UMAG Universidad de Magallanes | Facultad de Ingeniería

Informe Aplicación de Algoritmos a un Sistema de Gestión de Inventario

Miguel Loaiza[†], **Ignacio Contreras**[†] **y Rodolfo Cifuentes**[†]

[†] *Universidad de Magallanes*

Este manuscrito fue compilado el: 20 de abril de 2025

Resumen

Este proyecto tiene como objetivo implementar y analizar el rendimiento de tres algoritmos básicos de ordenamiento (Bubble Sort, Selection Sort e Insertion Sort) y dos algoritmos de búsqueda (Búsqueda Secuencial y Búsqueda Binaria), aplicándolos en un sistema de gestión de inventario. El sistema desarrollado permite al usuario interactuar con la aplicación, seleccionar el algoritmo de ordenamiento deseado y ordenar los datos según diferentes métricas disponibles (precio, stock, ID o nombre). Además, se muestra el tiempo de ejecución en segundos que cada algoritmo demora en ordenar cinco bases de datos con distintos volúmenes de productos (10k, 25k, 50k, 75k y 100k). Asimismo, se pueden utilizar los métodos de búsqueda disponibles para buscar productos por ID, nombre o por un rango de precios. El sistema también ofrece funcionalidades adicionales que permiten ejecutar los tres algoritmos de ordenamiento de forma simultánea y generar gráficos comparativos de su rendimiento. De manera similar, se pueden ejecutar ambos algoritmos de búsqueda para visualizar gráficamente sus diferencias de eficiencia.

Keywords: *Diseño de Algoritmos, Bubble Sort, Insertion Sort, Selection Sort, Sequential Search, Binary Search, Recursivo, Gráfico*

Índice

1. Introducción

El presente proyecto tiene como finalidad desarrollar un sistema de gestión de inventario que permita aplicar, analizar y comparar el rendimiento de diversos algoritmos clásicos de ordenamiento y búsqueda. A través de este sistema interactivo, el usuario puede ejecutar los algoritmos de forma individual o combinada, seleccionando las métricas por las cuales se desea ordenar o buscar información (ID, nombre, precio, stock o rango de precios).

El sistema ha sido diseñado con un enfoque práctico, permitiendo observar el comportamiento de los algoritmos frente a bases de datos con diferentes volúmenes de información: 10.000, 25.000, 50.000, 75.000 y 100.000 productos. Se mide y reporta el tiempo de ejecución de cada algoritmo, lo que permite al usuario visualizar diferencias de rendimiento y tomar decisiones fundamentadas respecto a cuál algoritmo es más adecuado en distintos contextos.

Los algoritmos de ordenamiento implementados en este proyecto son Bubble Sort, Selection Sort e Insertion Sort. Su análisis es fundamental para comprender los principios básicos del ordenamiento de datos. Su implementación permite identificar fortalezas y debilidades, dependiendo del tamaño de la base de datos y del tipo de orden requerido.

En cuanto a los algoritmos de búsqueda, se han implementado Sequential Search (búsqueda secuencial) y Binary Search (búsqueda binaria). Ambos permiten realizar consultas eficientes prediseñadas dentro del inventario, y su comparación permite observar cómo varía el rendimiento según la cantidad de datos y si los mismos se encuentran ordenados previamente.

El sistema también ofrece la opción de generar gráficos comparativos de tiempo de ejecución para los algoritmos de ordenamiento y búsqueda, facilitando un análisis visual del comportamiento de cada uno frente a distintos tamaños de entrada.

2. Objetivos

El presente proyecto tiene como finalidad consolidar los conocimientos adquiridos sobre el análisis de complejidad y rendimiento mediante el desarrollo de un sistema de gestión ficticio.

2.1. Objetivo General

Desarrollar un sistema de gestión de inventario interactivo que permita implementar, comparar y analizar el rendimiento de distintos algoritmos de ordenamiento y búsqueda, evaluando su eficiencia a través de pruebas con conjuntos de datos de distintos tamaños.

