Algoritmos de ordenamiento y multiplicación de matrices

IVÁN VILLARROEL R.[1]

[1] *Departamento de Ingeniería Informática y Ciencias de la Computación, Universidad de Concepción, 2023, Concepción, Chile*
[1] *ivillarroelr@inf.udec.cl*

Compiled April 28, 2023

Este trabajo explora algoritmos de ordenamiento y multiplicación de matrices, analizando su complejidad asintótica y casos extremos de rendimiento. Se discuten algoritmos clásicos, así como las notaciones O, Θ y Ω para caracterizar su eficiencia. El estudio compara resultados teóricos y experimentales, destacando la importancia de la verificación empírica en el desarrollo de algoritmos eficientes y aplicables a casos de uso modernos.

1. INTRODUCCIÓN

El análisis asintótico es una herramienta valiosa para evaluar el rendimiento de algoritmos en función del tamaño de los datos de entrada. Mediante la comparación de resultados teóricos y experimentales, este enfoque permite validar, refinar o desafiar las predicciones teóricas. La aplicación del análisis asintótico en resultados experimentales es esencial en áreas como el aprendizaje automático, optimización y procesamiento de datos, proporcionando una base sólida para seleccionar y mejorar algoritmos en función de casos de uso específicos, y permitiendo la identificación de discrepancias entre el comportamiento esperado y el observado en la práctica.

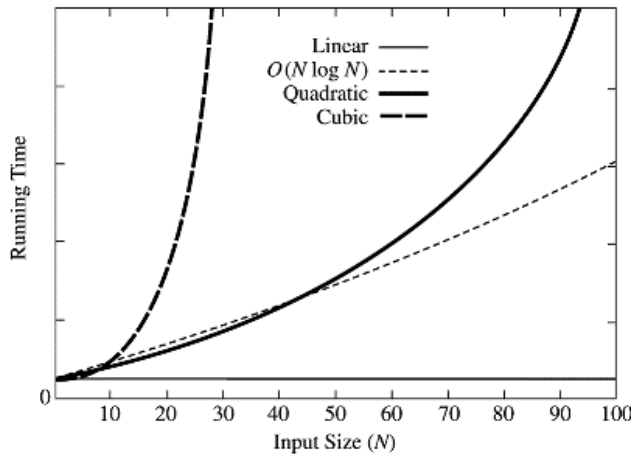

Fig. 1. Complejidades teóricas en función del input

A. Complejidad asintótica

La complejidad asintótica es un enfoque para describir el comportamiento de los algoritmos en función del tamaño de los datos de entrada, utilizando notaciones como $\mathcal{O}$. θ y Ω. Permite analizar y comparar la eficiencia de diferentes algoritmos en contextos específicos. El mejor y peor caso corresponden a los escenarios de rendimiento óptimo y más desfavorable, respectivamente, proporcionando una visión de los límites de eficiencia en diferentes situaciones. θ permite describir el comportamiento de un algoritmo con presición cuando realiza un trabajo promedio. Estos análisis son fundamentales para seleccionar y mejorar algoritmos adecuados a casos de uso particulares.

B. Algoritmos de ordenamiento

Los algoritmos de ordenamiento son técnicas esenciales en ciencias de la computación para organizar conjuntos de datos en un orden específico, como numérico o alfabético. Con algoritmos como QuickSort, MergeSort y BubbleSort, se abordan diversos problemas prácticos en áreas como búsqueda de información, procesamiento de datos, optimización y análisis estadístico. La eficiencia y complejidad de estos algoritmos varía, lo que influye en su aplicabilidad a diferentes situaciones. Comprender y seleccionar el algoritmo de ordenamiento adecuado permite desarrollar soluciones más eficientes y efectivas en la manipulación de datos en la vida real.

C. Algoritmos de multiplicación de matrices

Los algoritmos de multiplicación de matrices, como el método de multiplicación clásico, optimizado y el algoritmo de Strassen, tienen aplicaciones prácticas en áreas como el álgebra lineal, el procesamiento de señales, la optimización y el aprendizaje automático. La eficiencia y complejidad de cada algoritmo varían, influyendo en su aplicabilidad a diferentes situaciones.

2. DESCRIPCIÓN DE LOS ALGORITMOS A SER COMPARADOS

A. Referencia en C++

La implementación de los algoritmos en C++ puede ser encontrada en el repositorio público del proyecto desarrollado para este experimento, en los archivos source/SortingAlgorithms.cpp y source/MatrixAlgorithms.cpp[7].

B. Algoritmos de Ordenamiento

Un algoritmo de ordenamiento se define como una serie de pasos lógicos y matemáticos que se aplican a una colección de datos, transformando estos en una lista ordenada de acuerdo a diferentes criterios, siendo la definición de estos criterios el origen de su utilidad. Por ejemplo, en el ámbito comercial, una serie de datos se pueden ordenar por fecha de transacción o por cliente, a su vez, en el ámbito científico, una serie de datos se pueden ordenar por fecha de experimento o magnitud numérica de los resultados[p. 148 1].

Existe un creciente interés en este campo a nivel académico, en particular en hacerlos cada vez más eficientes y en nuevos enfoques para mejorar su escalabilidad[2].

Existen diversos algoritmos de ordenamiento, los cuales varían en complejidad y eficiencia, dependiendo del método ocupado para organizar los datos y de la naturaleza y características de los datos a organizar. Para poder sacar mayor provecho de estos, es importante entender las diferencias y elegir el más adecuado a cada situación[p. 39 3].

Entre los más conocidos, encontramos bubble sort, insertion sort, merge sort, quick sort, selection sort, heap sort, radix sort, counting sort, bucket sort, shell sort, cocktail sort, comb sort entre muchos otros[p. 8 3].

En el presente estudio, analizaremos experimentalmente la complejidad asintótica de los primeros cuatro y adicionalmente, la implementación sort de la Standard Template Library (STL por sus siglas en inglés) de C++[5]. El código fuente de los algoritmos utilizados puede ser encontrado en mi repositorio público [link a bibliografia] archivo source/SortingAlgorithms.cpp

B.1. Bubblesort

Bubblesort es un algoritmo simple 1 que funciona a través de la comparación y el intercambio de elementos adyacentes en una lista. Para una breve descripción del algoritmo, tenemos que este comienza en el primer elemento de la lista y va comparando cada par de elementos adyacentes, intercambiándolos de posición si están en el orden incorrecto, así, avanzará y repetirá el proceso hasta que toda la lista esté ordenada. El algoritmo termina cuando no se requiere ningún intercambio para completar un recorrido completo de la lista[6].

