Análisis asintótico de algoritmos

Sergio Revilla Velasco

Resumen

El objetivo de esta práctica es estudiar la complejidad asintótica de los algoritmos de ordenación más comunes: selección (selection-sort), inserción (insertion-sort), burbuja (bubble-sort), ordenación rápida (quick-sort) y ordenación por mezcla (merge-sort), por medio de un programa escrito en **C++**.

Índice

1. Instrucciones de compilación y dependencias

1. **Dependencias:**
   - **Windows:** MinGW `http://www.mingw.org`
   - **Linux:** GCC Compiler Collection `http://gcc.gnu.org`
   - **Mac Os X:** Xcode Tools con utilidades de línea de comando instaladas `https://developer.apple.com/xcode/`
   - **GnuPlot:** usado para generar las gráficas desde archivos de texto `http://www.gnuplot.info`
2. **Instrucciones de compilación:**
   - Compilar el archivo *main.cpp* contenido en la carpeta */src*
   - Ejecutar desde la consola el comando *gnuplot script.gnuplot* desde el mismo directorio donde se generaron los archivos de datos.
   - El script generará archivos de imagen en formato *png* con las gráficas.

2. Prólogo

Todos los algoritmos analizados en este trabajo parten de unas precondiciones y postcondiciones comunes. Partiendo de un array de números enteros positivos de tamaño $n \geq 0$, construimos un nuevo array con los elementos ordenados. La especificación formal puede expresarse como:

- **Precondición:**
  $P \equiv \{\text{lon}(V) = n \geq 0 \wedge V[n] \geq 0\}$ (Vector no vacío de enteros positivos)
- **Definición:**
  ordena(int &V[], int n) (Devuelve el vector ordenado por referencia)
- **Postcondición:**
  $Q \equiv \{\forall i : 0 \leq i < n-1 : V[i] \leq V[i+1]\}$

O expresado de otra manera:[1]

- **Entrada:** una secuencia de n enteros positivos (a_1, a_2, a_3, ... , a_n)
- **Salida:** una permutación (reordenación) de la secuencia de entrada tal que ($a_1 \leq a_2 \leq a_3$, ... , $\leq a_n$)

Cada método de ordenación se expresa como un algoritmo: un procedimiento de cálculo bien definido que toma algún valor o conjunto de valores como entrada y produce un cierto valor o conjunto de valores como salida.

3. Método de obtención de tiempos y gráficas

1. El programa preguntará por el tamaño del problema (la longitud del array)
2. También podemos elegir el mayor entero a generar (un entero puede tener un valor máximo de 2147483647)
3. El programa preguntará por el salto *(gap)* entre iteraciones. Cuanto menor sea el tamaño de la iteración, más puntos se generarán en el archivo de datos y más ajustada será la gráfica. Para asegurarnos que recorremos hasta el último elemento del array el salto debe ser múltiplo de 5.
4. En primera instancia se genera un array con números aleatorios que permanece inmutable y que sirve como base para las ordenaciones. Las copias del array se van procesando en incrementos de *gap* elementos y los algoritmos se ejecutan sobre un array con los mismos elementos que el array original.
5. El programa genera un archivo con los datos separados por un salto de línea. Cada línea contiene el tamaño del problema y el tiempo empleado para la ordenación expresado en segundos.
6. Con los archivo de datos, puede ejecutarse el script de órdenes con la sintaxis de *gnuplot* para crear los arhivos de imagen con las gráficas de las métricas generadas.

4. Análisis de resultados

- Tamaño del problema: 100000[1]
- Máximo entero generado: 125000
- Salto entre iteraciones: 100

[1]Todos los resultados han sido generados en un ordenador Apple Imac, Mac Os X 10.8.2, Procesador Intel Core i5 a 2,5Ghz (4 núcleos), 4 Gb de RAM a 1333 Mhz DDR3.

4.1. Ordenación por inserción

Su complejidad es de $O(n^2)$. Un buen algoritmo de ordenación para un número pequeño elementos. Funciona de la manera análoga a la que se ordenaría un mazo de cartas:[1]

1. Empezamos con la mano izquierda vacía y las cartas boca abajo sobre la mesa.
2. A continuación, cogemos una carta de la mesa, y la insertamos en la posición correcta en la mano izquierda.
3. Para encontrar la posición correcta en la que insertar la carta, se compara con cada una de las cartas que ya tenemos en la mano izquierda, recorriendo de derecha a izquierda.
4. En todo momento, las cartas sujetas con la mano izquierda se ordenan, permaneciendo las cartas originales sin ordenar encima de la pila en la mesa.

