Tarea 1 - Algoritmos y Complejidad: Algoritmos Dividir y Conquistar

Germán Fernández

2024 - 2

1 Repositorio

En el siguiente repositorio de github se encuentran todos los archivos, codigos, casos de prueba y su código que los genera, los archivos de jupyter utilizados para generar los gráficos, los archivos de excel en el cual se guardaron los resultados:
Repositorio GitHub: Tarea 1 - Algoritmos y Complejidad

2 Introducción

En este informe realizaré un análisis después de probar varios algoritmos de ordenamiento y multiplicación de matrices. Cada uno de los anteriores tiene una complejidad teórica propia. Observaremos los tiempos de ejecución que tarda cada uno, aplicándolos a datasets de distinto tipo y tamaño con la misión de observar si su tendencia es la complejidad esperada.

3 Algoritmos de ordenamiento

Los algoritmos de ordenamiento son aquellos que sirven para ordenar una serie de datos como uno desee. En este informe se estudiarán 3 algoritmos de ordenamiento, explicados a continuación:

3.1 Bubble Sort

Bubble Sort es un algoritmo de ordenamiento que recorre la lista de manera iterativa utilizando dos bucles anidados. El bucle externo controla el número de pasadas necesarias para ordenar la lista, iterando hasta que todos los elementos estén en su posición correcta. El bucle interno compara cada par de elementos adyacentes. Si el elemento actual es mayor que el siguiente (en caso de ordenar ascendentemente), se intercambian sus posiciones. A medida que avanza el bucle externo, el rango del bucle interno disminuye, ya que los elementos más grandes se van colocando al final de la lista.

La versión optimizada de Bubble Sort introduce una bandera (‘swapped‘) para verificar si se realizaron intercambios en una pasada completa. Si no se realizaron intercambios, el algoritmo concluye que la lista ya está ordenada y termina prematuramente, lo que reduce la complejidad a $O(n)$ en el mejor de los casos (cuando la lista ya está ordenada). Sin embargo, en el peor de los casos (cuando la lista está completamente desordenada), la complejidad sigue siendo $O(n^2)$ debido a los bucles anidados.

```
void bubbleSort(vector<int>& arr) {
    int n = arr.size();
    int i, j;               // Variables i and j are used for
        iteration
    bool swapped;            // A flag to check if any swapping
        happened

    // Outer loop to iterate over the entire array n-1 times
    for (i = 0; i < n - 1; i++) {
        swapped = false;     // Reset the swapped flag to false for
             each pass

        // Inner loop to perform the comparison and swap adjacent
            elements
        // As the largest elements "bubble up" to the end, we
            reduce the range by 'i'
        for (j = 0; j < n - i - 1; j++) {
            // Compare adjacent elements
            if (arr[j] > arr[j + 1]) {
                // Swap them if they are in the wrong order
                swap(arr[j], arr[j + 1]);
                swapped = true; // Set swapped to true if any
                    swapping occurred
            }
        }

        // If no two elements were swapped in the current pass,
        // the array is already sorted, and we can break early
        if (swapped == false)
            break; // Exit the loop early since the array is sorted
    }
}
```

3.2 Merge Sort

Este algoritmo comienza con una lista inicial de elementos, la cual se divide recursivamente en sub-arreglos cada vez más pequeños. El proceso continúa hasta que cada sub-arreglo tenga un solo elemento. Luego, se procede a unir estos sub-arreglos pequeños, combinándolos de forma ordenada, hasta formar un solo arreglo que contiene los elementos de la lista original en orden ascendente. La complejidad de Merge Sort es $O(n \log n)$, ya que el arreglo se divide recursivamente ($\log n$) y se requiere $O(n)$ para combinar las partes. Sin embargo, este algoritmo tiene un gasto adicional de memoria de $O(n)$ porque necesita espacio extra para almacenar los sub-arreglos durante el proceso de fusión.

```
// Funcion merge que une dos subarreglos ordenados (left y right)
    en el arreglo original arr
void merge(vector<int>& left, vector<int>& right, vector<int>& arr)
     {
    int size_izq = left.size();  // Tamano del subarreglo izquierdo
    int size_der = right.size(); // Tamano del subarreglo derecho

    // Indices para recorrer left, right y arr
    int i = 0, izq = 0, der = 0;

    // Mientras queden elementos en ambos subarreglos
    while (izq < size_izq && der < size_der) {
        // Si el elemento de left es menor que el de right, se
            coloca en arr
        if (left[izq] < right[der]) {
            arr[i] = left[izq];
            izq++;  // Avanzamos en el subarreglo izquierdo
        } else {
            arr[i] = right[der];
            der++;  // Avanzamos en el subarreglo derecho
        }
        i++;  // Avanzamos en el arreglo combinado
    }

    // Si quedan elementos en el subarreglo izquierdo, se anaden a
        arr
    while (izq < size_izq) {
        arr[i] = left[izq];
        izq++;
        i++;
    }

    // Si quedan elementos en el subarreglo derecho, se anaden a
        arr
    while (der < size_der) {
        arr[i] = right[der];
        der++;
        i++;
    }
}

// Funcion mergesort que aplica el algoritmo de ordenamiento por
    mezcla
void mergesort(vector<int>& arr) {
    int size = arr.size();  // Tamano del arreglo

    // Caso base: si el tamano es 1 o menor, ya esta ordenado
    if (size <= 1) return;

    int medio = size / 2;  // Dividimos el arreglo en dos mitades

    // Creamos los subarreglos para la izquierda y derecha
    vector<int> left(medio);  // Subarreglo izquierdo de tamano '
        medio'
    vector<int> right(size - medio);  // Subarreglo derecho con el
        resto
```

```
    // Llenamos los subarreglos con las correspondientes partes de
        arr
    for (int i = 0; i < medio; i++) left[i] = arr[i];  // Parte
        izquierda
    for (int i = medio; i < size; i++) right[i - medio] = arr[i];
        // Parte derecha

    // Recursivamente aplicamos mergesort en ambos subarreglos
    mergesort(left);
    mergesort(right);

    // Combinamos ambos subarreglos ordenados en arr usando la
        funcion merge
    merge(left, right, arr);
}
```

3.3 Quick Sort

Este algoritmo comienza eligiendo un pivote dentro de una lista de elementos; en este caso, el pivote se selecciona al inicio de la lista. Luego, se divide la lista en dos subarreglos: uno que contiene los elementos menores al pivote y otro con los elementos mayores. Este proceso se repite recursivamente para cada subarreglo, hasta que cada uno tenga un solo elemento, lo que garantiza que la lista completa quede ordenada. La complejidad de Quick Sort es $O(n \log n)$ en el mejor de los casos. Esto se debe a que la lista se divide recursivamente en dos mitades logarítmicas ($\log n$), y en cada nivel de la recursión se realiza una comparación lineal entre los elementos ($O(n)$). Sin embargo, en el peor de los casos, la complejidad puede llegar a $O(n^2)$ si el pivote se elige de manera desfavorable, como en este caso donde se selecciona el primer elemento. Si la lista ya está ordenada, se generarán n divisiones, lo que provoca que Quick Sort degrade su rendimiento a $O(n^2)$.

```
// Funcion de particion para QuickSort
int particion(vector<int>& arr, int inicio, int fin) {
    int pivote = arr[fin];  // Elige el ultimo elemento como pivote
    int i = inicio - 1;  // Indice del elemento mas pequeno (los
        que son menores que el pivote)

    // Recorremos desde el inicio hasta justo antes del pivote
    for (int j = inicio; j < fin; j++) {
        // Si encontramos un elemento menor que el pivote
        if (arr[j] < pivote) {
            i++;  // Incrementamos el indice de los menores que el
                pivote
            // Intercambiamos el elemento actual con el que esta en
                 la posicion 'i'
            swap(arr[i], arr[j]);
        }
    }
    // Finalmente, ponemos el pivote en su lugar correcto
    swap(arr[i + 1], arr[fin]);
    return i + 1;  // Retorna el indice del pivote
```

```
}

// Funcion QuickSort
void quickSort(vector<int>& arr, int inicio, int fin) {
    // Caso base: si inicio es menor que fin, seguimos con la
        particion
    if (inicio < fin) {
        // Obtenemos la posicion del pivote usando la funcion
            particion
        int pivote = particion(arr, inicio, fin);
        // Aplicamos quicksort recursivamente en el lado izquierdo
            del pivote
        quickSort(arr, inicio, pivote - 1);
        // Aplicamos quicksort recursivamente en el lado derecho
            del pivote
        quickSort(arr, pivote + 1, fin);
    }
}
```

3.4 Sort() de C++

Es un algoritmo de ordenamiento híbrido que aplica tres algoritmos distintos: Quick Sort, Insertion Sort y Heap Sort. La técnica que utiliza se llama IntroSort, que comienza con QuickSort. La profundidad de recursión se refiere al número de veces que una función se llama a sí misma en el proceso de particionamiento. Si esta profundidad excede un límite específico ($2 \log n$), el algoritmo cambia a HeapSort, que es un algoritmo basado en un árbol binario que garantiza un ordenamiento en $O(n \log n)$ utilizando la estructura de un heap. Si el tamaño de la partición es pequeño (menor a 16), se usa Insertion Sort, que ordena eficientemente listas pequeñas comparando e insertando elementos en su posición correcta de manera secuencial. De esta forma, `sort()` de C++ combina lo mejor de estos tres algoritmos, logrando un rendimiento promedio óptimo con una complejidad de $O(n \log n)$.

