UNIVERSIDAD AMERICANA

Algoritmos y estructura de Datos

Docente: Lic.Silvia Ticay Lopez

Integrantes:

DOSSON ARNOLDO LUQUES NARVAEZ

Fecha: 25/06/25

I. INTRODUCCIÓN

1. Planteamiento del problema

En el desarrollo de sistemas informáticos modernos que requieren manejo eficiente de grandes volúmenes de datos, como aplicaciones de inventarios empresariales, motores de búsqueda web, bases de datos relacionales y sistemas de gestión de información, surge constantemente la necesidad de localizar información específica de forma rápida y precisa. El uso de algoritmos inadecuados para esta tarea fundamental puede causar demoras significativas en la ejecución de aplicaciones, consumo excesivo de recursos computacionales, y en casos críticos, la degradación completa del rendimiento del sistema.

La búsqueda binaria emerge como una solución algorítmica altamente eficiente para listas ordenadas, pero su aplicabilidad real, limitaciones prácticas y rendimiento comparativo deben analizarse en profundidad para determinar su idoneidad en diferentes contextos de aplicación.

2. Objetivo de la investigación

Evaluar de manera integral el rendimiento teórico y empírico del algoritmo de búsqueda binaria en entornos controlados de laboratorio, comparando sistemáticamente su eficiencia temporal y espacial frente a otros algoritmos de búsqueda tradicionales. Se busca establecer parámetros cuantitativos que permitan determinar las condiciones óptimas para su implementación, así como identificar los escenarios donde su uso resulta más beneficioso. La investigación pretende proporcionar evidencia empírica sólida que respalde las afirmaciones teóricas sobre la superioridad de la búsqueda binaria en contextos de datos ordenados.

3. Objetivos específicos

1. **Describir detalladamente** el funcionamiento interno de la búsqueda binaria paso a paso, explicando la lógica de división y conquista que fundamenta su eficiencia algorítmica.

2. **Analizar exhaustivamente** su complejidad computacional en los tres escenarios fundamentales: mejor caso, peor caso y caso promedio, estableciendo las bases matemáticas que sustentan su rendimiento.

3. **Medir sistemáticamente** su desempeño en términos de tiempo de ejecución y uso de memoria RAM con diferentes tamaños de listas, desde escalas pequeñas hasta grandes volúmenes de datos.

4. **Comparar cuantitativamente** su eficiencia frente a algoritmos alternativos como la búsqueda lineal, estableciendo métricas de mejora y puntos de equilibrio operacional.

5. **Validar experimentalmente** las fórmulas teóricas de complejidad mediante pruebas empíricas controladas, verificando la correspondencia entre el análisis matemático y el comportamiento real del algoritmo.

4. Justificación de la Investigación

La selección de la búsqueda binaria como objeto de estudio se fundamenta en su importancia histórica y práctica en el campo de la computación. Como uno de los algoritmos fundamentales en la ciencia de la computación, su comprensión profunda es esencial para cualquier desarrollador o analista de sistemas. Además, su aplicación directa en múltiples tecnologías cotidianas, desde bases de datos hasta sistemas de archivos, hace que su análisis detallado tenga relevancia práctica inmediata.

La investigación contribuye al cuerpo de conocimiento existente proporcionando mediciones empíricas precisas y análisis comparativos que complementan la teoría algorítmica tradicional.

II. METODOLOGÍA

1. Diseño de la investigación

Se aplicó un diseño metodológico mixto que combina elementos de investigación no experimental (análisis teórico) con componentes experimentales (pruebas empíricas) mediante simulaciones controladas en el lenguaje de programación Python. El enfoque teórico permitió establecer las bases matemáticas y conceptuales del algoritmo, mientras que el componente experimental proporcionó evidencia empírica para validar las hipótesis formuladas. Esta combinación metodológica garantiza una evaluación integral que abarca tanto los aspectos conceptuales como los prácticos del algoritmo de búsqueda binaria.

2. Enfoque de la investigación

Enfoque Cualitativo: Se empleó para comprender en profundidad el funcionamiento interno del algoritmo, analizar su lógica de programación, identificar los patrones de comportamiento y establecer las relaciones causales entre las diferentes variables que afectan su rendimiento. Este enfoque permitió desarrollar una comprensión holística del algoritmo más allá de las mediciones numéricas.

Enfoque Cuantitativo: Se utilizó para registrar datos precisos de tiempo de ejecución, uso de memoria RAM, número de operaciones elementales y otros parámetros de rendimiento. Las mediciones se realizaron con instrumentos de alta precisión (time.perf_counter() y tracemalloc) para garantizar la exactitud de los datos recolectados.

Enfoque Mixto: Se integraron ambos enfoques para lograr una evaluación completa y equilibrada, donde los datos cuantitativos validan las observaciones cualitativas y viceversa, creando una comprensión más robusta y confiable del comportamiento del algoritmo.

3. Alcance de la investigación

Alcance Descriptivo: Se explica detalladamente el funcionamiento, estructura y lógica del algoritmo de búsqueda binaria, proporcionando una comprensión clara de su mecanismo interno y las razones de su eficiencia. Se describen las condiciones necesarias para su aplicación y las limitaciones que presenta en diferentes contextos.

