Universidad Simón
Bolívar

Reporte de Laboratorio de Semana 4

Análisis de algoritmos de ordenamiento

Autor:
Christopher Gómez

Profesor:
Guillermo Palma

Laboratorio de Algoritmos y Estructuras de Datos II (CI2692)

6 de junio de 2021

1. Metodología

El siguiente estudio experimental consiste en correr un conjunto de algoritmos de ordenamiento con tiempo de ejecución promedio $O(n \lg(n))$ sobre cinco secuencias de diferentes tamaños (10,000, 20,000, 30,000, 40,000 y 50,000 números enteros) y de distintas **clases**, midiendo sus tiempos de ejecución. Cada secuencia consiste en N números enteros almacenados en un objeto de tipo **Array<Int>**. La misma secuencia es copiada luego de ser generada y recibida 3 veces como entrada de cada algoritmo de ordenamiento del conjunto. Solamente se toma el tiempo de ejecución del algoritmo, no son partes de la medición los tiempos de copiar la secuencia ni de verificar que fue ordenada correctamente.

Las clases de secuencias a generar son:

- **random**: Secuencia de N elementos generados aleatoriamente en el intervalo $[0..N]$.
- **sorted**: Secuencia de N elementos generados aleatoriamente en el intervalo $[0..N]$, donde los elementos están ordenados.
- **inv**: Secuencia de N elementos generados aleatoriamente en el intervalo $[0..N]$, donde los elementos están ordenados de forma inversa.
- **cerouno**: Secuencia de N elementos de ceros y unos generados aleatoriamente.
- **mitad**: Secuencia de N elementos generados con la forma $\langle 1, 2, \ldots, \frac{N}{2}, \frac{N}{2}, \ldots, 2, 1\rangle$ si N es par, o $\langle 1, 2, \ldots, \lceil \frac{N}{2} \rceil, \ldots, 2, 1\rangle$ si N es impar.
- **repetido**: Secuencia de N elementos generados aleatoriamente en el intervalo $[0..\lceil N^{\frac{2}{3}} \rceil]$.

Los algoritmos a ejecutar, implementados en el lenguaje de programación Kotlin, serán los siguientes: *Merge Sort*, *Merge Sort Iterativo*, *Heap Sort*, *Quicksort Randomized Quicksort*, *Quicksort with 3-way Partitioning* y *Instrospective Sort (Introsort)*.

Los siguientes son los detalles de la máquina y el entorno donde se ejecutaron los algoritmos:

- **Sistema operativo:** Linux Mint 19.3 Tricia 32-bit.
- **Procesador:** Intel(R) Celeron(R) CPU G1610 Dual-Core @ 2.60GHz.
- **Memoria RAM:** 2,00 GB (1, 88 GB usables).
- **Compilador:** kotlinc-jvm 1.5.0 (JRE 11.0.11+9).
- **Entorno de ejecución:** OpenJDK Runtime Environment 11.0.11

2. Resultados experimentales

2.1. Secuencia 'random'

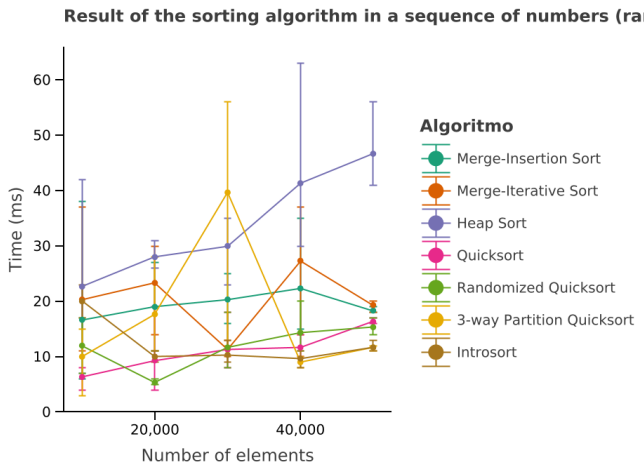

Figura 1: Tiempos de ejecución de los algoritmos $O(n \lg(n))$ en una secuencia de clase 'random'

