Universidad Técnica Federico Santa María
Campus San Joaquín
Departamento de informática
Segundo semestre 2024

Tarea 1:Algoritmos dividir y conquistar

Walter Sanhueza Neira, 202023564-3, walter.sanhueza@usm.cl , Paralelo 200

.

1. Introducción

Existe un gran interés en el ordenamiento de datos, ya que permiten reducir el tiempo de búsqueda u organización de los mismos, los cuales, en la mayoría de los casos, son grandes volúmenes de información, como sucede en servicios como YouTube, entre otros. Por otro lado, la multiplicación de matrices es de gran interés en la transformación de imágenes, objetos 3D y animación, pues estas requieren operaciones matriciales.

Para ahondar en este tema, en el presente informe se trabajará con 4 algoritmos de ordenamiento (Selection Sort, Merge Sort, Quick Sort, y std::sort) y 3 de multiplicación de matrices (Tradicional, Optimizado y Strassen) con el fin de comparar sus rendimientos y las variables que afectan estos mismos. Se hablará sobre sus características, cómo se definen los casos de prueba y por qué se utilizarán estos. Además, se abordarán sus tiempos algorítmicos (en el peor, mejor y caso promedio) con el objetivo de predecir su desempeño real y verificar si este se corresponde con el tiempo de ejecución. También se explicará en qué casos conviene utilizar uno sobre otro y que tanto impacto pueden llegar a ser características asociadas a la implementación del algoritmo o la localidad de los datos.

2. Descripción de algoritmos a ser comparados

El trabajo de dividió en 2 grupos de algoritmos, los de ordenamiento de arreglos y los de multiplicación de matrices. Notar que puede haber pequeñas modificaciones respecto al código de referencia (de la página GeekForGeek) con el utilizado debido a que este último usa vectores a diferencia listas, presentes en algunos de los códigos, lo cual fue modificado.

2.1 –Algoritmos de ordenamiento:

1. **Selection Sort (Algoritmos cuadrático de ordenamiento)**: Los algoritmos de ordenamiento cuadrático son fáciles de implementar en comparación otros que presentes en el siguiente informe, pero para arreglos grandes es muy notoria su poca eficacia debido a su complejidad temporal. Particularmente el algoritmo selection sort elige el elemento más pequeño de cada iteración y lo pone al final.
   - **Peor caso (O)**: $O(n^2)$
   - **Mejor caso (Ω)**: $O(n^2)$
   - **Caso promedio (Θ)**: $O(n^2)$
2. **Merge sort:** Algoritmo divide y vencerás que, divide recursivamente el vector en partes aún más pequeñas (hasta quedar con 1 elemento), ordena y combina.
   - **Peor caso (O)**: $O(\text{nlog}\, n)$
   - **Mejor caso (Ω)**: $O(\text{nlog}\, n)$
   - **Caso promedio (Θ)**: $O(\text{nlog}\, n)$

3. **Quick sort:** Algoritmo divide y vencerás que toma un pivote, y entorno a él reordena los elemento repitiendo el proceso de manera recursiva. Si selecciona el peor pivote, este es el mayor o menor número del arreglo tendrá el tiempo más lento.
   - **Peor caso (O)**: $O(n^2)$
   - **Mejor caso (Ω)**: $O(\text{nlog}\, n)$
   - **Caso promedio (Θ)**: $O(\text{nlog}\, n)$

4. **Función std::sort:** Función que particularmente en C++ utiliza algoritmos Intro sort (Tim sort en Python y Java), osea una combinación de los algoritmos: insertion sort (en arreglos pequeños), quickt sort (en arreglos grandes) y heap sort(cuando quick sort sobrepasa un tiempo límite), de forma que elige el mejor de estos según sea el caso. Los tiempos presentados son para el caso de Intro sort:
   - **Peor caso(O)**: $O(\text{nlog}\, n)$
   - **Mejor caso (Ω)**: $O(\text{nlog}\, n)$
   - **Caso promedio (Θ)**: $O(\text{nlog}\, n)$

2.2–Algoritmos de multiplicación de matrices:

Debido a que las complejidades temporales de estos algoritmos dependen más de la dimensión de las matrices que de los valores específicos en ellas. Daremos el tiempo en términos generales.