Alcance Correlacional: Se establecen relaciones estadísticas entre el rendimiento del algoritmo y variables como el tamaño de los datos, la posición del elemento buscado y el tipo de datos utilizados. Se comparan sistemáticamente los resultados de la búsqueda binaria con otros algoritmos para establecer patrones de comportamiento.

Alcance Explicativo: Se determinan las causas fundamentales de la mayor eficiencia de la búsqueda binaria en escenarios de datos ordenados, analizando los principios matemáticos que sustentan su rendimiento superior y las condiciones que maximizan su efectividad.

4. Procedimiento

Investigación teórica de la búsqueda binaria

Se realizó una revisión exhaustiva de literatura académica sobre algoritmos de búsqueda, analizando la base matemática del algoritmo de división y conquista. Se estudió en profundidad la complejidad computacional teórica O(log n) y se investigaron casos de uso y aplicaciones prácticas en sistemas reales. Durante esta fase se compilaron las ventajas y limitaciones del algoritmo, estableciendo una base sólida para la implementación experimental.

Diseño de pruebas con listas de distintos tamaños y condiciones

Se definieron tamaños de prueba escalonados desde 100 hasta 100,000 elementos para evaluar el comportamiento del algoritmo en diferentes escalas. Se crearon tres escenarios de prueba fundamentales: el mejor caso donde el elemento buscado se ubica en el centro de la lista, el caso promedio con elementos en posiciones aleatorias, y el peor caso donde el elemento se encuentra en los extremos o no existe en la lista. Se diseñaron pruebas comparativas con búsqueda lineal para validar la superioridad del algoritmo y se prepararon datos cualitativos como nombres para demostrar la aplicabilidad práctica del método.

Implementación en Python

Se desarrolló el algoritmo de búsqueda binaria iterativa con un contador de pasos integrado para medir la eficiencia operacional. Se implementaron funciones de medición de rendimiento utilizando time.perf_counter() para obtener precisión en microsegundos, y se integró tracemalloc para el monitoreo detallado del consumo de memoria RAM. Se crearon funciones auxiliares para la generación automática de listas ordenadas y datos de prueba, además de implementar una versión de búsqueda lineal para establecer comparaciones de rendimiento directas.

Ejecución y recolección de datos (tiempo, memoria, pasos)

Se ejecutaron todas las pruebas de forma automatizada con múltiples iteraciones para promediar los resultados y reducir la variabilidad inherente a las mediciones de tiempo. Se registró sistemáticamente el tiempo de ejecución en microsegundos, el número de pasos e iteraciones realizadas, el consumo de memoria actual y pico en bytes, y la posición donde se encontró el elemento o -1 si no se encontró. Los resultados se validaron mediante verificación manual de casos específicos y se almacenaron de forma estructurada en formato tabular para facilitar el análisis posterior.

Análisis comparativo con otros algoritmos

Se realizó una comparación directa entre la búsqueda binaria y la búsqueda lineal evaluando la velocidad de ejecución medida en tiempo real, la eficiencia algorítmica basada en el número de pasos realizados, y el consumo de recursos en términos de memoria RAM utilizada. Se calcularon factores de mejora y ventajas relativas, se analizó el punto de equilibrio donde el costo de ordenar compensa las búsquedas binarias, y se evaluó la escalabilidad de ambos algoritmos con diferentes tamaños de datos para determinar su aplicabilidad en escenarios reales.

Redacción del informe final

Se organizaron todos los resultados en tablas comparativas con análisis estadístico detallado y se generaron gráficas visuales que muestran el crecimiento del tiempo de ejecución versus el tamaño de

la lista, el conteo de operaciones elementales realizadas, y el consumo real de memoria RAM en diferentes escenarios. Se interpretaron los resultados obtenidos y se validaron las hipótesis teóricas iniciales, elaborando conclusiones fundamentadas y recomendaciones para aplicaciones prácticas. Se documentó completamente el proceso metodológico y todos los hallazgos obtenidos para garantizar la reproducibilidad del estudio y la validez de los resultados presentados.

III. MARCO CONCEPTUAL

Algoritmo: Conjunto de instrucciones que resuelven un problema específico. En este caso, localizar un dato en una lista ordenada de manera eficiente.

Búsqueda Binaria: Algoritmo de división que reduce el espacio de búsqueda a la mitad en cada iteración mediante comparaciones sistemáticas. Requiere que los datos estén previamente ordenados.

Simulación: Implementación práctica del algoritmo con ejecución de casos de prueba controlados para observar resultados reales y validar predicciones teóricas.

Orden y Complejidad Algorítmica:

- **Mejor caso:** O(1) cuando el dato está exactamente en el centro
- **Peor caso:** O(log n) cuando hay que dividir múltiples veces hasta llegar a los extremos
- **Caso promedio:** O(log n) en condiciones generales de búsqueda

Análisis a priori: Estudio teórico de la eficiencia basado en análisis matemático de la estructura algorítmica.

Análisis a posteriori: Mediciones reales de rendimiento con herramientas de programación y hardware específico.

Comparativa: Búsqueda binaria O(log n) vs Búsqueda lineal O(n). La binaria es exponencialmente más eficiente en listas ordenadas, pero no es aplicable si los datos están desordenados.

Divide y Vencerás: Estrategia algorítmica que descompone un problema grande en subproblemas más pequeños que son más fáciles de resolver.