Algorithm 1. Algoritmo Bubblesort

```
1: procedure BUBBLESORT(arr)                ▷ Ordena arr en orden ascendente
2:     n ← longitud(arr)
3:     for i = 0 hasta n − 2 do
4:         for j = 0 hasta n − i − 2 do
5:             if arr[j] > arr[j + 1] then
6:                 Intercambiar arr[j] y arr[j + 1]
```

Teniendo en cuenta el algoritmo 1, la complejidad temporal en el **peor caso** se puede calcular a partir de las comparaciones de cada iteración del primer bucle $(n-1)$ y segundo bucle $(n-1)$ es de $O(n^2)$, lo que significa que es ineficiente para grandes conjuntos de datos al tardar una cantidad de tiempo cuadrática en ordenarlos. El **mejor caso**, al ser una lista ordenada, tiene una complejidad temporal de $O(n)$, lo que significa que el algoritmo solo hace una pasada para verificar que la lista ya está ordenada

B.2. Insertion Sort

Insertion Sort es un algoritmo básico de ordenamiento2, su funcionamiento de manera general es que recorre una lista de elementos y coloca cada elemento en su posición correcta dentro de una sublista ordenada. Comienza con una sublista de tamaño uno, que se considera ordenada, luego va recorriendo la lista de elementos restantes y para cada uno, lo compara con los elementos de la sublista ordenada y los coloca en la posición correcta en la sublista[8].

La complejidad temporal para Insertion Sort en el **peor caso** es de $O(n^2)$, esto ocurre cuando la lista está en orden inverso, necesitando que cada elemento deba ser movido a la posición correcta en la sublista ordenada en cada paso del algoritmo. La complejidad temporal en el **mejor caso** es de $O(n)$ lo que ocurre cuando la lista ya está ordenada, por lo que el algoritmo solo necesita recorrer la lista una vez y verificar que está ordenada y sin necesitar hacer intercambios.

Algorithm 2. Algoritmo Insertion Sort

```
1: procedure INSERTIONSORT(arr)             ▷ Ordena arr en orden ascendente
2:     n ← length(arr)
3:     for i = 1 hasta n − 1 do
4:         key ← arr[i]
5:         j ← i − 1
6:         while j ≥ 0 y arr[j] > key do
7:             arr[j + 1] ← arr[j]
8:             j ← j − 1
9:         arr[j + 1] ← key
```

B.3. Quick Sort

Quick Sort es un algoritmo de ordenamiento que se basa en el paradigma de diseño divide y conquista3. El algoritmo selecciona un pivote en la lista de elementos y luego divide la lista en dos subconjuntos: uno con elementos menores que el pivote y otro con elementos mayores que el pivote. Finalmente, los dos subconjuntos se ordenan recursivamente usando Quick Sort.

Sobre su complejidad temporal, el **mejor caso** es $O(nlogn)$ y ocurre cuando el pivote se selecciona de manera que los subconjuntos tienen aproximadamente la misma longitud. En este caso, el algoritmo divide la lista en subconjuntos de tamaño similar en cada paso, lo que minimiza el número de divisiones necesarias y reduce la complejidad temporal total.

El **peor caso** de Quick Sort es $O(n^2)$. Este caso ocurre cuando el pivote se selecciona de manera que uno de los subconjuntos es significativamente más grande que el otro en cada paso, lo que lleva a una división desbalanceada y un mayor número de divisiones necesarias.

A pesar de su peor caso, Quick Sort es un algoritmo muy popular debido a su eficiencia en la mayoría de los casos. Es especialmente útil para ordenar conjuntos de datos grandes o complejos, como listas de objetos o estructuras de datos. Además, Quick Sort es un algoritmo in-place, lo que significa que no requiere memoria adicional para ordenar la lista.

Algorithm 3. Algoritmo Quicksort

```
1: procedure QUICKSORT(arr, low, high) ▷ Ordena arr en orden ascendente
2:     if low < high then
3:         pivot ← PARTITION(arr, low, high)
4:         QUICKSORT(arr, low, pivot − 1)
5:         QUICKSORT(arr, pivot + 1, high)
6: procedure PARTITION(arr, low, high)
7:     pivot ← arr[high]
8:     i ← low − 1
9:     for j = low hasta high − 1 do
10:        if arr[j] ≤ pivot then
11:            i ← i + 1
12:            Intercambiar arr[i] y arr[j]
13:    Intercambiar arr[i + 1] y arr[high]
14:    return i + 1
```

B.4. STL Sort

La función sort de la biblioteca estándar de C++ (STL) es una función de ordenamiento que se utiliza para ordenar los elementos de un contenedor, como un vector o una lista. La función sort utiliza el algoritmo de ordenamiento introsort, que combina Quick Sort, Heap Sort e Insertion Sort, para proporcionar una implementación eficiente y estable de ordenamiento.

Para la complejidad temporal, la función sort en el **peor caso** es de $O(nlogn)$, en el **mejor caso**, la complejidad temporal de la función sort es $O(n)$, donde n es el número de elementos en el contenedor. En la práctica, la función sort de STL es muy eficiente y generalmente supera a otros algoritmos de ordenamiento como Quick Sort y Merge Sort[9].

B.5. Merge Sort

Merge Sort es un algoritmo de ordenamiento que se basa en el enfoque divide y conquista. El algoritmo divide la lista de elementos en dos mitades iguales, las ordena recursivamente y luego las fusiona para obtener la lista ordenada completa.

