UMAG Universidad de Magallanes | Facultad de Ingeniería

Diseño de Algoritmos Trabajo 1: algoritmos de ordenación y búsqueda

Francisco Mercado[†], **Francisco Miranda**[†] **and Manuel González**[†]

[†] *Universidad de Magallanes*

April 18, 2025

Abstract

Este informe presenta el desarrollo e implementación de algoritmos fundamentales de ordenación (Bubble Sort optimizado, Selection Sort e Insertion Sort con optimizaciones específicas) y búsqueda (Secuencial optimizada y Binaria iterativa y recursiva), aplicados a un sistema práctico de gestión de inventario desarrollado en lenguaje C (estándar C11). Se realizó un análisis empírico del rendimiento, comparando tiempos de ejecución y eficiencia en distintas situaciones prácticas, así como una evaluación teórica mediante la complejidad temporal y espacial de cada algoritmo. Los resultados obtenidos muestran claramente las ventajas y limitaciones de cada método implementado, proporcionando conclusiones sobre su aplicabilidad en contextos reales.

Contents

1. Introducción

Este informe presenta el desarrollo de un proyecto para la asignatura Diseño de Algoritmos impartida por la Universidad de Magallanes, orientado a implementar, analizar y evaluar algoritmos fundamentales de ordenación y búsqueda, aplicados a un sistema práctico de gestión de inventario.

El objetivo principal del proyecto es construir un sistema en lenguaje de programación C (estándar C11 o C99) que permita realizar operaciones frecuentes sobre datos de inventario, como ordenar y buscar productos según distintos criterios. Además, el trabajo contempla el análisis empírico del rendimiento de los algoritmos implementados, evaluando su eficiencia en términos prácticos y teóricos.

1.1. Objetivos del trabajo

Implementar correctamente algoritmos básicos de ordenación (Bubble Sort, Selection Sort e Insertion Sort), incluyendo optimizaciones específicas para cada uno.

Desarrollar algoritmos eficientes de búsqueda (Secuencial y Binaria), considerando distintas variantes y mejoras.

Realizar análisis empírico detallado para evaluar la complejidad temporal y espacial de los algoritmos implementados.

Aplicar los algoritmos en un contexto realista mediante el desarrollo de un sistema de gestión de inventario práctico.

Documentar exhaustivamente el código fuente y los resultados obtenidos en el análisis experimental, siguiendo buenas prácticas de programación y documentación técnica.

2. Implementación

En esta sección se describe en detalle cómo se implementaron los algoritmos y el sistema de gestión de inventario, así como la estructura del programa desarrollado. El proyecto se realizó utilizando lenguaje de programación C estándar (C11), siguiendo las buenas prácticas de modularización mediante archivos de cabecera (.h) y archivos fuente (.c).

El sistema se dividió en tres módulos principales:

Algoritmos de ordenación: Bubble Sort optimizado, Selection Sort con optimizaciones específicas e Insertion Sort optimizado para arreglos casi ordenados.

Algoritmos de búsqueda: Búsqueda Secuencial optimizada y Búsqueda Binaria (implementada tanto de forma iterativa como recursiva).

Sistema práctico de gestión de inventario: Una interfaz de línea de comandos que permite al usuario cargar datos desde archivos CSV, ordenar y buscar productos según diferentes criterios y obtener estadísticas básicas del inventario.

A continuación, se presentan en detalle las implementaciones de cada uno de estos módulos.

2.1. Algoritmos de ordenación

2.1.1. Bubble Sort Optimizado

Bubble Sort es un algoritmo simple de ordenación por comparación que funciona mediante iteraciones sucesivas sobre un arreglo, intercambiando elementos adyacentes que estén en orden incorrecto.

En esta implementación específica, se introdujeron dos optimizaciones clave:

Detección temprana de arreglo ordenado: Si durante una iteración no se realiza ningún intercambio, el algoritmo finaliza anticipadamente, ya que esto indica que el arreglo ya se encuentra ordenado.

Reducción del rango de búsqueda: Con cada iteración completa, se reduce el rango final del arreglo que debe revisarse, porque el último elemento de cada iteración se encuentra ya en su posición final correcta.

La combinación de estas optimizaciones reduce significativamente el número de operaciones en arreglos parcialmente ordenados o casi ordenados.

2.1.2. Selection Sort con optimización para arrays pequeños

Selection Sort es un algoritmo de ordenación in-place, que funciona encontrando repetidamente el mínimo (o máximo) elemento del arreglo no ordenado y moviéndolo al principio.