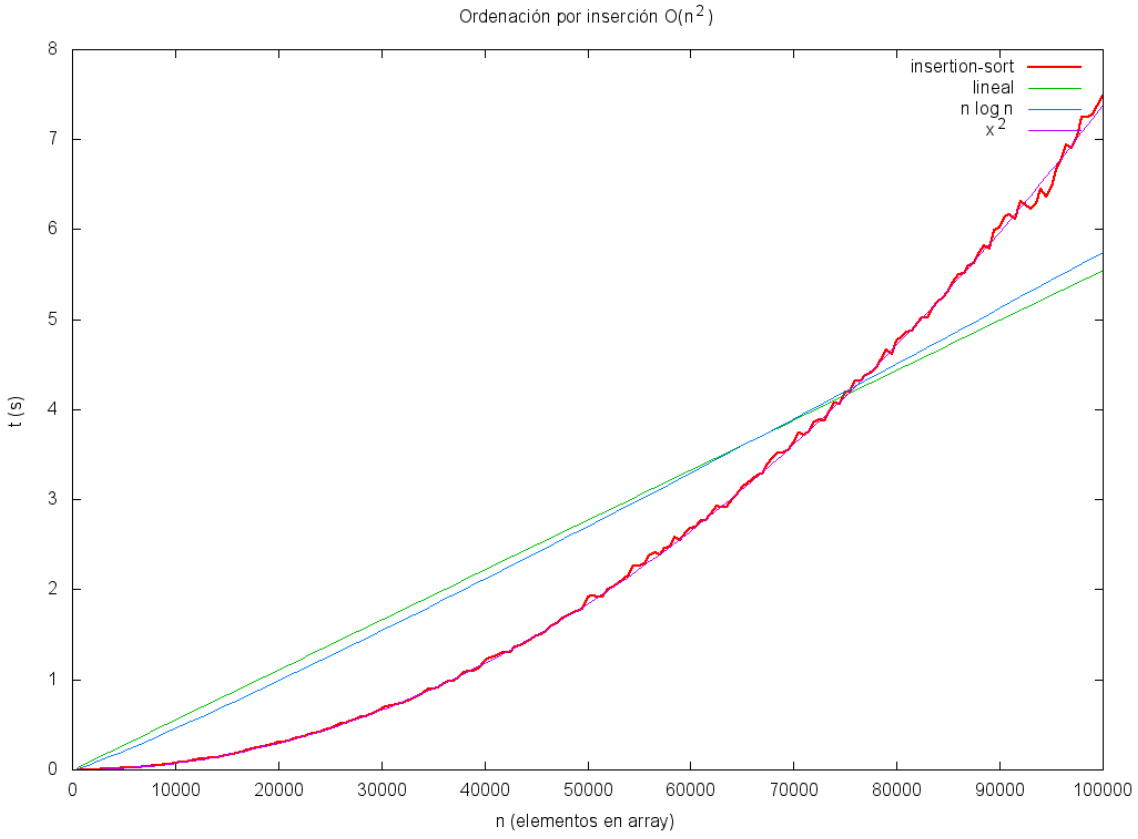

Figura 1: Ordenación por inserción

4.2. Ordenación por selección

Es un algoritmo de ordenación por comparación. Su complejidad es de $O(n^2)$. Resulta ineficiente en tamaños de problema grandes, y suele funcionar peor que la ordenación por inserción. Se caracteriza por su sencillez, y también tiene ventajas de rendimiento sobre algoritmos más complicados en determinadas situaciones, sobre todo cuando la memoria auxiliar es limitada.
El algoritmo divide la lista de entrada en dos partes: la lista secundaria de los elementos que ya están ordenados, dispuestos de izquierda a derecha, y la lista secundaria de los elementos restantes a ser ordenados, ocupando el resto de la lista. Inicialmente, la sublista ordenada está vacía y la sublista sin ordenar es la lista original menos los elementos ordenados. El algoritmo continúa encontrando siempre el elemento inmediatamente superior (según orden de clasificación) y lo cambia con el último elemento del array.

