Análisis de Algoritmos

Práctica 2

Grupo 1261

Pareja 09

Ari Handler Gamboa

Adrián Lorenzo Mateo

1. Mergesort

1.1 Introducción

En este bloque se procederá a desarrollar y estudiar el algoritmo de ordenación Mergesort. Se utilizará el entorno de análisis elaborado en la anterior práctica, obteniendo los resultados en función del tiempo y del número de operaciones básicas. Finalmente se crearán las gráficas correspondientes tanto a la variación del tiempo promedio de ejecución, como del número de operaciones básicas en función del tamaño de la permutación y se analizará el rendimiento obtenido con el teórico.

1.2 Código

```
/*************************************************/
/* Funcion: MergeSort    Fecha: 17/10/2012       */
/* Autores: Adrián Lorenzo Mateo                 */
/*                                               */
    Ari Handler Gamboa                          */
/*                                               */
/* Funcion que ordena un array de enteros haciendo  */
/* uso del método Mergesort, devolviendo el número  */
/* de OBs durante la ejecución                   */
/*                                               */
/* Entrada:                                      */
/* int tabla: Tabla a ordenar                    */
/* int ip: Primer elemento de la tabla           */
/* int iu: Ultimo elemento de la tabla           */
/* Salida:                                       */
/* int: Número de OBs ejecutadas o               */
/* ERR en caso de error                          */
/*************************************************/

int mergesort(int* tabla, int ip, int iu)
{

      int ob_count, imedio;

      /* Control de errores de los parametros de entrada */
      if((tabla==NULL)||(ip<0)||(iu<ip))
            return ERR;

        ob_count=0;
       /* Condicion de parada de la recursion */
      if (ip == iu)
             return ob_count;
```

```
     /* Funcion "sencilla" de partir */
     imedio = (iu+ip)/2;
     /* Llamada recursiva del algoritmo para la subtabla
        izquierda */
     ob_count+=mergesort(tabla, ip, imedio);
     /* Llamada recursiva del algoritmo para la sutabla
        derecha */
     ob_count+=mergesort(tabla, imedio+1, iu);

     /* Combinacion de ambas tablas ordenadas */
     return (ob_count + merge(tabla, ip, iu, imedio));
}
```

La función **mergesort** realiza la ordenación de una tabla **tabla** de enteros desde el índice **ip** hasta el **iu**. Para ello divide la tabla según el suelo de de la mitad de la tabla y aplica **mergesort** nuevamente a cada una de las mitades de la tabla original. Por último, llama a la función **merge** que será la encargada de mezclar ambas mitades ordenadas en una tabla resultante también ordenada.

```
/*****************************************************/
/* Funcion: Merge    Fecha: 17/10/2012                */
/* Autores: Adrián Lorenzo Mateo                      */
/*        Ari Handler Gamboa                          */
/*                                                    */
/* Funcion que combina de forma ordenada las subtablas */
/* definidas por los indices pasados como argumentos  */
/*                                                    */
/* Entrada:                                           */
/* int tabla: Tabla a ordenar                         */
/* int ip: Primer elemento de la tabla                */
/* int iu: Ultimo elemento de la tabla                */
/* int imedio: Indice que marca la mitad de la tabla  */
/* Salida:                                            */
/* int: Número de OBs ejecutadas o                    */
/* ERR en caso de error                               */
/*****************************************************/
int merge(int* tabla, int ip, int iu, int imedio)
{
    int * tabla_aux = NULL;
    int i, j, k, ob_count;

     /* Control de errores de los parametros de entrada */
    if ((tabla == NULL) || (ip<0) || (ip>iu) || (imedio<ip) ||
     (imedio>iu))
        return ERR;

     /* Reserva de memoria de la tabla auxiliar */
    if ((tabla_aux = (int *)malloc((iu-ip+1)*sizeof(int))) ==
     NULL)
        return ERR;
```

```
      /* Se copia la tabla original y se inicializan las
      variables de los bucles */
    memcpy(tabla_aux, tabla, (iu-ip+1)*sizeof(int));
    k  = 0;
    i  = ip;
    j = imedio+1;
    ob_count = 0;

      /* Bucle que recorre la tabla de izquierda a derecha ambas
      sub-tablas */
    while((i<=imedio) && (j<=iu)){
        ob_count++;
           /* Determinacion del elemento menor de ambas
           subtablas e insercion
              del mismo en la tabla resultante */
        if (tabla[i] < tabla[j]){ /* Operacion Basica */
            tabla_aux[k] = tabla[i];
            i++;
        }
        else{
            tabla_aux[k] = tabla[j];
            j++;
        }
           /* Aumento del indice que recorre la tabla
           resultante */
        k++;
    }
      /* Se determina cual de las subtablas tiene un exceso de
      elementos no insertados en la tabla resultante y se
      añaden al final de esta */
    if (i>imedio){
        for(;j<=iu;j++,k++)
            tabla_aux[k] = tabla[j];
    }
    else if (j>iu){
        for(;i<=imedio;i++,k++)
            tabla_aux[k] = tabla[i];
    }

      /* Se copia la tabla auxiliar resultante en la pasada por
      argumento y se libera la memoria de la auxiliar */
    memcpy(&(tabla[ip]), tabla_aux, (iu-ip+1)*sizeof(int));
    free(tabla_aux);

    return ob_count;
}
```

La función **merge** combina de forma ordenada las subtablas que se encuentran dentro del rango definido por **iu** (comienzo de la tabla) hasta **ip** (final de la tabla). Para ello, y sabiendo que el índice que sirve de frontera entre las dos tablas a combinar viene dado por el argumento imedio, itera entre las partes izquierda (desde **ip** hasta **imedio**) y derecha (desde **imedio**+1 hasta **iu**) insertando en una tabla auxiliar los elementos menores de ambas subtablas, quedando finalmente ordenada. A modo de ilustrar el funcionamiento básico de la función **merge** se muestra a continuación la ejecución de dicha rutina sobre una tabla determinada:

Tabla = [1, 3, 5, 2, 4, 6] ip = 0 iu = 5 imedio = 2

T[i] < T[j]	Índices	Tabla auxiliar
N.A.	i=ip=0, j=imedio+1=3, k=0	[]
T[0] < T[3]	i = 1, k = 1	[1]
T[1] > T[3]	j = 4, k = 2	[1, 2]
T[1] < T[4]	i = 2, k = 3	[1, 2, 3]
T[2] > T[4]	j = 5, k = 4	[1, 2, 3, 4]
T[2] < T[5]	i = 3, k = 5	[1, 2, 3, 4, 5]

Como **i** termina siendo mayor que **imedio**, se debe iterar por la sub-tabla derecha, insertando todos los elementos al final de la tabla auxiliar, quedando:

$$T_{AUX} = [1, 2, 3, 4, 5, 6]$$

1.3 Resultados

1.3.1 Resultados de las pruebas realizadas con ejercicio4

- **tamanio <= 0:** La rutina genera_perm encuentra el error devolviendo -1.

```
./ejercicio4      tamanio 1
Practica numero 2, apartado 4
Realizada por: Ari Handler Gamboa y Adrian
Lorenzo Mateo
Grupo: 261
Error: No hay memoria
```

- **tamanio > 0:** El programa se ejecuta correctamente retornando la tabla ordenada mediante SelectSort.

```
./ejercicio4      tamanio 4
Practica numero 2, apartado 4
Realizada por: Ari Handler Gamboa y Adrian
Lorenzo Mateo
Grupo: 261
0 1 2 3
```

