Análisis de Algoritmos 2017/2018

Práctica 1

Victoria Pelayo e Ignacio Rabuñal, grupo

Código	Gráficas	Memoria	Total

1. Introducción.

En esta práctica hemos implementado varias rutinas diferentes. En primer lugar, construímos las rutinas de generación de numeros aleatorios y permutaciones y comprobamos que tienen la aleatoreidad correcta. Después implementamos el algoritmo de ordenación BubbleSort y finalmente las funciones para generar y manejar los tiempos de ejecución pedidos.

2. Objetivos

2.1 Apartado 1

Construir una rutina que genere un numero aleatorio equiprobable entre dos enteros definidos.

2.2 Apartado 2

Implementar la rutina de generación de permutaciones aleatorias a partir del pseudocódigo dado.

2.3 Apartado 3

Implementar la rutina que genera un numero de permutaciones equiprobables de un numero determinado de elementos (ambos pasados como argumento)

2.4 Apartado 4

Implementar una rutina para el algoritmo de ordenación BubbleSort.

2.5 Apartado 5

Implementar la estructura y rutinas para poder determinar el tiempo medio de ejecución y el numero medio, máximo y mínimo de veces que se ejecuta la operación básica del algoritmo anteriormente implementado.
Estos datos se guardarán en un fichero.

3. Herramientas y metodología

En todos los apartados hemos utilizado el entorno Linux y las herramientas Gedit, gcc, Valgrind, Gnuplot, Sort, uniq y Make..

3.1 Apartado 1

Adaptamos la rutina rand para que generara los numeros aleatorios entre los enteros que deseemos. Comprobamos que funcionaba correctamente con el fichero ejercicio1.c y posteriormente cambiando los parametros de entrada por números más grandes para comprobar su correcta aleatoreidad. Por último, comprobamos que la gestión de memoria fuera correcta con Valgrind.

3.2 Apartado 2

Adaptamos el pseudocódigo dado en el ejercicio para crear la rutina de generación de permutaciones. Comprobamos que funcionaba correctamente con el fichero ejercicio2.c . Por último, comprobamos que la gestión de memoria fuera correcta con Valgrind.

3.3 Apartado 3

Partiendo de la función creada en el apartado 2 creamos la rutina que genera las permutaciones que queramos del numero de elementos que queramos. Comprobamos que funcionaba correctamente con el fichero ejercicio3.c y posteriormente cambiando los parametros de entrada por números más grandes para comprobar su correcta aleatoreidad. Por último, comprobamos que la gestión de memoria fuera correcta con Valgrind.

3.4 Apartado 4

Partiendo del pseudocódigo que aparece en las transparencias de la parte teórica de la asignatura creamos la rutina que implementa el algoritmo de ordenación BubbleSort. Comprobamos que funcionaba correctamente ejecutando el fichero ejercicio4.c varias veces para así ver que la ordenación fuera siempre correcta. Por último, comprobamos que la gestión de memoria fuera correcta con Valgrind.

3.5 Apartado 5

Para la primera función hermos contado el número mínimo y máximo de OBs que realizaba un método al ordenar una tabla. Hemos medido el tiempo antes y después de realizar el bucle porque clock no es tan exacta como para medir lo que tarda Bubble Sort en ordenar una sola permutación. Por ello luego lo dividimos entre el número de permutaciones y entre CLOCKS_PER_SEC, para obtener el tiempo medio .

En la función genera_tiempos_ordenación, en tiempos va guardando en cada una de sus posiciones el resultado de llamar a la función anterior pasándole el método (en esta práctica es Bubble Sort), el número de permutaciones, tamaño y en que posición de timepos lo guarda. Luego al final se llamará a la función guarda_tabla_tiempos.

La función guarda_tabla_tiempos escribirá en un fichero (de cada una de las posiciones del array tiempos) el tamaño, tiempo medio, número promedio que se ejecuta la OB, el número máximo y el número mínimo.

4. Código fuente

Código Fuente de las rutinas que hemos desarrollado en cada apartado.

4.1 Apartado 1

```
int aleat_num(int inf, int sup){

        return inf + (rand() % (sup - inf + 1));

}
```

4.2 Apartado 2

```
int* genera_perm(int N){

	int i;
	int* perm = (int *) malloc(N * sizeof(perm[0]));
	if (perm == NULL)
		return NULL;

	for(i=0; i < N; i++)
		perm[i] = i + 1;

	for(i=0; i < N; i++)
		swap (&perm[i], &perm[aleat_num(i, N - 1)]);

	return perm;

}
```

4.3 Apartado 3

```
int** genera_permutaciones(int n_perms, int N){

	int i;

	int **perms=(int**)malloc(n_perms*sizeof(perms[0]));

	if(perms == NULL) return NULL;

	for(i=0;i<n_perms;i++){

		perms[i]=genera_perm(N);

		if (perms[i]==NULL){

			int j;

			for (j=0;j<i;j++) free(perms[j]);

			free(perms);

		}

	}
	return perms;

}
```

