Informe 1: Algoritmos de Ordenamiento y Multiplicación de Matrices

Raúl Astete Elguin

1 de mayo de 2022

Resumen

Este trabajo presenta una comparación de distintos algoritmos de ordenamiento de datos y multiplicación de matrices. La sección 2 incluye una descripción teórica sobre el funcionamiento de los algoritmos, que fueron implementados en C++, con el objetivo de comparar sus tiempos de ejecución en los experimentos, frente a su tiempo de ejecución teórico.

1. Introducción

1.1. Contexto y objetivos

Dentro de los objetivos personales para este trabajo, destacan el lograr unificar los conocimientos teóricos sobre algoritmos obtenidos en clases, y lograr una implementación que funcione y permita observar las diferencias experimentales entre los distintos procedimientos. A su vez, interesa lograr conclusiones certeras respecto al rendimiento de los algoritmos a considerar. Por otro lado, como fue mencionado en clases, interesa obtener una comprensión de los algoritmos de ordenamiento que sea lo suficientemente robusta como para, en un futuro, sea posible utilizar y entender código ya implementado en librerías clásicas.

1.2. Notación

En lo que sigue, denotaremos por $A[1 \cdots n]$ o A un arreglo de n elementos. Para todo $1 \leq j \leq n$, $A[j]$ será el elemento en la posición j del arreglo A. Consideraremos a $\mathbb{N}$ como el conjunto de números naturales.

2. Descripción de los algoritmos

2.1. Algoritmos de Ordenamiento

2.1.1. Insertion Sort

El primer algoritmo de ordenamiento con el que trabajaremos es INSERTION SORT, y utilizaremos la formulación descrita en Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). Introduction to algorithms. MIT press.. Es decir, dado un arreglo de n números flotantes $A = [A_1, \cdots, A_n]$ se considera el siguiente procedimiento

Algoritmo INSERTION SORT

Input: $A[1 \cdots n], n \in \mathbb{N}$

1: **for** $j = 2$ to n **do**
2: Key $= A[j]$
3: $i = j - 1$
4: **while** $i > 0$ and $A[i] >$Key **do**
5: $A[i + 1] = A[i]$
6: $i = i - 1$
7: **end while**
8: $A[i + 1] =$Key
9: **end for**

Observamos que el for de la línea 1 conlleva $O(n)$ operaciones que se ejecutan en orden constante, mientras que el while de la línea 4, en el peor caso requerirá $O(n)$ ejecuciones, en las que cada iteración se compone de operaciones de orden constante. Por último, dado que la línea 8 solo consiste en una asignación (orden constante), se concluye que el tiempo de ejecución de INSERTION SORT en el peor caso es

$$T_{\text{InsertionSort}}(n) = O(n^2)$$

Por otro lado, cuando el arreglo ya está ordenado, INSERTION SORT solo requiere hacer $n-1$ comparaciones, pues en ningún caso se tendrá que $A[j-1] > A[j]$, por lo que no se entrará al while de la línea 4 para ningún j. En resumen, en el mejor caso

$$T_{\text{InsertionSort}}(n) = O(n)$$

Este algoritmo de ordenamiento es *In place* y estable. Es decir, no se requiere memoria adicional para generar la salida, y los elementos repetido preservan su orden relativo después del ordenamiento.

2.1.2. MergeSort

El algoritmo MERGE SORT está basado en el paradigma divide & conquer y dado un arreglo de tamaño $A[1 \cdots n]$, su funcionamiento es el siguiente:

- **Dividir:** Primero, dividimos el arreglo por la mitad, generando dos arreglos de tamaño $n/2$ cada uno.
- **Conquistar:** Posteriormente ordenamos los subarreglos mediante llamadas recursivas de MERGE SORT.
- **Combinar:** Mezclar en orden los subarreglos ya ordenados.

En este caso, el fondo de la recursividad será cuando tengamos arreglos de un elemento, pues en tal caso cada subarreglo estará ordenado por defecto. Primero describamos formalmente el paso de mezclar dos subarreglos en orden. Consideraremos que vamos a mezclar dos subarreglos consecutivos de un arreglo $A[1 \cdots n]$. Tales arreglos serán $A[l \cdots m]$ y $A[m+1 \cdots r]$, donde $m = l+(r-l)/2$ es el punto medio entre l y r. Utilizaremos tres índices i, j y k, que recorrerán respectivamente los subarreglos $A[l \cdots m]$, $A[m+1 \cdots r]$ y $A[1 \cdots n]$. Para tal efecto, inicializamos el valor de $k = l$, pues solo ingresaremos los elementos ordenados a partir del índice l. Realizaremos el siguiente procedimiento mientras que $i < |A[l \cdots m]|$ y $j < |A[m+1 \cdots, r]|$ (i.e. mientras sigan quedando elementos sin mezclar en ambos subarreglos.

