PRÁCTICA 1:
CÁLCULO DE LA EFICIENCIA DE ALGORITMOS

Carmona Molina, José Antonio.
Manzano Mata, Nuria.
Rodríguez Rodríguez, Antonio.

Algoritmo 1:

```
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
```

1. Cálculo eficiencia teórica:

El algoritmo dado, buscará el elemento máximo y el elemento mínimo en un subvector, cuyo tamaño delimitamos con las variables cini y cfin, del vector inicial A.

Teniendo en cuenta que el **tamaño del problema** dependerá de las variables cini y cfin, el **peor caso posible** que comience la posición inicial cini en 1 y que la posición final acabe en cfin = N, con lo que el problema debería de recorrer N-1 posiciones, siendo el subvector en el que buscamos, el máximo y el mínimo, el vector inicial completo.

Es un **algoritmo recursivo**. Por tanto, llamemos T(n) al tiempo de ejecución que tarda el algoritmo MaximoMinimoDyV en resolver un problema de tamaño n. Viendo nuestro algoritmo tendremos dos casos (uno general y uno base):

- Caso base: Se da cuando cini es mayor o igual a cfin, en el caso de que resulten ser iguales nos encontraremos con una eficiencia O(1), al tener tan solo una simple asignación. Por otro lado, para el caso en el que cfin < cini, tendríamos de nuevo operaciones simples en comparativas de if, quedándonos por tanto en el peor caso posible en el que la amplitud entre cini y cfin es máxima, que la eficiencia es O(1).

- Caso general: Se da cuando cini < cfin -1. En este caso:
    - El algoritmo calcula la parte central del subvector en “mitad”. Esta operación es O(1) porque todas las sentencias son operaciones simples.

- Luego hace una **llamada recursiva**, para resolver un subproblema desde cini hasta mitad, cuyo **tamaño es n/2**. Si T(n) es el tiempo que tarda el algoritmo para resolver el problema de tamaño n, entonces la llamada recursiva tendrá un tiempo de ejecución **T(n/2)**.
- Luego hace **otra llamada recursiva**, para resolver un subproblema desde mitad+1 hasta cfin, cuyo tamaño asintótico también se puede aproximar por n/2. Al igual que en la línea anterior, esta llamada recursiva tendrá un tiempo de ejecución **T(n/2)**.
- Por último, como habremos obtenido dos máximos y dos mínimos de cada mitad del subvector, bastará con una simple condición, podremos quedarnos con aquel que sea mayor y menor respectivamente, teniendo esto una eficiencia de orden O(1), al tratarse de operaciones simples.

Por tanto, podemos aproximar T(n) en el caso general como **T(n)= 2T(n/2)+1**. Hay que resolver la ecuación en recurrencias para obtener el orden de ejecución de MaximoMinimoDyV.

Como la ecuación no tiene las variables de la forma requerida por la ecuación característica, tenemos que hacer un cambio de variable. En este caso, 2^m , quedando:

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

Llevamos las "T's" a la izquierda, y obtenemos que es una ecuación lineal no homogénea:

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

Resolvemos la "**parte homogénea**":

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

Y obtenemos la parte homogénea del polinomio característico como PH(X) = (X-2)

Para resolver la "**parte no homogénea**", debemos conseguir un escalar "b1 " y un polinomio "q1(m)". De modo que:

$$1 = b1\cdot q1(m)$$

Con lo que deducimos que b1 = 1 y q1(m) tendrá un grado d1 = 0.
El **polinomio característico** se obtiene como: P(X)=PH(X)=(X-2)(X-1)= (X-2)(X-1)

De este modo, tenemos r=2 dos raíces diferentes, con valores 1 y 2; y multiplicidades M = 1 en ambos casos.

Aplicando la fórmula de la **ecuación característica**, tenemos que el tiempo de ejecución (en la variable m que teníamos) se expresa como:

$$T(2^m) = \sum_{i=1}^{r} \sum_{j=0}^{M-1} C_{ij} Ri^m m^j = c_{10} 2^m + c_{20} 1^m$$

Ahora **deshacemos el cambio de variable** para volver al espacio de tiempos inicial:

$$T(n) = c_{10} n + c_{20}$$

Aplicando la **regla del máximo** a la ecuación anterior, obtenemos que el orden de eficiencia del algoritmo en el peor de los casos es **O(n)**.

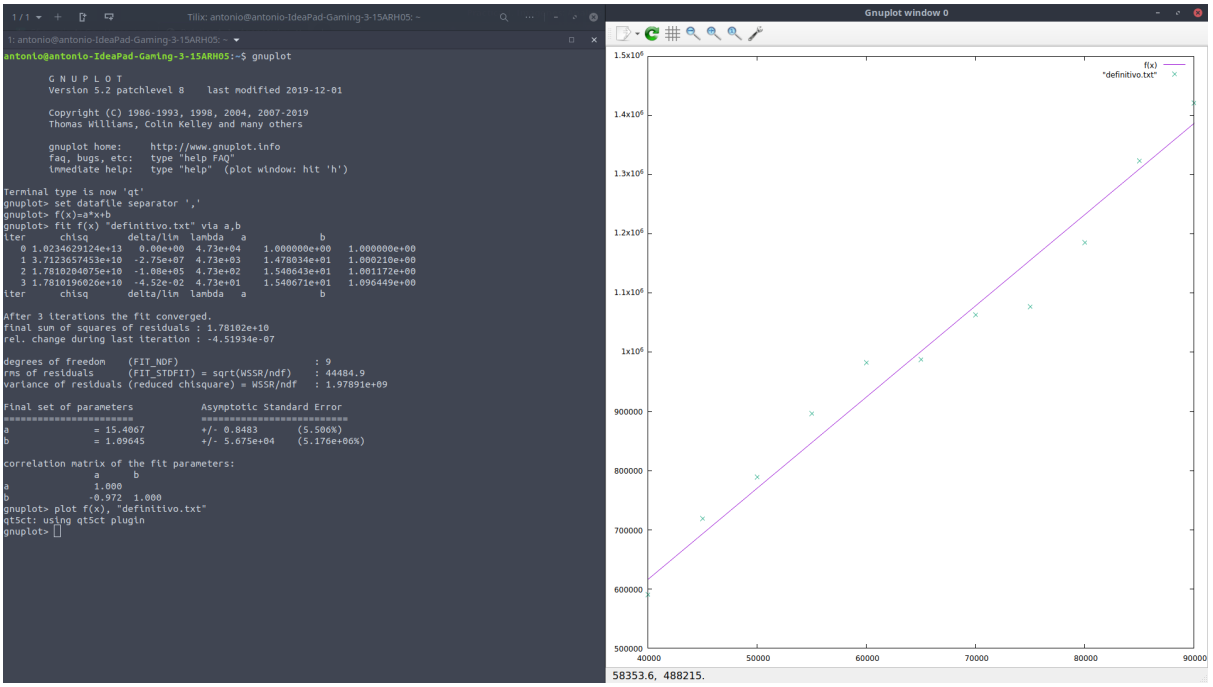
2. Cálculo eficiencia práctica:

En primer lugar, tuvimos que implementar la función basándonos en la explicación dada, la cual quedó de la siguiente forma:

```
pair<int, int> MaximoMinimoDyV(int * A, int Cini, int Cfin) {
    pair<int, int> resultados = {A[Cini],A[Cfin]};

    if(Cini < Cfin-1) {
        int mitad = (Cini + Cfin)/2;
        pair<int, int> resultados1 = MaximoMinimoDyV(A, Cini, mitad);
        pair<int, int>resultados2 = MaximoMinimoDyV(A, mitad + 1, Cfin);

        if(resultados1.first > resultados2.first)
            resultados.first = resultados1.first;
        else
            resultados.first = resultados2.first;

        if(resultados1.second < resultados2.second)
            resultados.second = resultados1.second;
        else
            resultados.second = resultados2.second;

    } else if (Cini == Cfin) {
        resultados.first = A[Cini];
        resultados.second = A[Cini];
    } else {

        if(A[Cini] > A[Cfin])
            resultados.first = A[Cini];
        else
            resultados.first = A[Cfin];

        if(A[Cini] <= A[Cfin])
            resultados.second = A[Cini];
        else
            resultados.second = A[Cfin];

    }

    return resultados;
}
```

Para el cálculo de la eficiencia práctica, creamos un programa basándonos en lo aprendido en la asignatura de Estructura de Datos y los programas de ejemplo para calcular la eficiencia de Burbuja y MergeSort (cuya implementación dejamos al final de la memoria).
Tan solo iremos variando la línea que hace la llamada a la función (dependiendo de la función de la que queremos calcular la eficiencia) entre el inicio y el fin del cronómetro, en este caso quedará como MaximoMinimoDyV(v, 0, n).

Una vez obtenidos los datos utilizamos gnuplot para generar la gráfica, de la manera que sigue:



3. Cálculo eficiencia híbrida:

a) Cálculo de constante oculta:

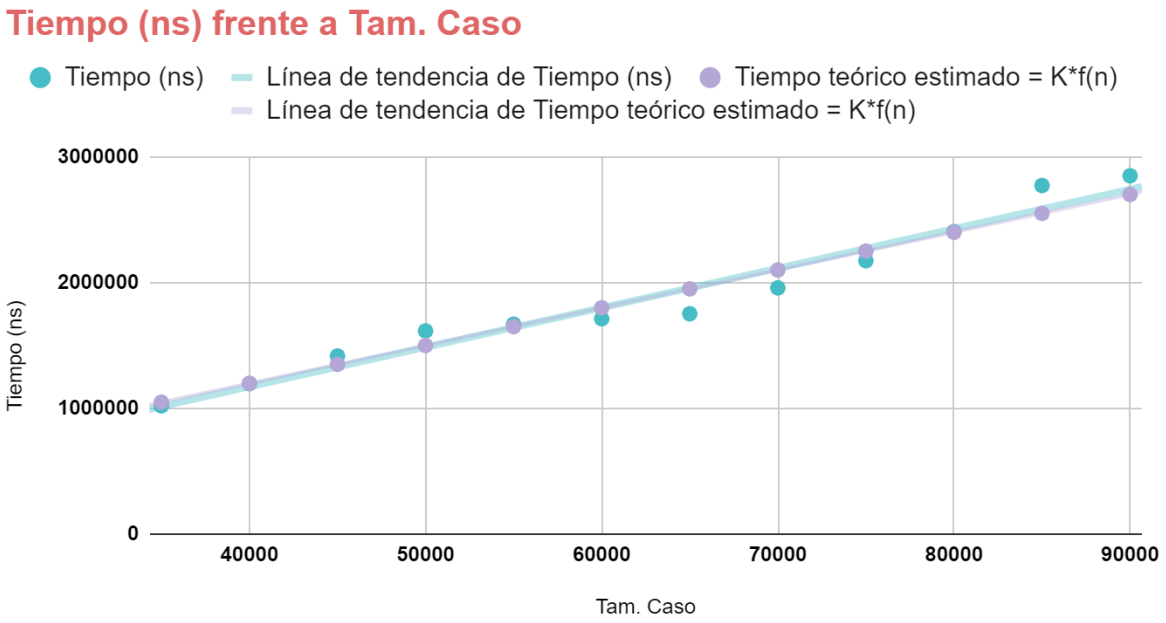
Teniendo en cuenta que para el cálculo de la constante oculta hay que hacer uso de la fórmula T(n) <= K*f(n), despejamos K y se calcula:

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

Tam. Caso	Tiempo (ns)	K=Tiempo/f(n)	Tiempo teórico estimado = K*f(n)
35000	1021260	29,17885714	1050568,399
40000	1197841	29,946025	1200649,599
45000	1417248	31,4944	1350730,799
50000	1616477	32,32954	1500811,999
55000	1670000	30,36363636	1650893,199
60000	1713552	28,5592	1800974,399
65000	1753254	26,97313846	1951055,599
70000	1958704	27,98148571	2101136,798
75000	2174769	28,99692	2251217,998
80000	2406415	30,0801875	2401299,198
85000	2772854	32,62181176	2551380,398
90000	2850271	31,66967778	2701461,598
	K promedio	30,01623998	

Se obtiene una **K promedio de 30,01623998**

b) Comparación gráfica:



En esta gráfica podemos observar como coinciden las líneas de tendencias del tiempo en ns (eficiencia gráfica) y el cálculo del tiempo teórico estimado con la constante oculta calculada en el apartado anterior, por lo tanto, el cálculo de ambos es correcto.

Algoritmo 2:

1. Cálculo eficiencia teórica:

Función 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);
    }
  }
}
```

El algoritmo dado, es un algoritmo de ordenación para un árbol APO, donde los valores mayores están en la posición más inferior (mayor profundidad).

Teniendo en cuenta que el **tamaño del problema** dependerá de las variables pos, en concreto el valor idx = pos - 1, el **peor caso posible** es que la segunda condición se cumpla, ya que, tendríamos una recursividad.

Es un **algoritmo recursivo**. Por tanto, llamemos T(n) al tiempo de ejecución que tarda el algoritmo insertarEnPos en resolver un problema de tamaño n. Viendo nuestro algoritmo tendremos dos casos (uno general y uno base):

- Caso base: Inicialmente se crea una variable de tipo entero “idx” al que se le asigna el valor pos -1, y otra con el nombre de “padre”. Posteriormente, aparece un bucle if que se lleva a cabo cuando el valor de idx es positivo, si dicho valor es par se le restan dos posiciones, se divide entre dos y se le asigna a la variable padre. En caso contrario, se realiza la misma asignación pero restando sólo una posición en lugar de dos. En este caso, concreto de árbol APO, nos permite acceder al nodo padre de la posición actual. Como en esta parte del código únicamente se realizan **operaciones simples** de asignación, el orden de eficiencia es O(1)
- Caso general: Se da en el segundo if, cuando apo[padre] > apo[idx]. En este caso:
    - Se inicializa una variable auxiliar que almacena el valor inicial del APO. Esta operación es O(1) porque todas las sentencias son operaciones simples.

- Luego hace de nuevo otra asignación apo[idx] = apo[padre], luego es una operación O(1) porque todas las sentencias son operaciones simples.
- La siguiente operación es de nuevo una asignación, por lo que, el orden es de nuevo O(1).
- Por último, hace una **llamada recursiva**, donde avanza una posición al padre. Si T(n) es el tiempo que tarda el algoritmo para resolver el problema de tamaño n, entonces la llamada recursiva tendrá un tiempo de ejecución **T(n/2)**.

Por tanto, podemos aproximar T(n) en el caso general como **T(n)= T(n/2)**. Hay que resolver la ecuación en recurrencias para obtener el orden de ejecución de insertarEnPos.

Como la ecuación no tiene las variables de la forma requerida por la ecuación característica, tenemos que hacer un cambio de variable. En este caso, $n=2^m$, quedando:

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

Llevamos las "T's" a la izquierda, y obtenemos que es una ecuación lineal no homogénea:

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

Resolvemos la **parte homogénea**:

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

Y obtenemos la parte homogénea del polinomio característico como PH(X) = (X-1)

Para resolver la "**parte no homogénea**", debemos conseguir un escalar "b1 " y un polinomio "q1(m)". De modo que:

$$0 = b1 \cdot q1(m)$$

Con lo que deducimos que b1 = 0 y q1(m) tendrá un grado d1 = 0.

El **polinomio característico** se obtiene como: P(X)=PH(X) = (X+1)(X - 0) = X(X-1)

De este modo, tenemos r=2 tenemos dos raíces, con valores 1 y 0; y multiplicidades M =1.

Aplicando la fórmula de la **ecuación característica**, tenemos que el tiempo de ejecución (en la variable m que teníamos) se expresa como:

$$T(2^m) = \sum_{i=1}^{r} \sum_{j=0}^{M-1} c_{ij} Ri^m m^j = c_{10} 1^m + c_{20} 1^m \, m$$

Ahora **deshacemos el cambio de variable** para volver al espacio de tiempos inicial:

$$T(n) = c_{10} + c_{20} \log(n)$$

Aplicando la regla del máximo a la ecuación anterior, obtenemos que el orden de eficiencia del algoritmo en el peor de los casos es **O(log(n))**.

a) **Función 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);
    }
  }
}
```

