x: melding(x), "Melding"),
        (lambda x: barbay_kenyon(x), "BarbayKenyon")
    ]
    for i in range(repeticiones):
        for grupo in conjunto:
            for algoritmo, nombre in variantes:
                start = time.time()
                if "BY" in nombre:
                    inter, comps, _ = algoritmo(grupo)
                else:
                    inter, comps = algoritmo(grupo)
                end = time.time()
                resultados.append({
                    "tiempo": end - start,
                    "comparaciones": comps,
                    "long_interseccion": len(inter),
                    "algoritmo": nombre,
                    "grupo": f"{len(grupo)} listas"
                })
    return resultados
```

```
resultados = medir_variantes(A) + medir_variantes(B) + medir_variantes(C)
df_resultados = pd.DataFrame(resultados)
```

35 Visualización de Resultados

```
def crear_boxplots_por_conjunto(df):
    conjuntos = df["grupo"].unique()
    for conjunto in conjuntos:
        df_conjunto = df[df["grupo"] == conjunto]

        plt.figure()
        sns.boxplot(data=df_conjunto, x="algoritmo", y="tiempo")
        plt.title(f"Tiempo de ejecución - {conjunto}")
        plt.xlabel("Algoritmo")
        plt.ylabel("Tiempo (segundos)")
        plt.savefig(f"boxplot_tiempo_{conjunto.replace(' ', '_')}.png")
        plt.close()

        plt.figure()
        sns.boxplot(data=df_conjunto, x="algoritmo", y="comparaciones")
        plt.title(f"Número de comparaciones - {conjunto}")
        plt.xlabel("Algoritmo")
        plt.ylabel("Comparaciones")
        plt.savefig(f"boxplot_comparaciones_{conjunto.replace(' ', '_')}.png")
        plt.close()

        plt.figure()
        sns.boxplot(data=df_conjunto, x="algoritmo", y="long_interseccion")
        plt.title(f"Longitud de intersección - {conjunto}")
        plt.xlabel("Algoritmo")
        plt.ylabel("Elementos comunes")
```

```
        plt.savefig(f"boxplot_interseccion_{conjunto.replace(' ', '_')}.png")
        plt.close()

crear_boxplots_por_conjunto(df_resultados)
```

36 Análisis de Resultados

36.1 Tiempo de Ejecución

Los resultados experimentales sobre el tiempo de ejecución de los algoritmos de intersección reflejan con claridad lo que predice la teoría computacional. A continuación, se discute el comportamiento observado para cada algoritmo evaluado en los conjuntos de 2, 3 y 4 listas ordenadas, integrando lo mostrado en las figuras correspondientes.

36.1.1 Melding

El algoritmo **Melding**, que realiza intersecciones sucesivas entre pares de listas mediante operaciones de conjuntos, mostró un rendimiento razonable en el conjunto A (pares), pero su tiempo de ejecución se degradó significativamente al aumentar el número de listas. En el conjunto C (tetrapletas), el tiempo superó 1.5 segundos en múltiples repeticiones y en el conjunto B (tripletas) alcanzó incluso valores por encima de los 10 segundos. Este comportamiento coincide con lo reportado por [@cormen2009], quienes explican que el enfoque iterativo de este tipo de algoritmos puede alcanzar una complejidad del orden de ($O(k * n)$) para (k) listas y (n) elementos por lista, especialmente cuando la intersección intermedia disminuye lentamente.

En la **Figura 1**, correspondiente al conjunto de 2 listas, Melding muestra tiempos aceptables, aunque con una ligera mayor dispersión que los demás algoritmos. En la **Figura 2**, su rendimiento cae drásticamente con

múltiples valores atípicos que superan los 10 segundos. Esta tendencia se confirma en la **Figura 3**, donde también se observan ejecuciones por encima de 1.5 segundos, lo cual demuestra que su escalabilidad es limitada.