- Si el elemento $A[i] \leq A[j]$, ingresamos en la posición $A[k]$ el menor, es decir $A[i]$. Posteriormente incrementamos i pues un elemento del subarreglo de la izquierda fue ingresado.
- En caso contrario, se tiene $A[i] > A[j]$, por lo que agregamos $A[j]$ al arreglo ordenado, i.e. $A[k] = A[j]$ e incrementamos j en uno.
- Terminando cualquiera de ambos casos incrementamos k en uno pues un elemento de alguno de los dos arreglos fue ingresado a la lista mezclada.

Ahora, si nos siguen quedando elementos solo en alguno de los dos subarreglos por ingresar al arreglo ordenado, los podemos ingresar todos a la vez pues ellos preservan un orden relativo para su mismo subarreglo, cuidando de incrementar los índices auxiliares respectivos. Es claro que el algoritmo de MERGE se traduce en $\Theta(r-l) = \Theta(n)$ operaciones,

La necesidad de crear copias de estos subarreglos auxiliares provoca que MERGESORT sea un alogoritmo que no es *In place* (i.e. es *Out of place*), es decir, se requiere memoria adicional para su realización. En resumen, podemos explicar el procedimiento de mezclar en orden con el siguiente pseudocódigo:

Algoritmo MERGE

Input: $A = [1 \cdots n]$, $l, r \in \{1, \ldots, n\}$, $l < r$

```
1:  i = 0; j = 0; k = l
2:  m = l + (r − l)/2
3:  while i < m + 1 − l and j < r − m do
4:     if A[i] ≤ A[j] then
5:        A[k] = A[i]
6:        i = i + 1
7:     else
8:        A[k] = A[j]
9:        j = j + 1
10:    end if
11:    k = k + 1
12: end while
13: while i < m + 1 − k do
14:    A[k] = A[i]
15:    i = i + 1
16:    k = k + 1
17: end while
18: while j < r − m do
19:    A[k] = A[j]
20:    j = j + 1
21:    k = k + 1
22: end while
```

Ahora, podemos utilizar este procedimiento para definir de manera recursiva el algoritmo de ordenamiento MERGE SORT

Algoritmo MERGESORT

Input: $A = [1 \cdots n], l, r$

```
1: if l < r then
2:    m = l + (r − l)/2
3:    MERGESORT(A[1 ··· n], l, m)
4:    MERGESORT(A[1 ··· n], m + 1, r)
5:    MERGE(A[1 ··· n], l, r)
6: end if
```

Respecto al tiempo de ejecución, observamos que para el caso base de ordenar un arreglo de un elemento, solo se requiere tiempo $O(1)$ (de hecho es nulo pues los arreglos de un elemento están por defecto ordenados) y cada subproblema se trata de ordenar un arreglo del mismo tamaño (más o menos uno, dependiendo la

paridad de n). Como en cada iteración se hacen dos llamadas recursivas, y la fase de MERGE conlleva un tiempo $O(n)$, se tiene que el tiempo de MERGESORT es

$$T_{\text{MERGESORT}}(n) = 2\,T_{\text{MERGESORT}}\left(\frac{n}{2}\right) + O(n)$$

Utilizando el Teorema Maestro de recurrencia [fuente], se concluye así que el tiempo de ejecución de MERGESORT es $O(n \log n)$. De manera más precisa, el peor caso se da cuando en cada paso de MERGE se deben hacer todas las comparaciones, esto ocurre cuando los dos valores más grandes están en listas opuestas, con lo que la fase MERGE requiere $n - 1 = O(n)$ comparaciones. Por otro lado, el mejor caso se da cuando el elemento más grande de un subarreglo ordenado es más pequeño que el primero del subarreglo opuesto, para cada fase de MERGE. En tal caso solo se requerirán $n/2 = O(n)$ comparaciones en cada paso. En definitiva, como en mejor y peor caso se requieren de igual modo $O(\log n)$ pasos recursivos para dividir el arreglo de tamaño n, y mezclar dos listas en cada paso es $O(n)$ tanto para el mejor y el peor caso, se concluye que el tiempo de ejecución de MERGESORT es $O(n \log n)$ para el mejor y el peor caso.

Cabe destacar que MERGE SORT es un algoritmo de ordenamiento estable, es decir, los datos repetidos mantienen su posición inicial en el arreglo ordenado. Además, como ya mencionamos, es *Out of place* pues requiere memoria adicional para guardar los subarreglos auxiliares generados.

2.1.3. QuickSort

Al igual que Merge sort, el algoritmo Quicksort aplica el paradigma Divide & conquer visto en clases. Utilizaremos la formulación descrita en [libro guia]. Sea $A[1 \cdots n]$ un arreglo, consideremos el siguiente procedimiento para ordenar el subarreglo $A[p \cdots r]$.

