Experimentación y análisis de algoritmos de búsqueda, ordenamiento e inserción sobre un directorio de archivos, usando C++ en Linux (WSL)

Eduardo Montecinos Gatica *eduardo.montecinos@alumnos.uach.cl* , Felipe Guevara *felipe.guevara@alumnos.uach.cl*, Ignacio Pérez *ignacio.perez01@alumnos.uach.cl* , Diego Pérez de Arce *diego.perezdearce@alumnos.uach.cl*

Escuela de Informática, Facultad de Ciencias de Ingeniería, Universidad Austral de Chile. Valdivia, Chile.

Resumen

En el ámbito de las ciencias de la computación, la selección adecuada de estructuras de datos constituye un pilar fundamental para la implementación y optimización de sistemas de gestión de información. El presente trabajo explora el rendimiento de operaciones básicas (búsqueda, inserción y eliminación) sobre un vector unidimensional que simula un sistema de archivos, analizando empíricamente los tiempos de ejecución conforme escala el volumen de datos. Además, se analiza la eficiencia de los algoritmos implementados, mediante experimentos controlados sobre directorios con 200.000, 1.000.000 y 10.000.000 archivos. Los resultados obtenidos demuestran la relevancia de la elección crítica, tanto de estructuras de datos, como de algoritmos, evidenciando diferencias de hasta tres órdenes de magnitud (notación Big O) en el rendimiento entre algoritmos secuenciales, y algoritmos eficientes.

Keywords: Estructuras de Datos, Algoritmos, Ordenamiento, Búsqueda, Optimización, Eficiencia.

Introducción.

En el desarrollo de sistemas informáticos, la eficiencia con la que se gestionan los datos es un factor crítico que incide directamente en el rendimiento de las aplicaciones. Este trabajo aborda la simulación de un sistema de archivos simplificado utilizando una estructura de datos lineal —específicamente, un vector dinámico (std::vector)— con el objetivo de analizar y comparar el desempeño de distintas operaciones fundamentales: búsqueda, inserción y eliminación de elementos.

El experimento consiste en leer los nombres de archivo almacenados en un directorio real del sistema operativo, almacenarlos en un vector, y luego aplicar distintas técnicas sobre esta estructura.

Se realizarán pruebas sistemáticas que midan el tiempo de ejecución para cada operación bajo diferentes volúmenes de datos, evaluando así su escalabilidad. Particularmente, se contrastarán métodos de búsqueda secuencial y binaria, así como algoritmos de ordenamiento —tanto implementados manualmente como utilizando las herramientas que ofrece la STL de C++—, para evidenciar cómo la elección de una estructura o técnica puede tener un impacto significativo en el comportamiento de un programa.

Además, este proyecto busca reforzar la importancia de comprender el análisis

algorítmico desde una perspectiva teórica (mediante la notación Big-O) y empírica (a través de la experimentación). El análisis de resultados obtenidos en distintos escenarios permitirá valorar la adecuación del vector como estructura base para simular un sistema de archivos, y reflexionar sobre sus limitaciones y ventajas en contextos reales de procesamiento de grandes volúmenes de datos.

Metodología.