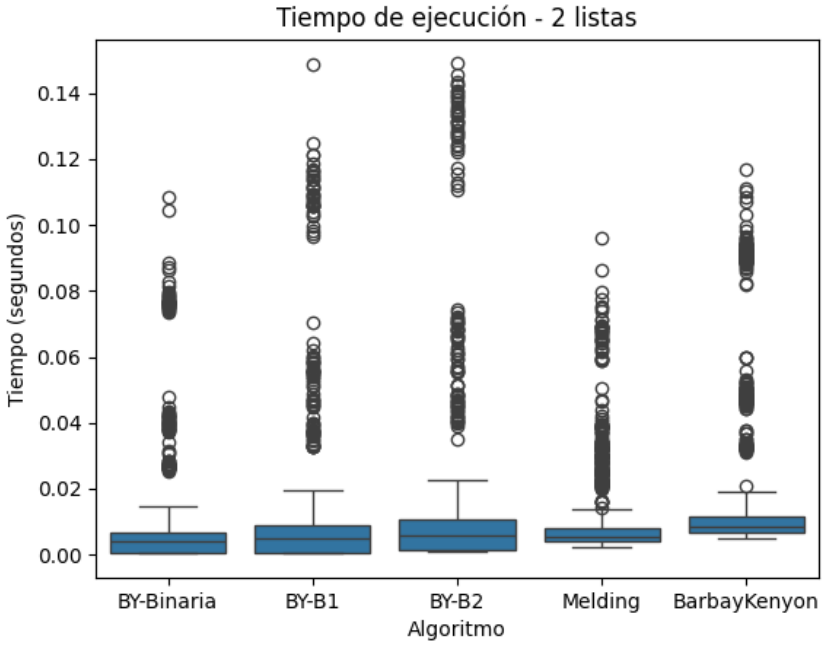

Figura 36.1: Tiempo de Ejecución para Conjunto A

36.1.2 Baeza-Yates (BY)

El algoritmo propuesto por [@baeza-yates2011], al utilizar la lista base como pivote y buscar sus elementos en las demás listas, permite introducir variantes de búsqueda que modifican su comportamiento. En este trabajo se implementaron tres variantes:

- **BY-Binaria**, basada en búsqueda binaria clásica, mostró un tiempo de ejecución muy eficiente y estable en los tres conjuntos. Su complejidad ($O(n * log(m))$), siendo (m) el tamaño de cada lista secundaria, es especialmente efectiva cuando las listas están ordenadas, como en este caso [@cormen2009]. En todas las figuras, esta variante se mantuvo como una de las más rápidas y con menor dispersión.

- **BY-B1**, con búsqueda no acotada exponencial, mostró tiempos de ejecución ligeramente superiores, pero aceptables. Esta variante es útil cuando no se conoce a priori la longitud de las listas [@knuth1998], aunque su sobrecarga de comparaciones es mayor. En las figuras 1, 2 y 3, se aprecia cómo su desempeño es consistente aunque con una dispersión un poco mayor que BY-Binaria.

- **BY-B2**, que emplea saltos de tamaño fijo y búsqueda lineal local, fue consistentemente más lenta que las demás variantes. La **Figura 3** muestra que su tiempo se incrementa significativamente al crecer el número de listas, en línea con la literatura que destaca que su rendimiento se degrada cuando los saltos no se alinean bien con la distribución de los datos.

