PRÁCTICA 1: EFICIENCIA DE ALGORITMOS

Autores:
José Teodosio Lorente Vallecillos
Miguel Torres Alonso
Mario Soriano Morón

ÍNDICE

0. Introducción

Este informe realiza un estudio integral sobre la eficiencia de una serie de algoritmos para resolver problemas específicos.

Se proponen dos bloques de algoritmos básicos: el primero incluye algoritmos de ordenación de vectores aleatorios de dimensión N ó también búsqueda del máximo y mínimo en vectores aleatorios de dimensión N mediante recursividad, y el segundo tiene los mismos objetivos y pueden ser iterativos o iterativos y recursivos.

En base a esto, todos estos códigos que analizaremos han sido ejecutados en las computadoras de los miembros del equipo y tienen todos los recursos disponibles para el proceso creado por el sistema operativo para el programa, y no se ven afectados por otros programas abiertos u otras interferencias que puedan afectar el desempeño de la ejecución, eficiencia práctica, y la investigación de la eficiencia empírica o teórica, para la futura comparación de la eficiencia híbrida.

A continuación especificamos los datos técnicos de nuestros PCs.

1. Especificaciones técnicas de los PCs

Ordenador de José Teo Lorente

MODELO: Lenovo IdeaPad Gaming 3
CPU: Intel® Core™ i7-10750H CPU @ 2.60GHz hasta 5.0GHz × 12
RAM: 16GB DDR4, 2933Hz
MEMORIA: 1TB SSD
TARJETA GRÁFICA: NVIDIA Corporation / NVIDIA GeForce GTX 1650/PCIe/SSE2
SISTEMA OPERATIVO: Ubuntu 20.04.2 LTS de 64 bits

Ordenador de Miguel Torres

MODELO: Lenovo IdeaPad 320
CPU: Intel® Core™ i5-8250U CPU @ 1.60GHz × 8
RAM: DDR4-SDRAM 2133 MHz, memoria interna de 8 GB
MEMORIA: 1TB SSD
TARJETA GRÁFICA: NV 138 / Mesa Intel® UHD Graphics 620 (KBL GT2)
SISTEMA OPERATIVO: Ubuntu 20.04.3 LTS de 64 bits

Ordenador de Mario Soriano

MODELO: ASUS ROG STRIX GL553VD-DM467T
CPU: Intel® Core™ i7-7700HQ CPU @ 2.80GHz × 8

RAM: 16 GB RAM DDR4 2133 MHz
MEMORIA: 512GB SSD
TARJETA GRÁFICA: NVIDIA® GeForce® GTX 1050 con 4GB
SISTEMA OPERATIVO: Ubuntu 20.04.2 LTS de 64 bits

2. Algoritmos recursivos

2.1. Algoritmo MaximoMinimoDyV

2.1.1. Eficiencia Teórica

Divide el array en dos partes para encontrar el máximo y el mínimo de cada mitad, dividiendo el vector en mitades hasta que el vector sea de 1 o 2 componentes y devuelve el máximo y mínimo encontrado.

Variable o variables de las que dependen el tamaño del caso: El problema del tamaño depende del número de componentes del vector. En concreto n = Cfin - Cini + 1.
Es un algoritmo recursivo, por tanto llamemos T(n) al tiempo de ejecución que tarda el algoritmo en resolver el problema de tamaño "n". Pueden existir tres casos:

- Caso primero: cuando n = 1, hace el "else if" y asigna Max y Min, por tanto es O(1).
- Caso segundo: cuando hace el "else", asigna Max y Min otra vez, por tanto O(1).
- Caso tercero: el caso general, el algoritmo calcula la parte central del vector en la variable "mitad", lo cual es O(1). Después hace una llamada recursiva para resolver un subproblema desde Cini hasta mitad, cuyo tamaño es n/2. Si T(n) es el tiempo que tarda el algoritmo para resolver el problema de tamaño n, entonces la llamada recursiva tendrá un tiempo de ejecución T(n/2). Luego hace otra llamada recursiva para resolver un subproblema desde centro+1 hasta posFin, cuyo tamaño asintótico también se puede aproximar por n/2. Al igual que la línea anterior, la llamada recursiva tendrá un tiempo de ejecución T(n/2). Luego asigna Max y Min que es O(1). Por tanto podemos aproximar T(n) como T(n) = 2T(n/2).

