Análisis de los Algoritmos de Ordenamiento por Selección, Inserción, Mezcla, Montículos, Rápido y por Residuos

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—Algoritmo, Ordenamiento.

I. Introducción

El análisis de algoritmos tiene una gran importancia en las ciencias de la computación y la informática, ya que nos permite comprender y predecir el comportamiento de un algoritmo con relación al tiempo, permitiéndonos establecer cotas teóricas para los diferentes casos en los que se utilicen. Estas cotas nos permiten realizar estimaciones sobre el tiempo de ejecución de cada algoritmo respecto al tamaño de los datos que procesan, sin embargo, la realización de pruebas y experimentos nos permiten comparar los resultados prácticos de los tiempos de ejecución con sus estimaciones teóricas (Cormen et al., 2022).

Asimismo, es importante mencionar que existen diferentes tipos de algoritmos de ordenamiento, como el Algoritmo de Ordenamiento por Selección, el Algoritmo de Ordenamiento por Inserción, el Algoritmo de Ordenamiento por Mezcla, el Algoritmo de Ordenamiento por Montículos, el Algoritmo de Ordenamiento Rápido, y el Algoritmo de Ordenamiento por Residuos, entre otros (Cormen et al., 2022).

El Algoritmo de Ordenamiento por Selección, busca el menor valor en un arreglo y lo coloca en la primera posición de la parte desordenada del arreglo. Posee una complejidad temporal de $\Theta(n^2)$ en su peor caso y en sus casos promedio (Cormen et al., 2022; W3Schools, n.d.).

El Algoritmo de Ordenamiento por Inserción, toma el primer valor en la parte desordenada de un arreglo y lo compara con el siguiente, intercambiándolos si el valor actual es mayor al siguiente. Posee una complejidad temporal de $\Theta(n^2)$ en su peor caso y en sus casos promedio (Cormen et al., 2022; W3Schools, n.d.).

El Algoritmo de Ordenamiento por Mezcla, divide un arreglo de valores a la mitad, y divide recursivamente los subarreglos hasta que estos contengan solamente un elemento. Luego, une los arreglos, dejando el valor menor siempre de primero, y continúa hasta unir todos los arreglos. Posee una complejidad temporal de $\Theta(n \log n)$ en su peor caso y en sus casos promedio (Cormen et al., 2022; W3Schools, n.d.).

El Algoritmo de Ordenamiento por Montículos, toma un arreglo de valores y lo reorganiza en una estructura de datos llamada montículo máximo, dejando el mayor elemento como su raíz. Luego, intercambia la raíz por el último elemento, dejando al valor que estaba en la raíz en su posición correcta. Posee una complejidad temporal de $\Theta(n \log n)$ en su peor caso (Cormen et al., 2022; W3Schools, n.d.).

El Algoritmo de Ordenamiento Rápido selecciona un valor en el arreglo como pivote, y ordena el arreglo de forma que los valores menores al pivote queden a su izquierda, y los valores mayores queden a su derecha. Luego, realiza el mismo proceso recursivamente para los subarreglos menores y mayores. Posee una complejidad temporal de $\Theta(n^2)$ en su peor caso, o $\Theta(n \log n)$ en sus casos promedio (Cormen et al., 2022; W3Schools, n.d.).

El Algoritmo de Ordenamiento por Residuos ordena un arreglo por dígitos individuales, de derecha a izquierda, ordenando respecto al dígito que se está tomando en cuenta, y luego respecto a los siguientes dígitos. Posee una complejidad temporal de $\Theta(d(n + k))$ en su peor caso y en sus casos promedio (Cormen et al., 2022; W3Schools, n.d.).

En este trabajo, se llevará a cabo una comparación entre las cotas teóricas de complejidad temporal y los tiempos de ejecución prácticos de los algoritmos de ordenamiento mencionados anteriormente. El propósito de este análisis es estudiar si, y hasta qué punto, las predicciones y el comportamiento teórico de los algoritmos se alinea con sus resultados prácticos. De esta forma, se podrá obtener una comprensión más completa del comportamiento real de los algoritmos de ordenamiento estudiados.