4.4 Apartado 4

```
int BubbleSort(int* tabla, int ip, int iu){
	int cont = 0;
	int i;
	int flag = 1;

	i = iu;

	if (tabla == NULL) return ERR;

	while (flag == 1 && i >= ip + 1) {
		int j;
		flag = 0;
		for (j = ip; j <= i - 1; j++){
			cont++;
			if (tabla[j] > tabla[j+1]){
				swap(&tabla[j], &tabla[j + 1]);
				flag = 1;
			}
		}
		i--;
	}

	return cont;
}
```

4.5 Apartado 5

```
short tiempo_medio_ordenacion(pfunc_ordena metodo,
                              int n_perms,
                              int N,
                              PTIEMPO ptiempo)
{
        clock_t ini,fn;
        int** perms= (int**)genera_permutaciones(n_perms,N);
        int i;
        if (perms == NULL) return ERR;

        ptiempo->N = N;
        ptiempo->min_ob = INT_MAX;
        ptiempo->min_ob = INT_MAX;
        ptiempo->max_ob = 0;
        ptiempo->medio_ob = 0;
        ini = clock( );
        for( i=0; i < n_perms; i++){
                int ob = metodo(perms[i], 0, N - 1);
                if(ob == ERR){
                        for(i=0; i < n_perms; i++) free(perms[i]);
                        free(perms);
                        return ERR;
                }
                if (ob < ptiempo->min_ob) ptiempo->min_ob = ob;
                if (ob > ptiempo->max_ob) ptiempo->max_ob = ob;
                ptiempo->medio_ob += ob;
```

