Práctica 1: Eficiencia

Antonio Manuel Fernández Cantos

11 de octubre de 2015

Índice

1. Introducción

La práctica consiste en calcular la eficiencia **teórica y empírica** de un código en c++ y **realizar un ajuste de la curva de eficiencia teórica a la empírica.** Se utilizará la biblioteca **ctime** para poder obtener los resultados empíricos. Dentro de la biblioteca ctime tenemos la función **clock()** que devuelve el número de ticks que han transcurrido desde un momento determinado, es esta función la que usaremos para medir la diferencia de tiempo entre el inicio del algoritmo y su finalización.

2. Componentes utilizados

En el cálculo empírico, el algoritmo tardará más o menos en función de:

- **Hardware usado**:
  - CPU
  - RAM

    - HDD
- **Sistema Operativo**
- **Compilador (y sus opciones de compilación)**
- **Bibliotecas**

Todos estos componentes se tienen en cuenta cuando se obtiene el tiempo que tarda nuestro algoritmo en ejecutar todas las sentencias. Dependiendo de la potencia de nuestro ordenador y de las librerias usadas, el algoritmo tardará más o menos. Para la realización del cálculo empírico de los ejercicios, he usado los siguientes componentes:

- **Hardware usado**:
    - Procesador: 8x Intel(R) Core(TM) i7-3630QM CPU@2.40MHz
    - RAM: 6GB
    - CPU clock: 1200 MHz
    - HDD: 750GB
- **Sistema Operativo**: Ubuntu 14.04.3 LTS
- **Compilador**: GCC sin opciones de compilación
- **Bibliotecas**:
    - iostream (E/S)
    - ctime (Para medir el tiempo de ejecución de un algoritmo)
    - cstdlib (Para generar números pseudoaleatorios)

3. Ejercicio 3: Problemas de precisón

3.1. Introducción

En algunos algoritmos las funciones de la biblioteca ctime no sirven para medir cuanto tarda en ejecutarse el algoritmo, ya que el propio algoritmo es muy rápido o tiene pocos datos con los que trabajar y su tiempo de ejecución nos da 0 con las funciones de ctime. En este ejercicios se pide que se ejecute el fichero ejercicio desc, explique que hace el algoritmo, calcular las dos eficiencias y posteriormente se explique y proponga una solución al problema que surja.

3.2. Código

```
1  int operacion(int *v, int n, int x, int inf, int sup) {
2      int med;
3      bool enc=false;
4      while ((inf<sup) && (!enc)) {
5          med = (inf+sup)/2;
6          if (v[med]==x)
7              enc = true;
8          else if (v[med] < x)
9              inf = med+1;
10         else
11             sup = med-1;
12     }
13     if (enc)
14         return med;
15     else
16         return -1;
17 }
```

3.3. Algoritmo

Como podemos observar en el código, se realiza una busqueda binaria. La busqueda binaria consiste:

Recibe cinco variables

- int *v que es puntero a entero y que contiene un vector ordenado.
- int n es un entero que supongo que contiene el tamaño del vector. En el código no se utiliza en ningún momento.
- int x entero que contiene el número que se busca.
- int inf entero que contiene la cota inferior de búsqueda del número.
- int sup entero que contiene la cota superior de búsqueda del número.

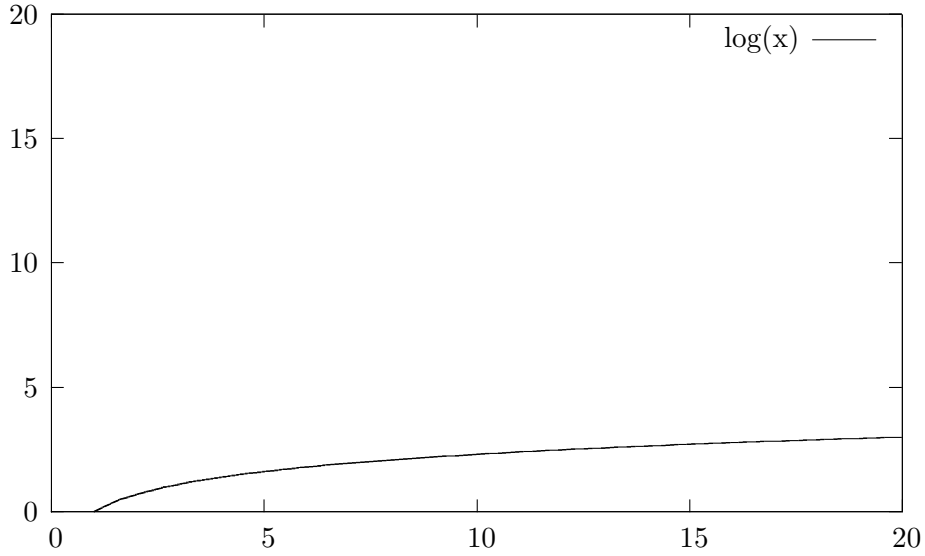
Básicamente el algoritmo consiste en comprobar primero que la cota inferior sea menor que la superior y que aún no se ha encontrado el número. Se calcula el punto medio entre las dos cotas. Posteriormente se comprueba si en la posición de la media de las dos cuotas se encuentra el número buscado. Si se encuentra, se pone enc (encontrado) a true y finaliza el algoritmo devolviendo la posición del número en el vector. Si no se encuentra, se comprueba si el número de la posición media es menor que el número buscado. De ser así, sustituimos el valor de la cota inferior por med+1. En el caso contrario es a la cota superior la que se le sustituye su valor por med-1. Despúes de sustituir el valor de alguna de las dos cotas, se vuelve a realizar los cálculos oportunos. En el caso de que no se encuentre el valor buscado en el vector, el algoritmo finaliza enviando un -1 indicando que no contiene el número buscado.

