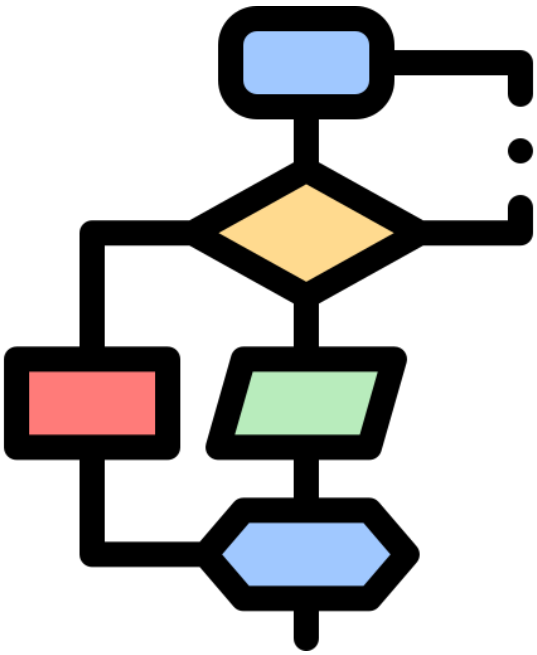

COMPLEJIDAD DE ALGORITMOS

ETSII - URJC

Sebastián Ramiro Entierros García,
Adrián Martin Martin y
Álvaro Martínez Quiroga

Fundamentos del diseño del software

Índice

Contenido

Introducción y ficheros

En esta memoria se va a desarrollar la primera practica de la asignatura de *Fundamentos del Diseño del Software* del grado de *Ingeniería de Computadores* de la Universidad Rey Juan Carlos. La práctica consiste en el estudio teórico y empírico de 3 algoritmos de ordenación, estos son (en inglés): ***Bubble Sort***, ***Selection Sort*** y **Merge Sort**. Al final de las explicaciones también responderemos a una serie de preguntas sobre la práctica y finalizaremos con las conclusiones personales.

El paquete entregado incluye los siguientes ficheros y archivos:

- Memoria de la práctica.
- Ejecutables en código Java con las clases pedidas en el enunciado, estas son:
  - "*Complejidad.java*", la clase que ejecutará los algoritmos.
  - "*MatrizDeTiempos.java*", la clase que guardará los tiempos en un fichero Excel.
  - "*VectorOrdenable.java*", la clase que contiene el código de los algoritmos.
  - "*VectorOrdenableVolatile.java*", igual que la anterior, pero haciendo uso del atributo *volatile* en Java.
  - "*Main.java*", la clase principal.
- Fichero Excel con los resultados.

Estudio teórico

En esta sección estudiaremos los resultados de T(n) para el mejor y peor para cada uno de los algoritmos. También describiremos a que O(n) pertenecen los algoritmos propuestos.

Bubble sort

Algoritmo

```
private static void burbuja(int[] v) {
        for (int i = 0; i < v.length - 1; i++) {
            for (int j = v.length - 1; j > i; j--){
                if (v[j-1] > v[j]){
                    int temp = v[j-1];
                    v[j-1] = v[j];
                    v[j] = temp;
                }
            }
        }
    }
```

Comenzamos a analizar nuestro algoritmo desde dentro hacia fuera.

T_{if}: 3 + 4 + 2 = 13.

Continuamos analizando el for interno:

$T_{for\text{-}interno}$:

$$\sum_{j=i+1}^{j=n} (13+3)+3+1 = 16(n-i)+4$$

Y por último analizamos el for externo:

$T_{for\text{-}externo}$:

$$(\sum_{i=0}^{n-1} (16(n-i)+4)+4)+5$$

$$= \sum_{i=0}^{n-1} 16n - \sum_{i=0}^{n-1} 16i + \sum_{i=0}^{n-1} 8 + 5$$

$$= 16n^2 - \left(\frac{((16(0)) + (16(n-1))) * n}{2}\right) + 8n + 5$$

$$= 16n^2 - 8n^2 + 8n + 8n + 5$$

$$= 8n^2 + 16n + 5$$

Una vez finalizado podemos concluir que el algoritmo pertenece a **O(n^2)**.