II. Metodología

Para lograr lo propuesto, se realizó la implementación de los algoritmos en el lenguaje de programación C++, utilizando el sistema operativo Windows 11, en el cual se instaló WSL 2 (Windows Subsystem for Linux) para facilitar la compilación de código utilizando el compilador g++ (Ubuntu 11.4.0). Para garantizar condiciones consistentes para todas las pruebas, se deshabilitó cualquier proceso de fondo que pudiera generar interferencias. Para la medición en milisegundos de los tiempos de ejecución se utilizó la biblioteca *chrono* (específicamente la

función *chrono::high_resolution_clock*) para obtener mediciones de alta precisión, capturando el tiempo de inicio y finalización de cada ejecución. El código se muestra en los apéndices, y está basado en el pseudocódigo del libro de Cormen y colaboradores.

Se realizaron tres pruebas con cuatro diferentes tamaños de arreglos de valores enteros (50 000, 100 000, 150 000 y 200 000) para cada algoritmo de ordenamiento, con el objetivo de obtener el tiempo de ejecución de cada prueba y el promedio de las tres pruebas. El arreglo de valores aleatorios utilizado fue el mismo en cada ejecución de cada algoritmo de ordenamiento (solamente se modificó su tamaño), y contenía números en el rango de $[0, (2^{32}) - 1]$.

Tras realizar las pruebas y obtener el tiempo de cada ejecución, estos se registraron junto con su promedio en el cuadro I. A partir de los tiempos promedio, se generaron gráficos individuales y comparativos para visualizar el comportamiento de cada algoritmo de ordenamiento según incrementa el tamaño del arreglo.

III. Resultados

Cuadro I

Tiempo de ejecución de los algoritmos de ordenamiento

Algoritmo	Tam.	Tiempo (ms) Corrida 1	2	3	Prom.
Selección	50000	2528.65	2526.57	2527.16	2527.46
	100000	10140.20	10122.50	10153.80	10138.90
	150000	22890.50	22810.10	22838.70	22846.40
	200000	40667.40	40472.10	40454.30	40531.30
Inserción	50000	1164.54	1164.68	1169.12	1166.11
	100000	4715.16	4845.85	6998.94	5519.98
	150000	14237.70	13579.70	13180.50	13666.00
	200000	20792.30	20344.20	21477.60	20871.40
Mezcla	50000	19.76	19.95	20.40	20.04
	100000	41.13	40.77	42.57	41.49
	150000	62.61	63.02	62.32	62.65
	200000	84.66	84.51	85.16	84.78
Montículos	50000	34.27	33.61	32.89	33.59
	100000	70.64	66.73	67.02	68.13
	150000	106.28	109.38	114.11	109.92
	200000	150.62	150.78	152.16	151.19
Rápido	50000	15.12	16.31	17.89	16.44
	100000	33.91	33.31	33.13	33.45
	150000	51.92	49.52	51.47	50.97
	200000	69.52	67.66	66.99	68.06
Residuos	50000	5.45	7.26	5.62	6.11
	100000	10.86	10.83	10.85	10.84
	150000	18.18	14.18	14.69	15.69
	200000	20.39	19.46	19.78	19.88

Figura 1

Gráfico de los tiempos de ejecución promedio del Algoritmo de Ordenamiento por Selección

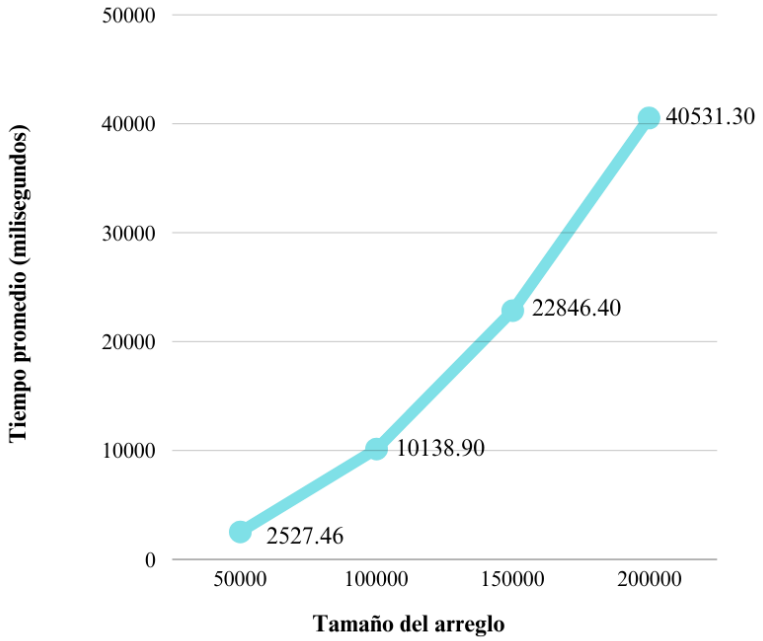


Figura 2

Gráfico de los tiempos de ejecución promedio del Algoritmo de Ordenamiento por Inserción

