Análisis de Algoritmos 2015/2016

Práctica 1

Óscar Gómez

Jose Ignacio Gómez

Pareja 4

Grupo 1201

|  |  |  |  |
| --- | --- | --- | --- |
| Código | Gráficas | Memoria | Total |
|  |  |  |  |

**1. Introducción.**

En esta práctica hemos fijado los fundamentos para las siguientes prácticas, generando librerías que usaremos más adelante, como la generación de números aleatorios, permutaciones o medición de tiempo de ejecución de un algoritmo.

**2. Objetivos**

2.1 Apartado 1

En el primer apartado realizaremos un algoritmo que genere números aleatorios entre dos números dados, siendo el resultado lo mas equiprobable posible.

2.2 Apartado 2

En este apartado buscamos un algoritmo que cree una lista de n números ordenados aleatoriamente.

2.3 Apartado 3

Ahora ampliaremos el alcance del apartado anterior, realizaremos un algoritmo que genere un número dado de permutaciones.

2.4 Apartado 4

En este apartado implementaremos el código para el algoritmo de ordenación InsertSort

2.5 Apartado 5

Llegado este punto buscaremos una manera de comprobar el tiempo de ejecución de nuestros algoritmos.

**3. Herramientas y metodología**

Para toda la práctica hemos utilizado el entorno Linux (Debian). Hemos utilizado como herramienta principal Sublime Text para la edición de código. Como control de versiones usamos git. En todo momento comprobamos los posibles errores de memoria mediante Valgrind.

3.1 Apartado 1

En este apartado decidimos utilizar un algoritmo basado en uno encontrado en la bibliografía recomendada, para ello utilizamos la función rand(), incluida en stdlib.h. Para comprobar la equiprobabilidad de los números aleatorios utilizamos Gnuplot.

3.2 Apartado 2

En este apartado decidimos utilizar la función anterior para crear la permutación. Usamos un algoritmo de “desordenación” donde intercambiamos cada posición con una posición aleatoria buscada utilizando la función anterior.

3.3 Apartado 3

Para los requisitos definidos buscamos un algoritmo que genere un número determinado de permutaciones. Para ello definimos un bucle llamando a la función anterior para realizarlo.

3.4 Apartado 4

Para este apartado buscamos en la bibliografía recomendada el algoritmo del método de ordenación InsertSort, implementándolo para conseguir la misma funcionalidad.

3.5 Apartado 5

Para este apartado utilizaremos la función clock(), incluida en time.h para realizar nuestra función. Por otra parte para escribir el resultado en un fichero decidimos llamar a la función remove() para eliminar el fichero en caso de su existencia ya que así conseguiremos un resultado limpio, sin interferencia de los resultados de las previas ejecuciones.

**4. Código fuente**

4.1 Apartado 1

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* Funcion: swap Fecha: 23/09/2016 \*/

/\* Autores: Óscar Gómez, Jose Ignacio Gómez \*/

/\* \*/

/\* Rutina que intercambia el contenido de dos \*/

/\* posiciones de un Array \*/

/\* \*/

/\* Entrada: \*/

/\* int n: Numero de elementos de la permutación \*/

/\* \*/

/\* Salida: \*/

/\* int \*: puntero a un array de enteros \*/

/\* que contiene a la permutacion \*/

/\* o NULL en caso de error \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int swap(int\* a, int\* b){

int aux;

if(!a || !b){

return ERR;

}

aux = \*(a);

\*(a) = \*(b);

\*(b) = aux;

return OK;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* Funcion: aleat\_num Fecha: 23/09/2016 \*/

/\* Autores: Óscar Gómez, Jose Ignacio Gómez \*/

/\* \*/

/\* 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){

float random;

if(inf < 0 || sup < 0 || sup < inf){

fprintf(stderr, "Error en la entrada de argumentos a aleat\_num");

return -1; /\*ERROR\*/

}

random = rand();

random /= RAND\_MAX;

random \*= (sup-inf+1);

random += inf;

if(random > sup){

random = sup;

}

return (int)random;

}

4.2 Apartado 2

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* Funcion: genera\_perm Fecha: 23/09/2016 \*/

/\* Autores: Óscar Gómez, Jose Ignacio Gómez \*/

/\* \*/

/\* Rutina que genera una permutacion aleatoria \*/

/\* \*/

/\* Entrada: \*/

/\* int n: Numero de elementos de la permutación \*/

/\* \*/

/\* 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 i, random;

int\* perm = NULL;;

if(n<1){

fprintf(stderr, "Error de entrada de argumentos de genera\_perm");

return NULL;