Para estudiar el algoritmo contamos el nº de asignaciones y de de elementos del tipo “tipo” en el caso mejor, peor y medio.

	Comparaciones	Asignaciones
Mejor caso	$n – 1 \in O(n)$	$2 \in O(1)$
Peor caso	$2(n - 1) \in O(n)$	$n + 1 \in O(n)$
Caso promedio	$(n-1) + \Sigma(i-1)/i = 2(n-1) + \Sigma 1/i \approx 2(n-1) - \ln n \in O(n)$	$2 + 2 \cdot \ln n \in O(\log n)$

2.1.2. Eficiencia Práctica

El tiempo de ejecución lo mediremos como la diferencia entre el instante de tiempo justo anterior al inicio del algoritmo, y el instante justamente posterior. Para ello, necesitaremos declarar las siguientes variables en nuestro programa:

```
#include <iostream>
#include <ctime>
#include <cstdlib>
#include <climits>
#include <cassert>
#include <chrono>
#include <fstream>
#include <utility>

using namespace std;
using namespace std;

int maximo, minimo;

int max(int max1, int max2){
  int aux=0;

  if(max1>max2)
```

```
    aux=max1;
  else
    aux=max2;
  return aux;
}

int min(int min1, int min2){
  int aux=0;

  if(min1<min2)
    aux=min1;
  else
    aux=min2;
  return aux;
}

int* MaximoMinimoDyV(int *A, int Cini, int Cfin){

    int *v1,*v2, *vf;
    v1=new int[2];  // v1[max1,min1]
    v2=new int[2];  // v2[max2,min2]
    vf=new int[2];  // vf[maximo, minimo]

    if (Cini<Cfin-1){
        int mitad = (Cini-Cfin)/2;
        v1=MaximoMinimoDyV(A,Cini,mitad);
        v2=MaximoMinimoDyV(A,mitad+1,Cfin-1);
        maximo=max(v1[0],v2[0]); //devuelve max de los 2
        minimo=min(v1[1],v2[1]); //devuelve min de los 2
    }else if (Cini==Cfin){
        maximo = minimo = A[Cini];
    }else{
        maximo=max(A[Cini],A[Cfin]);
        minimo=min(A[Cini],A[Cfin]);
    }
    vf[0]=maximo;
    vf[1]=minimo;

    return vf;
}
```

```
int main(int argc, char *argv[]) {

  int *v;
        int n, i, argumento;
  chrono::time_point<std::chrono::high_resolution_clock> t0, tf; // Para
medir el tiempo de ejecución
        double tejecucion; // tiempo de ejecucion del algoritmo en ms
        unsigned long int semilla;
        ofstream fsalida;

        if (argc <= 3) {
                cerr<<"\nError: El programa se debe ejecutar de la
siguiente forma.\n\n";
                cerr<<argv[0]<<" NombreFicheroSalida Semilla tamCaso1
tamCaso2 ... tamCasoN\n\n";
                return 0;
        }

        // Abrimos fichero de salida
        fsalida.open(argv[1]);
        if (!fsalida.is_open()) {
                cerr<<"Error: No se pudo abrir fichero para escritura
"<<argv[1]<<"\n\n";
                return 0;
        }

        // Inicializamos generador de no. aleatorios
        semilla= atoi(argv[2]);
        srand(semilla);

        // Pasamos por cada tamaÒo de caso
        for (argumento= 3; argumento<argc; argumento++) {

                // Cogemos el tamanio del caso
                n= atoi(argv[argumento]);

                v= new int[n];
                // Reservamos memoria para el vector
```

```
		// Generamos vector aleatorio de prueba, con componentes
entre 0 y n-1
		for (i= 0; i<n; i++)
			v[i]= rand()%n;

		cerr << "Ejecutando MaximoMinimoDyV para tam. caso: "
<< n << endl;

		t0= std::chrono::high_resolution_clock::now(); // Cogemos
el tiempo en que comienza la ejecuciÛn del algoritmo
		MaximoMinimoDyV(v, 0, n); // Ejecutamos el algoritmo para
tamaÒo de caso n
		tf= std::chrono::high_resolution_clock::now(); // Cogemos el
tiempo en que finaliza la ejecuciÛn del algoritmo

		unsigned long tejecucion=
std::chrono::duration_cast<std::chrono::microseconds>(tf - t0).count();

		cerr << "\tTiempo de ejec. (us): " << tejecucion << " para
tam. caso "<< n << " max= "<<maximo<<" min= "<< minimo<<endl;

		// Guardamos tam. de caso y t_ejecucion a fichero de
salida
		fsalida<<n<<" "<<tejecucion<<"\n";


		// Liberamos memoria del vector
		delete [] v;
	}

	// Cerramos fichero de salida
	fsalida.close();

	return 0;
}
```

