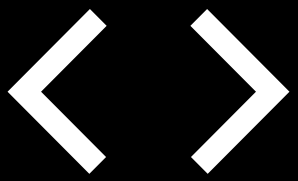


Algorítmica

Práctica 1

Cálculo de la eficiencia de algoritmos

Coman Venceslá, Alejandro
Fernández Aranda, José Antonio

Algoritmo MaximoMinimoDyV

```
75  pair<int, int> MaximoMinimoDyV(int *A, int Cini, int Cfin){
76      int mitad, max, min;
77      pair <int, int> pair1, pair2, result;
78      if (Cini < Cfin-1) {
79          mitad = (Cini+Cfin)/2;
80          pair1 = MaximoMinimoDyV(A, Cini, mitad);
81          pair2 = MaximoMinimoDyV(A, mitad+1, Cfin);
82          max = pair1.first > pair2.first ? pair1.first : pair2.first;
83          min = pair1.second > pair2.second ? pair1.second : pair2.second;
84      }  else if (Cini = Cfin) {
85          max = A[Cini];
86          min = A[Cini];
87      } else {
88          max = A[Cini] > A[Cfin] ? A[Cini] : A[Cfin];
89          min = A[Cini] < A[Cfin] ? A[Cini] : A[Cfin];
90      }
91      result.first = max;
92      result.second = min;
93      return result;
94  }
```

Estamos ante un algoritmo de búsqueda que devuelve un par con el elemento mínimo y máximo de un vector A, dadas dos posiciones inicial `Cini` y final `Cfin`, empleando la técnica de Divide y Vencerás, por la que va partiendo el tramo comprendido entre `Cini` y `Cfin` en dos mitades y así consecutivamente hasta llegar al caso base.

Las variable o variables de las cuales depende el tamaño del caso, serían `Cini y Cfin` ya que delimitan el intervalo del vector en el que se va a aplicar el algoritmo. También entra en juego posteriormente otra variable dependiente de las dos anteriores, que se trate del punto medio de dicho intervalo `mitad`, el cual actuará como punto divisorio de ambas mitades y se pasará como argumento en las subfunciones recursivas de la ejecución de la función principal para que éstas se ejecuten recursivamente sobre cada mitad y así sucesivamente.

Eficiencia Teórica

Se trata de un **algoritmo recursivo.** Por tanto denominaremos $\boldsymbol{T(n)}$ al **tiempo de ejecución** que tarda el algoritmo "`MaximoMinimoDyV`" en hallar un elemento máximo y otro mínimo en un vector para un tamaño igual a la diferencia entre los parámetros `Cini y Fini`. Dependiendo de los valores que tomen dichos parámetros, tendremos dos casos base y el caso general:

- **Primer caso base: (Cini = Cfin)** no entrará en el primer `if` de la línea 78 ya que no se cumplirá que `Cini < Cfini-1`. Saltará al segundo `if` de la línea 84 donde sí logrará entrar. Al estar el algoritmo buscando un máximo y un mínimo sobre la misma posición, es decir, actuando sobre un único elemento, el máximo y el mínimo resultarán ser ese mismo elemento. Al estar llevando a cabo dos asignaciones, $T(n)$ resultará ser 2, es decir, $O(1)$ o eficiencia constante.

- **Segundo caso base: (`Cini>Cfin+1`)** al comenzar con la ejecución del programa, nos saltamos las dos primeras condiciones ya que no se cumple, y entramos en else dónde se lleva a cabo las siguientes asignaciones, el máximo entre la posición `Cini` y `Fini`, análogamente para el caso del mínimo. Estas dos comprobaciones y asignaciones son $O(1)$.

- **Caso general: (`Cini< Cfin+1`),** en este caso:
  - Se calcula la parte central del vector en `mitad`. Esta operación es $O(1)$ porque todas las sentencias son operaciones simples.
  - Luego se hace una llamada recursiva en la línea 80 para resolver un problema desde `Cini` hasta `mitad`, por lo que el tamaño será de $n/2$. Si para n el tiempo de ejecución es $T(n)$, para $n/2$ será de $T(n/2)$.
  - Luego hace otra llamada recursiva en la línea 81 para resolver otro problema desde `mitad+1` hasta `Cfin`. Ocurre lo mismo de antes, al ser de tamaño $n/2$, el tiempo de ejecución será $T(n/2)$
  - Por último, en las dos últimas líneas del `if` (82 y 83) se realiza un par de asignaciones condicionadas que, al ser operaciones simples, serán $O(1)$.

Por tanto, podemos aproximar $T(n)$ en el caso general como $\mathbf{T(n) = 2T(n/2) + a}$. Hay que resolver la ecuación en recurrencias para obtener el orden de ejecución de `MaximoMinimoDyV`:

La ecuación no cumple con la forma requerida por la ecuación de característica, por lo que tendremos que hacer un cambio de variable de $\boldsymbol{n = 2^m}$:

Cambio de variable

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

Llevando la parte homogénea a la izquierda

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

Resolviendo primero la parte homogénea

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

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

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

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

Ahora deberemos resolver la parte no homogénea. Debemos conseguir un escalar b_1 y un polinomio $q_1(m)$ tal que:

$$\mathbf{a = b_1 q_1(m)}$$

Nos queda $b_1 = 1 y\ q_1(m) = a$ donde el grado del polinomio es $d_1 = 0$

El polinomio característico nos queda tal que $\mathbf{P(x) = P_H(x)(x-b_1)^{d1+1} = (x-2)(x-1)}$

Obtenemos dos raíces $r_1 = 1$ y $r_2 = 2$ ambas con multiplicidad $M_1 = M_2 = 1$

Aplicando la fórmula de la ecuación característica, el tiempo de ejecución será:

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

Deshaciendo el cambio de variable realizado anteriormente: ($n = 2^m$)

$$T(n) = c_1 n + c_2$$

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

Eficiencia Práctica

Para medir la eficiencia práctica de los algoritmos, tenemos que ejecutar el mismo algoritmo para diferentes tamaños de caso, y calcular su tiempo de ejecución. Para ello haremos uso de la biblioteca **chrono** incluida en el estándar **C++11**. No olvidar compilar cada programa con el compilador **g++** incluyendo este estándar (o superior) antes de compilar cada programa.

¿Cómo hemos procedido?

Entre la ejecución del algoritmo hemos tomado los instantes de tiempo, proporcionados por la función de la librería chrono, **now()**

```
t0= std::chrono::high_resolution_clock::now();
par = MaximoMinimoDyV(v, 0, n-1);
tf= std::chrono::high_resolution_clock::now();
```

Para el cálculo del tiempo, simplemente restaremos ambos tiempos para obtener la diferencia, que coincidirá con el tiempo de ejecución del algoritmo. Medimos la duración en microsegundos:

```
unsigned long tejecucion= std::chrono::duration_cast<std::chrono::microseconds>(tf - t0).count();
```

Se deberán medir **diferentes tiempos de ejecución**, (nosotros hemos tomado 10 en este caso) de modo que **se descartarán todos los tiempos de ejecución igual a 0** que obtengamos. Un tiempo de ejecución nunca es de 0 unidades; si esto ocurre, es debido a que la precisión del reloj no es suficiente como para poder medir el tiempo que tarda en ejecutarse un algoritmo. Los tamaños de caso y sus correspondientes tiempos de ejecución serán guardados a fichero para su posterior procesamiento.

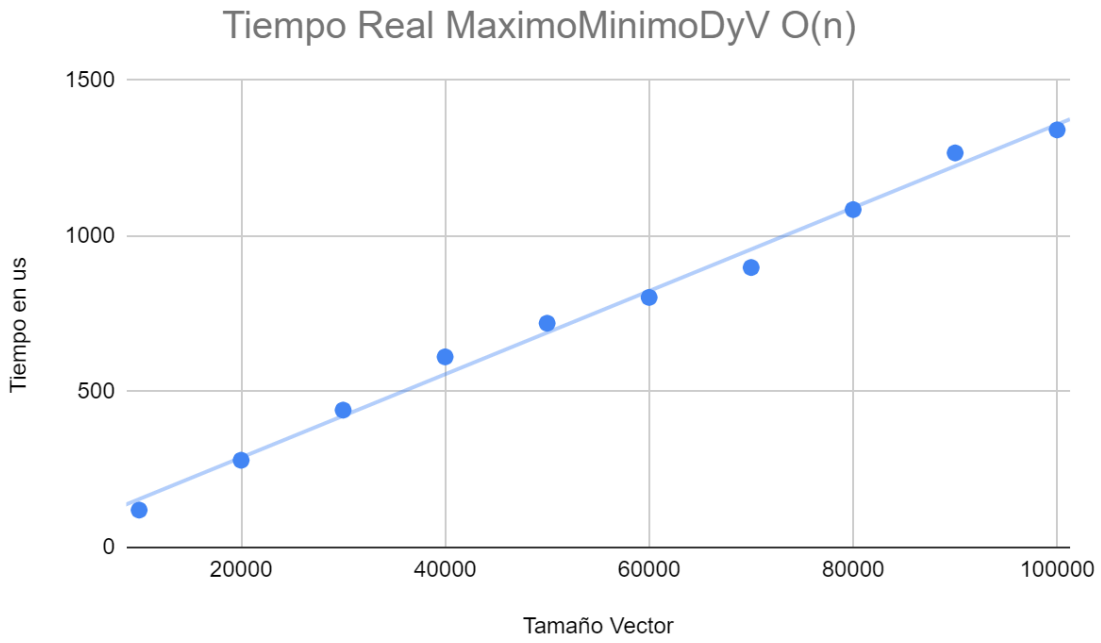
./MaximoMinimoDyV maxminDyV.txt 12345 10000 20000 30000 … 100000
(incrementos de 10^4)

Esta línea creará un fichero, `maxminDyV.txt`, con dos columnas de datos: en la primera encontraremos el tamaño de los casos de los vectores pasados como argumento. La segunda columna, muestra el tiempo de ejecución en este algoritmo en **µs**, para cada ejecución. Los valores

de los tiempos obtenidos dependen, además del tamaño de caso usado, del PC dónde se ejecuten y de su frecuencia de reloj.