Los resultados para las secuencias 'random' [Fig. 1] muestran el comportamiento esperado de cada uno de los algoritmos. Se puede notar que (salvo ciertas excepciones), los algortimos basados en *Quicksort* logran un mejor desempeños que los basados en *Merge*, y que el más lento de ellos, aunque del mismo orden de crecimiento respecto al tamaño de la entrada es *Heap Sort*.

Al ser *Quicksort* diseñado para secuencias totalmente aleatorias, se observa que el algoritmo original y la variante con pivoteo aleatorio logran varias veces tiempos de ejecución incluso menores a los de algoritmos híbridos como *Introsort* y versiones optimizadas como *Quicksort with 3-way Partitioning*.

2.2. Secuencia 'sorted'

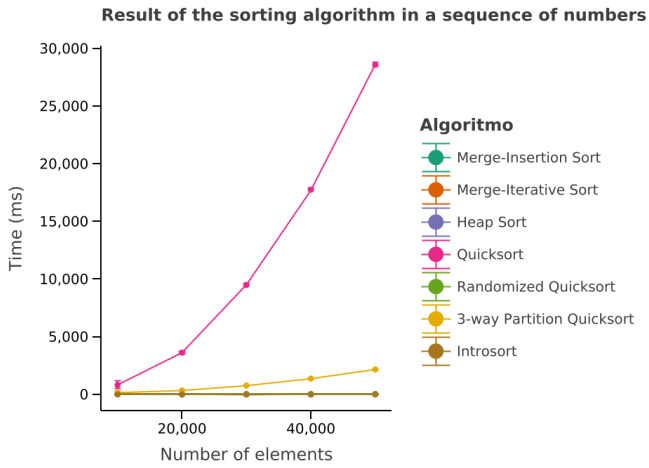

Figura 2: Tiempos de ejecución de los algoritmos $O(n \lg(n))$ en una secuencia de clase 'sorted'

Para las secuencias de tipo 'sorted' [Fig. 2] se puede ver que todos los algoritmos de ordenamiento de la familia -excepto por el *Quicksort* convencional y la variante de 3 particiones- se desempeñan similar. Tal como apunta la teoría, *Quicksort* tiene un tiempo de ejecución en el peor caso de $O(n^2)$, lo cual se ve reflejado en la gráfica. El peor caso ocurre cuando la secuencia a ordenar está desbalanceada; como esta clase de secuencia está ordenada, el algoritmo de partición escogerá siempre el mayor número como pivote y dividirá el problemas en dos instancias, una de las cuales será trivial (de tamaño 1) y la otra habrá disminuido el tamaño del problema por 1, lo cual aumenta la cantidad de llamadas recursivas y hace que el rendimiento del algoritmo decaiga.

Aunque *Quicksort with 3-way Partitioning* logra solucionar parte de este problema esquivando las repeticiones en el arreglo, su desempeño en secuencias ordenadas también llega a ser notablemente peor que el de los otros algoritmos.

Luego, *Randomized Quicksort* al escoger aleatoriamente un pivote se hace totalmente independiente de cuan balanceada esté la instancia, obteniendo un rendimiento parecido al que se espera para *Quicksort* en secuencias ordenadas.

Los algoritmos restantes, al no depender de las características del arreglo o de su forma, no observan cambios en sus tiempos de ejecución al ser probados en secuencias previamente ordenadas. Por su parte, aunque el algoritmo de *Introsort* usa *Quicksort* para ordenar, el escoger como pivote la mediana entre 3 elementos del arreglo y limitar la cantidad de llamadas recursivas aseguran que las particiones sean aproximadamente balanceadas y los tiempos de ejecución no sean asintóticamente más lentos que $O(n \lg(n))$, ya que termina de ordenar usando *Heap Sort* cuando la profundidad de la recursión sobrepasa $\lfloor 2 * \lg N \rfloor$ llamadas.