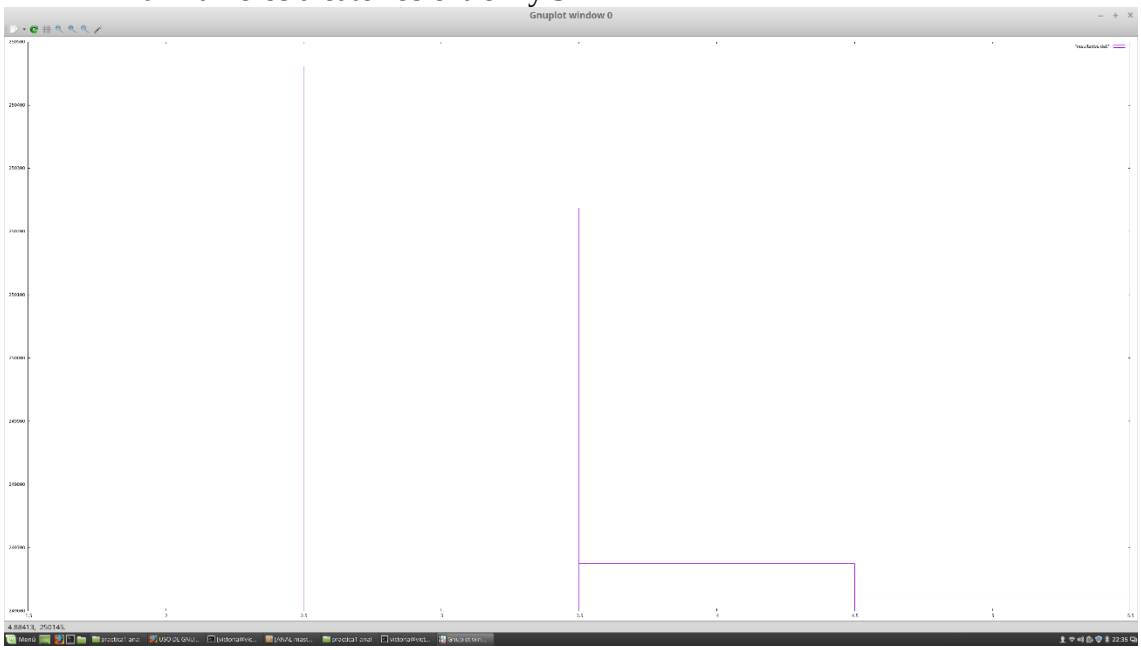
```
        }
        fn = clock();
        ptiempo->medio_ob /= n_perms;
        ptiempo->tiempo = (double)(fn - ini)/ n_perms / CLOCKS_PER_SEC;

        for(i=0; i < n_perms; i++) free(perms[i]);
        free(perms);

        return OK;
}

short genera_tiempos_ordenacion(pfunc_ordena metodo, char* fichero,
                        int num_min, int num_max,
                        int incr, int n_perms){
        int n = ((num_max - num_min) / incr) + 1;
        int j, tamanio;

        PTIEMPO tiempos = (PTIEMPO)malloc(n * sizeof(tiempos[0]));

        if(tiempos == NULL) free(tiempos);

        for (j = 0,tamanio = num_min; tamanio <= num_max; j++, tamanio+=incr){
                short codigo = tiempo_medio_ordenacion(metodo, n_perms, tamanio, &tiempos[j]);
                fprintf(stderr, "%d\n", tamanio);
                if (codigo == ERR){
                        free(tiempos);
                        return ERR;
```

```
                }


        }
        if (guarda_tabla_tiempos(fichero, tiempos, n) == ERR) return ERR;
        free(tiempos);

        return OK;
}

short guarda_tabla_tiempos(char* fichero, PTIEMPO tiempo, int n_tiempos){
        int i;

        FILE *f = fopen(fichero,"w");
        if (f == NULL) return ERR;

        for (i = 0; i < n_tiempos; i++){
                fprintf(f, "%d\t", tiempo[i].N);
                fprintf(f, "%f\t", tiempo[i].tiempo);
                fprintf(f, "%f\t", tiempo[i].medio_ob);
                fprintf(f, "%d\n", tiempo[i].max_ob);
                fprintf(f, "%d\t", tiempo[i].min_ob);

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

5. Resultados, Gráficas

5.1 Apartado 1

Gráfica del histograma de números aleatorios.
Eran números aleatorios entre 2 y 5

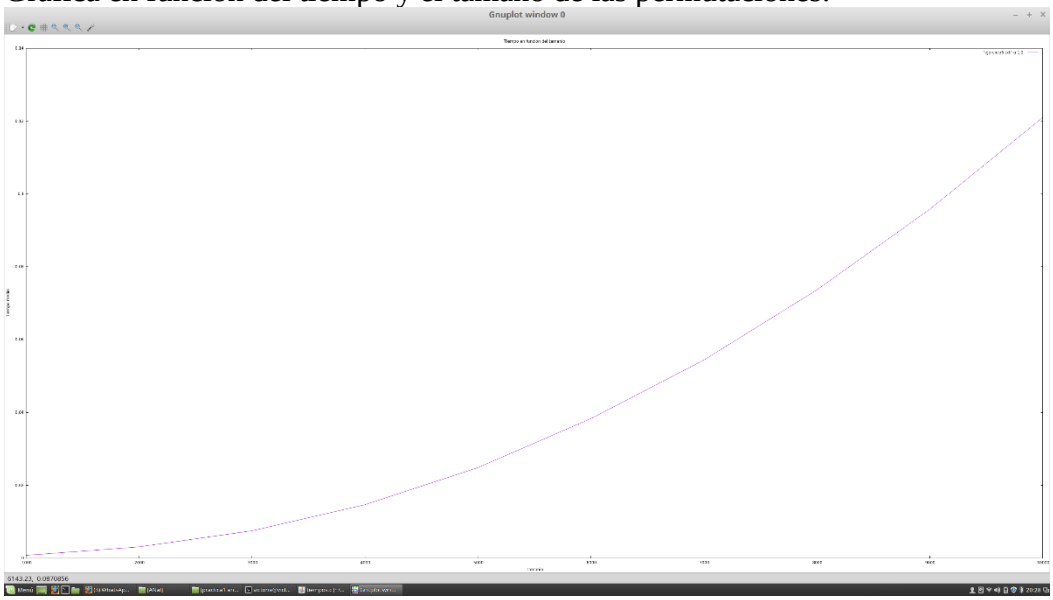


Viendo la gráfica parece que aparecen mas el 2 y 3 que el 4 y 5, sin embargo al mirar los números vemos que la diferencia no es tan grande, se ve así por la escala y el intervalo tomados.

5.2Apartado 5

Vamos a nalizar las gráficas obtenidas de los tiempos de la realización del algoritmo Bubble Sort. Estos datos los hemos tomado de un fichero que se ha creado con el ejercicio 5 (ejercicio5.txt) y como argumentos hemos pasado los siguientes : -num_min 1000 -num_max 10000 -incr 1000 -numP 10 -fichSalida ejercicio5.txt

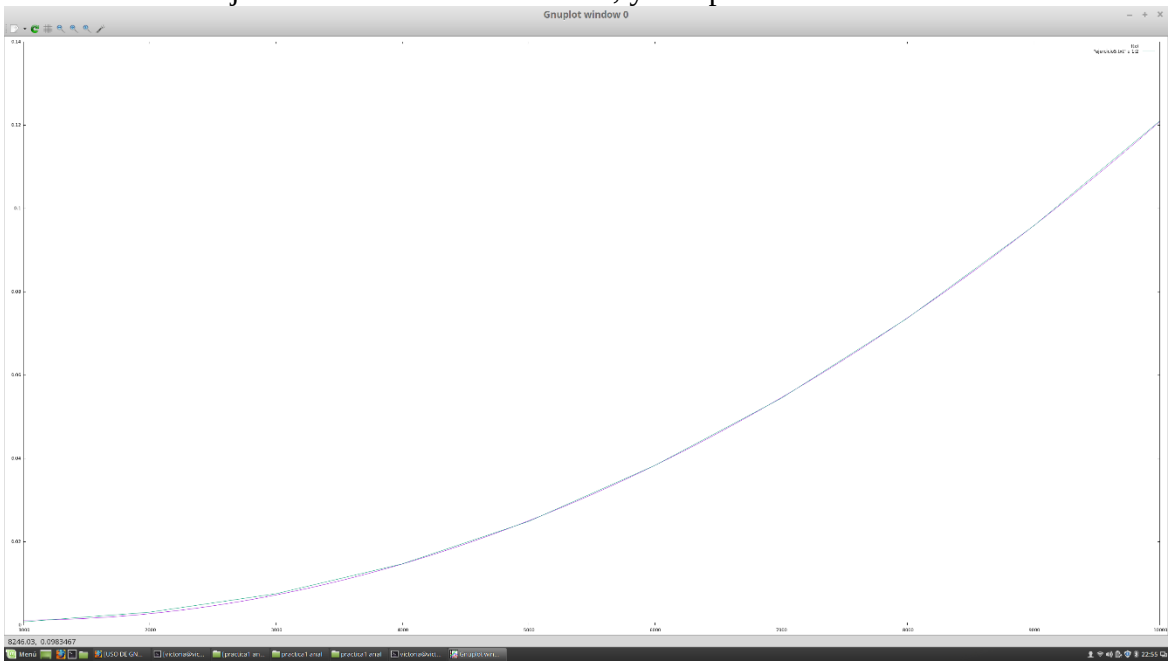
Gráfica en función del tiempo y el tamaño de las permutaciones:



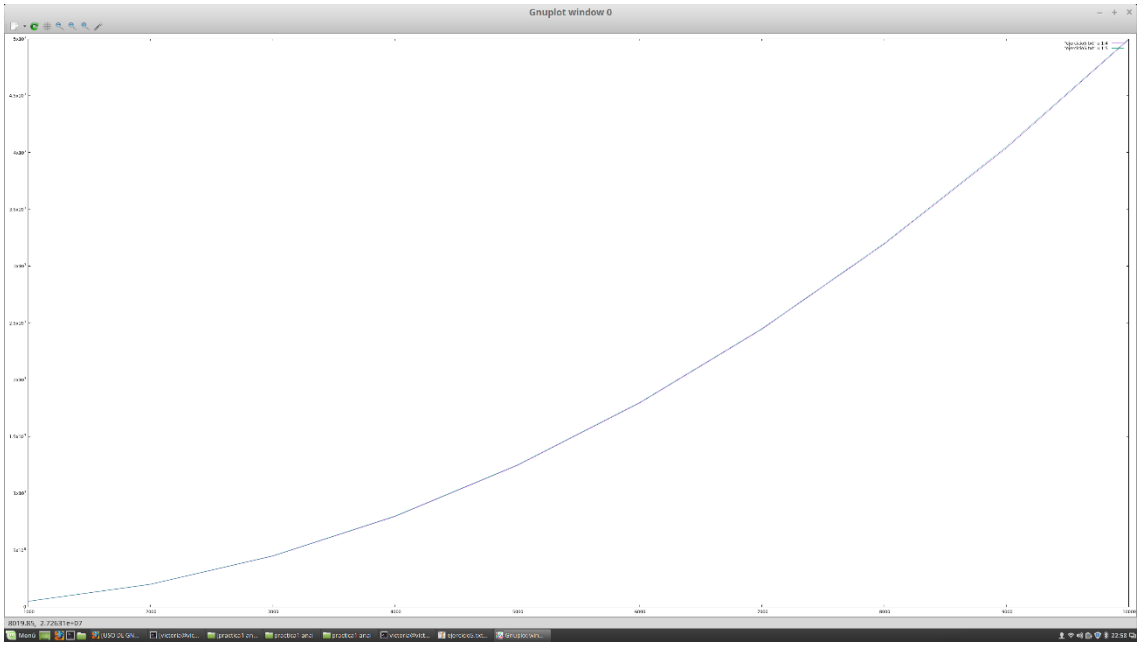
A simple vista no sabemos si es cuadrática, súbica, por ello utilizaremos la función fit f(x) de gnuplot y luego representaremos ambas gráficas.

