Análisis de Algoritmos 2022/2023

Práctica 2

Andrés Goldwasser, Ignacio Núñez, Grupo 1271.

Código	Gráficas	Memoria	Total

1. Introducción.

En esta práctica se implementarán en C dos nuevos algoritmos de ordenación: Mergesort y Quicksort. Se analizará su eficiencia y se utilizarán varios métodos para determinar el pivote en Quicksort.

2. Objetivos

2.1 Apartado 1

Se buscará implementar el algoritmo Mergesort en C, y la función merge para facilitar su realización. Esta versión de Mergesort devolverá el número de operaciones básicas realizadas.

2.2 Apartado 2

Se realizarán ligeros cambios al programa exercise5 para poder obtener los datos necesarios para el análisis del algoritmo Mergesort.

2.3 Apartado 3

Se implementará el algoritmo Quicksort en C, mediante la función partition y la subfunción median para determinar el pivote (en su caso, siempre el primer elemento). Esta versión de Quicksort además devolverá el número de operaciones básicas realizadas.

2.4 Apartado 4

Se realizarán ligeros cambios al programa exercise5 para poder obtener los datos necesarios para el análisis del algoritmo Quicksort.

2.5 Apartado 5

Se desarrollarán dos subfunciones para elegir el pivote en el algoritmo Quicksort. Una tomará el elemento en el medio de la tabla, y la otra elegirá un elemento de valor medio entre tres dentro de la tabla.

3. Herramientas y metodología

Todas las funciones se han escrito en C mediante Visual Studio Code. Se han compilado utilizando el makefile de la práctica 1, y en todo caso se ha verificado mediante ejecución que los algoritmos funcionan correctamente, y no hay errores o fugas de memoria (confirmado mediante valgrind).

3.1 Apartado 1

Se implementó el algoritmo como función en C mediante una interpretación del pseudocódigo de Mergesort y merge. Se ha incluido la variable count para mantener cuenta del número de operaciones básicas realizadas, y es lo que devuelve la función tras su realización.

3.2 Apartado 2

La modificación de exercise5 es mínima, necesitando de solo el cambio de "SelectSort" a "Mergesort" para ejecutar correctamente.

3.3 Apartado 3

Utilizando el pseudocódigo de Quicksort y partition, ligeramente adaptado al uso de la variable adicional "pos" para manejar la posición del pivote. La función median simplemente asigna la posición inicial (ip) a pos- que se declara inicialmente como un int en la función quicksort. Se mantiene la cuenta de las operaciones básicas mediante la variable count, que se incrementa cuando se ejecuta la cdc en partition.

3.4 Apartado 4

La modificación de exercise5 es mínima, necesitando de solo el cambio de "SelectSort" a "Quicksort" para ejecutar correctamente.

3.5 Apartado 5

La función median_avg es similarmente simple a la función median, simplemente asignándole a pos la posición intermedia entre ip e iu. Por otro lado, median_stat obtiene los valores de tres casillas (ip, iu y la intermedia) y elige la posición del valor intermedio entre los tres. Aunque requiere de tres operaciones básicas adicionales.

4. Código fuente

4.1 Apartado 1

```
int mergesort(int* tabla, int ip, int iu)
{
  int im=0, count=0, ret;

  /*CONTROL ERROR*/
  if(!tabla || ip > iu){
    return ERR;
  }

  /********BASE CASE********/
  if(ip == iu){
    return OK;
  }
```

```
  /********GENERAL CASE********/

  /*CALCULATES MEDIUM ELEMENT*/
  im = (ip+iu)/2;

  /*SORT FIRST HALF*/
  ret = mergesort(tabla, ip, im);
  if(ret==ERR){
    return ERR;
  }
  count += ret;

  /*SORT SECOND HALF*/
  ret = mergesort(tabla, im+1, iu);
  if(ret==ERR){
    return ERR;
  }
  count += ret;

  /*MERGE BOTH HALVES*/
  ret = merge(tabla, ip, iu, im);
  if(ret==ERR){
    return ERR;
  }
  count += ret;

  return count;

}

int merge(int* tabla, int ip, int iu, int imedio){
  int *aux;
  int count=0;
  int i,j,k;

  /*CONTROL ERROR*/
  if(!tabla || ip > imedio || imedio > iu){
    return ERR;
  }

  /*SAVES MEMORY FOR A SIZED TABLE*/
  aux = (int*) malloc((iu-ip+1)*sizeof(int));
  /*MEMORY CONTROL ERROR*/
  if(aux==NULL){
    return ERR;
  }

  /*FILLS THE NEW TABLE*/
  for(i=ip, j=imedio+1, k=0 ; i<imedio+1 && j<iu+1 ;k++){
```

```
    if(tabla[i] < tabla[j]){
      aux[k] = tabla[i];
      i++;
    }
    else{
      aux[k] = tabla[j];
      j++;
    }
    count++;
  }

  /*ADDS THE REST OF THE ELEMENTS*/
  while(i<imedio+1){
    aux[k] = tabla[i];
    i++;
    k++;
  }
  while(j<iu+1){
    aux[k] = tabla[j];
    j++;
    k++;
  }

  /*COPIES THE TABLE ON THE TABLE THAT WAS SENT*/
  if (icopytable(aux, tabla, ip, iu) == ERR){
    free(aux);
    return ERR;
  }

  /*FREES MEMORY*/
  free(aux);

  return count;

}
```

