PRÁCTICA 1

CÁLCULOS DE LA EFICIENCIA DE ALGORITMOS

UNIVERSIDAD DE GRANADA

Antonio Jiménez Rodríguez 2D (D2)

1. Solución a los problemas propuestos de eficiencia teórica

>> Algoritmos no recursivos

a) Pivotar

El algoritmo compara los elementos de un vector con el pivote (que lo coloca en la posición 0 del vector) y va intercambiandolos si el elemento de la derecha es menos que el de la izquierda. Finalmente cambia el pivote por el elemento que está en la posición j si este está en una posición mayor que 0.

PEOR CASO

Comenzaremos analizando la parte más interna del algoritmo y finalizaremos por las sentencias de primer nivel.

Así que lo primero será ver el orden de eficiencia que tiene el *if* dentro del bucle *while*. En la sentencia condicional tanto la evaluación de la condición como el bloque que ejecuta cuando este es verdadero son O(1) porque son sentencias simples (operaciones matemáticas, asignaciones, etc). Por lo tanto la eficiencia en el peor caso del *if* es el max{O(1), O(1)} = O(1).

Dentro del mismo bucle *while* tenemos otros dos bucles *while*. El primero de ellos en el peor caso se ejecutaría j - 1 veces, siendo j = fin = n = fin - ini ya que ini = 0. Por tanto el tiempo de ejecución seria k(n-1) siendo k una constante del tiempo de ejecución del bloque de sentencias al entrar en el while. Por tanto la eficiencia es igual a O(n).

Como podemos ver si uno de los 2 bucles se ejecuta por completo el otro no se ejecutará por las condiciones de cada uno, i<=j y j>=i.

Siguiendo con el bucle *while* la evaluación de la condición es O(1) y como podemos ver si uno de los 2 bucles se ejecutan n-1 veces en su peor caso, este bucle while solo se ejecutaría 1 vez (en el inicio del programa) por lo que podríamos considerarlo tiempo constante y con una eficiencia de O(1).

Por último tenemos el último condicional *if* que tanto la condición como el bloque de sentencias que ejecuta tienen una eficiencia O(1). Por lo tanto la eficiencia total del algoritmo es de O(n).

MEJOR CASO

Para el mejor caso ocurre exactamente lo mismo, no hay una manera de que el conjunto de los 3 bucles se ejecuten por debajo de las n veces ya que las 3 tienen las condiciones teniendo en cuenta las variables i y j. Por lo tanto la eficiencia en el mejor de los casos sería $\Omega(n)$.

b) Búsqueda

El algoritmo busca un elemento en un vector que ya está ordenado. Siempre mira la posición central y si el elemento que busca es menor, parte el vector por la mitad y busca en la parte izquierda. Si es mayor busca en la parte derecha. Y si el centro coincide con el valor que se está buscando lo devuelve.

PEOR CASO

En este ejemplo vamos a comenzar comentando el bucle *while* que donde se encuentra las principales operaciones del algoritmo.

Como podemos ver tiene una sentencia condicional *if else* y en ambas partes la eficiencia es O(1) ya que son operaciones elementales, al igual que la condición que también es O(1). Además a continuación del bloque *if else* hay una sentencia de O(1) también. Por lo tanto para ver la eficiencia del bucle debemos ver cuantas veces se ejecuta.

El algoritmo va dividiendo el problema en 2, depende si el elemento a buscar es más pequeño, más grande o igual al centro. Por lo tanto en el peor caso que el elemento esté en uno de los extremos por ejemplo el bucle *while* se ejecutará log(n) veces.

El resto de sentencias del algoritmo (*if* y asignaciones) tienen una eficiencia de O(1) así que la eficiencia final del algoritmo sería de O(log(n)).

MEJOR CASO

El mejor caso es muy sencillo, basta con que la posición del elemento que estemos buscando coincida con el centro por lo tanto del bucle while solo se ejecutaría la condición y no llegaría a entrar. En este caso la eficiencia en el mejor caso sería $\Omega(1)$.

c) EliminaRepetidos

Busca valores repetidos en un vector ya ordenado. Si encuentra uno en la posición k+1, desplaza el vector restante hacia la izquierda, machacando el valor repetido y decrementa en uno el tamaño del vector.

PEOR CASO

Comenzamos analizando la sentencia *if else* que se encuentra dentro del *do while*. La eficiencia es el máximo entre la evaluación de la condición, el bloque de sentencias del *if* y el bloque de sentencias del *else*. En este caso, el *else* únicamente incrementa el valor de j por lo que la eficiencia es un O(1) al igual que el de la evaluación de la condición. Dentro del *if* tenemos un bucle for el cual se ejecutará en el peor de los casos k $\sum_{i=1}^{n}$ n-i (la asignación, evaluación, incremento y cuerpo es un O(1)) por lo tanto la eficiencia de este *if else* es de O(n).