La implementación de este proyecto se estructuró en módulos específicos que permitieran realizar pruebas controladas sobre un vector unidimensional que simula un sistema de archivos. Para ello, se desarrollaron funciones encargadas de cargar los datos, realizar búsquedas, ordenar, eliminar e insertar elementos, todo a partir de nombres de archivos reales obtenidos desde un directorio del sistema. Dado que todo el código fuente está en el lenguaje de programación C++, el proyecto se estructuró de la siguiente manera: la carpeta "include" contiene los archivos de cabecera (headers) "vector.h" y "experimentación.h", dentro de los cuales están declaradas las funciones necesarias para los experimentos a realizar. Al interior de la carpeta "src", se encuentran los archivos "vector.cpp", "experimentación.cpp" y "main.cpp". Dentro de los archivos "vector.cpp" y "experimentación.cpp" están definidas las funciones declaradas anteriormente en los archivos "vector.h" y "experimentación.h" respectivamente. En el archivo "vector.cpp" se encuentran todas las funciones que realizan las operaciones necesarias sobre el vector (búsqueda, ordenamiento, eliminación e inserción) y en el archivo "experimentación.cpp", están las funciones que integran las estas operaciones y experimentan con ellas, midiendo el tiempo que tarda cada una en llevarse a cabo. Finalmente, dentro del archivo "main.cpp" se encuentra el código necesario para la ejecución del programa, implementando todo lo anterior (incluye a "experimentación.h" que a su vez incluye a "vector.h"). Luego de compilar el código mediante un archivo "Make" (Makefile), el archivo ejecutable, llamado "file_experiments" se almacena en la carpeta "bin". A continuación, se explica detalladamente cada parte del código:

1 . - Lectura del directorio y carga de datos

Para recorrer el directorio de archivos, se utilizó la librería estándar <filesystem> de C++ para recorrer de forma recursiva (es decir, recorre también subdirectorios) un directorio especificado y almacenar las rutas relativas de los archivos encontrados. Estas rutas se guardan como "strings" (std::string) en un vector (std::vector).

Esta funcionalidad se distribuyó en dos partes: Primero, en el archivo "vector.cpp", la función "crearVector" recibe como argumento la ruta del directorio de archivos y retorna el vector creado. Segundo, en el archivo "experimentación.cpp", la función "creacionVector" recibe como argumento por referencia un vector (el vector sobre el cuál se harán todas las operaciones) e implementa la función "crearVector" y la asigna a dicho vector, retornando la duración de este proceso, medida con "std::chrono", permitiendo analizar el costo temporal de crear la estructura base.

2 . - Búsqueda secuencial

Se implementó un algoritmo que recorre el vector (sin ordenar) desde el inicio, comparando cada elemento con el objetivo y retorna la posición en dónde se encuentra. Si el objetivo no está, se retorna -1. El pseudocódigo es:

```
function busquedaSecuencial (vector, objetivo) {
        for i = 0 to (vector.size() – 1) do {
            if (vector[i] == objetivo) then
                return i;
         }
        return -1;
}
```

Esta función se encuentra definida dentro del archivo "vector.cpp" como "busquedaSecuencial".

3 . - Ordenamiento

Se implementó un algoritmo QuickSort manual, utilizando pivotes aleatorios. Se optó por escoger QuickSort por sobre otros algoritmos eficientes (como MergeSort) debido a que ordena "in-place", o sea que no requiere espacio adicional en memoria para operar (o requiere muy poco) y dado que se está trabajando con grandes volúmenes de datos, nos pareció la mejor opción. Si bien, en el peor caso es de orden cuadrático ($O(n^2)$), el pivote aleatorio, sumado a la gran cantidad de datos, y el "orden" en que a priori se almacenan, hace prácticamente imposible que ocurra ese caso. Cabe destacar que también experimentamos escogiendo la conocida "mediana de tres" como pivote, pero como ofrecía rendimiento muy similar, optamos por dejar el pivote aleatorio para tener código más compacto. A grandes rasgos, QuickSort funciona siguiendo el paradigma de diseño algorítmico "divide y vencerás". Divide y vencerás, es una estrategia que resuelve un problema grande o complejo, dividiéndolo en problemas más pequeños y simples. El algoritmo de ordenamiento QuickSort, escoge un elemento del arreglo (vector en este caso) llamado pivote y lo ubica en su posición correcta, esto es, con todos los elementos menores o iguales a su izquierda y los elementos mayores a su derecha. Luego hace dos llamadas recursivas para repetir el mismo proceso en la parte izquierda del pivote y en la parte derecha, hasta que el vector esté ordenado ascendentemente. Dentro del código fuente, la función "partition" escoge un pivote aleatorio, lo ubica en su posición correcta y retorna el índice de dicha posición. correcta. La función "quickSort", hace las llamadas recursivas. Ambas funciones se encuentran en el archivo "vector.cpp".