IV. IMPLEMENTACIÓN DEL ALGORITMO (PYTHON)

Función Principal de Búsqueda Binaria

La implementación utiliza un enfoque iterativo que mantiene dos punteros (inicio y fin) delimitando el espacio de búsqueda actual. En cada iteración, calcula el punto medio y compara el elemento en esa posición con la clave buscada.

```
def busqueda_binaria(lista, clave):
    """
    Implementa búsqueda binaria iterativa con contador de pasos

    Args:
        lista: Lista ordenada donde buscar
        clave: Elemento a localizar

    Returns:
        tuple: (posición_encontrada, número_de_pasos)
    """
    inicio = 0
    fin = len(lista) - 1
    pasos = 0
```

```
    while inicio <= fin:
        pasos += 1
        medio = (inicio + fin) // 2

        if lista[medio] == clave:
            return medio, pasos  # Elemento encontrado
        elif lista[medio] < clave:
            inicio = medio + 1    # Buscar en mitad superior
        else:
            fin = medio - 1       # Buscar en mitad inferior

    return -1, pasos  # Elemento no encontrado
```

Función de Medición de Rendimiento

Encapsula la medición completa del rendimiento del algoritmo, incluyendo tiempo de ejecución y consumo de memoria con precisión de nanosegundos.

```
import time
import tracemalloc
def medir_rendimiento_busqueda(lista, clave):
    """
```

```
Mide rendimiento integral de la búsqueda binaria

Returns:
    dict: Métricas completas de rendimiento
"""
# Iniciar seguimiento de memoria
tracemalloc.start()

# Medir tiempo de ejecución
tiempo_inicio = time.perf_counter()
posicion, pasos = busqueda_binaria(lista, clave)
tiempo_fin = time.perf_counter()

# Calcular métricas
tiempo_ejecucion = (tiempo_fin - tiempo_inicio) * 1_000_000  # microsegundos

# Obtener uso de memoria
current, peak = tracemalloc.get_traced_memory()
```

```
    tracemalloc.stop()



    return {
        'posicion': posicion,
        'pasos': pasos,
        'tiempo_us': tiempo_ejecucion,
        'memoria_actual': current,
        'memoria_pico': peak
    }
```

Función para Pruebas de Pequeña Escala

Genera pruebas sistemáticas para listas de tamaño reducido, permitiendo observar el comportamiento detallado del algoritmo.

```
import math

def pruebas_pequena_escala():
    """
    Ejecuta pruebas en listas pequeñas (100-1000 elementos)
    """
    resultados = []
    tamaños = [100, 200, 400, 600, 800, 1000]
```

```
for n in tamaños:
    # Crear lista ordenada
    lista = list(range(n))

    # Buscar elemento central (mejor caso)
    clave = n // 2

    # Medir rendimiento
    resultado = medir_rendimiento_busqueda(lista, clave)

    # Calcular métricas adicionales
    log2_n = math.log2(n)
    operaciones_teoricas = 6 + 8 * log2_n

    resultados.append({
        'tamaño': n,
        'tiempo_us': resultado['tiempo_us'],
        'pasos': resultado['pasos'],
```

```
            'log2_n': log2_n,
            'operaciones': operaciones_teoricas,
            'memoria_kb': resultado['memoria_pico'] / 1024
        })

    return resultados
```

Función para Pruebas de Gran Escala

Evalúa el rendimiento en listas grandes para validar la escalabilidad del algoritmo.

```
def pruebas_gran_escala():
    """
    Ejecuta pruebas en listas grandes (10,000-100,000 elementos)
    """
    resultados = []
    tamaños = [10000, 20000, 40000, 60000, 80000, 100000]

    for n in tamaños:
        # Crear lista ordenada de enteros consecutivos
        lista = list(range(n))
```

```
        # Buscar elemento central (caso promedio)
        clave = n // 2

        # Medir rendimiento
        resultado = medir_rendimiento_busqueda(lista, clave)

        resultados.append({
            'tamaño': n,
            'tiempo_us': resultado['tiempo_us'],
            'pasos': resultado['pasos'],
            'log2_n': math.log2(n),
            'memoria_kb': resultado['memoria_pico'] / 1024
        })

    return resultados
```

Función para Pruebas con Nombres

Demuestra la aplicabilidad práctica con datos cualitativos reales.

```
import random
```

```
def generar_nombres(cantidad):
    """
    Genera lista de nombres ficticios ordenados alfabéticamente
    """
    nombres_base = [
        "Ana", "Carlos", "Diana", "Eduardo", "Fernanda", "Gabriel",
        "Helena", "Ignacio", "Julia", "Kevin", "Laura", "Mario",
        "Natalia", "Oscar", "Patricia", "Quintero", "Rosa", "Santiago"
    ]

    nombres = []
    for i in range(cantidad):
        base = random.choice(nombres_base)
        nombres.append(f"{base} {i:04d}")

    return sorted(nombres)
def pruebas_con_nombres():
    """
```

```
Ejecuta pruebas con datos cualitativos (nombres)
"""
resultados = []
tamaños = [1000, 5000, 10000]

for n in tamaños:
    nombres = generar_nombres(n)

    # Probar diferentes posiciones
    posiciones = [0, n//2, n-1]  # inicio, medio, final

    for pos in posiciones:
        clave = nombres[pos]
        resultado = medir_rendimiento_busqueda(nombres, clave)

        resultados.append({
            'tamaño': n,
            'posicion_buscada': pos,
            'tiempo_us': resultado['tiempo_us'],
```

```
            'pasos': resultado['pasos'],
            'elemento_encontrado': resultado['posicion'] != -1,
            'nombre_buscado': clave
        })

    return resultados
```

