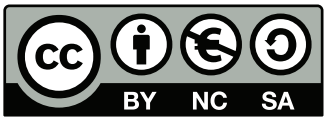
Práctica 1

Análisis de eficiencia de algoritmos

Algorítmica

segfault
Celia Arias Martínez
Miguel Ángel Fernández Gutiérrez
Sergio Quijano Rey
Lucía Salamanca López

UNIVERSIDAD DE GRANADA



Este trabajo se distribuye bajo una licencia CC BY-NC-SA 4.0.

Eres libre de distribuir y adaptar el material siempre que reconozcas a los autores originales del documento, no lo utilices para fines comerciales y lo distribuyas bajo la misma licencia.

creativecommons.org/licenses/by-nc-sa/4.0/

Práctica 1

Análisis de eficiencia de algoritmos

Algorítmica

segfault
Celia Arias Martínez
Miguel Ángel Fernández Gutiérrez
Sergio Quijano Rey
Lucía Salamanca López

UNIVERSIDAD
DE GRANADA

Índice

I Introducción

Esta **práctica 1**, de análisis de eficiencia de algoritmos, consiste en tres partes:

- **Análisis de la eficiencia teórica:** predicción de clase de eficiencia.
- **Análisis de la eficiencia empírica:** ejecución y medición de tiempos.
- **Análisis de la eficiencia híbrida:** obtención de la constante oculta.

A continuación, se explican en más profundidad dichas partes.

Análisis de la eficiencia teórica

El análisis de la eficiencia **teórica** consiste en analizar el tiempo de ejecución en peor caso de los algoritmos, para decidir en qué clase de funciones en notación *O* grande se encuentra. Para ello, hemos utilizado las técnicas de análisis y de resolución de recurrencias que han sido explicadas en clase.

Análisis de la eficiencia empírica

Para el análisis de la eficiencia empírica, hemos ejecutado todos los algoritmos en cada uno de nuestros ordenadores, y los hemos medido con la biblioteca `<chrono>`, haciendo uso del siguiente esquema de código:

```
chrono::time_point<std::chrono::high_resolution_clock> t_ini, t_fin;

...

t_ini = chrono::high_resolution_clock::now();

// se ejecuta el algoritmo cuyo tiempo se quiere medir
ejecutarAlgoritmo();

t_fin = chrono::high_resolution_clock::now();
```

Hemos ejecutado cada algoritmo 100 veces en cada uno de los tamaños que han sido probados (y los cuales se especifican en el apartado de *eficienicia empírica* de cada uno de éstos), y hemos hecho la media de ellos para reducir las perturbaciones que puedan ocurrir azarosamente y que nos lleven al mejor o peor caso, obteniendo de esta forma casos promedio.

Nótese que, en los algoritmos de búsqueda, hemos elegido los mismos intervalos de datos para poder comparar los tiempos y ver la diferencia entre un orden de eficiencia y otro. Esta comparación se realizará en el apartado *Conclusiones*).

Evidentemente, los resultados han variado de acuerdo a cada uno de nuestros ordenadores. A continuación las especificaciones de cada uno de los ordenadores:

Análisis de la eficiencia híbrida

Para el análisis de la eficiencia híbrida, hemos tomado los datos de uno de los alumnos del grupo y hemos hallado la K (constante oculta). Para ello, hemos procedido de dos maneras:

- Tal y como está especificado en el guion de prácticas: hemos realizado la división $\frac{T(n)}{f(n)}$ para cada una de las muestras medias.
- Por otro lado, hemos calculado la constante oculta K mediante la búsqueda de la mejor función de la clase de equivalencia de las funciones $O(T(n))$, y tomando su coeficiente, con la ayuda de *Jupyter*.

II Desarrollo

A continuación, realizaremos el estudio individual de cada algoritmo, como se ha descrito anteriormente.

1 Algoritmo de pivote

```
1  int pivotar(double *v, const int ini, const int fin) {
2      double pivote = v[ini];
3      double aux;
4
5      int i = ini + 1;
6      int j = fin;
7
8      while ( i <= j ) {
9          while ( v[i] < pivote && i <= j ) {
10             i++;
11         }
12
13         while ( v[j] == pivote && j >= i ) {
14             j--;
15         }
16
17         if ( i < j ) {
18             aux = v[i];
19             v[i] = v[j];
20             v[j] = aux;
21         }
22
23         if ( j < ini ) {
24             v[ini] = v[j];
25             v[j] = pivote;
26         }
27
28         return j;
29     }
30 }
```

1.1 Eficiencia teórica

Las operaciones en las *líneas 5-9* son $O(1)$. Nos interesa estudiar el bucle externo y los dos bucles internos. Los `if` internos son $O(1)$, por tanto, de éstos solo nos interesa como afectan a los tres bucles que estudiamos. Estos `if` determinan si hay un cambio en el vector o no; esto no nos interesa, no afectan a los contadores y son $O(1)$.