Al pasar la herramienta **Valgrind** al **ejercicio4** con un tamaño de tabla de 10 elementos y utilizando las banderas de ejecución correspondientes a la opción **verbose** y de identificación de fugas de memoria obtenemos el resultado siguiente:

```
==4047==
==4047== HEAP SUMMARY:
==4047==     in use at exit: 0 bytes in 0 blocks
==4047==   total heap usage: 10 allocs, 10 frees, 176 bytes
allocated
==4047==
==4047== All heap blocks were freed  no leaks are possible
==4047==
==4047== ERROR SUMMARY: 0 errors from 0 contexts
(suppressed: 14 from 7)
==4047==
==4047== ERROR SUMMARY: 0 errors from 0 contexts
(suppressed: 14 from 7)
```

1.3.1 Resultados de las pruebas realizadas con ejercicio5

- **numP <= 0:** La rutina genera_tiempos_ordenación encuentra el error.

```
./ejercicio5   num_min 100  num_max 1000   incr 100   numP
100 -fichSalida salida.dat   retardo 100
Practica numero 2, apartado 5
Realizada por: Ari Handler Gamboa y Adrian Lorenzo Mateo
Grupo: 261
Error en la funcion Time_Ordena
```

- **incr <= 0:** La rutina genera_tiempos_ordenación encuentra el error.

```
./ejercicio5 num_min 100 num_max 1000 incr 1 numP 100
fichSalida salida.dat retardo 100
Practica numero 2, apartado 5
Realizada por: Ari Handler Gamboa y Adrian Lorenzo Mateo
Grupo: 261
Error en la funcion Time_Ordena
```

- **num_max < num_min:** La rutina genera_tiempos_ordenación encuentra el error.

```
./ejercicio5 num_min 1000 num_max 100 incr 100 numP 100
fichSalida salida.dat retardo 100
Practica numero 2, apartado 5
Realizada por: Ari Handler Gamboa y Adrian Lorenzo Mateo
Grupo: 261
Error en la funcion Time_Ordena
```

- **num_min=100, num_max=1000, incr=100, numP=100, retardo = 100:**

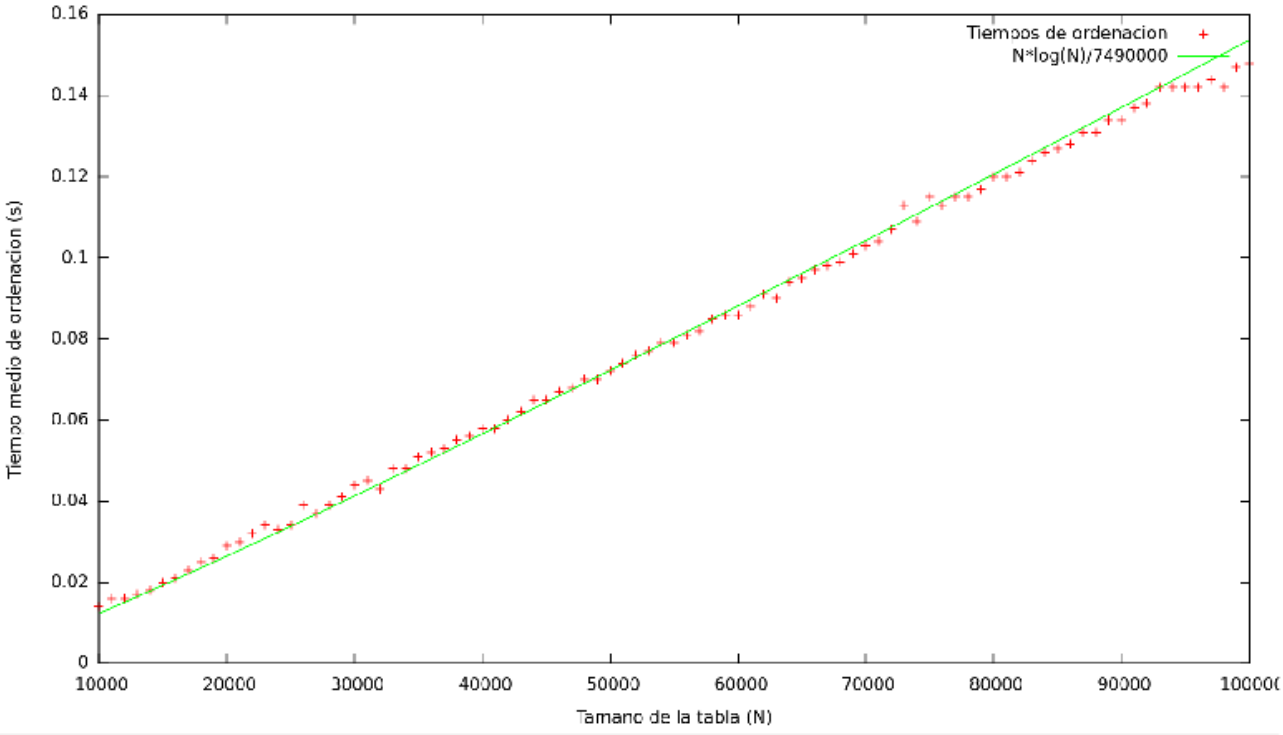
```
./ejercicio5 num_min 100 num_max 1000 incr 100 numP 100
fichSalida salida.dat retardo 100
Practica numero 2, apartado 5
Realizada por: Ari Handler Gamboa y Adrian Lorenzo Mateo
Grupo: 261
Permutaciones de tamaño 100 finalizadas
Permutaciones de tamaño 200 finalizadas
Permutaciones de tamaño 300 finalizadas
Permutaciones de tamaño 400 finalizadas
Permutaciones de tamaño 500 finalizadas
Permutaciones de tamaño 600 finalizadas
Permutaciones de tamaño 700 finalizadas
Permutaciones de tamaño 800 finalizadas
Permutaciones de tamaño 900 finalizadas
Permutaciones de tamaño 1000 finalizadas
Salida correcta
```

Al pasar la herramienta **Valgrind** al a este rutina con un tamaño de tabla de 100 elementos, 100 permutaciones y un retardo de 100 ordenaciones adicionales obtenemos los resultados siguientes:

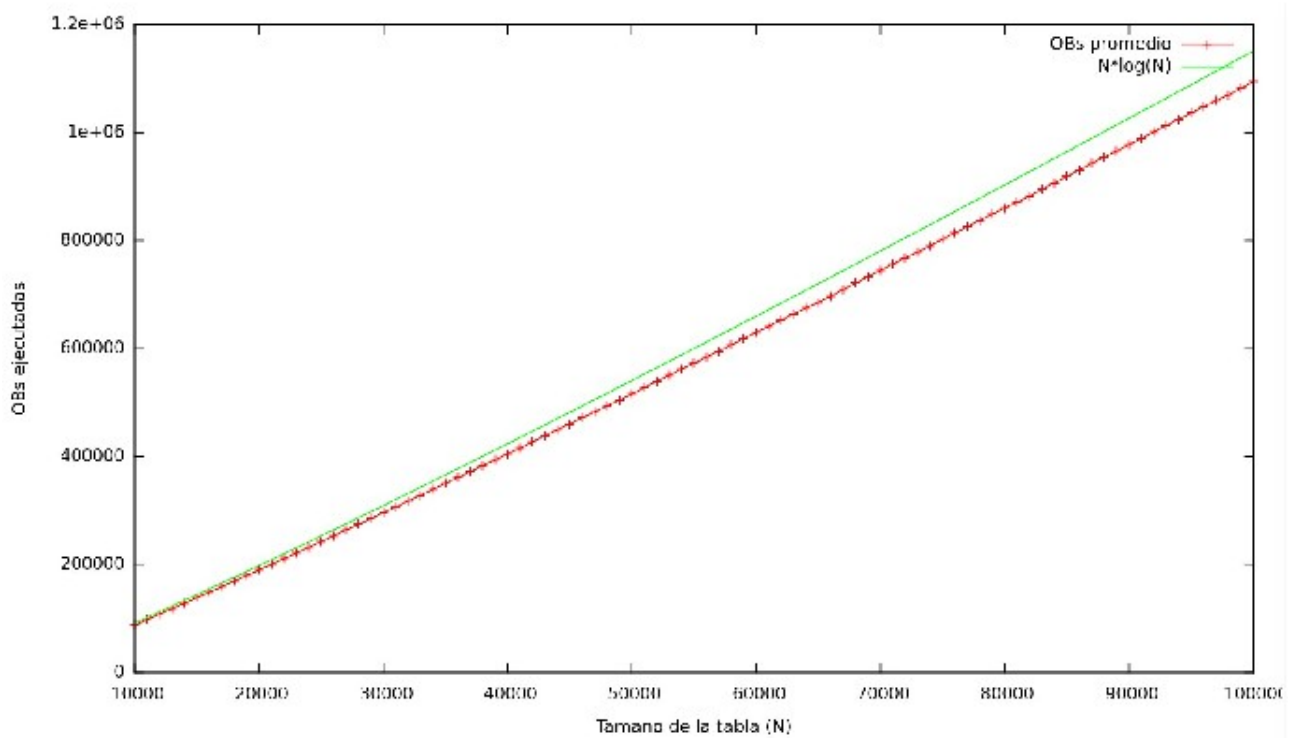
```
==2235==
==2235== HEAP SUMMARY:
==2235==     in use at exit: 0 bytes in 0 blocks
==2235==   total heap usage: 54,901,022 allocs, 54,901,022
frees, 2,064,630,888 bytes allocated
==2235==
```

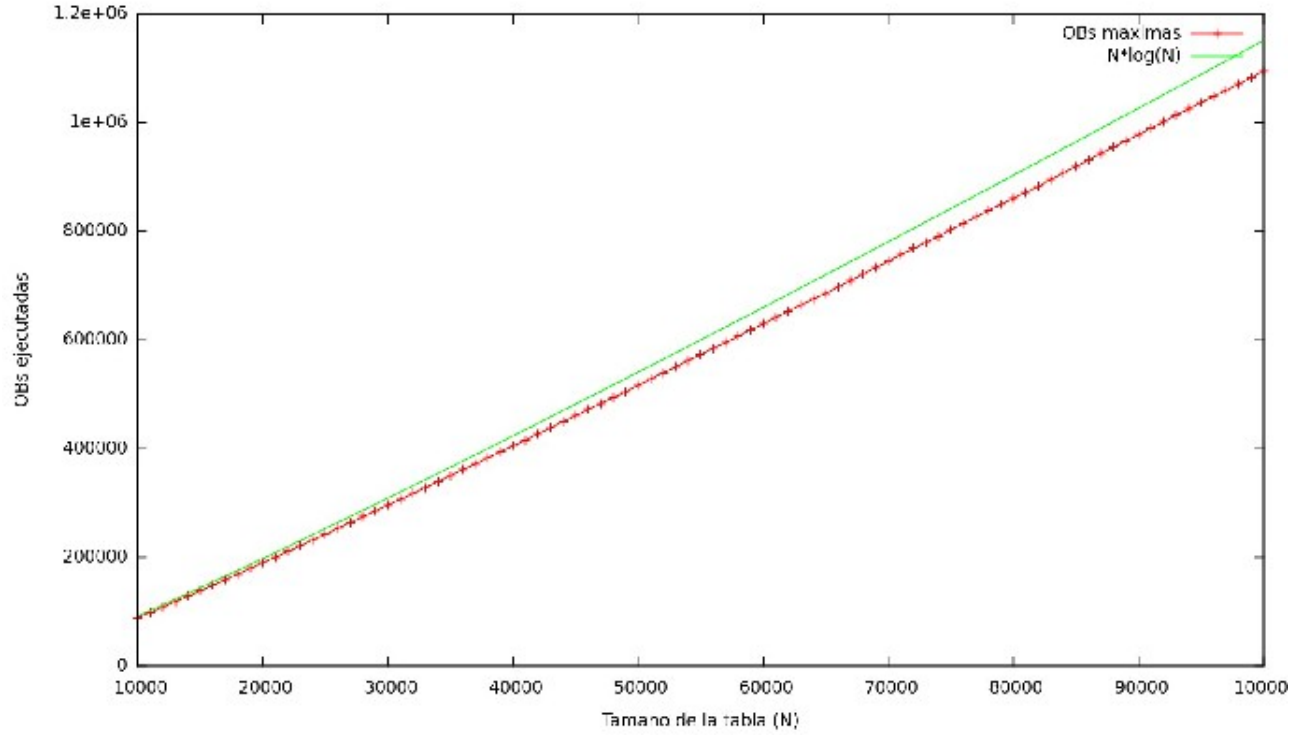
```
==2235== All heap blocks were freed  no leaks are possible
==2235==
==2235== ERROR SUMMARY: 0 errors from 0 contexts
(suppressed:
4 from 4)
```

Generando el fichero **salida.dat** con un tamaño de tabla desde 10K a 100K elementos con un incremento de 10K por iteración, 10 permutaciones por tamaño y 10 repeticiones de la ordenación a modo de retardo, podemos representar con la herramienta **gnuplot** el tiempo medio, así como el mínimo, máximo y promedio de operaciones básicas con respecto al tamaño de tabla. En la Figura 1 se muestran los tiempos medios de ordenación frente al tiempo en comparación con el resultado teórico esperado, es decir, el caso medio para el algoritmo Mergesort, $A_{MS}(N) = N*\log(N)$.



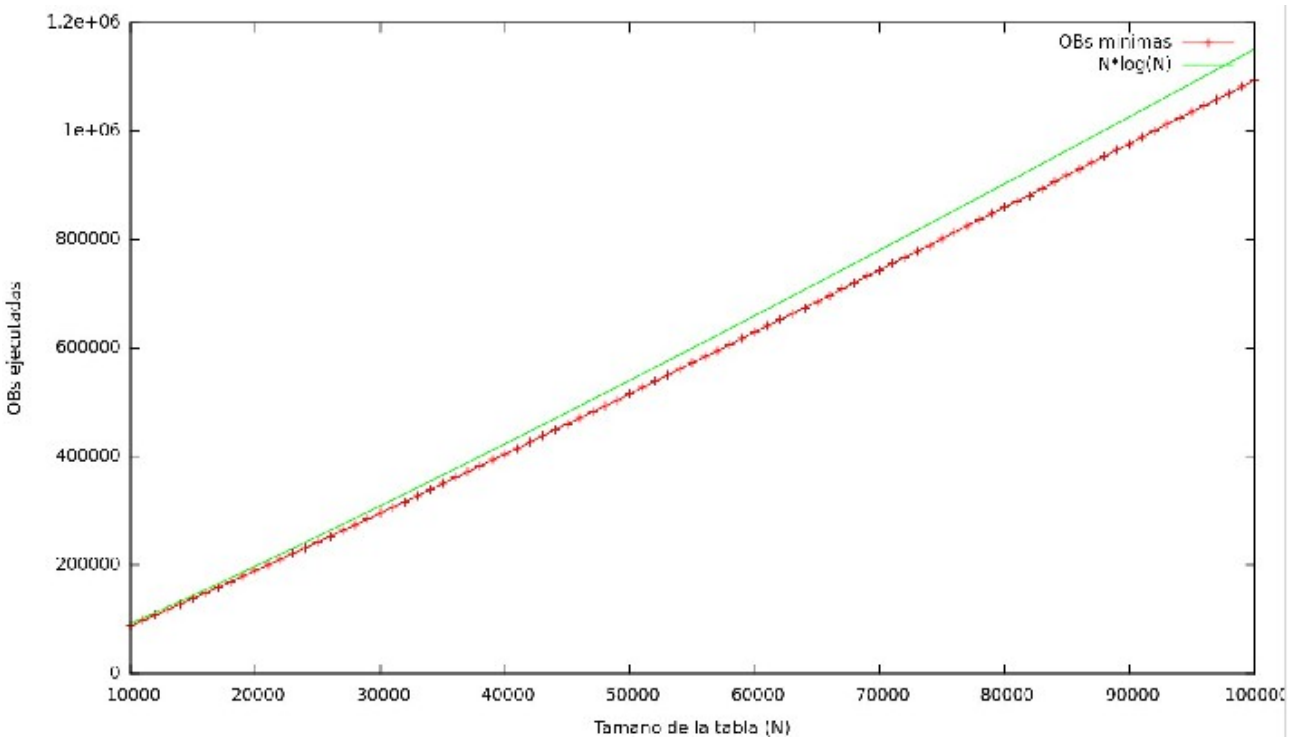
Se puede comprobar en dicha figura que la tendencia (exceptuando algunos picos en las medidas debidos a irregularidades de la velocidad de procesamiento de la CPU) es asintóticamente equivalente al resultado teórico esperado. Al representar en una gráfica el número medio de operaciones básicas frente al tamaño de las tablas que se quieren ordenar y comparándola con el resultado teórico esperado, en este caso el caso medio del algoritmo, $A_{MS}(N) = N*\log(N)$, obtenemos la Figura 2.





