Algoritmos de Ordenamiento: Tiempos de Duración Ordenando Arreglos de Diferentes Tamaños

José Manuel Mora Zúñiga

***Resumen*—El objetivo del trabajo es medir el tiempo de duración de distintos algoritmos de ordenamiento vistos en el curso, y compararlos con su medida esperada de acuerdo al análisis asintótico de los mismos. Para esto se van a ordenar arreglos de longitud cada vez más grande y se van a graficar los tiempos de duración para poder observar el comportamiento de cada algoritmo, esto con el fin de compararlo con el comportamiento esperado de cada uno.**

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

I. Introducción

Para este trabajo se implementaron 6 algoritmos: Insertion Sort, Selection, Merge Sort, Heap Sort, Quick Sort y Radix Sort, que implementa Counting Sort.

Un breve resumen de cada algoritmo:

Insertion: El Insertion Sort empieza en el segundo elemento del arreglo. Por cada iteración que realiza, toma el elemento en el que se encuentra y busca donde debe insertarlo entre los elementos que están antes de este. Por ejemplo, toma el cuarto elemento del arreglo y busca dónde debe ir entre los 3 elementos anteriores, insertándolo en dicha posición. El sub-arreglo desde el primer elemento hasta la última posición procesada se considera ordenado. El tiempo promedio de este algoritmo es $\Theta(n^2)$.

Selection: El Selection Sort busca el elemento más pequeño en el rango entre la posición de arreglo que se encuentra procesando en ese momento y el último elemento del arreglo, luego lo posiciona en dicha posición y se mueve a la siguiente. De esta forma el sub-arreglo desde el primer elemento hasta la última posición procesada se considera ordenado. El tiempo promedio de este algoritmo es $\Theta(n^2)$.

Merge: El Merge Sort va separando el arreglo en dos mitades recursivamente hasta terminar con dos arreglos de tamaño 1 o 0, el siguiente paso es volver a unir estas mitades pero de forma ordenada, para esto se van comparando los elementos de ambas mitades para determinar cual debe ir primero en el arreglo ordenado, este proceso continúa hasta que todas las mitades estén unidas y el arreglo queda finalmente arreglado. El tiempo promedio de este algoritmo es $\Theta(n \cdot log(n))$.

Heap: El Heap Sort utiliza la estructura del "montículo", que es visualmente similar a un árbol binario. El algoritmo No utiliza un montículo como tal, sino que trata al arreglo como uno, haciendo uso de los índices para conectar los "nodos"(los elementos). En el montículo, los hijos de un nodo se dan por las formulas $2i$ y $2i + 1$ para el hijo izquierdo y el derecho respectivamente, donde i es el índice del nodo "padre". De esto también se obtiene la fórmula $\lfloor \frac{i}{2} \rfloor$ para encontrar al padre de un nodo específico. El procedimiento para ordenar un arreglo utilizando esta información se resume en: 1. Formar un "Montículo Máximo.[es]to es, uno en el que el padre siempre es mayor que ambos hijos, sin importar el orden de estos dos. Lo importante de este paso es dejar el elemento más grande de primero, lo que lleva al paso 2. Intercambiar el primer y último elemento, esto deja el elemento más grande al final (ya ordenado), luego es separado del montículo, moviendo nuestra posición final hacia atrás, para poder ubicar el próximo elemento ahí. Al dejar el elemento más pequeño de primero, se debe volver al paso 1. y volver a maximizar el montículo. Esto se repite hasta que el arreglo queda ordenado. El tiempo promedio de este algoritmo es $\Theta(n \cdot loog(n))$.

