Análisis de Algoritmos 2017/2018

Práctica 1

Daniel Santo-Tomás y Lucía Rivas ,Grupo 1201

Código	Gráficas	Memoria	Total

1. Introducción.

En esta práctica hemos implementado una serie de funciones basadas en la generación de permutaciones aleatorias,que posteriormente hemos aplicado para el desarrollo del algoritmo local de ordenación Bubble Sort,para finalmente generar otras funciones que estudiaran y almacenaran los datos del estudio del algoritmo. A lo largo de esta memoria se expondrán los objetivos,las implementaciones y las herramientas de diseño utilizadas en cada apartado.

2. Objetivos

2.1 Apartado 1

Basándonos en la función rand(),de la librería stdlib de C,el objetivo de este apartado es implementar una función de generación de números aleatorios en un rango dado ,y que esta función sea equiprobable,es decir,que todos los números en ese rango salgan con una frecuencia similar

2.2 Apartado 2

Siguiendo el pseudocódigo dado en el enunciado y a partir de la función del apartado 1,el objetivo de este segundo apartado es crear una función que genere un tabla ordenada de tamaño N para después aplicarle N permutaciones aleatorias,devolviendo una tabla desordenada de tamaño N

2.3 Apartado 3

Haciendo uso de la función del apartado 2,el objetivo de esta sección es implementar una función que devuelva un array de n_perms punteros a tablas de tamaño N permutadas

2.4 Apartado 4

El objetivo de este apartado es programar el algoritmo local de ordenación Bubble Sort,al cual se le pasa una tabla,el primer indice desde el que se quiere ordenar y el último,y devuelve la tabla ordenada entre esos dos indices. Se prueba su correcto funcionamiento con un main dado que genera tuna tabla permutada con las funciones ya implementadas y les aplica el algoritmo para ordenarla.

2.5 Apartado 5

Definida la estructura tiempo en tiempo.h,en este apartado se pide estudiar el funcionamiento de Bubble Sort,implementando tres funciones. La primera estudia el funcionamiento de un cierto algoritmo de ordenación ,pasado como argumento,y almacena los datos obtenidos en una estructura de tipo tiempo. La segunda genera varias tablas permutadas de distintos tamaños y estudia mediante la primera función el funcionamiento del algoritmo para cada tabla,almacenando los resultados en un array de

tipo tiempo,para posteriormente llamar a la tercera función,que imprime estos resultados en un fichero txt de salida.

3. Herramientas y metodología

Hemos desarrollado la práctica desde Linux,programando con el editor de texto básico y compilando mediante el fichero Makefile incluido en el .zip de la práctica. Además ,hemos hecho uso de Valgrind para comprobar que no había problemas de memoria y de GNUplot para la creación de las gráficas.

3.1 Apartado 1

La función srand() establece la "semilla" a través de la cual se generan los números aleatorios. Normalmente se establece como semilla el tiempo de reloj que hay cuando se ejecuta el programa,pero con esas semilla puede surgir el problema de que ,al llamar varias veces a la función en el mismo main,salga constantemente el mismo numero. Para evitar esto,la hemos implementado de tal manera que la semilla se establece cada vez que se llama a la función,siendo esta semilla un numero aleatorio generado por rand. Para devolver un numero en el rango pedido,la función devolverá el limite inferior mas el numero aleatorio módulo limite superior menos inferior mas uno. Es decir,el resultado como poco será el inferior,en el caso de que el numero aleatorio sea múltiplo de inf-sup+1,y como mucho será el máximo,en el caso de que el numero modulo sup-inf+1 sea sup-inf.

Aparte,para el desarrollo del histograma pedido,hemos desarrollado un main llamado ejercicio1b.c,donde al igual que en ejercicio1.c,se llama varias veces a la función creada,dando lugar a una serie de números aleatorios,solo que el nuevo main lo que hace es contabilizar el número de veces que sale cada número e imprimirlo en un fichero txt llamado datos.txt,para facilitar la creación del gráfico con GNUplot.

