Análisis de algoritmos de ordenamiento

Jean Aguilar Mena, B70134, jean.aguilarmena@ucr.ac.cr

***Resumen*—En este trabajo se aborda el estudio y comparación de distintos algoritmos de ordenamiento, como selección, inserción, mezcla, montículos, rápido y residuos, con el objetivo de analizar su desempeño frente a distintos tamaños de arreglos. Para esto, se generaron arreglos aleatorios de distintos tamaños y se midieron los tiempos de ejecución de cada algoritmo en 3 corridas, calculando ele promedio para cada caso. El resultado fue que los algoritmos de tipo cuadrático, como selección e inserción, mostraron un aumento considerable en los tiempos conforme crecía el tamaño de los arreglos, mientras que algoritmos más eficientes, como mezcla, montículos y rápido, presentaron incrementos mucho más estables, asimismo, el ordenamiento por residuos se mantuvo rápido en todos los tamaños evaluados. Se concluye que la elección del algoritmo depende del tamaño de los datos, siendo los métodos eficientes más adecuados para grandes volúmenes de información y los simples útiles para arreglos pequeños o debido a que no desentonan en rendimiento y son sencillos de entender y desarrollar.**

***Palabras clave*—ordenamiento, selección, inserción, mezcla, rápido, residuos**

I. Introducción

En este trabajo se realiza un análisis comparativo del rendimiento de distintos algoritmos de ordenamiento, con el propósito de evaluar cómo se comportan frente a conjuntos de datos de diferentes tamaños y características. Este estudio busca contrastar los tiempos de ejecución observados con los resultados teóricos, identificando patrones de rendimiento característicos de cada algoritmo y evaluando, para cada uno, su eficiencia en escenarios prácticos.

En esta línea, se pretende verificar, por ejemplo, si los algoritmos de orden definido presentan un crecimiento más cercano a la curva que describe su función. De manera complementaria, se analiza la consistencia de los resultados ante múltiples ejecuciones, considerando la posible variación en los tiempos de procesamiento debido a la naturaleza aleatoria de los datos, así como el comportamiento relativo entre algoritmos de distinta naturaleza.

El análisis no solo permite validar las predicciones teóricas, sino también identificar fortalezas y limitaciones prácticas de cada método de ordenamiento. Esto resulta especialmente relevante al considerar la escalabilidad y eficiencia en la manipulación de grandes volúmenes de datos, lo que puede influir directamente en la elección del algoritmo más adecuado según los requerimientos específicos de cada aplicación. De esta manera, el trabajo proporciona una visión integral del comportamiento de los algoritmos, combinando perspectiva teórica y evidencia empírica, y contribuye a una comprensión más profunda de los factores que afectan su rendimiento.

Ante esto se propone abarcar 2 objetivos claros:

Objetivos

1. **Contrastar la eficiencia teórica:** Contrastar la eficiencia teórica de los algoritmos de ordenamiento con los resultados obtenidos experimentalmente, analizando cómo se comportan frente a distintos tamaños y características de datos.
2. **Identificar fortalezas y limitaciones:** Identificar las fortalezas y limitaciones prácticas de cada algoritmo de ordenamiento, evaluando su desempeño, estabilidad y aplicabilidad en escenarios de manejo de grandes volúmenes de datos.

II. Metodología

Para la realización del análisis se generaron distintos conjuntos de datos de entrada con el fin de evaluar de manera práctica el desempeño de los algoritmos de ordenamiento seleccionados. Los arreglos fueron construidos a partir de números enteros pseudoaleatorios, garantizando que cada ejecución partiera de datos desordenados y sin patrones predefinidos que pudieran favorecer a un algoritmo en particular.

Se definieron cuatro tamaños de arreglos: 1 000, 10 000, 100 000 y 1 000 000 elementos. En todos los casos, los valores de los elementos se generaron dentro del rango de 0 hasta $2^{32} - 1$, correspondiente al máximo representable en 32 bits sin signo. Cabe destacar que, aunque el rango completo de un entero sin signo de 32 bits es 0 a $2^{32} - 1$, en este estudio se utilizó 0 como valor mínimo posible, mientras que el máximo estuvo acotado por dicho límite.

Con el fin de asegurar la equidad en la comparación, cada arreglo fue generado una sola vez y posteriormente almacenado en un archivo de texto plano, txt. De esta manera, al momento de ejecutar los algoritmos, se pudo importar exactamente el mismo arreglo y aplicarlo a cada método en idénticas condiciones de desorden inicial. Esta decisión metodológica permitió eliminar sesgos y garantizar que las diferencias observadas en los tiempos de ejecución respondieran exclusivamente al desempeño propio de cada algoritmo, para ello se creo una función que recibe un valor, que representa el tamaño de array a generar, una función para guardar el array generado en un txt y otra para cargarlo desde ahí.