4 . - Búsqueda eficiente

Una vez ordenado el vector, se implementó la clásica búsqueda binaria para mejorar el rendimiento. Este algoritmo es increíblemente eficiente, sobre todo cuando el tamaño de la entrada es muy grande, aunque solo funciona si el vector está ordenado. El pseudocódigo es:

```
function busquedaBinaria (vector, objetivo){
        inicio = 0;
        fin = vector.size() – 1;
        while (inicio  <=  fin) do {
            medio = (inicio + fin) / 2;
            if vector[medio]==objetivo then
                return medio;
            if vector[medio] < objetivo then
                inicio = medio + 1;
            else
                fin = medio – 1;
        }
        return -1
    }
```

Esta función se encuentra definida dentro del archivo "vector.cpp" como "busquedaBinaria".

5 . – Eliminación

Para eliminar elementos del vector, se encuentra primeramente la posición de dichos elementos mediante una búsqueda binaria. Luego, se utiliza la función "std::vector::erase()". Esta función remueve el elemento en la posición especificada y desplaza todos los elementos posteriores una posición hacia la izquierda, a fin de mantener la contigüidad de la estructura.

6 . – **Inserción**

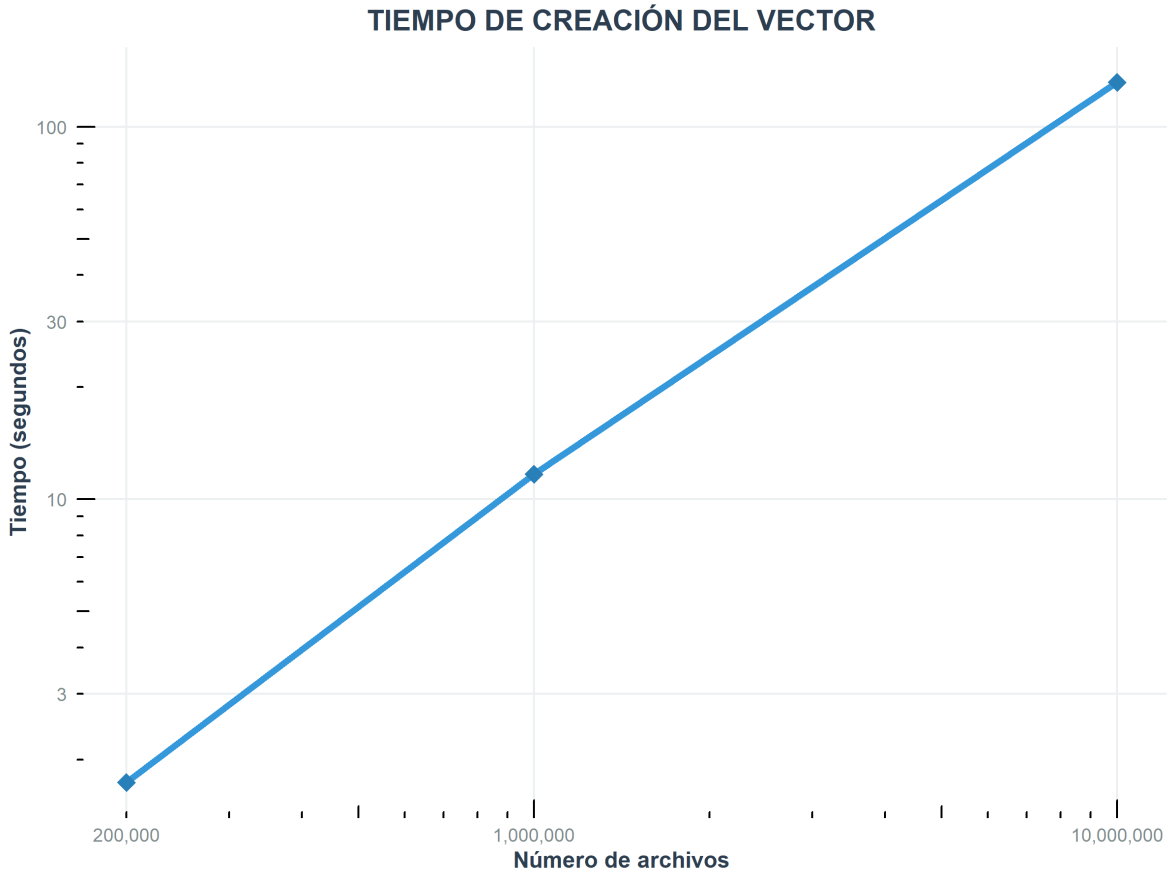
La inserción en un vector ordenado requiere determinar la posición adecuada donde debe ubicarse el nuevo elemento, manteniendo el orden lexicográfico. Para ello, se utiliza "std::lower_bound (inicio_vector, fin_vector, nuevo_valor)", una función del STL que encuentra el primer elemento **no menor** al valor buscado, dentro de un vector ordenado. La posición de ese elemento, es dónde debería insertarse el nuevo valor para que se mantenga el orden. Una vez obtenida el índice correcto, se usa "std::vector.insert(índice, valor)", función que inserta el nuevo valor en la posición dada. Si el índice no es el del último elemento del vector, todos los elementos desde el índice hasta el final del vector deben moverse una posición hacia la derecha para hacer espacio al nuevo elemento.