- **Dividir:** Particionamos el arreglo $A[p \cdots r]$ en dos (posiblemente vacíos) subarreglos $A[p \cdots q-1]$ y $A[q+1 \cdots r]$, de modo que cada elemento de $A[p \cdots q-1]$ sea menor o igual a $A[q]$ y que todos los elementos de $A[q+1 \cdots r]$ sean mayores o iguales a $A[q]$. El cálculo del índice q se realizará en la parte del procedimiento a la que llamaremos *partition*.
- **Conquistar:** Ordenamos los arreglos $A[p \cdots q-1]$ y $A[q+1 \cdots r]$ de manera recursiva utilizando quicksort.
- **Combinar:** Como $A[p \cdots q-1]$ y $A[q+1 \cdots r]$ quedan ordenados y $A[q]$ está en la posición correcta respecto a su orden $A[p \cdots r]$ está ordenado.

Primero, describiremos formalmente la idea clave de este algoritmo, que es la fase de *Partition*, en la cual ordenaremos el subarreglo $A[p \cdots r]$.

Algoritmo PARTITION

Input: $A = [p \cdots r]$,

```
1:  x = A[r]
2:  i = p − 1
3:  for j = p to r − 1 do
4:      if A[j] ≤ x then
5:          i = i + 1
6:          swap(A[i],A[j])
7:      end if
8:  end for
9:  swap(A[i + 1],A[r])
10: return i + 1
```

En la linea 1 definimos x como elemento en la ultima posición del arreglo A. A tal elemento lo llamaremos *pivote*. La idea del procedimiento PARTITION es retornar el índice q a utilizar en la fase de dividir. Para tal efecto, el for de la linea 3 se trata de observar aquellos elementos $A[j]$ para j entre p y a la izquierda del pivote para los cuales el orden relativo a x sea correcto. En tal caso, se incrementa i e intercambiamos el elemento en la posición i con el de la posición j. Con respecto al tiemo, las líneas 1,2 y 9 corresponden a operaciones que se ejecutan en $O(1)$, por lo que el tiempo de ejecución de PARTITION en el peor caso viene dado por el for de las líneas 3-8, que en su interior ejecuta $O(1)$ operaciones, $r-1-p$ veces. De tal modo, si consideramos un arreglo $A[p \cdots r]$ con $n = r - p$, en el peor caso el tiempo de ejecución de PARTITION es $O(n)$. Más precisamente, es posible concluir que $T_{\text{PARTITION}} = \Theta(n)$.

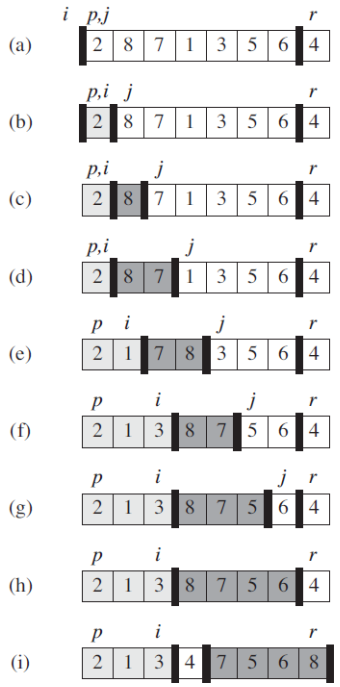


Figura 1: Partition en un arreglo especifico, fuente [libro-guia]

Consideremos el arreglo de la figura 1. Aquí $x = A[r]$ es el pivote, los elementos en gris claro son aquellos que son menores o iguales a x y en gris oscuro los que son mayores a x. Observamos que el valor con el que queda i al final es aquel índice a la izquierda de la posición que tendría x si estuviera en el arreglo ordenado. En [libro guia] se encuentra la demostración de la correctitud de este algoritmo, usando que al comienzo de cada iteración del for, para cualquier índice k se tienen los invariantes de ciclo:

1. Si $p \leq k \leq i$, $A[k] \leq x$.
2. Si $i+1 \leq k \leq j-1$, $A[k] > x$.
3. Si $k = r$, $A[k] = x$.

En vista de lo ya discutido, podemos definir el algoritmo QUICK SORT, en resumen elige un pivote (en nuestro caso lo consideraremos como el elemento de más a la derecha de la lista) y posteriormente se dejan a la izquierda todos los elementos menores que el pivote y a la derecha todos los mayores, luego cada subarreglo se ordena recursivamente. El fondo de la recursividad será cuando tengamos sub arreglos de tamaño 1 o 0, que ya están por defecto ordenados. Además, QUICKSORT es inplace pero genera un ordenamiento inestable.