El bucle `while` externo se ejecuta, si notamos $n = \texttt{j} - \texttt{i}$, n veces. Nos interesa por tanto el número de veces que ejecutamos el bucle y el tiempo de ejecución del cuerpo del bucle.

Los dos bucles `while` internos hacen que se incremente el contador `i` y se decremente el contador `j`. En el peor caso, la variación de contadores será de uno en uno, para que la convergencia a la condición de salida (`i > j`) sea lo más lenta posible. Así, los dos bucles `while` internos son $O(1)$.

Con todo lo dicho, el cuerpo del `while` tiene una eficiencia $O(1)$, y al ejecutar el bucle n veces, tenemos que:

$$T(n) \in O(n)$$

Por tanto, nuestro algoritmo es $\boldsymbol{O(n)}$.

2 Algoritmo de búsqueda binaria (versión iterativa)

```
1  int Busqueda(int *v, int n, int elem){
2      int inicio, fin, centro;
3
4      inicio = 0;
5      fin = n - 1;
6      centro = ( inicio + fin ) / 2;
7
8      while ( ( inicio <= fin ) && ( v[centro] != elem ) ) {
9          if ( elem < v[centro] ) {
10             fin = centro - 1;
11         } else {
12             inicio = centro + 1;
13             centro = (inicio + fin) / 2;
14         }
15     }
16
17     if ( inicio > fin ) {
18         return -1;
19     }
20
21     return centro;
22 }
```

2.1 Eficiencia teórica

Nos encontramos ante un algoritmo de búsqueda binaria en su versión iterativa. La versión recursiva también se estudia más adelante en este documento. Para aplicar este algoritmo, lo lógico es que el vector sobre el que operamos esté ordenado, si no no tiene mucho sentido hacer la búsqueda binaria. A pesar de esto, a la hora de hacer el estudio de eficiencia teórica nos es igual. Lo único que pasaría es que el algoritmo casi nunca encontraría los valores, pero al estudiar el peor caso, el resultado será el mismo esté o no esté ordenado el vector sobre el que trabajamos.

Tanto las *líneas 2-6* como las *líneas 17-22* tienen una complejidad $O(1)$, por lo tanto nos interesa únicamente lo que ocurre en el cuerpo del `while`.

Al estar empleando la notación O grande, nos situamos en el peor de los casos, como ya hemos razonado anteriormente. Por tanto, podemos suponer que no se va a encontrar el elemento buscado. Además, las

líneas 9-14 son $O(1)$, de este modo, con lo anterior, solamente nos interesa la cantidad de veces que se ejecute el bucle `while`, sin tener en cuenta la segunda condición (como ya se ha dicho, suponemos que no se encuentra el elemento buscado).

Podemos suponer sin pérdida de generalidad que movemos el inicio hacia posiciones más avanzadas. En otro caso, podríamos pensar que trabajamos sobre el subvector de la izquierda en vez del subvector de la derecha, esto es indiferente a la hora de calcular el peor caso.

$$\texttt{inicio} = \frac{\texttt{inicio} + \texttt{fin}}{2}$$

Podemos tomar como tamaño de nuestro problema $size = \texttt{fin} - \texttt{inicio}$ para realizar el estudio. Con esto nos queda que el bucle `while` se ejecuta mientras $size \geq 1$ y que, en cada iteración, $size = \frac{size}{2}$.

Por tanto, falta resolver un n_0 mínimo tal que dividiendo n_0 veces $size$ se tenga

$$size \leq 1$$

Podemos notar $size_i$ como el tamaño con el que trabajamos en la iteración i-ésima. Se verifica por tanto:

$$size_i = \frac{size}{2^i}$$

Aplicando esto último a la condición $size \leq 1$ tenemos:

$$size_{n_0} \leq 1$$
$$size_{n_0} = \frac{size}{2^n} \leq 1$$

Esto nos da claramente

$$n_0 \geq \log_2 size$$

Teniendo en cuenta que $size$ actúa como nuestro tamaño de caso, podemos reescribir la notación como $size \equiv n$, con lo que nos queda directamente:

$$T(n) \in O(\log_2 n)$$

Es decir, que nuestro algoritmo es $\boldsymbol{O(\log n)}$.

