Universidad Simón
Bolívar

Reporte de Laboratorio de Semana 3

# Análisis de algoritmos de ordenamiento

**Autor:**
Christopher Gómez

**Profesor:**
Guillermo Palma

**Laboratorio de Algoritmos y Estructuras de Datos II (CI2692)**

30 de mayo de 2021

# 1. Metodología

El siguiente estudio experimental consiste en correr tres conjuntos de algoritmos de ordenamiento sobre cinco secuencias de diferentes tamaños (10,000, 20,000, 30,000, 40,000 y 50,000 números enteros), midiendo sus tiempos de ejecución. Cada secuencia consiste en $N$ números enteros generados aleatoriamente en el intervalo $[0..N]$ 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 respectivo 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.

Los algoritmos de ordenamiento fueron implementados en el lenguaje de programación Kotlin, y divididos para el estudio en las siguientes tres categorías:

- **Algoritmos $O(n^2)$:** *Bubble Sort*, *Insertion Sort*, *Selection Sort* y *Shell Sort*.
- **Algoritmos $O(n \lg(n))$:** *Merge-Insertion Sort* (recursivo), *Merge-Iterative Sort* (variante iterativa) y *Heap Sort*.
- **Todos:** Los algoritmos que conforman las dos categorías anteriores.

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. Algoritmos $O(n^2)$

Los primeros resultados obtenidos fueron los correspondientes a la ejecución de los algoritmos $O(n^2)$, los cuales se muestran en la Figura 1.

Como se puede ver, la gráfica obtenida refleja cómo crece el tiempo de ejecución a medida que aumenta el número de elementos en la secuencia de entrada. De acuerdo a lo que dice la teoría, estos algoritmos (a exception de *Shell Sort*), al ser $O(n^2)$, se tiene que un aumento lineal en el tamaño de la entrada se verá reflejado en aumento cuadrático en el tiempo de ejecución, lo cual se corresponde notoriamente con lo observado en [1].

Es curioso observar que a pesar de que todos estos algoritmos se comportan asintóticamente equivalentes, cuando el tamaño de la entrada es relativamente pequeño sí es perceptible una diferencia entre los tiempos de ejecución.

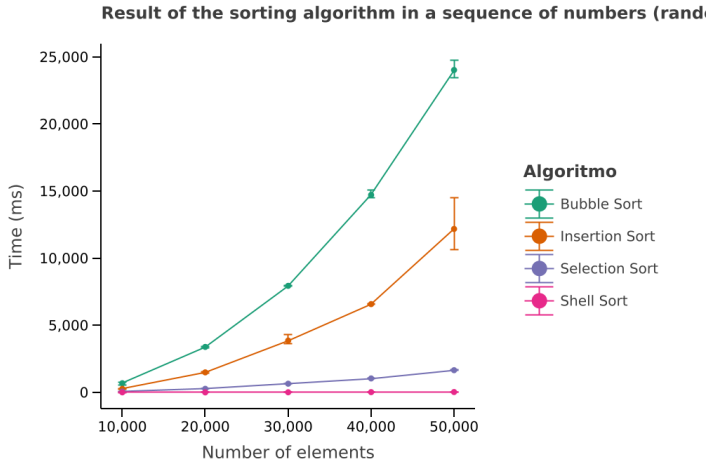


Figura 1: Tiempos de ejecución de los algoritmos $O(n^2)$

Para esta familia de algoritmos, se observa que el más lento es *Bubble Sort*, a pesar de ser el más fácil de implementar, seguido de *Insertion Sort*, esto se debe a que la implementación de *Bubble Sort* se basa en dos ciclos **for** anidados, lo cual hace que se pase repetidamente por el arreglo haciendo comparaciones, independientemente de si el arreglo ya estaba ordenado, por lo que incluso corriendo en secuencias ya ordenadas tardaría más que los demás algoritmos. Por otra parte, el algoritmo de *Insertion Sort* soluciona este problema al tener un ciclo **while** anidado dentro de un ciclo **for**, lo cual hace que ciertas veces no se termine de recorrer el arreglo al encontrar el lugar donde se quiere insertar cada número; de esta forma, cuando el arreglo está ordenado *Insertion Sort* llega a ser aún más rápido que algoritmos de la familia $O(n \log(n))$, sin embargo, como se ve en [1], en promedio su comportamiento asintótico es cuadrático.

