Tarea 1, Entrega 1

Josué Torres Sibaja, C37853, josue.torressibaja@ucr.ac.cr

***Resumen*—En este trabajo se presenta un análisis comparativo del comportamiento de los algoritmos de ordenamiento por selección, inserción y mezcla. Para esto, se requirió la implementación de los algoritmos en C++ y la medición de sus tiempos de ejecución utilizando arreglos de diferentes tamaños. Los resultados se visualizaron mediante gráficos de líneas que muestran el rendimiento de cada algoritmo. Se concluye que el algoritmo de mezcla es más eficiente que los algoritmos de selección e inserción en arreglos grandes, aunque estos últimos pueden resultar de utilidad a la hora de trabajar con arreglos pequeños o parcialmente ordenados. El trabajo destaca la importancia de elegir el algoritmo adecuado según el tamaño de los datos y las necesidades a cubrir.**

Palabras clave—ordenamiento, selección, inserción, mezcla.

I. Introducción

El ordenamiento de datos es una tarea fundamental en informática, ya que facilita la organización y búsqueda eficiente de información (Algorino, n.d.). Existen diversos algoritmos de ordenamiento, cada uno con diferentes características y aplicaciones. En este trabajo se analizarán los algoritmos de selección, inserción y mezcla, con el objetivo de comparar su rendimiento en función del tamaño de los datos. El algoritmo de selección realiza comparaciones sucesivas para encontrar el menor elemento y colocarlo en la posición correcta. El algoritmo de inserción introduce elementos en su posición correcta de forma iterativa. Finalmente, el algoritmo de mezcla utiliza la técnica de 'divide y vencerás', que le otorga una complejidad temporal de $\Theta(n \log n)$, haciéndolo altamente eficiente en arreglos grandes (Cormen et al., 2022). Para este estudio se implementaron estos algoritmos en C++ y se midieron sus tiempos de ejecución en diferentes tamaños de arreglos, con el fin de obtener conclusiones prácticas sobre su comportamiento y eficiencia.

II. Metodología

Para lograr lo propuesto se realizó la implementación de los algoritmos en el lenguaje C++, modificando el archivo 'Ordenador.hpp' facilitado por el profesor, y se creó un archivo 'main.c' con el que se pudieran generar informes sobre los tiempos de ejecución de los algoritmos. El código se muestra en los apéndices, y está basado en el pseudocódigo del libro de Cormen y colaboradores. El código para medir los tiempos de ejecución utiliza funciones para generar los arreglos aleatorios, medir el tiempo y poner a trabajar los algoritmos. Así mismo, se utilizan varios ciclos *for* anidados para imprimir de forma ordenada los resultados de tiempo de cada algoritmo. Es importante resaltar que el arreglo de números aleatorios es el mismo en cada ejecución de cada algoritmo (solamente se modifica su tamaño).

Tras correr el programa y obtener el tiempo de cada ejecución, estos se registraron junto con su promedio en el cuadro I.

Finalmente, se utilizó la plataforma Canva para crear los gráficos de líneas para cada algoritmo y el gráfico de comparación. Para esto se introdujeron los tamaños de los arreglos en el eje X, y los tiempos de ejecución en el eje Y, generando automáticamente la gráfica. Las imágenes también fueron editadas manualmente para agregar títulos y nombres a los ejes, así como crear una escala personalizada para la gráfica de comparación.

Cuadro I

Tiempo de ejecución de los algoritmos

		Tiempo (ms) *Corrida*					
Algoritm	Tam.	1	2	3	4	5	Prom.
Selecció	50000	2528.65	2526.57	2527.16			2527.46
	100000	10140.2	10122.5	10153.8			10138.9
	150000	22890.5	22810.1	22838.7			22846.4
	200000	40667.4	40472.1	40454.3			40531.3
Inserción	50000	1164.54	1164.68	1169.12			1166.11
	100000	4715.16	4845.85	6998.94			5519.98
	150000	14237.7	13579.7	13180.5			13666
	200000	20792.3	20344.2	21477.6			20871.4
Mezcla	50000	19.7667	19.9541	20.4075			20.0428
	100000	41.1392	40.7761	42.5718			41.4957
	150000	62.6132	63.0288	62.327			62.6563
	200000	84.6673	84.5185	85.1676			84.7845

Los tiempos de ejecución de las tres corridas de cada algoritmo se muestran en el cuadro I.

