

ESTRUCTURA DE DATOS

Práctica 1. Eficiencia de algoritmos

Doble Grado de Informática y Matemáticas

José María Martín Luque
Antonio Coín Castro

12 de octubre de 2016

Ejercicio 1.

En este ejercicio, comprobaremos tanto la eficiencia teórica como la eficiencia empírica del algoritmo de ordenación *burbuja*:

```
1 void ordenar_burbuja(int *v, int n)
2 {
3     for (int i=0; i<n-1; i++)
4         for (int j=0; j<n-i-1; j++)
5             if (v[j]>v[j+1]) {
6                 int aux = v[j];
7                 v[j] = v[j+1];
8                 v[j+1] = aux;
9             }
10 }
```

Comencemos analizando la eficiencia teórica del algoritmo, en el caso peor. Veamos primero el coste en operaciones elementales (OE) de cada línea:

Línea 3. Hay 5 OE: una asignación, una resta, una comparación y un incremento (incremento + asignación).

Línea 4. Hay 6 OE: igual que la línea anterior, pero se realizan dos restas.

Línea 5. Hay 4 OE: dos accesos a un vector, una suma y una comparación.

Línea 6. Hay 2 OE: asignación y acceso al vector.

Línea 7. Hay 4 OE: dos accesos, suma y asignación.

Línea 8. Hay 3 OE: acceso, suma y asignación.

Entonces, la eficiencia del algoritmo es:

$$\begin{aligned} T(n) &= \sum_{i=0}^{n-2} \left(5 + \sum_{j=0}^{n-i-2} 19 \right) = \sum_{i=0}^{n-2} (5 + 19(n-i-1)) = \sum_{i=0}^{n-2} 19n - \sum_{i=0}^{n-2} 19i - \sum_{i=0}^{n-2} 14 = \\ &= 19n(n-1) - 19 \left(\frac{0 + (n-2)}{2} \cdot (n-1) \right) - 14(n-1) = \frac{19}{2}n^2 - \frac{9}{2}n - 5 \end{aligned}$$

Por tanto, afirmamos que $T(n) \in O(n^2)$, y el algoritmo de ordenación burbuja es de orden de eficiencia $O(n^2)$.

