Carlos Quesada Pérez
Miguel Ángel López Sánchez
David Serrano Domínguez

Práctica 1

Ejercicio 1 - *MaximoyMinimoDyV*

Eficiencia Teórica

```
1   /**
2    * @brief Funcion que implementa un algortimo recursivo que devuelve el mínimo y
    máximo valor asociado a una estructura lineal (vector)
3    * @param a vector de componentes
4    * @param Cini posicion inicial
5    * @param Cfin posicion final
6    * @ret Devuelve una estructura con el maximo y el minimo conponente entre a[Cini] y
    a[Cfin]
7    * @pre Cini>=0
8    * @pre Cini<Cfin<N
9    */
10  template <typename T>
11  componentes<T> MaximoMinimoDyV(T *a, int Cini, int Cfin) {
12      // assert(Cfin<N);
13      componentes<T> salida;   O(1)
14
15      if(Cini<(Cfin-1)) {
16          int mitad = (Cini+Cfin)/2;
17          componentes<T> salida1 = MaximoMinimoDyV(a, Cini, mitad);    O(n/2)
18          componentes<T> salida2 = MaximoMinimoDyV(a, mitad+1, Cfin);  O(n/2)
19          salida.max = max(salida1.max, salida2.max);  O(1)
20          salida.min = min(salida1.min, salida2.min);  O(1)
21      } else if (Cini==Cfin) { // n=1   O(1)
22          salida.max = a[Cini];  O(1)
23          salida.min = a[Cini];  O(1)
24      } else { // Cini == (Cfin-1), n=2   O(1)
25          salida.max = max(a[Cini], a[Cfin]); // O(1) -
26          salida.min = min(a[Cini], a[Cfin]); // O(1) -
27      }
28
29      return salida;  O(1)
30  }
```

Caso Base : $n=1 \rightarrow T(1)=1$.

Caso General: $T(n) = T(n/2)+T(n/2)$ ← da constante se puede obviar porque vamos a estudar el comportamiento cuando $n \rightarrow \infty$

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

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

$T(2^k) = 2T(2^k/2) \Rightarrow T(2^k) - T(2^{k-1}) = 0$ (Recurrencia Homogénea)

$t_k - 2t_{k-1} = 0$

Ec. característica : $(x-2)=0 \Rightarrow t_k = c_1 \cdot 2^k$

Deshacemos cambio : $t_n = c_1 \cdot 2^{\log n} = c_1 \cdot n$

Aplicamos condiciones iniciales (caso base)