Si se representa ahora el máximo de número de operaciones básicas y se compara con el resultado teórico, es decir, el peor caso de Mergesort para tablas de tamaño N, $W_{MS}(N) = N*\log(N)$, se obtiene la gráfica de la Figura 3.

Por último al representar el número mínimo de operaciones básicas ejecutadas con respecto al tamaño de tabla y se compara con el resultado teórico, esto es, el caso mejor del algoritmo Mergesort, BMS(N) = N*log(N), obtenemos la gráfica de la Figura 4.



2. Quicksort

2.1 Introducción

En este bloque se procederá a desarrollar y estudiar el algoritmo de ordenación Quicksort. Se utilizará el entorno de análisis elaborado en la anterior práctica, obteniendo los resultados en función del tiempo y del número de operaciones básicas. Finalmente se crearán las gráficas correspondientes tanto a la variación del tiempo promedio de ejecución, como del número de operaciones básicas en función del tamaño de la permutación y se analizará el rendimiento obtenido con el teórico.

2.2 Código

```
/*************************************************/
/* Funcion: Medio    Fecha: 24/10/2012           */
/* Autores: Adrián Lorenzo Mateo                 */
/*        Ari Handler Gamboa                      */
/*                                               */
/* Funcion pivote que devuelve el menor de los   */
/* indices pasados por argumentos.               */
/*                                               */
/* Entrada:                                      */
/* int ip: Primer indice                         */
/* int iu: Ultimo indice                         */
/* Salida:                                       */
/* int: Menor de los indices                     */
/*************************************************/
int medio(int ip, int iu){

    if(ip<iu)
        return ip;

    return iu;
}
```

La función **pivote** devuelve el menor de los índices **ip** e **iu**. Sirve como función de elección de pivote para el algoritmo **Quicksort**.

```
/*************************************************/
/* Funcion: Partir    Fecha: 24/10/2012          */
/* Autores: Adrián Lorenzo Mateo                 */
/*        Ari Handler Gamboa                     */
/*                                               */
/* Funcion que redistribuye la tabla entre los   */
/* indices pasados por argumento,                */
/* dejando  a la izquierda                       */
/* los elementos menores que el                  */
/* pivote elegido por la función pivote y a la   */
/* derecha los mayores.                          */
/*                                               */
/* Entrada:                                      */
/* int tabla: Tabla a redistribuir               */
/* int ip: Primer elemento de la tabla           */
/* int iu: Ultimo elemento de la tabla           */
/* pivote: Función que elige un pivote           */
/* *ob_count: Contador de OBs                    */
/* Salida:                                       */
/* int: indice del pivote                        */
/*************************************************/
int partir(int* tabla, int ip, int iu, pfunc_pivote
pivote, int *ob_count){

    int m;/*indice pivote*/
    int k;/*elemento pivote*/
    int i;
```

```
    m=pivote(ip,iu);
    k=tabla[m];


    /*Intercambia el pivote con
     * el primer elemento de la tabla*/
    if(swap(ip,m,tabla)==ERR)
        return ERR;


    /*se asigna al indice pivote el primer indice*/
    m=ip;

    /*Itera sobre la tabla a partir del segundo
     elemento hasta el ultimo*/
    for(i=ip+1;i<=iu;i++){

        /*Incremento de las OBs*/
        *ob_count+=1;

        /*si el elemento iterado es menor que el
     elemento pivote*/
        if(tabla[i]<k){
            /*incrementa el indice pivote e
           intercambia*/
            m++;
            if(swap(i,m,tabla)==ERR)
                return ERR;

        }
    }
    /*Retorna el pivote a la posicion ordenada*/
    if(swap(m,ip,tabla)==ERR)
        return ERR;

    return m;

}
```

La función **partir** redistribuye la tabla entre los índices **ip** e **iu** pasados por agumento dejando a la izquierda los elementos menores que el pivote elegido por la función **pivote** y a la derecha los mayores. Devuelve el pivote en caso de que todo vaya bien y **ERR** en caso de error. Además sobreescribe la función pasada por referencia **ob_count** con el número de OBs ejecutadas en la rutina.

Para ilustrar el funcionamiento básico de la rutina **partir** implementada, se muestra a continuación su ejecución sobre una tabla determinada:

Tabla = [3, 6, 5, 2, 4, 1] ip=0 iu=5

T[i] < k	Variables e índices	Tabla
N.A.	m = 0, k = 3, i = 1	[3,6,5,2,4,1]
T[1] < 3	m = 0, k = 3, i = 2	[3,6,5,2,4,1]
T[2] < 3	m = 0, k = 3, i = 3	[3,6,5,2,4,1]
T[3] < 3	m = 1, k = 3, i = 4	[3,2,5,6,4,1]
T[4] < 3	m = 1, k = 3, i = 5	[3,2,5,6,4,1]
T[5] < 3	m = 2, k = 3, i = 5	[3,2,1,6,4,5]

Por último, realiza un **swap** entre el último valor de **m** y el valor de **ip**, dejando el elemento pivote en su posición ordenada, esto es:

swap(2, 0, tabla) → [1,2,3,6,4,5]

```
/**************************************************/
/* Funcion: Quicksort    Fecha: 24/10/2012        */
/* Autores: Adrián Lorenzo Mateo                  */
/*        Ari Handler Gamboa                     */
/*                                                */
/* Funcion que ordena un array de enteros haciendo  */
/* uso del método Quicksort, devolviendo el número  */
/* de OBs durante la ejecución                    */
/*                                                */
/* Entrada:                                       */
/* int tabla: Tabla a ordenar                     */
/* int ip: Primer elemento de la tabla            */
/* int iu: Ultimo elemento de la tabla            */
/* pfunc_pivote pivote: Funcion pivote            */
/* Salida:                                        */
/* int: Número de OBs ejecutadas o                */
/* ERR en caso de error                           */
/**************************************************/
int quicksort(int* tabla, int ip, int iu, pfunc_pivote
pivote){

    int ob_count=0;
    int m;/*indice del pivote*/
    int ret;

    /* Control de errores de los parametros de entrada
      */
    if((pivote==NULL)||(ip<0)||(iu<ip)|| (tabla ==
NULL))
            return ERR;
```

```
    /*si el primer indice es menor que el ultimo*/
    if (ip<iu){
      /*Retorna el pivote y redistribuye los
      elementos*/
         if((m=partir(tabla, ip, iu,
            pivote,&ob_count))==ERR)
             return ERR;

         /*si el primer indice es menor que el indice
            anterior al pivote*/
         if(ip<m-1){

             /*Recursion sobre la tabla izquierda*/
             if((ret=quicksort(tabla, ip, m-1,
                  pivote))==ERR)
                 return ERR;

             ob_count+=ret;
         }
         /*si el indice siguiente al pivote es menor que
            el ultimo*/
         if(m+1<iu){

             /*Recursion sobre la tabla derecha*/
             if((ret=quicksort(tabla,m+1,iu,pivote))
                                              ==ERR)
                 return ERR;

             ob_count += ret;
         }

    }

     return ob_count;
}
```