Ahora creamos un programa de prueba para analizar la eficiencia empírica, haciendo uso de la biblioteca *ctime* del lenguaje C++:

```

#include <iostream>
#include <ctime>      // Medir tiempos
#include <cstdlib>    // Generación de números pseudoaleatorios

using namespace std;

void ordenar_burbuja(int *v, int n) {
    for (int i=0; i<n-1; i++) {
        for (int j=0; j<n-i-1; j++)
            if (v[j]>v[j+1]) {
                int aux = v[j];
                v[j] = v[j+1];
                v[j+1] = aux;
            }
    }
}

void sintaxis()
{
    cerr << "Sintaxis:" << endl;
    cerr << "  TAM: Tamaño del vector (>0)" << endl;
    cerr << "  VMAX: Valor máximo (>0)" << endl;
    cerr << "Se genera un vector de tamaño TAM con elementos aleatorios en"
          << endl;
    exit(EXIT_FAILURE);
}

int main(int argc, char * argv[])
{
    // Lectura de parámetros
    if (argc!=3)
        sintaxis();
    int tam=atoi(argv[1]);    // Tamaño del vector
    int vmax=atoi(argv[2]);   // Valor máximo
    if (tam<=0 || vmax<=0)
        sintaxis();

    // Generación del vector aleatorio
    int *v=new int[tam];        // Reserva de memoria
    srand(time(0));              // Inicialización del generador de números
                                pseudoaleatorios
    for (int i=1; i<tam; i++)    // Recorrer vector
        v[i] = rand() % vmax;   // Generar aleatorio [0,vmax[

    clock_t tini;               // Anotamos el tiempo de inicio
    tini=clock();

    ordenar_burbuja(v,tam);

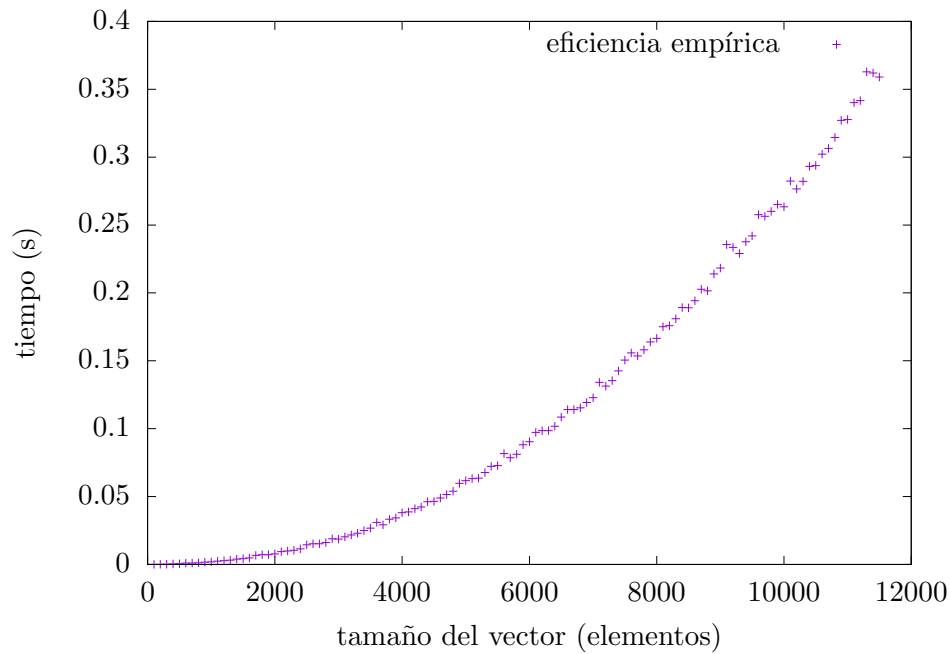
    clock_t tfin;               // Anotamos el tiempo de finalización
    tfin=clock();

    // Mostramos resultados
    cout << tam << "\t" << (tfin-tini)/(double)CLOCKS_PER_SEC << endl;

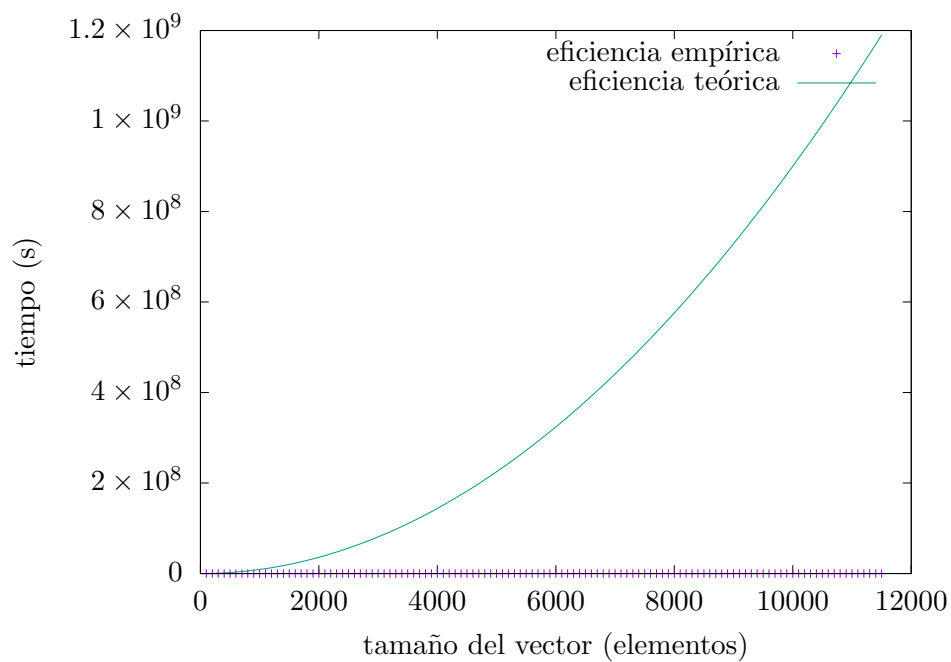
    delete [] v;                // Liberamos memoria dinámica
}

```

Al analizar la eficiencia empírica del algoritmo, obtenemos la siguiente gráfica:



Si representamos superpuestas la función de la eficiencia teórica y la empírica, obtenemos lo siguiente:

