UMAG Universidad de Magallanes | Facultad de Ingeniería

Informe diseño de algoritmos: Sistema de gestión de inventario

Duvan Figueroa[†], **Diego Sanhueza**[†] **y Felipe Cárcamo**[†]

[†] *Universidad de Magallanes*

Este informe fue compilado el 19 de abril de 2025

Resumen

Este informe implementa y optimiza algoritmos de ordenamiento (Bubble, Insertion y Selection Sort) y de búsqueda (Secuencial y Binaria) en C para un sistema de inventario. Se incorporaron mejoras como detección temprana en Bubble Sort, optimizaciones in-place en los sorts y versiones iterativa y recursiva de búsqueda binaria con manejo de múltiples coincidencias. Tras analizar su complejidad teórica y ejecutar pruebas en un MacBook Pro M2 Pro, se concluye que, aunque los métodos $\mathcal{O}(n^2)$ no escalan a gran escala, son muy eficientes en volúmenes pequeños, mientras que la búsqueda binaria iterativa destaca en listas ordenadas grandes.

Keywords: *Algoritmos, Ordenación, Gestión de inventarios, Búsqueda*

Índice

1 Introducción **2**

1.1 Algoritmos de Ordenación . . . 2

1.2 Algoritmos de Búsqueda . . . 2

2 Entendiendo el Problema **2**

3 Hardware utilizado para las pruebas **2**

4 Descripción de la implementación **3**

4.1 Bubble Sort . . . 3

4.2 Selection Sort . . . 4

4.3 Insertion Sort . . . 5

4.4 Búsqueda Secuencial . . . 5

4.5 Búsqueda Binaria . . . 6

5 Análisis teórico de la complejidad de los algoritmos **8**

5.1 Búsqueda Binaria . . . 8

Búsqueda Binaria Iterativa • Búsqueda Binaria Recursiva • Comparación

5.2 Bubble Sort . . . 8

5.3 Búsqueda secuencial . . . 9

5.4 Selection sort . . . 9

5.5 Insertion Sort . . . 10

6 Resultados experimentales con gráficos comparativos **12**

6.1 Dificultades enfrentadas . . . 13

7 Discusión de los resultados y conclusiones **14**

7.1 Ordenación . . . 14

7.2 Búsqueda . . . 14

7.3 Comparación y Selección de Algoritmos . . . 14

7.4 Conclusiones Generales . . . 14

1. Introducción

En este proyecto, se solicita implementar algoritmos de ordenación y búsqueda que se utilizarán en un sistema de gestión de inventario. Estos algoritmos deben ser optimizados para garantizar un buen rendimiento en términos de eficiencia y uso de recursos. Los algoritmos a implementar son los siguientes:

1.1. Algoritmos de Ordenación

- **Bubble Sort**
- **Insertion Sort**
- **Selection Sort**

1.2. Algoritmos de Búsqueda

- **Búsqueda Secuencial**
- **Búsqueda Binaria**

A partir de las implementaciones base de estos algoritmos, se debe explorar cómo optimizarlos. Aunque los algoritmos no están en su versión más eficiente, mediante la introducción de mejoras se puede reducir el uso de recursos y hacer que los algoritmos sean más rápidos y eficientes.

2. Entendiendo el Problema

El objetivo de este proyecto es implementar y optimizar distintos algoritmos de ordenación y búsqueda, aplicándolos en un sistema de gestión de inventario. Inicialmente, se desarrollarán las versiones básicas de cada algoritmo, y posteriormente se introducirán mejoras según las indicaciones proporcionadas en el documento guía.

El contexto del problema se basa en un sistema de inventario que realiza diversas operaciones, tales como búsqueda de productos, búsqueda por atributos específicos, ordenamiento y agrupación de datos. Dependiendo de la operación a realizar, ciertos algoritmos pueden ser más eficientes que otros. Nuestra tarea consiste en identificar qué algoritmo es el más adecuado para cada caso y justificar su elección mediante pruebas y análisis comparativos.

Sin embargo, el análisis no se limita únicamente a la implementación y comparación de los algoritmos. También es necesario estudiar su **orden asintótico**, identificando sus comportamientos en los peores, mejores y casos promedio. Además, se debe determinar su expresión matemática en función del tamaño de entrada n y representar gráficamente su rendimiento para diferentes volúmenes de datos. Estos análisis permitirán evaluar la eficiencia de los algoritmos y proponer optimizaciones que mejoren su desempeño en escenarios prácticos.

