PRÁCTICA 1

Cálculo de la eficiencia de los algoritmos

Algorítmica

Carlos Mata Carrillo
Buenaventura Porcel Esquivel
Lydia Vílchez López

UNIVERSIDAD
DE GRANADA

ÍNDICE

I. Introducción 2

II Desarrollo 2

1. Algoritmo 1: MaximoMinimoDyV 3
   - 1.1. Eficiencia teórica 3
   - 1.2. Eficiencia práctica 6
   - 1.3. Eficiencia híbrida 7
2. Algoritmo 2 9
   - 2.1. insertarEnPos 9
     - 2.1.1. Eficiencia teórica 9
     - 2.1.2. Eficiencia práctica 11
     - 2.1.3. Eficiencia híbrida 12
   - 2.2. reestructurarRaiz 14
     - 2.2.1. Eficiencia teórica 14
     - 2.2.2. Eficiencia práctica 15
     - 2.2.3. Eficiencia híbrida 16
3. Algoritmo 3: HeapSort 18
   - 3.1. Eficiencia teórica 18
   - 3.2. Eficiencia práctica 19
   - 3.3. Eficiencia híbrida 20

III Conclusiones 22

I. Introducción

A lo largo de esta primera práctica, vamos a llevar a cabo un análisis de la eficiencia teórica, práctica e híbrida de los algoritmos propuestos. Una vez hecho esto, basándonos en los datos obtenidos, compararemos las eficiencias de los algoritmos burbuja, mergesort y heapsort.

En cuanto al análisis de eficiencia que vamos a llevar a cabo, constará de 3 partes:

- **Análisis de eficiencia teórica**

Medir la eficiencia teórica implica analizar el tiempo de ejecución del algoritmo en el peor de los casos, para determinar qué clase de función es en notación O grande. Para realizar este estudio, aplicaremos los métodos y técnicas aprendidas en clase.

- **Análisis de eficiencia práctica**

Para medir la eficiencia práctica de los algoritmos, tenemos que ejecutar el mismo algoritmo para diferentes tamaños de caso, y calcular su tiempo de ejecución. Esto lo llevaremos a cabo usando la librería **chrono**, que nos permitirá hacer mediciones con una precisión de hasta nanosegundos.

- **Análisis de eficiencia híbrida**

Debemos comprobar si la eficiencia teórica se corresponde con la eficiencia práctica. Partiendo de la definición de orden de eficiencia, el tiempo de ejecución del algoritmo $T(n)$ está acotado por la función de orden de eficiencia $O(f(n))$, de modo que $T(n) <= K \cdot f(n)$, con K una constante real positiva. A "K" se le denomina constante oculta. A la hora de los análisis, hemos tenido en cuenta que el valor de la "K" variará dependiendo del ordenador en el que se ejecute el código, por lo que hemos decidido ejecutarlo todo en el mismo computador con el objetivo de que los resultados sean lo más precisos posible.

II. Desarrollo

1. Algoritmo 1 (MaximoMinimoDyV)

1.1 Eficiencia teórica

```
79 pair<int, int> MaximoMinimoDyV(int *v, int Cini, int Cfin){
80
81         int mitad;
82         pair<int, int> pareja1, pareja2, salida;
83         if (Cini < Cfin-1){
84                 mitad = (Cini+Cfin)/2;
85                 pareja1 = MaximoMinimoDyV(v, Cini, mitad);
86                 pareja2 = MaximoMinimoDyV(v, mitad+1 , Cfin);
87
88                 //comparo para ver cual es el mayor y el menor
89                 //comparo el mayor
90                 if (pareja1.first <= pareja2.first) salida.first = pareja2.first;
91                 else salida.first = pareja1.first;
92
93                 //comparo el menor
94                 if (pareja1.second >= pareja2.second) salida.second = pareja2.second;
95                 else salida.second = pareja1.second;
96
97         }
98         else if (Cini = Cfin){
99
100                salida.first=v[Cini];
101                salida.second=v[Cini];
102        }
103        else{
104
105                if (v[Cini] <= v[Cfin]){
106                        salida.first = v[Cfin];
107                        salida.second = v[Cini];
108                }
109                else {
110                        salida.second = v[Cfin];
111                        salida.first = v[Cini];
112                }
113
114
115        }
116
117        return salida;
118
119 }
120
```

Figura 1: *Código algoritmo* 1

El primer algoritmo propuesto resuelve un problema de encontrar el valor máximo y el mínimo dado un vector de n componentes y dado un rango acotado por "$Cini$" y "$Cfin$" .
El tamaño del problema depende del número de componentes que tiene el vector así como sus cotas.

Comenzamos analizando la función desde la parte más interna para ir poco a poco hacia fuera. Las líneas 106, 107 y 110, 111 contienen accesos a arrays y asignaciones lo cual son operaciones elementales y por tanto su eficiencia es $O(1)$. Así mismo, las sentencias condicionales, tanto "if" como "else", como ya hemos demostrado antes, su contenido es de eficiencia $O(1)$ ya que se tratan de operaciones elementales. A mitad de código (líneas

98-102), en la sentencia “else if”, nos volvemos a encontrar dentro del bloque accesos a arrays y asignaciones, es decir operaciones elementales, tal y como hemos descrito en el caso anterior, por tanto su eficiencia vuelve a ser $O(1)$.

Pasamos a analizar el primer bloque “if”, líneas 83-97. Lo dividiremos en partes para tenerlo más claro. Analizamos los dos bloques condicionales, líneas 90, 91 y 94, 95. Ambas tienen la misma eficiencia. Consisten en una comprobación cuya eficiencia es $O(1)$ y tanto si se cumple como si no lo hiciese la eficiencia será la misma ya que únicamente aparecen asignaciones en el código, es decir operaciones elementales de nuevo y por tanto eficiencia $O(1)$, aplicando la regla de máximo.

Llegamos a la parte interesante de analizar en la función, las líneas 85 y 86. Como podemos observar, se trata de un algoritmo recursivo ya que hace una llamada a la propia función dentro de ella misma. Por tanto vamos a decir que $T(n)$ es el tiempo de ejecución que tarda el algoritmo “MaximoMinimoDyV” en resolver un problema en el que el número de componentes del vector es “n”. Como todo algoritmo recursivo pueden existir dos casos, el general y el base:

- Caso base: Cuando $"Cini" >= "Cfin" - 1$. En este caso no entrará en el primer bloque condicional y pasará a ejecutarse uno de los bloques que hemos analizado antes y por tanto su eficiencia será de $O(1)$, tal y como hemos demostrado.
- Caso general. Este caso se dará cuando $"Cini" >= "Cfin" - 1$. Ya hemos analizado las sentencias no recursivas del caso general por tanto vamos a ver que sucede en las sentencias donde la función se llama a sí misma:
    - En la línea 86 nos encontramos una llamada recursiva en la que se pasa por parámetros el vector y una cota que va desde la mitad del vector hasta su posición final. Es decir, un subvector de tamaño $n/2$. Por tanto si hemos afirmado que el algoritmo completo tarda un tiempo de $T(n)$ con un un vector de n componentes, podemos afirmar que el tiempo de ejecución de esta llamada sería la mitad, $T(n/2)$.
    - De la misma manera en la línea 85 nos encontramos otra llamada recursiva. Es igual que el caso anterior, lo único que cambia esta vez es la cota. Ahora va desde el inicio del vector hasta la mitad. De nuevo, el mismo algoritmo pero con la mitad de componentes. Por tanto el tiempo de ejecución sería $T(n/2)$.

Por otro lado, la línea 84, consta de una simple suma y división y una asignación. Todas ellas operaciones elementales y por tanto con eficiencia $O(1)$. La sentencia condicional del if es una comprobación, que, tal y como ya hemos dicho es $O(1)$ y finalmente fuera de este (al principio del algoritmo), nos encontramos con dos declaraciones que son operaciones elementales y cuya eficiencia vuelve a ser $O(1)$.