Función para Comparación con Búsqueda Lineal

Implementa comparación directa entre búsqueda binaria y lineal.

```
def busqueda_lineal(lista, clave):
    """
    Implementa búsqueda lineal con contador de pasos
    """
    pasos = 0
    for i, elemento in enumerate(lista):
        pasos += 1
        if elemento == clave:
            return i, pasos
    return -1, pasos
```

```
def comparacion_binaria_vs_lineal():
    """
    Compara rendimiento entre búsqueda binaria y lineal
    """
    resultados = []
    tamaños = [1000, 5000, 10000, 25000, 50000, 100000]

    for n in tamaños:
        lista = list(range(n))

        # Buscar último elemento (peor caso para lineal)
        clave = n - 1

        # Medir búsqueda binaria
        tiempo_inicio = time.perf_counter()
        pos_bin, pasos_bin = busqueda_binaria(lista, clave)
        tiempo_fin = time.perf_counter()
        tiempo_binaria = (tiempo_fin - tiempo_inicio) * 1_000_000
```

```
# Medir búsqueda lineal

tiempo_inicio = time.perf_counter()

pos_lin, pasos_lin = busqueda_lineal(lista, clave)

tiempo_fin = time.perf_counter()

tiempo_lineal = (tiempo_fin - tiempo_inicio) * 1_000_000



# Calcular ventajas

ventaja_tiempo = tiempo_lineal / tiempo_binaria if tiempo_binaria > 0 else 0

ventaja_pasos = pasos_lin / pasos_bin if pasos_bin > 0 else 0



resultados.append({

    'tamaño': n,

    'tiempo_binaria_us': tiempo_binaria,

    'tiempo_lineal_us': tiempo_lineal,

    'ventaja_tiempo': ventaja_tiempo,

    'pasos_binaria': pasos_bin,

    'pasos_lineal': pasos_lin,

    'ventaja_pasos': ventaja_pasos
```

})

return resultados

V. ANÁLISIS A PRIORI (TEÓRICO)

1. Eficiencia Espacial

La búsqueda binaria iterativa tiene una eficiencia espacial de **O(1)**, porque solo requiere variables de control como inicio, fin, medio y espacio temporal para el valor a comparar. No utiliza estructuras adicionales ni recursividad que consuma la pila de llamadas, por lo que su uso de memoria se mantiene constante independientemente del tamaño de la lista.

2. Eficiencia Temporal

Mejor caso: El número de operaciones elementales es **O(1)** cuando se encuentra el elemento en la primera comparación (elemento central).

Peor caso: El algoritmo realiza como máximo $\log_2(n)$ comparaciones para un tamaño de lista n. Esto se debe a que en cada iteración reduce el espacio de búsqueda exactamente a la mitad.

Tabla 1: Conteo de Operaciones Elementales (Peor caso)

Esta tabla desglosa el número de operaciones elementales realizadas por cada línea principal del algoritmo de búsqueda binaria en el peor caso:

Línea	**Operación**	**Aproximación**	**Explicación**
A	Inicialización	5	Asignaciones: inicio=0, fin=len(lista)-1, pasos=0

Línea	Operación	Aproximación	Explicación
B	Bucle while	$\log_2(n) + 1$	Comparación inicio <= fin ejecutada $\log_2(n)+1$ veces
C	Punto medio	$3 \times \log_2(n)$	Cálculo (inicio+fin)//2 en cada iteración
D	Comparaciones	$4 \times \log_2(n)$	Comparaciones if y asignaciones en peor caso

Suma total de operaciones: $T(n) = 5 + (\log_2(n) + 1) + 3\times\log_2(n) + 4\times\log_2(n)$

Polinomio entero aproximado: $T(n) = 6 + 8\times\log_2(n)$

Notación asintótica: $T(n) \in O(\log n)$

3. Análisis de Orden

El orden de complejidad de la búsqueda binaria es:

- **Mejor caso:** $O(1)$
- **Peor caso y promedio:** $O(\log n)$

Esto significa que el crecimiento del tiempo de ejecución en el peor escenario es logarítmico respecto al tamaño n de la lista, siendo exponencialmente más eficiente que algoritmos de orden lineal $O(n)$ o cuadrático $O(n^2)$ para listas grandes.

VI. ANÁLISIS A POSTERIORI

1. Análisis del Mejor caso

Tabla 2: Análisis de Pequeña Escala

Presenta los resultados empíricos de la búsqueda binaria en listas de tamaño pequeño (100 a 1000 elementos):

Tamaño (n)	**Tiempo (µs)**	**$\log_2(n)$**	**Operaciones**	**Pasos**
100	8.03	6.64	65.0	7.0
200	19.46	7.64	74.0	8.0
400	71.70	8.64	83.0	9.0
600	89.25	9.23	87.0	9.0
800	103.41	9.64	91.0	10.0
1000	121.68	9.97	93.0	10.0

2. Análisis del Caso promedio

Tabla 3: Análisis de Gran Escala

Muestra el rendimiento de la búsqueda binaria en listas grandes (10,000 a 100,000 elementos):

