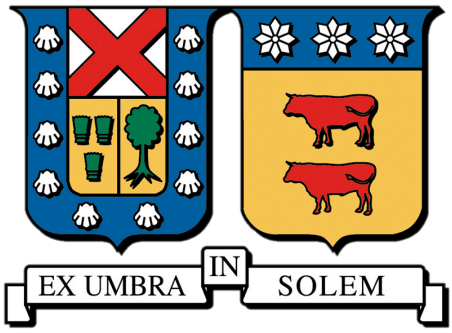


Algoritmos Dividir y Conquistar

2024-09-08

Joaquín Antonio Véliz Carmona
Univeridad tecnica Federico Santa Maria
INF-221
Algoritmos y complejidad

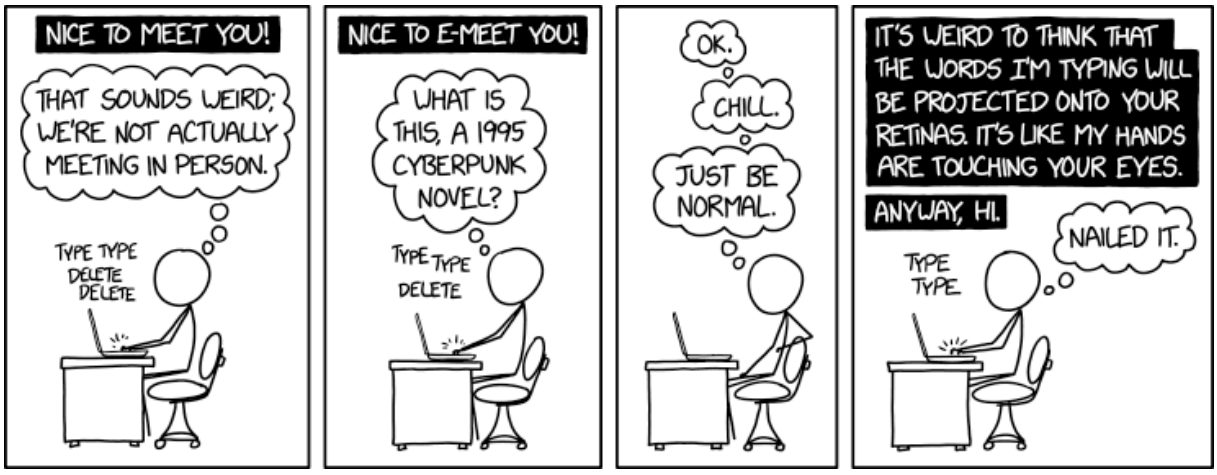
I. Introduccion

Este informe presenta un análisis de varios algoritmos de ordenamiento y multiplicación de matrices, con un enfoque en el paradigma Dividir y Conquistar. A lo largo del documento, se comparan algoritmos clásicos como Merge Sort, Quicksort y Strassen, evaluando su rendimiento en diferentes escenarios y tipos de datos, incluyendo casos semiordenados, aleatorios e invertidos. Además, se incluyen pruebas experimentales con matrices cuadradas y rectangulares, lo que permite observar el comportamiento de los algoritmos bajo diversas condiciones.

El análisis revela no solo las fortalezas y debilidades de cada algoritmo, sino también los entornos en los que pueden destacarse. Se estudia cómo cada algoritmo responde frente a variaciones en el tamaño y la disposición de los datos, lo que permite identificar en qué circunstancias son más eficientes. Por ejemplo, se explora el peor caso de Quicksort, cuando el conjunto de datos está ordenado de manera descendente, y el impacto de la transposición en la multiplicación de matrices.

Se estará trabajando con Python, para facilitar la ejecución de los códigos mediante la herramienta de Jupyter Notebook. Los resultados de los experimentos serán almacenados en DataFrames y en archivos de texto para los algoritmos de ordenamiento y para los de multiplicación de matrices, respectivamente. Los códigos, en su mayoría, son extraídos de internet, con ligeras modificaciones para poder adaptarse a las necesidades del trabajo y el manejo de los datos. En la parte superior de todos los códigos cuya autoría no poseo, está en un comentario el enlace de donde fue extraído.

https://github.com/juaquo15/Tarea01-INF221.git



II. Descripción de los algoritmos a ser comparados

II.1. Algoritmos de ordenamiento

Bubble sort funciona recorriendo el arreglo entregado un elemento a la vez y, por cada valor, lo compara con el siguiente. Si el valor actual es mayor que el siguiente, entonces hace el intercambio, de este modo recorriendo el arreglo hasta el final, tantas veces como sean necesarias.

Al recorrer el arreglo tantas veces como sea necesario implica que por cada n elementos se compare n veces, lo que nos da:

$$n * n = O(n^2)$$

- En el mejor de los casos su complejidad temporal es de $O(n)$ (el arreglo esta ordenado y solo hay que recorrerlo)
- En el caso promedio de los casos su complejidad temporal es de $O(n^2)$
- En el peor de los casos su complejidad temporal es de $O(n^2)$(el arreglo esta invertido y para cada elemento recorrido hay que hacer un intercambio).

```
#https://www.geeksforgeeks.org/python-program-for-bubble-sort/
def bubbleSort(arr):
    n = len(arr)
    # Traverse through all array elements
    for i in range(n):
        swapped = False
        # Last i elements are already in place
        for j in range(0, n-i-1):
            # Traverse the array from 0 to n-i-1
            # Swap if the element found is greater
            # than the next element
            if arr[j] > arr[j+1]:
                arr[j], arr[j+1] = arr[j+1], arr[j]
                swapped = True
        if (swapped == False):
            break
```

Insertion sort funciona separando el arreglo en dos partes: una ordenada y otra que todavía no ha sido ordenada. El algoritmo recorre la zona no ordenada y los agrega de forma ordenada en la zona ordenada.

En promedio, cada elemento del arreglo debe de ser comparado con $\frac{n}{2}$ elementos para ubicar su posicion en el arreglo de los ordenados, por lo tanto:

$$O\left(\frac{n}{2} * n\right) = O(n^2)$$

n elementos del arreglo por las $\frac{n}{2}$ comparaciones en promedio nos daria $\frac{n^2}{2}$ pero el $\frac{1}{2}$ desaparece al ser constante, entonces podemos concluir:

- En el mejor de los casos su complejidad temporal es de $O(n)$ (el arreglo esta ordenado y solo hay que recorrerlo)
- En el caso promedio de los casos su complejidad temporal es de $O(n^2)$
- En el peor de los casos su complejidad temporal es de $O(n^2)$(el arreglo esta invertido y para cada elemento recorrido hay que hacer un intercambio).

```
#https://www.geeksforgeeks.org/python-program-for-insertion-sort/
def insertionSort(arr):
  n = len(arr) # Get the length of the array
  if n <= 1:
    return # If the array has 0 or 1 element, it is already sorted, so return
  for i in range(1, n): # Iterate over the array starting from the second element
    key = arr[i] # Store the current element as the key to be inserted in the
right position
    j = i-1
    while j >= 0 and key < arr[j]: # Move elements greater than key one position
ahead
      arr[j+1] = arr[j] # Shift elements to the right
      j -= 1
    arr[j+1] = key # Insert the key in the correct position
```

Merge sort funciona separando el arreglo inicial en subarreglos más pequeños y luego juntándolos en el orden correcto. Es un algoritmo del tipo **dividir y conquistar**. La división del arreglo principal en varios subarreglos se realiza de forma recursiva hasta que el subarreglo tenga al menos un elemento. Luego, ordena los elementos de la rama de subarreglos según corresponda y continúa hasta que ya no queden subarreglos.