Presentación de los resultados

Como ya se mencionó anteriormente, el propósito de este trabajo es medir el tiempo que toman las operaciones de: creación de un vector de strings que simula un directorio de archivos, búsquedas dicho vector, ordenamiento del vector, eliminación de elementos e inserción de elementos. En base a estos experimentos, se busca sacar conclusiones sobre las ventajas y desventajas de implementar esta solución para gestionar grandes directorios de archivos. En el archivo de cabecera "experimentacion.h", se encuentra una variable global (un #define) llamada "REP", que indica la cantidad de veces que se llevan a cabo las operaciones de búsqueda, eliminación e inserción de elementos sobre el vector. Los experimentos se llevaron a cabo con REP = 100.000. Todas las funciones que miden el tiempo de las operaciones se encuentran en el archivo "experimentacion.cpp", y usan la librería "std::chrono", la cual permite manipular duraciones e instantes de tiempo. A continuación, se muestran los resultados:

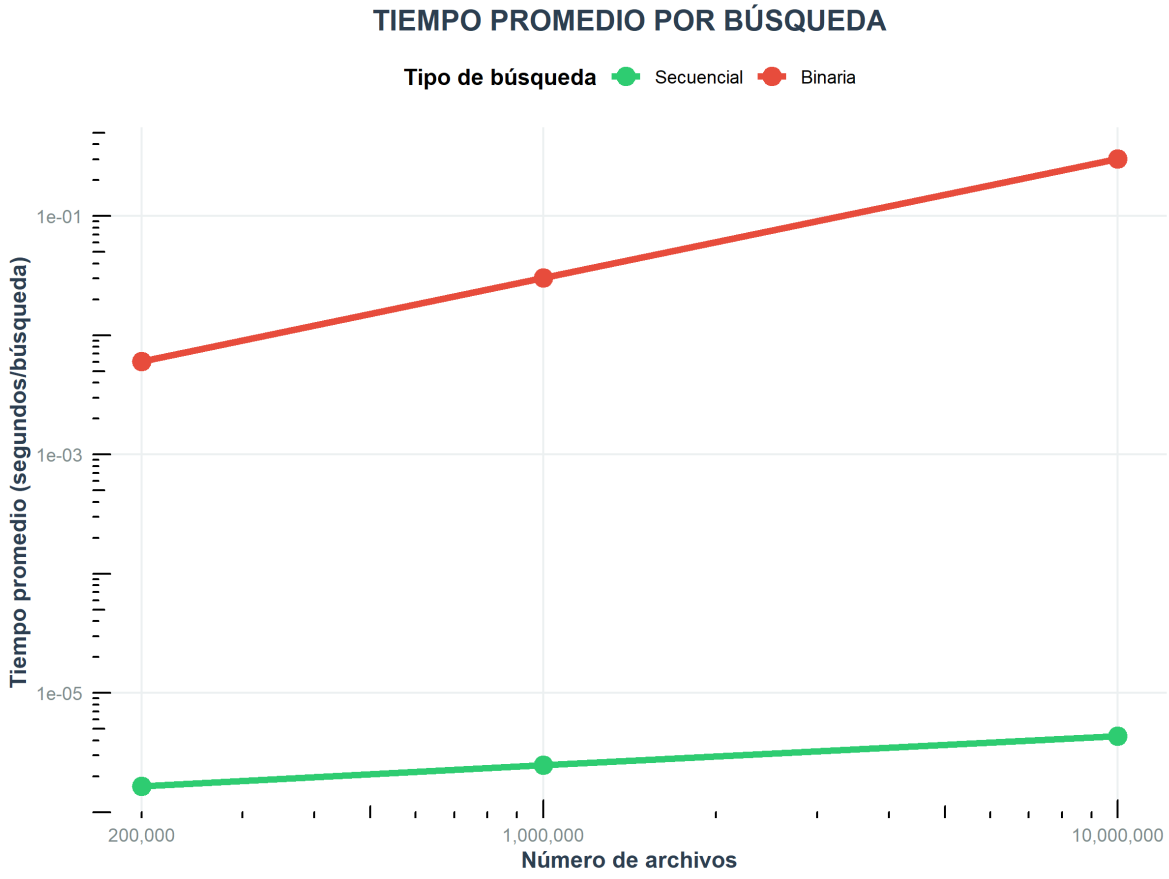
1) Creación del vector:

	Tamaño (cantidad de archivos del directorio cargado)		
	200.000	**1.000.000**	**10.000.000**
Tiempo de creación (microsegundos)	**1.735.337 (1,7 [s] aprox.)**	**11.673.429 (11,7 [s] aprox.)**	**131.548.146 (2,2 [min] aprox.)**