3.2 Apartado 2

Siguiendo el pseudocódigo dado,la función de este apartado crea una tabla de int de tamaño N,metiendo en cada posición i el número i,de manera que se genera una tabla ordenada desde 0 hasta N-1.A continuación,se permuta la tabla de la siguiente manera. Se hace un bucle for desde i hasta i menor que N,donde en cada iteración se guarda el valor de la posición i en un int auxiliar,y se genera un numero aleatorio entre i y N-1.Después,se guarda en la posición i el valor de la posición dada por el numero aleatorio,para posteriormente guardar en esta ultima posición el valor guardado en el int auxiliar(valor original de array[i]),quedando así la tabla permutada.

3.3 Apartado 3

La función de este apartado simplemente crea un doble puntero de tipo int de tamaño n_perms llamado perm ,y para cada posición desde i=0 hasta N-1,llama a la función del apartado 2 ,generando en cada iteración una lista de tamaño N permutada que se guarda en perm[i].

3.4 Apartado 4

En este apartado hemos implementado el algoritmo de ordenación Bubble Sort. Este recibe como argumentos una tabla,el primer indice desde el que ordenar y el último. Se implementa mediante dos bucles for,un primero descendente que hace variar i entre el último indice y el primero mas uno,y otro que varia j entre el primer indice e i-1. En cada iteración del segundo bucle se produce una comparación(operación básica de este algoritmo,contabilizada con un int),donde se comprueba si el indice j de la lista es mayor que el siguiente indice(j+1). En tal caso,se invierten los contenidos de ambas posiciones. De esta manera,los indices mas grandes van quedando mas a la derecha de la tabla,en su posición final,por tanto en la siguiente iteración del bucle más externo,se establece i de manera que el segundo bucle no llegue hasta el final de la tabla,sino a una posición antes,ya que en la posición final ya esta el numero mas grande,y así sucesivamente a lo largo del algoritmo,hasta ordenar la tabla entera. La función devuelve el valor del contador de OB realizadas por Bubble Sort.

3.5 Apartado 5

La primera función de este apartado recibe como argumento un numero de permutaciones n_perms,un tamaño N,un algoritmo de ordenación método y una estructura de tipo tiempo,la cual sirve para almacenar la información del análisis de un algoritmo,con datos tales como el tamaño de la tablas ordenadas,el numero de tablas ordenadas, el tiempo promedio de ordenación y el numero promedio ,máximo y mínimo de OB realizadas por el algoritmo método.

Esta función esta implementada de manera que primeramente genera un doble puntero a int de tablas de tamaño N permutadas(llamado perms) mediante la función genera_permutaciones del apartado 3.Posteriormente,mediante un bucle for llama al algoritmo método lo aplica a las diversas tabla de perms,ordenándolas y devolviendo el numero de operaciones básicas. Dentro del bucle,se almacenan en un contador el numero total de operaciones básicas que se realizan para perms,a igual que el valor máximo y mínimo,de manera que en cada iteración se comprueba si el numero de OB de la ultima ordenación es mayor o menor que el máximo o el mínimo,respectivamente,y se actualiza el valor de máx y nin si se da el caso. Además es necesario registrar el tiempo que tarda la función método en ser llamada,ejecutarse y devolver el resultado. Para eso,utilizamos la función gettimeofday (esta solución la hemos obtenido en la página https://davidcapello.com/blog/cpp/medir-el-tiempo-de-una-rutina/),y se necesitan las librerías time.h y sys/time.h y la estructura timeval. Esta estructura consta de dos campos que registran tiempo,uno en segundos y otro en microsegundos,de manera que la precisión es mayor que con otras funciones,como clock(la utilizamos y la descartamos debido a su baja precisión). Para medir los tiempos,declaramos t_ini y t_fin,de tipo timeval,y registramos los datos de la hora,minuto,segundo etc antes de llamar a método mediante gettimeofday,guardándolo en t_ini. Después de la llamada a método,se registra de nuevo el tiempo,pero guardándolo en t_fin. Finalmente,con una conversión a double,se suman los campos tv_sec(segundos) y tv_usec(microsegundos) de t_ini y t_fin,individualmente,y se resta el resultado de t_fin menos el de t_ini,sumándolo a un int llamado secs(inicializado a 0,va registrando el tiempo total de todas las llamadas a método).