Quick: El Quick Sort ordena mediante "particiones". Estas particiones del arreglo se manejan únicamente por índices, lo que evita la necesidad de crear arreglos extra, a diferencia del Merge, que crea arreglos adicionales. Para crear una partición, se debe elegir uno de los elementos (el valor, no su posición) como un pivote, hay varios métodos para elegir un pivote, en mi caso me quedé con la versión simple y elegí el último elemento del arreglo. El pivote se utiliza como "punto medio", no es exactamente la mitad pues se elige al azar, pero es el punto donde se parte el arreglo. Para eso, se utilizan dos índices, uno para recorrer el arreglo y otro para separar los valores mayores al pivote de los menores o iguales a este (<= a la izquierda y > a la derecha). El último paso es insertar el pivote en medio de los dos grupos. Ahora, a estos dos sub-arreglos resultantes, los valores a cada lado del pivote, se les aplica el mismo procedimiento hasta que el sub-arreglo tenga tamaño 1 o 0. El tiempo promedio de este algoritmo es $\Theta(n \cdot log(n))$.

Radix: El Radix Sort es simplemente ordenar el arreglo por "dígitos", esto es, separar los valores del arreglo en partes, estos pueden ser literalmente los dígitos como los conocemos o alguna cantidad arbitraria de bits, por ejemplo. Ya definidos los dígitos, se aplica un algoritmo de ordenamiento estable por cada posición de los dígitos, por ejemplo, usando los digitos en base 10: esto sería ordenar las unidades, luego las decenas y así hasta terminar. Un algoritmo de ordenamiento estable es aquel que preserva el orden en el que aparecen los elementos en el arreglo desordenado, esto es fácilmente ejemplificado en este algoritmo, por ejemplo: si tenemos 49 y 42 en el arreglo, cuando se ordenan las unidades quedarían en este orden: $[42, ..., 49]$ (suponiendo que hay más elementos en medio), al momento de ordenar las decenas, en este caso, ambos son 40, pero como en el arreglo está primero el 42, pues $2 < 9$, un algoritmo estable permite mantener ese orden, quedando así $[42, 49]$ en el arreglo ordenado. El tiempo promedio de este algoritmo es $\Theta(d(n + k))$, d siendo la cantidad de dígitos.

Counting: El Counting Sort es el algoritmo que comúnmente se utiliza para ordenar los dígitos del Radix. Este consiste en crear un arreglo auxiliar para posicionar los elementos en un nuevo arreglo ordenado. Este arreglo auxiliar pasa por diferentes "fases.[en] las cuales sus valores indican cosas distintas. El primer paso es crear el arreglo auxiliar de tamaño k, k es el valor de mayor tamaño en el arreglo original. Por esto es útil separar números más grandes en dígitos, pues entre más grande k, más espacio consume el algoritmo y más dura en ejecutarse. Una vez creado el arreglo, se documenta la frecuencia de cada uno de los dígitos, terminado este paso, el arreglo contiene la cantidad de apariciones de cada dígito en el arreglo original. El siguiente paso es, empezando en la segunda posición del arreglo auxiliar, sumarle el valor de la posición anterior, esto se hace con el valor .ªctualizado", o sea, se toma el valor de la posición anterior luego de que a esta se le haya realizado la suma. Al terminar este paso, el arreglo contiene la cantidad de elementos que son menores o iguales a cada dígito. Esto nos ayudará a posicionar los números en el siguiente paso, el cual es recorrer el arreglo original en reversa, y utilizar el arreglo auxiliar como una tabla de referencia para ir posicionando el valor en el nuevo arreglo ordenado. Para esto, se toma el elemento en el original, se revisa la posición que corresponde al dígito en el auxiliar, el cual contiene la posición en la que se debe colocar el elemento en el nuevo arreglo ordenado. Una vez colocado, se le resta 1 al valor en el arreglo auxiliar, pues en caso de encontrarse con otra aparición del mismo dígito, esta se coloca a la izquierda del que se acaba de colocar, esto y el hecho de que el arreglo original se recorre en reversa, es lo que asegura que el algoritmo es estable. Luego de terminar de recorrer el arreglo original, el nuevo arreglo ordenado estará completo, El tiempo promedio de este algoritmo es $\Theta(n+k)$, donde k es, nuevamente, el tamaño del arreglo auxiliar, el cual es recorrido en dos ocasiones.

