Diseño de un experimento

1er trabajo en grupo

Programación de Sistemas

Óscar García Lorenz 15166

Mario Musicò Cortés 15308

Alejandro Redondo Ayala 15361

Grupo V07

Índice

1. Introducción

En este apartado se muestran las ideas generales del experimento llevado acabo así como el reparto de tareas asignado a cada miembro del equipo.

1.1 . Ideas generales del trabajo

A la hora de comparar algoritmos de ordenación pudiera parecer adecuado tomar la medida del tiempo que tarda cada uno en ordenar un mismo vector de datos, sin embargo, debido a que el Sistema Operativo en el momento de ejecución del algoritmo puede estar atendiendo más procesos, no resulta fiable medir el tiempo absoluto que tarda cada uno. Además, a esta medida no sólo afectan los procesos que pudiera estar atendiendo el S.O, sino que dependerá también de la tecnología empleada en el computador en el que se realice el experimento, de forma que no se podrían usar los resultados si se ejecutan en distintos ordenadores. Por otra parte, a la velocidad del algoritmo afectan el número de comparaciones e intercambios llevados a cabo durante el mismo, por lo tanto decidimos contabilizar estos para realizar el experimento. Sin embargo, estas medidas puede que no sean del todo aclaratorias para determinar cuál es más rápido, por eso decidimos medir el tiempo relativo que tarda cada algoritmo. No obstante, hay que aclarar que esta medida aunque más fiable que el tiempo absoluto, no deja de tener cierta imprecisión por los mismos motivos ya comentados anteriormente, sin embargo, puede dar una idea aproximada del tiempo de ejecución de cada algoritmo.

En resumen, para el experimento tomaremos un vector de enteros y se ordenaran por los distintos algoritmos, tomando nota del número de comparaciones, de intercambios, y el tiempo relativo que tarda cada uno de ellos.

1.2 . Reparto de tareas

Una vez decidida la forma de abordar el trabajo procedimos al reparto de tareas:

Óscar: Diseño de experimentos, creación del Benchmark y pruebas.

Mario: creación de datos e interfaz de usuario.

Alejandro: elaboración de algoritmos y documentación.

2. Experimento y código

En este apartado explicaremos como se ha desarrollado el código y las librerías del trabajo, así como su funcionalidad. Consideramos experimento a cada vez que se ordena un vector con alguno de los algoritmos, para un tipo de dato y un tamaño de vector. Para entender bien todo el desarrollo comenzamos explicando cómo obtenemos los resultados y la notación empleada.

2.1. Obtención de resultados

El resultado de nuestros experimentos los mostramos en notación de Landau también llamada cota asintótica, mucho más útil que el número de comparaciones, movimientos y tiempo transcurrido para cada algoritmo, orden inicial de los datos y número de datos. Es más fácil su lectura también por quitar resultados al condensar los experimentos con número distinto de datos a ordenar.

El proceso llevado a cabo es el siguiente, en primer lugar fijamos un algoritmo y un tipo de orden inicial en los datos.

Tras computar movimientos, comparaciones y tiempo para distintos experimentos cuya única diferencia es el número de datos, intentaremos ajustar cada uno a un orden lineal $O(n)$, cuadrático $O(n^2)$ o lineal-logarítmico $O(n\ log\ n)$.

Para cada tipo de orden habrá una transformación, para $O(n)$ es la unidad, para $O(n^2)$ simplemente hay que dividir cada dato por el número de elementos y en $O(n\ log\ n)$ por el logaritmo del número de elementos.

Hecho esto efectuaremos una Regresión Lineal Simple.

Datos iniciales:

$\boldsymbol{n_i}$ (Número de elementos).

$\boldsymbol{y_i}$ (Número de comparaciones/movimientos/tiempo una vez transformados).

La recta de regresión es la recta que minimiza el error con respecto al conjunto de puntos anterior siendo $\boldsymbol{n}$ las ordenadas e $\boldsymbol{y}$ las abcisas. Esta recta se puede calcular de la siguiente forma:

$\boldsymbol{y} = \beta_1 \boldsymbol{n} + \beta_0$; donde $\beta_1 = \frac{\sum(\boldsymbol{n_i} - \bar{n})(\boldsymbol{y_i} - \bar{y})}{\sum(\boldsymbol{n_i} - \bar{n})^2}$ y $\beta_0 = \bar{y} - \beta_1 \bar{n}$

Siendo $\bar{n}$ n media e $\bar{y}$ y media.

Tras hacer esto se calcula el residuo de cada orden, $r = \sqrt{\sum(\beta_1 \boldsymbol{n_i} + \beta_0 - \boldsymbol{y_i})^2}$

Nos quedaremos con el residuo menor de todos siempre que $\beta_1 \neq 0$.

Finalmente en notación de Landau, el coeficiente será el β_1 de la regresión elegida anteriormente:

$$\boldsymbol{\beta_1 O(orden)}$$

2.2. Benchmark

Esta sección está compuesta por la librería "Benchmark.h" y el archivo "Benchmark.c". Como idea general se encarga de definir el tipo de datos básicos que se van a usar en el experimento, las funciones básicas y la toma de medidas. La estructura básica que se va a emplear la hemos llamado "Experiment" y en ella se guarda el número de comparaciones, el número de movimientos, el tiempo al inicio del experimento, el tiempo al final del experimento, y el número de elementos.

La función `New Experiment`, genera un nuevo experimento poniendo los contadores a cero.

La función `swap` intercambia dos valores, sumando 3 al contador de movimientos, la función `compare` compara dos valores y actualiza el valor del contador de número de comparaciones.

La función `startCount` inicia la cuenta del tiempo, por otra parte la función `endCount` finaliza la cuenta de tiempo. De esta forma cuando finaliza la ejecución de un algoritmo, tenemos guardados en la estructura el número de comparaciones, el número de intercambios, el tiempo al inicio y el tiempo al final.

La función `millis` y la función `nanos` devuelven respectivamente los milisegundos y nanosegundos que transcurre entre la llamada a `startCount` y la llamada a `endCount`, de forma que ya tenemos el tiempo relativo que ha tardado el algoritmo. Hasta ahora han quedado definidas las funciones y variables que interactúan con los algoritmos.

Para el cálculo de los costes computacionales además se define lo siguiente. Un nuevo tipo "trans_ptr" que es un puntero a función transformación (lineal, cuadrática o logarítmico lineal) del tipo float. Además se define la estructura "Cost", que almacena el tipo de coste computacional y los coeficientes del coste. Y el puntero "algorithm_ptr" a las funciones de los algoritmos de ordenación. Las funciones `unity`, `lin`, `quad` y `linlog` transforman los datos, teniendo como parámetros n e y (los mismos del apartado 2.1). La función `regrex` calcula la recta de regresión, para ello recibe como parámetros el número de elementos, el número de comparaciones/movimientos/tiempo según lo que se desee obtener y el tamaño de estos dos vectores anteriores. Devuelve por referencia, β_0 y β_1 parámetros de la recta de regresión, además la función devuelve el valor del residuo r.

La función pública `CostIdentification` es la encargada de calcular el coste computacional de los distintos algoritmos, devolviendo por referencia y como cadena de caracteres el coste computacional de movimientos, de comparaciones, de tiempo relativo, y el coste de memoria. Para ello hace uso de dos funciones `identify` y `costTostring`

La función `identify` identifica la transformación más adecuada, que será aquella que tenga menor residuo. Una vez determinada cual es la transformación óptima devuelve por referencia la transformación a realizar y el coeficiente (el de la notación de Landau) quedando almacenado en la estructura "Cost".

La función `cosToString` se encarga de transformar la estructura "Cost" en un Char para que pueda ser visualizado por pantalla.

La función `calculateTable` nos da una tabla con los algoritmos utilizados y sus distintos costes computacionales. Esta tabla es liberada por la función `freeTable`.

Como se puede observar en estos dos archivos se encuentra la base del experimento, pues sus tareas son, ejecutar los distintos experimentos (cada uno de ellos es la ordenación por uno de los algoritmos), almacenando las variables a medir (comparaciones, intercambios y tiempo), calcular los costes computacionales y expresarlos en notación de Landau.

2.3. Algoritmos

Esta sección está compuesta por los archivos "Algorithm.c" y "Algorithm.h". En la librería quedan definidas las funciones de los distintos algoritmos de ordenación utilizados `bubble, insertion, selection, heapsort, shell y quicksort.`

Todas las funciones tiene la misma arquitectura para pasar a la función la tabla a ordenar, el número de elementos a ordenar y un puntero a la estructura "Experiment".

Es necesario incluir la librería "Benchmark.h" ya que ahí está definida la estructura mencionada, y porque se hace uso de las funciones `startCount, endCount, swap, y compare.`

En la ejecución de los algoritmos el acceso a la estructura para actualizar los contadores se hace a través de las funciones `swap` y `compare` o bien directamente (i.e: `experiment -> movements++)`.

2.4. Generación de datos

Como se comentará en el siguiente apartado nuestro experimento da la opción al usuario de elegir entre que el programa genere los datos a ordenar o bien que sea el usuario quien proporcione los datos.

En la generación de datos están incluidos los archivos "Datacreator.c" y "Datacreator.h". La generación de datos por el programa es aleatoria. Su funcionamiento es el siguiente, crea un vector de datos ya ordenados, y le da la opción al programa de mandarle el vector tal cual, en orden inverso, en orden aleatorio, o bien en orden aleatorio y con datos duplicados.

A la función `datacreator` se le indica donde debe guardar los datos, el número de datos que debe crear, la disposición de estos (en orden creciente, decreciente…) y como debe ser el intervalo entre número y número una vez ya ordenados, es decir si queremos que sean consecutivos o bien que el intervalo sea aleatorio. Esta función se apoya en las siguientes: `startVector, inverVector, randomize y duplicate.`

También hay definido en este archivo una función `swap` que no tiene relación con la función del mismo nombre de la librería "Benchmark.h", pues aunque también intercambia valores no modifica el valor de los contadores.

2.5. Interfaz del experimento

La interfaz con la que interactúa el usuario del programa está definida en los archivos “TUI.c” y “TUI.h”.

Se le da opción al usuario de elegir entre tres modos de operación:

Modo automático: que compara todos los algoritmos con varios tipos de datos (ya ordenado, orden aleatorio o con duplicados).

Comparar la velocidad de un solo algoritmo con diferentes tipos de datos.

Comparar los distintos algoritmos dando un tipo de dato.

2.6. Funcionamiento del programa

El usuario que ejecute el programa podrá elegir entre dos opciones ejecutar el modo experimento o bien ordenar una serie de datos facilitados por el usuario (Ver anexo).

a) Modo automático: el programa realiza lo siguiente. Ejecuta cada algoritmo de ordenación (i=6) a cada tipo de dato (j=4 ordenado, en orden inverso, aleatorio y con elementos repetidos) y para varios tamaños de array. Después realiza la regresión entre los resultados individuales cada experimento y muestra por pantalla los resultados de la regresión en notación de Landau.
b) Modo un solo algoritmo: El usuario elige el algoritmo (i=1) y se evalúa para cada tipo de dato (j=4) y para varios tipos de array.
c) Modo un solo tipo de dato: El usuario elige el tipo de dato a analizar (j=1) y se evalúa en cada algoritmo (i=6) para distintos tamaños de array.

3. Pruebas, problemas y soluciones.

En esta sección se explican las pruebas realizadas, así como los problemas o dificultades que hemos encontrado en la elaboración del código y las soluciones que se han implementado.

3.1. Pruebas

A la hora de definir el experimento es importante definir las condiciones en las que se va a realizar como pueden ser número de datos, el orden previo de los datos, algoritmo a utilizar… Dado que el programa realiza una regresión de los resultados no es tan crítico la elección del número datos y muestra al usuario las posibles combinaciones entre algoritmos y orden previo

```
//*****************************************************************************
// Declaración e inicializacion de variables de tipos

// Nombres de los algoritmos a ejecutar
algorithm_ptr algorithmTypes[] = {bubble,insertion,selection,shell,heapsort,quicksort};
// Nombres de los algoritmos a visualizar
char algorithmNames[][NAMESIZE] = {"Burbuja  ", "Insercion", "Seleccion", "Shell    ", "Monticulo", "Quicksort"};
// Número de algoritmos                    (i)
int n_algorithms = sizeof(algorithmNames) / sizeof(*algorithmNames);

// Nombres de los tipos de datos a utilizar
dataType dataTypes[] = {INCREASING, DECREASING, RANDOM, REPEATED};
// Nombres de los tipos de datos a visualizar
char dataNames[][NAMESIZE] = {"Creciente", "Decreciente", "Aleatorio", "Repetidos"};
// Número de tipos de datos                (j)
int n_data = sizeof(dataNames) / sizeof(*dataNames);

// Nombres de los costes
char costNames[][NAMESIZE] = {"Comparaciones", "Movimientos  ", "Tiempo (ms)  "};
// Número de costes                        (k)

// Cantidades de datos con las que se iteran
size_t dataSizes[] = {10, 100, 1000, 2000, 3000, 4000};
// Número de iteraciones por experimento    (l)
int iterations = sizeof(dataSizes) / sizeof(*dataSizes);

//*****************************************************************************
```

de los datos. En la siguiente imagen se muestra un extracto del código en el que se pueden apreciar las condiciones elegidas para el experimento.

Tras varias pruebas de ejecución del experimento hemos llegado a la conclusión que los resultados obtenidos son coherentes con los resultados teóricos vistos en clase. En la siguiente imagen se muestra un ejemplo de lo que el programa muestra por pantalla en un experimento.

```
Que desea hacer?
a) Ejecutar experimento de obtencion de la complejidad de diferentes algoritmos de ordenacion de datos
b) Ordenar una serie de datos introducidos por el usuario, en forma de fichero o por teclado
Introducza su opcion: a

************************************************************
Ha entrado en modo EXPERIMENTO

Para cada caso, se realizara una regresion para obtener el residuo frente al modelo teorico,
        probando con las siguientes cantidades de datos: 10, 100, 1000, 2000, 3000, 4000,

Como desea realizar el experimento?
a) En modo automatico, comparar todos los algoritmos con todos los tipos de datos
b) Comparar la velocidad un algoritmo para diferentes tipos de datos
c) Comparar diferentes algoritmos dado un tipo de dato
Introducza su opcion: a


Algoritmos      Tipo de coste computacional:
                Comparaciones           Movimientos             Tiempo (ms)
** Burbuja   **
Creciente:      0.00 O(1)               0.50 O(n^2)             2.47 O(n^2)
Decreciente:    1.50 O(n^2)             0.50 O(n^2)             5.45 O(n^2)
Aleatorio:      0.75 O(n^2)             0.50 O(n^2)             5.19 O(n^2)
Repetidos:      0.74 O(n^2)             0.50 O(n^2)             5.19 O(n^2)

** Insercion **
Creciente:      2.00 O(n)               1.00 O(n)               1.26 O(n log n)
Decreciente:    0.50 O(n^2)             0.50 O(n^2)             2.81 O(n^2)
Aleatorio:      0.25 O(n^2)             0.25 O(n^2)             1.43 O(n^2)
Repetidos:      0.25 O(n^2)             0.25 O(n^2)             1.41 O(n^2)

** Seleccion **
Creciente:      0.00 O(1)               0.50 O(n^2)             2.34 O(n^2)
Decreciente:    1.50 O(n)               0.50 O(n^2)             2.46 O(n^2)
Aleatorio:      3.00 O(n)               0.50 O(n^2)             2.42 O(n^2)
Repetidos:      2.99 O(n)               0.50 O(n^2)             2.34 O(n^2)

** Shell     **
Creciente:      2.41 O(n log n)         1.21 O(n log n)         11.14 O(n log n)
Decreciente:    2.87 O(n log n)         1.57 O(n log n)         15.33 O(n log n)
Aleatorio:      3.68 O(n log n)         2.41 O(n log n)         0.04 O(n^2)
Repetidos:      3.66 O(n log n)         2.39 O(n log n)         0.04 O(n^2)

** Monticulo **
Creciente:      2.02 O(n log n)         2.62 O(n log n)         22.66 O(n log n)
Decreciente:    1.85 O(n log n)         2.42 O(n log n)         20.91 O(n log n)
Aleatorio:      1.94 O(n log n)         2.52 O(n log n)         0.03 O(n^2)
Repetidos:      1.94 O(n log n)         2.52 O(n log n)         0.04 O(n^2)

** Quicksort **
Creciente:      0.20 O(n log n)         1.34 O(n log n)         7.36 O(n log n)
Decreciente:    0.38 O(n log n)         1.34 O(n log n)         7.49 O(n log n)
Aleatorio:      1.09 O(n log n)         2.03 O(n log n)         0.02 O(n^2)
Repetidos:      1.14 O(n log n)         2.05 O(n log n)         0.02 O(n^2)
```

