Análisis de Algoritmos 2015/2016

Práctica 2

Alfonso Villar y Víctor García, Grupo 12.

Código	Gráficas	Memoria	Total

1. Introducción.

En esta práctica trabajamos con los algoritmos MergeSort y QuickSort estudiados en las clases de teoría en los cuales tendremos que utilizar recursividad al elaborar sendos algoritmos. Comprobaremos el número de operaciones básicas y los tiempos de ejecución de ambos algoritmos de ordenación, llevando a cabo diversas gráficas donde compararemos los resultados obtenidos con los supuestos teóricos.

2. Objetivos

2.1 Ejercicio 1

En el primer ejercicio desarrollaremos el algoritmo MergeSort. Para ello tendremos que dividir la tabla en unidades mediante recursividad, reconstruirla en una tabla temporal ya ordenada y luego copiarla en la final.

2.2 Ejercicio 2

En este ejercicio comprobaremos que el algoritmo MergeSort funciona correctamente creando una gráfica utilizando el ejercicio5.c de la práctica anterior el cual nos devolverá la variación del tiempo promedio de ejecución, máximo, mínimo y promedio de operaciones básicas en función del tamaño de la permutación. Después compararemos los resultados obtenidos con los teóricos.

2.3 Ejercicio 3

En este ejercicio desarrollaremos el algoritmo QuickSort. Para ello iremos ordenando la tabla colocando el pivote en la posición correcta de la tabla mediante recursividad. Además el pivote será el primer elemento de la tabla.

2.4 Ejercicio 4

En este ejercicio comprobaremos que el algoritmo QuickSort funciona correctamente creando una gráfica utilizando el ejercicio5.c de la práctica anterior el cual nos devolverá la variación del tiempo promedio de ejecución, máximo, mínimo y promedio de operaciones básicas en función del tamaño de la permutación. Después compararemos los resultados obtenidos con los teóricos.

2.5 Ejercicio 5 y Ejercicio 6

En estos últimos ejercicios desarrollaremos los algoritmos QuickSort_avg y QuickSort_stat, los cuales son iguales a QuickSort excepto en la selección del pivote de la tabla (QuickSort_avg elige como pivote el elemento en la posición media de la tabla y QuickSort_stat, para elegir el pivote, compara los valores de las posiciones primera, última y media de la tabla y devuelva la posición que contiene el valor intermedio entre las tres), y comprobaremos si los resultados que obtendremos de estos algoritmos son iguales a los de QuickSort o si difieren sus tiempos de ejecución.