Tablas de tiempos:

Teo:

Tamaño del caso (n)	Tiempo de ejecución (en µs)
1000	187
2000	270
3000	380
4000	487
5000	646
6000	708
7000	842
8000	951
9000	1043
10000	1157

Miguel:

Tamaño del caso (n)	Tiempo de ejecución (en µs)
1000	816
2000	788
3000	1151
4000	1531
5000	1877
6000	2250
7000	2622
8000	3000
9000	3374

10000	3760

Mario:

Tamaño del caso (n)	Tiempo de ejecución (en µs)
1000	344
2000	537
3000	794
4000	1066
5000	976
6000	1106
7000	1221
8000	1438
9000	1685
10000	1909

2.1.3. Eficiencia Híbrida

En este apartado comprobaremos cómo la eficiencia práctica medida en el apartado 2.1.2. se corresponde con la eficiencia teórica calculada en el apartado 2.1.1. Partiremos de la definición de orden de eficiencia, en la que se indica que, cuando el tamaño del caso es suficientemente grande (tiende a infinito), entonces el tiempo de ejecución del algoritmo T(n) está acotado por la función del orden de eficiencia O(f(n)), de modo que T(n)<= K·f(n), con K una constante real positiva. A "K" se le denomina constante oculta.

Atendiendo al ya calculado orden O(f(n)) del algoritmo anteriormente. Este orden O(f(n)) quiere decir que existe una constante K, para cada algoritmo, tal que el tiempo T(n) de ejecución del mismo para un tamaño de caso n es:

$$T(n) <= K*f(n)$$

El cálculo de la constante K se calcula despejando e igualando la fórmula anterior:

$$K= T(n)/f(n)$$

Este valor de K se calculará para todas las ejecuciones del mismo algoritmo para distintos tamaños de caso, produciendo valores aproximados para K. Aproximamos el valor final de K como la media de todos estos valores. La siguiente gráfica muestra el cálculo de este valor en LibreOffice Calc, para los resultados del algoritmo de búsqueda del máximo y mínimo en un vector de tamaño ‘n’, anteriormente expuesto, donde f(n)=n. El valor final de K será el valor promedio obtenido en todas las ejecuciones.

Cálculo de la constante oculta mínima y de los tiempos de ejecución teóricos aproximados:

Teo:

Tam. Caso	Tiempo (us)	K=Tiempo/f(n)	Tiempo teórico estimado= K*f(n)
1000	187	0,187	128,836626984127
2000	270	0,135	257,673253968254
3000	380	0,126666666666667	386,509880952381
4000	487	0,12175	515,346507936508
5000	646	0,1292	644,183134920635
6000	708	0,118	773,019761904762
7000	842	0,120285714285714	901,856388888889
8000	951	0,118875	1030,69301587302
9000	1043	0,115888888888889	1159,52964285714
10000	1157	0,1157	1288,36626984127