4.3 Apartado 3

```
int median(int *tabla, int ip, int iu,int *pos){
  if(!tabla||iu<ip||!pos) return ERR;

  *pos=ip;
  return 0;
}

int partition(int* tabla, int ip, int iu,int *pos){
  int val,auxval1,auxval2,i,count,check;

  if(!tabla||iu<ip||!pos) return ERR;
```

```
  check=median(tabla,ip,iu,pos);
  if(check==ERR)
    return ERR;

  count=0;

  val=tabla[*pos];

  auxval1=tabla[ip];
  tabla[*pos]=auxval1;
  tabla[ip]=val;

  *pos=ip;

  for(i=(*pos)+1;i<=iu;i++){
    if(tabla[i]<val){
      (*pos)++;

      auxval1=tabla[i];
      auxval2=tabla[*pos];
      tabla[i]=auxval2;
      tabla[*pos]=auxval1;
    }
    count++;
  }

  auxval1=tabla[ip];
  auxval2=tabla[*pos];
  tabla[ip]=auxval2;
  tabla[*pos]=auxval1;

  return count;
}

int quicksort(int* tabla, int ip, int iu){
  int pos,count;

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

  count=0;

  if(ip==iu)
    return 0;
  else{
    count+=partition(tabla,ip,iu,&pos);
    if(ip<pos-1){
      count+=quicksort(tabla,ip,pos-1);
    }
```

```
    if (pos+1<iu){
      count+=quicksort(tabla,pos+1,iu);
    }
  }

  return count;
}
```

4.5 Apartado 5

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

  *pos=((iu+ip)/2);

  return 0;
}

int median_stat(int *tabla, int ip, int iu, int *pos){
  int i,j,k;
  if (!tabla||iu<ip||!pos) return ERR;

  i=tabla[ip];
  j=tabla[iu];
  k=tabla[(ip+iu)/2];

  if((i<=j && i>=k) || (i>=j && i<=k)){
    *pos=ip;
    return 0;
  }
  if((j<=i && j>=k) || (j>=i && j<=k)){
    *pos=iu;
    return 0;
  }
  if((k<=j && k>=i) || (k>=j && k<=i)){
    *pos=(ip+iu)/2;
    return 0;
  }

  return ERR;