```
gnuplot> f(x) = a*x**2 + b*x + c
gnuplot> fit f(x) "ejercicio5.txt" via a,b,c
iter      chisq       delta/lim  lambda   a             b             c
   0 2.5339051096e+16   0.00e+00  2.91e+07    1.000000e+00   1.000000e+00   1.000000e+00
   1 2.6367402694e+13  -9.60e+07  2.91e+06    3.214251e-02   9.998844e-01   1.000000e+00
   2 2.6726452096e+07  -9.87e+10  2.91e+05   -1.086585e-04   9.998777e-01   1.000000e+00
   3 2.3785293712e+07  -1.24e+04  2.91e+04   -1.193752e-04   9.995961e-01   9.999999e-01
   4 2.2499868658e+07  -5.71e+03  2.91e+03   -1.161039e-04   9.722000e-01   9.999895e-01
   5 1.5444412316e+06  -1.36e+06  2.91e+02   -3.039558e-05   2.544317e-01   9.997170e-01
   6 2.0045806267e+01  -7.70e+09  2.91e+01   -7.613611e-08   5.201514e-04   9.996121e-01
   7 7.1743934116e-01  -2.69e+06  2.91e+00    3.147060e-08  -3.808974e-04   9.987589e-01
   8 6.0875171381e-01  -1.79e+04  2.91e-01    2.910821e-08  -3.511151e-04   9.202008e-01
   9 6.6602784059e-03  -9.04e+06  2.91e-02    4.358809e-09  -3.927267e-05   9.852066e-02
  10 4.5528734138e-07  -1.46e+09  2.91e-03    1.471123e-09  -2.887825e-06   2.649490e-03
  11 4.4622096364e-07  -2.03e+03  2.91e-04    1.467750e-09  -2.845323e-06   2.537501e-03
  12 4.4622096364e-07  -2.79e-07  2.91e-05    1.467750e-09  -2.845323e-06   2.537500e-03
iter      chisq       delta/lim  lambda   a             b             c