El algoritmo reestructurarRaiz, es un algoritmo de ordenación, donde se busca en un árbol APO pasado por referencia un valor mínimo entre sus hijos y reestructurar la raíz; intercambiandolo por un valor mayor a partir de una posición dada por parámetros.

Teniendo en cuenta que el **tamaño del problema** dependerá de la variable tamapo y el **peor caso posible** es que la segunda condición del primer if se cumpla, ya que, tendríamos una recursividad.

Es un **algoritmo recursivo**. Por tanto, llamemos T(n) al tiempo de ejecución que tarda el algoritmo reestructurarRaiz en resolver un problema de tamaño n. Viendo nuestro algoritmo tendremos dos casos (uno general y uno base):

- Caso base: Este caso se da cuando se cumple la condición de que 2*pos+1 < tamapo, posterior a la creación de una variable de tipo entero “minhijo” donde se albergará el valor de 2*pos+1 si se da dicha condición. Esto es de eficiencia O(1), ya que, solo se realizan operaciones simples de comparación y asignación. El siguiente if comprueba que la siguiente posición del árbol no se salga del tamaño del mismo, y si eso se da

además de que el valor del hijo de la derecha sea menor que el de la izquierda, este pasará a ser el minhijo. Finalmente, se realizará el intercambio con el padre si se dan las condiciones pertinentes como veremos en el caso general, pero hasta el momento la eficiencia es O(1).

Nota: el minhijo es necesario para no cometer el error de intercambiar el valor con el hijo incorrecto, ya que, si se intercambia un padre con el hijo mayor, sigue habiendo un fallo en la estructura debido a que el nuevo padre seguiría siendo mayor que uno de sus hijos.

- Caso general: Cuando nos encontramos dentro del primer if, en la segunda condición (apo[pos] > apo[minhijo]) estaríamos ante el caso general donde:
  - Se inicializa una variable auxiliar que almacena el valor de la posición pasada por parámetros del APO. Esta operación es O(1) porque todas las sentencias son operaciones simples.
  - Luego hace una asignación apo[pos] = apo[minhijo], que es una operación O(1) porque todas las sentencias son operaciones simples.
  - La siguiente operación es de nuevo una asignación, por lo que, el orden es de nuevo O(1).
  - Por último, hace una **llamada recursiva**, sin modificar los valores, ya que, durante el código se han ido actualizando. Si T(n) es el tiempo que tarda el algoritmo para resolver el problema de tamaño n, entonces la llamada recursiva tendrá un tiempo de ejecución T(n/2), dado a que, minhijo se multiplica por 2, lo que se posiciona en la mitad del APO.

  Por tanto, podemos aproximar T(n) en el caso general como **T(n) = T(n/2)**. Hay que resolver la ecuación en recurrencias para obtener el orden de ejecución de reestructurarRaiz .

  Como la ecuación no tiene las variables de la forma requerida por la ecuación característica, tenemos que hacer un cambio de variable. En este caso, $n=2^m$, quedando:

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

  Llevamos las "T's" a la izquierda, y obtenemos que es una ecuación lineal no homogénea:

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

  Resolvemos la "**parte homogénea**":

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

$$X^m - X^{m-1} = 0$$
$$X^{m-1} \cdot (X-1) = 0$$

Y obtenemos la parte homogénea del polinomio característico como PH(X) = (X-1)

Para resolver la "**parte no homogénea**", debemos conseguir un escalar "b1 " y un polinomio "q1(m)". De modo que:

$$0 = b1 \cdot q1(m)$$

Con lo que deducimos que b1 = 0 y q1(m) tendrá un grado d1 = 0.

El **polinomio característico** se obtiene como: P(X) = PH(X) = (X-1)(X - 0) = X(X-1)

De este modo, tenemos r=2 dos raíces, con valor 1 y 0; y multiplicidades M = 1.

Aplicando la fórmula de la **ecuación característica**, tenemos que el tiempo de ejecución (en la variable m que teníamos) se expresa como:

$$T(2^m) = \sum_{i=1}^{r} \sum_{j=0}^{M-1} C_{ij} Ri^m m^j = c_{10} 1^m + c_{20} 1^m m$$

Ahora **deshacemos el cambio de variable** para volver al espacio de tiempos inicial:

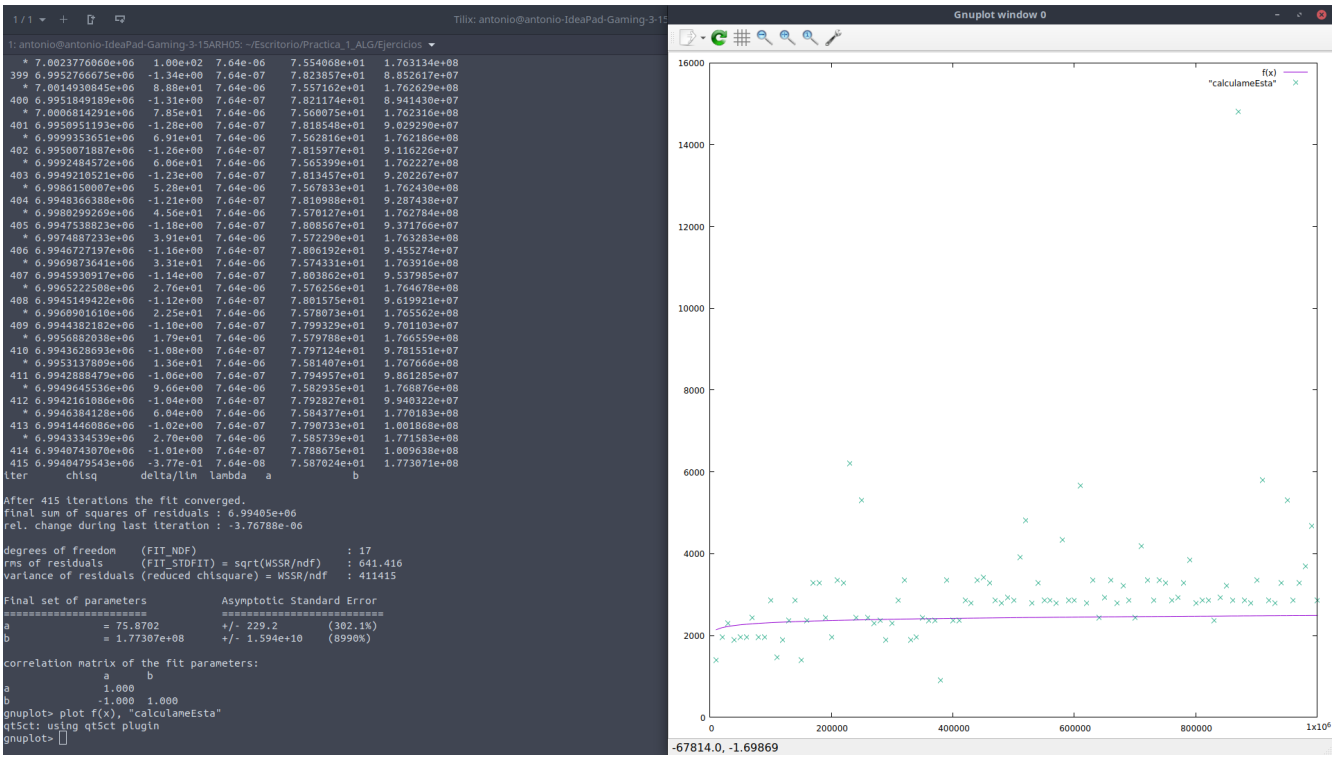
$$T(n) = c_{10} + c_{20} \log(n)$$

Aplicando la **regla del máximo** a la ecuación anterior, obtenemos que el orden de eficiencia del algoritmo en el peor de los casos es **O(log(n))**.

2. Cálculo eficiencia práctica:

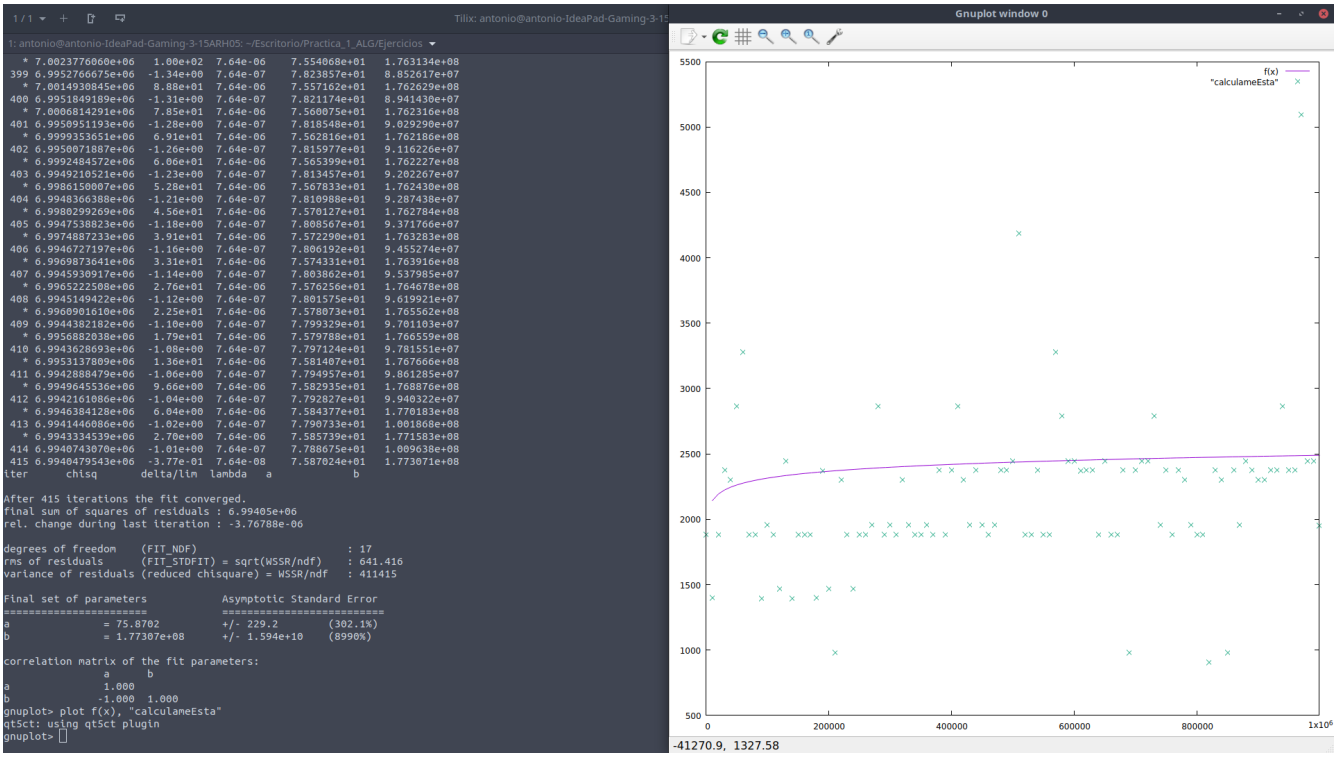
a) Función insertarEnPos:

Simplemente copiamos la función insertarEnPos y seguimos el procedimiento que hemos utilizado para calcular la eficiencia práctica de todos los algoritmos anteriores con el programa que indicamos al final de la memoria, sustituyendo la línea: insertarEnPos(v, int(n/2)).



b) Función reestructurarRaiz:

Simplemente copiamos la función reestructurarRaiz y seguimos el procedimiento que hemos utilizado para calcular la eficiencia práctica de todos los algoritmos anteriores con el programa que indicamos al final de la memoria, sustituyendo la línea: reestructurarRaiz(v, int(n/2), n).



