Análisis de Algoritmos 2018/2019

Práctica 2

Román García, 1271 9.

Código	Gráficas	Memoria	Total

1. Introducción.

En esta práctica se propone la comparación de dos algoritmos del tipo “Divide y vencerás”

A continuación se muestra el proceso seguido para la elaboración de la misma.

2. Objetivos

2.1 Apartado 1

En este apartado tenemos como objetivo la implementación del algoritmo “MergeSort”, así como las rutinas necesarias para su correcta ejecución.

2.2 Apartado 2

Una vez implementado el algoritmo, se pide el cálculo de tiempos de ejecución y de operaciones básicas en función del tamaño, así como su representación y comparación con resultados teóricos.

2.3 Apartado 3

Para este apartado se pide la implementación del algoritmo “QuickSort”, además de las rutinas necesarias para su correcto funcionamiento.

2.4 Apartado 4

Como en el apartado 2, se pide la relación de tiempos de ejecucion y operaciones básicas con el tamaño usado para la ejecucion del algoritmo, así como su representación y comparación con cálculos teóricos. Esta vez para “QuickSort”.

2.5 Apartado 5

Para finalizar, en este apartado, se comparará el rendimiento de “QuickSort” usando su implementación recursiva, con su implementación sin recursión de cola.

3. Herramientas y metodología

Para todos los apartados he utilizado la misma metodología y las mismas herramientas:

Para saber qué piden exactamente he usado un boceto en papel de lo que debería hacer cada una de las funciones a codificar.

Una vez analizado el problema y diseñada la solución, he usado la combinación del IDE “Visual Studio Code” para codificar, “GitHub” para mantener un repositorio de versiones y una “bash Ubuntu 18.04” de “Windows” que he utilizado para compilar y ejecutar el código.

Para la creación de gráficas he usado “Gnuplot” y bash script para el procesamiento de datos.

4. Código fuente

Aquí ponéis el código fuente **exclusivamente de las rutinas que habéis desarrollado vosotros** en cada apartado.

4.1 Apartado 1

```
int merge(int* tabla, int ip, int iu, int medio)
{
  int* tabAux;
  int i, j, k, ob = 0;

  tabAux = (int*) malloc(sizeof(int) * (iu-ip+1));

  if(!tabAux)
    return ERR;

  i = ip;
  j = medio +1;
  k = 0;

  while( i <= medio && j <= iu ){


    if(tabla[i] < tabla[j]){

      tabAux[k] = tabla[i];
      i++;

    } else {
      tabAux[k] = tabla[j];
      j++;

    }


    k++;
  }

  ob=k;

  if( i > medio){

    if(j > iu){

    }
```

```
    while(j <= iu){

      tabAux[k] = tabla[j];
      j++;
      k++;
    }

  } else if(j > iu){


    if(i > medio){

    }

    while(i <= medio){
      tabAux[k] = tabla[i];
      i++;
      k++;
    }

  }


  copiar(tabAux, tabla, ip, iu);
  free(tabAux);

  return ob;

}
int mergesort(int* tabla, int ip, int iu)
{
  int M = (ip + iu)/2;
  int ob = 0;

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

  if(ip == iu)
    return 0;

  ob += rec_mergesort(tabla, ip, M);
  ob += rec_mergesort(tabla, M+1, iu);

  return ob + merge(tabla, ip, iu, M);

}
int rec_mergesort(int* tabla, int ip, int iu)
{
```

```
  int M = (ip + iu)/2;
  int ob = 0;

  if(ip == iu)
    return 0;

  ob += rec_mergesort(tabla, ip, M);
  ob += rec_mergesort(tabla, M+1, iu);

  return ob + merge(tabla, ip, iu, M);

}
```

