PRÁCTICA 1

Mario Casas Pérez, Rocío Barragán Moreno y Ángela María Garrido Ruiz
Grupo: D1

1. ALGORITMOS ITERATIVOS

1. Diseñe un algoritmo que, teniendo como entrada un vector de números enteros de tamaño N, compruebe si existen elementos repetidos en el vector y elimine todas las ocurrencias repetidas del mismo elemento. Como salida, se devolverá el mismo vector con todos los elementos sin repetir. No se requiere que el vector de salida tenga los elementos en el mismo orden que el array original. Por ejemplo, para la entrada V= {5, 1, 9, 2, 2, 5, 1, 1, 7}, de tamaño N=9, una posible salida sería V'= {2, 9, 5, 7, 1}, de tamaño N= 5. Otra posible salida alternativa sería V'= {1, 2, 5, 7, 9}, también de tamaño N=5. El único requisito del vector de salida es que contenga los mismos elementos que el array de entrada, sin repetir.

```
8
9   void EliminaOcurrencias(int v[], int &utiles){
10
11      const int max = utiles;
12      int aux[max];
13      int util = 0;
14
15      for (int i=0; i<utiles; i++){
16
17          //Si la i es 0, se inicializará el vector auxiliar y añadimos el primer valor
18          //para ir comparando valores
19          if (i == 0){
20              aux[i] = v[i];
21              util++;
22
23          }
24
25          //Cuando la i>0, se irá seleccionando qué elementos se van añadiendo al vector auxiliar
26          //en función de si están o no repetidos en el vector auxiliar.
27          else{
28
29              //utilizamos un booleano para comprobar que el elemento de v[] que estamos analizando
30              //no coincide con ningún elemento de los que se encuentran en aux[]
31              bool repetido = false;
32              for (int j=0; j<util && !repetido; j++){
33
34                  //Si coinciden, significa que es un valor repetido, por tanto no se añadirá al vector auxiliar.
35                  if(aux[j] == v[i])
36                      repetido = true;
37
38              }
39
40              //Si el booleano sigue a false, significa que ese elemento de v[] no se encuentra en el vector aux
41              //por tanto, se añadirá ya que no sería un elemento repetido
42              if (!repetido){
43                  aux[util] = v[i];
44                      util++; //aumentamos el valor de util ya que hemos añadido un nuevo elemento al vector
45              }
46
47          }
48      }
49      //Redimensionamos utiles al valor de util, para rellenar v[] con los valores necesarios de aux[]
50      utiles = util;
51
52      //Volcamos los elementos de aux[] en v[]
53      for (int i=0; i<utiles; i++){
54          v[i] = aux[i];
55      }
56
57  }
58
```

2. Supongamos ahora que disponemos de un vector de entrada, también conteniendo Números enteros, que cumple con la precondición de que se encuentra ordenado. Diseñe un algoritmo lo más eficiente posible para eliminar los elementos repetidos del vector y devuelva, como salida, el mismo vector modificado con elementos sin repetir. Por ejemplo, para el vector de entrada V={1, 1, 1, 2, 2, 5, 5, 7, 9} de tamaño N=9, una posible salida sería V'= {1, 2, 5, 7, 9} de tamaño 5.

Hemos hecho la función OrdenarInserccion ya que el vector se inicializa con elementos aleatorios que no van a estar ordenados. Como para el método EliminaOcurrencias debemos suponer que dichos elementos están ordenados primero pasamos el vector por la función OrdenarInserccion para garantizar esa precondición y después eliminamos.
Para medir la eficiencia dejamos fuera de las mediciones de tiempo el algoritmo de ordenación para estudiar solamente el algoritmo nuevo de EliminaOcurrencias

```
8
9   void OrdenarInserccion(int v[], int utiles){
10      int a_desplazar;
11      int i;
12
13      for (int izda = 1; izda <= utiles; izda++){ //recorremos el vector del inicio al final
14          a_desplazar = v[izda]; //establezco el elemento candidato a desplazarse
15
16          for (i = izda; i > 0 && a_desplazar < v[i-1]; i--) //recorro del final al inicio el vector y si se cumple la condicion de
17                                                              // que el elemento seleccionado es menor que el ultimo elemento del vector
18              v[i] = v[i-1]; //realizo el intercambio
19
20          v[i] = a_desplazar; //establezco mi nuevo elemento a desplazar
21      }
22  }
23
24  void EliminaOcurrencias(int v[], int &utiles){
25      const int max = utiles;
26      int aux[utiles];
27      int util = 0;
28
29      //Recorre el vector hasta utiles -1 porque el último valor no se comparará con ninguno
30      for (int i=0; i<utiles-1; i++){
31          //Para i=0, se inicializa el auxiliar con el primer elemento del vector original para ir
32          //comparando elementos.
33          if(i==0){
34              aux[i] = v[i];
35              util++;
36              //Compara el primer valor del vector con el siguiente, y si el siguiente es mayor, se añadiría al vector aux
37              if (v[i] < v[i+1]){
38                  aux[util] = v[i+1];
39                  util++;
40              }
41          }
42
43          //Para i>1, va comparando con el valor siguiente (i+1) y como suponemos que el vector de entrada está ordenado,
44          //si v[i] < v[i+1] se añade v[i+1], pues  significará que es un nuevo elemento.
45          else{
46              if (v[i] < v[i+1]){
47                  aux[util] = v[i+1];
48                  util++;
49              }
50          }
51      }
52
53      //Redimensionamos utiles a util
54      utiles = util;
55
56      //Volcamos el vector original
57      for (int i=0; i<utiles; i++){
58          v[i] = aux[i];
59      }
60
61  }
```

3. Identifique de qué variable, o variables, depende el tamaño del problema en cada algoritmo diseñado.

En el ejercicio 1 el tamaño del problema depende de la variable útiles la cual se va decrementando (se pasa por referencia) conforme elimine ocurrencias.

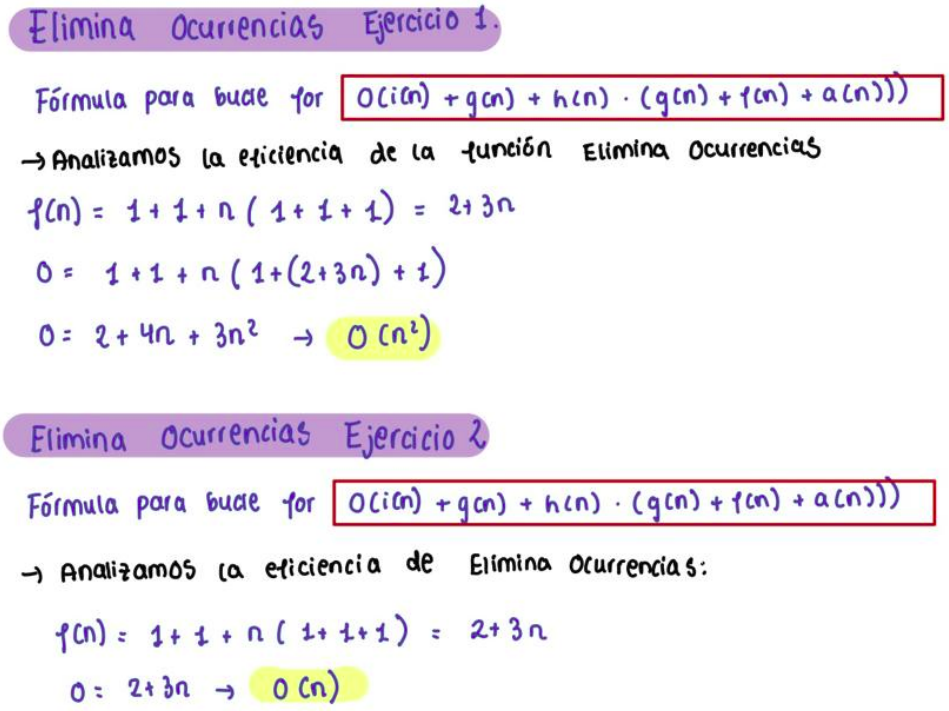
En el ejercicio 2 el tamaño del problema depende también de los útiles tanto en la función de eliminar ocurrencias como el de inserción.

4. Identifique cuáles serían los casos mejor y peor para cada algoritmo diseñado. Exponga un ejemplo para una instancia concreta de tamaño determinado para cada caso, y justifique porqué esa instancia hace que el algoritmo analizado ejecute el máximo número de operaciones (caso peor) o el mínimo (caso mejor).

- **Ejercicio 1:** el mejor caso que se pueda dar es que no se presente ningún tipo de ocurrencia en el vector del mismo número, y el peor caso es que todos los números que aparecen en el vector estén repetidos.

- **Ejercicio 2:** el mejor caso sería que el vector no tenga elementos repetidos y que se repitan n-1 elementos.

5. Calcule los órdenes de eficiencia de ambos algoritmos en el caso peor y mejor.



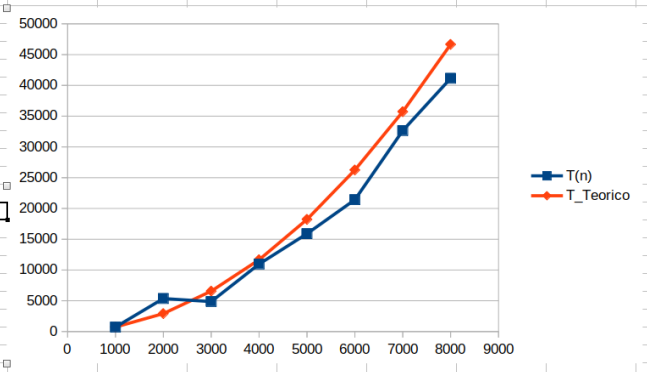
6. Realice una experimentación con instancias aleatorias de vectores de entrada de diferentes tamaños de caso (1000, 2000, 3000, ..., 10000, 20000, 30000, ...), midiendo tamaños de caso y tiempos de ejecución para cada uno de ellos. Calcule la constante oculta de cada algoritmo y muestre en una gráfica el tiempo real de cada algoritmo junto con la gráfica del tiempo de ejecución teórico (híbrido) con la constante calculada. Justifique los resultados que se visualizan en la gráfica para cada algoritmo.

Cálculos ejercicio 1

El T(n) lo hemos calculado probando con los distintos tamaños de vector, desde 1000 a 8000.
K se calcula como T(n)*f(n).

Y una vez calculada la constante podemos calcular el tiempo teórico como K*f(n).

n	T(n)	T_Teorico	K
1000	725	729,0021358	0,000725
2000	5363	2916,008543	0,00134075
3000	4869	6561,019222	0,000541
4000	10964	11664,03417	0,00068525
5000	15894	18225,05339	0,00063576
6000	21422	26244,07689	0,000595056
7000	32650	35721,10465	0,000666327
8000	41144	46656,13669	0,000642875
		K_media	**0,000729002**

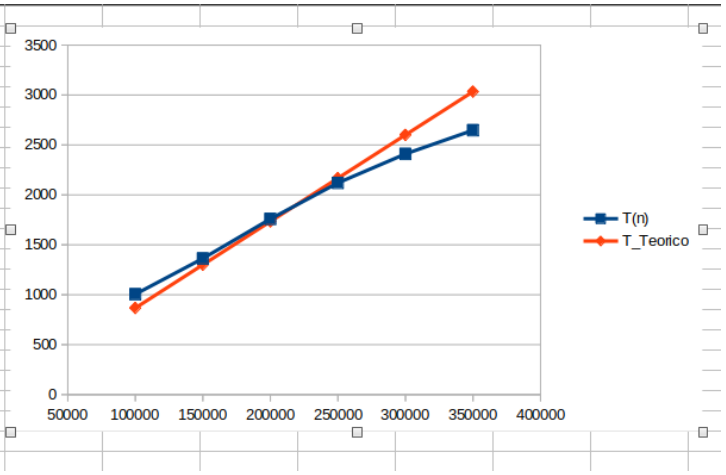


Como podemos observar el algoritmo tiene una tendencia cuadrática que concuerda con el orden de eficiencia calculado anteriormente (o(n²))

Cálculos ejercicio 2

El T(n) lo hemos calculado probando con los distintos tamaños de vector, desde 100000 a 350000. Hemos cambiado el rango de valores dado que hemos querido experimentar con valores mayores y comprobar que la tendencia empírica se asemeja a la teórica, mejor que con valores pequeños.
K se calcula como T(n)*f(n)
Y una vez calculada la constante podemos calcular el tiempo teórico como K*f(n)

n	T(n)	T_Teorico	K
100000	1005	866,797619	0,01005
150000	1363	1300,196429	0,009086667
200000	1759	1733,595238	0,008795
250000	2120	2166,994048	0,00848
300000	2410	2600,392857	0,008033333
350000	2647	3033,791667	0,007562857
		K_media	0,008667976



Como podemos ver en la gráfica este algoritmo de EliminarOcurrencias tiene una tendencia lineal como nos indicaba el cálculo anterior de la eficiencia (O(n)).

2. ALGORITMOS RECURSIVOS

1. Calcule la ecuación en recurrencias, y el orden del algoritmo en el caso peor, para las funciones Hanoi y HeapSort siguientes.

Ecuación de recurrencias de Hanoi

La función tiene la ecuación en recurrencias siguiente **T(n) = 2T(n-1)+1**.

Hemos llegado a esta conclusión estudiando el caso base y el caso general basándonos en el código:

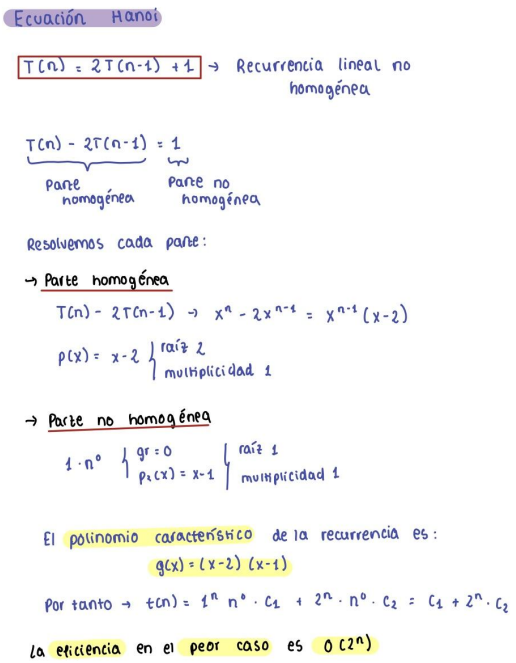
```
/**
Se trata del problema clásico de las torres de Hanoi.
Se tienen 3 barras, y hay que mover M anillos de la primera barra
a la segunda. Solo se puede mover un anillo en cada movimiento,
y ningún anillo de tamaño mayor puede ponerse sobre otro de tamaño
menor.
Los valores de "i" y "j" sólo pueden tomar los valores {1, 2, 3}

Si M=3, la llamada sería hanoi(3, 1, 2)

*/
void hanoi (int M, int i, int j)
{
  if (M > 0)
    {
      hanoi(M-1, i, 6-i-j);
      cout << i << " -> " << j << endl;
      hanoi (M-1, 6-i-j, j);
    }
}
```

- Caso base: si no se cumple la condición de que M sea mayor que 0 no se hace nada, solo se comprobaría la condición del if, por lo que O(1)
- Caso general: se cumple que M es mayor que 0 por lo que se hacen dos llamadas recursivas de la función con un tamaño M-1, por tanto T(n-1), y una salida de datos que tiene un orden O(1).

La ecuación ha sido resuelta de la siguiente manera:



Ecuación de recurrencias de HeapSort

- ### Recurrencias de la función insertarEnPos

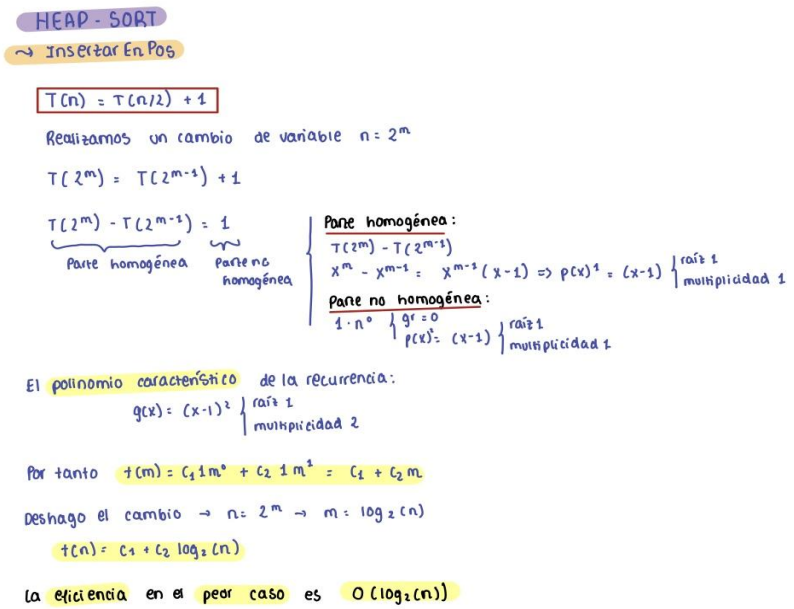
La función tiene la ecuación en recurrencias siguiente **T(n) = T(n/2) +1.**
Para sacar esta ecuación hemos estudiado los casos base y general basándonos en el código proporcionado :

```
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);
    }
  }
}
```

- Caso base: si no se cumple que el elemento en la posición padre es mayor que el elemento en la posición idx realizamos dos operaciones elementales dentro de una estructura if-else cuyo orden de eficiencia en el peor de los casos será de O(1)
- Caso general: si se cumple que el elemento en la posición padre es mayor que el elemento en la posición idx realizamos una serie de operaciones elementales y después hacemos llamadas recursivas a la función. Como vemos depende del valor de la variable padre que cambia su valor a la mitad por lo que el tiempo de ejecución será de T(n/2)

La ecuación ha sido resuelta de la siguiente manera:



- <u>Recurrencia de la función restructurarRaiz</u>

La función tiene la ecuación en recurrencias siguiente **T(n) = T(n/2) +1.**
Para sacar esta ecuación hemos estudiado los casos base y general basándonos en el código proporcionado :

```
void reestructurarRaiz(double *apo, int pos, int tamapo){
  int minhijo;
  if (2*pos+1< tamapo) {
    minhijo=2*pos+1;
    if ((minhijo+1 < tamapo) && (apo[minhijo]>apo[minhijo+1])) minhijo++;
    if (apo[pos]>apo[minhijo]) {
      double tmp = apo[pos];
      apo[pos]=apo[minhijo];
      apo[minhijo]=tmp;
      reestructurarRaiz(apo, minhijo, tamapo);
    }
  }
}
```

- Caso base: minhijo será una posición no válida o el vector en el elemento pos será mayor que el vector en el elemento minhijo, en este caso se realizará una operación elemental (minhijo++) con orden de eficiencia O(1)
- Caso general: el vector elemento del vector en la posición pos será mayor que el elemento en la posicion minhijo y se procederá a hacer una serie de operaciones elementales terminando con una llamada recursiva a la propia función. Cada vez que se llama a pos está se multiplica por 2 lo que hace que se divida a la mitad el tiempo de ejecución. (T(n/2))

La ecuación resultante es la misma que para la función insertarEnPos por lo que la resolución de la recurrencia se ha hecho de la misma forma obteniendo el mismo orden de eficiencia **$O(\log_2(n))$**

- <u>Eficiencia total</u>

```
void HeapSort(int *v, int n){

  double *apo=new double [n];
  int tamapo=0;

  for (int i=0; i<n; i++){
    apo[tamapo]=v[i];
    tamapo++;
    insertarEnPos(apo,tamapo);
  }
  for (int i=0; i<n; i++) {
    v[i]=apo[0];
    tamapo--;
    apo[0]=apo[tamapo];
    reestructurarRaiz(apo, 0, tamapo);
  }
  delete [] apo;
}
```

Como podemos observar en el código las dos primeras sentencias son operaciones elementales que tienen una eficiencia O(1)
El primer bucle for tiene la llamada a la función insertarEnPos que como hemos visto en los cálculos anteriores tiene una eficiencia de O(log(n)). El bucle se repite n veces por lo que la eficiencia total será de $O(n*\log_2(n))$
El segundo bucle for tiene el mismo orden de eficiencia que el primero ya que hace la llamada a la función reestructurar Raíz que tiene el mismo orden que insertarEnPos y se repite de igual forma n veces. Por tanto $O(n*\log_2(n))$.
La última sentencia es una operación elemental de orden de eficiencia O(1)

Por tanto, la eficiencia total del algoritmo HeapSort es de:

$$O(1) + O(n*\log_2(n)) + O(n*\log_2(n)) + O(1) \rightarrow \mathbf{O(n*\log_2(n))}$$

2. Compare la eficiencia del algoritmo HeapSort con el algoritmo MergeSort, explicado en clase de teoría, de forma teórica y de forma híbrida para el caso peor. Para ello, realice una experimentación con instancias aleatorias de vectores de entrada de diferentes tamaños de caso (1000, 2000, 3000, ..., 10000, 20000, 30000, ...), midiendo tamaños de caso y tiempos de ejecución para cada uno de ellos. Calcule la constante oculta de cada algoritmo y muestre en una gráfica el tiempo real de cada algoritmo junto con la gráfica del tiempo de ejecución teórico (híbrido) con la constante calculada. Justifique los resultados que se visualizan en la gráfica para cada algoritmo e indique porqué los algoritmos se comportan igual o, si se da el caso, uno es mejor que otro.
Hemos calculado primero la eficiencia del algoritmo MergeSort.
Hemos resuelto la ecuación en recurrencias de la siguiente forma:

MERGE - SORT

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

Realizamos un cambio de variable $n = 2^m$

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

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

Parte homogénea: $T(2^m) - 2T(2^{m-1}) \Rightarrow (x-2)$

Parte no homogénea: $2^m = (x-2)$

El polinomio característico de la recurrencia es:

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

Por tanto $t(m) = c_1 \cdot 2^m \cdot m^0 + c_2 \cdot 2^m \cdot m^1$

$$t(m) = c_1 \cdot 2^m + c_2 \cdot 2^m \cdot m$$

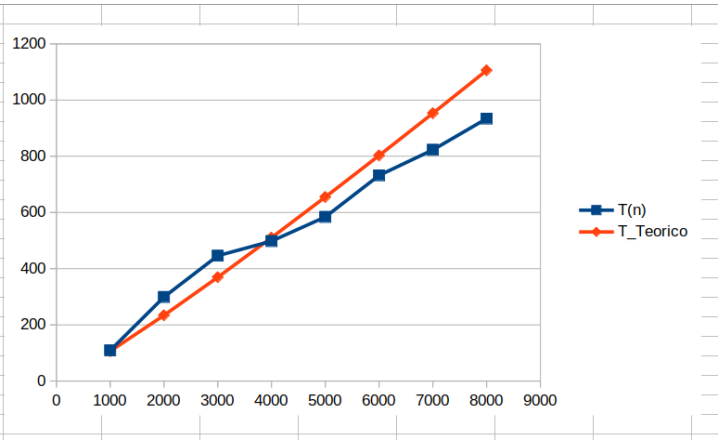
Deshacemos el cambio $n = 2^m \rightarrow m = \log_2(n)$