3. Herramientas y metodología

3.1 Ejercicio 1

Entorno de desarrollo y ejecución: Linux

Comprobación de pérdidas de memoria y violaciones de segmento: Valgrind

Compilación del código: gcc -ansi -pedantic Flags: -Wall -g

3.2 Ejercicio 2

Entorno de desarrollo y ejecución: Linux

Comprobación de pérdidas de memoria y violaciones de segmento: Valgrind

Elaboración de gráficas: Gnuplot

Compilación del código: gcc -ansi -pedantic Flags: -Wall -g

3.3 Ejercicio 3

Entorno de desarrollo y ejecución: Linux

Comprobación de pérdidas de memoria y violaciones de segmento: Valgrind

Compilación del código: gcc -ansi -pedantic Flags: -Wall -g

3.4 Ejercicio 4

Entorno de desarrollo y ejecución: Linux

Comprobación de pérdidas de memoria y violaciones de segmento: Valgrind

Elaboración de gráficas: Gnuplot

Compilación del código: gcc -ansi -pedantic Flags: -Wall -g

3.5 Ejercicio 5 y Ejercicio 6

Entorno de desarrollo y ejecución: Linux

Comprobación de pérdidas de memoria y violaciones de segmento: Valgrind

Elaboración de gráficas: Gnuplot

Compilación del código: gcc -ansi -pedantic Flags: -Wall -g

4. Código fuente

```
/**
 *
 * Descripcion: Implementacion de funciones de ordenacion
 *
 * Fichero: ordenacion.c
 * Autor: Alfonso Villar y Víctor García
 * Version: 3.2
 * Fecha: 4-11-2016
 *
 */

#include <stdio.h>
#include <stdlib.h>
#include "ordenacion.h"

/***************************************************/
/* Funcion: mergesort Fecha: 21/10/2016            */
/* Autores: Alfonso Villar y Victor Garcia         */
/*                                                 */
/* Entrada:   0 <= ip <= iu       tabla != NULL    */
/* int *tabla: Tabla con los numeros!=NULL         */
/* int ip: Posicion del primer elemento a ordenar  */
/* int iu: Posicion del ultimo elemento a ordenar  */
/*                                                 */
/* Salida: devuelve el numero de veces que se ha   */
/* ejecutado la OB si se ha ordenado la tabla o en */
/* caso de error devuelve ERR                      */
/***************************************************/

int mergesort(int* tabla, int ip, int iu){
        int m;
        int n;
        int ob = 0;

        if (ip > iu)
                return ERR;

        if (ip == iu)
                return OK;

        m = (int)(ip + iu) / 2;
        n = mergesort(tabla, ip, m);
        if (n == ERR)
                return ERR;

        ob += n;
```

```
	n = mergesort(tabla, m+1, iu);
	if (n == ERR)
		return ERR;

	ob += n;

	n = merge (tabla, ip, iu, m);

	ob += n;
	return ob;
}

/*****************************************************/
/* Funcion: merge Fecha: 21/10/2016                  */
/* Autores: Alfonso Villar y Victor Garcia           */
/*                                                   */
/* Entrada: 0 <= ip <= imedio <= iu   tabla != NULL*/
/* int *tabla: Tabla con los numeros!=NULL           */
/* int ip: Posicion del primer elemento a ordenar    */
/* int iu: Posicion del ultimo elemento a ordenar    */
/* int imedio: Posicion del elemento medio de la     */
/* tabla                                             */
/*                                                   */
/* Salida: devuelve el numero de veces que se ha     */
/* ejecutado la OB si se ha ordenado la tabla o en   */
/* caso de error devuelve ERR                        */
/*****************************************************/

int merge(int* tabla, int ip, int iu, int imedio){
	int i;
	int j;
	int k;
	int cont;
	int* tabla2;
	tabla2 = (int*)malloc((iu-ip+1)*sizeof(int));
	if (tabla2 == NULL)
		return ERR;

	i = ip;
	j = imedio + 1;
	k = 0;
	cont = 0;

	while (i <= imedio && j <= iu){
		cont ++;
		if (tabla[i] < tabla[j]){ /*OB del algoritmo*/
			tabla2[k] = tabla[i];
```

```
                        i++;
                }
                else{
                        tabla2[k] = tabla[j];
                        j++;
                }
                k++;
        }

        if (i > imedio){ /*copia el resto de la tabla derecha*/
                while (j <= iu){
                        tabla2[k] = tabla[j];
                        j++;
                        k++;
                }
        }

        else if (j > iu){ /*copia el resto de la tabla izquierda*/
                while (i <= imedio){
                        tabla2[k] = tabla[i];
                        i++;
                        k++;
                }
        }

        for(i=ip, j = 0; i <= iu; i++, j++){
                tabla[i] = tabla2[j];
        }

        free(tabla2);

        return cont;
}

/*****************************************************/
/* Funcion: quicksort Fecha: 4/11/2016           */
/* Autores: Alfonso Villar y Victor Garcia       */
/*                                               */
/* Entrada:   0 <= ip <= iu      tabla != NULL   */
/* int *tabla: Tabla con los numeros!=NULL       */
/* int ip: Posicion del primer elemento a ordenar  */
/* int iu: Posicion del ultimo elemento a ordenar  */
/*                                               */
/* Salida: devuelve el numero de veces que se ha   */
/* ejecutado la OB si se ha ordenado la tabla o en */
/* caso de error devuelve ERR                    */
/*****************************************************/
```

```
int quicksort(int* tabla, int ip, int iu){
	int m;
	int *pos = NULL;
	int ob = 0;

	pos = (int *)malloc(sizeof(pos[0]));
	if (pos == NULL)
		return ERR;

	if (ip > iu){
		free (pos);
		return ERR;
	}

	if (ip == iu){
		free (pos);
		return ob;
	}

	else{
		ob = partir(tabla, ip, iu, pos);
		if (ob == ERR){
			free (pos);
			return ERR;
		}

		m = *pos;

		if (ip < m)
			ob += quicksort (tabla, ip, m-1);
		if (m+1 < iu)
			ob += quicksort (tabla, m+1, iu);
	}
	free (pos);
	return ob;
}

/**************************************************/
/* Funcion: quicksort_avg Fecha: 4/11/2016        */
/* Autores: Alfonso Villar y Victor Garcia        */
/*                                                */
/* Entrada:   0 <= ip <= iu       tabla != NULL   */
/* int *tabla: Tabla con los numeros!=NULL        */
/* int ip: Posicion del primer elemento a ordenar */
/* int iu: Posicion del ultimo elemento a ordenar */
/*                                                */
/* Salida: devuelve el numero de veces que se ha  */
/* ejecutado la OB si se ha ordenado la tabla o en */
```

```
/* caso de error devuelve ERR                    */
/************************************************/

int quicksort_avg(int* tabla, int ip, int iu){
        int m;
        int *pos = NULL;
        int ob = 0;

        pos = (int *)malloc(sizeof(pos[0]));
        if (pos == NULL)
                return ERR;

        if (ip > iu){
                free (pos);
                return ERR;
        }

        if (ip == iu){
                free (pos);
                return ob;
        }

        else{
                ob = partir_avg(tabla, ip, iu, pos);
                if (ob == ERR){
                        free (pos);
                        return ERR;
                }

                m = *pos;

                if (ip < m)
                        ob += quicksort (tabla, ip, m-1);
                if (m+1 < iu)
                        ob += quicksort (tabla, m+1, iu);
        }
        free (pos);
        return ob;
}

/************************************************/
/* Funcion: quicksort_stat Fecha: 4/11/2016     */
/* Autores: Alfonso Villar y Victor Garcia      */
/*                                              */
/* Entrada:   0 <= ip <= iu       tabla != NULL */
/* int *tabla: Tabla con los numeros!=NULL      */
/* int ip: Posicion del primer elemento a ordenar */
/* int iu: Posicion del ultimo elemento a ordenar */
```

```
/*                                      */
/* Salida: devuelve el numero de veces que se ha   */
/* ejecutado la OB si se ha ordenado la tabla o en */
/* caso de error devuelve ERR                      */
/*************************************************/

int quicksort_stat(int* tabla, int ip, int iu){
        int m;
        int *pos = NULL;
        int ob = 0;

        pos = (int *)malloc(sizeof(pos[0]));
        if (pos == NULL)
                return ERR;

        if (ip > iu){
                free (pos);
                return ERR;
        }

        if (ip == iu){
                free (pos);
                return ob;
        }

        else{
                ob = partir_stat(tabla, ip, iu, pos);
                if (ob == ERR){
                        free (pos);
                        return ERR;
                }

                m = *pos;

                if (ip < m)
                        ob += quicksort (tabla, ip, m-1);
                if (m+1 < iu)
                        ob += quicksort (tabla, m+1, iu);
        }
        free (pos);
        return ob;
}

/*************************************************/
/* Funcion: partir Fecha: 4/11/2016              */
/* Autores: Alfonso Villar y Victor Garcia       */
/*                                      */
/* Entrada:   0 <= ip <= iu       tabla != NULL    */
```

```
/* int *tabla: Tabla con los numeros!=NULL         */
/* int ip: Posicion del primer elemento a ordenar  */
/* int iu: Posicion del ultimo elemento a ordenar  */
/* int *pos:Puntero que recibe el pivote           */
/*                                                 */
/* Salida: devuelve el numero de veces que se ha   */
/* ejecutado la OB si se ha ordenado la tabla o en */
/* caso de error devuelve ERR                      */
/*************************************************/

int partir(int* tabla, int ip, int iu,int *pos){
        int m;
        int k;
        int aux;
        int i;
        int ob;

        ob = medio (tabla, ip, iu, pos);
        if (ob == ERR)
                return ERR;

        m = *pos;
        k = tabla[m];

        aux = tabla[ip];
        tabla[ip] = tabla[m];
        tabla[m] = aux;

        m = ip;

        for (i = ip +1; i <= iu; i++){
                if (tabla[i] < k){
                        m++;
                        aux = tabla[i];
              tabla[i] = tabla[m];
              tabla[m] = aux;
                }
                ob ++;
        }


        aux = tabla[ip];
        tabla[ip] = tabla[m];
        tabla[m] = aux;

        *pos = m;

        return ob;
```

```
}

/*****************************************************/
/* Funcion: partir_avg Fecha: 4/11/2016              */
/* Autores: Alfonso Villar y Victor Garcia           */
/*                                                   */
/* Entrada:   0 <= ip <= iu       tabla != NULL      */
/* int *tabla: Tabla con los numeros!=NULL           */
/* int ip: Posicion del primer elemento a ordenar    */
/* int iu: Posicion del ultimo elemento a ordenar    */
/* int *pos:Puntero que recibe el pivote             */
/*                                                   */
/* Salida: devuelve el numero de veces que se ha     */
/* ejecutado la OB si se ha ordenado la tabla o en   */
/* caso de error devuelve ERR                        */
/*****************************************************/

int partir_avg(int* tabla, int ip, int iu,int *pos){
        int m;
        int k;
        int aux;
        int i;
        int ob;

        ob = medio_avg (tabla, ip, iu, pos);
        if (ob == ERR)
                return ERR;

        m = *pos;
        k = tabla[m];

        aux = tabla[ip];
        tabla[ip] = tabla[m];
        tabla[m] = aux;

        m = ip;

        for (i = ip +1; i <= iu; i++){
                if (tabla[i] < k){
                        m++;
                        aux = tabla[i];
              tabla[i] = tabla[m];
              tabla[m] = aux;
                }
                ob ++;
        }
```

```
	aux = tabla[ip];
	tabla[ip] = tabla[m];
	tabla[m] = aux;

	*pos = m;

	return ob;
}

/***************************************************/
/* Funcion: partir_stat Fecha: 4/11/2016           */
/* Autores: Alfonso Villar y Victor Garcia         */
/*                                                 */
/* Entrada:   0 <= ip <= iu       tabla != NULL    */
/* int *tabla: Tabla con los numeros!=NULL         */
/* int ip: Posicion del primer elemento a ordenar  */
/* int iu: Posicion del ultimo elemento a ordenar  */
/* int *pos:Puntero que recibe el pivote           */
/*                                                 */
/* Salida: devuelve el numero de veces que se ha   */
/* ejecutado la OB si se ha ordenado la tabla o en */
/* caso de error devuelve ERR                      */
/***************************************************/

int partir_stat(int* tabla, int ip, int iu,int *pos){
	int m;
	int k;
	int aux;
	int i;
	int ob;

	ob = medio_stat (tabla, ip, iu, pos);
	if (ob == ERR)
		return ERR;

	m = *pos;
	k = tabla[m];

	aux = tabla[ip];
	tabla[ip] = tabla[m];
	tabla[m] = aux;

	m = ip;

	for (i = ip +1; i <= iu; i++){
		if (tabla[i] < k){
			m++;
			aux = tabla[i];
```

```
                tabla[i] = tabla[m];
                tabla[m] = aux;
                    }
                    ob ++;
            }


            aux = tabla[ip];
            tabla[ip] = tabla[m];
            tabla[m] = aux;

            *pos = m;

            return ob;
}

/*****************************************************/
/* Funcion: medio Fecha: 4/11/2016                   */
/* Autores: Alfonso Villar y Victor Garcia           */
/*                                                   */
/* Entrada:   0 <= ip <= iu       tabla != NULL      */
/* int *tabla: Tabla con los numeros!=NULL           */
/* int ip: Posicion del primer elemento a ordenar    */
/* int iu: Posicion del ultimo elemento a ordenar    */
/* int *pos:Puntero que recibe el pivote             */
/*                                                   */
/* Salida: Devuelve la posicion del pivote           */
/*****************************************************/

int medio(int *tabla, int ip, int iu,int *pos){
            *pos = ip;
            return 0;
}


/*****************************************************/
/* Funcion: medio_avg Fecha: 4/11/2016               */
/* Autores: Alfonso Villar y Victor Garcia           */
/*                                                   */
/* Entrada:   0 <= ip <= iu       tabla != NULL      */
/* int *tabla: Tabla con los numeros!=NULL           */
/* int ip: Posicion del primer elemento a ordenar    */
/* int iu: Posicion del ultimo elemento a ordenar    */
/* int *pos:Puntero que recibe el pivote             */
/*                                                   */
/* Salida: Devuelve la posicion del pivote           */
/*****************************************************/
```

