Análisis de Algoritmos 2025/2026

Práctica 2

Alina Datsko Yaskerska, Lucas Manuel Blanco Rodríguez Grupo 1201

Código	Gráficas	Memoria	Total

Índice

Introducción…….. 3

Objetivos…….. 4

Herramientas y metodologías…….. 6

Código fuente…….. 9

Resultados, Gráficas…….. 20

Conclusiones finales…….. 39

1. Introducción.

En esta práctica vamos a ver de forma experimental el comportamiento de dos nuevos algoritmos de ordenación, el MergeSort y el QuickSort, los que comúnmente se denominan “divide y vencerás”. A diferencia de los estudiados en la anterior práctica, ambos dividen el array en numerosos subarrays, que se ordenarán usando funciones recursivas.

El objetivo es comparar lo que dice la teoría de sobre su complejidad O(nlogn), además de analizar y comprar sus mejores, medios y peores casos. Para ver todo esto se implementarán en primer lugar versiones básicas de ambos algoritmos en C. Además, en el caso del QuickSort se estudia también cómo la elección del pivote altera el número de OBs del algoritmo y su tiempo de ejecución. En esta práctica se tomarán tres casos: eligiendo como pivote el primer elemento, el elemento central, y la mediana entre el primer elemento, el central y el final.

Finalmente, estos resultados experimentales los mostraremos a través de gráficas y discutiremos los resultados obtenidos, además de analizar cómo llegar a los mejores y peores casos de cada algoritmo.

2. Objetivos

2.1 Apartado 1: Implementación MergeSort

En este apartado hay que implementar en sorting.c el algoritmo de ordenación MergeSort siguiendo el prototipo dado por las diapositivas de la parte teórica de la asignatura. Para ello se deben escribir dos funciones: la principal: mergesort(int *tabla, int ip, int iu) y la función de combinación: merge(int *tabla, int ip, int iu, int imedio), la que debe fusionar dos mitades ordenadas. Ambas funciones deben devolver el número de obs realizadas o ERR si ocurre algún error. Además, se debe comprobar su funcionamiento correcto empleando el ejercicio 4.

2.2 Apartado 2: Análisis MergeSort

En este apartado hay que modificar el exercise5.c para que utilice MergeSort y así poder analizar y medir su comportamiento. Más específicamente, se deben tener las tablas con el tiempo medio de ejecución y con el número de operaciones básicas (mínimo, máximo y promedio) para los distintos tamaños de permutación. Después, esos valores se verán en gráficas y se compararán y analizarán.

2.3 Apartado 3: Implementación Quicksort

En este apartado hay que implementar en sorting.c el algoritmo de ordenación QuickSort siguiendo el prototipo indicado en la parte teórica de la asignatura. Se debe escribir la función principal quicksort(int *tabla, int ip, int iu), que ordenará el tramo de la tabla entre las posiciones ip(fisrt) e iu(last). Además, es necesario implementar las funciones de apoyo partition(int *tabla, int ip, int iu, int *pos) y median(int *tabla, int ip, int iu, int *pos), que devuelven el número de OBs o ERR y dejan en pos la posición del pivote. En esta primera versión el pivote será siempre el primer elemento. Igual que en el caso anterior, se comprobará su funcionamiento modificando el exercise4.c.

2.4 Apartado 4: Análisis QuickSort

En este apartado hay que modificar el programa exercise5.c para que, en lugar del MergeSort, se utilice ahora el QuickSort añadido. Con esto, se deben obtener las tablas de tiempo medio de ejecución y de operaciones básicas (mínimo, máximo y promedio) para los distintos tamaños de permutación. Después, esos valores se verán en gráficas y se compararán y analizarán.

2.5 Apartado 5: Implementación y análisis funciones pivote

En este apartado hay que añadir dos nuevas funciones de selección de pivote para QuickSort. La primera, median_avg, debe devolver la posición central de la tabla. La segunda, median_stat, debe realizar una mediana de tres elementos, es decir, compara los valores en las posiciones inicial, central y final y devuelve la posición del valor intermedio, sumando también las OBs que suponga esa comparación. Después, la función partition debe modificarse para poder usar las nuevas funciones de elección de pivote y contar así sus OBs. Finalmente, se compararán en el apartado de gráficos tanto los tiempos y como las operaciones básicas obtenidas con cada una de las tres funciones.

3. Herramientas y metodología

Por un lado, un integrante de la pareja ha estado desarrollando su trabajo tanto en entorno MacOS como Linux. Usaba el primero para programar en VSCode, y posteriormente, gracias a una máquina virtual, compila los archivos en Linux. En este último entorno también usó otras herramientas como Valgrind y Gnuplot.

Por otro lado, el segundo integrante empleó un ordenador Windows, operando a la vez con una máquina virtual con Linux en ella. Para la toma de resultados de las gráficas se ha usado el ordenador del segundo integrante.

3.1 Apartado 1: Implementación MergeSort

En este apartado se ha seguido la estructura teórica del MergeSort: dividir, ordenar y mezclar. La función mergesort empieza comprobando con assert los parámetros. Si la tabla que entra tiene un solo elemento, no hay nada que ordenar y se devuelve 0 operaciones básicas.

Cuando el tramo tiene más de un elemento, se calcula el punto medio y se ordenan recursivamente las dos mitades: primero la izquierda (ip a imedio) y luego la derecha (imedio+1 a iu). Cada llamada devuelve el número de comparaciones que ha hecho y, si alguna falla, se devuelve ERR. Una vez que las dos mitades están ordenadas, se llama a merge, que es la función que junta las dos partes en orden. El total de operaciones básicas es la suma de las obs de la izquierda, la derecha y la fusión de ambas tablas.

En el caso de la función merge, reserva un array auxiliar del tamaño del tramo de tabla que se va a mezclar y usa dos índices: i recorre la mitad izquierda (ip hasta imedio) y j recorre la mitad derecha (imedio+1 hasta iu). Mientras queden elementos en las dos mitades, se compara tabla[i] con tabla[j] (la comparación es la OB) y se copia al auxiliar el más pequeño. Cuando una de las mitades ya no tiene elementos, se copian directamente los que quedan en la otra. Al final, copiamos el contenido del array auxiliar de vuelta al array original y se libera la memoria del array auxiliar.

3.2 Apartado 2: Análisis MergeSort

En este apartado se modificó exercise5.c para que ejecutara nuestro mergesort empleando permutaciones de distintos tamaños y guardara en un fichero log los datos de interés: tiempo medio de ejecución y número de operaciones básicas (media, mínimo y máximo). A partir de esos datos se añadieron al Makefile comandos que usan gnuplot para generar las gráficas de tiempo de ejecución y de las OBs en función de N.

Además, en el Makefile se incluyeron otros comandos específicos para representar en los casos mejor y peor del MergeSort. Todo esto nos permite ver cómo es su crecimiento O(nlogn) en un entorno práctico. La forma en la que se ha planteado e implementado las funciones que generan el mejor y peor caso se describirán más adelante.

3.3 Apartado 3: Implementación QuickSort

El quicksort que se ha hecho sigue la idea vista en clase: se coge un pivote, se coloca todo lo que es menor que él a su izquierda y lo demás a su derecha, y luego se aplica lo mismo a cada lado. La función comprueba primero que el array y los índices son válidos y, si el tramo tiene 0 o 1 elemento, termina ahí porque ya está ordenado. Si no, llama a partition, suma las comparaciones que ha hecho esa partición y después llama recursivamente a la parte izquierda y a la derecha, acumulando también sus OBs.

La función partition lo primero que hace es decidir el pivote (en esta versión de la función usamos el primer elemento). Luego recorre el resto de la tabla y va contando comparaciones con el pivote. Cada vez que encuentra un elemento más pequeño que el pivote lo va colocando al principio. Al final, cuando ya ha separado menores y mayores, coloca el pivote justo en medio, en su sitio definitivo, y devuelve esa posición. Así quicksort ya sabe dónde partir el array y seguir ordenando.

3.4 Apartado 4: Análisis QuickSort

En este apartado se crearon nuevos exercise5.c para que, en lugar de llamar a MergeSort, ejecutara nuestro QuickSort sobre permutaciones de distintos tamaños y fuera guardando en un fichero los valores que nos interesan: tiempo medio de ejecución y número de operaciones básicas (media, mínimo y máximo) para cada N. Igual que antes, esos datos se guardan en un .log y se añadieron al Makefile comandos que llaman a gnuplot para dibujar las gráficas de tiempo y de OBs en función del tamaño de la permutación.