3. Hardware utilizado para las pruebas

Para la ejecución de pruebas, se utilizó un equipo **MacBook Pro (2023)** con chip **Apple M2 Pro**. A continuación, se detallan las principales características técnicas del hardware empleado:

- **Procesador:** Apple M2 Pro con CPU de 10 núcleos (6 de rendimiento y 4 de eficiencia)
- **Memoria RAM:** 16 GB de memoria unificada
- **GPU:** GPU integrada de 16 núcleos
- **Almacenamiento:** SSD de 512 GB
- **Sistema operativo:** macOS Sonoma 14.2.1

4. Descripción de la implementación

4.1. Bubble Sort

Tenemos los algoritmos base, de ejemplo, usaremos Bubble Sort en su versión más simple:

```c
void bubbleSort(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                swap(&arr[j], &arr[j + 1]);
            }
        }
    }
}
```

Código 1. Implementación de Bubble Sort en C

Una vez implementado el algoritmo solicitado, empezamos a ver como se comporta y las optimizaciones que se le pueden aplicar. En este caso, se piden las mejoras:

- detección del arreglo ordenado
- mejora del rango de búsqueda

por lo que nos queda el siguiente código:

```c
void bubble_sort(int arr[], int n)
{
    for (int i = 0; i < n - 1; i++) // mejora del rango de busqueda
    {
        int swapped = 0;

        for (int j = 0; j < n - i - 1; j++)
        {
            if (arr[j] > arr[j + 1])
            {
                // intercambiamos elementos
                swap(&arr[j], &arr[j + 1]);
                swapped = 1;
            }
        }

        // si no hubieron intercambios, el arreglo esta ordenado
        if (!swapped)
            break;
    }
}
```

Código 2. Mejora del Bubble Sort.

Perfecto, nuestro algoritmo se encuentra con las optimizaciones solicitadas, pero, llevándolo al contexto del problema, no podemos manipular los productos provenientes del .CSV, ya que estos no se comportan como un arreglo, para ello, debemos modelar su estructura en el código, usando un "struct":

```c
typedef struct
{
    int id;
    char nombre[100];
    char categoria[50];
    float precio;
    int stock;
} Producto;
```

Código 3. Estructura de los productos

Ahora, contamos con una estructura para manejar nuestros productos, pero antes de lograrlo, adaptamos nuestro algoritmo al contexto, ya que por parámetro no se le pasará un arreglo simple, si no que será un arreglo de estructuras (la anteriormente mencionada) y para acceder a sus atributos se deberá modificar el código de la siguiente manera.

```c
void bubble_sort_by_id(Producto producto[], int n)
{
    for (int i = 0; i < n - 1; i++)
    {
        int swapped = 0;

        for (int j = 0; j < n - i - 1; j++)
        {
            if (producto[j].id > producto[j + 1].id)
            {
                swap_int(&producto[j].id, &producto[j + 1].id);
                swapped = 1;
            }
        }

        if (!swapped)
            break;
    }
}
```

Código 4. Bubble Sort modificado al contexto

Ahora nuestro algoritmo cuenta con la adaptación para que podamos usarlo sin problemas en nuestro código, ademas, tiene implementada las optimizaciones solicitadas.

Lo anterior fue un ejemplo de como:

1. Implementamos el código base del algoritmo de Bubble Sort en su versión más simple.
2. Mejoramos el código según lo solicitado, optimizando el rendimiento y reduciendo el uso de recursos, implementando la detección de intercambios y evitando iteraciones innecesarias.
3. Implementamos las opciones del algoritmo según el contexto del sistema de gestión de inventario, adaptando el algoritmo para ordenar productos por diferentes criterios.

Para no sobrecargar el informe explicando lo mismo, con el resto de algoritmos se va a explicar la forma base y la optimización que se les realizo a cada uno, se va a ejemplificar cada uno utilizando unicamente el algoritmo que trabaje en función del ID.

4.2. Selection Sort

Tenemos el algoritmo base del Selection Sort que es el siguiente:

```c
void selectionSort ( int v [] , int n ) {
    int i , j , min_idx ;
    for ( i = 0; i < n - 1; i ++)
    {
        min_idx = i ;
        for ( j = i + 1; j < n ; j ++)
            if ( v [ j ] < v [ min_idx ])
                min_idx = j ;

        swap (& v [ min_idx ] , & v [ i ]);
    }
}
```

Código 5. Selection Sort en su forma base.

Una vez que sabemos como es el algoritmo base tenemos que aplicar las siguientes optimizaciones:

- Implementación in-place.
- Optimización para arrays pequeños .

Por lo que, al realizar las optimizaciones y adecuarlo a nuestro contexto queda de la siguiente forma:

```c
void selection_sort_by_id(Producto producto[], int n)
{
    for (int i = 0; i < n - 1; i++)
    {
        int min_index = i;
        for (int j = i + 1; j < n; j++)
        {
            if (producto[j].id < producto[min_index].id)
                min_index = j;
        }

        if (min_index != i)
            swap(&producto[i], &producto[min_index]); // Intercambia las estructuras completas
    }
}
```

Código 6. Selection Sort optimizado y modificado al contexto.