}

perm = (int \*)malloc(n \* sizeof(int));

if(!perm){

return NULL;

}

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

perm[i] = i;

}

for(i = 0; i < n-1; i++){ /\*Llegamos hasta n-1 porque la última posición

del array solo se puede permutar consigo misma\*/

random= aleat\_num(i+1, n-1);

if(swap(&(perm[i]), &(perm[random])) == ERR){

fprintf(stderr, "Error");

free(perm);

return NULL;

}

}

return perm;

}

4.3 Apartado 3

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* Funcion: genera\_permutaciones Fecha: 23/09/2016 \*/

/\* Autores: Óscar Gómez, Jose Ignacio Gómez \*/

/\* \*/

/\* Funcion que genera n\_perms permutaciones \*/

/\* aleatorias de tamanio elementos \*/

/\* \*/

/\* Entrada: \*/

/\* int n\_perms: Numero de permutaciones \*/

/\* int tamanio: 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 tamanio)

{

int i;

int \*\*perms;

perms = (int \*\*) malloc (n\_perms \* sizeof(int \*));

if(!perms){

return NULL;

}

for (i = 0; i < n\_perms; i++){

perms[i] = genera\_perm(tamanio);

}

return perms;

}

4.4 Apartado 4

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* Funcion: InsertSort Fecha: 30/09/2016 \*/

/\* Función de ordenacion para tablas de menor a \*/

/\* mayor \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int InsertSort(int\* tabla, int ip, int iu)

{

int time = 0;

int i, j, aux;

for (i = ip + 1; i <= iu; i++) {

aux = tabla[i];

j = i-1;

while (j >= ip && tabla[j] > aux){

tabla[j+1] = tabla[j];

j--;

time++;

}

time++;

tabla[j+1] = aux;

}

return time;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* Funcion: InsertSortInv Fecha: 30/09/2016 \*/

/\* Función de ordenacion para tablas de mayor a \*/

/\* menor \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int InsertSortInv(int\* tabla, int ip, int iu)

{

int time = 0;

int i, j, aux;

for (i = ip + 1; i <= iu; i++) {

aux = tabla[i];

j = i-1;

while (j >= ip && tabla[j] < aux){

tabla[j+1] = tabla[j];

j--;

time++;

}

time++;

tabla[j+1] = aux;

}

return time;

}

4.5 Apartado 5

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* Funcion: tiempo\_medio\_ordenacion Fecha:2/10/2016\*/

/\* \*/

/\* Rutina que guarda en una escritura PTIEMPO los \*/

/\* datos necesarios para el análisis temporal \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

short tiempo\_medio\_ordenacion(pfunc\_ordena metodo,

int n\_perms,

int tamanio,

PTIEMPO ptiempo)

{

int \*\*perms;

int i; /\*ob = operación basica\*/

double suma = 0, sumaob=0;

clock\_t start, end;

double time, ob; /\*time = tiempo fisico\*/

if(!ptiempo){

return -1;

}

ptiempo->n\_perms = n\_perms;

ptiempo->tamanio = tamanio;

ptiempo->max\_ob = 0;

ptiempo->min\_ob = INT\_MAX;

perms = genera\_permutaciones(n\_perms, tamanio);

if(!perms){

return -1;

}

for(i = 0; i < n\_perms; i++){

start = clock();

ob = metodo(perms[i], 0, tamanio - 1);

end = clock();

time = (double)(end-start);

sumaob += ob;

suma += time;

if(ob > ptiempo->max\_ob){

ptiempo->max\_ob = ob;

}

if (ob< ptiempo->min\_ob){

ptiempo->min\_ob = ob;

}

}

ptiempo->medio\_ob = (double)(sumaob/n\_perms);

ptiempo->tiempo = (double)(suma/n\_perms)/CLOCKS\_PER\_SEC;

for(i = 0; i < n\_perms; i++){

free(perms[i]);

}

free(perms);

return 0;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* Funcion: genera\_tiempos\_ordenacion Fecha: 2/10/2016\*/

/\* \*/

/\* Rutina usada para generar todos los tiempos de \*/

/\* ordenación para su posterior escritura en fichero \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

short genera\_tiempos\_ordenacion(pfunc\_ordena metodo, char\* fichero,

int num\_min, int num\_max,

int incr, int n\_perms)

