**Informe de segundo taller:**

**Análisis de complejidad y pruebas de algoritmos de matrices poco pobladas en C++**

Integrantes:

Francisco Romero O.

Profesor:

José Luis Veas (Paralelo C1)

Fecha de entrega: 24 de octubre, 2025

**Índice**

Tabla de contenido

[1. Introducción 3](#_Toc212233468)

[2. Estructuración de la matriz 4](#_Toc212233469)

[2.1. Conceptual 4](#_Toc212233470)

[2.2. Implementación 5](#_Toc212233471)

[3. Análisis de los algoritmos de Matriz 7](#_Toc212233472)

[3.1. Inserción 7](#_Toc212233473)

[3.1.1 Búsqueda y creación de cabeceras 7](#_Toc212233474)

[3.1.2 Inserción de valor en la columna 8](#_Toc212233475)

[3.1.3 T(n) de la inserción 8](#_Toc212233476)

[3.2. Obtención 9](#_Toc212233477)

[3.3. Multiplicación 10](#_Toc212233478)

[4. Pruebas de tiempo a los algoritmos 12](#_Toc212233479)

[4.1. Tiempos de inserción 12](#_Toc212233480)

[4.2. Tiempos de multiplicación 14](#_Toc212233481)

[4.3. Tiempos de obtención 16](#_Toc212233482)

[4.3. Resultados totales 18](#_Toc212233483)

[4.4. Comparación de rendimiento entre dispositivos 20](#_Toc212233484)

[5. Conclusión de las pruebas 22](#_Toc212233485)

[Referencias bibliográficas 23](#_Toc212233486)

# 1. Introducción

Una *matriz poco poblada*, según el Doctor en Ingeniería Computacional, Fatih Karabiber, es un tipo especial de matriz en el que la mayoría de sus elementos son *cero* y, por tanto, puede ser representada solamente por aquellos elementos *no-cero* que la componen. Esta estructura *dispersa*, permite aprovechar los vacíos para apuntar a técnicas y estructuras de datos más eficientes y económicos sin almacenar tantos *ceros* (Cs, 2025).

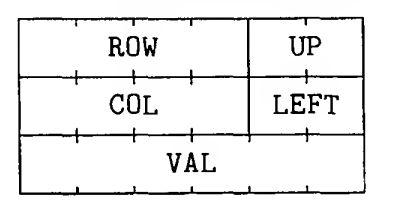
A partir de este enfoque en la optimización de recursos computacionales, el presente informe detalla un análisis algorítmico y experimental de un proyecto de implementación de *Matriz Poco Poblada* o *Sparse Matrix* en *C++* utilizando *listas enlazadas*.

# 2. Estructuración de la matriz

## 2.1. Conceptual

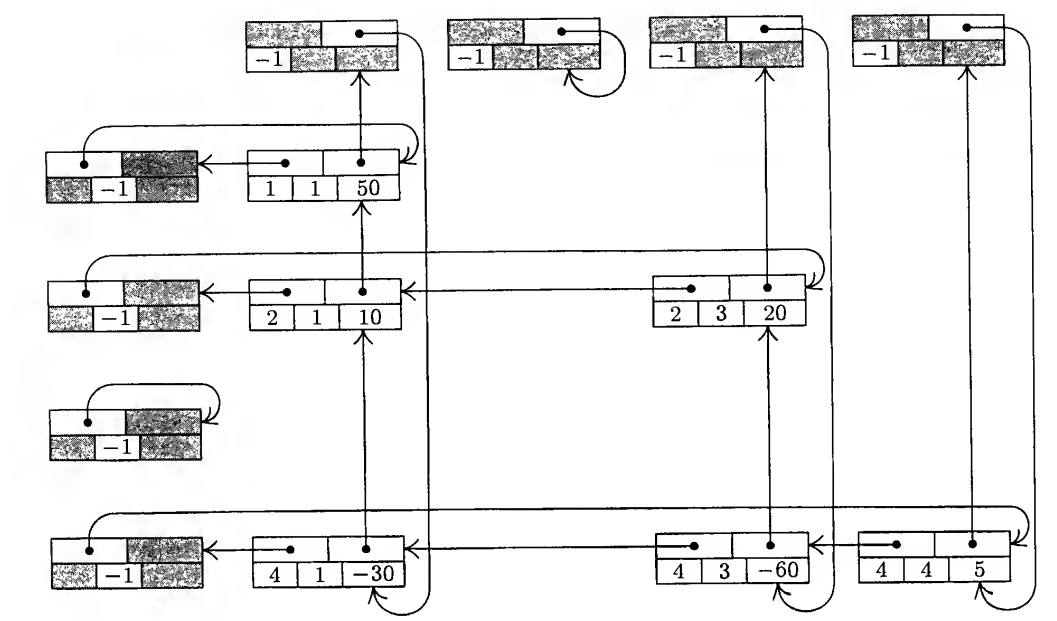
La estructura implementada en el proyecto se basa en la representación de las *matrices poco pobladas*, del matemático y científico computacional Donald Knuth.