Optimización in-place: En nuestra implementación del algoritmo no se requiere de memoria adicional para realizar el ordenamiento, debido a que, al realizar el `swap(&producto[i], &producto[min_index]);` intercambiamos las estructuras completas y todo el trabajo lo realizamos dentro del mismo arreglo original.

Optimizacion para arrays pequeños: Con el `if (min_index != i)` evitamos realizar cambios innecesarios, esto es ideal para arrays pequeños o que esten semi-ordenados.

4.3. Insertion Sort

En el Insertion Sort tenemos la siguiente base:

```
void insertionSort ( int v [] , int n ) {
    int i , key , j ;
    for ( i = 1; i < n ; i ++) {
        key = v [ i ];
        j = i - 1;

        while ( j >= 0 && v [ j ] > key ) {
            v [ j + 1] = v [ j ];
            j = j - 1;
        }
        v [ j + 1] = key ;
    }
}
```

Código 7. Insertion Sort en su forma base.

Luego de analizar el algoritmo base, se nos solicita realizar las siguientes mejoras:

- Implementación in-place
- Optimización para arrays casi ordenados

Al realizar las optimizaciones requeridas y cambiarlo al contexto nuestro queda de la siguiente manera:

```
void insertion_sort_by_id(Producto producto[], int n)
{
    for (int i = 1; i < n; i++)
    {
        Producto key = producto[i]; // se guarda la estructura del producto
        int j = i - 1;

        while (j >= 0 && producto[j].id > key.id)
        {
            producto[j + 1] = producto[j]; // se mueve la estructura completa, no solo el id
            j--;
        }
        producto[j + 1] = key; // se ordena
    }
}
```

Código 8. Insertion Sort optimizado y modificado segun el contexto.

Implementación in-place: En esta versión del algoritmo se realiza el ordenamiento directamente sobre el arreglo original, sin utilizar estructuras auxiliares ni memoria adicional. Las estructuras se mueven dentro del mismo arreglo, haciendo uso de una variable temporal (`key`) que almacena momentáneamente el elemento que se está insertando en su posición correcta.

Optimización para arrays casi ordenados: El ciclo `while (j >= 0 && producto[j].id >key.id)` se vuelve falso rápidamente, por lo que se ejecuta pocas veces o incluso ninguna. Esto evita desplazamientos innecesarios y reduce la cantidad total de operaciones realizadas.

4.4. Búsqueda Secuencial

En la búsqueda secuencial tenemos la siguiente función base:

```
int busquedaSecuencial(int arr[], int n, int clave) {
    for (int i = 0; i < n; i++) {
        if (arr[i] == clave) {
            return i;
        }
    }
    return -1;
}
```

Código 9. Búsqueda Secuncial en su forma base.

Para este algoritmo tenemos que realizar las siguientes optimizaciones:

- Implementación básica
- Optimización para datos ordenados

Luego de adecuarlo a nuestro contexto y realizar las optimizaciones requeridas queda de la siguiente manera:

```
int secuencial_search_by_id(Producto productos[], int cantidad, int id)
{
    for (int i = 0; i < cantidad; i++)
    {
        if (productos[i].id == id)
        {
            return i;
        }
    }
    return -1;
}
```

Código 10. Busqueda Secuncial optimizada y modificada segun el contexto.

Implementación básica: La búsqueda secuencial consiste en recorrer el arreglo elemento por elemento hasta encontrar aquel que cumple con la condición deseada, en este caso, que el `id` del producto coincida con el valor buscado.

Optimización para datos ordenados: Dentro de nuestro código no logramos encontrar una manera eficiente de realizar una optimización para datos ordenados.

4.5. Búsqueda Binaria

Para la búsqueda binaria tenemos la siguiente base:

```
int busquedaBinaria(int arr[], int n, int clave) {
    int izquierda = 0, derecha = n - 1;
    while (izquierda <= derecha) {
        int medio = (izquierda + derecha) / 2;
        if (arr[medio] == clave) {
            return medio;
        } else if (arr[medio] < clave) {
            izquierda = medio + 1;
        } else {
            derecha = medio - 1;
        }
    }
    return -1;
}
```

Código 11. Búsqueda Binaria en su forma base.