El procedimiento que utiliza el algoritmo en 4 se puede revisar en el apartado Apéndice 6, en el procedimiento **Merge** 8

Algorithm 4. Algoritmo Merge Sort

```
1: procedure MERGESORT(arr, left, right)  ▷ Ordena arr en orden ascendente
2:     if left < right then
3:         middle ← ⌊(left + right)/2⌋
4:         MERGESORT(arr, left, middle)
5:         MERGESORT(arr, middle + 1, right)
6:         MERGE(arr, left, middle, right)
```

El algoritmo divide la lista en mitades cada vez, lo que significa que el número de divisiones necesarias es log n. Luego, cada mitad se ordena en tiempo lineal, lo que da como resultado una complejidad temporal total de $O(nlogn)$ en el mejor y peor caso.

El enfoque divide y conquista de Merge Sort lo hace muy útil para ordenar grandes conjuntos de datos, ya que puede manejar eficientemente listas con cientos de miles o incluso millones de elementos. Además, Merge Sort es un algoritmo estable, lo que significa que preserva el orden relativo de los elementos iguales en la lista original.

C. Algoritmos de Multiplicación de Matrices

Las matrices son una estructura de datos fundamental para representar relaciones lineales en la matemática y la física, la multiplicación de matrices es una operación que se realiza con frecuencia en muchas áreas de la ciencia.

Para multiplicar dos matrices A y B, se multiplican las filas de A por las columnas de B. El resultado de esta operación es una matriz C, donde el elemento en la fila i y la columna j de C es el resultado de la multiplicación del elemento en la fila i de A con el elemento en la columna j de B[10], como se puede observar en la figura 2.

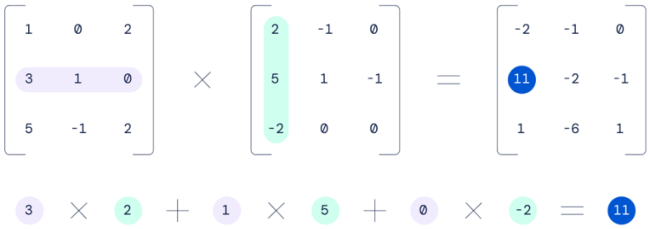

Fig. 2. Multiplicación de matrices

La multiplicación de matrices se puede realizar utilizando varios algoritmos, que varían en términos de complejidad temporal y espacio. Uno de los algoritmos más simples es el algoritmo de multiplicación de matrices iterativo cúbico tradicional, que realiza la multiplicación utilizando tres bucles anidados y tiene una complejidad temporal de $O(n^3)$.

Existen algoritmos más eficientes que pueden reducir la complejidad temporal de la multiplicación de matrices. Un ejemplo de estos es el algoritmo de Strassen, que utiliza una técnica de división y conquista para reducir la complejidad temporal a $O(n^{log_2 7})$. Este algoritmo divide las matrices originales en cuatro submatrices y realiza siete multiplicaciones de matrices más pequeñas para calcular el resultado final[11].

C.1. Algoritmo iterativo cúbico tradicional

El algoritmo de multiplicación de matrices iterativo cúbico tradicional 5 es uno de los algoritmos más simples para realizar la multiplicación de dos matrices. Este algoritmo se basa en la definición de la multiplicación de matrices, que consiste en multiplicar las filas de la primera matriz por las columnas de la segunda matriz y sumar los productos resultantes para obtener el valor de cada elemento de la matriz resultante.

Para implementar el algoritmo se utiliza un triple bucle anidado. El primer bucle recorre las filas de la matriz A, el segundo bucle recorre las columnas de la matriz B, y el tercer bucle calcula el producto de cada elemento de la fila de A por cada elemento de la columna de B y los suma para obtener el valor del elemento correspondiente en la matriz resultante C.

La complejidad temporal es de $O(n^3)$. Esto se debe a que el algoritmo realiza tres bucles anidados, cada uno de los cuales recorre n elementos. Como resultado, la cantidad total de operaciones necesarias para multiplicar dos matrices de tamaño n es proporcional a n^3.

C.2. Algoritmo iterativo cúbico optimizado

La multiplicación de matrices por algoritmo iterativo cúbico optimizado 6 es una variante de la multiplicación de matrices clásico, que consiste en transponer una de las matrices antes de realizar la multiplicación. La transposición de una matriz implica intercambiar sus filas por sus columnas, lo que significa

Algorithm 5. Algoritmo iterativo cúbico tradicional

```
1: function STANDARDMM(A, B)                          ▷ Calcula A x B
2:     n ← rows(A)
3:     m ← columns(A)
4:     k ← columns(B)
5:     Inicializa C como una matriz n × k con todas las entradas seteadas a 0
6:     for i = 0 hasta n − 1 do
7:         for j = 0 hasta k − 1 do
8:             for p = 0 hasta m − 1 do
9:                 C[i][j] ← C[i][j] + A[i][p] × B[p][j]
10:    return C
```

que las filas de la matriz original se convierten en columnas en la matriz transpuesta y viceversa.

Algorithm 6. Multiplicación de matrices por transposición

```
1: function TRANSPOSICIÓNMM(A, B)▷ Calcula A x B usando transposición
2:     n ← filas(A)
3:     m ← columnas(A)
4:     k ← columnas(B)
5:     Inicializar Bt como una matriz k × m con todos los elementos en 0
6:     for i = 0 hasta k − 1 do
7:         for j = 0 hasta m − 1 do
8:             Bt[i][j] ← B[j][i]
9:     Inicializar C como una matriz n × k con todos los elementos en 0
10:    for i = 0 hasta n − 1 do
11:        for j = 0 hasta k − 1 do
12:            productoEscalar ← 0
13:            for p = 0 hasta m − 1 do
14:                productoEscalar ← productoEscalar + A[i][p] × Bt[j][p]
15:            C[i][j] ← productoEscalar
16:    return C
```

El algoritmo de multiplicación de matrices optimizado utiliza el mismo triple bucle anidado que el algoritmo de multiplicación de matrices clásico. La diferencia es que una de las matrices se transpone antes de realizar la multiplicación. Esto significa que, en lugar de recorrer las filas de una matriz y las columnas de otra, se recorren las filas de una matriz y las filas transpuestas de la otra matriz.

La complejidad temporal de la multiplicación de matrices optimizada es la misma que la de la multiplicación de matrices clásica, que es $O(n^3)$. Sin embargo, en algunos casos puede ser más eficiente que la multiplicación de matrices convencional debido a la forma en que se accede a los elementos de las matrices en la memoria.