El bucle *do while* le pasa algo parecido, se ejecutará en el peor de los casos k $\sum_{i=1}^{n}$ n-i, pero en este caso la eficiencia del cuerpo es de O(n), por lo que la eficiencia de este bucle es de O(n^2).

Por último el bucle for se ejecuta n veces así que la eficiencia final en el peor caso de este algoritmo es de O(n^3).

MEJOR CASO

En el mejor de los casos el bucle *for* y el *do while* se ejecutan el mismo número de veces, sin embargo el *for* que se encuentra dentro del *if* no se ejecutaría ninguna vez ya que si no hubiera repetidos no entraría dentro. Por lo tanto la eficiencia en el mejor caso del algoritmo es $\Omega(n^2)$.

>> Algoritmos recursivos

a) hanoi

En el código del algoritmo tenemos un condicional *if* donde se realizan una llamada recursiva con el tamaño del caso, n-1, un *cout* de

tiempo constante (que podemos obviar) y otra llamada recursiva. Por lo tanto el tiempo de ejecución del algoritmo sería T(n)=T(n-1) + T(n-1)=2T(n-1).

Vamos resolver la ecuación recurrente no homogénea:

1) Pasamos todos los términos de T a la izquierda

$$T(n) - 2T(n-1) = 0$$

2) Resolvemos la ecuación homogénea

$$T(n) - 2T(n-1) = 0$$
$$t_n - 2\,t_{n-1} = 0$$
$$x^n - 2\,x^{n-1} = 0$$
$$x^{n-1}\,(x - 2) = 0$$
$$p\,(x) = x-2$$

3) Calculamos t_n

$$t_n = c_1\ 2^n\ n^0 \ \rightarrow \text{ El algoritmo es } O(2^n)$$

b) HeapSort

En el siguiente algoritmo tenemos dos llamadas a dos algoritmo recursivos por lo que empezaremos analizándolos

b.1) insertarEnPos

Este algoritmo inserta elementos en un vector de nombre apo con la estructura de un árbol APO.

Al ser un algoritmo recursivo tenemos que calcular el tiempo de ejecución del mismo. En el peor de los casos el nuevo elemento a añadir puede ser el más pequeño por lo que irá comprobando con el padre (dividiendo la posición del nuevo entre 2), intercambiandolos y posteriormente volver a llamarse recursivamente. Por lo tanto el tiempo de ejecución del algoritmo será:

$$T(n) = 1 + T(n/2)$$

Vamos resolver la ecuación recurrente no homogénea:

1) Realizamos un cambio de variable

$$n = 2^m \ \rightarrow\ m = \log(n)$$

2) Pasamos todos los términos de T a la izquierda con el cambio de variable aplicado

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

3) Resolvemos la parte de la izquierda (sacamos el polinomio característico homogéneo)

$$T(2^m) - T(2^{m-1}) = 0$$

$$t_m - t_{m-1} = 0$$

$$x^m - x^{m-1} = 0$$

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

$$p_h(x) = x-1$$

4) Resolvemos la parte de la derecha (parte no homogénea)

$$1 = b_1^m \bullet q_1(m)$$

$$b_1 = 1$$

$$q_1(m) = 1 \rightarrow d_1 = 0$$

5) Construimos el polinomio característico y calculamos t_n

$$p(x) = p_h(x) \bullet (x - b_1)^{d1+1} = (x - 1)^2$$

$$t_m = c_1\ 1^m\ m^0 + c_2\ 1^m\ m^1$$

6) Deshacemos el cambio de variable

m = log(n) $\rightarrow$ $t_n = c_1\ 1^{log(n)}\ log(n)^0 + c_2\ 1^{log(n)}\ log(n)^1$ $\rightarrow$ La función es O(log(n))

b.1) reestructurarRaiz

En esta función recursiva pasa algo parecido, reconstruye el vector APO para devolver el vector ordenado. Ser recursiva implica como ya

sabemos que se vuelve a llamar a ella misma. El tamaño del problema en este caso sería log(n) ya que el parámetro pos lo va incrementando el doble en cada llamada (valor de minhijo). Por tanto el tiempo de ejecución del algoritmo será:

$$T(n) = 1 + T(n/2)$$

Vamos resolver la ecuación recurrente no homogénea:

1) Realizamos un cambio de variable

$$n = 2^m \rightarrow m = \log(n)$$

2) Pasamos todos los términos de T a la izquierda con el cambio de variable aplicado

