Práctica 1:
Cálculo de la eficiencia de algoritmos

Santiago Carbó García y Carlos García Segura

Universidad de Granada, España

1 Algoritmo 1: MaximoMinimoDyV

1.1 Variables que influyen en el tamaño del caso

El tamaño del problema viene determinado por el tamaño del vector "A" y las variables "Cini" y "Cfin". Aunque el tamaño del vector sea uno, las variables "Cini" y "Cfin" son determinantes, ya que ellas indican cuánto se va a recorrer del vector.

1.2 Eficiencia teórica

```
Función [Max, Min]= MaximoMinimoDyV(A, Cini, Cfin)
Entradas:
    A: Vector de N componentes de tipo T, indexadas de 1 a N
    Cini: Componente inicial de A donde se inicia la búsqueda. Cini>=1
    Cfin: Componente final de A donde se finaliza la búsqueda. Cini<Cfin<=N
Salidas:
    Max: Máximo elemento de A entre A[Cini]... A[Cfin]
    Min: Mínimo elemento de A entre A[Cini]... A[Cfin]
INICIO-Algoritmo
    Si Cini < Cfin-1, hacer:
        mitad= parte entera de (Cini+Cfin)/2
        [Max1, Min1]= MaximoMinimoDyV(A, Cini, mitad)
        [Max2, Min2]= MaximoMinimoDyV(A, mitad+1, Cfin)
        Max= Máximo entre Max1 y Max2
        Min= Mínimo entre Min1 y Min2
    En otro caso, Si Cini=Cfin, hacer:
        Max=A[Cini], Min= A[Cini]
    En otro caso, hacer:
        Max= Máximo entre A[Cini] y A[Cfin]
        Min= Mínimo entre A[Cini] y A[Cfin]
    Devolver Max, Min
FIN-Algoritmo
```

Es un algoritmo recursivo. Existen tres casos (dos casos base y un caso general):

- Caso base 1: Cuando "Cini == Cfin" (Sentencia 7 del pseudocódigo). La ejecución del algoritmo termina después de salir del "if" ya que, al entrar en el caso base, no se producen más llamadas recursivas. Las dos asignaciones de dentro son O(1), y la evaluación de la condición es O(1), de forma que $max(O(1), O(1), O(1)) = O(1)$.

- Caso base 2: Cuando "Cini > Cfin" (Sentencia 9 del pseudocódigo). La ejecución del algoritmo termina después de salir del "if" ya que, al entrar en el caso base, no se producen más llamadas recursivas. Dentro, calcula en dos sentencias el máximo y el mínimo entre dos números, lo cuál son realmente dos "if", declarados de esta forma:

```
if(A[Cini]>A[Cfin])
      max = A[Cini];
else
      max = A[Cfin];
if(A[Cini]<A[Cfin])
      min = A[Cini];
else
      min = A[Cfin];
```

Cada una de las sentencias de asignación es O(1), y la evaluación de la condición es O(1), de forma que cada uno de los "if" por separado es $max(O(1), O(1)) = O(1)$. Sin embargo, al tratarse de dos declaraciones "if-else" estamos hablando de $O(1 + 1) = O(2)$.

- Caso general: Cuando "Cini < Cfin-1" (Sentencia 1 del pseudocódigo). Dispone de lo siguiente:
  - El algoritmo calcula la parte central del subvector en "mitad". Esta operación es O(1).
  - En la siguiente sentencia (Sentencia 3 del pseudocódigo) hace una llamada recursiva para resolver un subproblema desde "Cini" y "mitad" (es decir, calcular el máximo y el mínimo de ese subproblema). Si T(n) es el tiempo que tarda el algoritmo en resolver un problema de tamaño n, entonces la llamada recursiva tendrá un tiempo de ejecución T(n/2).
  - A continuación (Sentencia 4 del pseudocódigo) hace otra llamada recursiva para resolver un subproblema desde "mitad+1" y "Cfin" (es decir, calcular el máximo y el mínimo de ese subproblema). Si T(n) es el tiempo que tarda el algoritmo en resolver un problema de tamaño n, entonces la llamada recursiva tendrá un tiempo de ejecución T(n/2).
  - Finalmente (en las sentencias 5 y 6 del pseudocódigo), se calcula, en dos sentencias, el máximo y el mínimo entre dos números, lo cuál son realmente dos "if", declarados, también, de esta forma:

```
if(Max1>Max2)
      max = Max1;
else
      max = Max2;
if(Min1<Min2)
      min = Min1;
else
      min = Min2;
```

    Cada una de las sentencias de asignación es O(1), y la evaluación de la condición es O(1), de forma que cada uno de los "if" por separado es $max(O(1), O(1)) = O(1)$. Sin embargo, al tratarse de dos declaraciones "if-else" estamos hablando de $O(1 + 1) = O(2)$.

Concluimos en que obtenemos una Recurrencia Lineal No Homogénea, la cuál sería la siguiente:

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

Como la ecuación no tiene las variables de la forma requerida por la ecuación característica, tenemos que hacer el siguiente cambio de variable:

$$n = 2^m \; ; \; m = log_2 n$$

De forma que la ecuación quedaría así:

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

Despejamos, y quedaría así:

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

Resolvemos la parte homogénea:

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

$$x^m - 2x^{m-1} = 0$$

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

Obteniendo la parte homogénea del polinomio característico: $P_H(x) = (x - 2)$

Para resolver la parte no homogénea, debemos conseguir un escalar b_1 y un polinomio q_m :

$2 = {b_1}^m q_1(m)^{d_1}$

donde podemos ver que $b_1 = 2$ y $q_1(m) = 1$, donde el grado del polinomio es $d_1 = 0$

