Algoritmos de Ordenamiento

Ericka Araya-C20553

***Resumen*—En este trabajo, se evaluó el rendimiento de tres algoritmos de ordenamiento: selección, inserción y mezcla, mediante la ejecución de experimentos con arreglos de diferentes tamaños. Las gráficas resultantes mostraron las tendencias esperadas, reflejando las diferencias en la complejidad temporal teórica de los algoritmos. Se observó que el algoritmo de mezcla exhibió una mejor escalabilidad y consistencia de rendimiento en comparación con los algoritmos de selección e inserción. Estos hallazgos proporcionan una comprensión más profunda del rendimiento relativo de los algoritmos de ordenamiento en función del tamaño del arreglo.**

***Palabras clave*—ordenamiento, selección, inserción, mezcla, arreglo, algoritmo,**

I. Introducción

En este trabajo, se reporta y analiza los experimentos realizados para observar el rendimiento de tres de los algoritmos de ordenamiento: selección, inserción y mezcla. En las pruebas se utilizando arreglos de diferentes tamaños (50.000, 100.000, 150.000, 200.000 elementos) y se registraron los tiempos de ejecución en cinco corridas para cada tamaño del arreglo. Esto con el fin de ver como estos algoritmos se comportan ante arreglos con numeros y tamaños diferentes.

El artículo en general se divide en dos partes; la primera parte es analizar cada algoritmo por separado, sobre si los tiempos de ejecución tuvieron consistencia con el mismo tamaño del arreglo, y cuanta diferencia de promedio de duración hay entre cada cantidad distinta de numeros.

La segunda parte se encarga de comparar los tiempos promedio de los tres algoritmos. Ambas partes hechas con el objetivo comprender cómo varían los tiempos de ejecución de los algoritmos de diferentes configuraciones de entrada y cómo estos resultados pueden informar decisiones de diseño en aplicaciones del mundo real.

II. Metodología

Para lograr lo propuesto se implementaron los algoritmos de selección, inserción y mezcla en un entorno de desarrollo en C++ utilizando un enfoque basado en funciones miembro de una clase llamada Ordenador. Cada algoritmo se diseñó para ordenar arreglos de enteros en orden ascendente.

Se creó una función para generar arreglos de enteros aleatorios. Esta función utiliza la función rand() de la biblioteca estándar de C para generar números aleatorios y establece una semilla inicial utilizando la función srand() basada en el tiempo actual.

El código se muestra en los apéndices. Se diseñó un procedimiento experimental para evaluar el rendimiento de los algoritmos de ordenamiento. El código de los algoritmos de ordenamiento se basó en el seudocódigo proporcionado en los libros "Data Structures and Algorithms"de Aho, Hopcroft y Ullman [1], así como en Ïntroduction to Algorithms"de Cormen, Leiserson, Rivest y Stein. [2].

Se utilizó la biblioteca <chrono> de C++ para medir los tiempos de ejecución de los algoritmos con una precisión de milisegundos. Se registraron los tiempos de cada ejecución de los algoritmos y se calculó el tiempo promedio de ejecución para cada tamaño de arreglo. Se creo una tabla para guardar estos datos y se hicieron graficas basadas en estos registros, para poder analizar con más claridad cuanta diferencia hay entre los tiempos promedio, observando las curvas producidas.

III. Resultados

Los tiempos de ejecución de las X corridas de los algoritmos se muestran en el cuadro I.

Cuadro I
Tiempo de ejecución de los algoritmos.