}
```

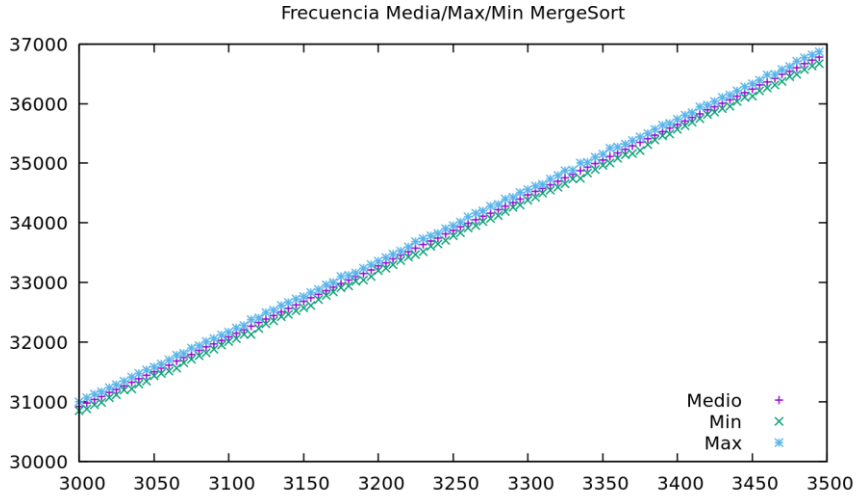
5. Resultados, Gráficas

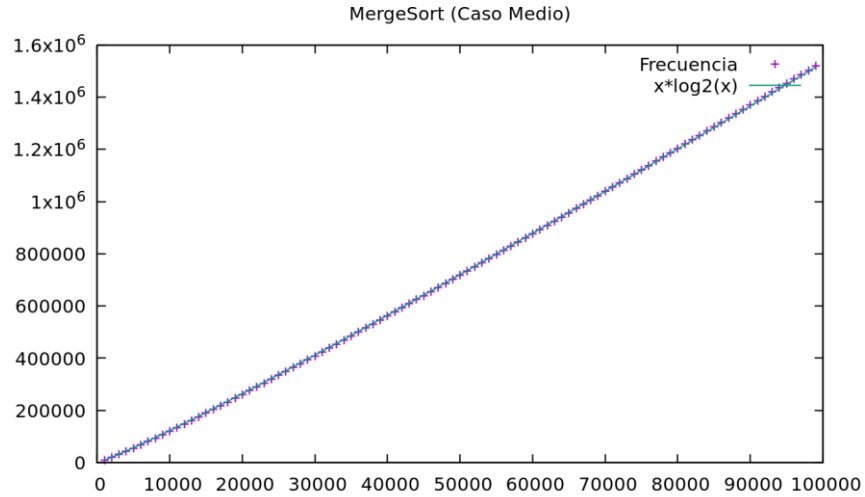
5.1 Apartado 1

El algoritmo de Mergesort, y su función merge asociada funcionan perfectamente, ordenando de manera correcta cuando se utiliza en exercise4.

5.2 Apartado 2

Resultados del apartado 2.

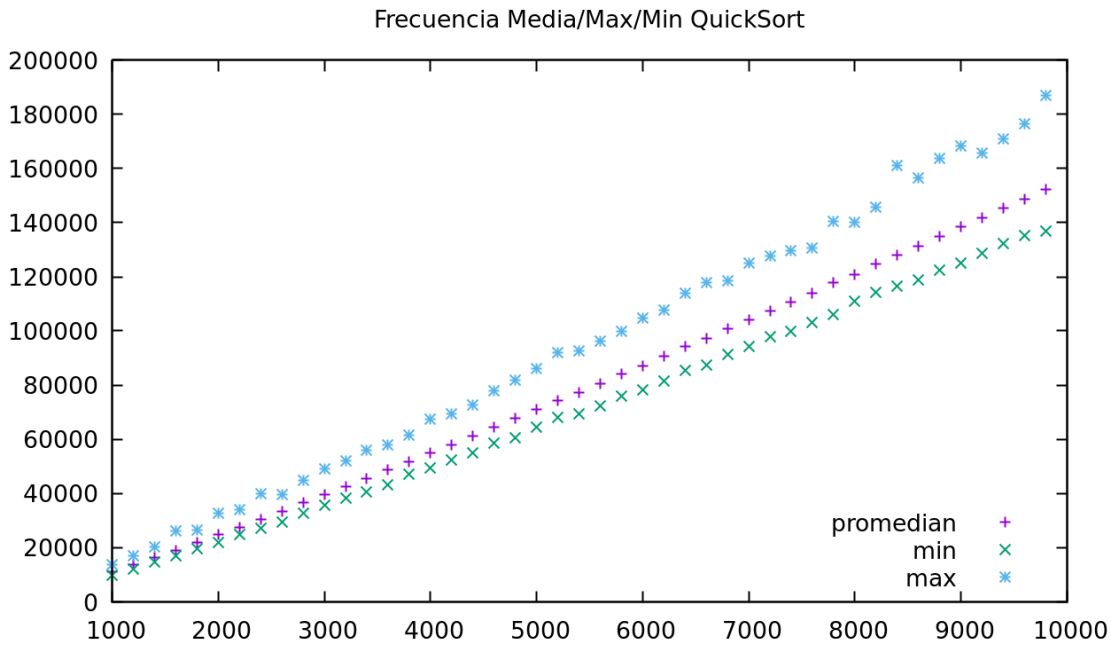




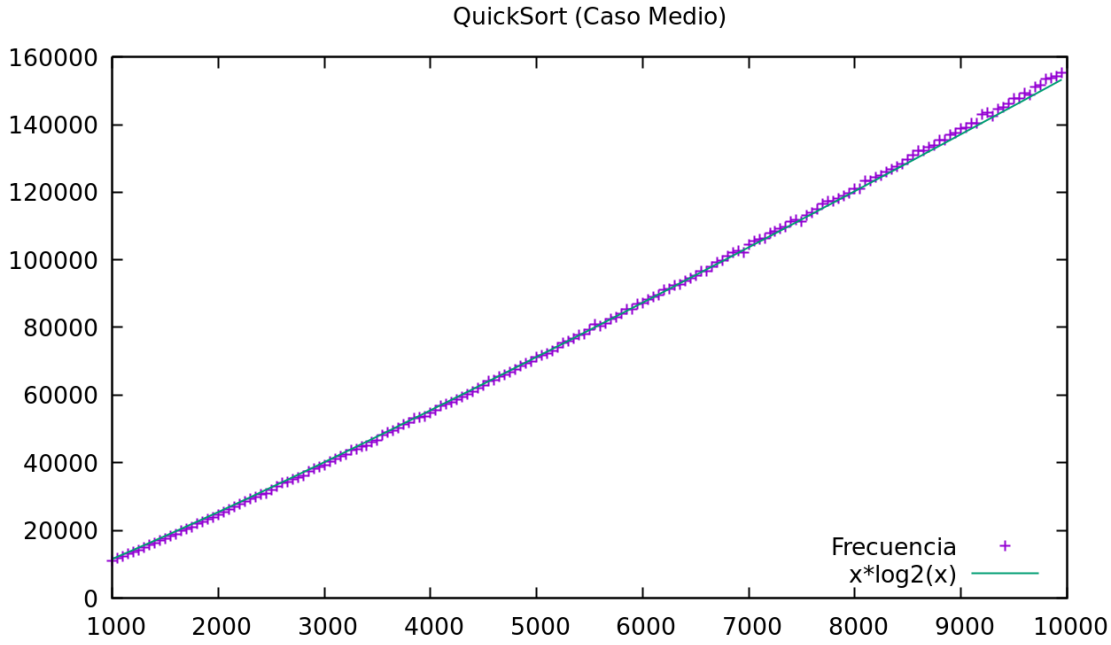
5.3 Apartado 3

Una vez más, la implementación se realizó sin problemas, y cuando se aplica la función Quicksort en exercise4 se ordena correctamente la permutación generada.

5.4 Apartado 4

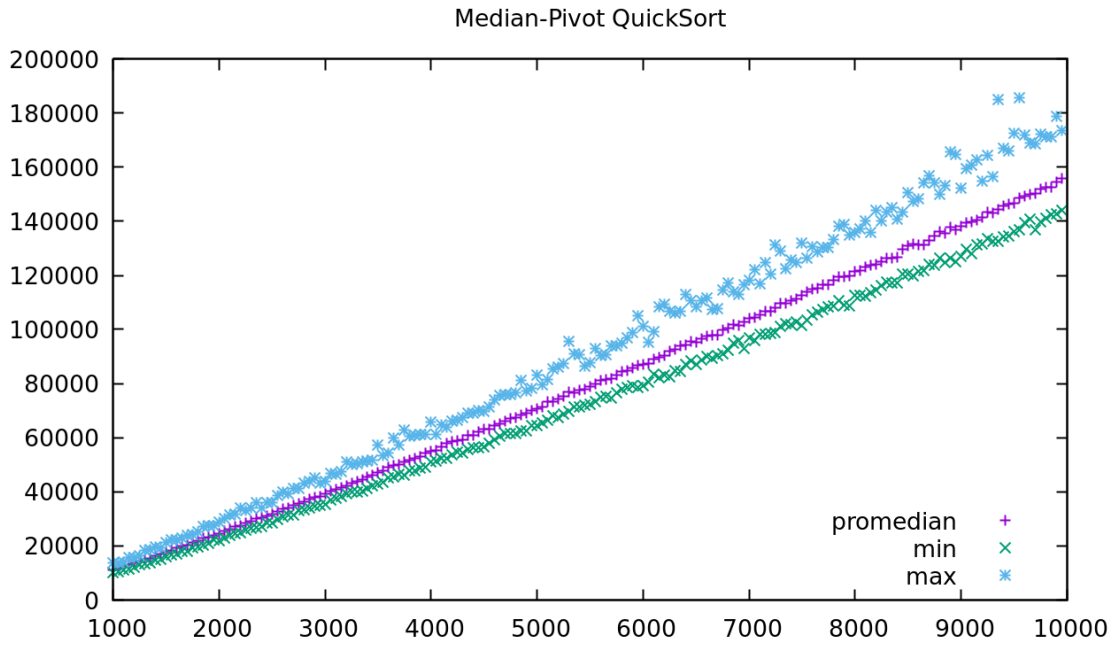


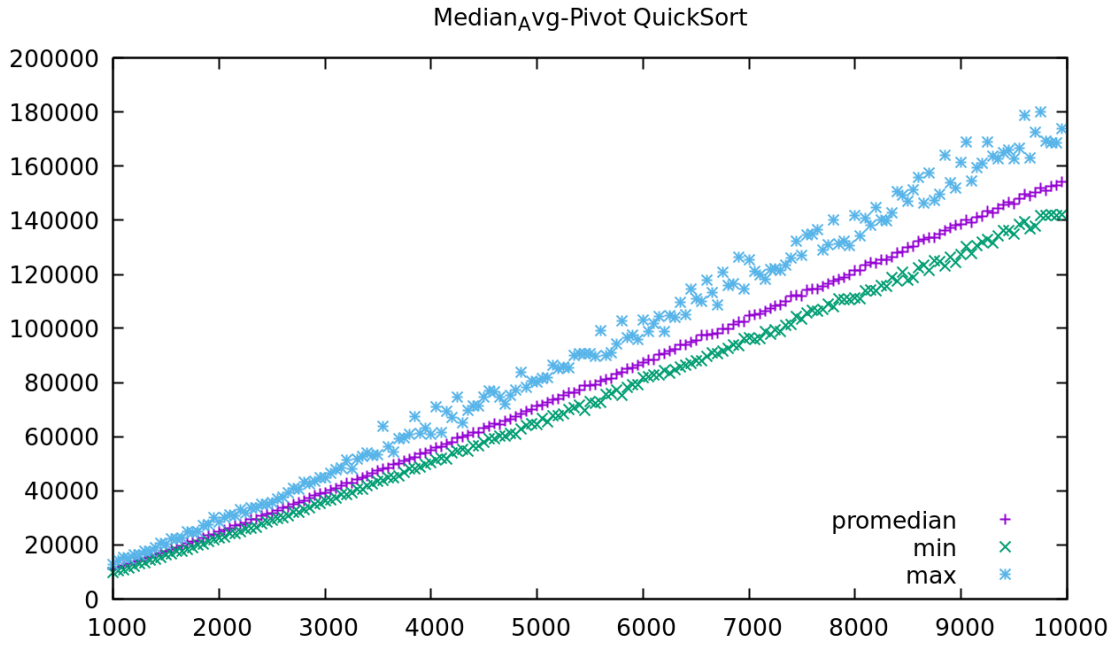
A diferencia de Mergesort, Quicksort tiene mucha más variabilidad, debido al azar involucrado en la elección del pivote. Por ende se observan máximos mayores y mínimos menores a los del otro algoritmo.



5.5 Apartado 5

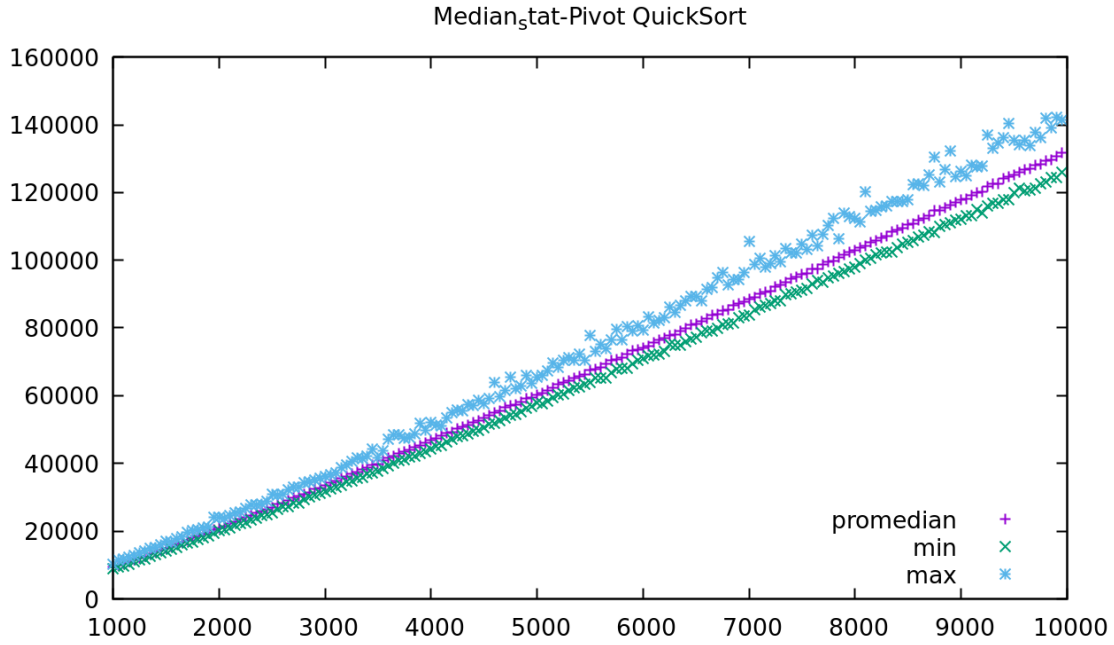