Cada algoritmo fue ejecutado tres veces sobre cada arreglo, registrándose la duración de cada corrida en milisegundos. Se seleccionó esta unidad de tiempo por ser lo suficientemente precisa y representativa para apreciar las variaciones de rendimiento entre algoritmos, además de que era una unidad que se encontraba un poco en el medio para poder medir cada una de las variaciones, ya que unidades muy pequeñas de tiempo, brindarían datos muy grandes, difíciles de seguir y con datos muy grandes, se perdería en parte las variaciones reales entre algoritmos con rendimientos similares, principalmente para los más eficientes. Posteriormente, a partir de los valores obtenidos, se calculó el promedio de cada conjunto de corridas, con el objetivo de disponer de una medida central que reflejara

Cuadro I
TIEMPO DE EJECUCIÓN DE LOS ALGORITMOS.

		Tiempo (Unidad)			
		Corrida			
Algoritmo	Tam. (k)	1	2	3	Prom.
Ordenamiento por Selección	1000	1	2	1	1,33
	10000	130	124	121	125
	100000	9446	9404	9600	9483,33
	1000000	949476	1355417	960069	1088320,67
Ordenamiento por Inserción	1000	0	0	0	0
	10000	10	14	19	14,33
	100000	830	803	810	814,33
	1000000	82121	80508	83458	82029
Ordenamiento por Mezcla	1000	0	0	0	0
	10000	1	3	3	2,33
	100000	25	20	20	21,67
	1000000	124	124	123	123,67
Ordenamiento por Montículos	1000	0	0	0	0
	10000	0	1	1	0,67
	100000	18	15	16	16,33
	1000000	119	119	124	120,67
Ordenamiento Rápido	1000	0	0	0	0
	10000	0	1	1	0,67
	100000	11	9	9	9,67
	1000000	57	57	57	57
Ordenamiento por Residuos	1000	0	0	0	0
	10000	0	0	0	0
	100000	1	0	0	0,33
	1000000	4	4	4	4

de manera más confiable el comportamiento de cada algoritmo frente a los diferentes tamaños de datos.

En el caso particular del algoritmo de ordenamiento por residuos (Radix Sort), se realizaron dos configuraciones de prueba: una utilizando el valor de r derivado de la fórmula $\lfloor \log_2(n) \rfloor$, considerado como el valor ideal según la teoría, y otra empleando el valor de r que resultó más eficiente en la práctica. Esta doble aproximación permitió contrastar la efectividad de la parametrización teórica frente a la experiencia empírica y valorar su impacto directo en el tiempo de ejecución, además, a modo de experimentar, se ejecutó con otros valores como 8, 10, 21 para apoyar un poco el análisis del r = 16 y el r = 19 y sustentar más la idea de que ambos son buenos valores para r, en este contexto.

Con este procedimiento se obtuvo un conjunto de resultados comparables, que sirvieron como base para el análisis gráfico y la discusión posterior acerca de la coherencia entre los tiempos experimentales y la complejidad temporal teórica de los algoritmos estudiados.

III. RESULTADOS

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

El cuadro 1 tiene los resultados de las 3 corridas de cada uno de los algoritmos con las 4 distintas cantidades de datos, mostrando de forma numérica la diferencia marcada entre un algoritmo y otro. Los tiempos están en unidades de milisegundos, unidad lo suficiente pequeña para evidenciar la diferencia entre uno y otro, pero lo suficientemente grande para no tener datos extremadamente grandes. Es importante, destacar que estos resultados provienen de ejecutar cada algoritmo 3 veces por tamaño de array, pero que entre cada corrida, no se modificó el array, por lo que las 3 corridas para cada tamaño son exactamente con la misma entrada de datos para todos los algoritmos.

Es importante aclarar que los rangos de los gráficos a continuación, están en una escala logarítmica, esto para poder tener una representción en un rango más adecuado para visualizar las diferencias entre una curva u otra.

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

En la figura 1, se observa que los tiempos promedio de ejecución del algoritmo de ordenamiento por selección aumentan conforme crece el tamaño del arreglo. Para 1,000 elementos, las ejecuciones son prácticamente inmediatas, mientras que para 100,000 elementos los tiempos ya se encuentran en el rango de varios miles de milisegundos, y para 1,000,000 de elementos los tiempos superan ampliamente el millón de milisegundos en algunas ejecuciones. Se puede apreciar una curva muy pronunciada a partir de un cierto rango de datos.

