ALGORÍTMICA

Eficiencia

PRÁCTICA 1

INFORMÁTICA-ADE

Mercedes León Chaves
Lydia van Dillewijn Soto
Laura Zafra Álarcos

Índice

ALGORITMOS ITERATIVOS

ELIMINAR REPETIDOS

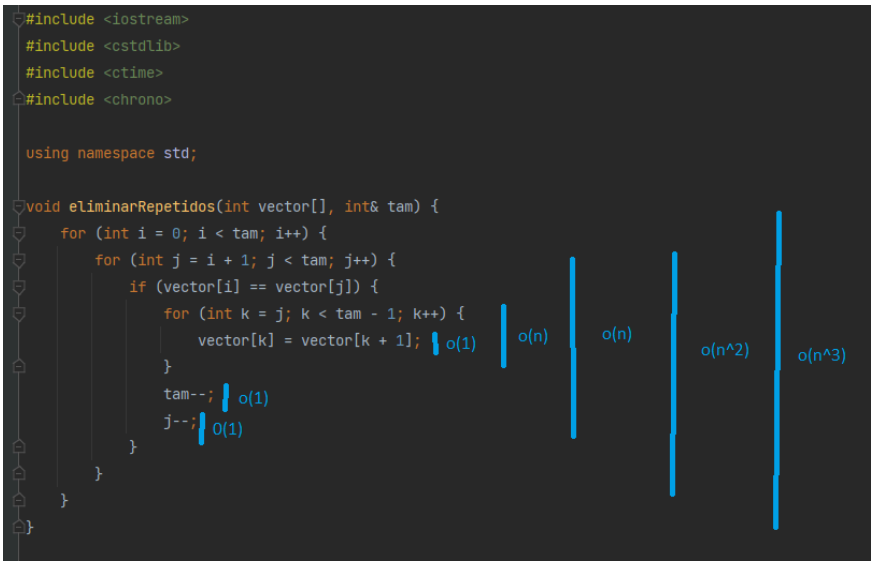
1. Algoritmo:

```cpp
#include <iostream>
#include <cstdlib>
#include <ctime>
#include <chrono>

using namespace std;

void eliminarRepetidos(int vector[], int& tam) {
    for (int i = 0; i < tam; i++) {
        for (int j = i + 1; j < tam; j++) {
            if (vector[i] == vector[j]) {
                for (int k = j; k < tam - 1; k++) {
                    vector[k] = vector[k + 1];
                }
                tam--;
                j--;
            }
        }
    }
}
```

```cpp
int main() {
    srand( seed: time( timer: NULL)); // semilla para números aleatorios
    int tam = 20; // tamaño del vector
    int vector[tam]; // declaración del vector
    chrono::time_point<std::chrono::high_resolution_clock> t0, tf;
    unsigned long duration;

    for (int i = 0; i < tam; i++) { // inicialización del vector con números aleatorios
        vector[i] = rand() % 10; // números aleatorios entre 0 y 9
    }


    t0= std::chrono::high_resolution_clock::now();

    eliminarRepetidos(vector, &tam); // eliminación de elementos repetidos

    tf= std::chrono::high_resolution_clock::now();

    duration= std::chrono::duration_cast<std::chrono::microseconds>( d: tf - t0).count();

    cout << "Tiempo del algortimo" << duration;

    return 0;
}
```

3. El tamaño del problema depende la variable ‘tam’, ya que el for del algoritmo es la instrucción que más va ha hacer aumentar el tiempo de este, que las veces que lo recorremos está determinado por esta variable int.

4 y 5.



Como podemos ver en el algoritmo:

- El mejor de los casos sería uno en el que al poner números aleatorios en el vector, estos no se repitieran en ningún momento, ya que en ese caso no se realizaría el ‘for’ del ‘if’ y por tanto es ese caso solo se ejecutarán dos bucles for que uno está dentro de otro por lo que tendríamos la eficiencia de o(n)*o(n)=o(n^2).

```
Ubuntu: /mnt/c/Users/merch/untitled5/cmake-build-debug/untitled5
Tamaño del vector: 100
Vector inicial:
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 5
Vector final:
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 5
Tiempo del algortimo: 14
Process finished with exit code 0
```

- En el peor de los casos tendríamos uno en el que todos los elementos del vector sean iguales, por lo tanto tiene que realizar los tres bucles for, con lo que obtendremos o(n)*o(n)*o(n)=o(n^3).

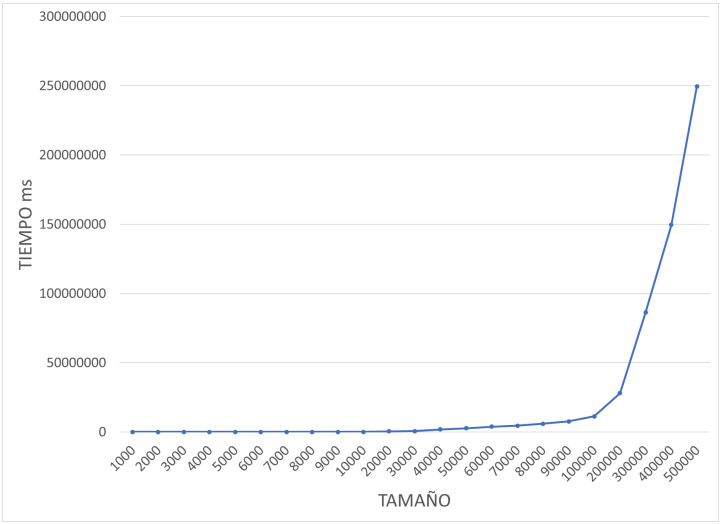
```
Ubuntu: /mnt/c/Users/merch/untitled5/cmake-build-debug/untitled5
Tamaño del vector: 100
Vector inicial:
10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 1
Vector final:
10
Tiempo del algortimo: 33
Process finished with exit code 0
```

Como podemos observar en este ejemplo el tiempo del mejor de los casos es de 14 ms, mientras que el peor de los casos es de 33ms.

Esto como se ha explicado antes, porque en el último caso se está realizando el tercer ‘for’ que hay dentro del ‘if’ y los demás ‘for’.

6.

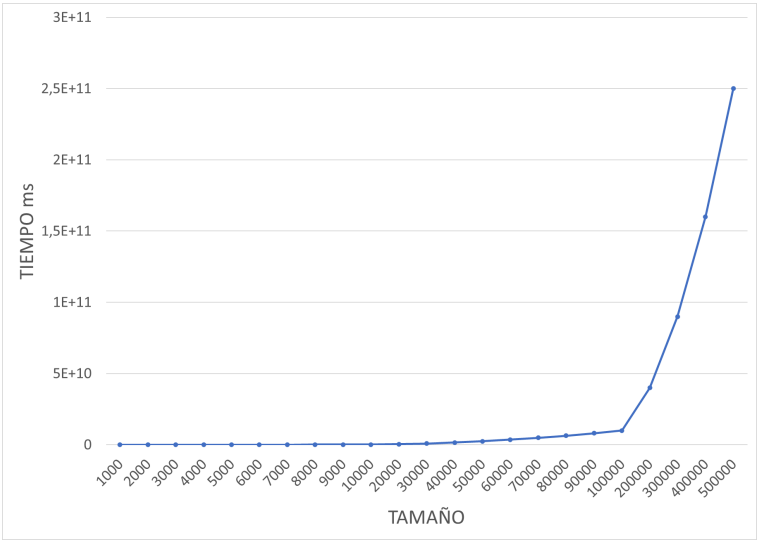
TAMAÑO	TIEMPO ms
1000	1467
2000	5800
3000	11217
4000	19859
5000	32.246
6000	46.779
7000	73.581
8000	78.903
9000	97.695
10000	116.558
20000	469.623
30000	674.944
40000	1.847.070
50000	2.758.390
60000	3.877.154
70000	4.624.629
80000	5.976.511
90000	7.718.738
100000	11.254.331
200000	28.050.625
300000	86.296.176
400000	149.572.950
500000	249.652.118



En esta gráfica podemos ver el tiempo (en microsegundos) de ejecución que conlleva este algoritmo para diferentes tamaños.