{

int i;

PTIEMPO tiempo;

tiempo = (PTIEMPO) malloc (sizeof(TIEMPO));

remove(fichero);

for(i = num\_min; i <= num\_max; i += incr){

tiempo\_medio\_ordenacion(metodo, n\_perms, i, tiempo);

guarda\_tabla\_tiempos(fichero, tiempo, 1);

}

free(tiempo);

return 0;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* Funcion: guarda\_tabla\_tiempos Fecha:02/10/2016 \*/

/\* \*/

/\* Rutina que escribe en un fichero los datos \*/

/\* obtenidos \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

short guarda\_tabla\_tiempos(char\* fichero, PTIEMPO tiempo, int N)

{

FILE\* f = NULL;;

f = fopen(fichero, "a");

if(!f){

return -1;

}

fprintf(f, "%-20d%-20.1f%-20.1f%-20d%-20d\n", tiempo->tamanio, tiempo->tiempo, tiempo->medio\_ob, tiempo->max\_ob, tiempo->min\_ob);

fclose(f);

return 0;

}

**5. Resultados, Gráficas**

5.1 Apartado 1

Tras realizar la rutina realizamos un histograma, para ello realizamos un programa que siguiendo el siguiente código escribe en un fichero la frecuencia en la que se obtienen los números:

int main(){

FILE \*f = NULL;

int tabla[NUMEROS];

int i;

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

tabla[i] = 0;

}

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

tabla[aleat\_num(1, NUMEROS) - 1]++;

}

f = fopen("histograma.txt", "w");

if(!f){

return -1;

}

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

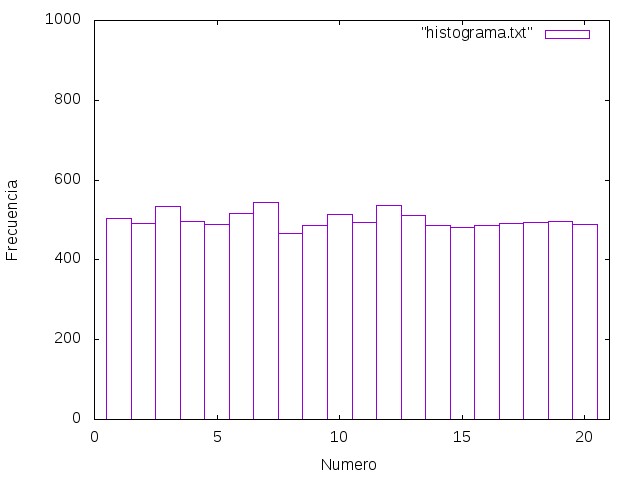
fprintf(f, "%d \t %d\n",i+1, tabla[i]);

}

fclose(f);

return 0;