Teniendo ya todos los datos,se almacena cada uno en el lugar que le corresponde de la estructura tiempo. En el caso del numero medio de OB,se guarda el valor obtenido del total de OB realizadas entre el numero de veces que se ha llamado a método(n_perms).Algo similar se hace con el tiempo promedio,almacenando el double resultad de dividir el tiempo total ,guardado en secs,entre n_perms. Finalmente,liberamos el espacio almacenado para perms y si no hay errores,la función devuelve OK(la función es de un tipo llamado short,que devuelve ERR si hay alguna incidencia y OK en caso contrario).

La segunda función recibe como argumento un algoritmo de ordenación método,el nombre de un fichero,y cuatro ints num_min,num_max,incr y n_perms. Esta función registra en un array de tipo tiempo los datos del funcionamiento del algoritmo método sobre n_perms tablas cuyo tamaño va desde n_min hasta n_max,aumentando el tamaño con un incremento igual a incr. Por tanto,el tamaño de este array sera de (num_max - num_min)/incr +1. Planteándolo de esta manera,solo se llegara a estudiar el funcionamiento sobre tablas de tamaño num_max si (num_max - num_min) modulo incr es 0.

La implementación se realiza mediante un bucle for que va de i igual a num_min a i menor o igual que num_max,sumando en cada iteracion incr a i. En interacciona,se llama a la primera función de este apartado,pasándole como argumentos el método,n_perms,i como tamaño de la tabla y un puntero aux que apunta a la posición del array de tiempo donde se va a almacenar los datos. Después de la llamada a la función,se aumenta el valor de este puntero aux y se continua.

Obtenidos y almacenados los datos,la función llama a la tercera función de este apartado. Esta recibe como argumentos el nombre de un fichero(en este caso el pasado por argumento en la segunda función),un array de tipo tiempo(en este caso,el generado en la función 2) y el tamaño de este array((num_max-num_min)/incr +1). El objetivo de esta función es imprimir los datos almacenados en el array en e fichero cuyo nombre se pasa por argumento. Sencillamente,mediante un bucle for se recorre ele array y con fprintf se imprimen los datos en el fichero. Si todo funciona correctamente,se cierra el fichero y se devuelve OK;en caso contrario se devuelve ERR.

Volviendo a la segunda función y después de llamar a la tercera,se libera el array de tiempos y se devuelve OK en caso de que todo funcione,ERR en caso contrario.

4. Código fuente

4.1 Apartado 1

Librerías: permutaciones.h y time.h

```
int aleat_num(int inf, int sup)
{
  if(inf < 0 || sup <=0) return -1;

  srand(rand());

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

4.2 Apartado 2

Librerías: permutaciones.h y tiempo.h

```c
int* genera_perm(int N)
{
  if(N <=0) return NULL;

  int *array = (int*)malloc(sizeof(int)*N);
  if(!array) return NULL;

  int aux, aleat, i;
  for(i = 0; i < N; i++){
        *(array + i) = i;
  }

  for(i = 0; i < N; i++){
        aux = *(array + i);
        aleat = aleat_num(i, N - 1);
        if(aleat == -1) return NULL;

        *(array + i) = *(array + aleat);
        *(array + aleat) = aux;
  }

  return array;
}
```

4.3 Apartado 3

Librerías: permutaciones.h y tiempo.h

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

        if(n_perms<=0 || N <=0) return NULL;
        int **perm=(int**)malloc(sizeof(int*)*n_perms);
        if(!perm) return NULL;
        int i;
        for(i=0;i<n_perms;i++){
                perm[i] = genera_perm(N);
                if(!perm[i]) return NULL;
        }
        return perm;
}
```

4.4 Apartado 4

Librerías : ordenación.h

```
int BubbleSort(int* tabla, int ip, int iu)
{
        if(!tabla || ip<0 || iu<0) return ERR;
        int cont,i,j,aux;
        cont =0;

        for(i = iu; i >= ip+1; i--){
                for(j = ip; j <= i-1; j++){
                        cont++;
                        if(tabla[j] > tabla[j+1]){
                                aux = tabla[j];
                                tabla[j] = tabla[j+1];
                                tabla[j+1] = aux;
                        }
                }
        }

        return cont;
}
```