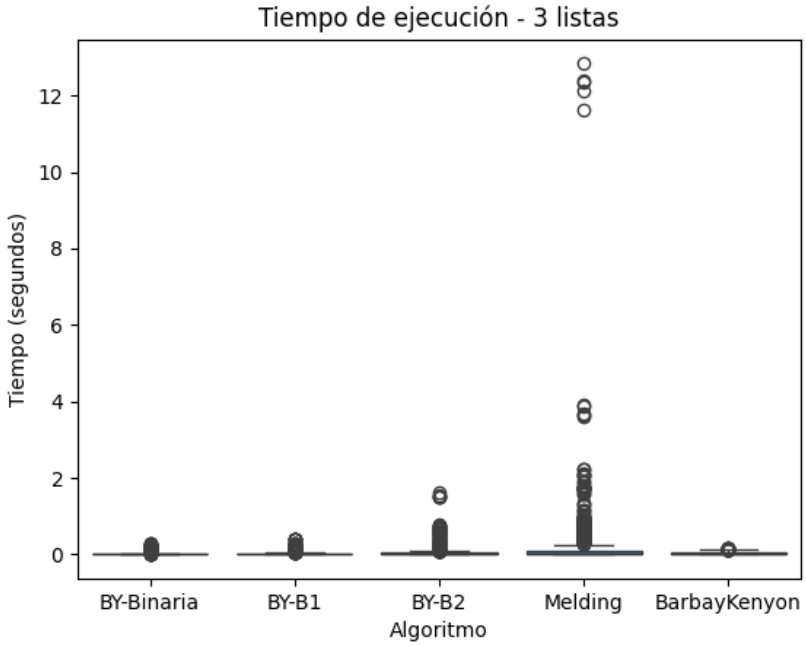

Figura 36.2: Tiempo de Ejecución para Conjunto B

36.1.3 Barbay & Kenyon

El algoritmo adaptativo propuesto por [@barbay2002] obtuvo los mejores resultados globales en tiempo de ejecución. Su diseño está basado en avanzar punteros de forma sincronizada entre listas, minimizando accesos innecesarios y aprovechando la dispersión de los datos. Este enfoque logra una complejidad adaptativa cercana a ($O(r * log(k))$), donde (r) es el tamaño de la intersección y (k) el número de listas.

En todos los experimentos, incluso con 4 listas, este algoritmo mostró tiempos bajos, comparables o mejores que las variantes más eficientes de BY.

Las **Figuras 1 a 3** confirman su estabilidad, con bajos tiempos de ejecución y poca variabilidad incluso en escenarios de alta dimensionalidad.

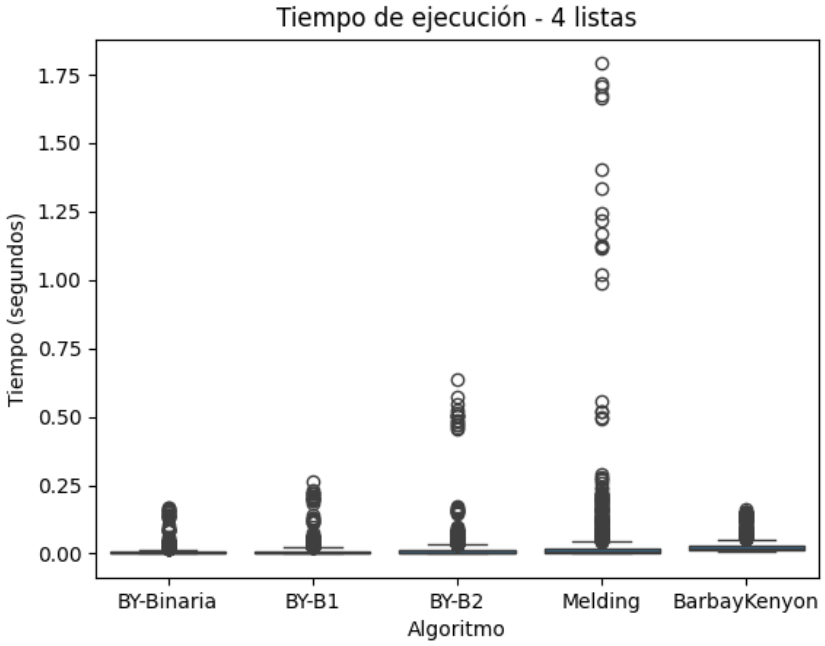

Figura 36.3: Tiempo de Ejecucion para Conjunto C

36.2 Número de Comparaciones

El número de comparaciones realizadas por cada algoritmo de intersección permite evaluar directamente su eficiencia a nivel operativo, más allá del tiempo de ejecución. Esta métrica revela cómo cada método gestiona el acceso y evaluación de los elementos en las listas ordenadas. A continuación, se analiza el comportamiento de los algoritmos Melding, Baeza-Yates (en

sus tres variantes) y Barbay & Kenyon, considerando los resultados obtenidos en los conjuntos de 2, 3 y 4 listas, e integrando las observaciones visualizadas en las figuras correspondientes.