Algoritmo QUICKSORT

Input: $A = [p \cdots r]$, $n = r - p$

```
1: if p < r then
2:     q = PARTITION(A[p ··· r])
3:     QUICKSORT(A[p ··· q − 1])
4:     QUICKSORT(A[q + 1 ··· r])
5: end if
```

Para analizar el mejor caso de QUICKSORT, podemos pensar cuando las

dos particiones definidas tienen el mismo tamaño. En tal caso, como ya vimos que $T_{\text{PARTITION}} = \Theta(n)$, se tiene que

$$T_{\text{QUICKSORT}}(n) = 2\,T_{\text{QUICKSORT}}\left(\frac{n}{2}\right) + O(n)$$

Utilizando el Teorema Maestro [https://doi.org/10.1145/1008861.1008865], obtenemos que en el mejor caso QUICKSORT tiene un tiempo de ejecución similar al de MERGE SORT

$$T_{\text{QUICKSORT}}(n) = O(n \log n)$$

Por otro lado, observamos que el peor caso se da cuando el arreglo es particionado en 1 y $n-1$ elementos, pues en tal situación debemos volver a aplicar QUICKSORT en un arreglo de $n-1$ elementos, es decir

$$\begin{aligned} T_{\text{QUICKSORT}}(n) &= T_{\text{QUICKSORT}}\,(n-1) + \Theta(n) \\ &= T_{\text{QUICKSORT}}\,(n-2) + \Theta(n) + \Theta(n-1) \\ &\vdots \\ &= \sum_{k=1}^{n} \Theta(k) \\ &= \Theta\left(\sum_{i=0}^{n} k\right) = \Theta(n^2) \end{aligned}$$

Cabe mencionar que es posible demostrar que el tiempo de ejecución para el caso promedio de QUICKSORT es $\Theta(n \log n)$, suponiendo que la distribución de las posibles permutaciones de entrada es uniforme.

2.1.4. Sort de la biblioteca estándar

Utilizaremos siguientes dos funciones de la biblioteca estándar:

SORT

La función `sort` de la biblioteca estándar es una implementación del algoritmo INTRO SORT (o Introspective Sort), el cual es un híbrido entre tres otros algoritmos: QuickSort, HEAPSORT e InsertionSort. El procedimiento empieza con QuickSort y si los pasos de recursión pasan un límite en particular, se cambia a HEAPSORT, para evitar el peor caso de QuickSort, que como vimos es $O(n^2)$. Si n es pequeño, se opta por utilizar Insertion Sort. Una descripción detallada de este algoritmo se encuentra en *Introspective Sorting and Selection Algorithms* , David R. Musser. Aquí se demuestra que el tiempo de ejecución de INTROSORT en el peor caso es de $O(n \log n)$, al igual que el peor caso y el promedio. Cabe destacar que este algoritmo es *In place*, pero no estable.

STABLE SORT

Corresponde a la función `stable_sort` de la biblioteca estándar, y es la implementación de un algoritmo adaptativo, es decir, que intenta asignar memoria temporal al buffer, por lo que su tiempo de ejecución depende de la cantidad de memoria que haya disponible. El peor caso se da cuando no hay memoria auxiliar disponible, y su tiempo de ejecución es $O(n \log^2 n)$, mientras que en el mejor caso una cantidad lo suficientemente grande de memoria auxiliar está disponible, para el cual STABLESORT ejecuta una variación de MERGESORT, lo que se traduce en un tiempo $O(n \log n)$. Como su nombre lo dice, este algoritmo es estable, pero no es *In place* si hay memoria disponible.

2.2. Algoritmos de Multiplicación de Matrices

Para el análisis siguiente, salvo que se diga lo contrario, denotaremos dos matrices A, B cuadradas de tamaño $n \times n$. Además, denotaremos por $\mathcal{M}_{n\times n}(\mathbb{R})$ al conjunto de matrices cuadradas de tamaño n, con elementos flotantes (o enteros, según el contexto lo indique). Además, obviaremos realizar el cálculo del tiempo de ejecución para mejor/peor caso, puesto que no asumimos ninguna propiedad sobre las matrices, y los procedimientos aquí descritos se realizarán de la misma manera (y mismo número de operaciones) para todo par de matrices, con lo que el tiempo en el peor caso y mejor caso coinciden.