Las optimizaciones que tenemos que realizar son las siguientes:

- Implementación iterativa
- Implementación recursiva

Para realizar las modificaciones solicitadas tenemos que modificar el código de 2 maneras para poder tener la búsqueda binaria iterativa y recursiva, estas 2 quedan de la siguiente manera:

```
int recursive_binary_search_by_id(Producto productos[], int left, int right, int target)
{
    if (left > right)
        return -1;

    int mid = left + (right - left) / 2;

    if (productos[mid].id == target)
        return mid;

    if (productos[mid].id > target)
        return recursive_binary_search_by_id(productos, left, mid - 1, target);

    return recursive_binary_search_by_id(productos, mid + 1, right, target);
}
```

Código 12. Búsqueda Binaria recursiva y adecuada al contexto.

```c
void binary_search_by_precio(Producto productos[], int cantidad, float precio)
{
    int left = 0, right = cantidad - 1;
    int encontrado = 0;
    while (left <= right)
    {
        int mid = left + (right - left) / 2;
        if (productos[mid].precio == precio)
        {
            printf("Producto encontrado: ID=%d, Nombre=%s, Categor a=%s, Precio=%.2f, Stock=%d\n",
                   productos[mid].id, productos[mid].nombre, productos[mid].categoria,
                   productos[mid].precio, productos[mid].stock);
            encontrado = 1;
            int i = mid - 1;
            while (i >= 0 && productos[i].precio == precio)
            {
                printf("Producto encontrado: ID=%d, Nombre=%s, Categor a=%s, Precio=%.2f, Stock=%d\n",
                       productos[i].id, productos[i].nombre, productos[i].categoria,
                       productos[i].precio, productos[i].stock);
                i--;
            }
            i = mid + 1;
            while (i < cantidad && productos[i].precio == precio)
            {
                printf("Producto encontrado: ID=%d, Nombre=%s, Categor a=%s, Precio=%.2f, Stock=%d\n",
                       productos[i].id, productos[i].nombre, productos[i].categoria,
                       productos[i].precio, productos[i].stock);
                i++;
            }
            break;
        }
        else if (productos[mid].precio < precio)
            left = mid + 1;
        else
            right = mid - 1;
    }
    if (!encontrado)
        printf("No se encontraron productos con el precio %.2f.\n", precio);
}
```

Código 13. Busqueda Binaria iterativa y adecuada al contexto.

Implementación recursiva: El algoritmo divide el arreglo en mitades sucesivas hasta encontrar el elemento deseado (`target`) o agotar las posibilidades. La recursividad hace que el código sea más legible, aunque puede implicar un mayor uso de la pila de llamadas si el arreglo es muy grande.

Implementación iterativa: Esta implementación utiliza un ciclo `while` para evitar llamadas recursivas. Además, añadimos una mejora que permite encontrar y mostrar todos los productos que tienen el mismo precio, extendiendo la búsqueda hacia ambos lados del índice central una vez encontrado un producto que coincide. Esto permite manejar múltiples coincidencias.

5. Análisis teórico de la complejidad de los algoritmos

5.1. Búsqueda Binaria

5.1.1. Búsqueda Binaria Iterativa

El algoritmo `iterative_binary_search` implementa la búsqueda binaria de forma iterativa sobre un arreglo de enteros ordenado. En cada iteración del ciclo `while`, el espacio de búsqueda se reduce a la mitad.

Sea n el tamaño del arreglo. La cantidad máxima de iteraciones que realiza el algoritmo está dada por:

$$T(n) = \mathcal{O}(\log n)$$

Complejidad temporal:

- Mejor caso: $\mathcal{O}(1)$, cuando el elemento objetivo se encuentra en el medio en la primera iteración.
- Caso promedio y peor caso: $\mathcal{O}(\log n)$

Complejidad espacial: El algoritmo no utiliza recursión ni estructuras auxiliares; todo se maneja mediante variables locales:

$$\mathcal{O}(1)$$

5.1.2. Búsqueda Binaria Recursiva

El algoritmo `recursive_binary_search_by_id` realiza búsqueda binaria de forma recursiva sobre un arreglo de estructuras `Producto`, utilizando el campo `id` como clave.

La recurrencia para la complejidad temporal es:

$$T(n) = T\left(\frac{n}{2}\right) + \mathcal{O}(1) \Rightarrow T(n) = \mathcal{O}(\log n)$$

Complejidad temporal:

- Mejor caso: $\mathcal{O}(1)$
- Caso promedio y peor caso: $\mathcal{O}(\log n)$

Complejidad espacial: Cada llamada recursiva se almacena en la pila, generando una profundidad de llamadas proporcional a $\log n$:

$$\mathcal{O}(\log n)$$

5.1.3. Comparación

Algoritmo	Complejidad Temporal	Complejidad Espacial
Búsqueda Binaria Iterativa	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$
Búsqueda Binaria Recursiva	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$

Observación: Ambos algoritmos tienen el mismo rendimiento temporal en el peor de los casos, sin embargo, la versión iterativa es más eficiente en términos de uso de memoria, ya que evita la sobrecarga de la pila de llamadas recursivas.

