UNIVERSIDAD DE GRANADA

Algorítmica

Curso 2023

Practica 1:

Análisis de Eficiencia en Algoritmos

Amador Carmona Méndez

Miguel Ángel López Sánchez

Analisis de la eficiencia de los algoritmos iterativos:

1. Algoritmo “eliminarRepetidos”: se introduce un vector y un tamaño total del vector.

Este algoritmo tiene una eficiencia teorica o(n²) debido a que se hace una busqueda con dos bucles for de tamaño n, por lo que en el peor de los casos se hacen n² iteraciones y en el mejor de los casos, n iteraciones.
Si se hicieran n², significa que todos los elementos del vector son iguales, por lo que se completan las n² iteraciones, por lo que significa que es el peor caso. Si se hicieran n iteraciones, significa que todos los elementos tienen valor distinto por lo que solo haria falta recorrerlo n veces, por lo que es el mejor caso.

Esto significa que el mejor caso es Ω(n) y el pero es O(n^2).

```
//Algoritmos creados por Amador Carmona Méndez y Miguel Ángel López Sánchez
void eliminarRepetidos(int *v, int n) {
    //Iterativo O(n^2)
    int i, j, k;
    for (i = 0; i < n; i++) {
        for (j = i + 1; j < n;) {
            if (v[j] == v[i]) {
                for (k = j; k < n; k++) {
                    v[k] = v[k + 1];
                }
                n--;
            } else {
                j++;
            }
        }
    }
    cout<<endl<<n<<endl;
    //Divide y venceras, Es mas lioso y acaba costando lo mismo o(n^2)
}
```

2. Algoritmo “eliminarRepetidosOrdenado”: funciona igual que el algoritmo anterior pero al estar ordenados podemos hacer una analisis del vector mas sencillo ya que solo tenemos que saber si el valor es distinto al que tenemos a los lados. Si la posicion en la que estamos y la siguiente son iguales entra en otro bucle y coloca ese valor al final del array y su tamaño maximo decrementa en uno.

La eficiencia de este algoritmo es la misma que la eficiencia del algoritmo anterior, el mejor caso seria un vector el cual no tenga elementos repetidos, y el peor caso seria un vector el cual tenga todos los elementos repetidos.
Esto significa que el mejor caso es Ω(n) y el pero es O(n^2).

Eficiencia Hibrida de los algoritmos iterativos:

eliminarRepetidos				
n	T(n)	T.E(n)=n^2	K	K*f(n)
1000	901	1000000	0,000901	0,7827953109
2000	3117	4000000	0,00077925	2,708072124
3000	7811	9000000	0,0008678888	7819,265037
4000	16023	16000000	0,0010014375	13900,91562
5000	20042	25000000	0,00080168	21720,18066
6000	28979	36000000	0,0008049722	31277,06015
7000	42271	49000000	0,0008626734	42571,55409
8000	57348	64000000	0,0008960625	55603,66248
9000	68371	81000000	0,0008440864	70373,38533
10000	89559	100000000	0,00089559	86880,72263
11000	99756	121000000	0,0008244297	105125,6744
12000	140037	144000000	0,0009724791	125108,2406
13000	148217	169000000	0,0008770236	146828,4212
14000	165433	196000000	0,0008440459	170286,2164
15000	193385	225000000	0,0008594888	195481,6259
		K=	0,0008688072	

eliminarRepetidosOrdenado				
n	T(n)	T.E(n)=n	K	f(n)*k
10000	126	10000	0,0126	73,59775225
20000	156	20000	0,0078	147,1955045
30000	215	30000	0,0071666666	220,7932567
40000	287	40000	0,007175	294,391009
50000	355	50000	0,0071	367,9887612
60000	415	60000	0,0069166666	441,5865135
70000	456	70000	0,0065142857	515,1842657
80000	516	80000	0,00645	588,782018
90000	834	90000	0,0092666666	662,3797702
100000	639	100000	0,00639	735,9775225
110000	734	110000	0,0066727272	809,5752747
120000	894	120000	0,00745	883,173027
130000	817	130000	0,0062846153	956,7707792
140000	889	140000	0,00635	1030,368531
150000	939	150000	0,00626	1103,966284
		K=	0,0073597752	

CURSO 2ºD

Amador Carmona Méndez

Miguel Ángel López Sánchez

Analisis de la eficiencia teorica y practica de los algoritmos recursivos

El Algoritmo de Hanoi:

```
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);
17       cout << i << " -> " << j << endl;
18       hanoi (M-1, 6-i-j, j);
19   }
20 }
```

Caso Base.