Los mejores algoritmos de esta familia terminan siendo *Selection Sort* y *Shell Sort*. Aunque *Selection Sort*, al igual que *Bubble Sort*, también consta de dos **for** anidados, los resultados muestran que su desempeño es casi diez veces mejor que los dos anteriores porque funciona mediante comparaciones y hace sólo un intercambio al final del ciclo externo, lo cual lo hace en general menos costoso. Sin embargo, aunque empíricamente es mucho más rápido que los algoritmos anteriores, su tiempo no se diferencia de el de los otros más que por un factor constante e independiente del tamaño de la entrada, lo que lo hace pertenecer todavía a la categoría de $O(n^2)$, ya que para una entrada lo suficientemente grande esta constante es despreciable.

Por último, notamos que el algoritmo de *Shell Sort*, aunque su tiempo de ejecución en el peor caso sea $O(n^2)$, supera en promedio por mucho a estos algoritmos, ya que su tiempo de ejecución según la teoría es $O(n^{\frac{3}{2}})$, lo cual explica la enorme ventaja que tiene con respecto a los demás algoritmos, ya que está acotado superiormente por un polinomio de grado menor a 2. Mientras más aumenta el tamaño de la secuencia, más exagerada es la diferencia entre *Shell Sort* y los demás algoritmos de este conjunto, a tal punto que ya para 50.000 elementos, ya es alrededor de 50 veces más rápido que *Selection Sort* y casi 700 veces más rápido que *Bubble Sort*.

## 2.2. Algoritmos $O(n \lg(n))$

Los próximos resultados que se obtuvieron fueron los correspondientes a los tiempos de los algoritmos $O(n \lg(n))$, mostrados en la Figura 2.

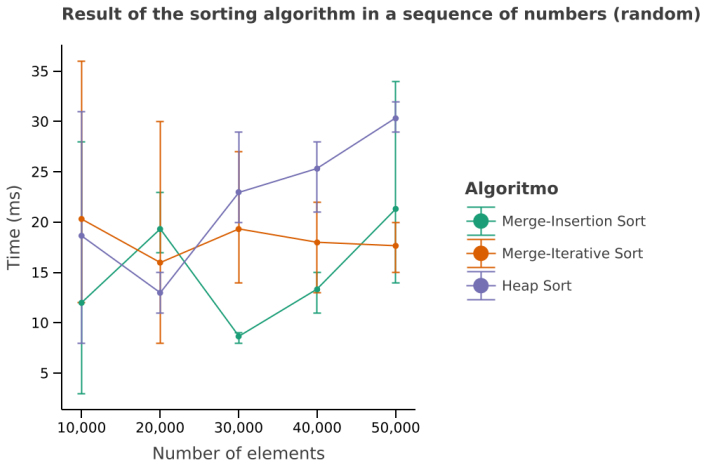


Figura 2: Tiempos de ejecución de los algoritmos $O(n \lg(n))$

A diferencia de en [1], donde la escala avanzaba de 5.000 en 5.000, vemos que el eje de ordenadas en [2] va de 0 a 35, lo cual deja constancia del extraordinario salto de rendimiento que supone un algoritmo que crece asintóticamente de forma logarítmico-lineal, con respecto a los algoritmos cuadráticos y polinómicos.