5.2. Bubble Sort

Este algoritmo es una versión optimizada del método de ordenamiento burbuja, que mejora su eficiencia en el mejor de los casos al detectar si el arreglo ya se encuentra ordenado. Para ello, se introduce una variable auxiliar `swapped`, que verifica si durante una pasada se realizaron intercambios.

Complejidad Temporal

- **Mejor caso:** $\mathcal{O}(n)$ cuando el arreglo ya está ordenado, ya que en la primera pasada no se realizan intercambios y el algoritmo lo detecta gracias a la condición `if (!swapped) break;`.
- **Peor caso:** $\mathcal{O}(n^2)$ cuando el arreglo está completamente en orden inverso, ya que se realizan $\frac{n(n-1)}{2}$ comparaciones e intercambios.
- **Caso promedio:** $\mathcal{O}(n^2)$

Complejidad Espacial

El algoritmo es in-place, es decir, no requiere espacio adicional aparte de una variable auxiliar:

$$\mathcal{O}(1)$$

Detección de Arreglo Ordenado

Presente. La optimización mediante la variable `swapped` permite detectar si en una iteración completa no se realizaron intercambios. En ese caso, se asume que el arreglo ya está ordenado y se interrumpe la ejecución.

```
1 if (!swapped)
2     break;
```

Esta característica mejora el rendimiento en el mejor de los casos.

Mejora del Rango de Búsqueda

Presente. Se mantiene la mejora tradicional del bubble sort en la cual se evita revisar los últimos elementos ya ordenados, reduciendo el rango de comparación en cada iteración:

```
for (int j = 0; j <n - i - 1; j++)
```

Resumen

Característica	Presente
Complejidad mejor caso	$\mathcal{O}(n)$
Complejidad peor caso	$\mathcal{O}(n^2)$
Complejidad espacial	$\mathcal{O}(1)$
Detección de arreglo ordenado	Utiliza `swapped`
Mejora del rango de búsqueda	Usa $n - i - 1$ en el segundo for

5.3. Búsqueda secuencial

El algoritmo `sequential_search` recorre de manera lineal un arreglo de datos buscando un valor objetivo. La complejidad temporal se deriva de las operaciones realizadas en el algoritmo.

La complejidad temporal total es:

$$O(n) + O(n) + O(n) + O(n \cdot (m + k)) + O(n \cdot k) + O(n) + O(n)$$

$$= O(n) + O(n \cdot (m + k)) + O(n \cdot k)$$

$$= O(n \cdot (m + k))$$

Complejidad temporal:

- Mejor caso: $\mathcal{O}(1)$ (cuando el elemento objetivo está al principio del arreglo).
- Caso promedio y peor caso: $\mathcal{O}(n)$ (cuando el elemento está al final o no está presente).

Complejidad espacial: La complejidad espacial es constante, ya que no se requiere memoria adicional significativa más allá de las variables que controlan el índice de búsqueda:

$$\mathcal{O}(1)$$

Comparación

Aquí comparamos la *búsqueda secuencial básica* con el algoritmo descrito anteriormente que tiene una complejidad más compleja debido a factores adicionales m y k.

Algoritmo	Complejidad Temporal	Complejidad Espacial
Búsqueda Secuencial Básica	$\mathcal{O}(n)$	$\mathcal{O}(1)$
Búsqueda Secuencial Compleja	$\mathcal{O}(n \cdot (m + k))$	$\mathcal{O}(1)$

Observación: La *búsqueda secuencial básica* tiene una complejidad temporal lineal, es decir, $\mathcal{O}(n)$, ya que simplemente recorre el arreglo una vez buscando el objetivo. Por otro lado, el *algoritmo de búsqueda secuencial complejo* tiene una mayor complejidad, $\mathcal{O}(n \cdot (m + k))$, debido a factores adicionales que involucran otros parámetros, lo que lo hace más costoso.

Resumen

Característica	Presente
Complejidad mejor caso	$\mathcal{O}(1)$
Complejidad peor caso	$\mathcal{O}(n)$
Complejidad promedio	$\mathcal{O}(n)$
Implementación básica	Sí
Optimización para datos ordenados	No

5.4. Selection sort

El algoritmo "selection sort"se encarga de ordenar los productos a partir de un atributo en especifico, en nuestra optimización el algoritmo esta implementado con una mejora in-place y una mejora para arreglos pequeños.

