Análisis de Algoritmos 2022/2023

Práctica 2

Diego Rodríguez Ortiz y Alejandro García Hernando, Grupo 1202.

Código	Gráficas	Memoria	Total

1. Introducción.

En esta práctica hemos comprobado de manera experimental lo visto en clase. Para ello, hemos diseñado un programa que implementa dos de los algoritmos de ordenación aprendidos, MergeSort y QuickSort. Además de ordenar listas, cuenta el número de operaciones básicas realizadas por el mismo y el tiempo que le ha llevado hacerlo (en función del tamaño de la entrada). Para ello recurrimos a los conocimientos obtenidos en la parte de teoría e hicimos una serie de aproximaciones teóricas para estimar los resultados que nuestro código iba a proporcionar.

2. Objetivos

2.1 Apartado 1

En este primer apartado desarrollamos la función *mergesort* que hace uso del algoritmo MergeSort para ordenar una lista de números. Para lograr lo anterior hay que crear una segunda función llamada *merge,* que ordena la fusión de dos listas ordenadas contiguas en la misma tabla.

2.2 Apartado 2

Modificando ligeramente el fichero *exercise5.c* de la práctica anterior, debemos obtener el tiempo promedio de ejecución del algoritmo de ordenación implementado en la función anterior. También hay que conseguir el número máximo, mínimo y promedio de operaciones básicas ejecutadas por el algoritmo en función del tamaño de la entrada.

2.3 Apartado 3

El objetivo es componer una segunda función, en este caso *quicksort*, para ordenar listas de números. Para ello, también hay que hacer *partition* (divide la tabla situando a la izquierda los números menores al pivote proporcionado) y *median* (devuelve el índice del primer elemento).

2.4 Apartado 4

Obtener las características del algoritmo de ordenación QuickSort (tiempo promedio de ejecución, número máximo, mínimo y promedio de operaciones básicas ejecutadas por el algoritmo en función del tamaño de la entrada). Para ello modificaremos ligeramente el fichero *exercise5.c* de la práctica anterior.

2.5 Apartado 5

Añadir las funciones *median_avg* (devuelve el índice de la posición media de la tabla) y *median_stat* (devuelve el índice de la posición del valor medio entre ip, iu y (ip+iu)/2). Con esto, hay que comparar los tiempos promedio de ejecución de *quicksort* con cada una de las tres rutinas pivote implementadas.

3. Herramientas y metodología

Cómo ambos tenemos ordenadores Windows, hemos decidido programar en Visual Code y ejecutar nuestros programas en Ubuntu. Además, para realizar las gráficas hemos empleado gnuplot y hemos comprobado que no haya fugas de memoria con Valgrind.

3.1 Apartado 1

Para desarrollar la función MergeSort nos hemos basado en el pseudocódigo facilitado en la presentación de la asignatura, donde hemos encontrado tanto el de la función *mergesort*, como el de *merge*, la auxiliar para combinar.

3.2 Apartado 2

Hemos comprobado el correcto funcionamiento del código escrito anteriormente cambiando el fichero de la práctica anterior *exercise5.c.* Dicho archivo fue adaptado para obtener el tiempo promedio de ejecución, máximo, mínimo y promedio de operaciones básicas realizadas por el algoritmo en cada caso. Posteriormente, graficamos con gnuplot las características obtenidas anteriormente.

3.3 Apartado 3

Para desarrollar la función QuickSort nos hemos basado en el pseudocódigo facilitado en la presentación de la asignatura. Implementando de este modo la función general *quicksort*. Además, ideamos cómo crear las rutinas *partitition* y *median* de apoyo a la función principal.

3.4 Apartado 4

Hemos comprobado el correcto funcionamiento del código escrito anteriormente cambiando el fichero de la práctica anterior *exercise5.c.* Obtuvimos el tiempo promedio de ejecución, máximo, mínimo y promedio de operaciones básicas realizadas por QuickSort en cada caso. Y graficamos sus características, al igual que con MergeSort, utilizando gnuplot.

3.5 Apartado 5

Construimos las funciones auxiliares *median_stat* y *median_avg* para medir cómo variaba el número de OBs realizadas por QuickSort al ordenar las distintas permutaciones. Seguimos el método descrito en el apartado anterior para obtener las características de las distintas rutinas, y las comparamos representándolas gráficamente con gnuplot.

4. Código fuente

4.1 Apartado 1

```c
int MergeSort(int* tabla, int ip, int iu){
  int imedio, m1, m2, m3;

  if(!tabla||ip>iu) return ERR;

  if(ip==iu) return OK;

  imedio=(iu-ip)/2+ip;
  m1=MergeSort(tabla, ip, imedio);
  if(m1==ERR) return ERR;
  m2=MergeSort(tabla,imedio+1, iu);
  if(m2==ERR) return ERR;


  m3=merge(tabla, ip, iu, imedio+1);
  if(m3==ERR) return ERR;

  return m1+m2+m3;
}
```

```c
int merge(int* tabla, int ip, int iu, int imedio){
  int j1=ip, j2=imedio, *t=NULL, m=0, coste=0;

  if(!tabla) return ERR; /*Control de errores*/

  t=(int*)calloc(iu-ip+1, sizeof(int));
  if(!t) return ERR;

  while((j1<imedio)&&(j2<=iu)){
    if(tabla[j1]<tabla[j2]){
      t[m]=tabla[j1];
      j1++;
      m++;
    }
    else{
      t[m]=tabla[j2];
      j2++;
      m++;
    }
  }
  coste=m;

  while(j1<imedio){
    t[m]=tabla[j1];
    j1++;
    m++;
  }
  while(j2<=iu){
    t[m]=tabla[j2];
    j2++;
    m++;
  }

  for(m=0; m<=iu-ip; m++) tabla[ip+m]=t[m]; /*Copiam
  free(t);

  return coste; /*Devolvemos el coste*/
}
```

