Memoria Práctica 1: Análisis de Métodos de Ordenación

Alejandro Trujillo Caballero

18 de Noviembre de 2014

Índice

1. Introducción

Los algoritmos de ordenación pretenden reorganizar los elementos de un array de manera que queden ordenados según el criterio que queramos (orden creciente o decreciente para números, orden alfabético para letras o palabras...).
En esta práctica vamos a crear un programa para estudiar y comparar la eficiencia temporal de cinco algoritmos que se utilizan para esta función: ordenación por burbuja, inserción, shell, MergeSort y QuickSort.

2. Análisis Teórico

En este apartado se mostrarán los códigos de los algoritmos a estudiar así como su orden de complejidad temporal. Los cálculos detallados se omitirán ya que el objetivo principal de la práctica es comprobar el ajuste de estos órdenes teóricos con la realidad, no el cálculo de órdenes en si mismo.

2.0.1. Burbuja

Código:

```
void Burbuja(int t[], int ini, int fin) {
    int i, j;
    int aux;
    int size = fin - ini;

    for (i = 0; i < size - 1; i++) {
        for (j = size - 1; j > i; j--) {
            if (t[j] < t[j - 1]) {
                aux = t[j];
                t[j] = t[j - 1];
                t[j - 1] = aux;
            }
        }
    }
}
```

En este caso fijandonos en que ejecuta dos bucles for anidados, es fácil ver que harán que para el caso medio el algoritmo sea de orden cuadrático.

$$T_{Burbuja}(n) \in O(n^2) \tag{1}$$

2.0.2. Inserción

Código:

```
void Insercion(int t[], int ini, int fin) {
    int size = fin - ini;
    for (int i = 1; i < size; i++) {
        int x = t[i];
        int j = i - 1;

        while (j >= 0 && x < t[j]) {
```

```
                t[j + 1] = t[j];
                j--;
            }

            t[j + 1] = x;
        }
    }
```

Para el caso medio, al igual que en el caso anterior, el algoritmo anida dos bucles, por tanto:

$$T_{Insercion}(n) \in O(n^2) \tag{2}$$

2.0.3. Shell

Código:

```
void Shell(int t[], int ini, int fin) {
    fin--;
    int i, j, inc, temp, TAM = fin + 1;
    for (inc = TAM / 2; inc > 0; inc /= 2) {
        for (i = inc; i < TAM; i++) {
            temp = t[i];
            for (j = i; j >= inc; j -= inc) {
                if (temp < t[j - inc]) {
                    t[j] = t[j - inc];
                } else {
                    break;
                }
            }
            t[j] = temp;
        }
    }
}
```

La complejidad temporal de este algoritmo varía en función de la secuencia de incrementos utilizada, para nuestro caso:

$$T(n) \in O(n \log(n)) \tag{3}$$

2.0.4. MergeSort

Código:

```
void mergesort(int *a, int*b, int low, int high) {
    int pivot;
    if (low < high) {
        pivot = (low + high) / 2;
        mergesort(a, b, low, pivot);
        mergesort(a, b, pivot + 1, high);
        merge(a, b, low, pivot, high);
    }
}
```

```
10
11  void merge(int *a, int *b, int low, int pivot, int high) {
12      int h, i, j, k;
13      h = low;
14      i = low;
15      j = pivot + 1;
16
17      while ((h <= pivot)&&(j <= high)) {
18          if (a[h] <= a[j]) {
19              b[i] = a[h];
20              h++;
21          } else {
22              b[i] = a[j];
23              j++;
24          }
25          i++;
26      }
27      if (h > pivot) {
28          for (k = j; k <= high; k++) {
29              b[i] = a[k];
30              i++;
31          }
32      } else {
33          for (k = h; k <= pivot; k++) {
34              b[i] = a[k];
35              i++;
36          }
37      }
38      for (k = low; k <= high; k++) a[k] = b[k];
39  }
```