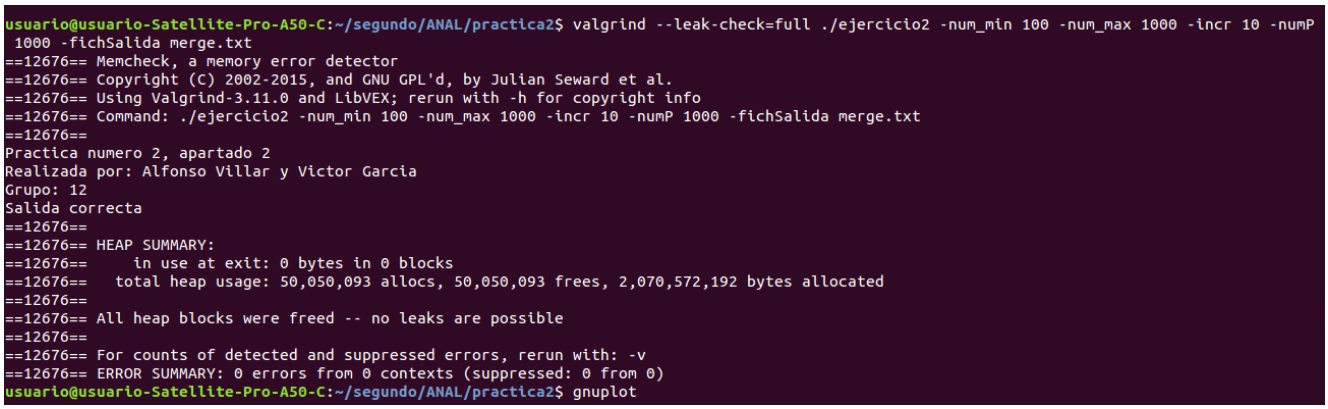
```
int medio_avg(int *tabla, int ip, int iu,int *pos){
	*pos = ip;
	return (int) (ip+iu)/2;
}


/*****************************************************/
/* Funcion: medio_stat Fecha: 4/11/2016              */
/* Autores: Alfonso Villar y Victor Garcia           */
/*                                                   */
/* Entrada:   0 <= ip <= iu       tabla != NULL      */
/* int *tabla: Tabla con los numeros                 */
/* int ip: Posicion del primer elemento a ordenar    */
/* int iu: Posicion del ultimo elemento a ordenar    */
/*                                                   */
/* Salida: Devuelve la posicion del pivote           */
/*****************************************************/

int medio_stat(int *tabla, int ip, int iu,int *pos){
	int m = (int) (ip+iu)/2;
	if (ip < m) {
		if (m < iu){
			*pos = m;
			return m;
		}
		else {
			if (ip < iu){
				*pos = iu;
				return iu;
			}
			*pos = ip;
			return ip;
		}}
	else {
		if (iu < m){
			*pos = m;
			return m;
		}
		else {
			if (ip < iu){
				*pos = ip;
				return ip;
			}
			*pos = iu;
			return iu;
		}}
	}
```

5. Resultados, Gráficas

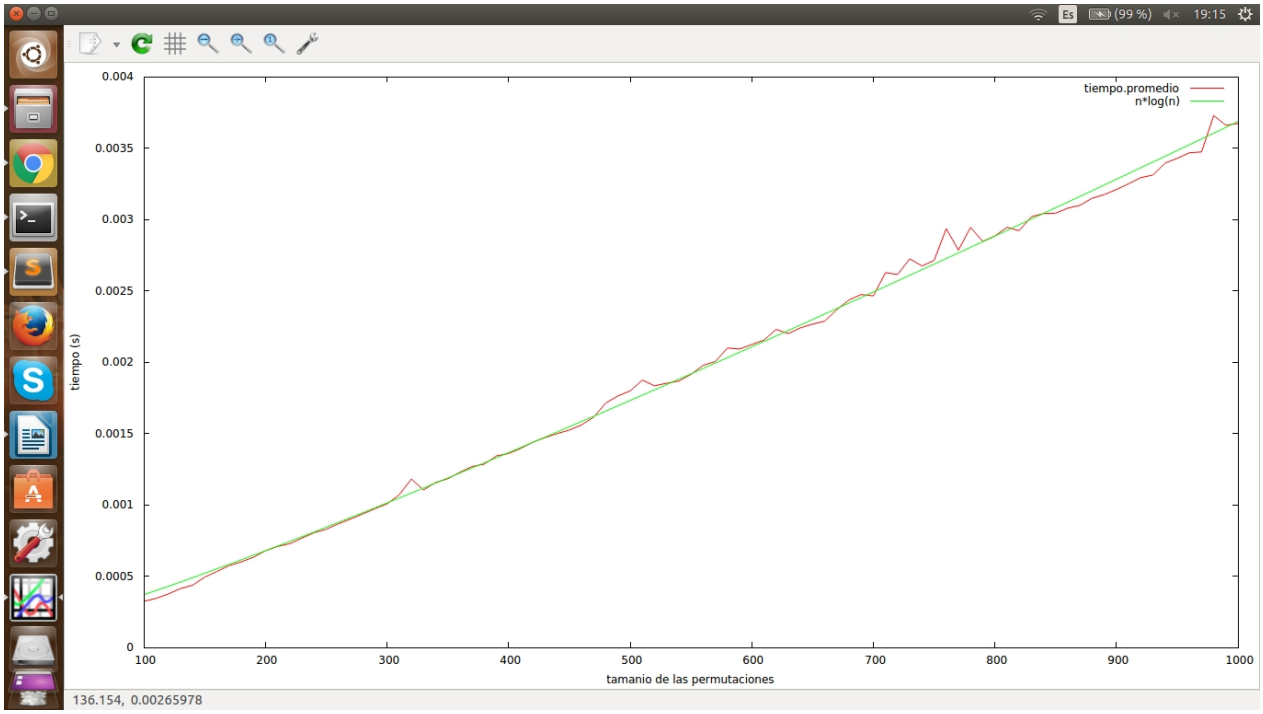
Ejercicio 1

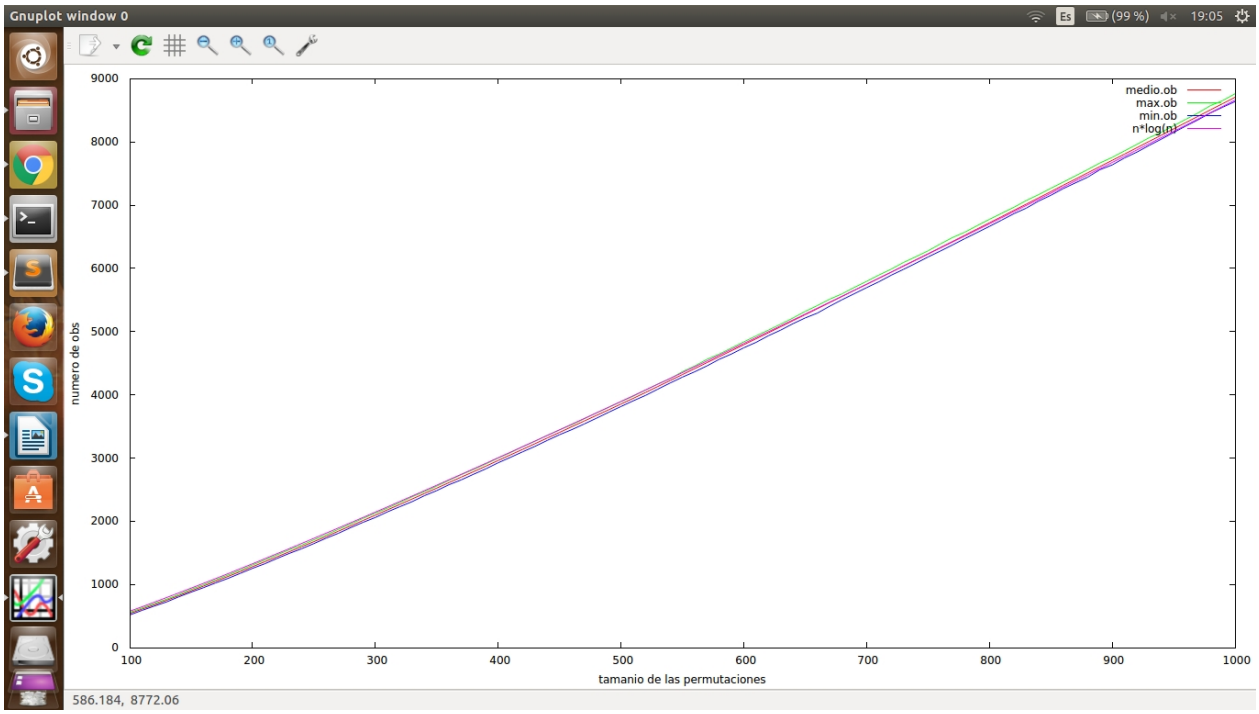
En este primer ejercicio, creamos un programa que a la entrada recibe el tamaño de la permutación a generar, y que utilizando el algoritmo de ordenación MergeSort, ordena la tabla y la imprime. En la imagen se puede ver un ejemplo de ejecución con una permutación de tamaño 20.