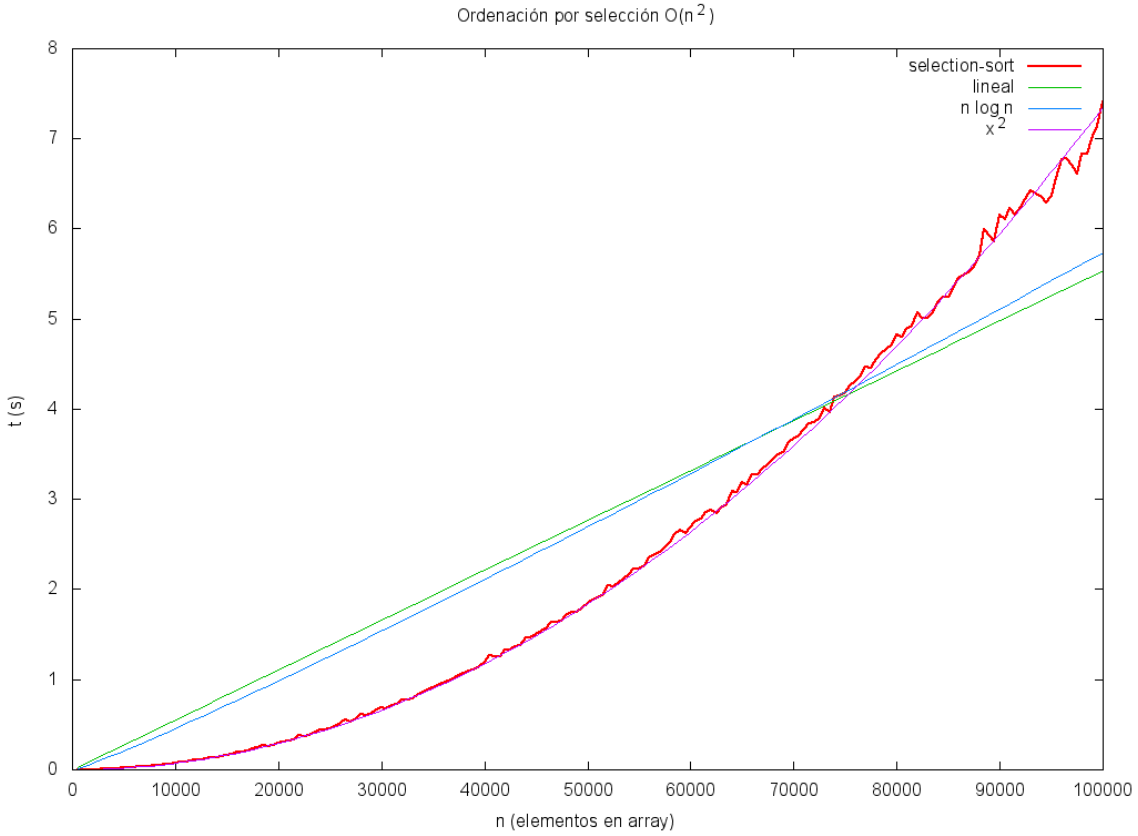

Figura 2: Ordenación por selección

4.3. Ordenación por burbuja

Es un algoritmo por comparación. Recorremos los elementos del array de derecha a izquierda, comparando cada elemento con el siguiente e intercambiándolos en caso de estar desordenados. Para ordenar un vector de n elementos, el algoritmo siempre realiza el mismo número de comparaciones:

$$\frac{n^2 - n}{2}$$

Esto lo convierte en un algoritmo ineficiente para tamaños de problema grandes y, en general, es menos eficiente que otros algoritmos de orden $O(n^2)$ como *insertion-sort*.