Tanto median como median_avg seleccionan una posición fija, por lo que su eficiencia es casi completamente al azar, dependiendo de que tan cerca a la media de la tabla se asemeja el número. Por ende, a pesar de elegir posiciones distintas, su eficiencia es prácticamente igual:



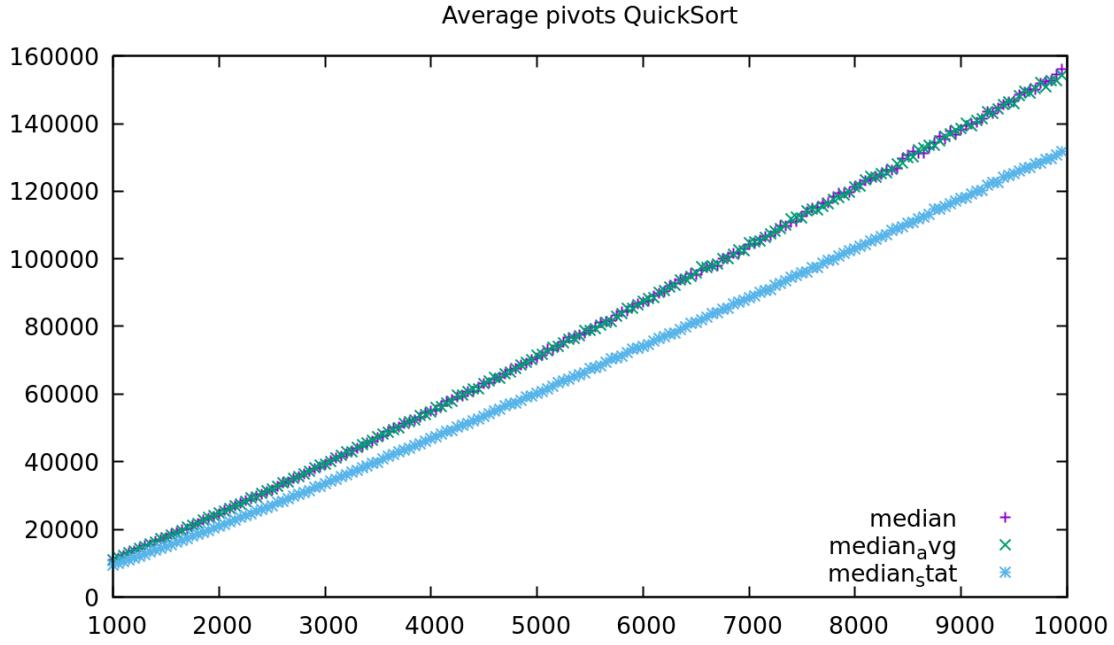


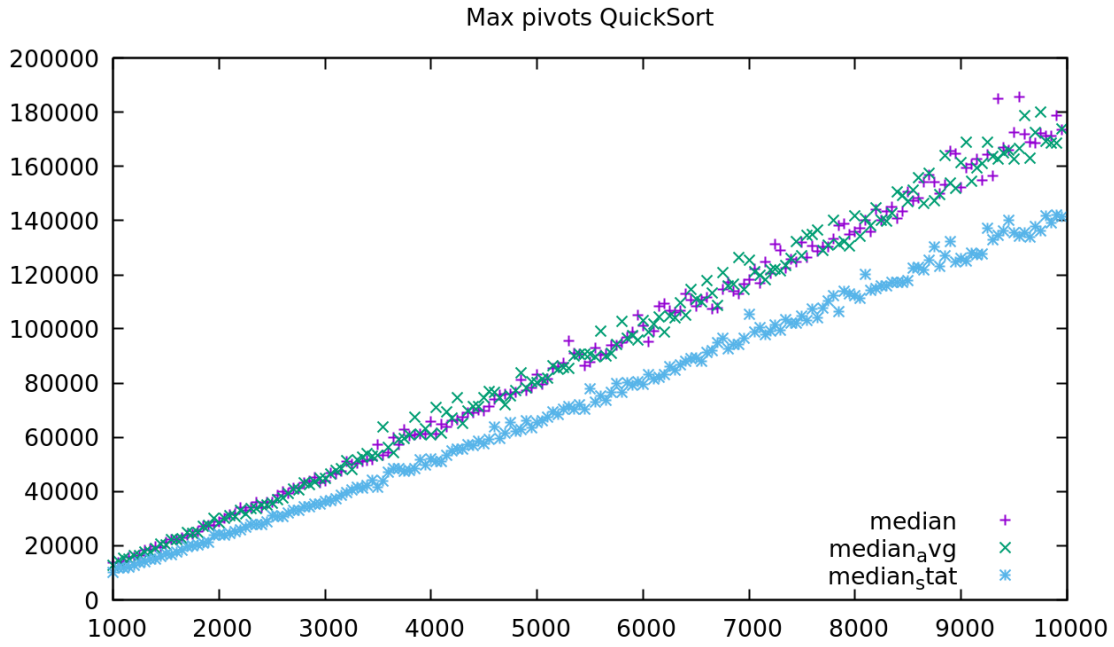
Pero por otro lado, median_stat selecciona un número intermedio entre tres posibilidades. Aunque esto significa más OBs en cada selección del pivote (las

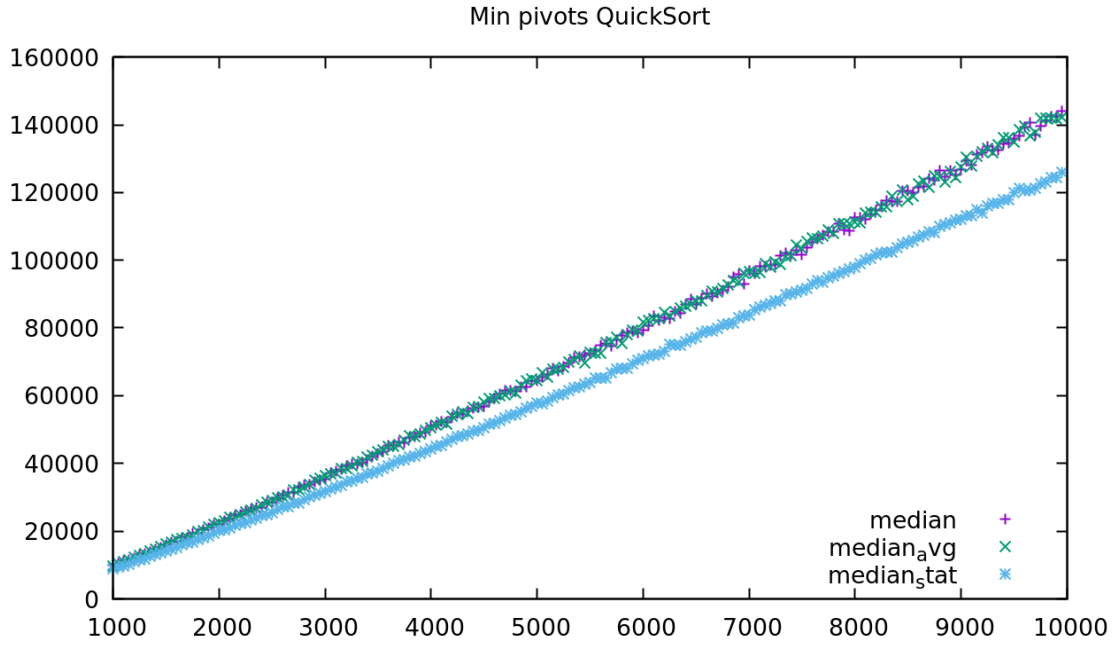
comparaciones), a la larga se vuelve mas eficiente el proceso de Quicksort, ya que el pivote es siempre el mejor candidato de los tres a la hora de ordenar la tabla.



Aquí se ilustra la clara diferencia entre los métodos, y como stat es más eficiente que las demás opciones.





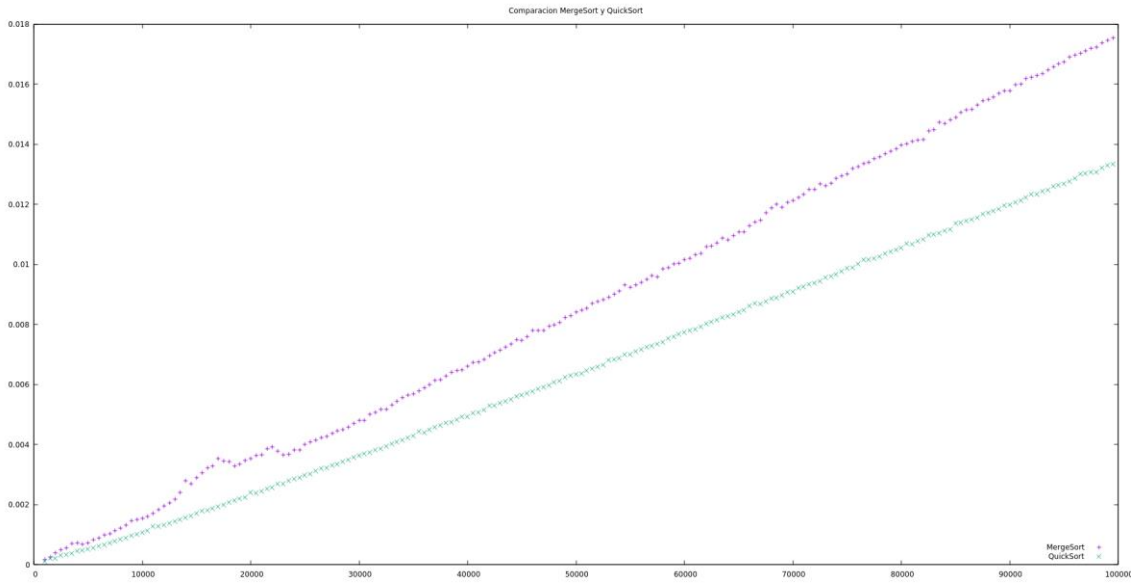


5. Respuesta a las preguntas teóricas.

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