Teniendo en cuenta las dos llamadas recursivas de MergeSort y que el método merge necesita 2n pasos, obtenemos la ecuación de recursividad:

$$T(n) = 2n + 2T(n/2) \tag{4}$$

Aplicando el teorema maestro obtenemos:

$$T(n) \in O(n \log(n)) \tag{5}$$

2.0.5. QuickSort

Código:

```
1  void QuickSort::quicksort(int t[], int ini, int fin) {
2      int pivote;
3      if (ini < fin) {
4          pivote = partition(t, ini, fin);
5          quicksort(t, ini, pivote);
6          quicksort(t, pivote+1, fin);
7      }
8  }
9
```

```
10 int QuickSort::partition(int* t, int p, int r) {
11     int piv = t[p];
12     int i = p-1;
13     int j = r+1;
14     do {
15         do {
16             j--;
17         } while (t[j]>piv);
18         do {
19             i++;
20         }while(t[i]<piv);
21         if (i < j) {
22             int aux = t[i];
23             t[i] = t[j];
24             t[j] = aux;
25         }
26     } while(i < j);
27     return j;
28 }
```

En este caso, con las llamadas recursivas y teniendo en cuenta que partition necesita n operaciones, la ecuación de recurrencia a la que llegamos es:

$$T(n) = n + 2T(n/2) \tag{6}$$

Aplicando el teorema maestro obtenemos:

$$T(n) \in O(n \log(n)) \tag{7}$$

3. Análisis Práctico

3.1. Análisis del problema

Para crear el programa, necesitaremos diferentes grupos funcionales diferentes: la implementación de los algoritmos a analizar, la logica de las pruebas, la interfaz gráfica (en este caso salida por consola), elemenos sobre los que ejecutar los algoritmos y el manejo de archivos y gráficas. Para implementar todo esto, utilizaremos:

- Algoritmos a analizar: Una clase abstracta que defina la interfaz de uso, de la que heredarán cada uno de los diferentes algoritmos.

- Lógica de las pruebas: Se implimentará en una clase donde se encontrarán las funciones principales del programa (prueba de los algoritmos, comparaciones...).

- Interfaz gráfica: Se manejara mediante funciones que escriban en la consola los menus necesarios.

- Elementos sobre los que ejecutar los algoritmos: Utilizaremos una clase que nos permita crear vectores con contenidos aleatorios.

- Manejo de archivos y gráficas: Se implementará una clase que unifique todo el sistema del salida de datos a ficheros y creación de representaciones gráficas.

3.2. Diseño de la aplicación

3.2.1. Ordenacion.h

```
#ifndef ORDENACION_H
#define ORDENACION_H

using namespace std;

class Ordenacion {
public:
    /**
     * Contructor vacio
     */
    Ordenacion(){}

    /**
     * Destructor
     */
    virtual ~Ordenacion(){}

    /**
     * Metodo que debera implementar cada clase hija
     * @param t Vector a ordenar
     * @param ini posicion del primer elemento
     * @param fin posicion ultimo elemento
     */
    virtual void ordenar(int t[], int ini, int fin) {}
};


#endif  /* ORDENACION_H */
```

Clase de la que hereda clada uno de los algoritmos estudiados, el método ordenar sera el que implementen las hijas y será llamado a la hora de realizar las pruebas. El código de cada algoritmo es el que se muestra en el análisis teórico por lo que no se mostrará en este apartado.

3.2.2. ConjuntoInt.h

```
////////////// Declaracion de la clase conjuntoInt ////////////

class ConjuntoInt {
private:
    int tamano;
    int nelementos;
    int *datos;
public:
    ConjuntoInt (int max= 0);
    ~ConjuntoInt ();
    void vaciar();
    void insertar(int n);
    bool miembro(int n);
    void ordena(int tam);
    void ordena();
    void escribe(int tam);
    void escribe();
```

```
18      void clonar(ConjuntoInt &c);
19      void generaVector ();
20      int* getDatos();
21  };
```

Esta clase es simplemente una pequeña modificación de la entregada antes de la realización de la práctica para que trabaje con arrays de forma dinámica.

