Análisis de Algoritmos 2017/2018

Práctica 1

Rafael Sánchez Sánchez y Sergio Galán Martín, 1201.

Código	Gráficas	Memoria	Total

1. Introducción.

En esta práctica definiremos ciertas rutinas que nos permitirán realizar un estudio exhaustivo de diferentes medios de ordenación. Implementaremos BubbleSort y probaremos las librerías creadas con esta función.

Las rutinas a definir sirven para crear tablas desordenadas y para medir cuanto tarda un método de ordenación en ordenarlas.

2. Objetivos

2.1 Apartado 1

Implementar una función en C la cual reciba los límites superior e inferior y nos devuelva un número aleatorio equiprobable en ese rango.

2.2 Apartado 2

Implementar una función en C la cual genere una permutación aleatoria de los N primeros naturales recibiendo ese cierto N como argumento de entrada.

2.3 Apartado 3

Usando la función del apartado anterior, generar un array de n_perms permutaciones aleatorias de los N primeros naturales, siendo n_perms y N los argumentos de entrada de dicha función.

2.4 Apartado 4

Programar el algoritmo de ordenación Bubble Sort entre cualquier par de índices de la tabla recibida en C.

2.5 Apartado 5

Implementar una función en C que mida el tiempo de reloj y el número de OB realizadas por un algoritmo de ordenación sobre n_perms permutaciones de los naturales de 1 a N, con n_perms y N argumentos de entrada. Usar dicha función para ver dicha información con el algoritmo Bubble Sort implementado en el Apartado 4 y escribirla en un fichero de salida.

3. Herramientas y metodología

Hemos realizado la práctica en Linux (Ubuntu), y los programas que hemos usado han sido Atom (como editor de texto), la propia consola de Ubuntu (para compilar, ejecutar y debuggear) y Valgrind (para comprobar los errores de memoria)

3.1 Apartado 1

Empezamos escribiendo la función típica que a cualquier persona se le ocurre para generar números aleatorios en C, usando rand(), srand() y módulos, pero vimos que había números que aparecían más que otros, por lo que seguimos la sugerencia de ir al libro sugerido, y tomamos una de sus implementaciones, consiguiendo así una mayor aleatoriedad.

Un pequeño detalle que tuvimos que subsanar es el problema de no poder devolver un mensaje de error en caso de que los límites del número aleatorio recibidos en la función estuviesen invertidos, es decir inf > sup, por lo que decidimos hacer una comprobación, y, en el caso en el que sucediese eso, daríamos la vuelta a los límites, ante la imposibilidad de devolver un mensaje de error y cortar la ejecución del programa.

3.2 Apartado 2

Simplemente generamos un array de enteros que contuviera los números de 1 a N, para después aleatorizarlo haciendo N cambios entre los que ocupaban el índice i y los que ocupaban el índice dado por un número generado aleatoriamente entre 1 y N por la función del Apartado 1. Obviamente, también tuvimos que reservar memoria para dicho array, ya que el objetivo es que devuelva dicho array para usarlo más adelante en otras funciones.

3.3 Apartado 3

Generalizando un poco más la función anterior, en la función genera_permutaciones reservamos memoria para un “array de nperms arrays de enteros”, y tras ello procedemos a rellenarlo usando la función del Apartado 2.

3.4 Apartado 4

Siguiendo el pseudocódigo visto en clases de teoría, implementamos Bubble Sort en C, con la variación de que, como la tabla viene dada por referencia, se modifica directamente en la función, y lo que devolvemos es el número de OB realizadas por dicho algoritmo en la tabla recibida entre los índices establecidos.

