ALGORITMOS DE ORDENACIÓN

Practica 1

Estructura de datos

Universitat de Lleida

Míriam Rodríguez Carranza y Abraham Ruiz Mosteirin

Índice:

Introducción:

En este trabajo ampliaremos nuestros conocimientos sobre algoritmos de ordenación para poder implementarlos, tal y como se nos pide, y a tomar decisiones sobre la implementación de los algoritmos teniendo en cuenta el coste de estos. Además aprenderemos a complementar nuestros proyectos con test unitarios.

insertionSort:

Este algoritmo ya implementado nos sirve para tener una referencia y poder empezar con el resto de las implementaciones.

La ordenación por inserción funciona insertando iterativamente cada elemento de una lista desordenada en su posición correcta en una parte ordenada de la lista. Es un algoritmo de ordenación estable, lo que significa que los elementos con valores iguales mantienen su orden relativo en el resultado ordenado.

Aunque se nos de este algoritmo echo, calcularemos los costes para poder compararlo con los otros métodos de ordenación.

bubbleSort:

Este algoritmo de ordenación trabaja revisando cada elemento del array que va a ser ordenado con el siguiente, intercambiándolos de posición si están en el orden equivocado.

El algoritmo implementado es el siguiente:

```
public void bubbleSort() {
    int end = 0;
    int pos = array.length;
    while (pos != end) {
      for (int i = pos - 1; i > end; i--) {
        if (array[i] < array[i - 1]) {
          swap(i, i - 1);
        }
      }
      end++;
    }
}
```

selectionSort:

Este algoritmo mejora ligeramente el algoritmo de la burbuja. En el caso de tener que ordenar un vector de enteros, esta mejora no es muy sustancial. Este algoritmo realiza muchas menos operaciones swap(i,j) que el de la burbuja, por lo que lo mejora un poco.

Otra desventaja de este algoritmo respecto a otros como el de burbuja o de inserción directa es que no mejora su rendimiento cuando los datos ya están ordenados o parcialmente ordenados. Es necesario recorrer el array n veces para saber si esta ordenado.

```
public void selectionSort() {
    for (int i = 0; i < array.length - 1; i++) {
      int min = i;
      for (int j = i + 1; j < array.length; j++) {
        if (array[j] < array[min]) {
          min = j;
        }
      }
      if (min != i) {
        swap(i, min);
      }
    }
}
```

quickSort:

Este es el algoritmo de ordenación predeterminado en las librerías de la mayoría de los lenguajes de programación. Este algoritmo se implementa de forma recursiva. La gracia de este algoritmo es que trabajamos usando una posición como pivot con la que “partimos” el array y comparamos los números con arrray[pivot] para poder ordenar los números de cada partición.

partitionIterative:

Se nos pide la implementación de la partición del array de forma iterativa.

En la primera implementación que hicimos cometimos el error de solucionar el problema con bucles que incrementan el coste del algoritmo, por lo tanto, hemos decidido que la implementación con condicionales es la mejor opción.

```
*Primera implementación*
private int partitionIterative(int pivotValue, int inf, int sup) {
    // 0 <= left <= inf <= sup <= right <= v.length
    while (inf != sup){
        while (array[inf] <= pivotValue) {
          inf++;
        }
        while (array[sup - 1] > pivotValue) {
          sup--;
        }
        swap(inf, sup - 1);
        inf++;
    }return inf;
}
```

Esta ha sido la implementación final, sobre la cual haremos el cálculo de costes:

```
*Implementación final*
private int partitionIterative(int pivotValue, int inf, int sup) {
    // 0 <= left <= inf <= sup <= right <= v.length
    while (inf != sup) {
```

```
        if (array[inf] <= pivotValue) {
            inf++;
        } else if (array[sup - 1] > pivotValue) {
            sup--;
        } else {
            swap(inf, sup - 1);
            inf++;
        }
    }
    return inf;
}
```

Test:

Hemos implementado tres test adicionales(“negative”, “sorted_negative” y “unsorted_negative”) para comprobar el funcionamiento de los algoritmos con números negativos. Estos test presentan la misma estructura que los proporcionados, de manera que, solo hemos cambiado los valores del array pasado como parámetro.

Costes:

Los costes los hemos calculado a mano y posteriormente los hemos comparado con los resultados obtenidos de ejecutar Mybenchmark, de esta forma tenemos una visión mas real de la complejidad computacional de nuestro código.

Coste insertionSort:

Este algoritmo, ya implementado, tiene un coste O(n^2) ya que tenemos un bucle “for” anidado dentro de otro, haciendo que el coste en el peor de los casos sea exponencial. El coste en el mejor de los casos es O(n).