4.5 Apartado 5

Librerías:tiempos.h,ordenación.h,permutaciones.h,stdio.h,stdlib.h,time.h,sys/time.h

```
short tiempo_medio_ordenacion(pfunc_ordena metodo,
                              int n_perms,
                              int N,
                              PTIEMPO ptiempo)
{
    if(!metodo || !ptiempo || n_perms<=0 || N<=0) return ERR;
    int **perms;
    int i, min = -1, max = -1, aux;
    double secs = 0, med = 0;
    struct timeval t_ini, t_fin;


    perms = genera_permutaciones(n_perms,N);
    if(!perms) return ERR;

    for(i = 0; i < n_perms; i++){
        gettimeofday(&t_ini, NULL);
        aux = metodo(perms[i], 0, N-1);
        gettimeofday(&t_fin, NULL);
        if(aux == ERR) return ERR;
        secs += (double)(t_fin.tv_sec + (double)t_fin.tv_usec/1000000) -
        (double)(t_ini.tv_sec + (double)t_ini.tv_usec/1000000);
        med += aux;
        if(min == -1 && max == -1) {
                min = aux;
                max = aux;
                continue;
        }

        if(aux < min) min = aux;
        if(aux > max) max = aux;

    }
```

```
    ptiempo->N = N;
    ptiempo->n_elems = n_perms;
    ptiempo->max_ob = max;
    ptiempo->min_ob = min;
    ptiempo->medio_ob = (med/n_perms);
    ptiempo->tiempo = (secs/n_perms);

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

    int i;

    PTIEMPO tiempo = (PTIEMPO)malloc(sizeof(TIEMPO)*
    (((num_max - num_min)/incr) + 1));
    if(!tiempo) return ERR;

    PTIEMPO aux;
    aux = tiempo;

    for(i = num_min ; i <= num_max ; i += incr){
        if(tiempo_medio_ordenacion( metodo,n_perms, i, aux) == ERR) return ERR;
        aux++;
    }

    if(guarda_tabla_tiempos(fichero, tiempo, (((num_max - num_min)/incr) + 1)) == ERR) return ERR;

    free(tiempo);

    return OK;

}
short guarda_tabla_tiempos(char* fichero, PTIEMPO tiempo, int n_tiempos)
{
    if(!fichero || !tiempo || n_tiempos <= 0 ) return ERR;

    FILE *f = NULL;
    int i;
    PTIEMPO aux;

    aux = tiempo;

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

    for(i = 0; i <n_tiempos ; i++){
         fprintf(f,"%d %.20f %lf %d %d \n",aux->N, aux->tiempo,
        | aux->medio_ob, aux->max_ob, aux->min_ob);
         aux++;
    }

    fclose(f);

    return OK;
}
```