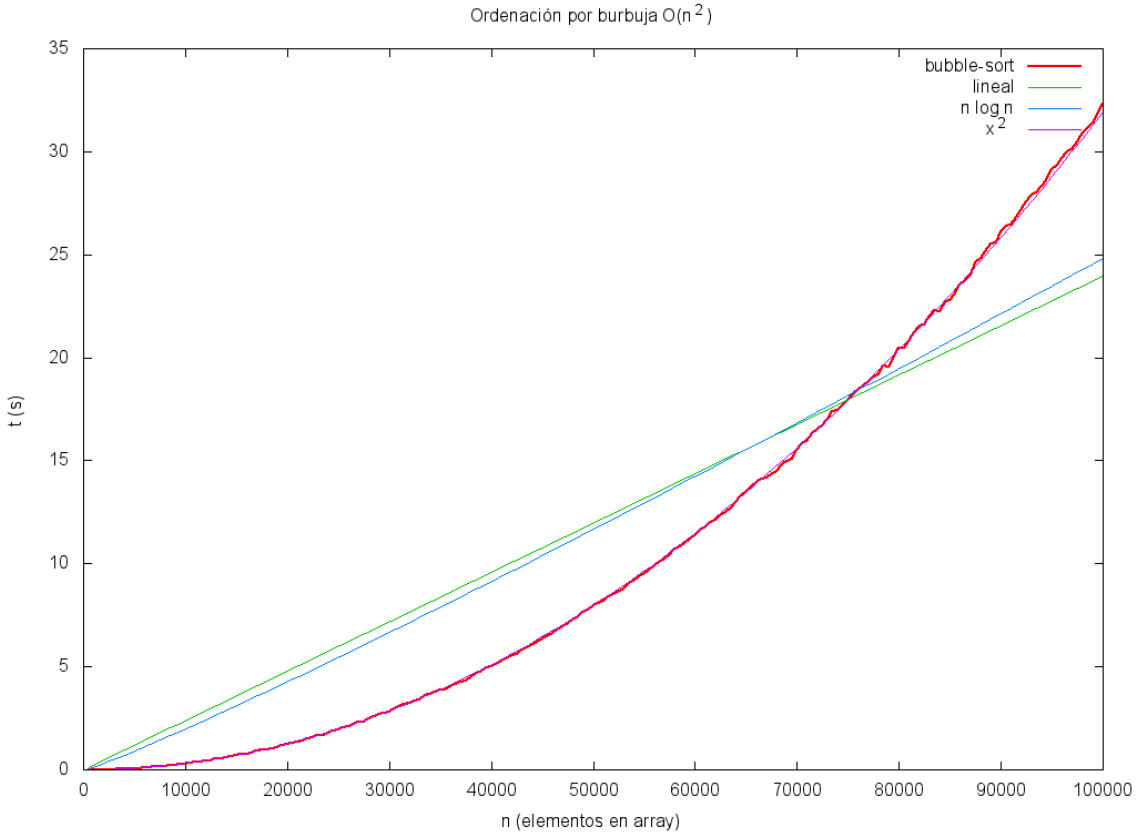

Figura 3: Ordenación por burbuja

4.4. Ordenación rápida

El algoritmo de ordenación rápida o *quick-sort*, aunque tiene una complejidad en el caso peor de orden $O(n^2)$, resulta uno de los algoritmos más eficientes debido a su complejidad media de orden $O(nlog(n))$. Sigue el esquema de *divide y vencerás* y su rendimiento depende en gran medida de cómo se realicen las particiones (qué elementos escojamos para las particiones). Si éstas son balanceadas, el algoritmo se ejecuta asintóticamente tan rápido como *merge-sort*. Si no son balanceadas, puede ejecutarse tan despacio como *insertion-sort*.[1]