En este caso el rango en tiempos se dejó en 50 000 y el rango del tamaño del array si se dejó en su totalidad, para poder apreciar mejor el comportamiento de la curva y como esta se va asemejando a una parábola.

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

En la figura 2, se observa que los tiempos promedio de ejecución aumentan a medida que crece el tamaño del arre-

glo. Para 1,000 elementos, las ejecuciones son prácticamente inmediatas. Con 10,000 elementos los tiempos son de unos pocos milisegundos, para 100,000 elementos se acercan a los 800 milisegundos, y para 1,000,000 de elementos superan los 80,000 milisegundos.

En este caso el rango en tiempos se dejó en 50 000 y el rango del tamaño del array si se dejó en su totalidad, para poder apreciar mejor el comportamiento de la curva y como esta se va asemejando a una parábola.

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

En la figura 3, se puede observar el comportamiento del algoritmo de ordenamiento por mezcla, en la cual se observa que los tiempos promedio de ejecución también crecen con el tamaño del arreglo, pero de manera mucho más gradual. Para 1,000 elementos son inmediatos, con 10,000 elementos apenas unos pocos milisegundos, para 100,000 elementos alrededor de 20 milisegundos, y para 1,000,000 de elementos se mantienen cerca de 124 milisegundos.

En este caso, el rango del tiempo en milisegundos va de 0 a 200, esto para poder apreciar un poco mejor el crecimiento que tiene el algoritmo al incrementar el tamaño de datos del eje x, aunque en una escala mucho más reducida que los anteriores.

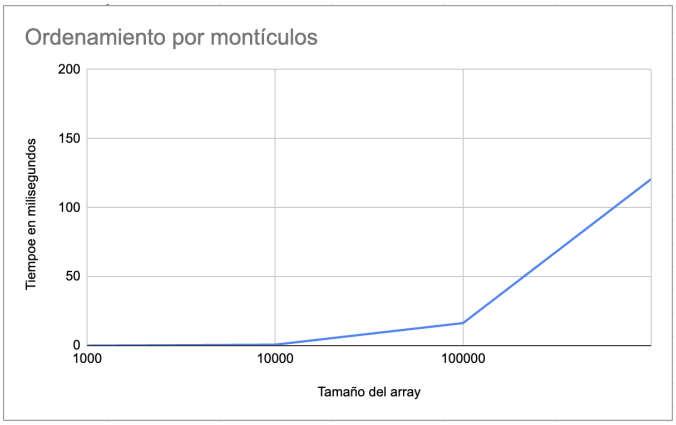


Figura 4. Tiempos promedio de ejecución del algoritmo de ordenamiento por montículos.

En la figura 4, se observa que los tiempos de ejecución aumentan conforme crece el tamaño del arreglo. Para 1,000 y 10,000 elementos las ejecuciones son prácticamente inmediatas. Con 100,000 elementos los tiempos alcanzan alrededor de 16 milisegundos, y para 1,000,000 de elementos superan los 120 milisegundos.

Se maneja un rango de tiempo promedio de 0 a 200, para poder visualizar el comportamineto real del algoritmo, como a medida que los datos crecen, el tiempo de duración también lo hace pero a un ritmo desacelerado.

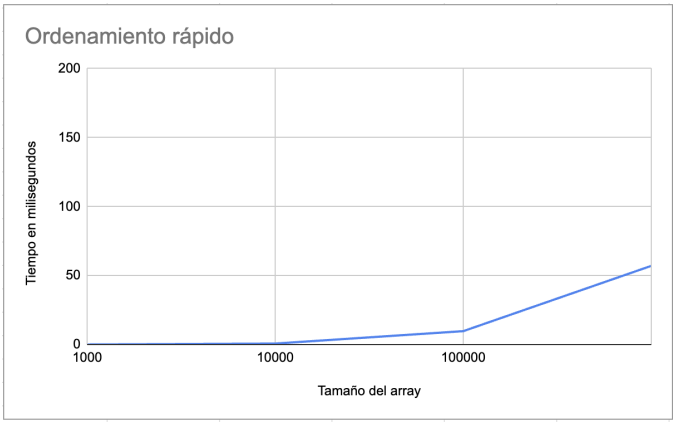


Figura 5. Tiempos promedio de ejecución del algoritmo de ordenamiento rápido.

En la figura 5, se puede observar, que los tiempos de ejecución crecen a medida que aumenta el tamaño del arreglo. Para 1,000 y 10,000 elementos las ejecuciones son prácticamente inmediatas. Con 100,000 elementos los tiempos rondan los 10 milisegundos, y para 1,000,000 de elementos se mantienen alrededor de 57 milisegundos

Se maneja un rango de tiempo en milisegundos muy reducido, de 0 a 200 para poder visualizar en algoritmos muy rápidos, como al tener que procesar estructuras más grandes, esto impacta en el rendimiento, aunque a muy baja escala.