De esto obtendremos la siguiente tabla de datos, y su correspondiente representación gráfica:

Tamaño del caso (n)	Tiempo de ejecución (en µs)
10000	120
20000	280
30000	441
40000	612
50000	720
60000	803
70000	899
80000	1085
90000	1267
100000	1341



Eficiencia Híbrida

En este apartado comprobaremos cómo la eficiencia práctica medida en el apartado anterior se corresponde con la teórica estimada en el primer apartado. Partiendo de la definición de orden de eficiencia, en la que se indica que, cuando el tamaño del caso es suficientemente grande (tiende a infinito), entonces el tiempo de ejecución del algoritmo $T(n)$ está acotado por la función del orden de eficiencia $O(f(n))$, de modo que $T(n) <= K \cdot f(n)$, con K una constante real positiva. A "K" se le denomina *constante oculta*.

Cálculo de la constante oculta

Aquí procederemos a calcular la constante oculta que se ha mencionado anteriormente, tal que:

$$T(n) <= K*f(n)$$

Despejando la constante nos queda que:

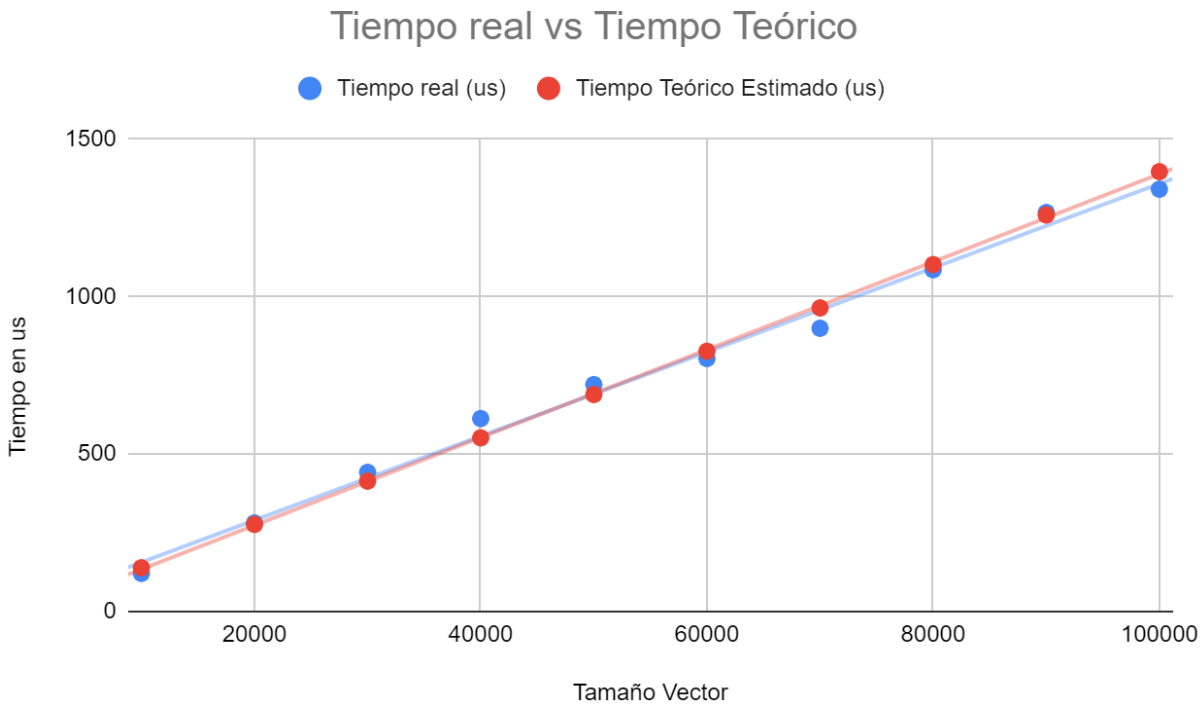
$$K = T(n)/f(n)$$

Este valor de K se calculará igual para cada una de las ejecuciones con diferentes tamaños del vector, no obstante obtendremos valores distintos, pero aproximadamente cercanos de K. Adicionalmente, realizaremos un promedio de todos los valores obtenidos para K para hallar la constante oculta. Nos queda tal que así:

Tamaño del caso (n)	Tiempo de ejecución (en µs)	K = Tiempo / F(n)	Tiempo teórico estimado = K * f(n)
10000	120	0,012	137,6764683
20000	280	0,014	275,3529365
30000	441	0,0147	413,0294048
40000	612	0,0153	550,705873
50000	720	0,0144	688,3823413
60000	803	0,01338333333	826,0588095
70000	899	0,01284285714	963,7352778
80000	1085	0,0135625	1101,411746
90000	1267	0,01407777778	1259,088214
100000	1341	0,01341	1396,764683
	K Promedio=	0,01376764683	

Comparación gráfica de órdenes de eficiencia

Sabiendo que el orden de eficiencia de `MaximoMinimoDyV` es $O(n)$ y que el valor de la constante oculta es K, deberemos comprobar de manera experimental que el orden $O(n)$ calculado con la constante oculta siempre será mayor que el tiempo de ejecución real del algoritmo. Para ello, en Google Sheets hemos representado en un gráfico ambas columnas para poder compararlas gráficamente, nos queda tal que de la siguiente manera:

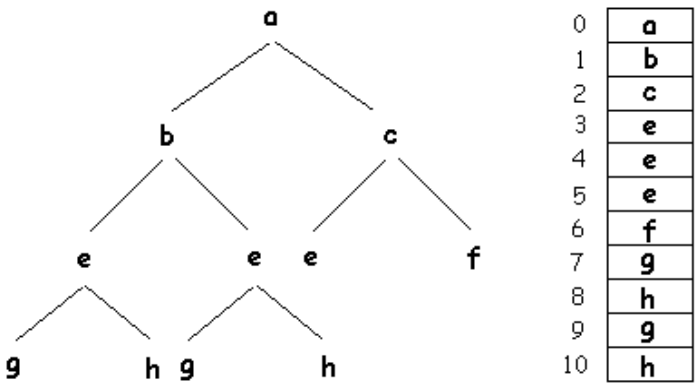


Tal y como podemos observar, la gráfica correspondiente al análisis teórico ($K*f(x)$) se ajusta bastante bien a la línea de tendencia generada sobre la dispersión de puntos correspondientes al tiempo real, por lo que podemos afirmar que tanto el análisis teórico como el cálculo de la constante oculta del algoritmo `MaximoMinimoDyV` se han realizado correctamente.

Algoritmo insertarEnPos

```
62  void insertarEnPos(double *apo, int pos){
63      int idx = pos-1;
64      int padre;
65      if (idx > 0) {
66          if (idx%2==0) {
67              padre = (idx-2)/2;
68          } else {
69              padre = (idx-1)/2;
70          }
71
72          if (apo[padre] > apo[idx]) {
73              double tmp = apo[idx];
74              apo[idx] = apo[padre];
75              apo[padre] = tmp;
76              insertarEnPos(apo, padre+1);
77          }
78      }
79  }
```

Estamos ante un algoritmo de inserción que trabaja con vector de double, el cual representa un árbol parcialmente ordenado (APO). Insertará en la posición dada por el argumento `pos` según la organización y condiciones de los APOs. La estructura de un APO es tal que:



Representación matricial de un APO.

Teniendo, en este caso, los nodos hijos valores mayores o igual que los de sus respectivos padres. Estando cada nodo en la posición `k` del vector, su hijo izquierdo estará en la posición `k+1` y su hijo derecho en la posición `k+2`. En el caso de que se inserte un elemento y tenga un valor menor o igual que su padre, se irán intercambiando padre e hijo hasta que el elemento insertado tenga un padre con valor menor o igual y unos hijos con valor mayor o igual que él mismo.

Las variable o variables de las cuales depende el tamaño del caso, viene dado por `pos`. Deberemos fijarnos en la profundidad en la que se halla en el árbol ya que, en el peor de los casos, en el que todos los elementos sean menores que el elemento en `pos`, se deberán realizar tantos intercambios como la profundidad mencionada anteriormente sea. Se da la casualidad de que la profundidad en la que se encuentra un elemento resulta ser log_2 de ese elemento. Por lo que el tamaño del caso será de log_2(`pos`).

Eficiencia Teórica

Se trata de un **algoritmo recursivo.** Por tanto denominaremos $\mathbf{T(n)}$ al **tiempo de ejecución** que tarda el algoritmo `insertarEnPos` en hallar una posición válida según las condiciones de nuestro APO y una vez comprobada dicha posición, realizamos el intercambio para que nuestro árbol siga ordenado hasta la posición que le corresponda. Dependiendo de los valores que tome `pos` y considerando que a la hora de insertar el elemento todos los valores ya existentes se encuentran ordenados según las condiciones del APO, tendremos:

- **Primer caso base:** en el caso de que la posición pasada como argumento, tenga un tamaño menor que 1, la primera condición no se cumple (`idx > 0`) línea 65, por lo que conlleva la no inserción del elemento en el APO y concluye. Según las reglas de análisis de eficiencia, en este caso caso base de la recurrencia el tiempo de ejecución es $O(1)$.

