Análisis de Algoritmos

18/10/2024

Práctica 1

Javier Moreno y Guillermo Santaolalla,

Grupo 1202

Código	Gráficas	Memoria	Total

1. Introducción.

En esta práctica vamos a implementar el código para generar permutaciones y analizar su tiempo de ejecución, para luego discutirlo en función de los resultados.

2. Objetivos

2.1. Rutina **random_num**.

Implementar la rutina **int random_num (int inf, int sup)** que deberá generar un número aleatorio equiprobable entre los enteros inf y sup, incluidos.

2.2. Rutina **generate_perm**.

Implementar la rutina i**nt *generate_perm(int N)** que, utilizando la rutina del apartado 1, genere una permutacion aleatoria de tamaño N.

2.3. Rutina **generate_permutations**.

Implementar la rutina **int **generate_permutations(int n_perms, int N)** que que genere mediante la función generate_perm del ejercicio anterior n_perms permutaciones equiprobables de N elementos cada una.

2.4. Función **BubbleSort**.

Implementar la función **int BubbleSort(int *array, int ip, int iu)** para el método de ordenación por ascenso de burbuja *BubbleSort*. Esta función devuelve ERR en caso de error o el número de veces que se ha ejecutado la OB en el caso de que la tabla se ordene correctamente, array es la tabla a ordenar, ip es el primer elemento de la tabla e iu es el último elemento de la tabla.

2.5. Rutinass de **times.c**.

1. Implementar la rutina **short average_sorting_time (pfunc_sort method, int n_perms, int N, PTIME_AA time)** donde pfunc_sort es un puntero a la función de ordenaciónh, n_perms representa el número de permutaciones a generar y ordenar por el método que se use (en este *BubbleSort*), N es el tamaño de cada permutación y ptime es un puntero a una estructura de tipo TIME que a la salida de la función contendrá: el número de permutaciones promediadas en el campo n_elems, el tamaño de las permutaciones en el campo N, el tiempo medio de ejecución (en segundos) en el campo tiempo, el número promedio de veces que se ejecutó la OB en el campo average_ob, el número mínimo de veces que se ejecutó la OB min_ob y el número máximo de veces que se ejecutó la OB en el campo max_ob sobre las permutaciones generadas por la función generate permutations. Devuelve ERR en caso de error y OK en el caso de que las tablas se ordenen correctamente.

2. Implementar la rutina **short generate_sorting_times (pfunc sort method, char * file, int num_min, int num_max, int incr, int n_perms)** que escribe en el fichero file los tiempos medios, y los números promedio, mínimo y máximo de veces que se ejecuta la OB en la ejecución del algoritmo de ordenación method con n_perms permutaciones de tamaños en el rango desde num_min hasta num_max, ambos incluidos, usando incrementos de tamaño incr. La rutina devolverá el valor ERR en caso de error y OK en caso contrario. **generate_sorting_times** llamará a una rutina **save_time_table.**

3. Implementar la rutina s**hort sabe_time_table (char * file, TIME_AA time_aa, int n_times)** con la que se imprime en un fichero una tabla con cinco columnas correspondientes al tamaño N, al tiempo de ejecución time y al número promedio bf average_ob, máximo bf max_ob y mínimo min_ob de veces que se ejecuta la OB; el array time_aa guarda los tiempos de ejecución y n_times es el número de elementos del array time_aa.

2.6. Función **BubbleSortFlag**.

Implementar una rutina mejora sobre el algoritmo *BubbleSort* que introduzca un flag que detecta si la tabla está ya ordenada. Comparad el tiempo mejor, peor y medio de OBs y el tiempo medio de reloj con los obtenidos por la rutina *BubbleSort.*

3. Herramientas y metodología

A lo largo de toda la práctica, hemos programado tanto en WSL como en Linux, utilizando herramientas como VSCode, Valgrind y GNuplot para el entorno, comprobaciones y gráficas, respectivamente. Para probar el funcionamiento correcto del código, hemos hecho uso de los ejercicios de prueba facilitados en la práctica. Para la implementación de la función generadora de números aleatorios nos hemos apoyado en el libro proporcionado en la propia práctica.

4. Código fuente

4.1. Apartado 1

```c
int random_num(int inf, int sup) {
  if (inf > sup) return ERR;
  return inf + (int)((double)(sup - inf + 1) * rand() / (RAND_MAX + 1.0));
}
```

4.2. Apartado 2

```c
int *generate_perm(int N) {
  int i, temp, random, *arr;

  if (N <= 0) return NULL;
  arr = (int *)malloc(N * sizeof(int));
  if (arr == NULL) return NULL;

  for (i = 1; i <= N; i++) {
    arr[i - 1] = i;
  }

  for (i = 0; i < N; i++) {
    random = random_num(0, N - 1);
    if (random == ERR) {
      free(arr);
      return NULL;
    }

    temp = arr[i];
    arr[i] = arr[random];
    arr[random] = temp;
  }
  return arr;
}
```

4.3. Apartado 3

```c
int **generate_permutations(int n_perms, int N) {
  int **matrix = NULL, i;

  if (n_perms <= 0 || N <= 0) return NULL;
  if (!(matrix = (int **)calloc(n_perms, sizeof(int *)))) return NULL;

  for (i = 0; i < n_perms; i++) {
    if (!(matrix[i] = generate_perm(N))) {
      for (i--; i >= 0; i--) {
        free(matrix[i]);
      }
      free(matrix);
      return NULL;
    }
  }

  return matrix;
}
```