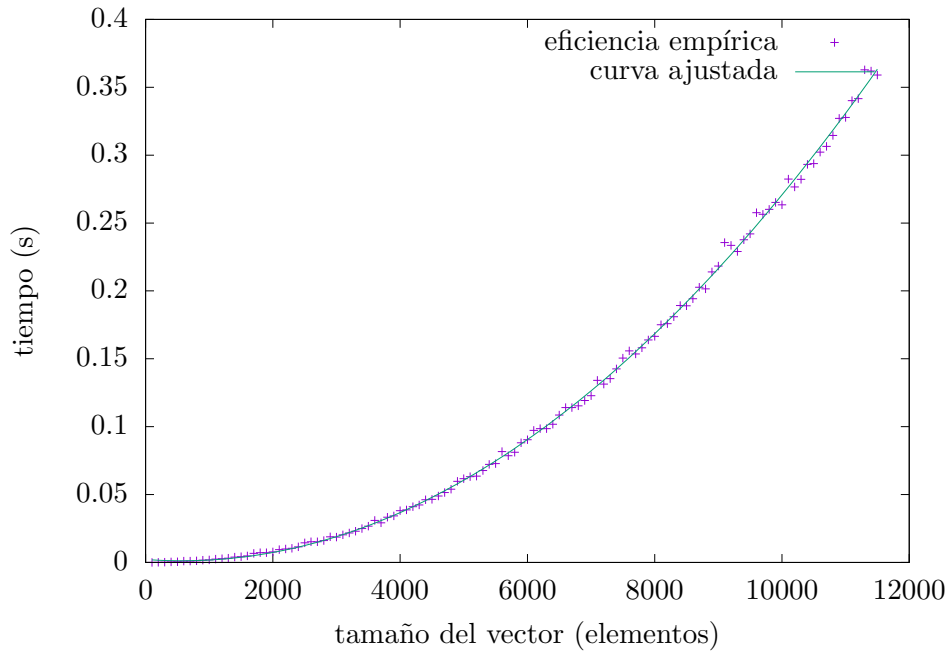


Observamos que

Ejercicio 2.

Veamos que podemos ajustar, mediante *gnuplot*, los puntos obtenidos al analizar la eficiencia teórica en una función de la forma $f(n) = an^2 + bn + c$.

En efecto, obtenemos que la función en cuestión es: $f(n) = n^2 + n$, y la representación gráfica es la siguiente:



Ejercicio 3.

a) En este primer apartado, describiremos el funcionamiento del algoritmo proporcionado en el archivo *ejercicio_desc.cpp*.

Se trata de un algoritmo de **búsqueda binaria**, donde dado un vector v de n elementos enteros, busca el entero x en él. Para ello, se pasan como parámetros el inicio (*inf*) y el final (*sup*) de dicho vector. La función devuelve la posición donde se ha encontrado el elemento, o -1 si no estaba en el vector.

Para el correcto funcionamiento del algoritmo, es imprescindible que el vector esté **ordenado**, pues el procedimiento es el siguiente:

- (i) Se establece la posición $med = (inf + sup)/2$.
- (ii) Se comprueba si el elemento x está en la posición med .
- (iii) Si está, hemos acabado. Si no está, se comprueba si el elemento $v[med]$ es mayor o menor que x .
 - Si $v[med] < x$, actualizamos $inf = med + 1$
 - Si $v[med] > x$, actualizamos $sup = med - 1$
- (iv) Repetimos el proceso, hasta encontrar el elemento, o concluir que no está en el vector.

En resumen, el procedimiento se basa en dividir el vector original por la mitad, y si no está ahí el elemento buscado, nos quedamos únicamente con el sub-vector donde se puede encontrar dicho elemento (recordemos que el vector está ordenado). Repitiendo el proceso, acabaremos encontrando el elemento, o descubriendo que no está en el vector, cuando ya no se puedan hacer más divisiones.