Del mismo modo, en esta otra gráfica podemos visualizar el gráfico para la eficiencia teórica calculada anteriormente, es decir, n^2. Asimismo en la siguiente tabla podemos ver el cálculo de la constante oculta.

TAMAÑO	TIEMPO ms	f(n)	k=T(n)/f(n)
1000	1467	1000000	0,001467
2000	5800	4000000	0,00145
3000	11217	9000000	0,00124633
4000	19859	16000000	0,00124119
5000	32.246	25000000	0,00128984
6000	46.779	36000000	0,00129942
7000	73.581	49000000	0,00150165
8000	78.903	64000000	0,00123286
9000	97.695	81000000	0,00120611
10000	116.558	100000000	0,00116558
20000	469.623	400000000	0,00117406
30000	674.944	900000000	0,00074994
40000	1.847.070	1600000000	0,00115442
50000	2.758.390	2500000000	0,00110336
60000	3.877.154	3600000000	0,00107699
70000	4.624.629	4900000000	0,0009438
80000	5.976.511	6400000000	0,00093383
90000	7.718.738	8100000000	0,00095293
100000	11.254.331	1E+10	0,00112543
200000	28.050.625	4E+10	0,00070127
300000	86.296.176	9E+10	0,00095885
400000	149.572.950	1,6E+11	0,00093483
500000	249.652.118	2,5E+11	0,00099861
		k=	0,00117765



Como conclusión, podemos decir que este algoritmo es eficiente para casos pequeños, pero es realmente lento en ocasiones en las que el vector sea de grandes dimensiones.

ELIMINAR REPETIDOS ORDENADOS

2. Algoritmo:

```
1  #include <iostream>
2  using namespace std;
3
4  int main(int argc, char ** argv){
5      //Comprobar que le pasamos 1 argumento
6      if (argc != 2) {
7          cout <<"\nError: No hay suficientes argumentos.\n\n";
8          return 0;
9      }
10
11     int TAMANIO=atoi(argv[1]);
12     int inicial[TAMANIO];
13     //Inicializar el vector con números aleatorios
14     for (int i=0;i<TAMANIO;i++){
15         cin >> inicial[i];
16     }
17
18     int final[TAMANIO];
19     int tam_final = 0;
20     for (int i=0;i<TAMANIO;i++){
21         if (inicial[i]!=inicial[i+1]){
22             final[tam_final]=inicial[i];
23             tam_final++;
24         }
25     }
26
27     //Salida del programa
28     for (int i=0; i<tam_final; i++){
29         cout << final[i] << " ";
30     }
31     cout << endl;
32 }
```

3. El tamaño del problema para este algoritmo depende de la variable TAMANIO, que es definida por el usuario en la ejecución, ya que de esta variable depende el número de iteraciones de los bucles for en las líneas 14 y 20, así como el tamaño máximo de tam_final, variable de la que a su vez depende el número de iteraciones del tercer bucle (línea 28).

4. El mejor caso para este algoritmo es un vector con todos los elementos iguales y el peor caso es un vector con todos los elementos diferentes. En el primer caso tan solo tendremos un entero en el vector final, mientras que para el segundo caso tendremos un vector con tantos elementos como el inicial. Esto supone la copia e impresión por pantalla de un mayor número de enteros. Es decir, el if de la línea 21 se ejecuta más veces y el for de la línea 28 tiene más iteraciones, por lo que el tiempo de ejecución aumenta.

Ejemplo para el mejor caso:

Vector inicial = {1,1,1,1,1,1,1,1,1,1}

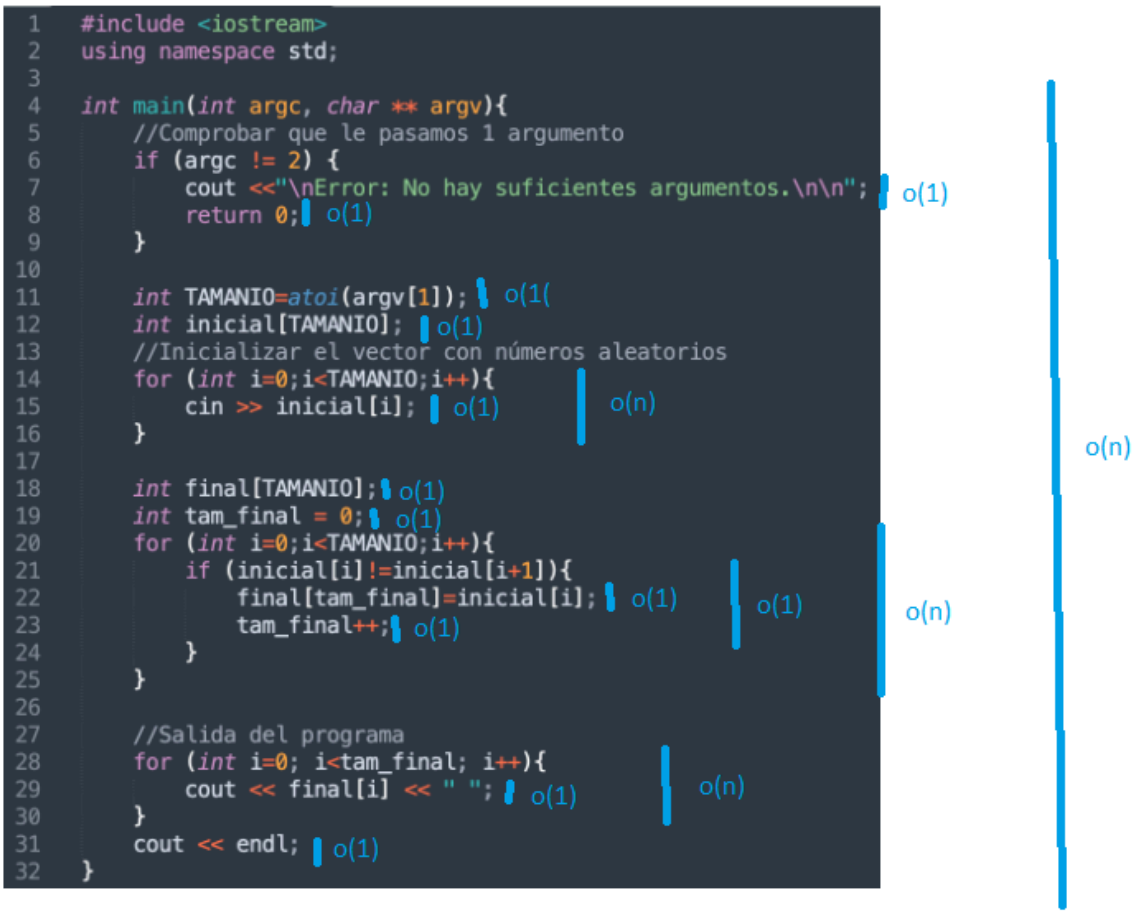
Vector final = {1}

Ejemplo para el peor caso:

Vector inicial = {1,2,3,4,5,6,7,8,9,10}

Vector final = {1,2,3,4,5,6,7,8,9,10}

5. Análisis teórico de la eficiencia del algoritmo:

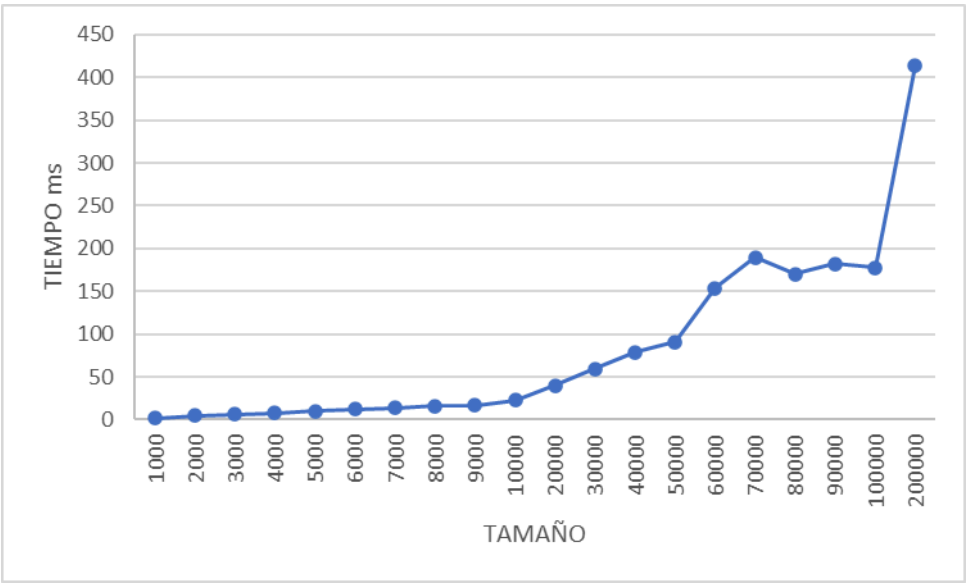


Por tanto, la eficiencia teórica de este algoritmo es O(n).

6.

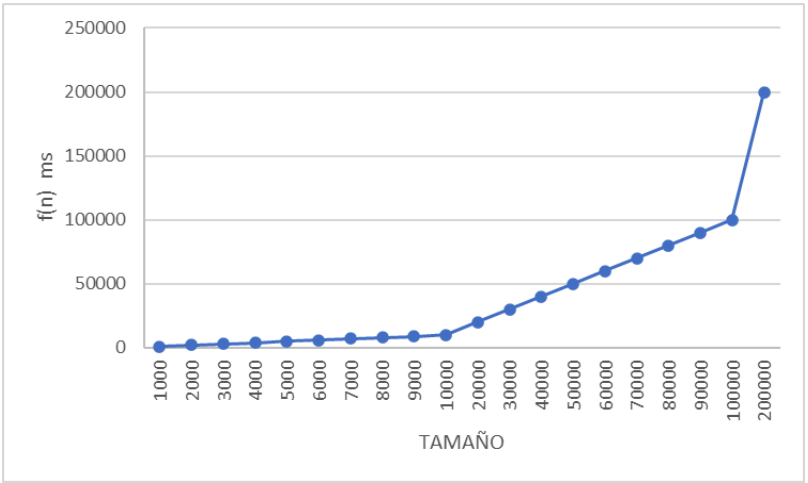
En la siguiente gráfica se muestra el tiempo real del algoritmo en microsegundos con respecto al tamaño dado y su tiempo de ejecución

TAMAÑO	TIEMPO ms
1000	2
2000	5
3000	6
4000	8
5000	10
6000	12
7000	14
8000	16
9000	17
10000	23
20000	40
30000	59
40000	79
50000	91
60000	154
70000	190
80000	170
90000	182
100000	178
200000	414



Siguiendo el mismo procedimiento que en el algoritmo anterior, se usa la eficiencia teórica (n) para construir la siguiente gráfica y calcular la constante oculta.

TAMAÑO	TIEMPO ms	f(n) ms	K=T(n)/f(n)
1000	2	1000	0,002
2000	5	2000	0,0025
3000	6	3000	0,002
4000	8	4000	0,002
5000	10	5000	0,002
6000	12	6000	0,002
7000	14	7000	0,002
8000	16	8000	0,002
9000	17	9000	0,001888888889
10000	23	10000	0,0023
20000	40	20000	0,002
30000	59	30000	0,001966666667
40000	79	40000	0,001975
50000	91	50000	0,00182
60000	154	60000	0,002566666667
70000	190	70000	0,002714285714
80000	170	80000	0,002125
90000	182	90000	0,002022222222
100000	178	100000	0,00178
200000	414	200000	0,00207
		k=	0,002086436508



De nuevo, al igual que en el primer algoritmo, se observa que en casos donde el tamaño es menor, el algoritmo es más eficiente. Al aumentar el tamaño, también lo hace el tiempo de ejecución, hasta llegar a instantes donde el tiempo de espera es significante.

ALGORITMOS RECURSIVOS

HANOI

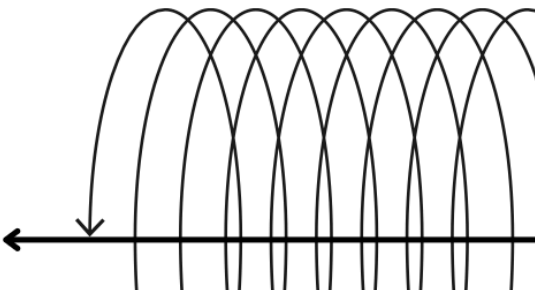
EFICIENCIA TEÓRICA

```
1  /**
2  Se trata del problema clásico de las torres de Hanoi.
3  Se tienen 3 barras, y hay que mover M anillos de la primera barra
4  a la segunda. Solo se puede mover un anillo en cada movimiento,
5  y ningún anillo de tamaño mayor puede ponerse sobre otro de tamaño
6  menor.
7  Los valores de "i" y "j" sólo pueden tomar los valores {1, 2, 3}
8
9  Si M=3, la llamada sería hanoi(3, 1, 2)
10
11 */
12 void hanoi (int M, int i, int j)
13 {
14   if (M > 0)
15     {
16       hanoi(M-1, i, 6-i-j);   o(2^n)
17       cout << i << " -> " << j << endl;   o(1)
18       hanoi (M-1, 6-i-j, j);   o(2^n)
19   }
20 }
```

o(2^n)