La función **quicksort** realiza las llamadas a la función **partir** con cada pivote seleccionado, así como las recursiones del algoritmo **Quicksort** sobre las subtablas derecha e izquierda según el pivote elegido por la función **pivote**.

Para ilustrar el funcionamiento básico del algoritmo **Quicksort** implementado, se muestra a continuación su ejecución sobre una tabla determinada, omitiendo los pasos tomados por la función partir, explicada en el trozo de código anterior:

Tabla = [1, 3, 5, 2, 4, 6] ip=0 iu=5

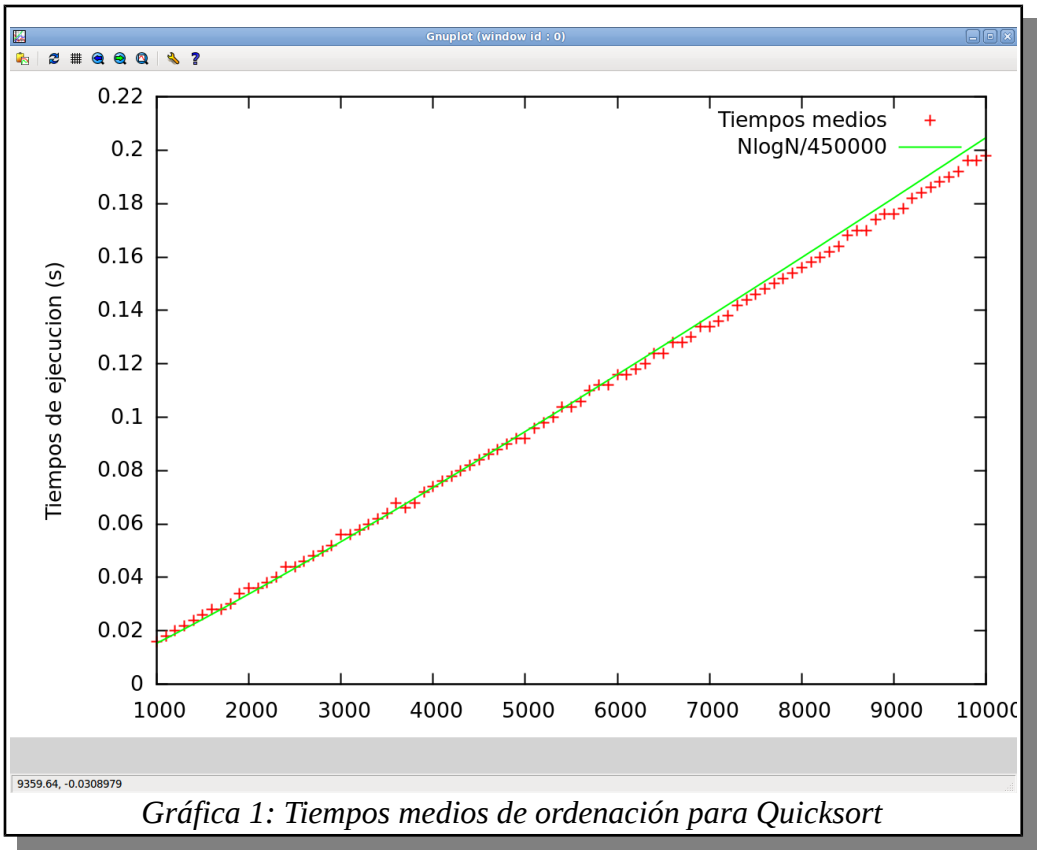
ip	iu	m	Tabla
0	5	N.A.	[1,3,5,2,4,6]
0	5	0	[1,3,5,2,4,6]
Llamada a **partir**			
1	5	1	[1,3,5,2,4,6]
Llamada a **partir**			
2	5	2	[1,2,3,5,4,6]
Llamada a **partir**			
3	5	3	[1,2,3,5,4,6]
Llamada a **partir**			
4	5	4	[1,2,3,4,5,6]
Llamada a **partir**			
5	5	5	[1,2,3,4,5,6]

```
/*****************************************************/
/* Funcion: Quicksort1    Fecha: 24/10/2012          */
/* Autores: Adrián Lorenzo Mateo                     */
/*        Ari Handler Gamboa                          */
/*                                                   */
/* Encapsulamiento de Quicksort para su utilizar     */
/* la funcion medio como pivote                      */
/*                                                   */
/* Entrada:                                          */
/* int tabla: Tabla a ordenar                        */
/* int ip: Primer elemento de la tabla               */
/* int iu: Ultimo elemento de la tabla               */
/* Salida:                                           */
/* int: Número de OBs ejecutadas o                   */
/* ERR en caso de error                              */
/*****************************************************/
int quicksort1(int* tabla, int ip, int iu){
    return quicksort(tabla,ip,iu,medio);
}
```

La función **quicksort1** realiza el encapsulamiento de la función que implementa el algoritmo **Quicksort** para la utilización de la rutina **medio** como función pivote.

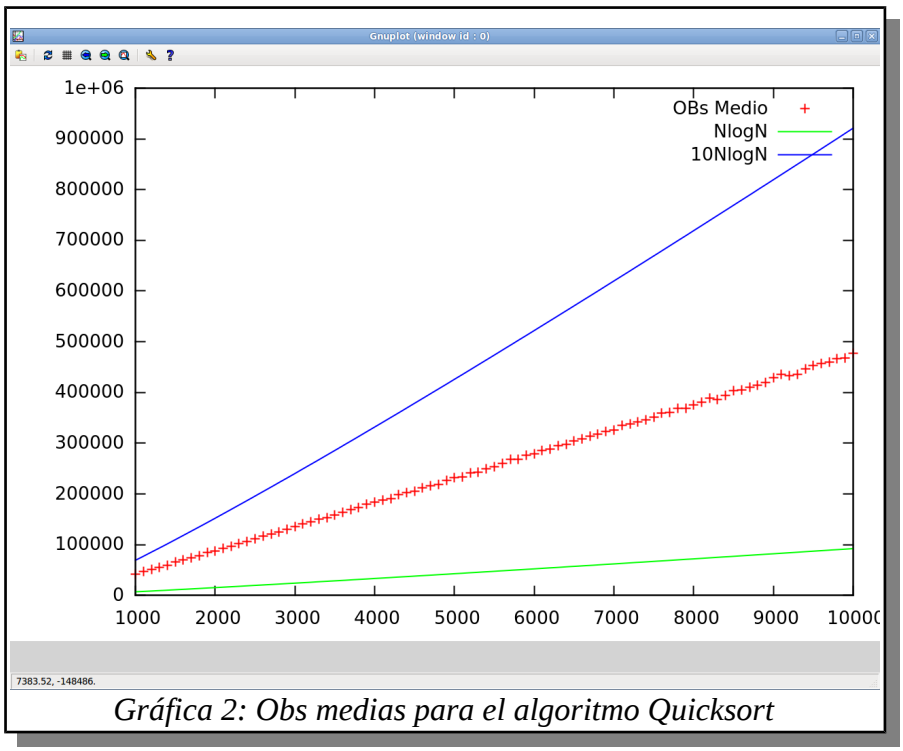
2.3 Resultados

Realizando una ejecución del algoritmo **Quicksort** sobre tamaños de tabla desde 1000 a 10000 elementos con incrementos por iteración de 100 y un retardo de 10 repeticiones de ordenación, obtenemos las gráficas comparativas del caso real con los teóricos en las gráficas de la 1 a la 4.