5. **Tradicional:** Realiza 3 bucles for anidados para recorrer todos los elementos de la matriz y en cada elemento del matriz producto realiza la suma correspondiente.
    - **General (O)**: $O(n^3)$

6. **Optimizado:** Similar al tradicional, pero mejorado en el acceso de memoria trasponiendo la segunda matriz, mejorando la localidad espacial.
    - **General (O)**: $O(n^3)$

7. **Strassen:** Este reduce el número de multiplicaciones agregando números en la suma, utilizando un enfoque recursivo. Haciendo una multiplicación menos (8 en los otros 2 métodos).
    - **General (O)**: $O(n^{2,81})$

Las referencias utilizadas sobre los algoritmos tanto de ordenamiento como de multiplicación de matrices, y todo el código utilizo en la fabricación de este informe se puede encontrar en la sección de Bibliografía.

3. Descripción de datasets

Tanto para los algoritmos de ordenamiento como los de multiplicación de matrices se crea un dataset a través de un código (dataset.cpp), que escribe el conjunto estudiado en un archivo de texto (test_cases.txt). Este código luego será leído para poder ser testeado por el algoritmo correspondiente.
Para la implementación se los algoritmos se ha preferido utilizar la librería vector, pues otorga una mayor flexibilidad, además de funciones como size() que permitieron escribir código de manera más sencilla.
Para la toma del tiempo se utilizó la librería chronos, que nos permite tomar el tiempo como la diferencia de una variable que tiene el tiempo inicial (inicio) y otra con el final (end). Se cada caso de ambos grupos de algoritmos se hizo itero más de una vez cada caso (10 veces los algoritmos de ordenamiento y 5 los de multiplicación de matrices) asignando la suma de esto a una variable llamada total_time que después sería dividida por el total de iteraciones, dándonos un tiempo promedio más representativo.
A continuación, se hablará de cómo se compone el dataset según el problema:

3.1 –Algoritmos de ordenamiento:

Son un total de 25 casos, dividido en 5 grupos (ordenado, 25% desordenado, 50% desordenado, 75% desordenado, 100% desordenado) separados por el salto de línea en el archivo de texto. Estos van desde el tamaño 10.000 hasta el 50.000, donde en cada caso se obtiene un promedio de 10 iteraciones. Para la creación se del caso se utiliza como referencia el grupo de datos ordenados. Así utilizamos el siguiente código que nos permite tomar parte del vector y hacer que genere un numero aleatorio que va desde 1 hasta el largo del arreglo.

```
chrono::duration<double> total_time(0);
for (int j = 0; j < 5; j++) {
    auto start = chrono::high_resolution_clock::now();
    result = multiply_matrix(mat1, mat2);
    auto end = chrono::high_resolution_clock::now();
    total_time += end - start;
}
```

Figura 1: Extracto de código que muestra cómo se calculó el promedio. Notar que para este caso se usó un j<5 (caso para matrices) y que para el otro grupo se usó un j<10.

Entrada: cada línea del archivo test_cases.txt es uno de los casos (25 líneas de texto, 25 casos), separados por el salto de línea, lo que permite al código determinar si está leyendo otro caso. Estos se almacenan en un variable vector de vectores.

Salida: al momento de ejecutar el código se muestra por consola el caso a ordenar y debajo el caso ordenado (para las 2 listas se muestra los 10 primeros y 10 últimos números) con su respectivo tiempo promedio, calculado de promediar los tiempos que demora el algoritmo un total de 10 veces cada caso.

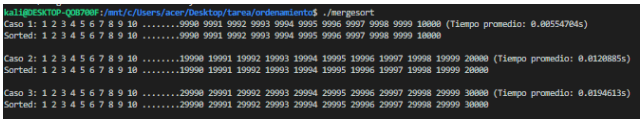


Figura 2: Muestra el output esperado.