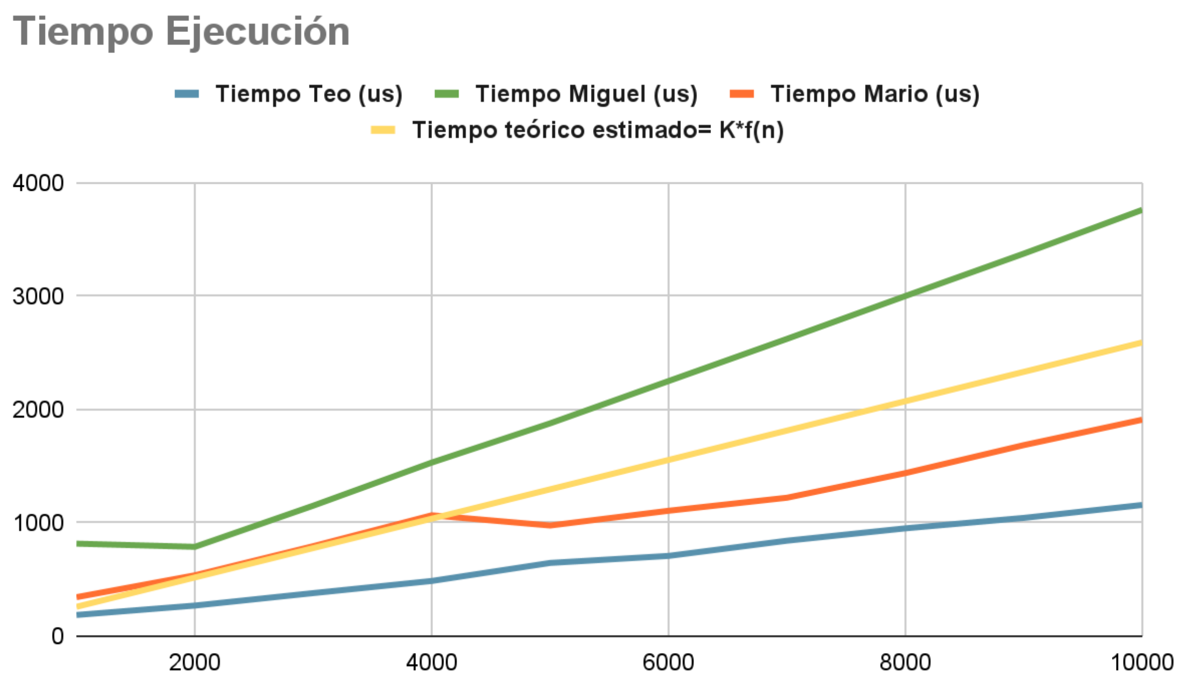
Miguel:

Tam. Caso	Tiempo (us)	K=Tiempo/f(n)	Tiempo teórico estimado= K*f(n)
1000	816	0,816	422,727698412698
2000	788	0,394	845,455396825397
3000	1151	0,383666666666667	1268,1830952381
4000	1531	0,38275	1690,91079365079
5000	1877	0,3754	2113,63849206349
6000	2250	0,375	2536,36619047619
7000	2622	0,374571428571429	2959,09388888889
8000	3000	0,375	3381,82158730159
9000	3374	0,374888888888889	3804,54928571429
10000	3760	0,376	4227,27698412698

Mario:

Tam. Caso	Tiempo (us)	K=Tiempo/f(n)	Tiempo teórico estimado= K*f(n)
1000	344	0,344	225,550079365079
2000	537	0,2685	451,100158730159
3000	794	0,264666666666667	676,650238095238
4000	1066	0,2665	902,200317460317
5000	976	0,1952	1127,7503968254
6000	1106	0,184333333333333	1353,30047619048
7000	1221	0,174428571428571	1578,85055555556
8000	1438	0,17975	1804,40063492064
9000	1685	0,187222222222222	2029,95071428571
10000	1909	0,1909	2255,50079365079

Gráfica:



2.2. Algoritmo 2: HeapSort

2.2.1. Eficiencia Teórica

Para el paso de pila, estamos examinando cada elemento en el árbol y moviéndolo hacia abajo hasta que sea más grande que sus hijos. Dado que la altura de nuestro árbol es O(lg(n)), podríamos hacer hasta O(lg(n)) movimientos. En todos los n nodos, esa es una complejidad de tiempo general de O(nlog(n)).

Después de transformar el árbol en un montón, eliminamos todos los elementos nn de él, un elemento a la vez. Eliminar de un montón lleva O(log(n)) tiempo, ya que tenemos que mover un nuevo valor a la raíz del montón y burbujear hacia abajo. Hacer n eliminar operaciones será O(nlog(n)) tiempo.