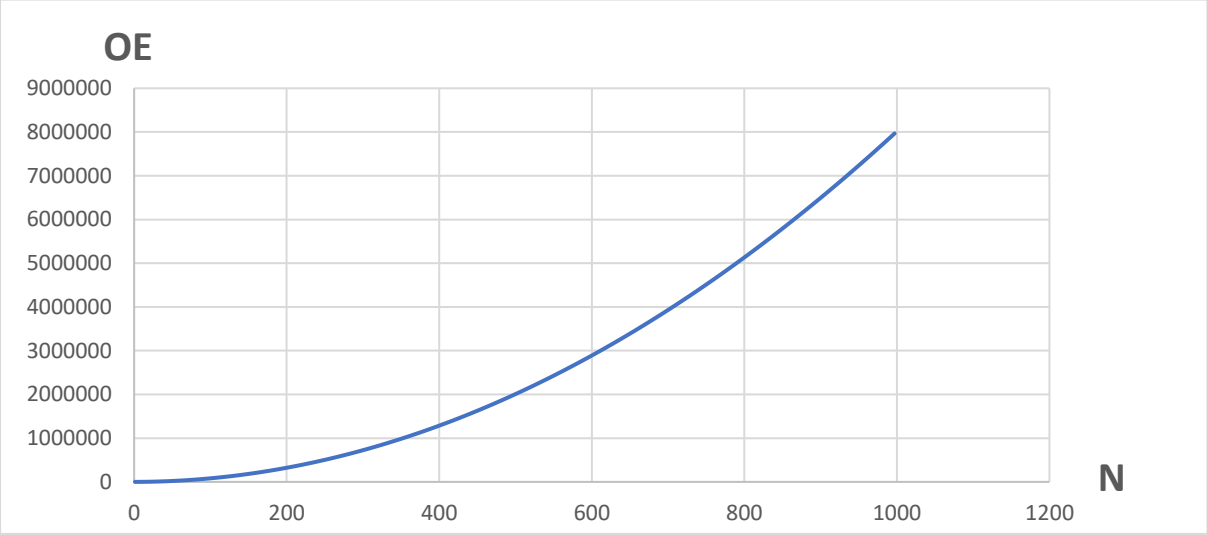

Ilustración 1: gráfica teórica algoritmo de selección

Selection sort

Algoritmo

```
private static void selectSort(int[] v) {
        int m;
        for (int i = 0; i < v.length - 1; i++) {
            m = i;
            for (int j = i + 1; j < v.length; j++)
                if (v[j] < v[m])
                    m = j;
            int aux = v[i];
            v[i] = v[m];
            v[m] = aux;
        }
    }
```

Comenzamos analizando primero el *if* interno del algoritmo:

T_{if}: 3 + 1 = 4.

Continuamos estudiando el comportamiento del *for* interno:

$T_{for\text{-}interno}$:

$$(\sum_{j=i+1}^{j=n} (4+3)+8)+4 = 15(n-i)+4$$

Finalizamos con el estudio del *for* externo:

$T_{for\text{-}externo}$:

$$(\sum_{i=0}^{i=n-1} (15(n-i)+4)+4)+4$$

$$= 15n\text{^}2 - 7{,}5n\text{^}2 - 7{,}5n + 8n + 4$$

$$= 7{,}5n\text{^}2 + 0{,}5n + 4$$

Tras estos cálculos podemos afirmar que el algoritmo pertenece a **O(n^2)**.

Esta fórmula aplicándola teóricamente y obtenemos la siguiente gráfica, donde el eje X se refiere al tamaño del vector y el eje Y al tiempo en ms:

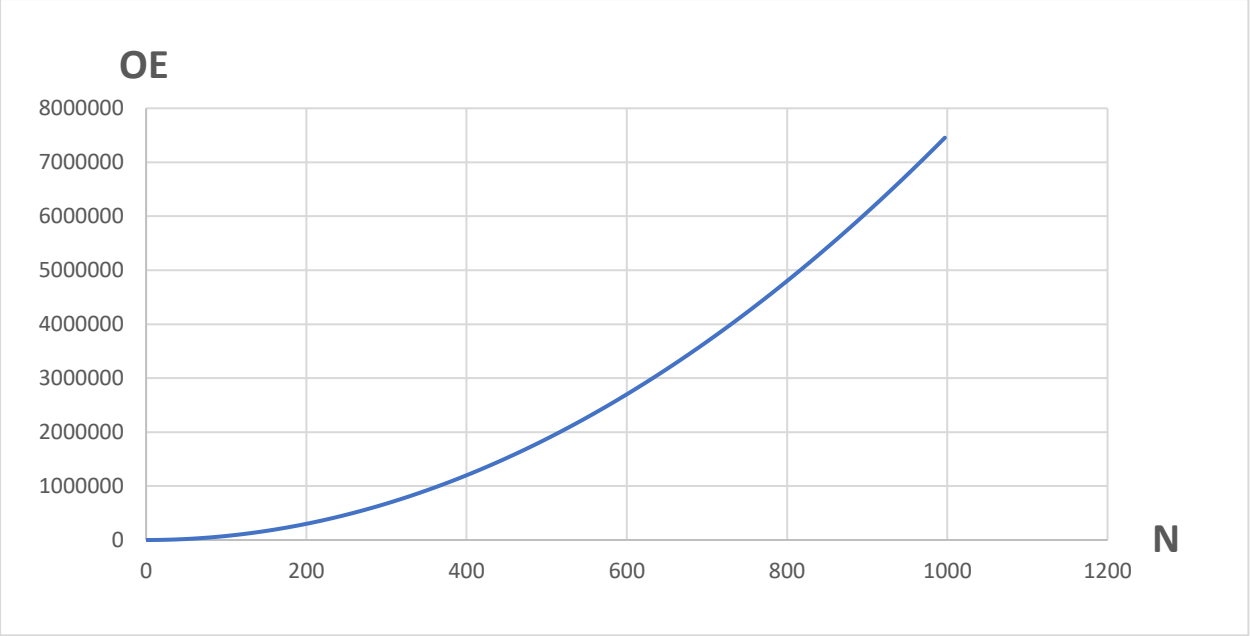

Ilustración 2: gráfica teórica algoritmo de selección

Merge sort

Algoritmo

```
private static void mergeSort(int[] v){
    sort (v, 0, v.length-1);
}
private static void sort(int[] v, int l, int r){
    if(l < r){
        int m = l + (r-l)/2;
        sort(v, l, m);
        sort(v, m + 1, r);
        merge(v, l, m, r);
    }
}
private static void merge(int[] v, int l, int m, int r){
    int n1 = m - l + 1;
    int n2 = r - m;
    int[] left = new int[n1];
    int[] right = new int[n2];
    for(int i = 0; i < n1; i++) left[i] = v[l+1];
    for(int j = 0; j < n2; j++) right[j] = v[m + 1 + j];
    int i = 0, j = 0;
    int k = l;
    while(i < n1 && j < n2){
        if(left[i] <= right[j]){
            v[k] = left[i];
            i++;
        }else{
            v[k] = right[j];
            j++;
        }
        k++;
    }
    while(i < n1){
        v[k] = left[i];
        i++;
        j++;
    }
    while(j < n2){
        v[k] = right[j];
        j++;
        k++;
    }
}
```