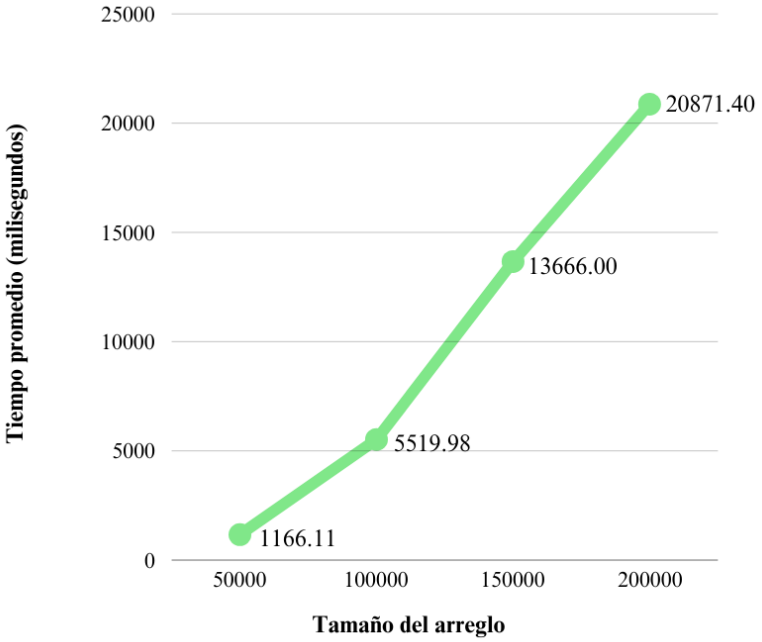


Figura 3

Gráfico de los tiempos de ejecución promedio del Algoritmo de Ordenamiento por Mezcla