- **Segundo caso base:** en el que no existen elementos ascendentes de `pos` tal que son mayores o iguales que él. En otras palabras, su padre será menor o igual y no entrará en el `if` de la línea 72, por lo que no se realizará ninguna llamada recursiva a la función y, ya que el resto de instrucciones de la función son operaciones simples con tamaños constantes, el orden será $O(1)$.

- **Caso general:** que es aquel que nos interesa analizar, en el que `pos` es un entero positivo tal que existen elementos ascendentes de `pos` tal que son mayores o iguales que él. Para este caso se realizarán las siguientes instrucciones:
  - Entre las líneas 73 a 75, se intercambiarán los elementos del elemento en `pos` con el del elemento de su padre. Se harán uso de 3 asignaciones con un registro temporal, las cuales son $O(1)$, por lo que las 3 líneas en conjunto serán también $O(1)$.
  - En la línea 76, se realiza una llamada recursiva a la función pero esta vez con el padre de `pos`, que, hablando en términos vectoriales, se trata de la posición `pos/2`. Si *T(n)* equivale a `pos`, `pos/2` será igual a $T(n/2)$.

Por tanto, podemos aproximar $T(n)$ en el caso general como $\mathbf{T(n) = T(n/2) + a}$. Hay que resolver la ecuación en recurrencias para obtener el orden de ejecución de `insertarEnPos`:

La ecuación no cumple con la forma requerida por la ecuación de característica, por lo que tendremos que hacer un cambio de variable de $\mathbf{n = 2^k}$:

Primero realizamos el cambio de variable:

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

Luego, llevamos la parte homogénea a la izquierda de la ecuación:

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

Resolvemos la parte homogénea:

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

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

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

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

A continuación, deberemos resolver la parte no homogénea. Hay que conseguir un escalar b_1 y un polinomio $q_1(m)$ tal que:

$$a = b_1 q_1(m)$$

Nos queda $b_1 = 1$ y $q_1(m) = a$ donde el grado del polinomio es $d_1 = 0$

El polinomio característico nos queda tal que $P(x) = P_H(x)(x-b_1)^{d1+1} = (x-1)(x-1) = (x-1)^2$

Obtenemos una raíz $r = 1$ con multiplicidad $M_1 = 2$

Aplicando la fórmula de la ecuación característica, el tiempo de ejecución será:

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

Deshaciendo el cambio de variable realizado anteriormente: ($n = 2^m$)

$$T(n) = c_1 log_2(n) + c_2$$

Aplicando la regla del máximo a la ecuación anterior, obtenemos que el orden de eficiencia del algoritmo en el peor de los casos es $O(log_2(n))$.

Eficiencia Práctica

Para medir la eficiencia práctica de los algoritmos, tenemos que ejecutar el mismo algoritmo para diferentes tamaños de caso, y calcular su tiempo de ejecución. Para ello haremos uso de la biblioteca **chrono** incluida en el estándar **C++11.** No olvidar compilar cada programa con el compilador **g++** incluyendo este estándar (o superior) antes de compilar cada programa.

¿Cómo hemos procedido?

Entre la ejecución del algoritmo hemos tomado los instantes de tiempo, proporcionados por la función de la librería chrono, **now()**

```
t0= std::chrono::high_resolution_clock::now();
insertarEnPos(v, m);
tf= std::chrono::high_resolution_clock::now();
```

Para el cálculo del tiempo, simplemente restaremos ambos tiempos para obtener la diferencia, que coincidirá con el tiempo de ejecución del algoritmo. Medimos la duración en nanosegundos:

```
unsigned long tejecucion= std::chrono::duration_cast<std::chrono::nanoseconds>(tf - t0).count();
```

Se deberán medir **diferentes tiempos de ejecución**, (hemos decidido tomar 10 en este caso) de modo que **se descartarán todos los tiempos de ejecución igual a 0** que obtengamos. Un tiempo de ejecución nunca es de 0 unidades; si esto ocurre, es debido a que la precisión del reloj no es suficiente como para poder medir el tiempo que tarda en ejecutarse un algoritmo. Los tamaños de caso y sus correspondientes tiempos de ejecución serán guardados a fichero para su posterior procesamiento.

./insertarEnPos salidapos.txt 12345 100 200 300 … 5000 (incrementos de 10^2)

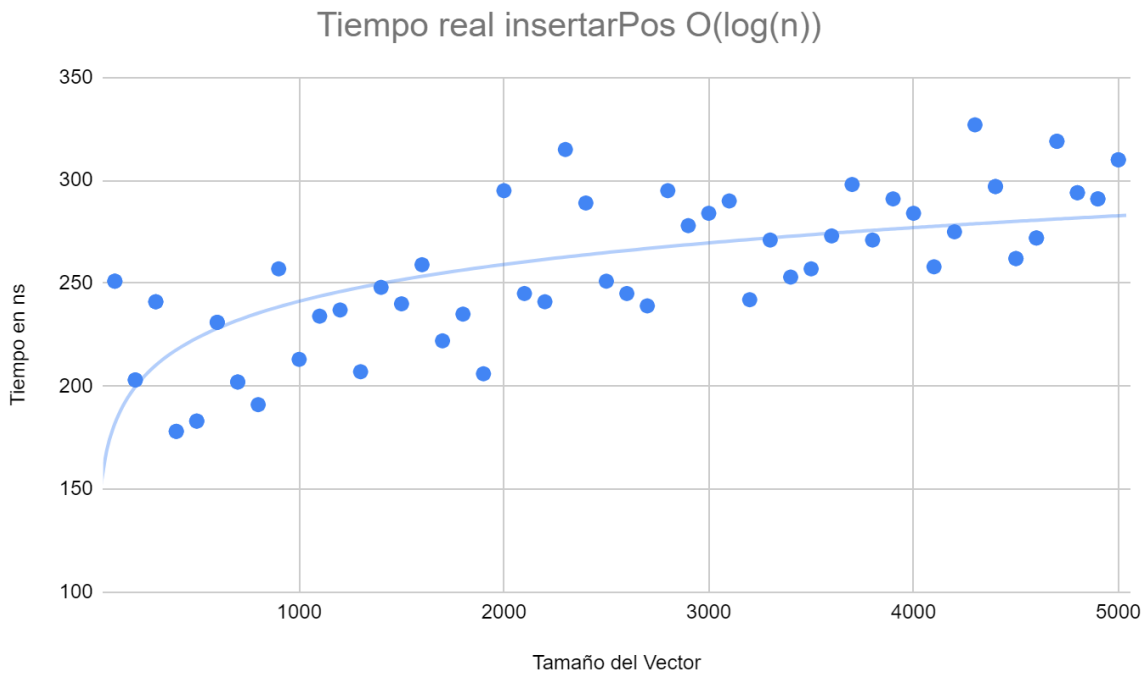
Esta línea creará un fichero, `insertar.txt`, con dos columnas de datos: en la primera encontraremos el tamaño de los casos de los vectores pasados como argumento. La segunda columna, muestra el tiempo de ejecución en este algoritmo en **ns**, para cada ejecución. Los valores de los tiempos obtenidos dependen, además del tamaño de caso usado, del PC dónde se ejecuten y de su frecuencia de reloj.

Como notas adicionales en lo que respecta a las ejecuciones del algoritmo, cabe remarcar un aspecto que hemos adaptado a este caso que se trata de cómo se organiza el vector. En el algoritmo anterior (`MaximoMinimoDyV`) los valores del vector eran completamente aleatorios, haciendo llamadas a la función `rand()`. Debido a las condiciones del árbol APO en el que los nodos hijo son mayores o iguales que los padres (es un APOmin), el vector de dicho árbol, contendrá valores ascendentes conforme se avanza en el vector.

Ya que queremos evaluar el algoritmo en el peor de los casos, hemos revertido el orden de los valores de los elementos del vector quedando éste ordenado de mayor a menor, para que siempre se pueda entrar en el `if` de la línea 72.

Resumiendo, hemos dejado de insertar números aleatorios en el vector y ahora tenemos uno ordenado de mayor a menor, que es el peor caso para este algoritmo de inserción, que esté completamente desordenado conforme a su forma de ordenación implementada.

De esto obtendremos la siguiente tabla de datos, y su correspondiente representación gráfica:



Tamaño del caso (n)	Tiempo de ejecución (en ns)
100	251
200	203
300	241
400	178
500	183
600	231
700	202
...	...
4400	297
4500	262
4600	272
4700	319
4800	294
4900	291
5000	310

(Al haber tomado 49 tamaños diferentes, hemos decidido incluir algunos de los primeros y algunos de los últimos datos por razones de espacio)

<>

Eficiencia Híbrida

En este apartado comprobaremos cómo la eficiencia práctica medida en el apartado anterior se corresponde con la teórica estimada en el primer apartado. Partiendo de la definición de orden de eficiencia, en la que se indica que, cuando el tamaño del caso es suficientemente grande (tiende a infinito), entonces el tiempo de ejecución del algoritmo $T(n)$ está acotado por la función del orden de eficiencia $O(f(n))$, de modo que $T(n) <= K \cdot f(n)$, con K una constante real positiva. A "K" se le denomina *constante oculta*.