4.3 Apartado 3

```
int quicksort(int* tabla, int ip, int iu){
  int m;
  int ob = 0;

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

  else if (ip==iu){
    return OK;
  }

  else{

    ob+= partir(tabla, ip, iu, &m);

    if(ip<m-1){
      ob+= quicksort(tabla, ip, m-1);
    }

    if (m+1 < iu){
      ob+= quicksort(tabla, m+1, iu);
    }

    return ob;
  }
}
int partir(int* tabla, int ip, int iu, int* pos){
  int k, i;
  int ob = 0;

  if(!tabla || !pos){
```

```
    return ERR;
  }

  medio(tabla, ip, iu, pos);

  k=tabla[*pos];

  swap( &tabla[ip], &tabla[*pos]);
  *pos=ip;

  for(i=ip+1; i<=iu; i++){
    ob++;
    if(tabla[i]<k){
      (*pos)++;
      swap( &tabla[i], &tabla[*pos]);
    }
  }
  swap( &tabla[ip], &tabla[*pos]);
  return ob;
}
int medio(int *tabla, int ip, int iu, int *pos)
{
  if(!tabla){
    return ERR;
  }

  *pos = ip;
  return 0;
}
```

4.5 Apartado 5

```
int quicksort_src(int* tabla, int ip, int iu){
  int m, aux, i;
  int ob = 0;

  if (!tabla ){
    return ERR;
  }

  if (ip>iu || ip==iu){
    return 0;
  }

  else{

    ob += partir(tabla, ip, iu, &m);
```

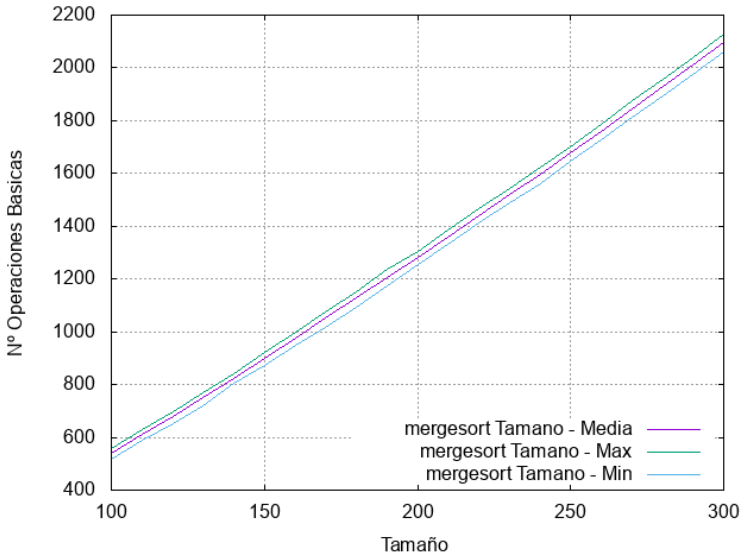
```
    for(aux = m-1, i=2; i  > 0; i--  ){
        ob += quicksort_src(tabla, ip, aux);
        ip = aux +2;
        aux = iu;
    }

    return ob;
  }
}
```

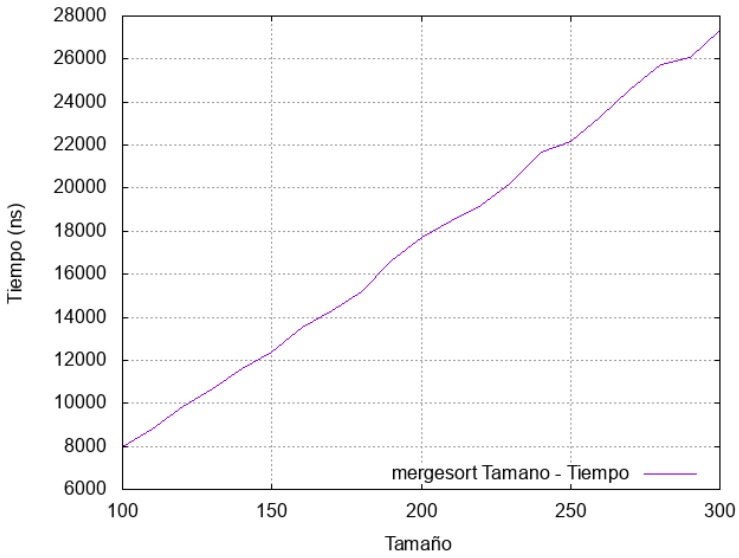
5. Resultados, Gráficas

Aquí ponis los resultados obtenidos en cada apartado, incluyendo las posibles gráficas.