Tenemos t(n) que se considera a la cantidad de movimientos a realizar para n discos. Para hallar la ecuación hay que aplicar una hipótesis que apoye la ecuación a demostrar:

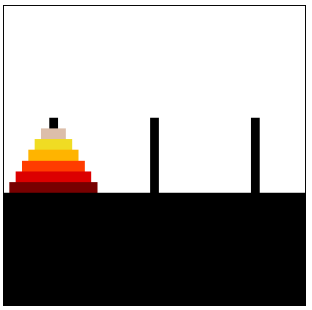
- Suponemos la hipótesis que para mover n discos se tiene que mover los (n-1) discos menores a la torre auxiliar y uno más, que es el disco n, a la torre destino. Además sabemos que esos discos deben moverse el doble de veces pues para 2 discos, hay que mover el disco pequeño dos veces, una para pasarlo a la torre auxiliar y otra para ponerlo en la cima de la torre destino.
- Para tres discos es exactamente lo anterior, debemos mover los 2 discos menores 2 veces, una para quitarlos de encima (despejar) del disco de abajo (más grande), 1 movimiento para mover el disco grande a la torre destino, y otra vez volver a mover los dos discos más pequeños encima del disco grande (en la torre destino).
- Sabemos que para un disco se necesita un movimiento, para dos discos tres movimientos y para tres discos se necesitan siete movimientos.



Otra forma de razonarlo es que observando el código, se ve claramente que la función se llama a sí misma dos veces para un caso un número más pequeño, es decir, 2t(n-1). Si a este tiempo le sumamos la eficiencia del caso base, O(1), también podemos llegar al resultado que buscamos.

Por lo que tenemos la siguiente ecuación:

$$t(n)=2t(n-1)+1,\ n>=1$$



Con la que obtenemos la ecuación de recurrencia:

$$(x-2)(x-1) = 0$$

$$tn = c1 \cdot 2^n + c2 \cdot 1^n = c1 \cdot 2^n + c2$$

Orden=$O(2^n)$

HEAPSORT

EFICIENCIA TEÓRICA

Considerando a "n" el parámetro del que depende el tamaño del problema, obtenemos la ecuación siguiente para el tiempo de ejecución del algoritmo: T(n) = 2T(n/2) +n, con la condición inicial T(1)=2. Vemos esto a continuación.

Analizando teóricamente la función **insertarEnPos** podemos observar que posee un tiempo T(n) = T(n/2) +1. Esto se debe a que el caso base tiene eficiencia de O(1), mientras que en el resto de casos, donde continúa la recursividad la eficiencia es de O(log2(n)), ya que en cada iteración se analiza la mitad del array.

Respecto a la eficiencia de **reestructurarRaiz** podemos decir que al igual que en la función anterior, el caso base (2*pos+1 >= tamapo) tiene eficiencia O(1). En el caso estándar, se llama a la función recursivamente con pos = 2*pos+1; por lo que de nuevo en este caso el tiempo de ejecución para el algoritmo es T(n) = T(n/2) +1 y su eficiencia O(log2(n)).

Por último analizamos la función principal HeapSort, en la que llamamos n (tamaño del vector) veces a insertarEnPos y que posteriormente, llamamos de nuevo n veces a reestructurarRaiz. El resto de operaciones tienen eficiencia O(1). Por este motivo, el tiempo de ejecución queda T(n) = T(n/2) + T(n/2) +1, es decir, T(n) = 2T(n/2)+n.

A partir de la ecuación T(n) = 2T(n/2)+n obtenemos la ecuación en recurrencias. Comenzamos realizando un cambio de variable como n=2^k

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

$$tk - 2(k-1) = 2^k$$

Obtenemos la ecuación característica:

$$(x-2)^2 = 0$$

$$tk = c1 \cdot 2^k + c2 \cdot k \cdot 2^k$$

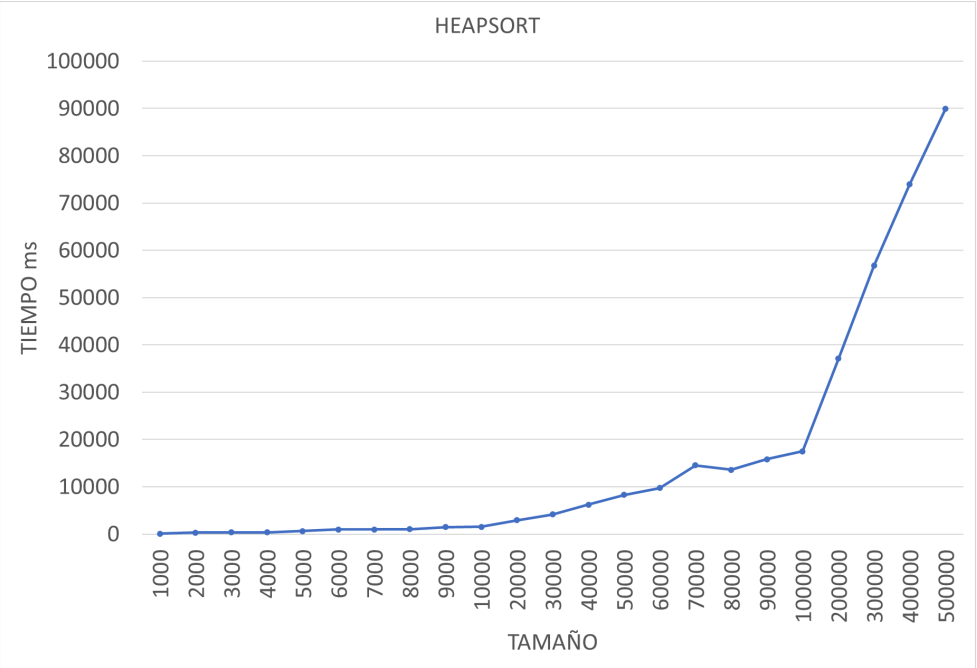
Deshaciendo el cambio de variable:

$$T(n) = c1 \cdot n + c2 \cdot n \cdot \log2(n)$$

Por lo que la eficiencia del algoritmo es de orden O(nlog(n)).

EFICIENCIA PRÁCTICA

En la siguiente gráfica podemos ver los tiempos de ejecución para heapSort con diferentes tamaños. En concreto, desde el tamaño 1000-10000 vamos pasando las comprobaciones de 1000 en 1000, desde 10000-100000 vamos pasando las comprobaciones de 10000 en 100000, y por último desde 100000-500000 vamos pasando las comprobaciones de 100000 en 100000. Podemos ver el tiempo en el eje Y medido en microsegundos.