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

3) Resolvemos la parte de la izquierda (sacamos el polinomio característico homogéneo)

$$T(2^m) - T(2^{m-1}) = 0$$

$$t_m - t_{m-1} = 0$$

$$x^m - x^{m-1} = 0$$

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

$$p_h(x) = x-1$$

4) Resolvemos la parte de la derecha (parte no homogénea)

$$1 = b_1^m \cdot q_1(m)$$

$$b_1 = 1$$

$$q_1(m) = 1 \rightarrow d_1 = 0$$

5) Construimos el polinomio característico y calculamos t_n

$$p(x) = p_h(x) \cdot (x - b_1)^{d1+1} = (x - 1)^2$$

$$t_m = c_1 1^m m^0 + c_2 1^m m^1$$

6) Deshacemos el cambio de variable

m = log(n) $\rightarrow$ $t_n = c_1\ 1^{log(n)}\ log(n)^0 + c_2\ 1^{log(n)}\ log(n)^1$ $\rightarrow$ La función es O(log(n))

Volviendo al código del algoritmo HeapSort como podemos ver, la llamada a la función insertarEnPos está dentro de un bucle *for* que se ejecuta n veces. Por lo tanto la eficiencia de ese bucle es de n por la eficiencia de la función, es decir n*log(n).

Igualmente en el otro bucle *for* que también se ejecuta n veces se encuentra la llamada a reestructurarRaiz por lo tanto la eficiencia será n*log(n).

Por lo tanto la eficiencia final del algoritmo es O(n*log(n)).

2. Solución a los problemas guiados de eficiencia teórica

>> MergeSort

En el código de este algoritmo tenemos dos llamadas recursivas a la función con tamaño n/2 ya que en una llamada le pasamos la mitad de un vector y en la otra llamada la otra mitad, y una llamada a la función combina a la cual vamos a estudiar la eficiencia para poder sacar la ecuación en recurrencia del algoritmo.

Como podemos ver en el primer while se recorren los vectores con una eficiencia O(n) comparando las componentes una a una de las dos partes y cogiendo el menor elemento. Cuando una de las dos partes termina se vuelca el resto de la otra que también se hace con eficiencia O(n).

Por tanto el tiempo de ejecución del algoritmo será:

$$T(n) = 2*T(n/2) + n$$

Vamos resolver la ecuación recurrente no homogénea:

1) Realizamos un cambio de variable

$$n = 2^m \rightarrow m = \log(n)$$

2) Pasamos todos los términos de T a la izquierda con el cambio de variable aplicado

$$T(2^m) - 2*T(2^{m-1}) = 2^m$$

3) Resolvemos la parte de la izquierda (sacamos el polinomio característico homogéneo)

$$T(2^m) - T(2^{m-1}) = 0$$

$$t_m - t_{m-1} = 0$$

$$x^m - x^{m-1} = 0$$

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

$$p_h(x) = x - 2$$

4) Resolvemos la parte de la derecha (parte no homogénea)

$$2^m = b_1^m \bullet q_1(m)$$

$$b_1 = 2$$

$$q_1(m) = 1 \rightarrow d_1 = 0$$

5) Construimos el polinomio característico y calculamos t_n

$$p(x) = p_h(x) \bullet (x - b_1)^{d1+1} = (x - 2)^2$$

$$t_m = c_1\, 2^m\, m^0 + c_2\, 2^m\, m^1$$

6) Deshacemos el cambio de variable

$m = \log(n) \rightarrow t_n = c_1\, n * log(n)^0 + c_2\, n * log(n)^1 \rightarrow$ La función es O(n*log(n))

>> Ordenación por burbuja (Practica1.cpp)

Este algoritmo lo que hace es recorrer un vector y comparar los elementos consecutivos. Si el elemento i es mayor que el elemento i+1 los intercambia y sigue recorriendo el vector. Repite esta operación hasta que al recorrer el vector entero no tenga que intercambiar más posiciones.

Por lo tanto como podemos ver la eficiencia del algoritmo es en el peor caso (que el vector esté ordenado descencentemente) $O(n^2)$ ya que ejecuta un bucle *for* dentro de un bucle *while* con el tamaño del caso n que es la longitud del vector.

En el mejor de los casos la eficiencia sería $\Omega(n)$ porque el bucle *while* solo lo ejecutaría una vez ya que el vector ya estaría ordenado y el bucle *for* recorrería se ejecutaría n veces.