4.4. Apartado 4

```c
int BubbleSort(int* array, int ip, int iu) {
  int i, j, temp, ob_counter = 0;

  if (!array || ip > iu) return ERR;
  for (i = iu; i >= ip + 1; i--) {
    for (j = ip; j <= i - 1; j++) {
      ob_counter++;
      if (array[j] > array[j + 1]) {
        temp = array[j];
        array[j] = array[j + 1];
        array[j + 1] = temp;
      }
    }
  }
  return ob_counter;
}
```

4.5. Apartado 5

```
int cceil(double x) {
  int integer_part = (int)x;
  if (x > integer_part) return integer_part + 1;
  return integer_part;
}

short average_sorting_time(pfunc_sort method, int n_perms, int N, PTIME_AA ptime)
{
  int **matrix = NULL, i, current_ob = 0;
  double total_ob = 0.0, avg_time = 0.0;
  clock_t start_time, end_time;

  if (!method || n_perms <= 0 || N <= 0 || !ptime) return ERR;

  ptime->n_elems = n_perms;
  ptime->N = N;

  if (!(matrix = generate_permutations(n_perms, N))) return ERR;

  for (i = 0; i < n_perms; i++) {
    if ((start_time = clock()) == ERR) return ERR;

    if ((current_ob = method(matrix[i], 0, N - 1)) == ERR) return ERR;

    if ((end_time = clock()) == ERR) return ERR;

    if (i == 0)
      ptime->min_ob = ptime->max_ob = current_ob;

    else if (current_ob < ptime->min_ob)
      ptime->min_ob = current_ob;
    else if (current_ob > ptime->max_ob)
      ptime->max_ob = current_ob;

    total_ob += current_ob;
    avg_time += (double)(end_time - start_time) /
                CLOCKS_PER_SEC; /*Calculate the average time per interaction. We
pass from ticks to seconds dividing by CLOCKS_PER_SEC*/
  }

  ptime->time = avg_time / n_perms;
  ptime->average_ob = total_ob / n_perms;
```

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

  return OK;
}

short generate_sorting_times(pfunc_sort method, char *file, int num_min, int
num_max, int incr, int n_perms) {
  PTIME_AA ptime;
  int i, n_rows;

  /* We check that the increment is greater than or equal to 0 to avoid infinite
loops */
  if (!method || !file || num_min > num_max || incr <= 0 || n_perms <= 0) return
ERR;

  n_rows = (int)cceil((num_max - num_min + 1) / (double)incr);
  ptime = (PTIME_AA)malloc(n_rows * sizeof(TIME_AA));
  if (!ptime) {
    printf("Error allocating memory\n");
    return ERR;
  }

  for (i = num_min; i <= num_max; i += incr) {
    /* (i - num_min) / incr is the n-1th iteration of this loop. e.g. when the
loop starts, i =
     * num_min, so (i - num_min) / incr = 0 */
    if (average_sorting_time(method, n_perms, i, ptime + (i - num_min) / incr) ==
ERR) {
      printf("Error in average_sorting_time for i = %d\n", i);
      free(ptime);
      return ERR;
    }
  }

  if (save_time_table(file, ptime, n_rows) == ERR) {
    free(ptime);
    return ERR;
  }

  free(ptime);
  return OK;
```

```
}

short save_time_table(char *file, PTIME_AA ptime, int n_times) {
  FILE *f;
  int i = 0;

  f = fopen(file, "w");
  if (!f) return ERR;

  for (i = 0; i < n_times; i++) {
    fprintf(f, "%d  %lf %lf %d  %d\n", ptime[i].N, ptime[i].time,
ptime[i].average_ob, ptime[i].max_ob, ptime[i].min_ob);
  }

  fclose(f);
  return OK;
}
```

4.6 Apartado 6

```
int BubbleSortFlag(int* array, int ip, int iu) {
  int i, j, temp, ob_counter = 0, flag = 1;