3.5 Apartado 5

La primera función necesaria para este apartado se basa en rellenar el PTIEMPOS recibido. min_ob, max_ob, medio_ob se rellenan a partir del resultado obtenido en las sucesivas llamadas al algoritmo de ordenación (En este caso Bubble Sort) para ordenar cada una de las permutaciones generadas por genera_permutaciones. El número de permutaciones que se generan viene dado dentro del propio PTIEMPO recibido. El tiempo que tarda en ejecutar el algoritmo de ordenación todas las veces necesarias lo obtenemos mediante la función clock antes y justo cuando termine cada uno de ellos, con lo que al restarlos obtenemos el tiempo de reloj exclusivamente del algoritmo de ordenación. Lo vamos guardando en una variable y añadiéndole el de cada iteración, para al final hacer la media.

La segunda función utiliza tanto la primera función como la tercera de este apartado para generar los PTIEMPOS para n_perms permutaciones de diferentes tamaños, y, tras ello, escribirlos en un fichero de texto. Reservamos memoria para poder almacenar un array de suficientes PTIEMPOS. Dicha memoria la calculamos teniendo en cuenta el límite superior, el inferior y el incremento que se pasan a esta función. Luego, para cada uno de esos PTIEMPOS se llama a la primera función, la cual genera las permutaciones del tamaño dado por el bucle de la segunda función, y las ordena, devolviendo PTIEMPOS relleno con los datos correspondientes. Tras ello, escribe todos esos datos en un fichero externo mediante la tercera función.

La tercera función lo único que hace es abrir un fichero de texto en modo de escritura y va imprimiendo en él todos los datos obtenidos a partir del bucle de llamadas a la primera función.

4. Código fuente

4.1 Apartado 1

```
/*************************************************/
/* Funcion: aleat_num Fecha: 19/10/2017          */
/* Autores: Rafael Sánchez, Sergio Galán         */
/*                                               */
/* Rutina que genera un numero aleatorio         */
/* entre dos numeros dados                       */
/*                                               */
/* Entrada:                                      */
/* int inf: limite inferior                      */
/* int sup: limite superior                      */
/* Salida:                                       */
/* int: numero aleatorio                         */
/*************************************************/
int aleat_num (int inf, int sup){
  int temp, r;
  if (inf > sup){
    temp = inf;
    inf = sup;
    sup = temp;
  }
  sup++;
  r = (int) inf + rand() / (RAND_MAX / (sup - inf) + 1);
  return r;
}
```

4.2 Apartado 2

```
/*************************************************/
/* Funcion: genera_perm Fecha: 19/10/2017        */
/* Autores: Rafael Sánchez, Sergio Galán         */
/*                                               */
```

```
/* Rutina que genera una permutacion                */
/* aleatoria                                        */
/*                                                  */
/* Entrada:                                         */
/* int n: Numero de elementos de la                 */
/* permutacion                                      */
/* Salida:                                          */
/* int *: puntero a un array de enteros             */
/* que contiene a la permutacion                    */
/* o NULL en caso de error                          */
/**************************************************/
int* genera_perm(int N)
{
  int* arr;
  int perm, temp;
  unsigned int i;
  if (N <= 0){
    return NULL;
  }
  arr = calloc(N, sizeof(int));
  if (!arr){
    return NULL;
  }
  for (i = 1; i <= N; i++){
    arr[i-1]=i;
  }
  for (i = 0; i < N; i++){
    perm = aleat_num(0, N-1);
    temp = arr[i];
    arr[i] = arr[perm];
    arr[perm] = temp;
  }
  return arr;
}
```

4.3 Apartado 3

```
/**************************************************/
/* Funcion: genera_permutaciones Fecha: 19/10/2017 */
/* Autores: Rafael Sánchez, Sergio Galán            */
/*                                                  */
/* Funcion que genera n_perms permutaciones         */
/* aleatorias de tamanio elementos                  */
/*                                                  */
/* Entrada:                                         */
/* int n_perms: Numero de permutaciones             */
/* int N: Numero de elementos de cada               */
/* permutacion                                      */
/* Salida:                                          */
/* int**: Array de punteros a enteros               */
```

```
/* que apuntan a cada una de las                    */
/* permutaciones                                    */
/* NULL en caso de error                            */
/****************************************************/

int** genera_permutaciones(int n_perms, int N)
{
  int **arr, i, j;
  arr = calloc(n_perms, sizeof(int*));
  if(!arr){
    return NULL;
  }
  for(i=0; i < n_perms; i++){
    arr[i] = genera_perm(N);
    if (!arr[i]){
      for(j=0; j<i; j++){
        free(arr[j]);
      }
      free(arr);
      return NULL;
    }
  }
  return arr;
}
```

