Reporte de Laboratorio 1: Comparación de Algoritmos de Búsqueda

Juan Diego Reyes

February 11, 2025

1 Introducción

La búsqueda de elementos dentro de una estructura de datos es una operación fundamental en el ámbito de la informática, con aplicaciones en bases de datos, motores de búsqueda y optimización de algoritmos. Existen distintos enfoques para realizar esta tarea, cada uno con ventajas y desventajas en términos de eficiencia computacional.

En este trabajo se comparan dos estrategias para la búsqueda de un elemento dentro de un conjunto ordenado de datos: la *búsqueda ingenua* (naive) y la *búsqueda binaria*. La búsqueda naive recorre secuencialmente el conjunto hasta encontrar el elemento deseado, lo que resulta poco eficiente en grandes volúmenes de datos. En contraste, la búsqueda binaria aprovecha la estructura ordenada del conjunto y reduce el espacio de búsqueda a la mitad en cada iteración, logrando un mejor desempeño en términos de complejidad computacional.

El objetivo de este reporte es analizar el rendimiento de ambas estrategias, implementarlas en Python y realizar experimentos para evaluar su eficiencia empíricamente. Finalmente, se presentan los resultados obtenidos y se discute cuál de los algoritmos es más adecuado según el contexto de uso.

2 Algoritmos Implementados

Para llevar a cabo la comparación, se implementaron dos algoritmos de búsqueda en Python.

2.1 Búsqueda Naive

Este algoritmo recorre el arreglo de manera secuencial, verificando elemento por elemento hasta encontrar el valor buscado. Su complejidad temporal es de $O(n)$, lo que significa que su rendimiento decrece linealmente conforme aumenta el tamaño del conjunto de datos. A continuación, se muestra la implementación en Python:

```
1
2  from generate_unimodal_array import generar_arreglo_unimodal
3
4  def naive(arr: list, start = 0, end = None):
5      for i in range(1, len(arr) - 1):
6          if arr[i] > arr[i - 1] and arr[i] > arr[i + 1]:
7              print(f"naive:: i: {i} val: {arr[i]}")
8              return i
9      return -1
10
11 def run(arr: list):
12     naive(arr)
13
14
15 if __name__ == "__main__":
```

```
    run(100, [1,2,3])
```

2.2 Búsqueda Binaria

Este algoritmo aprovecha la estructura ordenada del conjunto de datos, comenzando la búsqueda desde el elemento central. Si el valor buscado no se encuentra en el punto medio, el algoritmo reduce el espacio de búsqueda a la mitad y repite el proceso hasta encontrar el elemento o determinar su ausencia. Su complejidad temporal es de $O(\log n)$, lo que lo hace significativamente más eficiente que la búsqueda naive en conjuntos grandes. A continuación, se presenta su implementación en Python:

```python
from generate_unimodal_array import generar_arreglo_unimodal

def binary(arr: list):
    start, end = 0, len(arr) - 1
    while start <= end:
        mid = (start + end) // 2
        if arr[mid] > arr[mid - 1] and arr[mid] > arr[mid + 1]:
            print(f"binary:: i: {mid} val: {arr[mid]}")
            return mid
        elif arr[mid] < arr[mid - 1]:
            end = mid - 1
        else:
            start = mid + 1
    return -1  # Si no se encuentra un pico, aunque eso no deber a ocurrir en un arreglo adecuado


def run(arr: list):
    binary(arr)

if __name__ == "__main__":
    run(100, [1,2,3])
```

3 Experimentos

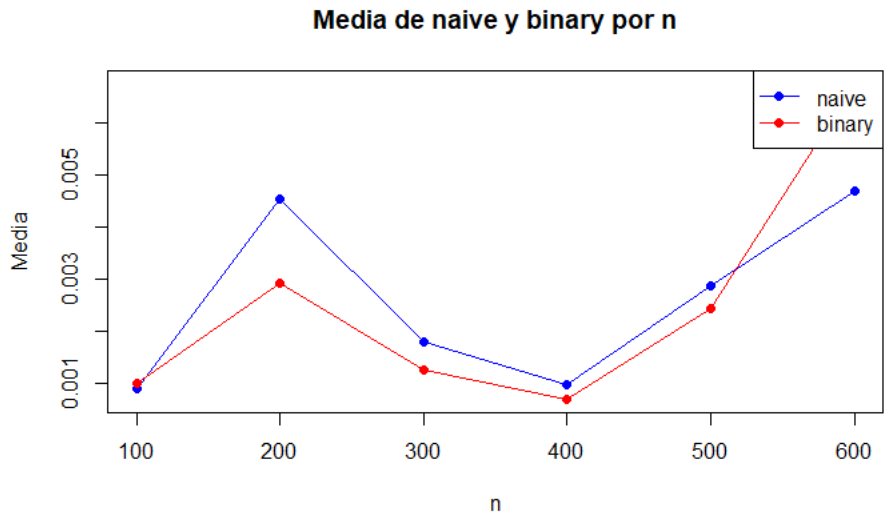
Para evaluar empíricamente el rendimiento de ambos algoritmos, se ejecutaron múltiples pruebas utilizando conjuntos de datos de diferentes tamaños. Se realizaron 10 ejecuciones por cada tamaño de arreglo, variando entre 100 y 600 elementos en incrementos de 100.

El siguiente código en Python fue utilizado para medir los tiempos de ejecución de ambos algoritmos bajo las mismas condiciones:

```python
import time
import csv

import naive_unimodal_search
import binary_unimodal_search
from generate_unimodal_array import generar_arreglo_unimodal

def mesure_time(func, arr:list):
    start_time = time.time()
    func(arr)
    end_time = time.time()
    return end_time - start_time

def test():
    with open('results.csv', 'w', newline='') as file:
```

```
        writer = csv.writer(file)
        writer.writerow(['n', 'naive', 'binary'])
        for n in [100, 200, 300, 400, 500, 600]:
            print(f"n: {n}")
            for j in range(10):
                arr = generar_arreglo_unimodal(n)
                naive_time = mesure_time(naive_unimodal_search.run, arr)
                binary_time = mesure_time(binary_unimodal_search.run, arr)
                writer.writerow([n, naive_time, binary_time])
                del arr


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

Los datos obtenidos fueron procesados y analizados mediante R, generando el siguiente gráfico que ilustra el comportamiento de ambos algoritmos en función del tamaño del conjunto de datos:



4 Conclusión

Los resultados obtenidos confirman que la búsqueda binaria es considerablemente más eficiente que la búsqueda naive en conjuntos de datos grandes. Mientras que la búsqueda naive presenta una relación lineal entre el tiempo de ejecución y el tamaño del conjunto, la búsqueda binaria muestra un crecimiento logarítmico, lo que la hace mucho más escalable.

Estos resultados refuerzan la importancia de elegir el algoritmo adecuado según el contexto. Si bien la búsqueda naive puede ser útil en conjuntos pequeños o cuando los datos no están ordenados, la búsqueda binaria es la mejor opción en estructuras ordenadas debido a su menor costo computacional.