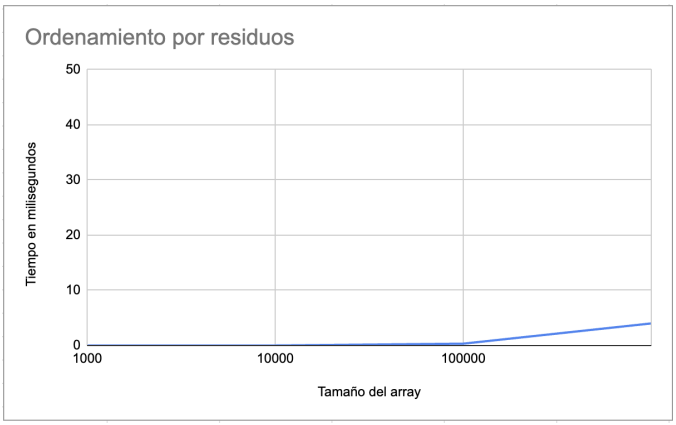


Figura 6. Tiempos promedio de ejecución del algoritmo de ordenamiento por Residuos.

En la figura 6, se observa que los tiempos de ejecución aumentan levemente conforme crece el tamaño del arreglo. Para 1,000 y 10,000 elementos las ejecuciones son inmediatas. Con 100,000 elementos los tiempos son de menos de 1 milisegundo, y para 1,000,000 de elementos alcanzan alrededor de

4 milisegundos. Estos datos mostrados no son los obtenidos con el r ideal calculado a partir de la fórmula, sino que es un r probado para buscar la máxima eficiencia. El r ideal obtenido a partir de la fórmula es 19, mientras que el r a partir del cual se obtuvo los datos anteriores es 16.

Se maneja un rango de tiempo en milisegundos, de 0 a 50, para poder visualizar de forma gráfica el comportamiento del algoritmo cuando se le ingresan datos más grandes.

Las curvas se muestran de forma conjunta en la figura **??**.

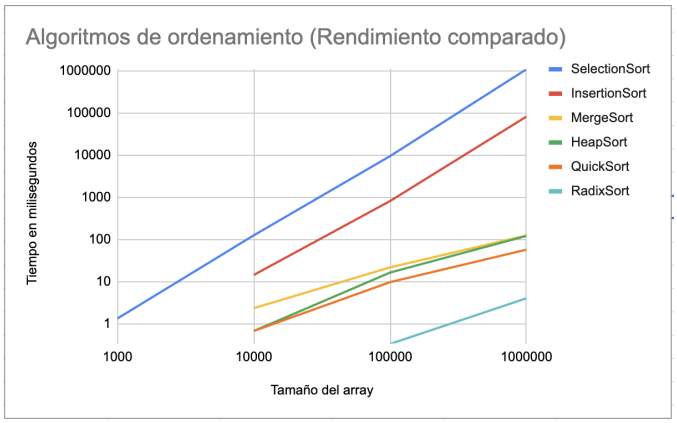


Figura 7. Gráfico comparativo de los tiempos promedio de ejecución de algoritmos de ordenamiento.

Se muestra en la imagen un grafico comparativo entre el rendimiento de cada uno de los algoritmos estudiados en este reporte, este gráfico se muestra en una escala logarítmica para visualizar mejor los distintos comportamientos de los algoritmos y tener un gráfico visual que muestre de una forma clara, las diferencias entre uno y otro, sin tener cosas poco entendibles, debido a la gran diferencia de rendimiento entre unos y otros.

El gráfico muestra en el eje y los tiempos promedio de ejecución y en el eje x, el tamaño de los datos y se muestra claramente como crece la curva de cada uno, considerando precisamente la escala logaritmica en el eje x.

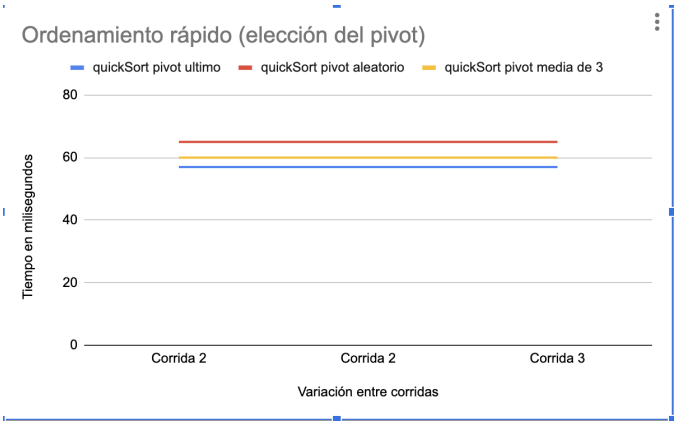


Figura 8. Gráfico comparativo de los tiempos promedio de ejecución del algoritmo de ordenamiento rápido con distintos pivot.