Un análisis más completo muestra que hacer n eliminaciones sigue siendo O(nlog(n)).
Poniendo estos pasos juntos, estamos en el tiempo O(nlog(n)) en el peor de los casos (y en promedio).
Cada vez que eliminemos un elemento de la raíz del árbol, el elemento que lo reemplace no tendrá que desaparecer en absoluto. En ese caso, cada eliminación requiere un tiempo O(1).
Dado que inteligentemente reutilizamos el espacio disponible al final de la matriz de entrada para almacenar el elemento que eliminamos, solo necesitamos espacio O(1) en general para heapsort.

2.2.2. Eficiencia Práctica

El tiempo de ejecución lo mediremos como la diferencia entre el instante de tiempo justo anterior al inicio del algoritmo, y el instante justamente posterior. Para ello, necesitaremos declarar las siguientes variables en nuestro programa:

```
#include <iostream>
#include <ctime>
#include <cstdlib>
#include <climits>
#include <cassert>
#include <chrono>
#include <fstream>
#include <utility>

using namespace std;
using namespace std;

void insertarEnPos(double *apo, int pos){
	int idx = pos -1;
	int padre;
	if (idx > 0){
		if(idx%2==0){
			padre=( idx-2)/2;
		}else{
			padre=(idx-1)/2;
```

```
                }

                if(apo[padre] > apo[idx]){
                        double tmp= apo[idx];
                        apo[idx]= apo[padre];
                        apo[padre]=tmp;
                        insertarEnPos(apo, padre+1);
                }
        }
}

void reestructurarRaiz(double *apo, int pos, int tamapo){
        int minhijo;
        if (2*pos+1< tamapo) {
           minhijo=2*pos+1;
           if ((minhijo+1< tamapo) && (apo[minhijo]>apo[minhijo+1]))
minhijo++;
           if (apo[pos]>apo[minhijo]) {
                                double tmp = apo[pos];
                                apo[pos]=apo[minhijo];
                                apo[minhijo]=tmp;
                                reestructurarRaiz(apo, minhijo, tamapo);
                        }
        }
}

void HeapSort(int *v, int n){

        double *apo=new double [n];
        int tamapo=0;

        for (int i=0; i<n; i++){
                apo[ tamapo]= v[ i ];
                tamapo++;
                insertarEnPos(apo , tamapo );
}
        for (int i=0; i<n; i++){
                v[ i ] =apo[ 0 ];
                tamapo--;
                apo[0]=apo[ tamapo ];
```

```
                reestructurarRaiz(apo, 0, tamapo);
    }
                delete [ ] apo;
    }

    int main(int argc, char *argv[]) {

            int *v;
      int *vaux;
            int n, i, argumento;
      chrono::time_point<std::chrono::high_resolution_clock> t0, tf; // Para
    medir el tiempo de ejecución
            double tejecucion; // tiempo de ejecucion del algoritmo en ms
            unsigned long int semilla;
            ofstream fsalida;

            if (argc <= 3) {
                cerr<<"\nError: El programa se debe ejecutar de la
    siguiente forma.\n\n";
                cerr<<argv[0]<<" NombreFicheroSalida Semilla tamCaso1
    tamCaso2 ... tamCasoN\n\n";
                return 0;
            }

            // Abrimos fichero de salida
            fsalida.open(argv[1]);
            if (!fsalida.is_open()) {
                cerr<<"Error: No se pudo abrir fichero para escritura
    "<<argv[1]<<"\n\n";
                return 0;
            }

            // Inicializamos generador de no. aleatorios
            semilla= atoi(argv[2]);
            srand(semilla);

            // Pasamos por cada tamaÒo de caso
            for (argumento= 3; argumento<argc; argumento++) {

                // Cogemos el tamanio del caso
```

```
                n= atoi(argv[argumento]);

                // Reservamos memoria para el vector
                v= new int[n];
        vaux= new int[n];

                // Generamos vector aleatorio de prueba, con componentes
entre 0 y n-1
                for (i= 0; i<n; i++)
                        v[i]= rand()%n;

                cerr << "Ejecutando HeapSort para tam. caso: " << n <<
endl;

                t0= std::chrono::high_resolution_clock::now(); // Cogemos
el tiempo en que comienza la ejecuciÛn del algoritmo
                HeapSort(v, n-1); // Ejecutamos el algoritmo para tamaÒo
de caso n
                tf= std::chrono::high_resolution_clock::now(); // Cogemos el
tiempo en que finaliza la ejecuciÛn del algoritmo

                unsigned long tejecucion=
std::chrono::duration_cast<std::chrono::microseconds>(tf - t0).count();

                cerr << "\tTiempo de ejec. (us): " << tejecucion << " para
tam. caso "<< n<<endl;

                // Guardamos tam. de caso y t_ejecucion a fichero de
salida
                fsalida<<n<<" "<<tejecucion<<"\n";


                // Liberamos memoria del vector
                delete [] v;
        delete [] vaux;
        }

        // Cerramos fichero de salida
        fsalida.close();
```

```
    return 0;
}
```