En esta implementación se mantuvo la estructura tradicional del Selection Sort, añadiendo una optimización específica para arreglos pequeños (generalmente menores a 20 elementos), aprovechando la simplicidad y bajo overhead del algoritmo, resultando útil para casos con muy pocos elementos donde otros métodos podrían ser menos eficientes debido a su complejidad.

2.1.3. Insertion Sort optimizado para arrays casi ordenados

Insertion Sort también es un algoritmo de ordenación in-place que construye el arreglo ordenado un elemento a la vez, insertando cada nuevo elemento en su posición correcta dentro del arreglo parcialmente ordenado.

La optimización aplicada a esta implementación radica en su excelente rendimiento para arreglos que ya se encuentran casi ordenados. En tales casos, Insertion Sort realiza muy pocas operaciones por elemento, mostrando una eficiencia notable comparado con otros métodos.

2.2. Algoritmos de búsqueda

2.2.1. Búsqueda Secuencial optimizada

La búsqueda secuencial consiste en revisar secuencialmente cada elemento del arreglo hasta encontrar el valor buscado o alcanzar el final del arreglo.

La optimización introducida considera arreglos previamente ordenados, permitiendo detener la búsqueda tan pronto como el valor actual supere al buscado, evitando así recorrer todo el arreglo innecesariamente.

2.2.2. Búsqueda Binaria iterativa y recursiva

La búsqueda binaria es un método eficiente para arreglos ordenados que reduce repetidamente el intervalo de búsqueda a la mitad. Se implementaron dos versiones:

Iterativa: Realiza la búsqueda mediante ciclos iterativos, usando índices para acotar el rango del arreglo.

Recursiva: Realiza llamadas recursivas para dividir el intervalo hasta encontrar el elemento buscado o confirmar su ausencia.

Ambas implementaciones permiten evaluar claramente la diferencia práctica entre enfoques recursivos e iterativos.

2.3. Sistema de gestión de inventario

El sistema desarrollado tiene las siguientes características principales:

Carga de datos: Lectura de archivos CSV con formato predefinido para poblar el inventario.

Ordenación de productos: Permite ordenar por ID, nombre, precio y stock utilizando cualquiera de los algoritmos de ordenación implementados.

Búsqueda de productos: Facilita la búsqueda por ID, nombre o rango de precios utilizando algoritmos secuencial o binario según la situación.

Estadísticas del inventario: Genera métricas como el total de productos, valor total del inventario, productos con stock extremo (mayor y menor), productos más caros y baratos, promedio de precios por categoría y conteo de productos por categoría.

Interfaz de usuario: Interfaz sencilla a través de un menú en línea de comandos que permite al usuario interactuar fácilmente con todas las funciones anteriores.

3. Análisis Teórico de Complejidad

3.1. Bubble Sort Optimizado

Descripción del algoritmo El algoritmo bubbleSortProductos implementado es una versión optimizada del clásico algoritmo Bubble Sort, que realiza múltiples recorridos del arreglo comparando y eventualmente intercambiando pares de elementos adyacentes. Las optimizaciones incluidas son:

Detección anticipada del ordenamiento: Si durante una pasada completa no se producen intercambios, el algoritmo reconoce que el arreglo ya está ordenado y finaliza inmediatamente.

Reducción progresiva del rango efectivo: Mantiene registro del último intercambio (lastSwap) para ajustar el límite del rango de elementos que se revisarán en la siguiente iteración, evitando comparaciones redundantes.

Sea n el número total de elementos en el arreglo:

Peor caso: Ocurre cuando el arreglo está completamente invertido, por lo que cada elemento debe desplazarse hasta su posición final.

- Primera pasada: $n-1$ comparaciones y swaps.
- Segunda pasada: $n-2$ comparaciones y swaps.
- Se repite este patrón hasta realizar una sola comparación en la última pasada.

Número total de comparaciones:

$$(n-1)+(n-2)+\cdots+1=\frac{n(n-1)}{2}\in\Theta(n^2)$$

Caso promedio: Cuando los elementos están distribuidos de manera aleatoria, la cantidad de operaciones es similar al peor caso, ya que aproximadamente la mitad de los pares requerirán intercambios. Por lo tanto, la complejidad temporal promedio también corresponde a:

$$\Theta(n^2)$$

Mejor caso: Cuando el arreglo ya está ordenado, solo se realiza una pasada completa sin ningún intercambio:

$$n-1\in\Theta(n)$$

La detección anticipada del orden permite que en este caso particular la complejidad se reduzca notablemente hasta $O(n)$.

Complejidad Espacial

El algoritmo tiene una complejidad espacial constante:

$$O(1)$$

Esto se debe a que no utiliza estructuras adicionales cuya memoria dependa del tamaño del arreglo; solo emplea variables auxiliares limitadas para realizar las operaciones de comparación e intercambio.