3.5 Datasets a ordenar

Para demostrar la eficiencia de cada algoritmo, se utilizaron cincuenta datasets con tamaños que van desde cero hasta un billón en algoritmos con una complejidad teórica de $O(n \log n)$, con el objetivo de reflejar un comportamiento uniforme a lo largo de una gran cantidad de datos. En cambio, en algoritmos con complejidad de $O(n^2)$, se aplicaron datasets que van desde cero hasta cien mil elementos. Además de la variedad en tamaño, se consideraron tres tipos distintos de datos:

- **Ordenados**: Listas con números enteros ordenados de forma ascendente.
- **Random**: Listas en las que cada elemento es un número generado al azar.
- **Semi-ordenados**: Listas ordenadas de forma ascendente, pero aproximadamente la mitad de sus elementos son números al azar.

Para generar los datasets, se utilizaron archivos `.txt`, de modo que cada programa, que contiene un algoritmo de ordenamiento, los lea uno a uno, almacenando los elementos en una lista para luego aplicar el algoritmo de ordenamiento. Este proceso se repite de forma consecutiva hasta que no queden más archivos por leer. Por cada dataset, después de aplicar el algoritmo, se recibe como salida en la terminal el tiempo de ejecución. Una vez finalizado el ordenamiento de todos los datasets, los tiempos de ejecución se copian en una columna de Excel.

3.6 Resultados

Para reflejar la complejidad algoritmica utilicé Pandas, el cual me permitió generar gráficos y estudiar las tendencias asociadas a cada algoritmo en los distintos datasets que fueron generados.

3.6.1 Datasets Random

A continuación presentaré una tabla con los resultados de cada sort al ser aplicados a datasets randoms de distinto tamaño:

RANDOM	MergeSort (ms)	QuickSort (ms)	StandardSort(ms)
0	0	0	0
2000000	889	387	471
4000000	1822	914	977
6000000	2792	1265	1506
8000000	3814	1723	2258
10000000	4757	2154	2587
12000000	5710	2582	3130
14000000	6784	3035	3725
16000000	7731	3828	4504
18000000	8740	4206	5129
20000000	9695	4757	5677
22000000	10789	5256	6425
24000000	11718	5625	6808
26000000	12784	6051	7625
28000000	13791	6842	7991
30000000	14737	7461	8386
32000000	15850	7761	9061
34000000	17001	8046	9712
36000000	18404	9131	10367
38000000	19425	9060	10982
40000000	20239	10345	11797
42000000	21308	10222	12071
44000000	22398	10738	13030
46000000	23085	11171	14018
48000000	24134	11759	13987
50000000	25145	12665	15166
52000000	26124	12480	15262
54000000	27496	13429	15692
56000000	28074	13812	15752
58000000	28984	14313	16233
60000000	30087	15279	16777
62000000	31062	14747	17944
64000000	31949	16489	18247
66000000	34552	16174	18687
68000000	35580	17758	19210
70000000	36554	17004	19880
72000000	36947	18269	20434
74000000	38724	19365	21106
76000000	39921	20000	21723
78000000	41390	20014	22232
80000000	41684	19864	23137
82000000	42937	20659	23592
84000000	43509	21614	24201
86000000	45589	21078	24780
88000000	45679	22725	25135
90000000	47095	23855	26148
92000000	48383	23283	26568
94000000	48972	23201	27124
96000000	50072	24764	28324
98000000	50802	25712	30627
100000000	50284	24295	30586

Table 1: Comparación de tiempos de ejecución con datasets aleatorios.

Como Bubble Sort tarda demasiado en ejecutar instrucciones desde 0 a 100000000, se utilizó un dataset más pequeño de 0 a 100000. En la siguiente pagina:

RANDOM	BubbleSort (ms)
0	0
2000	11
4000	48
6000	108
8000	209
10000	283
12000	429
14000	666
16000	767
18000	969
20000	1199
22000	1553
24000	1777
26000	2049
28000	2446
30000	2832
32000	3250
34000	3768
36000	3992
38000	4749
40000	5147
42000	5941
44000	6749
46000	7660
48000	8003
50000	8508
52000	9399
54000	9751
56000	10976
58000	11802
60000	13322
62000	13803
64000	14305
66000	15737
68000	16433
70000	17494
72000	19479
74000	25430
76000	27413
78000	28918
80000	31198
82000	32569
84000	34238
86000	35984
88000	37516
90000	40146
92000	41385
94000	42618
96000	43282
98000	44802
100000	46944

Table 2: Tiempos de ejecución del algoritmo BubbleSort en milisegundos para diferentes tamaños de datasets.

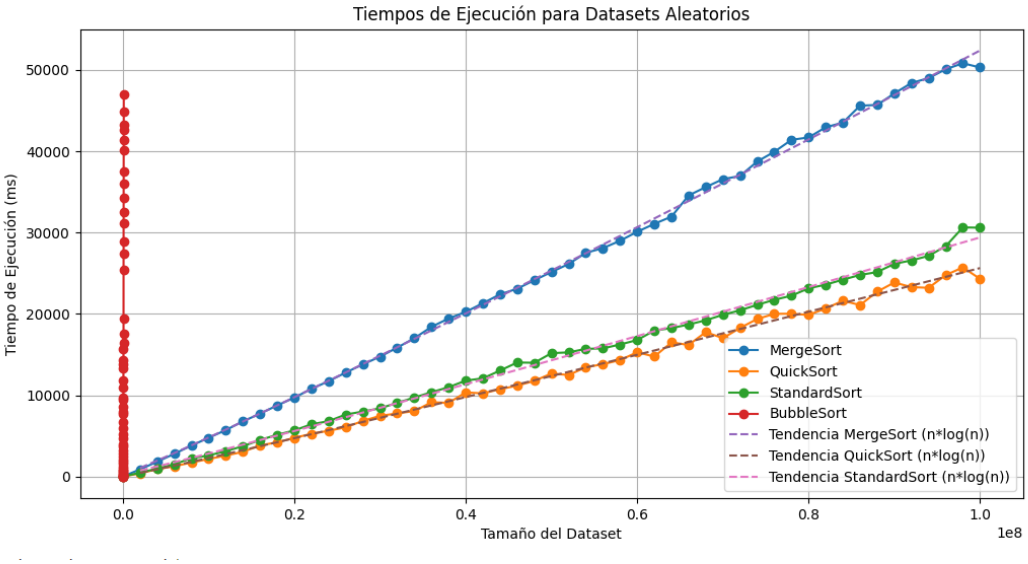

Figure 1: Algoritmos en datasets random

En las tablas presentadas, podemos observar que Bubble Sort, al ordenar cien mil datos aleatorios, toma aproximadamente el mismo tiempo que los demás algoritmos tardan en ordenar cien millones de datos. Esta diferencia significativa radica en la eficiencia de los algoritmos: mientras que MergeSort, QuickSort y la función sort() de C++ presentan una complejidad de $O(n \log n)$, lo que les permite manejar grandes volúmenes de datos de manera eficiente, Bubble Sort tiene una complejidad de $O(n^2)$, lo que lo hace ineficiente para ordenar listas grandes.

Para complementar estas tablas, a continuación, se presentarán gráficos que reflejan los tiempos de ejecución observados. Estos gráficos permitirán visualizar claramente cómo las complejidades teóricas se manifiestan de manera práctica, demostrando las diferencias de eficiencia entre los algoritmos.

Este gráfico nos permite visualizar claramente cómo MergeSort, QuickSort y StandardSort siguen de manera consistente su complejidad teórica $O(n \log n)$. Las líneas de tendencia mostradas para cada uno de estos algoritmos coinciden perfectamente con los puntos de los datos experimentales, lo que indica que, a medida que aumenta el tamaño del dataset, el tiempo de ejecución crece de manera proporcional a $n \log n$ de forma consistente a la teoria.