En el famoso *The art of computer programming* *(Vol. 1)*, el autor presenta un modelo de listas enlazadas circulares para cada fila y columna, donde cada nodo tiene cinco campos:



*Fuente de imagen: The art of computer programming (Vol. 1), pp. 302.*

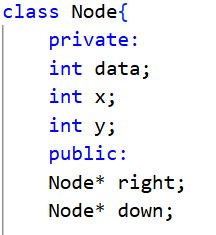
Aquí ROW y COL son los índices de fila y columna del nodo; VAL es el valor almacenado en esa parte de la matriz; LEFT y UP son enlaces a la siguiente entrada distinta de cero a la izquierda en la fila, o hacia arriba en la columna, respectivamente. Hay nodos especiales de encabezado de lista, BASEROW[i] y BASECOL[j], para cada fila y columna. (Knuth,1997, pp. 302)



*Fuente de imagen: The art of computer programming (Vol. 1), pp. 303.*

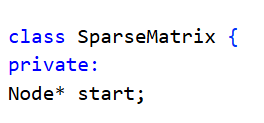
## 2.2. Implementación

De forma similar se modeló el proyecto, en el que cada nodo de la matriz contiene *X* e *Y* como sus coordenadas, *data* como su valor almacenado, y *right* y *down* como punteros hacia a la derecha (desplazamiento a lo largo de la fila) y abajo (desplazamiento a lo largo de la columna), respectivamente.

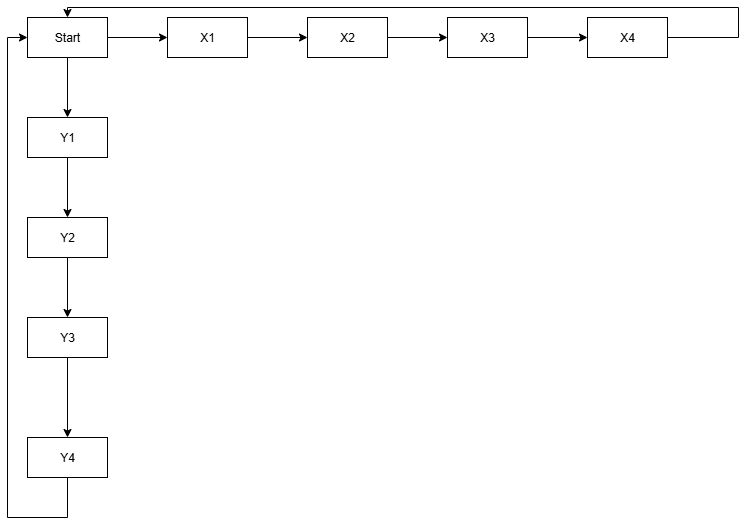


*Fuente de imagen: Creación propia.*

Y la matriz, a diferencia del modelo Knuth, se compone por un nodo Start, que representa una posición (0,0) y de donde se enlazan los nodos cabecera.



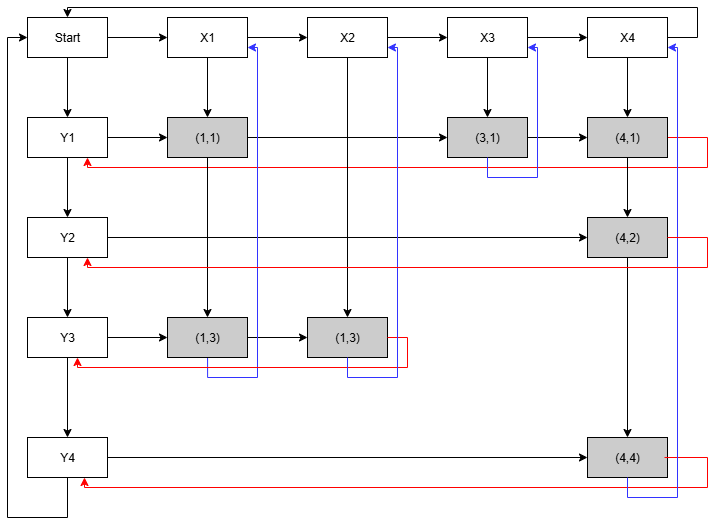
*Fuente de imagen: Creación propia*



Los nodos cabecera son nodos que contienen *data* = -1, y apuntan a aquellos nodos con los que coinciden en fila y/o columna; son útiles para organizar la matriz y permiten hacer una búsqueda por posición. Los enlaces al interior de la matriz son todos simples (solo de avance hacia abajo y derecha) y circulares.