4.3 Apartado 3

```c
int QuickSort(int *tabla, int ip, int iu)
{
    int pos, cont;
    if (!tabla || ip > iu)
    {
        return ERR;
    }

    if (ip == iu)
    {
        return 0;
    }
    else
    {
      cont = partition(tabla, ip, iu, &pos);
      if(cont==ERR){
        return ERR;
      }
      if(ip<pos){
         cont += QuickSort(tabla, ip, pos - 1);
      }
      if(iu>pos){
        cont += QuickSort(tabla, pos + 1, iu);
      }
    }

    return cont;
}
```

```c
int partition(int *tabla, int ip, int iu, int *pos)
{
    int cont, k, m, i;
    if (!tabla || ip > iu || !pos)
    {
        return ERR;
    }

    cont = median(tabla, ip, iu, pos);
    if(cont==ERR){
        return ERR;
      }
    m = *pos;
    k = tabla[m];

    swap1(&tabla[ip], &tabla[m]);
    m = ip;
    for (i = ip + 1; i <= iu; i++)
    {
        cont++;
        if (tabla[i] < k)
        {
            m++;
            swap1(&tabla[i], &tabla[m]);
        }
    }

    swap1(&tabla[ip], &tabla[m]);
    *pos = m;
    return cont;
}
```

```c
int median(int *tabla, int ip, int iu, int *pos)
{
    if (!tabla || ip > iu || !pos)
    {
        return ERR;
    }
    *pos = ip;
    return 0;
}
```

4.5 Apartado 5

```c
int median_stat(int *tabla, int ip, int iu, int *pos)
{
  int media;
    if (!tabla || ip > iu || !pos)
    {
        return ERR;
    }

    media=(iu-ip)/2+ip;

    if(tabla[ip]>tabla[iu]){
      if(tabla[ip]>tabla[media]){
        if(tabla[media]>tabla[iu]){
          *pos=media;
        }
        else{
          *pos=iu;
        }
      }
      else{
        *pos=ip;
        return 2;
      }
    }
    else{
      if(tabla[iu]>tabla[media]){
        if(tabla[ip]>tabla[media]){
          *pos=ip;
        }
        else{
          *pos=media;
        }
      }
      else{
        *pos=iu;
        return 2;
      }
    }

    return 3;
}
```

```c
int median_avg(int *tabla, int ip, int iu, int *pos)
{
    if (!tabla || ip > iu || !pos)
    {
        return ERR;
    }

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

Código que genera la permutación del caso peor dada la tabla ordenada:

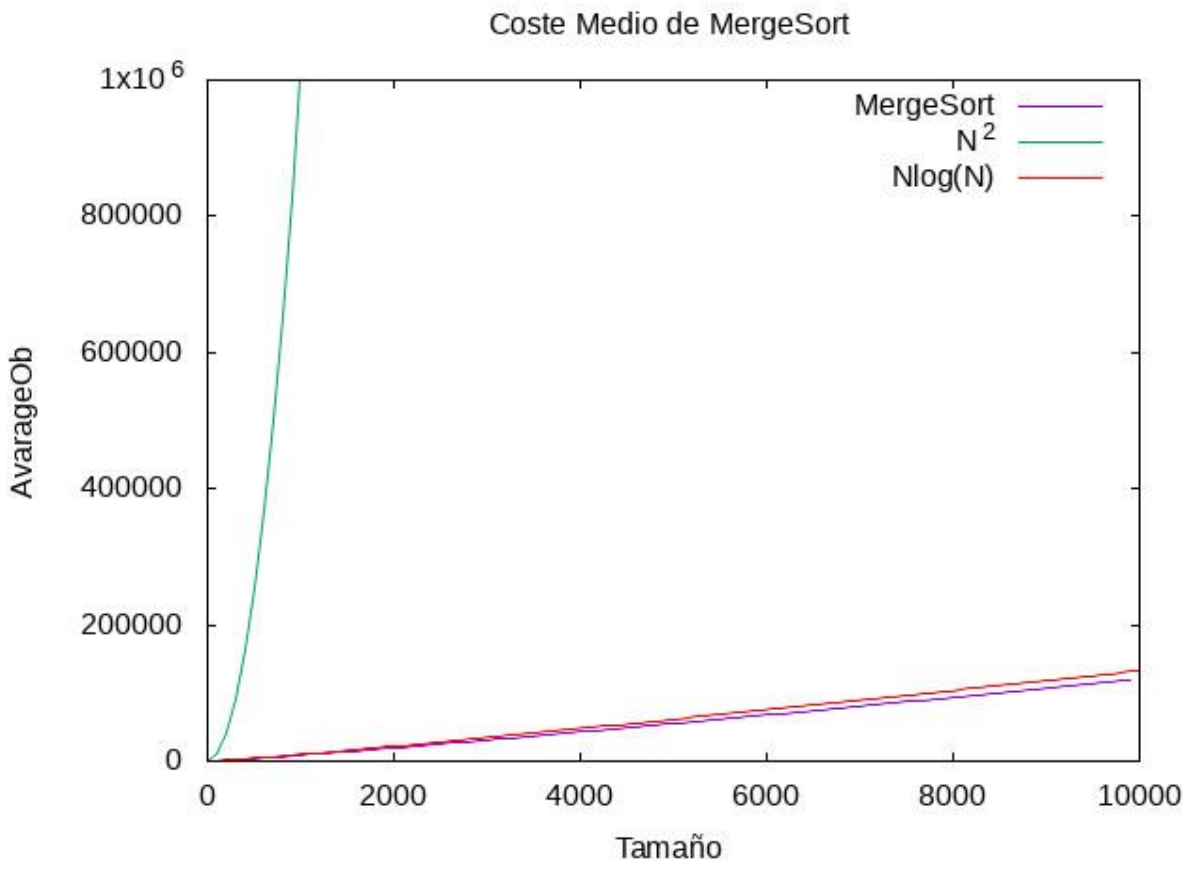
Para la pregunta 6.3

```c
void WorstCaseMergeRec(int *tabla,int *aux,int ip,int iu){
  int i,j;
    if(ip >= (iu -1 )){
        return ;
    }

    for(i=ip,j=ip;i<iu;i+=2,j++){
        aux[j]=tabla[i];
        aux[j+(iu-ip+1)/2]=tabla[i+1];
    }
     for(i=ip;i<=iu;i++){
      tabla[i]=aux[i];
    }