También se encuentran en el Makefile comandos hechos para representar el mejor y el peor caso de nuestro QuickSort (usando las permutaciones que hemos construido para forzarlos y en este caso solo para el caso de primer elemento como pivote, ya que para las demás elecciones de pivote sería bastante complejo). De esta forma se puede ver claramente que, aunque el caso medio se acerca al comportamiento asintótico teórico que es O(nlogn), el caso peor forzado se aleja mucho de un comportamiento de este tipo.

3.5 Apartado 5: Implementación y análisis funciones pivote

En este apartado se añadieron dos nuevas formas de elegir el pivote en QuickSort. La primera, median_avg, toma simplemente el elemento central (ip + iu)/2, la segunda, median_stat, aplica la elección basada en una mediana, comparando el primer elemento, el central y el último para quedarse con el valor intermedio y además cuenta esas comparaciones y las devuelve para tenerlas en cuenta a la hora de graficar.

Para usarlas sin tocar la versión básica se crearon funciones paralelas: quicksort_avg/partition_avg y quicksort_stat/partition_stat. Todas siguen el mismo patrón: elegir pivote con la función correspondiente, moverlo al principio y hacer la partición contando las OBs. Así se pueden medir y comparar los tiempos y operaciones de QuickSort con los tres pivotes distintos.

4. Código fuente

4.1 Apartado 1

```c
/*****************************************************/
/* Function: MergeSort                               */
/* Date: 03-10-2025                                  */
/* Authors: Lucas Manuel Blanco Rodríguez            */
/*                                                   */
/* Routine that sorts an array of integers           */
/* using the Merge Sort algorithm.                   */
/*                                                   */
/* Input:                                            */
/* int *tabla: pointer to  array to be sorted        */
/* int ip: index of the first element                */
/* int iu: index of the last element                 */
/*                                                   */
/* Output:                                           */
/* int: number of OBs made                           */
/* during the sorting process                        */
/* or ERR in case of problem                         */
/*****************************************************/
int mergesort(int* tabla, int ip, int iu) {
  int imedio = 0, ob_iz = 0, ob_dc = 0, ob_merge = 0, ob_total = 0;

  assert(tabla != NULL);
  assert(ip >= 0);
  assert(iu >= 0);
  assert(ip <= iu);

  if (ip == iu) return 0;
  imedio = (ip + iu)/2;

  ob_iz = mergesort(tabla, ip, imedio);
  if (ob_iz == ERR) return ERR;

  ob_dc = mergesort(tabla, imedio+1, iu);
  if (ob_dc == ERR) return ERR;

  ob_merge = merge(tabla, ip, iu, imedio);
  if (ob_merge == ERR) return ERR;

  ob_total = ob_iz + ob_dc + ob_merge;
  return ob_total;
}
```

```c
/*****************************************************************/
/* Merges the two sorted subarrays [ip, imedio] and [imedio+1, iu] */
/* into tabla, counting comparisons during the merge,              */
/* and returns the number of comparisons made.                     */
/*****************************************************************/
int merge(int* tabla, int ip, int iu, int imedio) {
  int i = ip, j = imedio+1, k = 0, ob = 0, *aux = NULL;

  assert(tabla != NULL);
  assert(ip >= 0);
  assert(iu >= 0);
  assert(imedio >= 0);


  aux = malloc((iu-ip+1)*sizeof(aux[0]));
  if (aux == NULL) return ERR;

  while (i <= imedio && j <= iu) {
    ob++;
    if (tabla[i] <= tabla[j]) {
      aux[k] = tabla[i];
      i++;
    }
    else {
      aux[k] = tabla[j];
      j++;
    }
    k++;
  }

  while (i <= imedio) {
    aux[k] = tabla[i];
    i++;
    k++;
  }

  while (j <= iu) {
    aux[k] = tabla[j];
    j++;
    k++;
  }

  for (i=0; i<(iu-ip+1); i++) {
    tabla[ip+i] = aux[i];
  }
  free(aux);
  return ob;
}
```

4.3 Apartado 3

```c
/**************************************************/
/* Function: QuickSort                            */
/* Date: 03-10-2025                               */
/* Authors: Lucas Manuel Blanco Rodríguez         */
/*                                                */
/* Routine that sorts an array of integers        */
/* using the Quick Sort algorithm.                */
/* and using the first element as pivot.          */
/*                                                */
/* Input:                                         */
/* int *tabla: pointer to  array to be sorted     */
/* int ip: index of the first element             */
/* int iu: index of the last element              */
/*                                                */
/* Output:                                        */
/* int: number of OBs made                        */
/* during the sorting process                     */
/* or ERR in case of problem                      */
/**************************************************/
int quicksort(int *tabla, int ip, int iu)
{
  int ob_total = 0;
  int ob_partir = 0, ob_izquierda = 0, ob_derecha = 0;
  int pos = 0;

  assert(tabla != NULL);
  assert(ip >= 0);
  assert(iu >= 0);
  assert(ip <= iu);

  if (ip >= iu) return 0;
  ob_partir = partition(tabla, ip, iu, &pos);
  if (ob_partir == ERR) return ERR;
  ob_total += ob_partir;

  if (pos - 1 >= ip)
  {
    ob_izquierda = quicksort(tabla, ip, pos - 1);
    if (ob_izquierda == ERR) return ERR;
    ob_total += ob_izquierda;
  }

  if (pos + 1 <= iu)
  {
    ob_derecha = quicksort(tabla, pos + 1, iu);
    if (ob_derecha == ERR) return ERR;
    ob_total += ob_derecha;
  }

  return ob_total;
}
```

```c
/*****************************************************************/
/* Partitions the subarray [ip, iu] using the first element as pivot,   */
/* moves smaller elements to the left and larger to the right,          */
/* and returns the number of comparisons performed.                     */
/*****************************************************************/
int partition(int *tabla, int ip, int iu, int *pos)
{
  int ob = 0;
  int ob_piv = 0;
  int pivot_val, i, j, tmp;

  assert(tabla != NULL);
  assert(ip >= 0);
  assert(iu >= 0);
  assert(ip <= iu);
  assert(pos != NULL);

  if (ip == iu)
  {
    *pos = ip;
    return 0;
  }

  ob_piv = median(tabla, ip, iu, pos);
  if (ob_piv == ERR) return ERR;
  ob += ob_piv;

  if (*pos != ip)
  {
    tmp = tabla[ip];
    tabla[ip] = tabla[*pos];
    tabla[*pos] = tmp;
  }

  pivot_val = tabla[ip];
  i = ip + 1;

  for (j = ip + 1; j <= iu; j++)
  {
    ob++;
    if (tabla[j] < pivot_val)
    {
      tmp = tabla[i];
      tabla[i] = tabla[j];
      tabla[j] = tmp;
      i++;
    }
  }

  *pos = i - 1;

  tmp = tabla[ip];
  tabla[ip] = tabla[*pos];
  tabla[*pos] = tmp;

  return ob;
}
```

```c
/*Select the first element as pivot*/
int median(int *tabla, int ip, int iu, int *pos)
{
  assert(tabla != NULL);
  assert(ip >= 0);
  assert(iu >= 0);
  assert(ip <= iu);
  assert(pos != NULL);

  *pos = ip;

  return 0;
}
```

