Algoritmos de Ordenamiento

Andres Murillo Murillo C15424

***Resumen*—Este reporte investiga los algoritmos de ordenamiento Selección, Inserción y Mezcla, con el fin de comparar sus tiempos de ejecución en variados tamaños de arreglos. Se realizó la implementación en C++, y se observaron las eficiencias en arreglos hasta de 200,000 elementos. Los resultados demuestran que el algoritmo de Mezcla es consistentemente más eficiente, especialmente en grandes volúmenes, apoyando su uso preferente en aplicaciones de procesamiento de datos intensivos. Este análisis destaca la relevancia de elegir el algoritmo adecuado basado en la escala de los datos a manejar.**

***Palabras clave*—ordenamiento, selección, inserción, mezcla.**

I. Introducción

En el presente estudio, se realiza un análisis comparativo entre varios algoritmos de ordenamiento: Selección, Inserción y Mezcla (Merge). El objetivo es no solo examinar las variaciones en los tiempos de ejecución debido a las diferentes implementaciones, sino también validar la complejidad teórica de cada algoritmo mediante el análisis de su comportamiento en casos de uso específicos. Este enfoque ofrece una comprensión más completa de cómo las características intrínsecas de cada método influyen en su eficiencia operativa en entornos controlados.

II. Metodología

Para alcanzar los objetivos planteados, se procedió a la implementación de los algoritmos utilizando el lenguaje de programación C++, haciendo uso de distintas bibliotecas que facilitan tanto la compilación como la ejecución adecuada de cada uno de ellos. El código fuente desarrollado se encuentra detallado en los apéndices de este documento, y se fundamenta en el pseudocódigo presentado en el libro "Introducción a los algoritmos" de Cormen et al. (2022).

Para la evaluación comparativa de la eficiencia entre los algoritmos, se procedió a generar series de arreglos compuestos por enteros aleatorios, tanto positivos como negativos, abarcando un rango de -1,000,000 a 1,000,000. Estos arreglos varían en tamaño, incluyendo 50,000, 100,000, 150,000 y 200,000 elementos, representando así distintas magnitudes de datos. Esta metodología permite una exploración detallada sobre el desempeño de los algoritmos ante variados contextos y volúmenes de información, ofreciendo una perspectiva amplia respecto a su comportamiento y eficacia operativa en diferentes escenarios. Para la creación de estos arreglos se ha hecho uso de la librería “random” para generar números aleatorios.

Cuadro I Tiempo de ejecución de los algoritmos

		Tiempo (ms) *Corrida*					
Algoritmo	Tam. (n)	1	2	3	4	5	Prom.
Selección	50000	978.875	997.458	998.049	1001.36	989.307	993.01
	100000	3898.62	3911.28	4051.78	4117.01	4005.22	3996.78
	150000	8951.81	9031.02	9022.42	9052.77	9232.67	9058.14
	200000	16529.2	16739.1	16391.5	15819.6	16234.2	16342.72
Inserción	50000	629.909	633.523	637.136	629.106	639.433	633.82
	100000	2556.47	2542.43	2567.26	2527.13	2537.96	2546.25
	150000	5746.79	5808.35	5813.27	5799.87	5891.93	5812.04
	200000	10104.1	10171.1	10155	10177.9	10229.9	10167.6
Mezcla	50000	5.296	5.819	5.664	5.248	5.702	5.55
	100000	11.141	11.264	11.254	10.92	11.824	11.40
	150000	17.575	17.52	17.917	17.922	17.384	17.66
	200000	23.431	23.05	25.041	23.902	23.561	23.80

Cada experimento se lleva a cabo midiendo el tiempo de ejecución en milisegundos, con el objetivo de evaluar la capacidad de los algoritmos para ordenar los números en una secuencia ascendente. Para determinar la duración exacta de la ejecución del programa, se emplea un cronómetro provisto por la biblioteca “chrono”. Este proceso consiste en registrar el tiempo justo antes de invocar al algoritmo, ejecutar el procedimiento de ordenamiento y, posteriormente, calcular el tiempo en milisegundos que ha transcurrido, sustrayendo el tiempo inicial del tiempo final. Con el fin de establecer un promedio de tiempo de ejecución para cada algoritmo en función de su respectivo tamaño de entrada, se realizan cinco ejecuciones distintas para cada conjunto de datos.