    WorstCaseMergeRec(tabla,aux,ip,(iu-ip)/2+ip);

    WorstCaseMergeRec(tabla,aux,(iu-ip)/2+1+ip,iu);
    return ;
}
void WorstCaseMerge(int *tabla, int ip,int iu){

  int *aux=NULL;
    if(!tabla || ip>iu){
        return ;
    }
    aux=(int*)calloc(iu-ip+1,sizeof(int));
    if(!aux){
      return;
    }

    WorstCaseMergeRec(tabla,aux,ip,iu);
    free(aux);
    return ;
}
```

5. Resultados, Gráficas

5.1/2 Apartado 1 y 2

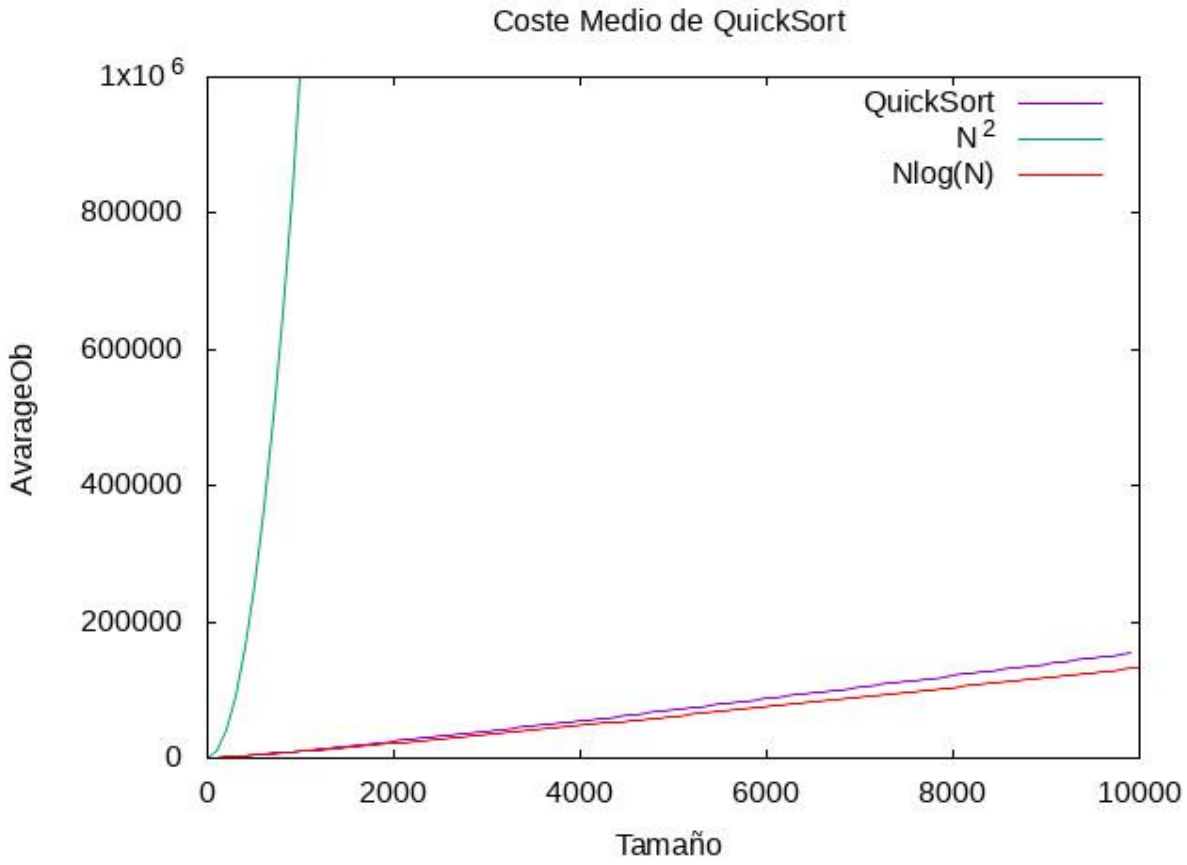
Cómo puede verse a continuación, el número medio de operaciones básicas realizadas por el algoritmo MergeSort crece de una manera muy similar a la función Nlog(N). Siendo ambas sensiblemente menores a la función N^2 (representativa del coste de los algoritmos de ordenación locales).



Como bien sabemos, el tiempo empleado por un algoritmo de ordenación es directamente proporcional al número de OBs realizadas en el proceso. Por lo tanto, es fácil entender que el crecimiento de la gráfica es, como en el caso anterior, igual al de Nlog(N). *Para facilitar la comparación de tiempos de ejecución entre los distintos algoritmos, analizaremos individualmente las OBs realizadas por cada uno de ellos y después compararemos gráficamente su eficiencia en cada caso.*

5.3/4 Apartado 3 y 4

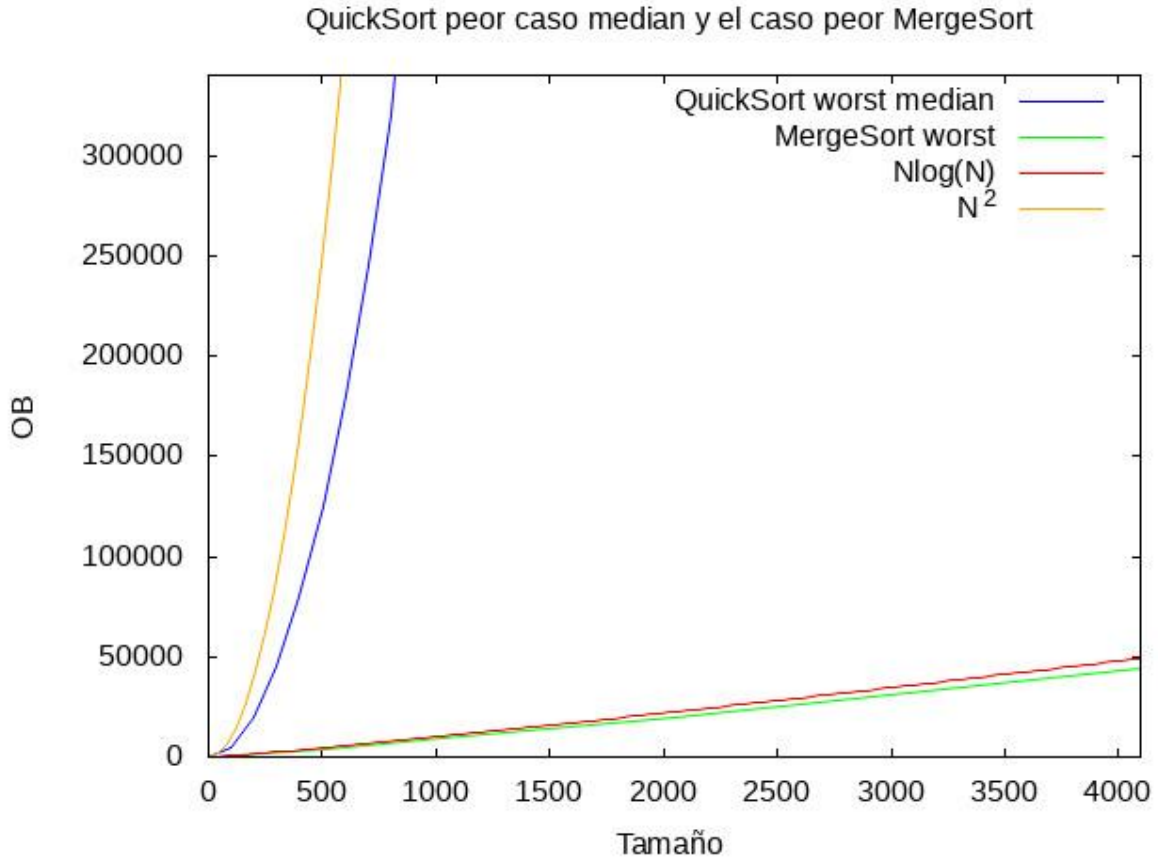
Al igual que MergeSort, las OBs realizadas por el algoritmo QuickSort presenta un crecimiento muy similar al de la función Nlog(N). Estos dos nuevos algoritmos, MergeSort y QuickSort, presentan una mejora significativa frente a todos los algoritmos de ordenación locales. Cuyo crecimiento es comparable al de la función N^2, siendo muchísimo más costoso ordenar con ellos.