2.2. Objetivos Específicos

- Implementar tres algoritmos clásicos de ordenamiento (Bubble Sort, Selection Sort e Insertion Sort) y dos algoritmos de búsqueda (Sequential Search y Binary Search) dentro de un sistema de gestión.
- Permitir al usuario interactuar con el sistema para seleccionar el algoritmo a utilizar, así como la métrica de ordenamiento o búsqueda (ID, nombre, precio, stock o rango de precios).
- Evaluar y registrar los tiempos de ejecución de cada algoritmo frente a bases de datos de diferentes tamaños (10k, 25k, 50k, 75k y 100k elementos).
- Generar gráficos comparativos que permitan visualizar y analizar las diferencias de rendimiento entre los algoritmos implementados.
- Fomentar la comprensión del comportamiento algorítmico a través de la experimentación práctica y la interpretación de resultados.

3. Equipo Utilizado

Para la ejecución y análisis del sistema de gestión de inventario desarrollado, así como de los algoritmos de ordenamiento y búsqueda implementados, se utilizó un equipo con especificaciones técnicas adecuadas para realizar pruebas de rendimiento sobre grandes volúmenes de datos. A continuación, se detallan las características del equipo:

- Tarjeta de video: NVIDIA GeForce RTX 3060
- Procesador: Ryzen 7 5700X3D
- Cantidad de Núcleos: 12 CPUs
- RAM: 32 GB
- Almacenamiento: 1TB SSD
- Sistema Operativo: Windows 10 Pro 64 bits

4. Bases de datos

Con el objetivo de evaluar el rendimiento de los algoritmos de ordenamiento y búsqueda implementados, se requirió el uso de conjuntos de datos que simularan un inventario de productos. Para ello, se generaron cinco bases de datos en formato `.csv`, cada una con diferentes volúmenes de elementos. Cada base contiene la misma estructura de columnas: `id`, `nombre`, `categoría`, `precio` y `stock`.

4.1. ¿Cómo se crean las bases de datos?

La creación de las bases de datos fue el primer paso desarrollado en el proyecto, ya que representaban la base sobre la cual se aplicarían los algoritmos de ordenamiento y búsqueda.

Para ello, se diseñó un programa auxiliar en lenguaje C, encargado de generar automáticamente los datos de manera aleatoria. Este generador implementa diversas funciones personalizadas para construir cada campo de los productos:

- ID: Se generó una secuencia de IDs únicos que luego fueron desordenados utilizando el algoritmo Fisher-Yates Shuffle, con el objetivo de evitar que los elementos estuvieran ordenados inicialmente. Esto permite poner a prueba la eficiencia de los algoritmos bajo condiciones más exigentes.
- Nombre y Categoría: Se generaron cadenas de texto aleatorias y únicas para simular la variedad de productos y categorías disponibles en un inventario real.
- Precio: Se asignó un valor decimal aleatorio dentro de un rango lógico, simulando precios de productos reales en dólares.
- Stock: Se generaron cantidades enteras aleatorias para representar la disponibilidad de cada producto.

Una vez generados, los productos eran escritos directamente en archivos `.csv`, cada uno correspondiente a un tamaño específico del conjunto de datos.

4.2. Volumen de datos

Los volúmenes seleccionados para las bases de datos fueron los siguientes:

- 10.000 productos.
- 25.000 productos.
- 50.000 productos.
- 75.000 productos.
- 100.000 productos.

Estos tamaños fueron elegidos de manera progresiva para observar el impacto que tiene el crecimiento del volumen de datos en el rendimiento de cada algoritmo. Durante pruebas preliminares se observó que, al superar los 100.000 productos, el sistema de creación de bases de datos presentaba errores o caídas, por lo que se decidió establecer ese límite como tope para las pruebas oficiales.

4.3. ¿Cómo se trabajan las bases de datos?

Una vez generados los archivos, el programa principal el cual contiene la lógica del sistema interactivo y la ejecución de los algoritmos se encarga de cargar los datos desde los archivos `.csv` hacia estructuras de datos diseñadas específicamente para este propósito. Estas estructuras permiten manipular el contenido del inventario dentro del programa.