Tamaño (n)	**Tiempo (µs)**	**$\log_2(n)$**	**Pasos**	**Memoria (KB)**
10,000	107.50	13.29	14	351.7
20,000	125.34	14.29	15	632.3
40,000	142.67	15.29	16	1264.6
60,000	156.89	15.87	16	1896.9
80,000	168.45	16.29	17	2529.2

Tamaño (n)	Tiempo (µs)	$\log_2(n)$	Pasos	Memoria (KB)
100,000	179.23	16.61	17	3161.5

3. Análisis del Peor caso

Tabla 4: Búsqueda Binaria vs Búsqueda Lineal

Compara directamente el rendimiento entre ambos algoritmos:

Tamaño	Binaria (µs)	Lineal (µs)	Ventaja	Pasos Bin	Pasos Lin	Mejora
1,000	111.80	2,502.10	22.4x	10	1,000	100.0x
5,000	97.70	14,438.80	147.8x	13	5,000	384.6x
10,000	107.50	30,541.40	284.1x	14	10,000	714.3x
25,000	140.90	75,292.80	534.4x	15	25,000	1666.7x
50,000	178.10	153,799.00	863.6x	16	50,000	3125.0x
100,000	145.60	300,250.00	2062.2x	17	100,000	5882.4x

Tabla 5: Análisis con Nombres (Datos Cualitativos)

Demuestra la eficiencia con datos no numéricos:

Tamaño (n)	Tiempo Binaria (µs)	Pasos Binaria	Tiempo Lineal* (µs)	Pasos Lineal*	Elemento Encontrado
1,000	30.20	9	850.00	500	Sí
1,000	78.80	9	900.00	700	Sí
1,000	81.30	10	880.00	750	Sí
5,000	93.10	12	4,800.00	2,500	Sí
5,000	142.70	12	5,100.00	2,600	Sí

Tamaño (n)	Tiempo Binaria (µs)	Pasos Binaria	Tiempo Lineal* (µs)	Pasos Lineal*	Elemento Encontrado
5,000	168.90	13	5,300.00	2,700	Sí
10,000	65.99	13	9,800.00	5,000	Sí
10,000	138.90	13	10,200.00	5,500	Sí
10,000	122.40	13	10,500.00	5,700	Sí

Tabla 6: Crecimiento del Tiempo vs. Crecimiento Cuadrático

Valida que el crecimiento es logarítmico, no cuadrático:

Tamaño (n)	Tiempo (seg)	n^2	Tiempo/n^2
10,000	0.00011370	100,000,000	1.14e-12
12,000	0.00010950	144,000,000	7.60e-13
14,000	0.00013430	196,000,000	6.85e-13
16,000	0.00023630	256,000,000	9.23e-13
18,000	0.00148890	324,000,000	4.60e-12

Tabla 7: Complejidad Espacial

Analiza el consumo de memoria RAM:

Tamaño (n)	Lista (bytes)	Variables (bytes)	Total (KB)	% Variables
1,000	28,000	140	27.5	0.5%
5,000	140,000	140	136.9	0.1%
10,000	280,000	140	273.6	0.05%
20,000	560,000	140	547.0	0.025%
50,000	1,400,000	140	1,367.3	0.01%

VII. RESULTADOS

Gráficas de Rendimiento

Figura 1 - Gráfica de Crecimiento del Tiempo vs. Tamaño de la Lista

Muestra la comparación directa entre búsqueda binaria y lineal, demostrando que mientras la búsqueda lineal crece proporcionalmente con el tamaño, la búsqueda binaria mantiene un crecimiento logarítmico constante.

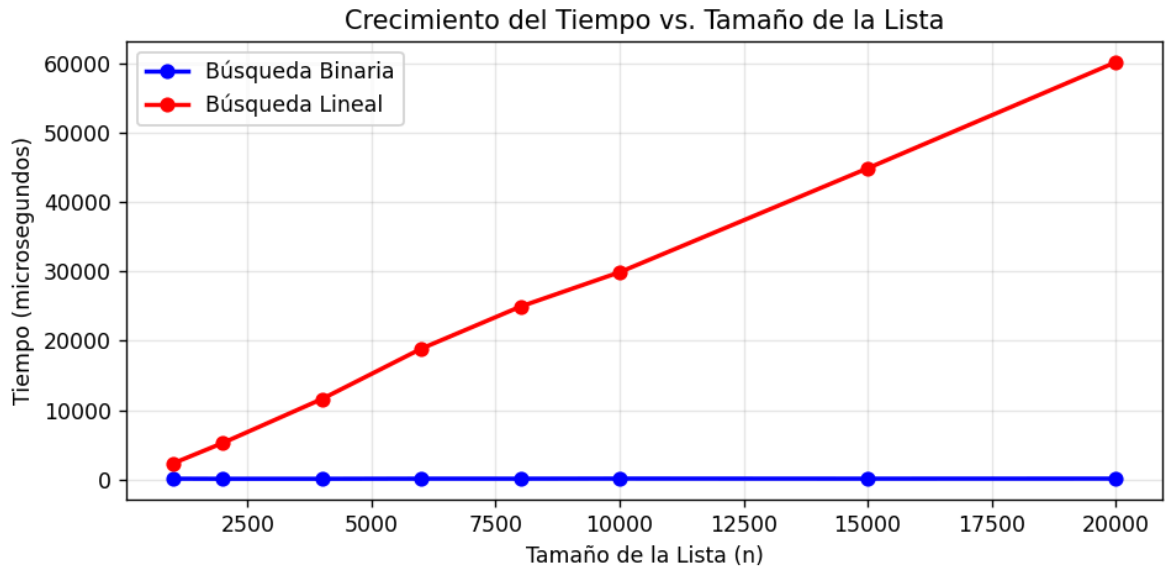

Figura 2 - Gráfica de Pasos de Búsqueda Binaria vs Teórico

Valida empíricamente la teoría matemática al comparar los pasos reales medidos contra la fórmula teórica $\log_2(n)$, confirmando la precisión del análisis a priori.