4.4 Apartado 4

```
/****************************************************/
/* Funcion: BubbleSort    Fecha: 19/10/2017         */
/* Autores: Rafael Sánchez, Sergio Galán            */
/*                                                  */
/* Funcion que ordena una tabla por el método de    */
/* la burbuja                                       */
/*                                                  */
/* Entrada:                                         */
/* int* tabla: Array de enteros a ordenar           */
/* int ip: Indice del elemento desde el que ordenar*/
/* int iu: Indice del elemento hasta el que ordenar*/
/* Salida:                                          */
/* int: Numero de veces que se ejecutó la OB        */
/* ERR en caso de error                             */
/****************************************************/
int BubbleSort(int* tabla, int ip, int iu)
{
  int i,j,temp, ret=0;
  for ( i = iu; i >= ip+1; i--){
    for (j = ip; j <= i-1; j++){
      ret++;
```

```
      if (tabla[j] > tabla[j+1]){
        temp = tabla[j];
        tabla[j] = tabla[j+1];
        tabla[j+1]=temp;
      }
    }
  }
  if(ret==0){
    return ERR;
  }
  return ret;
}
```

4.5 Apartado 5

```
/*****************************************************/
/* Funcion: tiempo_medio_ordenacion Fecha: 19/10/2017 */
/* Autores: Rafael Sánchez, Sergio Galán              */
/*                                                    */
/* Funcion que guarda en un tipo de dato TIEMPO, la   */
/* informacion relacionada con un metodo de ordenacion*/
/*                                                    */
/* Entrada:                                           */
/* pfunc_ordena metodo: Puntero a una funcion de      */
/* ordenacion                                         */
/* int n_perms: Numero de permutaciones               */
/* int N: Numero de elementos de cada permutacion     */
/* PTIEMPO ptiempo: Puntero al struct TIEMPO donde se */
/* guardaran los datos                                */
/* Salida:                                            */
/* short: OK si no surgió ningún fallo                */
/* ERR en caso de error                               */
/*****************************************************/
short tiempo_medio_ordenacion(pfunc_ordena metodo,
                              int n_perms,
                              int N,
                              PTIEMPO ptiempo)
{
  int **permutaciones, current_ob, min_ob, max_ob,
all_ob=0, i;
  clock_t t_start, t_end, t_total=0;
  double t_avg, avg_ob;

  if(!ptiempo || N < 0 || n_perms < 0){
    return ERR;
  }

  ptiempo->n_elems = n_perms;
```

```
  ptiempo->N = N;
  permutaciones = genera_permutaciones(n_perms, N);
  if(!permutaciones){
    return ERR;
  }
  for (i = 0; i < n_perms; i++){
    t_start = clock();
    current_ob = metodo(permutaciones[i], 0, N-1);
    t_end = clock();
    t_total += t_end - t_start;
    if (!i){
      min_ob = current_ob;
      max_ob = current_ob;
      }else{
      if (current_ob < min_ob) min_ob = current_ob;
      if (current_ob > max_ob) max_ob = current_ob;
    }
    all_ob += current_ob;
  }
  avg_ob = (double) all_ob/n_perms;
  t_avg = (double) t_total / n_perms;
  t_avg *= 1000.0;
  ptiempo->tiempo = (double) t_avg/(CLOCKS_PER_SEC);
  ptiempo->min_ob = min_ob;
  ptiempo->medio_ob = avg_ob;
  ptiempo->max_ob = max_ob;

  for(i = 0; i<n_perms; i++){
    free(permutaciones[i]);
  }
  free(permutaciones);

  return OK;
}

/***************************************************/
/* Funcion: genera_tiempos_ordenacion Fecha: 19/10/2017    */
/* Autores: Rafael Sánchez, Sergio Galán                   */
/*                                                         */
/* Funcion que genera un array de TIEMPO haciendo          */
/* llamadas sucesivas a la funcion tiempo_medio_ordenacion */
/* con un tamaño de permutacion distinto cada vez. Luego,  */
/* tras sucesivas llamadas a guarda_tabla_tiempos, genera  */
/* un fichero con la informacion del array de TIEMPO.      */
/*                                                         */
/* Entrada:                                                */
/* pfunc_ordena metodo: Puntero a una funcion de           */
/* ordenacion                                              */
```