El uso de estas estructuras facilita las operaciones de ordenamiento y búsqueda, así como la medición del rendimiento de cada algoritmo.

```c
// Estrucutra para representar los productos.
typedef struct
{
    int id;
    char name[MAX_NAME_LENGTH];
    char category[MAX_CATEGORY_LENGTH];
    double price;
    int stock;
} Product;

// Estructura para representar un inventario.
typedef struct
{
    Product *products;
    int count;
    int capacity;
} Inventory;
```

Código 1. Estructuras utilizadas

5. Implementación de los Algoritmos

La implementación de los algoritmos de ordenamiento y búsqueda se llevó a cabo dentro de un entorno interactivo diseñado específicamente para facilitar la ejecución y análisis de su rendimiento. Esta implementación se realizó mediante funciones modulares que contienen el código de cada algoritmo.

5.1. ¿Cómo es el sistema interactivo?

El sistema fue desarrollado con un enfoque en la experiencia del usuario. Al ejecutarlo, se despliega un menú principal en la terminal que permite al usuario seleccionar la operación que desea realizar: ordenar o buscar dentro de una base de datos.
Una vez elegida la operación, el sistema presenta un segundo menú con las métricas disponibles sobre las cuales aplicar el algoritmo seleccionado (ID, Nombre, Stock, Precio y Rango de precios).
Esta estructura de menús le da al usuario un control total sobre qué algoritmo utilizar y con qué criterio aplicarlo. Además, se ofrece la opción de ejecutar todos los algoritmos de una categoría (ordenamiento o búsqueda) en conjunto para generar comparaciones gráficas automáticas.

5.2. ¿Cómo se tomó el tiempo?

Uno de los objetivos principales del proyecto es analizar y comparar el rendimiento de los algoritmos, por lo tanto, fue esencial implementar un sistema de medición de tiempo. Para ello, se utilizó la biblioteca estándar time.h del lenguaje C. La metodología utilizada fue la siguiente:

- Al inicio de la ejecución del algoritmo, se registra el tiempo de inicio mediante clock().
- Al finalizar la ejecución, se toma un nuevo registro y se calcula la diferencia entre ambos puntos.
- Esta diferencia se convierte a segundos (dividiendo entre CLOCKS_PER_SEC) para obtener el tiempo total de ejecución del algoritmo sobre una base de datos específica.
- El proceso anterior se repite en un bucle para cada una de las cinco bases de datos de distinto tamaño, lo que permite obtener una visión clara del rendimiento del algoritmo frente a distintas cantidades de datos.

Los resultados son almacenados en variables específicas y posteriormente enviados a una función encargada de generar gráficos. Adicionalmente, en la terminal se muestra en tiempo real el resultado de cada ejecución, especificando cuánto demoró el algoritmo seleccionado en ordenar o buscar en cada base de datos.

En el caso de las búsquedas, se trabajó con un conjunto de datos de prueba predefinido para garantizar que la comparación entre algoritmos fuera consistente y válida en todos los volúmenes de datos.

5.3. ¿Cómo se grafican?

Una vez recolectados todos los tiempos de ejecución, estos son enviados a una función dedicada a la generación de gráficos. Esta función hace uso de una biblioteca de C que permite la integración con Gnuplot.

Estos gráficos son fundamentales para el análisis comparativo, ya que permiten observar visualmente cómo escalan los tiempos de ejecución al aumentar la cantidad de elementos, y cómo se comportan los algoritmos frente a distintos escenarios.

5.4. ¿Cómo se ejecutan los algoritmos?

Los algoritmos se ejecutan después después de ciertas elecciones del usuario, después se llama a la función que manipula a los algoritmos, toma tiempos y ejecuta los plots. se comienza a tomar el tiempo de ejecución, luego se llama al algoritmo dedicado a la búsqueda u ordenamiento, termina la toma de tiempo y se saca el total de tiempo de ejecución. luego los parámetros se pasan a la función dedicada a gráficar.

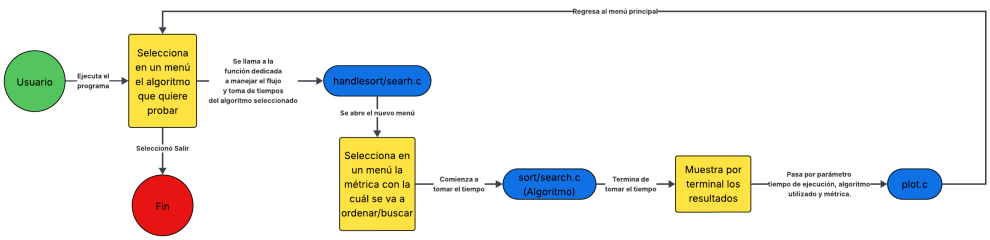

Figura 1. Flujo de Ejecución de los Algoritmos.