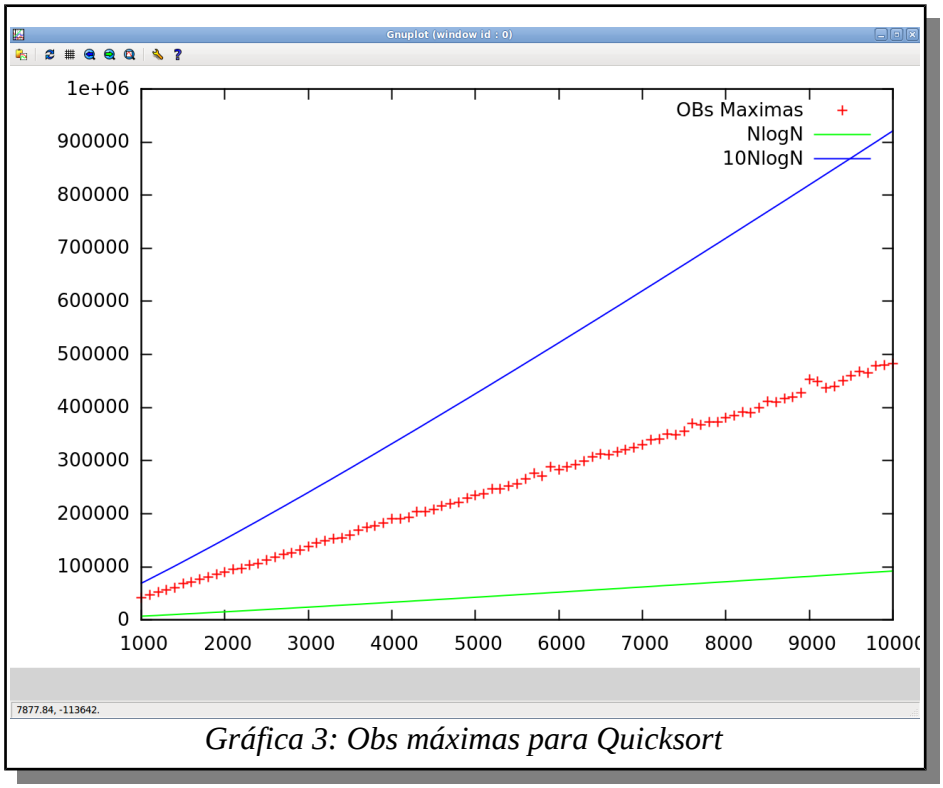
Gráfica 1: Tiempos medios de ordenación para Quicksort

Como el caso medio de **Quicksort** es $A_{QS}(N) = N\log(N)$, representando el resultado práctico frente al teórico divido por una constante de normalización lo suficientemente alta para que la escala en el eje de Y sea común en ambas, obtenemos la Gráfica 1, siendo ésta prácticamente igual a la teórica.



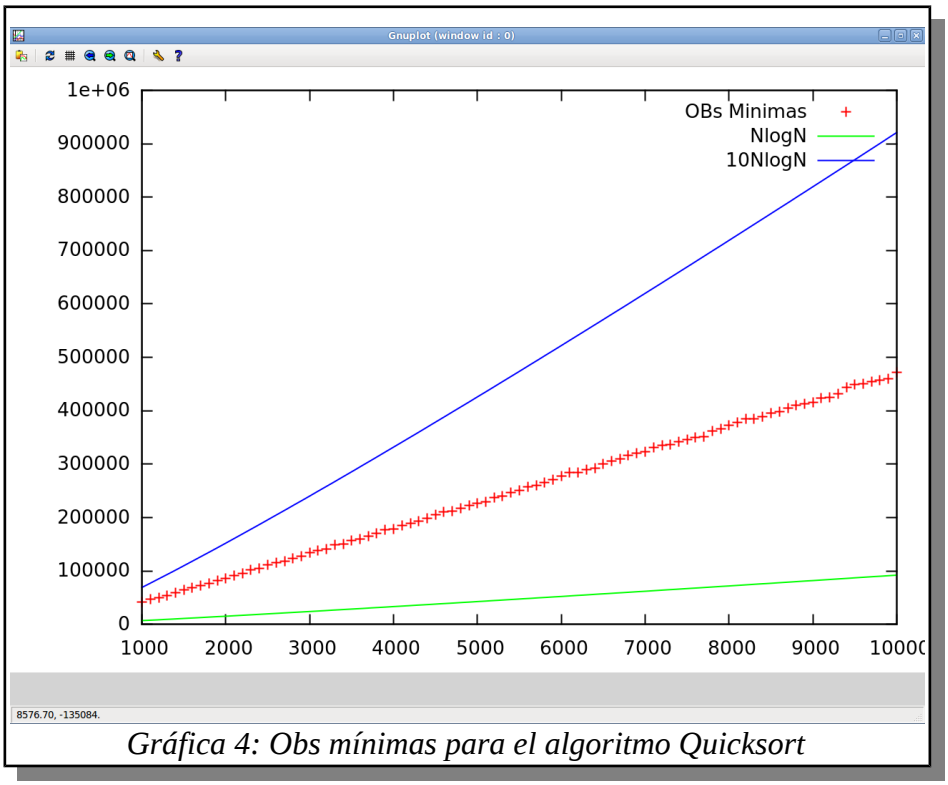
Gráfica 2: Obs medias para el algoritmo Quicksort

Con los mismos datos representamos el número medio de OBs ejecutadas con respecto al tamaño de tabla (Gráfica 2), y encerramos la curva entre las funciones NlogN y 10NlogN, podemos decir que el caso medio para el algoritmo **Quicksort** es asintóticamente equivalente a NlogN, es decir que $A_{QS}(N) = NlogN$.



Gráfica 3: Obs máximas para Quicksort

Con los mismos datos representamos el número máximo de OBs ejecutadas con respecto al tamaño de tabla (Gráfica 3), y encerramos la curva entre las funciones NlogN y 10NlogN, podemos decir que el caso peor para el algoritmo **Quicksort** es asintóticamente equivalente a NlogN, es decir que $W_{QS}(N) = NlogN$.



Gráfica 4: Obs mínimas para el algoritmo Quicksort

Con los mismos datos representamos el número mínimo de OBs ejecutadas con respecto al tamaño de tabla (Gráfica 4), y encerramos la curva entre las funciones NlogN y 10NlogN, podemos decir que el caso mejor para el algoritmo **Quicksort** es asintóticamente equivalente a NlogN, es decir que $B_{QS}(N) = NlogN$.

3. Quicksort con pivote aleatorio

3.1 Introducción

En este bloque se procederá a hacer las pruebas correspondientes sobre el algoritmo de Quicksort con método de pivote **aleat_num** para obtener las gráficas con los resultados de tiempos y número de operaciones básicas.