$T(1) = O(1)$ El

Caso General:

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

- $T(n) = 2T(n-1)+1 \Rightarrow T(n) - 2T(n-1) = 1$

→ 1º Resolvemos 'ELH': $x^2 - 2x$

Sacamos factor común 'x';

$P_H(x) = x - 2$ Raices: 0, 2

→ 2º Resolvemos 'ELNH'

Sabemos que $1 = b_1^n \cdot g_1(n)$; por lo que

$b_1 = 1$, $g_1(n) = 1$ con grado $d_1 = 0$

→ 3º Calculamos polinomio característico de ELNH

$p(x) = (x-2)\cdot(x-b_1)^{d_1+1} = (x-2)\cdot(x-1)^1 \Rightarrow$

$\Rightarrow p(x) = (x-2)\cdot(x-1)$

- Por ultimo calculamos 't_n'

$r = 0, 2, 1 \quad M = 1, 1, 1$

$t_n = 2^n + 1$

* Aplicando la regla de la suma sabemos que: $2^n + 1 = 2^n$

$t_n = 2^n \rightarrow O(2^n) \rightarrow$ Orden exponencial

CURSO 2ºD

Amador Carmona Méndez

Miguel Ángel López Sánchez

Algoritmo InsertaPos:

```
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);
    }
  }
}
```

→ O(1)
(1)
O(1)
O(1)
→ O((n-2)/2)
→ O((n-1)/2)
O((n-2)/2) o O((n-1)/2) ⇒ Padre mediocomún depende del caso es una u otra.
↳ O(padre-1)

- El peor caso sería: $\max\left(\frac{n-2}{2}, \frac{n-3}{2}\right)$

$$T(n) = T\left(\frac{n-2}{2}+1\right) + O(1)\cdot \log_2 n$$

$$T(n) = T(n/2) + \log_2 n$$

1 → Asignaciones los ciclos siempren son constantes

$\log_2 n$ → número máximo de veces que se puede llamar a la función

- Resolvemos la ecuación recurrente

$$T(n) = T(n/2) + \log_2 n$$

- Podemos resolverla por cambio de variable $\Longrightarrow n = 2^k$; $k = \log_2 n$

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

$$t_k - t_{k-1} = k \quad \text{← "RLNH"}$$

1. Ecuación característica

$$(x-1)\cdot(x-1) = 0 \Longrightarrow (x-1)^2 = 0$$

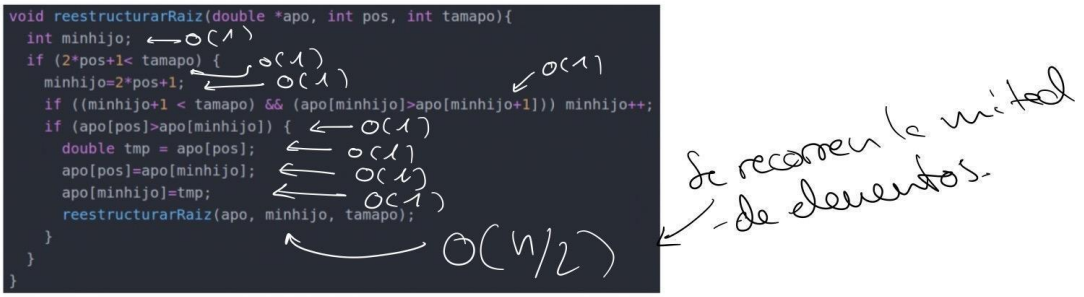
2. $t_k = c_1 \cdot 1^k + c_2 \cdot k \cdot 1^k = c_1 + k \cdot c_2$