6. Análisis de la complejidad

6.1. Bubble Sort

El algoritmo Bubble Sort recorre repetidamente la lisa de elementos adyacentes comparandolos y los intercambia si están en el orden incorrecto.

La versión optimizada mejora el algoritmo original al detenerse antes si no se realizan cambios (cuando ya está ordenado), dándo el mejor caso que es de O(n).

```
void bubble_sort_by_price(Inventory *inv)
{
    for (int i = 0; i < inv->count; i++)
        for (int j = 0; j < inv->count; j++)
            if (strcmp(inv->products[j].price,
                inv->products[j + 1].price) > 0)
            {
                Products temp = inv->products[j];
                inv->products[j] = inv->products[j + 1];
                inv->products[j + 1] = temp;
            }
}
```

Código 2. Bubble Sort no optimizado

Cálculo de complejidad Bubble Sort no optimizado:

$$T(n) = c_1 * n + c_1 * n^2 + c_3 * n^2 + c_4 * n^2 + c_5 * n^2 + c_6 * n^2$$

$$T(n) = c_1 * n + (c_2 + c_3 + c_4 + c_5 + c_6) * n^2$$

$$T(n) = O(n^2)$$

6.2. Insertion Sort

El algoritmo Insertion Sort recorre la lista desde le segundo elemento hasta el final, tomando cada elemento como una key y lo inserta en su lugar correcto en la parte izquierda.

La versión optimizada mejora el algoritmo original al evitar el while usando un if, esto reduce las comparaciones innecesarias en el mejor caso que tiene un orden de O(n).

```
void insertion_sort_by_price(Inventory *inv)
{
    int n = inv->count;
    for (int i = 1; i < n; i++)
    {
        Product key = inv->products[i];
        int j = i - 1;
        while (j >= 0 && strcmp(inv ->products[j].price
                , key.price) > 0)
        {
            inv->products[j + 1] = inv->products[j];
            j--;
        }
        inv->products[j + 1] = key;
    }
}
```

Código 3. Insertion Sort no optimizado

Cálculo de complejidad Insertion Sort no optimizado:

$$T(n) = c_1 + (n-1)(c_2 + c_3 + c_4 + c_8) + \frac{n(n-1)}{2}(c_5 + c_6 + c_7)$$

$$T(n) = a * n + b * n^2$$

$$T(n) = O(n^2)$$

6.3. Selection Sort

El algoritmo Selection Sort busca el menor elemento en la parte no ordenada del arreglo y lo intercambia con el primer elemento no ordenado.

La versión optimizada verifica si ya está ordenado y si no está ordenado, realiza el selection sort, pero solo se intercambia si encuentra un índice diferente al actual. En el mejor caso da orden O(n).

```c
void selection_sort_by_price(Inventory *inv)
{
    int n = inv->count;
    for (int i = 0; i < n - 1; i++)
    {
        int min_index = i;
        for (int j = i + 1; j < n; j++)
            if (strcmp(inv->products[j].price, inv->products[min_index].price) < 0)
        Product temp = inv->products[i];
        inv->products[i] = inv->products[min_index];
        inv->products[min_index] = temp;
    }
}
```

Código 4. Selection Sort no optimizado

Cálculo de complejidad Selection Sort no optimizado:

$$T(n) = (n-1)(c_1 + c_4) + \frac{n(n-1)}{2}(c_2 + c_3)$$

$$T(n) = a * n + b * n^2$$

$$T(n) = O(n^2)$$

6.4. Sequiential Search

El algoritmo Sequential Search realiza una búsqueda lineal recorriendo cada producto, uno por uno y comparando hasta encontrar el producto, si no lo encuentra devuelve un -1. El mejor caso es cuando el algoritmo encuentra el producto en la primera posición, este caso sería de orden O(1).

```c
int sequential_search_by_name(Inventory *inv, const char *name)
{
    for (int i = 0; i < inv->count; i++)
        if (strcmp(inv->products[i].name, name) == 0)
            return i;
    return -1;
}
```