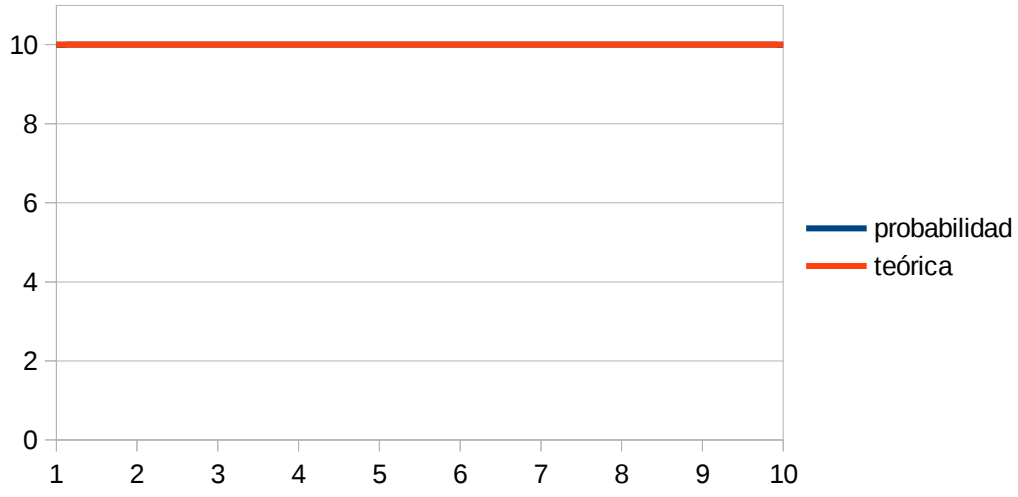
```
/* char* fichero: nombre del fichero al que escribir       */
/* int num_min: tamaño minimo de la permutacion            */
/* int num_max: tamaño maximo de la permutacion            */
/* int incr: Variación del tamaño de permutacion en cada   */
/* llamada a tiempo_medio_ordenacion                       */
/* int n_perms: Numero de permutaciones                    */
/* Salida:                                                 */
/* short: OK si no surgió ningún fallo                     */
/* ERR en caso de error                                    */
/***********************************************************/
short genera_tiempos_ordenacion(pfunc_ordena metodo, char*
fichero,
                                int num_min, int num_max,
                                int incr, int n_perms)
{
  PTIEMPO ptiempo;
  int i, size, j;

  if (!metodo || !fichero || incr <= 0 || num_min < 0 ||
num_max < 0 || num_min > num_max || n_perms < 0){
    return ERR;
  }

  size = ceil((num_max - num_min)/incr);
  ptiempo = calloc(size+1, sizeof(TIEMPO));
  if(!ptiempo){
    return ERR;
  }
  for(i=num_min, j=0; i<=num_max; i+=incr, j++){
    if(ERR == tiempo_medio_ordenacion(metodo, n_perms, i,
&ptiempo[j])){
      free(ptiempo);
      return ERR;
    }
  }
  for(j=0;j<=size;j++){
    if (ERR == guarda_tabla_tiempos(fichero, &ptiempo[j],
j)){
      free(ptiempo);
      return ERR;
    }
  }
  free(ptiempo);
  return OK;
}
```

```
/**************************************************************/
/* Funcion: genera_tiempos_ordenacion Fecha: 19/10/2017       */
/* Autores: Rafael Sánchez, Sergio Galán                      */
/*                                                            */
/* Funcion que abre un fichero y escribe en el la             */
/* informacion de un array de TIEMPO                          */
/*                                                            */
/* Entrada:                                                   */
/* char* fichero: nombre del fichero al que escribir          */
/* int num_min: tamaño minimo de la permutacion               */
/* PTIEMPO tiempo:Array que guarda los tiempos de ejecucion*/
/* int n_tiempos: Numero de elementos del array TIEMPO        */
/* Salida:                                                    */
/* short: OK si no surgió ningún fallo                        */
/* ERR en caso de error                                       */
/**************************************************************/
short guarda_tabla_tiempos(char* fichero, PTIEMPO tiempo,
int n_tiempos)
{
  FILE* fp;
  fp = fopen(fichero, "a");
  if(!fp){
    return ERR;
  }
  fprintf(fp, "%d %f %f %d %d\n",tiempo->N, tiempo->tiempo,
tiempo->medio_ob, tiempo->min_ob, tiempo->max_ob);
  fclose(fp);
  return OK;
}
```