		Tiempo (ms)					
		Corrida					
Algoritmo	Tam. (k)	1	2	3	4	5	Prom.
Selección	50000	2981	2979	2934	2928	2938	2952
	100000	11756	11922	11937	11879	11654	11829
	150000	26505	26166	26402	26416	26088	26315
	200000	46502	46485	46927	46473	46345	46546
Inserción	50000	1664	1613	1616	1624	1624	1628
	100000	6499	6453	6504	6479	6473	6481
	150000	14595	14613	14578	14560	14617	14592
	200000	25876	26771	25902	26122	26886	26311
Mezcla	50000	13	12	11	14	14	13
	100000	26	25	22	24	25	24
	150000	38	33	36	33	37	35
	200000	45	44	42	43	44	43

En el caso del algoritmo de selección, se observa que los tiempos de ejecución varían considerablemente entre las cinco corridas para cada tamaño de arreglo. Por ejemplo, para un tamaño de arreglo de 50,000 elementos, los tiempos oscilan entre 2,928 ms y 2,981 ms. Esta variabilidad se observa también en los otros tamaños de arreglos. Aunque los tiempos promedio tienden a ser consistentes entre las corridas para un mismo tamaño de arreglo, la variabilidad es notable.

Para el algoritmo de inserción, también se observa variabilidad en los tiempos de ejecución entre las corridas, aunque en menor medida que en el algoritmo de selección. Los tiempos oscilan en un rango más estrecho y los promedios son más consistentes entre las corridas para un mismo tamaño de arreglo.

Por último para el de mezcla, los tiempos de ejecución son más consistentes entre las corridas, con una variabilidad mínima. Los tiempos promedio son bastante estables para cada tamaño de arreglo. Para los tres algoritmos se observa que

entre más números se deban ordenar, mayor va a ser el tiempo de duración para hacerlo.

Los tiempos promedio se muestran gráficamente en las figuras 1, 2, 3.

Entrando en algoritmos de ordenamiento, se debe saber que estos tienen diferentes complejidades temporales teóricas. Por ejemplo, los algoritmos de selección e inserción tienen complejidades cuadráticas O(n^2), mientras que el mezcla tiene una complejidad O(nlogn). Las gráficas pueden reflejar estas diferencias en la forma en que los tiempos de ejecución aumentan con el tamaño del arreglo. Por ejemplo, es de esperar que los algoritmos cuadráticos muestren un aumento más pronunciado en los tiempos de ejecución a medida que el tamaño del arreglo aumenta, mientras que los algoritmos O(nlogn) pueden tener un crecimiento más gradual.

Las curvas generadas por las gráficas son consistentes con lo esperado porque reflejan las propiedades teóricas de los algoritmos de ordenamiento, sus eficiencias relativas y las características específicas de su implementación. Esto proporciona una validación empírica de las expectativas teóricas y ayuda a comprender mejor el rendimiento de los algoritmos en diferentes escenarios.

Las curvas se muestran de forma conjunta en la figura 4.

Observando las curvas, se puede notar que las curvas de los algoritmos de selección e inserción muestran un aumento más pronunciado en los tiempos de ejecución a medida que aumenta el tamaño del arreglo, mientras que la curva del algoritmo de mezcla muestra un crecimiento más gradual. Esto sugiere que el algoritmo de mezcla tiene una mejor escalabilidad en comparación con los algoritmos de selección e inserción, especialmente para arreglos de mayor tamaño.

Comparando los tiempos de ejecución promedio para un tamaño de arreglo dado, se observa que el algoritmo de mezcla tiene consistentemente los tiempos de ejecución más bajos, seguido por el algoritmo de inserción y luego el algoritmo de selección.

A lo largo de los diferentes tamaños de arreglo, se observa que la curva del algoritmo de mezcla es más estable y consistente en comparación con las curvas de los algoritmos de selección e inserción. Esto sugiere que el algoritmo de mezcla tiene un rendimiento más predecible y menos sensible a las variaciones en el tamaño del arreglo.

En términos de escalabilidad, se puede concluir que el algoritmo de mezcla es el más escalable, ya que mantiene tiempos de ejecución bajos incluso para arreglos de gran tamaño. Por otro lado, los algoritmos de selección e inserción muestran un deterioro más rápido en el rendimiento a medida que aumenta el tamaño del arreglo, lo que indica una escalabilidad limitada en comparación con el algoritmo de mezcla.