Código 5. Sequential Search no optimizado

Cálculo de complejidad Sequential Search no optimizado:

$$T(n) = c_1 * n + c_2 * n + c_4$$

$$T(n) = n * (c_1 + c_2) + c_4$$

$$T(n) = O(n)$$

6.5. Binary Search

El algoritmo Binary Search realiza una búsqueda dentro de un arreglo ordenado de productos, divide el rango de búsqueda en cada iteración comparando el elemento buscado con el elemento del medio, y elimina la mitad que no contiene el valor. El mejor caso sería si el elemento está justo en el medio de la primera iteración dando un orden O(1).

```c
int binary_search_by_name(Inventory *inv, const char *name)
{
    int left = 0;
    int right = inv->count - 1;
    while (left <= right)
    {
        int mid = left + (right - left) / 2; // Evitar el overflow.
        int cmp = strcmp(inv->products[mid].name, name);
        if (cmp == 0)
            return mid;
        if (cmp < 0)
            left = mid + 1;
        else
            right = mid - 1;
    }
    return -1;
}
```

Código 6. Binary Search no optimizado

Cálculo de complejidad Binary Search no optimizado:

$$T(n) = c_1 + c_2 + c_3 * log_2(n) + c_4 * log_2(n) * c_5 * log_2(n) + c_7 * log_2(n) + c_8$$