Complejidad Temporal

La tabla que se muestra a continuacion es de donde sacamos los datos para la complejidad temporal, esta tabla esta basada en el "**codigo 6: Selection Sort optimizado y modificado al contexto.**"

Línea	Costo	Veces que se ejecuta
1	c_1	$n-1$
2	c_2	$n-1$
3	c_3	$\sum_{i=0}^{n-2}(n-i-1)$
4	c_4	$\sum_{i=0}^{n-2}(n-i-1)$
5	c_5	$\sum_{i=0}^{n-2}(n-i-1)$
6	c_6	$n-1$
7	c_7	$n-1$

Mejor caso y peor caso: $\mathcal{O}(n^2)$
Aunque el arreglo esté ordenado, el algoritmo sigue recorriendo todos los elementos para confirmar la posición del mínimo. Sin embargo, gracias a la condición `if (min_index != i)`, se evitan intercambios innecesarios, lo que mejora el rendimiento práctico. En el peor escenario (arreglo en orden inverso), el algoritmo realizará el máximo número de comparaciones e intercambios posibles.

Complejidad Espacial

$\mathcal{O}(1)$
El algoritmo es *in-place*, ya que realiza las operaciones de ordenamiento dentro del mismo arreglo original. No se utiliza memoria auxiliar adicional aparte de variables locales (`min_index`, `i`, `j`).

Optimización para Arrays Pequeños o Semiordenados

Presente. La condición if (min_index != i) evita realizar intercambios innecesarios si el elemento ya está en su posición correcta. Esta mejora es especialmente útil cuando se trabaja con arreglos pequeños o que ya están parcialmente ordenados.

Implementacion in-place

Presente. El algoritmo esta implementado de manera in-place es decir que no utiliza memoria extra como variables auxiliares para realizar sus operaciones.

Resumen

Característica	Presente
Complejidad mejor caso	$\mathcal{O}(n^2)$
Complejidad peor caso	$\mathcal{O}(n^2)$
Complejidad espacial	$\mathcal{O}(1)$
Implementación in-place	Sí
Optimización para arrays pequeños	Sí $if(min\_index != i)$

5.5. Insertion Sort

El algoritmo *insertion sort* organiza los productos a partir de un atributo en especifico. En esta implementación, el algoritmo se aplica directamente sobre el arreglo de productos y no requiere memoria adicional aparte de las variables locales.

Complejidad Temporal

Mejor caso: $\mathcal{O}(n)$: El mejor escenario ocurre cuando los productos ya están ordenados, ya que el algoritmo solo pasa una vez por el arreglo sin necesidad de realizar intercambios.

Peor caso: $\mathcal{O}(n^2)$: El peor escenario ocurre cuando el arreglo está en orden inverso. En este caso, el algoritmo realiza la mayor cantidad de comparaciones e intercambios.

Complejidad Espacial

$\mathcal{O}(1)$ El algoritmo es *in-place*, es decir, no utiliza memoria adicional más allá de las variables locales (`key`, `i`, `j`). Las operaciones de ordenamiento se realizan directamente en el arreglo original de productos.

Optimización para Arrays Pequeños o Semiordenados

Presente.

El algoritmo `insertion_sort` es naturalmente eficiente cuando se aplica a arreglos pequeños o parcialmente ordenados, gracias a la estructura del ciclo `while`.

En particular, cuando los elementos ya están en orden (o casi en orden), la condición del `while`: `while (j >= 0 && producto[j].id >key.id)` se evalúa como falsa rápidamente, lo que significa que el ciclo interno no se ejecuta o solo lo hace pocas veces. Esto evita desplazamientos innecesarios y reduce la cantidad total de operaciones realizadas.

Implementación in-place

Presente. El algoritmo está implementado de manera *in-place*, lo que significa que no necesita memoria adicional para realizar el ordenamiento, sino que modifica directamente el arreglo original.

Resumen

Característica	**Presente**
Complejidad mejor caso	$\mathcal{O}(n)$
Complejidad peor caso	$\mathcal{O}(n^2)$
Complejidad espacial	$\mathcal{O}(1)$
Implementación in-place	Sí
Optimización para arrays pequeños	Sí