3. Deshacemos el cambio:

$t_n = c_2 \cdot \log_2 n + c_1 \longrightarrow$ "$O((\log_2 n)^1)$"

CURSO 2ºD
Amador Carmona Méndez
Miguel Ángel López Sánchez

El Algoritmo ReestructuraRaiz:



· Este algoritmo es de eficiencia $O(\log n)$ ya que se trata de una busqueda binaria de un arbol, entonces todas las decisiones se toman o izquierda o derecha lo que quiere decir que hace $T(n/2)$ iteraciones, esto significa que su caso general es $T(n) = T(n/2)$ y su caso base es $T(1) = O(1)$.

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

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

↳ Cambio de variable: $2^k = n$; $k = \log_2 n$

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

$t_k - t_{k-1} = 0 \Rightarrow (x-1)\cdot(x-1) = (x-1)^2$

$t_k = c_1 \cdot 1^k + c_2 \cdot k \cdot 1^k \Rightarrow$ Deshacemos el cambio.

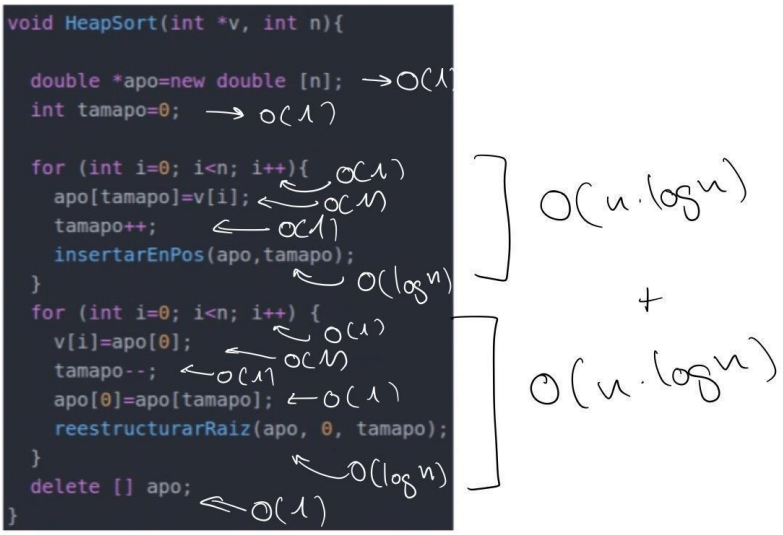
$t_n = c_2 \cdot \log_2 n + c_1 \Rightarrow O(\log_2 n)$

CURSO 2ºD

Amador Carmona Méndez

Miguel Ángel López Sánchez

El Algoritmo de HeapSort:



HeapSort es la suma de los dos algoritmos anteriores, esto significa que su eficiencia se calculará de la siguiente forma:

- Dando por secundarias las instrucciones O(1), ya que son 'cte', nos quedará que:

$$T(n) = O(n \cdot \log n) + O(n \cdot \log n) \rightarrow$$

$$\Rightarrow T(n) = n\log_2 n + n\log_2 n \Rightarrow$$

⇒ Aplicando la regla de la suma, sabemos que la eficiencia del algoritmo HeapSort es $T(n) = O(n \cdot \log n)$

También podríamos deducir que $n \cdot \log n + n \log n = 2(n \log n)$ y aplicando la regla de la suma se eliminan las 'cte' (el '2') y queda

$$O(n \cdot \log n)$$

CURSO 2ºD

Amador Carmona Méndez

Miguel Ángel López Sánchez

Algoritmos	*PeorCaso:*
Hanoi	O(2^n)
InsertaEnPos	O(logn)
ReestructuraRaiz	O(logn)
HeapSort	O(nlogn)

Eficiencia Practica e Hibrida:

Comparacion Entre HeapSort y MergeSort:

MergeSort				
n	T(n)	T.E(n)=nlogn	K	Tiempo teorico estimado Kpromedio*f(n)
1000	114	3000	0,038	119,447542855769
2000	328	6602,059991	0,049681463	262,866614583502
3000	447	10431,36376	0,04285154	415,333590087833
4000	596	14408,23997	0,041365219	573,676286910929
5000	684	18494,85002	0,036983268	736,388130191885
6000	834	22668,9075	0,036790481	902,581766791557
7000	1057	26915,68628	0,03927078	1071,67086347822
8000	1139	31224,7199	0,036477509	1243,23868930971
9000	1314	35588,18258	0,036922369	1416,97365482507
	K=	0,039815848		

CURSO 2ºD

Amador Carmona Méndez

Miguel Ángel López Sánchez

HeapSort				
n	T(n)	T.E(n)=n*log(n)	K	Tiempo teorico estimado Kpromedio*f(n)
1000	226702	3000	75,56733333	159676,501738311
2000	325466	6602,059991328	49,29764353	351397,947893903
3000	516127	10431,36376416	49,47838189	555214,558073561
4000	742363	14408,23996531	51,52350334	766885,784622371
5000	914325	18494,85002168	49,43673503	984397,650545532
6000	1145265	22668,9075023	50,52140249	1206563,94939897
7000	1508741	26915,6862801	56,05433888	1432600,87569739
8000	1512452	31224,71989594	48,43764828	1661951,34691387
9000	1733590	35588,18258495	48,71251843	1894198,83279657
	K=	53,22550057944		

CONCLUSION

Como hemos visto antes, ambos algoritmos tienen la misma eficiencia teorica, O(nlogn). Sin embargo, realizando las pruebas para calcular sus respectivas eficiencias practicas, resulta que el algoritmo MergeSort es mucho mas eficiente que el algoritmo HeapSort.

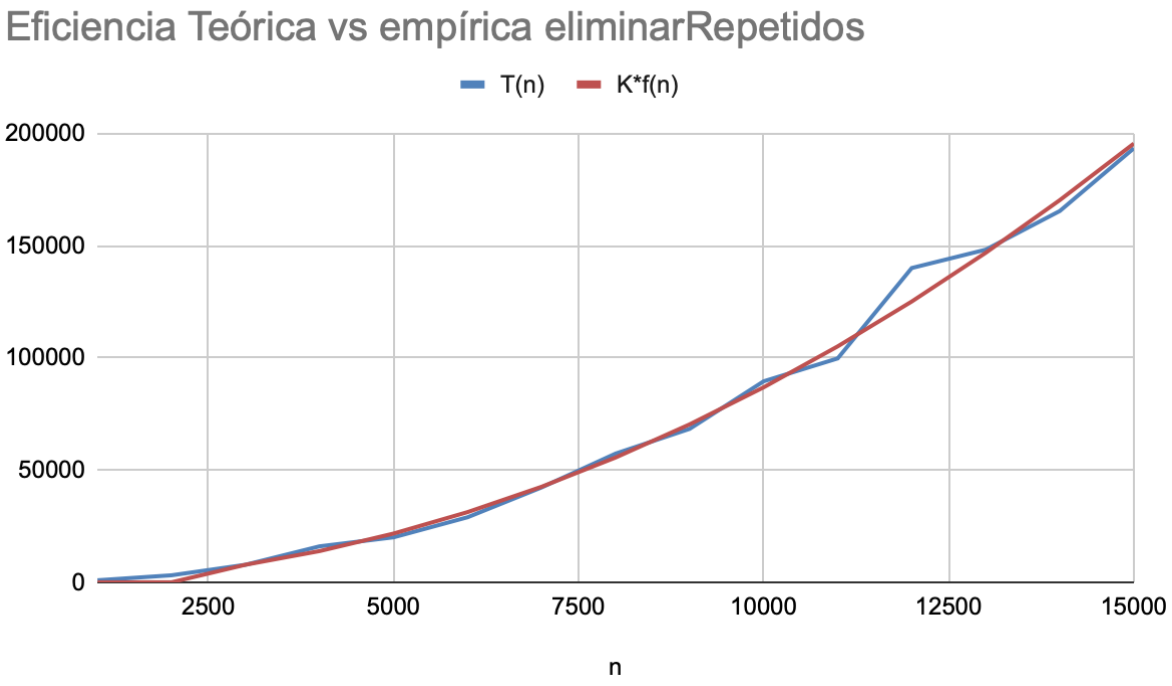
Ademas, como habiamos calculado antes, HeapSort tenia una eficiencia 2O(nlogn) lo que nos hacia pensar ya de antemano que seria mas lento que MergeSort que es O(nlogn), por lo que esto afecta mucho a la constante oculta k, la cual es bastante mas grande en Heap, que en Merge.