Su complejidad temporal es, en todos los casos:

$$\log_2(n) \rightarrow \begin{matrix}\text{es el numero de pasos} \\ \text{de la division recursiva}\end{matrix}$$

$$n \rightarrow \begin{matrix}\text{En cada uno de los logn niveles,} \\ \text{se realiza n operaciones} \\ \text{para combinar los elementos.}\end{matrix}$$

$$O(n * \log(n))$$

```
#https://www.w3schools.com/dsa/dsa_algo_mergesort.php
def merge(left, right):
    result = []
    i = j = 0
    while i < len(left) and j < len(right):
        if left[i] < right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1
    result.extend(left[i:])
```

```
    result.extend(right[j:])
    return result
def mergeSort(arr):
    step = 1  # Starting with sub-arrays of length 1
    length = len(arr)
    while step < length:
        for i in range(0, length, 2 * step):
            left = arr[i:i + step]
            right = arr[i + step:i + 2 * step]
            merged = merge(left, right)
            # Place the merged array back into the original array
            for j, val in enumerate(merged):
                arr[i + j] = val
        step *= 2  # Double the sub-array length for the next iteration
    return arr
```

Python sort es un algoritmo de ordenamiento llamado Timsort en la función .sort() de las listas. Timsort es un algoritmo híbrido que combina los conceptos de Merge Sort y Insertion Sort. Fue diseñado específicamente para ser eficiente en listas que ya están parcialmente ordenadas.

En el peor de los casos tiene una complejidad de $O(n * \log(n))$

En el caso promedio tiene una complejidad de $O(n * \log(n))$

En el mejor caso, $O(n)$ (lista ya ordenada)

```
lista.sort()
```

Quick sort toma un arreglo de elementos y elige uno como « pivote » y mueve los valores menores al pivote a la izquierda y los mayores a la derecha, luego cambia el pivote y realiza el mismo proceso de manera recursiva hasta terminar.

En el peor de los casos, cuando el pivote es el valor mas grande o el mas pequeño, la complejidad temporal es de $O(n^2)$

En el caso promedio es $O(n * \log(n))$

En el mejor caso es $O(n * \log(n))$

```
#https://www.w3schools.com/dsa/dsa_algo_quicksort.php
def partition(array, low, high):
    pivot = array[high]
    i = low - 1
    for j in range(low, high):
        if array[j] <= pivot:
            i += 1
            array[i], array[j] = array[j], array[i]
    array[i+1], array[high] = array[high], array[i+1]
    return i+1
def quicksort(array, low=0, high=None):
    if high is None:
        high = len(array) - 1
    if low < high:
```

```
        pivot_index = partition(array, low, high)
        try:
            quicksort(array, low, pivot_index-1)
        except RecursionError:
            pass

        try:
            quicksort(array, pivot_index+1, high)
        except RecursionError:
            pass
```

Nota: se tuvo que modificar el codigo para evitar los errores de recursion.

Selection sort elije el primer elemento del arreglo y recorre hasta encontrar el elemento mas pequeño, luego el mas pequeño lo envia a la primera posicion, luego con el segundo y asi, sucesivamente.

En promedio se hacen un total de $\frac{n}{2}$ comparaciones.

Siendo n elementos, entonces :

$$n * \frac{n}{2} \rightarrow O(n^2)$$

Siendo esta la complejidad en todos los casos.

```
#https://www.geeksforgeeks.org/python-program-for-selection-sort/
# Selection sort in Python
# time complexity O(n*n)
#sorting by finding min_index
def selectionSort(array):
  size = len(array)
  for ind in range(size):
    min_index = ind
    for j in range(ind + 1, size):
      # select the minimum element in every iteration
      if array[j] < array[min_index]:
        min_index = j
    # swapping the elements to sort the array
    (array[ind], array[min_index]) = (array[min_index], array[ind])
```

II.2. Algoritmos de multiplicacion de matrices

Algoritmo iterativo cúbico tradicional realiza la multiplicación mediante tres bucles anidados. El primer bucle recorre las filas de *A*, el segundo las columnas de B, y el tercero multiplica y acumula los productos correspondientes

El costo de la multiplicacion es de:

$$O(m * n * p)$$

- $m \rightarrow$ numero de filas de la matriz A
- $n \rightarrow$ numero de columnas de la matriz B
- $p \rightarrow$ numero de columnas/filas de A/B

lo cual resulta en la siguiente expresion:

$$O(m * n * p)$$

```
import time
def multiply_matrices(A, B):
    result = [[0 for _ in range(len(B[0]))] for _ in range(len(A))]
    for i in range(len(A)):
        for j in range(len(B[0])):
            for k in range(len(B)):
                result[i][j] += A[i][k] * B[k][j]
    return result
def multiply_and_time_matrices(input_filename, output_filename):
     with open(input_filename, 'r') as infile, open(output_filename, 'w') as
outfile:
        lines = infile.readlines()
        i = 0
        while i < len(lines):
            if "Matrix A" in lines[i]:
                size_A =tuple(map(int,lines[i].split()[2].strip('():').split('x')))
                A = []
                i += 1
                while lines[i].strip() != "":
                    A.append(list(map(int, lines[i].split())))
                    i += 1
                i += 2
                size_B =tuple(map(int,lines[i].split()[2].strip('():').split('x')))
                B = []
                i += 1
                while i < len(lines) and lines[i].strip() != "":
                    B.append(list(map(int, lines[i].split())))
                    i += 1
                i += 2
                start_time = time.time()
                result = multiply_matrices(A, B)
                end_time = time.time()
                elapsed_time = end_time - start_time
                outfile.write(f"Order of Matrices: A{size_A} x B{size_B}\n")
```

```
                outfile.write(f"Time taken: {elapsed_time:.6f} seconds\n\n")
multiply_and_time_matrices("dataset_matrices.txt", "multiplication_times.txt")
```

Algoritmo iterativo cúbico optimizado para mantener la localidad de los datos (transponiendo la segunda matriz)

El algoritmo posee su complejidad dividida en dos partes: la transposicion y la multiplicacion. En la transposicion el costo es de:

$$O(n * p)$$

Por otro lado (despues de transponer) el costo de la multiplicacion es de:

$$O(m * n * p)$$

- $m \rightarrow$ numero de filas de la matriz A
- $n \rightarrow$ numero de columnas de la matriz B
- $p \rightarrow$ numero de columnas/filas de A/B

lo cual resulta en la siguiente expresion:

$$O(m * n * p) + O(n * p)$$

$$\therefore O(m * n * p)$$

Este algoritmo es mejor que el anterior no tanto por la complejidad, sino que, aprovecha como esta constituido el hardware, ya que al estar toda la columna en una lista se almacena en la cache y por lo tanto no demora tanto en ir a buscar los datos.

```
import time
def transpose_matrix(matrix):
    rows, cols = len(matrix), len(matrix[0])
    transposed = [[0 for _ in range(rows)] for _ in range(cols)]
    for i in range(rows):
        for j in range(cols):
            transposed[j][i] = matrix[i][j]
    return transposed
def multiply_matrices_with_transpose(A, B):
    B_T = transpose_matrix(B)
    result = [[0 for _ in range(len(B_T))] for _ in range(len(A))]
    for i in range(len(A)):
        for j in range(len(B_T)):
            for k in range(len(A[0])):
                result[i][j] += A[i][k] * B_T[j][k]
    return result
defmultiply_and_time_matrices_with_transpose(input_filename, output_filename):
     with open(input_filename, 'r') as infile, open(output_filename, 'w') as
outfile:
        lines = infile.readlines()
        i = 0
        while i < len(lines):
            if "Matrix A" in lines[i]:
              size_A = tuple(map(int, lines[i].split()[2].strip('():').split('x')))
```

```
                A = []
                i += 1
                while lines[i].strip() != "":
                    A.append(list(map(int, lines[i].split())))
                    i += 1
                i += 2
                size_B = tuple(map(int,lines[i].split()[2].strip('():').split('x')))
                B = []
                i += 1
                while i < len(lines) and lines[i].strip() != "":
                    B.append(list(map(int, lines[i].split())))
                    i += 1
                i += 2
                start_time = time.time()
                result = multiply_matrices_with_transpose(A, B)
                end_time = time.time()
                elapsed_time = end_time - start_time
                outfile.write(f"Order of Matrices (with Transpose): A{size_A} x
B{size_B}\n")
                outfile.write(f"Time taken: {elapsed_time:.6f} seconds\n\n")
multiply_and_time_matrices_with_transpose("dataset_matrices.txt",
"multiplication_times_transpose.txt")
```

Algoritmo de Strassen

El algoritmo Strassen tiene una complejidad de:

$$O(n^{\log\{2\}7}) \approx O(n^{2.81})$$

Esto lo podemos deducir de aplicar el teorema maestro.

$$T(n) = 7 * T\left(\frac{n}{2}\right) + O(n^2)$$

- $a = 7$ numero de subproblemas generados apartir del algoritmo.
- $b = 2$ factor de reduccion de los subproblemas.
- $d = 2$ complejidad de la multiplicacion de las matrices fuera de recursion.