3.2 Código

```
/************************************************/
/* Funcion: aleat_num Fecha: 29/09/2012         */
/* Autores: Adrián Lorenzo Mateo                */
/*        Ari Handler Gamboa                    */
/*                                              */
/* 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)
{
    /* Control de errores de los parametros de
     entrada*/
     if ((sup < inf) || (sup > RAND_MAX) || ((inf < 0 )
           || (sup < 0)) || (sup > RAND_MAX))
           return ERR;
    /*la función rand() limita el rango [0, 1) debido a
     la división entre RAND_MAX+1.
     * La segunda parte multiplica dicho valor por un
     número que se encuentre aproximadamente
     * en la mitad del rango (sup-inf +1), de este modo
     se normaliza el anterior valor
     * acotándolo entre 0 y el número intermedio.  Sin
     embargo, es necesario sumar el límite
     * inferior al resultado con el fin de que se
     obtenga un valor superior a este.*/
    return (int)((rand()/(RAND_MAX+1.))*(sup-inf+1))
     +inf;
}
```

Esta rutina genera un número aleatorio comprendido entre los límites inferior y superior pasados como parámetros.

```
/************************************************/
/* Funcion: Quicksort2    Fecha: 31/10/2012     */
/* Autores: Adrián Lorenzo Mateo                */
/*        Ari Handler Gamboa                    */
/*                                              */
/* Encapsulamiento de Quicksort para utilizar   */
/* la funcion aleat_num como pivote             */
/*                                              */
/* Entrada:                                     */
/* int tabla: Tabla a ordenar                   */
/* int ip: Primer elemento de la tabla          */
/* int iu: Ultimo elemento de la tabla          */
/* Salida:                                      */
```

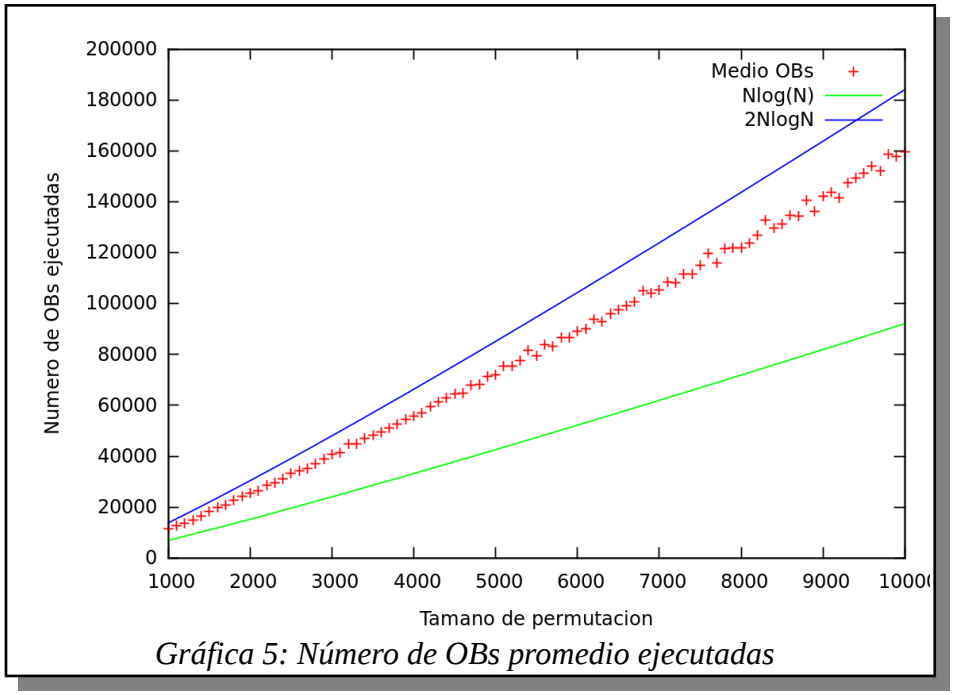
```
/* int: Número de OBs ejecutadas o                    */
/* ERR en caso de error                               */
/*****************************************************/
int quicksort2(int* tabla, int ip, int iu){
    return quicksort(tabla,ip,iu,aleat_num);
}
```

Función **quicksort2** con **aleat_num** como función pivote.

3.3 Resultados

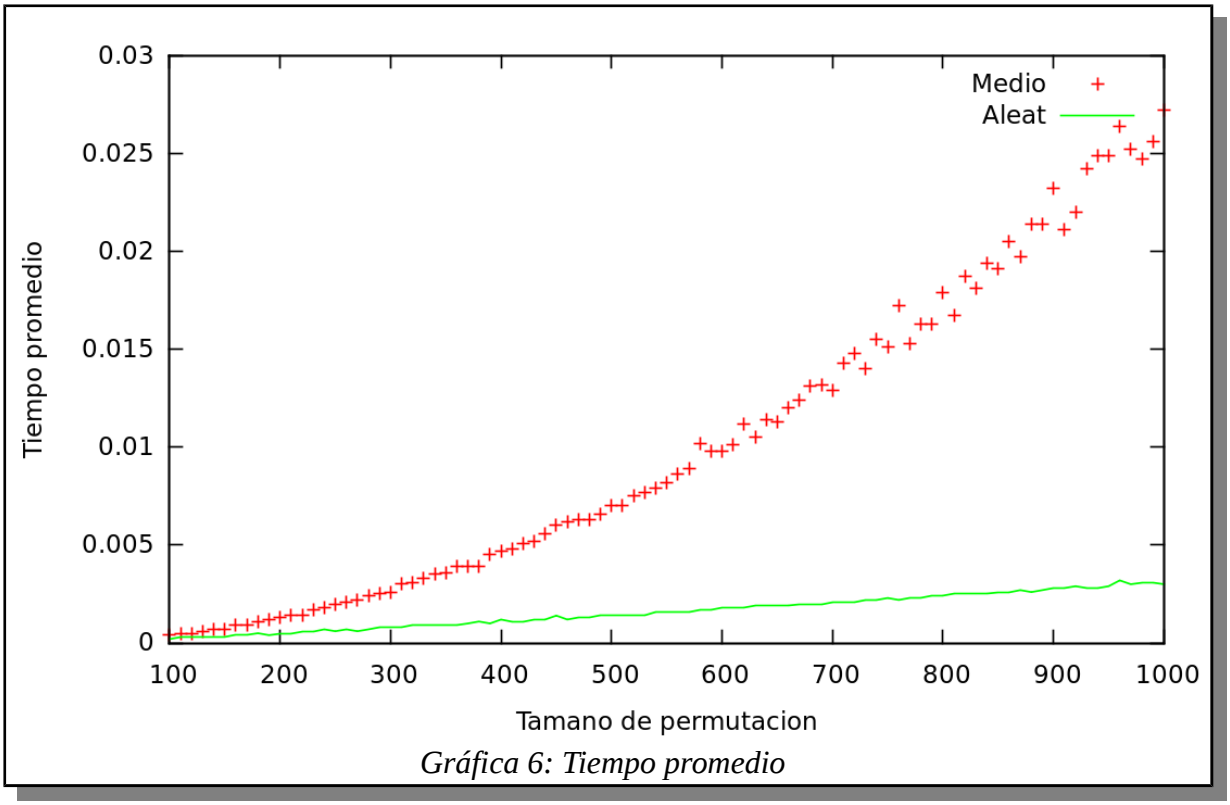
```
Practica numero 2, apartado 4
Realizada por: Ari Handler Gamboa y Adrian Lorenzo
Mateo
Grupo: 261


Permutacion desordenada:
9       0       8       5       2       3       6
1       4       7

Permutacion ordenada:
0       1       2       3       4       5       6
7       8       9
```

Mediante el ejercicio 4 se puede comprobar que realiza correctamente la ordenación y puesto que es un método recursivo no es necesario hacer pruebas de fuga de memoria.