Cálculo de la constante oculta

Aquí procederemos a calcular la constante oculta que se ha mencionado anteriormente, tal que:

$$T(n) <= K * f(n)$$

Despejando la constante nos queda que:

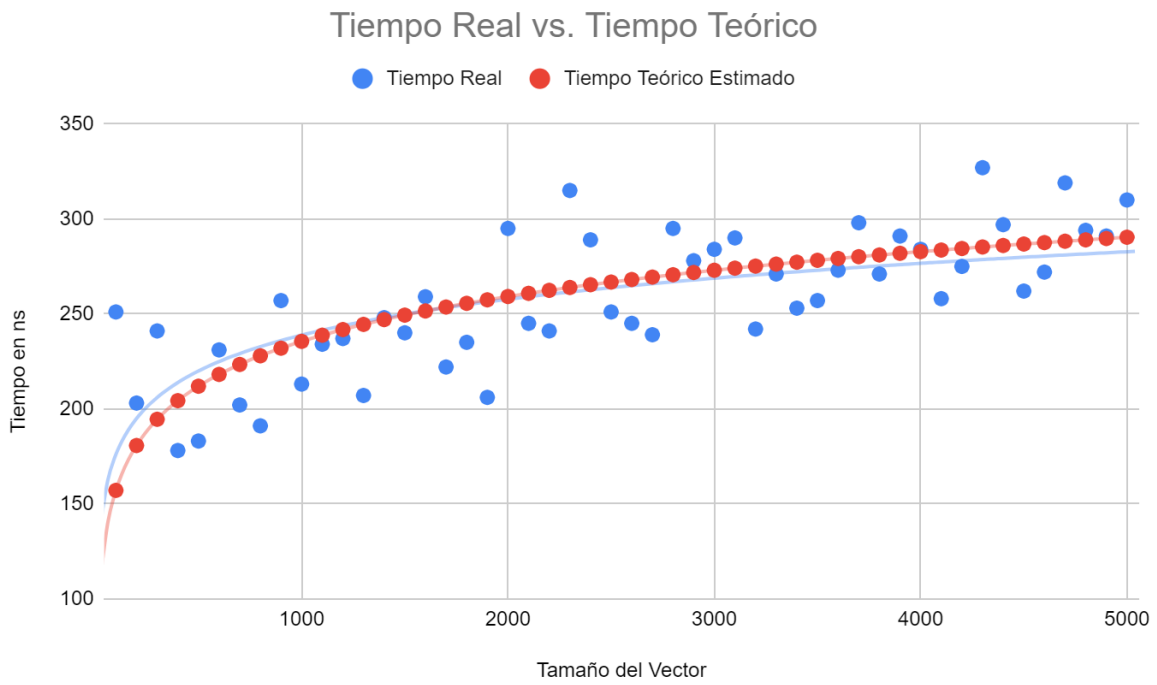
$$K = T(n) / f(n)$$

Este valor de K se calculará igual para cada una de las ejecuciones con diferentes tamaños del vector, no obstante obtendremos valores distintos, pero aproximadamente cercanos de K. Adicionalmente, realizaremos un promedio de todos los valores obtenidos para K para hallar la constante oculta. Nos queda tal que así:

Tamaño del caso (n)	Tiempo de ejecución (en ns)	K = Tiempo / F(n)	Tiempo teórico estimado = K * f(n)
100	251	125,5	157,0029591
200	203	88,2213619	180,6342592
300	241	97,29035248	194,4576836
400	178	68,40733903	204,2655592
500	183	67,8036435	211,8731386
600	231	83,14882063	218,0889836
700	202	70,9993108	223,3444056
...	...	...	...
4400	297	81,51608553	286,0164258
4500	262	71,71770025	286,7825875
4600	272	74,2609838	287,5319091
4700	319	86,87132324	288,265115
4800	294	79,86436667	288,9828838
4900	291	78,85759827	289,6858522
5000	310	83,80711377	290,3746182
	K=	79,73269879	

Comparación gráfica de órdenes de eficiencia

Sabiendo que el orden de eficiencia de insertarEnPos es $O(log_2(n))$ y que el valor de la constante oculta es K, deberemos comprobar de manera experimental que el orden $O(log_2(n))$ calculado con la constante oculta siempre será mayor que el tiempo de ejecución real del algoritmo. Para ello, en Google Sheets hemos representado en un gráfico ambas columnas para poder compararlas gráficamente, nos queda tal que de la siguiente manera:



Tal y como podemos observar, la gráfica correspondiente al análisis teórico $(K*f(x))$ se ajusta bastante bien a la línea de tendencia generada sobre la dispersión de puntos correspondientes al tiempo real, por lo que podemos afirmar que tanto el análisis teórico como el cálculo de la constante oculta del algoritmo `insertarEnPos` se han realizado correctamente.

Algoritmo reestructuraRaiz

```
63  void reestructurarRaiz(double *apo, int pos, int tamapo){
64      int minhijo;
65      if (2*pos+1 < tamapo) {
66          minhijo= 2*pos+1;
67          if ((minhijo+1 < tamapo) && (apo[minhijo]>apo[minhijo+1]))  minhijo++;
68          if (apo[pos] > apo[minhijo]) {
69              double tmp = apo[pos];
70              apo[pos] = apo[minhijo];
71              apo[minhijo]=tmp;
72              reestructurarRaiz(apo, minhijo, tamapo);
73          }
74      }
75  }
```

Estamos ante un algoritmo de ordenación que, dado un APO, reestructura su raíz, considerando ésta como el argumento `pos` pasado a la función. El método por el cual va reestructurando, trata de ir comparando los nodos con su hijo menor y, en el caso de que el hijo sea menor que el padre, ir intercambiando los nodos, así hasta tener la reestructuración completa.

Las variables que determinarán el tamaño del problema serán `pos` y `tamapo`. En este caso `pos` indicará el nodo del cual se partirá hacia abajo para ir comprobando y `tamapo` será el tamaño del árbol que actuará como tope para el algoritmo para que deje de comprobar hacia abajo. En el peor de los casos, siendo `pos` = 0 y `tamapo` = n, el algoritmo se ejecutará $log_2(n)$ veces, dándonos ya una pequeña pista de lo que podría ser la complejidad del algoritmo. Pero esto lo estudiaremos y comprobaremos en la eficiencia teórica.

Eficiencia Teórica

De nuevo se nos presenta un algoritmo recursivo. Por ende, llamemos $T(n)$ al tiempo de ejecución que tarda el algoritmo `reestructurarRaiz` en resolver un problema de tamaño $log_2(n)$ en el peor de los casos. Según el algoritmo, existen dos casos base y un caso general, es decir tres casos.

- **Primer caso base:** si queremos reestructurar un elemento que no tiene hijos, no es necesario equilibrar ningún elemento `if (2*pos+1<tamapo)` línea 65, por lo que esta sentencia sería $O(1)$ y con ella rompería el algoritmo.

- **Segundo caso base:** en el caso de que el nodo especificado por `pos` tenga uno o dos hijos (en el caso de que tenga dos se coge el más pequeño) y el padre sea el más pequeño de todos. En este caso el árbol que cuelga a partir de este nodo no precisa de reestructuración alguna. Como todas las operaciones previas a la comprobación son $O(1)$.

- **Caso general:** que es el que nos interesa estudiar. Se dará cuando el elemento en `pos` tenga uno o dos hijos y el menor sea mayor que el padre, es decir, el árbol necesite reestructuración. En este caso se llamará de nuevo a la función pero con dos veces el valor de `pos`. Al realizar el mismo recorrido que el algoritmo anterior pero esta vez de manera

inversa (de niveles menos profundos a los más profundos). Tendrá un tiempo de ejecución igual por lo que será $T(n/2)$

Por tanto, podemos aproximar $T(n)$ en el caso general como $\mathbf{T(n) = T(n/2) + a}$

Hay que resolver la ecuación en recurrencias para obtener el orden de ejecución de `reestructurarRaiz`:

La ecuación no cumple con la forma requerida por la ecuación de característica, por lo que tendremos que hacer un cambio de variable de $\boldsymbol{n = 2^m}$:

Primero realizamos el cambio de variable:

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

Luego, llevamos la parte homogénea a la izquierda de la ecuación:

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

Resolvemos la parte homogénea:

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

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

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

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

A continuación, deberemos resolver la parte no homogénea. Hay que conseguir un escalar b_1 y un polinomio $q_1(m)$ tal que:

$$\mathbf{a = b_1 q_1(m)}$$

Nos queda $b_1 = 1$ y $q_1(m) = a$ donde el grado del polinomio es $d_1 = 0$

El polinomio característico nos queda tal que $\mathbf{P(x) = P_H(x)(x-b_1)^{d1+1} = (x-1)(x-1) = (x-1)^2}$

Obtenemos una raíz $r = 1$ con multiplicidad $M_1 = 2$

Aplicando la fórmula de la ecuación característica, el tiempo de ejecución será:

$$T(2^m) = \sum_{i=1}^{r} \sum_{j=0}^{M_i - 1} c_{ij} R_i^m m^j = \mathbf{c_{10} 1^m m + c_{20} 1^m}$$

Deshaciendo el cambio de variable realizado anteriormente: ($n = 2^m$)

$$\mathbf{T(n) = c_1 log_2(n) + c_2}$$

Aplicando la regla del máximo a la ecuación anterior, obtenemos que el orden de eficiencia del algoritmo en el peor de los casos es $\mathbf{O(log_2(n))}$.

Eficiencia Práctica

Para medir la eficiencia práctica de los algoritmos, tenemos que ejecutar el mismo algoritmo para diferentes tamaños de caso, y calcular su tiempo de ejecución. Para ello haremos uso de la biblioteca **chrono** incluida en el estándar **C++11**. No olvidar compilar cada programa con el compilador **g++** incluyendo este estándar (o superior) antes de compilar cada programa.

¿Cómo hemos procedido?