EFICIENCIA HÍBRIDA

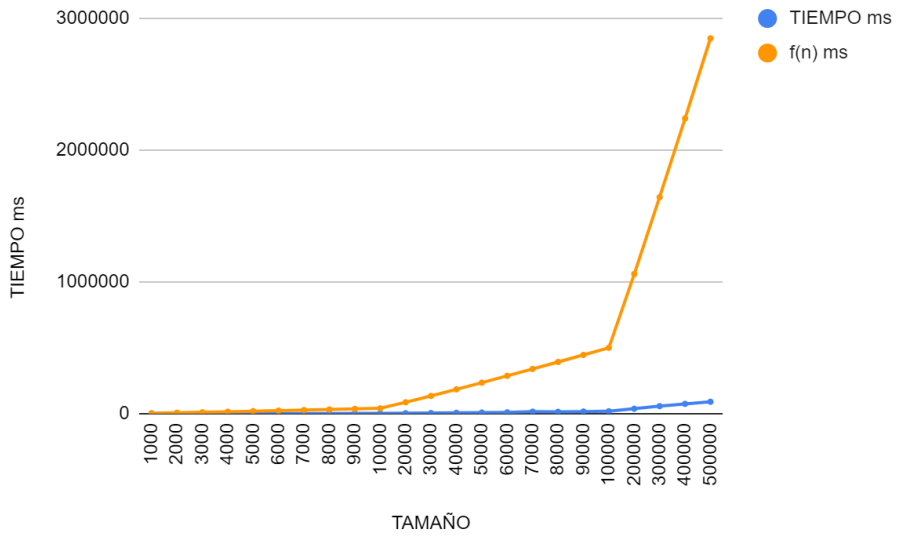
En la siguiente gráfica se muestra el tiempo real de ejecución del algoritmo junto con el teórico, ambos medidos en microsegundos. Los datos utilizados son los que se muestran en la tabla.

La constante oculta 'k' se ha calculado con la fórmula:

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

Y su valor promedio es **k=0,2342749**

TAMAÑO	TIEMPO ms	f(n) ms	K=T(n)/f(n)
1000	95	3000	0,33333333
2000	285	6602,05999	0,30293575
3000	347	10431,3638	0,28759423
4000	375	14408,24	0,27761892
5000	671	18494,85	0,27034553
6000	973	22668,9075	0,26467972
7000	979	26915,6863	0,26007139
8000	1.026	31224,7199	0,25620726
9000	1.496	35588,1826	0,25289294
10000	1.523	40000	0,25
20000	2.926	86020,5999	0,23250245
30000	4.154	134313,638	0,22335781
40000	6.250	184082,4	0,21729399
50000	8.270	234948,5	0,21281259
60000	9.742	286689,075	0,20928597
70000	14.549	339156,863	0,20639417
80000	13.604	392247,199	0,20395302
90000	15.822	445881,826	0,2018472
100000	17.530	500000	0,2
200000	37.102	1060206	0,18864258
300000	56.743	1643136,38	0,18257766
400000	74.013	2240824	0,17850576
500000	89.994	2849485	0,1754703
		k=	0,2342749