El algoritmo optimizado se utiliza en aplicaciones donde la memoria caché del procesador es limitada y se requiere un acceso eficiente a los elementos de las matrices. Al transponer una de las matrices, se puede acceder a los elementos de forma más eficiente y reducir la cantidad de veces que se accede a la memoria principal, lo que puede mejorar el rendimiento del algoritmo.

C.3. Algoritmo de Strassen

El algoritmo de Strassen 7 es un algoritmo de multiplicación de matrices que utiliza una técnica de división y conquista para reducir la cantidad de operaciones requeridas para la multiplicación. La idea principal es dividir cada una de las matrices de entrada en cuatro submatrices de igual tamaño, y luego realizar una serie de operaciones entre estas submatrices para obtener la matriz resultante.

La complejidad temporal del algoritmo de Strassen depende del tamaño de las matrices de entrada. En el mejor caso, la complejidad temporal es O(1) para matrices pequeñas. En el peor caso, la complejidad temporal es $O(n^{log_2 7})$, lo que es ligeramente mejor que la complejidad temporal $O(n^3)$ de la multiplicación de matrices iterativo cúbico tradicional.

Algorithm 7. Multiplicación de matrices de Strassen

```
1: function STRASSENMM(A, B)    ▷ Calcula A x B usando el algoritmo de Strassen
2:     if tamaño(A) = 1 then
3:         return A × B                                   ▷ Caso base
4:     Dividir A y B en submatrices de tamaño n/2 × n/2
5:     P1 ← STRASSENMM(A11 + A22, B11 + B22)
6:     P2 ← STRASSENMM(A21 + A22, B11)
7:     P3 ← STRASSENMM(A11, B12 − B22)
8:     P4 ← STRASSENMM(A22, B21 − B11)
9:     P5 ← STRASSENMM(A11 + A12, B22)
10:    P6 ← STRASSENMM(A21 − A11, B11 + B12)
11:    P7 ← STRASSENMM(A12 − A22, B21 + B22)
12:    Calcular las submatrices de C usando P1, P2, ..., P7
13:    C11 ← P1 + P4 − P5 + P7
14:    C12 ← P3 + P5
15:    C21 ← P2 + P4
16:    C22 ← P1 − P2 + P3 + P6
17:    return C
```

3. DESCRIPCIÓN DE LOS DATASETS

A. Sobre la parametrización del experimento

Para el siguiente experimento se generaron 20 conjuntos de números aleatorios (parámetro Total Amount of Files), cada conjunto se generó incrementalmente hasta llegar a 5000 filas *(parámetro Amount of Random Numbers)*, cada fila conteniendo un número aleatorio en un rango de 1 a 100000 (parámetro Distribution Maximum). Además, el experimento genera varias matrices, incluyendo matrices cuadradas y rectangulares (parámetros Matrix square & Matrix rectangle), con una anchura y altura máxima de 100 y 200, respectivamente. El conjunto total de archivos generados también es de 20 (parámetro Matrix Total Set of Files), con una distribución máxima de 10000 (parámetro Matrix Distribution Maximum). Estos parámetros permitirán analizar el rendimiento de algoritmos de ordenamiento y multiplicación de matrices bajo diferentes estructuras de datos, tamaños de archivo y estado del conjunto (conjunto randomizado, conjunto parcialmente ordenado, conjunto semi ordenado y conjunto inversamente ordenado). Los resultados de este experimento nos ayudarán a comprender la escalabilidad y eficiencia de estos algoritmos bajo estas diferentes condiciones.

Conjunto randomizado Este conjunto se encuentra definido como un vector de números aleatorios en un rango desde 1 hasta el parámetro *distribution maximum*.

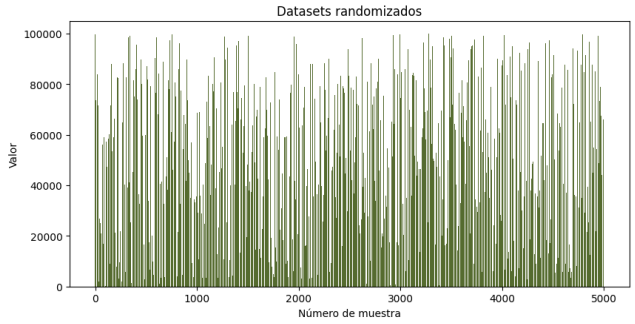

Fig. 3. Datasets con distribución de números randomizados

Conjunto parcialmente ordenado Este conjunto randomizado se encuentra ordenado desde el índice 0 hasta el índice $(largo\_del\_arreglo)/2$.

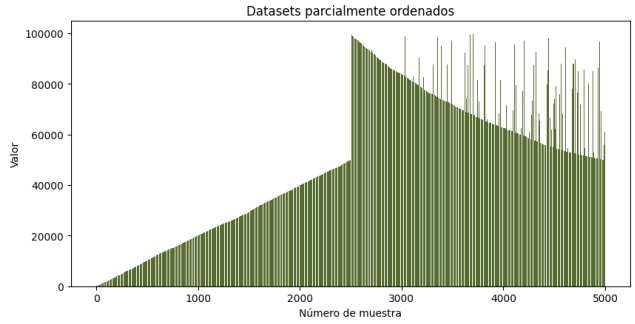

Fig. 4. Datasets con distribución de números parcialmente ordenados

Conjunto semi ordenado Este conjunto se encuentra definido como un conjunto randomizado, el cual se encuentra ordenado bajo un criterio definido, en este caso, se definió ordenar numeros pares, dejando los impares en su posición original. Tanto números pares mantienen su orden relativo entre ellos.

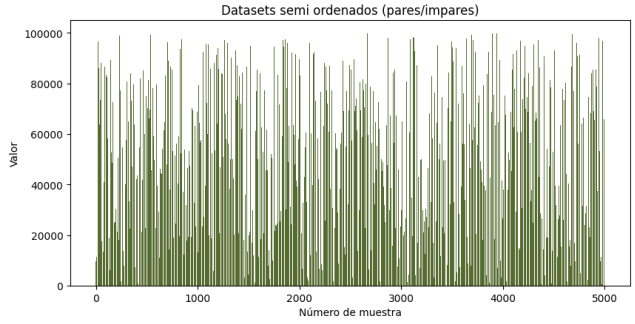

Fig. 5. Datasets con distribución de números semi ordenados (pares/impares)

Conjunto inversamente ordenado Este conjunto se encuentra definido como un conjunto randomizado, el cual se encuentra ordenado de *mayor a menor*.