Por otro lado, podemos observar que BubbleSort muestra un crecimiento mucho más abrupto. Desde los primeros tamaños de datasets, los tiempos de ejecución se disparan, confirmando su complejidad $O(n^2)$. A continuación se mostrará un gráfico de cómo se comporta el BubbleSort y su comportamiento de $O(n^2)$ en la práctica.

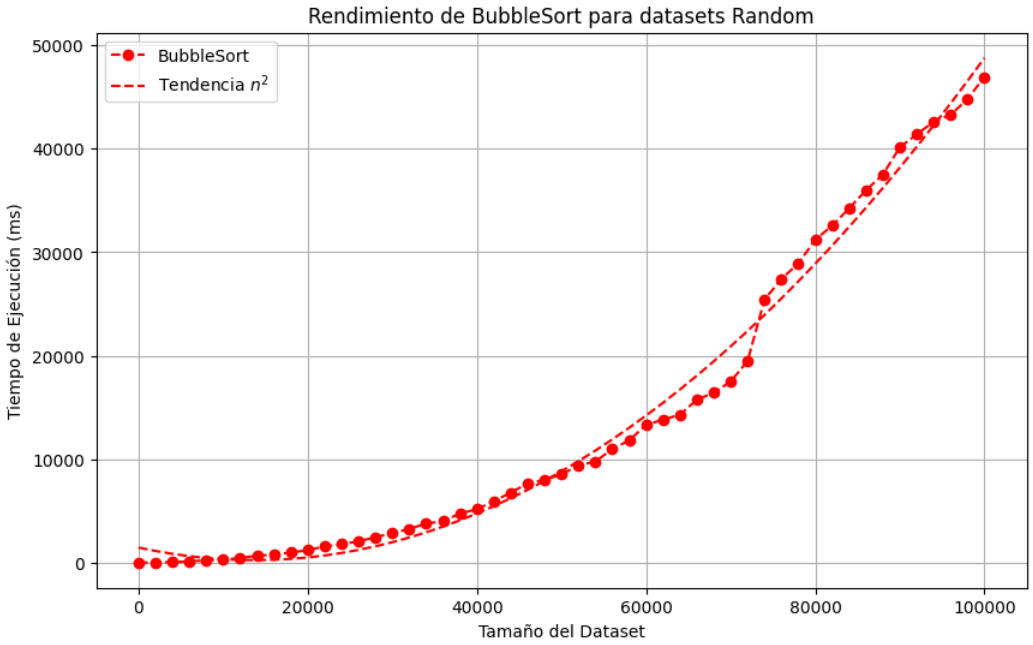

Figure 2: BubbleSort en dataset random

3.6.2 Datasets Ordenados

A continuación, se presentarán las tablas para los algoritmos que soportan bien datasets de tamaño desde 0 a cien millones, es decir, aquellos que cuestan $O(n \log n)$ se notarán con mayor claridad.

ORDENADOS	MergeSort (ms)	StandardSort (ms)	BubbleSort (ms)
0	0	0	0
2000000	630	266	5
4000000	1279	570	11
6000000	1969	988	16
8000000	2674	1181	22
10000000	3378	1670	27
12000000	3983	1872	34
14000000	4800	2356	39
16000000	5367	2489	42
18000000	6076	2763	51
20000000	6827	3498	55
22000000	7463	3586	58
24000000	8676	3846	65
26000000	9243	4107	70
28000000	9861	4651	87
30000000	10377	4767	90
32000000	11231	5814	96
34000000	12727	5946	107
36000000	13581	6039	106
38000000	13862	6513	113
40000000	14579	6858	118
42000000	15065	7480	132
44000000	15438	7726	125
46000000	16371	7805	190
48000000	17633	8080	139
50000000	18157	8613	145
52000000	18928	9262	151
54000000	19882	9504	163
56000000	19866	10092	159
58000000	20734	10124	238
60000000	20958	10033	168
62000000	22075	10303	175
64000000	22073	10844	180
66000000	22809	10918	185
68000000	24085	11585	190
70000000	24429	11141	199
72000000	25589	11387	237
74000000	26387	11730	206
76000000	26815	12665	212
78000000	27400	13206	216
80000000	28249	14198	221
82000000	29118	14638	230
84000000	30297	14701	237
86000000	30602	14735	252
88000000	31462	15255	242
90000000	31489	15697	249
92000000	32144	16168	253
94000000	32708	16779	258
96000000	33690	16716	260
98000000	34247	17341	-
100000000	34806	17150	-

Table 3: Tiempos de ejecución de BubbleSort, MergeSort y StandardSort con datasets ordenados.

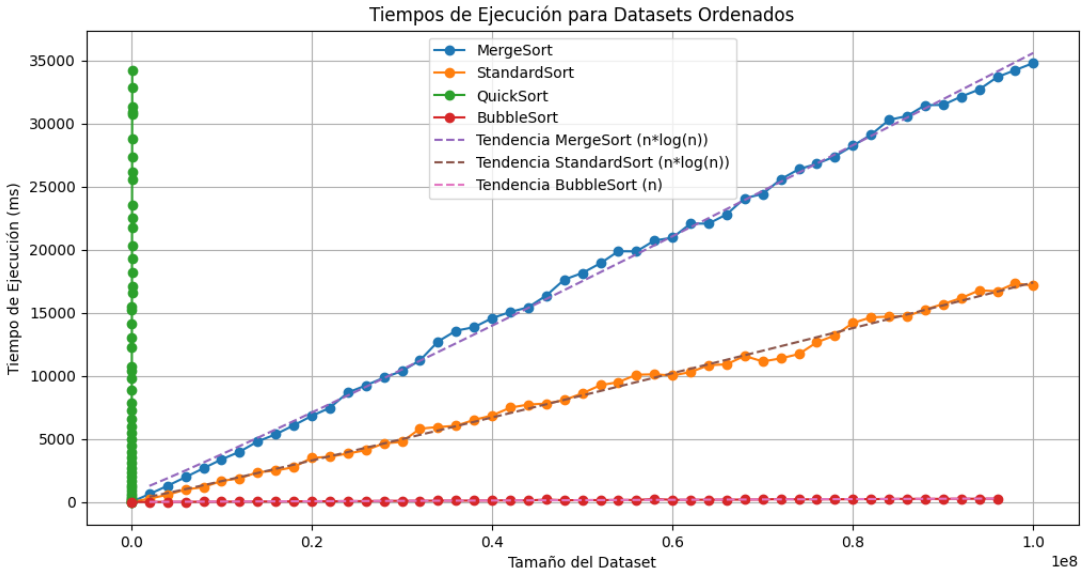

Figure 3: Grafico de algoritmos de ordenamiento en datasets ordenados

Es importante señalar que los algoritmos eficientes en el caso de datasets ordenados son MergeSort, la función `Sort()` de C++ y, en ciertas condiciones específicas, BubbleSort. Aunque `Sort()` utiliza InsertionSort en pequeñas particiones para acelerar el proceso cuando los datos ya están ordenados o casi ordenados, su complejidad sigue siendo $O(n \log n)$ en todos los casos. Esto quiere decir que, aunque en la práctica pueda ser más rápida en situaciones específicas, como con datasets ordenados, sigue manteniendo esa eficiencia teórica de $O(n \log n)$ al ser un algoritmo híbrido. MergeSort, por otro lado, es consistente en su rendimiento con una complejidad de $O(n \log n)$, sin importar si los datos están ordenados o no. En cuanto a BubbleSort, su eficiencia mejora notablemente en datasets ordenados, donde su optimización le permite alcanzar una complejidad de $O(n)$, deteniéndose después de una pasada al detectar que no se necesitan más intercambios, lo que lo convierte en una opción sorprendentemente efectiva en estos casos específicos.

Podemos destacar que QuickSort presenta problemas en este caso debido a que, al manejar datos ya ordenados, su eficiencia empeora considerablemente, alcanzando una complejidad de $O(n^2)$. Esto se debe a que el pivote elegido no realiza particiones equilibradas, lo que resulta en un comportamiento mucho más lento. A continuación se presentarán la tabla y el gráfico del comportamiento de QuickSort.

ORDENADOS	QuickSort (ms)
0	0
2000	14
4000	59
6000	123
8000	219
10000	341
12000	490
14000	681
16000	883
18000	1128
20000	1373
22000	1663
24000	1991
26000	2332
28000	2686
30000	3087
32000	3577
34000	3968
36000	4440
38000	4978
40000	5494
42000	6024
44000	6581
46000	7253
48000	7891
50000	8868
52000	9812
54000	10404
56000	10761
58000	12231
60000	13011
62000	14100
64000	15240
66000	15483
68000	16621
70000	17060
72000	18223
74000	19343
76000	20312
78000	21792
80000	22503
82000	23540
84000	25596
86000	26176
88000	27386
90000	28817
92000	30772
94000	30906
96000	31382
98000	32844
100000	34220

Table 4: Tiempos de ejecución para QuickSort en milisegundos para datos ordenados.