3.2 –Algoritmos de multiplicación de matrices:

Para los casos de prueba se decidió por elegir matrices cuadradas cuyo tamaño sea potencia de 2. Esto debido a que el algoritmo de Strassen no calcula bien la matriz cuadrada de otros tamaños (y rectangulares), ingresando valores incorrectos o rellenando con 0. Los números dentro de las matrices se generarán de forma aleatorio entre un rango de 1-100

Entrada: se guarda un número correspondiente al tamaño de la matriz cuadrada y luego de un salto de línea se encuentra la matriz A y B (separadas por un salto de línea).
-salida: esta se muestra por consola al momento de ejecutar el código:

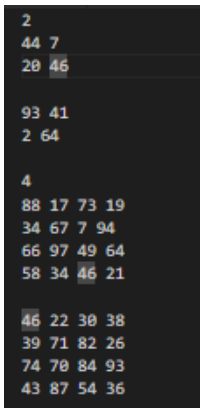


Figura 3: Muestra cómo queda el formato de los casos en el archivo test_cases.txt

Salida: existe una función que permite ver la matriz resultante en consola, pero debido a que resulta incómodo el formato se mantiene comentada, de tal forma que al que al borrar los "//" vuelve a funcionar.

4. Resultados experimentales

Para la experiencia se debe tomar en cuenta que se utilizó un computador con las siguientes características:

- **Procesador:** Intel(R) Core(TM) i5-7200U CPU @ 2.50GHz 2.70 GHz
- **Ram:** 16 GB
- **Memoria:** 500 GB
- **Gpu:** GeForce 920MX
- **Sistema operativo:** Microsoft Windows 10 Home Single Language

A la hora de ejecutar el código dentro del entorno de desarrollo de Visual Studio Code se dejó a la computadora sola hasta terminar habiendo ejecutado el programa anteriormente.

Debe considerar, además que para la visualización de los datos se prefirió una escala logarítmica. Sobre el código cabe aclarar que en el caso de quicksort se ha optado por una implementación que elige como pivote el *último elemento* del vector A continuación, se presentan los resultados de los experimentos:

4.1–Algoritmos de ordenamiento:

- ***Vectores ordenados:***

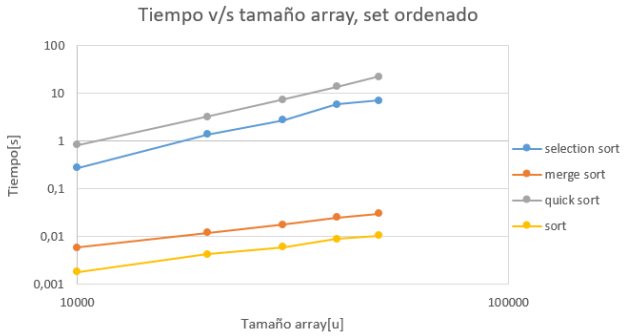


Figura 4: grafico de la tabla de elementos ordenados.

En el caso del set ordenado se observa quick sort es el más lento, seguido de selection sort, Merge sort y sort. Existe si una notoria diferencia entre selection sort, quick sort y mergesort y sort.

- ***Vectores 25% desordenados:***

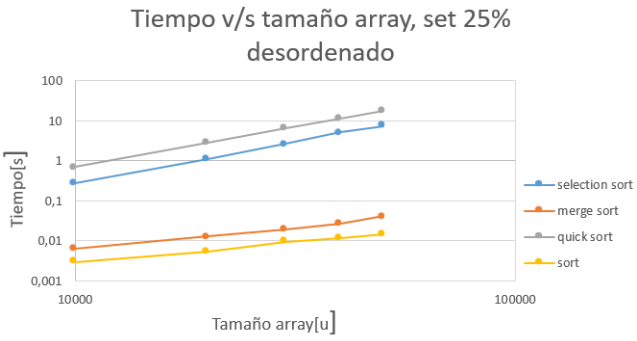


Figura 5: grafico de la tabla de elementos desordenados un 25%.

Para el caso de 25% desordenado podemos notar que el orden es muy similar al caso anterior, hasta el punto de que no existe ningún cambio respecto al orden anteior.