Se muestra un gráfico que tiene los resultados de la ejecución del algoritmo de ordenamiento rápido, con distintas formas de calcular un pivot. El eje y corresponde al tiempo en milisegundos, mientras que el eje x contiene cada una de las 3 corridas que se hizo por cada uno.

La línea azul representa la versión del algoritmo de ordenamiento rápido eligiendo como pivot al último elemento, la línea amarilla representa a la versión del algoritmo eligiendo el pivot como media de 3 y la línea roja representa a la versión eligiendo el pivot de manera aleatoria. Es importante destacar que para esta prueba, se corrió cada algoritmo 3 veces, utilizando todos el mismo array para la corrida 1, cambiando de array para la corrida 2, pero todos utilizando el mismo array en cada corrida.

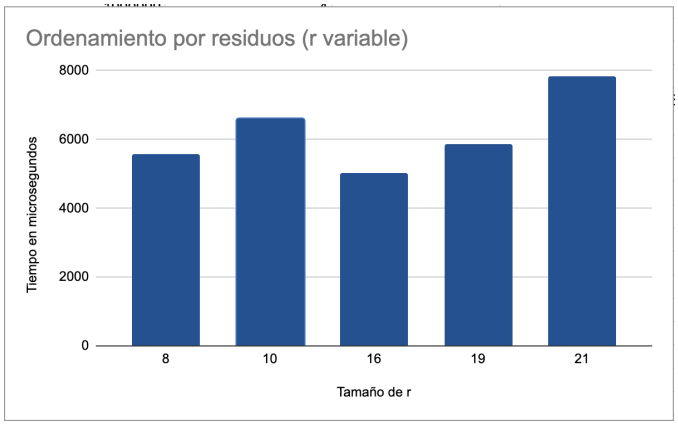


Figura 9. Gráfico comparativo de los tiempos promedio de ejecución del algoritmo de ordenamiento por residuos con distintos valores para r.

Se muestra en la imagen un gráfico comparativo entre distintos resultados obtenidos a partir de ejecutar el algoritmo de ordenamiento por residuos para distintos valores de r, desde valores bajos como lo es r=8 hasta valores altos como r=21, considerando que para este ejercicio el mayor numero de dígitos por número es de 32.

De estas columnas es muy importante notar que en el eje y se maneja el tiempo promedio de cada ejecución, en este caso, en microsegundos, ya que en general, las diferencias eran tan pequeñas, que dejarlas en milisegundos habría perdido riqueza en la comparación. Es importante prestar especial atención a la columna para r=16 que es el valor ideal buscado y la columna para r=19, que es el valor ideal calculado a partir de la fórmula, como el r ideal.

IV. Discusión

Como primer punto, con base en los resultados presentados, se puede observar que los algoritmos se comportaron de manera bastante alineada con lo que predice la teoría de su complejidad temporal. En algunos casos el crecimiento fue exactamente el esperado, mientras que en otros la eficiencia práctica superó lo que se anticipaba teóricamente, sobre todo en los algoritmos más avanzados.

Además del tiempo de ejecución, es importante destacar otras propiedades de los algoritmos. Por ejemplo, Merge Sort es estable, lo cual significa que conserva el orden relativo de elementos iguales; esta característica lo hace valioso en aplicaciones donde la estabilidad es crítica, aunque en los

resultados obtenidos, fue un poco más lento que QuickSort. Heap Sort, en cambio, no es estable pero tiene un rendimiento muy predecible, lo que lo convierte en una opción confiable cuando se necesita un tiempo de ejecución consistente.

Asimismo, los resultados muestran que QuickSort, pese a no ser estable, logra una ventaja considerable en velocidad, y que Radix Sort, aunque depende de parámetros adicionales, es el más eficiente en arreglos grandes cuando se trabaja con enteros de tamaño limitado, es decir, bajo ciertas circunstancias y evidentemente este no es un proyecto lo suficientemente robusto para hacer afirmaciones drásticas respecto a cual es mejor o peor, pero si se intenta, atendiendo a estos resultados, analizarlos y tratar de ver que elementos importantes se pueden concluir a partir de este experimento. Estos matices refuerzan la idea de que no existe un "mejor algoritmo absoluto", sino que la elección depende del contexto y de los datos a procesar.