5.1 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 por qué ocurre esto.

A continuación se muestra un gráfico con el tiempo de ejecución de mergesort y Quicksort para los mismos casos.



Podemos observar que Quicksort es más rápida (además también posee menos comparaciones de clave). Vemos que existe algún pico en la gráfica de mergesort pero no es muy significativo (puede tratarse de algún retraso puntual al guardar y liberar memoria). Aunque ambas gráficas posean el mismo rendimiento teórico (n*log2(n)), Quicksort es algo más rápido porque no necesita de liberar y guardar memoria.

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

Como podemos apreciar en el gráfico del apartado 5.4, median_stat es mucho más eficiente que los otros dos pivotes, que tienen un comportamiento prácticamente idéntico. Tenemos que entender que la elección del pivote Quicksort es clave para la eficiencia del mismo, pues una mala elección divide la tabla en dos partes desiguales y por lo tanto condena a realizar más cambios y más operaciones básicas. Evidentemente median no es una buena elección de pivote porque devuelve el primer elemento, que "dejará" al azar si se trata de una buena elección o no. Median_avg elige como pivote el elemento del medio. Puede parecer que mejora en algo a median pero en realidad se trata del mismo caso, vuelve a elegir un elemento al azar que podría o no ser buena elección. Por último, median_stat sí tiene un comportamiento más inteligente, ya que elegirá de entre el primer, último y elemento del medio del array cuál es el mejor pivote mediante una serie de comparaciones muy simple. Esta elección a tres es la que hace que el funcionamiento del algoritmo mejore notoriamente.

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

Para tanto Mergesort como para Quicksort, el caso medio es uno de distribución aleatoria. Como esta realizado actualmente el ejercicio (utilizando la creación de

