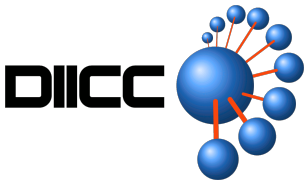


ESTRUCTURAS DE DATOS Y ALGORITMOS AVANZADOS

BOLETÍN N°1

SEPTIEMBRE 2023

ERICH GERMÁN GRÜTTNER DÍAZ

1. Introducción

El presente boletín tiene como objetivo la descripción, implementación y comparación de 3 algoritmos de búsqueda dentro de un arreglo. Se trata de búsqueda secuencial (lineal), búsqueda binaria y búsqueda galopante.

Se mostrarán los resultados de diferentes experimentos, con el objetivo de determinar la variabilidad de desempeño de los algoritmos en distintos escenarios. Se probará variando el tamaño de arreglo de búsqueda y cambiando la ubicación del elemento a buscar.

Estos experimentos se ejecutan mediante una aplicación construida en lenguaje C++, que contiene los algoritmos a probar y diversas utilidades para la medición de rendimiento, como, por ejemplo, la librería Chrono.

Posteriormente se muestran los gráficos asociados a los resultados, generados a través de la librería Matplotlib, usando Phyton.

Finalmente, en base a los resultados obtenidos, se presentan las conclusiones. Son comentados los rendimientos, las diferencias y los posibles escenarios óptimos para su uso.

2. Descripción de los algoritmos a ser comparados

2.1 Búsqueda secuencial

Es un algoritmo de búsqueda utilizado para encontrar un elemento específico en una lista o conjunto de datos.

La búsqueda se realiza secuencialmente, uno por uno, desde el principio hasta el final de la lista, comparando cada elemento con el valor objetivo que se está buscando. [1]

Pseudocódigo:

```
Función BusquedaLineal(lista, elemento_buscado):
    Para cada elemento en la lista:
        Si el elemento actual es igual al elemento_buscado:
            Retornar la posición actual
    Fin del loop
    Retornar -1  # Si no se encontró el objetivo en la lista
Fin de la función
```

2.2 Búsqueda binaria

Es un algoritmo de búsqueda eficiente utilizado para encontrar un elemento específico en una lista ordenada. La característica distintiva de la búsqueda binaria es que divide repetidamente la lista en dos mitades y determina en cuál de las mitades podría estar el elemento buscado, eliminando así la mitad de la lista en cada paso. Este proceso de división y eliminación continúa hasta que se encuentra el elemento objetivo o se determina que no está en la lista. [2]

Pseudocódigo:

```
Función BusquedaBinaria(lista, elemento_buscado):
    Inicializar izquierda a 0
    Inicializar derecha a longitud de lista - 1
    Mientras izquierda <= derecha:
        Calcular el punto medio como (izquierda + derecha) / 2
        Si lista[punto medio] es igual a elemento_buscado:
            Devolver punto medio
        Si lista[punto medio] es mayor que elemento_buscado:
            Establecer derecha a punto medio - 1
        Si lista[punto medio] es menor que elemento_buscado:
            Establecer izquierda a punto medio + 1
    Retornar -1  # El elemento_buscado no se encontró en la lista
Fin de la función
```

2.3 Búsqueda galopante

Es una estrategia de búsqueda que utiliza información parcial sobre la ubicación del elemento buscado para realizar saltos más grandes en lugar de buscar secuencialmente, lo que puede mejorar la eficiencia de la búsqueda en ciertos casos.

Generalmente implica saltar grandes distancias a través de los datos en lugar de examinar cada elemento uno por uno. Por lo general, se utiliza en combinación con otros algoritmos de búsqueda para reducir el tiempo de ejecución. [3]

Pseudocódigo:

```
Función BusquedaGalopante(lista, elemento_buscado):
    Inicializar tamaño de salto a sqrt(longitud de lista)
    Inicializar posición actual a 0
    Mientras pos_actual < largo de lista y lista[pos_actual] != elemento_buscado:
        Si lista[pos_actual] es mayor que elemento_buscado:
            Retroceder pos_actual en el tamaño de salto
        Sino:
            Avanzar pos_actual en el tamaño de salto
    Si pos_actual >= longitud de lista:
        Retornar -1  # El elemento_buscado no se encontró en la lista
    Sino:
        Para i desde pos_actual - tamaño de salto hasta posición actual:
            Si lista[i] == elemento_buscado:
                Retornar i
    Retornar -1  # El elemento_buscado no se encontró en la lista
Fin de la función
```

3. Descripción de los experimentos

3.1 Distintos tamaños del arreglo, posición fija del elemento buscado

Para este caso se consideraron 5 archivos de entrada de input, de diferentes tamaños: 100, 1000, 10000, 100000 y 1000000 registros. Cada uno de ellos contiene como primer elemento su tamaño y luego una lista ordenada de números enteros ascendentes, comenzando por el número 1 y finalizando con el número de su tamaño prestablecido.