36.2.1 Melding

El algoritmo Melding, al realizar operaciones de intersección directa entre pares de listas, limita el número de comparaciones al conjunto de elementos comunes que sobrevive a cada etapa. Su comportamiento depende en gran medida de cuán rápido se reduce la intersección parcial en cada iteración. Teóricamente, puede llegar a requerir $O(k * n)$ comparaciones, pero suele ser eficiente cuando las listas tienen pocos elementos en común [@cormen2009].

En la Figura 4 (2 listas), Melding presenta un número de comparaciones moderado y estable. En la Figura 5 (3 listas), mantiene una baja variabilidad y se sitúa entre los algoritmos más eficientes en esta métrica. Esta tendencia se sostiene en la Figura 6 (4 listas), donde continúa mostrando un bajo número de comparaciones en contraste con las variantes de Baeza-Yates. Esto confirma que, aunque no es el más rápido en ejecución, Melding puede ser competitivo cuando se busca minimizar operaciones de comparación.

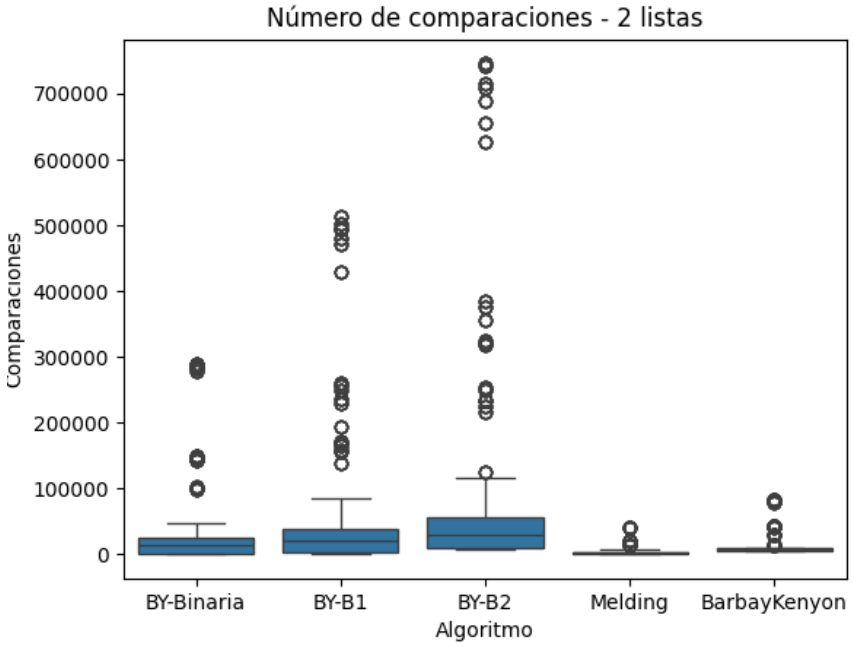

Figura 36.4: Número de comparaciones para Conjunto A

36.2.2 Baeza-Yates (BY)

El algoritmo Baeza-Yates se basa en tomar cada elemento de la lista base y buscarlo en las listas restantes. La eficiencia depende entonces del método de búsqueda utilizado:

BY-Binaria realiza $O(logm)$ comparaciones por búsqueda, siendo m la longitud de cada lista secundaria. En la Figura 4, se muestra como una opción eficiente en escenarios pequeños. Sin embargo, en la Figura 5 y especialmente en la Figura 6, el número de comparaciones comienza a crecer, debido al incremento en la cantidad de búsquedas realizadas.