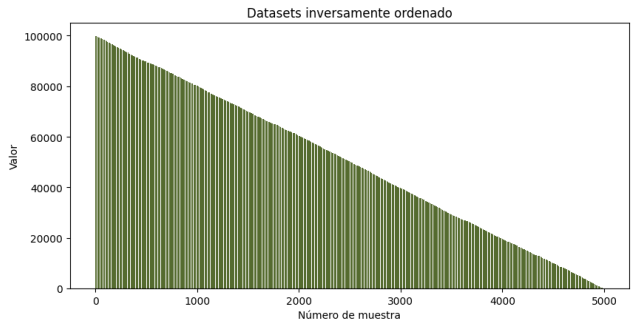

Fig. 6. Datasets con distribución de números inversamente ordenados

Table 1. Shape Functions for Quadratic Line Elements

Parameter	Value
Amount of Random Numbers	5000
Distribution Maximum	100000
Total Amount of Files	20
Matrix Square A Maximum Width	100
Matrix Square A Maximum Height	100
Matrix Square B Maximum Width	100
Matrix Square B Maximum Height	100
Matrix Rectangle A Maximum Width	100
Matrix Rectangle A Maximum Height	200
Matrix Rectangle B Maximum Width	200
Matrix Rectangle B Maximum Height	300
Matrix Total Set of Files	20
Matrix Distribution Maximum	10000

B. Sobre la importancia de datos variados

Es importante utilizar datasets variados en experimentos de complejidad asintótica de algoritmos de ordenamiento y multiplicación de matrices para obtener resultados más precisos y representativos del comportamiento de los algoritmos en diferentes escenarios. Si solo se utiliza un tipo de dataset, por ejemplo, listas ordenadas, es posible que se obtengan resultados que no reflejen el rendimiento real de los algoritmos en otros tipos de datos, como listas desordenadas o con elementos repetidos. Por lo tanto, al utilizar datasets variados, se puede obtener una evaluación más completa y precisa del comportamiento de los algoritmos, lo que puede ser útil en la selección y optimización de algoritmos para aplicaciones específicas.

4. RESULTADOS EXPERIMENTALES

Considerando el mecanismo empleado en la sección 3A sobre la caracterización de los datasets, se presentan los siguientes resultados.

Table 2. Equipo utilizado en el experimento

Hardware	Características
Procesador	11th Gen *R* Core(TM) i71165G7 2.80GHz 1.69 GHz
RAM	32 GB DDR4 SODIMM 2667MHz
Persistencia	NVMe PC SN530 512GB

Primero, se describe el equipo utilizado para el experimento, el cual se puede evaluar en la tabla 2.

A. Resultados experimentales para algoritmos de ordenamiento

La tabla completa de los resultados experimentales para el análisis de tiempo de ejecución de los distintos casos y algoritmos puede ser encontrada en el repositorio público para este proyecto en *statistics/sorting_results.csv* [7] En la siguiente tabla, encontraremos una muestra representativa del experimento para los datasets de 5000 entradas y sus tiempos de ejecución, ordenados por algoritmo evaluado. La representación gráfica de la totalidad de estos datos, incluidos los distintos casos de prueba, se encuentran graficados en las figuras 8, 9, 10 y 11.

Table 3. Resultados de los algoritmos de ordenamiento para los datasets de 5000 entradas

Algoritmo	Dataset	Entradas	Tiempo μs
bubblesort	randomizado	5000	182217
bubblesort	parcial	5000	120856
bubblesort	semi	5000	159738
bubblesort	inverso	5000	202690
insertionsort	randomizado	5000	51869
insertionsort	parcial	5000	16906
insertionsort	semi	5000	43906
insertionsort	inverso	5000	92466
mergesort	randomizado	5000	4477
mergesort	parcial	5000	2403
mergesort	semi	5000	3891
mergesort	inverso	5000	3448
quicksort	randomizado	5000	1247
quicksort	parcial	5000	57629
quicksort	semi	5000	1486
quicksort	inverso	5000	133315
stlsort	randomizado	5000	1333
stlsort	parcial	5000	1062
stlsort	semi	5000	1181
stlsort	inverso	5000	589

A.1. Análisis sobre compejidad temporal de los algoritmos de ordenamiento

En general, se espera que los algoritmos sean más eficientes cuanto menor sea su complejidad temporal. Sin embargo, la complejidad temporal de un algoritmo puede variar en función del caso de entrada, es decir, de las características de los datos que se están ordenando. Podemos ver en la Figura 7 las distintas expresiones de complejidad temporal que pueden tomar los algoritmos.

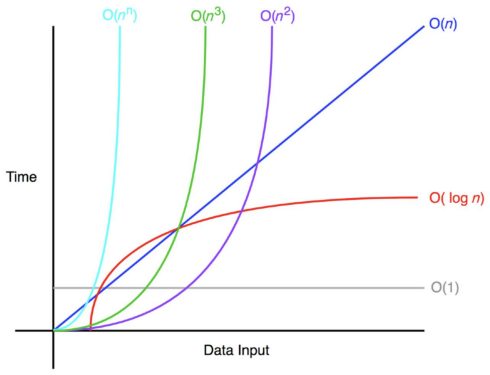

Fig. 7. Distintas complejidades temporales en función del input

Table 4. Rendimiento teórico de los algoritmos de ordenamiento

Algoritmo	Mejor Caso	Peor Caso	Caso Promedio
Bubblesort	$O(n)$	$O(n^2)$	$O(n^2)$
Insertionsort	$O(n)$	$O(n^2)$	$O(n^2)$
Quicksort	$O(n \log n)$	$O(n^2)$	$O(n \log n)$
Mergesort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Sort STL	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$

En este trabajo se han contemplado distintos casos para generar datasets variados, y poder evaluar los resultados experimentales del comportamiento de los distintos algoritmos, en la tabla 3 podemos ver una muestra del tiempo de ejecución cuando los input ya son lo suficientemente grandes. En los siguientes gráficos, revisaremos el comportamiento desde 1 hasta 5000 para los distintos algoritmos.