*Fuente de imagen: Creación propia*

De este modo, cada valor (nodo) de la matriz está enlazado a la lista de cada cabecera X e Y, junto con los otros valores. Además, el último valor de cada lista apunta hacia la cabecera, de forma circular.



*Fuente de imagen: Creación propia*

# 3. Análisis de los algoritmos de Matriz

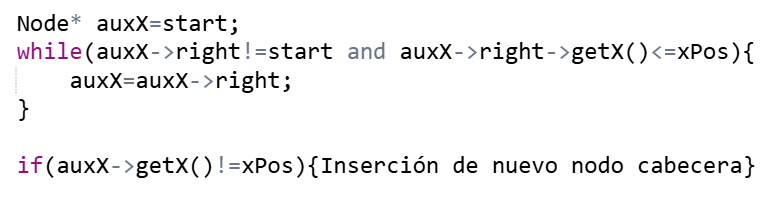
## 3.1. Inserción

El método de inserción de *SparseMatrix* consta de dos procesos, la búsqueda y creación de las cabeceras X e Y según las coordenadas ingresadas, y la inserción del nuevo nodo en su respectiva fila y columna, manteniendo siempre el orden y circularidad de las listas enlazadas.

A continuación, se analiza la complejidad algorítmica del método:

### 3.1.1 Búsqueda y creación de cabeceras

Para realizar la búsqueda de cabeceras, se itera a lo largo de la lista de cabeceras X hasta llegar el nodo anterior al buscado. Después, se revisa si el nodo buscado existe, o sea, si la columna del sucesor de auxX es igual a la columna donde se busca insertar o modificar el nuevo valor. De ser distintas, significa que el nodo cabecera aún no existe, y se crea e inserta.



*Fuente de imagen: Creación propia*

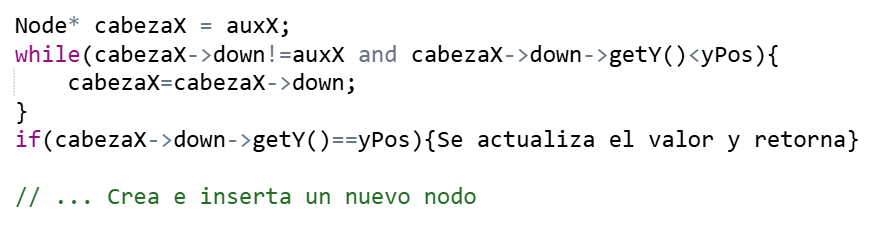
* La operación activa dentro de este proceso es el movimiento de punteros y comparar coordenada en cada iteración.
* El mejor caso en este algoritmo es cuando la matriz está vacía, porque no existen nodos cabecera que recorrer, de modo que ni siquiera entra al while.
* El peor caso es aquel donde se busca una cabecera lejana hasta al final de la lista, donde existen todos los nodos cabecera intermedios.

Tras crear o encontrar el nodo de cabecera X, se repite la misma operación para encontrar la cabecera Y, con la diferencia de que el puntero avanza hacia abajo, a lo largo de la lista enlazada de filas.

Se dan, entonces, las mismas operaciones activas con los mismos casos.

### 3.1.2 Inserción de valor en la columna

Tras encontrarse la cabecera deseada, se itera a lo largo de la columna de valores hasta llegar al nodo anterior al buscado. Después, se revisa si el nodo buscado existe, o sea, si la fila del sucesor de *cabezaX* es igual a la fila donde se busca insertar o modificar el nuevo valor. De ser distintas, significa que el nodo aún no existe, y se crea e inserta. De lo contrario, solo modifica su valor.



*Fuente de imagen: Creación propia*

* La operación activa dentro de este proceso es el movimiento de punteros y comparar coordenada en cada iteración.
* El mejor caso en este algoritmo es cuando la columna está vacía, porque no existen más nodos que recorrer, de modo que ni siquiera entra al while.
* El peor caso es aquel donde se busca una cabecera lejana hasta al final de la lista, donde existen todos los nodos intermedios.

Tras crear o encontrar el nodo, se repite la misma operación para la fila de la cabecera Y, con la diferencia de que el puntero avanza hacia la derecha, a lo largo de la lista enlazada de nodos de la fila Y.

Se dan, entonces, las mismas operaciones activas y los mismos casos.