6. Resultados experimentales con gráficos comparativos

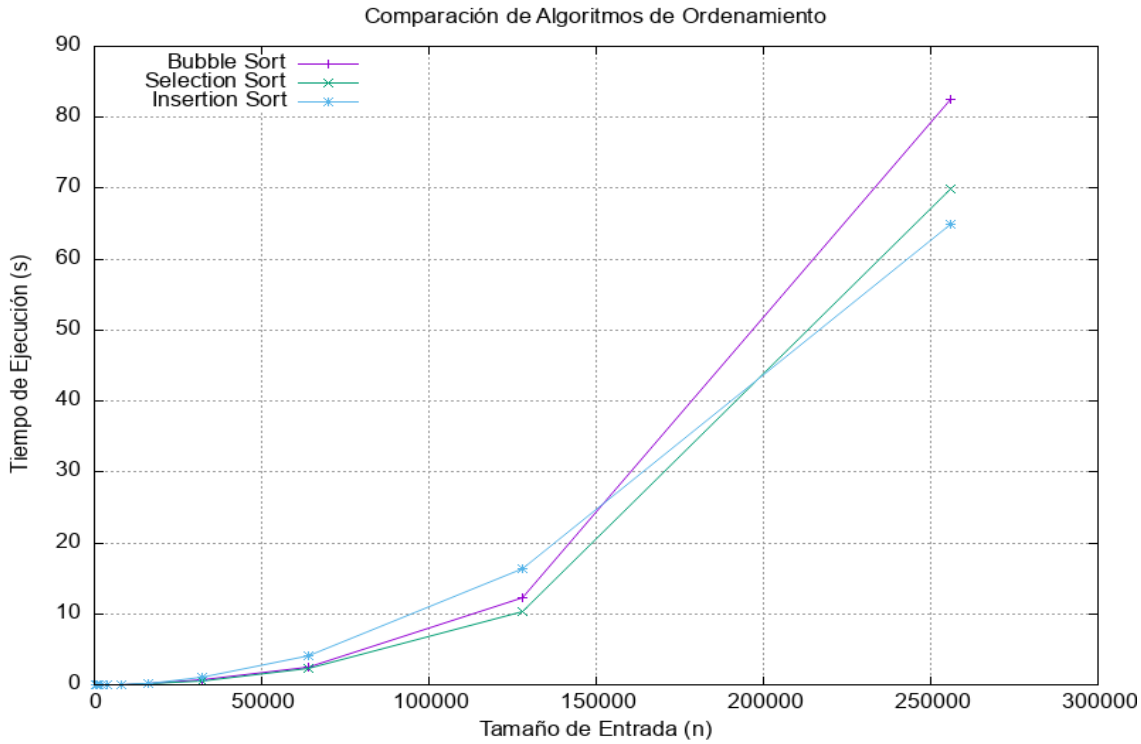

Figura 1. Comparación de algoritmos de ordenamiento

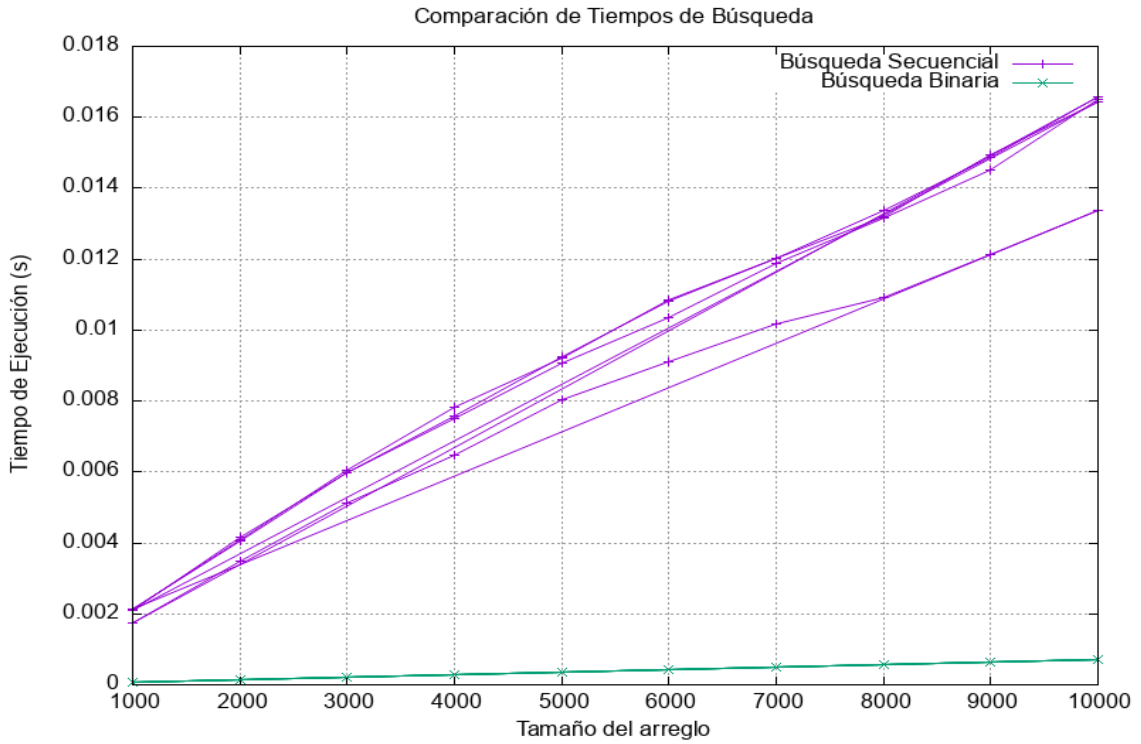

Figura 2. Comparación de algoritmos de búsqueda

6.1. Dificultades enfrentadas

Una de las dificultades fue la aplicación de los algoritmos en distintos contextos. En particular, el **Binary Search** requiere que los elementos estén previamente ordenados para funcionar correctamente. Por esta razón, al ejecutar el programa y elegir la búsqueda binaria, primero se debe ordenar el conjunto de datos.