2.3. Secuencia 'inv'

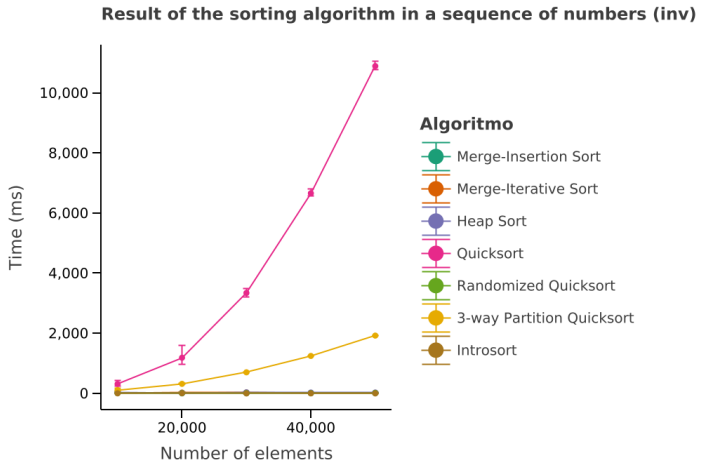

Figura 3: Tiempos de ejecución de los algoritmos $O(n \lg(n))$ en una secuencia de clase 'inv'

Para la clase de secuencias ordenadas descendentemente [Fig. 3] se observa un comportamiento similar al visto en la Fig. 2, lo cual ocurre porque este tipo de secuencia, a pesar de estar en orden inverso, también es desbalanceada, por lo que análogamente a los resultados de la Fig. 2 *Quicksort* y *Quicksort with 3-way Partitioning* alcanzan también su peor caso y obtienen un comportamiento

asintótico cuadrático.

Aunque con un comportamiento asintótico similar para el peor caso, se puede ver que los tiempos en el eje de las ordenadas son entre 2 y 3 veces menores para esta clase que para la anterior, alcanzando este algoritmo un tiempo máximo para 50,000 elementos de aproximadamente 10s, comparado con los cerca de 30s que se tarda en ordenar secuencias 'sorted'. Esto se corresponde con el funcionamiento de los algoritmos, ya que mientras que para las secuencias 'inv' se toma ahora como pivote el número menor y se hace un solo *swap* al final del ciclo, para las secuencias 'sorted' el algoritmo toma de pivote al número mayor y hace $N-1$ *swaps* de los elementos consigo mismos y $N-1$ asignaciones antes de dividir el problema en una instancia trivial y otra apenas menor. Al tener el *swap* un costo constante ($O(1)$), el tiempo de ejecución entre el ordenamiento de secuencias 'inv' y 'sorted' para *Quicksort* no se diferencia más que por un factor constante (entre 2 y 3 en este caso).

Para los demás algoritmos, todos tienen un tiempo de ejecución similiar al de los resultado para secuencias 'sorted'.

2.4. Secuencia 'cerouno'

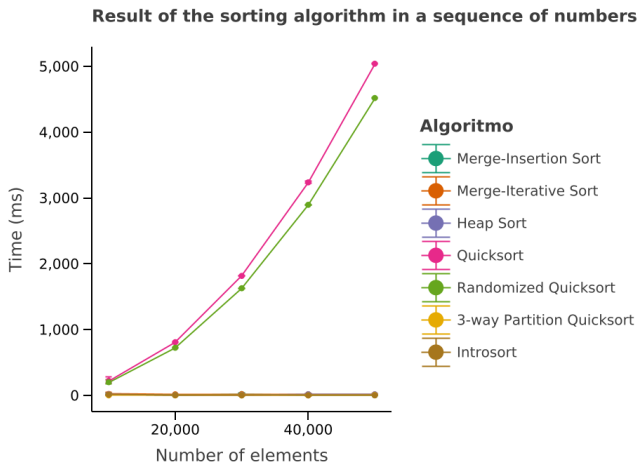

Figura 4: Tiempos de ejecución de los algoritmos $O(n\lg(n))$ en una secuencia de clase 'cerouno'