3. Solución a los problemas propuestos de eficiencia práctica

>> Tiempo de ejecución para diversos tamaños de caso para el algoritmo MergeSort

```
antonio@antonio-Lenovo-Y520-15IKBN:~/Documentos/5o Año/SEGUNDO CUATRIMESTRE/ALG/PRACTICAS/PRAC
ICA1/Practica1$ ./mergesort salidaMergeSort.txt 1000 2000 3000 4000 5000 6000 7000 8000 9000
10000 20000 30000 40000 50000 60000 70000 80000 90000 100000
        Tiempo de ejec. (us): 583 para tam. caso 1000
        Tiempo de ejec. (us): 1217 para tam. caso 2000
        Tiempo de ejec. (us): 905 para tam. caso 3000
        Tiempo de ejec. (us): 563 para tam. caso 4000
        Tiempo de ejec. (us): 737 para tam. caso 5000
        Tiempo de ejec. (us): 875 para tam. caso 6000
        Tiempo de ejec. (us): 1072 para tam. caso 7000
        Tiempo de ejec. (us): 1508 para tam. caso 8000
        Tiempo de ejec. (us): 1481 para tam. caso 9000
        Tiempo de ejec. (us): 1654 para tam. caso 10000
        Tiempo de ejec. (us): 3587 para tam. caso 20000
        Tiempo de ejec. (us): 5441 para tam. caso 30000
        Tiempo de ejec. (us): 7747 para tam. caso 40000
        Tiempo de ejec. (us): 8867 para tam. caso 50000
        Tiempo de ejec. (us): 10737 para tam. caso 60000
        Tiempo de ejec. (us): 14076 para tam. caso 70000
        Tiempo de ejec. (us): 15259 para tam. caso 80000
        Tiempo de ejec. (us): 17424 para tam. caso 90000
        Tiempo de ejec. (us): 19187 para tam. caso 100000
```

>> Tiempo de ejecución para diversos tamaños de caso para el algoritmo HeapSort

```
antonio@antonio-Lenovo-Y520-15IKBN:~/Documentos/5o Año/SEGUNDO CUATRIMESTRE/ALG/RACTICAS/PRACT
ICA1/Practica1$ ./heapsort salida.txt 1000 2000 3000 4000 5000 60
00 7000 8000 9000 10000 20000 30000 40000 50000 60000 70000 80000 90000 100000
        Tiempo de ejec. (us): 700 para tam. caso 1000
        Tiempo de ejec. (us): 1005 para tam. caso 2000
        Tiempo de ejec. (us): 598 para tam. caso 3000
        Tiempo de ejec. (us): 794 para tam. caso 4000
        Tiempo de ejec. (us): 938 para tam. caso 5000
        Tiempo de ejec. (us): 1638 para tam. caso 6000
        Tiempo de ejec. (us): 1684 para tam. caso 7000
        Tiempo de ejec. (us): 2164 para tam. caso 8000
        Tiempo de ejec. (us): 2097 para tam. caso 9000
        Tiempo de ejec. (us): 2487 para tam. caso 10000
        Tiempo de ejec. (us): 5088 para tam. caso 20000
        Tiempo de ejec. (us): 8111 para tam. caso 30000
        Tiempo de ejec. (us): 10756 para tam. caso 40000
        Tiempo de ejec. (us): 12239 para tam. caso 50000
        Tiempo de ejec. (us): 16208 para tam. caso 60000
        Tiempo de ejec. (us): 17634 para tam. caso 70000
        Tiempo de ejec. (us): 20667 para tam. caso 80000
        Tiempo de ejec. (us): 25249 para tam. caso 90000
        Tiempo de ejec. (us): 28731 para tam. caso 100000
```

>> Tiempo de ejecución para diversos tamaños de caso para el algoritmo Burbuja

```
antonio@antonio-Lenovo-Y520-15IKBN: ~/Documentos/5o Año/SEGUNDO CUATRIMESTRE/ALG/PRACTICAS/PRACTIC…
Archivo  Editar  Ver  Buscar  Terminal  Ayuda
antonio@antonio-Lenovo-Y520-15IKBN:~/Documentos/5o Año/SEGUNDO CUATRIMESTRE/ALG/RACTICAS/PRACTICA1/Practica1$ ./Practica1.bin salida.txt 12345 1000 2000 3000 4000 5000 6000 7000 8000 9000 10000 20000 30000 40000 50000 60000 70000 80000 90000 100000
Ejecutando Burbuja para tam. caso: 1000
        Tiempo de ejec. (us): 4854 para tam. caso 1000
Ejecutando Burbuja para tam. caso: 2000
        Tiempo de ejec. (us): 11749 para tam. caso 2000
Ejecutando Burbuja para tam. caso: 3000
        Tiempo de ejec. (us): 27257 para tam. caso 3000
Ejecutando Burbuja para tam. caso: 4000
        Tiempo de ejec. (us): 49022 para tam. caso 4000
Ejecutando Burbuja para tam. caso: 5000
        Tiempo de ejec. (us): 76183 para tam. caso 5000
Ejecutando Burbuja para tam. caso: 6000
        Tiempo de ejec. (us): 111930 para tam. caso 6000
Ejecutando Burbuja para tam. caso: 7000
        Tiempo de ejec. (us): 159761 para tam. caso 7000
Ejecutando Burbuja para tam. caso: 8000
        Tiempo de ejec. (us): 206137 para tam. caso 8000
Ejecutando Burbuja para tam. caso: 9000
        Tiempo de ejec. (us): 266069 para tam. caso 9000
Ejecutando Burbuja para tam. caso: 10000
        Tiempo de ejec. (us): 339252 para tam. caso 10000
Ejecutando Burbuja para tam. caso: 20000
        Tiempo de ejec. (us): 1437923 para tam. caso 20000
Ejecutando Burbuja para tam. caso: 30000
        Tiempo de ejec. (us): 3375119 para tam. caso 30000
Ejecutando Burbuja para tam. caso: 40000
        Tiempo de ejec. (us): 6149725 para tam. caso 40000
Ejecutando Burbuja para tam. caso: 50000
        Tiempo de ejec. (us): 10050783 para tam. caso 50000
Ejecutando Burbuja para tam. caso: 60000
        Tiempo de ejec. (us): 13724639 para tam. caso 60000
Ejecutando Burbuja para tam. caso: 70000
        Tiempo de ejec. (us): 18440699 para tam. caso 70000
Ejecutando Burbuja para tam. caso: 80000
        Tiempo de ejec. (us): 24112936 para tam. caso 80000
Ejecutando Burbuja para tam. caso: 90000
        Tiempo de ejec. (us): 30711603 para tam. caso 90000
Ejecutando Burbuja para tam. caso: 100000
        Tiempo de ejec. (us): 37930088 para tam. caso 100000
```