  if (!array || ip > iu) return ERR;
  for (i = iu; flag == 1 && i >= ip + 1; i--) {
    flag = 0;
    for (j = ip; j <= i - 1; j++) {
      ob_counter++;
      if (array[j] > array[j + 1]) {
        temp = array[j];
        array[j] = array[j + 1];
        array[j + 1] = temp;
        flag = 1;
      }
    }
  }
  return ob_counter;
}
```

5. Resultados, Gráficas

5.1. Apartado 1

Nos hemos apoyado en el capítulo 7 (Random numbers) del libro "Numerical recipes in C: the art of scientific computing" para diseñar una mejor funcion generadora de números aleatorios.

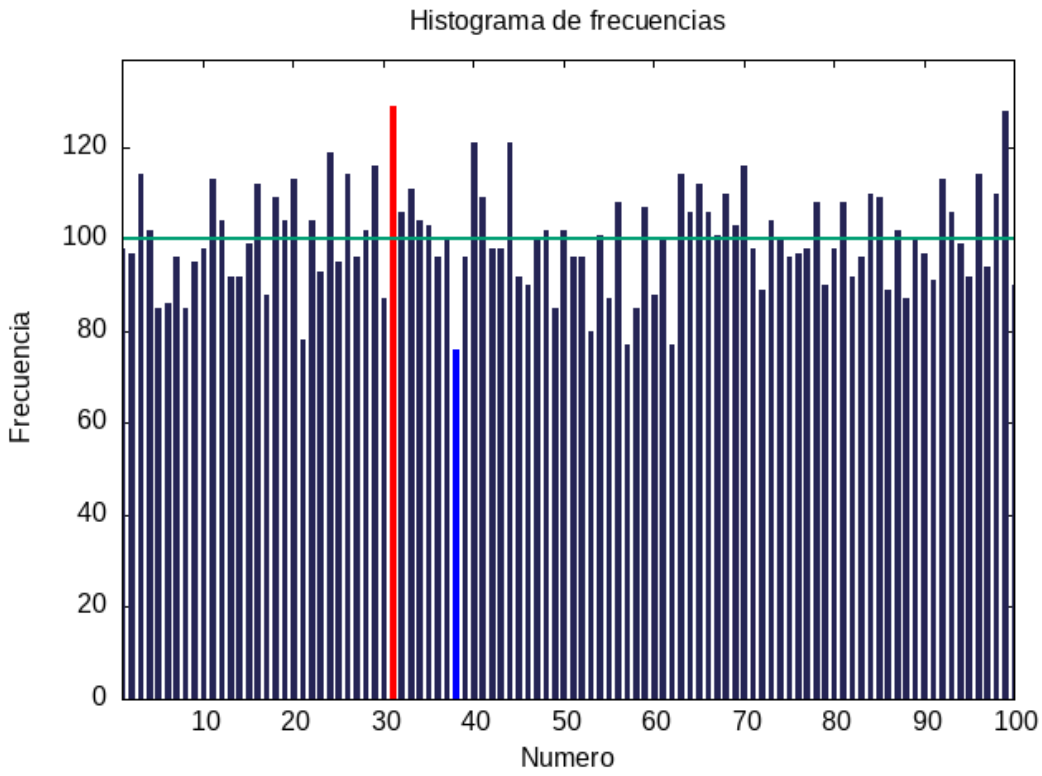
Generando entre 10 mil y 10 millones de números, podemos ver que cuántos más números generamos, menor es la desviación porcentual. Es decir, cuando consideramos muchos números, la selección de cada uno es equiprobable.

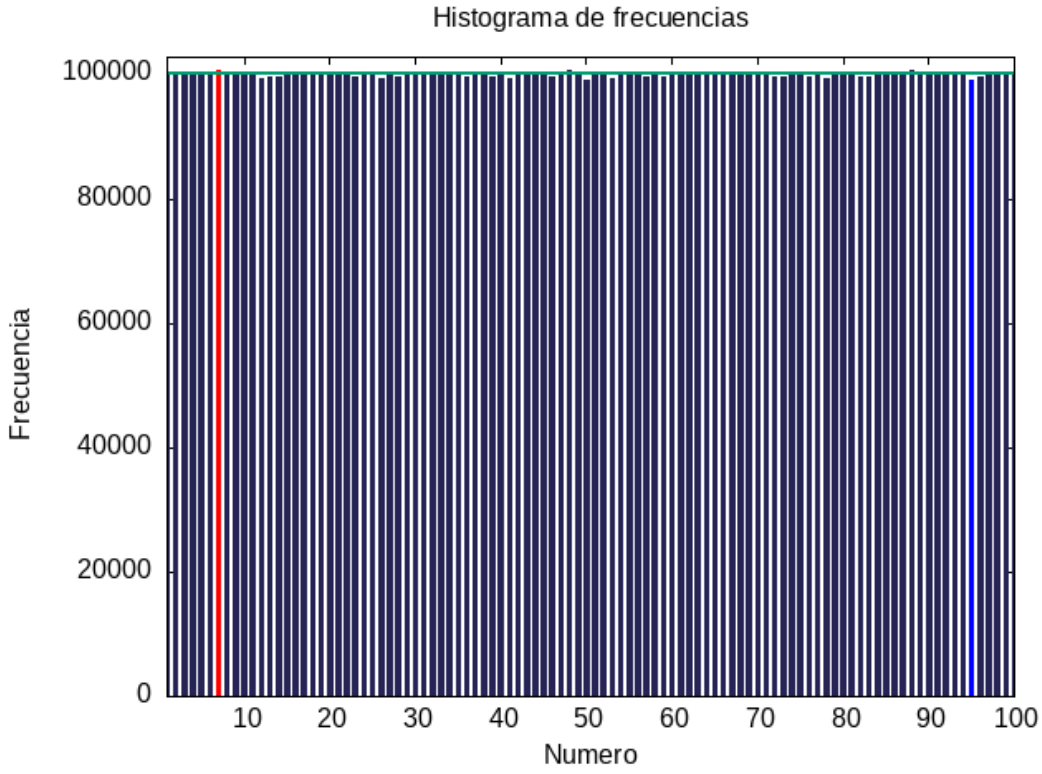
Números generados	Frencuencia esperada	Min	Max	100 * Desviación %
10.000	100	76	129	8.6%
100.000	1.000	916	1065	0.25%
1.000.000	10.000	9746	10234	0.008%
10.000.000	100.000	99126	100775	0.00026%

La fórmula empleada para calcular 100 * Desviación % es:

$$100 \cdot Desviacion \; = \; 100 \cdot \frac{\sum_{i=1}^{n} |x_i - VE|}{VE} \cdot 100\%$$

donde VE es el valor esperado y n la cantidad de números generados. En las gráficas 1 y 2 podemos ver la diferencia en el histograma y en la desviación porcentual cuando se generan 10 mil y 10 millones de números, respectivamente. La caja roja representa el número que fue generado con mayor frecuencia y en azul que el que fue generado con menor frencuencia. La línea verde representa el valor esperado.


Histograma de frecuencias
Frecuencia
Numero
0
20
40
60
80
100
120