La idea del trabajo es medir el tiempo de ejecución de los algoritmos y ver si coincide con la esperada en el análisis asintótico de los mismo.

II. Metodología

Para poder realizar las mediciones, se implementaron los algoritmos en un programa en C++ y se realizaron las pruebas midiendo el tiempo durante la ejecución misma haciendo uso de la biblioteca `chrono`, que es parte de la Biblioteca Estándar de Plantillas de C++ (STL por sus siglas en Inglés).

Para realizar la prueba, se generó un arreglo de un tamaño específico, luego, antes de cada ejecución de los algoritmos, se genera una copia del arreglo original desordenado, la cual es la proporcionada al algoritmo, esto con el fin de que todos los algoritmos tengan que ordenar el mismo algoritmo y no existan disparidades debido a que un algoritmo estuviera más ordenado o algo similar. Además, para estar más seguros de los resultados, cada algoritmo ordenó el mismo arreglo 3 veces. Para las pruebas realizadas, se utilizaron arreglos de 50, 100, 150 y 200 mil elementos.

El programa fue compilado con GCC y se realizaron dos pruebas, una compilando el programa de forma estándar, y otra indicándole al compilador que realizara optimizaciones utilizando la bandera `-O3`, esto disminuyó los tiempos de ejecución en gran medida, pero mantuvo los mismos patrones de crecimiento que el programa sin optimizar.

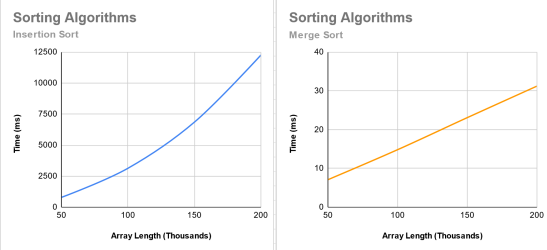

Figura 1. Tiempos promedio de ejecución de los algoritmos de ordenamiento por inserción y mezcla por separado.

Las pruebas fueron realizadas en una laptop Lenovo con 8GB de memoria RAM DDR5 de 5200 MT/s, un CPU Intel Core i5-13420H y corriendo en partición del SDD con Fedora 40 KDE Plasma (Linux).

III. Resultados

Los tiempos de ejecución de las 3 corridas de cada algoritmo se muestran en el cuadro I para los tiempos sin optimizar y en el cuadro II para los tiempos con optimizaciones.

Los resultados fueron obtenidos mediante el proceso descrito en la Metodología. Los tiempos obtenidos fueron impresos en la salida estándar a la terminal para poder ser copiados con facilidad y ser procesados de forma más sencilla. Las gráficas se crearon en Google Sheets.

Los cuadros también incluyen los promedios, los cuales fueron los usados para realizar las gráficas. Estas son: Las figuras 1 y 6 para los algoritmos por Inserción y Mezcla sin y con optimizaciones respectivamente. Los algoritmos por Selección y por Montículos se pueden apreciar en las figuras 2 sin optimizar y 7 con optimizaciones. Para los algoritmos Rápido y por Residuos, las figuras 3 y 8, nuevamente sin optimizar y optimizados respectivamente.

Finalmente, para las gráficas combinadas se hicieron 2 tipos, una con los 6 algoritmos para apreciar la diferencia entre los cuadráticos y los logarítmocos (o menores), y otra solo con estos últimos para poder apreciar mejor las diferencias ente ellos. Las gráficas incluyendo los 6 algoritmos se encuentran en las figuras 4 y 9, la figura 9 corresponde a la versión optimizada. Para la comparación de los algoritmos más rápidos, estas son las figuras 5 y 10, en este caso es la figura 10 la que corresponde a la versión optimizada.