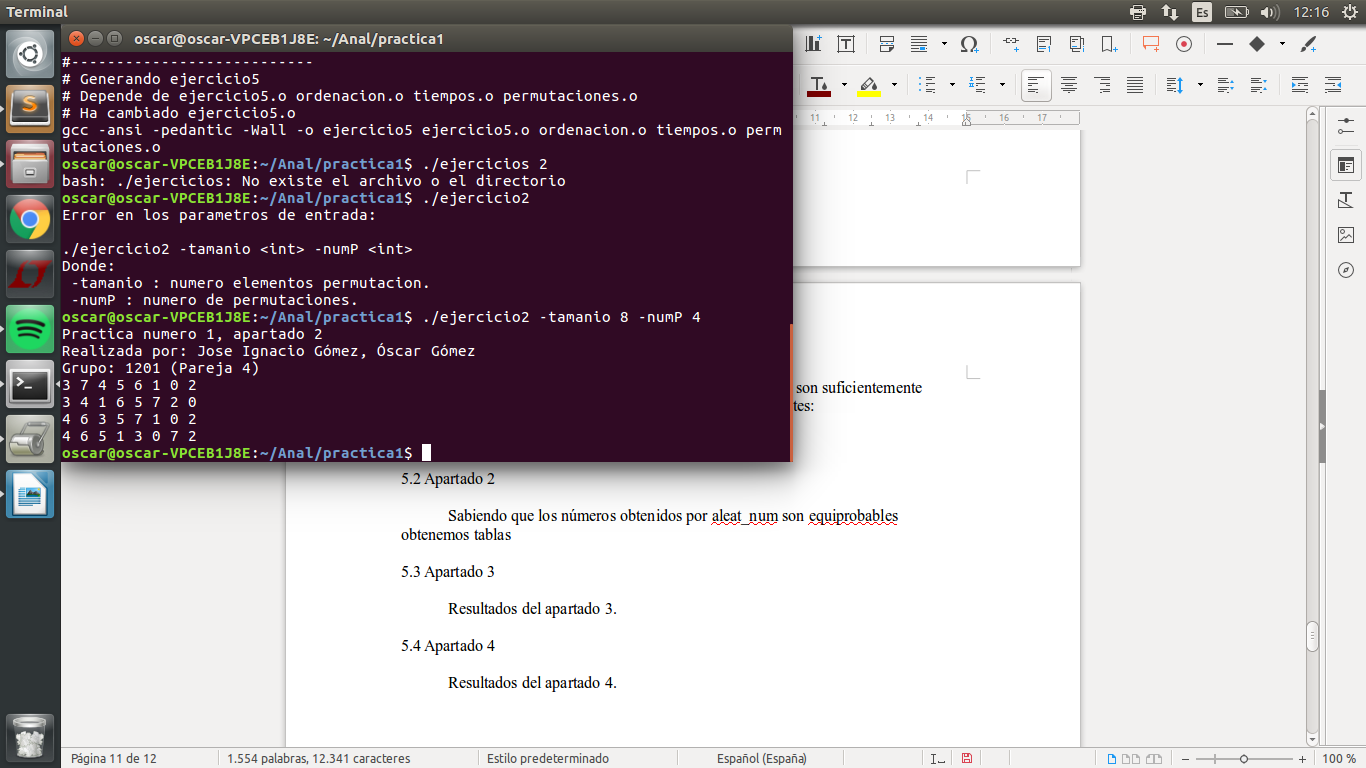
}



Tras analizar la gráfica creada por Gnuplot determinamos que los números son suficientemente equiprobables.