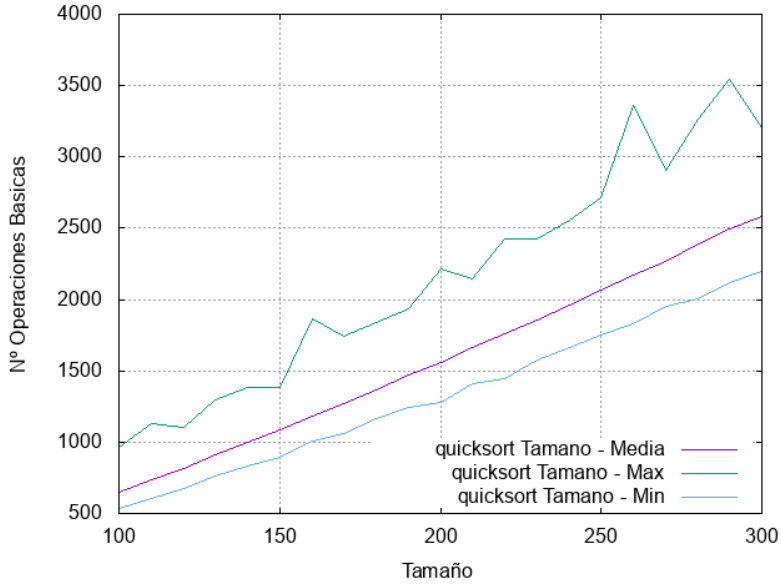
.