Comparemos con los cálculos del Benchmark:

N (Tamaño del array)	Tiempo promedio (ns/op)	Error (ns/op)
100	5,771.442	2,306.908
200	24,752.154	1,790.754
400	88,209.920	4,006.413
800	330,808.738	14,216.785

De N=100 a N=200, el tiempo pasa de 5,771.442 ns a 24,752.154 ns.

- Aumento en N: se duplica.
- Aumento en el tiempo: 24,752.154 / 5,771.442 ≈ 4.29

De N=200 a N=400, el tiempo pasa de 24,752.154 ns a 88,209.920 ns.

- Aumento en N: se duplica.
- Aumento en el tiempo: 88,209.920 / 24,752.154 ≈ 3.56

De N=400 a N=800, el tiempo pasa de 88,209.920 ns a 330,808.738 ns.

- Aumento en N: se duplica.
- Aumento en el tiempo: 330,808.738 / 88,209.920 ≈ 3.75

Vemos que el tiempo no augmenta de forma cuadrática como tal, aún asi, sigue una tendencia cercana a O(n^2). Podemos esperar este comportamiento en algoritmos cuadráticos, como el de Inserción o el de burbuja.

Coste bubbleSort:

El coste que hemos calculado es O(n^2) ya que tenemos un bucle “while” que hace ‘n’ vueltas y otro bucle “for” dentro de este que también hace ‘n’ vueltas en el peor caso, el resto de condicionales y definición de variables tiene un coste 0(1) el cual es irrelevante en este calculo.

Vamos a analizar los resultados del benchmark:

N (Tamaño del array)	Tiempo promedio (ns/op)	Error (ns/op)
100	17,231.339	5,951.690
200	64,428.466	11,595.393
400	216,120.579	8,449.885
800	729,304.026	45,115.504

Idealmente, para un algoritmo con complejidad O(n^2), el tiempo debería aumentar cuadráticamente.

De N=100 a N=200, el tiempo pasa de 17,231.339 ns a 64,428.466 ns.

- Aumento en N: se duplica.
- Aumento en el tiempo:64,428.466 / 17,231.339 ≈ 3.74

De N=200 a N=400, el tiempo pasa de 64,428.466 ns a 216,120.579 ns.

- Aumento en N: se duplica.
- Aumento en el tiempo: 216,120.579 / 64,428.466 ≈ 3.35

De N=400 a N=800, el tiempo pasa de 216,120.579 ns a 729,304.026 ns.

- Aumento en N: se duplica.
- Aumento en el tiempo: 729,304.026 / 216,120.579 ≈ 3.37

El tiempo de ejecución no aumenta exactamente al cuadrado, pero vemos que aumenta más de 3 veces cada vez que duplicamos el tamaño del array. Esto sugiere que el algoritmo sí tiene un crecimiento cercano a O(n^2) en la práctica, lo cual es coherente con el análisis teórico.

Coste selectionSort:

El algoritmo de selección vuelve a presentar un coste cuadrático anidando un bucle “for” dentro de otro. Con la ayuda del benchmark podremos determinar cuan se acerca el coste teórico al real.

N (Tamaño del array)	Tiempo promedio (ns/op)	Error (ns/op)
100	9,985.840	721.623
200	32,967.672	1,802.956
400	114,560.190	26,907.678
800	373,083.488	14,152.739

De N=100 a N=200, el tiempo pasa de 9,985.840 ns a 32,967.672ns.

- Aumento en N: se duplica.
- Aumento en el tiempo: 32,967.672 / 9,985.840≈ 3.30

De N=200 a N=400, el tiempo pasa de 32,967.672 ns a 114,560.190 ns.

- Aumento en N: se duplica.
- Aumento en el tiempo: 114,560.190 / 32,967.672 ≈ 3.47

De N=400 a N=800, el tiempo pasa de 114,560.190 ns a 373,083.488ns.

- Aumento en N: se duplica.
- Aumento en el tiempo: 373,083.488 / 114,560.190 ≈ 3.26

En este algoritmo observamos el mismo comportamiento que en los anteriores, volvemos a tener un coste cercano al cuadrático aunque no exactamente, diremos que el coste practico es cuadrático.

Coste quickSort:

Este algoritmo en el peor caso tiene un coste de O(n^2) pero un coste esperado de O(n*log n). Veamos los resultados obtenidos:

N (Tamaño del array)	Tiempo promedio (ns/op)	Error (ns/op)
100	4,693.669	1,723.208
200	10,644.799	919.742
400	24,012.142	1,105.154
800	52,524.070	1,889.187

De N=100 a N=200, el tiempo pasa de 4,693.669 ns a 10,644.799 ns.

- Aumento en N: se duplica.
- Aumento en el tiempo: 10,644.799 / 4,693.669 ≈ 2.27

De N=200 a N=400, el tiempo pasa de 10,644.799 ns a 24,012.142 ns.