El polinomio característico se obtiene como: $P(x) = P_H(x)(x - b_1)^{d_1+1} = (x - 2)(x - 2) = (x - 2)^2$

Tenemos $r = 1$ (ya que solo tenemos una raíz diferente), con valor $R_1 = 2$ y multiplicidad $M_1 = 2$

Si aplicamos la fórmula de la ecuación característica, tenemos que el tiempo de ejecución (expresado en términos de 2^m) es el siguiente:

$T(2^m) = \sum_{i=1}^{m} \sum_{j=0}^{M_i-1} c_{ij} {R_i}^m m^j = c_{10}2^m + c_{11}2^m m$

Deshacemos el cambio de variable:

$T(n) = c_{10}n + c_{11}nlog_2(n)$

Aplicando la **regla de la suma**, es decir, eligiendo el máximo, obtenemos que el orden de eficiencia del algoritmo en el peor de los casos es $O(n * log(n))$.

1.3 **Eficiencia práctica**

Tam. Caso	Tiempo (us)
10000	1281
20000	1531
30000	2212
40000	3247
50000	4368
60000	4535
70000	5217
80000	6540
90000	7646
100000	9322

1.4 **Eficiencia híbrida**

K=Tiempo/f(n)	Tiempo teórico estimado= K*f(n)
0,032025	744,7524835
0,017798062	1601,601386
0,016468916	2500,76038
0,01763884	3427,395608
0,018591308	4374,461976
0,015818531	5337,810016
0,015382263	6314,697897
0,016673159	7303,17689
0,017148041	8301,789929
0,018644	9309,406044
K promedio	0,018618812

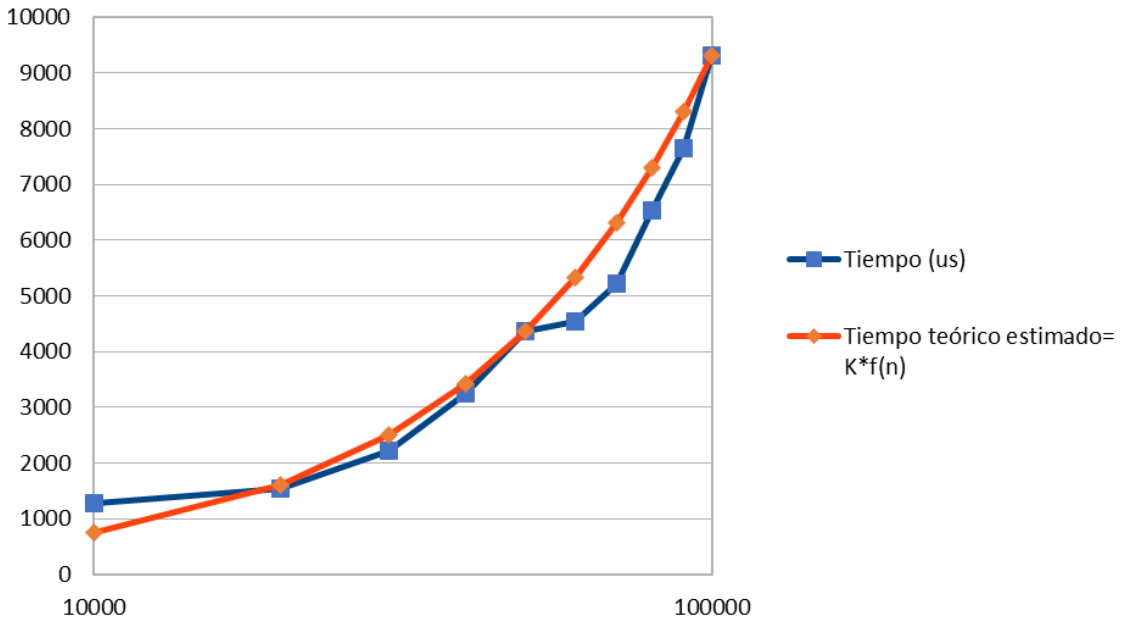

Fig. 1. Comparación entre tiempo práctico y tiempo teórico estimado del algoritmo MaximoMinimoDyV

Conclusión: El tiempo práctico se encuentra por debajo del tiempo teórico estimado (exceptuando el primer valor), el cuál es el tiempo teórico utilizando la constante oculta de nuestro ordenador. Además, sigue la tendencia de una función $nlog(n)$, lo cual nos confirma que el proceso de análisis se ha realizado correctamente.

2 Algoritmo 2: insertarEnPos y reestructurarRaiz

2.1 Variables que influyen en el tamaño del caso (insertarEnPos)

El tamaño del problema viene determinado por el tamaño del vector "apo" y la variable "pos". Aunque el tamaño del vector sea uno, la variable "pos" es determinante, ya que ella indica cuánto se va a recorrer del vector. Se va a recorrer un camino del Árbol Parcialmente Ordenado, y cuanto más grande sea "pos" más largo será el camino.

2.2 Eficiencia teórica (insertarEnPos)

```
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);
    }
  }
}
```

Es un algoritmo recursivo. El algoritmo, en esencia, va comparando, de abajo hacia arriba del árbol, los nodos hijos con los nodos padres, y si la etiqueta del nodo hijo es menor que la del nodo padre, entonces se intercambian sus etiquetas de valor. Es decir, recorre, como máximo, un único camino. Existen tres casos:

- Primer caso: Cuando "(idx <= 0)" (Sentencia 3 del pseudocódigo). La ejecución del algoritmo termina después de este caso, ya que no entra al "if" principal. La primera declaración es O(1), la segunda declaración es O(1) y la evaluación de la condición es O(1), de forma que $max(O(1), O(1), O(1)) = O(1)$.