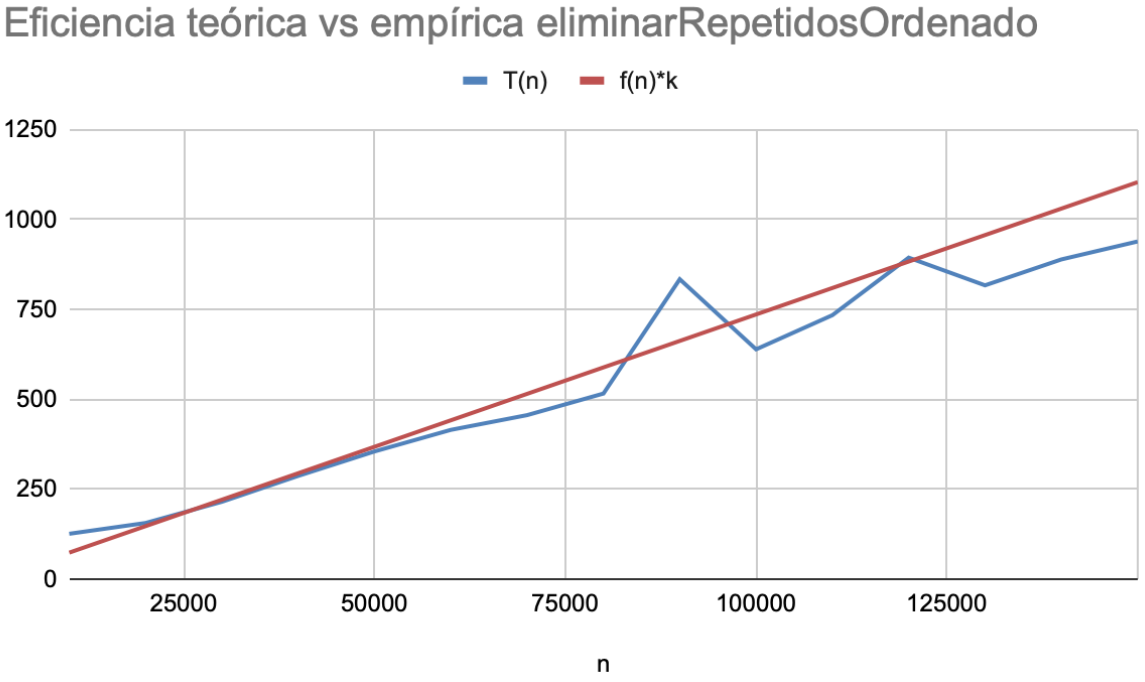
GRAFICAS DE LAS EFICIENCIAS COMPARADAS:

CURSO 2ºD

Amador Carmona Méndez

Miguel Ángel López Sánchez

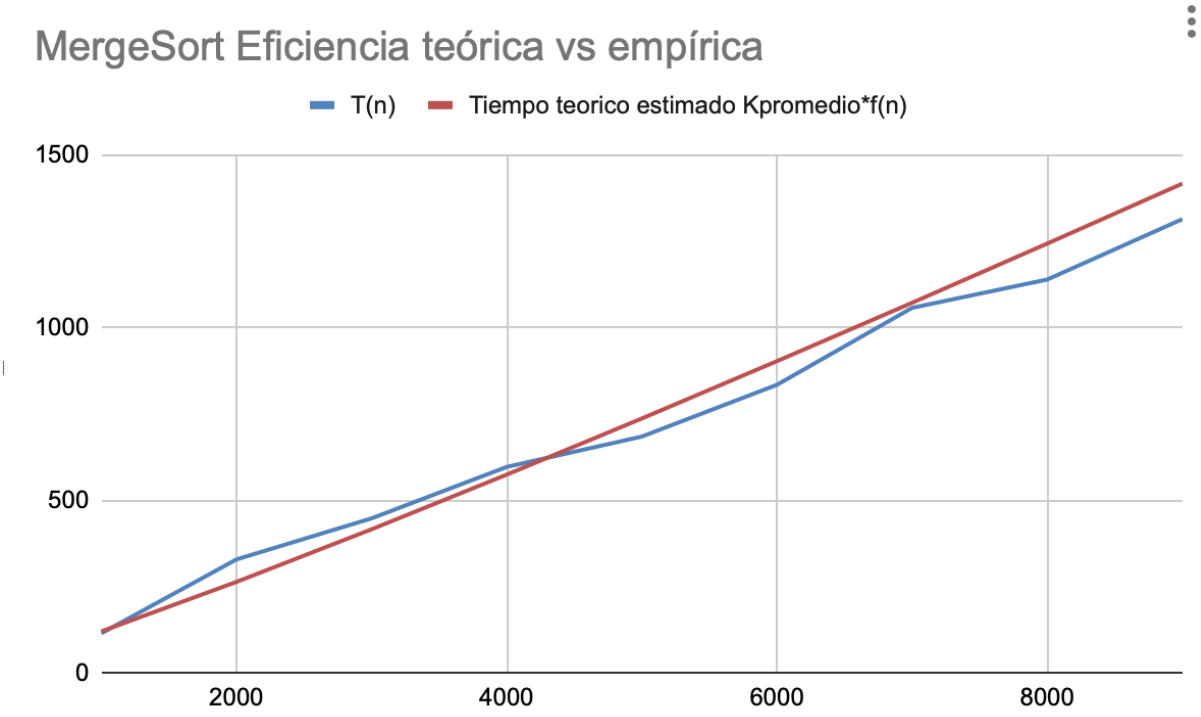




CURSO 2ºD

Amador Carmona Méndez

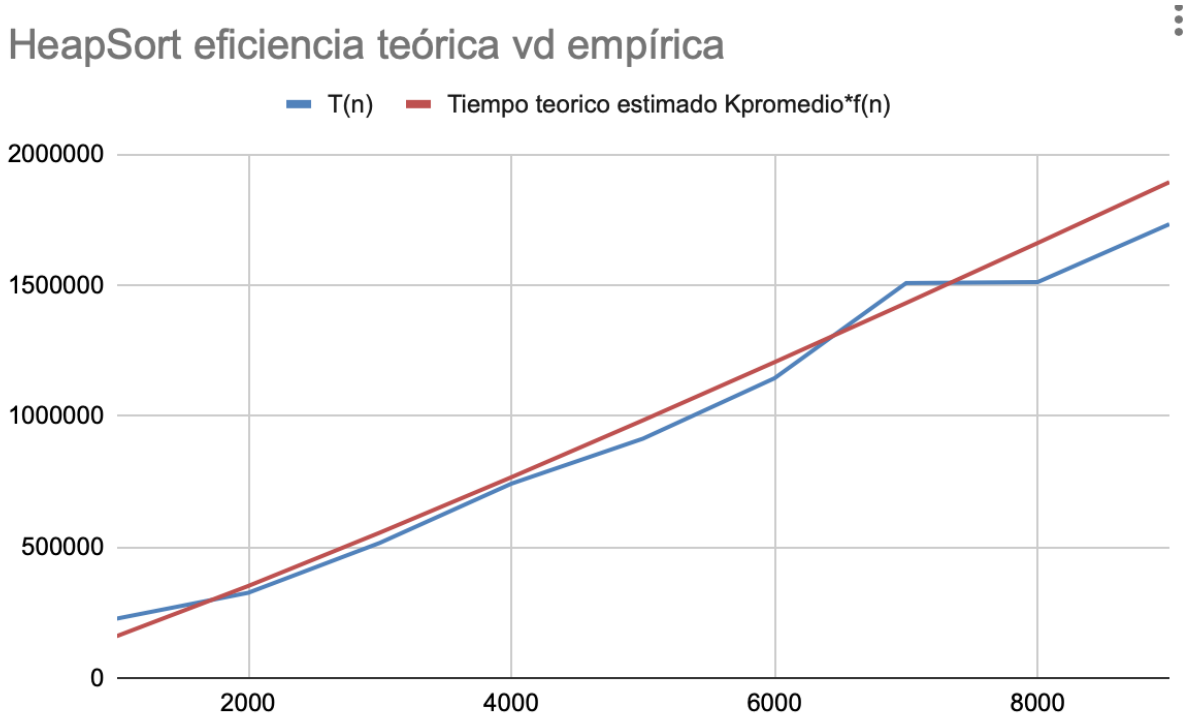