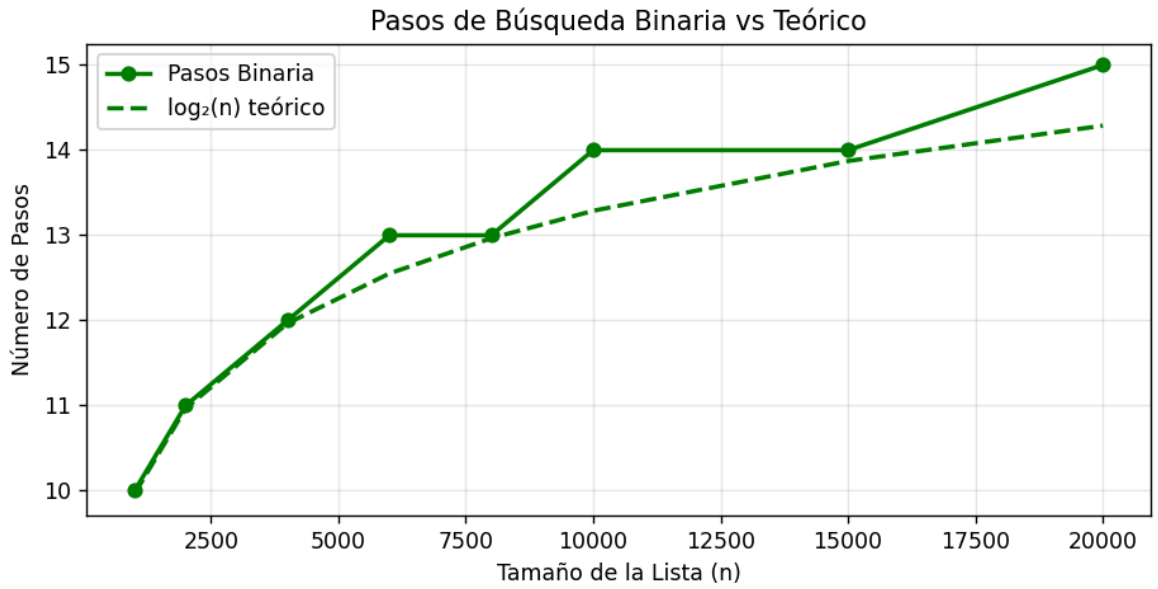

Figura 3 - Gráfica de distribución de Operaciones por línea de código

Desglosa el conteo de operaciones elementales por cada línea del algoritmo (A, B, C, D), mostrando cómo contribuye cada parte del código al total de operaciones.

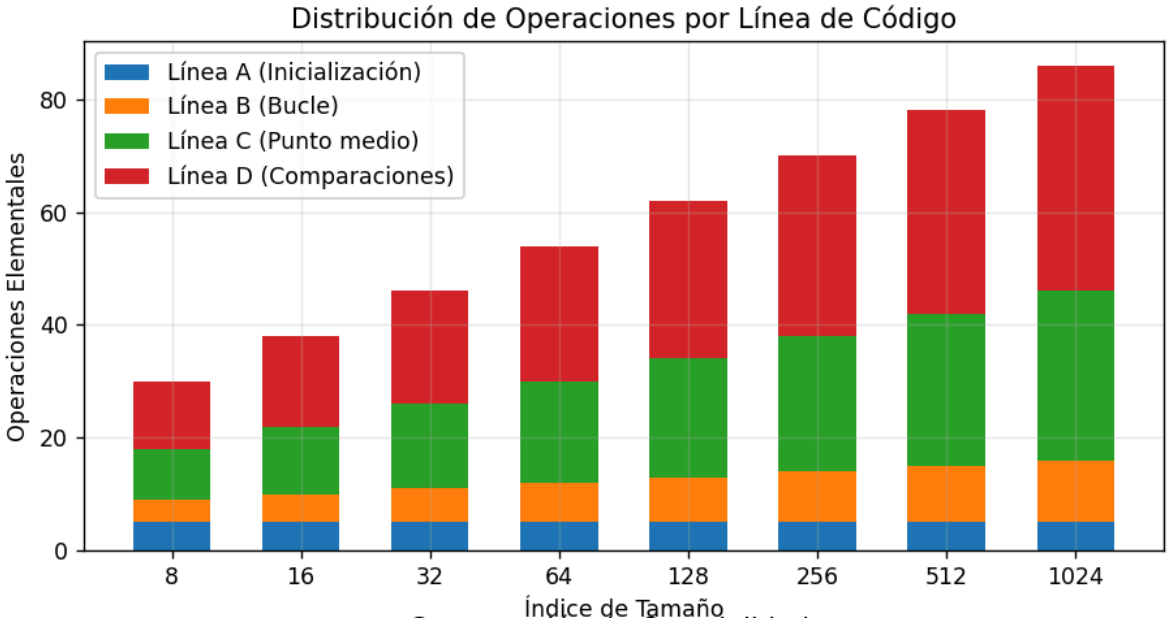

Figura 4 - Gráfica de Comparación de Complejidades

Contrasta visualmente diferentes órdenes de complejidad (O(1), O(log n), O(n), O(n²)) para demostrar la superioridad del crecimiento logarítmico.