Comparación con la teoría

En el caso de los algoritmos con complejidad $O(n^2)$, tanto Selección como Inserción mostraron un crecimiento muy evidente en sus tiempos de ejecución a medida que se incrementaba el tamaño del arreglo. Para arreglos de 1000 elementos, sus tiempos son mínimos e incluso despreciables (1,33 ms en Selección y 0 ms en Inserción), tal y como se menciona en Introduction to Algorithms: "El ordenamiento por inserción toma $\Theta(n^2)$ tiempo en el peor de los casos. Sin embargo, debido a que sus bucles internos son compactos, es un algoritmo de ordenamiento rápido para tamaños de entrada pequeños" (Cormen et al., 2022, p. 158). Conforme se aumenta a 100,000 y especialmente a 1,000,000 elementos, los tiempos se disparan de manera exponencial en la práctica: alrededor de 1,088,320 ms en Selección y más de 82,000 ms en Inserción. Este patrón confirma su comportamiento cuadrático, ya que pequeñas variaciones en el tamaño del arreglo generan saltos enormes en el tiempo.

A partir de esta diferencia no se puede aún así, afirmar que uno es mejor que otro, en 2 algoritmos bastante similares, sino, que pudo haber condiciones en los datos que beneficiaron a uno o perjudicaron de más al otro, como lo podría ser el hecho de que ordenamiento por selección siempre realiza exactamente $\frac{n(n-1)}{2}$ comparaciones, si el array tiene bloques ya ordenados, Insertion Sort los aprovecha y reduce drásticamente el trabajo.

En contraste, algoritmos como Merge Sort, Heap Sort y QuickSort sí se ajustaron a la teória $O(n \log n)$. El crecimiento fue mucho más moderado y estable: por ejemplo, con 1,000,000 de elementos, Merge Sort tardó en promedio 123,67 ms, Heap Sort 120,67 ms y QuickSort 57 ms. Tal como señalan Cormen et al. (2022): "Merge sort tiene un mejor tiempo de ejecución asintótico, $\Theta(n \log n)$, pero el procedimiento MERGE que utiliza no opera en el lugar" (p. 158), mientras que "Quicksort también ordena n números en el lugar, pero su tiempo de ejecución en el peor caso es $\Theta(n^2)$. Su tiempo de ejecución esperado es $\Theta(n \log n)$" (p. 159). Estos resultados experimentales confirman que, en escenarios promedio, QuickSort logra un rendimiento competitivo gracias a la elección aleatoria de pivotes, reduciendo la probabilidad de divisiones desbalanceadas.

Finalmente, Radix Sort se comportó como se esperaba: fue el algoritmo más eficiente en los arreglos grandes. En el caso de 1,000,000 elementos, logró ordenar en apenas 4 ms, un resultado que lo ubica muy por encima del resto. Esto concuerda con su naturaleza lineal $O(n \cdot k)$, donde k depende de la cantidad de dígitos. En palabras de Cormen et al. (2022): "RADIX-SORT ordena correctamente en tiempo $\Theta(d(n + k))$ si el algoritmo estable que utiliza toma tiempo $\Theta(n + k)$" (p. 211). Este comportamiento confirma por qué los algoritmos no comparativos, como Counting o Radix, pueden superar la cota inferior $\Omega(n \log n)$ de los algoritmos comparativos, siempre que los datos numéricos tengan un tamaño controlado.

Ahora bien, en la práctica, los resultados también muestran la relevancia de las decisiones de implementación. En QuickSort, por ejemplo, la elección de un pivote aleatorio suele dar lugar a divisiones más balanceadas y, por ende, a un rendimiento cercano a $O(n \log n)$; mientras que el uso de un pivote fijo como el último elemento incrementa el riesgo de caer en el peor caso $O(n^2)$, siempre lo que se busca es un pivote balanceado, tal y como se mencionaba en el libro data structures and algorithms, la idea es "...que el pivote esté cerca del valor medio de las claves en el arreglo, de modo que esté precedido por aproximadamente la mitad de las claves y seguido por la otra mitad."(Aho, A et al, 1983, o.261). De manera similar, en Radix Sort, el parámetro r que define la base de la descomposición influye de forma directa en la cantidad de pasadas y en el costo por iteración, lo que abre la discusión entre un r "óptimo" derivado teóricamente y un r ajustado manualmente según el hardware y la distribución de datos. Estas consideraciones prácticas refuerzan la idea de que el análisis asintótico debe complementarse con experimentación empírica para obtener conclusiones completas sobre el rendimiento real.