### 3.1.3 T(n) de la inserción

Considerando como O(1) las operaciones del tipo:

* Creación y movimiento de puntero.
* Comprobación de while.
* Creación de nodos.

El peor caso sería una matriz cuadrada completamente poblada a excepción de la esquina inferior derecha, donde se busca agregar un valor. Para ello se hacen cuatro recorridos equivalentes, uno en lista cabeceras X hasta el final, el segundo en la lista cabeceras Y hasta el final, el tercero a lo largo de la columna X hasta el final y el cuarto a lo largo de la fila Y hasta el final.

En el peor caso, dada una matriz :

Para el ciclo de búsqueda de cabecera X:

Para el ciclo de búsqueda de cabecera Y:

Para el ciclo de búsqueda en columna X:

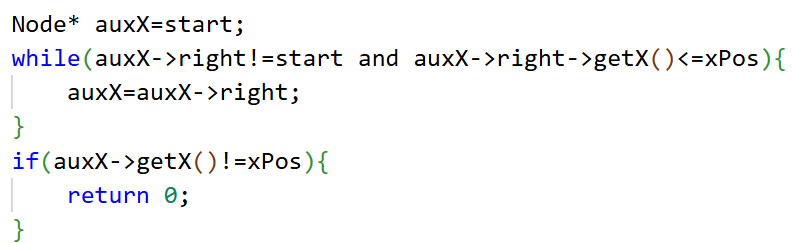
Para el ciclo de búsqueda en fila Y:

Entonces,

## 3.2. Obtención

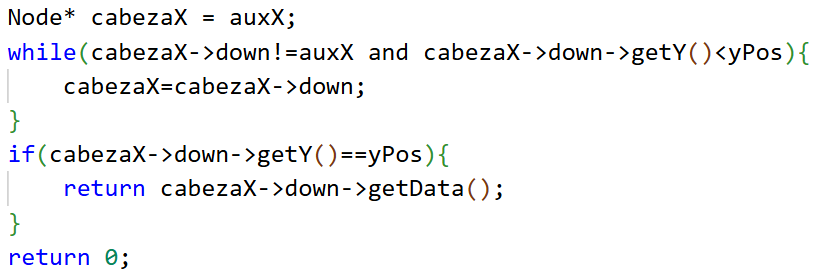
El método para obtener un valor de la matriz funciona de similar manera que la inserción. Consta de la búsqueda del nodo cabecera X, y en el caso positivo, en la búsqueda del nodo específico en esa columna.

Recorrido de lista de cabeceras X



*Fuente de imagen: Creación propia*

Recorrido de columna tras encontrar cabecera X



*Fuente de imagen: Creación propia*

* La operación activa dentro de este proceso, igual que en *insertar*, es el movimiento de punteros y comparar coordenada en cada iteración del ciclo.
* El mejor caso en este algoritmo es cuando la matriz está vacía, porque no existen nodos cabecera que recorrer, y tras terminar el primer ciclo, se retorna 0.
* El peor caso es aquel donde se busca un elemento presente en la esquina inferior derecha de la matriz, donde existen todos los nodos cabecera intermedios y cuya columna está completamente poblada, de modo que se deben recorrer todos los nodos de la columna hasta el final.

En el peor caso, dada una matriz :

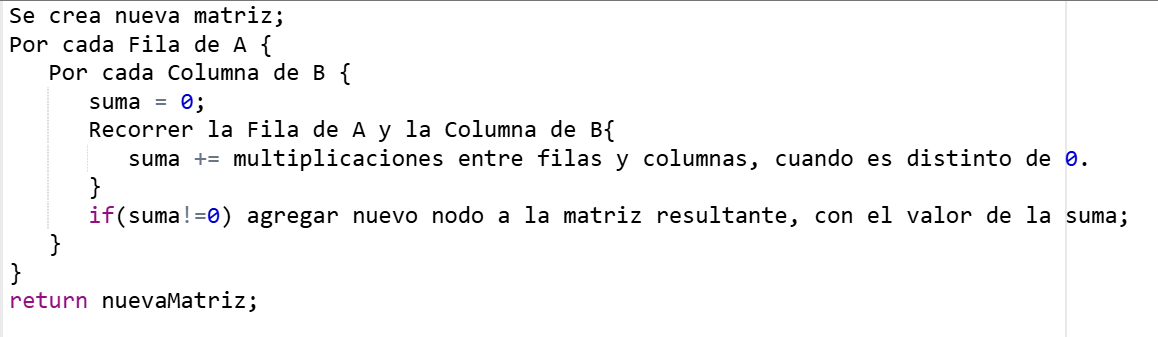
Para el ciclo de búsqueda de cabecera X:

Para el ciclo de búsqueda de nodo en columna X:

Entonces,

## 3.3. Multiplicación

El método multiplicar es significativamente distinto de los dos métodos anteriores. Para multiplicar una matriz poco poblada no se necesita tener dos matrices de dimensiones compatibles, puesto que al almacenarse solo los valores no-cero, las matrices pueden tener cualquier dimensión que se desee. De modo que deben multiplicarse de la manera tradicional, recorriendo con punteros las filas de la primera matriz y las columnas de la segunda matriz.