III. Resultados

El "Cuadro I Tiempo de ejecución de los algoritmos" presenta una recopilación meticulosa de los resultados temporales derivados de la ejecución de tres algoritmos de ordenamiento Selección, Inserción y Mezcla con distintas tamaño de entrada. Las dimensiones de los conjuntos de datos analizados oscilan entre 50,000 y 200,000 elementos, permitiendo evaluar la escalabilidad de cada algoritmo.

El algoritmo de Selección, cuya complejidad asintótica en el peor caso es $O(n2)$, muestra una progresión cuadrática en los tiempos de ejecución con un incremento proporcional al cuadrado del tamaño del arreglo. En los experimentos realizados, el tiempo requerido para ordenar 50,000 elementos fue de 993.01 ms, mientras que para 200,000 elementos ascendió a 16,342.72 ms, lo que indica una eficiencia decreciente acorde con el aumento del número de elementos y confirma el comportamiento esperado en su caso promedio asumido como $\Theta(n2)$ en la mayoría de los escenarios prácticos.

El algoritmo de Inserción, muestra un comportamiento análogo en sus tiempos de ejecución, comenzando con un tiempo promedio de 633.82 ms para arreglos de 50,000 elementos y ascendiendo a 10,167.6 ms para los de 200,000 elementos. Este patrón de crecimiento cuadrático se mantiene en línea con las expectativas teóricas y revela limitaciones en contextos de grandes volúmenes de datos, por lo que para estos mismos es menos eficiente.

En contraste con los algoritmos de Inserción y Selección, el algoritmo de Mezcla se caracteriza por una complejidad teórica de O(nlogn), lo cual se refleja en un incremento de tiempo de ejecución más gradual y controlado. Esta eficiencia se manifiesta claramente en los tiempos registrados: comienza con un promedio de solo 5.55 ms para ordenar arreglos de 50,000 elementos y aumenta a 23.80 ms para los de 200,000 elementos. Estos resultados evidencian que, incluso al escalar a grandes cantidades de datos, el algoritmo de Mezcla mantiene un rendimiento alto, siendo significativamente más eficaz para manejar grandes volúmenes de información.

En síntesis, los resultados evidencian que el algoritmo de Mezcla es altamente efectivo para procesar grandes conjuntos de datos, mientras que los algoritmos de Selección e Inserción experimentan una disminución notable en su rendimiento a medida que el tamaño de los datos aumenta. Esta observación reafirma las predicciones de la teoría algorítmica y subraya la relevancia de seleccionar un algoritmo que esté optimizado para las dimensiones específicas y la naturaleza del conjunto de datos a ordenar.

Ahora, se muestran gráficos en los que se puede observar el comportamiento de cada algoritmo de una mejor forma.