Discusión del rendimiento observado

El algoritmo más rápido en casi todos los escenarios fue Radix Sort, sobre todo al llegar a tamaños de 100,000 y 1,000,000 elementos, donde superó ampliamente a los demás. En el caso de los algoritmos de comparación, el QuickSort fue el que mostró el mejor rendimiento, incluso duplicando en rapidez a Merge Sort y Heap Sort en las pruebas con un millón de datos.

No se observaron comportamientos inesperados. Los algoritmos cuadráticos cumplieron con su rol de ser poco eficientes en entradas grandes, mientras que los algoritmos subcuadráticos y no basados en comparación se destacaron. La diferencia entre QuickSort con pivote fijo y uno aleatorio se podrá ver en una sección a continuación, pero es bien sabido que la elección del pivote es clave para evitar el peor caso. En estos resultados, QuickSort se mantuvo muy eficiente, lo que sugiere que la implementación utilizada escogía un pivote adecuado.

Discusión del rendimiento de quick sort con distintas formas de obtener un pivot

En este experimento se evaluó el impacto de distintas estrategias de selección de pivote en QuickSort utilizando

un arreglo de un millón de elementos en el rango de 1 a 10,000. Los resultados muestran que el uso del pivote como último elemento produjo tiempos cercanos a los 57 milisegundos, con muy poca variación entre corridas. Cuando se empleó un pivote aleatorio, los tiempos fueron ligeramente superiores, alrededor de 65 milisegundos, con un rango de 65338–65890 µs. Finalmente, con la estrategia de la media de tres elementos se obtuvieron resultados intermedios, en torno a los 60 milisegundos (60333–60372 µs), mostrando también gran estabilidad.

Estos hallazgos confirman lo que se establece en la teoría: la elección del pivote influye directamente en el balanceo de las particiones y, en consecuencia, en la eficiencia del algoritmo. Aunque el pivote aleatorio suele reducir la probabilidad de caer en el peor caso $O(n^2)$, en este escenario particular resultó más costoso que el último elemento, posiblemente por el costo adicional de generar números aleatorios. La estrategia de la media de tres, reconocida como una heurística eficaz para evitar particiones desbalanceadas, mostró un rendimiento bastante competitivo y más consistente que el pivote aleatorio. Estos resultados evidencian que, más allá de la complejidad teórica, la implementación concreta y el entorno de ejecución pueden alterar el rendimiento relativo de cada estrategia.

Variación entre radix sort con diferentes r

En la Figura 9 se muestra el rendimiento del algoritmo de ordenamiento por residuos al variar el número de bits procesados por pasada (r). Se realizaron pruebas con distintos valores de r, tanto escogidos de manera aleatoria como calculados mediante la fórmula teórica. En este sentido, los resultados evidencian que pequeñas variaciones en r producen cambios importantes en el tiempo de ejecución, lo cual pone en evidencia la relevancia de una adecuada parametrización en este algoritmo.

En primer lugar, al emplear valores aleatorios como $r = 8$, $r = 10$ y $r = 21$, los tiempos oscilaron entre 5,571 y 7,835 microsegundos. Estos resultados confirman que, aunque el algoritmo mantiene su eficiencia global, la elección de un r no adecuado puede introducir sobrecostos debido al mayor número de pasadas necesarias o, por el contrario, a la sobrecarga de procesar demasiados bits en una sola pasada.

Un aspecto particularmente interesante se observa con $r = 16$, valor encontrado de manera experimental como el más eficiente. En este caso, el tiempo fue de apenas 5,034 microsegundos, constituyendo el mejor resultado obtenido. Esto se debe a que $r = 16$ permite completar el ordenamiento en exactamente dos pasadas sobre números de 32 bits sin signo, equilibrando de manera óptima la cantidad de iteraciones y el tamaño de los contadores requeridos en cada pasada. En contraste, el valor teórico calculado a partir de la fórmula ($r = 19$) alcanzó un tiempo de 5,880 microsegundos, ligeramente superior al de $r = 16$. Este detalle demuestra que, si bien la teoría brinda una aproximación sólida, en la práctica pueden encontrarse valores que ajusten mejor al entorno real de ejecución, eso si, siempre destacando que esto se puede deber a una situación particular o aleatoria de los arrays utilizados, considerando que se calcula el valor máximo del array para no tener que procesar los 32 bits en caso de no ser necesario.

Finalmente, es importante destacar que las diferencias entre configuraciones de r son más evidentes cuando los resultados se expresan en microsegundos. Al analizar únicamente en milisegundos, estas variaciones resultaban casi imperceptibles, lo que podría llevar a conclusiones equivocadas sobre la relevancia de la elección de r. Esto subraya la importancia de medir con precisión en experimentos donde las diferencias de rendimiento son tan sutiles.