- Segundo caso: Cuando "(idx > 0) AND (apo[padre] <= apo[idx])" (Sentencias 3 y 10 del pseudocódigo). La ejecución del algoritmo también termina ya que, aunque entra al "if" principal, no entra al segundo "if" (el cuál es el importante). Las dos asignaciones de "if" y "else" son O(1), y la evaluación de la condición es O(1), y la evaluación de la condición del segundo "if" es O(1), de forma que $max(O(1), O(1), O(1)) = O(1)$. Comparándola con la parte que hemos hecho antes, fuera del "if" principal, que también es O(1), quedaría finalmente $max(O(1), O(1)) = O(1)$. Es importante añadir que la variable "padre" se reduce, aproximadamente, a la mitad. Esto es importante para el siguiente caso.

- Tercer caso (caso general): Cuando "(idx > 0) AND (apo[padre] > apo[idx])" (Sentencias 3 y 10 del pseudocódigo). Dispone de lo siguiente:
    - La primera sentencia de este último "if" es una declaración y asignación de una variable, de forma que es O(1).
    - La segunda y la tercera sentencia son asignaciones de una variable, de forma que son O(1).
    - Finalmente hacemos la llamada recursiva, pasándole como parámetros el vector y "padre +1". Cómo he indicado en el punto anterior, padre se reduce a la mitad en el "if" anterior. La llamada recursiva resuelve un subproblema desde el principio del vector hasta "padre+1". Si T(n) es el tiempo que tarda el algoritmo en resolver un problema de tamaño n, entonces la llamada recursiva tendrá un tiempo de ejecución T(n/2).

Concluimos en que obtenemos una Recurrencia Lineal No Homogénea, la cuál sería la siguiente:

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

Como la ecuación no tiene las variables de la forma requerida por la ecuación característica, tenemos que hacer el siguiente cambio de variable:

$$n = 2^m \; ; \; m = log_2 n$$

De forma que la ecuación quedaría así:

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

Despejamos, y quedaría así:

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

Resolvemos la parte homogénea:

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

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

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

Obteniendo la parte homogénea del polinomio característico: $P_H(x) = (x - 1)$

Para resolver la parte no homogénea, debemos conseguir un escalar b_1 y un polinomio q_m :

$1 = b_1{}^m q_1(m)^{d_1}$

donde podemos ver que $b_1 = 1$ y $q_1(m) = 1$, donde el grado del polinomio es $d_1 = 0$

El polinomio característico se obtiene como: $P(x) = P_H(x)(x - b_1)^{d_1+1} = (x - 1)(x - 1) = (x - 1)^2$

Tenemos $r = 1$ (ya que solo tenemos una raíz diferente), con valor $R_1 = 1$ y multiplicidad $M_1 = 2$

Si aplicamos la fórmula de la ecuación característica, tenemos que el tiempo de ejecución (expresado en términos de 2^m) es el siguiente:

$T(2^m) = \sum_{i=1}^{m} \sum_{j=0}^{M_i - 1} c_{ij} R_i{}^m m^j = c_{10} 1^m + c_{11} 1^m m$

Deshacemos el cambio de variable:

$T(n) = c_{10} + c_{11} log_2(n)$

Aplicando la **regla de la suma**, es decir, eligiendo el máximo, obtenemos que el orden de eficiencia del algoritmo en el peor de los casos es $O(log(n))$.

2.3 Variables que influyen en el tamaño del caso (reestructurarRaiz)

El tamaño del problema viene determinado por el tamaño del vector "apo", la variable "pos" y la variable "tamapo". Aunque el tamaño del vector sea uno, las variables "pos" y "tamapo" son determinantes, ya que ellas indican cuánto se va a recorrer del vector. Se va a recorrer un camino del Árbol Parcialmente Ordenado, y cuanta más distancia exista entre "pos" y "tamapo" más largo será el camino. Se escoge siempre el camino con el nodo hijo mínimo.

2.4 Eficiencia teórica (reestructurarRaiz)

```
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);
    }
  }
}
```

Es un algoritmo recursivo. El algoritmo, en esencia, va comparando, de arriba hacia abajo del árbol, el mínimo de los dos nodos hijo con el nodo padre. Si la etiqueta de éste es menor que la etiqueta del nodo padre, intercambia sus valores. Existen tres casos:

- Primer caso: Cuando "(2*pos+1 >= tamapo)" (Sentencia 2 del pseudocódigo). La ejecución del algoritmo termina después de este caso, ya que no entra al "if" principal. La declaración es O(1), la y la evaluación de la condición es O(1), de forma que $max(O(1), O(1)) = O(1)$.

- Segundo caso: Cuando "(2*pos+1 < tamapo) AND (apo[pos] <= apo[minhijo])" (Sentencias 2 y 5 del pseudocódigo). La ejecución del algoritmo también termina ya que, aunque entra al "if" principal, no entra al segundo "if" (el cuál es el importante). La primera asignación es O(1). A continuación, nos encontramos con un "if" (cuyo objetivo es elegir al menor de los dos nodos hijo). El incremento de dentro del "if" es O(1) y la evaluación de la condición es O(1), de modo que $max(O(1), O(1)) = O(1)$. La evaluación de la condición del "if" al que no se puede entrar es O(1), de modo que, finalmente, tenemos $max(O(1), O(1)) = O(1)$. Es importante añadir que la variable "minhijo" va, aproximadamente, en el doble. Esto es importante para el siguiente caso.