Por tanto podemos aproximar el tiempo de ejecución del algoritmo ($T(n)$) en el caso general como $T(n) = 2T(n/2)$. Ya que los otros casos al ser $O(1)$ nos son indiferentes.

Para obtener la ecuación característica necesitamos realizar un cambio de variable de forma que $n = 2^k$, por tanto la ecuación se nos queda como:
$T(2^k) = 2T(2^{k-1})$

Llevando todos los elementos a la izquierda, nos queda una ecuación lineal homogénea:
$T(2^k) - 2T(2^{k-1}) = 0$
Podemos resolverlo de forma más sencilla si la escribimos como:
$x^k - 2x^{k-1} = 0$
$x^{k-1}(x - 2) = 0$

El polinomio característico queda como:
$P(x) = (x - 2)$
De esta forma tenemos una única raíz, con valor 2 y cuya multiplicidad es 1. Aplicando la fórmula de la ecuación característica, tenemos que el tiempo de ejecución se expresa como:

$T(2^k) = c_1 2^k$

Ahora tenemos que deshacer el cambio de variable para volver al espacio de tiempos inicial:

$2^k = n$
$T(n) = c_1 n$

Por tanto, de la ecuación anterior obtenemos que el orden de la eficiencia del algoritmo en el peor de los casos es $O(n)$.

1.2 Eficiencia práctica

Para realizar el análisis de la eficiencia práctica de este primer algoritmo, hemos decidido ejecutarlo con diferentes tamaños de caso, con el objetivo de hacer el análisis lo más riguroso y libre de errores posible. Estos han sido los resultados obtenidos:

Tamaño del caso (n)	Tiempo de ejecución (en us)
10000	529
100000	2534
500000	9768
1000000	18324
1500000	37536
2000000	39594
2500000	53500
3000000	68281
3500000	72325
4000000	71924
5000000	97332
6000000	129885
7000000	139527
8000000	140687
9000000	154855

Figura 2: Tamaño de casos y tiempos de ejecución algoritmo 1

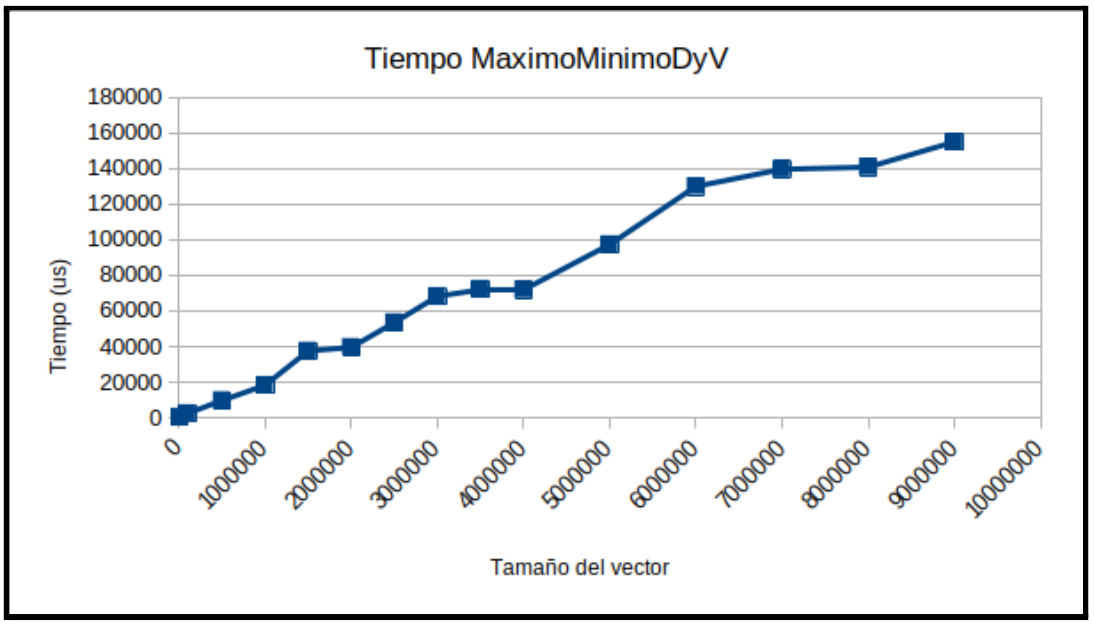

Figura 3: Gráfica tiempo de ejecución práctico algoritmo 1

1.3 Eficiencia híbrida

En el primer apartado hemos calculado el orden $O(f(n))$ de este primer algoritmo. Este orden quiere decir que existe una constante K, tal que el tiempo de ejecución del mismo para un caso de tamaño n es:

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

Por tanto, el cálculo de dicha constante K se calcula despejando de la fórmula anterior:

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

Una vez obtenidos los valores de K para cada uno de los tamaños utilizados, aproximaremos el valor final de dicha constante a partir de la media de todos los valores obtenidos.

Tam. caso	Tiempo (us)	K=Tiempo/f(n)	Tiempo teórico estimado = K*f(n)
10000	529	0.0529	226.376622486772
100000	2534	0.02534	2263.766225
500000	9768	0.019536	11318.83112
1000000	18324	0.018324	22637.66225
1500000	37536	0.025024	33956.49337
2000000	39594	0.019797	45275.3245
2500000	53500	0.0214	56594.15562
3000000	68281	0.02276033333	67912.98675
3500000	72325	0.02066428571	79231.81787
4000000	71924	0.017981	90550.64899
5000000	97332	0.0194664	113188.3112
6000000	129885	0.0216475	135825.9735
7000000	139527	0.01993242857	158463.6357
8000000	140687	0.017585875	181101.298
9000000	154855	0.01720611111	203738.9602
K promedio:		**0.02263766225**	

Figura 4: Cálculo de la constante oculta mínima y tiempos de ejecución algoritmo 1

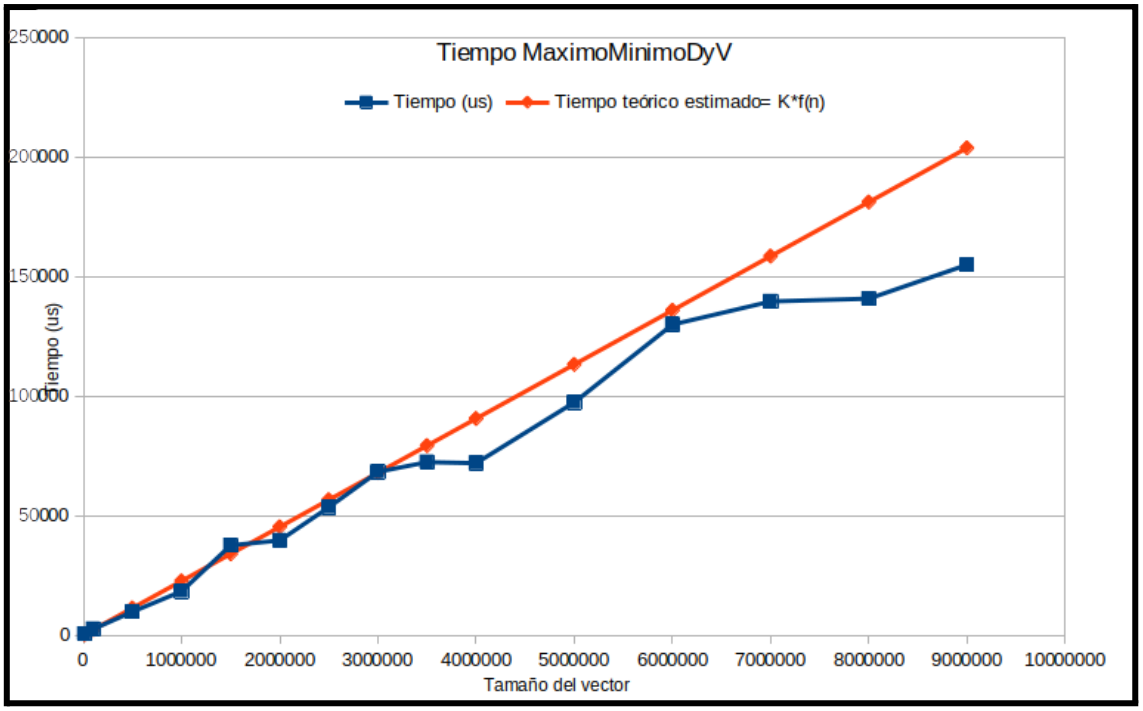

Figura 5: *Comparación tiempo de ejecución real vs. tiempo de ejecución teórico algoritmo* 1