Luego, se hacen ejecuciones considerando que el elemento buscado estará en las siguientes ubicaciones fijas: al 10%, 25%, 50%, 75% y 100% (al final) del archivo.

3.2 Tamaño de arreglo fijo, distintas posiciones del elemento buscado

Para este caso se consideraron 5 archivos de entrada de input, pero de tamaño fijo: 100000 registros.

En cada ejecución se varían las ubicaciones del elemento a buscar, utilizando el mismo criterio que en el experimento 1, vale decir: al 10%, 25%, 50%, 75% y 100% (al final) del archivo.

3.3 Procesamiento y gráficos

Cada prueba se realiza 30 veces y se obtiene el promedio del tiempo de ejecución, a través de la librería Chrono de C++.

Los tiempos quedan almacenados en archivos CSV, que luego permiten realizar gráficos de desempeño mediante Python y la librería Matplotlib.

3.4 Equipo de pruebas

Para la realización de las pruebas se utilizó un equipo MacbookPro con procesador M1 y 8Gb de memoria. El chip M1 tiene 8 núcleos (4 de alta eficiencia a 3.2 GHz + 4 de alto rendimiento a 2.0 GHz) y una velocidad de transferencia de 50Gb por segundo.

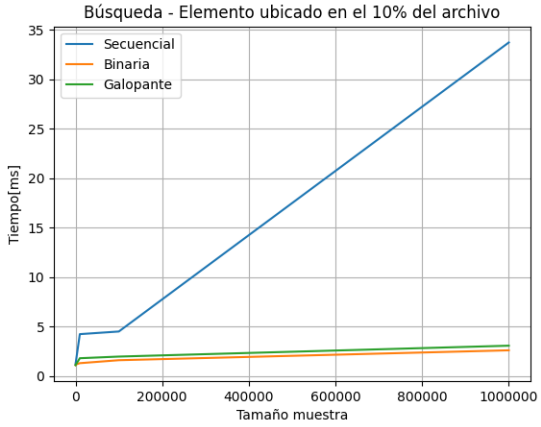
3.5 Código fuente

El código fuente de la aplicación se encuentra disponible en el repositorio de Github [4], donde además se pueden encontrar los archivos gráficos, los archivos CSV de resultados, y los datasets de input.

4. Resultado de los experimentos

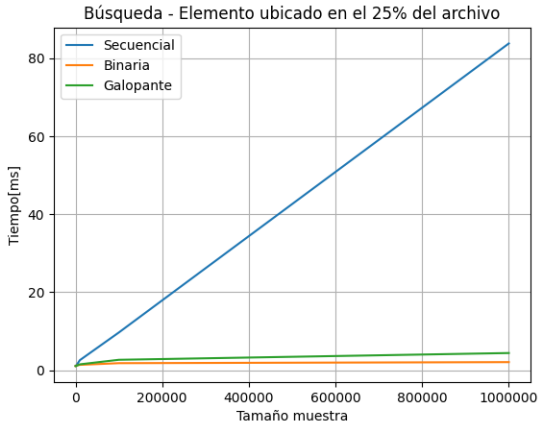
4.1 Distintos tamaños del arreglo, posición fija del elemento buscado

4.1.1 Elemento ubicado en la posición 10% del arreglo



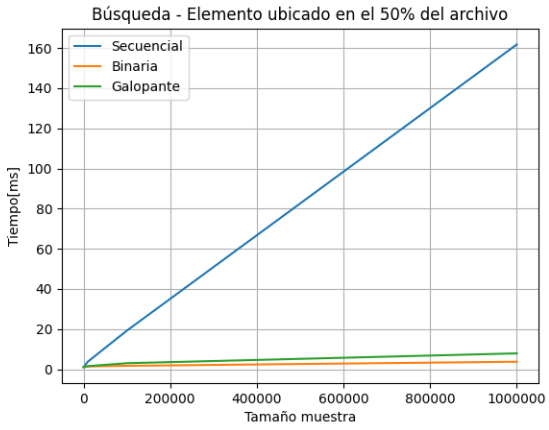
Tamaño muestra	Secuencial tiempo[ms]	Binaria tiempo[ms]	Galopante tiempo[ms]
100	1.23333	1.1	1.1
1000	1.46667	1.16667	1.16667
10000	4.23333	1.3	1.8
100000	4.5	1.6	1.96667
1000000	33.7	2.6	3.06667

