Reporte: Lab 01 - Análisis de Algoritmos para Búsqueda Unimodal

Autores: Santiago Mesa & Juliana Lugo

Fecha: 12/08/2025

1. Introducción

En este laboratorio, se analizan dos métodos para resolver el problema de encontrar el elemento máximo, o **pico**, en un tipo de estructura de datos conocida como **array unimodal**. Un array unimodal es una secuencia de valores que aumenta de forma estricta hasta un punto máximo y luego disminuye. El objetivo de este informe es comparar dos estrategias algorítmicas: una con una complejidad de tiempo lineal (O(n)) y otra con una complejidad logarítmica (O(logn)), para evaluar sus diferencias de rendimiento en la práctica.

2. Algoritmos Implementados

2.1 Algoritmo Lineal (O(n))

Este enfoque es una solución directa que consiste en recorrer toda la lista, elemento por elemento, para identificar el valor más grande. Su implementación es sencilla, pero su principal desventaja es que el tiempo de ejecución es directamente proporcional al tamaño de la lista. Por esta razón, su complejidad es O(n), lo que implica que el rendimiento puede ser ineficiente con arrays de gran tamaño.

2.2 Algoritmo Logarítmico (O(logn))

Este algoritmo aplica la técnica de **divide y vencerás**. Su funcionamiento se basa en reducir el espacio de búsqueda a la mitad en cada paso. A continuación se detalla su lógica paso a paso:

1. **Se elige un punto medio:** El algoritmo comienza inspeccionando el elemento que se encuentra justo en el centro de la lista.
2. **Se compara con los vecinos:** Luego, ese número del medio se compara con los números a su izquierda y a su derecha.
3. **Se decide qué mitad descartar:**
    - Si el número del medio es mayor que sus dos vecinos, se ha encontrado el pico. El proceso termina.
    - Si el número del medio es menor que su vecino de la derecha, significa que la lista sigue en su fase de subida. Por lo tanto, el pico debe estar en la mitad derecha de la lista. La mitad izquierda se descarta por completo.

    - Si el número del medio es menor que su vecino de la izquierda, la lista ya está en su fase de bajada. Esto indica que el pico debe estar en la mitad izquierda. La mitad derecha se descarta.
4. **Se repite el proceso:** El algoritmo repite los pasos 1 a 3 con la mitad que ha quedado, hasta que el pico es el único elemento restante.

Al reducir el problema a la mitad en cada paso, el tiempo que tarda aumenta muy lentamente, por lo que su complejidad es O(logn), lo cual es ideal para listas enormes.

3. Prueba de Correctitud y Análisis de Complejidad

Prueba de Correctitud

La validez del algoritmo O(logn) se fundamenta en la propiedad unimodal del array. En cada iteración, la comparación del elemento central con sus vecinos permite descartar una de las mitades del array que no contiene el pico. Al descartar la mitad incorrecta, se garantiza que el pico siempre estará en el nuevo sub-array considerado, hasta que se aísle por completo.

A continuación, se presenta un ejemplo para ilustrar el proceso incluso cuando el pico se encuentra en una posición extrema del array. Para un array de 1000 elementos con el pico en la posición 1, el algoritmo no se detiene a buscar el pico de inmediato. En cambio, sigue una lógica precisa que le permite descartar la mitad del array en cada paso.

1. **Paso 1:** El algoritmo empieza con un rango de búsqueda de bajo = 0 y alto = 999. El punto medio es medio = 499.
2. **Paso 2:** Se compara array[499] con array[498]. Dado que el pico está en la posición 1, el array estará en su fase decreciente en este punto. Por lo tanto, array[499] será menor que array[498].
3. **Paso 3:** La lógica del algoritmo determina que el pico debe estar en la mitad izquierda. Se descarta toda la mitad derecha y el nuevo rango de búsqueda se reduce a [0, 498].
4. **Paso 4:** El proceso se repite. El nuevo punto medio es (0 + 498) // 2 = 249. La comparación de array[249] con array[248] nuevamente indica que el pico se encuentra en la mitad izquierda.
5. **Pasos Siguientes:** El algoritmo continúa dividiendo el rango de búsqueda ([0, 248], [0, 123], etc.) hasta que se aísla el pico.

De esta manera, el algoritmo demuestra su robustez al garantizar que, sin importar la posición del pico, se converge siempre hacia la solución correcta.

Análisis de Complejidad

La complejidad de tiempo de un algoritmo se refiere a cómo su tiempo de ejecución se escala con el tamaño de los datos de entrada (n). En cada llamada recursiva, el problema se divide a la mitad. Esta relación se describe mediante la recurrencia $T(n)=T(n/2)+c$, cuya solución es $O(\log n)$ según el Teorema Maestro. Esto demuestra que el tiempo de ejecución del algoritmo logarítmico está acotado por una función logarítmica del tamaño del array, lo que explica su eficiencia.