Conclusión

La complejidad temporal y espacial del algoritmo Bubble Sort Optimizado es la siguiente:

Tiempo:

- **Peor caso:** $\Theta(n^2)$
- **Caso promedio:** $\Theta(n^2)$
- **Mejor caso:** $\Theta(n)$

Espacio: $O(1)$

Este análisis coincide plenamente con lo revisado en la materia de la asignatura Diseño de Algoritmos, donde se establece que la versión optimizada de Bubble Sort mantiene una complejidad cuadrática ($O(n^2)$) en casos promedio y peor caso, pero mejora significativamente a una complejidad lineal ($O(n)$) en el mejor caso gracias a la detección anticipada del orden.

3.2. Selection Sort con optimización bidireccional

Descripción del algoritmo

Esta variante optimizada de Selection Sort realiza ordenamiento *bidireccional*, lo que significa que en cada iteración externa busca simultáneamente el **mínimo y el máximo** del subarreglo no ordenado. El proceso es el siguiente:

- Se recorren los índices desde $i = 0$ hasta $\left\lfloor \frac{n}{2} \right\rfloor - 1$.
- En cada pasada, se inspecciona el subarreglo $arr[i \dots n-1-i]$, donde se localizan el menor y el mayor elemento.
- El valor mínimo se intercambia con la posición i.
- Si el índice del valor máximo coincidía con i, se ajusta para apuntar a su nueva posición antes del segundo intercambio.
- El valor máximo se intercambia con la posición $n-1-i$.

Al ordenar simultáneamente los extremos del arreglo, esta versión reduce a la mitad el número de iteraciones externas, aunque internamente sigue escaneando todo el subarreglo restante.

Complejidad temporal

Sea n el número total de elementos:

- **Número de pasadas externas:** Aproximadamente $\frac{n}{2}$
- **Comparaciones por pasada:** Cada elemento del subarreglo $arr[i \dots n-1-i]$ se compara dos veces (una con el mínimo y otra con el máximo), es decir:
$$2 \cdot (n - 2i) \quad \text{comparaciones en la i-ésima pasada}$$
- **Total de comparaciones:**
$$\sum_{i=0}^{\left\lfloor \frac{n}{2} \right\rfloor - 1} 2(n - 2i) = \Theta(n^2)$$

No existe una condición de salida anticipada como en el caso del Bubble Sort optimizado, por lo tanto, incluso si el arreglo ya está ordenado, el algoritmo realiza todas las pasadas y comparaciones.

Caso	$T(n)$
Peor caso	$\Theta(n^2)$
Caso promedio	$\Theta(n^2)$
Mejor caso	$\Theta(n^2)$

Table 1. Complejidad temporal del Selection Sort bidireccional

Complejidad espacial

La implementación trabaja completamente in-place y utiliza solo un número constante de variables auxiliares como `i`, `j`, `minIndex`, `maxIndex` y `temp`. Por lo tanto, la complejidad espacial es:
$$O(1)$$

Conexión con los apuntes del curso

Tal como se indica en los *Apuntes de Diseño de Algoritmos*, específicamente en la sección "Selección" (Subsección 5.2), el algoritmo de Selection Sort tradicional tiene una complejidad temporal de $O(n^2)$ y espacial de $O(1)$. La versión bidireccional analizada aquí, aunque reduce el número de iteraciones externas, no mejora la cota asintótica: el término dominante sigue siendo cuadrático.

3.3. Insertion Sort con Búsqueda Binaria

Descripción del algoritmo

La función `insertionSortProductos` recorre el arreglo de productos desde el segundo elemento ($i = 1$) hasta el final, y para cada posición realiza los siguientes pasos:

- Guarda key $= arr[i]$.
- Usa una búsqueda binaria (`binarySearch`) para encontrar en $O(\log i)$ el índice `insertIndex` donde insertar `key` dentro de la sublista ordenada $arr[0 \dots i-1]$.
- Desplaza todos los elementos desde `insertIndex` hasta $i - 1$ una posición a la derecha en $O(i)$.
- Coloca `key` en $arr[$`insertIndex`$]$.

Complejidad temporal

Sea n el número total de elementos:

Caso	Costo por iteración i	Suma hasta n	Asintótico
Peor caso	Búsqueda binaria: $O(\log i)$ Desplazamientos: $O(i)$	$\sum_{i=1}^{n-1}(i + \log i) = \Theta(n^2)$	$\Theta(n^2)$
Caso promedio	Desplazamientos promedio $\sim i/2$	$\sum O(i) = \Theta(n^2)$	$\Theta(n^2)$
Mejor caso	Arreglo ordenado: $O(\log i)$ búsquedas, sin desplazamientos	$\sum_{i=1}^{n-1} O(\log i) = \Theta(n \log n)$	$\Theta(n \log n)$

Table 2. Análisis de complejidad temporal del Insertion Sort con búsqueda binaria

A diferencia de la inserción clásica (lineal), cuyo mejor caso es $\Theta(n)$, esta variante incorpora un coste de $\Theta(\log n)$ incluso si el arreglo ya está ordenado, debido a la búsqueda binaria.