Miguel Ángel López Sánchez

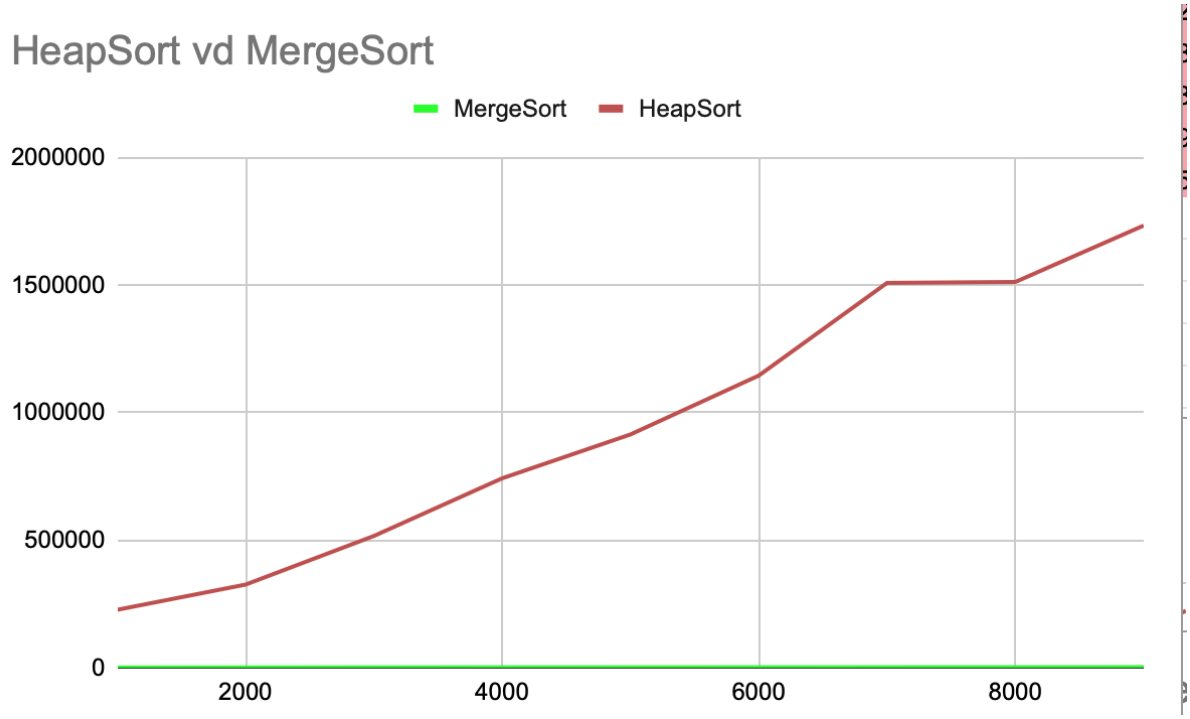


CURSO 2ºD

Amador Carmona Méndez

Miguel Ángel López Sánchez





CURSO 2ºD

Amador Carmona Méndez

Miguel Ángel López Sánchez