- Tercer caso (caso general): Cuando "(2*pos+1 < tamapo) AND (apo[pos] < apo[minhijo])" (Sentencias 2 y 5 del pseudocódigo). Dispone de lo siguiente:
  - La primera sentencia de este último "if" es una declaración y asignación de una variable, de forma que es O(1).
  - La segunda y la tercera sentencia son asignaciones de una variable, de forma que son O(1).
  - Finalmente hacemos la llamada recursiva, pasándole como parámetros el vector, "minhijo" y "tamapo". Cómo he indicado en el punto anterior, minhijo va aumentando en el doble, de forma que la llamada recursiva resuelve un subproblema desde "minhijo" hasta "tamapo". Si T(n) es el tiempo que tarda el algoritmo en resolver un problema de tamaño n, entonces la llamada recursiva tendrá un tiempo de ejecución T(n/2).

Concluimos en que obtenemos una Recurrencia Lineal No Homogénea, la cuál sería la siguiente:

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

Como la ecuación no tiene las variables de la forma requerida por la ecuación característica, tenemos que hacer el siguiente cambio de variable:

$$n = 2^m \; ; \; m = log_2 n$$

De forma que la ecuación quedaría así:

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

Despejamos, y quedaría así:

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

Resolvemos la parte homogénea:

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

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

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

Obteniendo la parte homogénea del polinomio característico: $P_H(x) = (x - 1)$

Para resolver la parte no homogénea, debemos conseguir un escalar b_1 y un polinomio q_m :

$1 = {b_1}^m q_1(m)^{d_1}$

donde podemos ver que $b_1 = 1$ y $q_1(m) = 1$, donde el grado del polinomio es $d_1 = 0$

El polinomio característico se obtiene como: $P(x) = P_H(x)(x - b_1)^{d_1+1} = (x - 1)(x - 1) = (x - 1)^2$

Tenemos $r = 1$ (ya que solo tenemos una raíz diferente), con valor $R_1 = 1$ y multiplicidad $M_1 = 2$

Si aplicamos la fórmula de la ecuación característica, tenemos que el tiempo de ejecución (expresado en términos de 2^m) es el siguiente:

$T(2^m) = \sum_{i=1}^{m} \sum_{j=0}^{M_i - 1} c_{ij} {R_i}^m m^j = c_{10} 1^m + c_{11} 1^m m$

Deshacemos el cambio de variable

$T(n) = c_{10} + c_{11} log_2(n)$

Aplicando la **regla de la suma**, es decir, eligiendo el máximo, obtenemos que el orden de eficiencia del algoritmo en el peor de los casos es $O(log(n))$.

2.5 **Eficiencia práctica (insertarEnPos)**

Tam. Caso	Tiempo (ns)
100000000	2204
200000000	2355
300000000	2555
400000000	2676
500000000	2755
600000000	2845
700000000	2845
800000000	2835
900000000	2735
1000000000	2725
1100000000	2966
1200000000	2885
1300000000	2846
1400000000	2975
1500000000	3086
1600000000	2935
1700000000	2986
1800000000	3025

2.6 **Eficiencia práctica (reestructurarRaiz)**

Tam. Caso	Tiempo (ns)
100000000	3647
200000000	3727
300000000	4208
400000000	4157
500000000	4238
600000000	4227
700000000	4298
800000000	4448
900000000	4309
1000000000	4247
1100000000	4439
1200000000	4468
1300000000	4569
1400000000	4668
1500000000	4549
1600000000	4699
1700000000	4548
1800000000	4520

2.7 **Eficiencia híbrida (insertarEnPos)**

K=Tiempo/f(n)	Tiempo teórico estimado= K*f(n)
82,93376381	2510,960002
85,40212963	2605,444287
90,73028636	2660,714051
93,64690193	2699,928572
95,33745232	2730,345718
97,56386205	2755,198335
96,82542057	2776,210924
95,85661146	2794,412857
91,94714542	2810,468099
91,14519313	2824,830003
98,75192885	2837,821925
95,65527044	2849,68262
94,00227043	2860,593403
97,91730813	2870,695209
101,2390287	2880,099766
95,99212515	2888,897142
97,38156559	2897,161006
98,38886282	2904,952384
K promedio	94,48428482

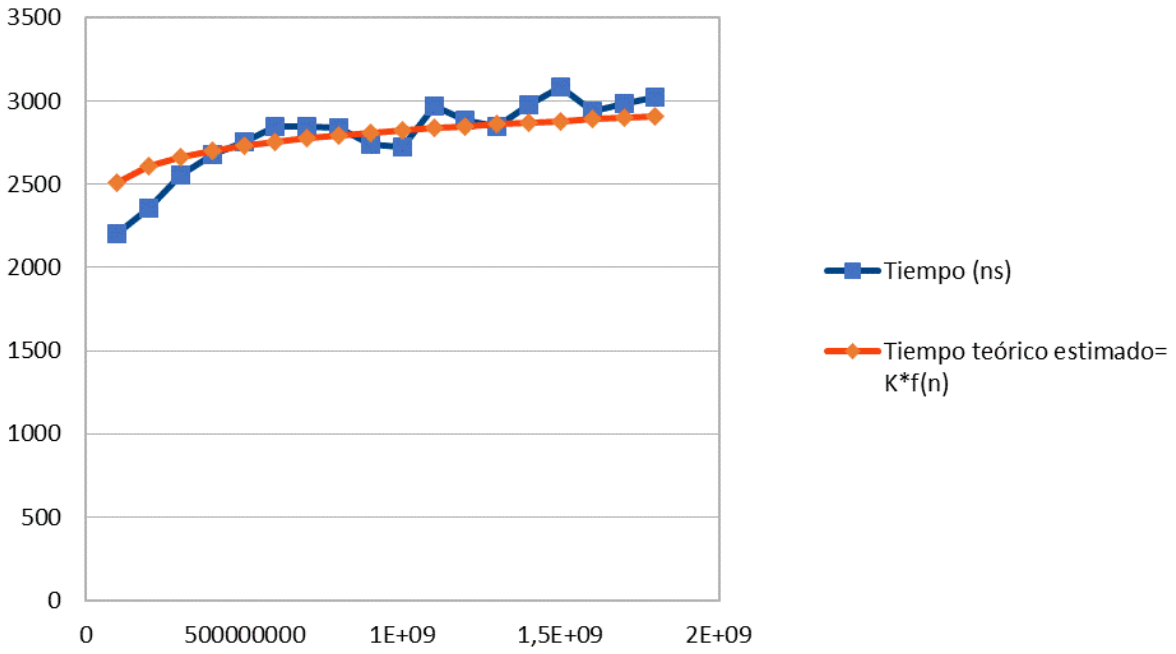

Fig. 2. Comparación entre tiempo práctico y tiempo teórico estimado del algoritmo insertarEnPos