Entre la ejecución del algoritmo hemos tomado los instantes de tiempo, proporcionados por la función de la librería chrono, **now( )**

```
t0= std::chrono::high_resolution_clock::now();
reestructurarRaiz(v, 0, m);
tf= std::chrono::high_resolution_clock::now();
```

Para el cálculo del tiempo, simplemente restaremos ambos tiempos para obtener la diferencia, que coincidirá con el tiempo de ejecución del algoritmo. Medimos la duración en **nanosegundos** (hacerlo en microsegundos no se consigue la suficiente precisión para poder realizar el análisis correctamente)

```
unsigned long tejecucion= std::chrono::duration_cast<std::chrono::nanoseconds>(tf - t0).count();
```

Se deberán medir **diferentes tiempos de ejecución**, (nosotros hemos tomado 10 en este caso) de modo que **se descartarán todos los tiempos de ejecución igual a 0** que obtengamos. Un tiempo de ejecución nunca es de 0 unidades; si esto ocurre, es debido a que la precisión del reloj no es suficiente como para poder medir el tiempo que tarda en ejecutarse un algoritmo. Los tamaños de caso y sus correspondientes tiempos de ejecución serán guardados a fichero para su posterior procesamiento.

./insertarEnPos salidapos.txt 12345 100 200 300 … 5000 (incrementos de 10^2)

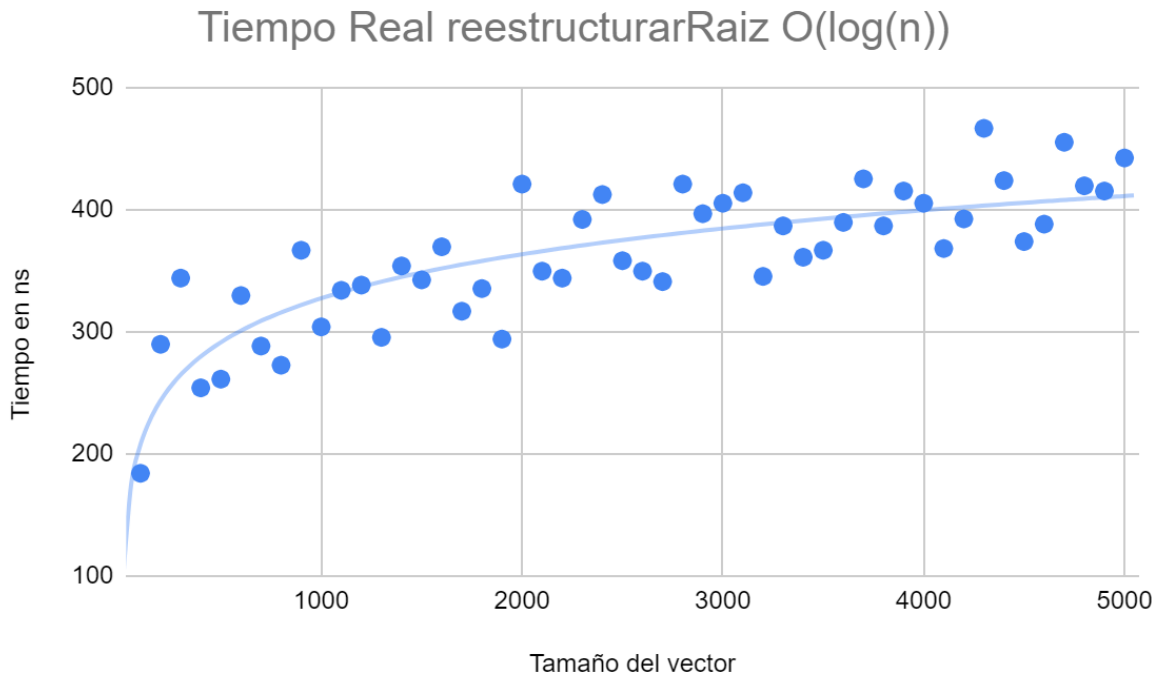
Esta línea creará un fichero, `reestructurar.txt`, con dos columnas de datos: en la primera encontraremos el tamaño de los casos de los vectores pasados como argumento. La segunda columna, muestra el tiempo de ejecución en este algoritmo en **ns**, para cada ejecución. Los valores de los tiempos obtenidos dependen, además del tamaño de caso usado, del PC dónde se ejecuten y de su frecuencia de reloj.

Para la ejecución de este algoritmo también hemos elegido un vector con valores decrecientes para poder asegurar que se da el peor caso posible en el que para todos los nodos, sus hijos serán menores que sus padres y, por lo tanto será necesario reestructurar.

Obtenemos para los diferentes tamaños, la tabla de valores y su respectiva representación gráfica para estudiar la evolución de los tiempos de ejecución al aumentar los tamaños de los vectores:

Tamaño del caso (n)	Tiempo de ejecución (en ns)
100	184
200	290
300	344
400	254
500	261
600	330
700	288
...	...
4400	424
4500	374
4600	388
4700	455
4800	420
4900	415
5000	442

(Al haber tomado 49 tamaños diferentes, hemos decidido incluir algunos de los primeros y algunos de los últimos datos por razones de espacio)



Eficiencia Híbrida

En este apartado comprobaremos cómo la eficiencia práctica medida en el apartado anterior se corresponde con la teórica estimada en el primer apartado. Partiendo de la definición de orden de eficiencia, en la que se indica que, cuando el tamaño del caso es suficientemente grande (tiende a infinito), entonces el tiempo de ejecución del algoritmo $T(n)$ está acotado por la función del orden de eficiencia $O(f(n))$, de modo que $T(n) <= K \cdot f(n)$, con K una constante real positiva. A K se le denomina *constante oculta*.

Cálculo de la constante oculta

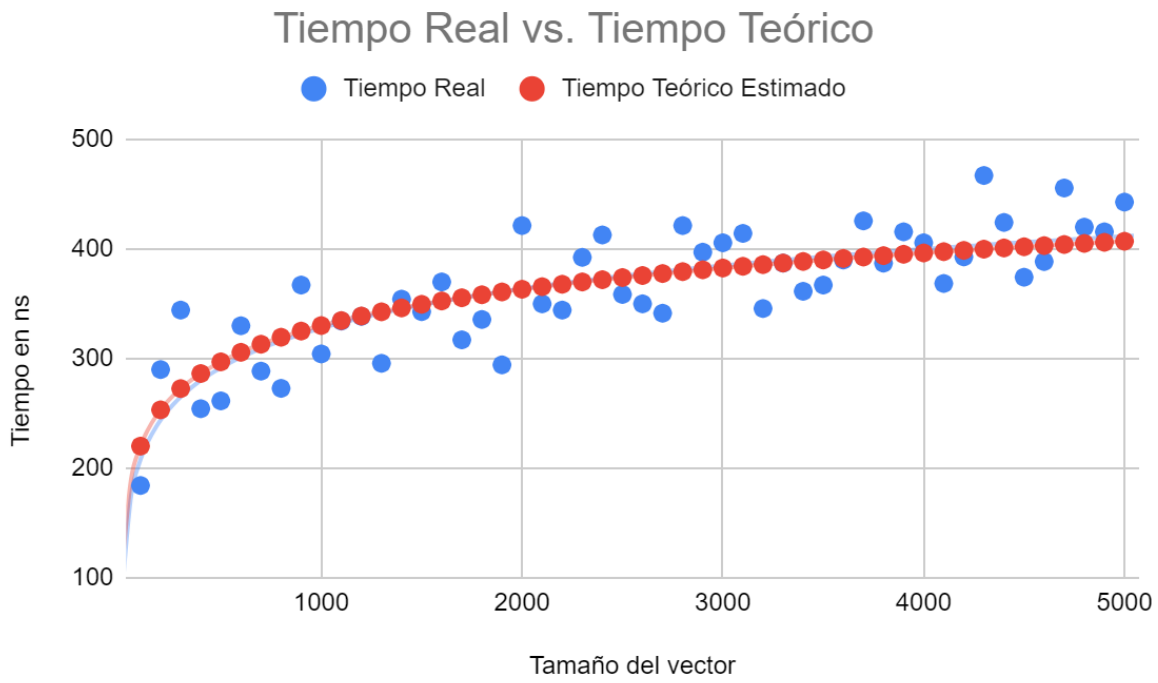
Aquí procederemos a calcular la constante oculta que se ha mencionado anteriormente, tal que: $T(n) <= K * f(n)$. Despejando la constante nos queda que: $K = T(n) / f(n)$

Este valor de K se calculará igual para cada una de las ejecuciones con diferentes tamaños del vector, no obstante obtendremos valores distintos, pero aproximadamente cercanos de K. Adicionalmente, realizaremos un promedio de todos los valores obtenidos para K para hallar la constante oculta. Nos queda tal que así:

Tamaño del caso (n)	Tiempo de ejecución (en ns)	K = Tiempo / F(n)	Tiempo teórico estimado = K * f(n)
100	184	92	219,8791445
200	290	125,8940815	252,9742534
300	344	138,835757	272,3336511
400	254	97,61897717	286,0693624
500	261	96,75748861	296,7236078
600	330	118,6554387	305,4287601
700	288	101,3177855	312,7888615
...	...	...	...
4400	424	116,325485	400,5596287
4500	374	102,3429451	401,6326211
4600	388	105,9722741	402,6820292
4700	455	123,9675427	403,7088677
4800	420	113,9684411	404,714087
4900	415	112,5317575	405,6985785
5000	442	119,5948394	406,66318
	K =	109,9395722	

Comparación gráfica de órdenes de eficiencia

Sabiendo que el orden de eficiencia de `reestructurarRaiz` es $O(log_2(n))$ y que el valor de la constante oculta es K, deberemos comprobar de manera experimental que el orden $O(n)$ calculado con la constante oculta siempre será mayor que el tiempo de ejecución real del algoritmo. Para ello, en Google Sheets hemos representado en un gráfico ambas columnas para poder compararlas gráficamente, nos queda tal que de la siguiente manera:



Tal y como podemos observar, la gráfica correspondiente al análisis teórico $(K*f(x))$ se ajusta bastante bien a la línea de tendencia generada sobre la dispersión de puntos correspondientes al tiempo real, por lo que podemos afirmar que tanto el análisis teórico como el cálculo de la constante oculta del algoritmo `reestructurarRaiz` se han realizado correctamente.