3. Cálculo eficiencia híbrida:

a) Función insertarEnPos:

- **Cálculo de constante oculta:**

Teniendo en cuenta que para el cálculo de la constante oculta hay que hacer uso de la fórmula T(n) <= K*f(n), despejamos K y se calcula:

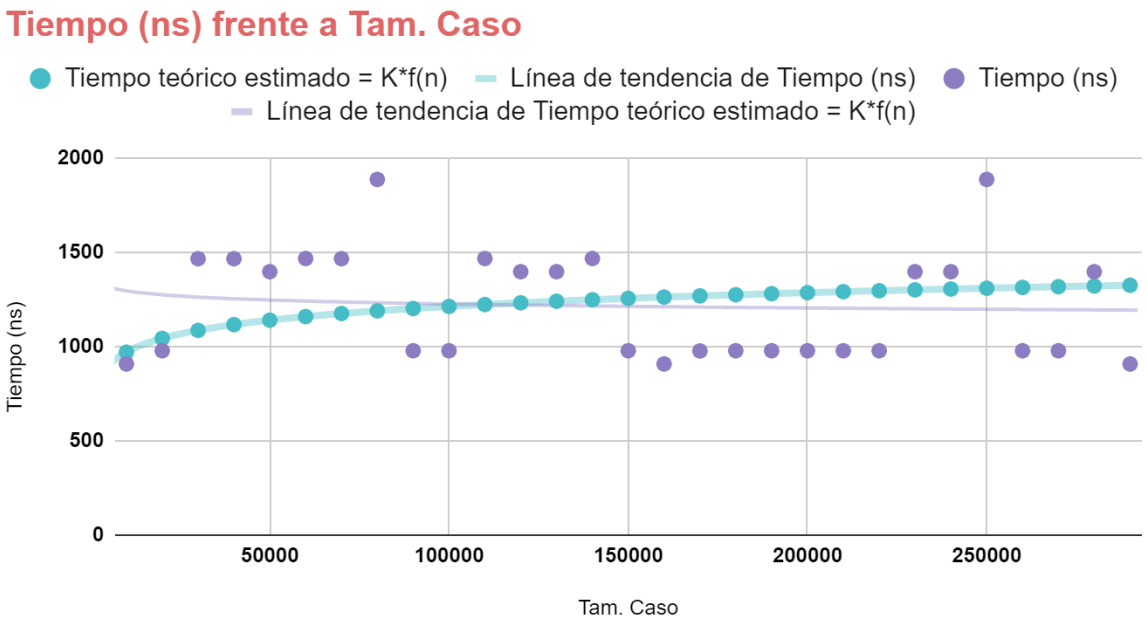
$$K=T(n)/f(n)$$

Tam. Caso	Tiempo (ns)	K=Tiempo/f(n)	Tiempo teórico estimado = K*f(n)
10000	908	227	970,3560224
20000	978	227,3873935	1043,38259
30000	1466	327,4425499	1086,100393
40000	1.466	318,5529964	1116,409157
50000	1.397	297,2991951	1139,918461
60000	1.467	307,0225121	1159,12696
70000	1.466	302,5738567	1175,367516
80000	1.886	384,6553918	1189,435724
90000	978	197,4065658	1201,844764
100000	978	195,6	1212,945028
110000	1.467	290,9910201	1222,986438
120000	1.397	275,0443295	1232,153528
130000	1397	273,1747115	1240,586432
140000	1467	285,0686943	1248,394083
150000	978	188,945664	1255,662831
160000	908	174,4771456	1262,462292
170000	977	186,79085	1268,849403
180000	978	186,0988177	1274,871331
190000	978	185,2710079	1280,567587
200000	978	184,4924478	1285,971595
210000	978	183,7579299	1291,111886
220000	978	183,0630144	1296,013006
230000	1.397	260,550338	1300,696224
240000	1.397	259,6552323	1305,180095
250000	1.886	349,3925455	1309,480899
260000	978	180,6103072	1313,613

270000	978	180,0652732	1317,589135
280000	1397	256,4640115	1321,42065
290000	908	166,2273603	1325,117698
	K promedio	242,5890056	

Se obtiene una **K promedio de 242,5890056**

- **Comparación gráfica:**



En esta gráfica podemos observar como los valores de la eficiencia práctica son más dispersos que la teórica, esto es debido a que los tamaños de casos escogidos son muy pequeños en comparación con la eficiencia práctica del algoritmo, por lo que sería necesario repetir la experimentación para valores más grandes (100000, 500000, 1000000, etc.) de modo que, se aprecie debidamente la tendencia de log(n). El cálculo del tiempo teórico estimado con la constante oculta calculada en el apartado anterior, si se aprecia mucho mejor la tendencia logarítmica del algoritmo log(n).

b) <u>Función reestructurarRaiz:</u>

- **Cálculo de constante oculta:**

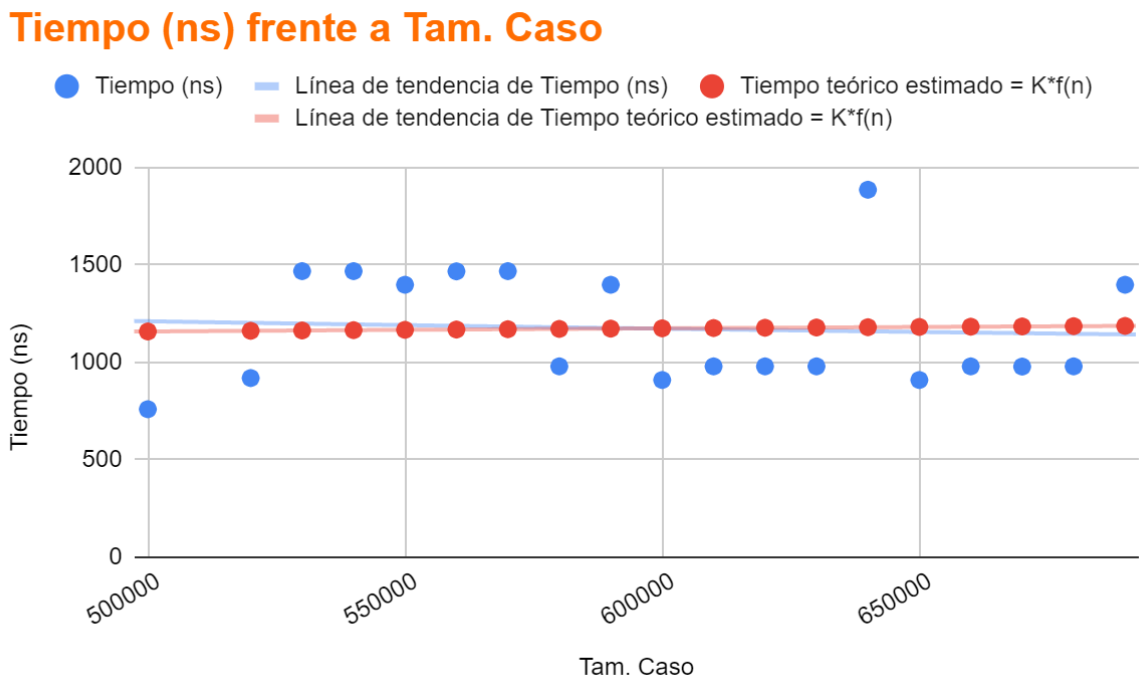
Teniendo en cuenta que para el cálculo de la constante oculta hay que hacer uso de la fórmula T(n) <= K*f(n), despejamos K y se calcula:

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

Tam. Caso	Tiempo (ns)	K=Tiempo/f(n)	Tiempo teórico estimado = K*f(n)
500000	758	133,0064905	1157,209148
520000	918	160,6017255	1160,667867
530000	1467	256,2769568	1162,347651
540000	1467	255,914032	1163,996036
550000	1397	243,3644136	1165,614174
560000	1466	255,0368904	1167,203154
570000	1467	254,8700305	1168,76401
580000	978	169,6906773	1170,297719
590000	1397	242,0786389	1171,805209
600000	908	157,1436885	1173,287362
610000	978	169,04827	1174,745016
620000	978	168,842173	1176,178966
630000	978	168,6398637	1177,589973
640000	1885	324,6540769	1178,978758
650000	908	156,2039453	1180,346011
660000	978	168,0544061	1181,692389
670000	977	167,6943797	1183,018519
680000	978	167,6808411	1184,325003
690000	1397	239,2594805	1185,612413
	K promedio	203,0558411	

Se obtiene una **K promedio de 203,0558411**

- **Comparación gráfica:**



En esta gráfica podemos observar como los valores de la eficiencia práctica son más dispersos que la teórica, esto es debido a que los tamaños de casos escogidos son muy pequeños en comparación con la eficiencia práctica del algoritmo, por lo que sería necesario repetir la experimentación para valores más grandes (100000, 500000, 1000000, etc.) de modo que, se aprecie debidamente la tendencia de log(n). El cálculo del tiempo teórico estimado con la constante oculta calculada en el apartado anterior, si se aprecia mucho mejor la tendencia logarítmica del algoritmo log(n).