Dado lo mínimo de la escala, se puede ver que la desviación estándar es –relativamente– grande, llegando a haber márgenes de hasta 30 ms entre el mejor y el peor tiempo de ejecución del algoritmo, tratándose de que la secuencia que se está ordenando es exactamente la misma. Sin embargo, teóricamente, la desviación relativa debería ir disminuyendo conforme el tamaño de la entrada crece (a, por ejemplo, millones de números enteros), lo cual se corresponde con lo que podemos observar en los casos de prueba del estudio, ya que la desviación estándar es grande para los tres algoritmos sobre la misma secuencia de 10.000 enteros, y para la secuencia de 50.000 disminuyó razonablemente.

Para secuencias no muy grandes, los tres algoritmos de ordenamiento rinden de forma similar, aunque la diferencia entre los tres llega a ser notoria con instancias del problema más grandes, a pesar de tener el mismo orden de crecimiento asintóntico. Para 4 de las 5 secuencias, el algoritmo más rápido fue el *Merge Sort recursivo* en su mejor tiempo, seguido de su variante iterativa y luego *Heap Sort*. Esta diferencia puede deberse a que los algoritmos de *Merge* dividen el problema y el paso de combinar es posiblemente menos costoso que para el *Heap Sort*, que aunque no se basa en *Dividir, conquistar y combinar*, se debe restaurar la estructura de Heap para mantener sus propiedades por cada iteración, lo cual puede ser más costoso que el paso de combinar, aunque no por más de un factor constante. Además, lo que hace para este estudio el algoritmo de *Merge Sort recursivo* sea el más eficiente es que la variante implementada termina de ordenar las secuencias cuando estas tienen a lo sumo 10 elementos usando *Insertion Sort*, el cual a pesar de ser mucho más lento para secuencias grandes, es más conveniente para ordenar secuencias de números muy pequeñas, y se desempeña mejor que hacer más llamadas recursivas a *Merge Sort* [1]

[1] Cormen, T., Leirserson, C., Rivest, R., and Stein, C. Introduction to Algorithms, 3ra ed. McGraw Hill, 2009.