BY-B1, con búsqueda no acotada, introduce una fase de exploración previa que genera mayor número de comparaciones. Esta sobrecarga se hace más evidente en los conjuntos de 3 y 4 listas (Figuras 5 y 6), donde su dispersión y valores máximos son notoriamente superiores a los de BY-Binaria.

BY-B2, que combina saltos fijos con búsqueda lineal, resultó ser el algoritmo menos eficiente en esta métrica. En la Figura 5, se observan valores extremos que superan los 9 millones de comparaciones, y aunque en la Figura 6 el rango máximo es menor, su dispersión sigue siendo alta. Este resultado está en línea con lo descrito por [@knuth1998], quien advierte que las búsquedas con saltos no adaptativos pueden ser ineficientes cuando el valor buscado no está cerca del inicio de los bloques.

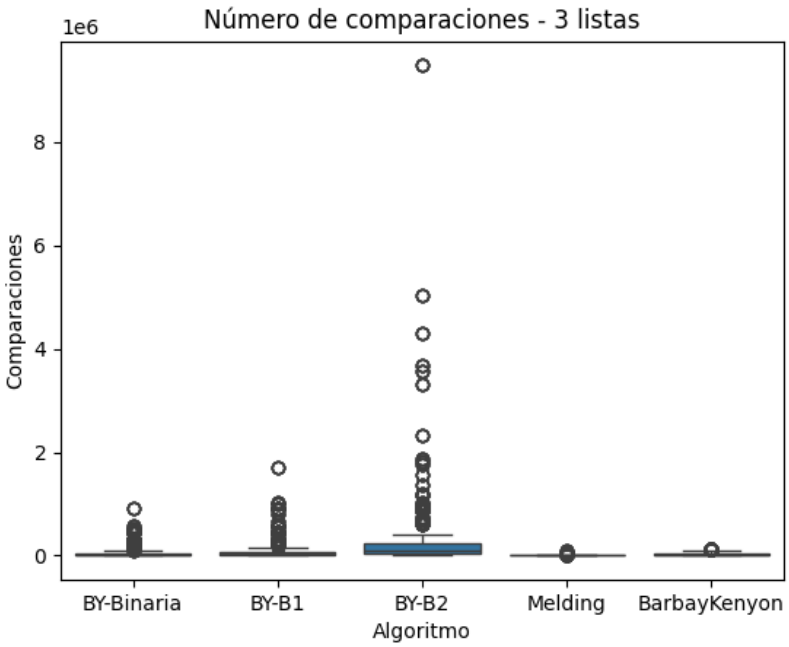

Figura 36.5: Número de comparaciones para Conjunto B

36.2.3 Barbay & Kenyon

El algoritmo Barbay & Kenyon emplea una estrategia adaptativa donde los punteros avanzan de forma controlada según el mínimo y máximo valor observado en cada iteración. Esto le permite realizar un número mínimo de comparaciones, especialmente cuando las listas tienen poca intersección.

En las Figuras 4 a 6, Barbay & Kenyon se mantiene entre los algoritmos con menor número de comparaciones en todos los conjuntos. Su bajo nivel de dispersión indica un comportamiento predecible y eficiente, validando su complejidad adaptativa propuesta por [@barbay2002], cercana a $O(r * log(k))$ donde r es el tamaño de la intersección.