IV. Conclusiones

Se puede concluir basados en los resultados de los experimentos, que la variabilidad en los tiempos de ejecución de los algoritmos de ordenamiento depende del algoritmo y del tamaño del arreglo. Mientras que el de selección exhibe una mayor variabilidad, el de inserción muestra una variabilidad moderada y mezcla muestra una variabilidad mínima. Esto sugiere que, en general, mezcla puede ser más confiable en términos de tiempo de ejecución predecible, especialmente para tamaños de arreglos más grandes. Sin embargo, la elección del algoritmo adecuado depende del contexto específico de la aplicación y de las características del conjunto de datos.

Referencias

[1] A. Alfred V, H. John E, and U. Jeffrey D, *Data Structures and Algorithms*, 1st ed. Micheal A, Harrison, 1983.

[2] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 4th ed. The MIT Press, 2009.

Created by Miguel Rocha
from Noun Project

Ericka Me considero alguien a quien le gusta la carrera que está estudiando, tengo otros intereses como el fútbol, hacer hiking, escuchar música. Me considero alguien feliz y con mucha perseverancia.

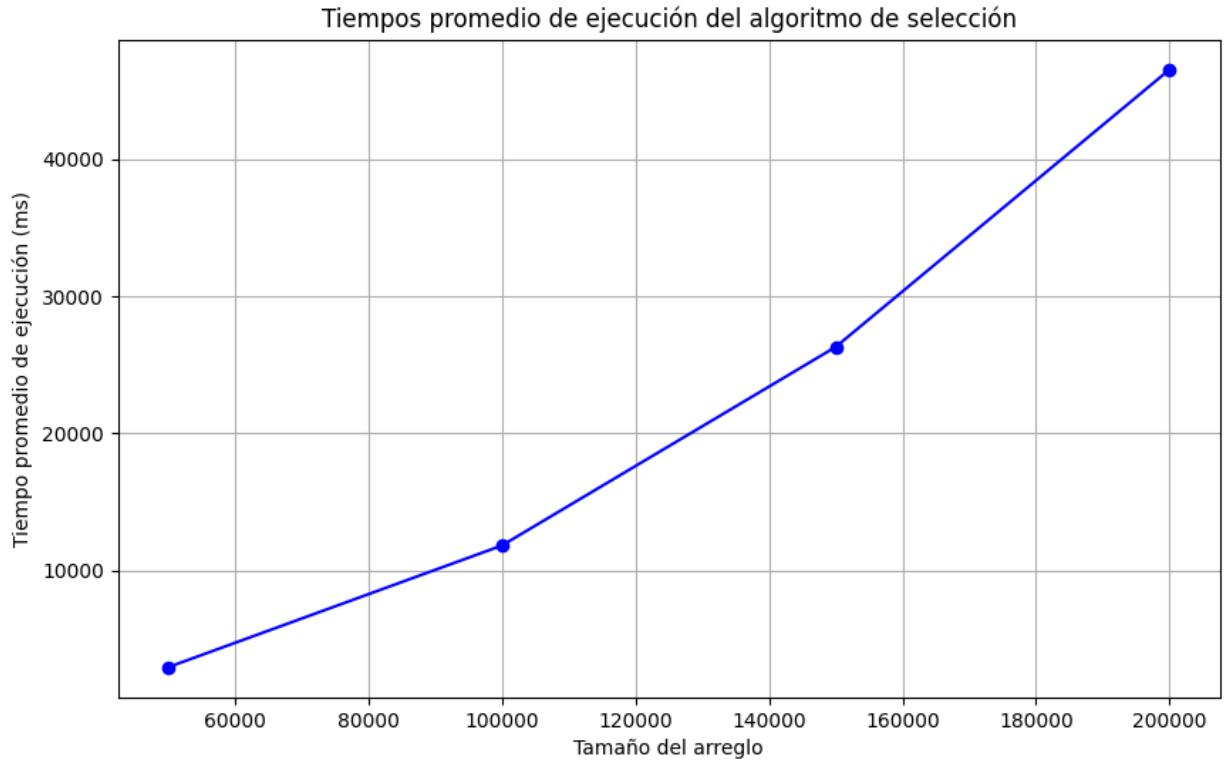

Figura 1. Tiempos promedio de ejecución del algoritmo de ordenamiento por selección.