5. Resultados, Gráficas

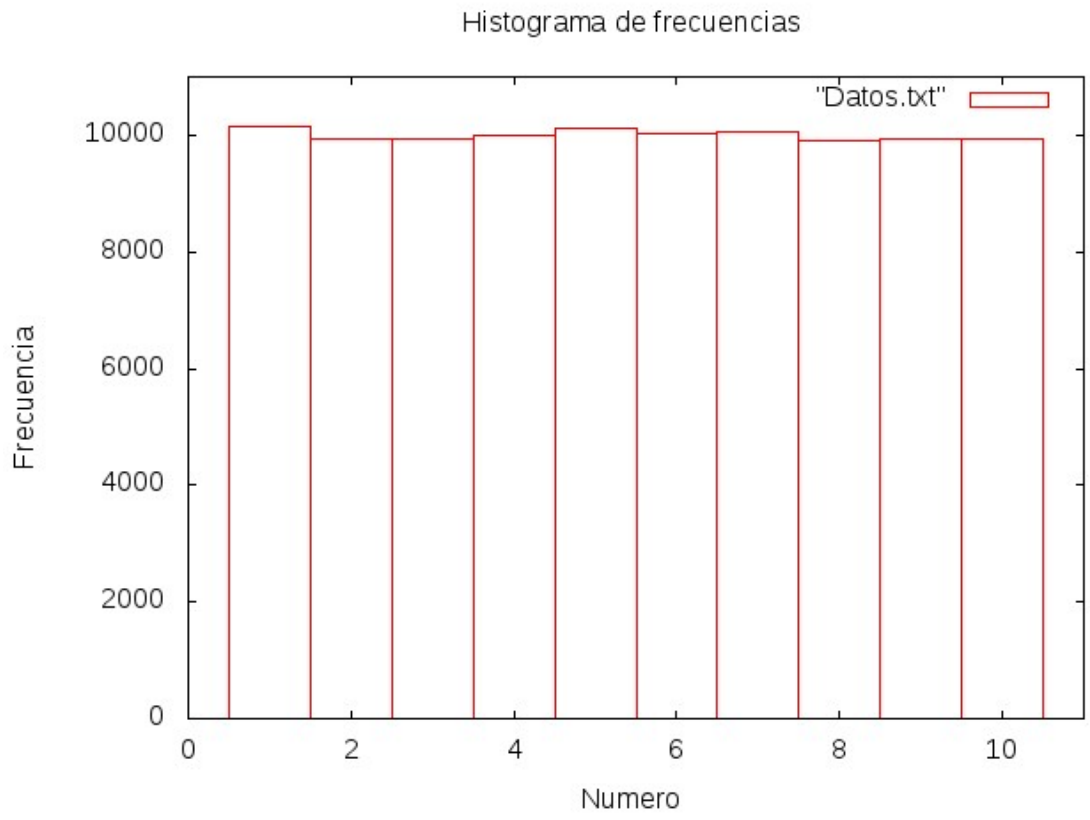
5.1 Apartado 1

Resultados ejercicio1.c

```
danist@danist-Lenovo-E51-80:~/Documentos/UAM/PRACTICAS ANALISIS/ANAL/practica1 final$ ./ejercicio1 -limInf 1 -limSup 5 -numN 5
Practica numero 1, apartado 1
Realizada por: Lucía Rivas Molina y Daniel Santo-Tomás López
Grupo: 1201
4
5
1
3
3
```

Efectivamente,se generan 5 números aleatorios entre 1 y 5.

El siguiente histograma refleja los resultados obtenidos en la generación de 100000 números aleatorios entre 1 y 10.



Como se puede observar,la gráfica es prácticamente constante,por lo que la función de generación de números aleatorios es eficaz.

5.2 Apartado 2

```
danist@danist-Lenovo-E51-80:~/Documentos/UAM/PRACTICAS ANALISIS/ANAL/practica1 final$ ./ejercicio2 -tamanio 10 -numP 5
Practica numero 1, apartado 2
Realizada por: Lucía Rivas y Daniel Santo-Tomás
Grupo: 1201
6 5 4 1 0 3 9 7 8 2
9 4 8 6 3 2 5 1 7 0
5 8 0 3 4 9 6 7 1 2
5 8 9 6 1 3 7 0 4 2
8 0 4 9 5 6 2 1 3 7
```

Este main genera numP listas permutadas de tamaño 10,llamando varias veces a genera_perm

5.3 Apartado 3

```
danist@danist-Lenovo-E51-80:~/Documentos/UAM/PRACTICAS ANALISIS/ANAL/practica1 final$ ./ejercicio3 -tamanio 10 -numP 5
Practica numero 1, apartado 3
Realizada por: Lucía Rivas y Daniel Santo-Tomás
Grupo: 12011 5 0 4 8 3 2 9 6 7
8 2 9 0 6 1 4 5 3 7
0 4 6 5 7 8 3 1 2 9
6 7 0 9 2 5 3 1 8 4
0 7 6 9 5 4 8 1 2 3
```

Al igual que en el apartado 3,se generan numP listas permutadas de tamaño N,pero esta vez llamando una única vez a genera_permutaciones

5.4 Apartado 4

```
danist@danist-Lenovo-E51-80:~/Documentos/UAM/PRACTICAS ANALISIS/ANAL/practica1 final$ ./ejercicio4 -tamanio 10
Practica numero 1, apartado 4
Realizada por:  Lucía Rivas y Daniel Santo-Tomás
Grupo: 1201
0       1       2       3       4       5       6       7       8       9
```

Ejercicio4.c genera una lista permutada de tamaño 10 y la ordena mediante Bublle Sort. La salida es la lista ordenada.