2.2.1. Algoritmo iterativo Cúbico Tradicional

Para el algoritmo cúbico tradicional de multiplicación de matrices, consideramos el procedimiento siguiente:

Para $A = (a_{ij})_{1\leq i,j\leq n}$, $B = (b_{ij})_{1\leq i,j\leq n}$, calculamos el producto

$$AB = C = (c_{ij})_{1\leq i,j\leq n}, \quad c_{ij} = \sum_{k=1}^{n} a_{ik}b_{kj}$$

Algoritmo MATPROD

Input: $A, B \in \mathcal{M}_{n\times n}(\mathbb{R}), n \in \mathbb{N}$

```
1: for i = 1 to n do
2:    for j = 1 to n do
3:       for k = 1 to n do
4:          c_ij = c_ij + a_ik b_kj
5:       end for
6:    end for
7: end for
```

Dado que este algoritmo consiste en tres ciclos for anidados, con una asignación interna en la línea 4, su tiempo de ejecución es

$$T_{\text{MATPROD}}(n) = O(n^3)$$

2.2.2. Algoritmo iterativo cúbico optimizado para mantener la localidad de los datos

Denotemos por A_{i*} a la fila i de la matriz A y por B_{*j} a la columna j de la matriz B. Observemos ahora que si representamos por $\langle \cdot\,;\cdot \rangle$ al producto escalar usual (o producto punto) de dos vectores en $\mathbb{R}^n$ (columnas), el producto de dos matrices A y B se puede escribir como

$$AB = \left[\begin{array}{c} A_{1*} \\ \hline \vdots \\ \hline A_{n*} \end{array}\right] \left[B_{*1} \mid \cdots \mid B_{*n}\right] = \begin{bmatrix} \langle A_{1*}^T ; B_{*1} \rangle & \cdots & \langle A_{1*}^T ; B_{*n} \rangle \\ \vdots & \ddots & \vdots \\ \langle A_{n*}^T ; B_{*1} \rangle & \cdots & \langle A_{n*}^T ; B_{*n} \rangle \end{bmatrix}$$

Por esta razón, resulta útil pensar en las columnas de la matriz del lado derecho a la hora de hacer la multiplicación. Sin embargo, dado que guardamos las filas de la matriz como arreglos distintos, para cada columna B_{*i} de la matriz B tendremos que revisar n arreglos distintos. Una forma de mantener cada columna en un mismo arreglo es cambiar la forma en la que almacenamos B, de modo que cada columna de B sea un arreglo distinto. Otra forma de realizar esto es trasponer la matriz del lado derecho, pero eso significaría realizar trabajo $O(n^2)$ operaciones extra. Por tal motivo, optamos por guardar la matriz del lado derecho traspuesta, desde el momento en que la leemos de la entrada.

2.2.3. Algoritmo de Strassen

El algoritmo de multiplicación de matrices de Strassen es un procedimiento basado en el paradigma Divide & Conquer para la multiplicación de dos matrices cuadradas de dimensiones $n \times n$, donde n es una potencia de dos[1]. A grandes rasgos, el algoritmo consiste en, dadas dos matrices de $n \times n$, $A = (a_{ij})$, $B = (b_{ij})$:

- **Dividir:** Segmentamos las matrices A y B en bloques. Es decir, definimos las siguientes ocho submatrices:

$$\begin{aligned}
A_{11} &:= a_{ij}, \quad 1 \le i, j \le \frac{n}{2} & \qquad B_{11} &:= b_{ij}, \quad 1 \le i, j \le \frac{n}{2} \\
A_{12} &:= a_{ij}, \quad \frac{n}{2} \le i \le n, \; 1 \le j \le \frac{n}{2} & \qquad B_{12} &:= b_{ij}, \quad \frac{n}{2} \le i \le n, \; 1 \le j \le \frac{n}{2} \\
A_{21} &:= a_{ij}, \quad \frac{n}{2} \le j \le n, \; 1 \le i \le \frac{n}{2} & \qquad B_{21} &:= b_{ij}, \quad \frac{n}{2} \le j \le n, \; 1 \le i \le \frac{n}{2} \\
A_{22} &:= a_{ij}, \quad \frac{n}{2} \le i, j \le \frac{n}{2} & \qquad B_{22} &:= b_{ij}, \quad \frac{n}{2} \le i, j \le \frac{n}{2}
\end{aligned}$$

Es decir, escribimos nuestras matrices a multiplicar de la siguiente manera

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}, \quad B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

[1] Es posible generalizar el algoritmo a matrices cuadradas de cualquier orden, para lo que se requiere completar espacios de las matrices con ceros. Sin embargo, tal variación no fue implementada en este trabajo.