$$t(n) = c_1 \cdot n + c_2 \cdot n \cdot \log_2(n)$$

La eficiencia en el peor caso es $O(n \log_2(n))$

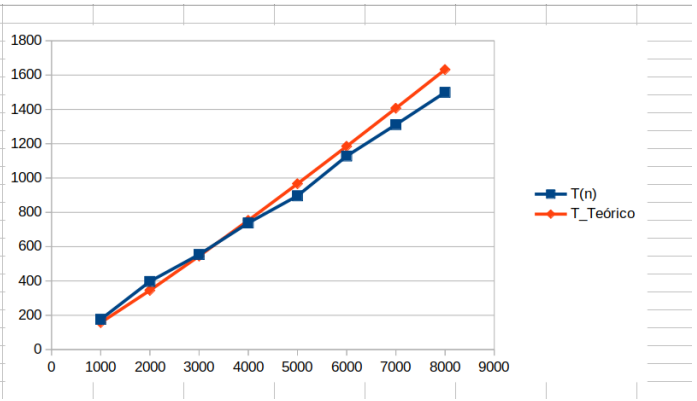
Los cálculos del tiempo empírico, el tiempo teórico y la K lo hemos realizado como en el ejercicio 6 de la parte iterativa, obteniendo los resultados y la gráfica siguiente:

n	T(n)	T_Teorico	K
1000	109	106,2367142	0,036333333
2000	299	233,7937202	0,045288895
3000	446	369,397937	0,042755675
4000	498	510,2280239	0,034563555
5000	584	654,9440321	0,031576358
6000	732	802,7567493	0,032290925
7000	823	953,1446905	0,030576965
8000	934	1105,737215	0,029912198
		K_MEDIA	0,035412238

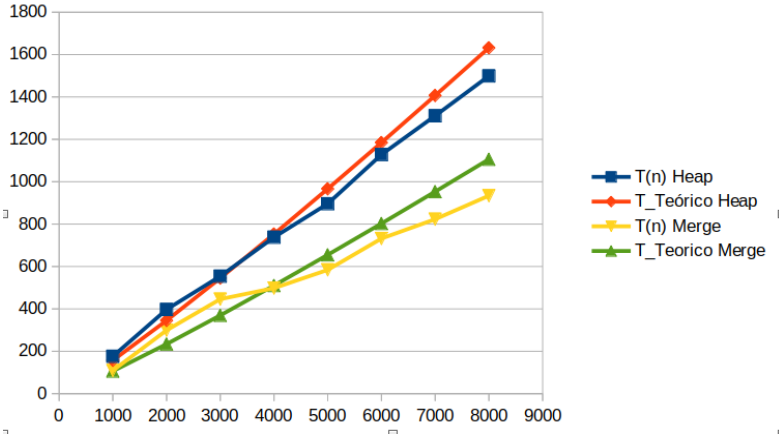


A continuación analizamos la eficiencia del HeapSort cuya ecuación de recurrencias y eficiencia hemos calculado en el ejercicio anterior obteniendo O(n*log2(n))

n	T(n)	T_Teórico	K
1000	176	156,7685132	0,058666667
2000	397	344,9983762	0,060132747
3000	554	545,1031292	0,053109067
4000	738	752,9194523	0,051220691
5000	896	966,4700464	0,048445919
6000	1128	1184,590308	0,049759787
7000	1311	1406,510706	0,048707656
8000	1499	1631,684304	0,048006836
		k_media	0,052256171



Como hemos visto ambos algoritmos tienen el mismo orden de eficiencia por lo que sus gráficas y resultados son muy similares. Para comparar ambos hemos realizado una gráfica con las eficiencias teóricas y empíricas de ambos:



Estudiando la gráfica podemos ver que el algoritmo MergeSort se comporta mejor que el HeapSort en el rango de valores estudiado.