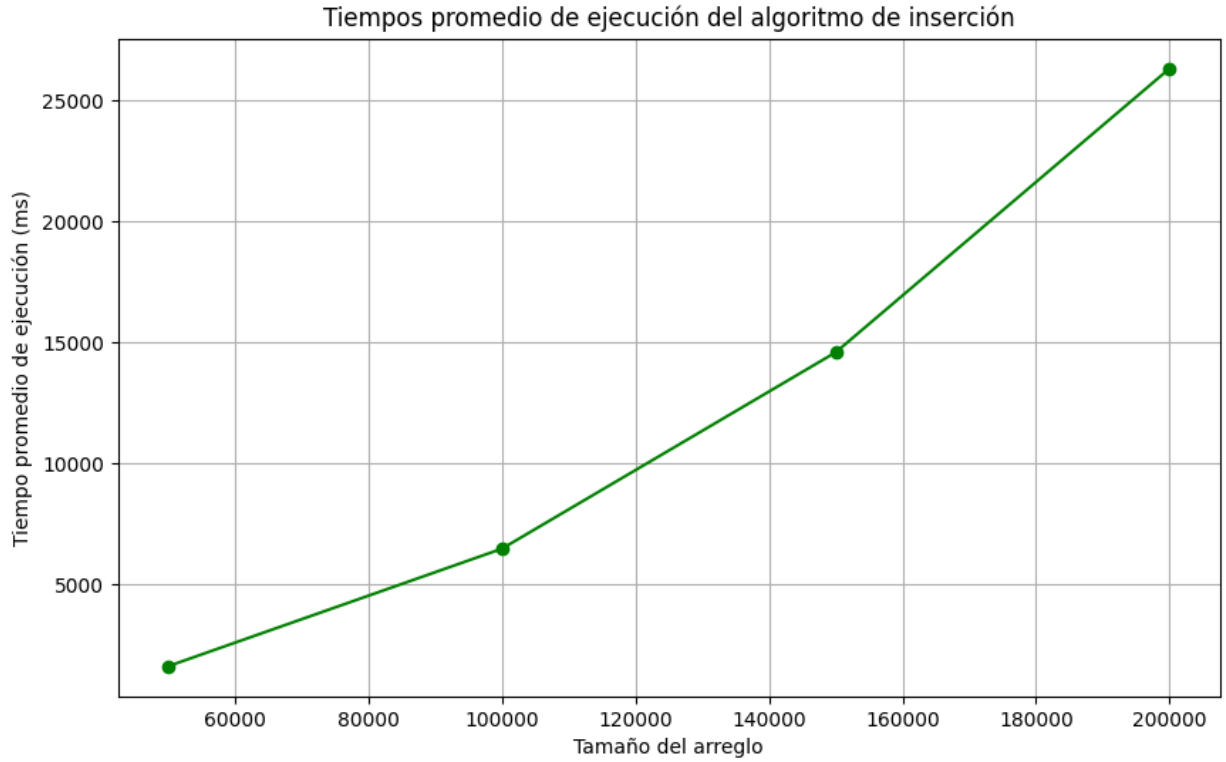

Figura 2. Tiempos promedio de ejecución del algoritmo de ordenamiento por inserción.