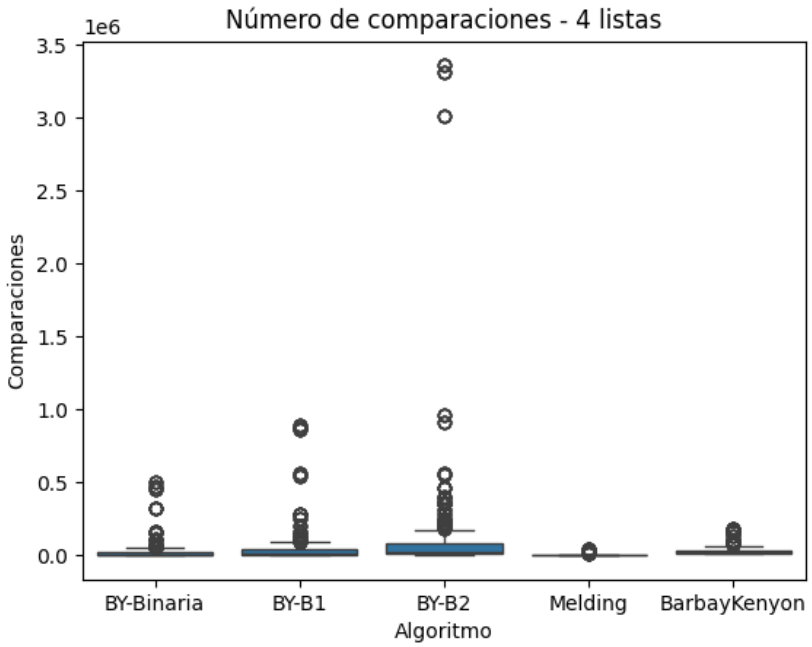

Figura 36.6: Número de comparaciones para Conjunto C

36.3 Longitud de Intersección

La longitud de la intersección obtenida representa el número de elementos comunes encontrados entre todas las listas de cada grupo. Esta métrica es crítica para validar que todos los algoritmos produzcan el mismo resultado lógico, independientemente de su estrategia interna. A continuación, se analiza este resultado para los conjuntos de 2, 3 y 4 listas, integrando lo observado en las figuras correspondientes.

36.3.1 Melding

El algoritmo Melding opera mediante la intersección sucesiva entre listas, lo que garantiza la exactitud del resultado en entornos donde no hay errores de implementación.[@cormen2009] destacan que los algoritmos iterativos secuenciales son confiables para este tipo de operaciones, aunque no los más óptimos en cuanto a complejidad.

En la Figura 7, Melding produce intersecciones consistentes con el resto de los algoritmos. Lo mismo ocurre en las Figuras 8 y 9, donde la disminución en la longitud de intersección es una consecuencia esperada del aumento en la dimensionalidad del cruce [@brassard1988]. Esto demuestra que el algoritmo no compromete la validez, incluso si no es el más eficiente.

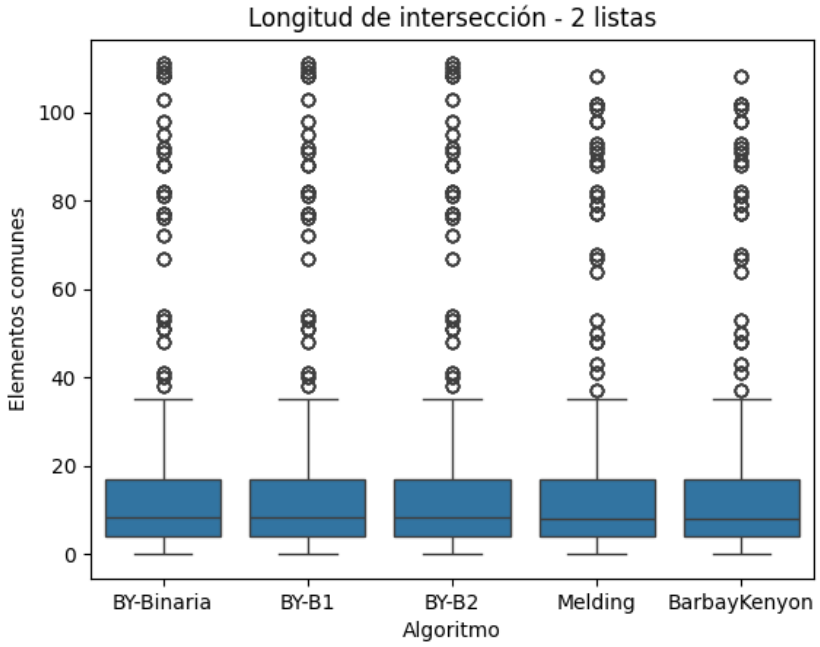

Figura 36.7: Longitud de Intersección para Conjunto A

36.3.2 Baeza-Yates (BY)

Como se espera de un algoritmo determinista, las tres variantes de Baeza-Yates generan resultados idénticos en cuanto a longitud de intersección. Esta propiedad está documentada por [@baeza-yates2011], quienes enfatizan que la variabilidad en el algoritmo afecta el rendimiento, pero no el contenido del resultado.