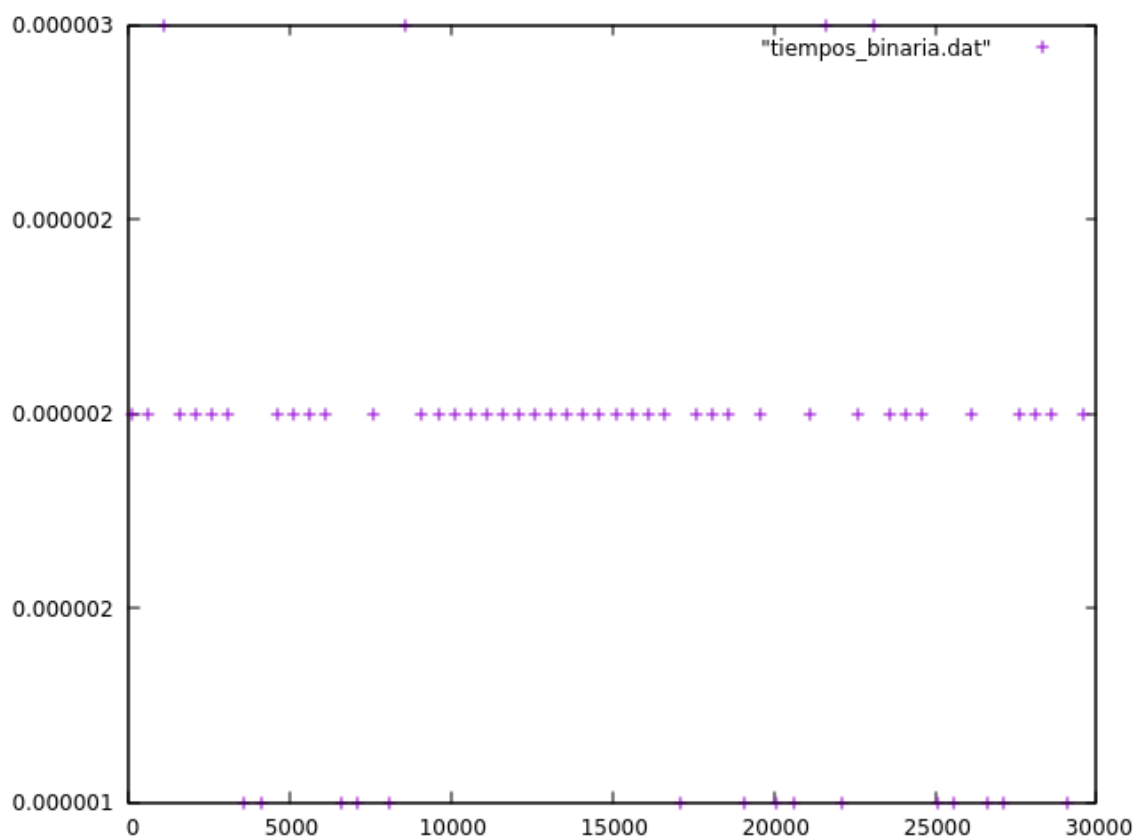
b) Para el cálculo de la eficiencia teórica de la *búsqueda binaria*, nos basamos en el hecho de que se divide sucesivamente en dos un vector de tamaño n . En el caso peor, el proceso continuará

hasta que no se puedan hacer más divisiones del vector.

Por definición, el número máximo de veces que se puede dividir por la mitad un vector de tamaño n es $\log_2(n)$ veces. Para ilustrarlo, consideramos un vector de n elementos. Sabemos que tras m divisiones, el número de elementos restantes será, como mucho, $\lfloor \frac{n}{2^m} \rfloor$, y se detendrá cuando dicho número sea menor que 1. Aplicando logaritmos, se tendría que $\log_2(n) < m$.

Como el resto de operaciones son elementales ($O(1)$), concluimos que la eficiencia del algoritmo es logarítmica, es decir, $T(n) \in O(\log(n))$.

c) Para analizar la eficiencia empírica, repetimos el proceso de los ejercicios anteriores, obteniendo la siguiente nube de puntos:



Vemos que ocurre algo extraño: la eficiencia parece constante. Esto se contradice con nuestro análisis teórico, pero ocurre por dos motivos: el algoritmo se ejecuta tan rápidamente que no es posible medir el tiempo con *ctime*, y la función logaritmo crece lentamente. Para solucionarlo, podemos optar por realizar un número grande de ejecuciones para cada tamaño del vector, y dividir el tiempo total entre el número de ejecuciones.

Vemos la gráfica resultante, con 1000 ejecuciones para cada n , y la curva de regresión correspondiente. Ahora sí es bueno el ajuste de la eficiencia empírica a la teórica, mediante la función $f(n) =$

Ejercicio 4.

a) Para este apartado, construimos un vector ordenado para representar el caso mejor. Usaremos, por ejemplo, el siguiente código, y elegiremos $vmax > tam$:

```

// Generación del vector ordenado
int *v=new int[tam];           // Reserva de memoria
v[0] = 0;
for (int i=1; i<tam; i++)      // Recorrer vector
    v[i] = v[i-1] + 1;        // Generar aleatorio [0,vmax[

```

La gráfica que representa la eficiencia empírica es la siguiente, cuya función ajustada a la eficiencia teórica es: $f(n) =$

b) Para este apartado, construimos un vector ordenado en orden inverso, para representar el caso peor. Usaremos, el siguiente código:

```

// Generación del vector ordenado
int *v=new int[tam];           // Reserva de memoria
v[0] = vmax;
for (int i=1; i<tam; i++)      // Recorrer vector
    v[i] = v[i-1] - 1;        // Generar aleatorio [0,vmax[

```

La gráfica que representa la eficiencia empírica es la siguiente, cuya función ajustada a la eficiencia teórica es: $f(n) =$

Si comparamos ambas funciones ajustadas con la del *Ejercicio 1*, observamos que el algoritmo es más eficiente en el caso mejor, y menos eficiente en el caso peor, tal y como indicaba la teoría.