*Fuente de imagen: Creación propia*

Dado el pseudocódigo anterior, el método itera a lo largo de cada fila de la matriz A una cantidad de veces dada por el número de columnas de la matriz B, coincidan éstas o no con las filas.

* La operación activa de este algoritmo es el movimiento de punteros y comparar coordenadas por cada avance de los punteros a lo largo de las filas y columnas.
* El mejor caso en este algoritmo es cuando la matriz está vacía, porque no existen nodos cabecera que recorrer, y tras terminar el primer ciclo, se retorna 0.
* El peor caso es la multiplicación de dos matrices densas cuadradas de la misma dimensión, donde cada fila de A tiene todos sus elementos coincidentes con los de cada columna de B. De modo que siempre existe la operación de suma-producto e inserción en la nueva matriz.

En el peor caso, dada la matriz A y la matriz B :

Para el recorrido de todas las filas de A:

Para el recorrido de todas las columnas de B:

Para el recorrido de una fila de A:

Para el recorrido de una columna de B:

Para la multiplicación de una fila de A por una columna de B (suma-producto nodo a nodo):

Para la multiplicación de una fila de A por todas las columnas de B:

Para la multiplicación todas las filas de A por todas las columnas de B:

Entonces,

# 4. Pruebas de tiempo a los algoritmos

Para evaluar temporalmente los algoritmos previamente analizados, se realizaron pruebas de tiempo de ejecución dadas distintas condiciones para las matrices, y en dos distintos dispositivos.

Las matrices se alimentaron con cinco sets de datos: 50, 250, 500, 1000 y 5000 nodos distintos y en posiciones aleatorias, pero sujetos a distintas densidades: densidad baja (menor o igual al 40%) y alta (mayor o igual al 70%).

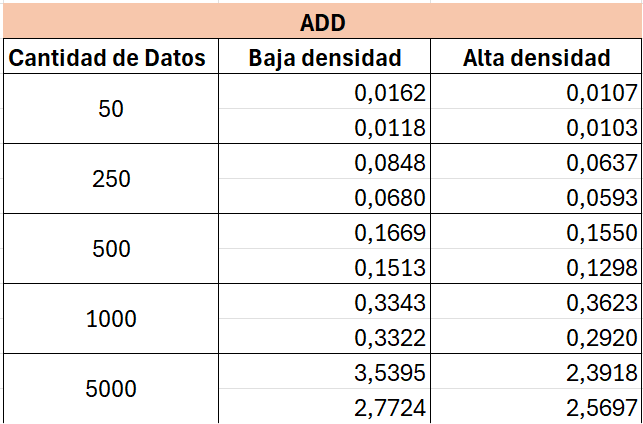
Además, se probó en dos computadores distintos:

1. HP Pavilion x360:
   1. Memoria RAM: 8 GB
   2. Procesador: 12th Gen Intel® Core™ i5-1235U (12CPUs)
   3. Sistema operativo: Windows 11 Home
2. MacBook Air:
   1. Memoria RAM: 8 GB
   2. Procesador: Apple M1
   3. Sistema operativo: macOS 15.6.1

Se realizaron dos pruebas por cada caso de densidad y por cada set de datos. En total, 20 pruebas por cada algoritmo y en cada computador.

## 4.1. Tiempos de inserción

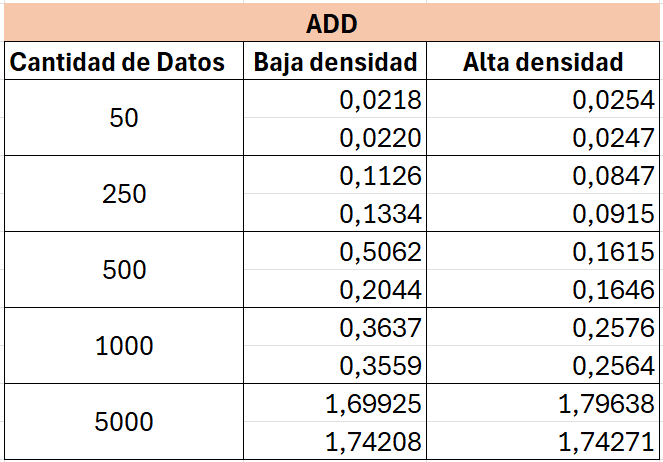
El HP Pavilion x360 presentó los siguientes resultados en milisegundos (ms):