III. Resultados

Para el algoritmo de selección, los tiempos de ejecución se mantuvieron prácticamente iguales en las tres corridas de cada tamaño, siendo la mayor diferencia 213.1 ms entre la primera y la última corrida con el arreglo de 200000 números.

Para el algoritmo de inserción, los tiempos tienden a variar un poco entre la primera y la última corrida con cada tamaño. Para la mayoría de arreglos la primera corrida es más rápida que la última, siendo la diferencia más grande 2283.78 ms con el arreglo de 100000. Para el arreglo de 150000 la última corrida fue más rápida que la primera.

Para el algoritmo de mezcla, para los arreglos de 100000 y 150000, la corrida más diferente es la segunda. En el caso del arreglo de 50000 la primera corrida es la más rápida y la última la más lenta, y para el arreglo de 200000 la segunda corrida es la más rápida y la tercera la más lenta.

La forma de las curvas fue la esperada, ya que las curvas de los algoritmos $\Theta(n^2)$ tuvieron una forma aproximadamente parabólica y la curva del algoritmo $\Theta(n \log n)$ tuvo una forma aproximadamente lineal.

Los tiempos promedio se muestran gráficamente en la figura 1.

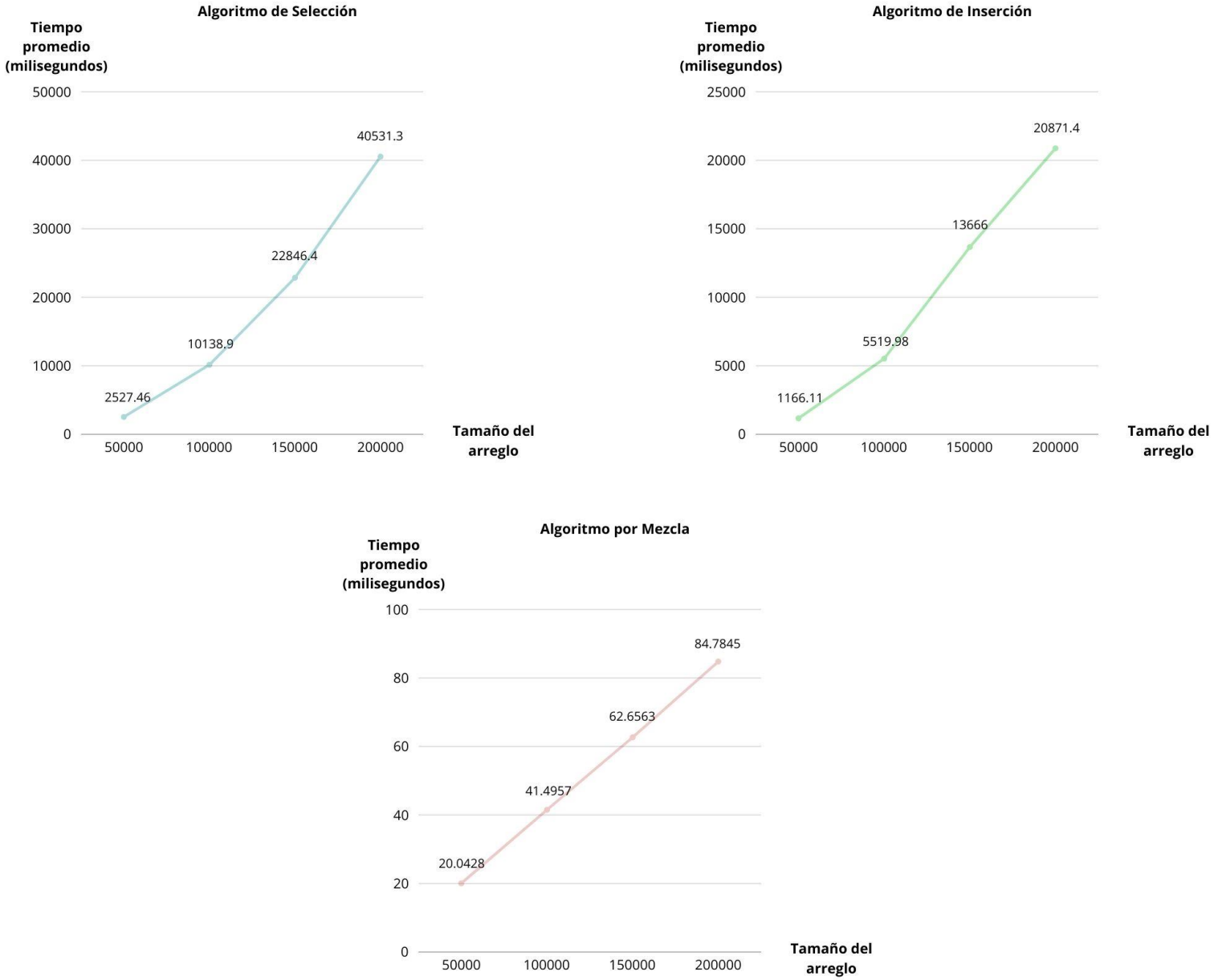

Figura 1. Tiempos promedio de ejecución de los algoritmos de ordenamiento por selección, inserción y mezcla.

Las curvas se muestran de forma conjunta en la figura 2.

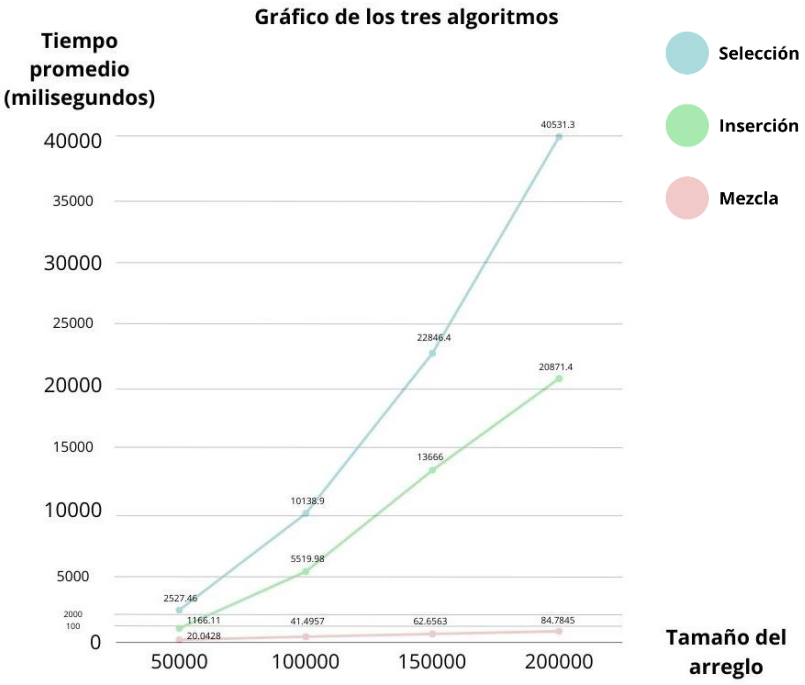

Figura 2. Gráfico comparativo de los tiempos promedio de ejecución de algoritmos de ordenamiento.

En la gráfica se aprecian enormes diferencias entre los tiempos de ejecución, ya que el algoritmo de mezcla tuvo promedios sumamente bajos en comparación de los otros dos algoritmos, manteniéndose por debajo de los 100 milisegundos, cuando los otros rebasaban los 1000 milisegundos en sus ejecuciones más rápidas.