Ejercicio 2



En el segundo ejercicio, ponemos a prueba el algoritmo de ordenación MergeSort. En la primera imagen se puede ver la ejecución en la terminal del programa, sin pérdidas de memoria y con salida correcta.





La primera gráfica representa el tiempo promedio de ejecución, y la segunda el número máximo, promedio y mínimo de operaciones básicas del algoritmo.En estas 2 gráficas se puede apreciar que los valores obtenidos experimentalmente se aproximan mucho a los obtenidos teóricamente (n*log(n)) sin picos inusuales significativos.

Ejercicio 3

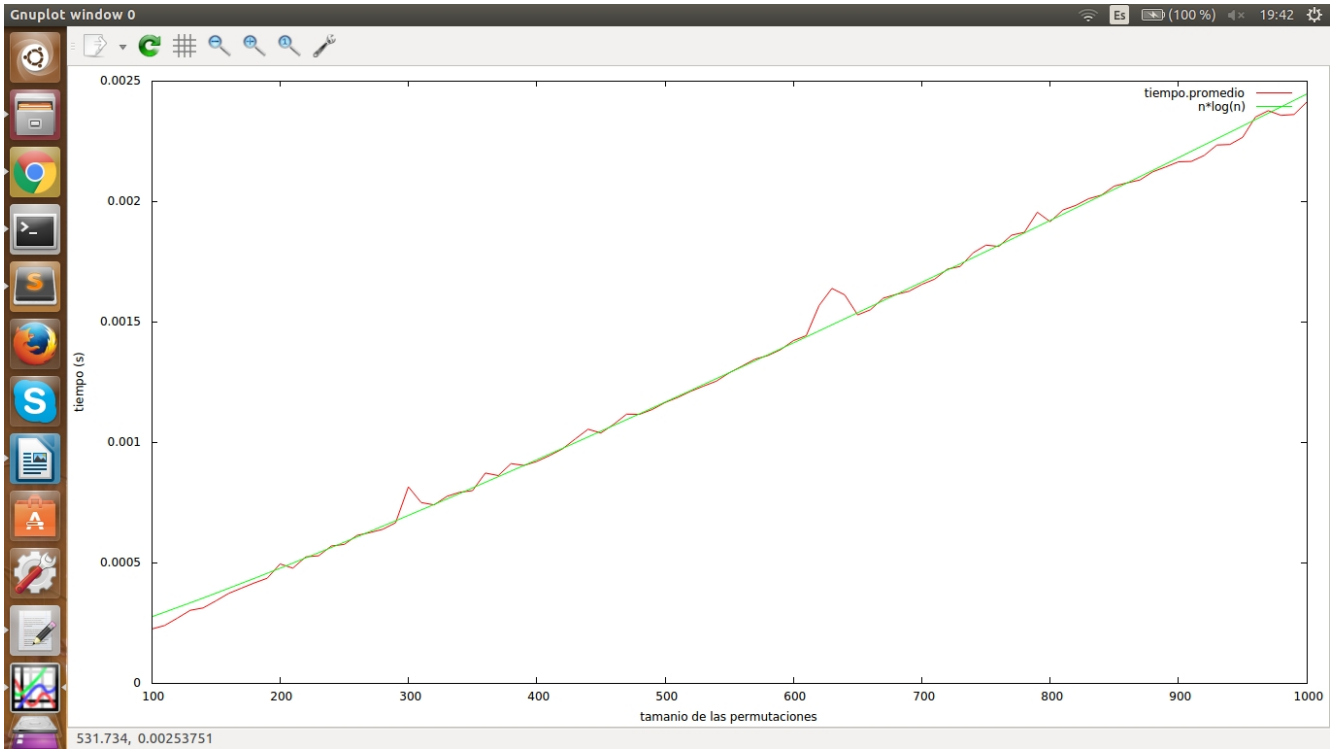
```
usuario@usuario-Satellite-Pro-A50-C:~/segundo/ANAL/practica2$ valgrind --leak-check=full ./ejercicio3 -tamanio 25
==2361== Memcheck, a memory error detector
==2361== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==2361== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==2361== Command: ./ejercicio3 -tamanio 25
==2361==
Practica numero 2, apartado 1
Realizada por: Alfonso Villar y Victor Garcia
Grupo: 12
1       2       3       4       5       6       7       8       9       10      11      12      13      14      15      16      17      18      1
9       20      21      22      23      24      25
==2361==
==2361== HEAP SUMMARY:
==2361==     in use at exit: 0 bytes in 0 blocks
==2361==   total heap usage: 21 allocs, 21 frees, 180 bytes allocated
==2361==
==2361== All heap blocks were freed -- no leaks are possible
==2361==
==2361== For counts of detected and suppressed errors, rerun with: -v
==2361== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
usuario@usuario-Satellite-Pro-A50-C:~/segundo/ANAL/practica2$
```

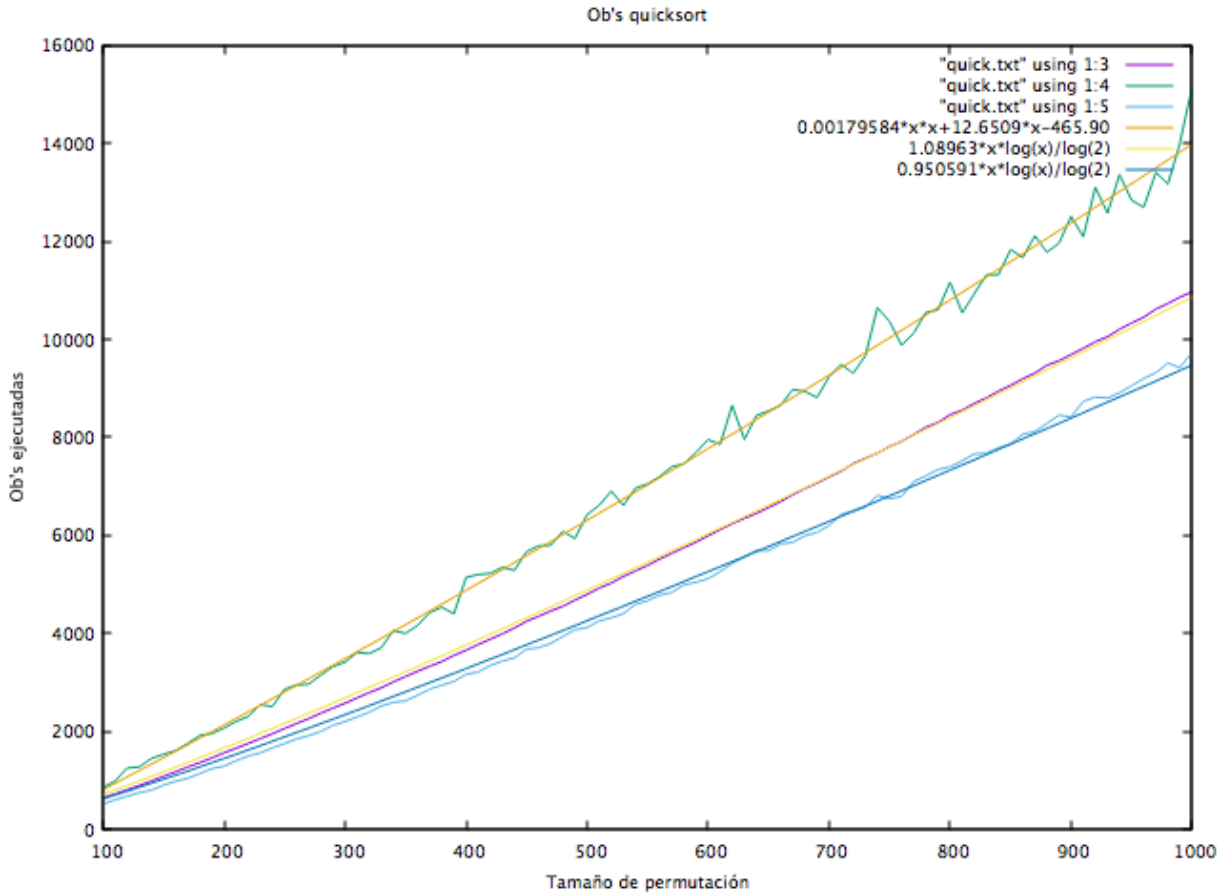
En este ejercicio, creamos un programa que a la entrada recibe el tamaño de la permutación a generar, y que utilizando el algoritmo de ordenación QuickSort, ordena la tabla y la imprime. En la imagen se puede ver un ejemplo de ejecución con una permutación de tamaño 25.