Conclusión: Al medir en nanosegundos, el ruido causado por la precisión de medir en esta unidad se hace mucho más notoria, lo que causa, que, la gráfica del tiempo práctico se sitúe, en muchos casos, por encima de la gráfica del tiempo teórico estimado. Sin embargo, sigue claramente la tendencia de la misma, de forma que podemos confirmar que esto es a causa del ruido. Por otra parte, podemos observar que, tanto el tiempo empírico como el teórico estimado, recorren una función $log(n)$, lo cual nos confirma que el proceso de análisis se ha realizado correctamente.

2.8 **Eficiencia híbrida (reestructurarRaiz)**

K=Tiempo/f(n)	Tiempo teórico estimado= K*f(n)
137,2320493	3899,557536
135,1565763	4046,29301
149,4297632	4132,127759
145,474653	4193,028483
146,6570319	4240,266754
144,9569226	4278,863233
146,2761538	4311,49609
150,395135	4339,763957
144,8629797	4364,697982
142,0527102	4387,002228
147,7949468	4407,178873
148,1413339	4425,598706
150,9122887	4442,54331
153,639662	4458,231563
149,2340705	4472,836978
153,6855183	4486,49943
148,3226257	4499,333331
147,0141025	4511,433456
K promedio	146,7354735

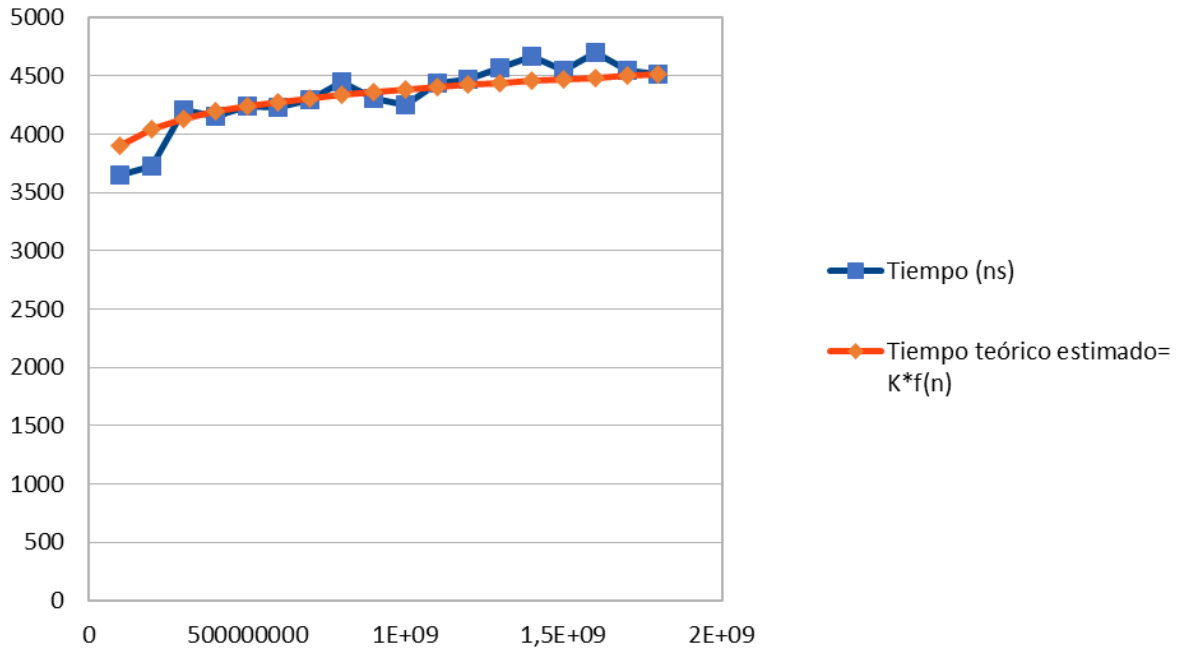

Fig. 3. Comparación entre tiempo práctico y tiempo teórico estimado del algoritmo reestructurarRaiz

Conclusión: Al medir en nanosegundos, el ruido causado por la precisión de medir en esta unidad se hace mucho más notoria, lo que causa, que, la gráfica del tiempo práctico se sitúe, en muchos casos, por encima de la gráfica del tiempo teórico estimado. Sin embargo, sigue claramente la tendencia de la misma, de forma que podemos confirmar que esto es a causa del ruido. Por otra parte, podemos observar que, tanto el tiempo empírico como el teórico estimado, recorren una función $log(n)$, lo cual nos confirma que el proceso de análisis se ha realizado correctamente.

3 Algoritmo 3: HeapSort

3.1 Variables que influyen en el tamaño del caso

El tamaño del problema viene determinado por el parámetro “n”, que indica cuántas veces se van a recorrer los bucles y el tamaño del vector.

3.2 Eficiencia teórica

```
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;
}
```

Calculamos el algoritmo HeapSort como un algoritmo iterativo, ya que tenemos la eficiencia de los algoritmos insertarEnPos y reestructurarRaiz, los cuáles son recursivos. Lo analizaremos de dentro hacia fuera.