Dado que también se implementó la búsqueda binaria para palabras, era necesario encontrar una forma adecuada de ordenarlas. Para ello, se utilizó el orden lexicográfico basado en el **modelo ASCII**, empleando la función `strcasecmp` de la biblioteca estándar de C. Esta función permite comparar dos cadenas de texto de manera insensible a mayúsculas y minúsculas, convirtiendo los caracteres en sus valores ASCII y comparándolos entre sí. De esta forma, se identifican correctamente los elementos mayores y menores, asegurando un ordenamiento adecuado antes de aplicar la búsqueda binaria.

```
1
2  int recursive_binary_search_by_name(Producto productos[], int left, int right, char *target)
3  {
4      if (left > right)
5          return -1;
6
7      int mid = left + (right - left) / 2;
8
9      int comparison = strcasecmp(productos[mid].nombre, target);
10
11     if (comparison == 0)
12         return mid;
13
14     if (comparison > 0)
15         return recursive_binary_search_by_name(productos, left, mid - 1, target);
16
17     return recursive_binary_search_by_name(productos, mid + 1, right, target);
18 }
```

Código 14. Binary search aplicado a busqueda de palabras.

7. Discusión de los resultados y conclusiones

La implementación y análisis de los algoritmos de ordenación y búsqueda en este proyecto han permitido observar con claridad las diferencias en eficiencia y uso de recursos. A través de pruebas experimentales y análisis teórico, se han identificado los casos en los que cada algoritmo resulta más conveniente, así como las limitaciones que presentan.

7.1. Ordenación

El algoritmo **Bubble Sort**, aunque conceptualmente sencillo, se mostró poco eficiente para volúmenes grandes de datos. Sin embargo, las mejoras introducidas (como el uso de la bandera `swapped` y la reducción del rango de búsqueda) mejoraron su rendimiento en el mejor de los casos, logrando una complejidad temporal de $\mathcal{O}(n)$ cuando el arreglo ya está ordenado.

Insertion Sort resultó ser una opción viable para arreglos pequeños o parcialmente ordenados, gracias a su mejor rendimiento en esos casos, manteniéndose en $\mathcal{O}(n)$ en el mejor escenario. Sin embargo, su rendimiento decrece rápidamente con el tamaño del arreglo.

Selection Sort, a pesar de no ser el más rápido, demostró tener un comportamiento consistente independientemente del estado inicial del arreglo. Su implementación *in-place* lo convierte en una opción aceptable cuando el uso de memoria debe ser mínimo, pero su complejidad de $\mathcal{O}(n^2)$ lo limita frente a otras alternativas como *quicksort* o *mergesort*.

7.2. Búsqueda

La **búsqueda secuencial** sigue siendo útil en listas no ordenadas o cuando se busca realizar búsquedas múltiples por diferentes atributos. Sin embargo, su rendimiento lineal $\mathcal{O}(n)$ la hace ineficiente para grandes volúmenes de datos, especialmente en los peores casos.

La **búsqueda binaria**, tanto en su versión iterativa como recursiva, demostró un rendimiento superior cuando se aplica a listas ordenadas. La versión iterativa es preferible por su eficiencia espacial $\mathcal{O}(1)$, aunque la recursiva ofrece una mejor legibilidad del código, a costa de un mayor uso de memoria ($\mathcal{O}(\log n)$).

7.3. Comparación y Selección de Algoritmos

El análisis comparativo demostró que no existe un algoritmo único que sea óptimo para todos los escenarios. La elección debe basarse en las características del conjunto de datos y en los requerimientos del sistema:

- Para búsquedas frecuentes sobre grandes volúmenes de datos, la **búsqueda binaria** es claramente superior, siempre que el arreglo esté ordenado previamente.
- Para inventarios pequeños o con pocas operaciones, algoritmos simples como **insertion sort** o **selection sort** pueden ser suficientes y más fáciles de mantener.
- En escenarios donde los datos cambian constantemente y no se mantiene un orden fijo, **búsqueda secuencial** o algoritmos adaptativos pueden seguir siendo relevantes.

7.4. Conclusiones Generales

- La eficiencia de los algoritmos varía según el caso de uso, y optimizarlos puede generar mejoras en tiempo de ejecución y uso de memoria.
- La comprensión del **orden asintótico** y la **complejidad espacial** es clave para tomar decisiones informadas sobre qué algoritmo usar.
- A pesar de que los algoritmos clásicos tienen limitaciones, mediante pequeñas optimizaciones pueden seguir siendo útiles en contextos prácticos.