5.5 Apartado 5

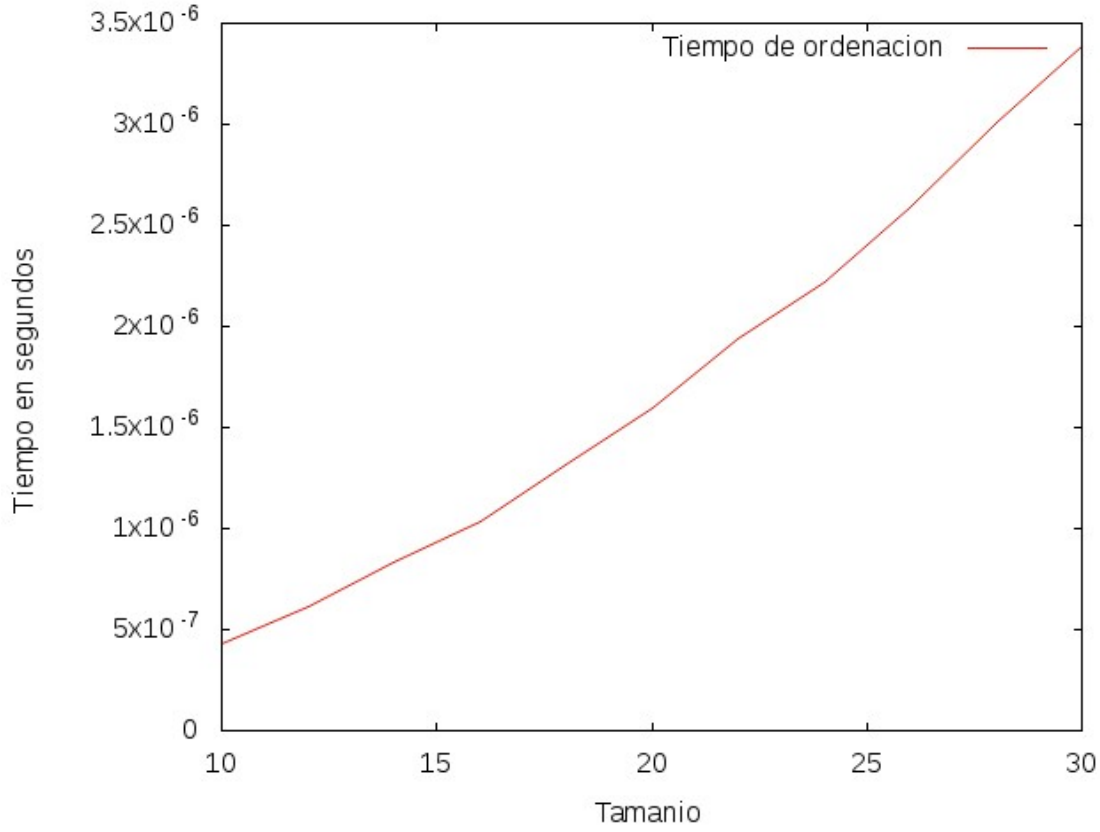
```
danist@danist-Lenovo-E51-80:~/Documentos/UAM/PRACTICAS ANALISIS/ANAL/practica1 final$ ./ejercicio5 -num_min 10 -num_max 30 -incr 2 -numP 100000 -fichSalida tiempos.txt
Practica numero 1, apartado 5
Realizada por:  Lucía Rivas y Daniel Santo-Tomás
Grupo: 1201
Salida correcta
```

Este programa llama a la función genera tiempos ordenación ,que estudia la ordenación de tablas de tamaño variante entre 10 y 30(de dos en 2),generando y ordenando 100000 tablas por tamaño. Los resultados del análisis se guardan en el fichero tiempos txt,y están expresados en las gráficas que vienen a continuación.

La primera gráfica expresa la relación entre el tamaño de las tablas y las OB media,máxima y mínima de Bubble Sort al ordenar las tablas. Al ser un algoritmo local y no haber implementado una bandera que determine si la tabla esta ya ordenada o no,para cualquier tabla del mismo tamaño,se realizan el mismo numero de operaciones básicas,por lo que las operaciones media,máxima y mínima serán iguales para cada grupo de tablas del mismo tamaño. Por tanto,la gráfica queda así:

Como se puede observar,cuanto más tamaño,más operaciones básicas se realizan.