$T(1) = 1 \Rightarrow c_1 = 1$

Caso General : $t_n = n \Rightarrow O(n)$ orden LINEAL

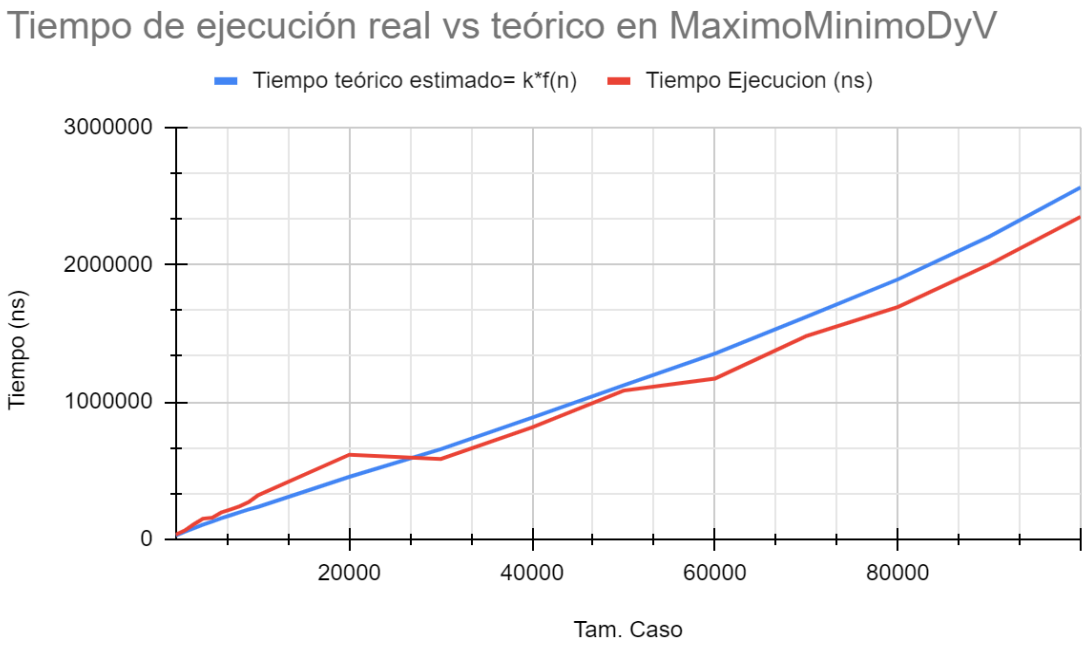
Algorítmica
2º GII
Grupo B3

Carlos Quesada Pérez
Miguel Ángel López Sánchez
David Serrano Domínguez

Eficiencia Práctica e Híbrida

Tam. Caso	Tiempo Ejecución (ns)	f(n) = n	k = T(n)/f(n)	Tiempo teórico estimado= k*f(n)
1000	33583	1000	33,583	27748,76116
2000	64099	2000	32,0495	54849,27356
3000	109925	3000	36,64166667	81557,12053
4000	150531	4000	37,63275	106517,8957
5000	156992	5000	31,3984	129708,8458
6000	196436	6000	32,73933333	153467,9626
7000	218026	7000	31,14657143	175465,2866
8000	240258	8000	30,03225	196790,1549
9000	271336	9000	30,14844444	217313,8138
10000	320517	10000	32,0517	236003,0066
20000	616078	20000	30,8039	455103,2146
30000	584789	30000	19,49296667	655825,691
40000	814447	40000	20,361175	886273,7032
50000	1082917	50000	21,65834	1120668,326
60000	1170130	60000	19,50216667	1352352,256
70000	1480127	70000	21,14467143	1620262,826
80000	1690359	80000	21,1294875	1891767,751
90000	2000357	90000	22,22618889	2203767,001
100000	2348395	100000	23,48395	2561635,558

K promedio = 27,74876116



Carlos Quesada Pérez
Miguel Ángel López Sánchez
David Serrano Domínguez

Ejercicio 2a - *insertarEnPos*

Eficiencia Teórica

```
 7  /**
 8   * @brief Función que reestructura el vector con forma de apo
 9   * @param apo vector sobre el que trabaja la funcion (árbol binario parcialmente
    ordenado)
10   * @param pos elemento a tratar (última posición del vector -> tam. caso n)
11   * @post apo reestructurado
12   */
13  void insertarEnPos(double *apo, int pos) {
14      int idx = pos-1;
15      int padre;
16      if (idx > 0) {
17          if (idx%2==0) {
18              padre = (idx-2)/2;
19          } else {
20              padre = (idx-1)/2;
21          }
22
23          if (apo[padre] > apo[idx]) {
24              double tmp = apo[idx];
25              apo[idx] = apo[padre];
26              apo[padre] = tmp;
27              insertarEnPos(apo, padre+1);
28          }
29      }
30  }
```

pos = n

idx = pos - 1 = n-1

padre $\begin{cases} = (idx-1)/2 = (n-2)/2 \\ = (idx-2)/2 = (n-3)/2 \end{cases}$

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

Asignaciones de tiempo cte

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

↘ núm. de veces que es llamada la función como máximo.

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

Cambio variable $n = 2^k$; $k = \log_2 n$

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

$t_k - t_{k-1} = k$ Rec. Lineal No Homogénea

Ec. característica : $(x-1)(x-1) = 0 \Longrightarrow (x-1)^2 = 0$

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

Deshacemos cambio:

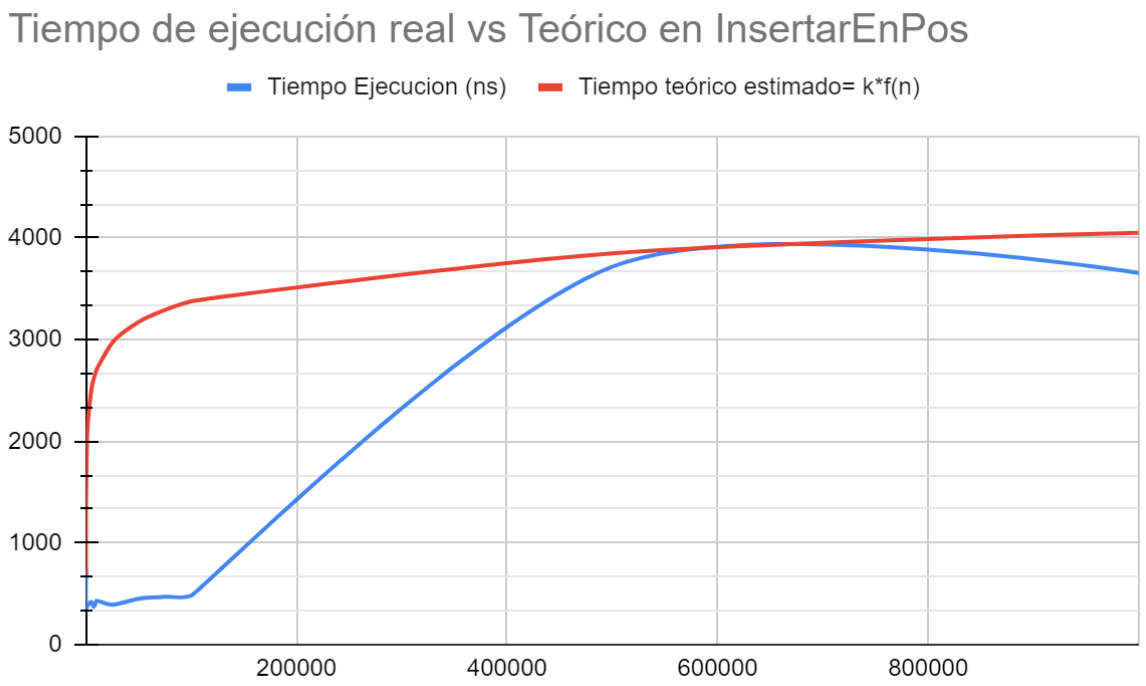
$t_n = c_2 \log_2 n + c_1 \Longrightarrow O(\log_2 n)$

Carlos Quesada Pérez
Miguel Ángel López Sánchez
David Serrano Domínguez

Eficiencia Práctica e Híbrida

Tam. Caso	Tiempo Ejecución (ns)	f(n) = logn	k = T(n)/f(n)	Tiempo teórico estimado= k*f(n)
10	741	1	741	675,1666667
100	381	2	190,5	1350,333333
1000	381	3	127	2025,5
5000	421	3,698970004	113,8154674	2497,421248
7500	371	3,875061263	95,74042184	2616,312196
10000	431	4	107,75	2700,666667
25000	391	4,397940009	88,90526001	2969,342496
50000	451	4,698970004	95,9784803	3172,587915
75000	470	4,875061263	96,40904485	3291,478863
100000	481	5	96,2	3375,833333
500000	3717	5,698970004	652,2231205	3847,754581
1000000	3656	6	609,3333333	4051

K promedio = 675,1666667



NOTA: El peor caso en este algoritmo es tener el vector con estructura de apo e insertar en la última posición el elemento mínimo (se necesita reestructurar el vector para convertir dicho elemento en la raíz del apo, elemento 0).