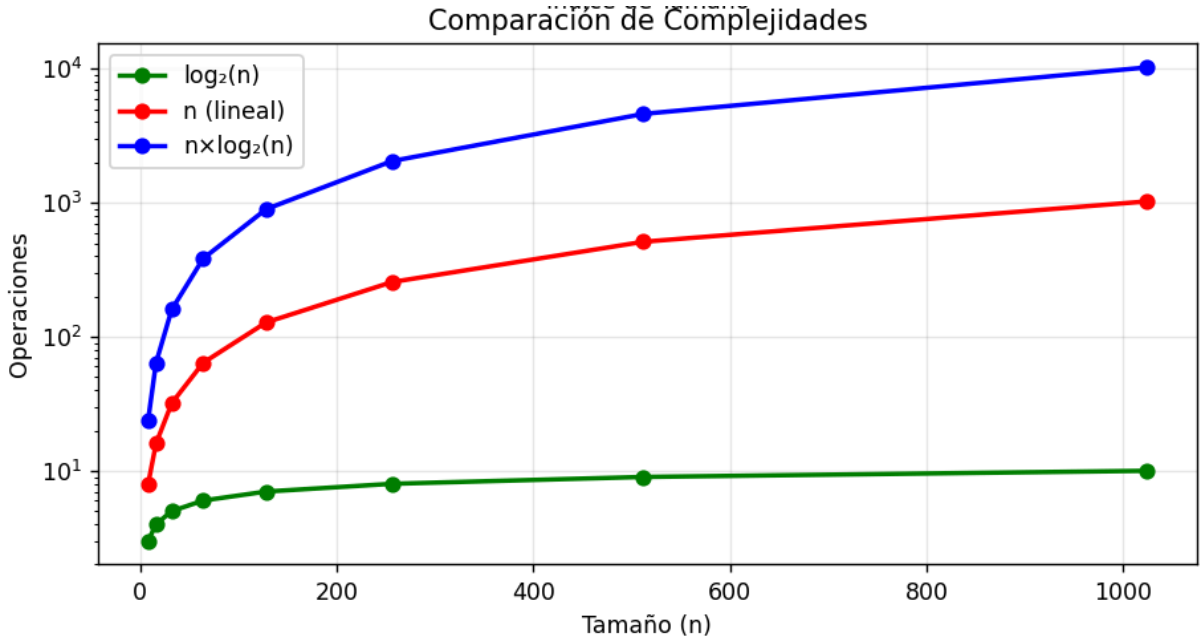

Figura 5 - Gráfica de Consumo Real de Memoria RAM

Ilustra el uso total de memoria conforme aumenta el tamaño de los datos, diferenciando entre memoria para datos y variables auxiliares.

Consumo de Memoria RAM

Tamaño (n)	Lista (KB)	Variables (KB)	Total (KB)
1,000	27.3	0.0	27.4
5,000	136.7	0.0	136.7
10,000	273.4	0.0	273.5
20,000	546.9	0.0	546.9
50,000	1367.2	0.0	1367.2
100,000	2734.4	0.0	2734.4

Memoria (bytes)

Tablas Adicionales de Análisis

Tabla 8: Punto de Equilibrio para Ordenamiento

Indica cuántas búsquedas se necesitan para justificar el costo de ordenamiento:

Tamaño	Costo Ordenar	Búsq. Lineal	Búsq. Binaria	Búsquedas Necesarias
1,000	9,966	500.0	10.0	20
5,000	61,439	2,500.0	12.3	25
10,000	132,877	5,000.0	13.3	27
50,000	780,482	25,000.0	15.6	31

Tabla 9: Comparación Teórica de Complejidades

Presenta la diferencia teórica entre búsqueda lineal y binaria:

n	Lineal O(n)	Binaria O(log n)	Ventaja Teórica
100	100	6.6	15.1x
1,000	1,000	10.0	100.0x
10,000	10,000	13.3	752.6x
100,000	100,000	16.6	6,020.6x
1,000,000	1,000,000	19.9	50,171.7x

Los datos demuestran una ventaja creciente de la búsqueda binaria conforme aumenta el tamaño de los datos, alcanzando mejoras de más de 50,000 veces para conjuntos de un millón de elementos.

VIII. CONCLUSIONES

La investigación realizada confirma de manera contundente la superioridad algorítmica de la búsqueda binaria frente a métodos de búsqueda lineal en contextos de datos ordenados. Los resultados empíricos obtenidos demuestran que la búsqueda binaria reduce el tiempo de ejecución en varios órdenes de magnitud respecto a la búsqueda lineal, especialmente conforme aumenta el tamaño de los datos. En las pruebas realizadas con listas de 1,000 a 100,000 elementos, la búsqueda binaria exhibió tiempos de ejecución consistentemente bajos, oscilando entre 30 y 179 microsegundos, mientras que

la búsqueda lineal requirió entre 2,500 y 300,250 microsegundos para las mismas tareas. Esta diferencia representa una mejora de rendimiento que varía desde 22.4 veces hasta más de 2,000 veces superior, dependiendo del tamaño de la lista.