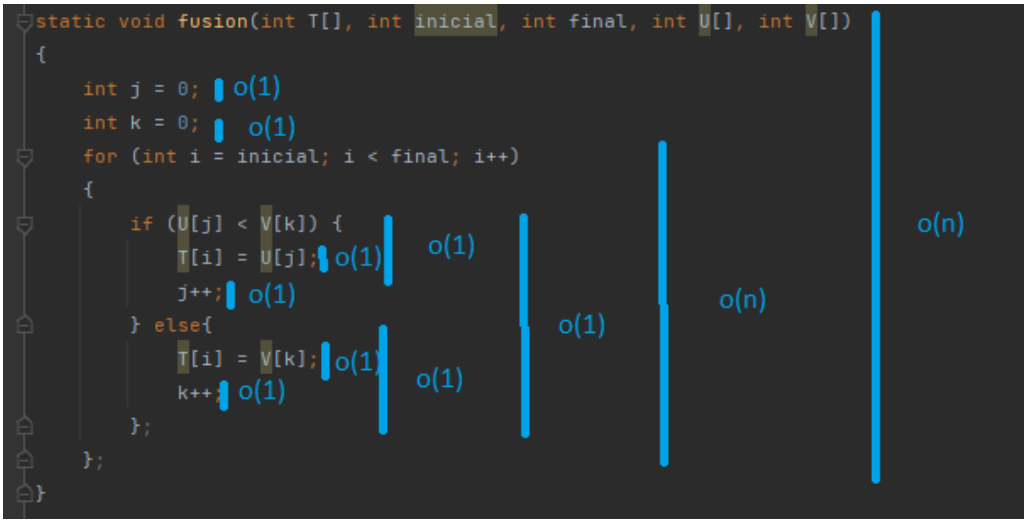


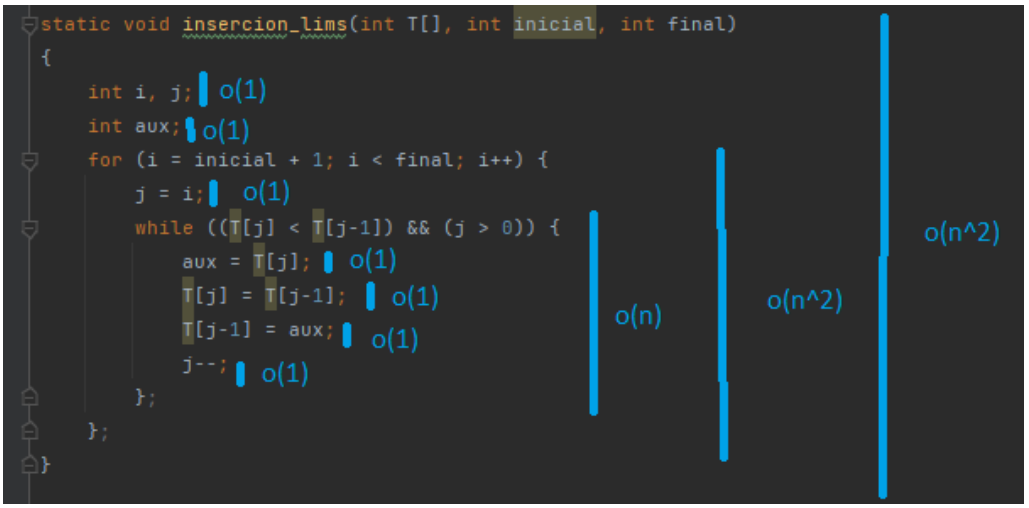
MERGESORT

EFICIENCIA TEÓRICA

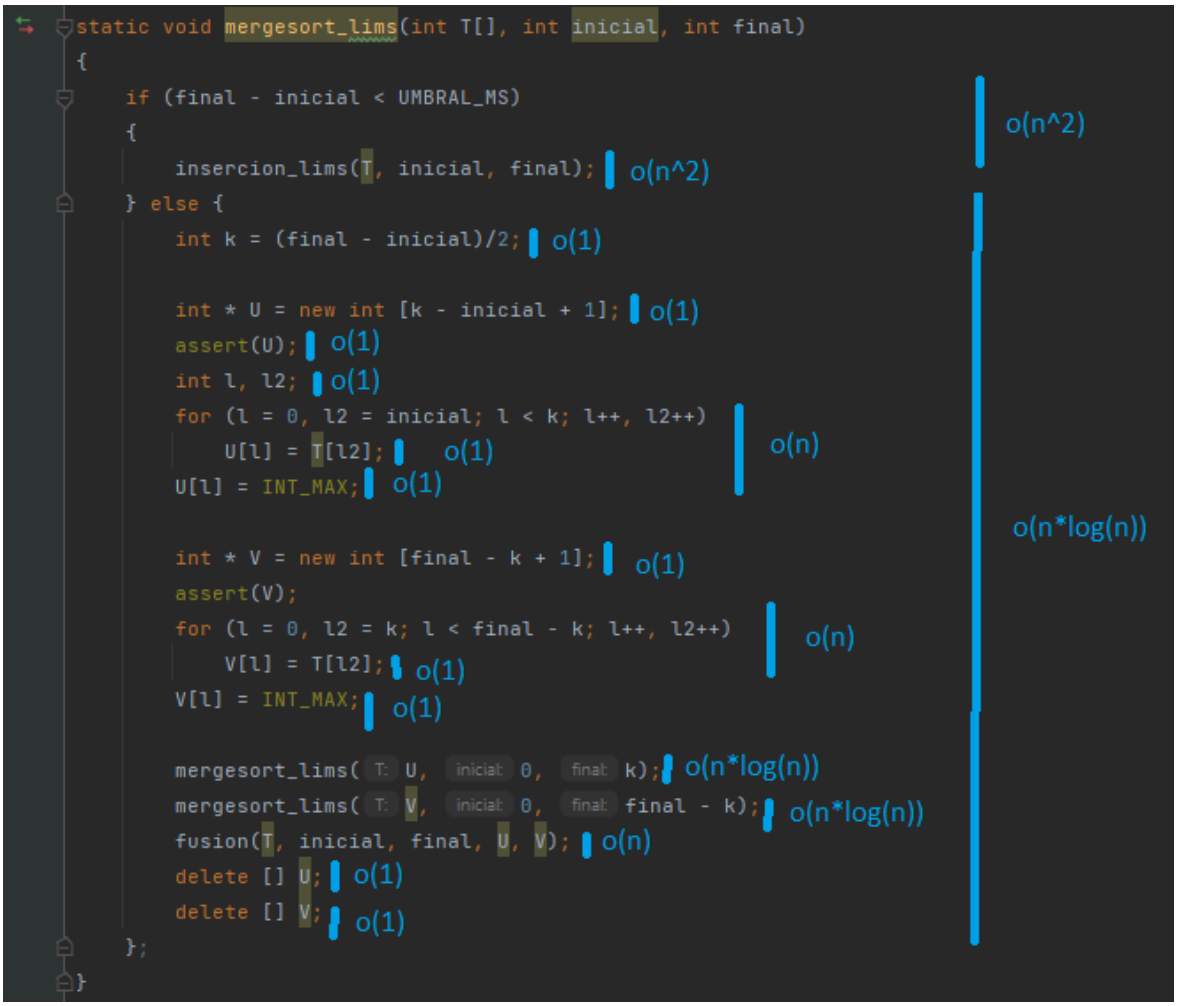
El funcionamiento de mergeSort consiste en dividir el vector del que se parte en dos partes iguales, es decir, si el tamaño del array es k, el tamaño de cada uno de estos subproblemas (nuevas llamadas a mergeSort) serán de tamaño k/2, con orden O(log2(n)).

En las imágenes siguientes podemos observar la eficiencia teórica de mergeSort:





La eficiencia de fusion es O(n) y la eficiencia de insercion_lims es O(n).



Las dos llamadas recursivas se hacen sobre dos vectores que constituyen cada uno de ellos la mitad del vector inicial, por tanto, el tiempo de ejecución de mergeSort será $T(n) = 2 \cdot T(n/2) + n$. El resto de operaciones realizadas en esta función poseen eficiencia O(1).

A partir de esta ecuación, podemos obtener la ecuación en recurrencias:

Haciendo un cambio de variable n=2^k:

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

Resolvemos la recurrencia:

$$(x-2)(x-2) = (x-2)^2 = 0$$

$$tk = c \cdot 2k + c2 \cdot k \cdot 2k$$

Deshacemos el cambio de variable:

$$tn = c1 \cdot n + c2 \cdot \log2(n) \cdot n$$

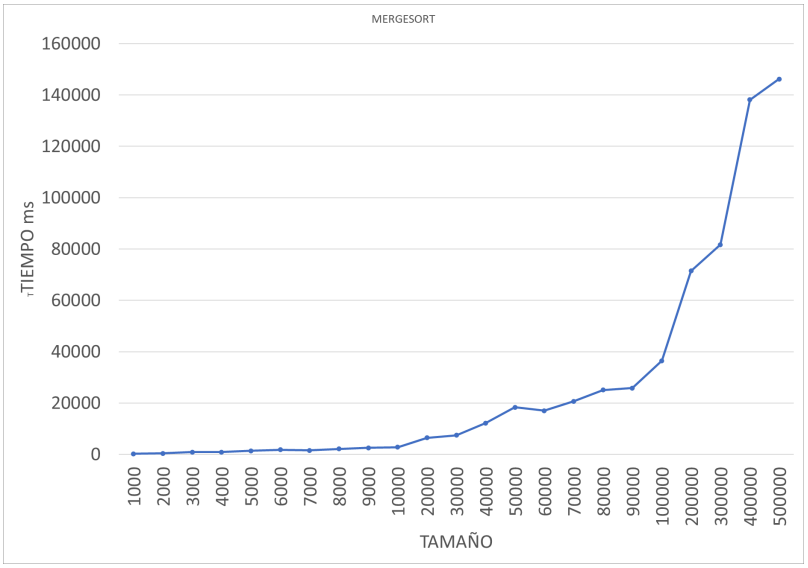
Por ello, el orden de eficiencia de mergeSort es O(n·log2(n)).

Hay que tener en cuenta que, si el vector es muy pequeño, el algoritmo sería O(n^2), mientras que en los demás casos sería de O(n·log(n)).

EFICIENCIA PRÁCTICA

Con el objetivo de comprobar experimentalmente la eficiencia de este algoritmo, hemos realizado varias mediciones del tiempo de ejecución que se requiere para distintos tamaños.

Desde el tamaño 1000-10000 vamos pasando las comprobaciones de 1000 en 1000, desde 10000-100000 vamos pasando las comprobaciones de 10000 en 100000, y por último desde 100000-500000 vamos pasando las comprobaciones de 100000 en 100000.

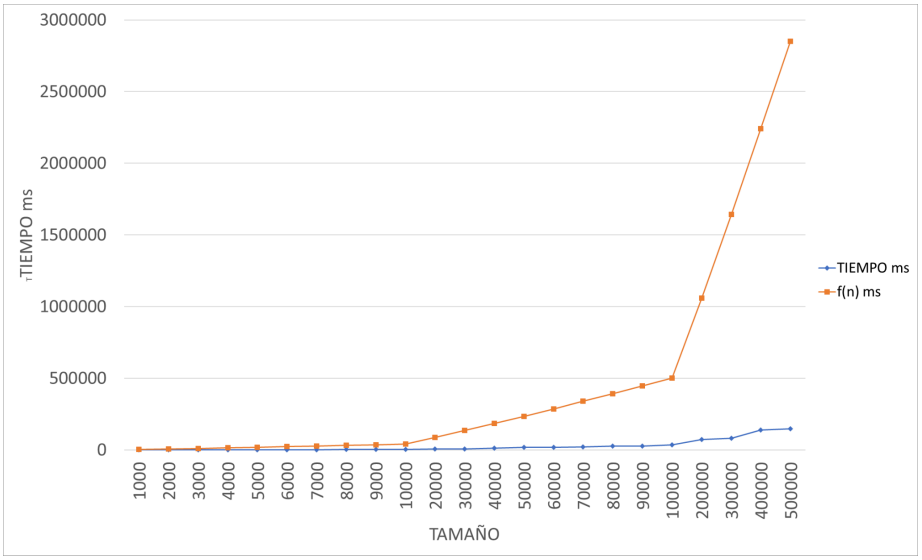


TAMAÑO	TIEMPO ms
1000	95
2000	285
3000	347
4000	375
5000	671
6000	973
7000	979
8000	1.026
9000	1.496
10000	1.523
20000	2.926
30000	4.154
40000	6.250
50000	8.270
60000	9.742
70000	14.549
80000	13.604
90000	15.822
100000	17.530
200000	37.102
300000	56.743
400000	74.013
500000	89.994

En esta tabla y este gráfico el tiempo está medido en microsegundos.