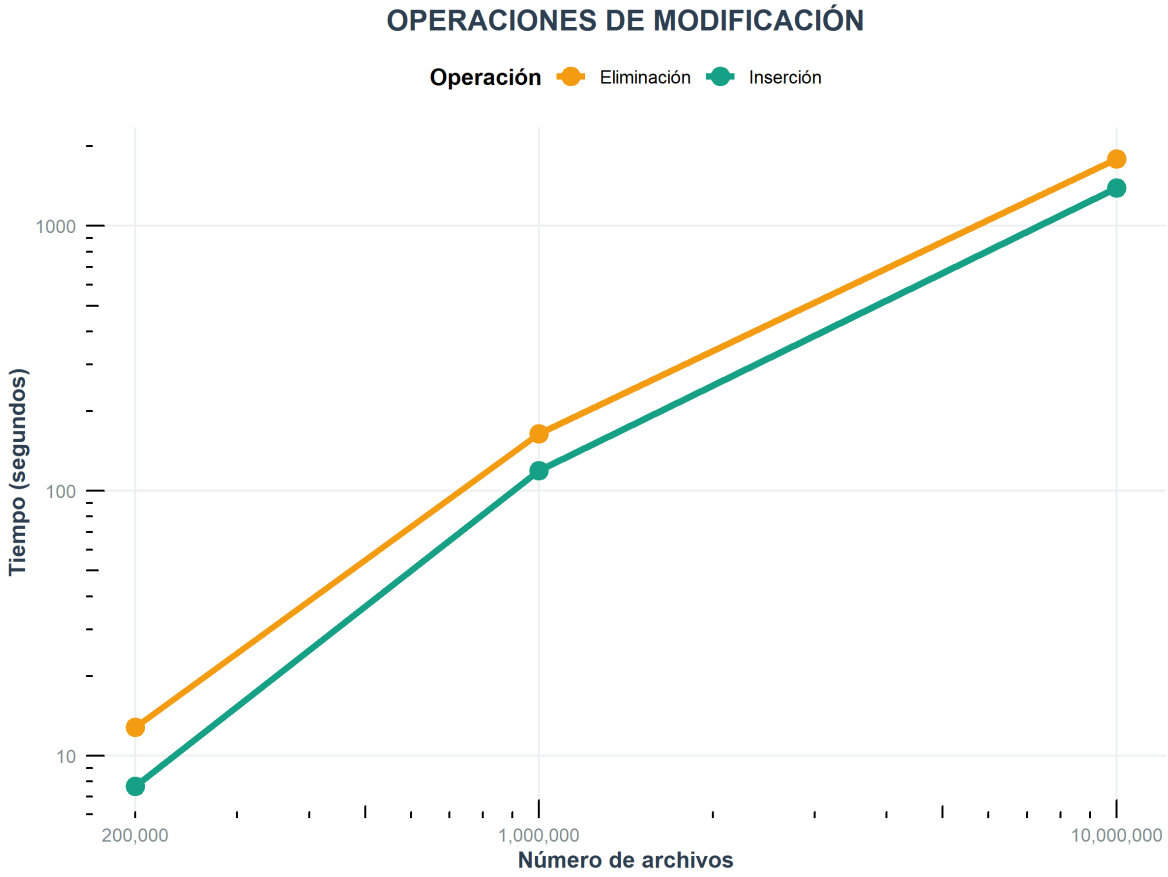
2) Búsquedas en el vector: 100.000 búsquedas aleatorias.

	Tamaño (cantidad de archivos del directorio cargado)		
	200.000	**1.000.000**	**10.000.000**
Tiempo búsqueda secuencial (microsegundos)	**605.675.310 (10 [min] aprox.)**	**3.030.113.927 (50 [min] aprox.)**	**30.096.504.682 (8,4 [hrs] aprox.)**
Tiempo búsqueda binaria (microsegundos)	**165.288 (0,17 [s] aprox.)**	**249.485 (0,25 [s] aprox.)**	**434.229 (0,43 [s] aprox.)**



3) Eliminación e Inserción: 100.000 eliminaciones aleatorias y luego 100.000 inserciones.

	Tamaño (cantidad de archivos del directorio cargado)		
	200.000	**1.000.000**	**10.000.000**
Tiempo eliminación (microsegundos)	**12.817.875 (12 [s] aprox.)**	**164.168.855 (2,7 [min] aprox.)**	**1.790.086.813 (30 [min] aprox.)**
Tiempo inserción (microsegundos)	**7.672.949 (7 [s] aprox.)**	**119.399.899 (2 [min] aprox.)**	**1.393.369.260 (23 [min] aprox.)**



Análisis asintótico (Big O)

El análisis asintótico es una poderosa herramienta que nos permite analizar que tan eficiente va a ser un algoritmo sin siquiera haberlo implementado. Como dice su nombre, analiza el comportamiento de nuestro algoritmo cuándo el tamaño de la entrada es significativamente grande. De esto, hay mucha información en internet (ver bibliografía). A continuación, se muestra el análisis asintótico de los algoritmos de búsqueda, eliminación e inserción.