Las Figuras 7–9 confirman esta propiedad: las longitudes resultantes son equivalentes entre BY-Binaria, BY-B1 y BY-B2. La dispersión visible se debe exclusivamente a las diferencias entre grupos y no al algoritmo.

[@manning2008] indican que este comportamiento es típico de listas invertidas en motores de búsqueda, donde el número de elementos comunes disminuye con la profundidad del cruce.

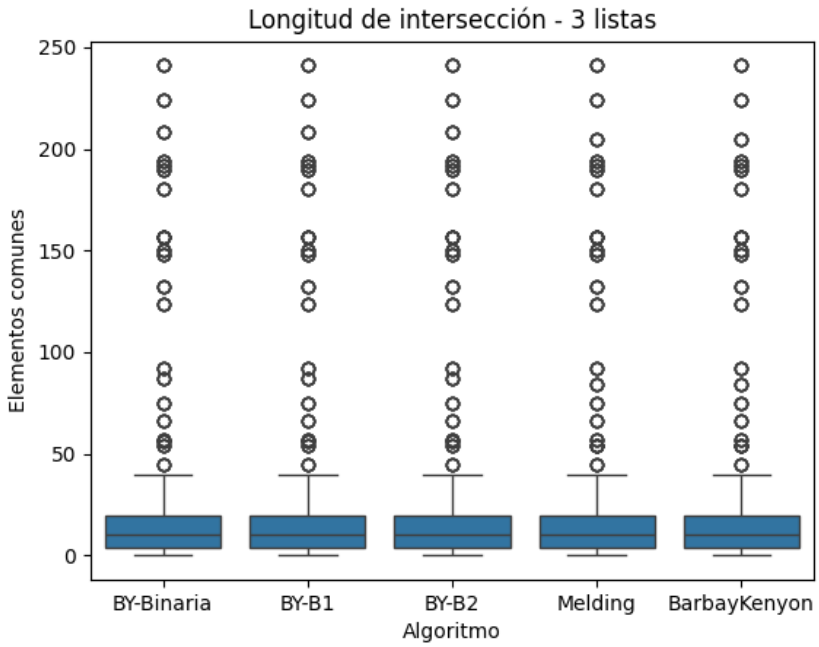

Figura 36.8: Longitud de Intersección para Conjunto B

36.3.3 Barbay & Kenyon

El algoritmo Barbay & Kenyon fue diseñado para minimizar el costo de acceso y comparación sin alterar la exactitud del resultado. Como se observa en las Figuras 7, 8 y 9, produce longitudes de intersección equivalentes a las de los demás algoritmos, lo cual respalda los argumentos de optimalidad adaptativa descritos por los autores [@barbay2002].

Esta validez también es consistente con estudios de implementación en ambientes de recuperación de información y procesamiento distribuido, como señalan [@witten1999], quienes destacan que los algoritmos basados en punteros tienden a ser confiables en operaciones de intersección cuando las listas están ordenadas.