EFICIENCIA HÍBRIDA

En la gráfica se muestra el tiempo real del algoritmo junto con el tiempo de ejecución teórico con la constante calculada. En la tabla se muestran los resultados empíricos de la ejecución del programa para cada tamaño. El tiempo tanto en la tabla como en la gráfica se encuentra expresado en microsegundos.



TAMAÑO	TIEMPO ms	f(n) ms	k=T(n)/f(n)
1000	197	3000	0,065666667
2000	432	6602,05999	0,065434122
3000	854	10431,3638	0,08186849
4000	924	14408,24	0,06412997
5000	1.388	18494,85	0,075047919
6000	1.774	22668,9075	0,078256969
7000	1.564	26915,6863	0,05810738
8000	2.148	31224,7199	0,06879165
9000	2.549	35588,1826	0,071624899
10000	2.775	40000	0,069375
20000	6.457	86020,5999	0,075063415
30000	7.455	134313,638	0,055504416
40000	12.193	184082,4	0,066236642
50000	18.316	234948,5	0,07795751
60000	17.009	286689,075	0,059329083
70000	20.706	339156,863	0,061051396
80000	25.075	392247,199	0,063926524
90000	25.816	445881,826	0,057898749
100000	36.481	500000	0,072962
200000	71.462	1060206	0,067403882
300000	81.664	1643136,38	0,049700074
400000	138.133	2240824	0,061643842
500000	146.220	2849485	0,051314536
		K=	0,069013415

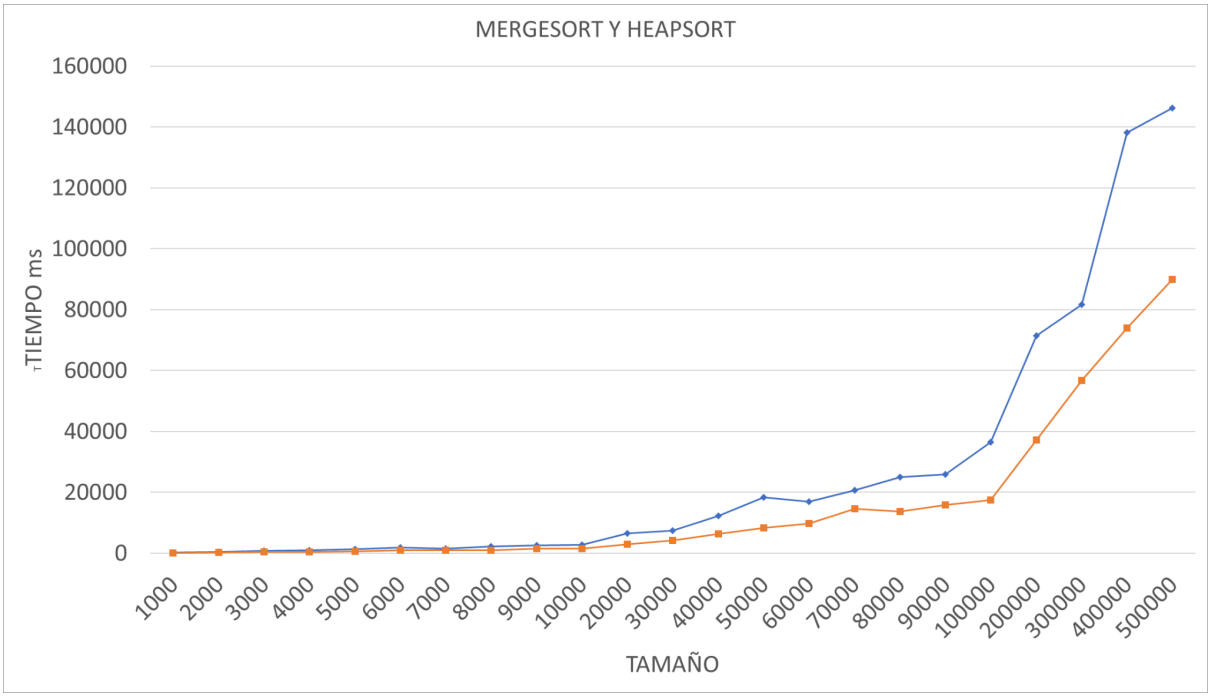
Por lo que podemos decir que en promedio, la constante oculta, **k=0,069013415**. Esta constante ha sido calculada como T(n)/f(n), es decir, t. ejecución / orden, como se especifica en las instrucciones de la práctica.

COMPARACIÓN: HEAPSORT Y MERGESORT

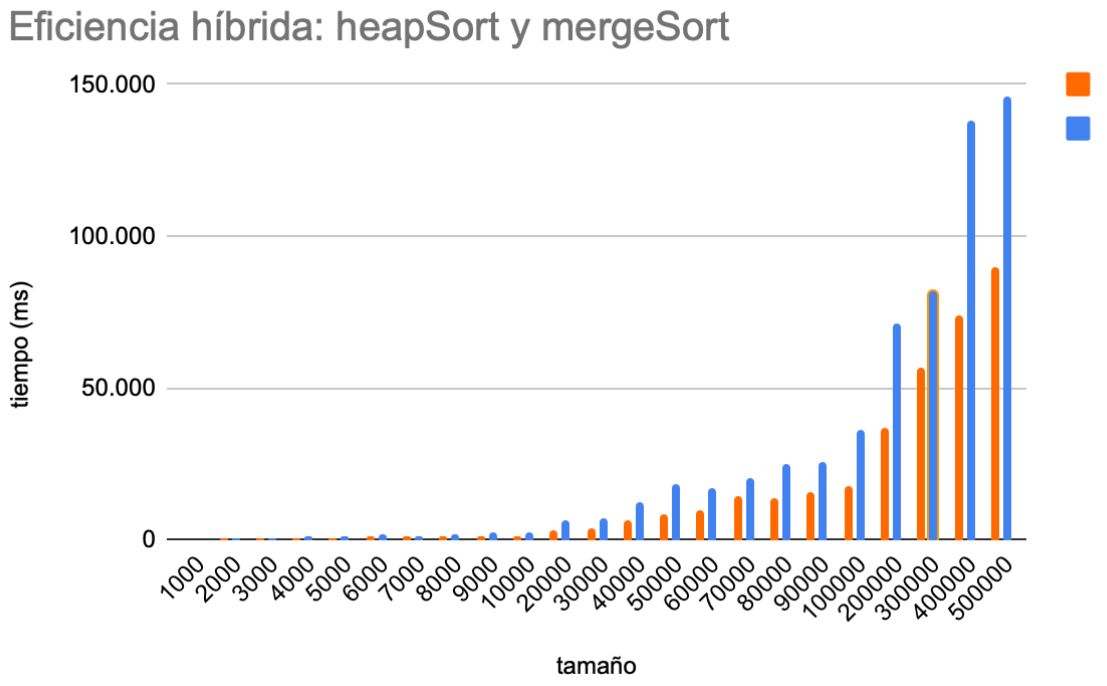
A continuación, pasamos a realizar una comparación entre los dos algoritmos de ordenación analizados de forma teórica, práctica e híbrida con anterioridad.