Del mismo modo que MergeSort, el tiempo medio de ejecución presenta un crecimiento muy similar a Nlog(N), pues es directamente proporcional a las OBs realizadas. Coincidiendo con el comportamiento asintótico de la gráfica anterior, dónde se puede ver el número de operaciones básicas ejecutadas en función del tamaño de la lista a ordenar. *Para facilitar la comparación de tiempos de ejecución entre los distintos algoritmos, analizaremos individualmente las OBs realizadas por cada uno de ellos y después compararemos gráficamente su eficiencia en cada caso.*

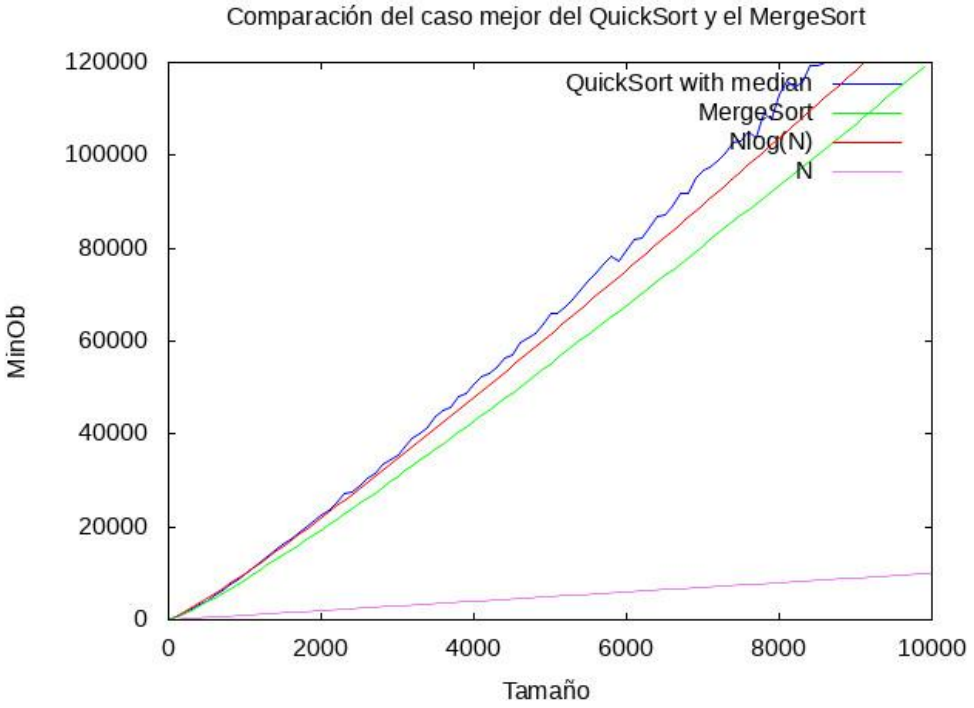
Comparación de OBs entre los dos algoritmos:

-Caso peor: Como podemos ver en la imagen adjunta, el caso peor de MergeSort es muy similar al medio. Por ello, podemos deducir que es un algoritmo muy fiable, pues incluso encontrándonos en un caso muy malo su coste no será demasiado alto. a lo largo del tiempo (ligeramente menor a la función auxiliar Nlog(N)). Mientras que el crecimiento del caso peor de QuickSort con *median* es mayor que el de MergeSort, pues su crecimiento es de la misma forma que el cuadrado del tamaño de la lista a ordenar.

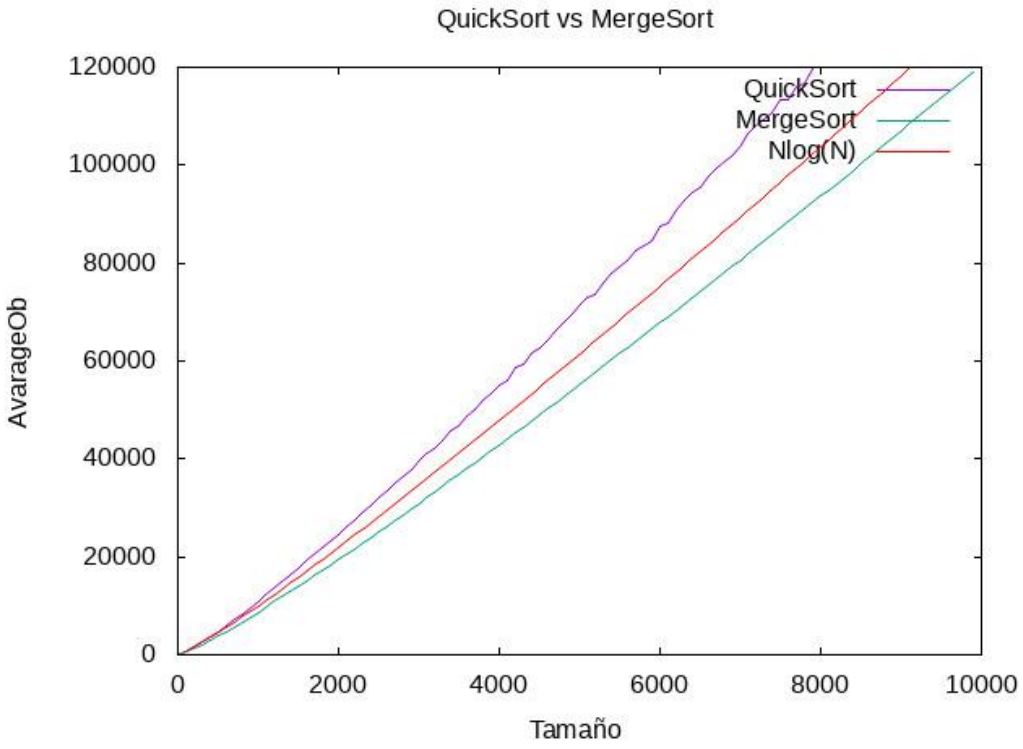


De este modo, podemos deducir que si priorizamos la constancia de un algoritmo y que nunca tarde "demasiado", MergeSort será mejor que QuickSort.

-Caso mejor: Mientras que en el caso mejor QuickSort hace algo más de Nlog(N) operaciones básicas y MergeSort algo menos, ambas funciones están extremadamente cerca de la función auxiliar mencionada y presentan un crecimiento igual a la misma. *Es muy picuda porque como ordena un número limitado de listas, hay ocasiones que el mejor caso de dicho conjunto no es el mejor caso de todas las opciones, por eso hay veces que se almacena como mayor tiempo uno ligeramente mayor que el mejor tiempo "real".*

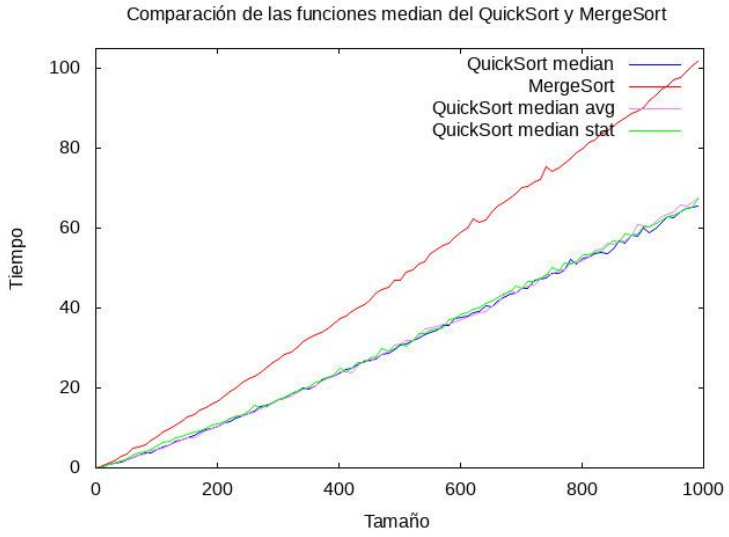


-Caso medio: Se puede apreciar que ambas funciones son muy similares. QuickSort necesita hacer ligeramente más comparaciones de clave que MergeSort, pero ambas son, a efectos prácticos, iguales.



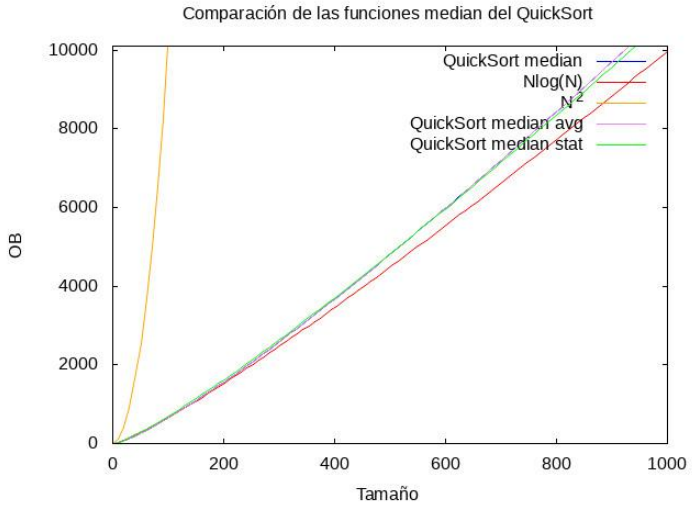
Comparación de tiempos entre los dos algoritmos:

Sorprendentemente MergeSort, el algoritmo que realiza menos OBs, es el que más tiempo tarda. Esto se debe a que realiza operaciones más costosas que no se tienen en cuenta al contar OBs; el mejor ejemplo de ello es la reserva de memoria. Al reservar memoria se pierde un valioso tiempo que permite que las tres variantes de QuickSort, a pesar de necesitar comparar claves en más ocasiones, sean más rápidas que MergeSort. Por ello, en términos de memoria empleada y tiempo medio de ejecución, QuickSort es mejor opción que MergeSort.



5.5 Apartado 5

Gráfica con las OBs realizadas comparando las versiones de Quicksort con las funciones pivote **median, median_avg** y **median_stat.**



Para hacer más sencilla la comparación entre las operaciones básicas realizadas, hemos graficado las funciones auxiliares Nlog(N) y N^2. Cabe resaltar la dificultad de distinguir las líneas que representan las OBs realizadas por cada variación de QuickSort, pues la diferencia entre ellas es mínima. Su diferencia es tan pequeña que en la gráfica son apenas distinguibles y en la práctica podemos decir que presentan el mismo coste en términos de OBs realizadas.

Continuamos la comparación de la quicksort con los distintos pivotes en la pregunta 6.2.

6. Respuesta a las preguntas teóricas.

6.1 Compara el rendimiento empírico de los algoritmos con el caso medio teórico en cada caso. Si las trazas de las gráficas del rendimiento son muy picudas razonad porqué ocurre esto.

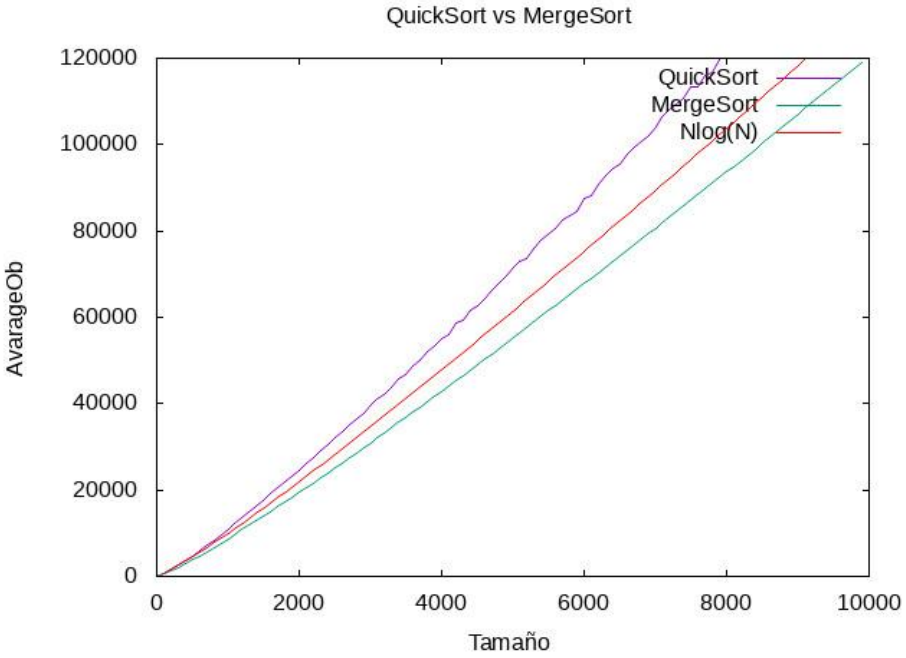
Para la predicción teórica usaremos los datos facilitados en la presentación de la asignatura:

Wms(N)<=Nlg(N) +O(N)	Bms(N)>=(½)Nlg(N)	Ams(N)=Θ(Nlg(N))
Wqs(N)=N^2 -N/2	Bqs(N)=?	Aqs(N)=2Nlg(N) +O(N)

Con los datos superiores, podemos predecir que el crecimiento asintótico del número de OBs ejecutadas es el mismo para ambos algoritmos y para Nlog(N).

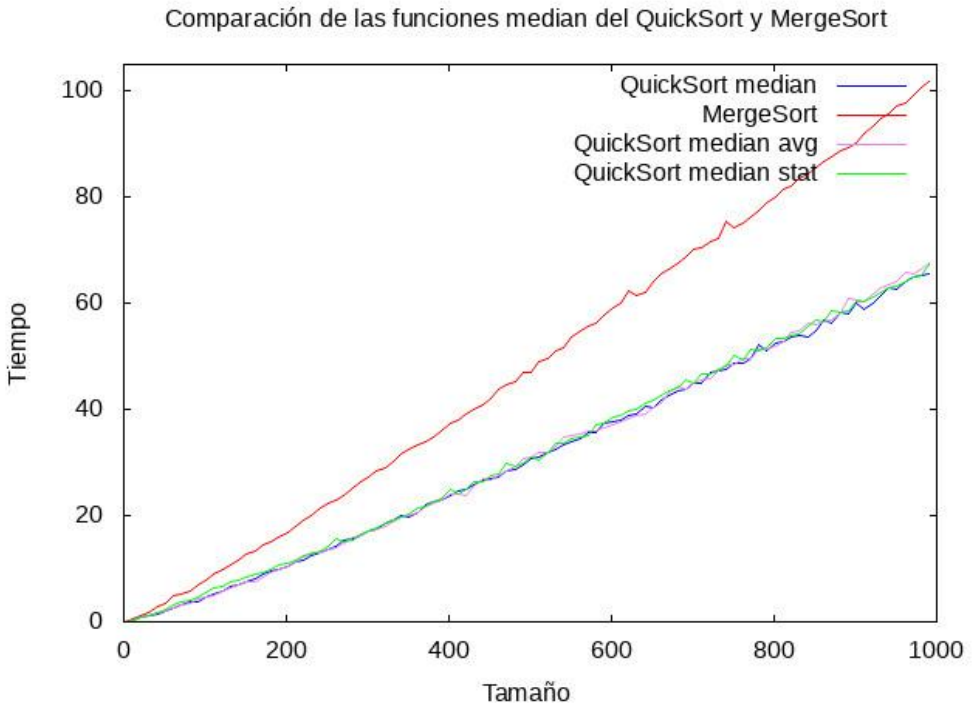