permutaciones en orden al azar con generate_perm), ya se tiene el método para calcular el tiempo medio (y ya se ha calculado en las gráficas anteriores).

Por otro lado, para obtener el tiempo máximo para ambos algoritmos se necesita una tabla perfectamente ordenada. Para obtener el tiempo máximo, bastaría con modificar la función generate_perm para no cambiar el orden de los elementos en un array, y asi siempre se obtendrá una cadena ordenada que causará el peor caso posible para los algoritmos.

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

Empíricamente, Mergesort es un poco más eficiente que Quicksort. Esto se evidencia en el caso teórico (gráficas), donde se aprecia una pequeña diferencia entre ambos algoritmos. Desde el punto de vista de gestión de memoria, por otro lado, Quicksort es mucho mas eficiente, ya que opera directamente sobre la misma tabla, solo manejando distintos segmentos, mientras que Mergesort aloca y libera memoria múltiples veces durante su ejecución.

6. Conclusiones finales.

Una vez finalizada la práctica podemos determinar que los objetivos de la misma se han cumplido de manera muy satisfactoria. Se han realizado todos los objetivos y se han respondido todas las preguntas de forma coherente a los resultados obtenidos. Los resultados obtenidos tanto en “MergeSort” como en “QuickSort” se ajustan mucho a los teóricos, lo que implica que tanto la función de ordenación como la función de medición de tiempo están codificadas de manera correcta.

Se ha continuado usando los mismos entornos de trabajo que para la práctica anterior que han seguido siendo adecuados y cómodos para cada miembro de la pareja (Linux, Ubuntu, VSC, Gnuplot, GitHub…). El reparto de la tarea ha sido equitativo y ambos miembros de la pareja han trabajado satisfactoriamente. Cada miembro de la pareja se ocupó específicamente en el trabajo de una de las funciones de ordenación pero aun así el otro miembro se interesó por el trabajo del compañero.