10
20
30
40
50
60
70
80
90
100

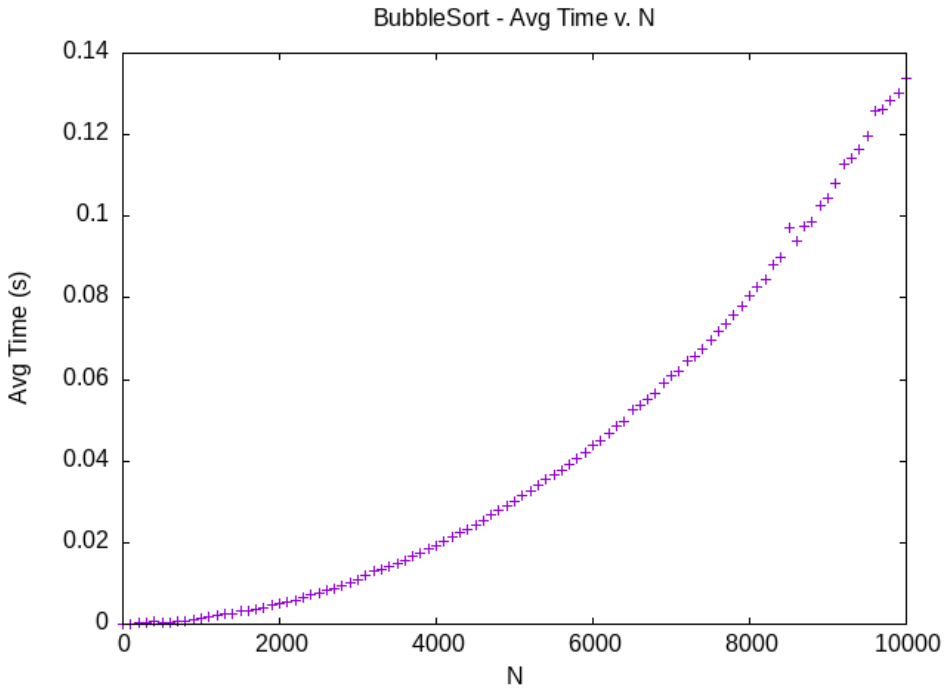

Histograma de frecuencias
Frecuencia
Numero
0
20000
40000
60000
80000
100000
10
20
30
40
50
60
70
80
90
100

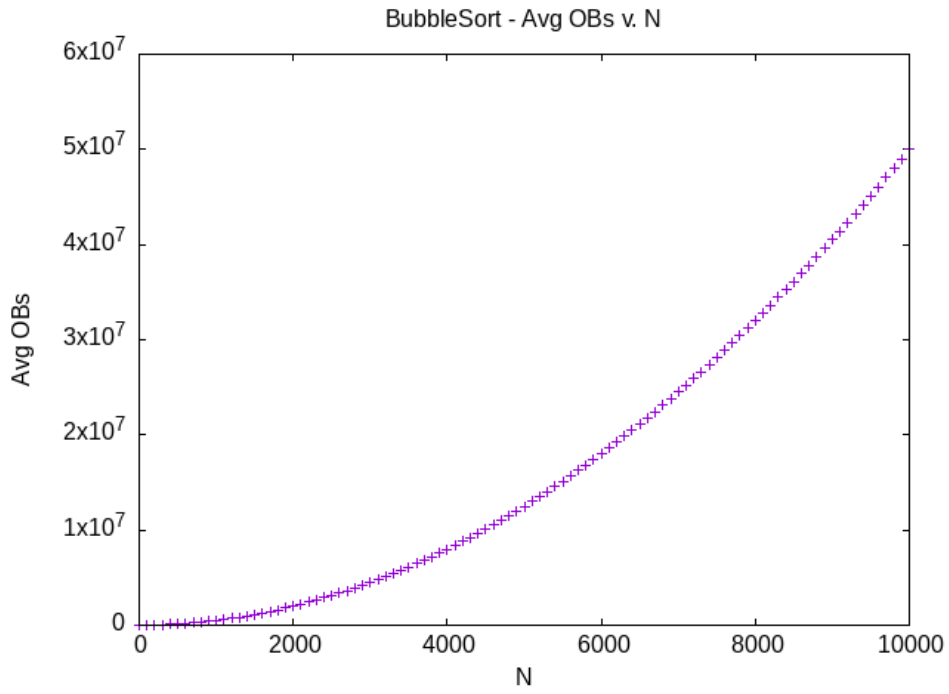
5.5. Apartado 5

El BubbleSort implementado en este apartado no tiene flag, lo que quiere decir que no distinguirá entre un *array* ordenada y una desordenada. Es por este motivo que, independientemente de lo ordenada que esté un *array*, el método realizará el mismo número de operaciones para un *array* de tamaño N. Sabiendo esto, no es necesario promediar más de 1 array para cada tamaño N puesto que todas resultarán en el mismo número de OBs y tiempo de ejecución.

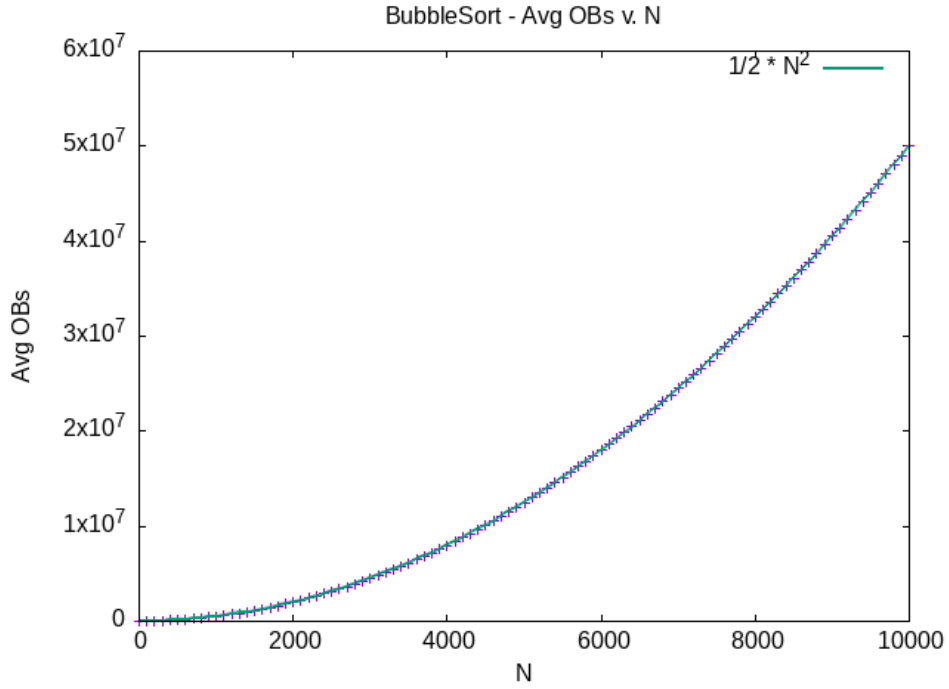
Podemos concluir que el **mejor, peor y tiempo medio** del BubbleSort es el **mismo**.

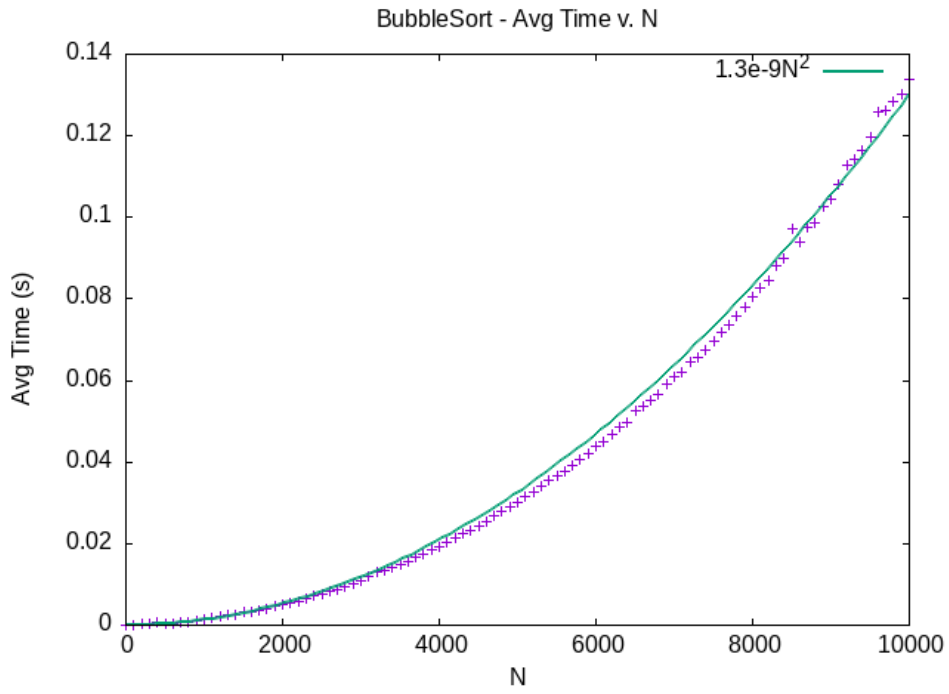
Para evitar que puntos se "escapen" de la gráfica aparentemente cuadrática (luego lo comprobaremos) hemos utilizado incrementos de 50. Esto minimiza el efecto que tienen sobre la gráfica aquellas *arrays* que, por un motivo u otro, tarden más o menos tiempo del debido, (en cuanto al número de operaciones realizadas para ordenar) en ordenarse.





Las gráficas parecen ser de la forma $f(N) = k \cdot N^2$ lo que quiere decir que el crecimiento del tiempo/OBs es cuadrático respecto a N.

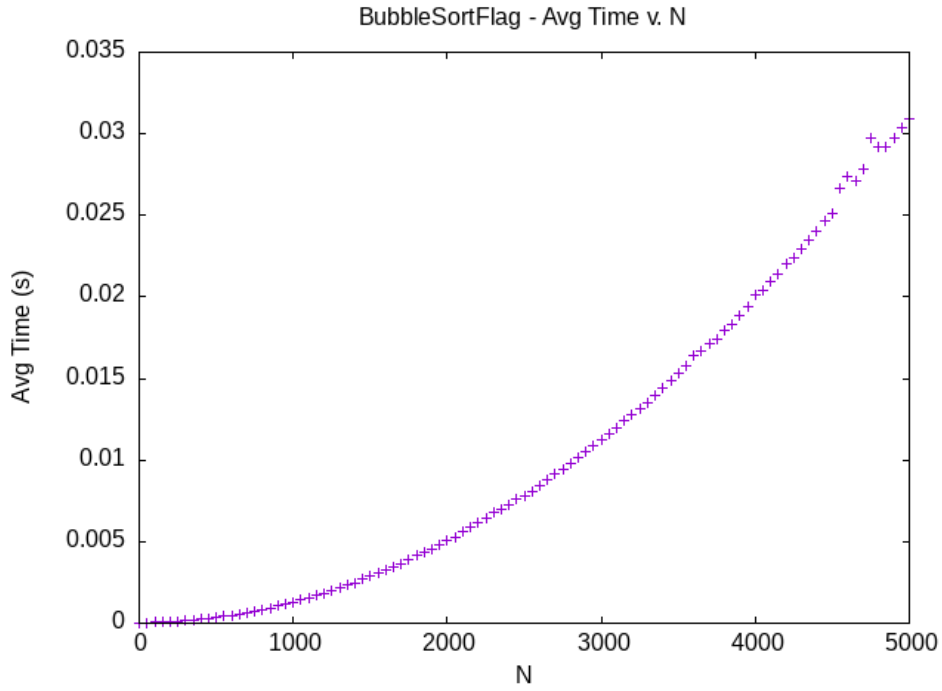


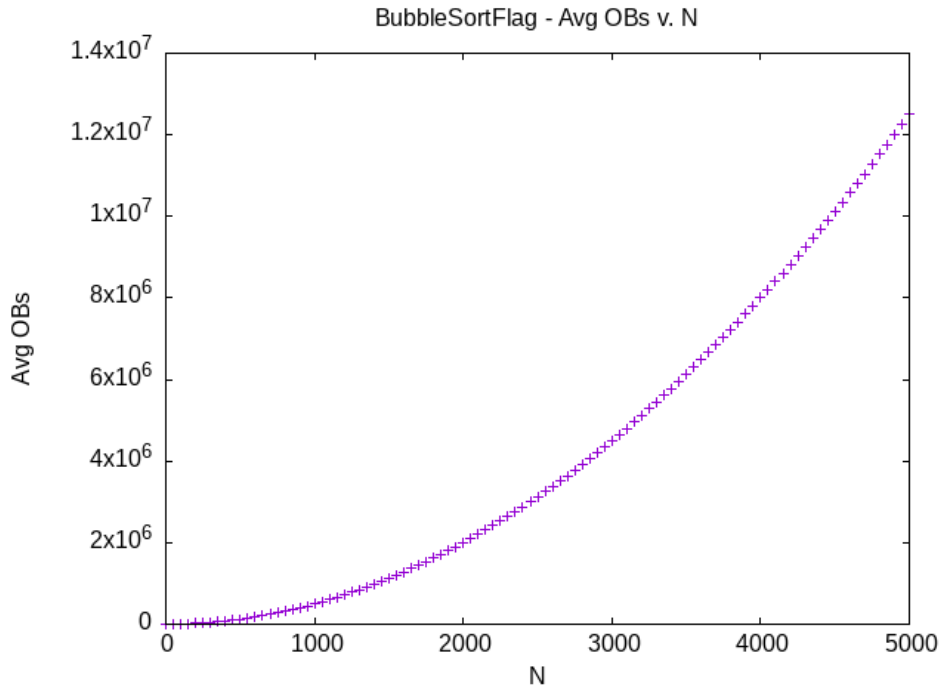


5.6. Apartado 6

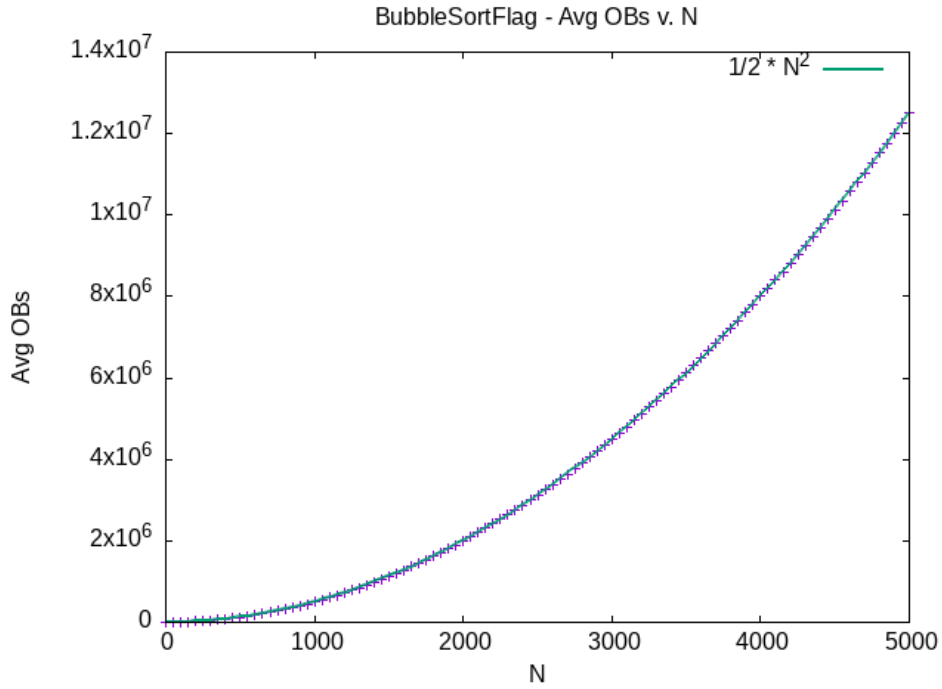