Como podemos observar en esta gráfica, el tiempo teórico calculado a partir de la constante oculta y la eficiencia teórica $(O(n))$, y el tiempo práctico obtenido con las ejecuciones del algoritmo, son realmente parecidas, obviando las pequeñas diferencias que podemos observar debido a que el programa sobre el que ejecutamos el algoritmo no es el único que estamos ejecutando en nuestro ordenador en ese instante de tiempo, lo que deriva en que no todos los recursos del ordenador estén a disposición de la ejecución y por tanto, los valores teóricos no correspondan completamente con los prácticos. Independientemente de este apunte, se observa claramente la tendencia lineal de ambas funciones.

2. Algoritmo 2

2.1. insertarEnPos

2.1.1. Eficiencia teórica

```
78 void insertarEnPos(double *apo, int pos){
79
80     int idx = pos -1;
81     int padre;
82
83     if (idx > 0) {
84
85         if (idx%2 == 0) {
86             padre = (idx - 2)/2;
87         }else{
88             padre = (idx -1)/2;
89         }
90
91         if (apo[padre] > apo[idx]) {
92             double tmp = apo [idx];
93             apo[idx] = apo[padre];
94             apo[padre] = tmp;
95             insertarEnPos(apo, padre + 1);
96         }
97     }
98 }
99
```

Figura 6: *Código algoritmo* 2. 1

Este algoritmo 2.1 tiene la función de permutar las posiciones de un apo, dada la posición de un nodo "pos", almacenado en un vector de forma que se respete que el nodo padre sea menor que sus dos hijos. En este caso el tamaño del problema dependerá del número de nodos que tenga el árbol, variable a la que denominaremos n.

Al ser un algoritmo recursivo, llamaremos $T(n)$ al tiempo de ejecución que tarda el algoritmo en reordenar un apo de n nodos, por tanto, un caso de tamaño n. Basándonos en el algoritmo en cuestión, podemos distinguir entre:

- Caso base: Cuando n<=1 no entrará en el if de la línea 86, y por tanto la ejecución terminará. Siguiendo las reglas de análisis de eficiencia, en este caso base, el tiempo de ejecución sería $O(1)$.

- Caso general: Este caso se dará cuando n>1. Si se da esta condición:
    - Lo primero que se hace es calcular la posición del padre, proceso que llevaremos a cabo de forma diferente dependiendo de si la posición que se nos ha pasado por parámetro - 1, es par o impar. Esta condición, compuesta por un if y un else, viene reflejada entre las líneas 85 y 89 del código. Si analizamos la eficiencia, nos damos cuenta que realmente no hay más que asignaciones y operaciones elementales de cálculo, y como ya sabemos, estas tienen eficiencia O(1).
    - A continuación, en la línea 91, nos encontramos con un if, que compara si el valor de la posición padre en en vector apo es mayor que el valor de la posición idx, que como se puede observar en el código es el valor del parámetro pos - 1. Al tener simples accesos a arrays, la eficiencia será también de O(1).
    - Una vez dentro del if, en las líneas 92, 93 y 94 encontramos accesos al vector apo y asignaciones, todas ellas operaciones O(1).
    - Sin embargo, a diferencia de las sentencias que nos hemos encontrado hasta ahora, en la línea 95 aparece una llamada recursiva a InsertarEnPos. En esta ocasión se le pasan por parámetros el apo y la posición padre+1, por lo que el problema se reduce a la mitad de tamaño ya que hemos descartado todos los nodos que no pertenecen a la rama del nodo padre+1. Por tanto, si el problema tenía un tiempo inicial de T(n), ahora tendremos un tiempo de T(n/2).

Una vez hecho este análisis podemos aproximar el tiempo con la siguiente ecuación $T(n) = T(n/2) + c$, donde "c" es una constante de todas las operaciones elementales que hay en el algoritmo.

De la misma forma que hicimos en el análisis del primer algoritmo, realizamos un cambio de variable donde $n = 2^k$. Haciendo esto la ecuación queda de la siguiente forma:

$$T(2^k) = T(2^{k-1}) + c$$

Se trata de una ecuación lineal no homogénea donde $b = 1\, y\, p = 1\ (polinomio\ de\ grado\ cero)$.

$$T(2^k) - T(2^{k-1}) = 1$$

Resolviendo primero la parte homogénea:

$$x^k - x^{k-1} = 0$$
$$x^{k-1}(x - 1) = 0$$

Para la parte homogénea, el polinomio característico es $P_H(x) = (x - 1)$

Como ya hemos determinado antes los coeficientes de la parte no homogénea y sabiendo que el polinomio característico nos quedaría de la siguiente forma: $P(x) = P_H(x)(x - b)^{d+1} = (x - 1)(x - 1) = (x - 1)^2$

De esta forma tenemos una única raíz, $r = 1$ pero de multiplicidad 2, es decir una raíz doble.

Ya podemos sacar el tiempo (primero en función de 2^k).

$$T(2^k) = c_1 1^k + c_2 k 1^k = c_1 + c_2 k$$

Ahora deshaciendo el cambio de variable, donde $k = log_2 n$

$$T(n) = c_1 + c_2 log_2 n$$

Por tanto, obtenemos que el orden de la eficiencia de la función en el peor de los casos es $O(log(n))$.

2.1.2. Eficiencia práctica

Una vez realizado el análisis de la eficiencia teórica, hemos analizado la eficiencia práctica de este algoritmo 2.1. Para ello hemos tomado diferentes tamaños de caso, y hemos medido los tiempos de ejecución para cada uno de ellos.

Tamaño del caso (n)	Tiempo de ejecución (en ns)
1000	297
10000	384
100000	559
500000	586
1000000	603
2500000	625
5000000	703
7500000	792
10000000	667
15000000	746
20000000	737
25000000	784
30000000	814