$$T(n) = c_1 + c_2 + c_8 + log_2(n) * (c_3 + c_4 + c_5 + c_7)$$

$$T(n) = O(logn)$$

7. Análisis de los gráficos

7.1. Orden general

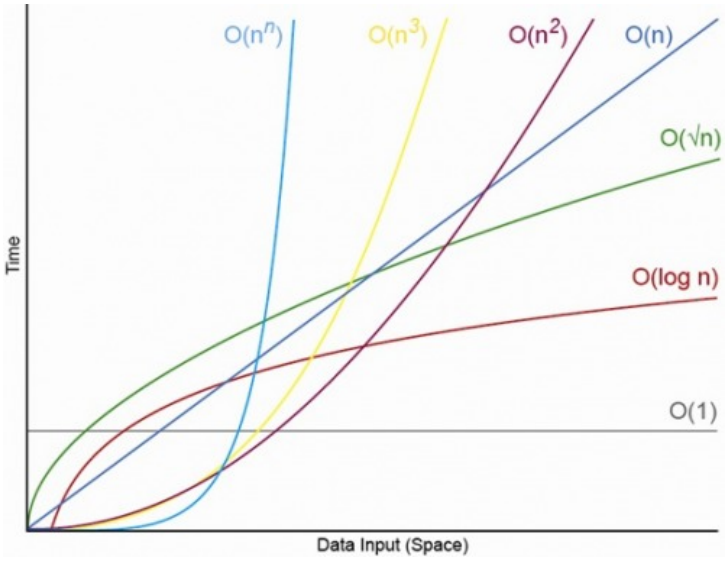

Figura 2. Orgen

7.2. Ordenamientos

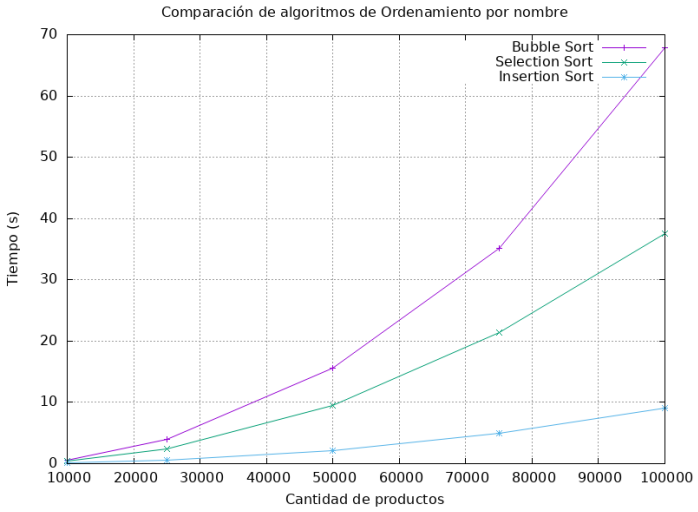

Figura 3. Comparative sort sin optimizar

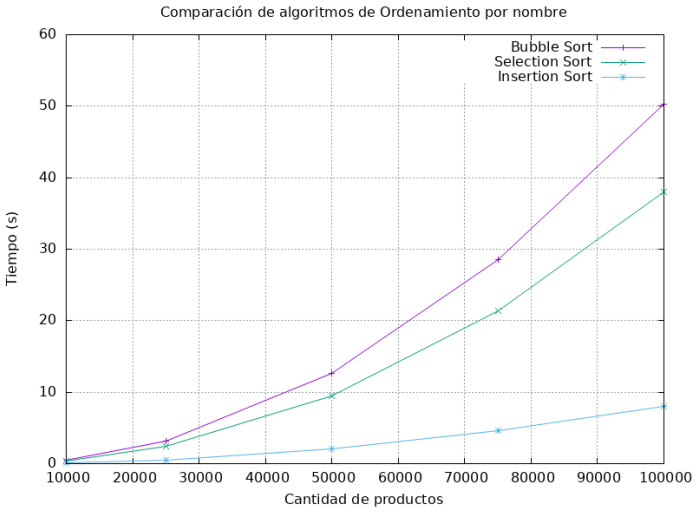

Figura 4. Comparative sort optimizado

Ambos gráficos muestran el comportamiento de tres algoritmos de ordenamiento clásicos —Bubble Sort, Selection Sort e Insertion Sort—, todos con una complejidad de

$$O(n^2)$$

, como se muestra en la Figura 2, lo cual se refleja en el crecimiento no lineal de las curvas a medida que aumenta la cantidad de datos. Las diferencias entre los gráficos radican en la optimización aplicada: en el gráfico optimizado, todos los algoritmos presentan una mejora general en su rendimiento, aunque sin alterar su complejidad teórica.

Bubble Sort es consistentemente el menos eficiente, especialmente en su versión sin optimizar, mientras que Insertion Sort se comporta mejor en ambos casos, mostrando un crecimiento más moderado. Esto se debe a que es más sensible a entradas parcialmente ordenadas y puede beneficiarse más de pequeñas optimizaciones. Selection Sort se mantiene en un punto medio entre los otros dos.

Las curvas coinciden con lo esperado para algoritmos cuadráticos, y aunque las optimizaciones reducen el tiempo de ejecución, no modifican el orden de crecimiento. Para volúmenes de datos como los utilizados, Insertion Sort resulta ser la opción más eficiente entre los tres, aunque ninguno de ellos es ideal para conjuntos grandes.