- ***Vectores 50% desordenados:***

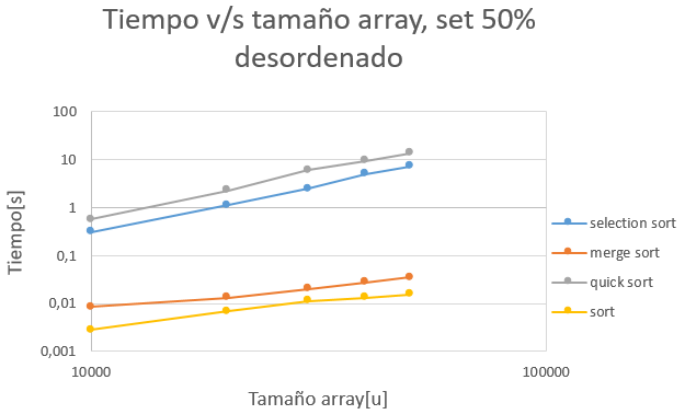


Figura 6: grafico de la tabla desordenados un 50%.

En el 50% se mantiene la misma tendencia que los graficos anteriores, pero con una leve mejoría en rendimiento por parte de quicksort.

- ***Vectores 75% desordenados:***

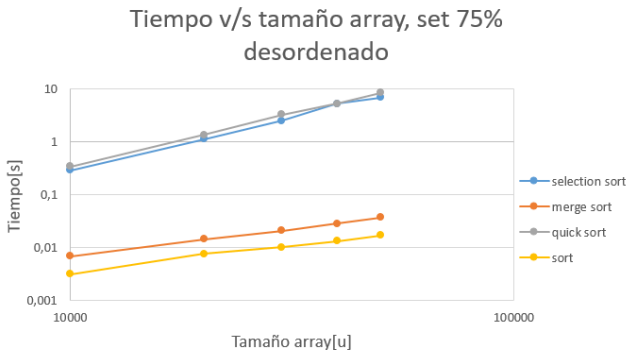
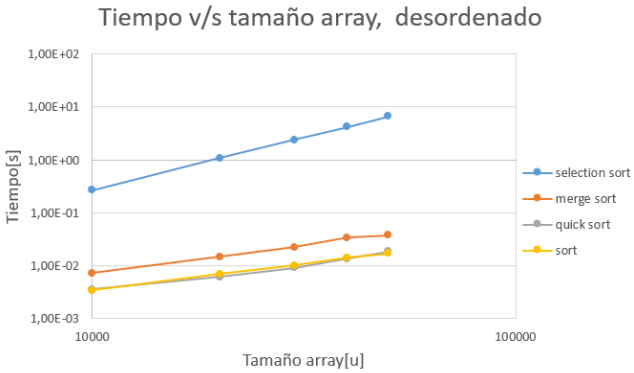


Figura 7: grafico de la tabla de elementos desordenados un 75%

Al 75% quick sort es capaz de rivalizar y a veces superar al algoritmo de selection sort en tiempo.

- ***Vectores 100% desordenados:***



Figuta 8: grafico de la tabla de elementos desordenados.

En al caso del arreglo totalmente desordenado existe un cambio brusco de rendimiento por parte del algoritmo quick sort que es capaz de rivalizar con sort y a veces superarlo incluso.
Pese al cambio de ordenamiento durante los 5 casos, selection sort, mergesort, y sort; esto ha influido levemente con sus rendimientos.

4.2–Algoritmos de multiplicación de matrices:

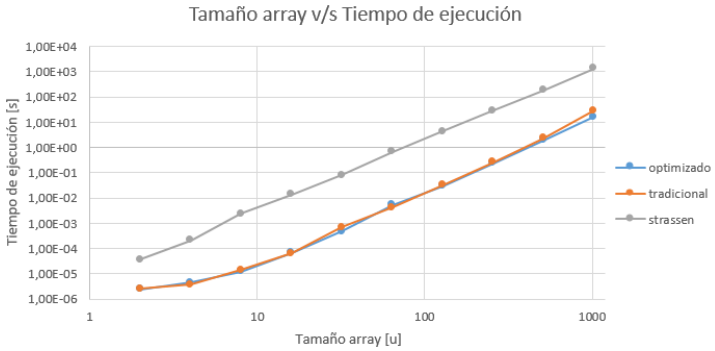


Figura 9: grafico de la tabla de elementos desordenados.