Al ordenar secuencias de clase 'cerouno', *Quicksort* y *Randomized Quicksort* tienen tiempos de ejecución de $O(n^2)$, como se puede ver la Figura 4, ya que según la teoría su peor caso se da tambuén cuando la instancia del problema tiene demasiados elementos repetidos. Por otra parte, se observa que *Quicksort with 3-way partitioning* cumple su propósito y no desmejora su desempeño en secuencias con muchas repeticiones,ya que crea una partición especial para elementos iguales al pivote, cualquiera que este sea (0 o 1), haciendo que en estos casos mantenga un comportamiento asintóticos de $O(n\lg(n))$.

A pesar de que el pivote se escoge aleatoriamente, *Randomized Quicksort* no soluciona el problema de los arreglos con pocas entradas distintas, aunque igualmente obtiene un desempeño sutilmente mejor -pero no asintóticamente mejor-. Esto ocurre porque estos dos algoritmos separan el arreglo en dos partes (una de elementos menores o iguales al pivote y otra de elementos mayores), y cuando se escoge a 0 como pivote esta división genera dos particiones ordenadas (una de ceros y la otra de unos) que se ordenarán ahora mediante llamadas recursivas, cada una de las cuales produce el peor caso del algoritmo. Además, como el tamaño de las instancias ordenadas será aproximadamente de $\frac{1}{2}$,

se puede notar que el desempeño es alrededor de dos veces más rápido que el observado que en la Fig. 3.

Por otro lado, *Introsort* tampoco sufre cambios en sus tiempos de ejecución ya que el algoritmo de partición de Hoare genera un lado de la partición ordenado y la limitación de las llamadas recursivas hace una llamada a *Heap Sort* una vez se supere el límite de profundidad establecido, lo que hace que el algoritmo mantenga su tiempo de ejecución de $O(n \lg(n))$. Los demás algoritmos no presentan cambios drásticos en sus tiempos de ejecución.

2.5. Secuencia 'mitad'

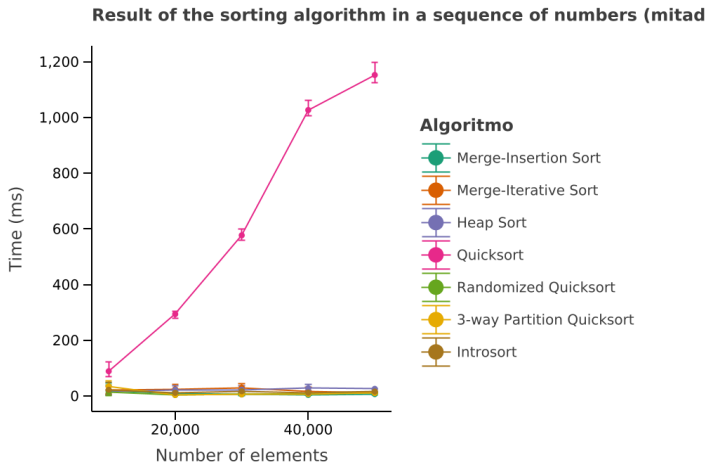

Figura 5: Tiempos de ejecución de los algoritmos $O(n \lg(n))$ en una secuencia de clase 'mitad'