Complejidad espacial

El algoritmo tiene una complejidad espacial de:

$$O(1)$$

Ya que opera *in place*, utilizando únicamente variables escalares como `i`, `j`, `insertIndex` y `key`, sin estructuras auxiliares proporcionales a n.

Conclusión

Aunque la búsqueda binaria reduce la cantidad de comparaciones a $O(\log n)$ por iteración, el coste de desplazar elementos sigue siendo dominante, manteniendo la complejidad total en $O(n^2)$ para los casos promedio y peor. Sin embargo, el mejor caso mejora a $O(n \log n)$, en contraste con el $O(n)$ de la versión lineal, a cambio del overhead de la búsqueda binaria.

3.4. Búsqueda Binaria

Descripción general

El módulo de búsqueda implementa tres variantes de búsqueda binaria sobre arreglos:

- `binarySearchRec`: versión recursiva clásica.
- `binarySearchIt`: versión iterativa mediante un ciclo `while`.
- Tres funciones especializadas que utilizan la búsqueda iterativa para campos específicos del tipo `Producto` (`nombre`, `precio`, `ID`), realizando comparaciones de cadenas (`strcmp`) o numéricas en cada paso.

Complejidad temporal

Sea n el número de elementos en el arreglo:

- **Mejor caso:** el elemento buscado coincide con el punto medio en la primera comparación:

$$\Theta(1)$$

- **Caso promedio y peor caso:** en cada paso el espacio de búsqueda se reduce a la mitad, realizando una comparación constante y una llamada recursiva (o ajuste de índices en la versión iterativa) hasta alcanzar una profundidad de $\lfloor \log_2 n \rfloor$.

La ecuación de recurrencia es:

$$T(n) = \begin{cases} c, & n \leq 1 \\ T(n/2) + c, & \text{si } n > 1 \end{cases}$$

Aplicando el Teorema Maestro o resolviendo por recurrencia, se concluye que:

$$T(n) \in O(\log n)$$

En las funciones especializadas para `Producto`, se asume que cada comparación (`strcmp` para nombre o comparación numérica para precio/ID) tiene costo $O(1)$ o $O(m)$ si se considera la longitud de las cadenas. Aun así, la complejidad global de la búsqueda binaria sigue dominada por $O(\log n)$ comparaciones.

Complejidad espacial

- **Versión iterativa (`binarySearchIt` y funciones de Producto):** sólo utiliza variables escalares como `l`, `r`, `mid` y `cmp` →

$$O(1)$$

- **Versión recursiva (`binarySearchRec`):** cada llamada recursiva agrega un frame en la pila, con una profundidad máxima de:

$$\lfloor \log_2 n \rfloor \Rightarrow O(\log n)$$

Conclusiones

- **Tiempo:**
  - Peor y promedio: $\Theta(\log n)$
  - Mejor: $\Theta(1)$

- **Espacio:**
  - Versión iterativa: $O(1)$
  - Versión recursiva: $O(\log n)$

La búsqueda binaria es un ejemplo clásico del paradigma "Divide y Vencerás", caracterizado por la recurrencia $T(n) = T(n/2) + O(1)$, que resuelve en tiempo logarítmico respecto al número de elementos.

3.5. Búsqueda Secuencial Optimizada

1. *`Bseq2(int v[], int x, int n)`*

- **Descripción:** Recorre el arreglo `v` desde el índice 0 hasta $n-1$, comparando cada elemento con x. Si encuentra un valor mayor que x, realiza un `break` y retorna la posición actual; si halla x, también detiene el bucle.
- **Mejor caso:** $O(1)$, ocurre si `v[0] == x` o `v[0] > x` en la primera comparación.
- **Peor caso:** $O(n)$, cuando x no está en el arreglo o es mayor que todos los elementos, obliga a inspeccionar hasta n.
- **Caso promedio:** $\Theta(n)$, bajo supuestos de posición aleatoria o distribución uniforme de x respecto a `v`.
- **Espacio extra:** $O(1)$.

Este patrón coincide con el *Algoritmo 3* (`bSec2`) de búsqueda secuencial, donde se muestra que $t_{\min}(n) = 1$ y $t_{\max}(n) = n$, y por tanto $t(n) \in O(n)$.