Se Observa un aumento muy similar entre los algoritmos Tradicional y Optimizado a lo largo de los casos respecto al rendimiento del algoritmo tanto tradicional, optimizado; donde existe una diferencia notable en el rendimiento cercano a 1000 datos, donde el Optimizado es mejor. El algoritmo de Strassen se observa que es el de menor rendimiento en todos los casos testeados, y que el optimizado es un poco más eficiente en el caso de mayor tamaño (1024 elementos).

5. Conclusiones

Podemos afirmar que el poder predictivo del análisis asintótico ayuda a inferir una tendencia sobre la eficiencia de los algoritmos. En el análisis del comportamiento de los algoritmos de ordenamiento, Merge Sort muestra un comportamiento con una complejidad de $O(n \log n)$, acorde a lo esperado si se compara con Selection Sort, que es cuadrático. Sin embargo, este no es el único factor a considerar. Quick Sort, que también debería tener un comportamiento similar al de Merge Sort, termina siendo peor que Selection Sort en la mayoría de los casos, salvo cuando el arreglo está completamente desordenado, donde su tiempo se compara e incluso supera al de la función proporcionada por la biblioteca estándar. Esto último se debe al pivote utilizado en el experimento, ya que un mal pivote puede provocar el peor caso, que es de complejidad cuadrática (igual a la de Selection Sort). Por esta razón, el algoritmo funciona mejor cuando el vector está desordenado.

Una forma de mejorar el rendimiento de Quick Sort sería elegir un pivote aleatorio o uno cercano a la media del vector. Cabe destacar que Quick Sort es un algoritmo in-place, lo que le permite ser más eficiente en el uso del espacio, a diferencia de Merge Sort, que implica un mayor requerimiento de memoria, lo que puede restringir su uso en situaciones donde el espacio de memoria es limitado.

Por otro lado, en términos generales, el rendimiento del algoritmo de multiplicación de matrices optimizado es mayor en comparación con el tradicional y con el de Strassen (a pesar de tener un mejor rendimiento teórico), debido a una mejor preservación de la localidad de los datos, lo que optimiza los accesos a la memoria.

En resumen, el análisis asintótico es útil para comprender la tendencia de los algoritmos, especialmente en casos de gran tamaño. No obstante, características como la implementación del algoritmo (como el caso del pivote en Quick Sort) y la preservación de la localidad de los datos (en el algoritmo de Strassen) son factores necesarios a tener en cuenta para un entendimiento más detallado del comportamiento del algoritmo con los datos.

6. Bibliografía

- Cormen, Thomas H., Leiserson, Charles E. (2009). Rivest, Ronald L., Stein, Clifford. "Introduction to Algorithms" (3a edición). MIT Press.
- GeeksforGeeks. (2024a, febrero 22). *Time Complexities of all Sorting Algorithms*. GeeksforGeeks. https://www.geeksforgeeks.org/time-complexities-of-all-sorting-algorithms/
- GeeksforGeeks. (2022, 22 diciembre). *Introsort C++'s Sorting Weapon*. GeeksforGeeks. https://www.geeksforgeeks.org/introsort-cs-sorting-weapon/

Lo códigos de los algoritmos utilizados en este informe puede ser encontrada en las siguientes páginas:

- Selection sort: GeeksforGeeks. (2024b, agosto 6). *Selection sort Algorithm*. GeeksforGeeks. https://www.geeksforgeeks.org/selection-sort-algorithm-2/
- Merge Sort: GeeksforGeeks. (2024b, agosto 6). *Merge Sort Data Structure and Algorithms Tutorials*. GeeksforGeeks. https://www.geeksforgeeks.org/merge-sort/
- Quick sort: GeeksforGeeks. (2024c, septiembre 1). *Quick sort*. GeeksforGeeks. https://www.geeksforgeeks.org/quick-sort-algorithm/
- Tradicional y Optimizado: GeeksforGeeks. (2024a, julio 23). *Program to multiply two matrices*. GeeksforGeeks. https://www.geeksforgeeks.org/c-program-multiply-two-matrices/
- Strassen: GeeksforGeeks. (2023, 1 junio). *Divide and Conquer | Set 5 (Strassen's Matrix Multiplication)*. GeeksforGeeks. https://www.geeksforgeeks.org/strassens-matrix-multiplication/