Ejercicio 4

```
usuario@usuario-Satellite-Pro-A50-C:~/segundo/ANAL/practica2$ valgrind --leak-check=full ./ejercicio4 -num_min 100 -num_max 1000 -incr 10 -numP
 1000 -fichSalida quick.txt
==16840== Memcheck, a memory error detector
==16840== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==16840== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==16840== Command: ./ejercicio4 -num_min 100 -num_max 1000 -incr 10 -numP 1000 -fichSalida quick.txt
==16840==
Practica numero 2, apartado 4
Realizada por: Alfonso Villar y Victor Garcia
Grupo: 12
Salida correcta
==16840==
==16840== HEAP SUMMARY:
==16840==     in use at exit: 0 bytes in 0 blocks
==16840==   total heap usage: 41,784,081 allocs, 41,784,081 frees, 367,704,144 bytes allocated
==16840==
==16840== All heap blocks were freed -- no leaks are possible
==16840==
==16840== For counts of detected and suppressed errors, rerun with: -v
==16840== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
usuario@usuario-Satellite-Pro-A50-C:~/segundo/ANAL/practica2$ gnuplot
```

En el cuarto ejercicio, ponemos a prueba el algoritmo de ordenación QuickSort. En la primera imagen se puede ver la ejecución en la terminal del programa, sin pérdidas de memoria y con salida correcta.


Gnuplot window 0
tiempo.promedio
n*log(n)
tiempo (s)
tamanio de las permutaciones
0.0025
0.002
0.0015
0.001
0.0005
0
100
200
300
400
500
600
700
800
900
1000
531.734, 0.00253751


Ob's quicksort
"quick.txt" using 1:3
"quick.txt" using 1:4
"quick.txt" using 1:5
0.00179584*x*x+12.6509*x-465.90
1.08963*x*log(x)/log(2)
0.950591*x*log(x)/log(2)
Ob's ejecutadas
Tamaño de permutación
16000
14000
12000
10000
8000
6000
4000
2000
0
100
200
300
400
500
600
700
800
900
1000

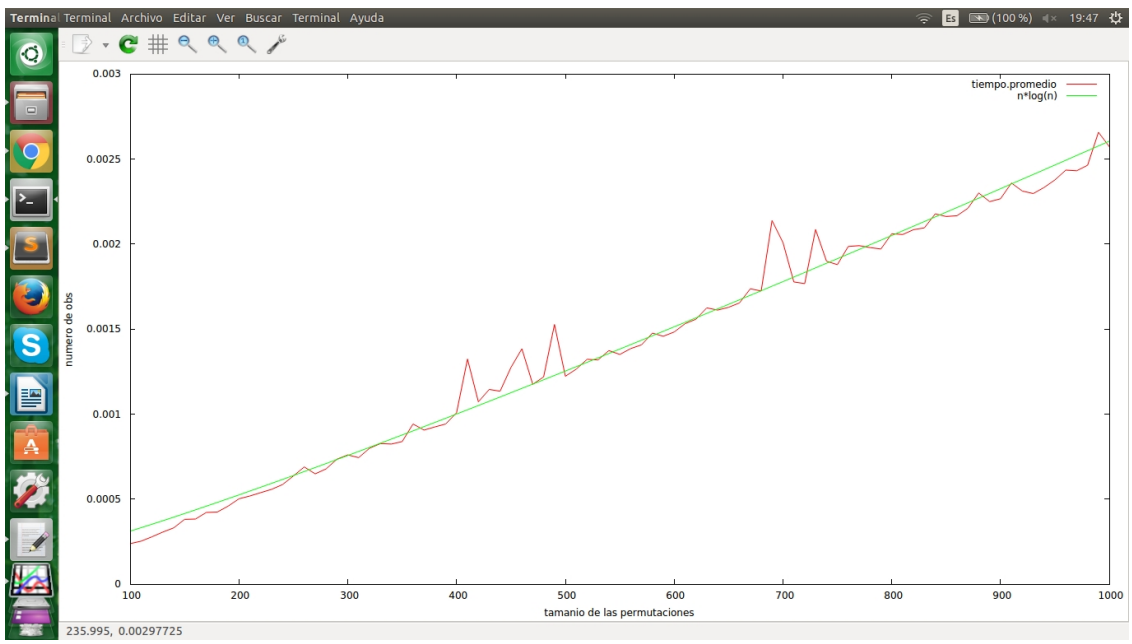
La primera gráfica representa el tiempo promedio de ejecución, y la segunda el número máximo, promedio y mínimo de operaciones básicas del algoritmo. En estas 2 gráficas se puede apreciar que los valores obtenidos experimentalmente se aproximan mucho a los obtenidos teóricamente (n*log(n) en tiempo medio de ejecución y ob's medias y para el caso ob's max se asemeja a n*n) sin picos inusuales significativos.

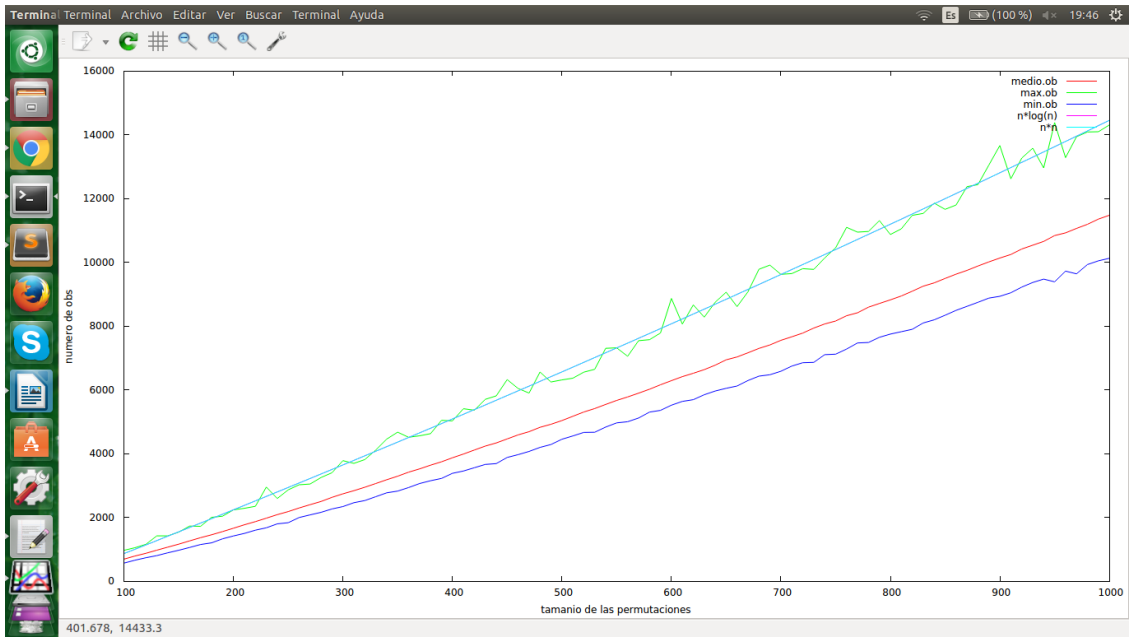
Nota: En la leyenda, aquellas gráficas con valores “extraños” son las aproximaciones teóricas del algoritmo.