4.1.2 Elemento ubicado en la posición 25% del arreglo



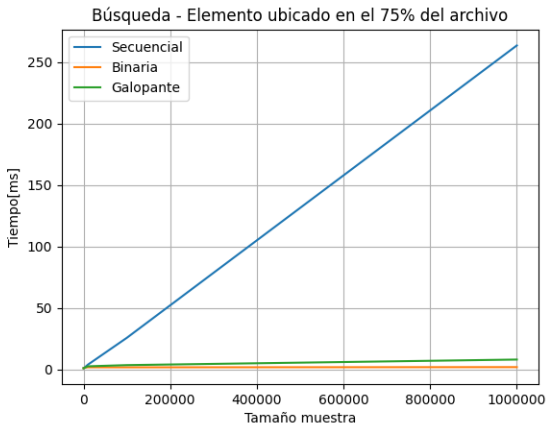
Tamaño muestra	Secuencial tiempo[ms]	Binaria tiempo[ms]	Galopante tiempo[ms]
100	1.1	1.06667	1.06667
1000	1.03333	1.16667	1.06667
10000	2.56667	1.36667	1.5
100000	9.66667	1.8	2.66667
1000000	83.8	2.06667	4.4

4.1.3 Elemento ubicado en la posición 50% del arreglo



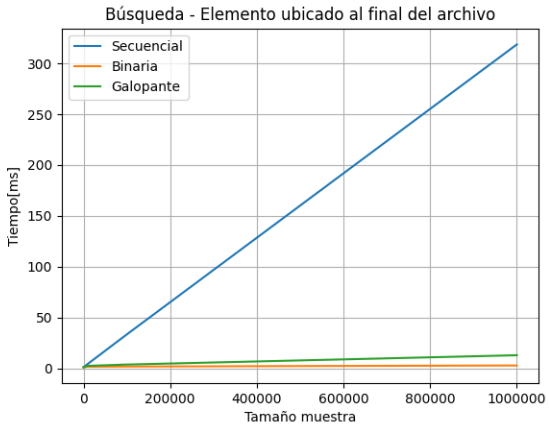
Tamaño muestra	Secuencial tiempo[ms]	Binaria tiempo[ms]	Galopante tiempo[ms]
100	1.1	1.06667	1.1
1000	1.23333	1.06667	1.1
10000	3.8	1.43333	1.56667
100000	19.2333	1.73333	3.03333
1000000	161.733	3.76667	7.93333

4.1.4 Elemento ubicado en la posición 75% del arreglo



Tamaño muestra	Secuencial tiempo[ms]	Binaria tiempo[ms]	Galopante tiempo[ms]
100	1.06667	1.06667	1.06667
1000	1.1	1.33333	1.06667
10000	3.86667	1.76667	2.56667
100000	25.7333	1.73333	3.53333
1000000	263.433	1.9	8.06667

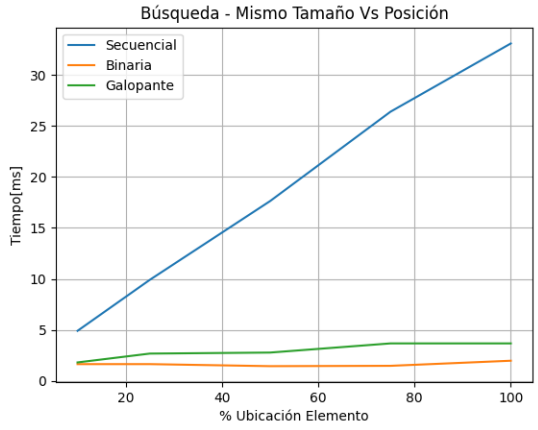
4.1.5 Elemento ubicado en la posición final del arreglo