*Fuente de tabla: Creación propia*

*Fuente de gráfico: Creación propia*

El MacBook Air presentó los siguientes resultados en milisegundos (ms):

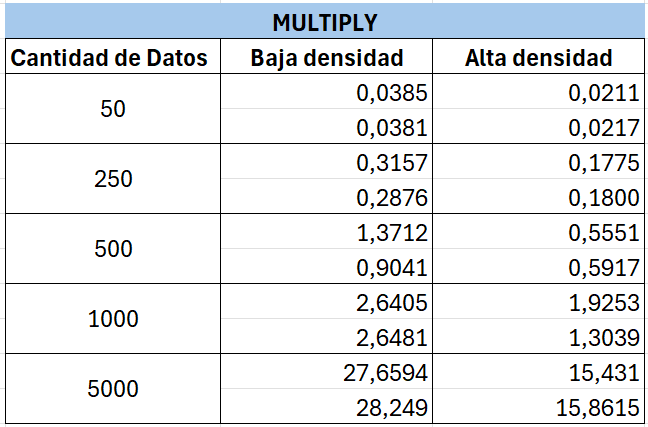


*Fuente de imagen: Creación propia*

*Fuente de gráfico: Creación propia*

## 4.2. Tiempos de multiplicación

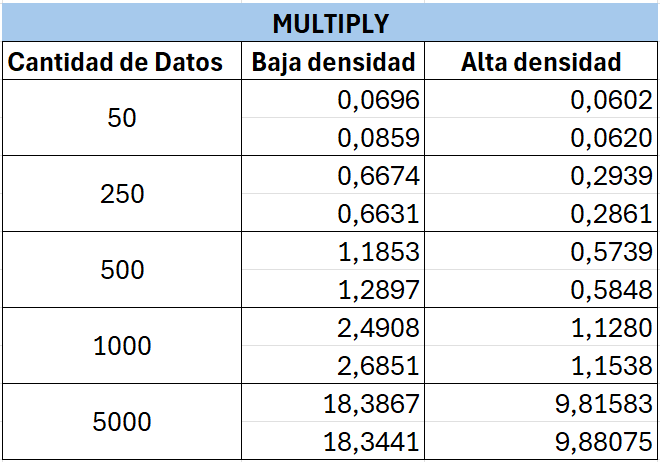
El HP Pavilion x360 presentó los siguientes resultados en milisegundos (ms):



*Fuente de imagen: Creación propia*

*Fuente de gráfico: Creación propia*

El MacBook Air presentó los siguientes resultados en milisegundos (ms):

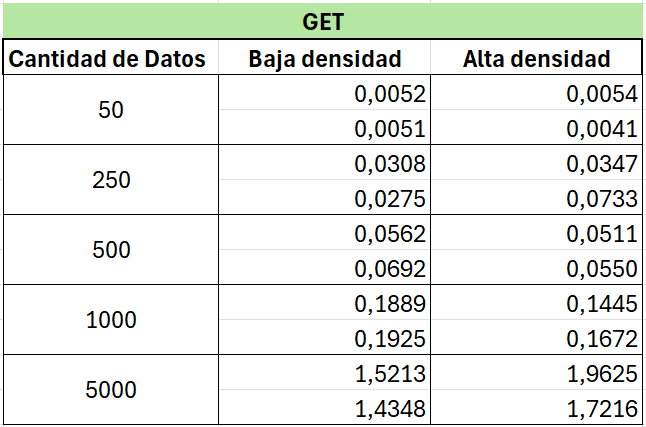


*Fuente de imagen: Creación propia*

*Fuente de gráfico: Creación propia*

## 4.3. Tiempos de obtención

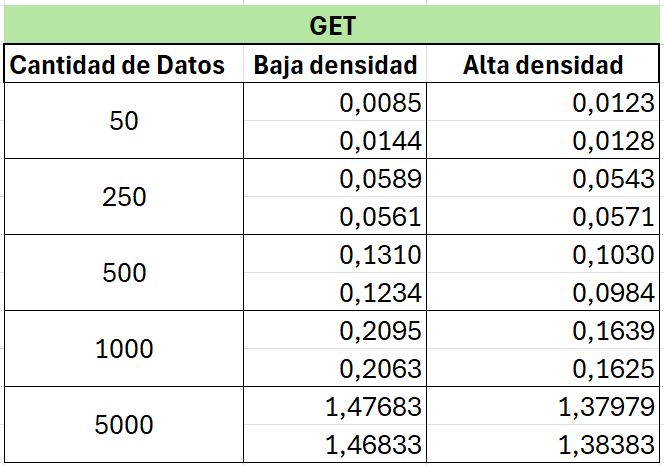
El HP Pavilion x360 presentó los siguientes resultados en milisegundos (ms):