Ejercicio 5

```
usuario@usuario-Satellite-Pro-A50-C:~/segundo/ANAL/practica2$ valgrind --leak-check=full ./ejercicio5 -num_min 100 -num_max 1000 -incr 10 -numP
 1000 -fichSalida quickavg.txt
==18352== Memcheck, a memory error detector
==18352== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==18352== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==18352== Command: ./ejercicio5 -num_min 100 -num_max 1000 -incr 10 -numP 1000 -fichSalida quickavg.txt
==18352==
Practica numero 2, apartado 5
Realizada por: Alfonso Villar y Victor Garcia
Grupo: 12
Salida correcta
==18352==
==18352== HEAP SUMMARY:
==18352==     in use at exit: 0 bytes in 0 blocks
==18352==   total heap usage: 41,787,997 allocs, 41,787,997 frees, 367,719,808 bytes allocated
==18352==
==18352== All heap blocks were freed -- no leaks are possible
==18352==
==18352== For counts of detected and suppressed errors, rerun with: -v
==18352== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
usuario@usuario-Satellite-Pro-A50-C:~/segundo/ANAL/practica2$ gnuplot
```

En el quinto ejercicio, ponemos a prueba el algoritmo de ordenación QuickSort_avg, que difiere de QuickSort en la selección del pivote (ver en objetivos). En la primera imagen se puede ver la ejecución en la terminal del programa, sin pérdidas de memoria y con salida correcta.



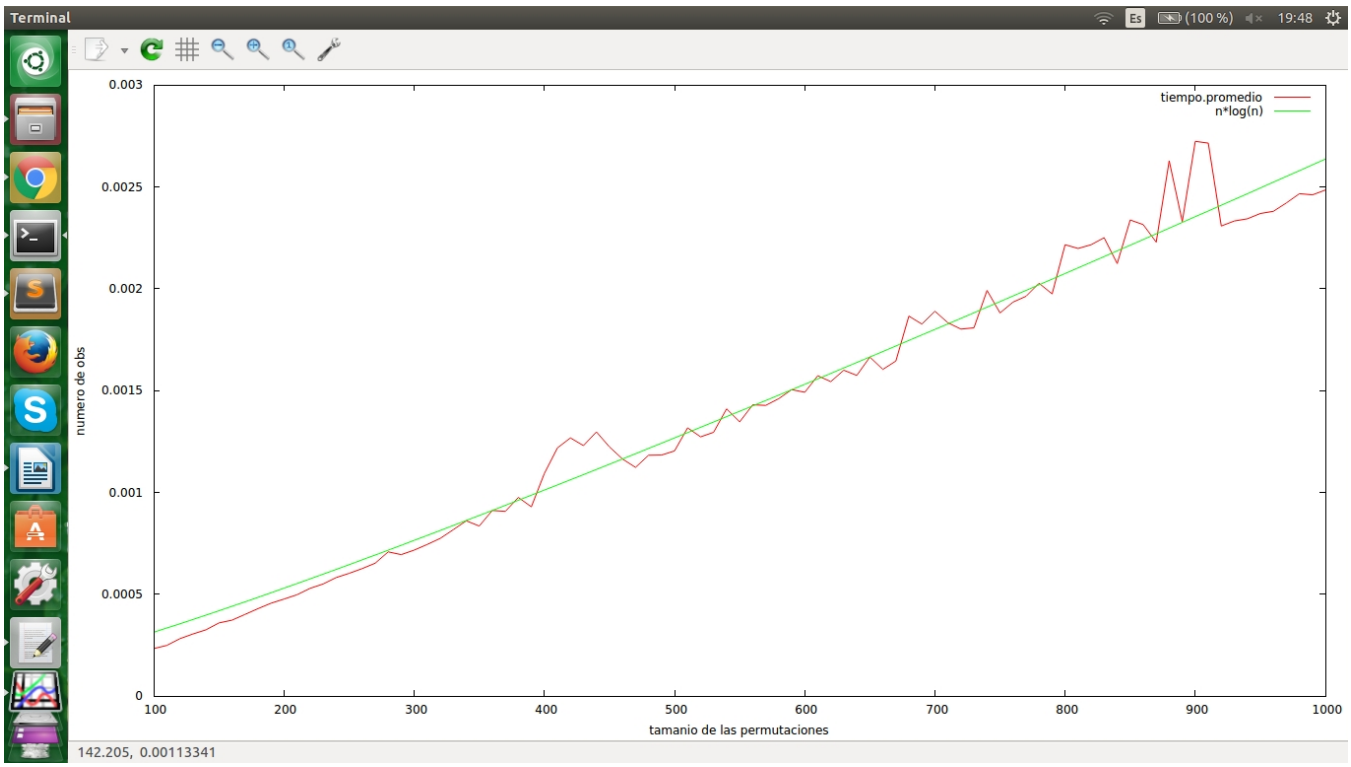


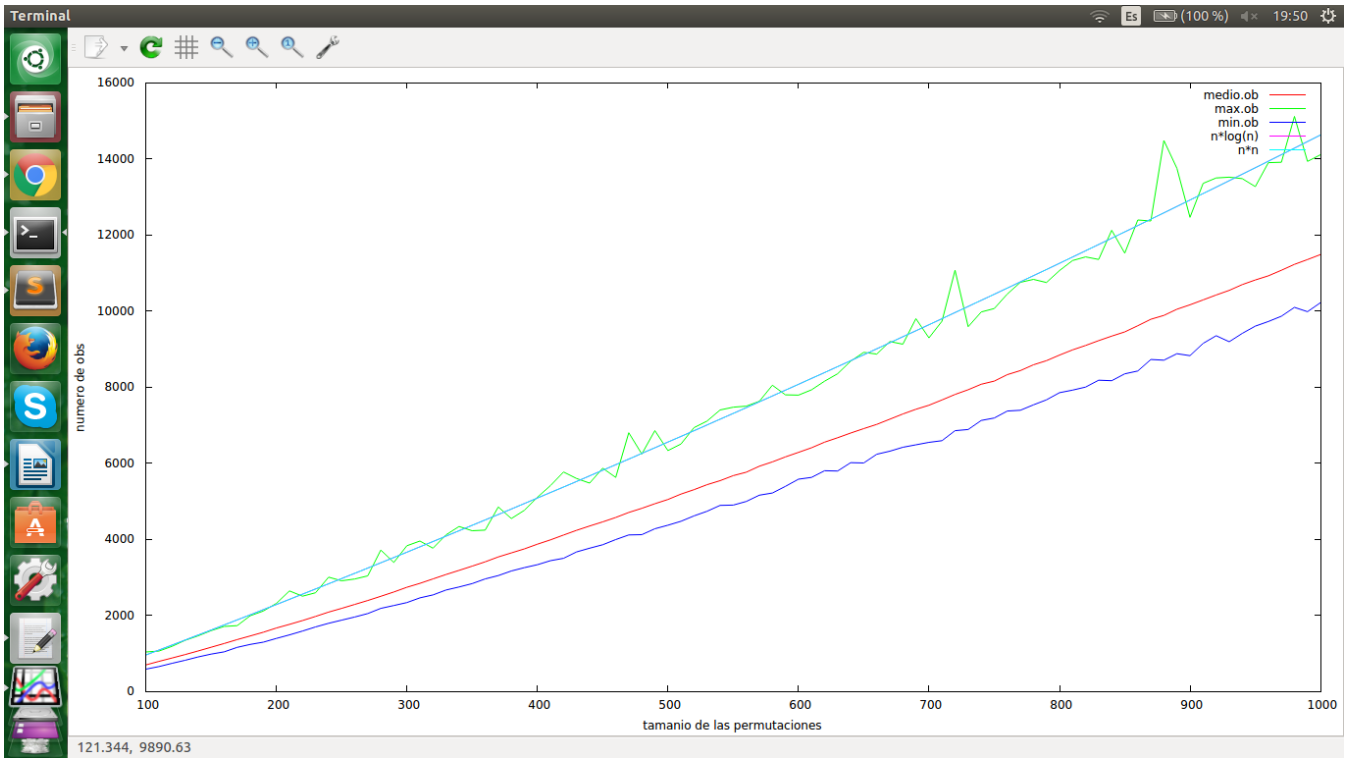
La primera gráfica representa el tiempo promedio de ejecución, y la segunda el número máximo, promedio y mínimo de operaciones básicas del algoritmo. En estas 2 gráficas se puede apreciar que los valores obtenidos experimentalmente se aproximan mucho a los obtenidos teóricamente (n*log(n) en tiempo de ejecucion y en el caso de ob's medias y n*n en la de max ob's) sin picos inusuales significativos