Sin embargo, dado que requiere dividir las matrices en submatrices y hacer múltiples llamadas recursivas, el algoritmo es más eficiente que la multiplicación estándar solo para matrices grandes. Para matrices pequeñas, el algoritmo clásico de multiplicación puede ser más rápido debido a la sobrecarga recursiva de Strassen.

```
#https://www.geeksforgeeks.org/strassens-matrix-multiplication-algorithm-
implementation/
import numpy as np
import time
def strassen(A, B):
    n = A.shape[0]
    # base case: 1x1 matrix
    if n == 1:
        return A * B
    else:
```

```
        # split input matrices into quarters
        mid = n // 2
        A11, A12, A21, A22 = A[:mid, :mid], A[:mid, mid:], A[mid:, :mid], A[mid:,
mid:]
        B11, B12, B21, B22 = B[:mid, :mid], B[:mid, mid:], B[mid:, :mid], B[mid:,
mid:]
        # calculate p1 to p7
        P1 = strassen(A11 + A22, B11 + B22)
        P2 = strassen(A21 + A22, B11)
        P3 = strassen(A11, B12 - B22)
        P4 = strassen(A22, B21 - B11)
        P5 = strassen(A11 + A12, B22)
        P6 = strassen(A21 - A11, B11 + B12)
        P7 = strassen(A12 - A22, B21 + B22)
        # calculate the 4 quarters of the resulting matrix
        C11 = P1 + P4 - P5 + P7
        C12 = P3 + P5
        C21 = P2 + P4
        C22 = P1 + P3 - P2 + P6
        # combine the 4 quarters into a single result matrix
        C = np.vstack((np.hstack((C11, C12)), np.hstack((C21, C22))))
        return C
def read_matrix_from_file(file, rows, cols):
    matrix = []
    for _ in range(rows):
        line = file.readline().strip()
        row = list(map(int, line.split()))
        matrix.append(row)
    return np.array(matrix)
def main():
    output_file = "multiplication_times_strassen.txt"
    with open(output_file, "w") as time_file:
        # Open the dataset file
        with open("dataset_matrices.txt", "r") as f:
            while True:
                line = f.readline()
                if not line:
                    break
                if "Matrix A" in line:
                    # Parse matrix A dimensions
                    dimensions = line.strip().split("(")[1].split(")")[0]
                    rows_A, cols_A = map(int, dimensions.split("x"))
                    # Read matrix A
                    A = read_matrix_from_file(f, rows_A, cols_A)
                    f.readline()  # Skip empty line
                if "Matrix B" in line:
                    # Parse matrix B dimensions
                    dimensions = line.strip().split("(")[1].split(")")[0]
                    rows_B, cols_B = map(int, dimensions.split("x"))
                    # Read matrix B
                    B = read_matrix_from_file(f, rows_B, cols_B)
                    f.readline()  # Skip empty line
```

```
                    # Check if matrices need padding
                    max_size = max(A.shape[0], A.shape[1], B.shape[0], B.shape[1])
                    m = 1
                    while m < max_size:
                        m *= 2
                    A = np.pad(A,((0,m-A.shape[0]),(0,m-A.shape[1])),mode='constant')
                    B = np.pad(B,((0,m-B.shape[0]),(0,m-B.shape[1])),mode='constant')
                    # Perform Strassen's algorithm on matrices A and B
                    start_time = time.time()
                    C = strassen(A, B)
                    end_time = time.time()
                    # Calculate the execution time
                    exec_time = end_time - start_time
                    # Write the execution time to the file
                    time_file.write(f"Order of Matrices: A({rows_A}x{cols_A}) x
B({rows_B}x{cols_B})\n")
                    time_file.write(f"Time taken: {exec_time:.6f} seconds\n\n")
if __name__ == "__main__":
    main()
```

III. Datasets

Codigo generador de dataset para algoritmos de ordenamiento

```
import random
import time
import pandas as pd
import matplotlib.pyplot as plt
def generar_dataset(tamano, tipo, nombre_archivo):
    if tipo == 'aleatorio':
        datos = [random.randint(1, 10000) for _ in range(tamano)]
    elif tipo == 'semi_ordenado':
          datos = sorted([random.randint(1, 10000) for _ in range(tamano)])
[:int(0.7*tamano)]
        datos += [random.randint(1, 10000) for _ in range(tamano - len(datos))]
    elif tipo == 'invertido':
          datos = sorted([random.randint(1, 10000) for _ in range(tamano)],
reverse=True)
    elif tipo == 'parcialmente_ordenado':
        datos = sorted([random.randint(1, 10000) for _ in range(tamano//2)])
        datos += [random.randint(1, 10000) for _ in range(tamano//2)]
    elif tipo == 'ordenado':
      datos = sorted([random.randint(1, 10000) for _ in range(tamano)])
    with open(nombre_archivo, 'w') as file:
        file.write(" ".join(map(str, datos)))
def measure_time(sort_function, data, nombre = ''):
    start_time = time.time()
    sort_function(data)
    end_time = time.time()
    elapsed_time = end_time - start_time
    return elapsed_time
df = pd.DataFrame(columns=['algoritmo','orden', 'tipo', 'tiempo'])
# Generar un dataset de 1000 elementos aleatorios
orden = 5 #-1
for i in range(1,orden):
  generar_dataset(10**i, 'ordenado', 'dataset'+str(i)+'O.txt')
  generar_dataset(10**i, 'aleatorio', 'dataset'+str(i)+'A.txt')
  generar_dataset(10**i, 'semi_ordenado', 'dataset'+str(i)+'S.txt')
  generar_dataset(10**i, 'invertido', 'dataset'+str(i)+'I.txt')
  generar_dataset(10**i, 'parcialmente_ordenado', 'dataset'+str(i)+'P.txt')
```

El código anterior fue utilizado para generar datasets que serían empleados en la ejecución y análisis de diferentes algoritmos de ordenamiento de arreglos, así como para la posterior generación de gráficos que visualizan su rendimiento.

Tamaño de los datasets: Los archivos generados pueden contener desde 10 hasta 10,000 elementos. Cada elemento es un número entero dentro del rango de 1 a 10,000.

Categorías de datasets:

1. Aleatorios: Los elementos se generan sin ningún orden particular.
2. Parcialmente ordenados: El 50% de los elementos están ordenados y el otro 50% está desordenado.

3. Semi ordenados: El 70% de los elementos están ordenados y el resto está desordenado.
4. Invertidos: Los elementos están generados de manera descendente.
5. Ordenados: Los datos están ordenados.

Los resultados de la ejecución de los algoritmos de ordenamiento sobre estos datasets se almacenan en dataframes de pandas para su posterior análisis.

La idea detras de los tipos de dataset es emular el comportamiento de los algoritmos en los diferentes ambientes de uso. Sobre todo ver como se comportan los algoritmos en los casos en los que esta invertido, el cual es un recurrente peor caso.