Para el aspecto empírico utilizaremos la gráfica adjunta a continuación, que relaciona el número de OBs medias, el algoritmo de ordenación empleado y el tamaño del array. Con ella, podemos deducir que MergeSort realiza ligeramente menos OBs de media que Nlog(N), mientras que QuickSort realiza algo más.

Esto, coincide con los datos obtenidos empíricamente pues MergeSort presenta el mismo crecimiento que Nlg(N), y QuickSort también tiene este mismo crecimiento además de estar algo por encima de la función citada.

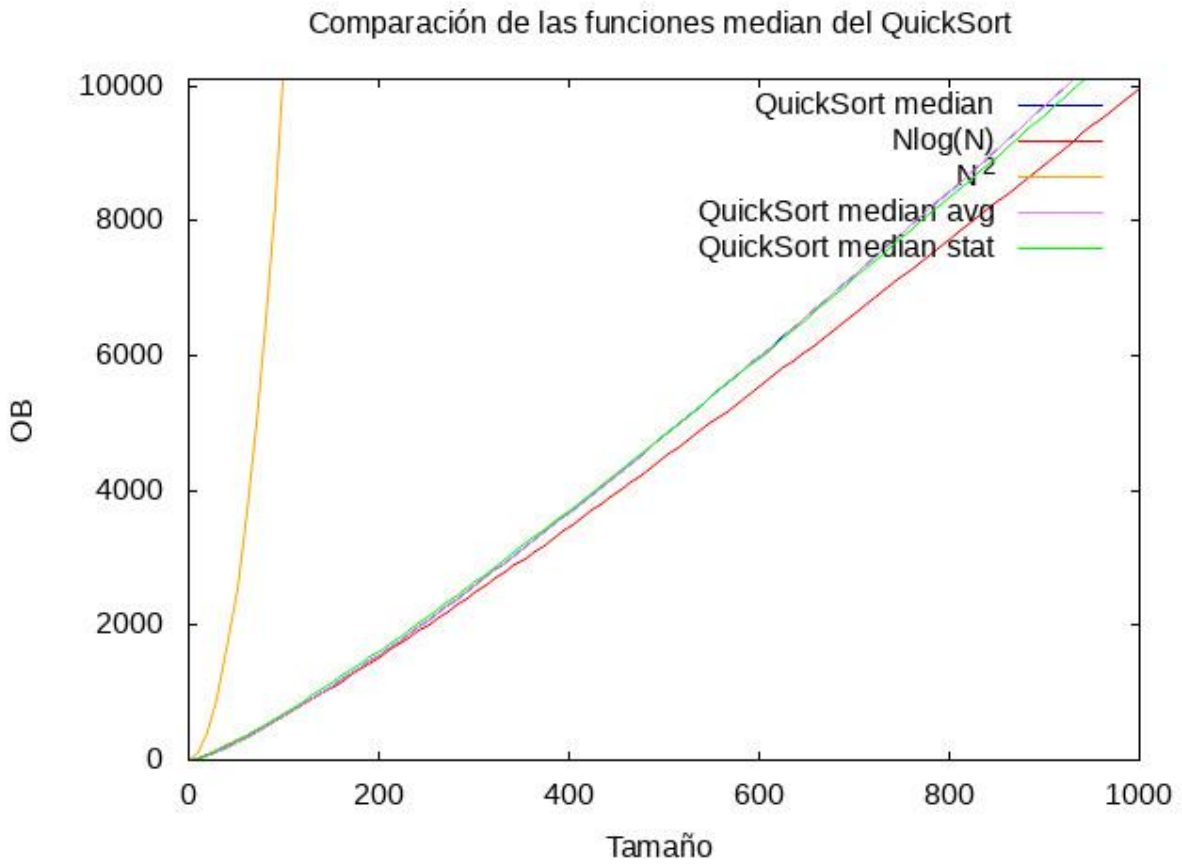


6.2 Razonad el resultado obtenido al comparar las versiones de quicksort con los diferentes pivotes tanto si se obtienen diferencias apreciables como si no.

Para tener una referencia hemos decidido añadir a la gráfica de tiempos la función MergeSort. No cabe la opinión en este aspecto debido al gran parecido de tiempo de las tres versiones del QuickSort. Como podemos ver, la diferencia en el tiempo de ejecución de las tres versiones es irrisoria, por lo tanto, concluimos que no hay diferencias apreciables entre ellas en términos de tiempo.



Por otro lado, vemos que si tenemos en cuenta las OBs la situación es igual, las funciones que representan el número de comparaciones de claves realizadas de cada uno de los *median* se superponen, pues la diferencia entre ellas es mínima.

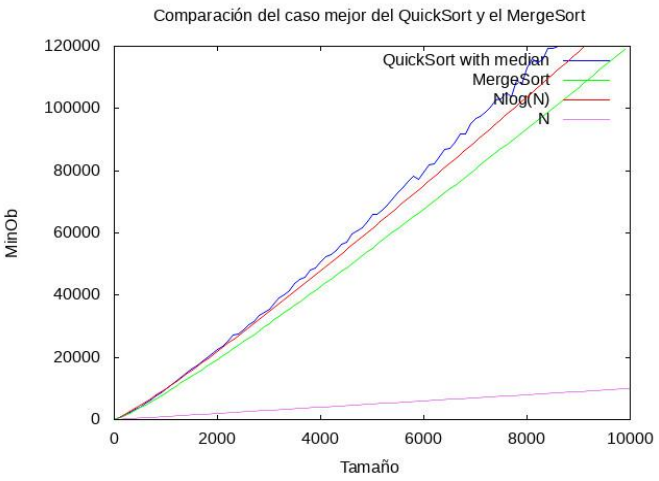


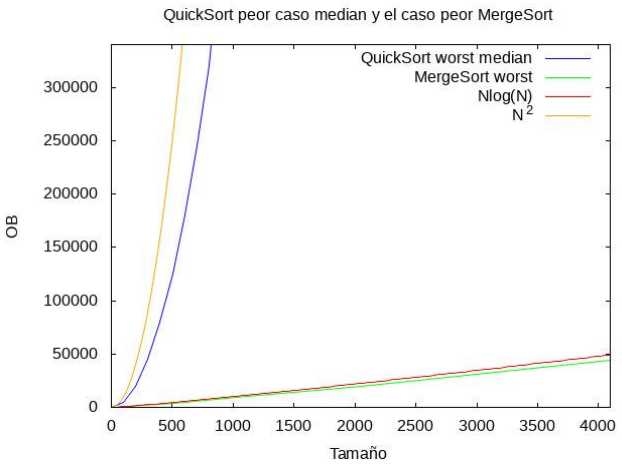
6.3 ¿Cuáles son los casos mejor y peor para cada uno de los algoritmos? ¿Qué habrá que modificar en la práctica para calcular estrictamente cada uno de los casos (también en el caso medio)?

Usando los conocimientos adquiridos en la teoría, sabemos que el mejor caso de MergeSort es la tabla ordenada. Mientras que el peor es algo más complejo y difícil de describir, para 8 elementos sería una lista de la siguiente forma: 4 8 2 6 3 7 1 5.