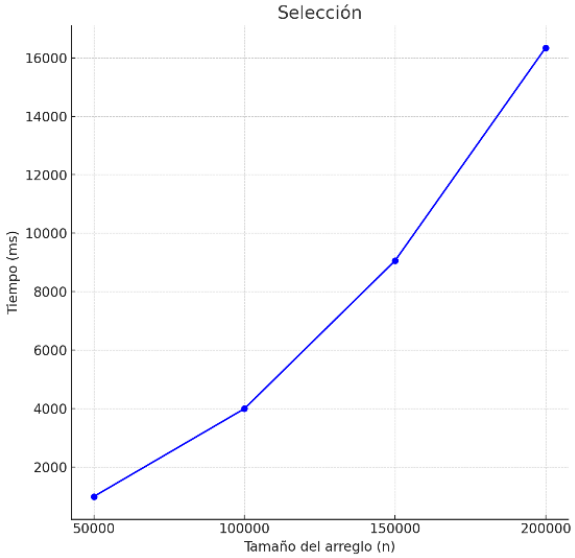

Figura 1 Tiempos promedio de ejecución Algoritmo Selección

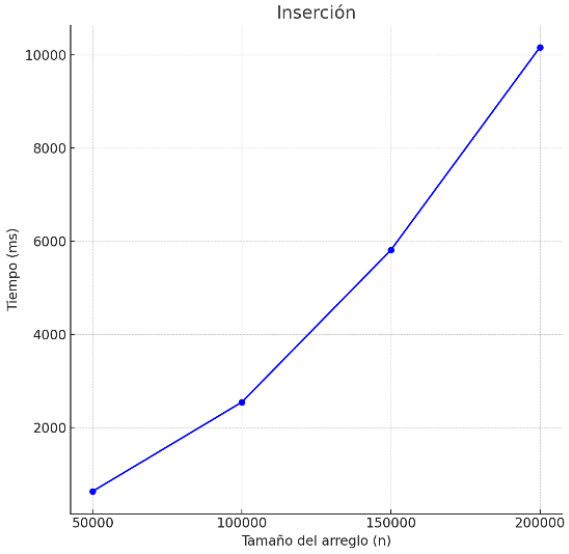

Figura *2* Tiempos promedio de ejecución Algoritmo Inserción

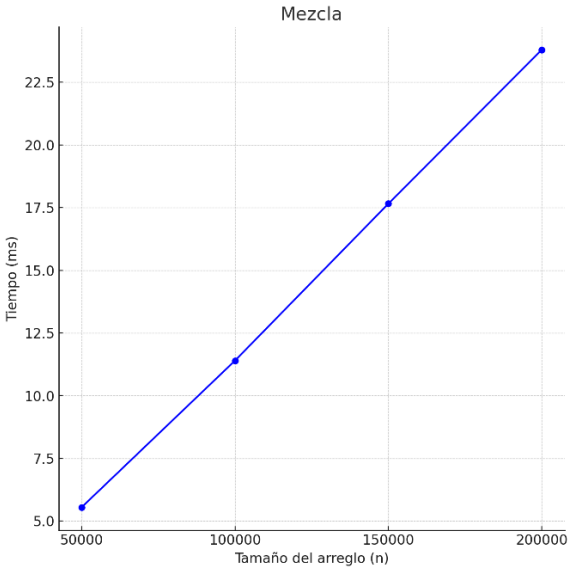

Figura 3 Tiempos promedio de ejecución Algoritmo Selección

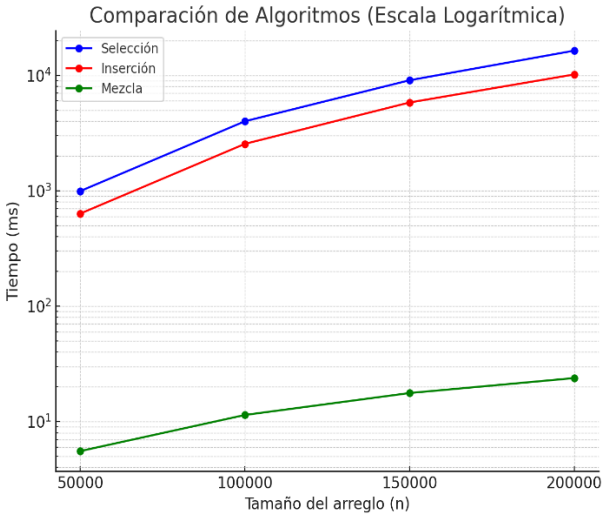

Figura *4* Tiempos promedio de ejecución de los tres algoritmos

La Figura 4 ilustra el tiempo promedio de ejecución de los tres algoritmos en cuestión, permitiendo una comparación directa de su rendimiento. De la visualización se deduce claramente que el algoritmo de Mezcla sobresale por su estabilidad durante la ejecución frente a variaciones en el volumen de datos de entrada. Este algoritmo manifiesta una progresión temporal casi lineal, en contraposición a los patrones de ejecución más lentos e ineficientes de los algoritmos de Selección e Inserción. La consistencia del algoritmo de Mezcla, sin importar la magnitud del conjunto de datos, respalda su preferencia en contextos donde la eficiencia y previsibilidad son fundamentales.

IV. Conclusiones

Los resultados obtenidos de la comparación entre los algoritmos de Selección, Inserción y Mezcla ofrecen perspectivas claras sobre su rendimiento al escalar con tamaños de entrada desde 50,000 hasta 200,000 elementos. Conforme el volumen de datos incrementa, el algoritmo de Selección muestra un aumento significativo en el tiempo de ejecución, alineándose con su complejidad algorítmica de $O(n^2)$., lo cual sugiere que no es el más adecuado para manejar grandes conjuntos de datos debido a su eficiencia decreciente. El algoritmo de Inserción, aunque similar en complejidad teórica al de Selección, también resulta ser ineficiente para grandes volúmenes de datos, evidenciado por su tiempo de ejecución que aumenta exponencialmente con el tamaño del conjunto. Por otro lado, el algoritmo de Mezcla se destaca como el más competente de los tres, manteniendo un crecimiento mucho más controlado y predecible en sus tiempos de ejecución, reflejando su complejidad de de $O(n \log n)$, Este comportamiento casi lineal lo valida como una elección robusta y eficiente para aplicaciones que requieren la ordenación de grandes cantidades de datos. En conclusión, el estudio refuerza la importancia de seleccionar un algoritmo de ordenamiento adecuado basado en la magnitud de la tarea de procesamiento de datos a mano. Mientras que los algoritmos de Selección e Inserción pueden ser suficientes para conjuntos de datos pequeños y tareas menos demandantes, el algoritmo de Mezcla emerge como la alternativa superior para conjuntos de datos extensos, equilibrando eficacia y velocidad de una manera que sus contrapartes no pueden igualar. Esta distinción es esencial para optimizar el rendimiento en aplicaciones de procesamiento de datos en la vida real.