3.2.3. TestOrdenacion.h

```
1
2   #ifndef TESTORDENACION_H
3   #define TESTORDENACION_H
4
5   using namespace std;
6
7   #include <string>
8   #include <vector>
9   #include "Ordenacion.h"
10
11  /**
12   * Struct utilizado para facilitar el manejo de medidas de
13   * tiempo para cada cantidad de elementos
14   */
15  struct Tiempos {
16      int elementos;
17      double tiempo;
18      Tiempos(){}
19      Tiempos(int elem, double t) {
20          elementos = elem;
21          tiempo = t;
22      }
23  };
24
25  struct tOrdenacion {
26      int num_pruebas; // Numero de pruebas para un metodo de ordenacion
27      string nombre_metodo; // Nombre del metodo de ordenacion
28      Ordenacion *estrategia; // puntero al tipo de ordenacion a realizar
29  };
30
31  class TestOrdenacion {
32      vector<tOrdenacion> testMetodos; // contiene todos los metodos a
            estudiar
33      int num_pruebas_al_comparar;
34
35      double ordenarArray(int v[],int size,int metodo);
36  public:
37      /**
38       * Anade los metodos de ordenacion a probar al vector testMetodos
39       */
40      TestOrdenacion();
41      ~TestOrdenacion();
42
43      /**
```

```
44          * Comprueba que los metodos de ordenacion funcionan bien
45          */
46         void comprobarMetodos();
47 
48         /**
49          * Compara todos los metodos de ordenacion entre si.
50          * Permite las opciones de crear el fichero de datos y la grafica
                correspondiente.
51          * met: indice de los metodos de testMetodos a comparar
52          */
53         void compararMetodos(vector<int> met);
54 
55         /**
56          * Estudia un metodo de ordenacion concreto.
57          * Permite las opciones de crear el fichero de datos y la grafica
                correspondiente.
58          */
59         void estudiarMetodo(int m);
60 
61         /**
62          * Facilita la creacion de los menus permitiendo acceder
63          * a los nombres de algoritmos fuera de esta clase
64          * @return vecto con los metodos a analizar
65          */
66         vector<tOrdenacion> getTestMetodos() {
67             return testMetodos;
68         }
69 
70 };
71 
72 #endif  /* TESTORDENACION_H */
```

Esta clase maneja todo el sistema de pruebas de la práctica, la explicación detallada de cada método y atributos se puede leer en los comentarios.

3.2.4. FicherosYGraficas.h

```
1  #ifndef FICHEROSYGRAFICAS_H
2  #define FICHEROSYGRAFICAS_H
3  using namespace std;
4  #include <vector>
5  #include <iostream>
6  #include <fstream>
7  #include <cstdlib>
8 
9  #include "TestOrdenacion.h"
10 
11 class FicherosYGraficas {
12     static TestOrdenacion *test;
13 
14     static void dibujar(string archivo);
15 public:
16 
17     /**
```

```
18          * Funcion que debe llamarse antes de utilizar la clase para definir
19          * la clase TestOrdenacion con la que trabajara y por tanto los nombres
20          * de los algoritmos que imprimira en archivos y graficas
21          * @param _test Istancia de TestOrdenacion con la que trabajar
22          */
23         static void setTest(TestOrdenacion *_test) {
24             test = _test;
25         }
26 
27         /**
28          * Guarda en un archivo los datos de una prueba
29          * @param T array que contiene tallas y tiempos
30          * @param size tamano de T
31          * @param metodo metodo que se esta analizando
32          */
33         static void GuardarEnArchivo(Tiempos *T, int size, int metodo);
34         /**
35          * Dibuja en una grafica los datos de una prueba
36          * @param T array que contiene tallas y tiempos
37          * @param size tamano de T
38          * @param metodo metodo que se esta analizando
39          */
40         static void GenerarGrafica(Tiempos *T, int size, int metodo);
41 
42 
43         /**
44          * Guarda en un archivo los datos de una comparacion
45          * @param T tabla de dos dimensiones con los datos de las pruebas
46          * @param size tamano de T
47          * @param metodos Diferentes metodos que se estan comparando
48          */
49         static void GuardarEnArchivo(Tiempos **T, int size, vector<int> metodos
               );
50         static void GenerarGrafica(Tiempos **T, int size, vector<int> metodos);
51 };
52 
53 
54 #endif  /* FICHEROSYGRAFICAS_H */
```

Esta clase controla la salida de datos a ficheros y gráficas del programa.