Figura 7: Tamaño de casos y tiempos de ejecución algoritmo 2.1

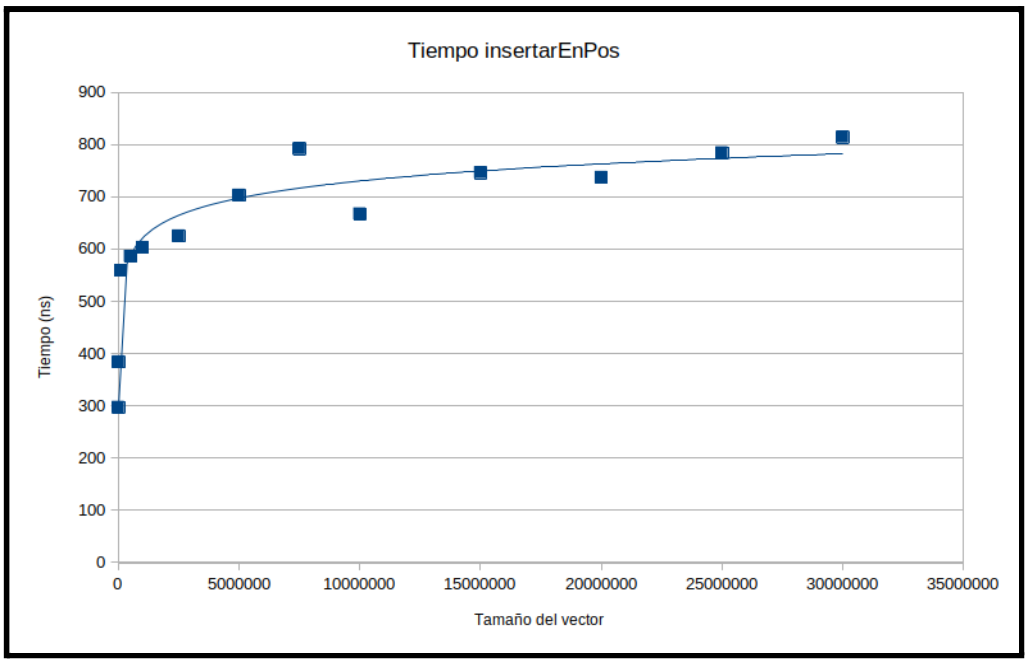

Figura 5: Gráfica tiempo ejecución práctico algoritmo 2.1

2.1.3. Eficiencia híbrida

Una vez que tenemos la eficiencia teórica y la práctica, podemos realizar un análisis híbrido de la eficiencia. Siguiendo el mismo procedimiento que en el apartado anterior, llegamos a los siguientes resultados:

Tam. caso	Tiempo (us)	K=Tiempo/f(n)	Tiempo teórico estimado = K*f(n)
1000	297	99	309,9187696
10000	384	96	413,2250262
100000	559	111,8	516,5312827
500000	586	102,8255982	588,7392573
1000000	603	100,5	619,8375393
2500000	625	97,68769309	660,9472319
5000000	703	104,9415059	692,0455139
7500000	792	115,1989735	710,2368426
10000000	667	95,28571429	723,1437958
15000000	746	103,9563145	741,3351246
20000000	737	100,9446613	754,2420778
25000000	784	105,9754471	764,2534884
30000000	814	108,8654273	772,4334066
K promedio:		103,3062565	

Figura 9: Cálculo de la constante oculta mínima y tiempos de ejecución algoritmo 2.1

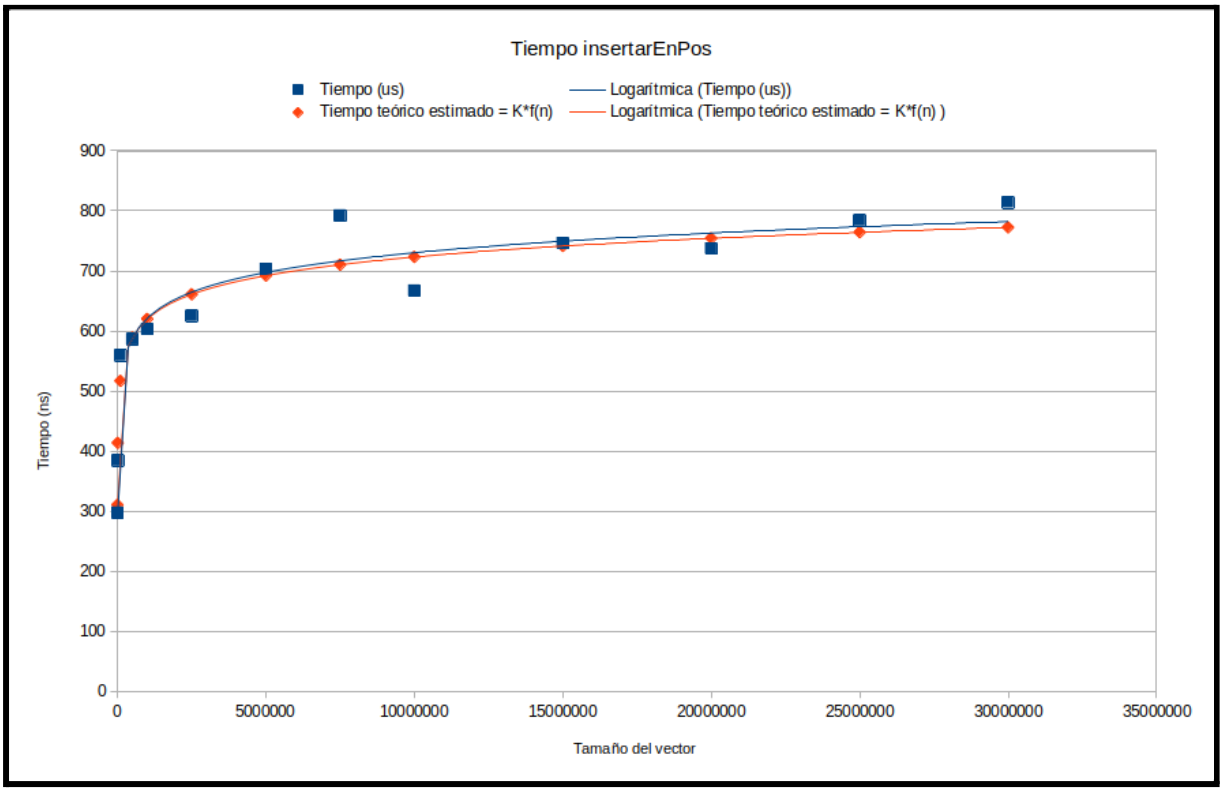

Figura 10: *Comparación tiempo de ejecución real vs. tiempo de ejecución teórico algoritmo* 2. 1

Se puede observar en la gráfica superior y en la tabla de la *figura* 9 como los valores de los tiempos teóricos y prácticos se asemejan bastante, siguiendo una clara tendencia logarítmica. Las pequeñas imprecisiones o diferencias que existen para algunos valores se deben principalmente al "ruido" que existe derivado de que la ejecución del algoritmo sobre el programa, no es la única que se está llevando a cabo en ese instante de tiempo en el ordenador.

2.2. reestructurarRaiz

2.2.1. Eficiencia teórica

```
81
82 void reestructurarRaiz(double *apo, int pos, int tamapo){
83
84     int minhijo;
85
86     if (2*pos+1 < tamapo) {
87         minhijo = 2*pos+1; //hijo izquierda de la posicion "pos"
88
89         if ((minhijo+1 < tamapo) && (apo[minhijo] > apo[minhijo+1])) minhijo++;
90
91         if (apo[pos] > apo[minhijo]) { //compruebas si el menor de los hijos es
92             double tmp = apo[pos]; //si el hijo es menor, haces el cambio
93             apo[pos] = apo[minhijo];
94             apo[minhijo] = tmp;
95             reestructurarRaiz(apo, minhijo, tamapo);
96         }
97     }
98 }
```

Figura 11: *Código algoritmo* 2.2

El algoritmo 2.1 consiste en la reestructuración de un nodo "pos" de un árbol parcialmente ordenado cuyo tamaño es "tamapo" pero que denominaremos "n". Este comprueba si existe algún hijo cuyo valor sea menor al del padre (padre es el nodo que se encuentra en la posición "pos"). En caso afirmativo, debido a la condición del APO, se realiza el cambio (en los árboles parcialmente ordenados el nodo padre tiene un valor menor al del hijo).