IV. Discusión

Este trabajo permite apreciar claramente las tendencias de los algoritmos. Estos reflejan hasta cierto punto el comportamiento esperado, aunque la teoría solo llega hasta cierto punto, pues, aunque el Selection Sort y el Insertion Sort tengan el mismo tiempo promedio en su análisis asintótico, realizar las pruebas revela una diferencia significativa entre estos dos, siendo el Insertion más rápido.

También se puede apreciar la gran diferencia entre los dos anteriores y los 4 algoritmos restantes, ya que, mientras desde

Cuadro I
TIEMPO DE EJECUCIÓN DE LOS ALGORITMOS SIN OPTIMIZAR.

		Tiempo (ms)			
		Corrida			
Algoritmo	Tam. (k)	1	2	3	Prom.
Selection	50	1073,33	1071,01	1079,93	1074,76
	100	4256,46	4265,48	4256,01	4259,32
	150	9749,87	9772,76	9797,66	9773,43
	200	17292,1	17117,7	17047,9	17152,6
Insertion	50	791,998	789,629	790,384	790,670
	100	3135,61	3140,86	3142,31	3139,59
	150	6807,31	6853,96	6916,56	6859,28
	200	12315,8	12281,5	12196,7	12264,7
Merge	50	7,24855	7,00321	7,02119	7,09098
	100	15,0256	14,7110	14,7661	14,8342
	150	23,3413	23,1157	22,8620	23,1063
	200	31,8727	30,9997	30,8776	31,2500
Heap	50	10,2534	10,2018	10,0237	10,1596
	100	21,5738	21,4606	21,5896	21,5413
	150	34,3166	34,2931	33,4536	34,0211
	200	45,7525	46,7721	46,8987	46,4744
Quick	50	4,99613	4,81915	4,73328	4,84952
	100	9,82995	9,71603	10,0754	9,87379
	150	15,6686	15,4350	15,7706	15,6247
	200	20,6738	20,6762	20,6187	20,6562
Radix	50	5,61949	5,33699	5,51827	5,49158
	100	10,7876	10,7770	10,6891	10,7512
	150	16,0524	15,7967	16,0956	15,9816
	200	21,1658	21,1113	21,0263	21,1011

Cuadro II
TIEMPO DE EJECUCIÓN DE LOS ALGORITMOS CON OPTIMIZACIONES.

		Tiempo (ms)			
		Corrida			
Algoritmo	Tam. (k)	1	2	3	Prom.
Selection	50	450,999	449,940	450,245	450,395
	100	1780,26	1777,55	1778,25	1778,69
	150	4025,09	4019,35	4007,27	4017,24
	200	7098,96	7089,60	7113,74	7100,77
Insertion	50	179,757	167,504	169,844	172,368
	100	666,475	664,661	664,598	665,245
	150	1510,16	1492,00	1492,16	1498,11
	200	2659,35	2657,40	2656,81	2657,85
Merge	50	4,96629	4,80054	4,81503	4,86062
	100	10,2664	10,0176	10,0141	10,0994
	150	17,1048	15,6074	15,5401	16,0841
	200	21,6525	21,1471	21,1040	21,3012
Heap	50	4,10077	4,06520	4,05043	4,07213
	100	8,71234	8,76943	8,67366	8,71848
	150	13,5788	13,6333	13,5926	13,6016
	200	19,1258	18,9227	18,8382	18,9622
Quick	50	2,75852	2,75342	2,76466	2,75887
	100	5,73226	5,72967	5,75171	5,73788
	150	8,98334	8,99751	9,00806	8,99630
	200	12,0172	13,5572	12,2865	12,6203
Radix	50	2,35826	2,18643	2,16373	2,23614
	100	4,45099	4,31981	4,33997	4,37026
	150	6,66563	6,44411	6,42298	6,51091
	200	9,09782	8,85010	8,73678	8,89490

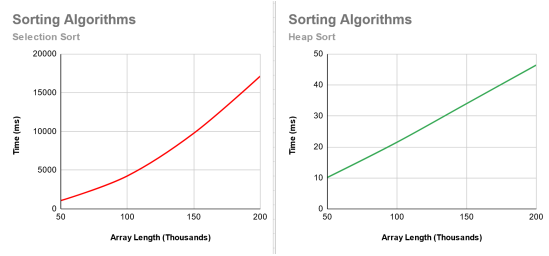

Figura 2. Tiempos promedio de ejecución de los algoritmos de ordenamiento por selección y montículos por separado.