7.3. Búsqueda secuencial

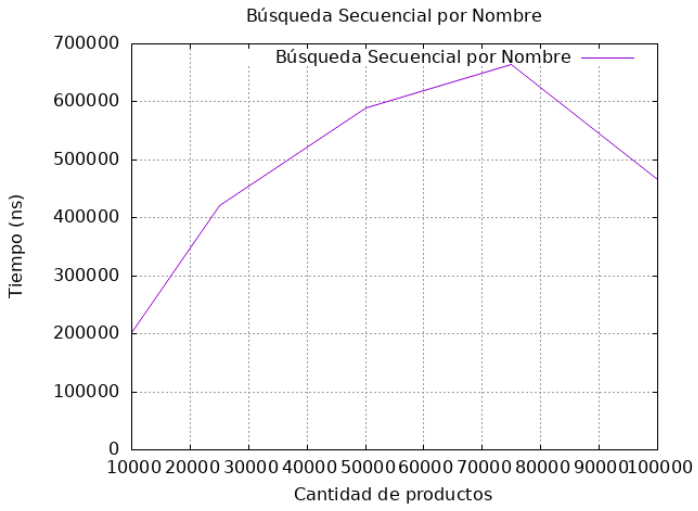

Figura 5. Búsqueda secuencial sin optimizar

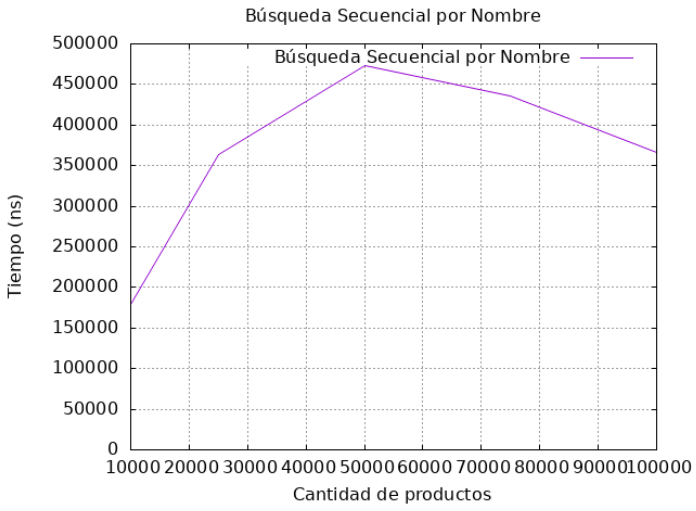

Figura 6. Búsqueda secuencial optimizado

Ambos gráficos evidencian un crecimiento lineal en el tiempo de ejecución a medida que aumenta la cantidad de datos, lo cual es coherente con la naturaleza de la búsqueda secuencial. La versión optimizada presenta una pendiente más suave, lo que indica una reducción en el número promedio de comparaciones. Sin embargo, esta mejora no modifica la complejidad teórica del algoritmo, que continúa siendo, en el peor caso

$$O(n)$$

La búsqueda secuencial, al no aprovechar estructuras de datos ni orden en la información, escala proporcionalmente al tamaño del conjunto. Aunque las optimizaciones mejoran el rendimiento práctico, no alteran el orden de crecimiento. Además, los volúmenes utilizados en la prueba son útiles para observar tendencias lineales, pero no son adecuados para evaluar diferencias significativas entre algoritmos de distintas complejidades. Para ello, sería necesario trabajar con conjuntos de datos mucho más grandes, donde los algoritmos más eficientes en términos teóricos comienzan a mostrar ventajas claras.

Cabe destacar que, debido a lo pequeños que son los tiempos de ejecución (en nanosegundos), las gráficas pueden no reflejar perfectamente el comportamiento teórico del algoritmo. Las pequeñas variaciones del sistema, como interrupciones del sistema operativo, efectos de caché, o fluctuaciones aleatorias, pueden tener un impacto considerable sobre estos tiempos tan bajos, haciendo que el patrón O(n) no se vea tan limpio o exacto como se esperaría en un entorno controlado o con tiempos más altos.