De nuevo, nos encontramos con un algoritmo recursivo. Procedemos de la misma manera que llevamos haciendo hasta ahora. $T(n)$ será el tiempo que tarda en ejecutarse para un APO con "n" nodos, es decir, un problema de tamaño "n". Calculamos el caso general y el caso base:

- Caso base: Cuando n<=1. No llegará a entrar en el "if" de la línea 86 y acabará. Su tiempo de ejecución es $O(1)$.
- Caso general: Puede darse la recursividad. Sucede cuando n>1 (ya que si el nodo no tiene ningún hijo no hace falta ordenarlo).
    - Lo primero que se hace es calcular el hijo izquierda del nodo en la posición "pos" y se guarda su posición en la variable "minhijo". Esto es un operación elemental cuya eficiencia es $O(1)$.
    - A continuación nos encontramos con una sentencia condicional (línea 89) con operaciones simples en la que se comprueba cuál de los dos hijos posee un valor menor (el hijo izquierda es "minhijo" y el hijo derecho es "minhijo + 1"). En caso de que se cumpla la condición la máxima operación que se realiza es una suma, por tanto la línea entera tiene una eficiencia de $O(1)$.
    - La siguiente sentencia que nos encontramos (línea 91) comprueba si el padre es mayor que el hijo, en caso afirmativo es necesario hacer el cambio (este sería el peor de los casos ya que si el padre fuese menor no tendríamos que

hacer reestructuración y la función acabaría con una eficiencia de $O(1)$). De nuevo, la sentencia condicional es $O(1)$.

- En las líneas 92-94 se llevan a cabo accesos a arrays y asignaciones y por tanto operaciones simples cuyo conjunto nos da una eficiencia de $O(1)$.
- Finalmente llegamos a la llamada recursiva. Los parámetros no cambian salvo la posición que le pasamos. La nueva posición es la del hijo menor (que ahora es padre). Como podemos comprobar el tamaño del problema se reduce a la mitad ya que hemos descartado todos los nodos que quedan con su otro hermano. Por tanto si el tiempo del problema inicial era $T(n)$, ahora tendremos un tiempo de $T(n/2)$.

Una vez hecho este análisis podemos aproximar el tiempo con la siguiente ecuación $T(n) = T(n/2) + c$, donde "c" es una constante de todas las operaciones elementales que hay en el algoritmo.

Como podemos comprobar tenemos la misma ecuación que en el algoritmo anterior. Por tanto ya podemos decir que el orden de la eficiencia en el peor de los casos es de $O(log(n))$.

2.2.2. Eficiencia práctica

Una vez calculada la eficiencia teórica, que hemos obtenido que vale $O(log(n))$, podemos pasar a analizar la eficiencia práctica. Para ello, usando los tamaños de caso indicados en la siguiente tabla, hemos obtenido los siguientes tiempos:

Tamaño del caso (n)	Tiempo de ejecución (en ns)
5000	390
20000	450
100000	536
500000	622
1500000	651
3000000	677
5000000	694
7000000	717
10000000	704
12500000	722
15000000	796
20000000	788

Figura 12: Tamaño de casos y tiempos de ejecución del algoritmo 2.2

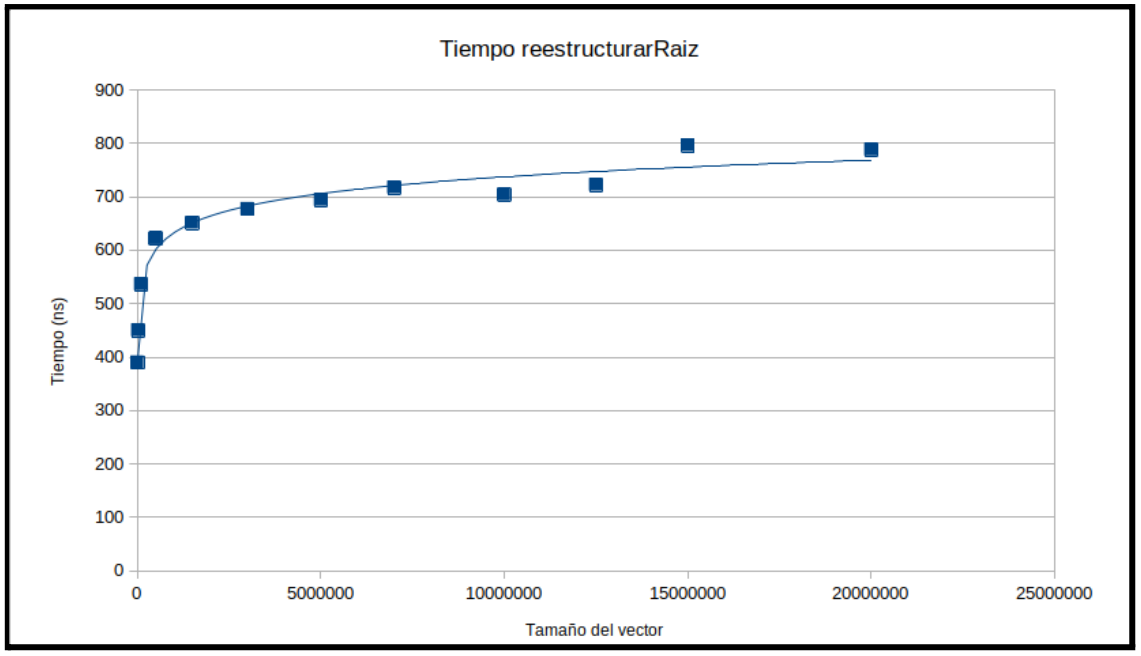

Figura 13: *Gráfica tiempo de ejecución práctico algoritmo* 2. 2

2.2.3. Eficiencia híbrida

Con los valores de la ejecución práctica ya obtenidos y representados en las gráficas anteriores, procedemos a analizar la eficiencia híbrida del algoritmo, teniendo en cuenta que el valor de la eficiencia teórica es $O(log(n))$ realizamos los siguientes cálculos:

Tam. Caso	Tiempo (us)	K=Tiempo/f(n)	Tiempo teórico estimado= K*f(n)
5000	390	105,434756	389,8683659
20000	450	104,6261013	453,3249888
100000	536	107,2	526,9958467
500000	622	109,1425292	600,6667045
1500000	651	105,4064736	650,9548885
3000000	677	104,5217425	682,6832
5000000	694	103,5980157	706,0658739
7000000	717	104,746491	721,4676475
10000000	704	100,5714286	737,7941854
12500000	722	101,7344166	748,0084202
15000000	796	110,9238959	756,3540578
20000000	788	107,9299771	769,5224969
K promedio:		105,3991693	

Figura 14: *Cálculo de la constante oculta mínima y tiempos de ejecución algoritmo* 2. 2