La implementación tanto de los algoritmos como el código utilizado para la creación del dataset y de la lectura de los archivos de texto, etc; puede encontrase en el siguiente repositorio de Github:

https://github.com/WalterSanhueza/Tarea-1-Algoritmo-Dividir-y-Conquistar.git

7. Anexos

Tablas:

ordenado				
Tamaño [u]	selection_sort [s]	merge_sort [s]	quick_sort [s]	sort [s]
10000	0,278825	0,00586043	0,827194	0,00185241
20000	1,40089	0,0120318	3,27729	0,00429547
30000	2,79092	0,0179842	7,3588	0,00605272
40000	5,9014	0,0251227	13,7251	0,0089802
50000	7,12402	0,0303611	22,4694	0,0106187

Figura 10: Tabla con los datos de tiempo según el tamaño de los vectores para los casos ordenados.

25%				
Tamaño [u]	selection_sort [s]	merge_sort [s]	quick_sort [s]	sort [s]
10000	0,281316	0,00628422	0,698706	0,00299309
20000	1,11E+00	1,28E-02	2,81637	5,50E-03
30000	2,64297	0,0190072	6,38756	0,00939129
40000	5,09326	0,0264877	11,2555	0,0118941
50000	7,28E+00	0,0409945	17,544	1,48E-02

Figura 11: Tabla con los datos de tiempo según el tamaño de los vectores para los de 25% desordenados.

50%				
Tamaño [u]	selection_sort [s]	merge_sort [s]	quick_sort [s]	sort [s]
10000	0,307394	0,00854033	0,561883	0,00284689
20000	1,17E+00	1,33E-02	2,26048	6,88E-03
30000	2,54308	0,0198787	6,07472	0,011273
40000	4,95414	0,0271772	9,46214	0,0131961
50000	7,34E+00	0,0349877	13,6276	1,55E-02

Figura 12: Tabla con los datos de tiempo según el tamaño de los vectores para los casos de 50% desordenados.

75%				
Tamaño [u]	selection_sort [s]	merge_sort [s]	quick_sort [s]	sort [s]
10000	0,291448	0,00677499	0,345416	0,00308405
20000	1,12208	1,43E-02	1,35999	7,51E-03
30000	2,50769	0,0208219	3,27891	0,0100318
40000	5,31925	0,0282615	5,30107	0,0131436
50000	6,97665	0,037139	8,51167	1,68E-02

Figura 13: Tabla con los datos de tiempo según el tamaño de los vectores para los de 75% desordenados.

100%				
Tamaño [u]	selection_sort [s]	merge_sort [s]	quick_sort [s]	sort [s]
10000	2,68E-01	0,00715698	0,00355829	0,00337518
20000	1,06669	1,45E-02	0,0061135	6,81E-03
30000	2,37535	0,0219111	0,00894778	0,0100668
40000	4,19451	0,0332528	0,013399	0,0139712
50000	6,59508	0,0376772	0,0184888	1,69E-02

Figura 14: Tabla con los datos de tiempo según el tamaño de los vectores para los desordenados.

tamaño [u]	Tradicional [s]	Optimizado [s]	Strassen [s]
2	2,48E-06	2,42E-06	3,54E-05
4	3,64E-06	4,44E-06	0,00021082
8	1,41E-05	1,20E-05	0,0023353
16	6,55E-05	6,68E-05	0,0138192
32	0,00067546	0,00046954	0,078591
64	0,0041487	0,00533602	0,664494
128	0,0321181	0,0300178	4
256	0,257785	0,242301	29,17
512	2	2	188,48
1024	29	16	1372,63

Figura 15: Tabla con los datos de tiempo según el tamaño de las matrices.