1) Búsqueda secuencial:

```
function busquedaSecuencial (vector, objetivo) {
        for i = 0 to (vector.size() – 1) do {
            if (vector[i] == objetivo) then
                return i;
         }
         return -1;
}
```

Este algoritmo de búsqueda es el más simple de todos. Comienza a recorrer el vector desde el inicio, comparando cada elemento con el objetivo, y se detiene en cuánto lo encuentra. Sin embargo, si el objetivo no está en el vector, sigue comparando elementos hasta llegar al final (peor caso). La operación primitiva del algoritmo es la comparación de dos elementos del vector (el del recorrido y el buscado), ya que es la operación que permite resolver el problema. Si n es el tamaño del vector, entonces, este algoritmo hace n comparaciones en el peor caso, ya que debe recorrer todo el vector, comparando cada elemento con el objetivo. Por lo tanto, su complejidad temporal en el peor caso es: **O(n)**.

2) Búsqueda eficiente (binaria):

```
function busquedaBinaria (vector, objetivo){
        inicio = 0;
        fin = vector.size() – 1;
        while (inicio  <=  fin) do {
            medio = (inicio + fin) / 2;
            if vector[medio]==objetivo then
                return medio;
            if vector[medio] < objetivo then
                inicio = medio + 1;
            else
                fin = medio – 1;
        }
        return -1
    }
```

El algoritmo de búsqueda binaria es un poderoso algoritmo de ordenamiento que sigue la estrategia “divide y vencerás”, aunque solo funciona en estructuras ordenadas. Funciona buscando justo en la posición de en medio del vector, comparando el elemento de esa posición con el elemento buscado. Si coincide, se retorna esa posición del vector. Si el elemento buscado es mayor al elemento del vector con el que se está comparando, entonces se busca el elemento en la posición de en medio de la parte derecha del vector, desde la posición actual de búsqueda hasta el final. Por el contrario, si el elemento buscado es menor, entonces se busca el elemento en la posición de en medio de la parte izquierda del vector, desde el inicio hasta la posición actual de búsqueda. Se repite este proceso hasta encontrar el objetivo, o hasta concluir que éste no está en el vector. Dada esta descripción, es claro que el espacio de búsqueda se divide por dos en cada iteración, y que se hacen como máximo, $\log_2 n$ iteraciones (siendo n el tamaño del vector). Cada iteración tiene un costo

constante: una suma, una división, y hasta dos comparaciones.
Por lo tanto, su complejidad temporal en el peor caso es: **O(log(n))**.

3) Eliminación (con vector ordenado):

La función "eliminaElemento" recibe como argumento el string del elemento a eliminar en el vector. Internamente implementa una búsqueda binaria (**O(log(n)**) para encontrar la posición (pos) de ese string y eliminarlo. Luego, usando vector.erase(pos), se elimina dicho elemento del vector, lo que implica mover todos los elementos posteriores una posición hacia la izquierda, por lo que el costo del erase() es proporcional al número de elementos que deben moverse. El peor caso, sería eliminar el primer elemento del vector, lo que implicaría mover los n – 1 elementos restantes una posición hacia la izquierda, esto es, **O(n)**. Como el coste dominante es el desplazamiento lineal de elementos, la complejidad temporal del algoritmo de eliminación es: **O(n).**

4) Inserción (con vector ordenado):

Para insertar un elemento, se elaboró una búsqueda binaria adaptada, que retorna la posición (pos) dónde se debería insertar el nuevo elemento sin que se pierda el orden en el vector. Sin embargo, para asegurarse de que no hubiera errores, se optó por usar la función de la STL "std::lower_bound", la cual cumple la misma función. Luego se inserta con vector.insert(pos, valor), desplazando todos los elementos una posición hacia la derecha, desde pos en adelante. El coste del insert() es lineal en la cantidad de elementos que deben moverse. El peor caso sería insertar un elemento en la primera posición, lo que nos obligaría a mover n elementos una posición hacia la derecha en memoria. Como el paso de búsqueda no domina el tiempo total, la complejidad temporal del algoritmo de inserción en el peor caso es: **O(n)**.