3.2.5. main

```
1  #include <iostream>
2  #include <cstdlib>
3  #include "TestOrdenacion.h"
4 
5  using namespace std;
6 
7  #ifdef __linux__
8  #define CLEAR "clear"
9  #else
10 #define CLEAR "cls"
11 #endif
```

```
12
13
14  int Menu() {
15
16      int opcion;
17      do {
18          system(CLEAR);
19          cout << "\n\t\t*** Estudio de Metodos de Ordenacion ***\n\n"
20               << "\t\t1- Probar los metodos de ordenacion" <<  endl
21               << "\t\t2- Obtener el caso medio de un metodo" << endl
22               << "\t\t3- Comparar metodos" << endl
23               << "\t\t0- Salir" << endl
24               << "\t\t--------------------"
25               << "\n\t\t >>> ";
26               cin >> opcion;
27      } while (opcion < 0 || opcion > 3);
28      return opcion;
29  }
30
31
32  int subMenuCasoMedio(TestOrdenacion &test) {
33      int opcion;
34
35      do{
36          cout << "\n\t\t*** Metodo a estudiar para el caso medio ***" <<
                endl;
37          for (int i = 0; i < test.getTestMetodos().size(); i++)
38              cout << "\t\t"<<i+1<<"- " << test.getTestMetodos()[i].
                    nombre_metodo << endl;
39
40          cout << "\n\t\t >>> ";
41          cin >> opcion;
42      } while (opcion < 0 || opcion > test.getTestMetodos().size());
43      return opcion - 1;
44  }
45
46  void subRutinaComparacion(TestOrdenacion &test){
47      vector<int> metodos;
48
49
50
51      system(CLEAR);
52      cout <<  "\n\t\t *** Comparacion de metodos de ordenacion" << endl <<
            endl;
53      for (int i = 0; i < test.getTestMetodos().size(); i++)
54          cout << "\t\t " <<i+1<<"- "<< test.getTestMetodos()[i].
                nombre_metodo << endl;
55
56      cout << "\t\t--------------------\n";
57      int aux;
58      int i = 1;
59      do {
60          do {
61              if (i <= 2)
62                  cout << "\n\t\t Introduzca metodo" << i << ": ";
```

```
63              else
64                  cout << "\n\t\t Introduzca metodo" << i << " (o 0 para
                        terminar): ";
65              cin >> aux;
66          } while (aux == 0 && i <= 2);
67          if (aux <= 5 && aux >= 1) {
68              metodos.push_back(aux-1);
69              i++;
70          }
71      } while (aux != 0);
72
73      test.compararMetodos(metodos);
74
75  }
76
77
78  int main()
79  {
80      int opcion;
81      TestOrdenacion test;
82
83      do {
84          opcion = Menu();
85          switch (opcion){
86          case 1:
87              test.comprobarMetodos();
88              break;
89          case 2:
90              test.estudiarMetodo(subMenuCasoMedio(test));
91              break;
92          case 3:
93              subRutinaComparacion(test);
94              break;
95          }
96      } while (opcion != 0);
97
98      return 0;
99  }
```

En la función main se realizan las llamadas necesarias a la clase TestOrdenacion para el funcionamiento del programa y se implementan los menús como funciones genéricas.