## 2.3. Todos los algoritmos

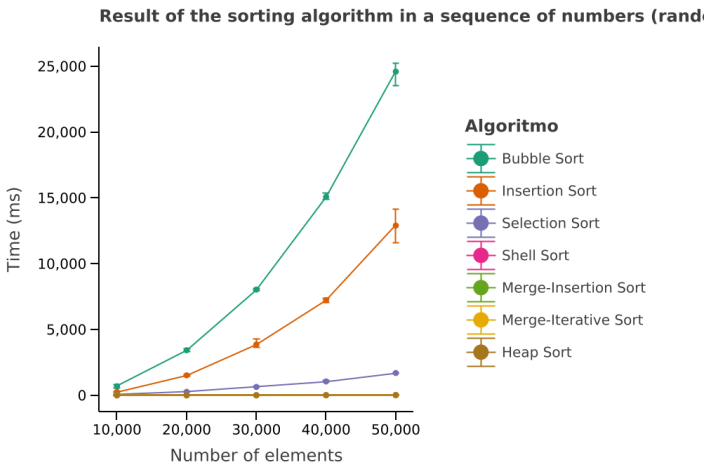


Figura 3: Tiempos de ejecución de todos los algoritmos

| | | Tamaño de la secuencia | | | | |
|---|---|---|---|---|---|---|
| **Algoritmo** | **Tiempo (ms)** | 10.000 | 20.000 | 30.000 | 40.000 | 50.000 |
| Bubble Sort | Peor | 826 | 3528 | 8129 | 15385 | 25240 |
| | Mejor | 573 | 3331 | 7981 | 14930 | 23518 |
| | Promedio | 702 | 3440,35 | 8048 | 15089 | 24609,68 |
| Insertion Sort | Peor | 291 | 1600 | 4294 | 7411 | 14134 |
| | Mejor | 248 | 1496 | 3661 | 7075 | 11585 |
| | Promedio | 271 | 1533 | 3877 | 7247 | 12911 |
| Selection Sort | Peor | 106 | 279 | 656 | 1147 | 1738 |
| | Mejor | 52 | 274 | 647 | 1019 | 1660 |
| | Promedio | 73,67 | 276,67 | 650,34 | 1067.32 | 1707 |
| Shell Sort | Peor | 27 | 15 | 27 | 30 | 48 |
| | Mejor | 10 | 11 | 17 | 27 | 34 |
| | Promedio | 16,32 | 13 | 21,68 | 29 | 39,36 |
| Merge Sort (rec.) | Peor | 29 | 11 | 12 | 17 | 18 |
| | Mejor | 5 | 8 | 8 | 11 | 15 |
| | Promedio | 16 | 9,34 | 10 | 13,34 | 17 |
| Merge Sort (it.) | Peor | 19 | 13 | 12 | 47 | 21 |
| | Mejor | 9 | 8 | 10 | 11 | 16 |
| | Promedio | 15,34 | 10 | 10.67 | 23,68 | 17,68 |
| Heap Sort | Peor | 14 | 38 | 26 | 28 | 33 |
| | Mejor | 8 | 11 | 17 | 20 | 30 |
| | Promedio | 10,6 | 24,32 | 21,32 | 24,32 | 31,68 |

Cuadro 1: Resumen de tiempos todos los algoritmos

Por último, se corrieron una vez más todos los algoritmos de ordenamiento y esta vez se tomó nota también de los resultados exactos sobre el mejor y peor tiempo de ejecución, junto con su tiempo

promedio, los cuales se pueden ver en la Figura 3 y en el Cuadro 1.

De esta forma, el gráfico de los tiempos de ejecución de todos los algoritmos en un mismo plano da una visión general de la gran diferencia de rendimiento entre ambas familias. Podemos diferenciar claramente cada algoritmo $O(n^2)$, sin embargo, los algoritmos $O(n \lg(n))$ son comparativamente tan rápidos que terminan solapándose en [3], e incluso con *Shell Sort*, que aunque asintóticamente es más lento que estos últimos, sigue siendo más rápido que $O(n^2)$ en la escala de órdenes de crecimiento, al ser $O(n^{\frac{3}{2}})$.

## 2.4. Conclusiones

Como observaciones finales, los datos de la tabla nos permiten notar que incluso el peor tiempo del algoritmo de ordenamiento más rápido (*Merge Sort recursivo*) en la secuencia más grande es al menos 30 veces más rápido que el mejor tiempo alcanzado por el algoritmo más lento (*Bubble Sort*) en la secuencia más pequeña. Además, podemos ver cómo crecen numéricamente los tiempos de acuerdo a lo que teóricamente debería ser; así, por ejemplo, vemos que el tiempo promedio de *Insertion Sort* crece 5 veces cuando se dobla el tamaño de la entrada, pero 47 veces cuando se alcanzan los 50.000 elementos, mientras que para *Heap Sort*, al tener un orden de crecimiento menor, vemos que el tiempo de ejecución se dobla al doblar el tamaño de la entrada, pero al verse esta multiplicada por 5, el tiempo apenas triplica el primero.

Con todo lo analizado, podemos concluir que:

- Los factores constantes de costo en los algoritmos, aunque teóricamente –y al hablar de crecimiento asintótico–, suelen ser insigmificantes e ignorados, en la práctica pueden llegar a tener peso y hacer una diferencia notable en el tiempo de ejecución de los algoritmos.
- Los algoritmos de orden de crecimiento menor serán, por lo general, notablementente más rápidos que los algoritmos de orden de crecicimiento mayor para entradas lo suficientemente grandes.
- Para instancias pequeñas, los algoritmos de menor orden de crecimiento no son necesariamente más eficientes que los de mejor rendimiento, lo cual ocurre por los terminos constantes de costo que son ignorados con la notación asintótica.
- Cada ejecución del mismo algoritmo sobre exactamente la misma instancia del problema puede variar en tiempo a pesar de estar realizando las mismas instrucciones, es por esto que se hace necesario una notación que busca expresar la eficiencia de los mismos independientemente del computador, la administración de memoria, etc.