4.5 Apartado 5

```c
/*Select the median of first, central and last elements as pivot*/
int median_stat(int *tabla, int ip, int iu, int *pos)
{
  int imedio, comps, a, b, c;

  assert(tabla != NULL);
  assert(ip >= 0);
  assert(iu >= 0);
  assert(ip <= iu);
  assert(pos != NULL);

  imedio = (ip + iu) / 2;
  comps = 0;
  a = tabla[ip];
  b = tabla[imedio];
  c = tabla[iu];

  comps++;
  if (a < b)
  {
    comps++;
    if (b < c)
    {
      *pos = imedio;
    }
    else
    {
      comps++;
      if (a < c)
      {
        *pos = iu;
      }
      else
      {
        *pos = ip;
      }
    }
  }
  else
  {
    comps++;
    if (a < c)
    {
      *pos = ip;
    }
    else
    {
      comps++;
      if (b < c)
      {
        *pos = iu;
      }
      else
      {
        *pos = imedio;
      }
    }
  }

  return comps;
}
```

```c
/*Select the central element as pivot*/
int median_avg(int *tabla, int ip, int iu, int *pos)
{
  assert(tabla != NULL);
  assert(ip >= 0);
  assert(iu >= 0);
  assert(ip <= iu);
  assert(pos != NULL);

  *pos = (ip + iu) / 2;
  return 0;
}
```

```c
/**************************************************/
/* Function: QuickSort (stat)                     */
/* Date: 03-10-2025                               */
/* Authors: Lucas Manuel Blanco Rodríguez         */
/*                                                */
/* Routine that sorts an array of integers        */
/* using the Quick Sort algorithm.                */
/* and using the median of                        */
/* first, central and last elements as pivot.     */
/*                                                */
/* Input:                                         */
/* int *tabla: pointer to  array to be sorted     */
/* int ip: index of the first element             */
/* int iu: index of the last element              */
/*                                                */
/* Output:                                        */
/* int: number of OBs made                        */
/* during the sorting process                     */
/* or ERR in case of problem                      */
/**************************************************/
int quicksort_stat(int *tabla, int ip, int iu)
{
  int ob_total = 0;
  int ob_partir = 0, ob_izquierda = 0, ob_derecha = 0;
  int pos = 0;

  assert(tabla != NULL);
  assert(ip >= 0);
  assert(iu >= 0);
  assert(ip <= iu);

  if (ip >= iu) return 0;

  ob_partir = partition_stat(tabla, ip, iu, &pos);
  if (ob_partir == ERR) return ERR;
  ob_total += ob_partir;

  if (pos - 1 >= ip)
  {
    ob_izquierda = quicksort_stat(tabla, ip, pos - 1);
    if (ob_izquierda == ERR) return ERR;
    ob_total += ob_izquierda;
  }

  if (pos + 1 <= iu)
  {
    ob_derecha = quicksort_stat(tabla, pos + 1, iu);
    if (ob_derecha == ERR) return ERR;
    ob_total += ob_derecha;
  }

  return ob_total;
}
```

```c
/*****************************************************************/
/* Partitions the subarray [ip, iu] using the median of first, central, */
/* and last elements as pivot,  moves smaller elements to the left and  */
/* larger to the righ and returns the number of comparisons performed.  */
/*****************************************************************/
int partition_stat(int *tabla, int ip, int iu, int *pos)
{
  int ob = 0;
  int ob_piv = 0;
  int pivot_val, i, j, tmp;

  assert(tabla != NULL);
  assert(ip >= 0);
  assert(iu >= 0);
  assert(ip <= iu);
  assert(pos != NULL);

  if (ip == iu)
  {
    *pos = ip;
    return 0;
  }

  ob_piv = median_stat(tabla, ip, iu, pos);
  if (ob_piv == ERR) return ERR;
  ob += ob_piv;

  if (*pos != ip)
  {
    tmp = tabla[ip];
    tabla[ip] = tabla[*pos];
    tabla[*pos] = tmp;
  }

  pivot_val = tabla[ip];
  i = ip + 1;

  for (j = ip + 1; j <= iu; j++)
  {
    ob++;
    if (tabla[j] < pivot_val)
    {
      tmp = tabla[i];
      tabla[i] = tabla[j];
      tabla[j] = tmp;
      i++;
    }
  }

  *pos = i - 1;

  tmp = tabla[ip];
  tabla[ip] = tabla[*pos];
  tabla[*pos] = tmp;
  return ob;
}
```

```c
/**************************************************/
/* Function: QuickSort (avg)                      */
/* Date: 03-10-2025                               */
/* Authors: Lucas Manuel Blanco Rodríguez         */
/*                                                */
/* Routine that sorts an array of integers        */
/* using the Quick Sort algorithm.                */
/* and using the central element as pivot.        */
/*                                                */
/* Input:                                         */
/* int *tabla: pointer to  array to be sorted     */
/* int ip: index of the first element             */
/* int iu: index of the last element              */
/*                                                */
/* Output:                                        */
/* int: number of OBs made                        */
/* during the sorting process                     */
/* or ERR in case of problem                      */
/**************************************************/
int quicksort_avg(int *tabla, int ip, int iu)
{
  int ob_total = 0;
  int ob_partir = 0, ob_izquierda = 0, ob_derecha = 0;
  int pos = 0;

  assert(tabla != NULL);
  assert(ip >= 0);
  assert(iu >= 0);
  assert(ip <= iu);

  if (ip >= iu) return 0;

  ob_partir = partition_avg(tabla, ip, iu, &pos);
  if (ob_partir == ERR) return ERR;
  ob_total += ob_partir;

  if (pos - 1 >= ip)
  {
    ob_izquierda = quicksort_avg(tabla, ip, pos - 1);
    if (ob_izquierda == ERR) return ERR;
    ob_total += ob_izquierda;
  }
  if (pos + 1 <= iu)
  {
    ob_derecha = quicksort_avg(tabla, pos + 1, iu);
    if (ob_derecha == ERR) return ERR;
    ob_total += ob_derecha;
  }

  return ob_total;
}
```

```c
/*****************************************************************/
/* Partitions the subarray [ip, iu] using the central element as pivot, */
/* moves smaller elements to the left and larger to the right,          */
/* and returns the number of comparisons performed.                     */
/*****************************************************************/
int partition_avg(int *tabla, int ip, int iu, int *pos)
{
  int ob = 0;
  int ob_piv = 0;
  int pivot_val, i, j, tmp;

  assert(tabla != NULL);
  assert(ip >= 0);
  assert(iu >= 0);
  assert(ip <= iu);
  assert(pos != NULL);

  if (ip == iu)
  {
    *pos = ip;
    return 0;
  }

  ob_piv = median_avg(tabla, ip, iu, pos);
  if (ob_piv == ERR) return ERR;
  ob += ob_piv;

  if (*pos != ip)
  {
    tmp = tabla[ip];
    tabla[ip] = tabla[*pos];
    tabla[*pos] = tmp;
  }

  pivot_val = tabla[ip];
  i = ip + 1;

  for (j = ip + 1; j <= iu; j++)
  {
    ob++;
    if (tabla[j] < pivot_val)
    {
      tmp = tabla[i];
      tabla[i] = tabla[j];
      tabla[j] = tmp;
      i++;
    }
  }

  *pos = i - 1;

  tmp = tabla[ip];
  tabla[ip] = tabla[*pos];
  tabla[*pos] = tmp;

  return ob;
}
```