Ejercicio 6

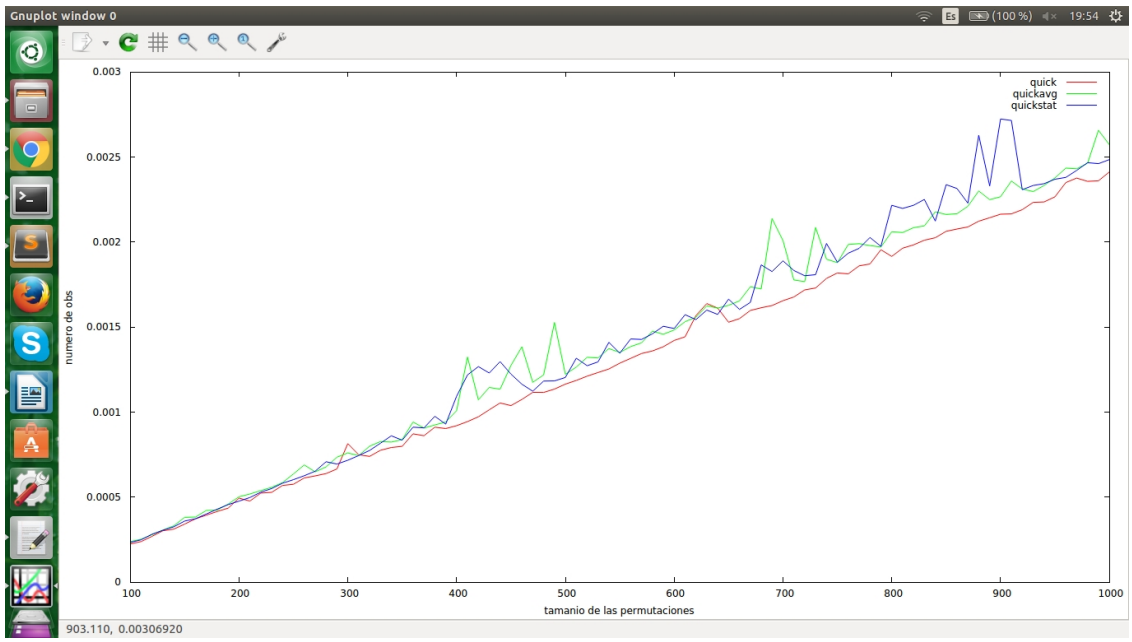
```
usuario@usuario-Satellite-Pro-A50-C:~/segundo/ANAL/practica2$ valgrind --leak-check=full ./ejercicio6 -num_min 100 -num_max 1000 -incr 10 -numP
 1000 -fichSalida quickstat.txt
==20342== Memcheck, a memory error detector
==20342== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==20342== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==20342== Command: ./ejercicio6 -num_min 100 -num_max 1000 -incr 10 -numP 1000 -fichSalida quickstat.txt
==20342==
Practica numero 2, apartado 6
Realizada por: Alfonso Villar y Victor Garcia
Grupo: 12
Salida correcta
==20342==
==20342== HEAP SUMMARY:
==20342==     in use at exit: 0 bytes in 0 blocks
==20342==   total heap usage: 41,782,910 allocs, 41,782,910 frees, 367,699,460 bytes allocated
==20342==
==20342== All heap blocks were freed -- no leaks are possible
==20342==
==20342== For counts of detected and suppressed errors, rerun with: -v
==20342== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
usuario@usuario-Satellite-Pro-A50-C:~/segundo/ANAL/practica2$ gnuplot
```

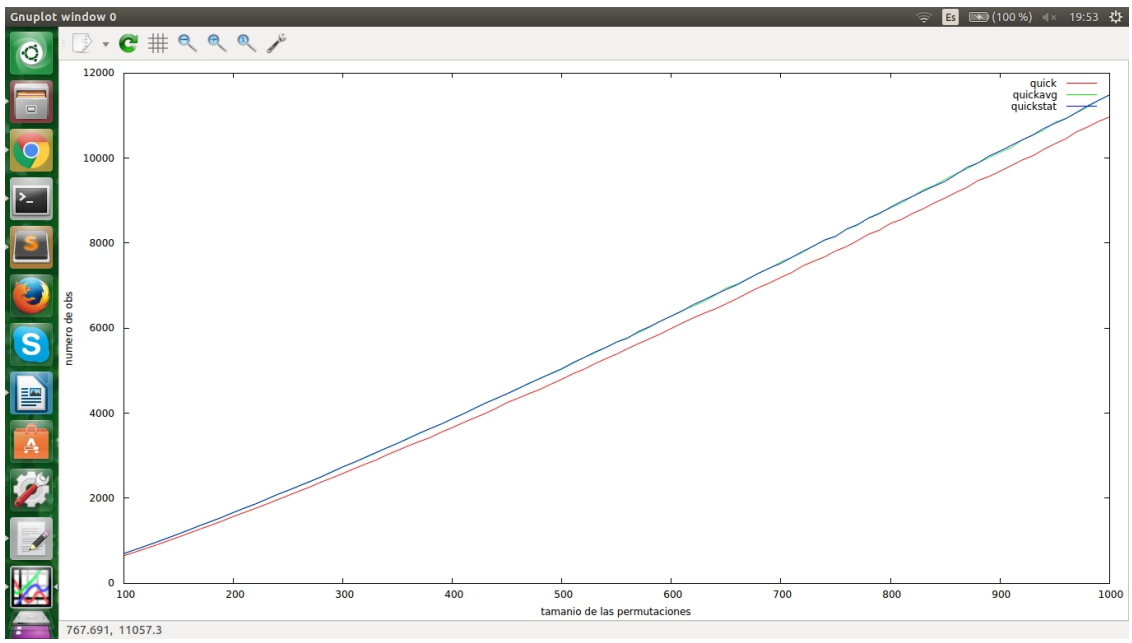
En el sexto ejercicio, ponemos a prueba el algoritmo de ordenación QuickSort_stat, que difiere de QuickSort en la selección del pivote (ver en objetivos). En la primera imagen se puede ver la ejecución en la terminal del programa, sin pérdidas de memoria y con salida correcta.


Terminal
(100 %)
19:48
tiempo.promedio
n*log(n)
numero de obs
tamanio de las permutaciones
0.003
0.0025
0.002
0.0015
0.001
0.0005
0
100
200
300
400
500
600
700
800
900
1000
142.205, 0.00113341


Terminal
(100 %)
19:50
medio.ob
max.ob
min.ob
n*log(n)
n*n
numero de obs
tamanio de las permutaciones
16000
14000
12000
10000
8000
6000
4000
2000
0
100
200
300
400
500
600
700
800
900
1000
121.344, 9890.63

La primera gráfica representa el tiempo promedio de ejecución, y la segunda el número máximo, promedio y mínimo de operaciones básicas del algoritmo. En estas 2 gráficas se puede apreciar que los valores obtenidos experimentalmente se aproximan mucho a los obtenidos teóricamente (n*log(n) en tiempo de ejecucion y en el caso de ob's medias y n*n en la de max ob's) sin picos inusuales significativos





La primera gráfica representa los tiempos promedios de QuickSort, QuickSort_avg y QuickSort_stat, y la segunda gráfica representa las operaciones básicas medias de los algoritmos anteriores. Como se aprecian en estas gráficas, los tiempos son muy semejantes entre si como cabía esperar debido a que sus codigos son muy semejantes al igual que su comparación de ob's. (ERROR EN LA 2NDA GRAFICA: las leyendas de quick y quickstat están intercambiadas)