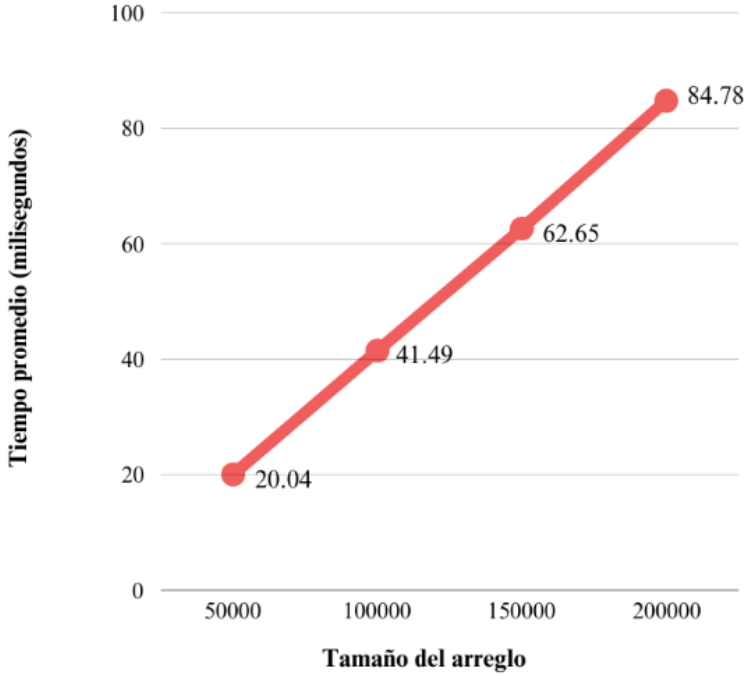


Figura 4

Gráfico de los tiempos de ejecución promedio del Algoritmo de

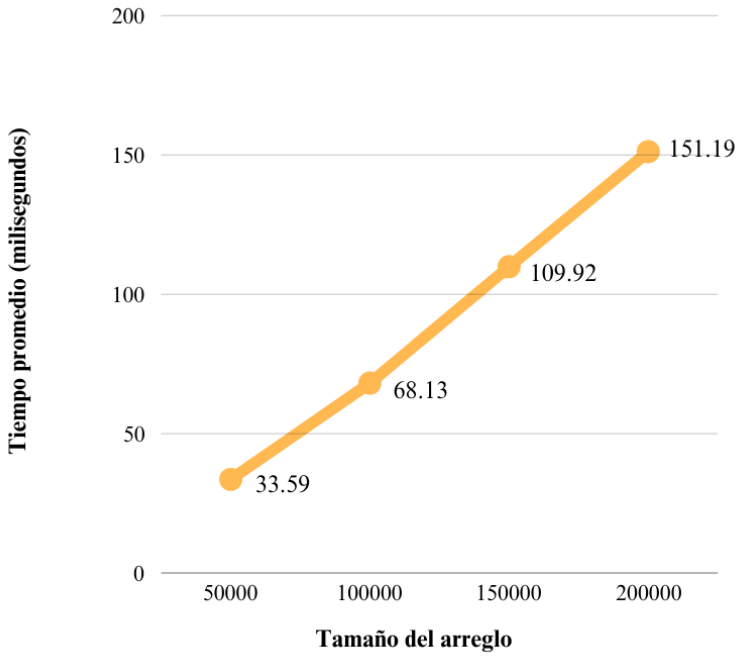


Figura 5

Gráfico de los tiempos de ejecución promedio del Algoritmo de Ordenamiento Rápido

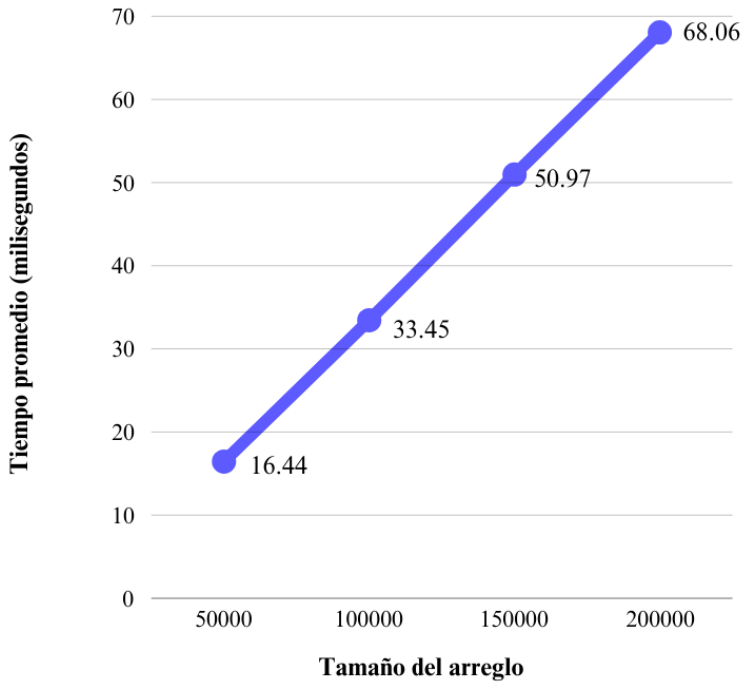


Figura 6

Gráfico de los tiempos de ejecución promedio del Algoritmo de Ordenamiento por Residuos