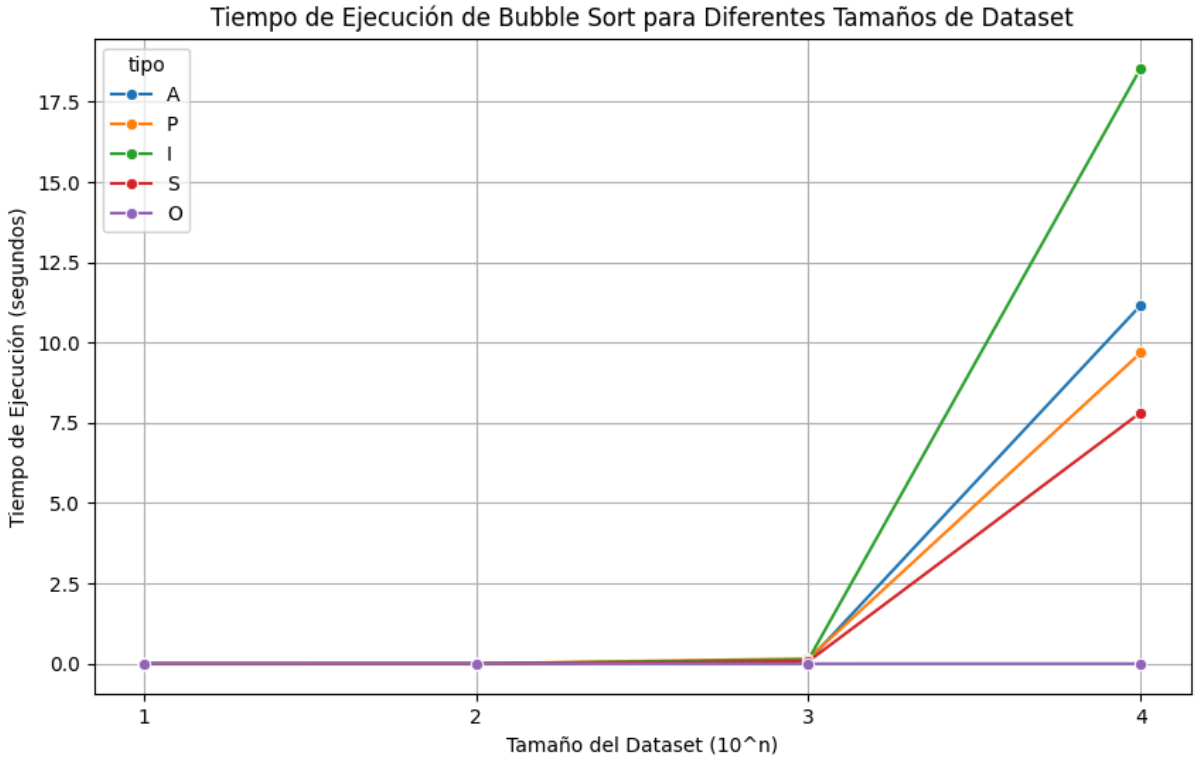
Codigo generador de dataset para algoritmos de multiplicacion de matrices

```
import numpy as np
def generate_matrices_no_multiplication(filename, sizes):
    with open(filename, 'w') as f:
        for size in sizes:
            A = np.random.randint(1, 100, (size, size))
            f.write(f"Matrix A ({size}x{size}):\n")
            np.savetxt(f, A, fmt='%d')
            f.write("\n\n")
            B = np.random.randint(1, 100, (size, size))
            f.write(f"Matrix B ({size}x{size}):\n")
            np.savetxt(f, B, fmt='%d')
            f.write("\n\n")
            if size > 1:
                A = np.random.randint(1, 100, (size, size-50))
                f.write(f"Matrix A ({size}x{size-50}):\n")
                np.savetxt(f, A, fmt='%d')
                f.write("\n\n")
                B = np.random.randint(1, 100, (size-50, size))
                f.write(f"Matrix B ({size-50}x{size}):\n")
                np.savetxt(f, B, fmt='%d')
                f.write("\n\n")
sizes = [10*(i*10) for i in range(1, 8)] #8 para traspuesta y normal, 2 para
strassen :skull:
generate_matrices_no_multiplication("dataset_matrices.txt", sizes)
```

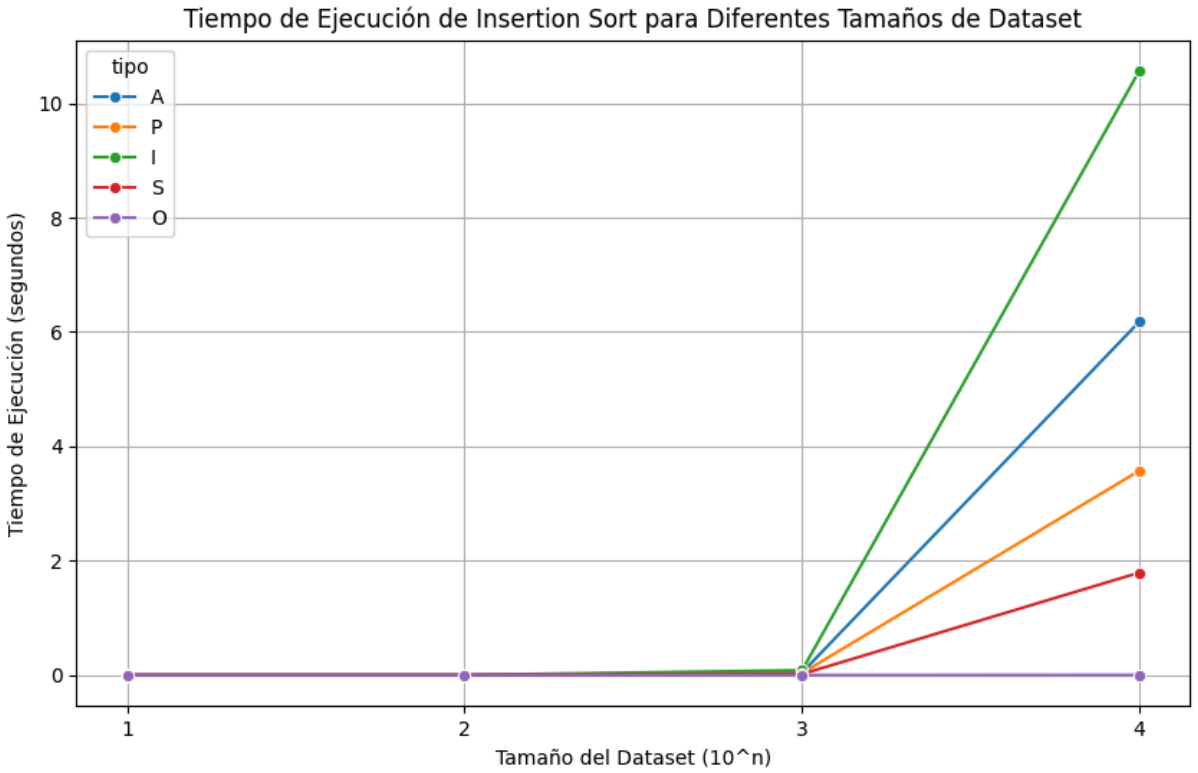
El código anterior genera un archivo .txt en el cual se guardan matrices generadas de manera aleatoria. Se generan matrices de 100x100, 100x50, 200x200, 200x150, etc.

La idea detrás de la generación de estas matrices es probar el algoritmo en casos de matrices cuadradas y rectangulares, de hasta 700x700 en los primeros dos casos y de hasta 400x400 en Strassen (ya que este último demora demasiado en ejecutarse).