- **Conquistar:** Calculamos la matriz $C = AB$, de elementos

$$C = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

de la siguiente manera:

$$\begin{aligned} C_{11} &= M_1 + M_4 - M_5 + M_7 \\ C_{12} &= M_3 + M_5 \\ C_{21} &= M_2 + M_4 \\ C_{22} &= M_1 - M_2 + M_3 + M_6 \end{aligned}$$

donde

$$\begin{aligned} M_1 &:= (A_{11} + A_{22})(B_{11} + B_{22}) \\ M_2 &:= (A_{21} + A_{22})B_{11} \\ M_3 &:= A_{11}(B_{12} - B_{22}) \\ M_4 &:= A_{22}(B_{21} - B_{11}) \\ M_5 &:= (A_{11} + A_{12})B_{22} \\ M_6 &:= (A_{21} - A_{11})(B_{11} + B_{12}) \\ M7 &:= (A_{12} - A_{22})(B_{21} + B_{22}) \end{aligned}$$

y cada producto es calculado de manera recursiva con el algoritmo de Strassen. El fondo de la recursividad es multiplicar matrices 1×1 como si fuesen números[2]

Para este algoritmo no incluiremos un pseudocódigo, pues el detalle anterior ya describe los pasos fundamentales. Con respecto al tiempo de ejecución, observamos que en cada llamada recursiva, se resuelven 7 subproblemas de la mitad del tamaño original. Además, en cada paso calcular las matrices M_i requiere sumar y restar matrices de tamaño $n/2 \times n/2$, lo que se traduce en $\Theta(n^2)$ operaciones. En definitiva, podemos establecer que

$$T_{\text{STRASSEN}}(n) = 7\, T_{\text{STRASSEN}}\left(\frac{n}{2}\right) + O(n^2)$$

lo que en virtud del Teorema Maestro, corresponde a

$$T_{\text{STRASSEN}}(n) = O(n^{\log 7}) \approx O(n^{2,8074})$$

[2]En este trabajo utilizamos el fondo de recursividad como el caso en que $n = 2$, y ocupamos la fórmula de Strassen para las submatrices, que en este caso serían de un elemento.

3. Descripción de los Datasets

3.1. Datasets de arreglos

Para la generación de conjuntos de datos para testear los algoritmos de ordenamiento, se utilizó el script `random_array_generator.cpp` (disponible en `https://github.com/raulsebastian1999/FEDA.git`), el cual utiliza el generador de números pseudoaleatorios Mersenne twister para generar un arreglo A con n valores del tipo `long long int`, en un rango de $(-9999, 9999)$, distribuidos uniformemente. La generación de los datasets como archivos de entrada se realizó con ayuda del operador `>`. Se incluye dentro de este mismo script la posibilidad de generar arreglos ordenados de mayor a menor, ordenados totalmente o parcialmente ordenados. En `https://github.com/raulsebastian1999/FEDA.git` se incluyen las entradas utilizadas para generar las tablas y gráficos de la sección 4. Cabe destacar que se considerarán principalmente cuatro tipos de datasets:

- **D1:** Arreglos desordenados de tamaños entre 500 y 500000. Para obtener información general del comportamiento de los algoritmos implementados en lo que podría considerarse un caso promedio.
- **D2:** Arreglos ordenados de mayor a menor para tamaños entre 2^{10} y 2^{15}. No se siguió incrementando el tamaño pues para arreglos ordenados en reversa InsertionSort presentó una violación de segmento.
- **D3:** Arreglos de tamaño n cuyos primeros $n/2$ elementos ya están ordenados. Consideraremos tamaños entre 2^{10} y 2^{16}
- **D4:** Arreglos ya ordenados de tamaños entre 2^{12} y 2^{16} (posterior a eso QuickSort arroja Segmentation Fault).

3.2. Datasets de Matrices

De manera similar al generador de arreglos aleatorios, se utilizó el script `random_matrix_generator.cpp` (disponible en []). A diferencia del script para arreglos, se había pensado inicialmente en generar números `double` aleatorios. Sin embargo, para simplificar el trabajo se optó por realizar solo casos de prueba con valores enteros, pero dado que las funciones de matrices estaban definidas para valores double, se tuvo que hacer el casting de los números aleatorios a double, con el objetivo de en el futuro generalizar los datasets a números double. Dado que la implementación realizada del algoritmos recursivo de Strassen solo multiplica matrices cuadradas con igual cantidad de filas (potencia de dos), la generación de datasets que permitieran observar cualidades de los tres algoritmos simultaneamente se dificultó.

4. Resultados Experimentales

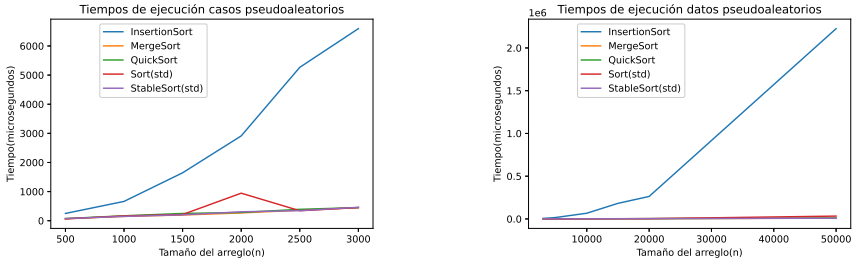
Principalmente se usaron 2 programas para los experimentos:

- Ordenamiento.cpp: Incluye todos los algoritmos de ordenamiento
- Algoritmos_multiplicacion_matrices.cpp: Todos los algoritmos de matrices.