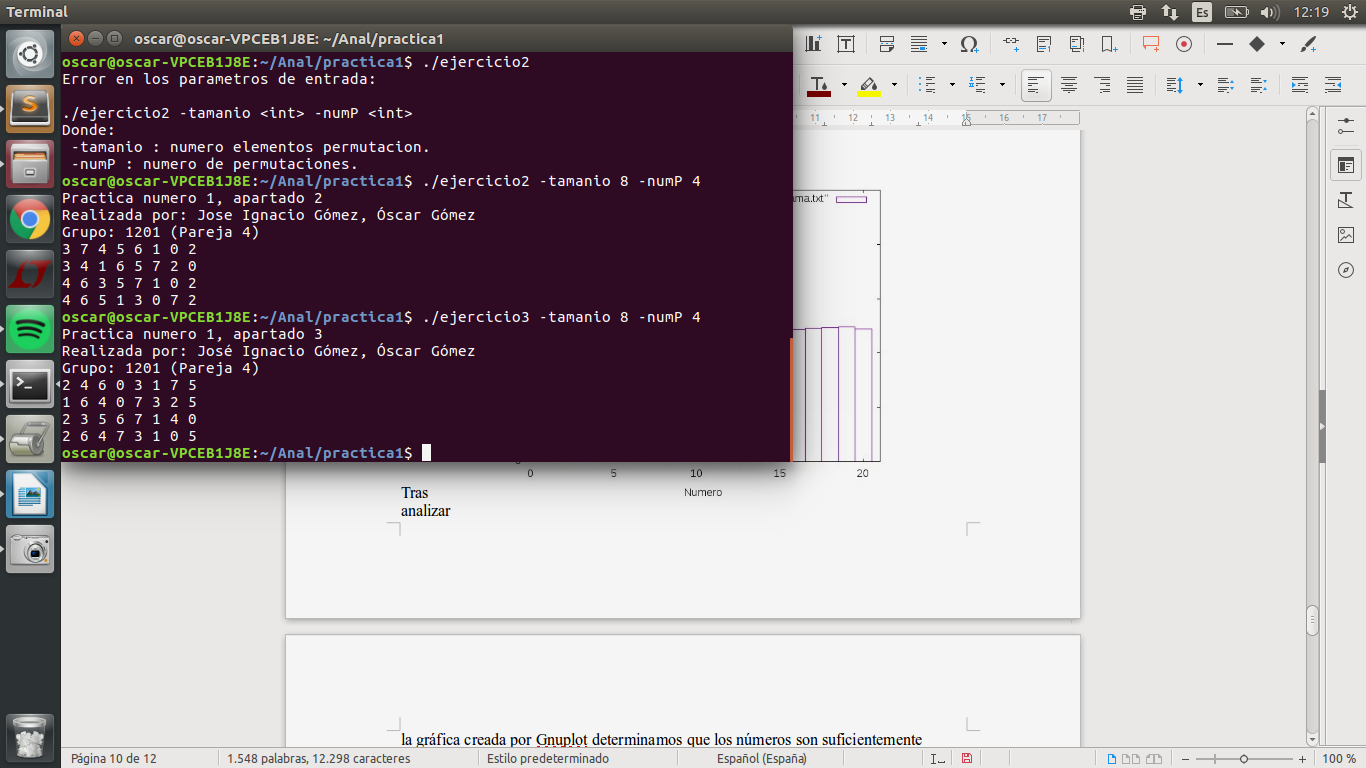
5.2 Apartado 2

Sabiendo que los números obtenidos por aleat\_num son equiprobables obtenemos arrays aleatorias como las siguientes:

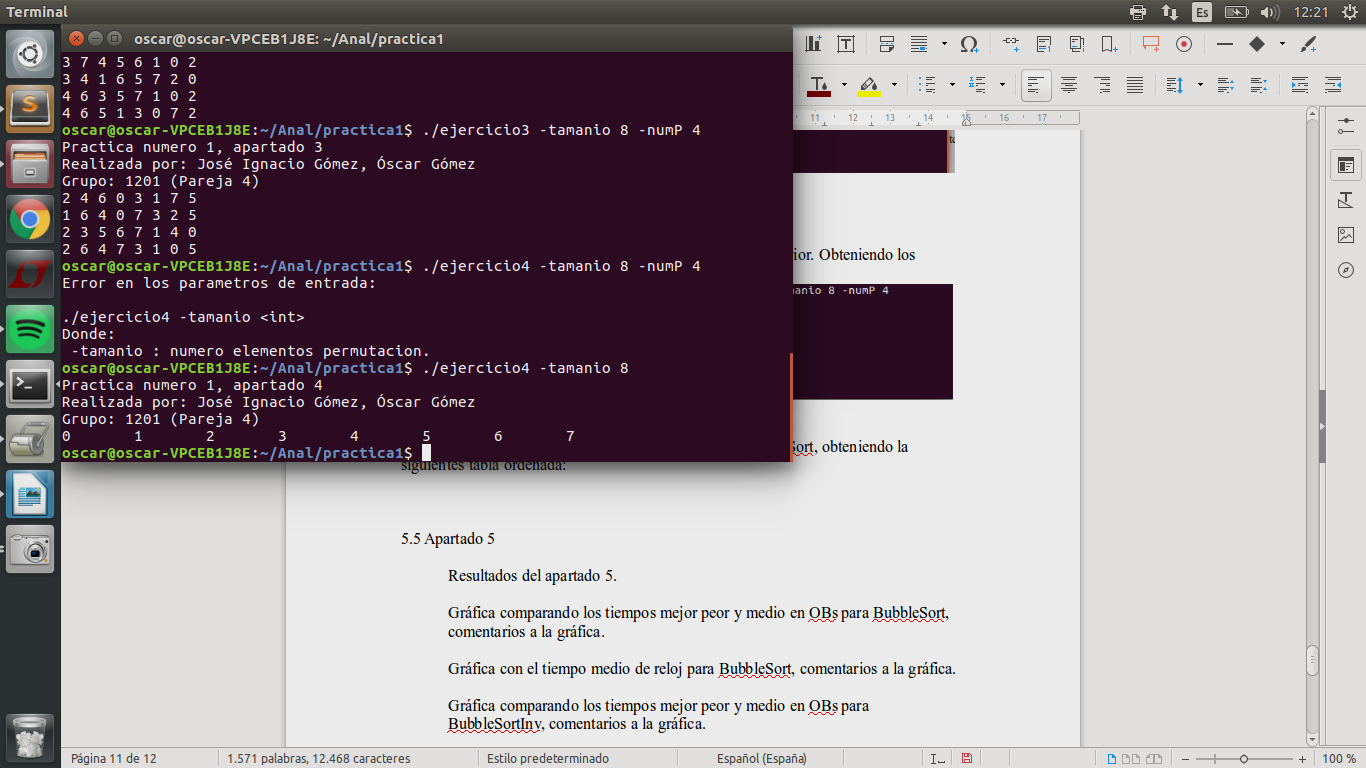


5.3 Apartado 3

En este ejercicio la rutina se asemeja al apartado anterior. Obteniendo los siguientes resultados:

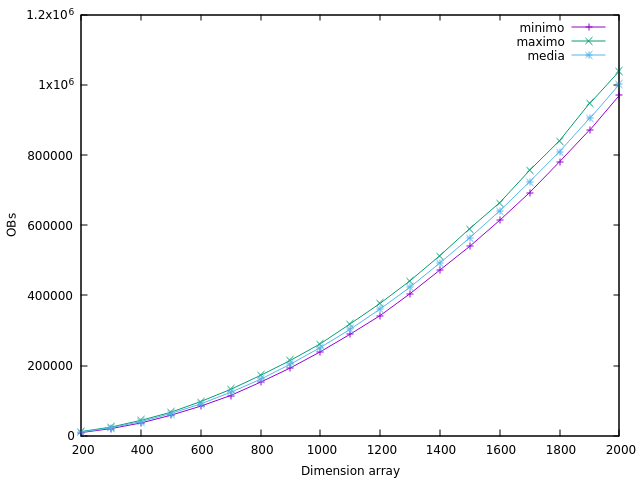


5.4 Apartado 4

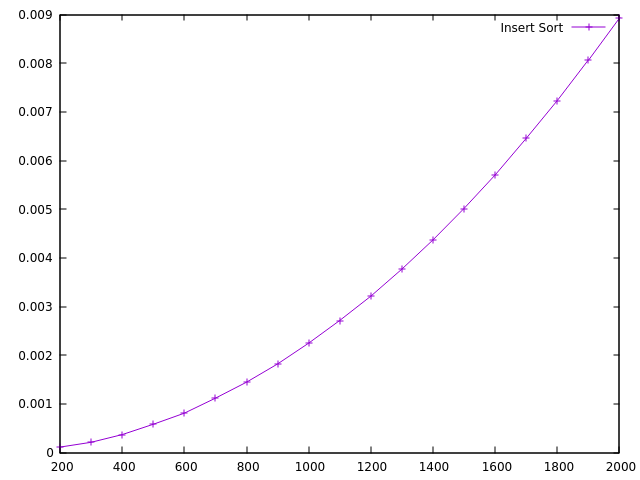
En este apartado ordenamos un array mediante InsertSort, obteniendo la siguiente tabla ordenada: 

5.5 Apartado 5

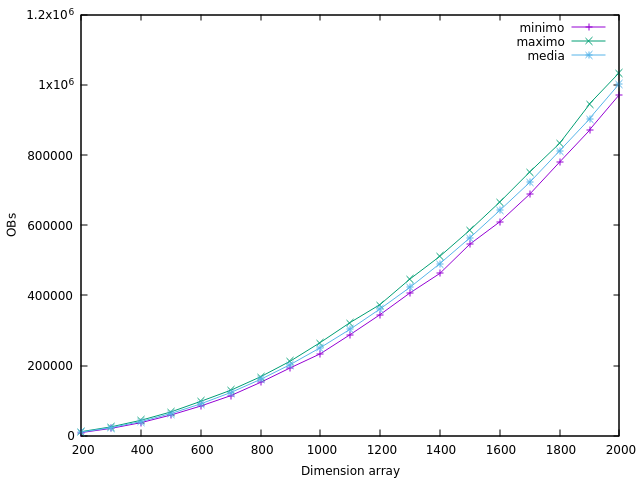
Tras ejecutar el programa y analizar los datos con Gnuplot obtenemos las siguientes gráficas:



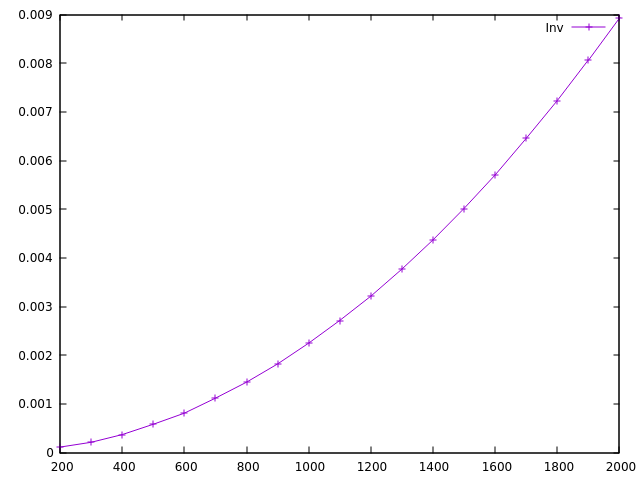
Se puede apreciar que la diferencia va aumentando según aumenta el tamaño del array, tomando un orden cuadrático



Se puede apreciar que el tiempo es cuadrático respecto a la dimensión del array, siendo el tiempo de ejecución con un array de tamaño 2000 cercano a los 0.009s



Podemos observar que ambos algoritmos realizan aproximadamente el mismo número de Obs.



Como realizan el mismo número de OBs, tardarán aproximadamente lo mismo

**5. Respuesta a las preguntas teóricas.**

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

5.1 Pregunta 1

Nuestra implementación de aleat\_num inicial consistía en obtener un número aleatorio generado por la función rand(), que va a devolver un número equiprobablemente aleatorio entre 0 y RAND\_MAX. Después dividimos este número aleatorio entre el máximo, y obtenemos un número entre 0 y 1, para posteriormente multiplicarlo por la diferencia entre el superior y el inferior entre los que queremos obtener el número aleatorio. De esta forma obtenemos un rango de valores que va de 0 a (sup - inf). A este resultado le sumamos el mínimo, para trasladar este rango a los valores que nos interesan, es decir, entre inf y sup. La función devuelve el resultado haciendo un casting a entero, de modo que, por ejemplo, todos los valores entre 6.0 y 6.9 van a devolver un 6.

Sin embargo, esta implementación no era totalmente equiprobable, ya que si, por ejemplo, queríamos obtener un valor aleatorio entre 1 y 6, la única forma de obtener un 6 sería que la función rand() nos devolviese el valor RAND\_MAX, cuya probabilidad de aparecer es de 1/RAND\_MAX, es decir, casi nula. Por ello, sería muy poco probable obtener un 6 en nuestro aleat\_num(). Por ese motivo, en la implementación anterior realizamos una modificación que consistía en, a la hora de multiplicar el número que obtenemos entre 0 y 1 (después de haber dividido entre RAND\_MAX), en vez de multiplicar por (sup - inf), multiplicamos por (sup – inf + 1). De esta forma obtendremos un valor entre inf y (sup + 1) y, si el número obtenido es mayor que sup (es decir, que rand() nos haya devuelto RAND\_MAX) mandamos este valor a sup. De esta forma, si queremos obtener un valor entre 1 y 6, la probabilidad de obtener un 6 va a ser 1 unidad mayor que la de obtener cualquier otro número, lo cual hace que esta implementación sea mucho más equiprobable que la anterior.

A la hora de mejorar la probabilidad de nuestro algoritmo, recurrimos a un artículo muy interesante acerca del tema.

FINE, T. L (1973). Theories of Probabiliry: An examinarion of foundations. New York: Academic Press.

5.2 Pregunta 2

La corrección del algoritmo InsertSort se basa en que, cada vez que avanzamos una posición en el array (i+1), una serie de swaps van a situar este elemento en su orden correcto respecto a los i primeros elementos, sabiendo que los i primeros ya van a estar ordenados. Realizando un razonamiento de inducción, sabemos que cuando i = N (siendo N el tamaño de la tabla a ordenar), el algoritmo va a haber ordenado la tabla al completo.

5.3 Pregunta 3

El bucle de InsertSort comienza a partir del segundo elemento de la tabla, ya que de lo contrario, si analizamos el primer elemento, no tendríamos con qué compararlo. Por ello las comparaciones de clave comienzan a partir del segundo elemento, que puede ser comparado con el primero.

5.4 Pregunta 4

La comparación entre los dos elementos, tal que si el elemento de la posición i < i-1, realiza un swap.

5.5 Pregunta 5

En el caso peor de InsertSort, el tiempo de ejecución es: , es decir

En el caso mejor,

5.6 Pregunta 6

**6. Conclusiones finales.**

Discusión final sobre la práctica y los resultados obtenidos.