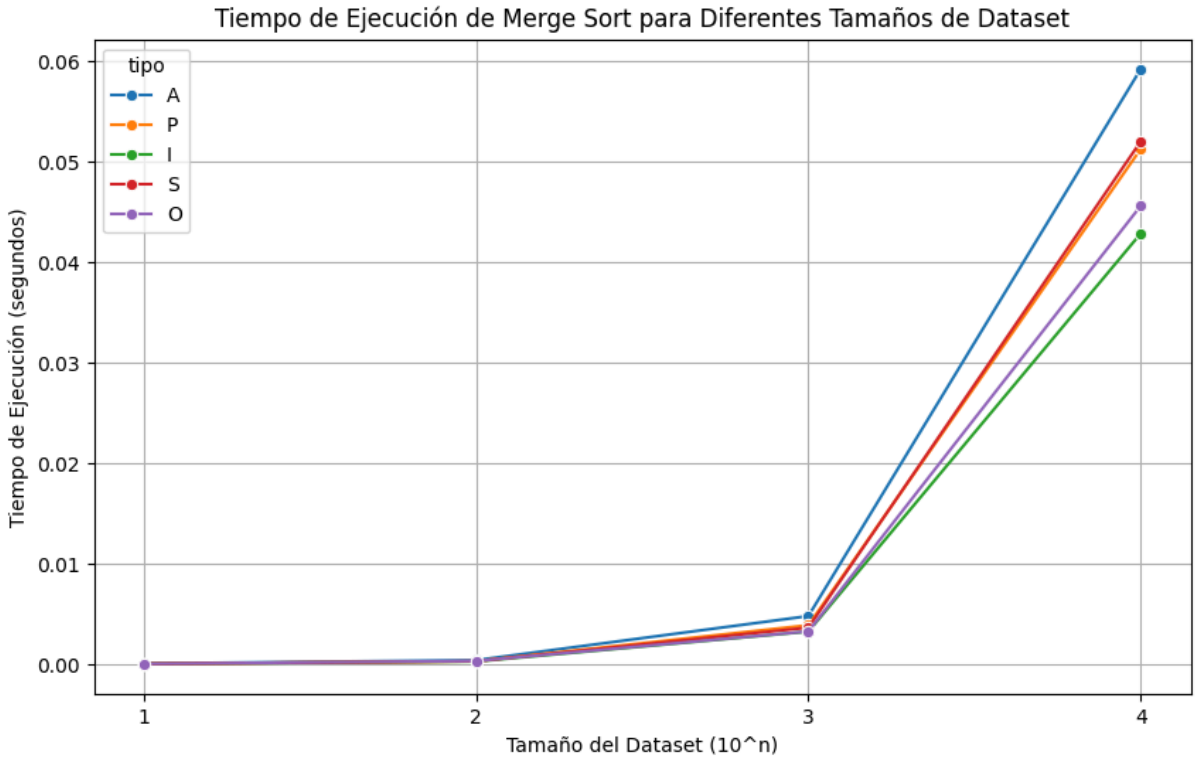
IV. Resultados Experimentales



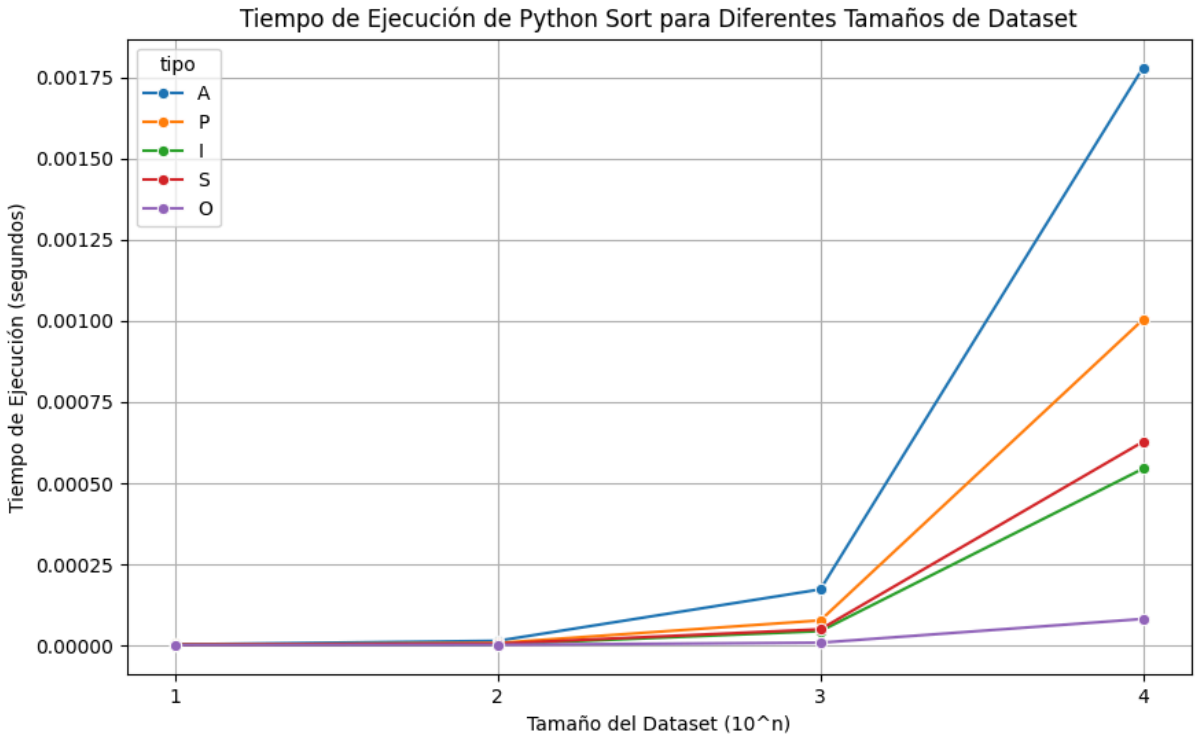
El gráfico permite visualizar tanto el mejor caso, donde el arreglo ya está ordenado, como el peor caso, donde está ordenado de manera invertida. Podemos observar que el crecimiento de los otros casos, aunque en menor grado, sigue la tendencia del caso invertido. Esto se debe a que la complejidad del peor caso y del caso promedio es la misma: $O(n^2)$.



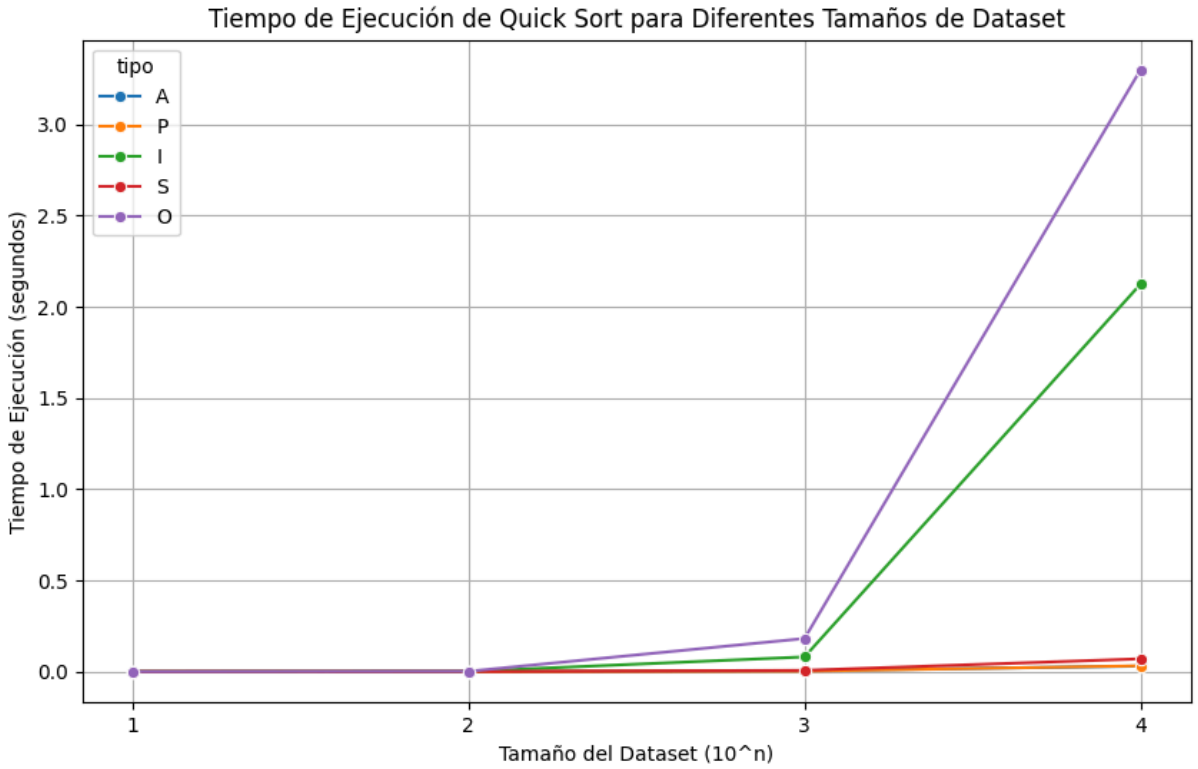
En este gráfico, además de visualizar el evidente peor caso (n permutaciones por cada n elementos al estar generados de manera invertida), también podemos observar el mejor caso, los datos ya están ordenados, y los intermedios, donde la mayoria lo está, lo que reduce el número de permutaciones necesarias.