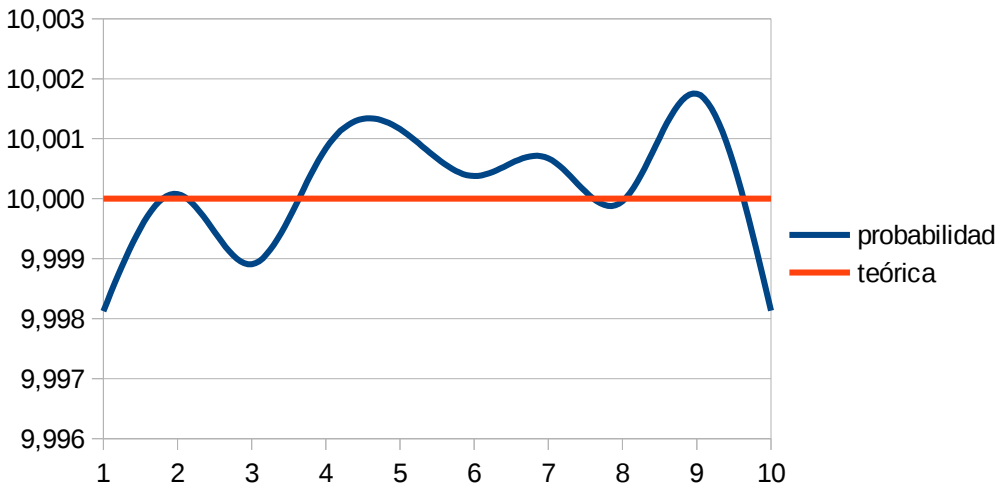
5. Resultados, Gráficas

5.1 Apartado 1

```
./ejercicio1 -limInf 1 -limSup 10 -numN 20
Practica numero 1, apartado 1
Realizada por: Sergio Galán Martín, Rafael Sánchez
Sánchez
Grupo: 1201
7
9
7
7
9
1
2
6
6
7
```



```
9
9
2
10
5
9
8
9
2
6
```



Podemos comprobar las variaciones mínimas que tiene nuestra implementación si mostramos la escala desde 9.996-10,003. Donde vemos que nuestra función es equiprobable con una tolerancia del ±0,002% a lo largo de 10^6 iteraciones.

En la escala absoluta vemos como la probabilidad coincide con la teórica.