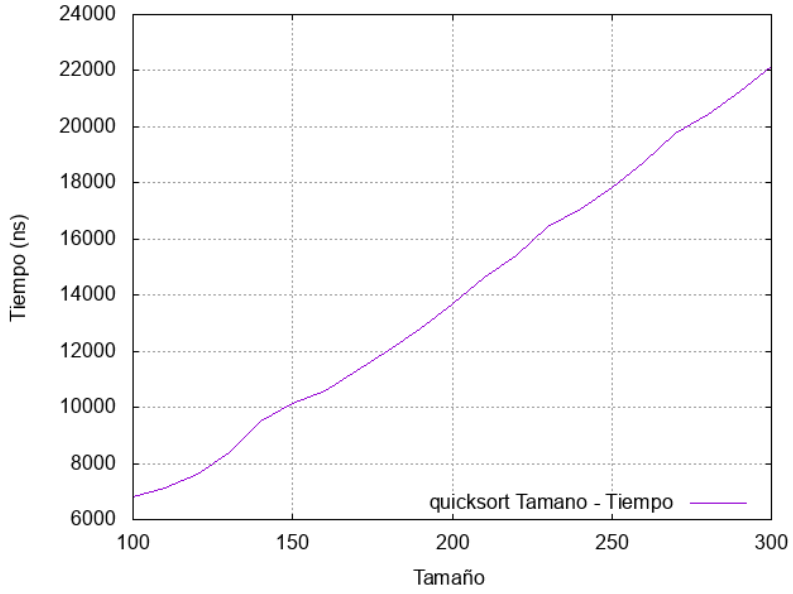
5.2 Apartado 2





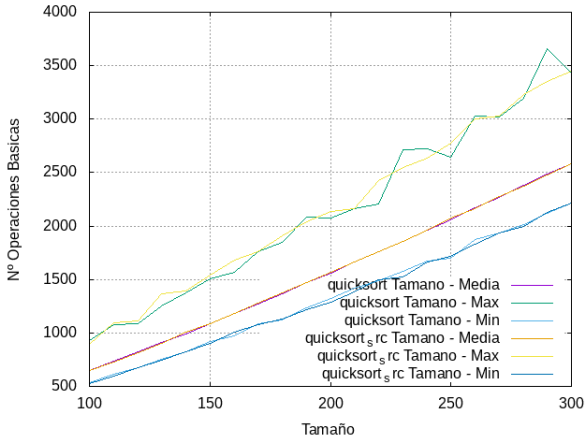
En las gráficas anteriores podemos observar el rendimiento de Mergesort en base a su tamaño.

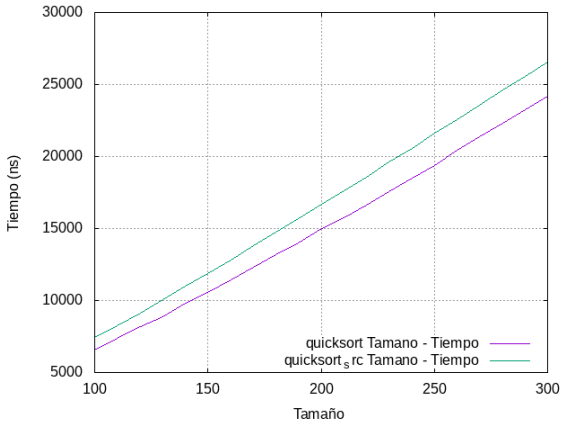




En las gráficas anteriores podemos observar el rendimiento de Quicksort en base a su tamaño.

5.5 Apartado 5





En las gráficas anteriores podemos apreciar la diferencia entre la implementación recursiva e iterativa de QuickSort.

5. Respuesta a las preguntas teóricas.

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

Sabemos que el rendimiento medio de QS es:

$$A_{QS}(N) = 2NLog(N) + O(N)$$

De esta igualdad podemos extraer que el rendimiento tiene una inclinación mayor que una funcion lineal y que es ascendente. Como podemos ver en las gráficas, la información extraída de las mismas coincide con los cálculos teóricos.

El rendimiento de MS es similar:

$$A_{MS}(N) = \Theta(NLog(N))$$

En ambos casos podemos observar algunos picos relacionados con máximos y mínimos. Se dan porque las tablas se desordenan de forma pseudoaleatoria, por lo tanto, no siempre será el mejor caso, ni el peor ni el medio.

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)?

$$W_{QS}(N) = \frac{N^2}{2} - \frac{N}{2}$$

$$W_{MS}(N) = NLg(N) + O(N)$$

$$B_{QS}(N) = NLg(N)$$

$$B_{MS}(N) => \frac{1}{2} * Nlg(N)$$

Para conseguir los tiempos mejor y peor de los algoritmos lo que hace falta es generar una permutación que tenga el orden que genera dichos tiempos: en Quicksort este tiempo se consigue con una tabla ya ordenada mientras que con Mergesort se consigue con una tabla ordenada inversamente.

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.

En ambos casos la respuesta sería QuickSort. Tiene un rendimiento medio mejor que MergeSort y MergeSort necesita memoria dinámica.

5.5 Indica si observas alguna diferencia en el rendimiento al eliminar la recursión de cola de Quicksort. Razona las diferencias de rendimiento que has encontrado

Existe una diferencia debida al numero de instrucciones reales que ejecuta cada algoritmo: ambos tienen la misma operación básica pero la implementación iterativa necesita cambiar los índices, lo cual hace que se retrase en la llamada, con respecto a la implementación recursiva

6. Conclusiones finales.

Como hemos observado a lo largo de esta práctica, hay distintos tipos de ordenación basada en “divide y vencerás”. No obstante, una característica que puede hacer semejante o diferente unos de otros es la implementación.