Por otro lado, el peor caso de QuickSort usando *median* son tanto la tabla ordenada como la invertida, usando *median_avg* es una tabla de esta forma 8 6 4 2 1 3 5 7, y usando *median_stat* el caso peor es mucho más difícil de encontrar, pero en cualquier caso se necesitan realizar N(N-1)/2 OBs para ordenarlas. Su mejor caso es muy complicado de calcular y no ha sido visto en teoría, pero tiene un coste muy similar al de MergeSort.

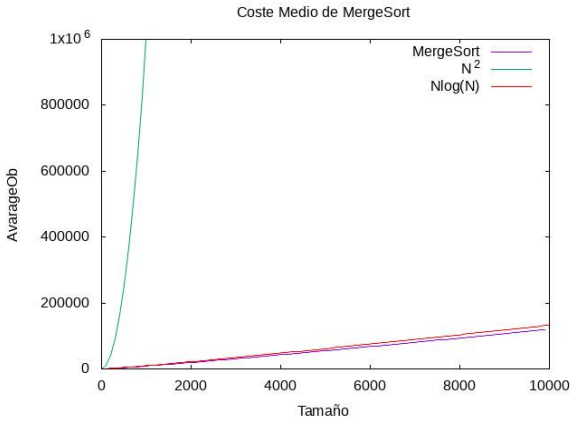
Para calcular estrictamente el mejor y peor caso de ambos algoritmos, hay que saber cuáles son las permutaciones para las que son menos eficientes, construirlas artificialmente y posteriormente ordenarlas mientras se cuentan las OBs ejecutadas en el proceso. Tras hacer todo lo anterior, hemos obtenido las siguientes gráficas:

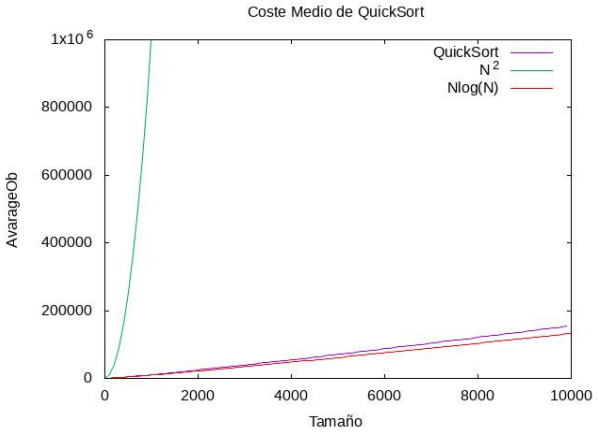




Como podemos ver a simple vista, los datos obtenidos empíricamente coinciden con la predicción teórica: un peor caso de QuickSort mucho más costoso que el de MergeSort, y dos mejores casos muy similares.

Por otro lado para calcular estrictamente el caso medio para un determinado tamaño habría que introducir una vez cada una de las distintas permutaciones, y hacer la media de las operaciones básicas ejecutadas. El problema principal para tamaños medianos y grandes es que el número de permutaciones posibles es de N!, por lo tanto el coste de dicho programa sería demasiado alto. Hemos obtenido una aproximación introduciendo un alto volumen de listas desordenadas:





6.4 ¿Cuál de los dos algoritmos estudiados es más eficiente empíricamente? Compara este resultado con la predicción teórica. ¿Cual(es) de los algoritmos es/son más eficientes desde el punto de vista de la gestión de memoria? Razona este resultado.

En términos de tiempo medio (suponiendo la equiprobabilidad de las entradas), se puede afirmar que el algoritmo QuickSort es ligeramente más eficiente que el MergeSort. Como puede verse en la gráfica adjunta en el apartado 5.2, QuickSort suele tardar algo menos que MergeSort.