El gráfico evidencia cómo en merge sort los tiempos de ejecución son similares, lo que concuerda con lo planteado previamente, ya que merge sort siempre realiza el mismo número de operaciones, independientemente del orden de la lista.

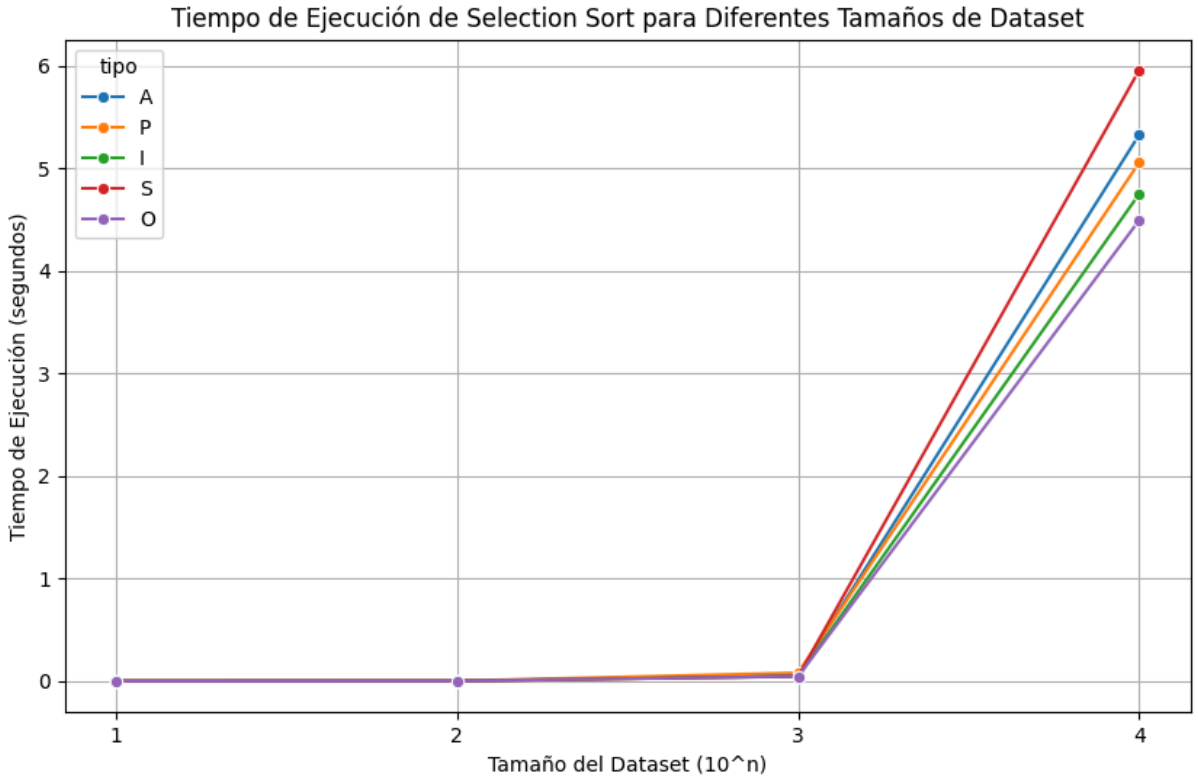


Al estudiar el gráfico, lo primero que podemos notar son los bajos tiempos en todos los casos, siendo este el mejor algoritmo analizado en este informe. También podemos observar, como mencionamos anteriormente, que Python sort es especialmente eficiente con arreglos que presentan algún tipo de orden o patrón.

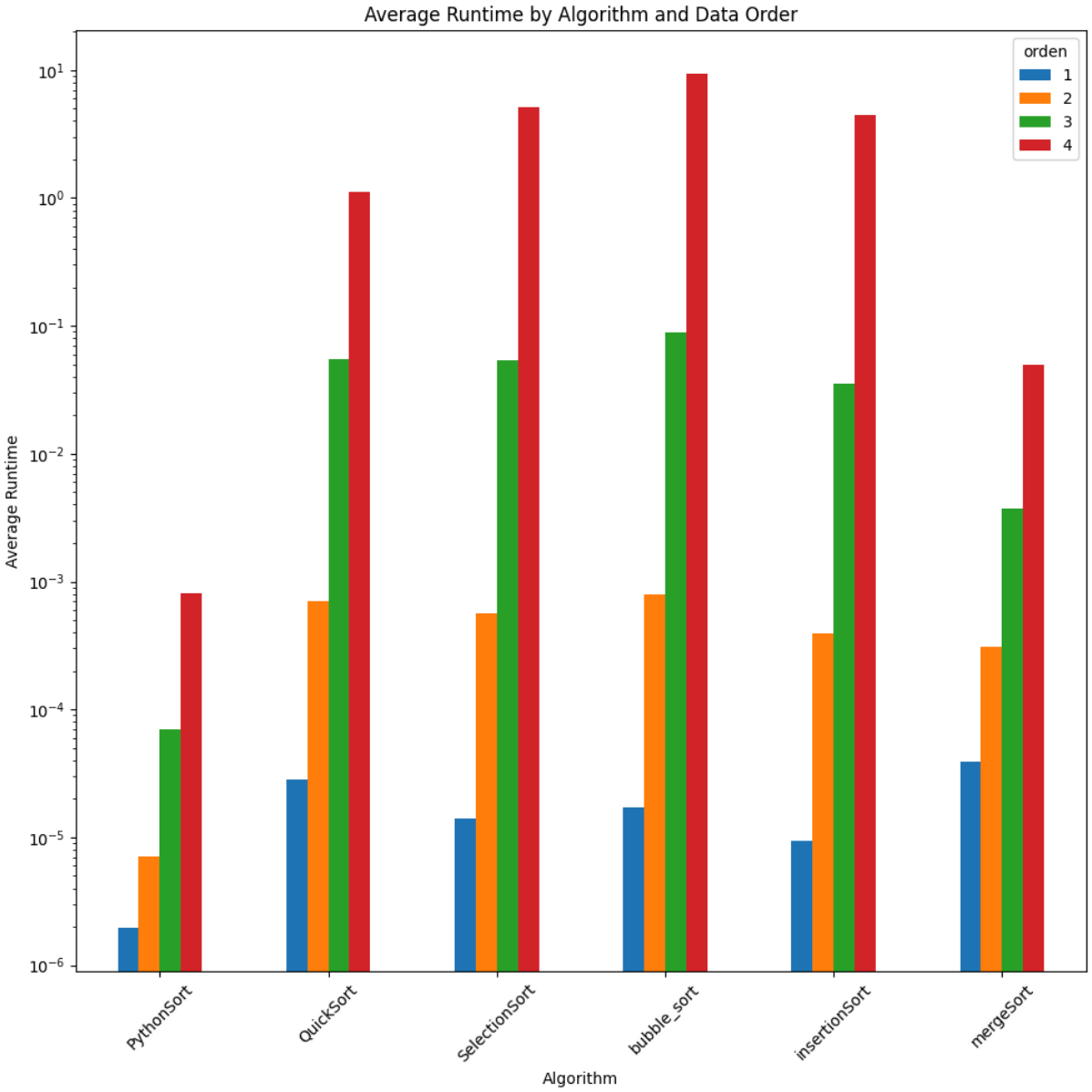


Queda claramente evidenciado el peor caso, ordenado de manera ascendente o descendente. Como quicksort elige el primer elemento del arreglo como pivote, en este caso, todos