3.4. Definición búsqueda binaria

Se utiliza cuando el vector en el que queremos determinar la existencia de un elemento está **previamente ordenado**.

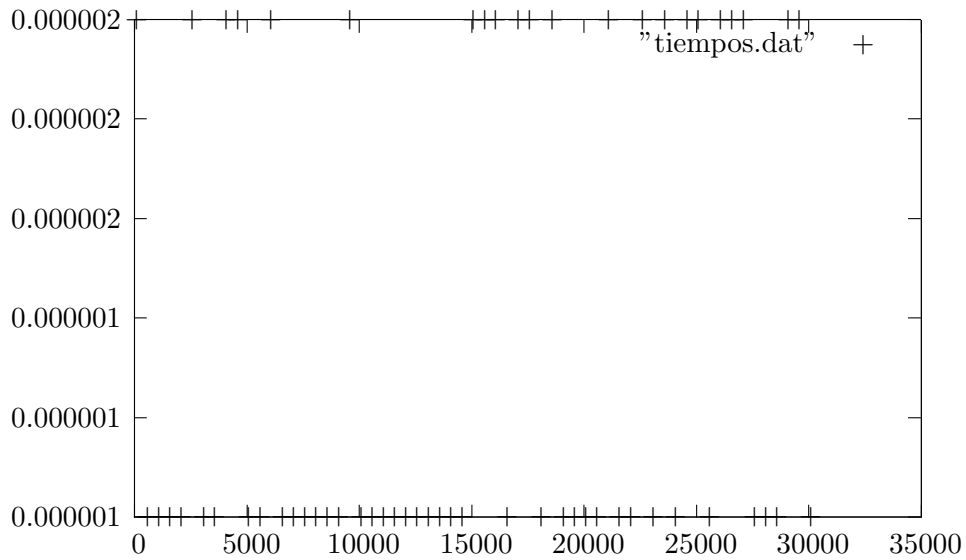
3.5. Eficiencia teórica

El orden de eficiencia en el peor de los casos es OE(log(n)). Viene representado por la siguiente gráfica:



3.6. Eficiencia empírica

El problema que aparece a la hora de calcular la eficiencia empírica es que el algoritmo es tan rápido que no consigue medir bien los tiempos. Este problema lo podemos apreciar en la gráfica siguiente:

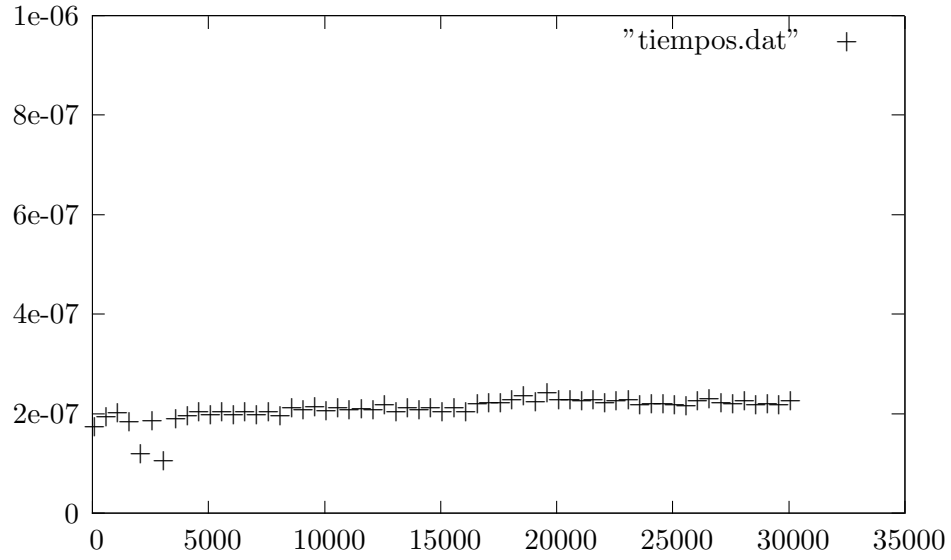


Como podemos comprobar en la gráfica no tiene sentido de crecimiento o decrecimiento conforme va aumentando de tamaño el vector, sino que crece y decrece de forma irregular. En esta sección vamos a solucionar el problema de la eficiencia empírica con dos códigos un poco distintos. Si nos fijamos en el código que nos han pasado para la gráfica, en ningún momento el vector que se pasa a la búsqueda binaria está ordenado. Por este motivo mediremos la eficiencia empírica para la búsqueda binaria sin el vector ordenado y otra con el vector ordenado. En los dos casos, para medir la eficiencia empírica se hará una batería de pruebas, se realizará un millón de ejecuciones para cada tamaño del vector, se obtendrá una suma total y se hará la media entre un millón. Realizando este procedimiento que es como da a entender en el pdf de la práctica obtendremos la eficiencia empírica. Tengo que recalcar que cuando analicemos la eficiencia empírica con el vector ordenado, la ordenación del vector no entrará en el análisis de tiempo a medir.

- En primer lugar vamos a medir la eficiencia empírica del vector sin estar ordenado. La parte del código que ha sido modificada es la siguiente:

```
long double suma=0.0;
long double media=0.0;
for(int j=0; j<1000000;j++){
srand(time(0));
  for (int i=0; i<tam; i++)
  v[i] = rand() %tam;
clock_t tini;
tini=clock();

// Algoritmo a evaluar
operacion(v,tam,tam+1,0,tam-1);

clock_t tfin;
tfin=clock();
suma +=(tfin-tini)/(double)CLOCKS_PER_SEC;
}

media=suma/1000000.0;

// Mostramos resultados
cout << tam << "\t" << media << endl;
```

He creado un bucle for con 1.000.000 de iteraciones para poder calcular el tiempo que tarda en ejecutarse el algoritmo. El tiempo de cada ejecución se guarda en una variable y al finalizar el bucle, se hace la media, obteniendo el resultado final. La gráfica de los tiempos obtenidos es la siguinte:

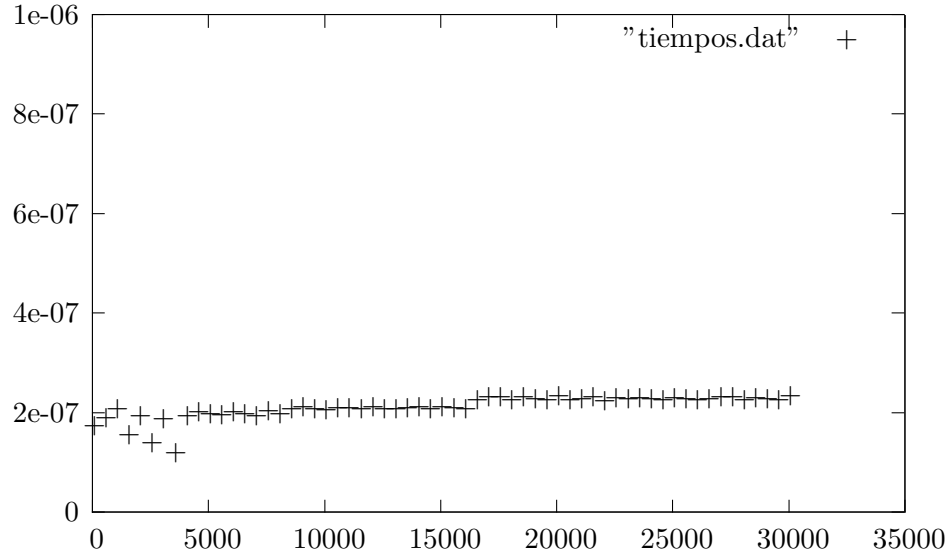


Como podemos observar hemos obtenido unos resultados más lógicos, obteniendo los mismos resultados de tiempo. Los datos son constantes en la gráfica, excepto algún punto, pero esta diferencia es mínima.

- Ahora vamos a modificar el código para obtener la eficiencia empírica del algoritmo de busqueda binaria pero con el vector ordenado. La parte del código modificada es la siguiente:

```
long double suma=0.0;
long double media=0.0;
for (int i=0; i<tam; i++)
  v[i] = i;
for(int j=0; j<1000000;j++){

  clock_t tini;
  tini=clock();


  operacion(v,tam,tam+1,0,tam-1);

  clock_t tfin;
  tfin=clock();
  suma +=(tfin-tini)/(double)CLOCKS_PER_SEC;
  }

media=suma/1000000.0;

// Mostramos resultados
cout << tam << "\t" << media << endl;
```

Como podemos observar la creación del vector con los datos se realiza fuera, además de tener el vector ordenado. Esto no influye en la rapidez del algoritmo, ya que lo que nos interesa es cuanto tarda el algoritmo en recorrer el vector en el peor de los casos. Ahora vamos a observar la eficiencia empírica en la gráfica siguiente:



Seguimos obteniendo el mismo intervalo de tiempo, y la linea de puntos

es constante.

3.7. Regresión para ajustar la curva teórica a la empírica

Es de tipo a*log(x) + b
a = 1.54749*e^{-}08 b = 6.7656*e^{-}08
El gráfico es este: