PRÁCTICA Nº 1: 1 sesión

(del 11 al 15 de Marzo de 2002)

Costes temporales en algoritmos de Ordenación de Vectores

0. OBJETIVO

En esta práctica se realizará un estudio comparativo de los costes temporales de diferentes algoritmos de ordenación de vectores, mediante la obtención de los costes medios en asignaciones y comparaciones de cada uno de los algoritmos, con distintas tallas del vector.

En concreto se implementarán los algoritmos de inserción, selección, burbuja y quick-sort, tal y como se comentaron en las clases teóricas.

1. ALGORITMOS DE ORDENACIÓN

Algoritmo de Inserción

El algoritmo de inserción se basa en la idea de ir insertando en la parte ordenada del vector, uno a uno, los elementos de la parte desordenada. De esta manera el algoritmo general que nos ordenaría un vector 'vec' de 'n_ele' elementos sería el siguiente:

```
i <- 1
Mientras (i < n_ele) Hacer
   aux <- vec[i]
   k <- (i - 1)
   Mientras (k >= 0) && (aux < vec[k]) Hacer
      vec[k + 1] <- vec[k]
      k <- k - 1
   Fin_mientras
   vec[k + 1] <- aux
   i <- i + 1
Fin_mientras
```

Algoritmo Selección

El algoritmo de selección se basa en la idea de ir seleccionando para cada una de las posiciones del vector el elemento que debería estar en esa posición (en la primera posición el menor, en la segunda el menor de los restantes y así sucesivamente hasta el último que ya está en su lugar)

```
i <- 0
Mientras (i < n – 1) hacer
   pos_min <- i

   j <- i + 1
   Mientras (j < n) hacer
      Si (v[j] < v[pos_min])
         pos_min <- j
      Fin_si
      j <- j + 1
   Fin_mientras

   Si (pos_min != i)
      aux <- v[i]
      v[i] <- v[pos_min]
      v[pos_min] <- aux
   Fin_si
   i <- i + 1
Fin_mientras
```

Algoritmo de intercambio directo o método de la Burbuja

Si consideramos el vector en posición vertical se puede imaginar que los componentes son burbujas con un peso acorde con su valor, así, en cada pasada se produce el ascenso de una burbuja hasta su nivel de peso correspondiente. De esta manera el algoritmo general que nos ordenaría un vector 'vec' de 'n_ele' elementos sería el siguiente:

```
i <- 1
Mientras (i < n_ele) Hacer
   j <- n_ele - 1

   Mientras (j >= i) Hacer

      Si (v[j] < v[j - 1])
         aux <- v[j - 1]
         v[j - 1] <- v[j]
         v[j] <- aux
      Fin_si

      j <- j - 1
   Fin_mientras

   i <- i + 1
Fin_mientras
```

Algoritmo Quick-Sort o de ordenación rápida

En este algoritmo se trata de ir agrupando los elementos en dos subgrupos: Un grupo de elementos 'pequeños' y otro grupo de elementos 'grandes' respecto de una referencia, que llamaremos pivote, y que será un elemento cualquiera del vector (generalmente el elemento central del vector)

```
Algoritmo Q_S
   Entradas: Por referencia Vector i_vec
             Entero izda, dcha;
Inicio
   i <- izda
   j <- dcha
   pivo <- i_vec[(i + j) / 2]

   Hacer
      Mientras (i_vec[i] < pivo] Hacer
         i <- i + 1
      Fin_mientras

      Mientras (i_vec[j] > pivo] Hacer
         j <- j - 1
      Fin_mientras

      Si (i <= j) Entonces
         i_vec[i] <-> i_vec[j]
         i <- i + 1
         j <- j - 1
      Fin_si
   Mientras (i <= j)

   Si (izda < j) Entonces
      Llamar a Q_S con entradas: i_vec, izda, j
   Fin_si

   Si (i < dcha) Entonces
      Llamar a Q_S con entradas: i_vec, i, dcha
   Fin_si
Fin
```

2. TAREAS A REALIZAR

Se trata de implementar en C++ los cuatro algoritmos de ordenación, para utilizarlos como funciones a las que se llamará desde el programa principal. Hay que incluir en los algoritmos **dos parámetros** que pasaremos por referencia para retornar al programa principal el número de asignaciones y comparaciones efectuadas durante la ordenación y también hay que añadir las **instrucciones que contabilizan** estos valores.

Dado que la talla del vector es un factor fundamental en el coste temporal de los algoritmos, cada uno de ellos debe ser ejecutado para la siguiente secuencia de tallas.