Ambos están disponibles en `https://github.com/raulsebastian1999/FEDA.git`.

4.1. Algoritmos de Ordenamiento

Los siguientes gráficos muestran el tiempo de ejecución de los algoritmos considerados a partir de evaluarlos en un mismo dataset del tipo D1.



(a) Tamaños entre 500 y 3000 **(b)** Tamaños entre 3000 y 50000

Figura 2: Tiempos de ejecución sobre datasets del tipo D1

Dado que los datos provienen de una distribución de probabilidades uniforme, pseudoaleatoria, podríamos considerar que es una simulación del caso promedio. De tal modo, podemos observar que para tamaños pequeños (< 500) el tiempo de ejecución de InsertionSort no es tan distinto al de los otros algoritmos, lo que puede ser un indicio para explicar su utilización en los casos pequeños de IntroSort. En ese sentido, sería intuitivo esperar que InsertionSort y Sort de la librería estándar estén cerca de coincidir en algún punto, lo que no es observable a simple vista para los casos considerados. Frente a esto, conjeturamos la posibilidad de que la variante de InsertionSort utilizada por Sort sea de mayor rendimiento, o que esto ocurre para casos de arreglos mucho más pequeños de los considerados en este informe.

Por otro lado, para datasets del tipo D2 observamos el siguiente comportamiento:

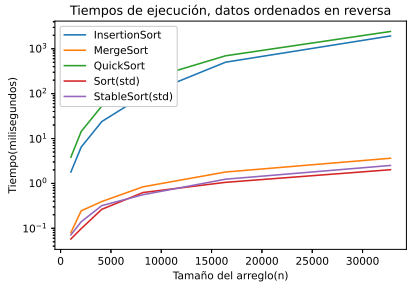


Figura 3: Tiempos de ejecución para datasets del tipo D2 (escala logarítimica)

Donde notamos que las implementaciones de Sort, StableSort y MergeSort son las que tienen un menor tiempo de ejecución y presentan un comportamiento similar. En cambio, los algoritmos de InsertionSort y QuickSort presentan un mayor tiempo de ejecución.

Para el conjunto de datasets del tipo D3, se observó el siguiente comportamiento:

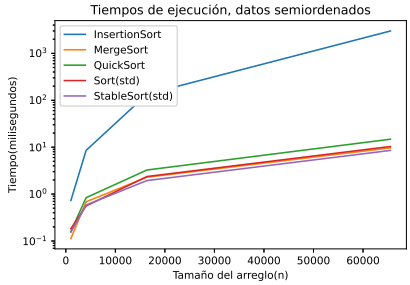


Figura 4: Tiempos de ejecución para datasets del tipo D2 (escala logarítimica)

Notamos que salvo InsertionSort, todos los algoritmos presentan un tiempo de ejecución similar, lo que coincide con la observación hecha anteriormente sobre el peor caso en el final de la sección 2.1.1.

A continuación presentamos el comportamiento gráfico de la evaluación de los algoritmos sobre los datasets del tipo D4, i.e. arreglos completamente ordenados. En el mismo espíritu del caso D3, se observa que el algoritmo con menor tiempo de ejecución es InsertionSort, lo que coincide con al análisis del mejor caso hecho en 2.1.1.

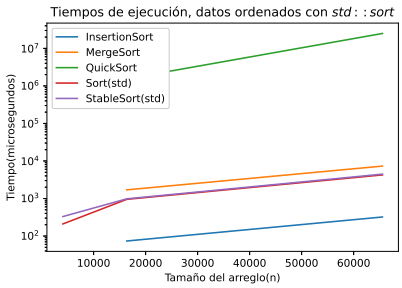
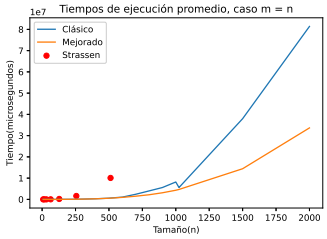


Figura 5: Tiempos de ejecución para datasets del tipo D4 (escala logarítimica)

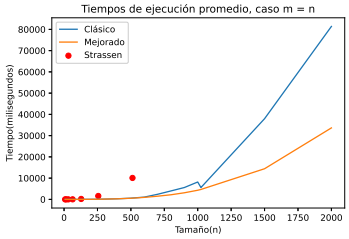
Además, observamos que QuickSort presenta un alto tiempo de ejecución, lo que no fue observado como peor caso en la sección teórica de este trabajo.

