Universidad Simón Bolívar

Reporte de Laboratorio de Semana 5

Análisis de algoritmos de ordenamiento

Autor:
Christopher Gómez

Profesor:
Guillermo Palma

Laboratorio de Algoritmos y Estructuras de Datos II (CI2692)

13 de junio de 2021

1. Metodología

El siguiente estudio experimental consiste en correr un conjunto de algoritmos de ordenamiento sobre cinco secuencias de diferentes tamaños (200,000, 400,000, 600,000, 800,000 y 1,000,000 números enteros), midiendo sus tiempos de ejecución. Cada secuencia consiste en N números enteros generados aleatoriamente en el rango $[0..N]$ y 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 al inicio ni de verificar que fue ordenada correctamente.

Los algoritmos a ejecutar, implementados en el lenguaje de programación Kotlin, serán los siguientes: *Quicksort*, *Instrospective Sort (Introsort)*, *Counting Sort* y *Radix Sort*.

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. Detalles de implementación

La implementación del algoritmo de *Counting Sort* está basada en el pseudocódigo mostrado en la página 195 de *Introduction to algorithms*[1]. Dicho pseudocódigo tiene un parámetro k que corresponde, según se explica, al máximo valor que pueden tomar los elementos de A –para crear un arreglo C de tamaño $k+1$ que guardará en su i-ésima entrada la frecuencia de apariciones de i en A–, y un arreglo B del mismo tamaño de A que obtendrá la salida del programa (el arreglo A ordenado); en la implementación realizada se crea una función adicional que halla la k apropiada y crea el arreglo de que recibirá la llamada del algoritmo, y mismamente, llama a la función.

Para k, esta es asignada antes de comenzar el algoritmo como el entero más grande del arreglo de entrada, el cual es hallado inspeccionando el arreglo completo; dado que solamente se itera sobre el arreglo una vez y se hace (a lo sumo) una comparación y una asignación por cada iteración, el tiempo de encontrar el mayor valor de A es lineal ($O(n)$), por lo cual no afectará al desempeño del algoritmo más que por un factor constante; además, no se toma k como el tamaño del arreglo porque, aunque se sabe que en las pruebas a realizar las secuencias solo contendrán enteros entre 0 y n, se diseñó el algoritmo para ser utilizado en cualquier secuencia de números no negativos, evitando por otra parte un gasto adicional de memoria cuando el mayor elemento del arreglo es mucho menor que su tamaño, o un error de acceso cuando este es mayor que el mismo. Por otro lado, para B y C se inicializan arreglos de tamaño $A.size$ y k+1, respectivamente, usando la estructura **IntArray** de Kotlin ya que proporcionan una fácil inicialización y mayor eficiencia para este propósito. Finalmente, se añadió al final de la función un ciclo que copia el arreglo ordenado B en el arreglo original A, tomando nuevamente tiempo lineal, y manteniendo el tiempo de ejecución de *Counting Sort* en el orden de $O(n+k)$.

En cuanto a la implementación de *Radix Sort*, el algoritmo estable utilizado fue una modificación de *Counting Sort* que ordena en tiempo lineal los elementos del arreglo de entrada según el d-ésimo dígito. Se escogió *Counting Sort* debido a que es estable, ordena en tiempo lineal, y es eficiente en el uso de la memoria al solo tener que ordenar usando un dígito de los números, por lo que se crea únicamente un arreglo C de tamaño 10 ([0..9]) para almacenar las frecuencias. No se usó *Bubble Sort* o *Insertion Sort* porque, a pesar de ser estables, sus tiempos de ejecución son $O(n^2)$, lo que haría al algoritmo en general ineficiente, luego, tampoco se usó *Merge Sort* porque, incluso siendo considerablemente eficiente (y además estable), al ser un algoritmo basado en comparaciones se ejecuta en tiempo logarítmico-lineal, por lo que se prefirió un algoritmo con comportamiento

[1]Cormen, T., Leiserson, C., Rivest, R., and Stein, C. Introduction to algorithms, 3rd ed. MIT press, 2009.

asintótico lineal. Luego, este algoritmo también recibe un parámetro d (pseudocódigo en la página 198 de *Introduction to Algorithms*[2]) con la cantidad de cifras del mayor valor del arreglo, por lo que –de forma análoga a *Counting Sort*– se crea en la implementación una función adicional en la que se halla el valor más grande de A mediante un escaneo lineal ($O(n)$), para calcular su número de dígitos mediante división entera repetida ($O(\log(\text{máx}(A)))$) y obtener el valor d que recibirá la llamada del algoritmo principal. Así, esta implementación tiene una complejidad de $O(d(n + 10))$.