After 12 iterations the fit converged.
final sum of squares of residuals : 4.46221e-07
rel. change during last iteration : -2.78614e-12

degrees of freedom    (FIT_NDF)                        : 7
rms of residuals      (FIT_STDFIT) = sqrt(WSSR/ndf)    : 0.000252479
variance of residuals (reduced chisquare) = WSSR/ndf   : 6.37459e-08

Final set of parameters            Asymptotic Standard Error
=======================            =========================
a               = 1.46775e-09      +/- 1.099e-11    (0.7486%)
b               = -2.84532e-06     +/- 1.24e-07     (4.359%)
c               = 0.0025375        +/- 0.000297     (11.7%)

correlation matrix of the fit parameters:
                a      b      c
a               1.000
b              -0.975  1.000
c               0.814 -0.909  1.000
gnuplot>
```

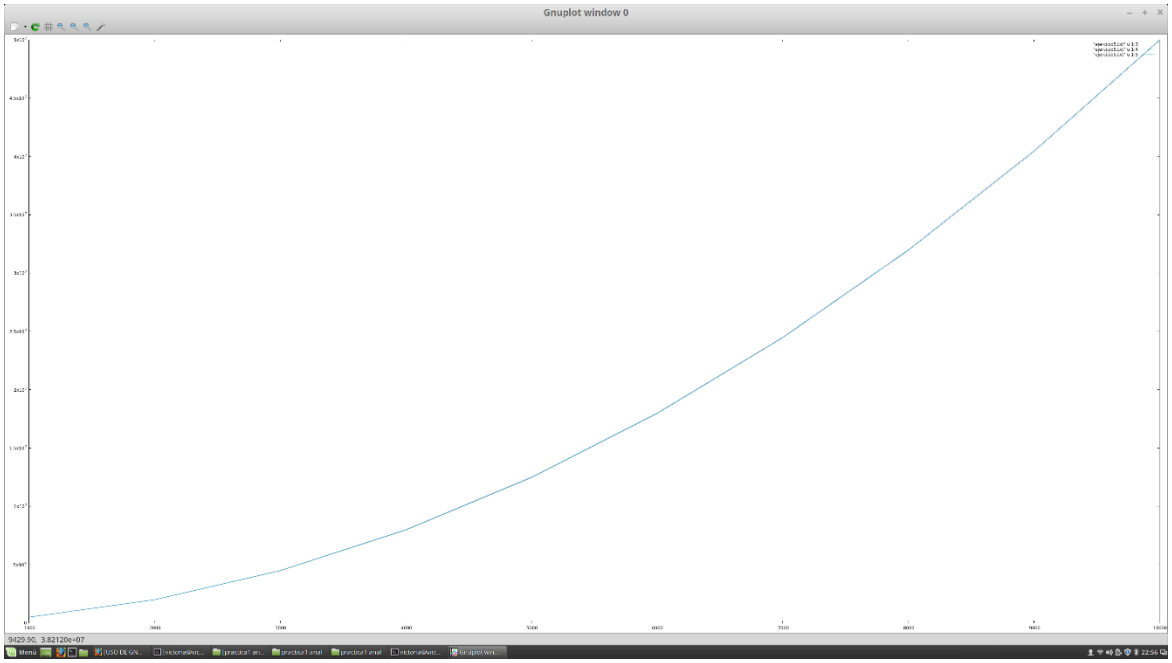
Vemos como se ajusta a una función cuadrática, y al representar ambas se ve claramente:



Gráfica comparando el número máximo y mínimo de ejercución de OBs para BubbleSort. En estas gráficas la columna de máximo y mínimo están al revés de como el código finalmente las crea porque en clase lo teníamos así y luego en casa nos dimos cuenta y cambiamos el orden de imprimir en el fichero del código.



No se ve claramente, porque las líneas están muy juntas pero si haces zoom se ve.
Ahora al añadir el número medio (sin zoom se ve como una línea), pero realmente va entre ambas líneas (el máximo y mínimo):



5. Respuesta a las preguntas teóricas.

5.1 Pregunta 1

Utilizamos la función rand() que devuelve un número aleatorio entre 0 y RAND_MAX. Dado que queremos un número entre el ínfimo y el supremo, este número será un número aleatorio en un intervalo de (sup-inf +1) número.

Primero hacemos rand() % (sup-inf+1), así obtenemos un número entre 0 y sup-inf. Para obtener un número entre el ínfimo y el supremo a ese número obtenido hay que sumarle el ínfimo. Y este número finalmente será el que devuelva la función.

Un método alternativo podría ser:

```
int aleat_num(int inf, int sup){
        int num;
        num = rand % sup;
        while (num < inf)
                num = rand() %sup;

        return num;
}
```

Las desventajas de este otro método principalmente serían el tiempo de ejecución ya que si el intervalo entre el ínfimo y el supremo fuera muy pequeño podría tardarse bastante en encontrar un número aleatorio entre ellos.

La nuestra simplemente toma un número aleatorio y realiza una serie de operaciones para encontrar un número aleatorio en el intervalo que se nos pide.

5.2 Pregunta 2

El algoritmo Bubble Sort empieza ordenando desde el final. Dentro del bucle principal hay otro dentro que se encarga de ir comparando desde el primer element hasta en el que nos encontremos y ir realizando los swaps correspondientes.
Hay dos implementaciones una con dos bucles for, esta no la hemos realizado ya que se t arda lo mismo tanto con la table ordenada como desordenada.
La otra version, la que hemos implementado, tiene un flag, el bucle principal se realizará siempre que el flag esté a 1. Al principio se incializa a 1 para que entre en el bucle, una vez dentró s epodrá a 0, excepto si se produce algún swap, de esta manera si el resto de la tabla está ordenada el algortimo parará.

5.3 Pregunta 3

Porque no es necesario hacer una última interación ya que los dos últimos elementos en el peor de los casos se ordenan a la vez

5.4 Pregunta 4

La operación básica de BubbleSort es la comparación entre un element de la table y su consecutivo.

5.5 Pregunta 5

Caso peor: W(n) = O(n^2)

Caso mejor: B(n) = O(n)

6. Capturas Valgrind.

Ejercicio1 :

```
==2823== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
victoria@victoria-HP-ENVY-Notebook-13-ab0XX ~/Escritorio/practica1 anal $ valgrind --leak-check=fu
ll ./ejercicio1 -limInf 1 -limSup 20 -numN 15
==2852== Memcheck, a memory error detector
==2852== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==2852== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==2852== Command: ./ejercicio1 -limInf 1 -limSup 20 -numN 15
==2852==
Practica numero 1, apartado 1
Realizada por: Vuestros nombres
Grupo: Vuestro grupo
2
4
16
19
8
9
3
10
8
17
10
9
17
18
19
==2852==
==2852== HEAP SUMMARY:
==2852==     in use at exit: 0 bytes in 0 blocks
==2852==   total heap usage: 1 allocs, 1 frees, 1,024 bytes allocated
==2852==
==2852== All heap blocks were freed -- no leaks are possible
==2852==
==2852== For counts of detected and suppressed errors, rerun with: -v
==2852== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
victoria@victoria-HP-ENVY-Notebook-13-ab0XX ~/Escritorio/practica1 anal $
```

Ejercicio2:

```
victoria@victoria-HP-ENVY-Notebook-13-ab0XX ~/Escritorio/practica1 anal
Archivo Editar Ver Buscar Terminal Ayuda
victoria@victoria-HP-ENVY-Notebook-13-ab0XX ~/Escritorio/practica1 anal $ valgrind --leak-check=fu
ll ./ejercicio2 -tamanio 10 -numP 10
==2789== Memcheck, a memory error detector
==2789== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==2789== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==2789== Command: ./ejercicio2 -tamanio 10 -numP 10
==2789==
Practica numero 1, apartado 2
Realizada por: Vuestros nombres
Grupo: Vuestro grupo
2 6 10 7 3 8 9 5 4 1
5 7 10 1 2 6 3 4 9 8
3 7 2 5 6 4 9 8 10 1
6 5 2 1 7 8 3 10 9 4
10 4 9 8 3 6 5 7 1 2
1 2 5 10 7 8 4 3 6 9
8 4 7 9 1 6 10 2 5 3
8 3 7 6 2 9 4 10 1 5
4 5 1 2 10 7 3 8 6 9
10 2 6 8 5 4 9 1 7 3
==2789==
==2789== HEAP SUMMARY:
==2789==     in use at exit: 0 bytes in 0 blocks
==2789==   total heap usage: 11 allocs, 11 frees, 1,424 bytes allocated
==2789==
==2789== All heap blocks were freed -- no leaks are possible
==2789==
==2789== For counts of detected and suppressed errors, rerun with: -v
==2789== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
victoria@victoria-HP-ENVY-Notebook-13-ab0XX ~/Escritorio/practica1 anal $
```

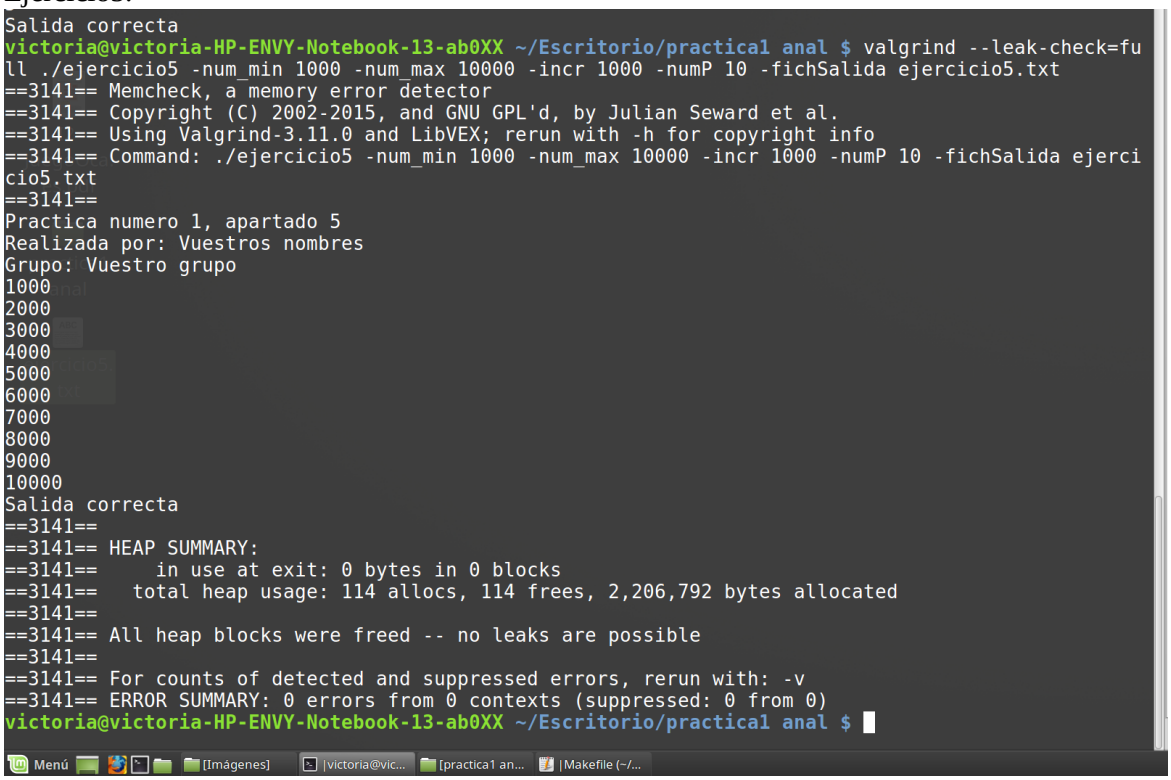
Ejercicio3:

```
victoria@victoria-HP-ENVY-Notebook-13-ab0XX ~/Escritorio/practica1 anal
Archivo Editar Ver Buscar Terminal Ayuda
8 2 10 5 3 9 6 4 7 1
5 1 2 9 10 4 6 3 7 8
4 3 7 2 8 9 1 6 10 5
6 8 7 4 3 2 5 1 9 10
9 1 10 5 7 2 3 4 8 6
7 8 2 5 10 3 6 1 9 4
==2873==
==2873== HEAP SUMMARY:
==2873==     in use at exit: 0 bytes in 0 blocks
==2873==   total heap usage: 12 allocs, 12 frees, 1,504 bytes allocated
==2873==
==2873== All heap blocks were freed -- no leaks are possible
==2873==
==2873== For counts of detected and suppressed errors, rerun with: -v
==2873== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
victoria@victoria-HP-ENVY-Notebook-13-ab0XX ~/Escritorio/practica1 anal $ valgrind --leak-check=fu
ll ./ejercicio3 -tamanio 20 -numP 20
==2882== Memcheck, a memory error detector
==2882== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==2882== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==2882== Command: ./ejercicio3 -tamanio 20 -numP 20
==2882==
Practica numero 1, apartado 3
Realizada por: Vuestros nombres
Grupo: Vuestro grupo
16 11 15 20 17 8 10 5 2 4 9 7 6 19 13 18 12 3 1 14
2 7 13 10 11 3 4 6 5 18 8 20 14 19 17 12 9 16 15 1
17 18 10 16 9 15 11 2 14 12 13 3 8 20 6 19 7 4 5 1
1 6 5 19 17 20 14 10 2 13 9 12 15 11 16 3 4 18 7 8
4 6 3 15 17 8 18 12 7 10 5 16 13 2 14 11 9 19 20 1
13 19 16 17 2 12 15 6 3 11 10 4 18 14 20 7 1 5 8 9
20 1 7 12 15 18 14 6 11 5 17 3 9 2 16 4 10 19 13 8
4 9 2 10 15 11 18 19 6 1 5 20 14 12 8 3 7 17 16 13
7 3 8 11 17 16 6 9 4 20 14 18 13 1 19 2 12 15 5 10
12 7 15 19 4 20 1 2 9 5 16 3 14 8 17 13 6 18 11 10
4 18 16 19 6 12 17 20 11 1 15 13 5 14 7 2 8 10 9 3
13 5 10 20 8 7 17 3 4 15 2 1 6 9 19 11 18 16 12 14
1 8 16 6 5 10 15 13 3 12 11 20 2 9 14 7 18 19 17 4
16 3 14 5 11 15 8 4 2 7 18 20 1 17 12 13 6 9 10 19
7 15 9 17 14 13 5 12 10 19 11 20 4 8 16 18 3 2 1 6
2 4 15 3 9 17 1 5 6 14 8 7 13 18 19 20 11 12 16 10
19 6 2 20 16 10 13 15 14 8 18 11 3 17 4 9 7 5 12 1
3 5 8 9 19 6 13 7 11 14 10 18 4 15 12 17 16 2 20 1
14 11 13 15 2 19 20 1 4 18 17 5 7 8 6 12 9 16 3 10
3 14 1 17 19 5 7 13 12 18 11 8 9 4 6 16 20 2 15 10
==2882==
==2882== HEAP SUMMARY:
==2882==     in use at exit: 0 bytes in 0 blocks
==2882==   total heap usage: 22 allocs, 22 frees, 2,784 bytes allocated
==2882==
==2882== All heap blocks were freed -- no leaks are possible
==2882==
==2882== For counts of detected and suppressed errors, rerun with: -v
==2882== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
victoria@victoria-HP-ENVY-Notebook-13-ab0XX ~/Escritorio/practica1 anal $
```

Ejercicio4:

```
==2882== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
victoria@victoria-HP-ENVY-Notebook-13-ab0XX ~/Escritorio/practica1 anal $ valgrind --leak-check=fu
ll ./ejercicio4 -tamanio 32
==2909== Memcheck, a memory error detector
==2909== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==2909== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==2909== Command: ./ejercicio4 -tamanio 32
==2909==
Practica numero 1, apartado 4
Realizada por: Vuestros nombres
Grupo: Vuestro grupo
 1       2       3       4       5       6       7       8       9       10      11      12      1
3        14      15      16      17      18      19      20      21      22      23      24      2
5        26      27      28      29      30      31      32
==2909==
==2909== HEAP SUMMARY:
==2909==     in use at exit: 0 bytes in 0 blocks
==2909==   total heap usage: 2 allocs, 2 frees, 1,152 bytes allocated
==2909==
==2909== All heap blocks were freed -- no leaks are possible
==2909==
==2909== For counts of detected and suppressed errors, rerun with: -v
==2909== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
victoria@victoria-HP-ENVY-Notebook-13-ab0XX ~/Escritorio/practica1 anal $
```

Ejercicio5:



7.Conclusiones finales.

Esta práctica, a parte de introducción para las siguientes, nos ha servido para familiarizarnos con el manejo de numeros aleatorios y la metodología para medir los tiempos de ejecución. Los resultados finales de los primeros apartados generaban grandes cantidades de numeros aleatorios correctamente y pudimos comprobar que nuestra implementación de BubbleSort cumplía con los tiempos de ejecución teóricos.

También nos ha servido para familiarizarnos con algunas herramientas como gnuplot, que no habíamos usado anteriormente.