4.2. Multiplicación de Matrices

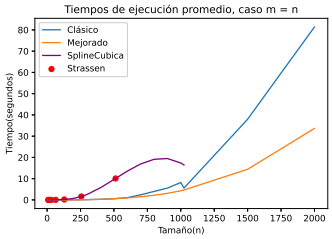
Para las primeras comparaciones, consideramos el producto de dos matrices cuadradas de tamaños desde 8×8 hasta 2000. En los siguientes gráficos observamos el tiempo de ejecución del algoritmo clásico de multiplicación, del mejorado y del algoritmo recursivo de Strassen para los casos 8×8, 16×16, 32×32, 64×64, 128×128, 256×256 y 512×512. Para los casos con mas de 1024 filas/columnas, la implementación considerada del algoritmo de Strassen provocaba una violación de segmento. Conjeturamos que esto se debe al carácter *naïve* del programa, pues genera muchas copias de matrices en cada paso de recursividad.



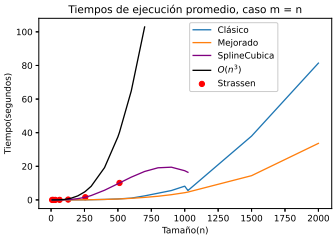
(a) Strassen en $n = 2^k$, $k = 3, 4, 5, 6, 7, 8, 9$



(b) Se observa la reducción en el tiempo que genera el algoritmo modificado.



(c) Se incluye la interpolación de los puntos en los que se usó Strassen



(d) Con $f(n) = 0{,}00003n^3$

Figura 6: Tiempos de ejecución, algoritmos de multiplicación de matrices

Como observamos en el caso de la figura 2.c, se calculó la spline cúbica interpolante de los puntos para los cuales fue posible realizar el producto de las matrices mediante el algoritmo de Strassen, con el objetivo de mostrar que, si bien para todos los casos de prueba el tiempo de ejecución fue mucho mayor que el de los otros algoritmos cúbicos, la función spline interpolante presenta una naturaleza a decaer. Se optó por interpolar con la spline cúbica dado que esta es un polinomio de grado 3 en los valores internos de la partición inducida por nuestras mediciones, lo que a priori puede ser una buena aproximación al comportamiento de nuestro algoritmo, que ya vimos tiene un tiempo de ejecución

aproximado de $O(n^{2,8074})$. El gráfico de la figura 2.d incluye en color negro una función particular de la clase $O(n^3)$ donde se aprecia que el comportamiento experimental de los algoritmos de multiplicación considerados coincide con el esperado.

5. Conclusiones

- Entendemos este trabajo como la proyección a pequeña escala de un proyecto estándar en ciencias de la computación: no se puede separar la implementación del conocimiento teórico de los algoritmos. Durante la realización de este trabajo fue esencial recurrir a la revisión de conceptos teóricos a la hora de programar, y las conclusiones obtenidas de la parte experimental eran esperables si se hace una lectura correcta de la parte matemática.

- *No basta con analizar solo el tiempo.* La situación que ejemplifica esta máxima es el fracaso al implementar el algoritmo recursivo de Strassen, con respecto al tiempo de ejecución teórico esperado. Frente a esto, conjeturamos que hay dos factores que generan esta desviación teórica/experimental: el manejo de la memoria a la hora de programar el algoritmo y la posibilidad de que la constante detrás del rendimiento $O(n^{2,8074})$ sea muy grande.

- Los resultados relativos a mejor y peor caso de QuickSort coinciden con los esperados tras el análisis teórico, y haber leído referencias. En definitiva, la existencia de estos contrastes empodera la idea de escoger un algoritmo de ordenamiento adecuado para cada problema.

    - Si queremos preservar orden en elementos repetidos y tenemos tiempo para programar un código extenso, es recomendable programar MergeSort.
    - Si no nos interesa preservar el orden de elementos repetidos, vamos a testear una gran cantidad de datos que no están ni ordenados de mayor a menor ni de menor a mayor, es recomendable usar QuickSort.
    - Si nos interesa preservar orden de elementos repetidos, no tenemos una gran cantidad de datos, y sabemos que existe una probabilidad de que los datos ya vengan ordenados, es recomendable InsertionSort.
    - Además, para sus respectivos casos útiles es recomendable usar Sort y StableSort, pues como ya vimos en los casos de prueba, están optimizados para ser eficientes en la mayoría de los casos.

- Respecto a los algoritmos de matrices, el mejor rendimiento del algoritmo modificado frente al clásico es un motivo más por el cual tener en cuenta que no sólo es importante el tiempo del algoritmo, sino también lo que el computador haga detrás, y cómo maneja la memoria.