Comenzamos analizando la fórmula de recurrencia:

$$T(n) = \begin{cases} 1 & ,n = 1 \\ 2T(\frac{n}{2}) + 16n + 17 & ,\forall n > 1 \end{cases}$$

Esta sigue la forma de una formula maestra:

$$Si\ a, b, b' \in \mathbb{R}^+, c \in \mathbb{N}\ con\ c > 1, d \in \mathbb{R},$$

$$f(n) = \begin{cases} b' & ,1 \leq n \leq n_0 \\ af\left(\frac{n}{c}\right) + bn + d & ,n > n_0 \end{cases}$$

$$Entonces:$$

$$Si\ a > c \rightarrow f \in O\left(n^{\log_c a}\right)$$

$$Si\ a < c \rightarrow f \in O(n)$$

$$Si\ a = c \rightarrow f \in O(n \log_c n)$$

$$Nuestra\ T(n) tiene\ b' = 1, n_0 = 1, a = 2, c = 2, b = 16, d = 17$$

$$Por\ lo\ que\ cumple\ que\ a = c, y\ t \in O(n \log_2 n)$$

Para resolver la ecuación de recurrencia primero haremos un cambio de variable:

$$T(n) = 2T\left(\frac{n}{2}\right) + 16n + 17$$

$$Si\ n = 2^k\ entonces:$$

$$T\left(2^k\right) = 2T\left(2^{k-1}\right) + 2^{k+4} + 17$$

$$Si\ T\left(2^k\right) = f(k) entonces:$$

$$f(k) = 2f(k-1) + 2^{k+4} + 17$$

$$f(k) - 2f(k-1) = 2^{k+4} + 17$$

Sacamos la ec. caracteristica:

$$k = 1, a_0 = 1, a_1 = -2b_1 = 1, p_1(n) = 1, d_1 = 0$$

$$ec.caracteristica: (x-2)^2 = 0$$

Por lo que la nueva f(k) es:

$$f(k) = c_1 2^k + c_2 k 2^k$$

Desacemos los cambios de variables:

$$T\left(2^k\right) = c_1 2^k + c_2 k 2^k$$

$$T(n) = c_1 n + c_2 n \log_2 n$$

Calculamos c_1 y c_2 con T(1):

$$T(1) = 1 = c_1 n + c_2 n \log_2 n \rightarrow c_1 + c_2\,(0) = 1 \rightarrow c_1 = 1, c_2 = 1$$

Por lo que queda:

$$T(n) = n + n\log_2 n \subset O(n\log_2 n)$$

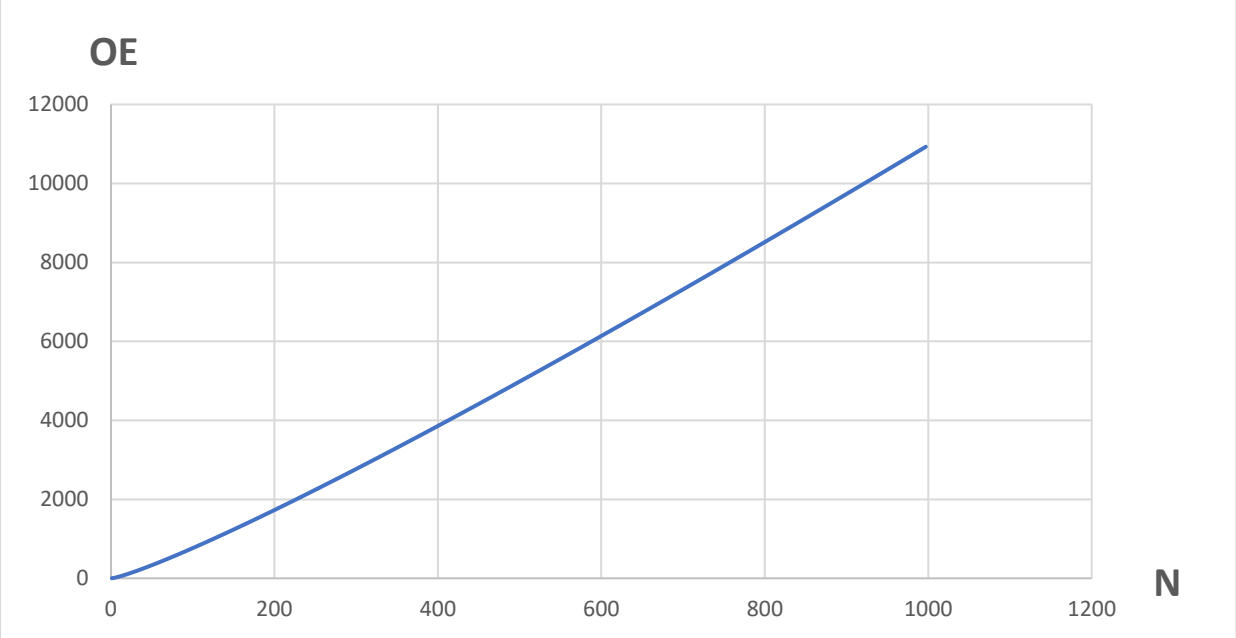

Ilustración 3: gráfica teórica algoritmo de mezcla

Estudio empírico

Las ilustraciones siguientes están medidas en:

- Eje x: tamaño del array de datos.
- Eje y: tiempo en ms de ejecución de ordenación.

Los colores de las líneas representan:

- **Azul**: algoritmo que **no hace uso del atributo "*volatile*"**.
- **Naranja**: algoritmo que **hace uso del atributo "*volatile*"**.