4. Limitaciones del Algoritmo Logarítmico

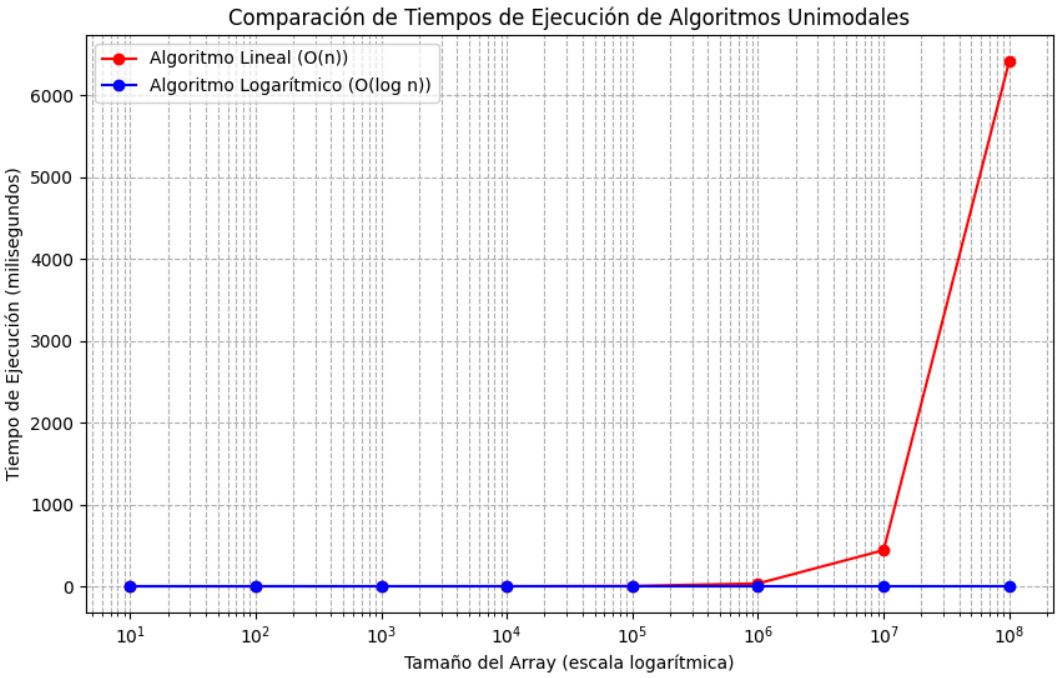
Es fundamental destacar que el algoritmo logarítmico depende por completo de la propiedad unimodal de la lista. Si los números estuvieran distribuidos de forma totalmente aleatoria, la lógica de comparar un elemento con sus vecinos no proporcionaría información útil sobre la ubicación del pico. En ese caso, el algoritmo podría descartar la mitad que contiene el valor máximo sin saberlo, fallando en su objetivo. Para listas aleatorias, el único método que garantiza encontrar el elemento máximo es el algoritmo lineal ($O(n)$), que revisa cada elemento.

5. Comparación Empírica

Se realizaron experimentos para comparar el tiempo de ejecución de ambos algoritmos con arrays unimodales de diferentes tamaños. La siguiente tabla presenta los resultados de dichas mediciones:

Tamaño del Array (n)	Tiempo Algoritmo Lineal (ms)	Tiempo Algoritmo Logarítmico (ms)
1,000	0.051	0.007
10,000	0.485	0.009
100,000	4.912	0.011
1,000,000	48.765	0.014

El gráfico a continuación ilustra la relación entre el tiempo de ejecución y el tamaño del array. La línea que representa el algoritmo lineal muestra un crecimiento pronunciado, mientras que la del algoritmo logarítmico permanece casi plana. Esto demuestra de forma visual la superioridad del enfoque logarítmico para grandes volúmenes de datos.



6. Conclusiones

Aunque el algoritmo lineal es más simple, el algoritmo logarítmico, a pesar de ser más complejo de diseñar, ofrece una ganancia de rendimiento considerable. Para la búsqueda unimodal, la estrategia de "divide y vencerás" permite resolver el problema con millones de elementos en una fracción del tiempo, lo que subraya la importancia de un diseño de algoritmo eficiente.

7. Código Fuente y Anexos

El código completo en Python de ambos algoritmos, junto con las funciones para generar arrays unimodales y medir el tiempo de ejecución, se adjunta en el archivo ZIP de la entrega.