Ejercicio 5.

Consideramos la siguiente representación alternativa del algoritmo de ordenación **burbuja**:

```

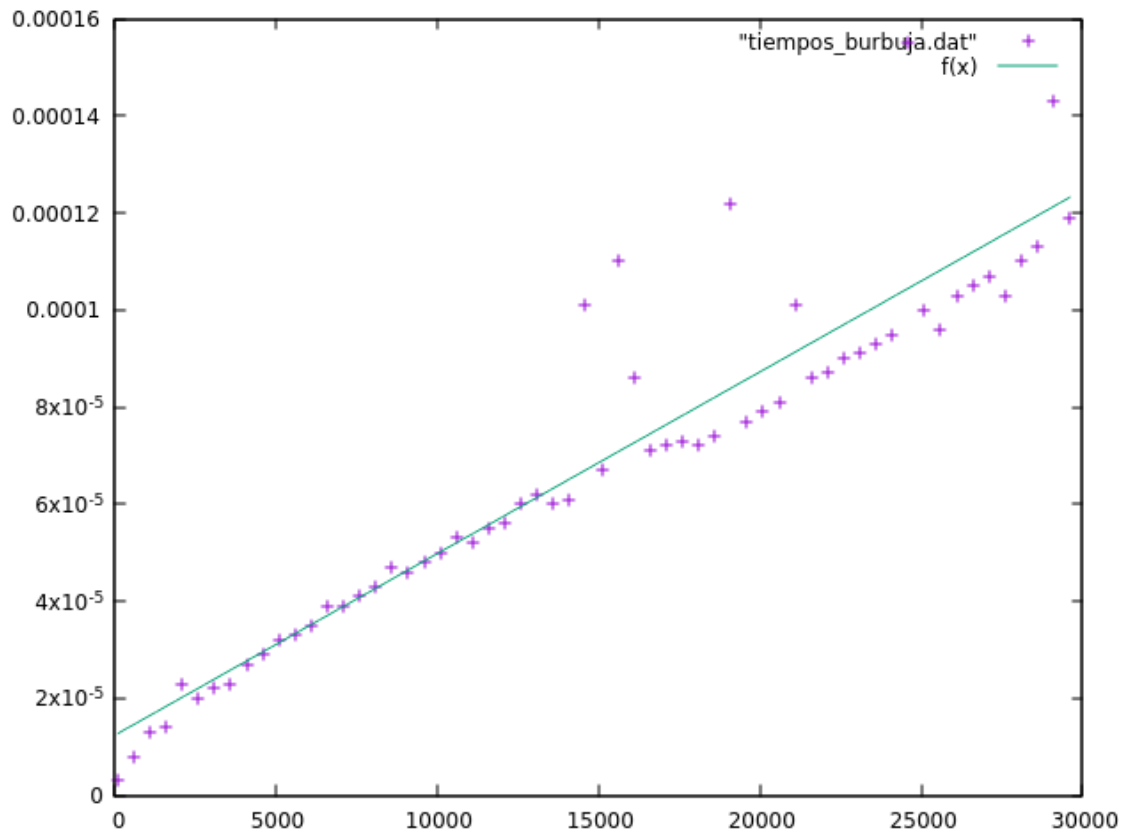
void ordenar(int *v, int n) {
    bool cambio=true;
    for (int i=0; i<n-1 && cambio; i++) {
        cambio=false;
        for (int j=0; j<n-i-1; j++)
            if (v[j]>v[j+1]) {
                cambio=true;
                int aux = v[j];
                v[j] = v[j+1];
                v[j+1] = aux;
            }
    }
}

```

En esta versión, se ha añadido una variable que permite saber si, en una iteración del bucle externo, no se ha producido un cambio en el vector. Si esto ocurre, el vector ya está ordenado, y no hay que continuar.

Si realizamos un estudio teórico de la eficiencia en el mejor caso, es decir, cuando el vector de entrada está ya ordenado, nos damos cuenta rápidamente de que el orden de eficiencia es lineal, es decir, $T(n) \in O(n)$. Esto es así porque tan sólo habría que recorrer una vez $n - 1$ elementos del vector, hasta darse cuenta de que está ordenado.

Analicemos ahora la eficiencia empírica de este algoritmo en el caso mejor. Veamos la gráfica asociada:



Como podemos observar, el orden de eficiencia es efectivamente lineal, tal y como había predicho nuestro estudio teórico.

Ejercicio 6.