6. Respuesta a las preguntas teóricas.

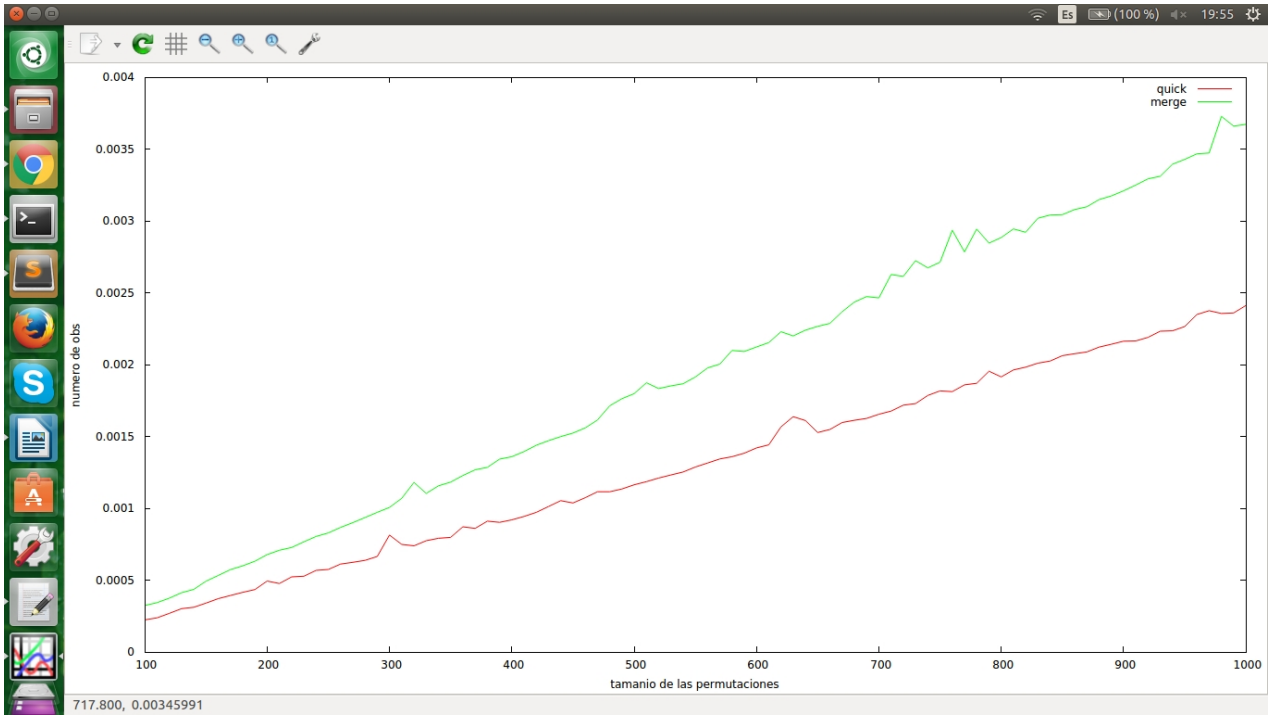
6.1
El rendimiento experimental prácticamente es igual al dado teóricamente. Los picos que tenemos en la gráfica son debidos a que, para eliminarlo, habría que realizar un número mayor de permutaciones, para que se ajustasen más

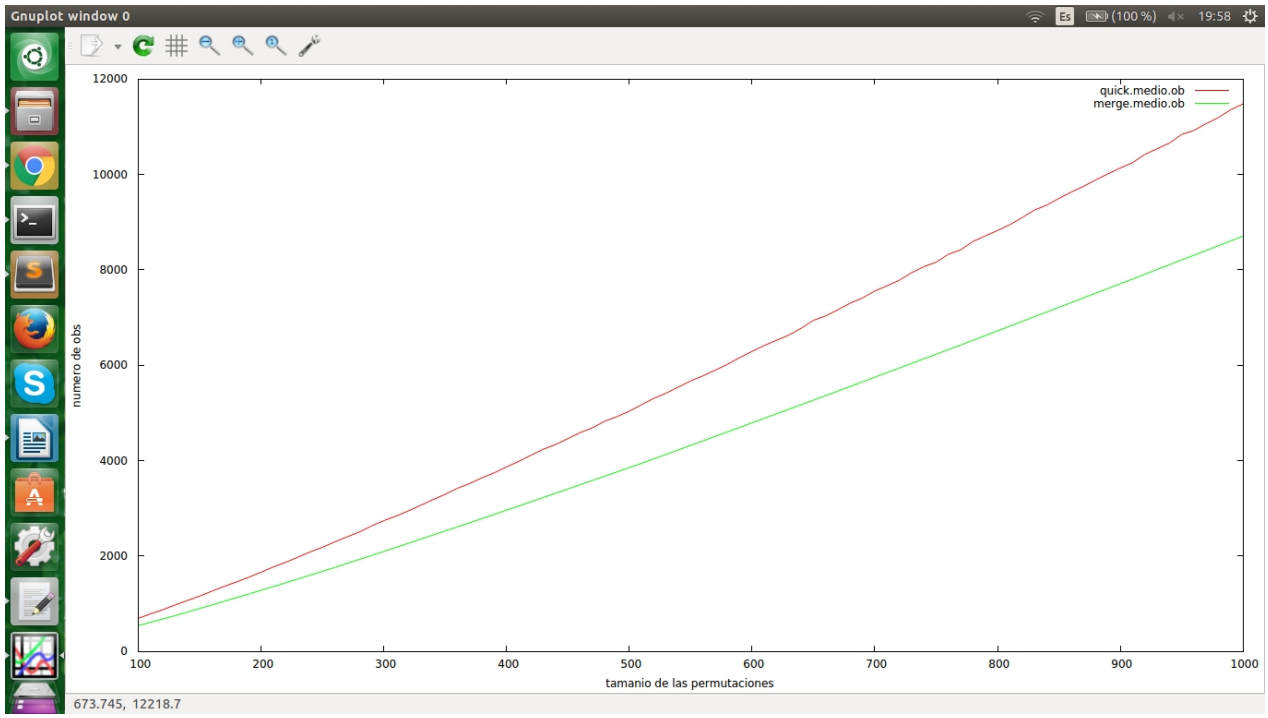
6.2
Teóricamente los resultados al seleccionar el pivote no deberían suponer diferencia alguna significativa, aunque cabe destacar que la versión QuickSort_stat a cuanto mayor es la permutación más significativa es la diferrencia de tiempo con las demás debido a que al seleccionar el pivote tal y como lo elige equilibra las tablas siguientes lo que provoca que se reduzca ligeramente el tiempo de ejecucción

6.3
MergeSort
W: $Wms(N) \leq N*lg(N)+O(N)$
B: $Bms(N) \geq (1/2)Nlg(N)$
QuickSort
W: $Wqs(N) = N2 /2-N/2$
B: $Bqs(N) = O (N*lg(N)$

Pues podríamos utilizar en vez de una permutación aleatoria, para la tabla usaríamos la permutación que siempre tardase más en ejecutar (por tanto sería el caso peor). Al igual lo realizariamos para el caso mejor y para el caso medio, realizariamos todas las posibles permutaciones y con ello calcularíamos el tiempo medio exacto.

6.4
La primera gráfica representa los tiempos promedios de MergeSort y QuickSort, y la segunda gráfica representa las operaciones básicas promedio de ambos algoritmos descritos. QuickSort es más eficiente empíricamente y esto es debido a que la gestión de memoria de MergeSort provoca que tarde más tiempo que QuickSort, a pesar de que MergeSort es más estable que QuickSort y que realiza menos ob's que QuicSort, ya que este último puede llegar a tiempos de n^2.


quick
merge
numero de obs
tamanio de las permutaciones
0.004
0.0035
0.003
0.0025
0.002
0.0015
0.001
0.0005
0
100
200
300
400
500
600
700
800
900
1000
717.800, 0.00345991


Gnuplot window 0
quick.medio.ob
merge.medio.ob
numero de obs
tamanio de las permutaciones
12000
10000
8000
6000
4000
2000
0
100
200
300
400
500
600
700
800
900
1000
673.745, 12218.7

7. Conclusiones finales

En la realización de esta práctica cabe destacar el hecho de que, al implementar nosotros mismos ambos algoritmos de ordenación, hemos podido comprender con mayor profundidad el funcionamiento de estos, además de que el hecho de implementar Quicksort cambiando la posición del pivote nos permitió ver que, en lo que respecta al tiempo de ejecución, independientemente de la posición de éste, no hay diferencias apreciables. Además, ambos algoritmos, de coste medio N*log(N), son, hasta la fecha, los más rápidos (en general), por lo que su estudio tiene gran valor.