Algoritmo 3:

1. Cálculo 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;
}
```

El algoritmo HeapSort, es un algoritmo de ordenación de un vector pasado por referencia apoyándose de una estructura de árbol parcialmente ordenado, generando un vector "apo", dónde reserva la memoria n pasada por parámetro. En el primer bucle, se insertan los valores de v en el APO y hace uso de la función insertarEnPos, organizando los valores que va insertando con estructura APO. En el siguiente bucle, va insertando la raíz del APO auxiliar y lo reestructura de manera que se cumpla la geometría de la estructura APO mediante la función recursiva "reestructurarRaíz". Finalmente, destruye el APO auxiliar para limpiar memoria.

Teniendo en cuenta que el **tamaño del problema** dependerá de la variable n, el **peor caso posible** es que el valor más pequeño del APO, es decir, lo que equivaldría a la raíz, se encontrara en la posición más profunda de dicho árbol, por lo que, el segundo bucle for, debe recorrerlo completamente hasta llevarlo a la raíz.

Dicho algoritmo **no sería directamente recursivo**, es decir, la recursividad vendría implícita en la llamada a las funciones insertarEnPos y reestructurarRaiz, ya que, como hemos analizado en los apartados anteriores, estos algoritmos son recursivos, por lo tanto, su llamada tendrá implícita una recursividad en el caso de que cumplan dichas condiciones dentro de su código.

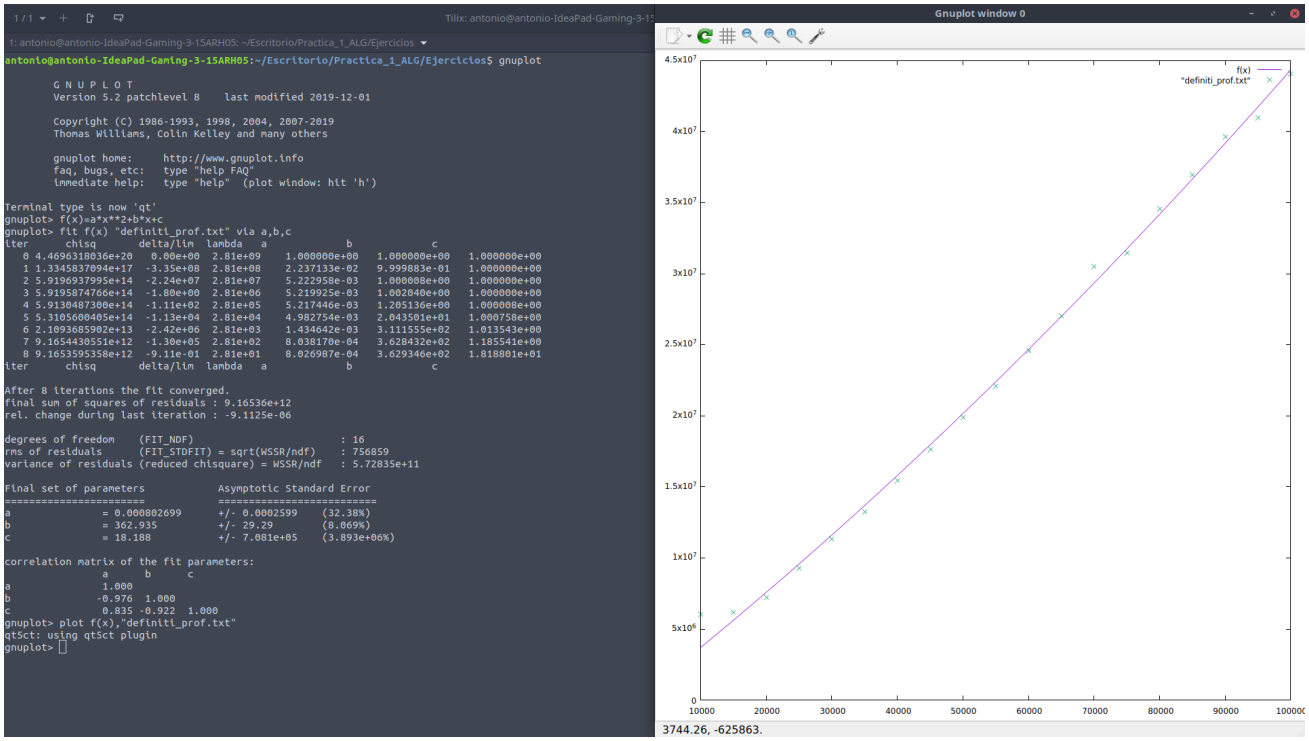
En conclusión a lo anterior, solo existe un caso base que sería todo el código en sí, donde:

- Comenzamos por la declaración de variables y asociaciones cuya operación es O(1) porque todas las sentencias son operaciones simples.
- El **primer bucle** for que recorre todo el vector insertando los valores uno a uno a la estructura APO, cuyo valor es de **O(n)**, pero, cuando llama a la función recursiva insertarEnPos cuya orden estudiada en el apartado anterior es de **O(log(n))**, modifica el orden de eficiencia del bucle completo a **O(n*log(n))**.
- El **segundo bucle** for recorre el árbol APO desde el hijo más profundo hasta la raíz donde el valor de la eficiencia es **O(n)**, además, a su vez, va reestructurando la raíz llamando a la función recursiva reestructurarRaiz, con el fin de que en la raíz siempre se encuentre el valor más pequeño de todo el APO; por lo que, el valor de eficiencia es de **O(log(n))**. Finalmente, quedaría una eficiencia completa del bucle como **O(n*log(n))**.

Por último, aplicando la **regla del máximo** a las eficiencias de cada uno de los bucles, obtenemos que el orden de eficiencia del algoritmo en el peor de los casos es **O(n·log(n))**.

2. Cálculo eficiencia práctica:

Simplemente copiamos la función HeapSort y seguimos el procedimiento que hemos utilizado para calcular la eficiencia práctica de todos los algoritmos anteriores con el programa que indicamos al final de la memoria, sustituyendo la línea: HeapSort(v, n).