Tablas de tiempos:

Teo:

Tamaño del caso (n)	Tiempo de ejecución (en µs)
10000	1834
20000	3597
30000	5284
40000	7728
50000	9524
60000	11630
70000	14429
80000	16422
90000	18924
100000	20693

Miguel:

Tamaño del caso (n)	Tiempo de ejecución (en µs)
10000	4663
20000	9813
30000	15423
40000	21113
50000	27071
60000	33491

70000	39960
80000	45601
90000	51691
100000	60738

Mario:

Tamaño del caso (n)	Tiempo de ejecución (en µs)
10000	2316
20000	5437
30000	9255
40000	10178
50000	12337
60000	14822
70000	16949
80000	20208
90000	22714
100000	25038

2.2.3. Eficiencia Híbrida

En este apartado comprobaremos cómo la eficiencia práctica medida en el apartado 2.1.2. se corresponde con la eficiencia teórica calculada en el apartado 2.1.1. Partiremos de la definición de orden de eficiencia, en la que se indica que, cuando el tamaño del caso es suficientemente grande (tiende a infinito), entonces el tiempo de ejecución del algoritmo

T(n) está acotado por la función del orden de eficiencia O(f(n)), de modo que T(n)<= K·f(n), con K una constante real positiva. A "K" se le denomina constante oculta.

Atendiendo al ya calculado orden O(f(n)) del algoritmo anteriormente. Este orden O(f(n)) quiere decir que existe una constante K, para cada algoritmo, tal que el tiempo T(n) de ejecución del mismo para un tamaño de caso n es:

$$T(n) \le K * f(n)$$

El cálculo de la constante K se calcula despejando e igualando la fórmula anterior:

$$K = T(n)/f(n)$$

Este valor de K se calculará para todas las ejecuciones del mismo algoritmo para distintos tamaños de caso, produciendo valores aproximados para K. Aproximamos el valor final de K como la media de todos estos valores. La siguiente gráfica muestra el cálculo de este valor en LibreOffice Calc, para los resultados del algoritmo de búsqueda del máximo y mínimo en un vector de tamaño ‘n’, anteriormente expuesto, donde f(n)=n*log(n). El valor final de K será el valor promedio obtenido en todas las ejecuciones.

Cálculo de la constante oculta mínima y de los tiempos de ejecución teóricos aproximados:

Teo:

Tam. Caso	Tiempo (us)	K=Tiempo/f(n)	Tiempo teórico estimado= K*f(n)
10000	1834	0,04585	1673,31436229933
20000	3597	0,041815565150979	3598,48763221237
30000	5284	0,039340755658038	5618,72347295849
40000	7728	0,041981199802711	7700,69307965219
50000	9524	0,040536543077362	9828,56749533648
60000	11630	0,040566596404367	11993,0236687582
70000	14429	0,042543735900949	14187,9012399323
80000	16422	0,041866455754351	16408,8217897593
90000	18924	0,042441738826076	18652,511577057