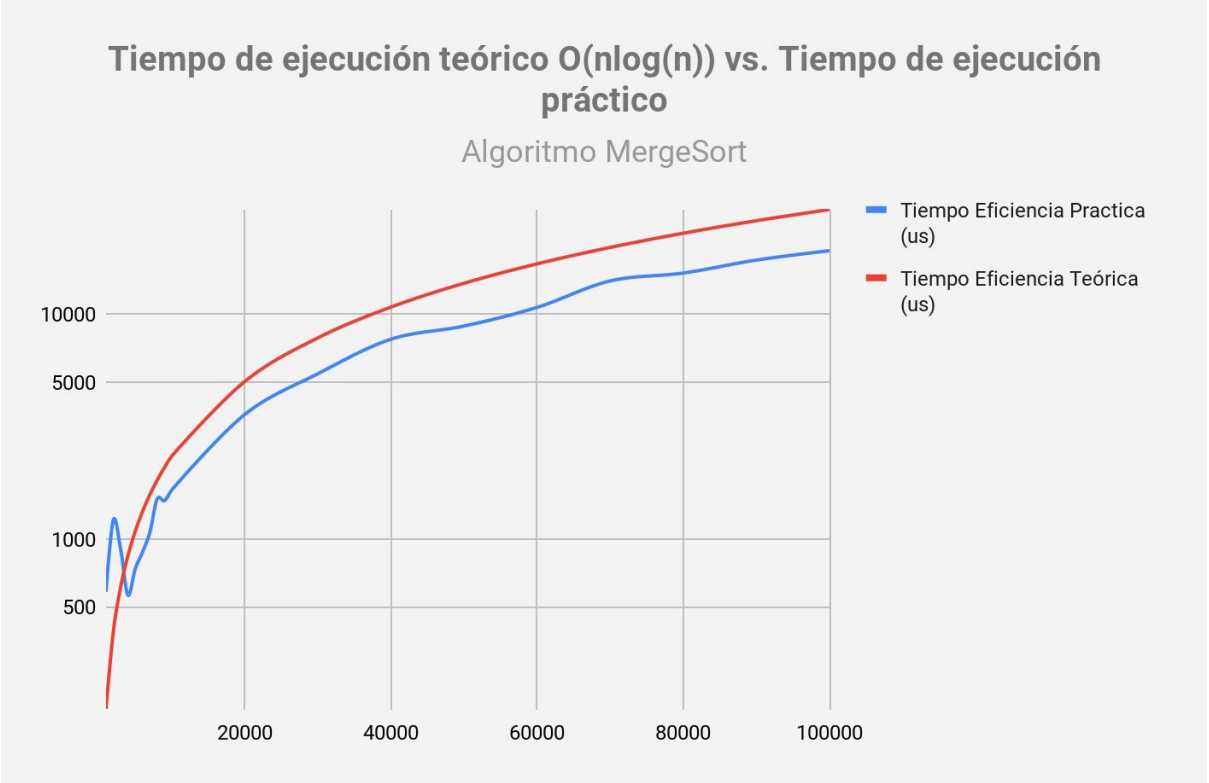
4. Eficiencia híbrida y cálculo de la constante oculta