Para las secuencias de clase 'mitad', se observa en la Fig. 5 que el único algoritmo que ve afectado sus tiempos de ejecución es *Quicksort*, de acuerdo a lo visto en los resultados anteriores, en los que obtiene un tiempo de ejecución cuadrático en secuencias "casi ordenadas". En este caso, se puede notar además que los tiempos de ejecución son 4 veces menores a los de los resultados de la Fig. 3, debido a que esta vez con cada llamada se genera una llamada recursiva trivial para ordenar 1 elemento y otra para ordenar un subarreglos de $N - 2$ elementos en el cual el pivote será nuevamente el menor elemento del mismo; esto hace que el algoritmo deje de hacer un número constante de operaciones y obtenga un comportamiento asintótico igualmente cuadrático pero no difente al desempeño del mismo para secuencias 'inv' más que por un factor constante. Los demás algoritmos no presentan tiempos de ejecución especialmente remarcables, siendo de $O(n \lg(n))$como indica la teoría.

2.6. Secuencia ‘repetido’

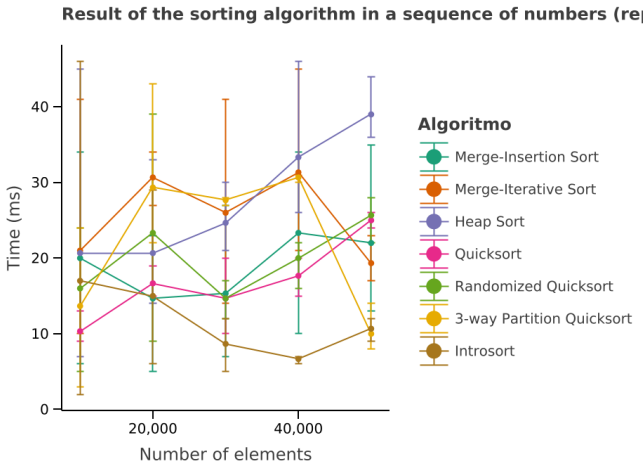

Figura 6: Tiempos de ejecución de los algoritmos $O(n \lg(n))$ en una secuencia de clase ‘repetido’

Los resultados para las secuencias de clase ‘repetido’ [Fig. 6] se muestran óptimos para todos los algoritmos, al tratarse de elementos generados aleatoriamente. A pesar de contar con repetición de elementos, no llegan a ser tantos como para afectar el comportamiento de los algoritmos de *Quicksort* y *Randomized Quicksort*, que como vimos anteriormente, se ven afectados en arreglos con una cantidad ínfima de elementos distintos. A pesar de ello, podemos ver como disminuye por poco el desempeño de estos últimos dos algoritmos con respecto a los resultados en la Fig. 1, estando estos acotados entre 10ms y 25ms aproximadamente, y aquellos entre 5ms y 15ms.

Se observa que los algoritmos de *Merge Sort* y *Merge Sort Iterativo* son por lo general más lento (no asintóticamente) que los algoritmos basados en *Quicksort*, a pesar de ser (por lo general) más rápidos que *Heap Sort*. El algortimo que alcanza los menores tiempos de ejecución es *Introsort*, incluso en secuencias que aparentemente representaron un problema para *Quicksort with 3-way Partitioning* (por ejemplo, las secuencias de 20,000, 30,000 y 40,000 elementos en 6).

2.7. Conclusiones

El análisis presentado permite concluir que:

- A pesar de tener un tiempo de ejecución esperado en el orden de $O(n \lg(n))$para secuencias aleatorias, es preferible usar alguna variante optimizada de *Quicksort* cuando la secuencia a ordenar puede estar ordenada o tener muchas repeticiones.
- *Heap Sort* puede ser más útil como procecedimiento auxiliar para ser usado en algoritmos híbridos que por sí solo.
- El desempeño de *Quicksort* depende fuertemente de la forma que puedan tener los datos a ordenar, por lo que es importante tenerlo en cuenta antede de elegir algún algoritmo de ordenamiento.
- Los algoritmos híbridos (como *Introsort*), y optimizados (como *Quicksort with 3-way Partitioning*) son los más recomendables al necesitar un algoritmo de ordenamiento con un desempeño razonable en *cualquier* tipo de secuencia, independientemente de su forma.