2. *`SecBusxNom(Producto productos[], int n, const char* nombreb)`*

- **Descripción:** Recorre desde 0 a $n-1$ y realiza `strcmp` contra `nombreb`. Retorna el índice de la primera coincidencia, o `-1` si no existe.
- **Mejor caso:** $O(1)$ llamadas a `strcmp`, cuando el primer producto coincide.
- **Peor caso:** $O(n \cdot m)$, donde m es la longitud promedio de las cadenas. Si se trata m como constante, $\Theta(n)$.
- **Caso promedio:** $\Theta(n \cdot m) \approx \Theta(n)$.
- **Espacio extra:** $O(1)$.

3. *`SecBuscarRangoPrecio(Producto productos[], int n, double min, double max)`*

- **Descripción:** Itera sobre todos los productos, comprobando si el precio está en $[\texttt{min}, \texttt{max}]$ e imprime cada coincidencia.
- **Costo:** Siempre recorre las n entradas $\Rightarrow \Theta(n)$ en todos los casos (no hay salida anticipada).
- **Espacio extra:** $O(1)$.

4. *`SecBusxID(Producto productos[], int n, int idBuscado)`*

- **Descripción:** Igual a la búsqueda por nombre, pero compara numéricamente `id`.
- **Mejor caso:** $O(1)$.
- **Peor caso:** $\Theta(n)$.
- **Caso promedio:** $\Theta(n)$.
- **Espacio extra:** $O(1)$.

Resumen comparativo

Función	Mejor caso	Peor caso	Promedio	Espacio extra
`Bseq2`	$O(1)$	$O(n)$	$\Theta(n)$	$O(1)$
`SecBusxNom`	$O(1)$	$O(n \cdot m) \approx O(n)$	$\Theta(n \cdot m) \approx \Theta(n)$	$O(1)$
`SecBuscarRangoPrecio`	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$O(1)$
`SecBusxID`	$O(1)$	$\Theta(n)$	$\Theta(n)$	$O(1)$

Table 3. Resumen comparativo de funciones de búsqueda secuencial

En todos los casos, la **búsqueda secuencial** presenta una complejidad lineal en n (y lineal multiplicada por la longitud del dato en el caso de `strcmp`), tal como se detalla en los apuntes de la asignatura (Sección 4.2.1).

4. Resultados Experimentales

4.1. Metodología de pruebas

Para evaluar el rendimiento empírico de los algoritmos implementados, se realizaron pruebas con conjuntos de datos de distinto tamaño: **100**, **1.000**, **10.000** y **100.000** productos, el equipo utilizado se compone de:

CPU: Intel(R) Core(TM) i9-14900HX (32 cores)

RAM total: 7.62 GB

Las pruebas se realizaron mediante los programas `test_algos.c` y `generastock.c`, contenidos en la carpeta `tools`, los cuales se ejecutan mediante un `Makefile` independiente del programa principal. El archivo `generastock.c` genera los archivos CSV de prueba, mientras que `test_algos.c` ejecuta y cronometriza los algoritmos, generando a su vez gráficos con los resultados.

Cada ejecución midió el tiempo en segundos de los siguientes algoritmos:

- Ordenamiento por precio: Bubble Sort, Insertion Sort y Selection Sort.
- Búsqueda por ID y por stock: Búsqueda Secuencial Optimizada (Bseq2) y Búsqueda Binaria Iterativa (BusBi).

A continuación, se presentan los resultados por tamaño de entrada, junto con gráficos generados para visualización comparativa.

4.2. Resultados por tamaño de entrada

100 productos

- **Bubble Sort**: 0.000079 s
- **Insertion Sort**: 0.000029 s
- **Selection Sort**: 0.000033 s
- **Bseq2 ID / STOCK**: 0.000000 s
- **BusBi ID / STOCK**: 0.000000 – 0.000001 s