El BubbleSortFlag se diferencia del BubbleSort en que detecta cuando el *array* está ordenado. Esto ayuda a reducir mucho el tiempo que tarda el BubbleSort en ordenar, evitando iteraciones redundantes sobre el *array* cuando este ya está ordenado. Es por esto por lo que el caso medio, peor y mejor es distintos. En el apartado 6.5 le damos valores al mejor, peor y caso medio.

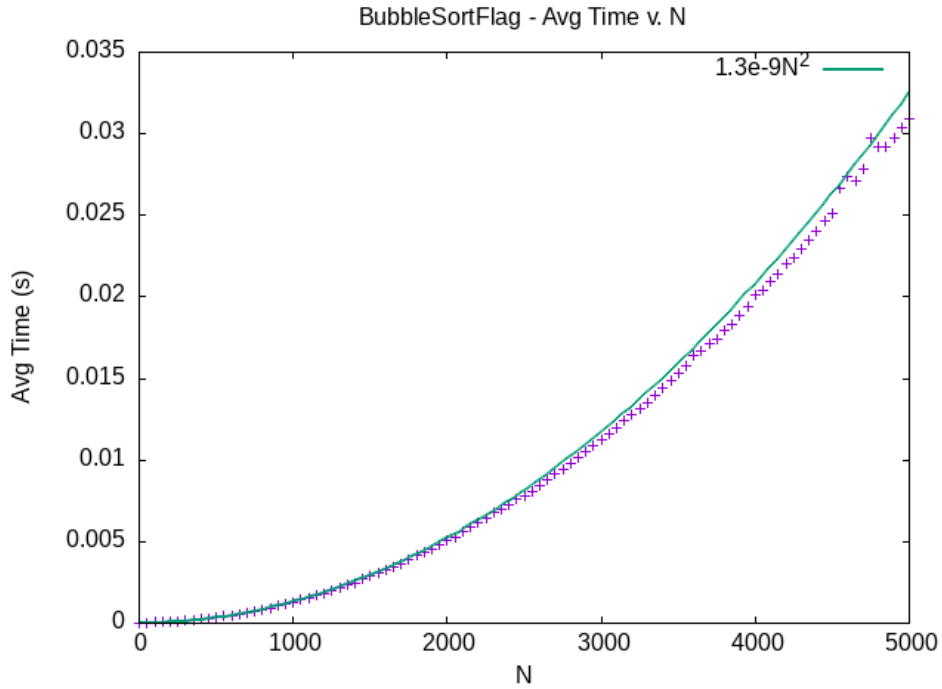
Como el tiempo/número de OBs de BubbleSortFlag depende de lo ordenada que el *array* que recibe esté, vamos a generar múltiples *arrays* para cada tamaño N para promediar el tiempo y número de OBs y evitar así poder considerar solo buenos o malos casos. En el apartado 6.5 diremos que constituye un "buen caso" y un "mal caso".





Las gráficas parecen ser de la forma $f(N) = k \cdot N^2$ lo que quiere decir que el crecimiento del tiempo/OBs es cuadrático respecto a N.





6. Respuesta a las preguntas teóricas.

6.1. Nuestra implementación de aleat_num se basa en el capítulo 7 (Random numbers) del libro "Numerical recipes in C: the art of scientific computing" y se basa en la idea de que el rand de C es un generador lineal congruencial en el que los dígitos más significativos son más aleatorios que los dígitos menos significativos.

Como método alternativo, podemos implementar un generador lineal congruencial. Esto es:

$$I_{j+1} = aI_j + c(mod\, m)$$

Es decir, el i-ésimo número generado corresponde al módulo del (i-1)-ésimo número generado multiplicado y sumado por una constante. Si quisiésemos imitar el rand de C, m sería RAND_MAX. Uno de los inconvenientes de este tipo de algoritmos es tras generar un número "aleatorio" se conocen todos los siguientes números generados, pues solo hay que aplicar la fórmula. Una ventaja de este tipo de algoritmos es que son fáciles de entender y muy eficientes.