3. Cálculo eficiencia híbrida:

a) Cálculo de constante oculta:

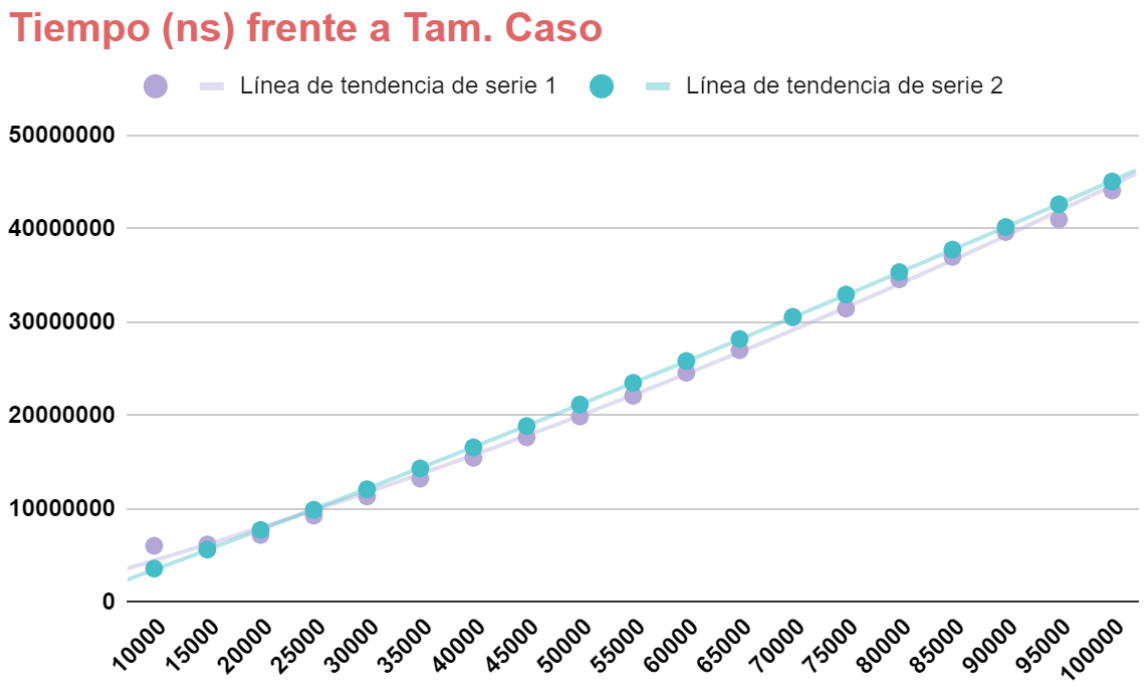
Teniendo en cuenta que para el cálculo de la constante oculta hay que hacer uso de la fórmula T(n) <= K*f(n), despejamos K y se calcula:

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

Tam. Caso	Tiempo (ns)	K=Tiempo/f(n)	Tiempo teórico estimado = K*f(n)
10000	6049710	151,24275	3603898,502
15000	6195678	98,90712975	5643828,388
20000	7213962	83,86319099	7750237,78
25000	9277838	84,38348847	9906080,881
30000	11332564	84,37388935	12101317,94
35000	13241672	83,2587633	14329315,02
40000	15468627	84,03099389	16585357,11
45000	17658355	84,33053245	18865918,81
50000	19889640	84,655318	21168263,7
55000	22110031	84,80375658	23490205,75
60000	24569416	85,70056602	25829958,2
65000	26991158	86,27800956	28186033,26
70000	30491456	89,9036975	30557172,75
75000	31439200	85,98647498	32942298,72
80000	34582751	88,16570543	35340477,32
85000	36973553	88,24224174	37750891,65
90000	39632196	88,88497737	40172821,11
95000	40984460	86,66920919	42605625,28
100000	44085547	88,171094	45048731,28
	K promedio	90,09746256	

Se obtiene una **K promedio de 90,09746256**

b) Comparación gráfica:



En esta gráfica podemos observar como coinciden prácticamente las líneas de tendencias del tiempo en ns (eficiencia gráfica) y el cálculo del tiempo teórico estimado con la constante oculta calculada en el apartado anterior, aunque no se aprecia tan bien como en el algoritmo burbuja. Esto es debido a que los tamaños de casos escogidos aunque son muy grandes, en comparación con la eficiencia práctica del algoritmo, no se llega a apreciar del todo la tendencia de n*log(n).

Comparación entre la eficiencia teórica e híbrida entre los algoritmos de ordenación por Burbuja, MergeSort y HeapSort:

La eficiencia teórica del algoritmo de ordenación por **Burbuja** es de O(n^2). La híbrida es:

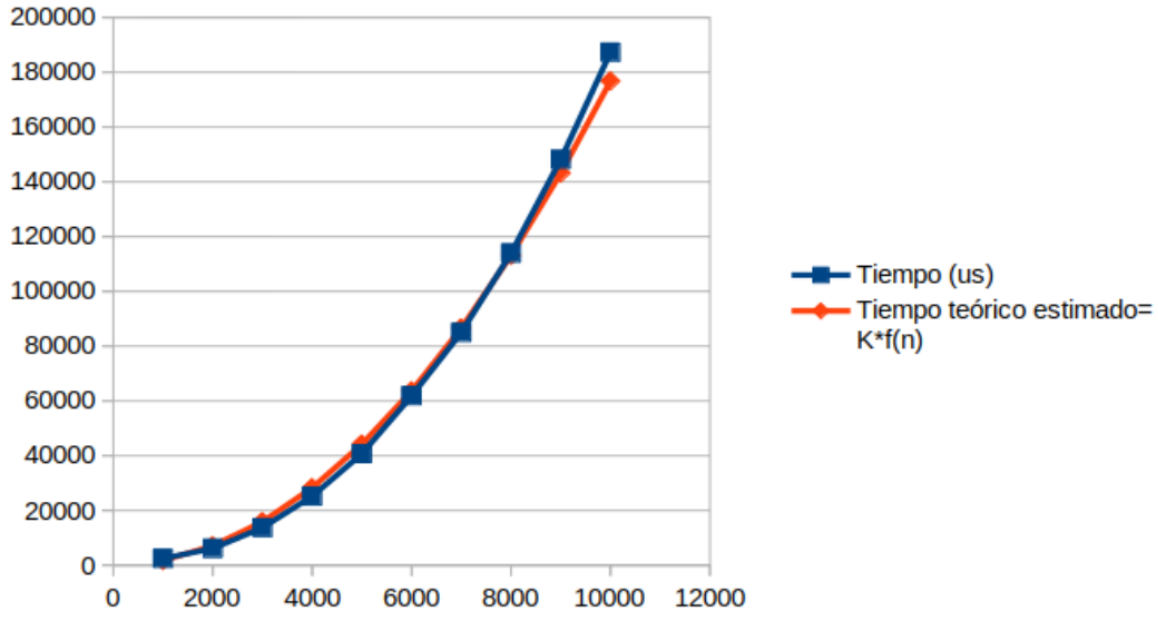


Figura 6: Tiempo de ejecución real vs teórico en burbuja

La eficiencia teórica del algoritmo de ordenación por **MergeSort** es de O(nlog(n)). La híbrida es:

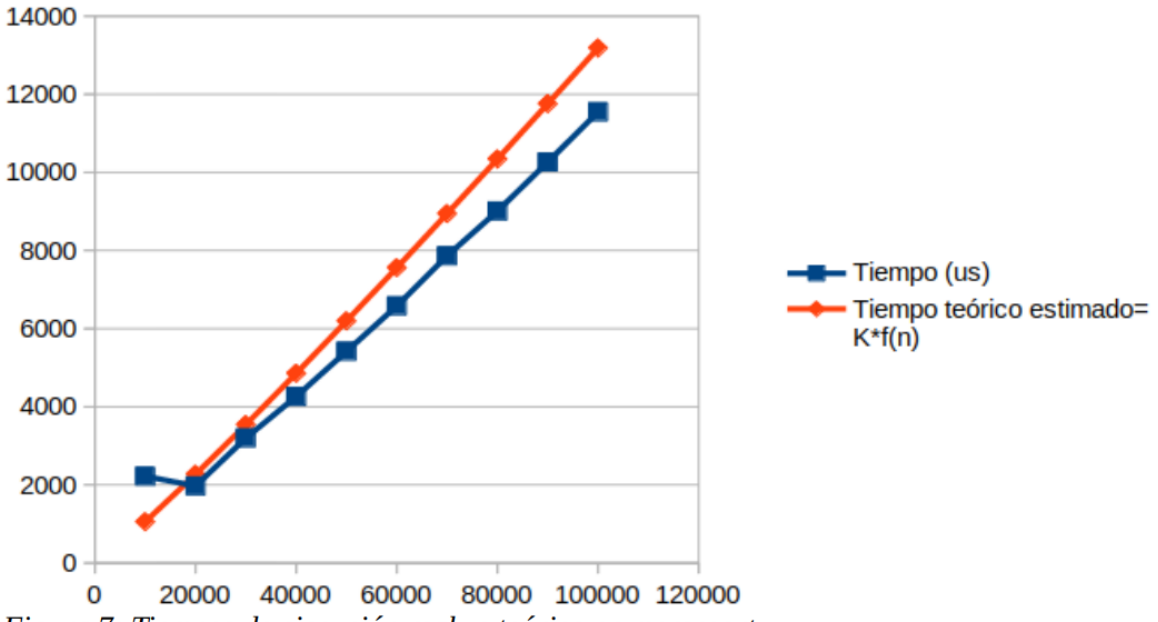
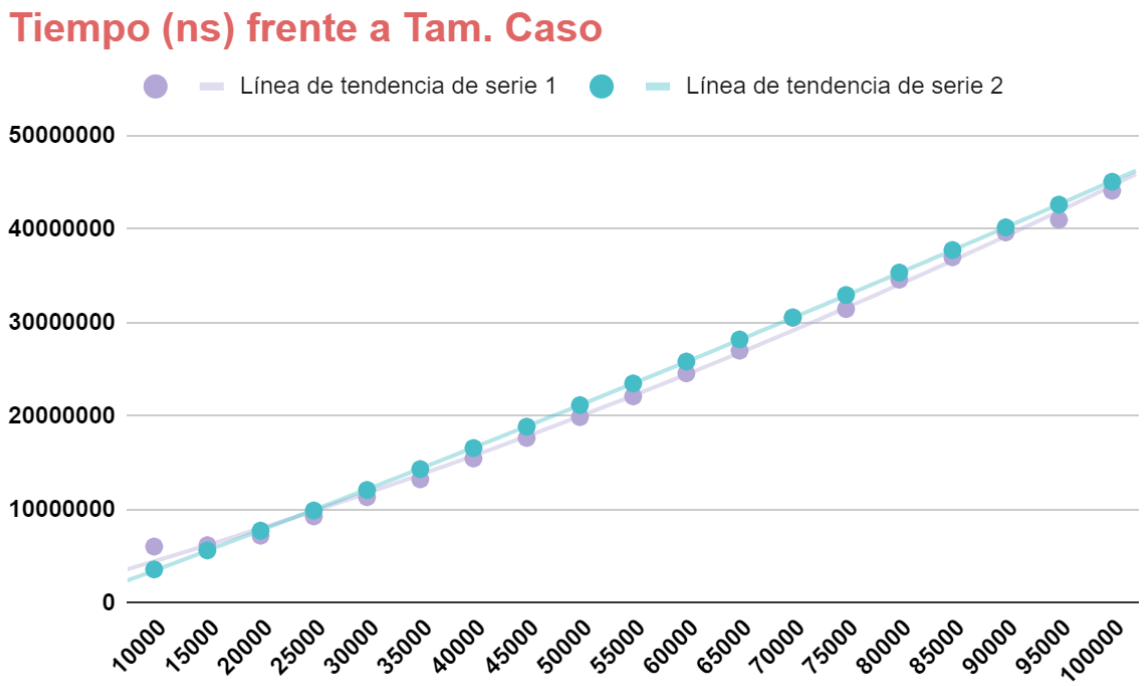


Figura 7: Tiempos de ejecución real vs teórico en mergesort

La eficiencia teórica del algoritmo de ordenación por **HeapSort** es de O(nlog(n)).



Según los datos **teóricos** de las eficiencias de los diferentes códigos, no cabe duda de que el algoritmo de ordenación por burbuja es el menos eficiente, dado que la complejidad de este es cuadrática frente a las logarítmicas de los otros dos ejemplos.

En el caso de la eficiencia **híbrida**, como se puede observar en las diferentes gráficas, los algoritmos MergeSort y HeapSort son gráficas prácticamente iguales, ya que, la eficiencia es la misma; por ello, son los más eficientes, frente al algoritmo burbuja.

CÓDIGO MAIN EFICIENCIA:

```
int main(int argc, char *argv[]) {

    string salida;
    int *v;
    int n, argumento;

    // Comprobar validez de la llamada
    if (argc <= 3) {
        cerr<<"\nError: El programa se debe ejecutar de la siguiente forma.\n\n";
        cerr<<argv[0]<<" NombreFicheroSalida Semilla tamCaso1 tamCaso2 ... tamCasoN\n\n";
        return 0;
    }

    // Obtener argumentos
    salida = argv[1];

    ofstream a(salida);
    if (!a.is_open()) {
        cerr << "Fail creating " << salida << " file" << endl;
    }

    vector<chrono::nanoseconds> times;
    vector<int> sizes;
    int k = 0;
        // Pasamos por cada tamaÒo de caso
    for (argumento= 3; argumento<argc; argumento++) {

        // Cogemos el tamanio del caso
        n= atoi(argv[argumento]);

        // Reservamos memoria para el vector
        v= new int[n];

        // Generamos vector aleatorio de prueba, con componentes entre 0 y n-1
        for (int i= 0; i<n; i++)
            v[i]= rand()%n;

        srand(time(NULL));
        int pos = rand()%n;

        // Chrono start, operation to measure, chrono stop
        auto start = chrono::high_resolution_clock::now();
        while (k <= 100) {
            MaximoMinimoDyV(v, 0, n);
            k++;
        }
        auto stop = chrono::high_resolution_clock::now();
        auto duration = chrono::duration_cast<chrono::nanoseconds>((stop - start)/k);
        // Hago la media de ejecutar 10 veces lo mismo para conseguir valores más estables.
        k = 0;

        // Info storing
        times.push_back(duration);
        sizes.push_back(n);
    }

    // Printing
    cout << "Sizes: ";
    for (int i = 0; i < sizes.size(); ++i){
        cout << sizes[i] << ",";
    }
    cout << endl;
    cout << "Times: ";
    for (int i = 0; i < times.size(); ++i){
        cout << times[i].count() << ",";
    }
    cout << endl;
    for (int i = 0; i < times.size(); ++i){
        a << sizes[i] << "," << times[i].count() << endl;
    }
    return 0;
}
```