3. Resultados experimentales

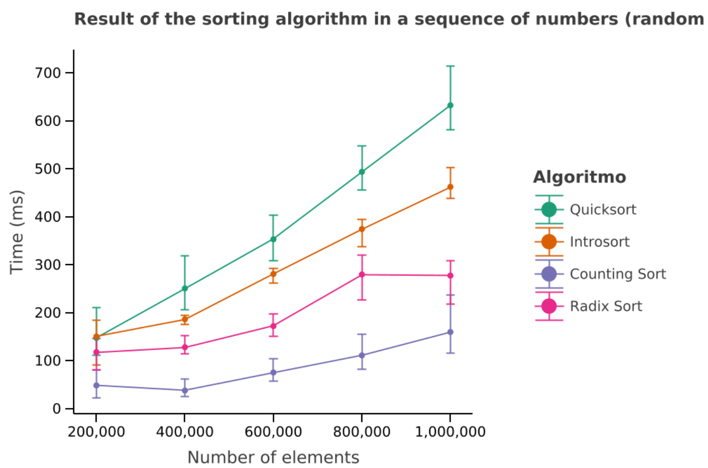

Figura 1: Tiempos de ejecución de los algoritmos 'ln' en secuencias aleatorias

Los resultados obtenidos se muestran en la Figura 1. Como se puede observar, se obtienen de estos algoritmos el comportamiento esperado según la teoría, la cual apunta una diferencia notoria en la práctica entre los órdenes de crecimiento de los algoritmos involucrados en la gráfica. El más lento de la familia es *Quicksort*, seguido de *Introsort*, y esto es debido a que *Introsort*, limitando la cantidad de llamadas recursivas y usando una forma distinta de partición logra ser una versión optimizada de *Quicksort*, que aunque mantiene sus tiempos de ejecución (asintóticamente hablando, ambos son en promedio $\Theta(n \lg(n))$), en la práctica logra ser más rápido debido a las constantes involucradas y se desempeña de forma razonable sea cual sea la forma de la secuencia a ordenar.

Asimismo se observan, un orden de crecimiento por debajo, los algoritmos de *Counting Sort* y *Radix Sort*, en consonacia con la teoría, dado que tienen un tiempo de ejecución en el peor de los casos de $O(n + k)$ y $O(d(n + k))$ respectivamente, donde k es el mayor número presente en la secuencia (acotado por n en este caso para *Counting Sort* en secuencias de clase 'random', e igual a 10 para *Radix Sort*) y d es la cantidad de dígitos del número más grande de la secuencia. El algoritmo más rápido de la familia es *Counting Sort* y esto se debe a que, a pesar de tener que crear un arreglo de tamaño aproximadamente igual a n para ejecutar el algoritmo, lo hace solo una vez, y en cambio *Radix Sort* ordena la secuencia de 5 a 6 veces, lo que hace que los factores constantes involucrados sean más elevados y obtenga un tiempo en este caso ligeramente mayor. Sin embargo, *Radix Sort* sigue manteniendo un comportamiento asintóntico lineal.

Con los datos del Cuadro 1 (que corresponden a los tiempos de la Figura 1), se puede verificar que, efectivamente, el crecimiento de los algoritmos de *Counting Sort* y *Radix Sort* es lineal. Para *Counting Sort*, vemos que el mejor tiempo del algoritmo aumenta aproximadamente entre 30 y 40 ms cuando el tamaño de la secuencia aumenta por 200.000 elementos, tanto al comienzo como al final, y con *Radix Sort*, pese a tener una gráfica más dispareja y un pico a los 800.000 elementos, también crece de forma aproximadamente lineal. En cambio, para *Quicksort* e *Introsort* se puede ver que para las secuencias pequeñas logran tiempos no tan alejados de

[2] Ib.

		Tamaño de la secuencia				
Algoritmo	**Tiempo (ms)**	200.000	400.000	600.000	800.000	1.000.000
Quicksort	Peor	210	319	403	548	714
	Mejor	111	206	308	456	582
	Promedio	147,33	250,67	353	493,67	632,33
Introsort	Peor	184	195	293	395	502
	Mejor	91	176	262	338	439
	Promedio	150,33	186,33	280,67	374,33	462,33
Counting Sort	Peor	80	61	104	155	237
	Mejor	23	25	58	82	115
	Promedio	48,33	37,67	75,33	111	157,67
Radix Sort	Peor	147	152	197	320	308
	Mejor	80	114	151	226	218
	Promedio	117,67	128	172	279,67	277,33

Cuadro 1: Tiempos de los algoritmos de ordenamiento de la familia 'ln'

los algoritmos lineales (alrededor de 100 ms más), pero debido a su orden de crecimiento más elevado terminan ordenando las secuencias más grandes con una diferencia ya notoria de 200 a 500 ms.

3.1. Conclusiones

El proceso de implementación, los resultados obtenidos y el análisis sobre ellos permite concluir que:

- Los algoritmos de *Counting Sort* y *Radix Sort* (y en general, los algoritmos de ordenamiento en tiempo lineal) son notablemente más eficientes para ordenar secuencias de enteros no negativos, al no basarse en comparaciones. Sin embargo, son limitados en cuanto a los tipos de arreglos que pueden ordenar.
- Aunque no es notorio en las gráficas, ni en las secuencias de las pruebas, cuando se busca eficiencia en la administración de la memoria puede ser mejor utilizar algoritmos como *Quicksort* o alguna variación.
- Cuando el mayor elemento presente en una secuencia es notablemente más grande que el tamaño de la misma el desempeño de *Counting Sort* puede decaer (por ejemplo, si se quiere ordenar una secuencia de n enteros no negativos entre 0 y $(1+\epsilon)^n$, para $\epsilon > 0$).
- La estabilidad es muchas veces una característica deseable y útil en los algoritmos de ordenamiento.
- Un mayor gasto de memoria no implica un peor desempeño en tiempo, pero puede ser un punto a considerar a la hora de escoger qué algoritmo de ordenamiento usar.