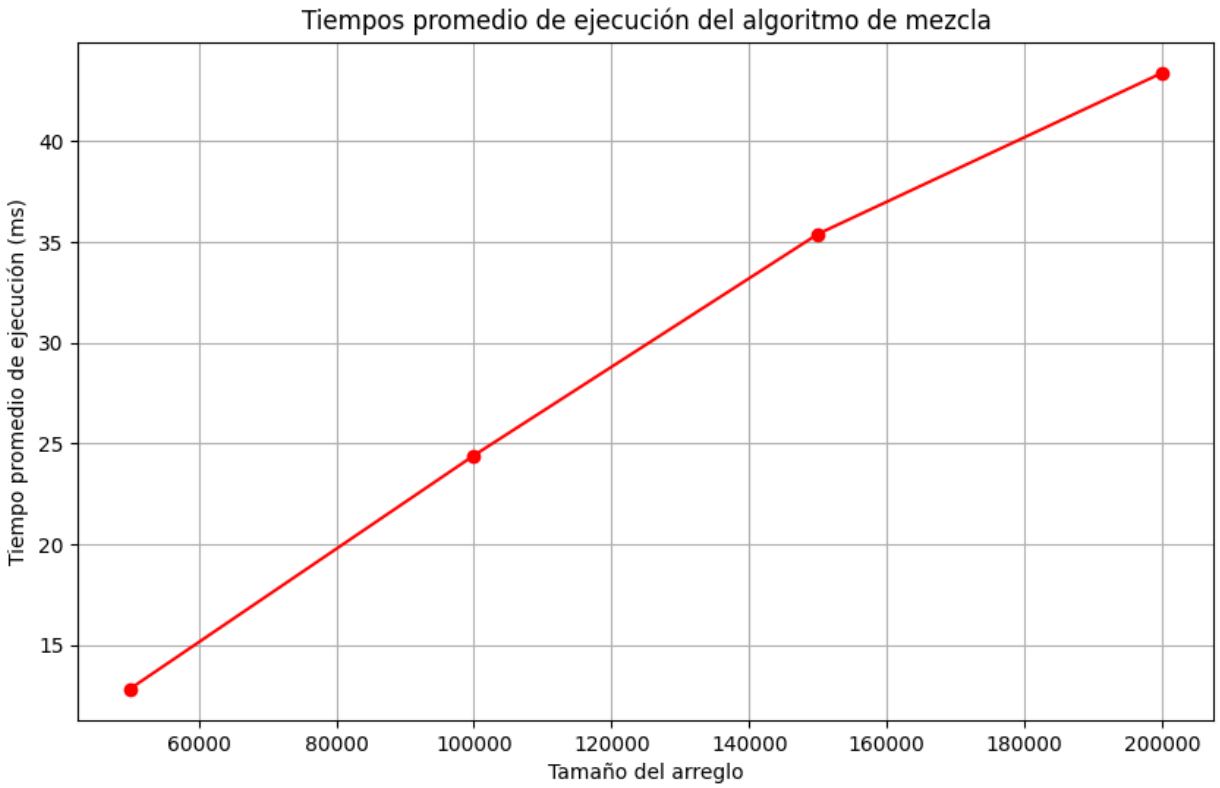

Figura 3. Tiempos promedio de ejecución del algoritmo de ordenamiento por mezcla.