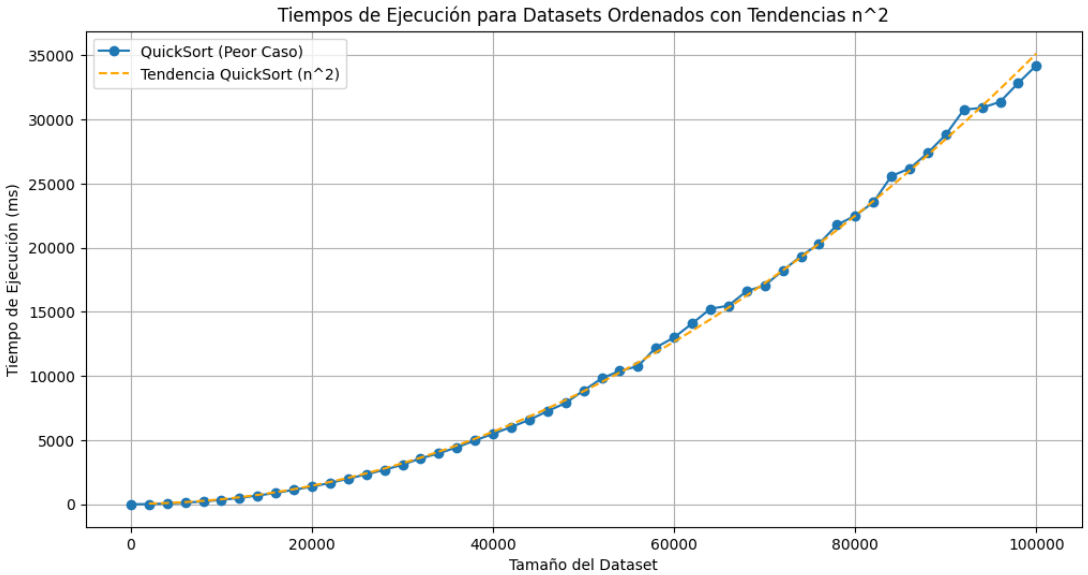

Figure 4: QuickSort en datasets ordenados

Podemos notar que QuickSort tarda casi el triple en ejecutar cada dataset con respecto a BubbleSort.

3.6.3 Datasets Semi-Ordenados

En esta sección presentaré los resultados de aplicar los distintos algoritmos de ordenamiento en datasets semi-ordenados, donde la mitad de los datos estan ordenados, mientras que, la otra mitad son distribuidos con numeros al azar. A Continuación se presentarán las tablas:

SEMI_ORDENADO	MergeSort (ms)	QuickSort (ms)	StandardSort (ms)
0	0	0	0
2000000	828	571	436
4000000	1686	889	879
6000000	2568	1660	1337
8000000	3541	2365	1801
10000000	4479	2778	2342
12000000	5573	3158	2979
14000000	6404	3305	3209
16000000	7406	4143	3813
18000000	8273	4586	4441
20000000	9286	5210	4754
22000000	10086	5770	5480
24000000	11400	5993	5933
26000000	12477	8310	6403
28000000	13564	8368	7169
30000000	14591	8785	7706
32000000	15415	9358	8107
34000000	16004	10032	8907
36000000	16992	10057	9558
38000000	18041	10164	9517
40000000	19299	12843	10289
42000000	20160	12270	10632
44000000	21486	12910	11171
46000000	22645	13961	11933
48000000	24067	14337	12334
50000000	25911	13390	12850
52000000	29455	16110	13157
54000000	31480	15536	14363
56000000	32114	14958	15292
58000000	32651	17289	16408
60000000	33986	17669	17137
62000000	35342	18085	18077
64000000	35134	18689	19527
66000000	35571	20230	19795
68000000	39092	19898	21284
70000000	39679	21342	21682
72000000	38983	21645	23235
74000000	39936	21044	21237
76000000	42079	21692	23013
78000000	40517	21455	22581
80000000	44469	23475	24074
82000000	44249	25730	25293
84000000	44536	24506	24188
86000000	43308	25439	25451
88000000	41247	26252	26405
90000000	42675	25646	27019
92000000	43423	23765	26856
94000000	16 43422	24089	28199
96000000	44906	24621	27973
98000000	45913	24578	28678
100000000	46602	24628	28912

Table 5: Tiempos de ejecución en milisegundos para MergeSort, QuickSort y StandardSort con datos semi-ordenados.

En la tabla presentada, podemos observar que MergeSort, QuickSort y StandardSort se comportan de manera eficiente en los datasets semi-ordenados. Este tipo de dataset, que combina datos previamente ordenados con números distribuidos al azar, pone a prueba la capacidad de cada algoritmo para manejar tanto el orden preexistente como la desorganización añadida.

MergeSort muestra tiempos de ejecución consistentes y adecuados a lo largo del crecimiento del dataset, reafirmando su robustez con su complejidad de $O(n \log n)$ para todos los casos. QuickSort, a pesar de ser sensible al orden de los datos, también presenta una eficiencia razonable en este caso, ya que no encuentra los problemas típicos que enfrenta con datos completamente ordenados. Finalmente, StandardSort mantiene un rendimiento excelente, demostrando por qué es el algoritmo de ordenamiento más confiable en la práctica para datasets de diverso tipo.

A continuación se presentará la tabla asociada al BubbleSort que tiene la complejidad esperada:

SEMI-ORDENADO	BubbleSort (ms)
0	0
2000	9
4000	38
6000	83
8000	148
10000	231
12000	338
14000	451
16000	630
18000	786
20000	1028
22000	1143
24000	1394
26000	1775
28000	1890
30000	2328
32000	2630
34000	2800
36000	3331
38000	3639
40000	4203
42000	4365
44000	5178
46000	5577
48000	5904
50000	6830
52000	7287
54000	7867
56000	7899
58000	8927
60000	9393
62000	10202
64000	10807
66000	11932
68000	13256
70000	14094
72000	15192
74000	16367
76000	17048
78000	18039
80000	19767
82000	20099
84000	22273
86000	22573
88000	23853
90000	24167
92000	25386
94000	26671
96000	28103
98000	28523
100000	30686

Table 6: Tiempos de ejecución en milisegundos para BubbleSort con datos semi-ordenados.

El comportamiento de BubbleSort es el esperado en datasets semi-ordenados, mostrando una tendencia cuadrática ($O(n^2)$). A medida que aumenta el tamaño del dataset, los tiempos de ejecución crecen exponencialmente, lo que es característico de este algoritmo. A pesar de manejar la mitad de los datos ya ordenados, la naturaleza ineficiente de BubbleSort no se ve beneficiada significativamente, ya que sigue requiriendo múltiples pasadas y comparaciones innecesarias en cada iteración.

A continuación presentaré como se observan gráficamente los datos de las tablas:

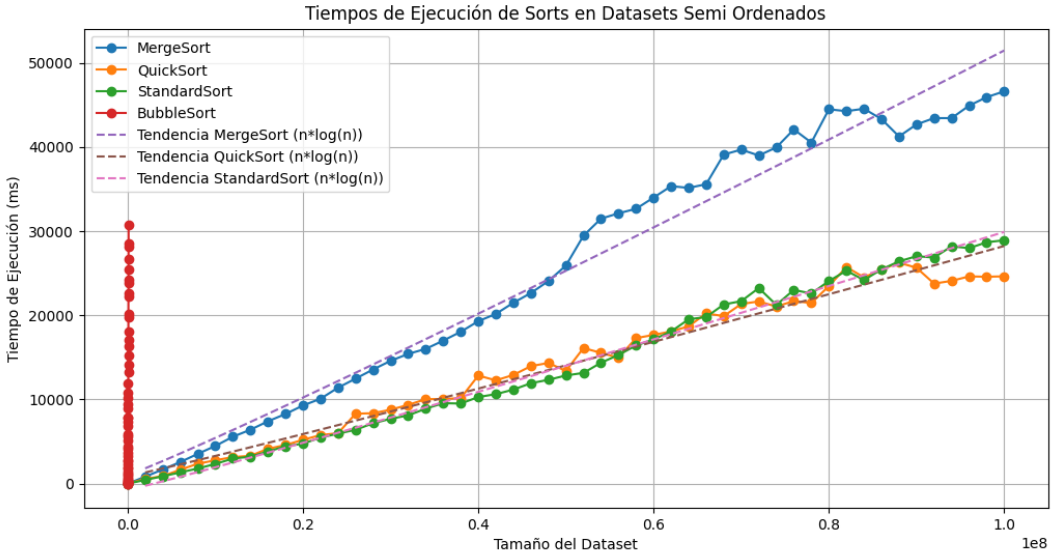

Figure 5: Sorts en datasets semiordenados

En el gráfico se observa el comportamiento esperado de los algoritmos de ordenamiento en datasets semiordenados. MergeSort y StandardSort siguen una tendencia clara de $O(n \log n)$, mientras que QuickSort también muestra un comportamiento cercano a esa eficiencia, aunque con algunas variaciones debido a la naturaleza de los datos semiordenados. BubbleSort, por otro lado, claramente exhibe una tendencia cuadrática $O(n^2)$, creciendo mucho más rápido en términos de tiempo de ejecución a medida que el tamaño del dataset aumenta. Las líneas de tendencia ajustadas para $n \log n$ confirman que tanto MergeSort como StandardSort se alinean con su complejidad esperada, destacando su eficiencia en comparación con BubbleSort. El gráfico perteneciente al algoritmo de BubbleSort, como es esperado, es de la siguiente forma:

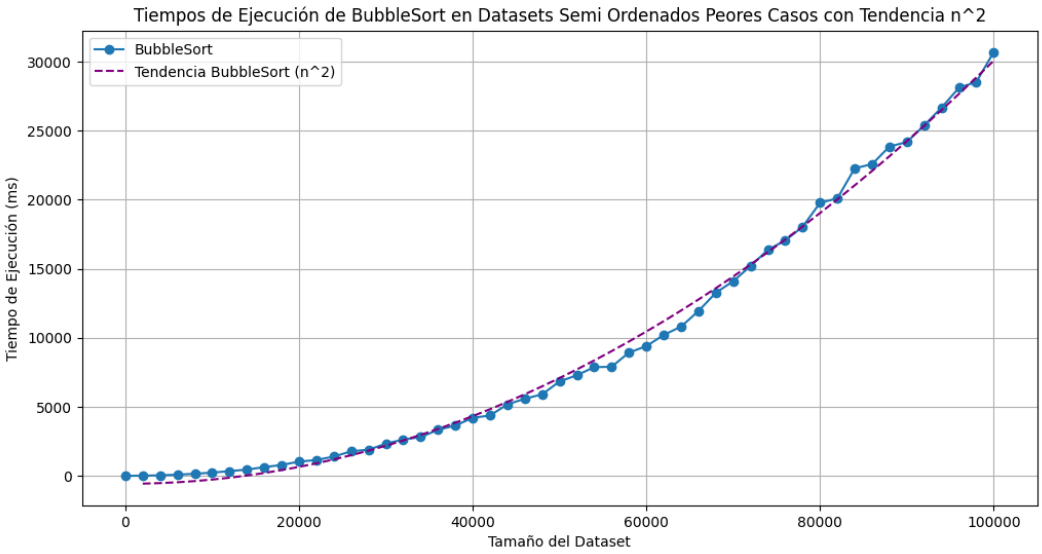

Figure 6: BubbleSort en datasets semiordenados

3.7 Conclusiones de los algoritmos de ordenamiento

En esta sección realizaremos conclusiones sobre el desempeño y comportamiento de cada uno de los algoritmos: BubbleSort, QuickSort, MergeSort y la función de Sort() de C++. Analizaremos cómo cada uno de ellos responde frente a cada uno de los datasets y qué tan bien cumplen con sus complejidades teóricas, destacando sus puntos fuertes y limitaciones.

3.7.1 BubbleSort

El algoritmo BubbleSort mostró el comportamiento esperado en cada uno de los casos. Debido a su naturaleza de $O(n^2)$, este algoritmo se desempeña de manera ineficiente cuando los datasets son grandes, ya que el número de comparaciones y pasadas necesarias para ordenar la lista crece de manera cuadrática.

Datasets Ordenados: En este caso, BubbleSort se benefició de la optimización que permite detener el algoritmo si no se realizan intercambios en una iteración completa. Esto redujo su complejidad a $O(n)$, mostrando tiempos de ejecución mucho más bajos en comparación con los datasets desordenados.

Datasets Semiordenados: A pesar de que la mitad de los datos ya estaban ordenados, el rendimiento de BubbleSort siguió mostrando su tendencia cuadrática. Esto se debe a que, aunque algunos elementos están en su lugar, el algoritmo sigue realizando comparaciones innecesarias en cada iteración.

Datasets Aleatorios: En este caso, el algoritmo presentó los peores tiempos de ejecución, ya que su naturaleza $O(n^2)$ se mostró sin optimización alguna. BubbleSort es claramente ineficiente en escenarios con datos completamente

aleatorios y desordenados.

3.7.2 QuickSort

QuickSort, siendo un algoritmo con una complejidad promedio de $O(n \log n)$, mostró un buen desempeño en la mayoría de los casos, excepto cuando se aplicó a datasets completamente ordenados. Debido a la elección del pivote (el primer elemento de la lista), el algoritmo sufrió una degradación en su eficiencia.

Datasets Ordenados: En este escenario, QuickSort alcanzó su peor rendimiento, mostrando una complejidad de $O(n^2)$. Esto ocurre porque el pivote elegido no divide la lista de manera equilibrada, lo que genera que el número de particiones aumente significativamente.

Datasets Semiordenados y Aleatorios: QuickSort se comportó de acuerdo con su complejidad esperada en estos casos, con tiempos de ejecución cercanos a $O(n \log n)$. A pesar de la sensibilidad del algoritmo al orden de los datos, los resultados fueron eficientes en estos casos.

3.7.3 MergeSort

MergeSort, con su complejidad garantizada de $O(n \log n)$, mostró un rendimiento consistente y eficiente en todos los tipos de datasets. Independientemente del estado inicial de los datos, MergeSort demostró ser robusto y mantener su complejidad teórica.

Datasets Ordenados, Semiordenados y Aleatorios: En todos los casos, MergeSort mantuvo su comportamiento esperado. Aunque consume más memoria que otros algoritmos debido a su necesidad de espacio extra para las divisiones recursivas, su consistencia y eficiencia lo hacen una excelente opción para grandes volúmenes de datos.

3.7.4 `Sort()` de C++

La función `Sort()` de C++ es un algoritmo híbrido que combina QuickSort, InsertionSort y HeapSort, lo que le permite adaptarse a diferentes tamaños de datasets de manera eficiente. Su rendimiento fue el mejor en la mayoría de los casos, debido a su capacidad para cambiar entre algoritmos dependiendo de las circunstancias.

Datasets Ordenados y Semiordenados: `Sort()` mantuvo su complejidad $O(n \log n)$ en estos casos, ajustando su estrategia según el estado de los datos.

Datasets Aleatorios: Al igual que en los otros casos, `Sort()` fue el más rápido y eficiente, aprovechando su capacidad para utilizar el algoritmo adecuado en cada situación.

En resumen, los algoritmos de ordenamiento analizados presentan diferencias significativas en su desempeño dependiendo del tipo de dataset. Mientras que MergeSort y `Sort()` de C++ se mantuvieron eficientes en todos los escenarios, QuickSort mostró vulnerabilidades cuando los datos estaban completamente

ordenados. Por su parte, BubbleSort demostró ser ineficiente en la mayoría de los casos, a excepción de los datasets ya ordenados, donde su optimización le permitió mejorar bastante su rendimiento en el caso de que el arreglo tenga sus datos ordenados.

4 Algoritmos de Multiplicación de Matrices

4.1 Introducción

Ahora, realizaré un análisis detallado de varios algoritmos de multiplicación de matrices, cada uno con su respectiva complejidad teórica. Evaluaré los tiempos de ejecución de estos algoritmos cuando se aplican a matrices de distintos tamaños, con el objetivo de observar si los resultados obtenidos corresponden a las expectativas teóricas. En este análisis, consideraremos tres algoritmos principales de multiplicación de matrices:

4.1.1 Algoritmo iterativo cúbico tradicional

Este es el enfoque clásico para la multiplicación de matrices, el cual se basa en un esquema iterativo que utiliza tres bucles anidados. Cada bucle recorre los índices de las matrices y acumula los productos de sus elementos en la matriz resultado.