3.2. Medida de tiempo

Las funciones de `time.h` del estándar ISO o ANSI, son insuficientes para poder medir el tiempo con precisión y resolución. Por tanto hemos visto necesario una solución alternativa.

En sistemas GNU/Linux y MacOS que son compatibles con POSIX, una interfaz estándar de sistema operativo. Utilizamos la función `clock_gettime` que devuelve la estructura `timespec`, que tiene una resolución de nanosegundos y una precisión elevada. Restando los campos de tiempo de estas estructuras generadas antes de comenzar el algoritmo de ordenación y tras acabarlo, podemos obtener una medida muy precisa del tiempo transcurrido aún para poca cantidad de datos iniciales. Además, esta función permite contabilizar solo el

tiempo del proceso en cuestión, así minimizando falsas mediciones causadas por el uso de CPU por parte de otros procesos

Desgraciadamente el sistema operativo Windows no es compatible con la solución anterior y por tanto hemos reproducido la función anterior mediante funciones propias de Windows como `QueryPerformanceCounter` de `Windows.h`, también tiene una alta resolución y precisión mayor de 1µs, pero a diferencia del método anterior no nos permite calcular el tiempo exclusivamente del proceso, sino que es del sistema entero. Este problema lo intentaremos minimizar tomando el tiempo como una medida relativa y no absoluta, es decir la relación entre número de elementos y tiempo transcurrido.

4. Anexos

4.1. Modo ordenador

Dado que ya tenemos definidos los algoritmos de ordenación se ofrece al usuario la posibilidad de introducir su propio vector.

En este modo del programa se le da la opción al usuario de introducir los datos por pantalla o bien que los proporcione a través de un documento de texto (.txt). Este archivo de texto puede ser generado:

a) Con el programa `filecreator.c` siguiendo las instrucciones mostradas por pantalla.
b) Manualmente. Atención: la primera línea contiene la cantidad de datos a ordenar seguido de una coma (“,”) y los datos en las sucesivas líneas.

Para esta tarea se usan las funciones definidas en “Dataorganizer.c” y “Dataorganizer.h”.

La función `inputData` es la que pide al usuario los datos por teclado y la función `fileReader` permite que los datos sean aportados a través de un fichero “.txt” que debe estar en el mismo directorio del programa.

Ordena ese vector de datos introducidos con los diferentes algoritmos y visualiza los resultados de coste computacional (comparaciones, intercambios y tiempo) para cada algoritmo. Además, guarda los datos ordenados en un nuevo fichero cuyo nombre tendrá el sufijo “-ordenado”.

4.2. Control de versiones con Git

A la hora de realizar el trabajo uno de los aspectos a tener en cuenta era la forma de compartir los archivos y controlar las versiones de estos. Finalmente se decidió alojar los archivos en una cuenta de Bitbucket y llevar el control de versiones con Git usando como interfaz gráfica el programa Gitkracken que permite el control de las nuevas versiones de cada fichero y permite visualizar los cambios que se realizaron de una versión a otra de forma sencilla. Se puede ver la evolución de versiones del programa en el documento arbol_Git.pdf