Algoritmo HeapSort

```
101   void HeapSort (int *v, int n) {
102       double *apo= new double [n];
103       int tamapo=0;
104       for(int i =0; i < n; i++){
105           apo[tamapo] = v[i];
106           tamapo++;
107           insertarEnPos(apo,tamapo);
108       }
109       for (int i = 0; i < n; i++){
110           v[i]=apo[0];
111           tamapo--;
112           apo[0]=apo[tamapo];
113           reestructurarRaiz(apo, 0, tamapo);
114       }
115       delete [] apo;
116   }
```

Esta vez no se nos presenta un algoritmo recursivo, sino iterativo (llamará a funciones recursivas pero no se llama a sí mismo). Se trata de un algoritmo de ordenación bastante conocido denominado `HeapSort`. Lo que hace este algoritmo de ordenación es ir almacenando todos los elementos del vector a ordenar `v` en un montículo o heap `apo`, el cual se comporta como un APOmin, para luego ir extrayendo el nodo que queda como nodo raíz del heap en sucesivas iteraciones para así obtener el montículo ordenado.

Para el peor de los casos, el tamaño de este problema **depende** de la variable **n**, la cual indicará el tamaño del vector a ordenar con el que trabajaremos. No obstante, será la variable **`tamapo`** la que se le pasará como argumento a las funciones recursivas estudiadas anteriormente. Esta variable se comportará de la siguiente manera:

- En el primer bucle se incrementará en una unidad tantas veces como iteraciones haya (hay `n` iteraciones). Por lo que al final del bucle tendremos que será justo igual a `n`.
- En el segundo bucle ocurre justo lo contrario, es decir, se va decrementando `tamapo` en una unidad por cada iteración. Como hay `n` iteraciones pues al final del bucle `tamapo` volverá a valer 0.

Eficiencia Teórica

Se nos presenta un algoritmo iterativo, es decir no es recursivo. Llamemos $T(n)$ al tiempo de ejecución que tarda el algoritmo `HeapSort` en resolver un problema de tamaño $n*log_2(n)$ en el peor de los casos. Podemos apreciar que el tamaño del caso Según el algoritmo, existen dos casos base y un caso general, es decir tres casos.

- **Caso base:** El tamaño del vector a ordenar es 1. En este caso el vector ya estará ordenado porque sólo tendrá un único elemento. Al entrar en los bucles, sólo se llevará a cabo una iteración en cada uno. En cuanto a las funciones estudiadas anteriormente `insertarEnPos` y `reestructurarRaiz`, el tamaño vendrá dado como la variable `tamapo` la cual se incrementa en función del número de iteraciones en el primer bucle y se decrementará según el número de iteraciones en el segundo bucle. Ya que n es 1. Cada bucle llegará a ejecutarse con 1 iteración por lo que el algoritmo será del orden $O(1)$.

- **Caso general:** que es el que nos interesa estudiar. Se dará cuando el tamaño n sea mayor que 1. En este caso tenemos de primeras que cada bucle se ejecutará n veces por lo que podemos afirmar que, como mínimo, tendríamos una complejidad lineal. Ahora deberemos ver qué es lo que ocurre dentro de los bucles. Ya que ambos algoritmos recursivos que se llaman en cada iteración presentan una complejidad logarítmica, finalmente tendremos que el algoritmo `HeapSort` será del orden $\boldsymbol{O(n*log_2(n))}$

Para este caso no realizaremos ningún análisis en cuanto al establecimiento de la ecuación característica ni desarrollo de la misma ya que no se trata de un algoritmo recursivo. Ya anteriormente en los casos base y general concluimos que el algoritmo es del orden $O(n*log_2(n))$.

Eficiencia Práctica

Para medir la eficiencia práctica de los algoritmos, tenemos que ejecutar el mismo algoritmo para diferentes tamaños de caso, y calcular su tiempo de ejecución. Para ello haremos uso de la biblioteca **`chrono`** incluida en el estándar **`C++11`**. No olvidar compilar cada programa con el compilador **`g++`** incluyendo este estándar (o superior) antes de compilar cada programa.

¿Cómo hemos procedido?

Entre la ejecución del algoritmo hemos tomado los instantes de tiempo, proporcionados por la función de la librería chrono, **`now()`**

```
t0= std::chrono::high_resolution_clock::now();
HeapSort(v, m);
tf= std::chrono::high_resolution_clock::now();
```

Para el cálculo del tiempo, simplemente restaremos ambos tiempos para obtener la diferencia, que coincidirá con el tiempo de ejecución del algoritmo. Medimos la duración en microsegundos, ya que en nanosegundos aporta valores muy grandes.

```
unsigned long tejecucion= std::chrono::duration_cast<std::chrono::microseconds>(tf - t0).count();
```

Se deberán medir **diferentes tiempos de ejecución**, (hemos tomado 50 en este caso) de modo que **se descartarán todos los tiempos de ejecución igual a 0** que obtengamos, que al cambiarlos a nanosegundos, han desaparecido. Un tiempo de ejecución nunca es de 0 unidades; si esto ocurre, es debido a que la precisión del reloj no es suficiente como para poder medir el tiempo que tarda en ejecutarse un algoritmo. Los tamaños de caso y sus correspondientes tiempos de ejecución serán guardados a fichero para su posterior procesamiento.

./heapsort salidaheap.txt 12345 100 200 300 … 4900 (incrementos de 10^2)

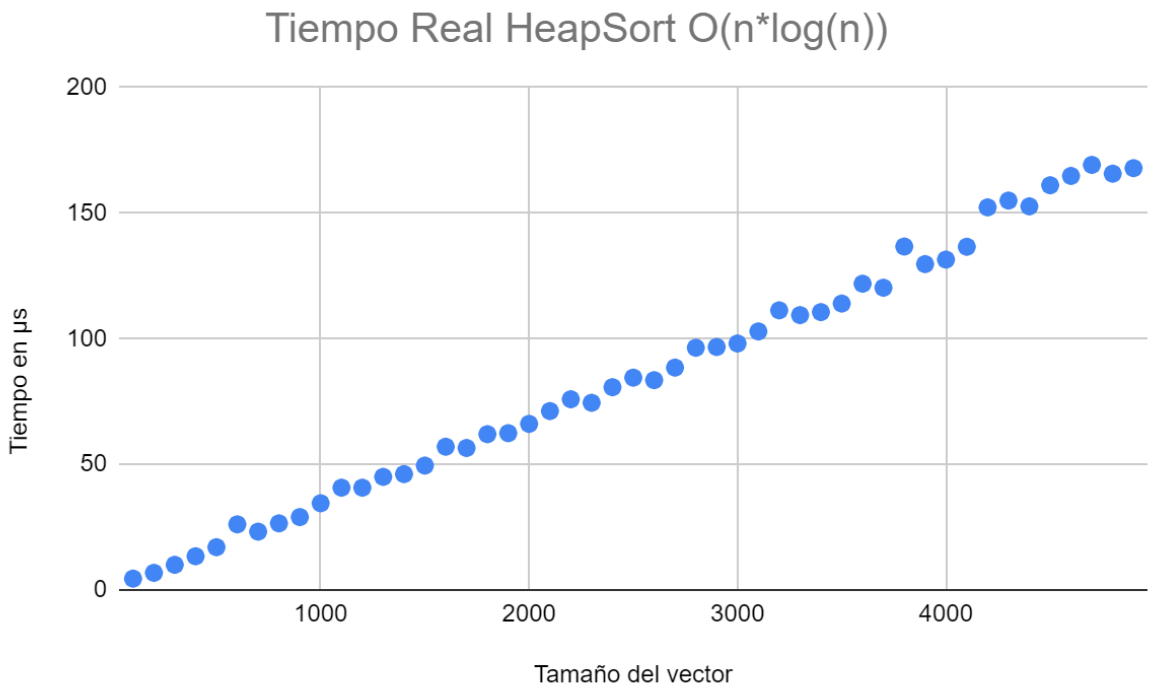
Esta línea creará un fichero, `salidaheap.txt`, con dos columnas de datos: en la primera encontraremos el tamaño de los casos de los vectores pasados como argumento. La segunda columna, muestra el tiempo de ejecución en este algoritmo en **ns**, para cada ejecución. Los valores de los tiempos obtenidos dependen, además del tamaño de caso usado, del PC dónde se ejecuten y de su frecuencia de reloj, dónde se desprende el valor de K).

Como notas adicionales en lo que respecta a las ejecuciones del algoritmo, cabe remarcar un aspecto que hemos adaptado a este caso que se trata de cómo se organiza el vector. En el algoritmo anterior (`HeapSort`) los valores del vector son completamente desordenados, en otras palabras de mayor a menor.