4. Conclusiones

4.1. Análisis individual de algoritmos

4.1.1. Burbuja

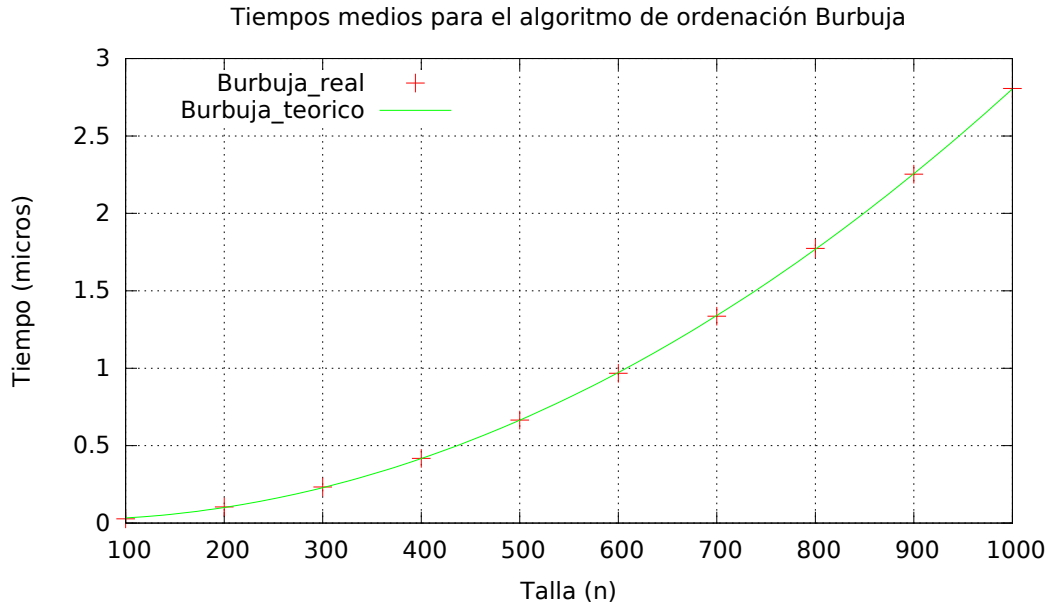

Figura 1: Tiempos medios para el algoritmo burbuja

Podemos ver que las medidas tomadas se ajustan perfectamente a la curva teórica, confirmando que el orden del algoritmo es:

$$O(n^2) \tag{8}$$

4.1.2. Inserción

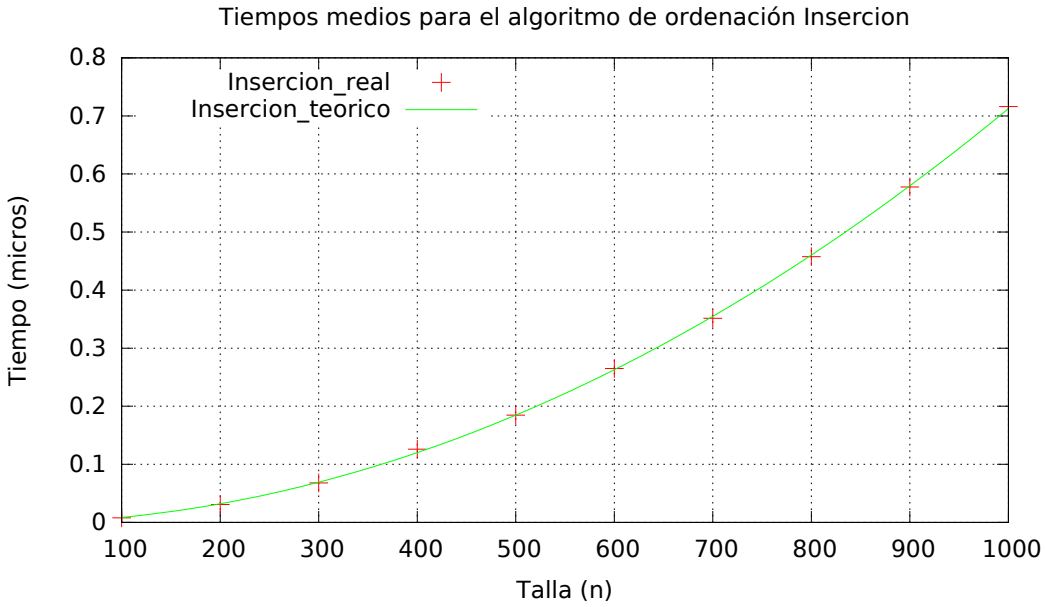

Figura 2: Tiempos medios para el algoritmo inserción

Podemos ver que las medidas tomadas se ajustan perfectamente a la curva teórica, confirmando que el orden del algoritmo es:

$$O(n^2) \tag{9}$$

4.1.3. Shell

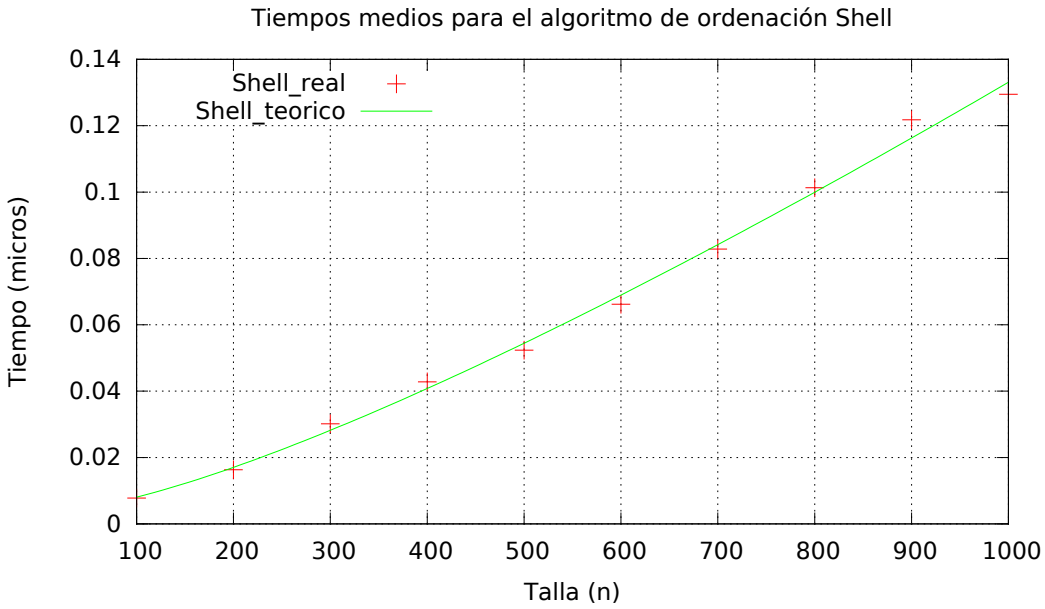

Figura 3: Tiempos medios para el algoritmo Shell

Podemos ver que las medidas se ajustan perfectamente a la teoría, confirmando un orden de:

$$T(n) \in O(n \log(n)) \tag{10}$$

4.1.4. MergeSort

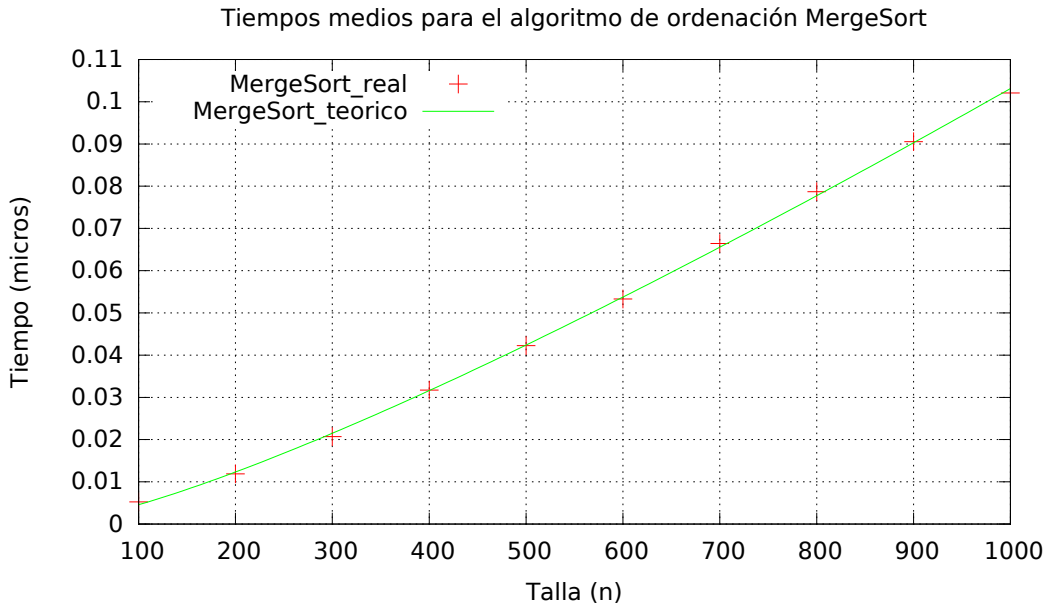

Figura 4: Tiempos medios para el algoritmo MergeSort

Una vez más, podemos ver que los datos se ajustan perfectamente a la teoría, confirmando un orden de:

$$T(n) \in O(n \log(n)) \tag{11}$$

4.1.5. QuickSort

Al igual que anteriormente, podemos ver que las medidas se ajustan perfectamente a la teoría, confirmando un orden de:

$$T(n) \in O(n \log(n)) \tag{12}$$

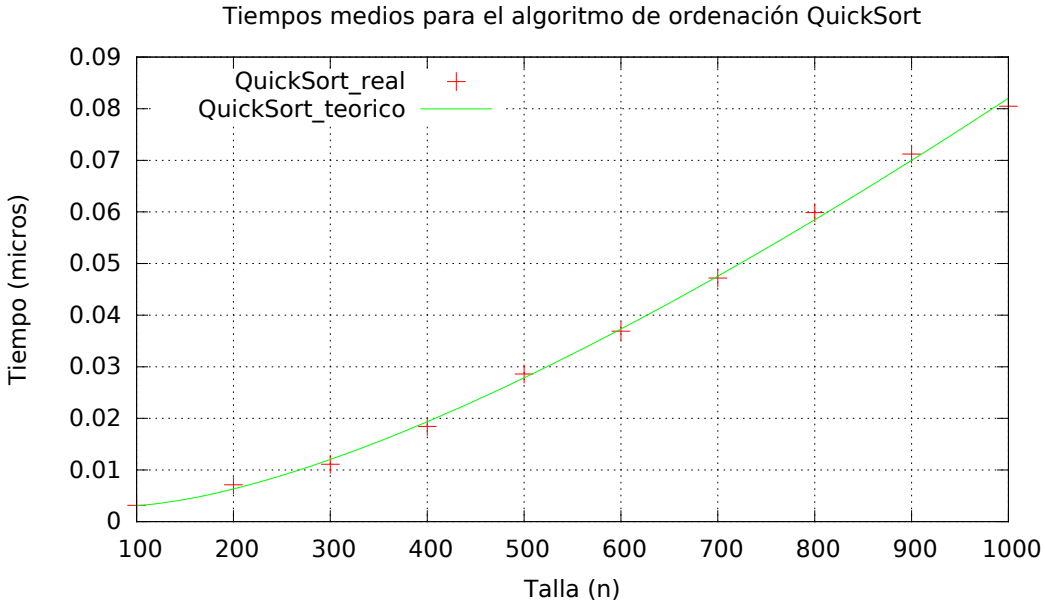

Figura 5: Tiempos medios para el algoritmo QuickSort

4.2. Comparación entre algoritmos

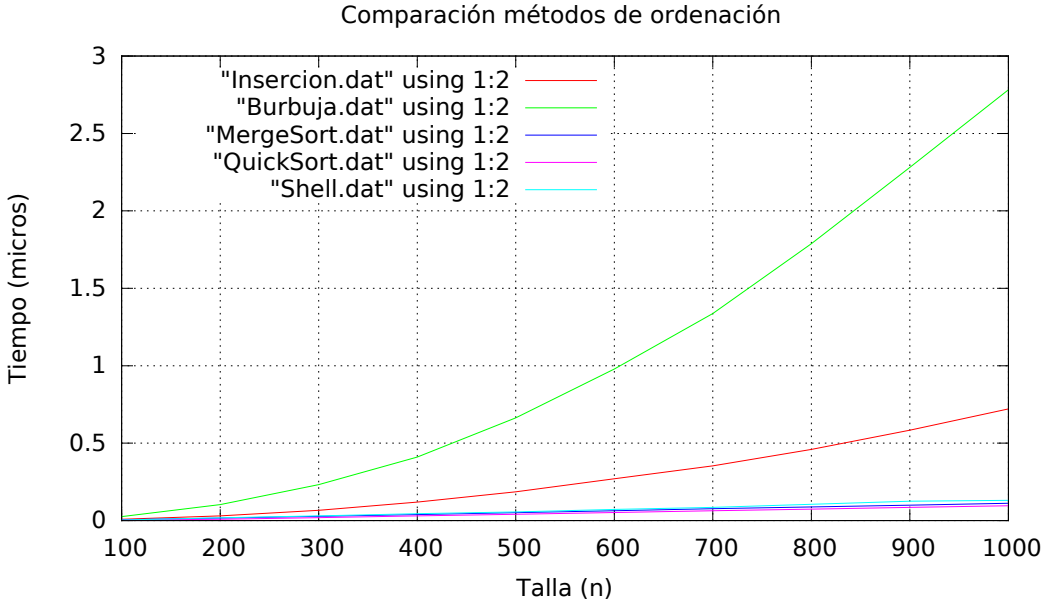

Figura 6: Comparación de los cinco algoritmos estudiados

Tal como podemos ver en la gráfica, el algoritmo de ordenación por burbuja es significativamente más lento que el resto, seguido por el algoritmo de ordenación por inserción. Los tres algoritmos restantes (al observarlos con esta escala) tienen una eficiencia muy parecida, por lo que vamos a compararlos en una gráfica en la que aparezcan unicamente Shell, MergeSort y QuickSort.