Variación entre corridas

En términos de estabilidad, los tiempos entre ejecuciones fueron bastante consistentes. Por ejemplo, en QuickSort, el tiempo para 1,000,000 elementos fue exactamente el mismo (57 ms) en dos de las tres corridas, y solo varió en 1 ms en otra. En Heap Sort y Merge Sort la variación también fue mínima (2–3 ms de diferencia). En Selección e Inserción, aunque los tiempos fueron mucho mayores, la diferencia entre ejecuciones fue relativamente pequeña en comparación con la magnitud total, lo que confirma que no hubo anomalías.

No hubo necesidad de eliminar corridas, ya que ninguna se salió de lo esperado. La diferencia entre la mejor y la peor corrida fue baja en casi todos los algoritmos, lo que refleja estabilidad en las mediciones.

Observaciones prácticas

En la práctica, queda claro que los algoritmos cuadráticos no son recomendables para grandes volúmenes de datos. Si bien funcionan para arreglos pequeños y son fáciles de implementar, sus tiempos se vuelven imprácticos muy rápido. Los algoritmos como QuickSort, Merge Sort y Heap Sort son opciones mucho más viables para datos de gran tamaño, con QuickSort destacando como el más rápido en este caso.

Por otro lado, Radix Sort resulta una opción excelente en contextos donde los datos a ordenar son enteros o cadenas de longitud fija, ya que su desempeño lineal lo hace prácticamente imbatible en escalas masivas. Sin embargo, su eficiencia depende de la parametrización: si la base o la cantidad de dígitos no se ajusta adecuadamente, puede perder rendimiento.

En términos generales, la teoría predice bastante bien el comportamiento, y lo observado en la práctica coincide con lo esperado, confirmando que la complejidad temporal es un buen indicador de eficiencia real.

Limitaciones

Algunas limitaciones a considerar son que la medición de tiempos depende del sistema operativo, la precisión de la función de reloj usada y la carga del procesador en el momento de la ejecución. Factores externos como procesos en segundo plano pudieron afectar mínimamente los resultados. Sin embargo, dado que la variación entre corridas fue pequeña, se puede concluir que el impacto de estas limitaciones no fue significativo.

V. Conclusiones

Retomando los objetivos de este trabajo, se puede concluir que los resultados experimentales confirmaron, en gran medida, las predicciones teóricas sobre su complejidad, los algoritmos cuadráticos como selección e inserción mostraron un crecimiento exponencial de los tiempos al aumentar el tamaño del arreglo, mientras que los algoritmos $O(n \log n)$ como ordenamiento por mezcla, ordenamiento por montículos y ordenaminto rápido presentaron un crecimiento más moderado y estable. Ordenamiento por residuos, por su parte, evidenció la eficiencia de los algoritmos no comparativos en arreglos grandes de enteros, cumpliendo con su comportamiento lineal esperado.

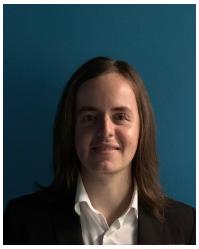
Asimismo, se identificaron fortalezas y limitaciones prácticas de cada algoritmo. Algoritmos como ordenamiento rápido y ordenamiento por residuos se destacaron por su rapidez y eficiencia en grandes volúmenes de datos, aunque ordenamiento por residuos requiere una adecuada parametrización de r para maximizar su rendimiento. ordenamiento por mezcla y ordenamiento por montículos ofrecieron un desempeño estable y predecible, mientras que los algoritmos cuadráticos resultaron poco viables en entradas de gran tamaño. Además, la experimentación permitió observar cómo decisiones de implementación, como la selección del pivote en QuickSort, afectan de manera significativa la eficiencia real del algoritmo.

En conjunto, este análisis evidencia que la elección del algoritmo más adecuado depende del contexto, la naturaleza de los datos y las condiciones de ejecución, y que la comparación entre teoría y práctica es esencial para una comprensión completa del desempeño de los métodos de ordenamiento.

Referencias

Aho, A. V., Hopcroft, J. E., Ullman, J. D. (1983). Data structures and algorithms. Addison-Wesley.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., Stein, C. (2022). Introduction to algorithms (4th ed.). MIT Press.



Jean Aguilar Mena Soy una persona bastante alegre, que le gusta mucho aprender cosas nuevas. Me gusta mucho aprender sobre tecnología y soy un amante de la música. Me gusta hablar y conocer personas, porque me dan una perspectiva diferente, lo cual, en mi caso, siempre es bienvenido.