6.2. El algoritmo *BubbleSort* ordena una lista comparando elementos adyacentes y permutándolos si están en el orden incorrecto. Recorre repetidamente la lista, intercambiando pares de elementos hasta que no se realizan más intercambios, lo que significa que la lista está ordenada. En cada pasada, el elemento más grande "burbujea" hacia su posición correcta al final de la lista. El proceso se repite hasta que toda la lista está ordenada. Aunque es sencillo de implementar, su eficiencia es baja en comparación con otros algoritmos de ordenación, con un tiempo promedio y peor caso de $O(n^2)$.

6.3. El bucle exterior de BubbleSort no actúa sobre el primer elemento de la tabla porque el *BubbleSort* va colocando el mayor elemento al final de la tabla en cada interación del bucle, por lo que, tras la última iteración del bucle, el primer elemento (en ip) estará automáticamente ordenado si el resto ya lo está.

6.4. La operación básica (OB) es | if (array[j] > array[j + 1]) | puesto que se encuentra en el bucle más interno del algoritmo y es representativa del mismo. Además, es representativa de *SelectSort* y de otros muchos algoritmos de ordenación.

6.5. El BubbleSort normal no sabe cuándo un *array* está ordenado, por lo que si consideramos la OB | if (array[j] > array[j + 1]), el peor y el mejor caso será el mismo --> $\frac{N^2}{2} + O(N)$

El BubbleSortFlag si sabe cuándo un *array* está ordenado. El mejor caso será aquel en el que el *array* de tamaño N tenga forma [1, 2, 3, ..., N] es decir $O(N)$, pues solo iterará una vez sobre el bucle exterior y N – 1 veces el bucle interior.