Carlos Quesada Pérez
Miguel Ángel López Sánchez
David Serrano Domínguez

Ejercicio 2b - *reestructurarRaiz*

Eficiencia Teórica

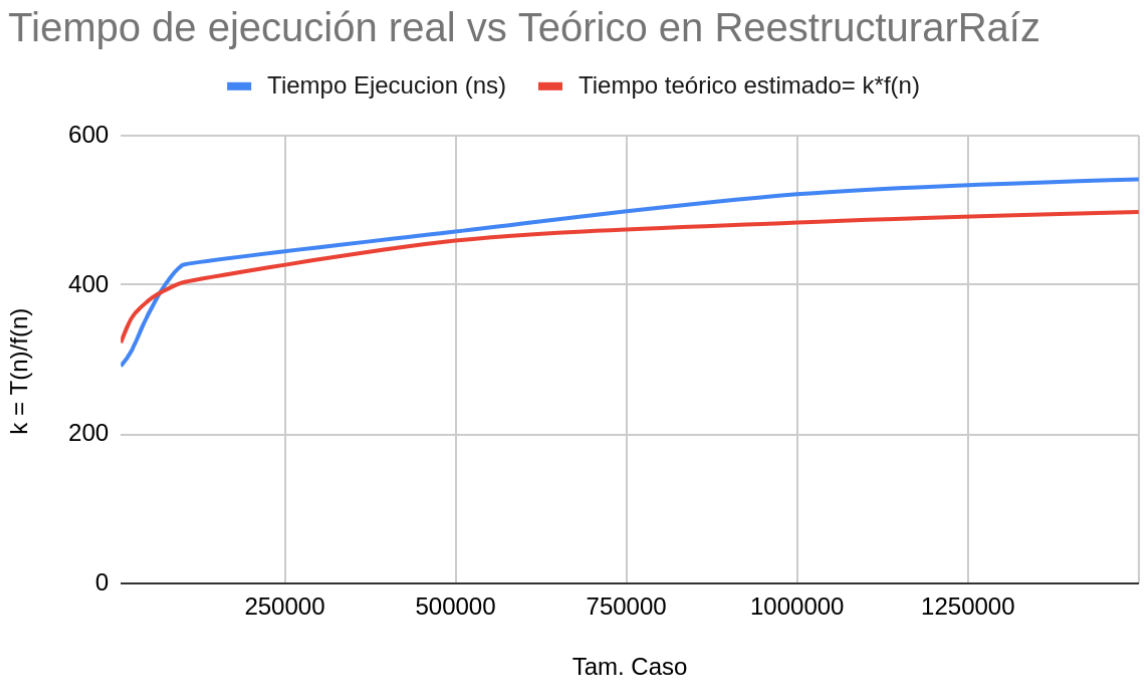
```
 7  /**
 8   * @brief Reestructura la posicion en función a la característica de un apo
 9   * @param apo vector sobre el que trabaja la funcion
10   * @param pos posicion a tratar
11   * @param tamapo tamaño total del vector
12   */
13  void reestructurarRaiz(double *apo, int pos, int tamapo) {
14      int minhijo;
15      if ((2*pos+1) < tamapo) {
16          minhijo=2*pos+1;
17
18          if ((minhijo+1 < tamapo) && (apo[minhijo]>apo[minhijo+1]))
19              minhijo++;
20
21          if (apo[pos]>apo[minhijo]) {
22              double tmp = apo[pos];
23              apo[pos] = apo[minhijo];
24              apo[minhijo] = tmp;
25              reestructurarRaiz(apo, minhijo, tamapo);
26          }
27      }
28  }
29
```

Eficiencia Práctica e Híbrida

Tam. Caso	Tiempo Ejecución (ns)	f(n) = logn	k = T(n)/f(n)	Tiempo teórico estimado= k*f(n)
10000	291	4	72,75	322,088017
25000	310	4,397940009	70,4875463	354,130944
50000	360	4,698970004	76,61253417	378,3704826
75000	400	4,875061263	82,05025094	392,5497037
100000	426	5	85,2	402,6100212
500000	471	5,698970004	82,64651325	458,8924868
1000000	521	6	86,83333333	483,1320254
1500000	541	6,176091259	87,59585591	497,3112465