Obtenemos para los diferentes tamaños, la tabla de valores y su respectiva representación gráfica para estudiar la evolución de los tiempos de ejecución al aumentar los tamaños de los vectores:

Tamaño vector	Tiempo µs	Tamaño vector	Tiempo µs
100	5	…4000	132
200	7	4100	137
300	10	4200	152
400	13	4300	155
500	17	4400	153
600	26	4500	161
700	23	4600	165
800	27	4700	169
900	29	4800	166
1000 …	35	4900	168

Debido a la gran cantidad de datos que hemos usado en este algoritmo, hemos simplificado la tabla para colocar los valores más significativos.



Debido a que la función que representa la eficiencia de este algoritmo no se encuentra disponible para representar como línea de tendencia en el programa usado (Google Sheets) hemos decidido sustituirla por una cantidad mayor de valores cuya representación en conjunto se aproxima más a lo que llegaría a ser una curva $x*log(x)$.

Eficiencia Híbrida

En este apartado comprobaremos cómo la eficiencia práctica medida en el apartado anterior se corresponde con la teórica estimada en el primer apartado. Partiendo de la definición de orden de eficiencia, en la que se indica que, cuando el tamaño del caso es suficientemente grande (tiende a infinito), entonces el tiempo de ejecución del algoritmo $\mathbf{T(n)}$ está acotado por la función del orden de eficiencia $\mathbf{O(f(n))}$, de modo que $\mathbf{T(n) <= K \cdot f(n)}$, con $\mathbf{K}$ una constante real positiva. A "K" se le denomina *constante oculta*.

Cálculo de la constante oculta

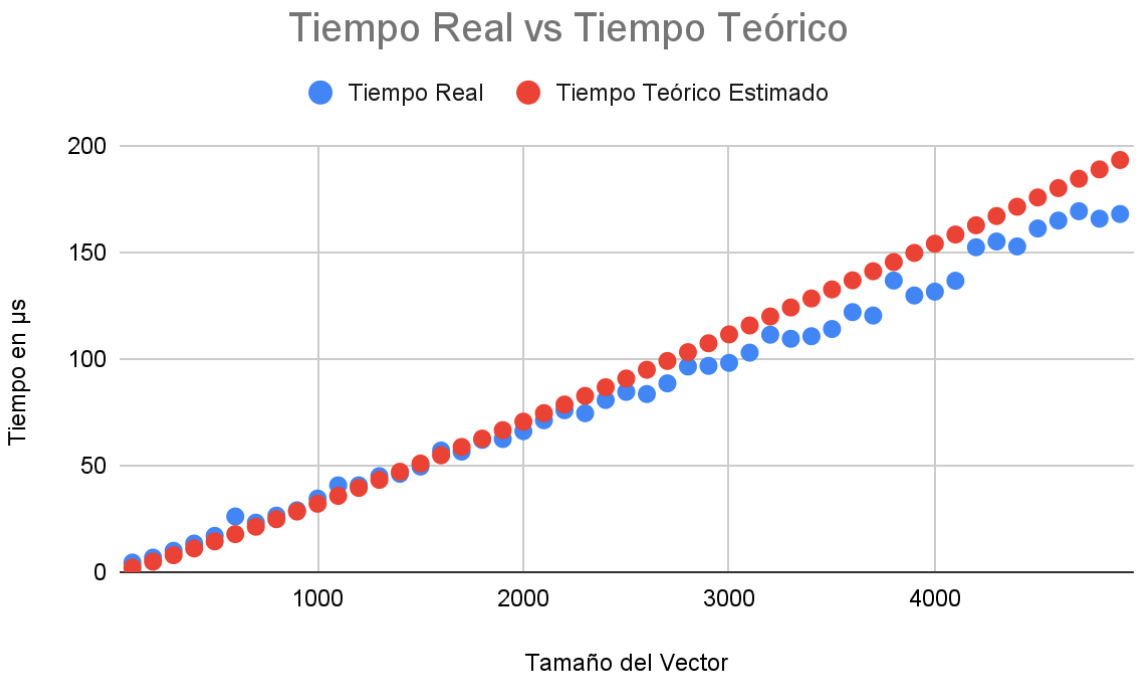
Aquí procederemos a calcular la constante oculta que se ha mencionado anteriormente, tal que: $\mathbf{T(n) <= K*f(n).}$ Despejando la constante nos queda que: $\mathbf{K = T(n)/f(n)}$

Este valor de K se calculará igual para cada una de las ejecuciones con diferentes tamaños del vector, no obstante obtendremos valores distintos, pero aproximadamente cercanos de K. Adicionalmente, realizaremos un promedio de todos los valores obtenidos para K para hallar la constante oculta. Nos queda tal que así:

Tamaño vector	Tiempo en µs	K = Tiempo/F(n)	Tiempo Teórico estimado
100	5	0,0225	2,137070231
200	7	0,01477599165	4,917462704
300	10	0,01345648029	7,940673137
400	13	0,01287441493	11,12156989
500	17	0,01259739825	14,41972112
600	26	0,01565789479	17,811313
...	...	...	...
4400	153	0,009525180793	171,2989135
4500	161	0,009799594156	175,6613635
4600	165	0,009781178173	180,0341267
4700	169	0,009803660303	184,4169789
4800	166	0,00937749827	188,8097055
4900	168	0,009285497404	193,2121006
	K =	0,01068535115	

Comparación gráfica de órdenes de eficiencia

Sabiendo que el orden de eficiencia de `HeapSort` es $O(n*log(n))$ y que el valor de la constante oculta es K, deberemos comprobar de manera experimental que el orden $O(n*log(n))$ calculado con la constante oculta siempre será mayor que el tiempo de ejecución real del algoritmo. Para ello, en Google Sheets hemos representado en un gráfico ambas columnas para poder compararlas gráficamente, nos queda tal que de la siguiente manera:



Tal y como podemos observar, la gráfica correspondiente al análisis teórico $(K*f(x))$ se ajusta bastante bien a la línea de tendencia generada sobre la dispersión de puntos correspondientes al tiempo real, por lo que podemos afirmar que tanto el análisis teórico como el cálculo de la constante oculta del algoritmo HeapSort se han realizado correctamente.

Conclusiones

Tras realizar los análisis teóricos de los diversos algoritmos, hemos podido comprobar que los datos obtenidos empíricamente y su curva de regresión se han podido ajustar adecuadamente a la función que deberíamos obtener. Con todo este análisis, deducimos que nuestros análisis teóricos han sido correctos.

También hemos visto que los datos obtenidos en la práctica depende de la arquitectura del ordenador dónde se ejecute el algoritmo, aún así las funciones son las mismas, siendo esto de mayor relevancia en el análisis de algoritmos.

Comparación de los algoritmos de ordenación

Comparación Teórica

Burbuja

Basándonos en la información proporcionada por el documento de prácticas tal y como se ha demostrado el algoritmo `Burbuja` tiene una complejidad del orden $\boldsymbol{O(n^2)}$.

Mergesort

Basándonos en la información provista en la guía de esta práctica se ha demostrado que el algoritmo `Mergesort` tiene una complejidad del orden $\boldsymbol{O(n*log(n))}$.

Heapsort

Tomaremos como referencia, la eficiencia calculada en el apartado anterior, por lo que el orden del algoritmo `Heapsort` es $\boldsymbol{O(n*log(n))}$.

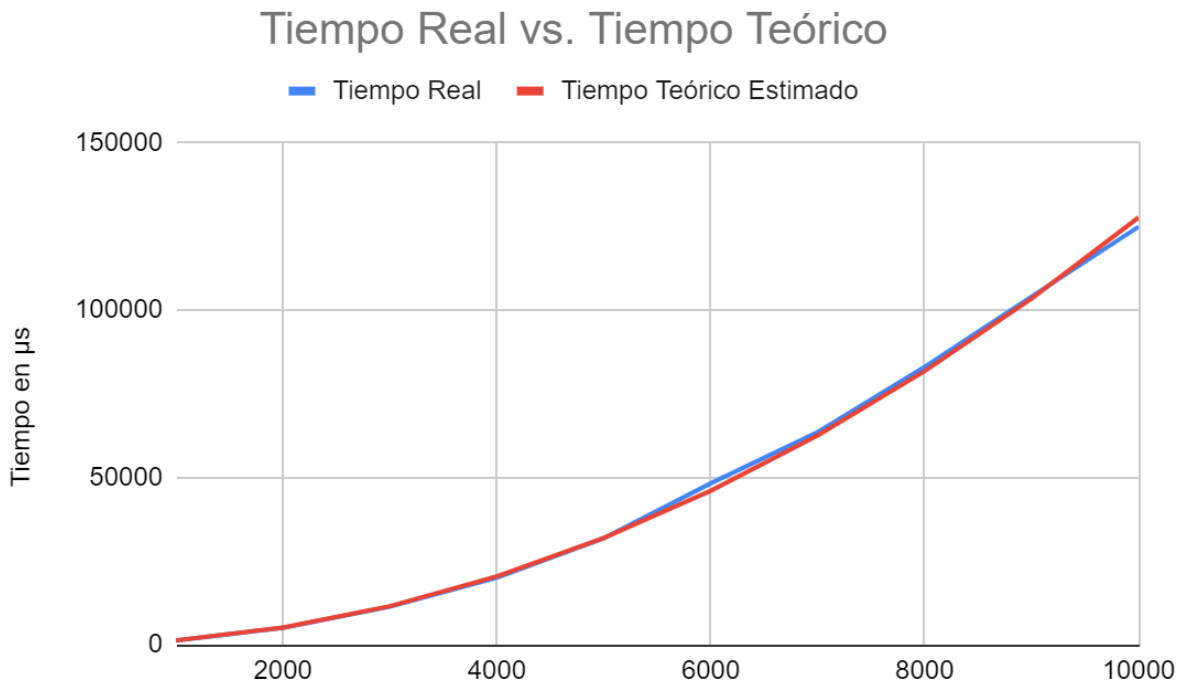
Pasando a la comparación desde un punto de vista teórico entre los tres algoritmos podemos apreciar como los algoritmos **`Mergesort` y `Heapsort` son del mismo orden** $O(n*log(n))$. No obstante, deberemos hallar posteriormente en el análisis híbrido la constante oculta de cada uno de ellos para poder comparar cual es asintóticamente más eficiente. Por otra parte, en cuanto al algoritmo de la **`burbuja`**, al ser $O(n^2)$, que es mayor que la complejidad de los otros dos anteriores, será obviamente **más ineficiente.**