Dentro del primer bucle “for” tenemos una asignación, la cuál es O(1), el incremento de una variable, el cuál es O(1), y la ejecución del algoritmo insertarEnPos, la cuál, como ya hemos visto anteriormente, tiene una eficiencia de O(log(n)). De forma que $max(O(1), O(1), O(log(n))) = O(log(n))$. Además el bucle se ejecuta siempre “n” veces, de forma que $n * (O(log(n)) = O(n * log(n))$.

Dentro del segundo bucle “for” tenemos una asignación, la cuál es O(1), el decremento de una variable, el cuál es O(1), y la ejecución del algoritmo reestructurarRaiz, la cuál, como ya hemos visto anteriormente, tiene una eficiencia de O(log(n)). De forma que $max(O(1), O(1), O(log(n))) = O(log(n))$. Además el bucle se ejecuta siempre “n” veces, de forma que $n * (O(log(n)) = O(n * log(n))$.

Las primeras dos sentencias de algoritmo consisten en la declaración de un vector y una variable, la cuáles son O(1). La última sentencia consiste en la llamada al operador “delete” para desasignar memoria dinámica. Como no sabemos cómo está implementado, pensamos en el peor de los casos, es decir, que consista en un bucle “for” que recorra “apo” y vaya eliminando sus valores, siendo su eficiencia de O(n).

Finalmente, podemos comprobar que $max(O(1), O(1), O(n * log(n), O(n * log(n), O(n) = O(n * log(n))$. Obtenemos que el orden de eficiencia del algoritmo en el peor de los casos es $O(n * log(n))$.

3.3 **Eficiencia práctica**

Tam. Caso	Tiempo (us)
1000	517
2000	1030
3000	1556
4000	2153
5000	2774
6000	1216
7000	1280
8000	1491
9000	1682
10000	2875
20000	4093
30000	6353
40000	8731
50000	11180
60000	13761
70000	16534
80000	18858
90000	21404
100000	24234

3.4 **Eficiencia híbrida**

K=Tiempo/f(n)	Tiempo teórico estimado= K*f(n)
0,172333333	150,920917
0,156011912	332,1296494
0,149165539	524,7703283
0,149428383	724,8349294
0,149987699	930,4199085
0,053641756	1140,404103
0,047555912	1354,046685
0,047750629	1570,82112
0,04726288	1790,333717
0,071875	2012,278894
0,047581626	4327,43594
0,047299739	6756,912454
0,047429847	9260,628187
0,047584896	11819,5477
0,047999736	14422,45937
0,048750304	17061,95492
0,048076825	19732,76899
0,048003751	22430,96468
0,048468	25153,48617
K promedio	0,050306972

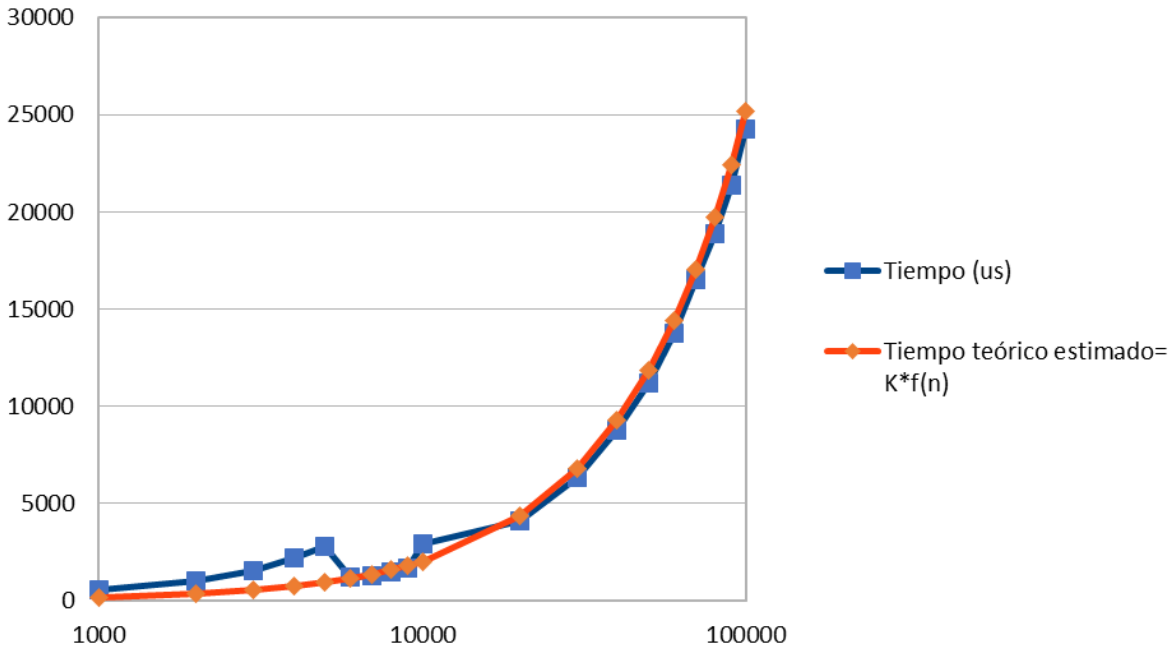

Fig. 4. Comparación entre tiempo práctico y tiempo teórico estimado del algoritmo HeapSort

Conclusión: El tiempo práctico se encuentra por debajo del tiempo teórico estimado (exceptuando los primeros casos), el cuál es el tiempo teórico utilizando la constante oculta de nuestro ordenador. Además, sigue la tendencia de una función $nlog(n)$, lo cual nos confirma que el proceso de análisis se ha realizado correctamente.