- Aumento en N: se duplica.
- Aumento en el tiempo: 24,012.142 / 10,644.799 ≈ 2.26

De N=400 a N=800, el tiempo pasa de 24,012.142 ns a 52,524.070 ns.

- Aumento en N: se duplica.

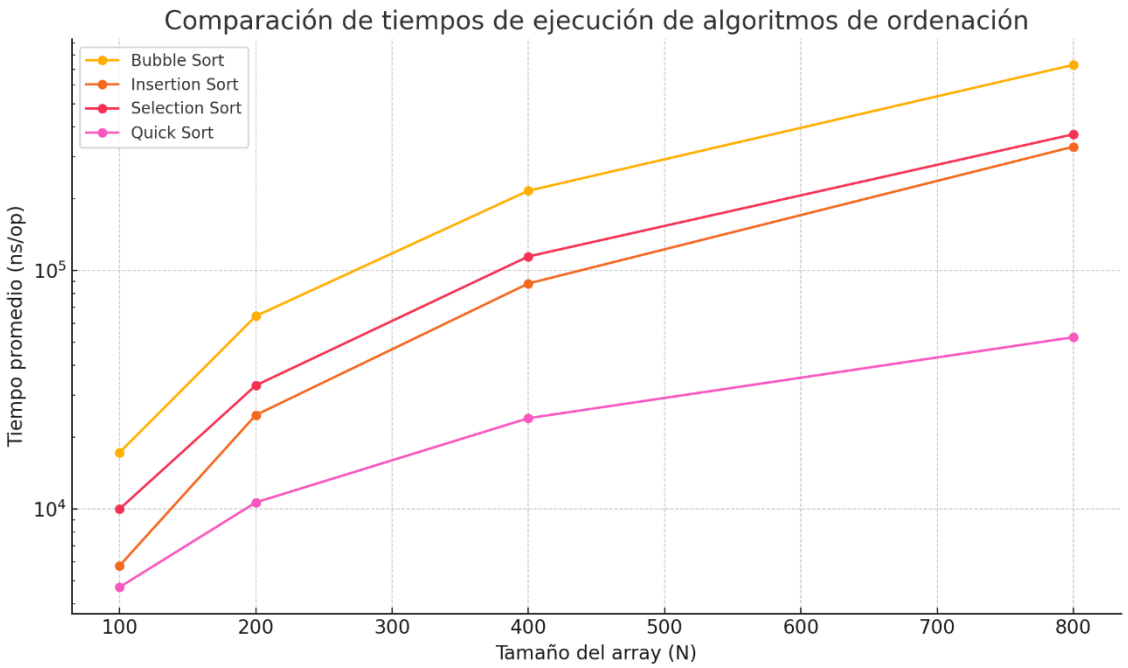
Aumento en el tiempo: 52,524.070 / 24,012.142 ≈ 2.19

Observamos que el tiempo de ejecución aumenta, aproximadamente, de forma lineal-logarítmica, como era el caso esperado, aun viendo que el tiempo se duplica un poco más del doble concuerda con el comportamiento normal de un algoritmo QuickSort promedio.

Conclusiones:

En conclusión vemos que, como era de esperar, el algoritmo QuickSort es el más eficiente de los cuatro que hemos trabajado en esta práctica, lo vemos tanto en los costes teóricos que hemos calculado como en los datos obtenidos por el benchmark.

Hemos hecho un grafico con los datos del benchmark de forma que sea más fácil visualizar el comportamiento de estos algoritmos.



Al trabajar con algoritmos conocidos ha sido fácil comparar los resultados esperados con los que teníamos.

Algunas observaciones del grafico:

Quick Sort es claramente el algoritmo más eficiente, con un crecimiento más lento en comparación con los otros tres, reflejando su complejidad esperada de O(n*logn).

Bubble Sort es el más ineficiente, mostrando tiempos significativamente mayores, lo cual concuerda con su complejidad O(n^2).

Insertion Sort y Selection Sort tienen rendimientos intermedios, ambos también con un crecimiento cuadrático, pero Selection Sort tiende a ser un poco más lento.

Esto confirma que Quick Sort es superior para tamaños de entrada grandes, mientras que los otros algoritmos cuadráticos no escalan bien con el tamaño del array.

Bibliografía:

Apuntes Estructura de datos 24/25:

Algoritmes d'Ordenació (v1).pdf

QuickSort (v1).pdf

Gráficos:

https://chatgpt.com/

Video quickSort:

https://youtu.be/UrPJLhKF1jY?si=wG652BIHP0XBz3YN

Wikipedia:

https://www.geeksforgeeks.org/insertion-sort-algorithm/

https://es.wikipedia.org/wiki/Ordenamiento_de_burbuja

https://es.wikipedia.org/wiki/Ordenamiento_por_selecci%C3%B3n