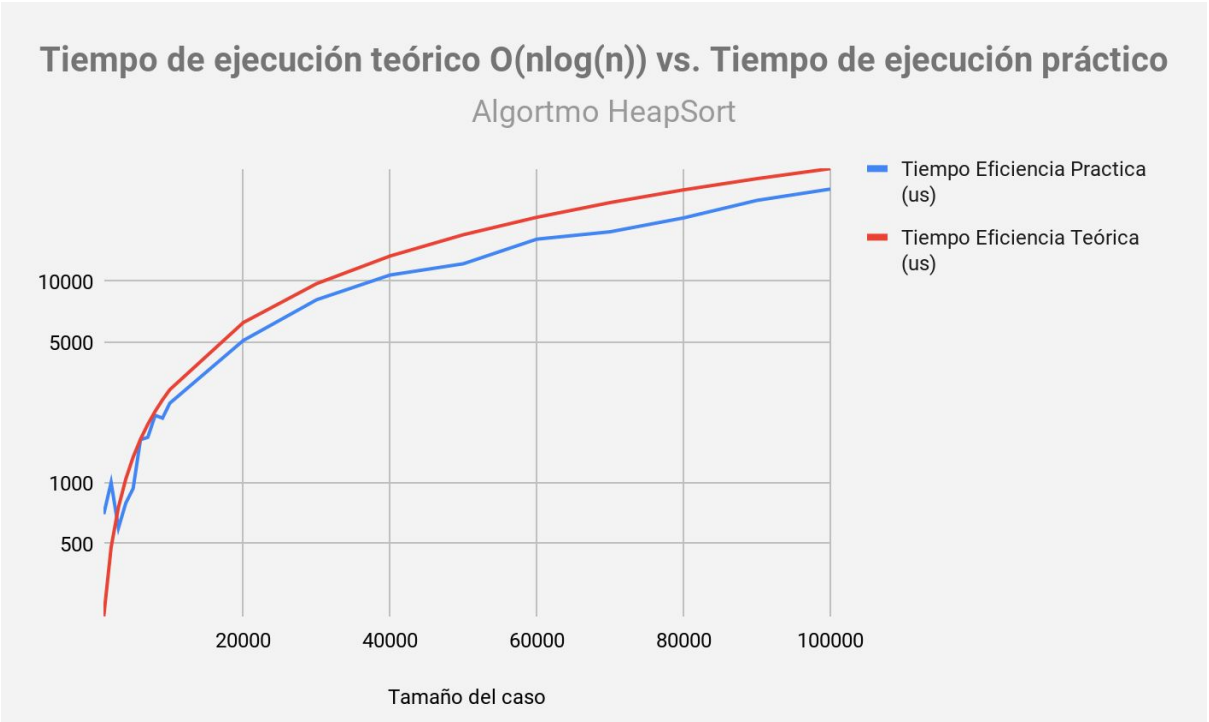
Vamos a calcular la constante K para cada algoritmo, tal que el tiempo T(n) de ejecución del mismo para un tamaño de caso n es:

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

K la calculamos despejando de la fórmula anterior.