El peor caso del BubbleSortFlag se da cuando el *array* está completamente desordenado, es decir, si el *array* tiene tamaño N, será de la forma [N, N – 1, ... , 1]. En estos casos, el algoritmo nunca parará de ordenar por la flag, sino por llegar al fin del bucle, por lo que el tae será máximo $\frac{N^2}{2} + O(N)$

6.6. Comparando las gráficas del apartado 5.6 podemos observar que no existe gran diferencia entre ambos. Esto se debe a que el *BubbleSortFlag* solo aporta una mejora significativa en los casos más favorables, en los que se encuentre la tabla ordenada. Un "buen caso" es aquel en el que la tabla ya está ordenada, por lo que no deberá iterar en dicha tabla y ahorrará tiempo de ejecución; por otro lado, un "mal caso" es aquel en el que no encuentra tabla ordenada y, por tanto, es indistinguible del *BubbleSort*. De esta manera, al generar tantas permutaciones no se aprecia mucho la diferencia, porque todo se promedia.

7. Conclusiones.

En esta práctica, nos centramos en implementar y analizar algoritmos de ordenación, en concreto el ***BubbleSort*** y una versión mejorada con un flag. Hemos llegado a las siguientes conclusiones:

1. **Generación de Permutaciones y Aleatoriedad**: Para generar números aleatorios y permutaciones, utilizamos un método basado en el libro "Numerical Recipes in C". Observamos que cuanto mayor es el número de datos generados, más se acerca la frecuencia real a la esperada, lo que demuestra que el generador funciona correctamente para grandes volúmenes.

2. ***BubbleSort***: Implementamos *BubbleSort*, un algoritmo de ordenación que compara y ordena elementos adyacentes. Como era de esperar, el tiempo de ejecución sigue un patrón cuadrático (O(N2)O(N^2)O(N2)), lo que significa que, para listas grandes, se vuelve bastante ineficiente. Además, no importa si la lista está ordenada o no, siempre realiza el mismo número de operaciones.