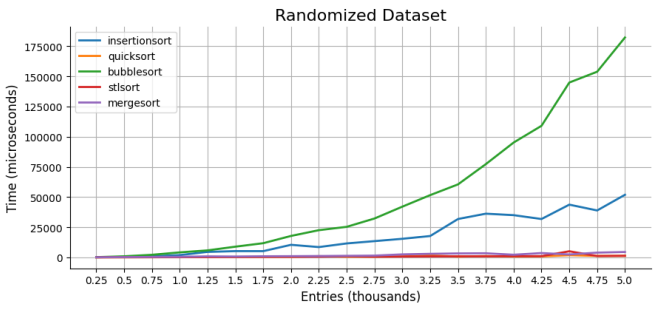

Fig. 8. Complejidad temporal para Dataset randomizado

Bubblesort e insertionsort son algoritmos de ordenamiento de complejidad temporal $O(n^2)$, lo que significa que su rendimiento empeora rápidamente a medida que el tamaño de la lista aumenta. Estos algoritmos pueden ser útiles para ordenar listas pequeñas, pero pueden ser muy ineficientes para listas grandes.

En la figura 8 se visualizan los **resultados experimentales** para los algoritmos estudiados en el contexto de una lista numérica *completamente randomizada*. Al compararlos con la tabla 4 podemos identificar inmediatamente las **tendencias cuadráticas** de bubblesort, y **logarítmicas** de mergesort y STL sort.

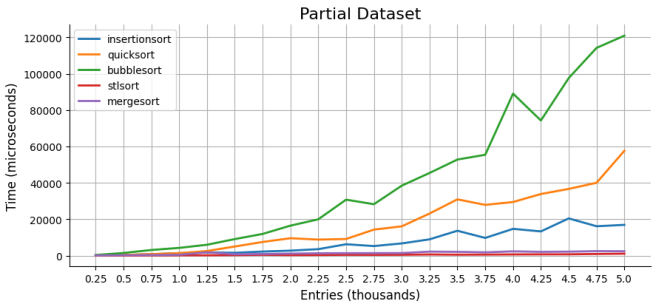

Fig. 9. Complejidad temporal para Dataset parcialmente ordenado

Mergesort y quicksort son algoritmos de ordenamiento de complejidad temporal $O(nlogn)$ en el caso promedio, lo que significa que tienen un rendimiento mucho mejor que los algoritmos $O(n^2)$, a menos que se encuentren en el peor caso (para quicksort). Mergesort es un algoritmo estable que utiliza un enfoque de dividir y conquistar para ordenar una lista. Quicksort es un algoritmo no estable que también utiliza un enfoque de dividir y conquistar.

En la figura 9 Podemos ver un caso no extremo, donde medio dataset se encuentra ordenado y el resto aleatorio, los resultados experimentales se acercan favorablemente a los valores teorizados para bubblesort, quicksort y mergesort.

Sin embargo, para el caso de *insertionsort* la gráfica se asemeja más a una función $nlog(n)$. La discusión para este punto se fundamenta en que diferencia de otros algoritmos de ordenamiento, como Bubblesort y Quicksort, el tiempo de ejecución del Insertionsort no depende solo del tamaño del arreglo a ordenar, sino también de la distribución de los elementos en el arreglo. Por lo tanto, es posible que en algunos casos específicos, el Insertionsort tenga un comportamiento más logarítmico que cuadrático en el caso promedio.

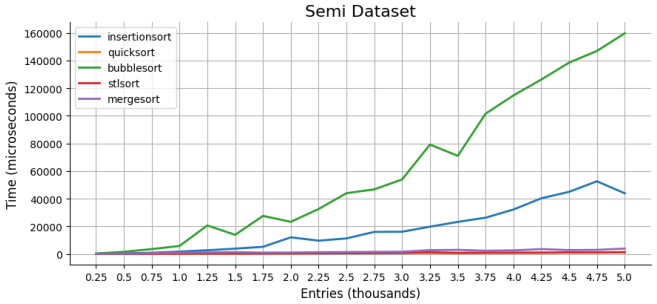

Fig. 10. Complejidad temporal para Dataset semi ordenado

Un ejemplo de esto podría ser cuando el arreglo a ordenar ya está casi ordenado. En este caso, el Insertionsort tendría que realizar muy pocas comparaciones y movimientos de elementos, lo que podría hacer que su tiempo de ejecución se aproxime a $O(nlogn)$ en lugar de $O(n^2)$.

Sort de STL, el algoritmo de ordenamiento proporcionado por la biblioteca estándar de C++ es un algoritmo híbrido que utiliza una variante del algoritmo introsort, que combina el Quicksort con el Heapsort y el Insertionsort. dependiendo de las características de la lista que se está ordenando. El rendimiento de sort suele ser muy bueno en la mayoría de los casos de entrada, como se puede visualizar en la figura 10, y para todas en general, el rendimiento de STL sort es muy bueno, siendo el menor para todos los casos.

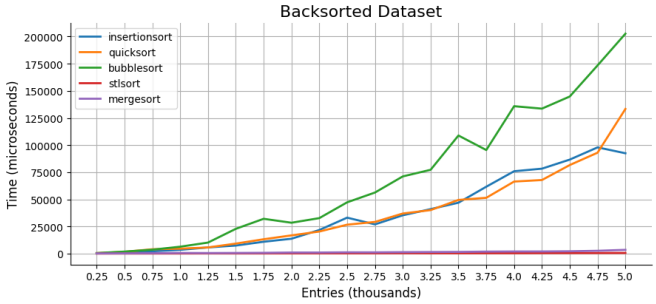

Fig. 11. Complejidad temporal para Dataset inversamente ordenado

Finalmente, en la figura 11 uno de los peores escenarios para este trabajo, experimentalmente obtenemos resultados similares a los teorizados en la tabla 4. Podemos entonces afirmar que la elección del algoritmo de ordenamiento adecuado dependerá de las características de la lista que se está ordenando y de los requisitos de rendimiento específicos del problema que se está resolviendo.

