|  |  |  |
| --- | --- | --- |
|  |  |  |

ESTRUCTURAS DE DATOS Y ALGORITMOS AVANZADOS

**BOLETÍN N°1**

SEPTIEMBRE 2023

**ERICH GERMÁN GRÜTTNER DÍAZ**

**Índice**

[1. Introducción 3](#_Toc144557501)

[2. Descripción de los algoritmos a ser comparados 4](#_Toc144557502)

[2.1 Ordenamiento 4](#_Toc144557503)

[2.1.1 Selection Sort 4](#_Toc144557504)

[2.1.2 Merge Sort 5](#_Toc144557505)

[2.1.3 Quick Sort 6](#_Toc144557506)

[2.1.4 Sort Interno C++ 7](#_Toc144557507)

[2.1.5 Tabla resumen algoritmos de ordenamiento a analizar en este informe 8](#_Toc144557508)

[3. Descripción de los datasets 9](#_Toc144557509)

[3.1 Datasets para algoritmos de ordenamiento 9](#_Toc144557510)

[3.1.1 Inputs 9](#_Toc144557511)

[3.1.2 Outputs 11](#_Toc144557512)

[3.1.3 Archivos para graficar 11](#_Toc144557513)

[4. Resultados experimentales 12](#_Toc144557514)

[4.1 Medición rendimiento algoritmos de ordenamiento 13](#_Toc144557515)

[4.1.1 Caso 1: Datos desordenados repetidos 13](#_Toc144557516)

[4.1.2 Caso 2: Datos desordenados con valores únicos 14](#_Toc144557517)

[4.1.3 Caso 3: Datos ordenados repetidos 15](#_Toc144557518)

[4.1.4 Caso 4: Datos ordenados reversa (descendente) 16](#_Toc144557519)

[4.1.5 Caso 5: Datos parcialmente ordenados 17](#_Toc144557520)

[5. Conclusiones 18](#_Toc144557521)

[5.1 Algoritmos de ordenamiento 18](#_Toc144557522)

[6. Referencias 19](#_Toc144557523)

# 1. Introducción

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

Se realizarán diferentes experimentos para determinar la variabilidad del desempeño de los algoritmos en distintos escenarios, variando el tamaño del arreglo de búsqueda y cambiando la ubicación del elemento a buscar. Para tal fin, se construirá una aplicación en lenguaje C++, que se combinará con Python para la obtención de gráficos de desempeño y rendimiento.

Se mostrarán conclusiones …

# 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.

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 bucle  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.

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 algoritmos de búsqueda binaria o estructuras de datos especiales para reducir el tiempo de búsqueda.

Pseudocódigo:

|  |
| --- |
| **Función BusquedaGalopante(lista, elemento\_buscado):**  Inicializar tamaño de salto a sqrt(longitud de lista)  Inicializar posición actual a 0  Mientras posición actual < longitud de lista y lista[posición actual] != elemento\_buscado:  Si lista[posición actual] es mayor que elemento\_buscado:  Retroceder posición actual en el tamaño de salto  Sino:  Avanzar posición actual en el tamaño de salto  Si posición actual >= longitud de lista:  Retornar -1 # El elemento\_buscado no se encontró en la lista  Sino:  # Realizar una búsqueda lineal dentro del rango actual  Para i desde posición 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.

# 4.1 Resultado de los experimentos

## 4.1 Distintos tamaños del arreglo, posición fija del elemento buscado

### 4.1.2 Caso 2: Datos desordenados con valores únicos

|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |

Observaciones:

* El algoritmo “Selection” es el de peor rendimiento al ser de orden O(n2)
* Debido a la gran diferencia de rendimiento con “Selection”, se separó el análisis en los 3 restantes. De ello se puede inferir que el peor fue “Mergesort”, y que el “SortInterno” fue ligeramente superior al “Quicksort”.

### 4.1.3 Caso 3: Datos ordenados repetidos

|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |

Observaciones:

* El algoritmo “Selection” es el de peor rendimiento al ser de orden O(n2)
* Debido a la gran diferencia de rendimiento con “Selection”, se separó el análisis en los 3 restantes. En este caso “Quicksort” tuvo el peor rendimiento, siendo el mejor “SortInterno” seguido por “Mergesort”.