*Fuente de imagen: Creación propia*

*Fuente de gráfico: Creación propia*

El MacBook Air presentó los siguientes resultados en milisegundos (ms):

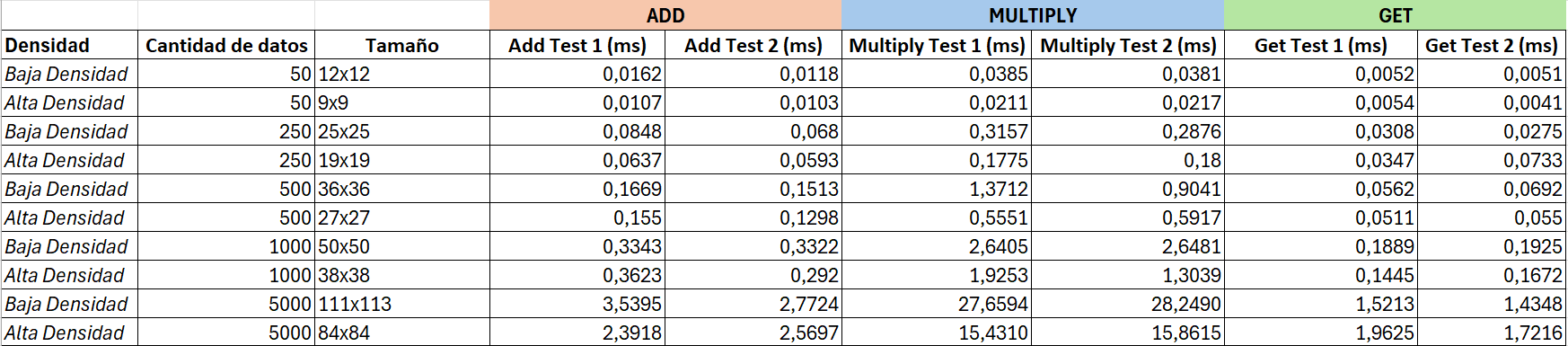


*Fuente de imagen: Creación propia*

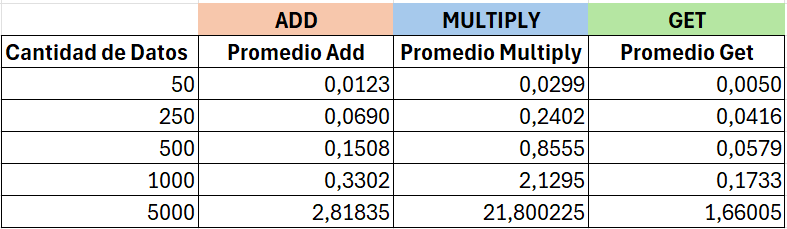
*Fuente de gráfico: Creación propia*

## 4.3. Resultados totales

Para el HP Pavilion x360, se tuvieron los resultados:



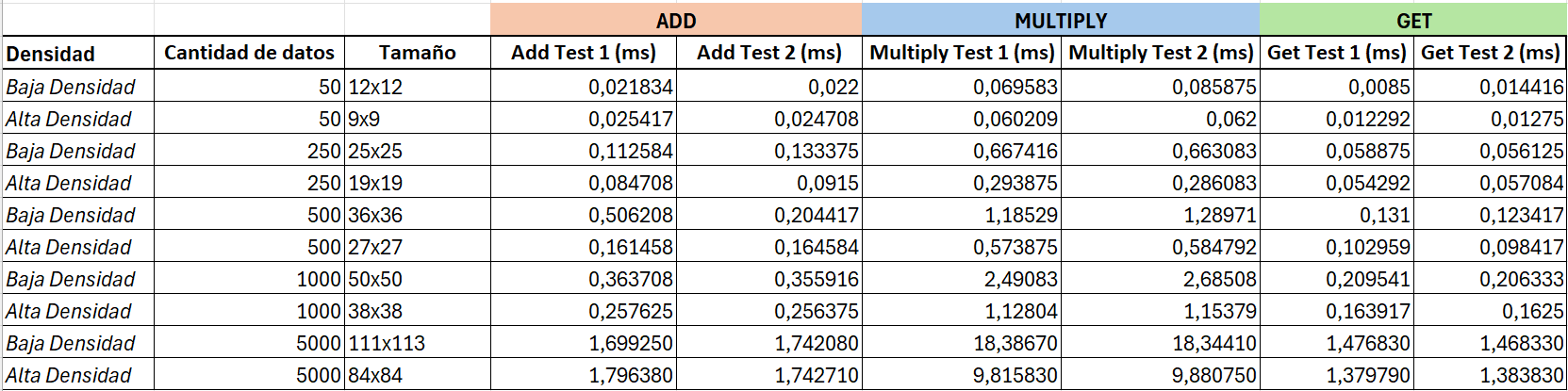
*Fuente de imagen: Creación propia*



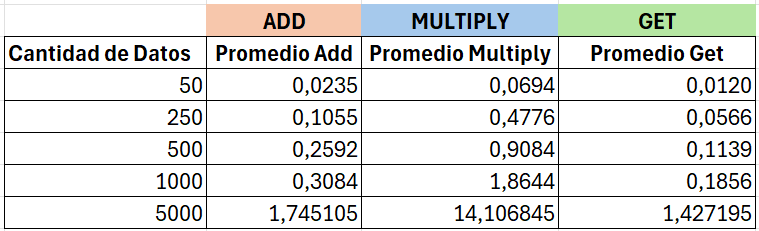
*Fuente de imagen: Creación propia*

*Fuente de gráfico: Creación propia*

Para el MacBook Air, se tuvieron los resultados:



*Fuente de imagen: Creación propia*



*Fuente de imagen: Creación propia*

*Fuente de gráfico: Creación propia*

## 4.4. Comparación de rendimiento entre dispositivos

Comparando los resultados de tiempo del método de inserción:

*Fuente de gráfico: Creación propia*

Comparando los resultados de tiempo del método de multiplicación:

*Fuente de gráfico: Creación propia*

Comparando los resultados de tiempo del método de obtención:

*Fuente de gráfico: Creación propia*

# 5. Conclusión de las pruebas

Los resultados de las pruebas son esclarecedores respecto al efecto de la complejidad algorítmica de las distintas funciones. Todas las pruebas demuestran un crecimiento en su tiempo de ejecución a medida que la cantidad de datos aumentan, pero las pendientes visibles de cada algoritmo y sus pruebas son diferentes entre sí, y esto es por la complejidad de las funciones.

Esto puede observarse al comparar los promedios de tiempos de ejecución de los tres algoritmos, donde la inserción y obtención mantienen curvas casi constantes, pero la multiplicación presenta un rápido crecimiento exponencial dado su , de modo que, mientras la cantidad de datos se mantiene relativamente pequeña (menores a 1000) su crecimiento no explota, cosa que sí sucede entre los 1000 y 5000 datos.

Además de las diferencias de complejidad, otro fenómeno que afecta el tiempo de ejecución es la densidad de las *matrices poco pobladas,* donde se observa que los algoritmos demoran más a la hora de trabajar con matrices de baja densidad (o mayor dispersión). Esto se da, principalmente, porque a mayor dispersión se necesita mayor tiempo para recorrer la totalidad de la matriz (la dimensión máxima es mayor), a diferencia de una densa, que tiene menor tamaño y menos cabeceras que recorrer.

Las diferencias de hardware entre los dispositivos en los que se probaron los distintos algoritmos también marcaron una diferencia significativa en los resultados de tiempo. MacBook Air demostró superioridad de rendimiento a medida que la cantidad de datos aumentaba. En los resultados de los tres algoritmos, Hp Pavilion x360 presenta ejecuciones más rápidas en el rango entre 50 y 1000 datos, desde donde sus tiempos comienzan a elevarse por sobre los de MacBook Air; así el procesador Apple M1 pone en evidencia su potencia.

Para concluir, las matrices poco pobladas son estructuras eficientes e indispensables para almacenar grandes cantidades de datos, ya que solo utilizan la información no vacía y permiten organizar los datos, de modo que resulta ser un método esencial, y una solución optimizada en recursos.

# Referencias bibliográficas

Karabiber, F. (s. f.). Sparse Matrix.

https://www.learndatasci.com/glossary/sparse-matrix/

Cs. (2025, 2 octubre). Sparse matrices · CS 357 Textbook. https://cs357.cs.illinois.edu/textbook/notes/sparse.html#goals

Knuth, D. E. (1997). The Art of Computer Programming: Fundamental Algorithms (Vol. 1). https://ia803207.us.archive.org/8/items/compatitive-programming/Books/The\_Art\_of\_Computer\_Programming%20-%20Vol%201\_text.pdf