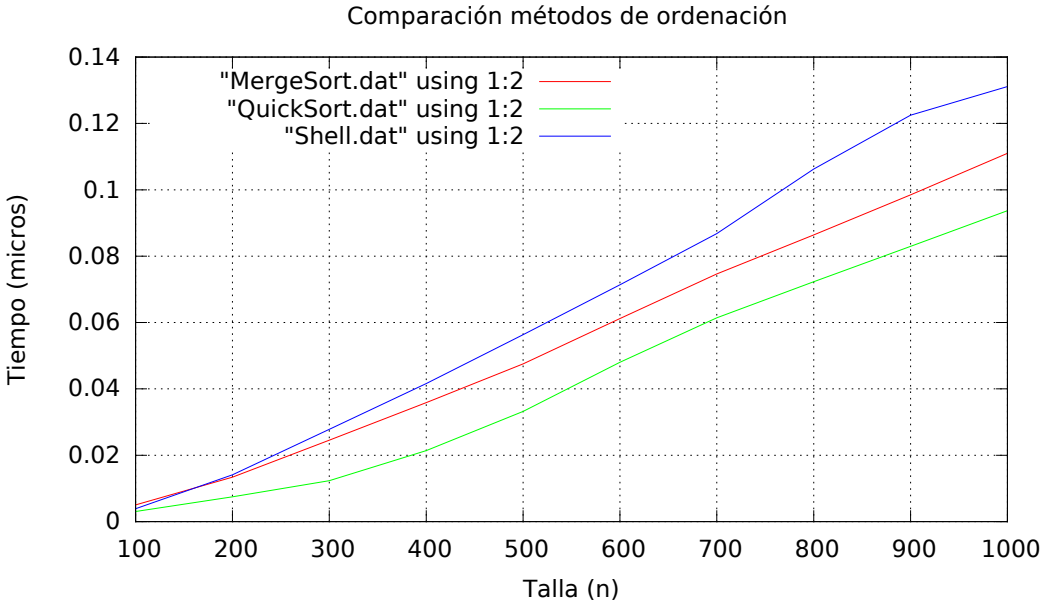

Figura 7: Comparación Shell, Merge y Quick

Como podemos ver, la velocidad de los tres algoritmos es muy similar, pero hay un orden claro entre ellos, QuickSort es el más rápido, sequido por MergeSort y por último Shell.

4.3. Análisis conjunto de los resultados

En resumen, todos los algoritmos se ajustan individualmente de manera clara a los cálculos teóricos. Con respecto a la comparación, los dos algoritmos de orden cuadrático (inserción y burbuja) se muestran claramente más lentos que el resto tal y como se esperaba.
En cuanto a los algoritmos Shell, QuicSort y MergeSort:

- QuickSort es el más rápido en general pero en su caso peor puede llegar a ser cuadrático por lo que habría que valorar si ese caso puede llegar a darse con frecuencia en el sistema que queramos implementar.

- MergeSort es algo más lento (o mejor dicho, ligeramente menos rápido), y es más estable en el sentido de que su caso peor es del mismo orden que el medio. Sin embargo requiere memoria adicional ya que utiliza un vector auxiliar y no unicamente el que queremos ordenar.

-Shell es el más lento de los tres (aunque enormemente más rápido que burbuja o inserción), no requiere memoria adicional y no es recursivo, por lo que es el ideal para entornos con memoria limitada (tanto QuickSort como MergeSort son recursivos por lo que podrian ser un problema en un entorno de este tipo). Por otro lado al depender su eficiencia de la secuencia de incrementos utilizada a la hora de implementarlo puede ser útil en entornos en los cuales tengamos mucha información sobre como serán los vectores que tendrá que ordenar, pudiendo así implementar la secuencia más adecuada y ganando optimización (aunque esto es algo bastante complejo).

5. Valoraciones Personales

En mi opinión la práctica permite ver de manera clara la importancia de la elección de algoritmos adecuados a la hora de realizar programas de cara a la eficiencia temporal de los mismos. Permite comprobar de manera empírica los cálculos matemáticos sobre la eficiencia teórica de los algoritmos que

unicamente vistos sobre papel pueden ser algo confusos.

Por otro lado, creo que la práctica aporta poco comparandola con las prácticas de la asignatura de algorítmica de primero (Sobre todo teniendo en cuenta el tiempo que ocupa y el poco que tenemos en una asignatura cuatrimestral), varios algoritmos (como burbuja, inserción o QuickSort) se estudian en una práctica muy similar.