La segunda gráfica representa la relación entre el tamaño de las listas y el tiempo medio,en segundos,de ordenación



Se puede observar que a mayor tamaño de las tablas,mayor es el tiempo promedio de ordenación,a que se tienen que realizar mas iteraciones en Bubble Sort para ordenar las tablas.

6. Respuesta a las preguntas teóricas.

6.1 Pregunta 1:Justifica tu implementación de aleat_num ¿en qué ideas se basa? ¿de qué libro/artículo, si alguno, has tomado la idea? Propón un método alternativo de generación de números aleatorios y justifica sus ventajas/desventajas respecto a tu elección.

Nuestra implementación de aleat_num se basa en la idea de generar cada vez una semilla nueva para la función rand(),aumentando así la aleatoriedad de la generación de números. El funcionamiento de rand() consiste en que a raíz de una semilla ,realiza una serie de operaciones,obteniendo así números aleatorios,pero esto puede generar errores,por lo que al modificar la semilla cada vez que se llama a aleat_num,se garantiza que la salida es más aleatoria. Un método alternativo,que planteamos primeramente,es que la semilla sea el tiempo de ejecución:Esto suele funcionar,ya que el programa no se va a ejecutar nunca en dos tiempos iguales. Sin embargo,tal y como comprobamos,esto puede generar errores al llamar varias veces a la función en el mismo programa,por lo que cambiar la semilla a otro numero aleatorio es más seguro para garantizar que no se repitan constantemente los números generados

6.2 Pregunta 2:Justifica lo más formalmente que puedas la corrección (o dicho de otra manera, el porqué ordena bien) del algoritmo BubbleSort.

La corrección de Bubble Sort se basa en que a cada iteración del bucle for más exterior,el número más alto esta ya en su posición. Esto sucede gracias a que el bucle for interior va haciendo comparaciones de clave e intercambiando números,de manera que los más grandes se van moviendo a la derecha,generando al final que el mas grande este a la derecha. Al reducir en uno el limite de ordenación la siguiente vez que se ejecuta este bloque,se garantiza que los números mas grandes van quedando poco a poco en su posición,hasta finalmente tener una tabla ordenada.

6.3 Pregunta 3:¿Por qué el bucle exterior de BubbleSort no actúa sobre el primer elemento de la tabla?

Porque el bucle va hasta ip+1,ya que en el caso de una tabla de tamaño mayor que uno,al ir desplazando hacia la derecha los números más grandes,te garantizas que en la última iteración del bucle más exterior(i=ip+1),solo faltan dos números por ordenar,del cual el más grande quedará en la posición 1,por lo tanto el otro,no solo sera mas grande que el oro,sino que podemos asegurar que es el más pequeño de la lista,quedando en la posición 0.Si la tabla es de tamaño 1,no se llega a ejecutar el bucle mas exterior,ya que una tabla con un único numero ya esta ordenada.

6.4 Pregunta 4:. ¿Cuál es la operación básica de BubbleSort?

La comparación del bucle más interior ente tabla[j] y tabla[j+1]

6.5 Pregunta 5:Dar tiempos de ejecución en función del tamaño de entrada n para el caso peor WBS(n) y el caso mejor BBS(n) de BubbleSort. Utilizar la notación asintotica (O, Θ, o, Ω,etc) siempre que se pueda.

La forma en la que esta implementado Bubble Sort hace que tarde exactamente lo mismo en ordenar un tabla ordenada que una desordenada,realizando siempre $N^2/2 -N/2$ comparaciones de clave.Por lo tanto el caso peor y el caso mejor son el mismo, corresponden a $N^2/2 -N/2$.Asi pues,los tiempo de ejecución respecto al tamaño son los expresados en la gráfica del apartado 5.5 de esta memoria

7. Conclusiones finales.

Esta práctica nos ha servido para estudiar el funcionamiento de un algoritmo local de ordenación,viendo así en la práctica lo expuesto en teoría. La implementación de algoritmo elegida ha sido la más sencilla,con la que se puede observar porque no es del todo eficaz y porque es necesario plantear varias implementaciones,para que si una tabla esta ya ordenada,se detenga el algoritmo. Además,hemos aprendido como registrar tiempos de ejecución de programas,además de como utilizar GNUplot para graficar de una forma sencilla desde e terminal de Linux.