5. Resultados, Gráficas

5.1 Apartado 1

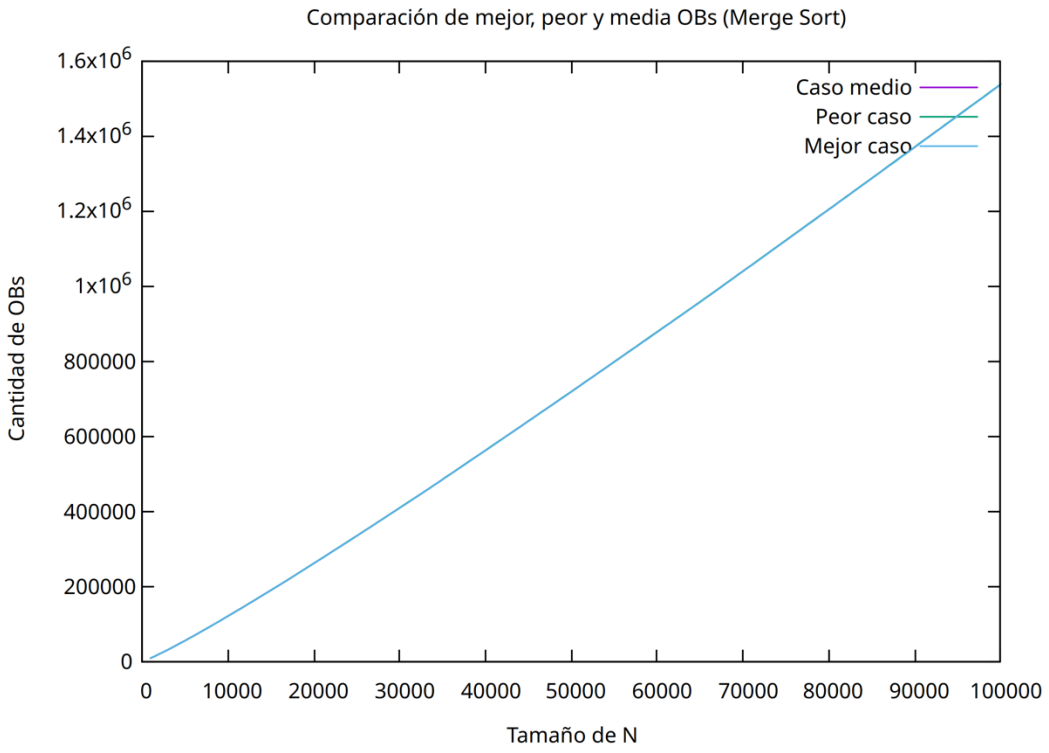
```
Running exercise4
Practice number 1, section 4
Done by: Lucas Blanco y Alina Datsko
Group: 1201
Ordenando empleando MergeSort
1       2       3       4       5       6       7       8       9       10      11      1
2       13      14      15      16      17      18      19      20      21      22      2
3       24      25      26      27      28      29      30      31      32      33      3
4       35      36      37      38      39      40      41      42      43      44      4
5       46      47      48      49      50      51      52      53      54      55      5
6       57      58      59      60      61      62      63      64      65      66      6
7       68      69      70      71      72      73      74      75      76      77      7
8       79      80      81      82      83      84      85      86      87      88      8
9       90      91      92      93      94      95      96      97      98      99      1
00
```

En esta imagen se muestra la ejecución del ejercicio 4: se ordena una tabla de 100 elementos utilizando el algoritmo MergeSort para comprobar que la implementación funciona correctamente. Con un tamaño tan pequeño no se aprecia una diferencia grande de tiempo respecto a QuickSort, e incluso puede parecer similar a métodos menos eficientes como InsertSort o BubbleSort, a pesar de que estos últimos son $O(n^2)$. Esto es normal porque la práctica, en este ejercicio, busca confirmar el funcionamiento y no su rendimiento.

En los apartados siguientes, con tablas más grandes y usando las gráficas, sí se podrá ver mejor el crecimiento asintótico de cada algoritmo.

5.2 Apartado 2

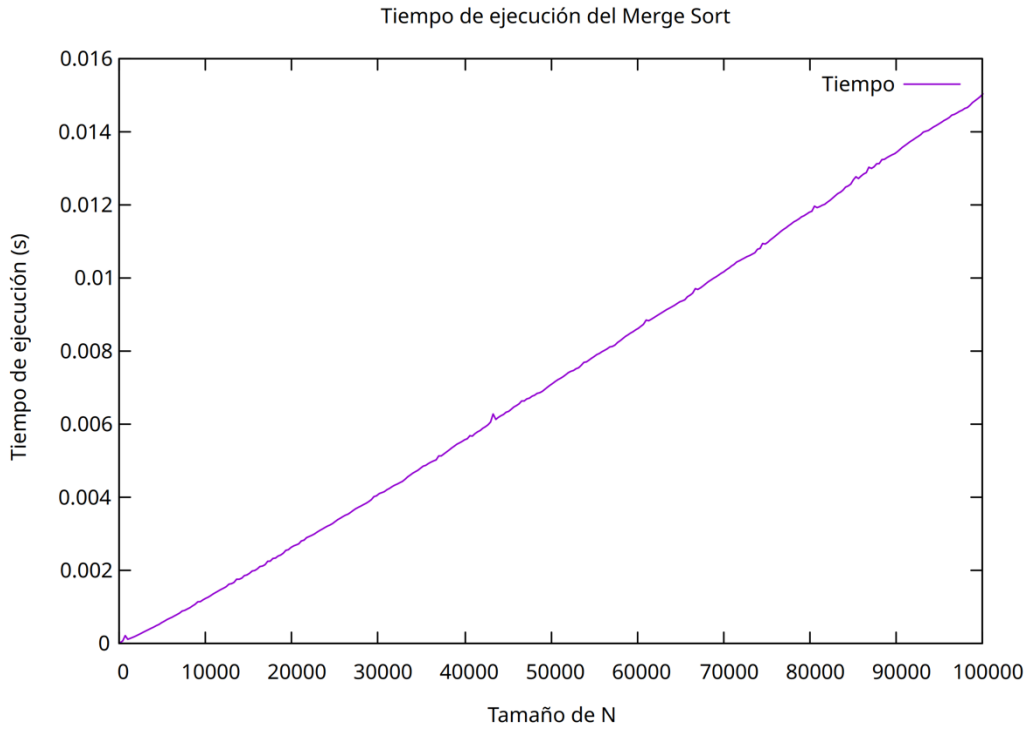
Gráfica comparando los tiempos mejor peor y medio en OBs para MergeSort, comentarios a la gráfica.



En esta gráfica se comparan las OBs del MergeSort en los tres escenarios (mejor, medio y peor caso). Lo primero que se aprecia es que las tres curvas prácticamente se solapan: el algoritmo hace casi el mismo número de operaciones básicas en cualquier situación. Eso confirma que MergeSort es muy estable y que su coste no depende apenas del orden inicial de la tabla.

Aunque en la imagen no se distingan las líneas, en el fichero de datos sí se ve que hay pequeñas diferencias. Para N= 100.000 las OBs están muy cerca entre sí (unos 1,53 millones en todos los casos). Esa diferencia tan pequeña, comparada con el total de operaciones, es lo que justifica su estabilidad y similitud.

Gráfica con el tiempo medio de reloj para MergeSort, comentarios a la gráfica.



En la gráfica se muestra el tiempo medio de ejecución de nuestro MergeSort para distintos tamaños de N. La curva crece de forma muy regular y eso encaja con la complejidad teórica del algoritmo, que es O(nlogn).

Lo más llamativo es la estabilidad: apenas hay picos y, si existen, apenas son perceptibles, el tiempo aumenta casi de manera suave conforme crece la entrada. Esto contrasta con QuickSort, donde sí aparecían pequeñas subidas cuando las particiones no quedan tan equilibradas.

Más adelante se ajustarán estos datos a una función para comprobar numéricamente que el comportamiento experimental coincide con el teórico.

5.3 Apartado 3

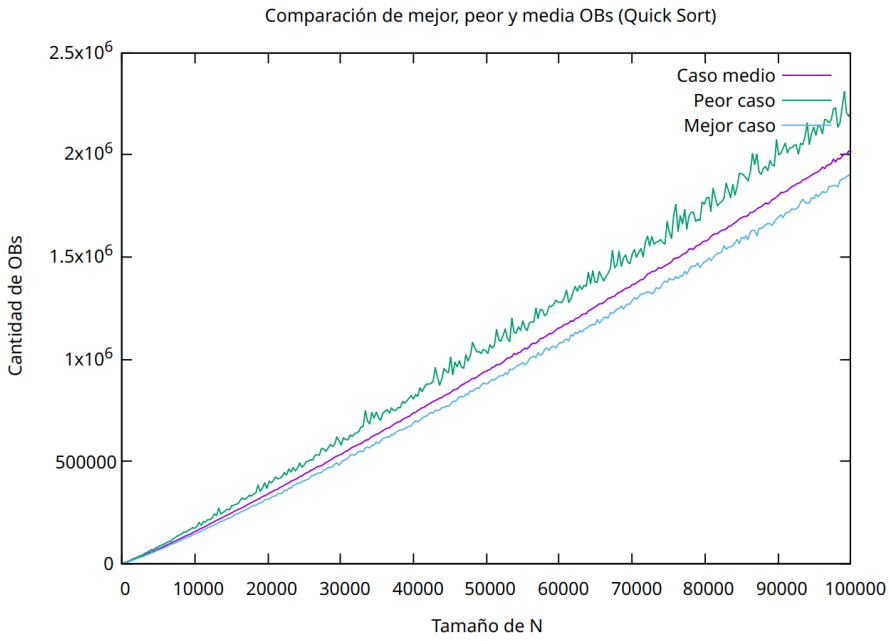
```
Running exercise4
Practice number 1, section 4
Done by: Lucas Blanco y Alina Datsko
Group: 1201
Ordenando empleando QuickSort
1       2       3       4       5       6       7       8       9       10      11      1
2       13      14      15      16      17      18      19      20      21      22      2
3       24      25      26      27      28      29      30      31      32      33      3
4       35      36      37      38      39      40      41      42      43      44      4
5       46      47      48      49      50      51      52      53      54      55      5
6       57      58      59      60      61      62      63      64      65      66      6
7       68      69      70      71      72      73      74      75      76      77      7
8       79      80      81      82      83      84      85      86      87      88      8
9       90      91      92      93      94      95      96      97      98      99      1
00
```

En esta imagen se muestra la ejecución del ejercicio 4: se ordena una tabla de 100 elementos utilizando el algoritmo QuickSort para comprobar que la implementación funciona correctamente. Con un tamaño tan pequeño no se aprecia una diferencia grande de tiempo respecto a MergeSort, e incluso puede parecer similar a métodos menos eficientes como InsertSort o BubbleSort, a pesar de que estos últimos son $O(n^2)$. Esto es normal porque la práctica, en este ejercicio, busca confirmar el funcionamiento y no su rendimiento.