> Desde **TALLA_MIN = 10** hasta **TALLA_MAX = 500**, incrementando la talla de **20 en 20** unidades. Declara constantes para estos valores.

Como se trata de obtener costes medios, vamos a ejecutar cada ordenación **tantas veces como talla** tenga el vector, es decir:

> Si la talla=10, ejecutar 10 veces el algoritmo y obtener la media de asignaciones y comparaciones.
> Si la talla=30, ejecutar 30 veces el algoritmo y obtener la media de asignaciones y comparaciones
> y así sucesivamente.

Para cada algoritmo, vamos a obtener entonces un número medio de asignaciones y comparaciones, para cada una de las tallas analizadas.

Talla	Nº medio Asignaciones
10	72
30	513
50	1338
...	

Talla	Nº medio Comparaciones
10	53
30	454
50	1239
...	

Esta información, sin las cabeceras, debe ser almacenada en dos ficheros por cada algoritmo:

> Para Inserción, **ins_asig.dat** e **ins_comp.dat**
> Para Selección, **sel_asig.dat** y **sel_comp.dat**
> Para Burbuja, **bur_asig.dat** y **bur_comp.dat**
> Para Quicksort, **qs_asig.dat** y **qs_comp.dat**

A continuación, se muestra el esquema del programa principal para el análisis del algoritmo de Inserción:

```
// Se suponen ya declaradas las constantes y tipos
// Con cada ejecución analizamos un algoritmo distinto
// En este caso analizamos el algoritmo de insercion

int main()
{
    vector v;
    int n, i;
    int asig = 0, comp = 0;
    int suma_asig, suma_comp;
    ofstream fasig, fcomp;

    fasig.open("ins_asig.dat");
    fcomp.open("ins_comp.dat");

    n = MIN_TALLA;

    while (n <= MAX_TALLA)
    {
        suma_asig = 0;
        suma_comp = 0;

        for (i = 0; i < n; i++)
        {
```

```
            rellena_vector(v, n); //Llena el vector con valores aleatorios
            insercion(v, n, asig, comp);
            suma_asig += asig;
            suma_comp += comp;
        }
        fasig << n << "\t" << suma_asig / n << endl;  //Media de asig.
        fcomp << n << "\t" << suma_comp / n << endl;  //Media de comp.

        n += INC_TALLA;
    }
    fasig.close();
    fcomp.close();

return 0;
}
```

Para comprobar si los costes medios obtenidos son correctos, visualízalos gráficamente utilizando el programa **GNUPLOT**. Esta herramienta nos permite comparar de forma gráfica, los costes medios calculados por nuestro programa junto con el coste medio teórico del algoritmo.

Algunos **comandos básicos** del programa son:

- Para ejecutar el programa utiliza la orden **gnuplot** y para terminar **exit**.
- **plot 'sel_asig.dat', n*(log(n)/log(2))** :Para visualizar el coste medio en asignaciones del alg. de Selección junto con su coste medio teórico.
- **plot 'ins_asig.dat', 'sel_asig.dat', 'qs_asig.dat'** :Para visualizar el coste medio en asignaciones obtenido por los tres algoritmos.
- **plot 'ins_comp.dat', 'sel_comp.dat', 'qs_comp.dat'** :Para visualizar el coste medio en comparaciones obtenido por los tres algoritmos.
- **save 'fichero.plt'** : Guarda la última gráfica realizada en el fichero 'fichero.plt'.
- **load 'fichero.plt' : Lee 'fichero.plt' y muestra la gráfica.**

3. REQUISITOS

Recordar que para poder acceder a la sesión de prácticas correspondiente es necesario entregar un pequeño esquema de lo que se va a realizar en la práctica. Si no se presenta el resumen ANTES de acceder al laboratorio, no se permitirá la entrada a la sesión de prácticas.

4. ENTREGA

La entrega de esta segunda práctica debe incluir **el código** con el programa principal y las funciones (**`ordena##.cpp`**), los ocho ficheros **`".dat"`** con los costes medios obtenidos en asignaciones y comparaciones para cada algoritmo y dos ficheros **`"asig.plt" y "comp.plt"`**, con las gráficas correspondientes al coste medio en asignaciones y comparaciones obtenido por los cuatro algoritmos, respectivamente.

La entrega puede realizarse por e-mail a la dirección del profesor/a encargad@ del grupo o en disquette al iniciar la siguiente sesión de prácticas, pero siempre ANTES de empezar la siguiente sesión de prácticas.

Cualquier práctica entregada fuera de plazo no será admitida para su corrección.