K promedio = 80,52200424

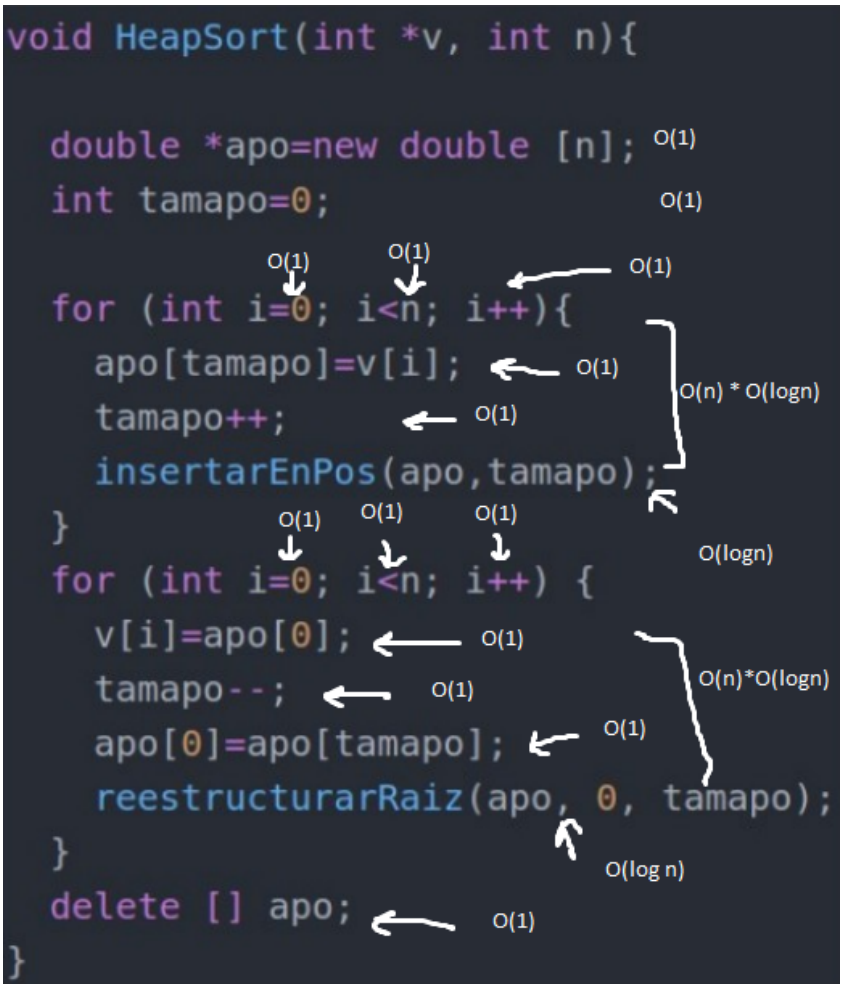
Carlos Quesada Pérez
Miguel Ángel López Sánchez
David Serrano Domínguez



NOTA: El peor caso en este algoritmo es tener el vector con estructura de apo e insertar en la última posición la que será la nueva raíz ya que tendrá el valor más pequeño de todo el vector y por tanto deberá ser la nueva raíz

Carlos Quesada Pérez
Miguel Ángel López Sánchez
David Serrano Domínguez

Ejercicio 3 - *HeapSort*



Eficiencia Teórica:

Dado a que las declaraciones de las variables y punteros son eficiencia O(1), no resultan ineficientes. El bucle “for” es de eficiencia O(n) sin embargo en cada iteración llama a la función “insertaEnPos” cuya eficiencia es O(log n) por lo que la eficiencia del bucle es O(n)*O(logn) o O(n)*O(logn)*O(1), es decir, O(nlogn). Ocurre lo mismo en el siguiente bucle “for”, la eficiencia del bucle sería tal cual el primer bucle,O(n)*O(logn)*O(1), que en su forma simplificada y correcta quedaría así, O(nlogn), ya que las tres primeras sentencias del bucle son de eficiencia O(1) y la llamada a la función “reestructuraRaiz” es de O(logn). Y por último la sentencia de eliminar de memoria el puntero “apo” es de eficiencia O(1).

Esto quiere decir que la eficiencia de la función “HeapSort” es de O(nlogn)+O(nlogn) (que es lo mismo a: T(n) = 2*O(n*log(n))), pero aplicando la regla de la suma, se descarta la constante 2 y quedaría: **T(n) = O(n*log(n))**

*(Todos los logaritmos en base 2).

Carlos Quesada Pérez
Miguel Ángel López Sánchez
David Serrano Domínguez

Eficiencia práctica:

Tam. Caso (n)	Tiempo Ejecución (us)	f(n) = n*log(n)	k = T(n)/f(n)	Tiempo teórico estimado= k*f(n)
1000	424,772	9965,784285	0,04262303777	306,4310597
2000	841,729	21931,56857	0,03837979018	674,3587464
3000	1071,856	34652,24036	0,03093179514	1065,497951
4000	1418,132	47863,13714	0,02962889783	1471,710747
5000	1705,1	61438,5619	0,02775292825	1889,132163
6000	2092,055	75304,48071	0,02778128181	2315,485782
7000	2485,323	89411,97445	0,02779631045	2749,267423
8000	2867,648	103726,2743	0,02764630293	3189,408002
9000	3250,323	118221,3836	0,02749352868	3635,108167
10000	3647,379	132877,1238	0,02744926211	4085,747462

K promedio = 0,03074831352

Ejercicio 4 - ***Comparación entre la eficiencia teórica e híbrida entre los algoritmos de ordenación por Burbuja, MergeSort y HeapSort, indicando cuál de ellos es más eficiente de forma justificada.***