IV. Discusión

Como primer punto, con base en los resultados presentados, se puede observar que el algoritmo de mezcla es muchísimo más rápido que los otros algoritmos. Esto se debe a la complejidad temporal de cada uno, ya que los algoritmos de selección e inserción tienen una complejidad de $\Theta(n^2)$, lo que los hace preferibles para arreglos pequeños (Kemeny, 2023). Por otro lado, el algoritmo de mezcla tiene una complejidad de $\Theta(n \log n)$, lo que lo hace superior al trabajar con tamaños de arreglos grandes (LinkedIn Algorithms, n.d.). En este caso, como todos los arreglos son de miles de números, el algoritmo de mezcla es el más eficiente, pero si se usaran arreglos mucho más pequeños probablemente los otros dos algoritmos destacarían por encima de este (templatetypedef, 2020).

Adicionalmente, los experimentos realizados revelan que, aunque tanto el algoritmo de selección como el de inserción comparten la misma complejidad de $\Theta(n^2)$, sus tiempos de ejecución no son idénticos. En todos los casos de este experimento, el algoritmo de selección fue más lento que el de inserción, sin embargo, el algoritmo de inserción tiene la ventaja de ser más eficiente en arreglos que ya están parcialmente ordenados (CoderSaty, 2023).

Por otra parte, con los datos recolectados en la tabla, se puede observar un patrón en los tiempos de ejecución. Mientras que el algoritmo de selección muestra diferencias mínimas entre las distintas corridas, el algoritmo de inserción presenta más variabilidad, especialmente en arreglos de tamaño intermedio. Por su parte, el algoritmo de mezcla se mantuvo consistentemente rápido en todas sus corridas, sin grandes variaciones entre ellas, lo que refuerza su estabilidad y eficiencia en comparación con los otros dos algoritmos. Estos resultados sugieren que la eficiencia del algoritmo de mezcla lo hace más adecuado para situaciones donde se requiere consistencia y rapidez, independientemente del tamaño del arreglo.

V. Conclusiones

A partir de los resultados obtenidos, podemos concluir que el algoritmo de mezcla es considerablemente más eficiente que los algoritmos de selección e inserción cuando se trabaja con arreglos de gran tamaño. Su complejidad $\Theta(n \log n)$ le permitió mantener tiempos de ejecución bajos y consistentes, lo que lo hizo destacar como la opción preferida para manejar grandes volúmenes de datos. Por otro lado, aunque los algoritmos de selección e inserción presentan una complejidad cuadrática $\Theta(n^2)$ y por lo tanto son menos eficientes, se valora que estos podrían ser adecuados a la hora de trabajar con arreglos pequeños o parcialmente ordenados. Finalmente, considero que el trabajo demuestra la importancia de elegir el algoritmo adecuado según el tamaño y las características del arreglo a ordenar, y destaca la utilidad de algoritmos con menor complejidad temporal en aplicaciones prácticas.

Referencias

[1] Algorino. (n.d.). La Relevancia de la Ordenación y Búsqueda en Informática. Algor Cards. https://cards.algoreducation.com/es/content/XYD4QRuN/ordenacion-busqueda-informatica

[2] CoderSaty. (30 de marzo de 2023). Difference between Insertion sort and Selection sort. GeeksforGeeks. https://www.geeksforgeeks.org/difference-between-insertion-sort-and-selection-sort/

[3] Cormen, T. et al. (2022). Introduction to algorithms (4th ed.). MIT Press.

[4] Kemeny, J. (28 de marzo de 2021). How does size of list in merge-sort, quick-sort, insertion-sort, matter? Computer Science Stack Exchange. https://cs.stackexchange.com/questions/138201/how-does-size-of-list-in-merge-sort-quick-sort-insertion-sort-matter

[5] LinkedIn Algorithms. (n.d.). What are the benefits and drawbacks of using merge sort over insertion sort? LinkedIn. https://www.linkedin.com/advice/3/what-benefits-drawbacks-using-merge-sort-over-insertion#:~:text=One%20of%20the%20main%20benefits,as%20the%20input%20size%20increases

[6] MohdArsalan. (29 de enero de 2024). Merge Sort vs. Insertion Sort. GeeksforGeeks. https://www.geeksforgeeks.org/merge-sort-vs-insertion-sort/

[7] templatetypedef. (23 de abril de 2020). When would you use Selection sort versus Merge sort? Stack Overflow. https://stackoverflow.com/questions/61393831/when-would-you-use-selection-sort-versus-merge-sort

Josué Torres Sibaja. Me interesa profundamente el mundo de la informática, su historia, su importancia en la actualidad y sus posibles avances y aplicaciones a futuro.