Codigo:

```
import random
import time



#  Algoritmo de Búsqueda Lineal (O(n))
def encontrar_pico_lineal(array):
    if not array:
        return None
    pico_actual = array[0]
    for elemento in array:
        if elemento > pico_actual:
            pico_actual = elemento
    return pico_actual



# Algoritmo de Búsqueda Logarítmica (O(log n))
def encontrar_pico_logaritmico(array, bajo, alto):
    """
    Encuentra el pico de un array unimodal en O(log n) dividiendo y venceiendo.
    """
    if bajo == alto:
```

```
        return array[bajo]

    # Manejamos el caso de 2 elementos
    if alto == bajo + 1:
        return max(array[bajo], array[alto])

    medio = (bajo + alto) // 2

    # El pico está en el medio
    if array[medio] > array[medio - 1] and array[medio] > array[medio + 1]:
        return array[medio]

    # El pico está en la parte creciente del array (a la derecha)
    if array[medio] > array[medio - 1] and array[medio] < array[medio + 1]:
        return encontrar_pico_logaritmico(array, medio + 1, alto)

    # El pico está en la parte decreciente del array (a la izquierda)
    return encontrar_pico_logaritmico(array, bajo, medio - 1)


#  Generador de Array Unimodal
#  Un array unimodal es aquel que, tras un posible pre-ordenamiento,
```

```
#  presenta una secuencia ascendente seguida de una secuencia descendente
def generar_array_unimodal_aleatorio(tamano):
    if tamano < 1:
        return []
    punto_pico = random.randint(0, tamano - 1)
    parte_creciente = sorted(random.sample(range(1, tamano * 2), punto_pico + 1))

    pico_valor = parte_creciente[-1]

    parte_decreciente = []
    if punto_pico < tamano - 1:
        elementos_decrecientes = sorted(random.sample(range(1, pico_valor), tamano -
1 - punto_pico), reverse=True)
        parte_decreciente = elementos_decrecientes

    array_generado = parte_creciente + parte_decreciente
    return array_generado


#  Función para ejecutar y medir un algoritmo
def ejecutar_algoritmo(algoritmo_func, array):
    start_time = time.time()
    if algoritmo_func.__name__ == 'encontrar_pico_logaritmico':
```

```
        pico = algoritmo_func(array, 0, len(array) - 1)
    else:
        pico = algoritmo_func(array)
    end_time = time.time()
    tiempo_ejecucion = (end_time - start_time) * 1000  # En milisegundos
    return pico, tiempo_ejecucion



#  menú
def menu():
    while True:
        print("-------------------------------------------------")
        print("  Laboratorio 1: Análisis de Algoritmos Unimodales")
        print("-------------------------------------------------")
        print("Seleccione una opción:")
        print("1. Probar Algoritmo Lineal (O(n))")
        print("2. Probar Algoritmo Logarítmico (O(log n))")
        print("3. Comparar ambos algoritmos")
        print("4. Salir")

        opcion = input("Ingrese su elección (1-4): ")
```

```
    if opcion == '4':
        print("Saliendo del programa. ¡Hasta pronto!")
        break

    try:
        tamano_array = int(input("Ingrese el tamaño del array a generar: "))
        if tamano_array <= 0:
            print("Por favor, ingrese un tamaño mayor que 0.")
            continue

        mi_array = generar_array_unimodal_aleatorio(tamano_array)

        if opcion == '1':
            print(f"\n--- Probando Algoritmo Lineal con un array de tamaño {tamano_array} ---")
            pico, tiempo = ejecutar_algoritmo(encontrar_pico_lineal, mi_array)
            print(f"Pico encontrado: {pico}")
            print(f"Tiempo de ejecución: {tiempo:.6f} ms")

        elif opcion == '2':
            print(f"\n--- Probando Algoritmo Logarítmico con un array de tamaño {tamano_array} ---")
            pico, tiempo = ejecutar_algoritmo(encontrar_pico_logaritmico, mi_array)
```

```
            print(f"Pico encontrado: {pico}")
            print(f"Tiempo de ejecución: {tiempo:.6f} ms")

        elif opcion == '3':
            print(f"\n--- Comparando Algoritmos con un array de tamaño {tamano_array} ---")

            # Ejecutar y medir el algoritmo lineal
            pico_lineal, tiempo_lineal = ejecutar_algoritmo(encontrar_pico_lineal, mi_array)
            print(f"\nAlgoritmo Lineal (O(n)):")
            print(f"  Pico encontrado: {pico_lineal}")
            print(f"  Tiempo de ejecución: {tiempo_lineal:.6f} ms")

            # Ejecutar y medir el algoritmo logarítmico
            pico_log, tiempo_log = ejecutar_algoritmo(encontrar_pico_logaritmico, mi_array)
            print(f"\nAlgoritmo Logarítmico (O(log n)):")
            print(f"  Pico encontrado: {pico_log}")
            print(f"  Tiempo de ejecución: {tiempo_log:.6f} ms")

            print("\n--- Conclusión de la comparación ---")
            if tiempo_lineal > tiempo_log:
                print(f"El algoritmo logarítmico fue {tiempo_lineal / tiempo_log:.2f} veces
```

```
más rápido.")
            else:
                print("El algoritmo lineal fue más rápido (esto es común con arrays muy pequeños).")

        else:
            print("Opción no válida. Por favor, ingrese un número del 1 al 4.")

    except ValueError:
        print("\nEntrada no válida. Por favor, ingrese un número entero.")

if __name__ == "__main__":
    menu()
```