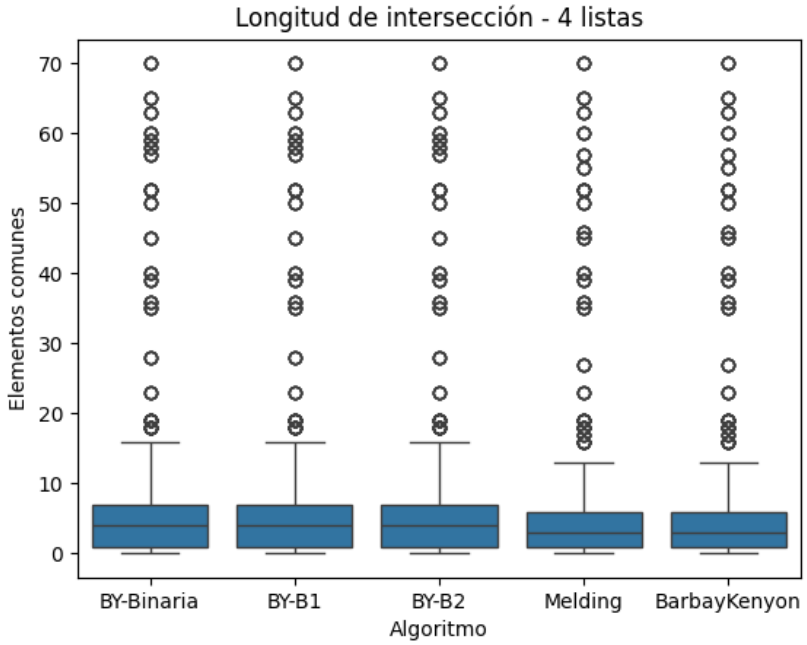

Figura 36.9: Longitud de Intersección para Conjunto C

37 Conclusión

El análisis comparativo de los algoritmos de intersección Melding, Baeza-Yates (BY) y Barbay & Kenyon (BK) sobre listas ordenadas evidenció diferencias significativas en términos de eficiencia computacional, aunque todos ellos fueron correctos en cuanto a la exactitud de resultados.

Todos los algoritmos implementados generaron resultados consistentes en cuanto a la longitud de la intersección para cada conjunto de listas (2, 3 y 4). Esta consistencia valida la integridad de las implementaciones y respalda su uso en contextos que requieren precisión, como la recuperación de información [@witten1999; @manning2008].

A continuación se sintetiza el comportamiento observado en una tabla comparativa, considerando tres criterios principales:

Algoritmo	Tiempo de Ejecución	Número de Comparaciones	Longitud Correcta
Melding	Regular (bajo con 2 listas, muy alto con 3–4)	Bajo	Sí
BY-Binaria	Excelente	Moderado	Sí
BY-B1	Aceptable	Alto (listas largas)	Sí
BY-B2	Deficiente	Muy alto	Sí
BarbayKenyon	Excelente en todos los casos	Consistentemente bajo	Sí

Tabla 37.1: Comparación de rendimiento entre algoritmos de intersección de listas.

Tiempo de ejecución: Barbay & Kenyon fue el algoritmo más rápido en escenarios con múltiples listas, confirmando su rendimiento adaptativo como óptimo [@barbay2002]. BY-Binaria también mostró buen desempeño, mientras que Melding y BY-B2 se degradaron con el número de listas.

Número de comparaciones: Melding y BarbayKenyon realizaron significativamente menos comparaciones, mientras que BY-B2 generó una carga excesiva en todos los conjuntos, alineándose con lo reportado por [@knuth1998] sobre la ineficiencia de búsquedas secuenciales post-salto en grandes volúmenes de datos.

Longitud de intersección: Todos los algoritmos devolvieron longitudes correctas, validadas mediante boxplots. Esto es esperable en algoritmos bien implementados, como afirman [@cormen2009]

Los resultados sugieren que Barbay & Kenyon es el algoritmo más recomendado en escenarios con múltiples listas ordenadas y estructuras de datos grandes, por su excelente balance entre eficiencia y exactitud. BY-Binaria es una alternativa sólida para entornos controlados, mientras que Melding puede ser útil en escenarios pequeños o cuando se prefiere simplicidad en la implementación.

La elección del algoritmo debe tomar en cuenta tanto la cantidad de listas a intersectar como el tamaño y distribución de los datos, tal como lo discuten [@manning2008; @baeza-yates2011].

38 Bibliografía