APÉNDICE A

Código de los Algoritmos

El código se muestra en los algoritmos 1, 2 y 3.

Algoritmo 1 Algoritmo de selección.

```
   void ordenamientoPorSeleccion(int *A, int n) const
{
       if (A == nullptr || n <= 0) return;   /**
Verificación defensiva de entrada. */

       for (int i = 0; i < n - 1; ++i) {
         int m = i;  /** m es el índice del elemento
más pequeño. */
         for (int j = i + 1; j < n; ++j) {
           if (A[j] < A[m]) {
             m = j;  // Actualizar m si se encuentra
un elemento más pequeño. */
           }
         }
         /** Se intercambia el elemento más pequeño
encontrado con A[i]. */
         swap(A[i], A[m]);
       }
  }
```

Algoritmo 2 Algoritmo de inserción.

```
   void ordenamientoPorInsercion(int *A, int n) const {
       if (A == nullptr || n <= 0) return;  /** Verificación defensiva de entrada. */

       for (int i = 1; i < n; ++i) {
         int key = A[i];  /** Se guarda el elemento actual. */
         int j = i - 1;
         /** Se mueven los elementos mayores que key una posición adelante. */
         while (j >= 0 && A[j] > key) {
           A[j + 1] = A[j];
           --j;
         }
         /** Insertar key en la posición correcta. */
         A[j + 1] = key;
       }
  }
```

Algoritmo 3 Algoritmo de mezcla.

```
     void ordenamientoPorMezcla(int *A, int n) const {
       if (A == nullptr || n <= 0) return;  /** Verificación defensiva de entrada. */

       /** Llamado a la función recursiva para ordenar el arreglo completo. */
       mezclaRec(A, 0, n - 1);
  }

     void mezclaRec(int *A, int p, int r) const {
       if (p >= r) return;  /** Caso de arreglo de un elemento o rango incorrecto. */

       int q = (p + r) / 2;  /** Calcular el punto medio. */
       mezclaRec(A, p, q);  /** Ordenar la primera mitad. */
       mezclaRec(A, q + 1, r);  /** Ordenar la segunda mitad. */
       mezclar(A, p, q, r);  /** Mezclar ambas partes. */
  }

    void mezclar(int* A, int p, int q, int r) const {
       int nI = q - p + 1;  /** Tamaño del subarreglo izquierdo. */
       int nD = r - q;  /** Longitud del subarreglo derecho. */

       /** Crear los subarreglos temporales L y R. */
       vector<int> I(nI);
       vector<int> D(nD);
       /** Copiar los elementos del subarreglo A[p:q] en L. */
       for (int i = 0; i < nI; ++i) {
         I[i] = A[p + i];
       }
       /** Copiar los elementos del subarreglo A[q+1:r] en R. */
       for (int j = 0; j < nD; ++j) {
         D[j] = A[q + 1 + j];
       }
       /** i: índice del subarreglo L. j: índice del subarreglo R. k: índice del arreglo original. */
       int i = 0, j = 0, k = p;

       /** Se mezclan los subarreglos L y R de regreso en A[p:r]. */
       while (i < nI && j < nD) {
         if (I[i] <= D[j]) {
           A[k] = I[i];
           i = i + 1;
```

```
          } else {
            A[k] = D[j];
            j = j + 1;
          }
          k = k + 1;
        }
        /** Si quedan elementos en L, se copian en A. */
        while (i < nI) {
          A[k] = I[i];
          i = i + 1;
          k = k + 1;
        }
        /** Si quedan elementos en R, se copian en A. */
        while (j < nD) {
          A[k] = D[j];
          j = j + 1;
          k = k + 1;
        }
    }
```