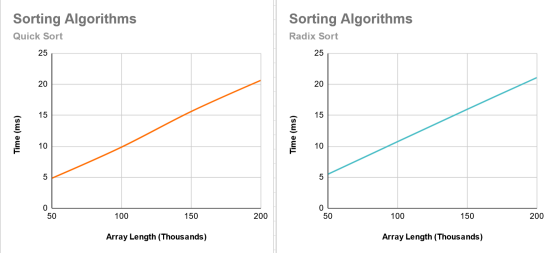

Figura 3. Tiempos promedio de ejecución de los algoritmos de ordenamiento rápido y por residuos por separado.

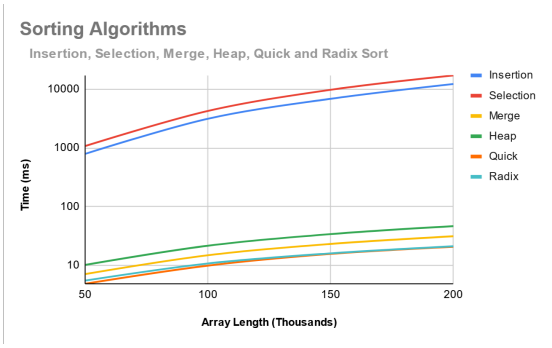

Figura 4. Tiempos promedio de ejecución de Todos los algoritmos probados, en escala logarítmica para poder apreciar los de menor tiempo de ejecución.

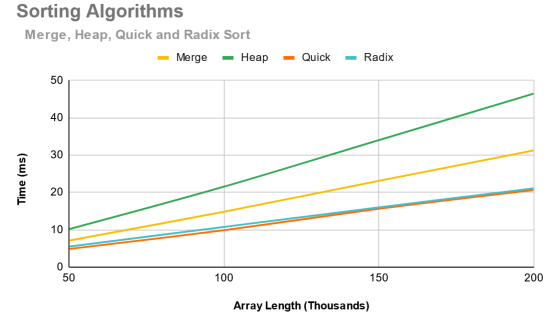

Figura 5. Tiempos promedio de ejecución de los algoritmos de ordenamiento de duración $\Theta(n \cdot log(n))$ o menor. Para poder apreciar mejor las diferencias entre los 4.

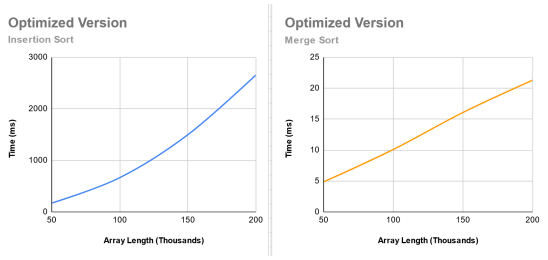

Figura 6. Tiempos promedio de ejecución de los algoritmos de ordenamiento por inserción y mezcla por separado. Con optimizaciones del compilador.

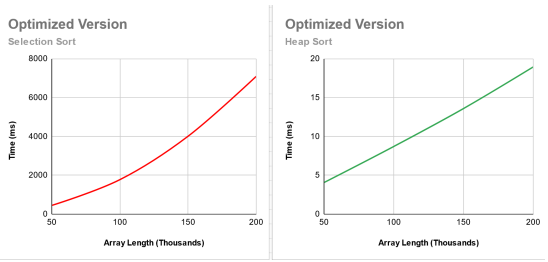

Figura 7. Tiempos promedio de ejecución de los algoritmos de ordenamiento por selección y montículos por separado. Con optimizaciones del compilador.