Carlos Quesada Pérez
Miguel Ángel López Sánchez
David Serrano Domínguez

Comparación Teórica

Burbuja	MergeSort	HeapSort
$O(n^2)$	O(n*log(n))	O(n*log(n))

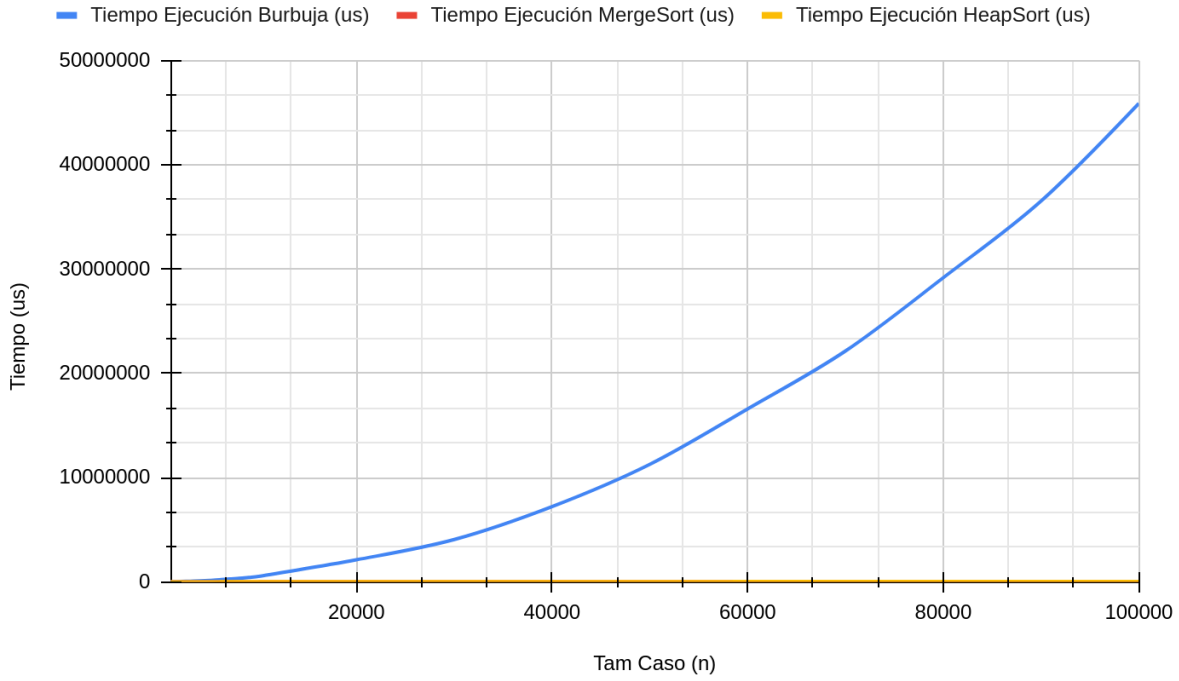
Comparación Híbrida

(Ejecución en el nodo de cálculo atcgrid)