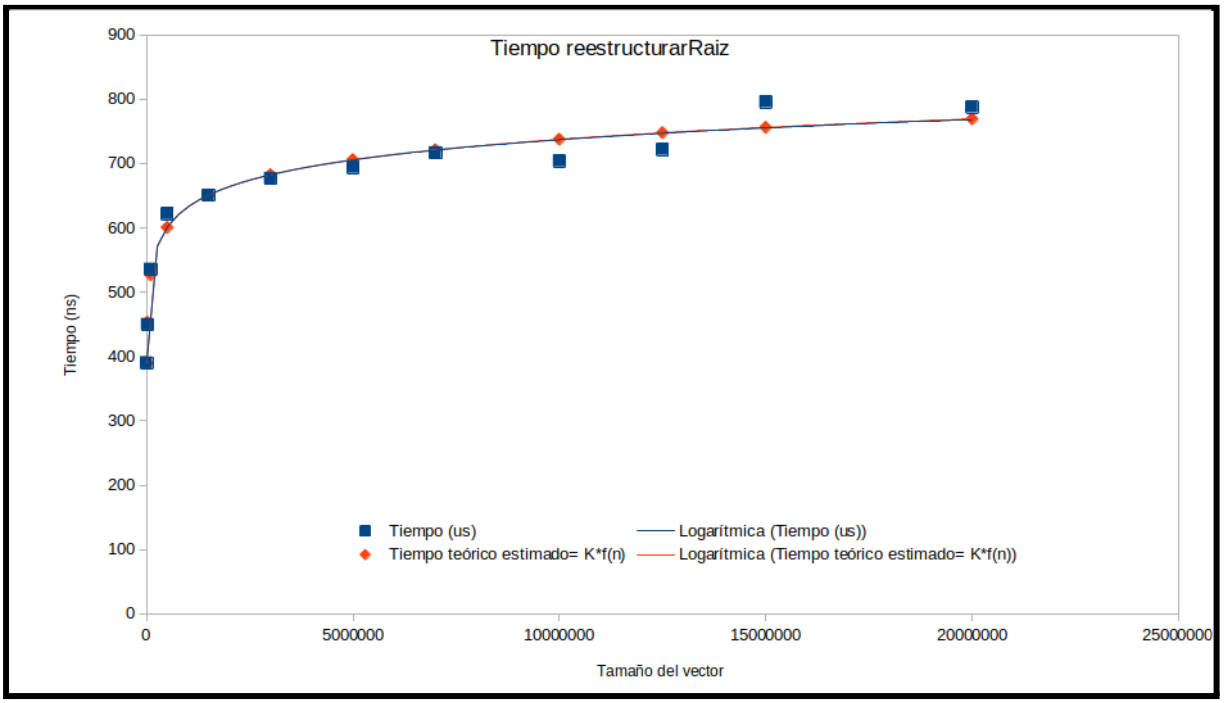

Figura 15: *Comparación tiempo de ejecución real vs. tiempo de ejecución teórico algoritmo* 2. 2

Como podemos observar en los valores de la tabla y en el propio gráfico (*figuras* 14 *y* 15), los valores prácticos se asemejan bastante a los valores teóricos calculados (se puede apreciar la tendencia logarítmica de la gráfica), si bien es cierto que hay ciertas fluctuaciones, que pueden proceder del ruido que haya en el computador, pues el programa del algoritmo no es el único que está siendo ejecutado en el ordenador en ese instante de tiempo.

3. Algoritmo 3 (HeapSort)

3.1. Eficiencia Teórica

```
14 void HeapSort (int *v; int n) {
15
16     double *apo = new double [n];
17     int tamapo = 0;
18
19     for (int i=0; i<n; i++) {
20         apo[tamapo]=v[i];
21         tamapo++;
22         insertarEnPos(apo, tamapo);
23     }
24
25     for (int i=0; i<n; i++) {
26         v[i] = apo[0];
27         tamapo--;
28         apo[0] = apo[tamapo];
29         reestructurarRaiz(apo, 0, tamapo);
30     }
31     delete [] apo;
32 }
33
```

Figura 16: *Código algoritmo* 3

El tercer y último algoritmo es "HeapSort". Este usa los dos algoritmos anteriores, que, pese a que son recursivos, este es iterativo. Se encargará de usar las dos estrategias anteriores para estabilizar el "APO". Puesto que es un algoritmo iterativo, el cálculo de su eficiencia es mucho más sencillo y más aún sabiendo que "insertarEnPos" y "reestructurarRaíz" son $O(log(n))$.

Como podemos ver, se resuelve para un problema de un vector "v" y de tamaño "n".

Comenzamos a analizar por las líneas 16-17. Esto son operaciones elementales y por tanto de $O(1)$ ambas.

Seguimos por el primer bucle "for". Si vamos de dentro hacia fuera, en la línea 22 tenemos la primera llamada a una de las funciones que analizamos anteriormente. Como ya vimos su eficiencia es $O(log(n))$. Las otras dos operaciones que tenemos dentro del bucle (líneas 20-21) son accesos a arrays y asignaciones y por tanto $O(1)$. Finalmente el bucle "for" tiene asignación, comprobación y actualización, todas ellas $O(1)$. Este se ejecutará siempre "n" veces. Por tanto su orden de eficiencia viene dado por la expresión $O(n)$ * $(O(1) + O(log(n)))$, que por la regla del máximo $(max(O(n), O(n\,log(n)))$ la eficiencia es $O(n\,log(n))$.

El segundo bucle “for” es similar al anterior. Se repite “n” veces siempre y dentro del bucle se ejecutan operaciones elementales y una llamada a uno de los algoritmos anteriores cuyo orden de eficiencia vuelve a ser $O(log(n))$. Por tanto el orden de eficiencia de este bucle también es $O(n\ log(n))$.

Finalmente, la última línea de código elimina el puntero del vector auxiliar que hemos usado, es decir, una operación elemental de orden $O(1)$.

En conclusión, el orden final que nos quedaría en “HeapSort” sería:

$$O(n\ log(n)) \ + \ O(n\ log(n)) \ + \ O(1)$$

Que si aplicamos la regla del máximo nos queda un orden de eficiencia: $O(n\ log(n))$.

3.2. Eficiencia Práctica

A continuación, después de realizar el análisis teórico del código, y obtener que la eficiencia teórica del algoritmo es $O(n\ *\ log(n))$, hemos decidido dar el siguiente rango de valores que viene reflejado en la $figura\ 17$ y así obtener los tiempos prácticos que han sido los siguientes:

Tam. Caso	Tiempo (us)
100	50
200	85
400	166
800	453
1600	899
3200	1691
6400	2960
12800	5840
25600	9241
51200	20558
102400	42584
204800	88120
409600	187098
819200	406993
1638400	854632
3276800	1788246
6553600	3828945
13107200	7831863

Figura 17: Tamaño de casos y tiempos de ejecución algoritmo 3

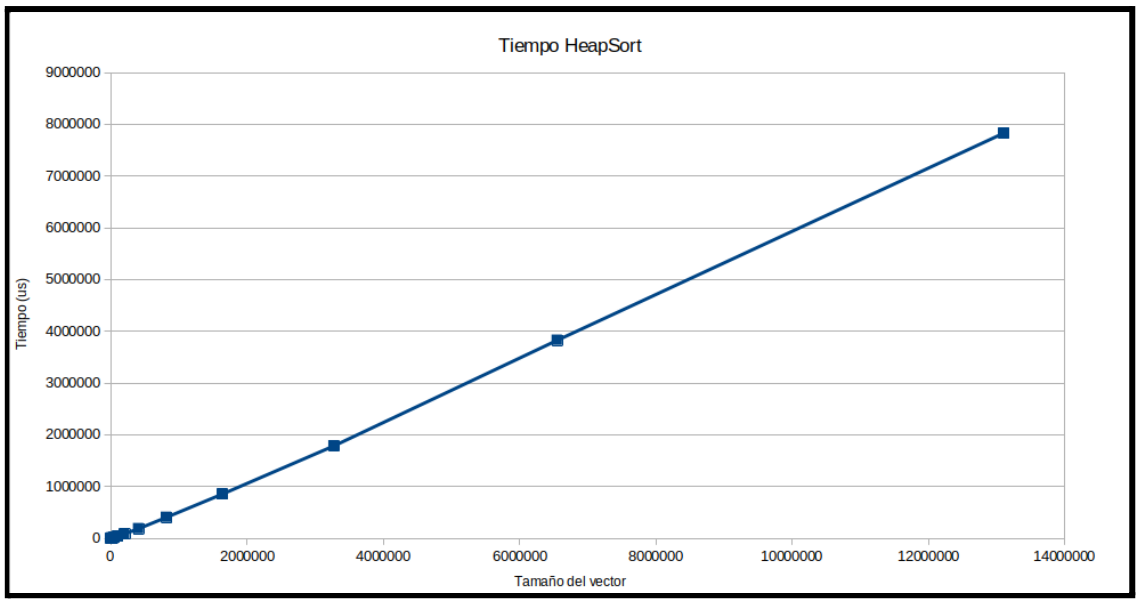

Figura 18: Gráfica tiempo de ejecución práctico algoritmo 3

3.3. Eficiencia Híbrida

Por último, una vez analizada la parte teórica y obtenidos los valores prácticos de la ejecución del algoritmo, podemos pasar a realizar un análisis híbrido:

Tam. Caso	Tiempo (us)	K=Tiempo/f(n)	Tiempo teórico estimado= K*f(n)
100	50	0,25	24,24314516
200	85	0,1846998956	55,7842042
400	166	0,1594890208	126,1642362
800	453	0,1950507916	281,5201279
1600	899	0,1753601622	621,4235668
3200	1691	0,1507603108	1359,613756
6400	2960	0,1215129088	2952,760756
12800	5840	0,1110851413	6372,588
25600	9241	0,08188677689	13679,30898
51200	20558	0,08526235294	29226,88391
102400	42584	0,08300089388	62190,29972
204800	88120	0,08101048916	131853,6633
409600	187098	0,08138861919	278653,4541
819200	406993	0,08401570532	587199,1636
1638400	854632	0,08393799931	1234182,838
3276800	1788246	0,08375927612	2587934,696
6553600	3828945	0,08571148845	5415007,435
13107200	7831863	0,08395123205	11308290,95
K promedio		**0,1212157258**	

Figura 19: Cálculo de la constante oculta mínima y tiempos de ejecución algoritmo 3

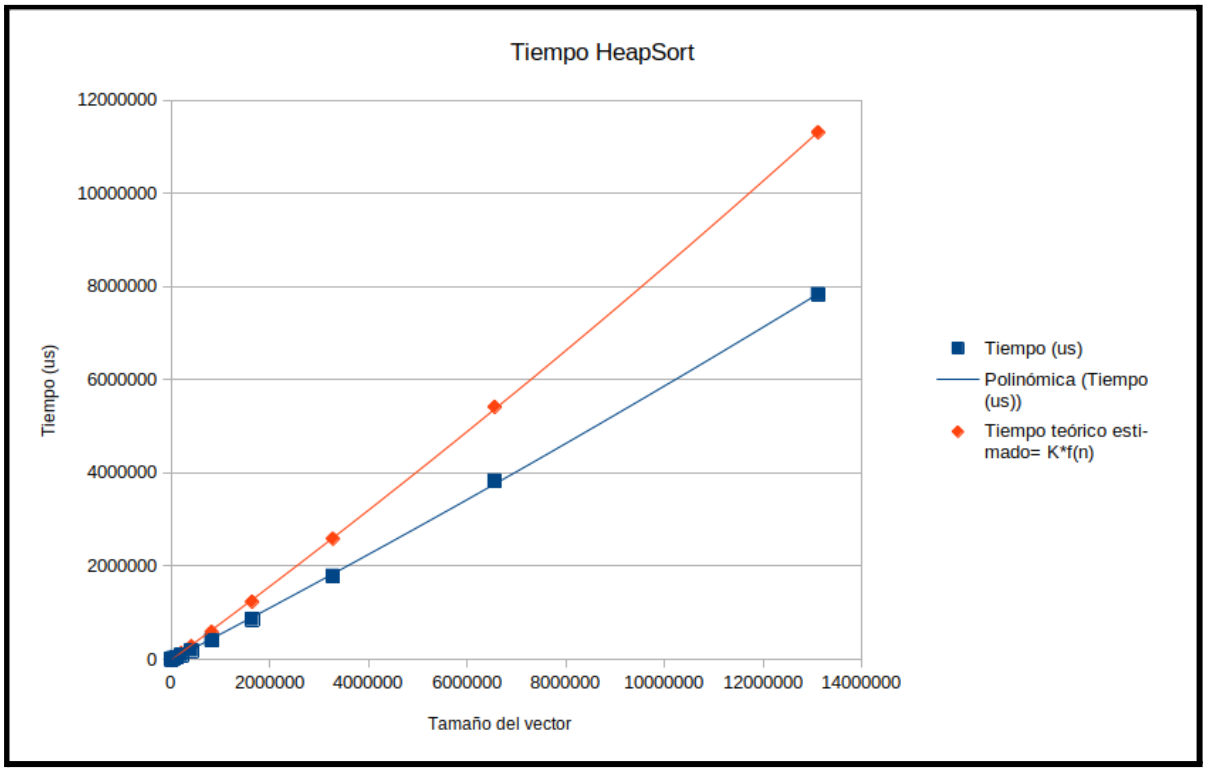

Figura 20: *Comparación tiempo de ejecución real vs. tiempo de ejecución teórico algoritmo* 3

Como podemos observar, los valores de la eficiencia teórica y práctica son bastante parecidos, aunque haya ciertas diferencias, sobre todo para tamaños del vector muy grandes, ya que como hemos comentado anteriormente no todos los recursos del ordenador están disponibles y siempre existe cierto "ruido". Independientemente de este apunte, se observa claramente que las dos funciones son $n * log(n)$.

III. Conclusiones

Como conclusión vamos a hacer una comparación entre los diferentes algoritmos de ordenación que hemos visto en la práctica (en cuanto a eficiencia teórica e híbrida se refiere): "algoritmos burbuja", "mergesort" y "heapsort". Antes de comenzar cabe mencionar que la eficiencia teórica de ordenación por burbuja es $O(n^2)$ mientras que la de MergeSort es $O(n\,log(n))$. Por tanto, ya podríamos hacer la primera comparación y afirmar que "teóricamente", ordenación por burbuja es menos eficiente que los otros dos y que, los restantes tienen la misma eficiencia. Como bien sabemos, realmente esto no es así, ya que para poder comparar dos algoritmos es necesario hacer pruebas reales y tener en cuenta varios factores que intervienen en la ejecución.

En la figura de abajo podemos observar los resultados obtenidos de la ejecución de estos tres algoritmos con el mismo tamaño del vector.