Referencias

[1] Cormen, T. H., Leiserson, C. E., Rivest, R. L. y Stein, C. Introduction to Algorithms, IV ed. MIT Press, 2022.

Andres Murillo Murillo

Estudiante de Ingenierida en Computacion en la Universidad de Costa Rica.

Apéndice A
Código de los Algoritmos

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

Algoritmo 1 El algoritmo de selección perfecciona la organización de una colección de elementos a través de un proceso iterativo que sistemáticamente identifica el elemento mínimo dentro del segmento aún no ordenado. En cada iteración, este elemento es trasladado al inicio de la sección ya ordenada. Este procedimiento se repite sucesivamente, avanzando el límite entre las partes ordenada y no ordenada, hasta que se alcanza un estado en el que toda la serie de datos se encuentra secuencialmente organizada.

```
void Ordenador::seleccion(int *A, int n) {
  int m;
  for (int i = 0; i < n - 1; i++){
    m = i;
    for (int j = i + 1 ; j < n; j++){
      if (A[j] < A[m]) {
        m=j;
      }
    }
    std::swap(A[i], A[m]);
  }
}
```

Algoritmo 2 El algoritmo de inserción trabaja seleccionando secuencialmente cada elemento de la porción no ordenada del conjunto de datos y lo incorpora en la ubicación exacta dentro de la subsecuencia ya organizada. Este método asegura que, tras cada inserción, la sección ordenada se expanda manteniendo su orden hasta que todo el conjunto esté sistemáticamente ordenado. Con cada operación, el algoritmo compara el elemento seleccionado con los precedentes, desplazándolos si es necesario, para encontrar la posición precisa donde el elemento debe ser ubicado, garantizando así la continuidad de la ordenación.

```
void Ordenador::insercion(int *A, int n) {
  int key;
  int i;
  for (int j = 1; j < n; j++) {
    key = A[j];
    i = j - 1;
    while (i >= 0 && A[i] > key) {
      A[i + 1] = A[i];
      i = i - 1;
    }
    A[i + 1] = key;
  }
}
```

Algoritmo 3 El algoritmo de mezcla adopta un enfoque dividir para conquistar, dividiendo el arreglo original en mitades más manejables. Estas subsecciones se ordenan de manera independiente mediante llamadas recursivas. Una vez que cada mitad está ordenada, el algoritmo procede a fusionarlas meticulosamente, ensamblando así el conjunto completo en un arreglo finalmente ordenado. Este proceso garantiza que los elementos dispersos se reintegren en una secuencia coherente y organizada.

```
void Ordenador::merge(int *A, int p, int q, int r) {
  int n1 = q - p + 1;
  int n2 = r - q;
  int L[n1], R[n2];

  for (int i = 0; i < n1; i++)
    L[i] = A[p + i];
  for (int j = 0; j < n2; j++)
    R[j] = A[q + 1 + j];

  int i = 0, j = 0, k = p;
  while (i < n1 && j < n2) {
    if (L[i] <= R[j]) {
      A[k++] = L[i++];
    } else {
      A[k++] = R[j++];
    }
  }

  while (i < n1) {
    A[k++] = L[i++];
  }

  while (j < n2) {
    A[k++] = R[j++];
  }
}

void Ordenador::mergeSortAux(int *A, int p, int r) {
  if (p < r) {
    int q = (p + r) / 2;
    mergeSortAux(A, p, q);
    mergeSortAux(A, q + 1, r);
    merge(A, p, q, r);
  }
}

void Ordenador::mergesort(int *A, int n) {
  mergeSortAux(A, 0, n - 1);
}
```