5.2 Apartado 2

```
./ejercicio2 -tamanio 15 -numP 10
Practica numero 1, apartado 2
Realizada por: Sergio Galán Martín, Rafael Sánchez
Sánchez
Grupo: 1201
1 9 5 6 3 7 15 10 11 8 14 2 12 13 4
1 2 8 12 10 5 14 3 7 9 6 13 15 11 4
2 13 10 8 14 5 11 1 12 7 9 4 15 6 3
10 1 2 8 4 5 12 15 3 11 13 9 6 7 14
11 6 9 14 10 15 13 12 5 2 4 7 3 1 8
3 6 12 13 9 2 8 5 1 10 11 4 7 15 14
10 6 15 4 9 7 5 2 1 11 13 14 3 8 12
3 4 13 9 12 5 6 2 10 11 7 8 15 1 14
2 7 3 9 8 10 1 12 14 15 13 6 5 11 4
6 11 4 10 5 1 13 14 12 2 3 15 9 8 7
```

5.3 Apartado 3

```
./ejercicio3 -tamanio 15 -numP 10
Practica numero 1, apartado 3
Realizada por: Sergio Galán Martín, Rafael Sánchez
Sánchez
```

```
Grupo: 1201
14  4  7 11 13  3  8  6  5  2 12  9 10 15  1
 9  4 15  5  2  8  7 13 11 12  3 10  6 14  1
 4  5 15 12  7  8  6 14  9  3 10  1 11  2 13
10  8  2  4 11 13 14  5  1  9  7 15 12  6  3
 8 12  2 13  6  9  7 11  5  1 10 15 14  3  4
 9  7  4  5  3 14 15 11  1 10 13  6  2 12  8
12 15  1  2  3  5  4 13  9 10  6  8 14  7 11
 4  5 13  2  7 10  9  1 11  3 15  8 14  6 12
 5 11 14 10  9  4  8  2 12 13 15  7  3  1  6
 7  5 14  4  2  9 10  6  8 11  1 15 13  3 12
```

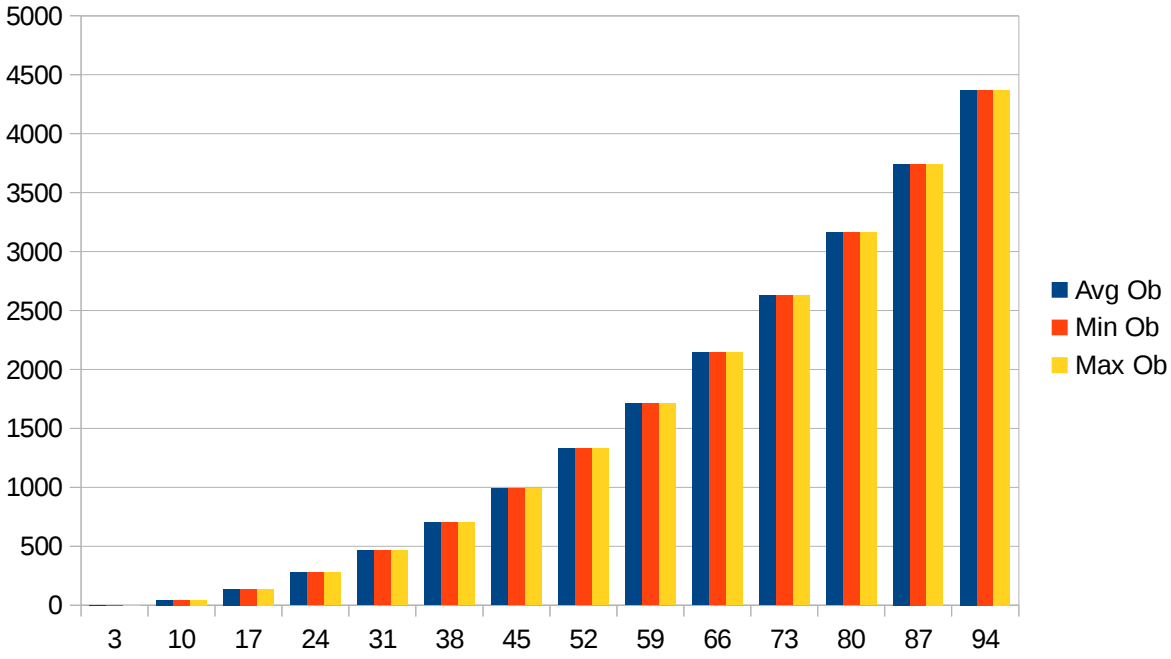
5.4 Apartado 4

```
./ejercicio4 -tamanio 50
Practica numero 1, apartado 4
Realizada por: Sergio Galán Martín, Rafael Sánchez
Sánchez
Grupo: 1201
1    2    3    4    5    6    7    8    9    10   11
12   13   14   15   16   17   18   19   20   21   22
23   24   25   26   27   28   29   30   31   32   33
34   35   36   37   38   39   40   41   42   43   44
45   46   47   48   49   50
```