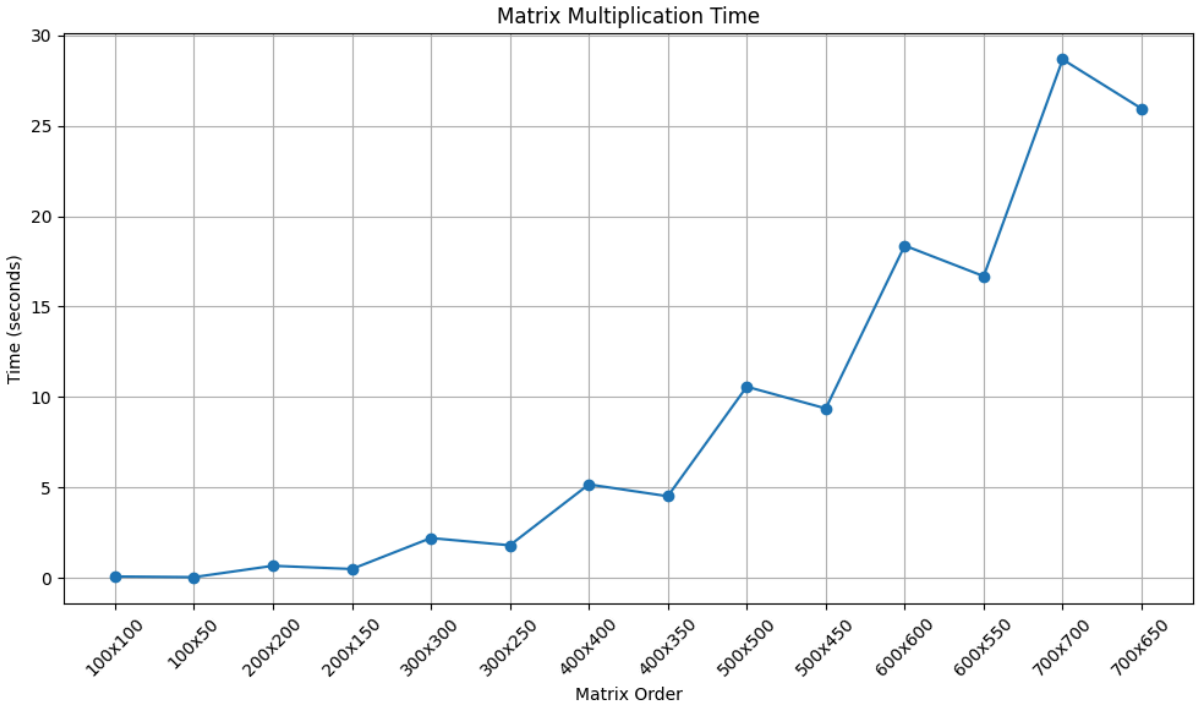
los elementos menores se ingresan a la izquierda (o mayores a la derecha). Los otros casos no son distinguibles entre ellos, ya que son similares.



Al tener la misma complejidad, se realiza un numero similar de pasos en el ordenamiento de los diferentes datasets y por lo tanto, tienen un tiempo similar.

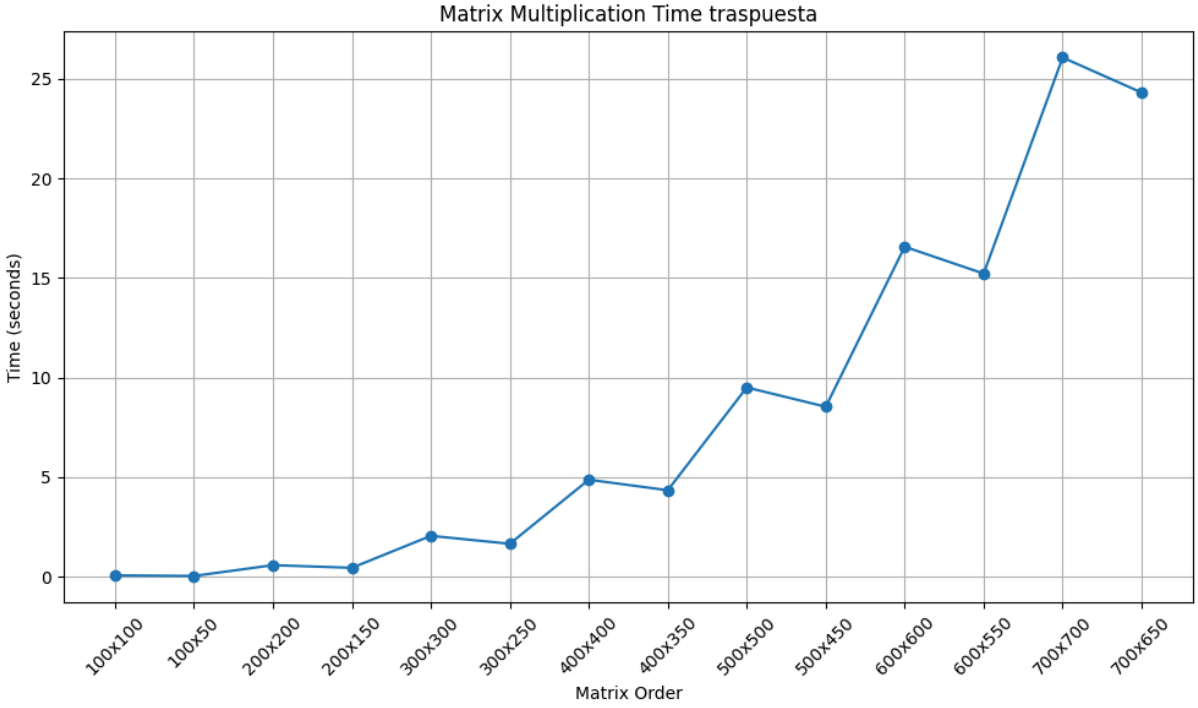


Podemos concluir que si estamos trabajando con python, en cualquier orden de magnitud, el algoritmo de la libreria estandar de python: **sort()** es por lejos el mejor algoritmo de ordenamiento.