7.4. Búsqueda binaria

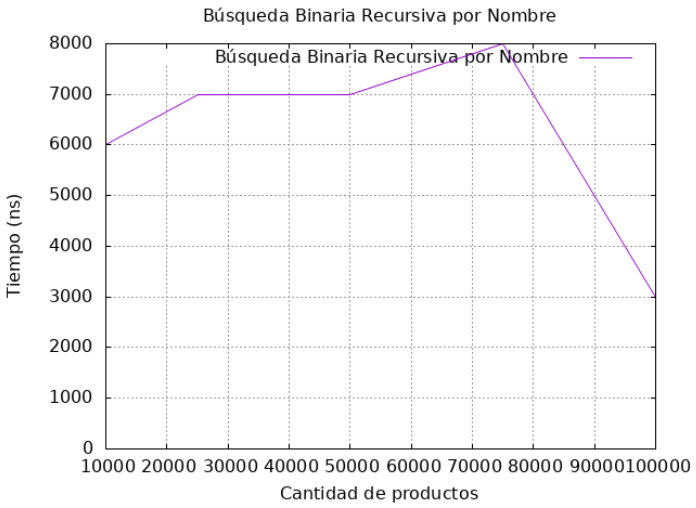

Figura 7. Búsqueda binaria recursivar

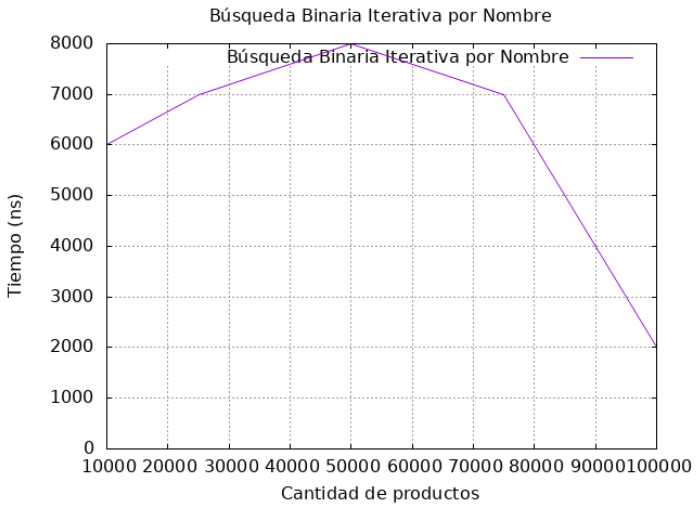

Figura 8. Búsqueda binaria iterativa

El tiempo medido en ambos gráficos corresponde a la búsqueda de múltiples nombres, no solo uno. Por eso, los gráficos reflejan el rendimiento acumulado de varias búsquedas.

Tanto la búsqueda binaria iterativa como la recursiva tienen una complejidad de orden

$$O(logn)$$

, lo cual significa que, a medida que aumenta la cantidad de elementos, el número de comparaciones crece lentamente. Esto las hace eficientes incluso con grandes volúmenes de datos.

Sin embargo, en los gráficos no se aprecia claramente esa relación logarítmica. Una posible razón es que los tiempos de ejecución son muy pequeños (medidos en nanosegundos), y esto hace que cualquier ruido del sistema, optimización del compilador o efecto del caché tenga un impacto relativamente grande en los resultados.

En el caso del gráfico de la búsqueda binaria iterativa, la caída repentina en el tiempo podría deberse a efectos de caché o alguna optimización del sistema operativo.

Por otro lado, el comportamiento inusual del gráfico recursivo, donde el tiempo disminuye con 100,000 productos, podría deberse a:

- Variaciones aleatorias en el rendimiento del sistema
- Características específicas de los datos de prueba utilizados
- Posibles optimizaciones de la pila de llamadas realizadas por el compilador

8. Conclusión

El desarrollo de este proyecto nos permitió aplicar de forma práctica los conceptos fundamentales del diseño y análisis de algoritmos, especialmente en el contexto de la gestión de inventario. A través de la implementación de algoritmos clásicos de ordenamiento (Bubble Sort, Selection Sort e Insertion Sort) y búsqueda (secuencial y binaria), pudimos analizar cómo varía el rendimiento de cada uno según el volumen de datos y el tipo de operación requerida.

Los resultados demostraron que, si bien los algoritmos de ordenamiento implementados son sencillos y didácticos, su eficiencia disminuye considerablemente a medida que aumenta la cantidad de datos, particularmente en el caso de Bubble Sort, incluso con optimizaciones. Por el contrario, algoritmos como Insertion Sort mostraron mejor desempeño en listas parcialmente ordenadas, lo que valida su uso en contextos específicos. En cuanto a la búsqueda, se evidenció una clara ventaja de la búsqueda binaria sobre la secuencial, siempre y cuando los datos estuvieran previamente ordenados.

Además de la implementación algorítmica, se integraron funcionalidades para visualizar los tiempos de ejecución mediante gráficos, lo que facilitó un análisis empírico del comportamiento de cada técnica. Esta etapa fue clave para comprender las diferencias entre el análisis teórico y la observación práctica.

Finalmente, el trabajo colaborativo fue fundamental para lograr una solución completa y funcional. Esto fortaleció nuestras habilidades en programación estructurada, análisis de complejidad, diseño modular y documentación de sistemas. En conclusión, este proyecto no solo consolidó nuestro entendimiento de los algoritmos estudiados, sino que también nos preparó para enfrentar desafíos más complejos a nivel académico y profesional

9. Contacto y Recursos

A continuación, puedes conocer más sobre nuestros trabajos y contactarnos a través de los siguientes medios:

- miloaiza@umag.cl — EhMigueh
- igcontre@umag.cl — Dysnomia9
- rcifuent@umag.cl — Fitooooooooo

El proyecto completo está alojado en GitHub. Puedes visitarlo en el siguiente enlace: github.com/EhMigueh/grupo_5