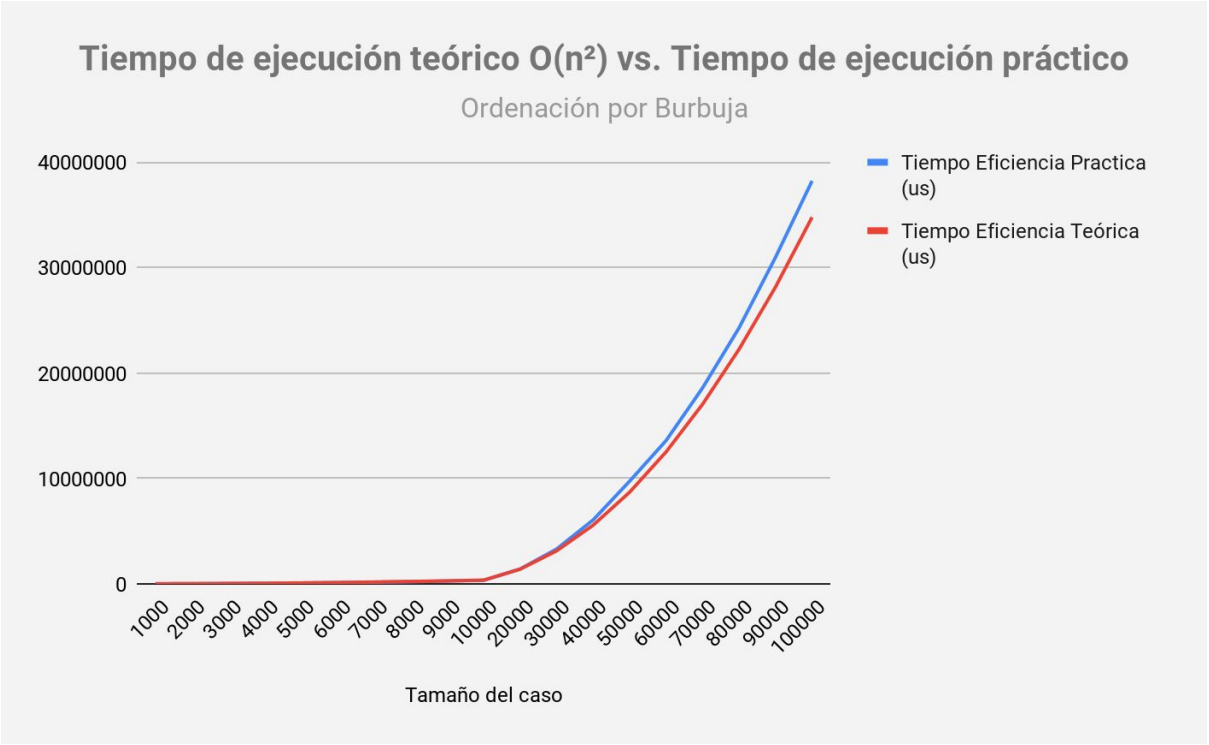
MergeSort **f(n) = n*log(n)**			
Tamaño del caso **n**	Tiempo Eficiencia Práctica (us)	Tiempo Eficiencia Teórica (us) f(n)*K	Constante K T(n)/f(n)
1000	583	175,4809985	0,05850016249
2000	1217	386,1786932	0,055490787
3000	905	610,1687098	0,0261166375
4000	563	842,7907787	0,01176270578
5000	737	1081,831583	0,01199572349
6000	875	1325,987508	0,01161949451
7000	1072	1574,397168	0,01198944556
8000	1508	1826,448342	0,01453826439
9000	1481	2081,683272	0,01252734451
10000	1654	2339,746647	0,01244759032
20000	3587	5031,660255	0,01255274429
30000	5441	7856,497083	0,01219462324
40000	7747	10767,65443	0,0126686711
50000	8867	13742,99914	0,0113609279
60000	10737	16769,49505	0,0112740922
70000	14076	19838,52831	0,0124936237
80000	15259	22943,97671	0,01171051499
90000	17424	26081,26267	0,0117635354
100000	19187	29246,83309	0,01155172505
		K media	0,01760834807


Tiempo de ejecución teórico O(nlog(n)) vs. Tiempo de ejecución práctico
Algoritmo MergeSort
Tiempo Eficiencia Practica (us)
Tiempo Eficiencia Teórica (us)
10000
5000
1000
500
20000
40000
60000
80000
100000


Tiempo de ejecución teórico O(nlog(n)) vs. Tiempo de ejecución práctico
Algortmo HeapSort
Tiempo Eficiencia Practica (us)
Tiempo Eficiencia Teórica (us)
10000
5000
1000
500
20000
40000
60000
80000
100000
Tamaño del caso