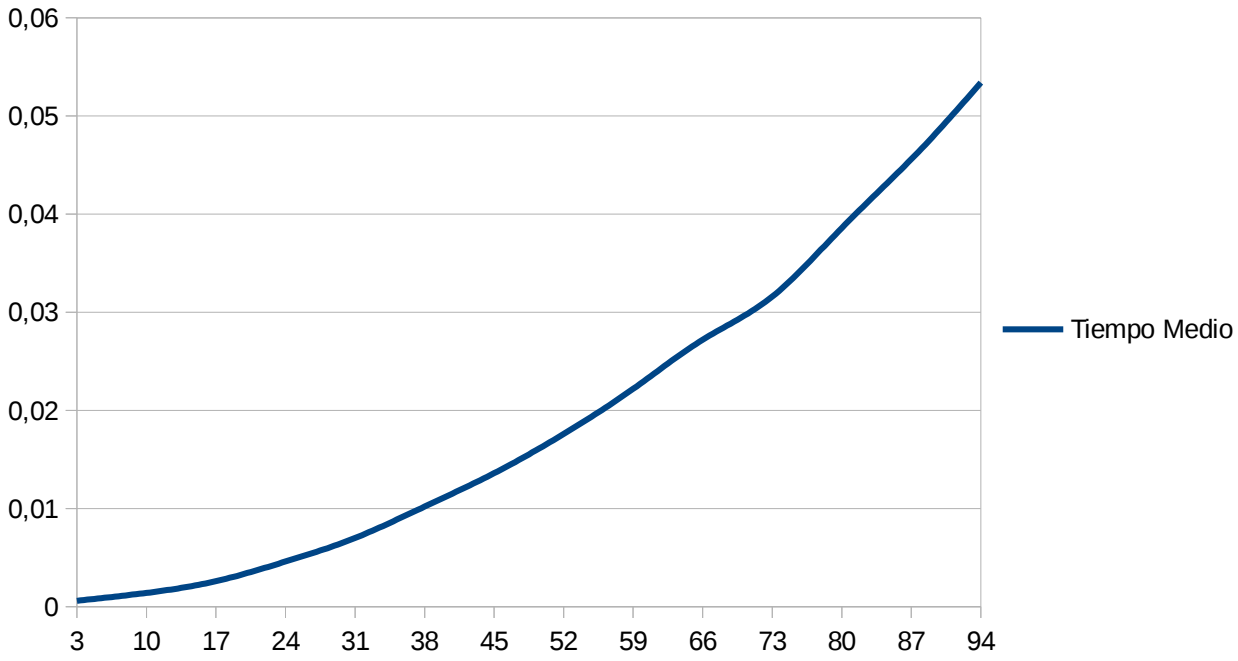
5.5 Apartado 5

```
./ejercicio5 -num_min 3 -num_max 100 -incr 7 -numP 5
-fichSalida Salida.txt

Practica numero 1, apartado 5
Realizada por: Sergio Galán Martín y Rafael Sánchez
Sánchez
Grupo: 1201
Salida correcta

Salida.txt:
3 0.000600 3.000000 3 3
10 0.001400 45.000000 45 45
17 0.002600 136.000000 136 136
24 0.004600 276.000000 276 276
31 0.007000 465.000000 465 465
38 0.010200 703.000000 703 703
45 0.013600 990.000000 990 990
52 0.017600 1326.000000 1326 1326
59 0.022200 1711.000000 1711 1711
66 0.027200 2145.000000 2145 2145
73 0.031600 2628.000000 2628 2628
80 0.038600 3160.000000 3160 3160
```

```
87 0.045600 3741.000000 3741 3741
94 0.053400 4371.000000 4371 4371
```