En cuanto a la complejidad, este algoritmo es $O(n^3)$, lo que significa que, para multiplicar dos matrices de tamaño $n \times n$, el número de operaciones necesarias aumenta cúbicamente con el tamaño de las matrices.

```
// Funcion para realizar la multiplicacion de matrices utilizando
    el algoritmo cubico tradicional
vector<vector<int>> traditional(vector<vector<int>>& matrix1,
    vector<vector<int>>& matrix2) {
    int rows1 = matrix1.size();          // Numero de filas de la
        primera matriz
    int cols1 = matrix1[0].size();       // Numero de columnas de
        la primera matriz (y numero de filas de la segunda matriz)
    int cols2 = matrix2[0].size();       // Numero de columnas de
        la segunda matriz

    // Inicializacion de la matriz resultado con tamano adecuado y
        valores iniciales de 0
    vector<vector<int>> result(rows1, vector<int>(cols2, 0));

    // Iteracion cubica para la multiplicacion de matrices,
        adaptada para matrices no necesariamente cuadradas
    for (int i = 0; i < rows1; i++) {             // Recorre las
        filas de la primera matriz
        for (int j = 0; j < cols2; j++) {         // Recorre las
            columnas de la segunda matriz
            for (int k = 0; k < cols1; k++) {     // Recorre las
                columnas de la primera matriz (y filas de la
                segunda matriz)
                // Multiplica los elementos correspondientes de la
                    primera y segunda matriz y acumula el resultado
```

```
                result[i][j] += matrix1[i][k] * matrix2[k][j];
            }
        }
    }

    // Retorna la matriz resultante de la multiplicacion
    return result;
}
```

4.1.2 Algoritmo iterativo cúbico optimizado (transposición)

Este algoritmo es una optimización del enfoque cúbico tradicional, y la mejora principal se logra mediante la transposición de la segunda matriz antes de comenzar la multiplicación. Esta optimización se basa en aprovechar la localidad de los datos, mejorando el acceso secuencial a los elementos en la memoria, lo cual reduce el tiempo de acceso a la memoria caché.

Aunque transponer la matriz introduce un pequeño costo adicional, el algoritmo reduce considerablemente los tiempos de ejecución, especialmente en matrices grandes, donde el acceso a la memoria se convierte en un factor crítico. A pesar de esta optimización, la complejidad teórica del algoritmo sigue siendo $O(n^3)$, pero en la práctica, su rendimiento es mucho mejor que el del algoritmo cúbico tradicional en datasets de menor tamaño.

```
// Funcion para transponer una matriz
vector<vector<int>> transpose(vector<vector<int>>& matrix) {
    int rows = matrix.size();                // Numero de filas de
        la matriz original
    int cols = matrix[0].size();             // Numero de columnas
        de la matriz original
    // Inicializa una nueva matriz transpuesta con dimensiones
        invertidas
    vector<vector<int>> transposed(cols, vector<int>(rows));

    // Recorre cada elemento de la matriz original
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            // Asigna el valor transpuesto, intercambiando filas y
                columnas
            transposed[j][i] = matrix[i][j];
        }
    }

    // Retorna la matriz transpuesta
    return transposed;
}

// Funcion para realizar la multiplicacion de matrices optimizada
    transponiendo la segunda matriz
vector<vector<int>> optimized(vector<vector<int>>& matrix1, vector<
    vector<int>>& matrix2) {
    int rows1 = matrix1.size();              // Numero de filas de
        la primera matriz
    int cols1 = matrix1[0].size();           // Numero de columnas
        de la primera matriz (y numero de filas de la segunda
```

```
        matriz)
    int cols2 = matrix2[0].size();           // Numero de columnas
        de la segunda matriz

    // Transponer la segunda matriz para mejorar la localidad de
        los datos
    vector<vector<int>> matrix2T = transpose(matrix2);

    // Inicializa la matriz resultado con el tamano adecuado y
        valores iniciales de 0
    vector<vector<int>> result(rows1, vector<int>(cols2, 0));

    // Iteracion cubica optimizada, utilizando la matriz
        transpuesta
    for (int i = 0; i < rows1; i++) {              // Recorre las
        filas de la primera matriz
        for (int j = 0; j < cols2; j++) {          // Recorre las
            columnas de la segunda matriz (transpuesta)
            for (int k = 0; k < cols1; k++) {      // Recorre las
                columnas de la primera matriz (y filas de la
                segunda matriz transpuesta)
                // Multiplica los elementos correspondientes y
                    acumula el resultado
                result[i][j] += matrix1[i][k] * matrix2T[j][k];
            }
        }
    }

    // Retorna la matriz resultante de la multiplicacion
    return result;
}
```

4.1.3 Algoritmo de Strassen

Strassen es un algoritmo de multiplicación de matrices que reduce el número de multiplicaciones necesarias mediante una técnica recursiva. En lugar de realizar 8 multiplicaciones, como en el enfoque cúbico tradicional, Strassen reduce este número a 7, utilizando una serie de identidades algebraicas que permiten optimizar el proceso.

La complejidad de Strassen es $O(n^{\log_2 7})$, que equivale aproximadamente a $O(n^{2.81})$. Esto lo hace considerablemente más rápido que los algoritmos cúbicos para matrices grandes. Sin embargo, es importante mencionar que la constante oculta en su complejidad y el coste asociado a la recursión pueden hacer que Strassen sea ineficiente para matrices de tamaño pequeño o moderado. En esos casos, su rendimiento no supera al de los algoritmos cúbicos optimizados.

Según el Teorema Maestro, Strassen resuelve una recurrencia de la forma $T(n) = 7T(n/2)+O(n^2)$, lo que corresponde al caso 1 del teorema, donde $a = 7$, $b = 2$ y $d = 2$. Esto genera una complejidad final de $O(n^{\log_b a}) = O(n^{\log_2 7})$. Aunque esto es asintóticamente más rápido que el enfoque cúbico ($O(n^3)$), las constantes asociadas a las operaciones adicionales (como las 7 multiplicaciones de matrices más pequeñas y las operaciones de suma) generan una sobrecarga significativa. En las matrices pequeñas que analizaremos, estas constantes no se

amortizan, lo que hace que Strassen no sea bueno frente a los algoritmos cúbicos optimizados, que tienen una implementación más directa y menos sobrecarga. Sin embargo, Strassen es mucho más eficiente para matrices de tamaño gigante.

```
// Funcion Strassen para multiplicar matrices
vector<vector<int>> Strassen(vector<vector<int>>& matrix_A, vector<
    vector<int>>& matrix_B) {
    int col_1 = matrix_A[0].size();      // Numero de columnas de
        la matriz A
    int row_1 = matrix_A.size();         // Numero de filas de la
        matriz A
    int col_2 = matrix_B[0].size();      // Numero de columnas de
        la matriz B
    int row_2 = matrix_B.size();         // Numero de filas de la
        matriz B

    // Verifica que el numero de columnas de A sea igual al numero
        de filas de B (condicion para multiplicacion de matrices)
    if (col_1 != row_2) {
        cout << "\nError: The number of columns in Matrix A must be
            equal to the number of rows in Matrix B\n";
        return {}; // Retorna un resultado vacio si las matrices no
             son multiplicables
    }

    // Inicializa la matriz resultante con el numero correcto de
        filas y columnas
    vector<int> result_matrix_row(col_2, 0);               // Fila
        de la matriz resultado con ceros
    vector<vector<int>> result_matrix(row_1, result_matrix_row); //
         Matriz de resultado

    // Caso base: si las matrices son de 1x1, multiplica los dos
        valores directamente
    if (col_1 == 1) {
        result_matrix[0][0] = matrix_A[0][0] * matrix_B[0][0];
    } else {
        // Caso recursivo: divide las matrices en 4 submatrices mas
             pequenas
        int split_index = col_1 / 2;                          //
            Punto de division de las matrices
        vector<int> row_vector(split_index, 0);              // Fila
             auxiliar con ceros para inicializar submatrices

        // Inicializacion de las submatrices A y B
        vector<vector<int>> a00(split_index, row_vector);
        vector<vector<int>> a01(split_index, row_vector);
        vector<vector<int>> a10(split_index, row_vector);
        vector<vector<int>> a11(split_index, row_vector);
        vector<vector<int>> b00(split_index, row_vector);
        vector<vector<int>> b01(split_index, row_vector);
        vector<vector<int>> b10(split_index, row_vector);
        vector<vector<int>> b11(split_index, row_vector);

        // Llena las submatrices A y B dividiendo las matrices
            originales
        for (int i = 0; i < split_index; i++) {
```

```
        for (int j = 0; j < split_index; j++) {
            a00[i][j] = matrix_A[i][j];
                // Parte superior izquierda de A
            a01[i][j] = matrix_A[i][j + split_index];
                // Parte superior derecha de A
            a10[i][j] = matrix_A[split_index + i][j];
                // Parte inferior izquierda de A
            a11[i][j] = matrix_A[i + split_index][j +
                split_index]; // Parte inferior derecha de A
            b00[i][j] = matrix_B[i][j];
                // Parte superior izquierda de B
            b01[i][j] = matrix_B[i][j + split_index];
                // Parte superior derecha de B
            b10[i][j] = matrix_B[split_index + i][j];
                // Parte inferior izquierda de B
            b11[i][j] = matrix_B[i + split_index][j +
                split_index]; // Parte inferior derecha de B
        }
    }

    // Evita pasar valores temporales directamente (Rvalues),
        almacena los resultados en variables intermedias

    // Calculo de las 7 multiplicaciones segun el algoritmo de
        Strassen
    vector<vector<int>> temp1 = add_matrix(b01, b11,
        split_index, -1); // b01 - b11
    vector<vector<int>> p = Strassen(a00, temp1);
                            // p = a00 * (b01 - b11)

    vector<vector<int>> temp2 = add_matrix(a00, a01,
        split_index);     // a00 + a01
    vector<vector<int>> q = Strassen(temp2, b11);
                            // q = (a00 + a01) * b11

    vector<vector<int>> temp3 = add_matrix(a10, a11,
        split_index);     // a10 + a11
    vector<vector<int>> r = Strassen(temp3, b00);
                            // r = (a10 + a11) * b00

    vector<vector<int>> temp4 = add_matrix(b10, b00,
        split_index, -1); // b10 - b00
    vector<vector<int>> s = Strassen(a11, temp4);
                            // s = a11 * (b10 - b00)

    vector<vector<int>> temp5a = add_matrix(a00, a11,
        split_index);    // a00 + a11
    vector<vector<int>> temp5b = add_matrix(b00, b11,
        split_index);    // b00 + b11
    vector<vector<int>> t = Strassen(temp5a, temp5b);
                           // t = (a00 + a11) * (b00 + b11)

    vector<vector<int>> temp6a = add_matrix(a01, a11,
        split_index, -1); // a01 - a11
    vector<vector<int>> temp6b = add_matrix(b10, b11,
        split_index);    // b10 + b11
    vector<vector<int>> u = Strassen(temp6a, temp6b);
```

```
                        // u = (a01 - a11) * (b10 + b11)

        vector<vector<int>> temp7a = add_matrix(a00, a10,
            split_index, -1); // a00 - a10
        vector<vector<int>> temp7b = add_matrix(b00, b01,
            split_index);    // b00 + b01
        vector<vector<int>> v = Strassen(temp7a, temp7b);
                        // v = (a00 - a10) * (b00 + b01)

        // Combina los resultados de las 7 multiplicaciones para
            obtener las submatrices de la matriz resultante
        vector<vector<int>> temp8 = add_matrix(t, s, split_index);
                    // t + s
        vector<vector<int>> temp9 = add_matrix(temp8, u,
            split_index);     // (t + s) + u
        vector<vector<int>> result_matrix_00 = add_matrix(temp9, q,
             split_index, -1);  // result_matrix_00 = (t + s + u -
            q)

        vector<vector<int>> result_matrix_01 = add_matrix(p, q,
            split_index);           // result_matrix_01 = p + q

        vector<vector<int>> result_matrix_10 = add_matrix(r, s,
            split_index);           // result_matrix_10 = r + s

        vector<vector<int>> temp10 = add_matrix(t, p, split_index);
                     // t + p
        vector<vector<int>> temp11 = add_matrix(temp10, r,
            split_index, -1);// (t + p - r)
        vector<vector<int>> result_matrix_11 = add_matrix(temp11, v
            , split_index, -1); // result_matrix_11 = (t + p - r -
            v)

        // Reconstruccion de la matriz resultante desde las
            submatrices calculadas
        for (int i = 0; i < split_index; i++) {
            for (int j = 0; j < split_index; j++) {
                result_matrix[i][j] = result_matrix_00[i][j];
                                // Parte superior izquierda
                result_matrix[i][j + split_index] =
                    result_matrix_01[i][j]; // Parte superior
                    derecha
                result_matrix[split_index + i][j] =
                    result_matrix_10[i][j]; // Parte inferior
                    izquierda
                result_matrix[i + split_index][j + split_index] =
                    result_matrix_11[i][j]; // Parte inferior
                    derecha
            }
        }
    }
    return result_matrix;  // Retorna la matriz final resultante
}
```

4.2 Datasets

Para demostrar la eficiencia de los algoritmos de multiplicación de matrices, se utilizaron matrices cuadráticas y matrices de dimensiones distintas. Las matrices cuadráticas permiten la aplicación de todos los algoritmos, incluyendo el algoritmo de Strassen, mientras que para las matrices de dimensiones distintas se excluyó este último debido a sus propias limitaciones.

Se trabajó con matrices de diferentes tamaños, generadas en archivos `.txt`, los cuales son leídos por cada programa que implementa los algoritmos de multiplicación. Las matrices utilizadas variaron en tamaño, desde matrices de dimensiones 50×40 hasta matrices de dimensiones 1000×800. Estas dimensiones fueron elegidas para reflejar cómo se comportan los algoritmos con diferentes tamaños y proporciones de matrices.

En el caso de las matrices cuadráticas, el tamaño de las filas y columnas es el mismo, permitiendo el uso del algoritmo de Strassen, que se destaca por su eficiencia teórica con una complejidad de $O(n^{2.81})$. Sin embargo, cuando las matrices no son cuadráticas, el algoritmo de Strassen no puede ser aplicado, ya que dicho algoritmo está diseñado para dividir las matrices en submatrices de igual tamaño, lo que es imposible si las matrices no son cuadradas. En estos casos, se utilizaron únicamente los algoritmos tradicionales y optimizados, ambos con una complejidad cúbica de $O(n^3)$.

Los tipos de datasets utilizados incluyen:

- **Matrices cuadráticas**: Matrices donde el número de filas es igual al número de columnas, permitiendo la aplicación de todos los algoritmos.
- **Matrices de dimensiones distintas**: Matrices en las que el número de filas no es igual al número de columnas, donde se excluyó el algoritmo de Strassen.

Después de la multiplicación, se registró el tiempo de ejecución de cada algoritmo para cada dataset, con el objetivo de comparar el rendimiento entre los métodos tradicionales, optimizados y Strassen.

4.3 Resultados

Para reflejar la complejidad algoritmica utilicé Pandas, el cual me permitió generar gráficos y estudiar las tendencias asociadas a cada algoritmo de multiplicación de matrices en los distintos datasets que fueron generados.

4.3.1 Matrices Cuadráticas

A continuación presentamos una tabla con los tiempos de ejecución del algoritmo de multiplicación de matrices tradicional, optimizado y Strassen en matrices cuadráticas.

Para poder realizar un mejor análisis utilizaremos un gráfico que modele los datos de la tabla.

Table 7: Tiempos de ejecución para matrices cuadráticas

Tamaño de la matriz	Tradicional (ms)	Optimizado (ms)	Strassen (ms)
100x100	6	6	161
200x200	55	52	1135
300x300	182	175	7347
400x400	443	421	8009
500x500	855	825	8221
600x600	1482	1413	51110
700x700	2320	2252	52359
800x800	3449	3353	55952
900x900	4937	4762	57265
1000x1000	6733	6573	57602