Comparación Híbrida

Para poder comparar, desde el análisis híbrido, estos tres algoritmos hemos tomado los mismos tamaños de vector para los tres casos y los hemos ejecutado en la misma máquina para así obtener sus respectivos tiempos de ejecución real y poder calcular la constante oculta y el tiempo teórico estimado. Finalmente, hemos representado y comparado las gráficas obtenidas de los tres algoritmos:

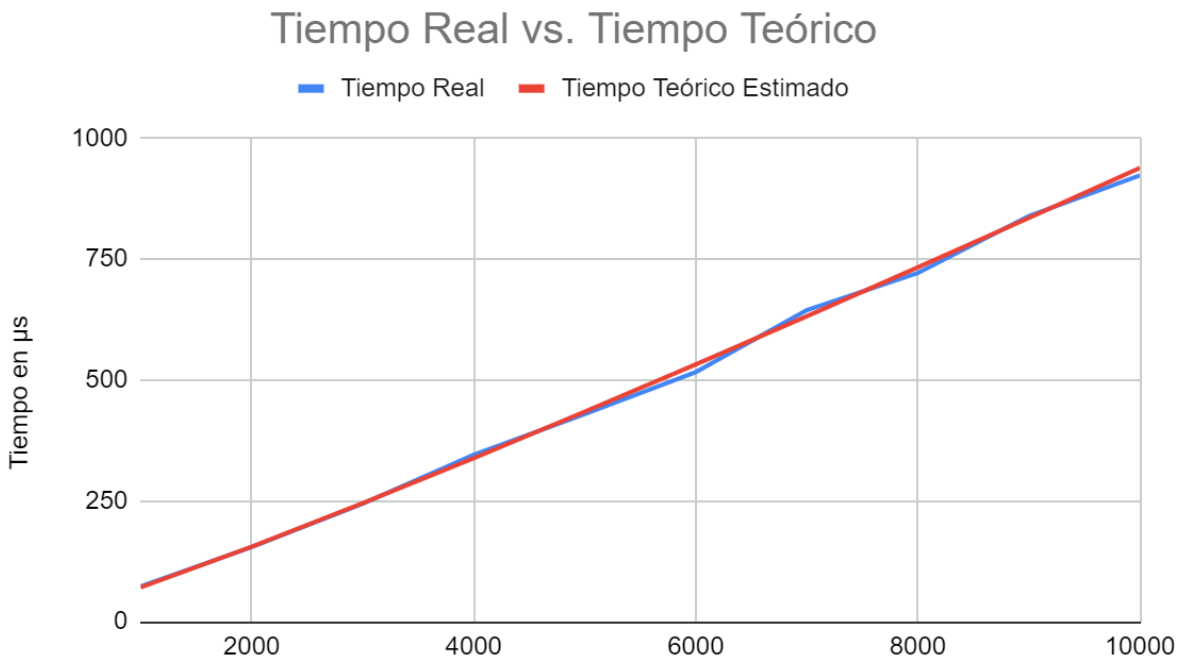
Burbuja

Tamaño Vector	Tiempo en µs.	K = Tiempo / F(n)	Tiempo teórico estimado
1000	1263	0,001263000	1.276,548276502
2000	5029	0,001257250	5.106,193106009
3000	11347	0,001260778	11.488,934488520
4000	20012	0,001250750	20.424,772424036
5000	31783	0,001271320	31.913,706912557
6000	48214	0,001339278	45.955,737954082
7000	63481	0,001295531	62.550,865548611
8000	82959	0,001296234	81.699,089696145
9000	103941	0,001283222	103.400,410396684
10000	124812	0,001248120	127.654,827650227
	K=	0,001276548	



Mergesort

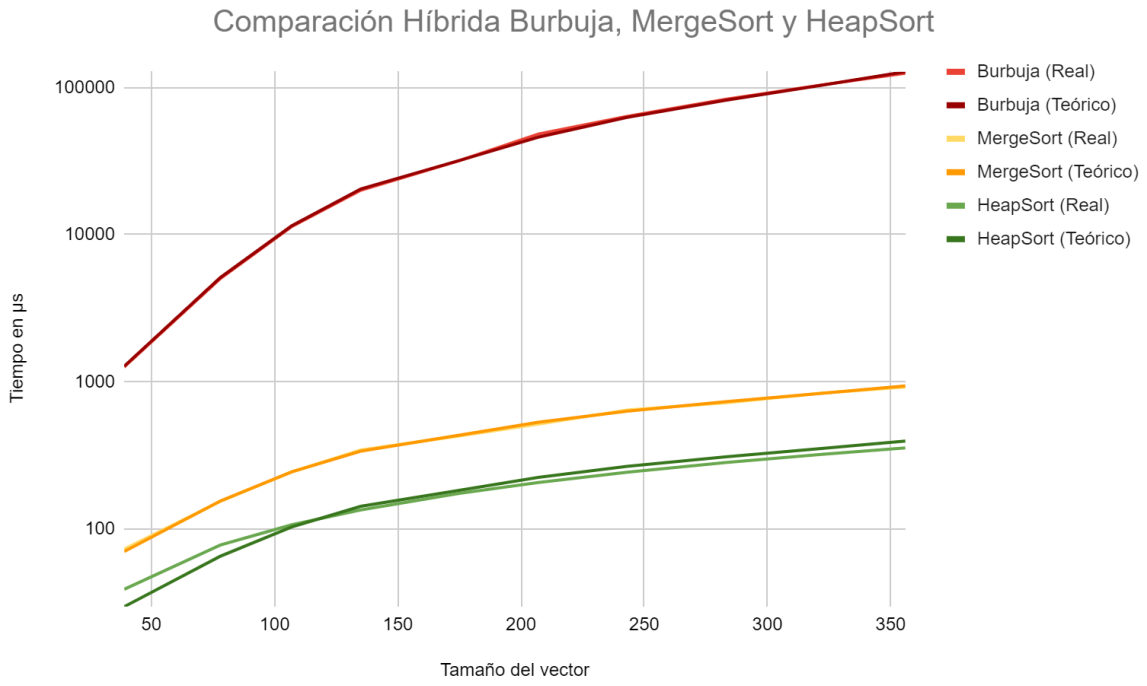
Tamaño Vector	Tiempo en µs.	K = Tiempo / F(n)	Tiempo teórico estimado
1000	73	0,02433333333	70,38618181
2000	154	0,02332605281	154,8979316
3000	244	0,02339099714	244,7412888
4000	345	0,02394463174	338,0469993
5000	429	0,02319564633	433,9272921
6000	516	0,02276245558	531,8592817
7000	644	0,02392656807	631,4974627
8000	721	0,02309067951	732,5962706
9000	839	0,02357524153	834,9720966
10000	923	0,023075	938,4824242
	K=	0,0234620606	



Heapsort

Tamaño Vector	Tiempo en µs.	K = Tiempo / F(n)	Tiempo teórico estimado
1000	39	0,013	29,72240271
2000	78	0,01181449428	65,40969527
3000	107	0,01025752744	103,3483982
4000	135	0,009369638507	142,7491702
5000	175	0,009462093491	183,2371268
6000	207	0,009131450202	224,591466
7000	243	0,009028192611	266,666289
8000	283	0,009063331903	309,3578998
9000	322	0,009047947285	352,5887649
10000	356	0,0089	396,2987029
	K=	0,009907467571	

Obtenidos a partir de los tiempos reales de ejecución de los tres algoritmos las constantes ocultas y los tiempos teóricos estimados, podemos llevar a cabo la comparación híbrida entre los tres algoritmos. Para ello, vamos a representar en la misma gráfica tanto el tiempo real como el teórico estimado de los tres algoritmos para así poder sacar conclusiones de ello:



(Nótese como la gráfica tiene el eje de ordenadas en escala logarítmica debido a la gran diferencia de valores entre el algoritmo de la `burbuja` y los otros dos de `MergeSort` y `HeapSort`.

Una vez hemos representado en la misma gráfica tanto los tiempos reales como los teóricos estimados podemos apreciar varios puntos:

- Definitivamente el **algoritmo más ineficiente es el de la burbuja** ya que es de una complejidad cuadrática y, por lo tanto, ambos tiempos real y teórico estimado del mismo van a aumentar mucho más rápido conforme aumento el tamaño del vector que los otros dos algoritmos.

- Por otra parte, ya podemos resolver la duda que teníamos en la comparación teórica acerca de qué quién del `MergeSort` y del `HeapSort` era más eficiente. En la comparación teórica no podíamos saber esto ya que no podíamos tener en cuenta y, además, no conocíamos las constantes ocultas para cada uno de los algoritmos. Ya en esta comparación, hemos podido apreciar que la constante relativa al algoritmo `HeapSort` es más pequeña que la del `MergeSort`. Por ello, podemos concluir que el **algoritmo de ordenación más eficiente** para un vector de tamaño n es ni más ni menos que el del **`HeapSort`.**