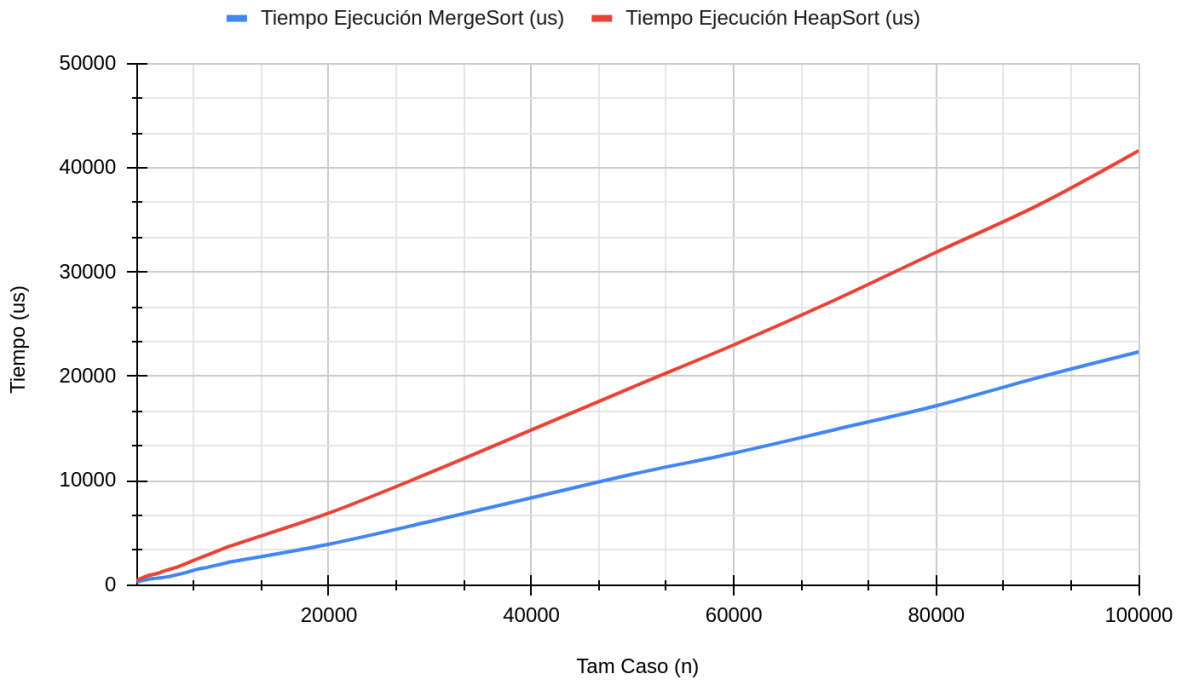
Tam. Caso (n)	**Tiempo Ejecución Burbuja (us)**	**Tiempo Ejecución MergeSort (us)**	**Tiempo Ejecución HeapSort (us)**
1000	7902,12	261,7	424,772
2000	31404,621	509,551	841,729
3000	60174,774	634,848	1071,856
4000	87073,063	759,064	1418,132
5000	131589,712	967,744	1705,1
6000	189234,078	1213,16	2092,055
7000	279001,163	1492,72	2485,323
8000	336168,656	1681,091	2867,648
9000	426126,095	1906,658	3250,323
10000	525363,647	2142,73	3647,379
20000	2113668,376	3898,215	6882,232
30000	4056046,303	6078,921	10765,285
40000	7213709,107	8345,024	14852,895
50000	11269542,9	10623,675	18971,74
60000	16587699,39	12642,078	23017,987
70000	22139857,15	14894,112	27333,445
80000	29170276,96	17176,934	31908,741
90000	36527659,51	19865,775	36394,976
100000	45895614,29	22343,621	41658,103

Gráfica con los tiempos del algoritmo de Burbuja ($O(n^2)$). No sirve para comparar los tres algoritmos puesto que es mucho más ineficiente que los otros dos.

Carlos Quesada Pérez
Miguel Ángel López Sánchez
David Serrano Domínguez



Gráfica comparativa (sin tener en cuenta el algoritmo de Burbuja). En esta sí que se puede apreciar la diferencia entre los dos algoritmos de misma eficiencia teórica.



Nota: Las condiciones de ejecución han sido exactamente iguales. Se han ordenado los mismos vectores (mismo contenido y mismo tamaño) en el mismo programa y ejecutado en el mismo proceso.

Carlos Quesada Pérez
Miguel Ángel López Sánchez
David Serrano Domínguez

Conclusión

Comenzamos la comparativa descartando al algoritmo de ordenación por Burbuja, que claramente tanto en la teoría como en la práctica era el menos eficiente. La pregunta realmente es cuál es más eficiente ¿MergeSort o HeapSort?

Teóricamente, ambos tienen la misma eficiencia, pero no significa que ambos sean igual de rápidos. La eficiencia teórica de los algoritmos se ha calculado obviando las constantes, puesto que sólo interesa saber su comportamiento en el infinito. De hecho, al calcular la eficiencia de HeapSort, se podía observar como su eficiencia era aproximadamente 2*O(n*log(n)) frente a la aparente O(n*log(n)) de MergeSort. Se podía ahí sospechar de que MergeSort podría ser más eficiente (mismo comportamiento pero menor constante oculta (k)).

Es en la práctica donde realmente se observa que el algoritmo *MergeSort es el más eficiente* de los dos (y de los tres). Sus tiempos de ejecución son aproximadamente el doble de rápidos que los de HeapSort aunque su comportamiento sea muy similar o prácticamente idéntico.