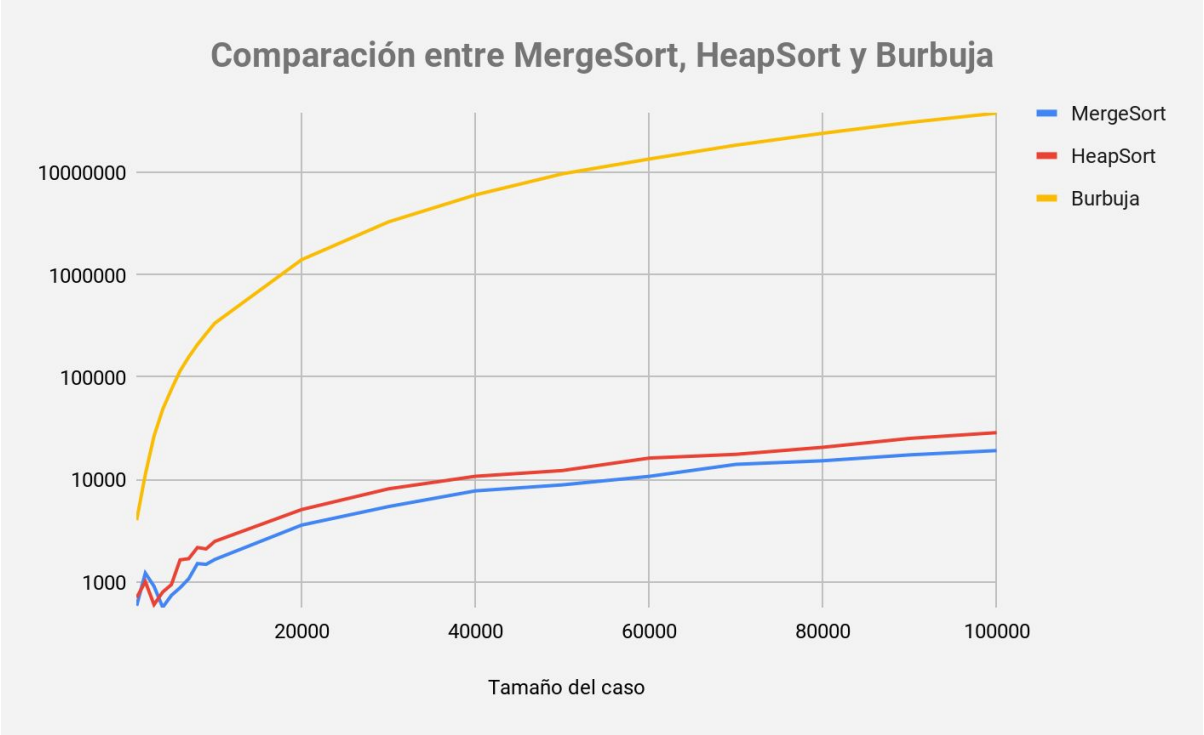
HeapSort **f(n) = n*log(n)**			
Tamaño del caso **n**	Tiempo Eficiencia Práctica (us)	Tiempo Eficiencia Teórica (us) f(n)*K	Constante K T(n)/f(n)
1000	700	217,7846211	0,07024033232
2000	1005	479,2757112	0,04582435574
3000	598	757,2635349	0,01725718147
4000	794	1045,96436	0,01658896695
5000	938	1342,631301	0,01526728444
6000	1638	1645,646477	0,02175169372
7000	1684	1953,940846	0,01883416635
8000	2164	2266,754597	0,02086260222
9000	2097	2583,51962	0,01773790778
10000	2487	2903,794948	0,01871653998
20000	5088	6244,654585	0,0178055096
30000	8111	9750,481559	0,0181787519
40000	10756	13363,43855	0,01758928958
50000	12239	17056,0567	0,0156813349
60000	16208	20812,15719	0,01701876561
70000	17634	24621,04962	0,01565164538
80000	20667	28475,13586	0,01586088298
90000	25249	32368,73483	0,01704645922
100000	28731	36297,43684	0,01729778561
		K media	0,02185323451

Ordenación por burbuja **f(n) = n²**			
Tamaño del caso **n**	Tiempo Eficiencia Práctica (us)	Tiempo Eficiencia Teórica (us) f(n)*K	Constante K T(n)/f(n)
1000	4001	3478,562383	0,004001
2000	11214	13914,24953	0,0028035
3000	26582	31307,06145	0,002953555556
4000	48467	55656,99813	0,0030291875
5000	76013	86964,05958	0,00304052
6000	115407	125228,2458	0,00320575
7000	157967	170449,5568	0,003223816327
8000	208669	222627,9925	0,003260453125
9000	266626	281763,553	0,003291679012
10000	337902	347856,2383	0,00337902
20000	1410814	1391424,953	0,003527035
30000	3304987	3130706,145	0,003672207778
40000	6063618	5565699,813	0,00378976125
50000	9743186	8696405,958	0,0038972744
60000	13597553	12522824,58	0,003777098056
70000	18598032	17044955,68	0,003795516735
80000	24290228	22262799,25	0,003795348125
90000	30996211	28176355,3	0,003826692716
100000	38232697	34785623,83	0,0038232697
		K media	0,003478562383


Tiempo de ejecución teórico O(n²) vs. Tiempo de ejecución práctico
Ordenación por Burbuja
40000000
30000000
20000000
10000000
0
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000
20000
30000
40000
50000
60000
70000
80000
90000
100000
Tamaño del caso
Tiempo Eficiencia Practica (us)
Tiempo Eficiencia Teórica (us)

5. Análisis de resultados

Como podemos ver en las gráficas anteriores y en la resolución teórica de la eficiencia el mejor algoritmo para la ordenación de vectores es el HeapSort o MergeSort con una eficiencia de O(nlog(n)) frente al de ordenación por burbuja con una eficiencia de $O(n^2)$ llegando a superar en tiempo de ejecución hasta en un 33% (1,33 veces más lento).



Este gráfico está realizado a escala logarítmica para poder comparar los tres algoritmos ya que al tener unos valores tan altos el de burbuja no se apreciaba la curvatura del MergeSort y HeapSort.

Cuando el tamaño del caso tiende a infinito en los algoritmos recursivos podemos ver que el tiempo de ejecución tienda a constante. En cambio en el algoritmo de ordenación por burbuja cuando el tamaño del caso tiende a infinito el tiempo también tiende a infinito.

Si tuviéramos que resolver un problema de ordenación y solo pudiéramos escoger uno de los trés, la mejor opción son el MergeSort y el HeapSort. Y en nuestro caso práctico la mejor elección sería el MergeSort que tiene el mejor comportamiento.