En los apartados siguientes, con tablas más grandes y usando las gráficas, sí se podrá ver mejor el crecimiento asintótico de cada algoritmo.

5.4 Apartado 4

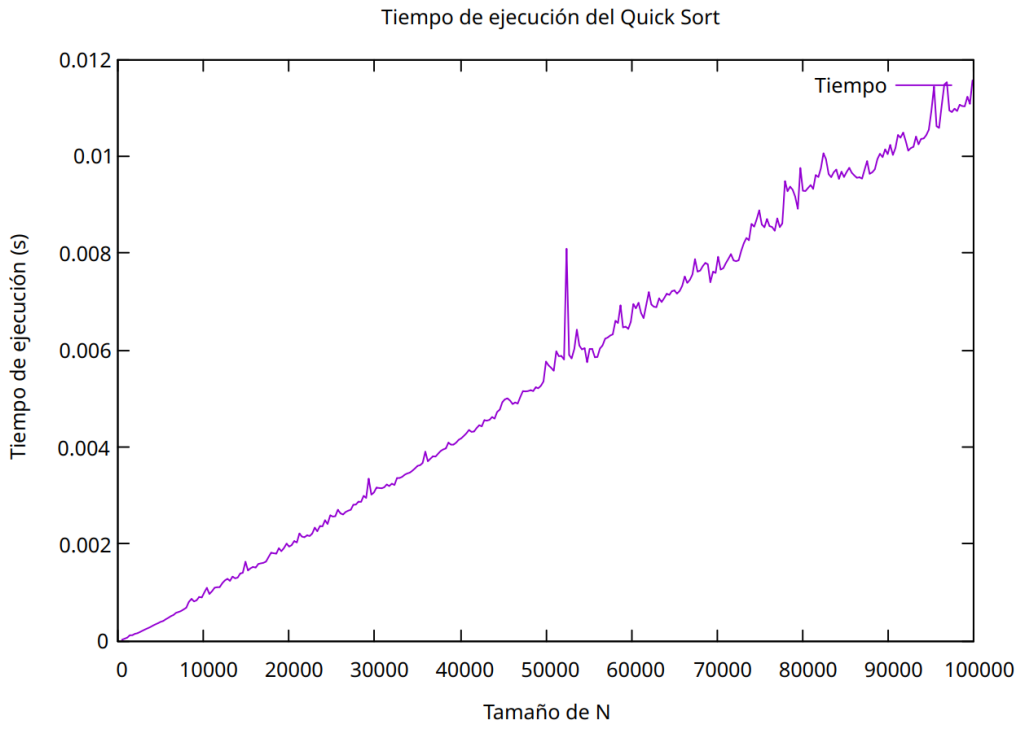
Gráfica comparando los tiempos mejor peor y medio en OBs para QuickSort, comentarios a la gráfica.



La gráfica permite ver bastante bien el comportamiento real de QuickSort según el tipo de entrada. La curva del caso medio crece de forma bastante regular y casi lineal en este rango, que es lo que esperamos para un algoritmo O(nlogn) cuando no hay particiones especialmente malas. La del mejor caso va siempre por debajo y mantiene una forma similar, lo que indica que cuando el pivote parte el array de manera equilibrada el número de comparaciones se reduce, pero el modelo de crecimiento sigue siendo el mismo.

La curva del peor caso es más alta y aparece con picos. Eso refleja que, cuando el pivote no divide bien y deja casi todo a un lado, QuickSort necesita más comparaciones en esa llamada concreta. No llega al peor caso teórico $O(n^2)$ porque no estamos forzándolo en todas las particiones, pero sí se ve que es la situación menos favorable. Esto nos deja ver que es un algoritmo bastante inestable, aunque eficiente.

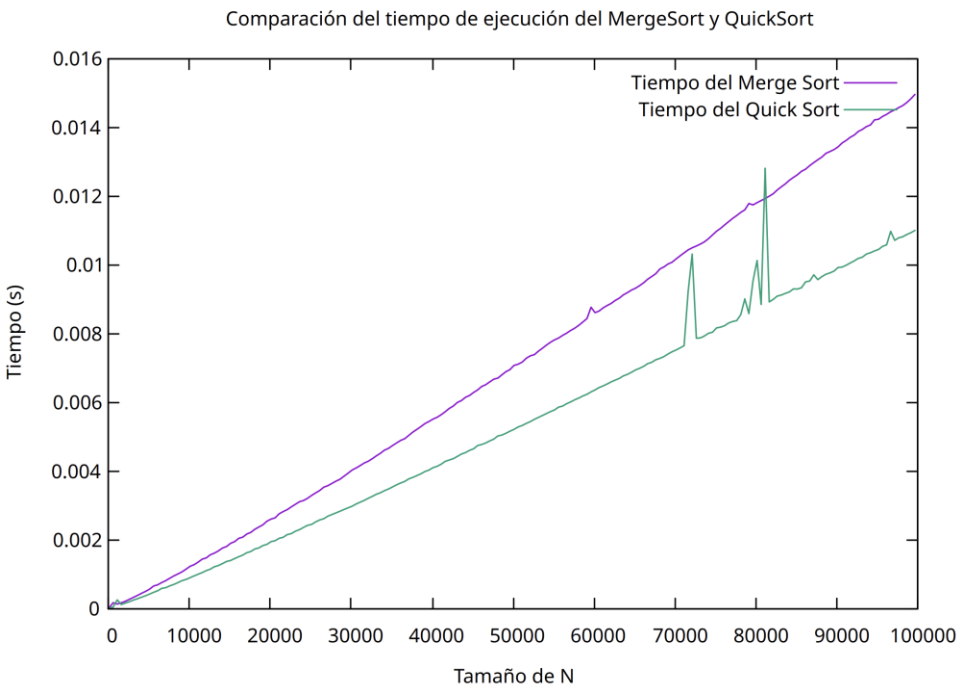
Gráfica con el tiempo medio de reloj para QuickSort, comentarios a la gráfica.



En esta gráfica vemos el tiempo de ejecución del QuickSort para distintos tamaños de entrada. La forma de la curva es la que esperamos: crece de manera suave y casi lineal con nlogn, aunque aparecen pequeños picos cuando alguna partición sale menos equilibrada. Aun así, la tendencia general es muy clara.

Más adelante esta misma serie de datos se puede ajustar con una función del tipo nlogn para comprobar que el comportamiento experimental coincide con la complejidad teórica del algoritmo. Los picos no invalidan eso: solo reflejan que QuickSort, al trabajar sobre datos concretos, a veces se acerca un poco a casos peores de partición.

Grafica comparando el tiempo medio de reloj de MergeSort y QuickSort



En la gráfica se ve claramente que, para los mismos tamaños de N, QuickSort (línea verde) acaba antes que MergeSort (línea morada). Los dos crecen más o menos lineal con crecimiento asintótico O(nlogn), pero la curva de MergeSort va siempre por encima.