Por otro lado, MergeSort presenta un coste mucho más constante (las OBs <= Nlg(N) +O(N)) mientras que el peor caso de QuickSort es N^2 -N/2 (el array ordenado). Esto puede empeorar significativamente el rendimiento del algoritmo si es empleado para ordenar listas que están "demasiado" ordenadas o "demasiado" desordenadas, pues el coste de QuickSort crece muy rápido en dichos casos. Por lo tanto, definir qué algoritmo es más eficiente depende mucho de la aplicación y la probabilidad de que aparezca cada entrada.

Además, desde el punto de vista de la gestión de memoria QuickSort es indiscutiblemente mejor que MergeSort. Esto se debe a que MergeSort va haciendo numerosas reservas de memoria a lo largo del proceso de ordenación, mientras que QuickSort no necesita reservar memoria extra.

Con todo esto, concluimos que cada algoritmo presenta sus propias ventajas e inconvenientes. No hay un ganador indiscutible, hay que evaluar cuáles son las prioridades y recursos del programa que deseamos implementar: MergeSort necesita emplear memoria y es ligeramente más lento en su caso medio, por otro lado ofrece un tiempo máximo de ordenación mucho más bajo que QuickSort; mientras que QuickSort es perfecto para programas con memoria disponible limitada y equiprobabilidad de permutaciones de entrada, pero no es una gran idea si se pretende ordenar listas con "pocas" o "demasiadas" inversiones.

7. Conclusiones finales.

Podemos concluir que a diferencia de otros análisis en este caso no hay una solución clara. A lo largo de esta práctica hemos podido ver las luces y sombras de cada uno de los algoritmos. MergeSort y QuickSort son algoritmos que se complementan muy bien, pues uno funciona en lo que el otro falla. Además, tal y como hemos visto en teoría son algoritmos de ordenación que rozan la perfección en términos de eficiencia y son significativamente mejores que los locales. Por lo tanto, ambos son muy buenos y la elección entre ellos depende principalmente teniendo en cuenta las características del programa en el que serán implementados.