En cuanto a la comparación teórica de la eficiencia de los algoritmos, tenemos que tanto el orden de heapSort como el de MergeSort es O(n·log(n)). Con solo esta información no podemos sacar en claro cuál es mejor para casos grandes, pero sabemos que para casos que no sean excesivamente grandes el rendimiento de ambos algoritmos va a ser similar. Claramente, si tratamos de representar sus órdenes en una gráfica obtendremos la misma representación, por lo que no vamos a hacer este análisis.

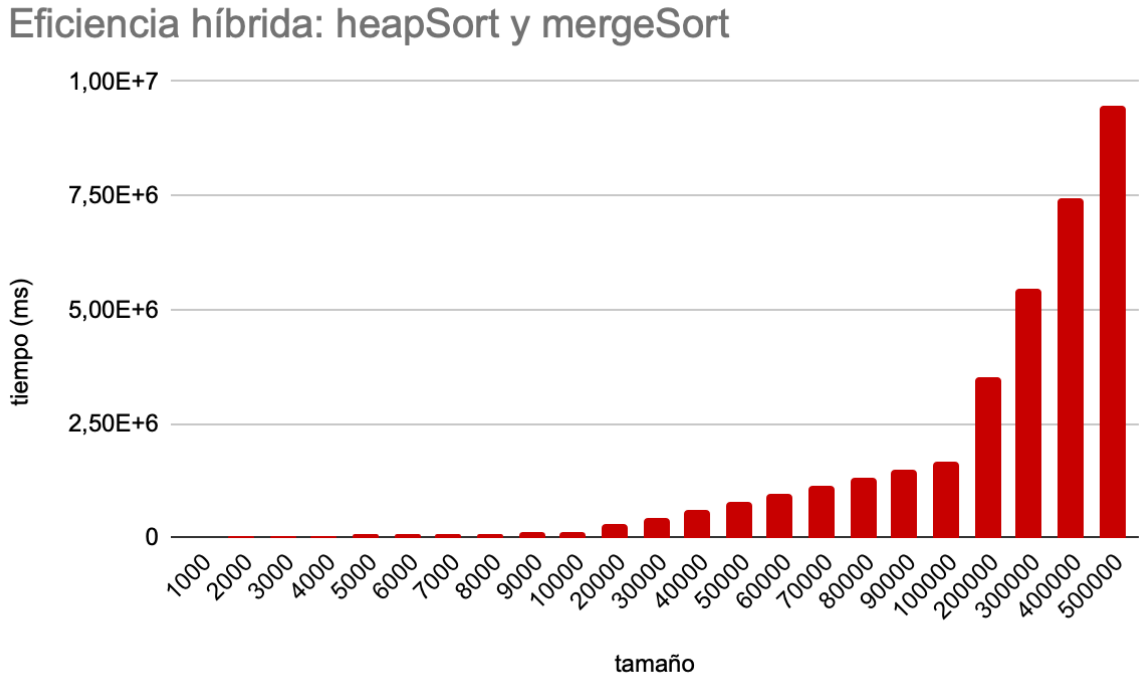
En la siguiente gráfica se puede observar el tiempo de ejecución en microsegundos (eje y) para distintos tamaños (eje x) tanto de heapSort (en naranja) como de mergeSort (en azul). A simple vista podemos ver que el algoritmo heapSort es más rápido que el otro, y que la diferencia en su rendimiento va incrementando con el tamaño del problema.



Respecto a la eficiencia híbrida de los algoritmos, en los siguientes dos gráficos podemos ver que la eficiencia práctica y la eficiencia teórica tienen una gráfica con forma parecida, pese a que los tiempos de ejecución teóricos son mucho mayores que los prácticos, ya que el tiempo teórico contempla el peor caso, que no va a ser casi nunca el caso real. Respecto a sus constantes, en los apartados anteriores hemos obtenido que sus constantes ocultas son 0,2342749 para heapSort y 0,069013415 para mergeSort. Con esto tenemos que mergeSort es más eficiente pero que heapSort es más rápido.



Tiempo de ejecución de heapSort en naranja y de mergeSort en azul en microsegundos.



Tiempos de ejecución en micro segundos teóricos según el orden O(nlog(n)) para ambos algoritmos.

E J E C U C I Ó N D E F I C H E R O S

Con la finalidad de realizar análisis empíricos de la eficiencia de los distintos algoritmos vistos a lo largo de la práctica, hemos seguido una serie de pasos comunes.

En primer lugar, hemos desarrollado/conseguido el código en lenguaje C++ de cada algoritmo. Este código se ha transcrito a un IDE, en nuestro caso CLion, con el que hemos compilado y ejecutado.

Por otro lado, hemos necesitado medir el tiempo que requiere cada algoritmo. Para ello hemos usado la biblioteca *chrono*, la cual nos permite medir la hora con precisión (microsegundos).

Como se indicaba en el guión, al principio de cada archivo hemos incluido la biblioteca con:

```
#include<chrono>
```

Antes de empezar la ejecución del algoritmo en sí, hemos declarado dos variables que nos sirven para almacenar los tiempos en los que inicia (t0) y finaliza (tf) la ejecución:

```
chrono::time_point<std::chrono::high_resolution_clock> t0, tf;
```

Damos valores a estas variables justo antes y después de la ejecución del algoritmo:

```
// Comenzamos

t0= std::chrono::high_resolution_clock::now();

// Algoritmo

// Terminamos

tf= std::chrono::high_resolution_clock::now();
```

Por último, calculamos el tiempo total que se ha necesitado para completar el proceso:

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

Por último, imprimimos la duración por terminal, lo que nos sirve de muestra.

Este proceso se ha repetido para cada tamaño que se ha querido experimentar y para cada algoritmo. Los resultados se han agrupado en una hoja de cálculo (Excel en nuestro caso). Con ellos se han elaborado tablas a partir de las cuales

hemos podido obtener tanto la constante oculta (k) como las gráficas comparativas.

En el caso de los dos primeros algoritmos, elaborados por nosotras, en el primero hemos usado la función rand para generar números aleatorios como elementos del vector, mientras que en el segundo hemos usado un bucle for que ha generado números desde el 0 de esta forma:

```
int num = 0;
for (int i=0; i<TAMANIO; i++){
        inicial[i] = num/2;
        num++;
}
```

Siendo TAMANIO el número de elementos que posee el vector e inicial el vector con elementos repetidos.