Esto encaja con lo que hemos visto en el código: MergeSort, cada vez que fusiona, tiene que reservar y copiar en un array auxiliar, que en tiempo real se nota. QuickSort, en cambio, trabaja sobre el propio array y solo va partiendo en subarrays, así que tiene menos sobrecarga. Por eso, aunque MergeSort sea muy estable en número de operaciones, en la práctica resulta más rápido QuickSort.

Grafica comparando el número medio de OBs de MergeSort y QuickSort

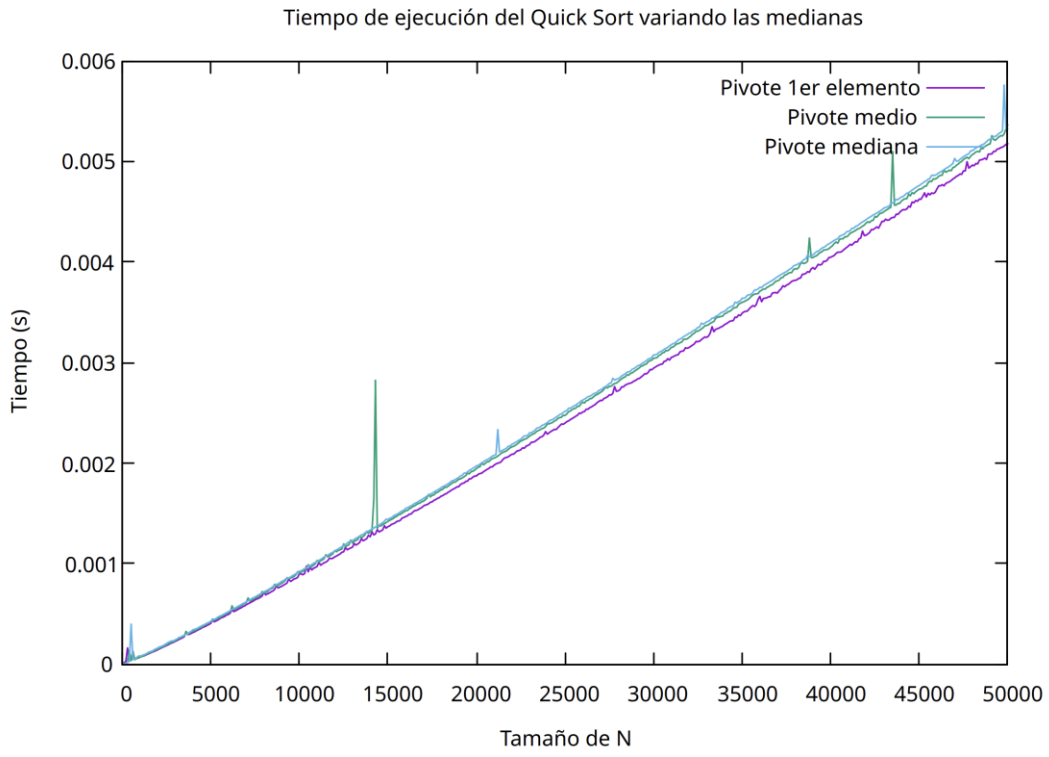
Como hemos visto en clase, comparar el número de OBs de dos algoritmos de ordenación distintos no tiene sentido, sino que para que se compare usando la misma medida se deben comparar los tiempos de reloj.

Grafica comparando el número máximo de OBs de MergeSort y QuickSort

Como hemos visto en clase, comparar el número de OBs de dos algoritmos de ordenación distintos no tiene sentido, sino que para que se compare usando la misma medida se deben comparar los tiempos de reloj.

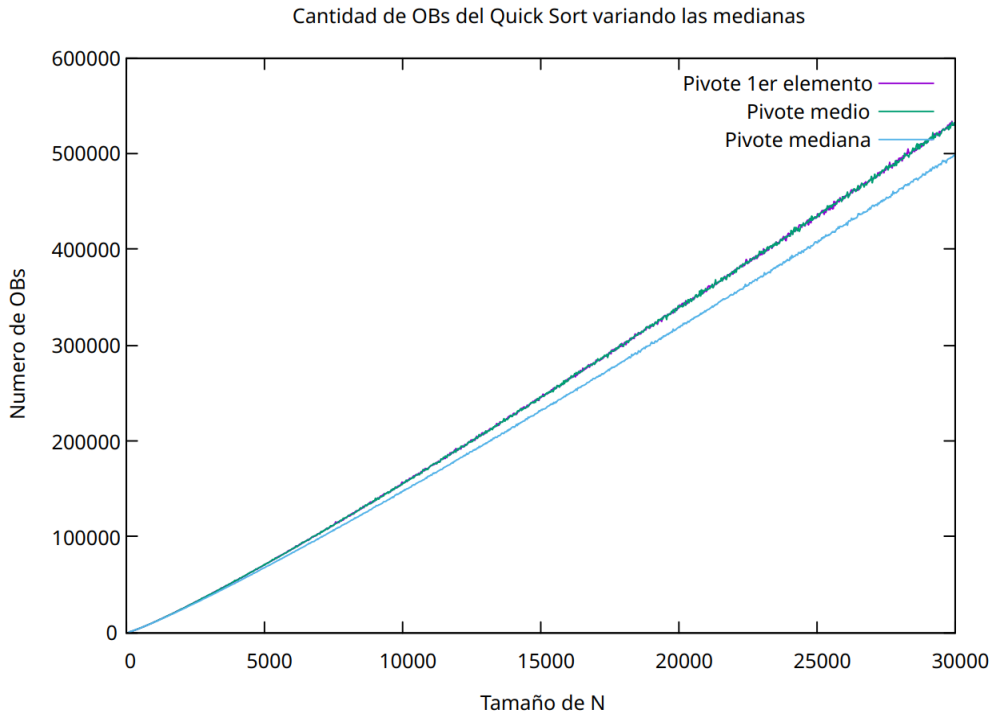
5.5 Apartado 5

Gráfica comparando el tiempo medio de reloj de Quicksort usando los pivotes **median**, **median_avg** y **median_stat**.



En esta gráfica se ven los tiempos de ejecución, que son bastante semejantes entre ellos. A pesar de que hemos llegado a un número de N elementos muy grande, las diferencias son muy pocas. Se ve que el mejor de ellos es la toma como pivote el primer elemento. Esto se debe principalmente a que, para elegir el pivote, elegir el primer no toma tiempo. Sin embargo, tomar el central tiene que realizar una división, que, a mayor número de elementos, esta decisión aumenta ligeramente el tiempo de ejecución. Por este mismo razonamiento la que queda por encima de todas ligeramente es la que toma la mediana de tres elementos. Esto sucede ya que, además de dividir para encontrar el elemento del medio, realiza comparaciones para encontrar la mediana.

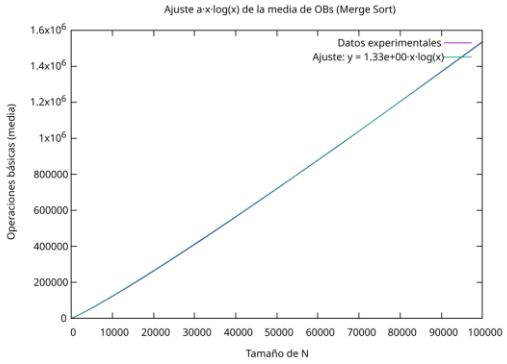
Gráfica comparando el tiempo medio de OBs de Quicksort usando los pivotes **median**, **median_avg** y **median_stat**.

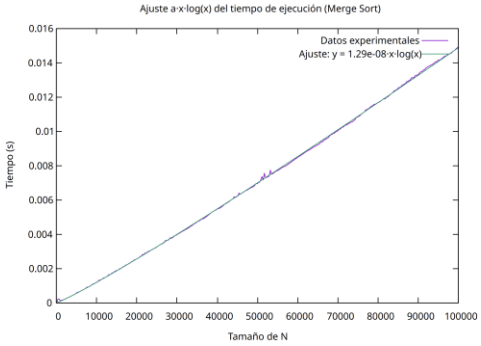


En la figura se ve que las tres versiones de QuickSort crecen de forma muy parecida (todas son O(nlog n)), pero la curva del pivote de la mediana de 3 (median_stat) va ligeramente por debajo de las otras. Esto pasa porque elegir la mediana entre primero, centro y último hace que las particiones suelan quedar más equilibradas y, por tanto, se hagan algo menos de comparaciones de media.