B. Resultados experimentales para algoritmos de multiplicación de matrices

De la misma manera que en A, el repositorio público para este proyecto en *statistics/matrix_square_results.csv* y *statistics/matrix_rectangle_results.csv* [7] contiene la tabla en su totalidad. En la siguiente, encontraremos una muestra representativa del experimento para los datasets de 100 y 200 filas y sus tiempos de ejecución, ordenados por algoritmo evaluado, respectivamente 5, 6. La representación gráfica de la totalidad de estos datos, incluidos los distintos casos de prueba, se encuentran graficados en las figuras 12 y 13.

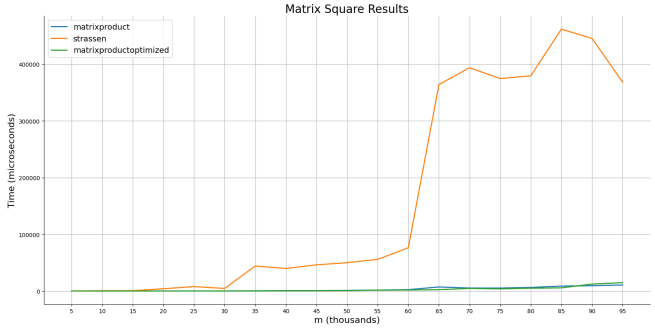

Fig. 12. Complejidad temporal para matrices cuadradas

El algoritmo clásico de multiplicación de matrices (método iterativo cúbico tradicional) que utiliza tres bucles anidados para

calcular el producto de dos matrices. La complejidad temporal de este algoritmo es de $O(n^3)$, lo que lo hace ineficiente para matrices grandes.

Table 5. Tiempos de ejecución de los algoritmos de multiplicación de matrices para diferentes tamaños de matrices cuadradas

Algoritmo	Tamaño (m)	Tiempo μs
Classic	5	2
Optimizado	5	2
Strassen	5	82
Classic	20	62
Optimizado	20	47
Strassen	20	4041
Classic	45	729
Optimizado	45	536
Strassen	45	46226
Classic	70	5289
Optimizado	70	4643
Strassen	70	393727
Classic	95	10682
Optimizado	95	14751
Strassen	95	368179

Podemos ver en la figura 12 que, por el contrario de lo teorizado, los algoritmos clásicos están muy por debajo de **Strassen** en cuanto a complejidad temporal en este experimento, una justificación de esto puede ser que incluso matrices de $100x100$ son demasiado pequeñas para marcar una diferencia significativa al momento de comparar *Strassen con el algoritmo clásico*.

El algoritmo clásico mejorado de multiplicación de matrices utiliza una técnica de reordenamiento de bucles que reduce la cantidad de operaciones necesarias para calcular el producto de dos matrices. La complejidad temporal de este algoritmo es de $O(n^3/logn)$, lo que lo hace más eficiente que el algoritmo clásico para matrices grandes, pudiendo evidenciar esto en la figura 13 de manera muy tenue.

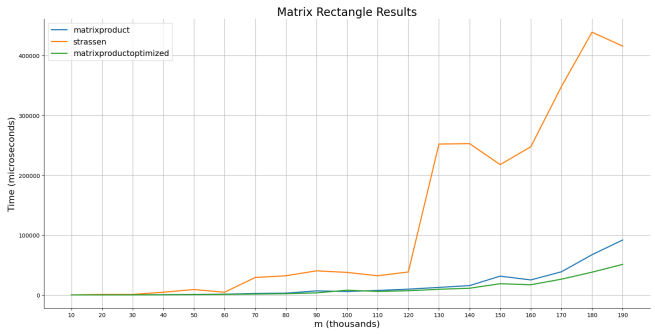

Fig. 13. Complejidad temporal para matrices rectangulares

El algoritmo de Strassen divide las matrices en submatrices más pequeñas y utiliza fórmulas matemáticas para calcular el producto de las submatrices. La complejidad temporal de este algoritmo es de $O(n^{log_27})$, lo que lo hace más eficiente que el algoritmo clásico y el algoritmo clásico mejorado para matrices grandes, **sin embargo esto no ha podido ser demostrado experimentalmente en este trabajo**, se considera que es producto de que el tamaño de las matrices es significativamente más pequeña que el teorizado para que Strassen tenga un margen de mejora versus los otros algoritmos.

Table 6. Tiempos de ejecución de los algoritmos de multiplicación de matrices para diferentes tamaños de matrices rectangulares

Algoritmo	m	Tiempo μs
Classic	10	7
Optimizado	10	9
Strassen	10	80
Classic	40	393
Optimizado	40	272
Strassen	40	4563
Classic	90	6787
Optimizado	90	3440
Strassen	90	40264
Classic	140	15598
Optimizado	140	11173
Strassen	140	252750
Classic	190	91783
Optimizado	190	51041
Strassen	190	415723

5. CONCLUSIONES

La evaluación exhaustiva de algoritmos de ordenamiento y multiplicación de matrices representa un tema intrincado y de suma importancia en el ámbito de las ciencias de la computación. Con frecuencia, se observa una discrepancia entre los resultados teóricos y experimentales de dichos algoritmos. Comprender las causas subyacentes de estas divergencias es esencial para implementar mejoras considerables en la eficacia y exactitud de los algoritmos.

En relación con los algoritmos de ordenamiento, se han ideado diversas metodologías para clasificar conjuntos de datos, cada una con su respectiva complejidad teórica y experimental. Algunos de los algoritmos más destacados incluyen el bubblesort, insertionsort, quicksort, mergesort y el algoritmo de ordenamiento integrado de la Standard Template Library (STL) en C++.

La complejidad teórica de un algoritmo se describe como la cantidad de tiempo y memoria necesaria para llevar a cabo una tarea en función del tamaño de la entrada. Algoritmos de ordenamiento más elementales, como el bubblesort y el insertionsort, exhiben una complejidad de $O(n^2)$, lo cual indica que el tiempo de ejecución incrementa cuadráticamente en relación al tamaño

de la entrada. En contraste, algoritmos más sofisticados, como el quicksort y el mergesort, poseen una complejidad de $O(nlogn)$, lo que sugiere que su tiempo de ejecución se incrementa logarítmicamente con respecto al tamaño de la entrada.