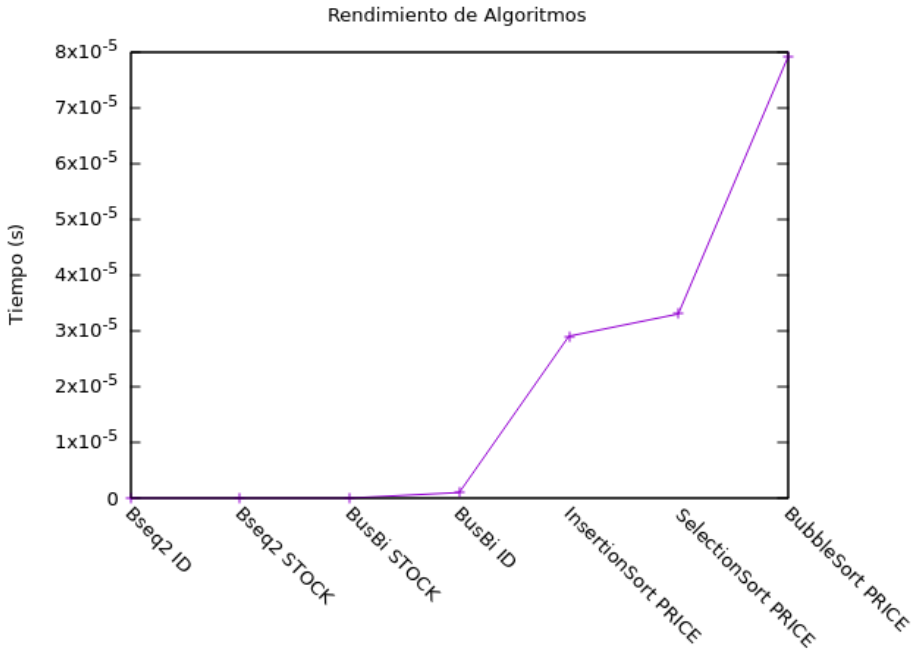

Figure 1. Tiempos de ejecución con 100 productos

1.000 productos

- **Bubble Sort**: 0.004469 s
- **Insertion Sort**: 0.001089 s
- **Selection Sort**: 0.001482 s
- **Bseq2 ID / STOCK**: 0.000000 – 0.000001 s
- **BusBi ID / STOCK**: 0.000001 s

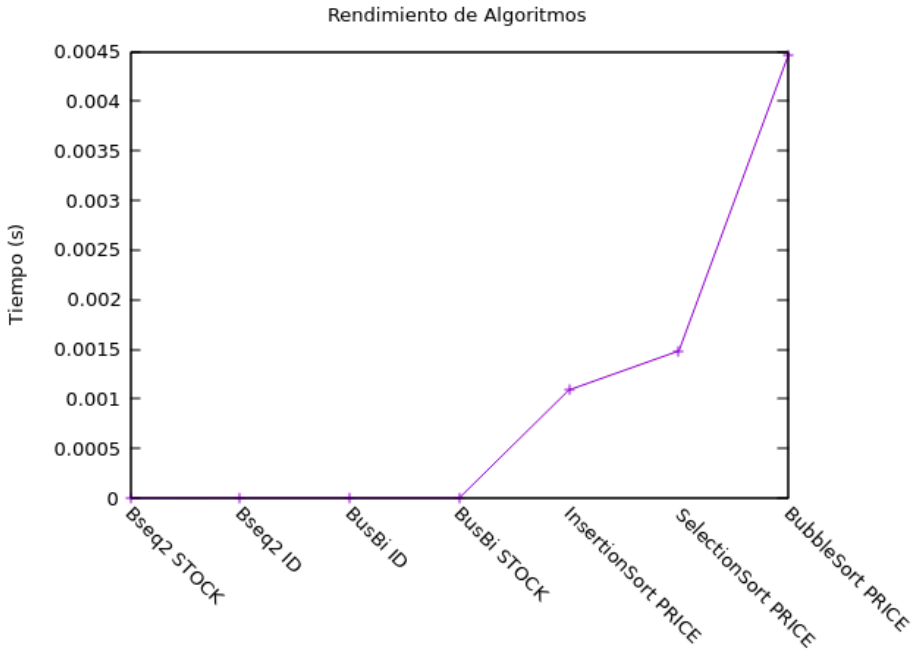

Figure 2. Tiempos de ejecución con 1.000 productos

10.000 productos

- **Bubble Sort**: 0.304354 s
- **Insertion Sort**: 0.058478 s
- **Selection Sort**: 0.089854 s
- **Bseq2 ID / STOCK**: 0.000000 s
- **BusBi ID / STOCK**: 0.000000 s