Conclusiones

A lo largo de los experimentos realizados, podemos concluir varias cosas:

1. **Creación del vector**

La operación de carga inicial de los archivos mostró un crecimiento lineal en el tiempo respecto al número de elementos. Esto es coherente con una complejidad teórica de **O(n)**, ya que se recorren y almacenan *n* elementos en el vector. Los tiempos aumentaron de forma proporcional con vectores de 200.000, 1.000.000 y 10.000.000 archivos.

2. **Búsqueda secuencial**

Tal como predice la teoría, esta operación presenta una complejidad **O(n)** en el peor caso. Los tiempos de ejecución crecieron significativamente con el tamaño del vector, validando el comportamiento esperado: si se duplican los elementos, el tiempo casi se duplica también. En el directorio de 10M de archivos, se demoró **más de 8 horas** en completar todas las búsquedas. Los resultados reflejan que esta estrategia es muy poco eficiente para conjuntos de datos grandes.

3. **Búsqueda binaria**

Luego de ordenar el vector, se aplicó búsqueda binaria, que teóricamente tiene una complejidad **O(log n)**. En la práctica, los tiempos se mantuvieron extremadamente bajos incluso para 10 millones de elementos, en dónde, si comparamos las 8 horas de la búsqueda

secuencial vs. los 0,43 segundos de la búsqueda binaria, esta última fue unas **69.000 veces más rápida** que la búsqueda secuencial, lo que la hace increíblemente más eficiente. Esto valida que el algoritmo escala muy bien y que el costo de buscar en un conjunto ordenado es casi despreciable en comparación con otras operaciones, reforzando su uso cuando el costo de ordenamiento puede amortizarse.

4. **Ordenamiento (QuickSort)**

La operación de ordenamiento arrojó tiempos muy bajos en todos los casos, incluso en el de 10 millones de elementos. Quicksort tiene un comportamiento promedio de **O(n log n)** y estos resultados concuerdan con dicha expectativa, ya que el crecimiento fue muy controlado respecto al tamaño del vector.

5. **Eliminación de elementos**

Esta operación, basada en búsquedas y borrados aleatorios en el vector, presentó un crecimiento del tiempo mucho más elevado, especialmente en los vectores grandes. Esto puede explicarse por una complejidad cercana a **O(n)** por eliminación, ya que implica desplazamientos internos dentro del vector. El tiempo total creció fuertemente, lo que confirma su naturaleza lineal (o incluso peor en algunos contextos, si se considera el reacomodo de memoria).

6. **Inserción**

Similar al caso anterior, insertar elementos en un vector implica posibles desplazamientos. Si bien los datos fueron insertados al final o en posiciones aleatorias, el costo de recolocación afecta el rendimiento. Teóricamente, insertar al final es **O(1)** amortizado, pero insertar en posiciones arbitrarias es **O(n)**. El comportamiento experimental concuerda con esta evaluación, pues el tiempo de inserción también se incrementó significativamente con el tamaño del vector.

En general, los resultados empíricos obtenidos se alinean con las expectativas teóricas derivadas del análisis asintótico. Las operaciones **lineales** (creación, búsqueda secuencial, eliminación, inserción) escalaron de forma proporcional al tamaño del vector, mientras que las operaciones **logarítmicas o linealogarítmicas** (búsqueda binaria y ordenamiento) mostraron un crecimiento mucho más moderado. Esto reafirma la utilidad de la notación **Big O** para anticipar el comportamiento de algoritmos sobre estructuras de datos a gran escala. Respecto a si la estructura elegida (vector unidimensional), las grandes ventajas son la facilidad para tener los datos ordenados y el tiempo constante O(1) para el acceso aleatorio, y las desventajas son el gasto de recursos computacionales a la hora de eliminar o insertar elementos.

Bibliografía

Asymptotic Analysis. (n.d.). GeeksforGeeks. https://www.geeksforgeeks.org/asymptotic-notation-and-analysis-based-on-input-size-of-algorithms/

Big O notation. (n.d.). GeeksforGeeks. https://www.geeksforgeeks.org/asymptotic-notations-and-how-to-calculate-them/