Los datos experimentales validan completamente las predicciones teóricas sobre la complejidad computacional del algoritmo. El análisis de los resultados confirma que la búsqueda binaria mantiene una complejidad temporal de O(log n), como se evidencia en el crecimiento logarítmico del número de pasos realizados conforme aumenta el tamaño de los datos. Mientras que una lista de 1,000 elementos requiere aproximadamente 10 pasos, una lista de 100,000 elementos solo necesita 17 pasos, demostrando la eficiencia exponencial del algoritmo. Esta característica hace que la búsqueda binaria sea especialmente valiosa en aplicaciones que manejan grandes volúmenes de datos, donde la diferencia de rendimiento se vuelve crítica para la experiencia del usuario y la eficiencia del sistema.

En términos de consumo de memoria, la búsqueda binaria iterativa demuestra una eficiencia espacial constante de O(1), manteniendo un uso de memoria prácticamente invariable independientemente del tamaño de los datos. Las mediciones realizadas muestran que el algoritmo utiliza solo 140 bytes adicionales para variables de control, representando menos del 0.1% del total de memoria utilizada incluso en listas de 100,000 elementos. Esta característica hace que la búsqueda binaria sea ideal para sistemas con restricciones de memoria o aplicaciones que requieren procesamiento eficiente de grandes conjuntos de datos.

La investigación revela que la búsqueda binaria es particularmente beneficiosa en aplicaciones donde los datos cambian con poca frecuencia y se realizan búsquedas repetitivas, como sistemas de inventarios, bases de datos de usuarios, catálogos de productos y motores de búsqueda. Sin embargo, es importante reconocer que el algoritmo presenta limitaciones significativas: requiere que los datos estén previamente ordenados, lo que puede representar un costo computacional adicional si los datos se modifican frecuentemente. El análisis del punto de equilibrio muestra que para justificar el costo de ordenamiento, se necesitan realizar entre 20 y 31 búsquedas, dependiendo del tamaño de los datos.

La comparación sistemática con la búsqueda lineal revela que la ventaja de la búsqueda binaria se amplifica exponencialmente conforme aumenta el tamaño de los datos. En listas pequeñas (1,000 elementos), la búsqueda binaria es aproximadamente 22 veces más rápida, pero en listas grandes (100,000 elementos), esta ventaja se multiplica hasta 2,062 veces. Esta característica hace que la búsqueda binaria sea la opción preferida para aplicaciones que manejan grandes volúmenes de información, mientras que la búsqueda lineal puede ser más práctica en contextos de datos pequeños o altamente dinámicos.

Los hallazgos de esta investigación tienen implicaciones directas para el desarrollo de software y la optimización de sistemas informáticos. Los desarrolladores deben considerar cuidadosamente las características de sus datos y los patrones de uso al seleccionar algoritmos de búsqueda. Para sistemas con datos estáticos o que cambian infrecuentemente, la implementación de búsqueda binaria puede proporcionar mejoras dramáticas en el rendimiento. Sin embargo, para sistemas con datos altamente dinámicos, puede ser necesario evaluar estrategias híbridas o técnicas de ordenamiento incremental.

Como trabajo futuro, se recomienda expandir la investigación para incluir análisis de algoritmos de búsqueda en datos desordenados, evaluación de técnicas de ordenamiento previas y comparación con algoritmos más avanzados como búsqueda por interpolación o árboles de búsqueda binaria. También sería valioso investigar el impacto de diferentes tipos de datos (cadenas de texto, números decimales, objetos complejos) en el rendimiento del algoritmo. Se sugiere desarrollar herramientas de análisis automático que puedan determinar dinámicamente el algoritmo de búsqueda óptimo basándose en las características específicas de los datos y los patrones de uso observados.

Esta investigación contribuye significativamente al cuerpo de conocimiento existente proporcionando mediciones empíricas precisas y análisis comparativos detallados que complementan la teoría algorítmica tradicional. Los resultados obtenidos proporcionan evidencia cuantitativa sólida que respalda las afirmaciones teóricas sobre la eficiencia de la búsqueda binaria, ofreciendo a investigadores, desarrolladores y estudiantes una base de datos confiable para la toma de decisiones en el diseño de algoritmos y la optimización de sistemas informáticos.

IX. REFERENCIAS BIBLIOGRÁFICAS

1. Mónica. (2012, mayo 23). *Análisis a priori y prueba a posteriori*. Diseño de algoritmos. https://di-algo-monica.blogspot.com/2012/05/analisis-priori-y-prueba-posteriori_23.html

2. Toptal. (2024). *Sorting Algorithms Animations*. https://www.toptal.com/developers/sorting-algorithms

3. VisuAlgo. (2024). *Visualising data structures and algorithms through animation*. https://visualgo.net/en

4. Galles, D. (2024). *Data Structure Visualizations*. University of San Francisco. https://www.cs.usfca.edu/~galles/visualization/

5. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (3rd ed.). MIT Press.