Tamaño muestra	Secuencial tiempo[ms]	Binaria tiempo[ms]	Galopante tiempo[ms]
100	1.06667	1.06667	1.1
1000	1.16667	1.06667	1.06667
10000	4.3	1.5	2.36667
100000	33.3	1.66667	3.63333
1000000	318.633	2.73333	12.8

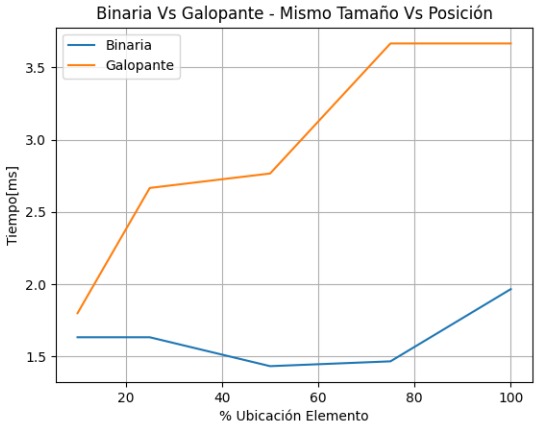
4.2 Distintas posiciones dentro del arreglo, con un tamaño fijo

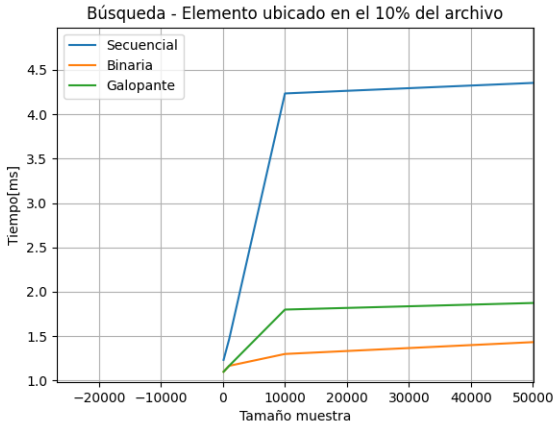
4.2.1 Arreglo con 100.000 registros, posición del elemento variable



% Ubicación elemento	Secuencial tiempo[ms]	Binaria tiempo[ms]	Galopante tiempo[ms]
10	4.9	1.63333	1.8
25	9.9	1.63333	2.66667
50	17.6333	1.43333	2.76667
75	26.4	1.46667	3.66667
100	33.1	1.96667	3.66667

4.2.2 Experimentos adicionales





1.- Se observa que la búsqueda binaria tiene un desempeño mejor cuando los valores a buscar se encuentran en la zona central de arreglo, y la búsqueda galopante tiene un desempeño cercano a la búsqueda binaria, solamente cuando la ubicación del elemento está cerca del principio del arreglo.

2.- En búsquedas con elementos cercanos al principio del arreglo (10%) y de tamaño menor (Ej.: 100 registros), los 3 algoritmos resultan competitivos, incluso la búsqueda secuencial.

5. Conclusiones

- De acuerdo a los gráficos obtenidos, se observa que la complejidad temporal de la búsqueda secuencial es O(n) y la búsqueda binaria se acerca a O(log n). Mientras que para la búsqueda secuencial, si bien su desempeño está entre ambas, se acerca mucho más a O(log n), pero "interferido" por el paso utilizado.
- La búsqueda secuencial es fácil de implementar y funciona bien para listas pequeñas o cuando no se conoce nada sobre la estructura de datos. Se puede usar para listas no ordenadas. No es recomendable para listas grandes.
- La búsqueda binaria es altamente eficiente para listas grandes que están ordenadas. Reduce el número de comparaciones significativamente.
- La búsqueda galopante depende mucho de la implementación, en particular del factor de salto. Mejora la búsqueda secuencial al realizar saltos más grandes. Puede ser más eficiente que la búsqueda secuencial, pero no tan eficiente como la búsqueda binaria en listas grandes y ordenadas. Su rendimiento depende del tamaño de salto y de la distribución de los datos.

6. Referencias

[1] Wikipedia, Linear Search. [En línea]. Disponible: https://en.wikipedia.org/wiki/Linear_search

[2] Wikipedia, Binary Search. [En línea]. Disponible: https://en.wikipedia.org/wiki/Binary_search_algorithm

[3] Wikipedia, Exponential Search. [En línea]. Disponible: https://en.wikipedia.org/wiki/Exponential_search

[4] Github, Repositorio de código fuente. [En línea]. Disponible: https://github.com/egruttner/FEDA2-Boletin_1/tree/main/code