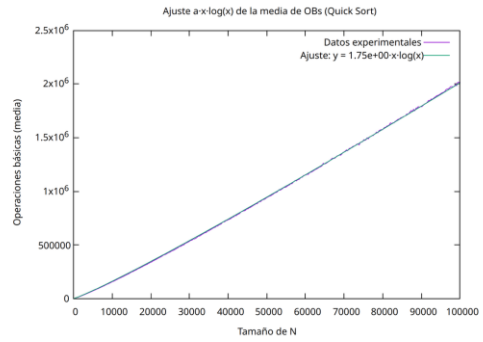
En cambio, usar siempre el primer elemento o el elemento central como pivote da resultados casi iguales: en ambos casos la elección es ciega y no garantiza que el array se parta en dos trozos parecidos. Por eso esas dos curvas van prácticamente juntas y un poco por debajo la de la mediana de 3.

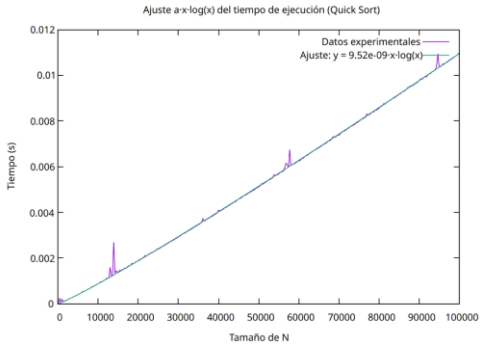
Ajuste que comprueba el crecimiento asintótico del MergeSort O(nlogn)





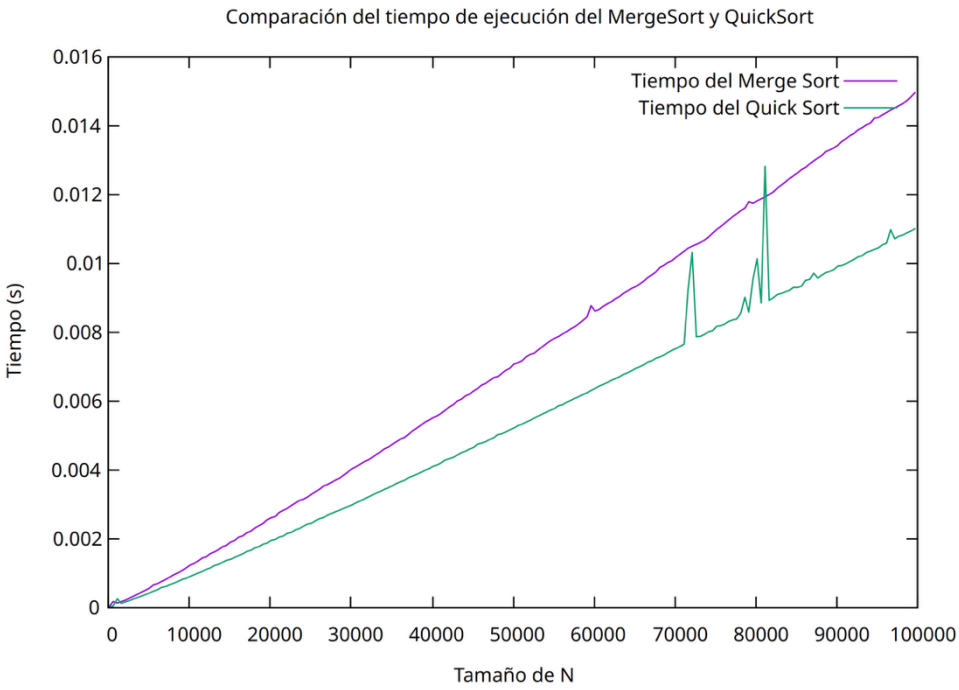
Ajuste que comprueba el crecimiento asintótico del QuickSort O(nlogn)





5. Respuesta a las preguntas teóricas.

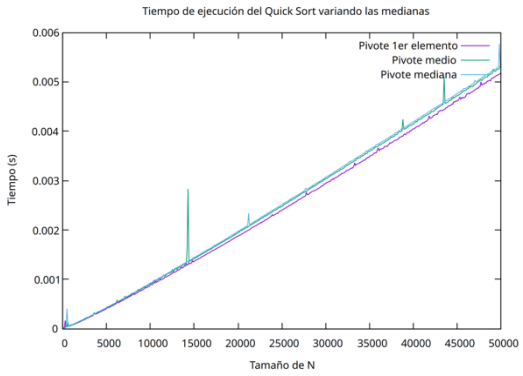
5.1 Pregunta 1

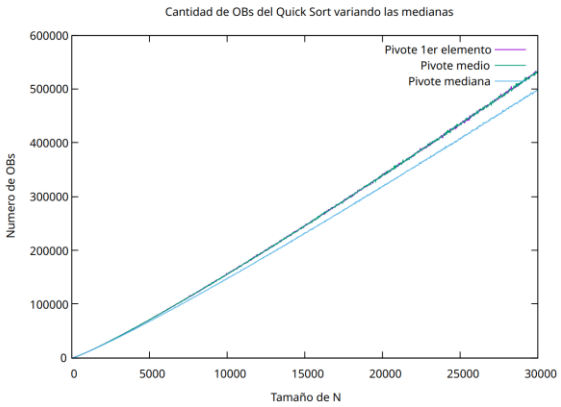


En la gráfica se ve que los dos algoritmos siguen la tendencia que dice la teoría: a medida que crece N, el tiempo crece de forma casi lineal con nlogn. No hay saltos bruscos de órdenes ni comportamientos raros, así que el rendimiento experimental cuadra con el caso medio teórico tanto para MergeSort como para QuickSort. También se aprecia que, para estos datos, QuickSort suele ir por debajo de MergeSort, algo razonable porque tiene menos coste de memoria.

Lo que sí llama la atención son los picos que aparecen de vez en cuando. En QuickSort pueden aparecer cuando alguna partición sale un poco descompensada y esa llamada concreta hace más trabajo que las demás. Como estamos midiendo tiempos muy pequeños, cualquier pequeña variación en la forma de dividir el array se nota, pero la tendencia general sigue siendo la de O(nlogn).

5.2 Pregunta 2





En las gráficas de tiempo se observa que cambiar la forma de elegir el pivote no modifica demasiado la duración total del algoritmo: las tres versiones de QuickSort tardan más o menos lo mismo para un mismo tamaño de entrada. Lo que sí se nota es que, cuando usamos la mediana de tres como pivote, desaparecen casi por completo los “picos” de ejecución, porque esta elección consigue particiones más equilibradas y, por tanto, evita caer en llamadas muy desfavorables. Sin embargo, en el tiempo de ejecución concluimos que la elección del primer elemento es la mejor en general, ya que no tiene que realizar una división ni tiene que comparar.

Donde la diferencia sí es más visible es en las gráficas de operaciones básicas. Ahí se ve que la versión con pivote por mediana hace menos comparaciones que la que siempre coge el primer elemento o la central. Esto tiene sentido: al elegir un pivote más central (en términos de valor numérico), las tablas que se introducen en la recursión quedan más equilibradas y el algoritmo necesita menos trabajo a nivel de OBs.

5.3 Pregunta 3

Para el MergeSort:

El **mejor caso** del MergeSort es aquel en el que la tabla está en orden descendente. Hemos llegado a esta conclusión porque, sin pérdida de generalidad, en el último paso de la rutina, cuando se combinan las dos mitades, los n elementos de la primera tabla son mayores que los n elementos de la tabla de la derecha. Como los elementos de la tabla derecha son más pequeños, se colocarán primero y, a continuación, se copiarán todos los mayores de una vez. Esta misma lógica podría aplicarse a una tabla ordenada de menor a mayor; sin embargo, si el número de elementos es impar, la parte izquierda queda con un elemento más que la derecha y, estrictamente, ya no sería el caso más eficiente.

Lo hemos implementado así:

```
if (metodo == mergesort){
  for (i = 0; i < n_perms; i++) {
    for (j = 0; j < N; j++) {
      tabla[i][j] = N - j;
    }
  }
}
```

En el **peor caso** del MergeSort hemos planteado un problema bastante más complejo que el anterior. La lógica detrás de esto es que queremos que cada vez que merge tenga que fusionar dos mitades tenga que comparar casi todos los elementos en cada iteración. Para ello creamos la función recursiva de peor_merge.