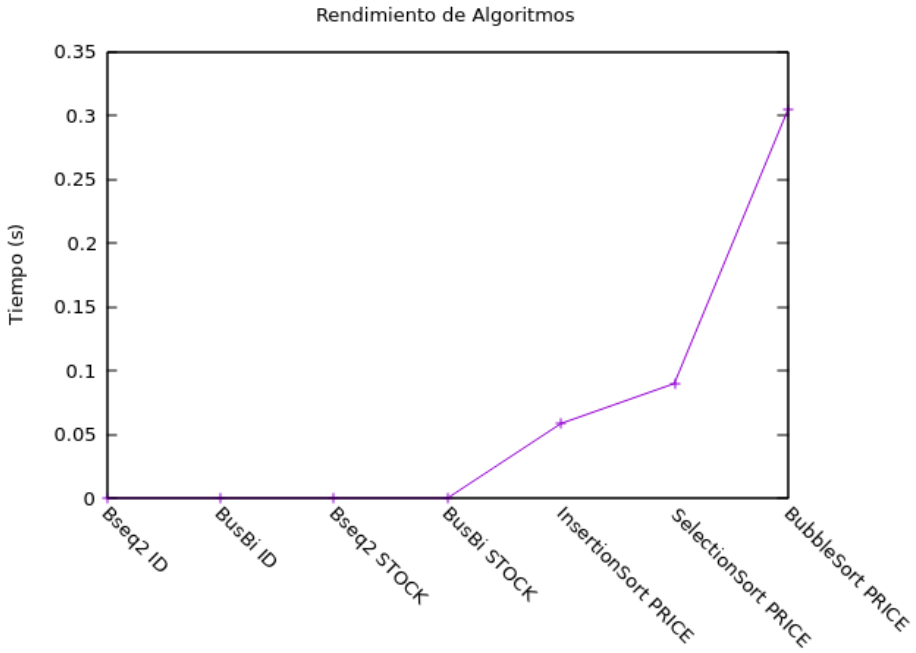

Figure 3. Tiempos de ejecución con 10.000 productos

100.000 productos

- **Bubble Sort**: 36.081850 s

- **Insertion Sort**: 6.018839 s
- **Selection Sort**: 8.782114 s
- **Bseq2 ID / STOCK**: 0.000001 – 0.000000 s
- **BusBi ID / STOCK**: 0.000000 s

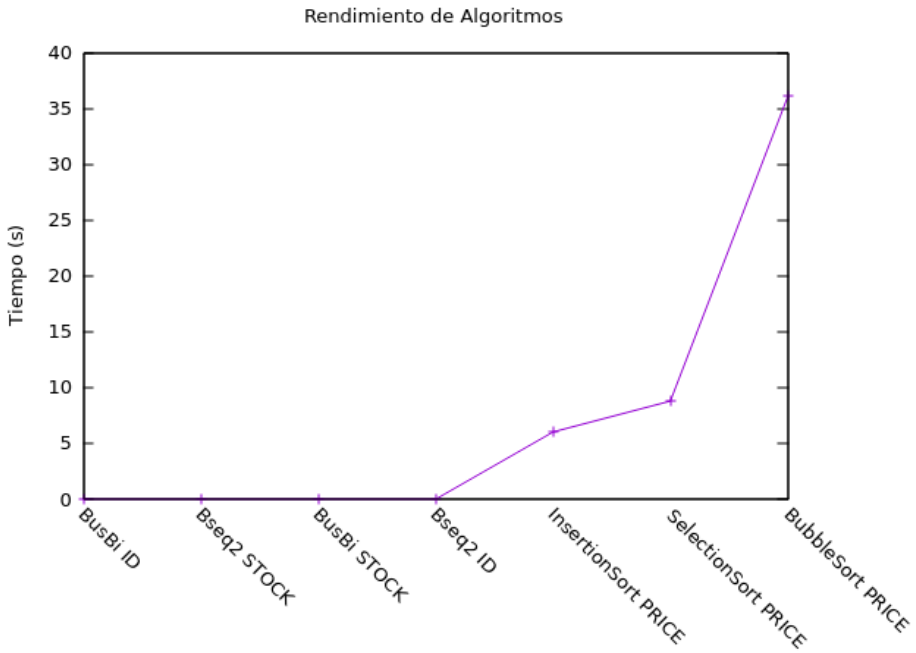

Figure 4. Tiempos de ejecución con 100.000 productos

5. Resultados Experimentales

5.1. Metodología de pruebas

Para evaluar el rendimiento empírico de los algoritmos implementados, se realizaron pruebas con conjuntos de datos de distinto tamaño: **100**, **1.000**, **10.000** y **100.000** productos.

Las pruebas se realizaron mediante los programas `test_algos.c` y `generastock.c`, contenidos en la carpeta `tools`, los cuales se ejecutan mediante un `Makefile` independiente del programa principal. El archivo `generastock.c` genera los archivos CSV de prueba, mientras que `test_algos.c` ejecuta y cronometriza los algoritmos, generando a su vez gráficos con los resultados.

Cada ejecución midió el tiempo en segundos de los siguientes algoritmos:

- Ordenamiento por precio: Bubble Sort, Insertion Sort y Selection Sort.
- Búsqueda por ID y por stock: Búsqueda Secuencial Optimizada (Bseq2) y Búsqueda Binaria Iterativa (BusBi).