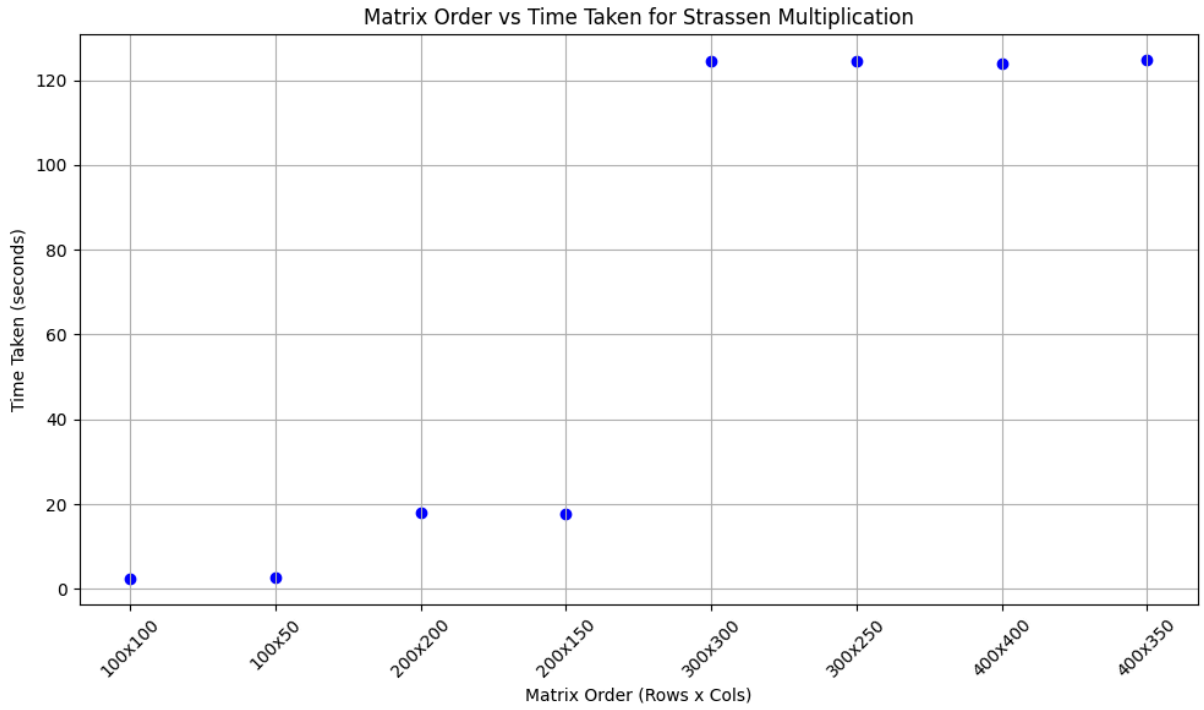

```
Order of Matrices: A(100, 100) x B(100, 100)
Time taken: 0.074192 seconds

Order of Matrices: A(100, 50) x B(50, 100)
Time taken: 0.037002 seconds

Order of Matrices: A(200, 200) x B(200, 200)
Time taken: 0.664860 seconds

Order of Matrices: A(200, 150) x B(150, 200)
Time taken: 0.487147 seconds

Order of Matrices: A(300, 300) x B(300, 300)
Time taken: 2.199622 seconds

Order of Matrices: A(300, 250) x B(250, 300)
Time taken: 1.797446 seconds

Order of Matrices: A(400, 400) x B(400, 400)
Time taken: 5.165237 seconds

Order of Matrices: A(400, 350) x B(350, 400)
Time taken: 4.512139 seconds

Order of Matrices: A(500, 500) x B(500, 500)
Time taken: 10.568744 seconds

Order of Matrices: A(500, 450) x B(450, 500)
Time taken: 9.365395 seconds

Order of Matrices: A(600, 600) x B(600, 600)
```

```
Time taken: 18.371070 seconds

Order of Matrices: A(600, 550) x B(550, 600)
Time taken: 16.678059 seconds

Order of Matrices: A(700, 700) x B(700, 700)
Time taken: 28.663327 seconds

Order of Matrices: A(700, 650) x B(650, 700)
Time taken: 25.941370 seconds
```




```
Order of Matrices (with Transpose): A(100, 100) x B(100, 100)
Time taken: 0.074005 seconds

Order of Matrices (with Transpose): A(100, 50) x B(50, 100)
Time taken: 0.044001 seconds

Order of Matrices (with Transpose): A(200, 200) x B(200, 200)
Time taken: 0.585997 seconds

Order of Matrices (with Transpose): A(200, 150) x B(150, 200)
Time taken: 0.452670 seconds

Order of Matrices (with Transpose): A(300, 300) x B(300, 300)
Time taken: 2.058176 seconds

Order of Matrices (with Transpose): A(300, 250) x B(250, 300)
Time taken: 1.659084 seconds

Order of Matrices (with Transpose): A(400, 400) x B(400, 400)
Time taken: 4.874554 seconds
```

```
Order of Matrices (with Transpose): A(400, 350) x B(350, 400)
Time taken: 4.346737 seconds

Order of Matrices (with Transpose): A(500, 500) x B(500, 500)
Time taken: 9.505733 seconds

Order of Matrices (with Transpose): A(500, 450) x B(450, 500)
Time taken: 8.537140 seconds

Order of Matrices (with Transpose): A(600, 600) x B(600, 600)
Time taken: 16.569030 seconds

Order of Matrices (with Transpose): A(600, 550) x B(550, 600)
Time taken: 15.219381 seconds

Order of Matrices (with Transpose): A(700, 700) x B(700, 700)
Time taken: 26.064738 seconds

Order of Matrices (with Transpose): A(700, 650) x B(650, 700)
Time taken: 24.306307 seconds
```




```
Order of Matrices: A(100x100) x B(100x100)
Time taken: 2.526360 seconds

Order of Matrices: A(100x50) x B(50x100)
Time taken: 2.620964 seconds

Order of Matrices: A(200x200) x B(200x200)
Time taken: 17.913724 seconds
```

```
Order of Matrices: A(200x150) x B(150x200)
Time taken: 17.669644 seconds

Order of Matrices: A(300x300) x B(300x300)
Time taken: 124.416624 seconds

Order of Matrices: A(300x250) x B(250x300)
Time taken: 124.555813 seconds

Order of Matrices: A(400x400) x B(400x400)
Time taken: 124.007385 seconds

Order of Matrices: A(400x350) x B(350x400)
Time taken: 124.657517 seconds
```

Comentario general: En los dos primeros métodos podemos ver cómo sube el tiempo a medida que aumenta la dimensión de la matriz. También podemos apreciar la disminución del tiempo de ejecución al transponer la segunda matriz, aunque no es tan significativo, al menos en este experimento. Nos damos cuenta de que el algoritmo de Strassen **no es** mejor debido a sus características. Sus ventajas podrían apreciarse en escenarios más grandes, los cuales no se han testeado, ya que la prueba aún estaría en ejecución…

V. Conclusiones

El análisis realizado sobre los algoritmos de ordenamiento y multiplicación de matrices expone la importancia de seleccionar el método adecuado según las características del problema y los datos involucrados. Algoritmos como Merge Sort y Quicksort, aunque eficientes en la mayoría de los casos, muestran limitaciones en escenarios específicos. Por ejemplo, Quicksort, a pesar de su complejidad promedio de $O(n * \log n)$, experimenta una notable pérdida de rendimiento en su peor caso, cuando los datos están ordenados, lo que lo convierte en un algoritmo menos adecuado en estos contextos. Merge Sort, por su parte, mantiene un rendimiento constante debido a su enfoque basado en dividir y conquistar, pero puede ser menos eficiente en términos de uso de memoria comparado con otros métodos.

En el caso de la multiplicación de matrices, el algoritmo de Strassen, aunque teóricamente más rápido que el método cúbico estándar para matrices grandes, resulta menos eficiente para matrices pequeñas debido a la sobrecarga recursiva que implica su implementación. Esto refleja que, si bien el rendimiento teórico es tentador, en la práctica el tamaño y la estructura de los datos son mas determinantes en la elección del algoritmo más apropiado.

En conclusión, se destaca la necesidad de considerar no solo la complejidad asintótica de los algoritmos, sino también el contexto en el que se implementan. Factores como el tamaño del dataset, el grado de orden de los datos y el tipo de estructura involucrada deben guiar la selección del algoritmo más adecuado. Esta conclusión refuerza la idea de que la eficiencia de los algoritmos no depende solo de su formulación teórica, sino de cómo responden a las condiciones reales de los problemas que intentan resolver.