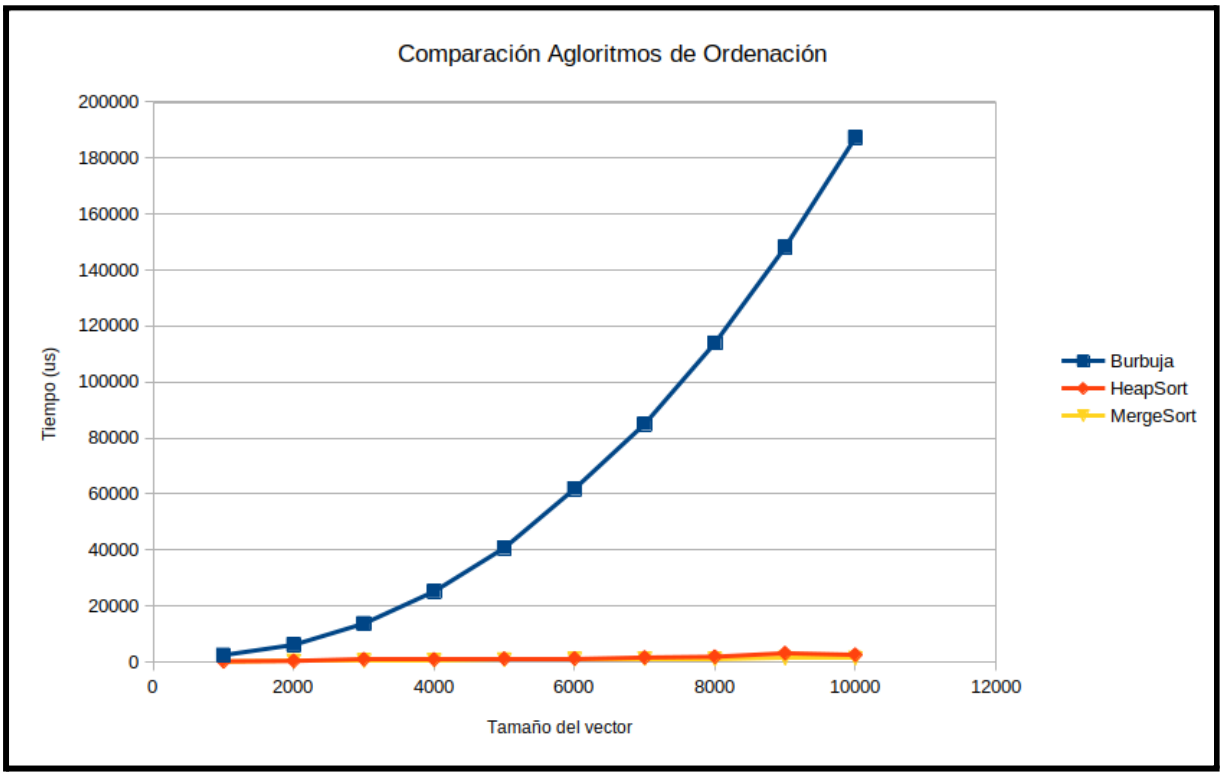

Figura 21: Comparación entre algoritmos de búsqueda

Como podemos comprobar el algoritmo de ordenación por burbuja, al ser cuadrático tiene tiempos de ejecución mucho mayores que los otros dos. Para ver mejor los otros algoritmos vamos a hacer una gráfica para ellos.

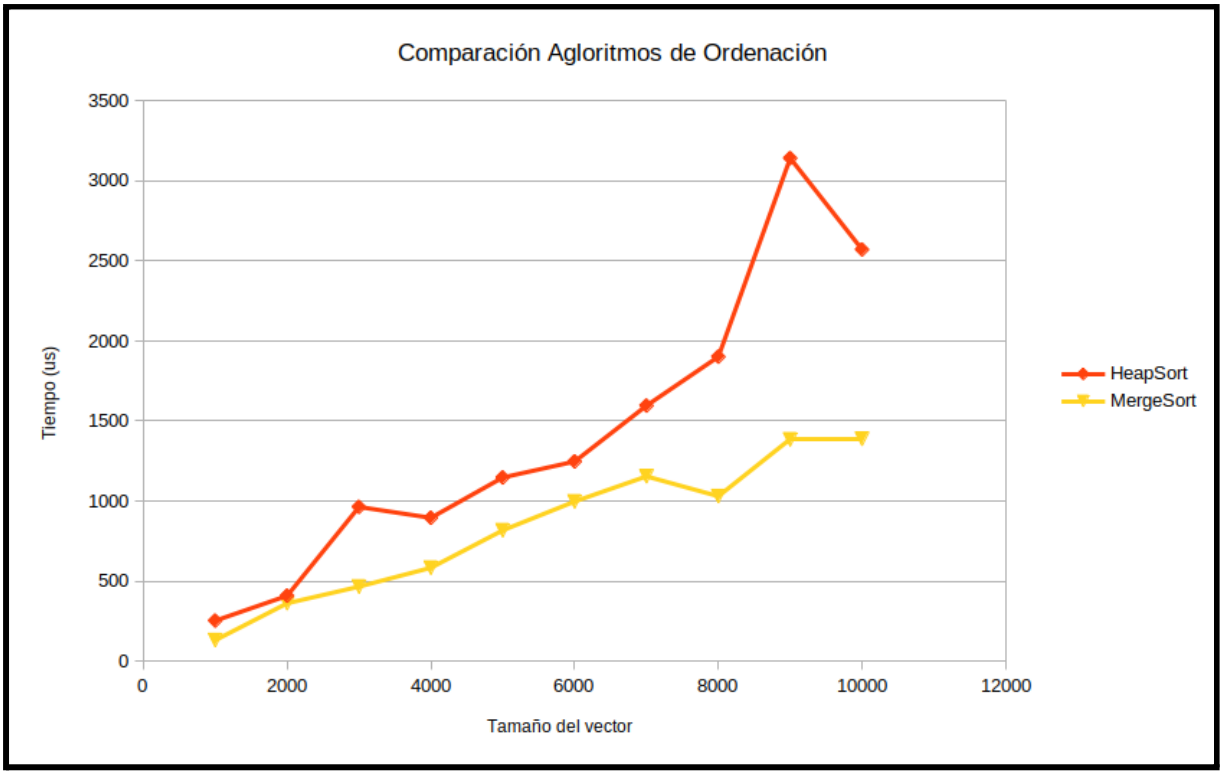

Figura 22: Comparación de tiempo práctico de: "MergeSort" y "HeapSort"

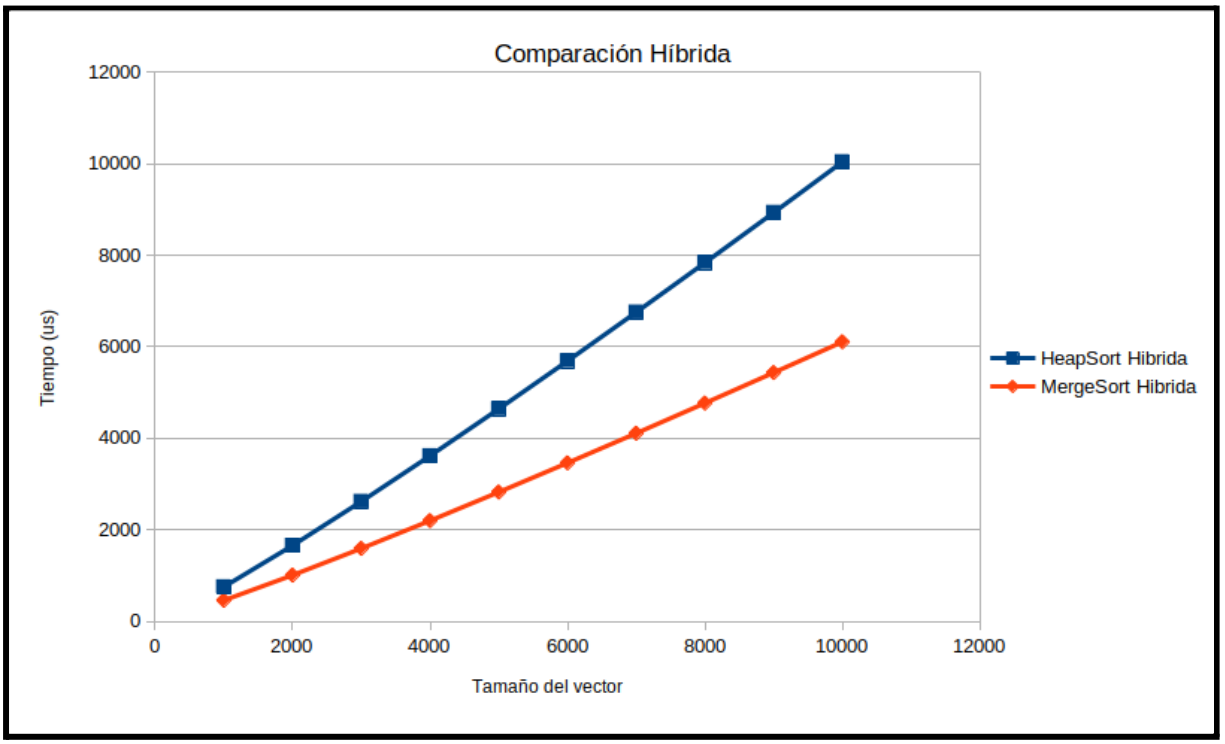

Figura 23: Comparación del tiempo teórico estimado de "HeapSort" y "MergeSort"

Aunque por muy poco, los tiempos de ejecución del algoritmo “MergeSort” son ligeramente menores que “HeapSort”, tanto en el tiempo práctico como en el teórico estimado, por tanto podemos afirmar que en la práctica, “MergeSort” es mejor. Sin embargo deberíamos tener en cuenta otra serie de factores para determinar cuál es el algoritmo más apropiado para usar en cada situación.