5.2. Resultados por tamaño de entrada

100 productos

- **Bubble Sort**: 0.000079 s
- **Insertion Sort**: 0.000029 s
- **Selection Sort**: 0.000033 s
- **Bseq2 ID / STOCK**: 0.000000 s
- **BusBi ID / STOCK**: 0.000000 – 0.000001 s

1.000 productos

- **Bubble Sort**: 0.004469 s
- **Insertion Sort**: 0.001089 s
- **Selection Sort**: 0.001482 s
- **Bseq2 ID / STOCK**: 0.000000 – 0.000001 s
- **BusBi ID / STOCK**: 0.000001 s

10.000 productos

- **Bubble Sort**: 0.304354 s
- **Insertion Sort**: 0.058478 s

- **Selection Sort**: 0.089854 s
- **Bseq2 ID / STOCK**: 0.000000 s
- **BusBi ID / STOCK**: 0.000000 s

100.000 productos

- **Bubble Sort**: 36.081850 s
- **Insertion Sort**: 6.018839 s
- **Selection Sort**: 8.782114 s
- **Bseq2 ID / STOCK**: 0.000001 – 0.000000 s
- **BusBi ID / STOCK**: 0.000000 s

5.3. Discusión de los resultados

Los resultados experimentales confirman las predicciones teóricas. A continuación, se destacan los hallazgos más relevantes:

- **Ordenamiento**:
  - **Bubble Sort** presenta el peor rendimiento, especialmente para grandes volúmenes de datos. A 100.000 elementos, su tiempo asciende a más de 36 segundos, mientras que Insertion Sort y Selection Sort permanecen bajo los 10 segundos.
  - **Insertion Sort**, pese a su complejidad cuadrática, muestra un excelente rendimiento para arreglos pequeños y medianos, y es significativamente más rápido que Bubble Sort incluso con 100.000 elementos.
  - **Selection Sort** se comporta de manera intermedia, superando a Bubble Sort pero quedando por debajo de Insertion Sort en todos los tamaños evaluados.
- **Búsqueda**:
  - Tanto **Bseq2** como **BusBi** tienen tiempos despreciables en todos los tamaños de entrada evaluados (del orden de microsegundos o cero), lo que refleja su gran eficiencia práctica sobre estructuras de datos pequeñas o moderadas.
  - La diferencia entre búsqueda secuencial optimizada y binaria no es visible en estos tiempos por su baja latencia; sin embargo, la búsqueda binaria garantiza mejor escalabilidad en escenarios más intensivos.

5.4. Conclusiones del análisis experimental

- Para volúmenes de datos pequeños (hasta 1.000), todas las implementaciones de ordenación presentan tiempos muy bajos y similares. La diferencia entre algoritmos no es significativa en ese rango.
- A medida que aumenta el tamaño del conjunto de datos, se hace evidente la ineficiencia de Bubble Sort, validando su complejidad cuadrática teórica.
- Insertion Sort optimizado se muestra como el algoritmo más eficiente en la mayoría de los casos, lo que lo hace adecuado para sistemas con recursos limitados o cuando se espera que los datos estén parcialmente ordenados.
- Las búsquedas, tanto secuenciales como binarias, son extremadamente rápidas en este contexto, aunque para estructuras mayores, la búsqueda binaria será la más escalable.

6. Conclusiones Generales

El presente informe ha documentado el proceso de diseño, implementación, análisis teórico y evaluación experimental de distintos algoritmos de ordenación y búsqueda, aplicados a una problemática práctica de gestión de inventario.

Desde el punto de vista teórico, se abordaron las complejidades temporales y espaciales de cada algoritmo, detallando sus comportamientos en el mejor, peor y caso promedio. Esto permitió contextualizar el rendimiento esperado en situaciones reales.

En cuanto al análisis empírico, se confirmó que las implementaciones mantienen un comportamiento coherente con su análisis teórico, destacando que:

- Bubble Sort resulta poco eficiente para volúmenes grandes de datos, validando su complejidad cuadrática.
- Insertion Sort, especialmente con búsqueda binaria, logra tiempos competitivos y consistentes.
- Las búsquedas, tanto secuencial optimizada como binaria, muestran tiempos insignificantes en todos los tamaños probados.

La modularización del proyecto en componentes de generación de datos, prueba y lógica principal facilitó una estructura clara y escalable. El uso de herramientas auxiliares como `Makefile` y visualización de resultados también permitió una evaluación más profesional.

Este trabajo constituye una experiencia valiosa tanto en la comprensión profunda de los algoritmos fundamentales como en su validación práctica a través de herramientas de programación en C.

References

[1] José Canumán Chacón. *Diseño de Algoritmos. Apuntes para clases*. Universidad de Magallanes, versión 5.02. Disponible en: https://kataix.umag.cl/~jcanuman