3. **Mejora con *BubbleSortFlag***: Al introducir un flag que detecta si el *array* ya está ordenado, el rendimiento mejora bastante en casos favorables. Esto ayuda a reducir el número de operaciones cuando el *array* está parcial o totalmente ordenado, evitando iteraciones innecesarias. Sin embargo, al promediar resultados, la mejora no es tan notable porque muchos *arrays* son desordenados.

4. **Cálculo de Tiempos y Operaciones Básicas (OBs)**: En el apartado 4 de la práctica, implementamos varias funciones para medir el rendimiento del algoritmo. La función *average_sorting_time* se encarga de calcular el tiempo medio y las operaciones básicas (OBs) promedio para un conjunto de permutaciones, mientras que *generate_sorting_times* genera y almacena estos datos en un archivo. Finalmente, la función *save_time_table* permite guardar los resultados en un fichero, detallando el tamaño de las permutaciones, el tiempo de ejecución, y el número promedio, mínimo y máximo de OBs para cada caso.

En definitiva, no solo implementamos algoritmos de ordenación, sino que también analizamos su rendimiento en base al tiempo de ejecución y el número de operaciones básicas necesarias. Aunque *BubbleSort* muestra un crecimiento cuadrático y es ineficiente para grandes conjuntos de datos, el uso del *BubbleSortFlag* mejora su rendimiento en casos favorables. Además, las funciones implementadas para calcular los tiempos y las OBs fueron clave para poder visualizar estos resultados, dándonos una visión más clara sobre cómo optimizar algoritmos y medir su eficiencia en diferentes escenarios.