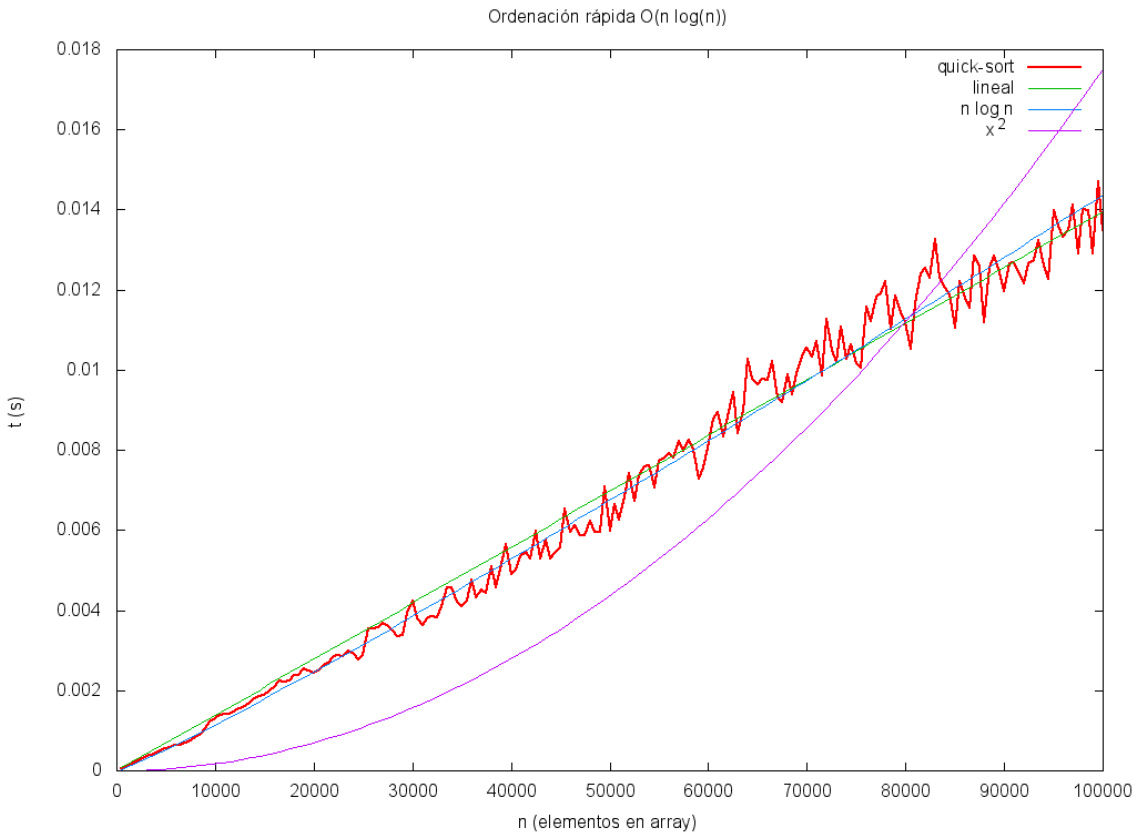

Figura 4: Ordenación rápida

4.5. Ordenación por mezcla

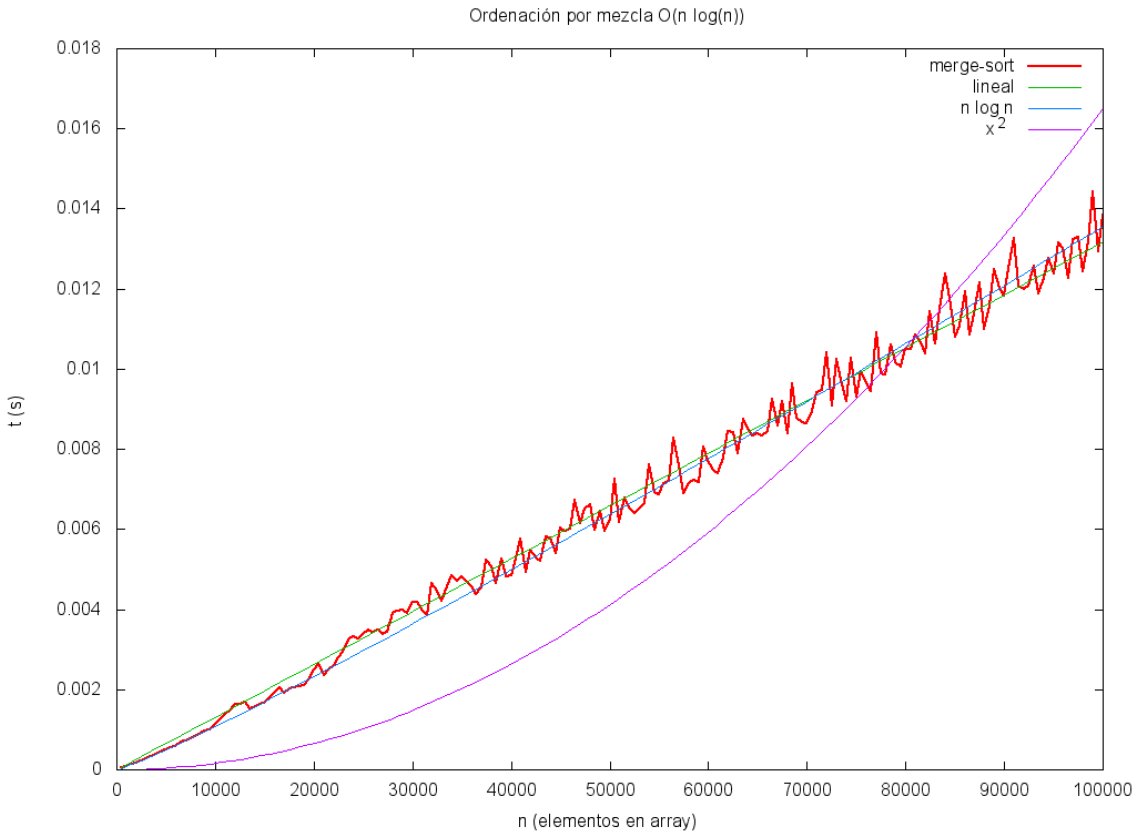

Figura 5: Ordenación por mezcla

5. Comentarios adicionales

Referencias

[1] Cormen, Thomas H. and Leiserson, Charles E. and Rivest, Ronald L. and Stein, Clifford. *Introduction to Algorithms, Second Edition*, MIT Press.