4 Comparación en eficiencia teórica e híbrida de los algoritmos Burbuja, MergeSort y HeapSort

Burbuja:

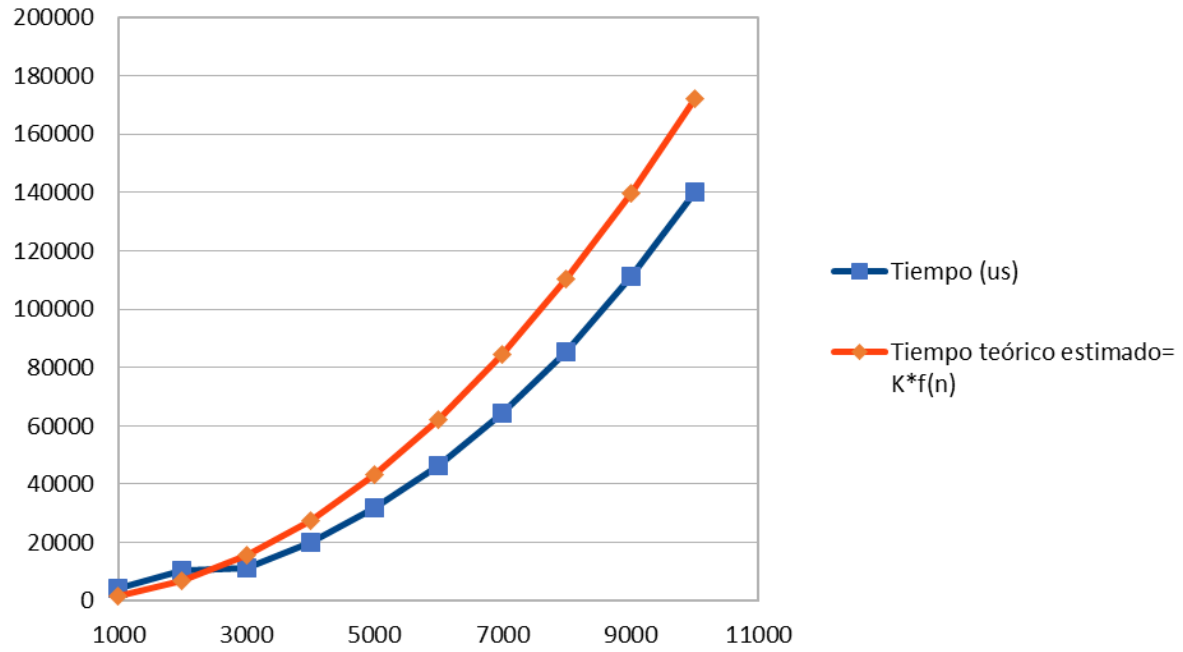

Fig. 5. Comparación entre tiempo práctico y tiempo teórico estimado del algoritmo Burbuja

Mergesort:

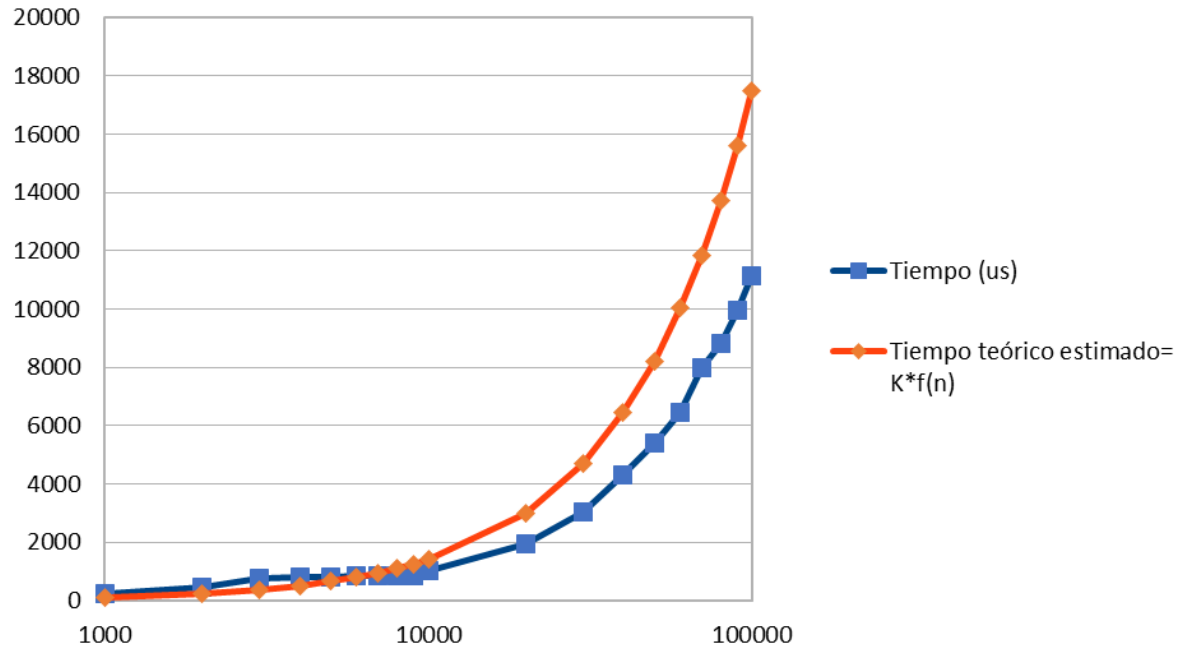

Fig. 6. Comparación entre tiempo práctico y tiempo teórico estimado del algoritmo MergeSort

Heapsort:

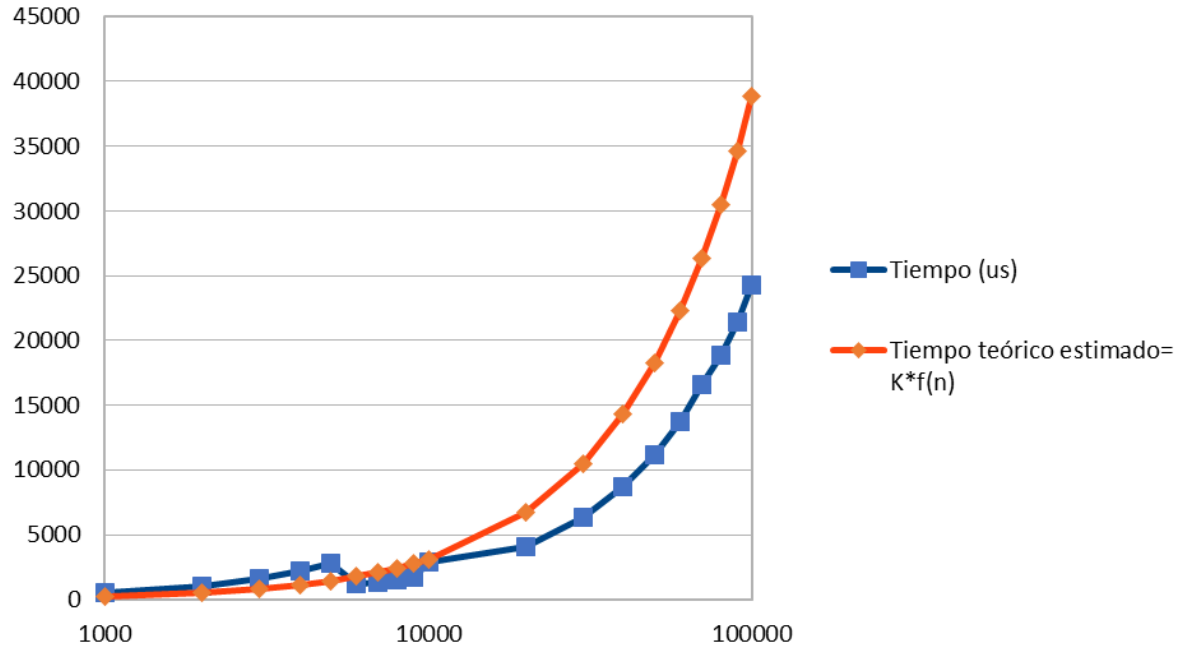

Fig. 7. Comparación entre tiempo práctico y tiempo teórico estimado del algoritmo HeapSort

Gráfica Comparativa:

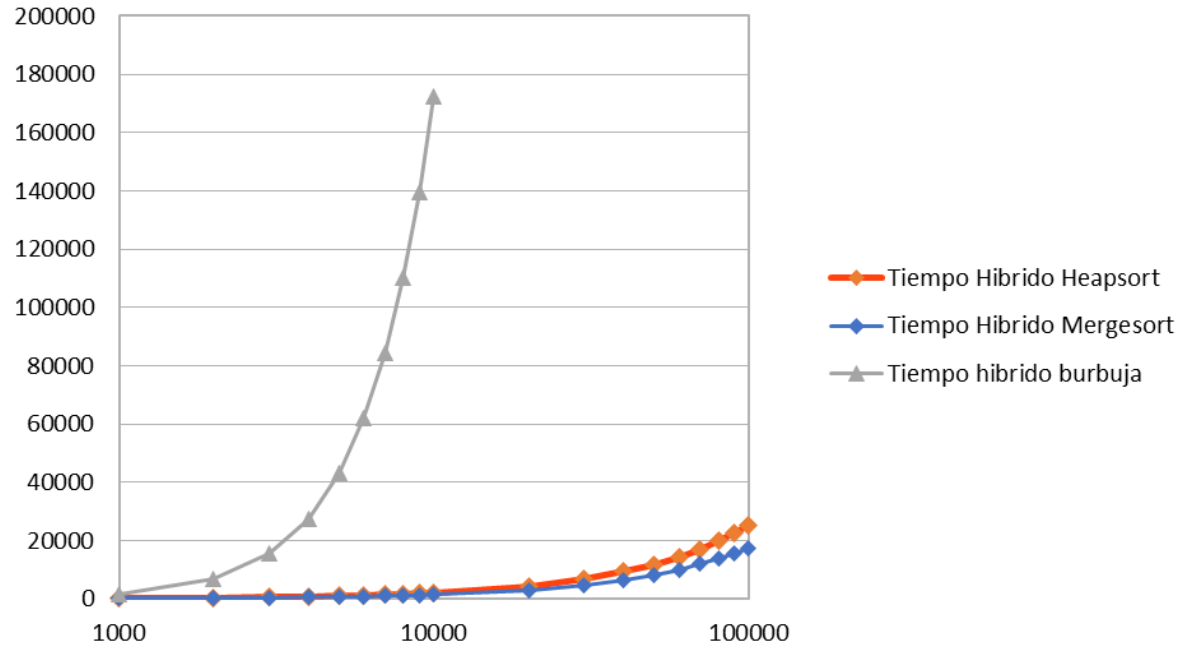

Fig. 8. Tabla comparativa de los algoritmos Burbuja, MergeSort y HeapSort

Como podemos observar, en términos generales, el algoritmo MergeSort es el más eficiente, aunque muy empatado con el HeapSort. El algoritmo Burbuja, en el caso de nuestro ordenador, no es más eficiente que ninguno de los otros dos en ningún caso (tampoco para casos pequeños), y hemos supuesto que esto se debe a la constante oculta, ya que la potencia de los ordenadores actuales ha cambiado considerablemente.