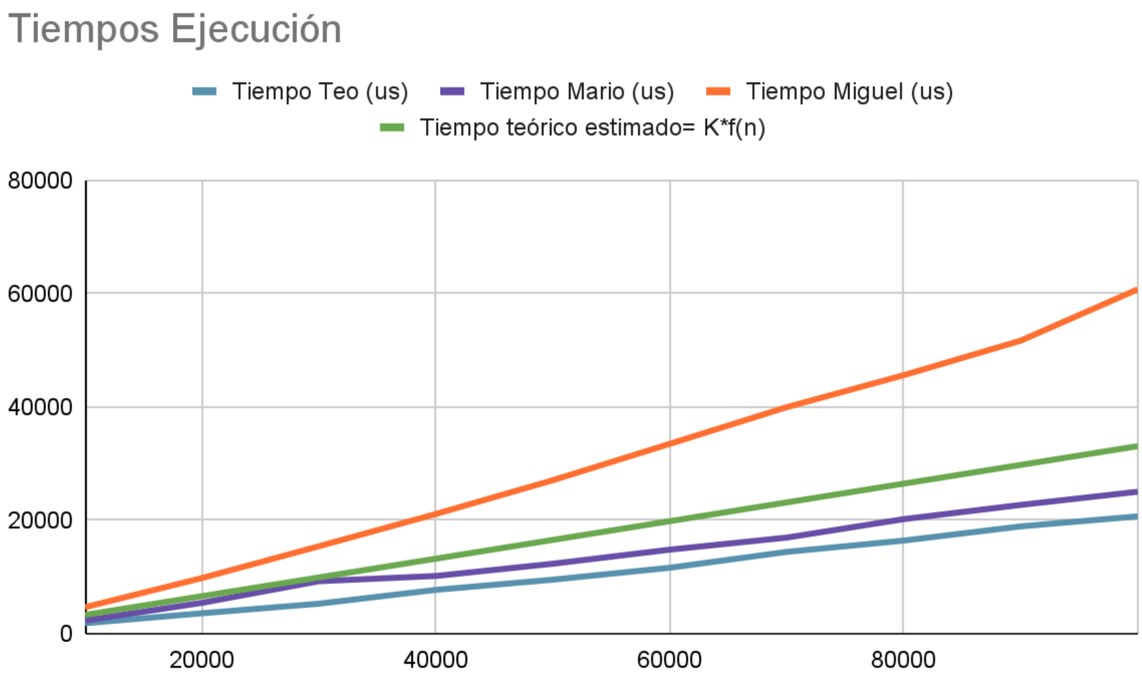
100000	20693	0,041386	20916,4295287416

Miguel:

Tam. Caso	Tiempo (us)	K=Tiempo/f(n)	Tiempo teórico estimado= K*f(n)
10000	4663	0,116575	4654,79155888187
20000	9813	0,114077325778858	10010,1990591572
30000	15423	0,114828250286509	15630,0496684198
40000	21113	0,114693202825392	21421,6300011014
50000	27071	0,115220995133061	27340,907389528
60000	33491	0,116819938106504	33361,94716102
70000	39960	0,117821587539117	39467,6125525736
80000	45601	0,11625576962941	45645,7237677767
90000	51691	0,115929820421617	51887,1739805818
100000	60738	0,121476	58184,8944860234

Mario:

Tam. Caso	Tiempo (us)	K=Tiempo/f(n)	Tiempo teórico estimado= K*f(n)
1000	2316	0,772	2115,34209416202
2000	5437	0,823530838426444	4655,20513594633
3000	9255	0,887228190795067	7355,30095661396
4000	10178	0,706401338713386	10159,4521671372
5000	12337	0,667050556535375	13040,9782586911
6000	14822	0,653847125120385	15984,1647560948
7000	16949	0,629707146368818	18978,6280605181
8000	20208	0,647179544519483	22016,9881247636
9000	22714	0,638245573394442	25093,7268922256
10000	25038	0,62595	28204,5612554936



Comparación entre Burbuja, MergeSort y HeapSort:

Atendiendo a los órdenes de eficiencia teórica de cada uno, los cuales son $f(n)=n^2$ para burbuja, f(n)=n*log(n) para MergeSort y f(n)=n*log(n) para HeapSort; tras un primer análisis se puede observar que el algoritmo de burbuja ni se va a considerar en las comparaciones debido a su ineficiencia en comparación a los otros dos dados.

Ahora de entre el MergeSort o el HeapSort, por el que fallamos a favor del algoritmo más eficiente es el HeapSort, ya que el HeapSort en todos sus casos tiene una eficiencia constante de O(n*log(n)), mientras que el MergeSort en los casos normales tiene O(n*log(n)), sin embargo, su peor caso es O(n) y O(n)<O(n*log(n)).