### 4.1.4 Caso 4: Datos ordenados reversa (descendente)

|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |

Observaciones:

* El algoritmo “Selection” es el de peor rendimiento al ser de orden O(n2)
* Debido a la gran diferencia de rendimiento con “Selection”, se separó el análisis en los 3 restantes. Pese a la cercanía del desempeño en el gráfico, “SortInterno” sigue siendo el que tiene mejor desempeño por una amplia diferencia.

### 4.1.5 Caso 5: Datos parcialmente ordenados

|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |

Observaciones:

* El algoritmo “Selection” es el de peor rendimiento al ser de orden O(n2)
* Debido a la gran diferencia de rendimiento con “Selection”, se separó el análisis en los 3 restantes. Para tamaños menores (20.000) no es tan notoria como al llegar a tamaños grandes (100.000), donde nuevamente “Sort Interno” tiene mejor desempeño.

# 5. Conclusiones

* 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. Ideal para listas grandes y ordenadas, donde se puede aprovechar la propiedad de ordenamiento. No es adecuada para listas no ordenadas.
* 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 secuencia, pero no tan eficiente como la búsqueda binaria en listas grandes y ordenadas. Útil en cuando la lista está ordenada y se quiere acelerar la búsqueda en comparación a la búsqueda secuencial. Su rendimiento depende del tamaño de salto y de la distribución de los datos.

# 

# 6. Referencias

[1] Wikipedia, Selection sort. [En línea]. Disponible: https://en.wikipedia.org/wiki/Selection\_sort

[2] Wikipedia, Merge sort. [En línea]. Disponible: https://en.wikipedia.org/wiki/Merge\_sort

[3] Wikipedia, Quick sort. [En línea]. Disponible: https://es.wikipedia.org/wiki/Quicksort

[4] Wikipedia, Sort STL. [En línea]. Disponible: https://en.wikipedia.org/wiki/Sort\_(C%2B%2B)

[5] Medium, Algorithm Analysis & Time Complexity Simplified. [En línea]. Disponible: https://randerson112358.medium.com/algorithm-analysis-time-complexity-simplified-cd39a81fec71

[6] Wikipedia, Multiplicación de matrices. [En línea]. Disponible: https://es.wikipedia.org/wiki/Multiplicación\_de\_matrices

[7] De User:Bilou - Trabajo propio, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=1538693

[8] Github, Explicación del algoritmo de Strassen. [En línea]. Disponible: https://alu0100881677.github.io/DAA\_L2\_1\_Strassen/Strassen.html

[9] Github, Herramienta para la generación de datasets para algoritmos de ordenamiento. [En línea]. Disponible: https://github.com/egruttner/FEDA-informe1/tree/main/ordenamiento/code/datasets

[10] Github, Outputs de ejecución de algoritmos de ordenamiento. [En línea]. Disponible: https://github.com/egruttner/FEDA-informe1/tree/main/ordenamiento/code/output

[11] Github, Archivos para graficar. [En línea]. Disponible: https://github.com/egruttner/FEDA-informe1/tree/main/ordenamiento/code/csv

[12] Github, Herramienta para la generación de datasets para multiplicación de matrices. [En línea]. Disponible: https://github.com/egruttner/FEDA-informe1/tree/main/matrices/code/datasets

[13] Github, Outputs de ejecución de algoritmos de ordenamiento. [En línea]. Disponible: https://github.com/egruttner/FEDA-informe1/tree/main/matrices/code/output

[14] Github, Archivos para graficar. [En línea]. Disponible: https://github.com/egruttner/FEDA-informe1/tree/main/matrices/code/csv

[15] Github, Gráficos para algoritmos de ordenamiento. [En línea]. Disponible: https://github.com/egruttner/FEDA-informe1/tree/main/ordenamiento/gr%C3%A1ficos

[16] Github, Gráficos para multiplicación de matrices. [En línea]. Disponible: https://github.com/egruttner/FEDA-informe1/tree/main/matrices/gr%C3%A1ficos

[17] Github, Código fuente herramienta análisis algoritmos de ordenamiento. [En línea]. Disponible: https://github.com/egruttner/FEDA-informe1/tree/main/ordenamiento/code

[18] Github, Código fuente herramienta análisis algoritmos de ordenamiento. [En línea]. Disponible: https://github.com/egruttner/FEDA-informe1/tree/main/matrices/code