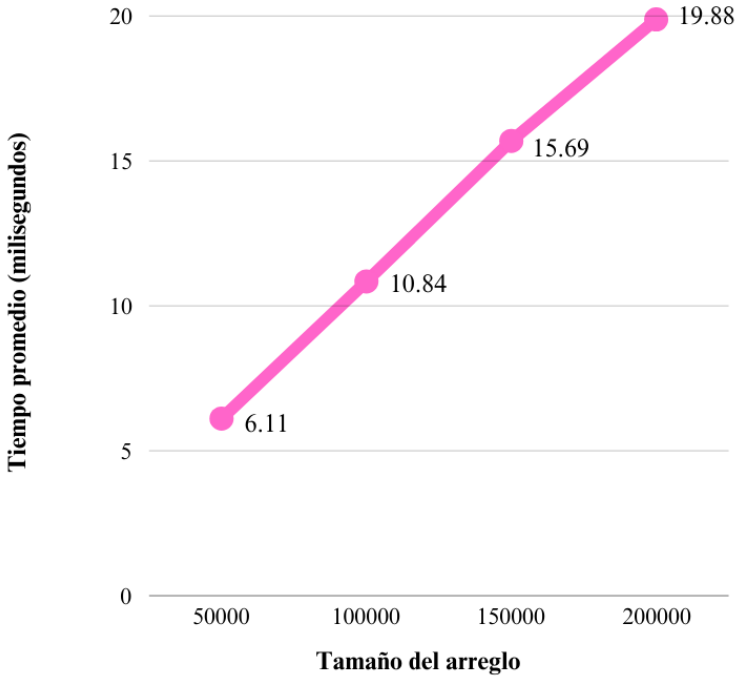
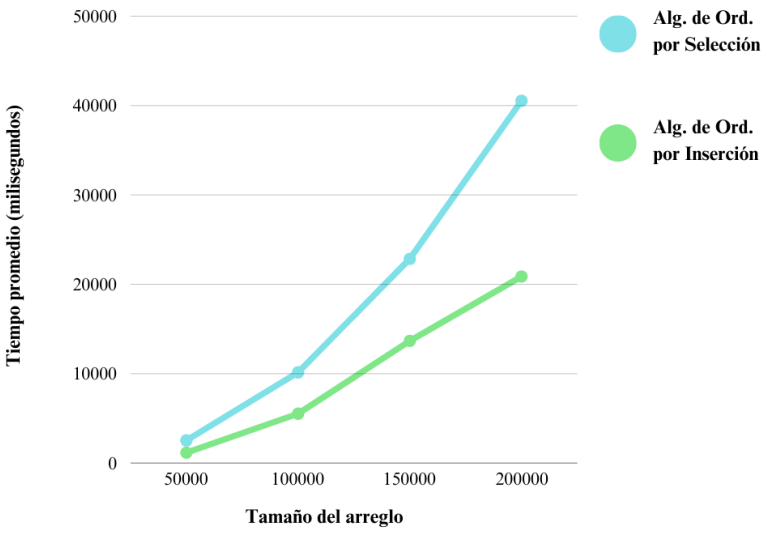


Figura 7

Gráfico comparativo de los tiempos de ejecución promedio del Alg. de Ord. por Selección y el Alg. de Ord. por Inserción



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 8

Gráfico comparativo de los tiempos de ejecución promedio del Alg. de Ord. por Mezcla, el Alg. de Ord. por Montículos, el Alg. de Ord. Rápido y el Alg. de Ord. por Residuos

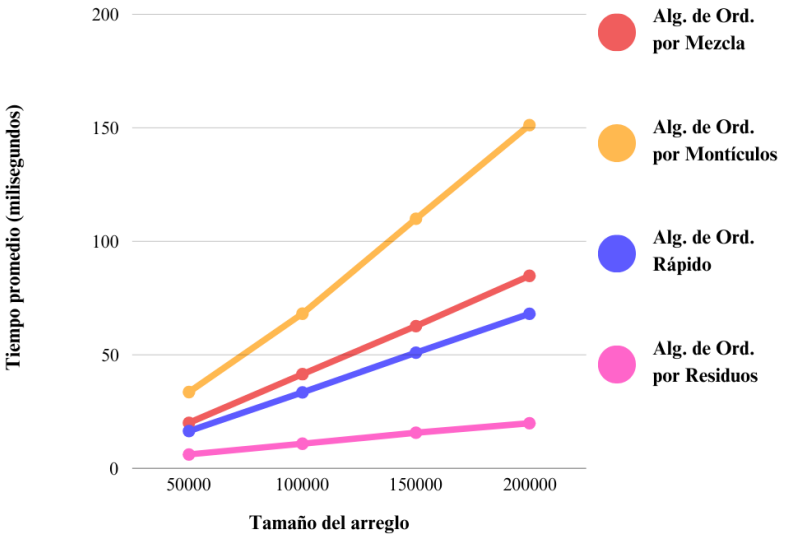


Figura 1. Tiempos promedio de ejecución de los algoritmos de ordenamiento por selección, inserción y mezcla.

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

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

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/

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

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

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

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

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;
        }
    }
```