No obstante, en situaciones prácticas, los resultados experimentales frecuentemente no reflejan la complejidad teórica del algoritmo. Esto puede atribuirse a diversos factores, tales como la implementación del algoritmo, el hardware empleado en las pruebas y la complejidad intrínseca del conjunto de datos de entrada. Por ejemplo, el bubblesort y el insertionsort pueden desempeñarse mejor en conjuntos de datos pequeños y casi ordenados, mientras que el quicksort y el mergesort podrían ser más eficientes en conjuntos de datos más extensos y desorganizados. Además, la aplicación de técnicas de optimización, como el paralelismo y la distribución de carga, puede potenciar significativamente el rendimiento de los algoritmos.

En el ámbito de los algoritmos de multiplicación de matrices, se utilizan comúnmente los enfoques clásico, optimizado y de Strassen. El algoritmo clásico presenta una complejidad de $O(n^3)$, lo cual implica que el tiempo de ejecución aumenta cúbicamente en función del tamaño de la entrada. El algoritmo optimizado, por su parte, emplea técnicas de optimización para disminuir la cantidad de operaciones necesarias, resultando en una complejidad teórica inferior a la del algoritmo clásico. El algoritmo de Strassen, en cambio, se basa en una estrategia de "divide y vencerás" con el objetivo de reducir la complejidad teórica a $O(nlogn)$.

A pesar de ello, los resultados experimentales de estos algoritmos pueden variar de manera notable respecto a su complejidad teórica. En particular, los algoritmos optimizados y de Strassen podrían ser menos eficientes en la práctica debido a la sobrecarga de memoria y a la necesidad de efectuar un mayor número de operaciones en coma flotante, las cuales suelen ser costosas en términos de tiempo.

Para optimizar los resultados experimentales de los algoritmos de ordenamiento y multiplicación de matrices, es fundamental considerar varios aspectos. En primer lugar, resulta esencial emplear hardware optimizado para la tarea en cuestión, como procesadores de alta velocidad y unidades de procesamiento gráfico (GPU) especializadas, que pueden acelerar significativamente el rendimiento computacional.

Además, se deben explorar y adaptar enfoques de optimización, como la vectorización y el uso de bibliotecas de alto rendimiento, para maximizar la eficiencia de los algoritmos en escenarios específicos. También es crucial evaluar el impacto del hardware, el sistema operativo y los compiladores en el rendimiento de los algoritmos, ya que estos factores pueden generar diferencias notables en los resultados experimentales.

Por último, es necesario desarrollar una comprensión profunda de las características de los conjuntos de datos de entrada y adaptar los algoritmos de manera adecuada para garantizar un rendimiento óptimo en función de estas características. Al considerar cuidadosamente todos estos factores, se pueden lograr mejoras significativas en la eficiencia y precisión de los algoritmos de ordenamiento y multiplicación de matrices, lo cual resulta vital para impulsar avances en el campo de las ciencias de la computación y en diversas aplicaciones prácticas.

6. APÉNDICE

A. Procedimientos para algoritmos

Algorithm 8. Procedimiento Merge del algoritmo Merge Sort

```
1:  procedure MERGE(arr, left, middle, right)
2:      Crear los arreglos L[0...middle − left + 1] y R[0...right − middle]
3:      for i = 0 hasta middle − left do
4:          L[i] ← arr[left + i]
5:      for j = 0 hasta right − middle − 1 do
6:          R[j] ← arr[middle + 1 + j]
7:      i ← 0
8:      j ← 0
9:      k ← left
10:     while i ≤ middle − left y j ≤ right − middle − 1 do
11:         if L[i] ≤ R[j] then
12:             arr[k] ← L[i]
13:             i ← i + 1
14:         else
15:             arr[k] ← R[j]
16:             j ← j + 1
17:         k ← k + 1
18:     while i ≤ middle − left do
19:         arr[k] ← L[i]
20:         i ← i + 1
21:         k ← k + 1
22:     while j ≤ right − middle − 1 do
23:         arr[k] ← R[j]
24:         j ← j + 1
25:         k ← k + 1
```

REFERENCES

1. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (3rd ed.). Cambridge, MA: The MIT Press.
2. Akhter, N., Idrees, M., & Rehman, F. (2016). Sorting Algorithms - A Comparative Study. *International Journal of Computer Science and Information Security (IJCSIS)*, 14(12), 937-943.
3. Mishra, Aditya Dev. *Selection of Best Sorting Algorithm for a Particular Problem*. Tesis de Maestría, Departamento de Ciencias de la Computación e Ingeniería, Universidad Thapar, 2009.
4. Gill, Sandeep Kaur. *A Comparative Study of Various Sorting Algorithms*. En *Special Issue based on proceedings of 4th International Conference on Cyber Security (ICCS) 2018*.
5. GeeksforGeeks. (26 de Abril 2023 00:09 hrs). *Sort in C++ Standard Template Library (STL)*. Recuperado de https://www.geeksforgeeks.org/sort-c-stl/.
6. Astrachan, Owen. *Bubble Sort: An Archaeological Algorithmic Analysis*. Computer Science Department, Duke University, 2003.
7. Villarroel, Iván. *Informe 1: Proyecto en C++*. Github, 2023. https://github.com/ivillarroelr/FEDA/blob/main/informe1.
8. Sodhi, Tarundeep Singh. *Enhanced Insertion Sort Algorithm*. International Journal of Computer Applications, vol. 64, no. 21, Feb. 2013, pp. 35.
9. cplusplus.com. (2022). std::sort. Recuperado el 1 de mayo de 2022, de https://cplusplus.com/reference/algorithm/sort/.
10. Press, W. H., Teukolsky, S. A., Vetterling, W. T., & Flannery, B. P. (2007). *Numerical Recipes: The Art of Scientific Computing*. Cambridge University Press.
11. Fawzi, A., Balog, M., & Huang, A. (2019). Discovering faster matrix multiplication algorithms with reinforcement learning.