En este bloque lo que hacemos es reordenar el array de una forma concreta. Primero recorremos el array original cogiendo solo las posiciones pares (0, 2, …) y las vamos guardando seguidas en el array auxiliar. Después hacemos otra pasada cogiendo las posiciones impares (1, 3, …) y las ponemos a continuación en el auxiliar. Al final copiamos todo el contenido del auxiliar de vuelta al array original.

El resultado es que el array queda intercalado: primero van todos los elementos que antes estaban en posiciones pares y después todos los que estaban en posiciones impares.

La implementación es la siguiente:

```
  if (metodo == mergesort){
    for (i = 0; i < n_perms; i++) {
      for (j = 0; j < N; j++){
        tabla[i][j] = j + 1;
      }
      peor_merge(tabla[i], N, aux);
    }
  }
```

```
void peor_merge(int *a, int n, int *aux) {

  int i=0, mid=0, k=0;

  assert(a != NULL);
  assert(aux != NULL);
  assert(n >= 0);

    if (n <= 1){
        return;
    }

    for (i = 0; i < n; i += 2){
      aux[k++] = a[i];
    }

    for (i = 1; i < n; i += 2){
      aux[k++] = a[i];
    }

    for (i = 0; i < n; i++){
      a[i] = aux[i];
    }

    mid = (n + 1) / 2;

    peor_merge(a, mid, aux);
    peor_merge(a + mid, n - mid, aux);
}
```

En MergeSort casi no hay **caso medio** que se note: da igual cómo venga la lista (ordenada, inversa o aleatoria): el algoritmo siempre parte en mitades y luego fusiona las dos mitades. Por eso, una lista aleatoria describe bien el comportamiento medio, que en la práctica es muy parecido al mejor y al peor caso. Según la teoría: $A_{MS}(N)=\Theta(N\lg(N))$

Para el QuickSort:

El **mejor caso** también hemos planteado una función recursiva que nos plasma el mejor caso. Estas funciones que se plantena después de este párrafo generan de forma recursiva una permutación que provoca el mejor caso. La recursiva va tomando siempre el elemento central de cada subarray y lo coloca en una posición auxiliar, y luego hace lo mismo con la parte izquierda y la derecha. De este modo, cuando el QuickSort real use ese array y escoja el pivote, las particiones le saldrán muy equilibradas en todos los niveles. El pequeño desplazamiento que se hace en la parte izquierda sirve para ajustar el orden y mantener ese equilibrio. Por último, la función externa copia la permutación construida al array original.

La implementación es la siguiente:

```
  if (metodo == quicksort)
  {
    aux = (int *)malloc(N * sizeof(int));
    if (aux == NULL)
    {
      for (i = 0; i < n_perms; i++)
      {
        free(tabla[i]);
      }
      free(tabla);
      return ERR;
    }

    for (i = 0; i < n_perms; i++)
    {
      for (j = 0; j < N; j++)
      {
        tabla[i][j] = j + 1;
      }

      mejor_quick(tabla[i], N, aux);
    }

    free(aux);
  }
```

```c
void mejor_quick(int *a, int n, int *aux)
{
  int i = 0;

  assert(a != NULL);
  assert(aux != NULL);
  assert(n >= 0);

  quick_rec_best(a, 0, n - 1, aux, 0);
  for (i = 0; i < n; i++)
  {
    a[i] = aux[i];
  }
}
```

```c
int quick_rec_best(int *a, int l, int r, int *aux, int pos)
{
  int i, mid, left_start, left_len, first;

  assert(a != NULL);
  assert(aux != NULL);
  assert(pos >= 0);

  if (l > r)
    return pos;

  mid = (l + r) / 2;

  aux[pos++] = a[mid];

  left_start = pos;
  pos = quick_rec_best(a, l, mid - 1, aux, pos);
  left_len = pos - left_start;

  if (left_len > 1)
  {
    first = aux[left_start];
    for (i = left_start; i < left_start + left_len - 1; i++)
      aux[i] = aux[i + 1];
    aux[left_start + left_len - 1] = first;
  }

  pos = quick_rec_best(a, mid + 1, r, aux, pos);

  return pos;
}
```

El **peor caso** del QuickSort hemos implementado, el pivote se elige siempre como el primer elemento del array. Si a ese algoritmo le damos como entrada una tabla ordenada de forma descendente (del mayor al menor), ese primer elemento será siempre el valor más grande. Al hacer la partición, todos los demás elementos resultan ser menores que el pivote, de modo que se colocan en el mismo subarray y la partición queda totalmente desequilibrada: una parte con N−1 elementos y la otra vacía. Esto nos da el peor caso del Quick, que hace un total de N(N-1)/2 OBs. Esta forma coincide con el peor caso del BubbleSort e InsertSort de la anterior práctica.

Lo implementación es la siguiente:

```
   if(metodo == BubbleSort || metodo == InsertSort || metodo == quicksort){
    for (i = 0; i < n_perms; i++) {
      for (j = 0; j < N; j++) {
        tabla[i][j] = N - j;
      }
    }
  }
```

El caso **medio**: en QuickSort cuando eliges siempre el primer elemento como pivote, el caso medio es la situación normal en la que ese pivote deja algunos valores por debajo y otros por encima, pero sin partir el array en dos mitades exactas ni dejar casi todo a un lado. Esa partición es razonable (aunque no perfecta) se repite recursivamente y, sumando el coste lineal de cada partición, lleva a una complejidad O(nlogn) sin llegar al caso peor que es $O(n^2)$. En la teoría se puede ver: 2NlogN+ O(N)

5.4 Pregunta 4

Si nos quedamos con lo que hemos medido, el que sale mejor parado es el Quicksort. En teoría los dos tienen el mismo crecimiento asintótico (ambos son O(nlogn)), pero en la práctica nuestro MergeSort pierde tiempo porque tiene que estar reservando memoria y copiando datos en cada merge. Quicksort, en cambio, trabaja siempre sobre el mismo array y eso se nota en el tiempo de reloj.

También se ve que MergeSort es más suave, ya que sus OBs crecen muy regular y no tiene picos, mientras que Quicksort sí puede tenerlos si la partición no sale del todo bien. Sin embargo, hay formas que hacen que se reduzan el número de OBs. Además, como las OBs no son comparables entre algoritmos distintos, lo que manda aquí es el tiempo.

Y si hablamos de memoria, ahí no hay comparación entre ambos: Quicksort es más eficiente ya que no necesita un array auxiliar grande; MergeSort sí. Así que, con esta implementación y estos datos, se puede decir que empíricamente y en memoria es mejor Quicksort, aunque MergeSort tenga el punto a favor de la estabilidad.

6. Conclusiones finales.

Con esta práctica hemos aprendido el funcionamiento de nuevos algoritmos de ordenación que, a diferencia de los de la anterior práctica (InsertSort y BubbleSort), tienen una complejidad asintótica que deja a los anteriores muy por detrás. De hecho, a modo de curiosidad hemos hecho varios comandos en el Makefile que muestra en modo de gráfica como trabajan con el mismo número de condiciones y parece que los algoritmos O(nlogn) responden casi a la constante y = 0.

Hemos aprendido a cómo implementar dichas funciones y a emplear la recursión en casos útiles y prácticos. Además, hemos podido comprobar y comparar cuál de los dos algoritmos es más eficientes en términos de tiempo de ejecución y discutir cuál de ellos era óptimo en términos de memoria del dispositivo. De ello sacamos la conclusión de que el algoritmo de QuickSort, a pesar de ser bastante más inestable que el MergeSort, es óptimo para la gran mayoría de casos, sobre todo en aquellos en los que el tamaño de N es muy grande.

Por otro lado, tratamos de mantener un estilo de código correcto, usando asserts como forma de controlar los errores de las funciones y escribiendo los comentarios en el código necesarios para la comprensión del código pero sin caer en simplismos.

Mantuvimos el desarrollo de todo el trabajo siguiendo el estándar C90, ya que de esta forma el código puede compilarse y ejecutarse correctamente en cualquier versión del lenguaje C, cosa que también se nos ha enseñado en clase.