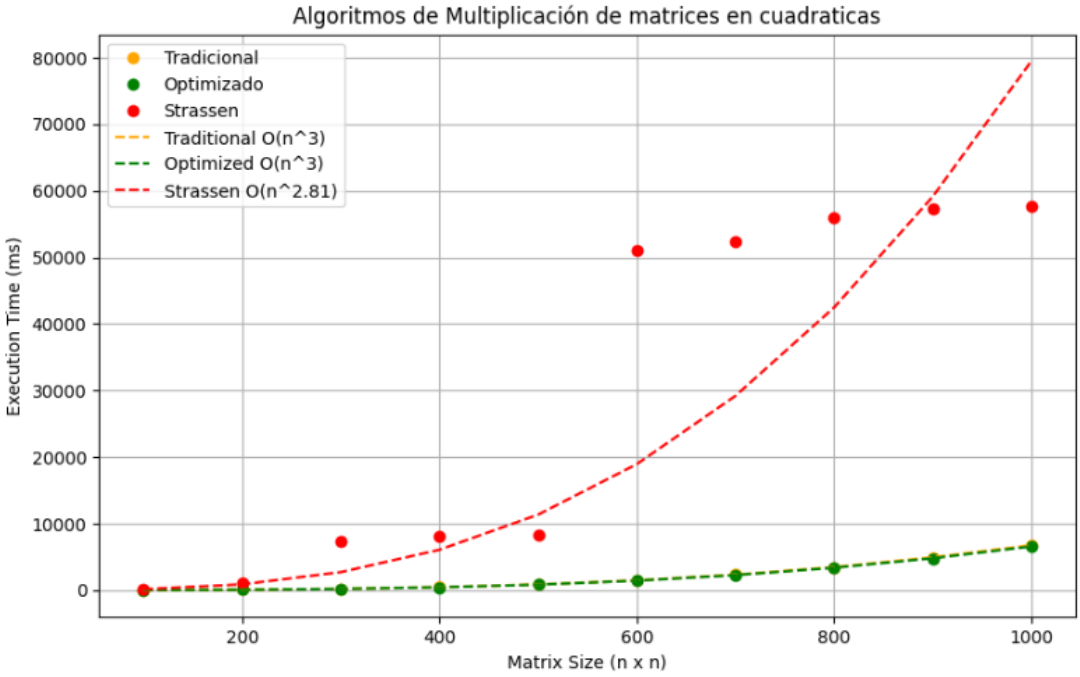

Figure 7: Algoritmos de multiplicación de matrices en cuadráticas

El gráfico refleja que los algoritmos **Tradicional** y **Optimizado**, ambos con complejidad $O(n^3)$, se comportan de manera predecible y eficiente en matrices pequeñas y medianas, siendo el optimizado un poco superior gracias a la utilización de la transposición de las matrices. Podemos notar que escalarmente, el rendimiento varía muy poco entre el algoritmo tradicional y el optimizado. Sin embargo, el algoritmo **Strassen**, con una complejidad teórica de $O(n^{2.81})$, resulta mucho más lento en la práctica. Esto se debe a los costos adicionales

que introduce, como la división y recombinación de matrices, además de las operaciones recursivas que aumentan el *overhead*, haciendo que pierda eficiencia frente a los otros dos métodos. Al final, la ventaja teórica de Strassen solo se manifiesta en matrices extremadamente grandes, donde el costo de las multiplicaciones empieza a superar la sobrecarga que genera la recursión.

4.3.2 Matrices de distintas dimensiones

Para las matrices de distintas dimensiones no podemos probar Strassen debido a sus propias limitaciones, sin embargo, probaremos estos casos de prueba para observar qué tal se comportan el algoritmo de multiplicación de matrices tradicional y el optimizado. A continuación, se presenta una tabla con los tiempos de ejecución de ambos algoritmos:

Table 8: Tiempos de ejecución para matrices de tamaños distintos

Filas x Columnas de 2 matrices a multiplicar	**Tradicional (ms)**	**Optimizado (ms)**
50x40 y 40x30	0	0
100x80 y 80x60	3	3
150x120 y 120x90	11	11
200x160 y 160x120	26	25
250x200 y 200x150	50	47
300x240 y 240x180	84	80
350x280 y 280x210	133	130
400x320 y 320x240	199	192
450x360 y 360x270	285	274
500x400 y 400x300	398	378
550x440 y 440x330	528	502
600x480 y 480x360	675	652
650x520 y 520x390	857	826
700x560 y 560x420	1084	1038
750x600 y 600x450	1322	1283
800x640 y 640x480	1613	1543
850x680 y 680x510	1923	1854
900x720 y 720x540	2277	2206
950x760 y 760x570	2677	2596
1000x800 y 800x600	3120	3015

Podemos notar, a partir de la tabla, que el comportamiento de los algoritmos de multiplicación de matrices tradicional y optimizado se comportan muy similar a sus tiempos de ejecución al multiplicar matrices cuadraticas. El algoritmo optimizado no genera una gran ventaja, debido a que, la traspoción al mejorar la localidad de los datos, es un poco más eficiente. Sin embargo, para matrices más grandes que aquellas de los casos de prueba, este algoritmo al tener que realizar todo un recorrido extra por la matriz para generar la transpuesta, el costo extra de esta operación va a hacer que sea más ineficiente que el algoritmo

tradicional.

4.4 Conclusiones

4.4.1 Algoritmo Tradicional

El algoritmo de multiplicación de matrices tradicional, con complejidad $O(n^3)$, es un método facil de implementar y con tiempos dejecucuión predecibles tanto para matrices cuadraticas como de distinto tamaño. Su rendimiento, aunque no optimizado en términos de operaciones de acceso a memoria, mantiene una consistencia en los tiempos de ejecución, lo que lo convierte en una opción robusta para situaciones donde no se necesiten grandes optimizaciones. A medida que el tamaño de las matrices aumenta, el costo computacional crece de manera cúbica.

4.4.2 Algoritmo Optimizado (Transposición)

El algoritmo optimizado utiliza la transposición para mejorar el acceso a la memoria, reduciendo el tiempo de búsqueda y lectura de datos de las matrices a multiplicar. Aunque su complejidad sigue siendo $O(n^3)$, en tanto las matrices cuadraticas como de distinto tamaño logra tiempos ligeramente mejores que el tradicional, especialmente en matrices de tamaño moderado. Sin embargo, a medida que el tamaño de las matrices aumenta considerablemente, el costo adicional de transponer la matriz y acceder a los elementos de manera secuencial puede contrarrestar las ganancias de rendimiento, lo que eventualmente lo hace inferior al método tradicional.

4.4.3 Algoritmo de Strassen

El algoritmo de Strassen, con complejidad $O(n^{2.81})$, presenta, teóricamente, una mejora significativa sobre los métodos cúbicos. No obstante, en la práctica, este algoritmo introduce una sobrecarga importante debido a las operaciones adicionales de división y recombinación de matrices, lo que lo hace menos eficiente en matrices pequeñas o medianas. Además, es importante destacar que Strassen solo funciona para matrices cuadradas de igual tamaño, ya que el algoritmo divide las matrices en submatrices iguales de manera recursiva, lo que no es posible cuando las dimensiones no coinciden. Es únicamente en matrices cuadradas de gran tamaño donde se puede observar una verdadera ventaja, ya que la reducción en el número de multiplicaciones empieza a compensar el costo de la recursión y las operaciones de recombinación.

5 Conclusión

5.1 Algoritmos de Ordenamiento

A partir de los tres casos de prueba realizados (datos ordenados, semiordenados y random), podemos concluir que no es recomendable utilizar Mergesort, Quick-

sort, y mucho menos BubbleSort, dado que la función `std::sort()` de C++ es extremadamente eficiente y está muy bien optimizada. Esto se refleja claramente en su superioridad en los tres tipos de casos de prueba, como se muestra en los gráficos. Esta ventaja de `std::sort()` proviene de su implementación basada en IntroSort, un algoritmo híbrido que combina de manera inteligente InsertSort, HeapSort y QuickSort, aprovechando lo mejor de cada uno de estos algoritmos de ordenamiento. Aunque BubbleSort mostró un comportamiento $O(n)$ en el caso donde los datos ya estaban ordenados, en la práctica, cuando se trabaja con grandes volúmenes de datos, es poco probable que queramos ordenar algo que ya esté ordenado, o que los datos al azar se presenten de manera ordenada de forma natural.

5.2 Algoritmos de Multiplicación de Matrices

En caso de que deseemos trabajar con la multiplicación de matrices cuadráticas, es importante considerar el tamaño de las matrices para elegir el algoritmo más adecuado. Para matrices pequeñas y medianas, el algoritmo Tradicional Optimizado resulta ser una buena opción debido a la mejora que ofrece al transponer una de las matrices, optimizando el acceso a la memoria y reduciendo el tiempo de ejecución en comparación con el algoritmo Tradicional. Sin embargo, a partir de cierto tamaño, la ventaja del Tradicional se vuelve evidente, ya que su implementación más directa evita el overhead adicional de la transposición, haciendo que su rendimiento sea superior.

Por otro lado, en el caso de matrices extremadamente grandes, el algoritmo de Strassen empieza a demostrar su verdadera fortaleza. Aunque en matrices pequeñas y medianas introduce un overhead considerable debido a las operaciones adicionales de división y recombinación de submatrices, en matrices de gran tamaño logra reducir significativamente el número de multiplicaciones, compensando estos costos y superando a los otros dos algoritmos en términos de eficiencia. Así, Strassen se convierte en la mejor opción para trabajar con matrices de gran envergadura, mientras que el Tradicional y su versión optimizada optimizada mantienen una ventaja en tamaños más pequeños o medianos. Hubiera sido interesante probar Strassen en matrices cuadráticas de mayor tamaño, pero debido a las limitaciones de tiempo y potencia computacional, no fue posible.

References

[1] Geeks for Geeks, *Algoritmos de Ordenamiento y Multiplicación de Matrices*, Disponible en: www.geeksforgeeks.org/

[2] ChatGPT, *Asistente Virtual Basado en Inteligencia Artificial*, OpenAI, Disponible en: https://openai.com/chatgpt

[3] Bro Code, *Learn Quick Sort in 13 minutes*, Disponible en: https://www.youtube.com/watch?v=Vtckgz38QHs&t=2s