Como observamos, tanto el máximo, el mínimo y la media coinciden. Esto se debe a que BubbleSort siempre ejecuta el mismo numero de OBs. Podemos ver como sigue una progresión cuadrática.



El tiempo medio (en ms) también sigue una progresión más o menos cuadrática. No es exacta ya que el tiempo depende del sistema en que se ejecute el código.

5. Respuesta a las preguntas teóricas.

Aquí respondéis a las preguntas teóricas que se os han planteado en la práctica.

5.1 Justifica tu implementación de aleat num ¿en qué ideas se basa? ¿de qué libro/artículo, si alguno, has tomado la idea? Propón un método alternativo de generación de números aleatorios y justifica sus ventajas/desventajas respecto a tu elección.

Se basa en que los bits más significativos de los números devueltos por la función rand() son más aleatorios que los bits menos significativos del resultado de la misma función por cuestiones de implementación. Sacado de: *Numerical Recipes in C. Página 277.*

Como método alternativo podemos usar la implementación con módulos de aleat_num. En este caso el código de la función es mucho más legible, pero menos aleatoria.

Por otro lado, los *"true" random-number-generators* funcionan a partir de inputs físicas como micrófonos en distintos lugares, como bosques o ciudades. Sin embargo, su implementación es mucho más costosa.

5.2 Justifica lo más formalmente que puedas la corrección (o dicho de otra manera, el porqué ordena bien) del algoritmo BubbleSort.

El algoritmo es correcto por que como hemos visto, resuelve el problema para el que fue diseñado, es decir, ordena los elementos.

Además, para cada entrada (tabla de enteros) produce la salida deseada (la tabla ordenada).

Finalmente, tiene un tiempo de ejecución finito que depende del tamaño del input, siguiendo la formula $N^2/2 + N/2$.

5.3 ¿Por qué el bucle exterior de BubbleSort no actúa sobre el primer elemento de la tabla?

Por que al ser un bucle decreciente, el último elemento ya estará ordenado para cuando el bucle interno llegue a él.

5.4 ¿Cuál es la operación básica de BubbleSort?

La comparación de claves

5.5 Dar tiempos de ejecución en función del tamaño de entrada n para el caso peor WBS(n) y el caso mejor BBS(n) de BubbleSort. Utilizar la notación asintótica (O, Θ, o, Ω, etc) siempre que se pueda.

Al ser la implementación sin flag, WBS(n) y BBS(n) harán exactamente el mismo número de operaciones básicas (comparaciones de clave), y esto implica que WBS(n) = BBS(n) = (N^2)/2 + O(N)

6. Conclusiones finales.

Hemos visto de manera práctica y experimental lo visto en la teoría. Además, hemos creado librerías para poder probar cualquier algoritmo de ordenación. Por ello, quizá habría sido más interesante analizar el algoritmo con flag, para que el minimo de OBs y el máximo de OBs no coincidieran.

Los resultados obtenidos eran los esperados. Por otro lado, hemos comprobado que los reportes de valgrind sean correctos. No se adjuntan por la extensión de los mismos.

El Makefile fue modificado para poder utilizar la función ceil(); de math.h, añadiendo *-lm* a la línea que compila la función.