Gráficas burbuja

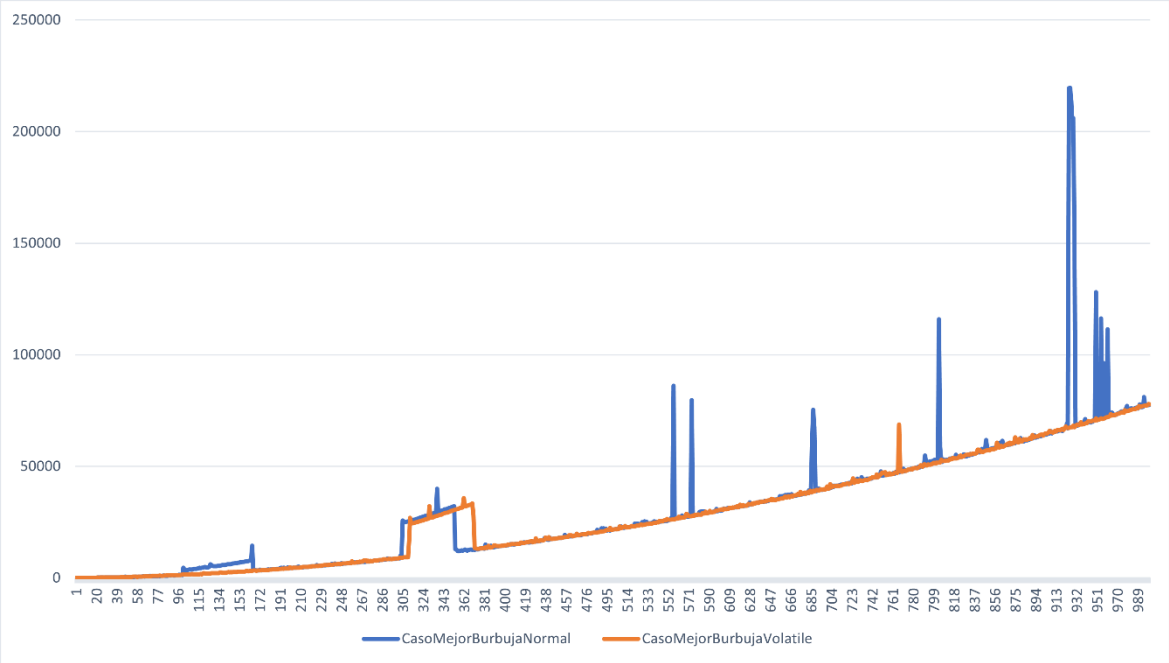

Ilustración 4: Tiempo medido algoritmo burbuja – caso mejor

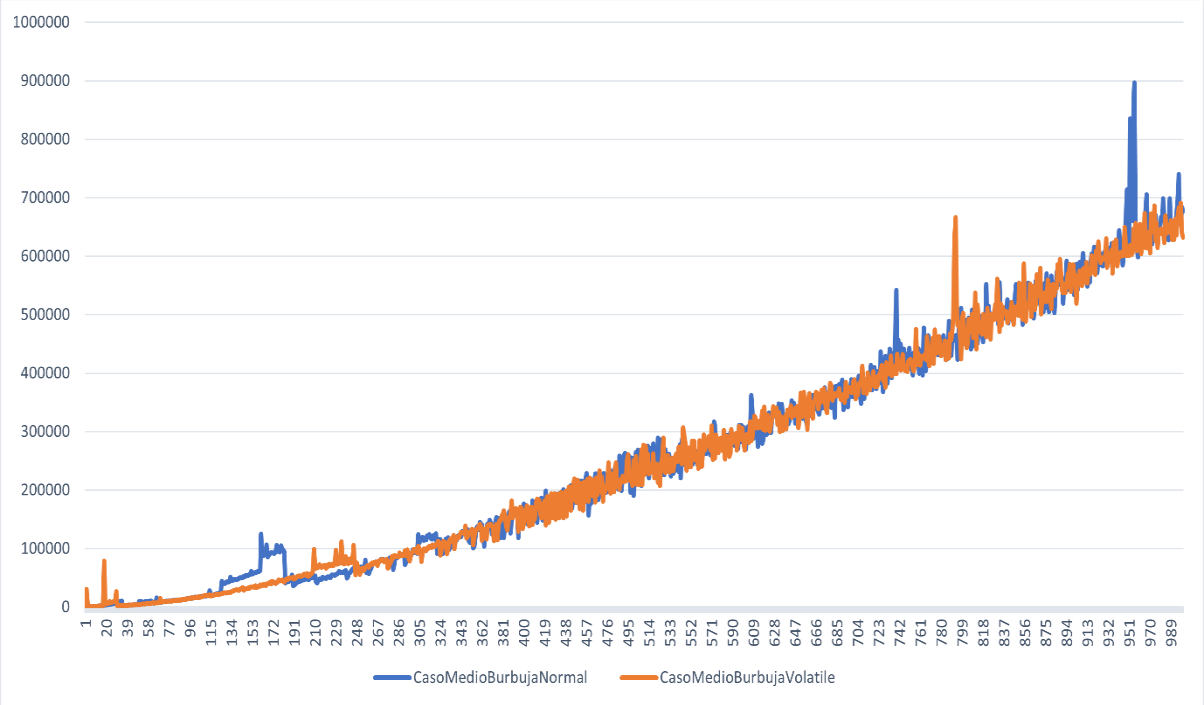

Ilustración 5: Tiempo medido algoritmo burbuja – caso medio

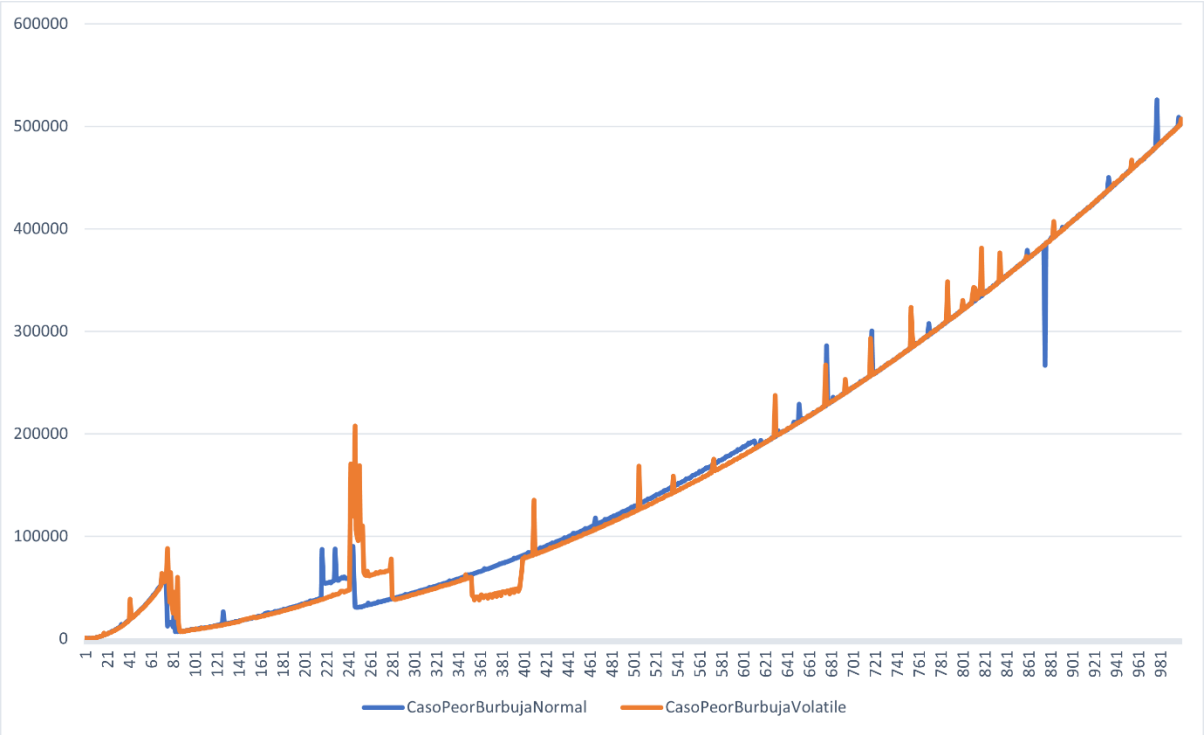

Ilustración 6: Tiempo medido algoritmo burbuja - caso peor

Gráficas selección

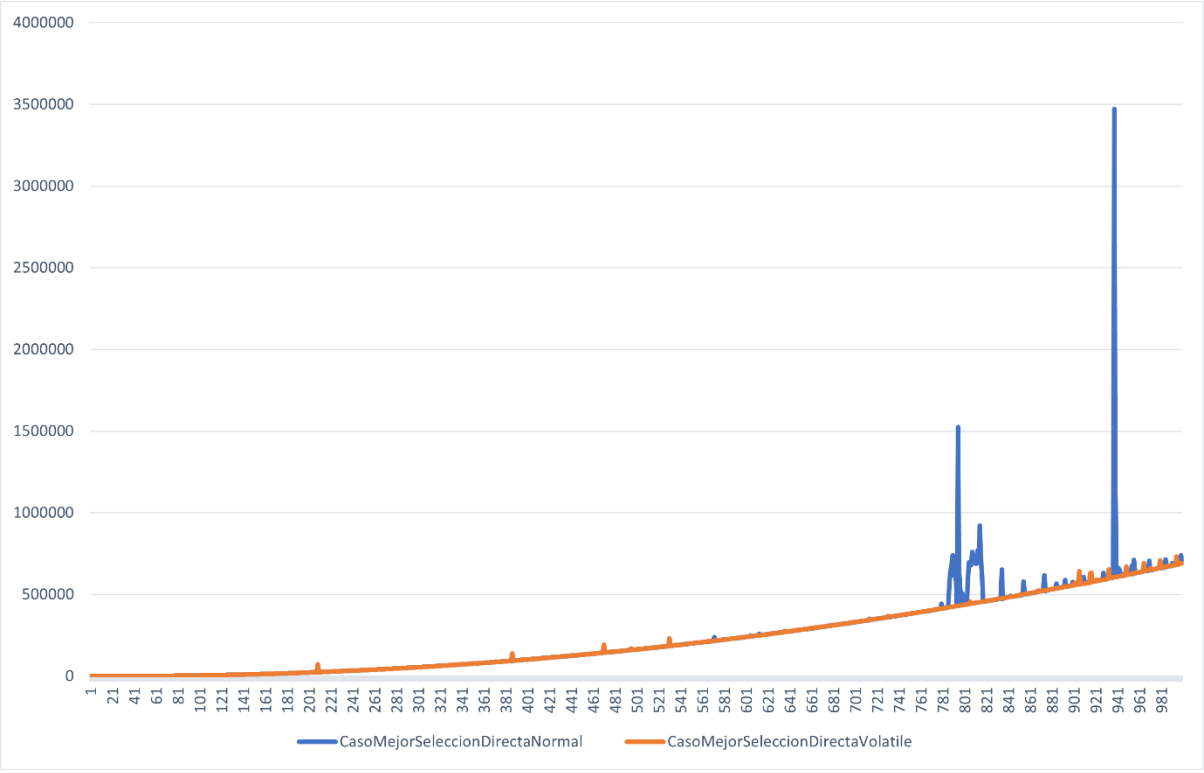

Ilustración 7: Tiempo medido algoritmo selección - caso mejor

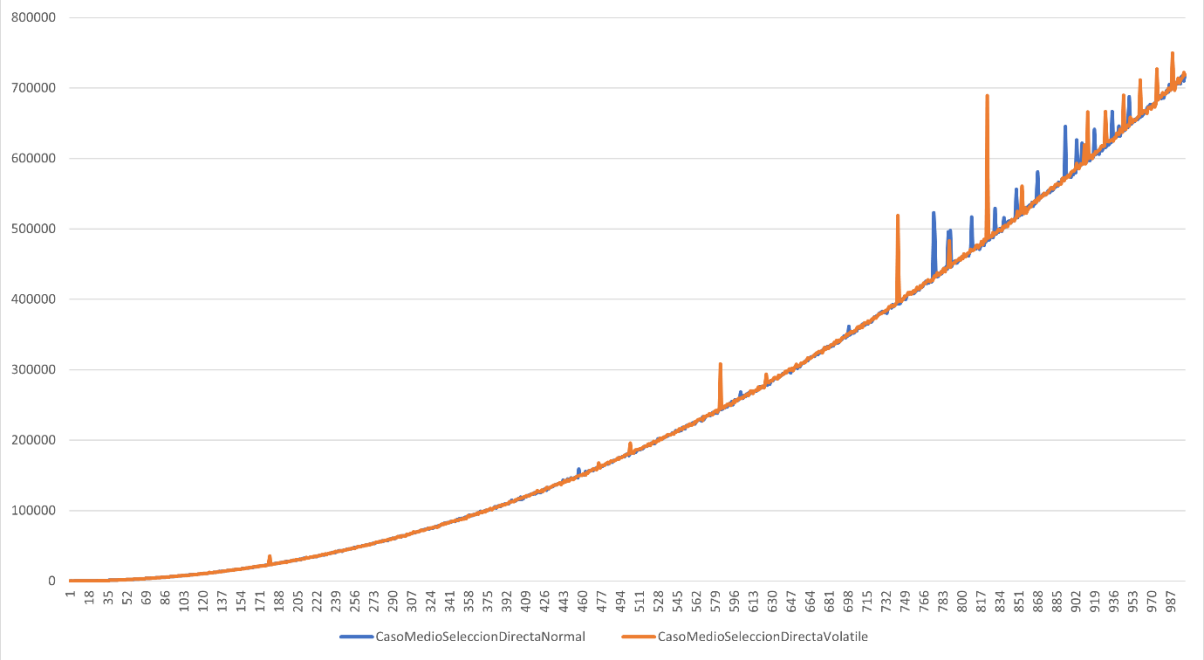

Ilustración 8: Tiempo medido algoritmo selección - caso medio

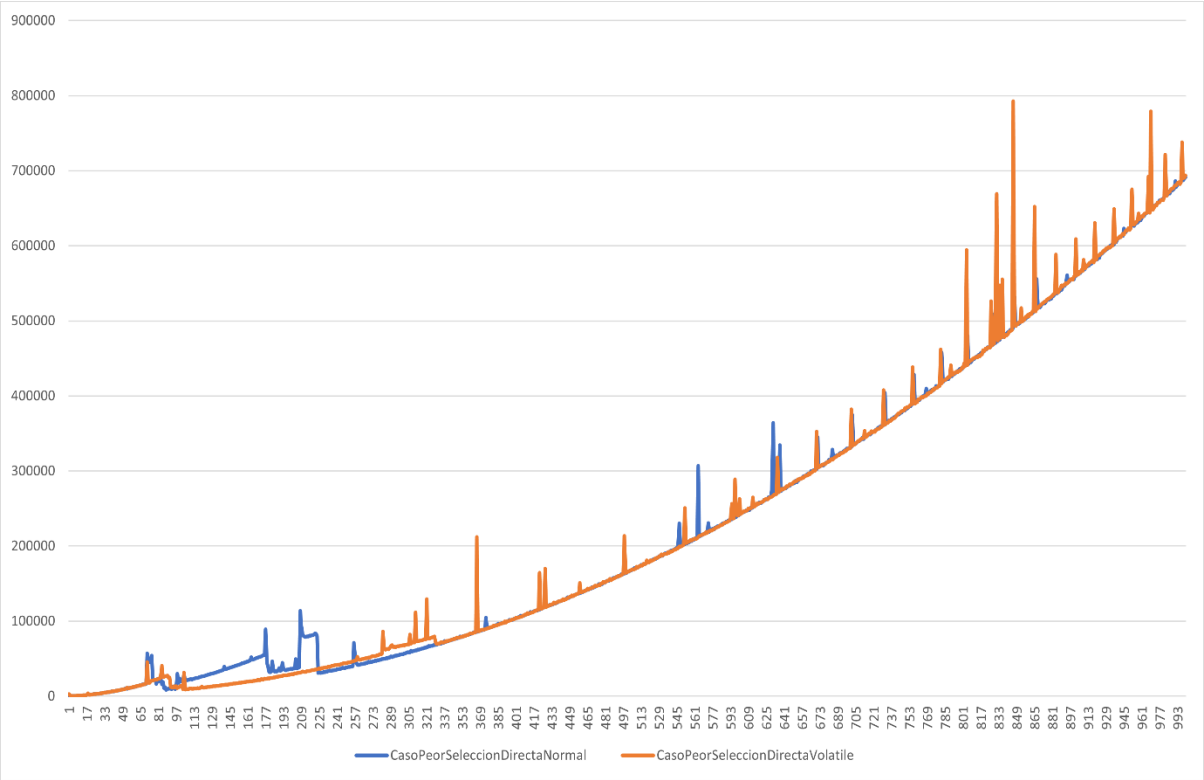

Ilustración 9: Tiempo medido algoritmo selección - caso peor

Gráficas mezcla

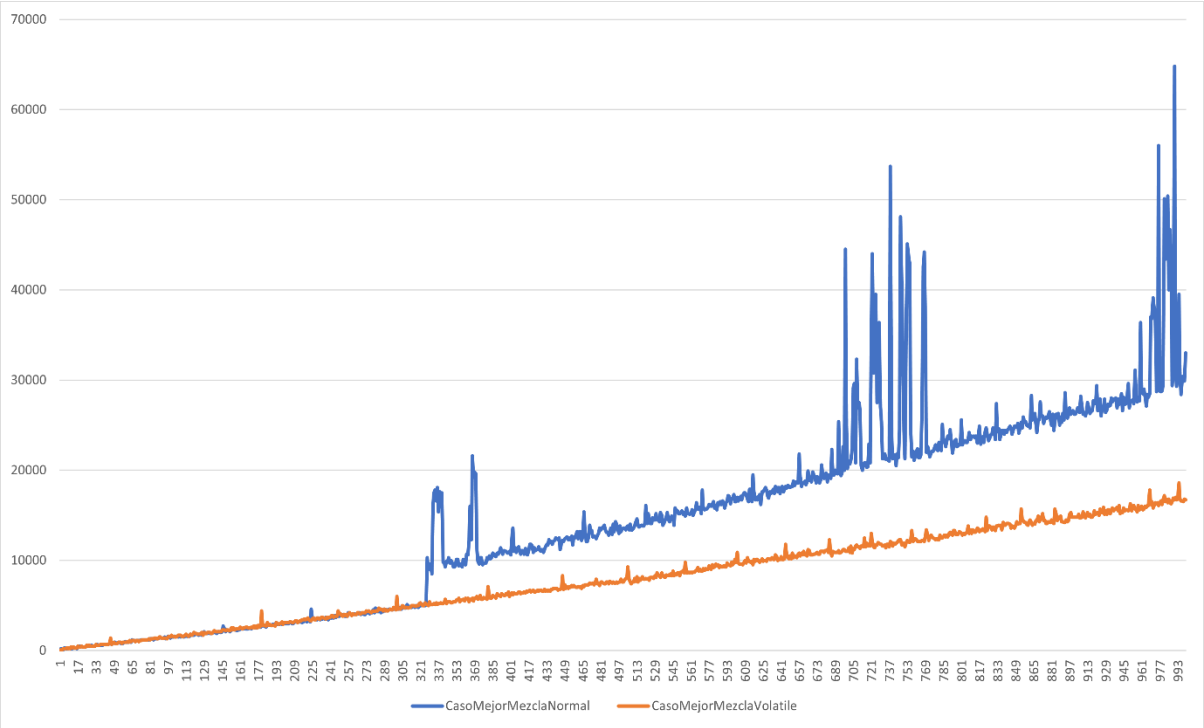

Ilustración 10: Tiempo medido algoritmo mezcla - caso mejor

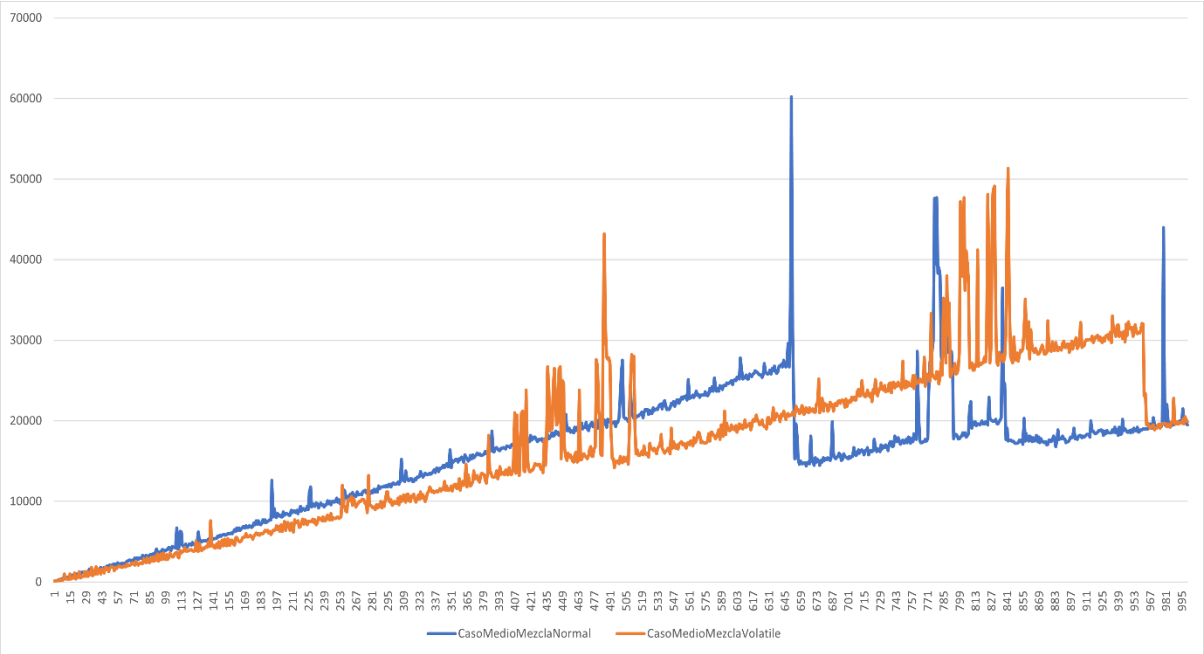

Ilustración 11: Tiempo medido algoritmo mezcla - caso medio

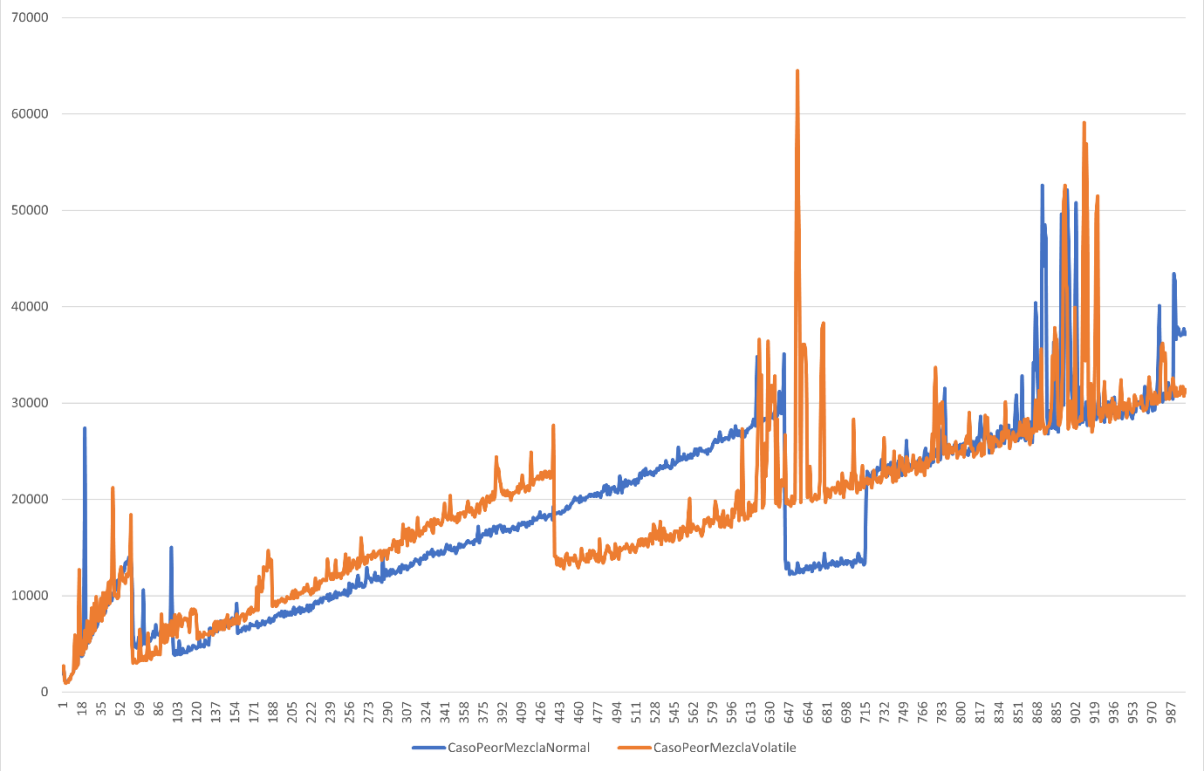

Ilustración 12: Tiempo medido algoritmo mezcla - caso peor

Preguntas cortas

¿Corresponden los resultados empíricos con los experimentales?

Podemos afirmar que las gráficas con mayor parecido entre los resultados empíricos y experimentales son las gráficas tóricas para los resultados empíricos y las de mejor caso para cada uno de los algoritmos presentados.

Sin embargo, observamos como se cumplen los órdenes de O(n) para todos los algoritmos a medida que van incrementando los tamaños de los vectores a ordenar.

¿Cuál de los tres algoritmos de ordenación se comporta mejor?

Tras estudiar caso por caso, es obvio pensar que el algoritmo de mezcla no solo tuvo tiempos menores que el de selección y el de la burbuja, sino que también sus curvas son de pendiente menor. Por estos dos aspectos claves el algoritmo más optimo es sin duda el de mezcla, tanto para vectores grandes como pequeños. Podemos deducir esto ya que el algoritmo utiliza la técnica de divide y vencerás.

¿Existen diferencias importantes entre el mejor, peor y caso intermedio?

En el caso de la burbuja, el mejor caso es el que menos tarda, pero en los casos peor y medio ocurre algo curioso, el caso peor tarda menos que el medio por una diferencia de hasta 10.000 nanosegundos.

Por parte del algoritmo de selección, se observa que en los tres casos los tiempos son muy semejantes, por lo que no hay casi diferencia entre casos.

Y por último en el algoritmo de mezcla, aunque menos similares que en el de selección si se puede decir que no hay casi diferencias entre casos.

¿Qué conjunto de datos es mejor para analizar los algoritmos de ordenación (los generados con volatile o los generados sin volatile)?

La teoría presenta que aquellos datos con "*volatile*" son más rápidos de acceder, esto quiere decir que aquellos datos con este atributo serán iterados y utilizados antes que ningún otro. Sin embargo y en la práctica, podemos observar en las anteriores gráficas que los resultados son muy parecidos. El atributo presentado "*volatile*" sí que genera cierta rapidez a la hora de ejecutar los algoritmos, sin embargo, esta velocidad también depende de factores externos como pueden ser los componentes del ordenador donde se está ejecutando el algoritmo, del numero de llamadas y creación de objetos para la ejecución e incluso de lo ocupado que este el procesador del equipo en ese momento.

Conclusiones

Consideramos esta practica fundamental para comprobar que en muchas ocasiones la teoría puede dar un vuelco en la práctica como hemos visto en las gráficas presentadas y que hay que tener en cuenta otros factores a la hora de calcular los tiempos de los algoritmos y no solo los datos teóricos.

También consideramos interesante la comparación de las gráficas experimentales o teóricas frente a las empíricas.

Referencias

- Temario del aula virtual
- https://es.wikipedia.org/wiki/Ordenamiento_de_burbuja
- https://es.wikipedia.org/wiki/Ordenamiento_por_selección
- https://es.wikipedia.org/wiki/Ordenamiento_por_mezcla