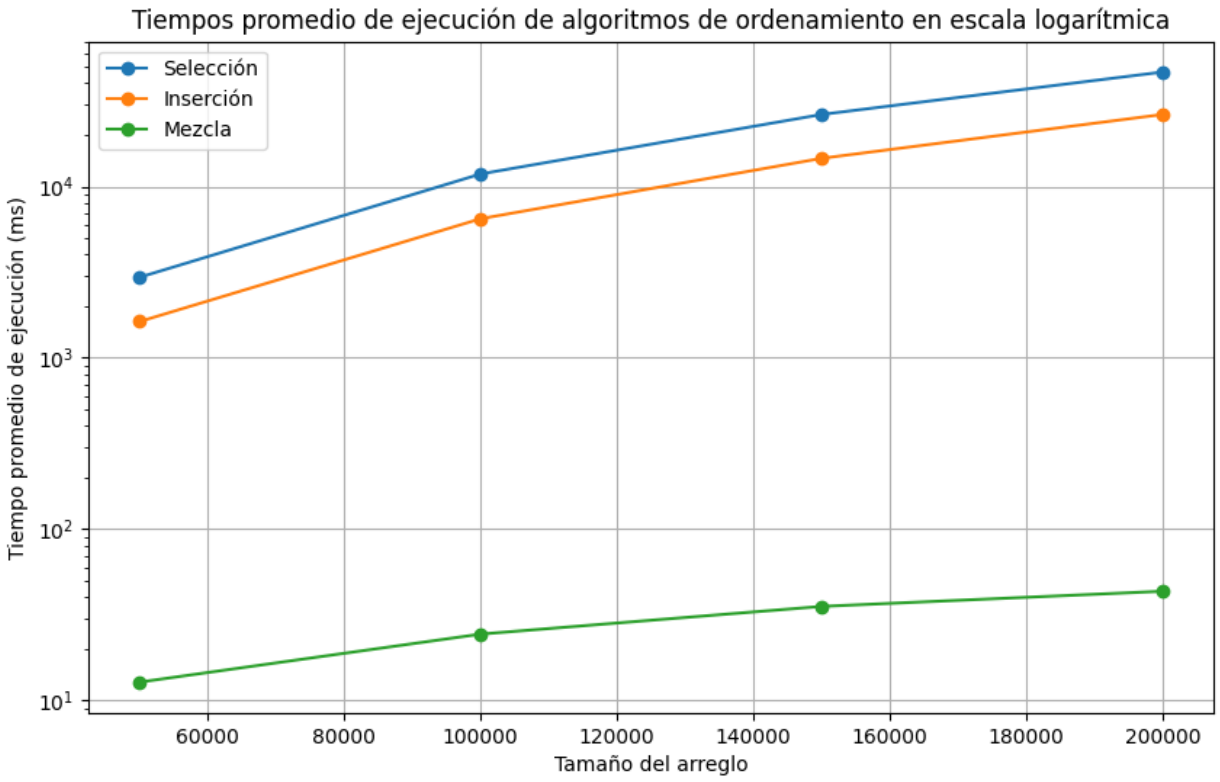

Figura 4. Gráfico comparativo de los tiempos promedio de ejecución de algoritmos de ordenamiento.

APÉNDICE A
CÓDIGO DE LOS ALGORITMOS

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

Algoritmo 1 Algoritmo que recorre repetidamente el arreglo para encontrar el elemento más pequeño y lo coloca en la posición correcta. A medida que avanza, va seleccionando el siguiente elemento más pequeño y lo intercambia con el elemento actual en la posición correcta:

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

Algoritmo 2 Algoritmo que recorre el arreglo desde el segundo elemento hasta el final, insertando cada elemento en su posición correcta entre los elementos ya ordenados. Para hacer esto, compara cada elemento con los elementos previos hasta encontrar su posición correcta. :

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

Algoritmo 3 Algoritmo que utiliza la estrategia de dividir y conquistar. Divide el arreglo en dos mitades, ordena cada mitad de forma recursiva y luego combina las dos mitades ordenadas en un solo arreglo ordenado. Para combinar las mitades, compara los elementos uno por uno y los coloca en orden en un nuevo arreglo auxiliar:

```
void mergesort(int *A, int n){
    mergeSortHelper(A, 0, n - 1);
}
void mergeSortHelper(int *A,
    int left, int right){
  if (left < right) {
    int mid = left + (right - left) / 2;

    mergeSortHelper(A, left, mid);
    mergeSortHelper(A, mid + 1, right);

    merge(A, left, mid, right);
  }
}

void merge(int *A, int left,
    int mid, int right){
  int n1 = mid - left + 1;
  int n2 = right - mid;

  int L[n1], R[n2];

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

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

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

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