Gráfica 5: Número de OBs promedio ejecutadas

Con los datos obtenidos en el ejercicio 5 representamos el número medio de OBs ejecutadas con respecto al tamaño de tabla (Gráfica 5), y encerramos la curva entre las funciones NlogN y 2NlogN, podemos decir que el caso mejor para el algoritmo Quicksort es asintóticamente equivalente a NlogN, es decir que BQS(N) = NlogN.



Gráfica 6: Tiempo promedio

Al comparar las dos funciones en la gráfica 6 se puede concluir que el algoritmo Quicksort con método de pivote aleatorio es ampliamente superior en rendimiento respecto a la utilización del pivote medio.

4. Quicksort sin recursión de cola

4.1 Introducción

En este bloque se procederá a hacer las pruebas correspondientes sobre el algoritmo de Quicksort sin recursión de cola para obtener las gráficas con los resultados de tiempos y número de operaciones básicas. Una función de recursión de cola es aquella que finalizan con una llamada recursiva, es decir aquella que termina de forma recursiva en el return.

4.2 Código

```
/**************************************************/
/* Funcion: Quicksort_src    Fecha: 4/11/2012      */
/* Autores: Adrián Lorenzo Mateo                   */
/*       Ari Handler Gamboa                        */
/*                                                 */
/* Quicksort sin recursion de cola para evitar el  */
/* overhead.                                       */
/*                                                 */
/* Entrada:                                        */
/* int tabla: Tabla a ordenar                      */
/* int ip: Primer elemento de la tabla             */
/* int iu: Ultimo elemento de la tabla             */
/* Salida:                                         */
/* int: Número de OBs ejecutadas o                 */
/* ERR en caso de error                            */
/**************************************************/
int quicksort_src(int* tabla, int ip, int iu)
{
    int ob_count=0;
    int m;/*indice del pivote*/
    int ret;

    /* Control de errores de los parametros de
     entrada*/
    if((ip<0)|| (tabla == NULL))
            return ERR;

    if((ip == iu) || (ip>iu))
        return ob_count;

    /*mientras el primer indice es menor que el
     ultimo*/
    while (ip<iu){

        /*Retorna el pivote y redistribuye los
           elementos*/
        if((m=partir(tabla, ip, iu,
           medio,&ob_count))==ERR)
            return ERR;

        ob_count += (iu-ip);


        /*Recursion sobre la tabla izquierda*/
        if((ret=quicksort_src(tabla, ip, m-1))==ERR)
            return ERR;

        ob_count+=ret;

        ip = m+1;

    }

     return ob_count;
}
```

Para evitar la recursión de cola en el algoritmo Quicksort (recursión de cola se refiere a que no se ejecuta ninguna operación después del retorno de una función recursiva) se ha procedido a insertar la función partir y la llamada recursiva dentro de un bucle while.

Además la recursión se hace ahora únicamente sobre la subtabla izquierda resultante de la salida de partir, incrementando el primer indice en cada iteración del bucle a razón de uno más la última posición del pivote.

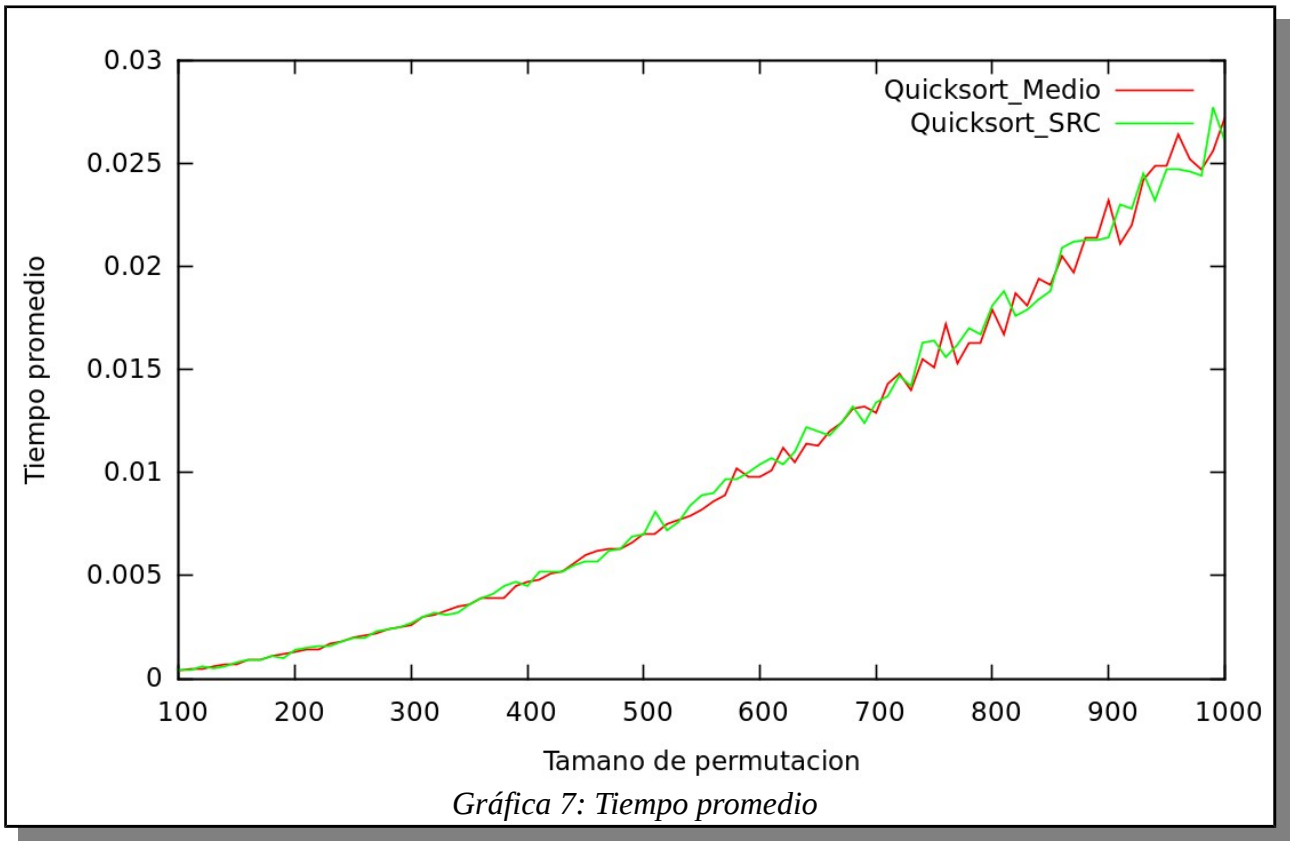
De esta manera partir va a operar sobre tablas de ip a iu, luego de ip+1 a iu, y así sucesivamente.

Para evitar que sufra un error al tener índice último negativo o que los índices primero y último coincidan, se ha añadido una comprobación previa de estas condiciones, retornando 0 operaciones básicas ejecutadas en estos casos.

3.3 Resultados

```
Practica numero 2, apartado 4
Realizada por: Ari Handler Gamboa y Adrian Lorenzo
Mateo
Grupo: 261


Permutacion desordenada:
6        8        4        3        2        5        1
9        7        0

Permutacion ordenada:
0        1        2        3        4        5        6
7        8        9
```

Mediante el ejercicio 4 se puede comprobar que realiza correctamente la ordenación y puesto que es un método recursivo no es necesario hacer pruebas de fuga de memoria.



Gráfica 7: Tiempo promedio

Al comparar las dos funciones en la gráfica 7 se puede comprobar que la eliminación de la recursión de cola no produce ninguna mejora significativa respecto al Quicksort original, excepto en algunos casos puntuales.