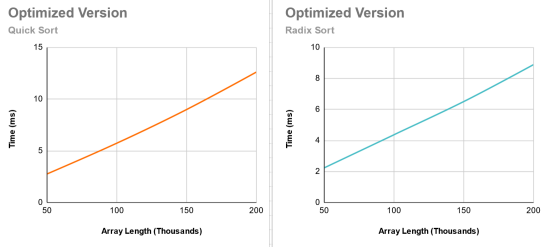

Figura 8. Tiempos promedio de ejecución de los algoritmos de ordenamiento rápido y por residuos por separado. Con optimizaciones del compilador.

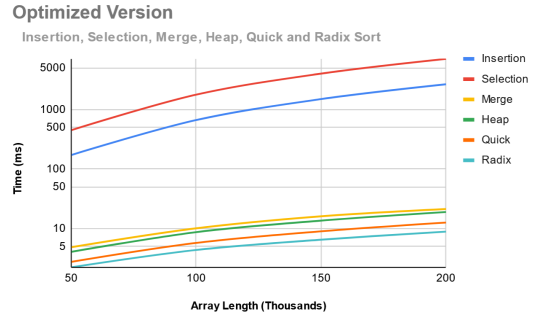

Figura 9. Tiempos promedio de ejecución de Todos los algoritmos probados, en escala logarítmica para poder apreciar los de menor tiempo de ejecución. Con optimizaciones del compilador.

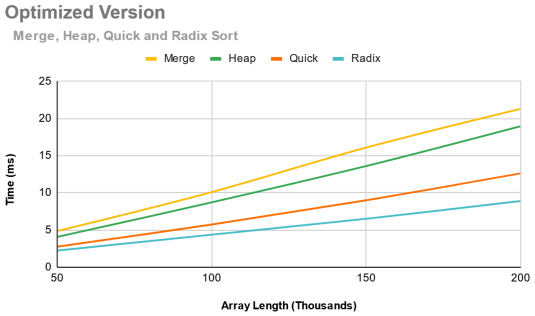

Figura 10. Tiempos promedio de ejecución de los algoritmos de ordenamiento de duración $\Theta(n \cdot log(n))$ o menor. Para poder apreciar mejor las diferencias entre los 4. Con optimizaciones del compilador.

el inicio el Selection y el Insertion estaban durando alrededor de 1 segundo, el Merge, Heap, Quick y Radix duraron 10 milisegundos o menos (esto sin tomar optimizaciones). Tomando en cuenta las optimizaciones, la diferencia es menor, pero igual existe una brecha entre los dos grupos de algoritmos.

Es curioso ver como la diferencia entre los algoritmos de orden $\Theta(nlog(n))$ y el Radix ($\Theta(d(n+k))$) no es tanta, por lo menos en este caso. El experimento se realizó con el tipo de datos `int`, por lo que puede que usando tipos de datos que almacenen números más grandes, como `uint64_t` o `uint32_t` se pueda apreciar más diferencia entre estos.

V. Conclusiones

Los algoritmos de orden cuadrático tienden a ser considerablemente más lentos que aquellos de orden $\Theta(nlog(n))$ o aproximadamente linear, en este caso $\Theta(d(n+k))$.

La notación asintótica nos da una idea aproximada de cuanto tarda un algoritmo en ejecutarse, pero no ofrece diferenciación entre algoritmos del mismo orden, como el Selection Sort y el Insertion Sort, ambos son de orden $\Theta(n^2)$ pero el Selection Sort tarda considerablemente más tiempo que el Insertion.

Referencias

José Manuel Estudiante en la ECCI, UCR.

Created by Miguel Rocha
from Noun Project