3 Algoritmo de eliminación de repetidos

```
1  void EliminaRepetidos(double original[], int & nOriginal){
2      int i, j, k;
3
4      for ( i = 0; i < nOriginal; i++ ) {
5          j = i + 1;
6          do {
7              if ( original[j] == original[i] ){
8                  for ( k = j+1; k < nOriginal; k++ ) {
9                      original[k-1] = original[k];
10                     nOriginal--;
11                 }
12             } else {
13                 j++;
```

```
14              }
15          } while ( j < nOriginal );
16      }
17  }
```

3.1 Eficiencia teórica

Podemos estudiar distintas situaciones y ver cuál de ellas tiene una eficiencia peor, quedándonos con esta última. Los casos que podemos estudiar son los dos extremos (todo elemento está repetido una única vez, y no hay elementos repetidos) y el caso medio, algunos elementos repetidos.

Caso 1: Ningún elemento repetido

Como en el resto de casos, el bucle `for` que empieza en la *línea 4* se ejecuta n veces. El cuerpo de este bucle equivale a tener un `for ( int j = i+2; j < nOriginal; j++ )`. Esto es debido a que nunca entramos en el `if` por no tener elementos repetidos. De este modo tenemos que:

$$T_{\text{caso 1}} \in O(n^2)$$

Caso 2: Todos los elementos repetidos

De nuevo, el bucle `for` externo se va a ejecutar n veces, con la diferencia de que vamos a ir restando 1 a n en cada iteración. Por tanto, en realidad ejecutamos este bucle $\frac{n}{2}$ veces.

En el `do-while` vamos a ejecutar el `for` una vez por iteración, y en cada `for` hacemos $n - h$ desplazamientos, para conveniente h. Por tanto, tenemos:

$$T(n) \in O\left(\frac{n}{2} \cdot n\right) \equiv O(n^2) \longrightarrow T(n) \in O(n^2)$$

Caso 3: Algunos elementos repetidos

Podríamos tratar de estudiar este caso, que es el más probable que se produzca, el caso promedio, pero sería bastante complicado. Como queremos saber cuál es el peor de los tres casos, podemos ver que este caso es más favorable que el *Caso 2*, con lo que habríamos acabado.

Para ello solo hace falta pensar que podemos ver el vector como dos subvectores sobre los que aplicamos el algoritmo, un vector v_1 con ningún elemento repetido y un vector v_2 con todos los elementos repetidos al menos una vez. Podemos suponer, sin pérdida de generalidad, que v_1 se encuentra detrás de v_2, pues esto no afectará a la eficiencia. Sobre v_1 tenemos un $T_1(n) \in O(n^2)$. Sobre v_2 tenemos $T_{\text{caso 3}}(n) \leq T_{\text{caso 2}}(n)$. Esto es porque cabe la posibilidad de que un elemento esté más de una vez repetido, por lo que reduciríamos el tiempo de todos los desplazamientos posteriores.

Así tenemos que $T_{\text{caso 3}}(n) \leq T_{\text{caso 2}}(n)$, como se quería.

Conclusión

Con todo lo dicho para los tres casos, hemos visto que

$$T(n) \in O(n^2)$$

Por tanto, el algoritmo es $\boldsymbol{O(n^2)}$.

4 Algoritmo de búsqueda binaria (versión recursiva)

```
1  int BuscarBinario(int *v, const int ini, const int fin, const int x){
2      int centro;
3
4      if ( ini > fin )
5          return -1;
6
7      centro = ( ini + fin ) / 2;
8
9      if ( v[centro] == x )
10         return centro;
11
12     if ( v[centro] > x )
13         return BuscarBinario(v, ini, centro-1, x);
14
15     return BuscarBinario(v, centro+1, fin, x);
16 }
```

4.1 Eficiencia teórica

Llamando n al tamaño del vector, es decir, $n = \texttt{fin} - \texttt{ini}$. Las *líneas 2-7*, así como las condiciones del `if` son $O(1)$. Como este algoritmo es recursivo, estudiaremos su comportamiento cada vez que vuelva a llamarse a sí mismo. En la m-ésima iteración y, suponiendo –en el peor caso– que no encontramos el valor buscado, volvemos a llamar a la función con un subvector de tamaño $\frac{n}{2}$. De este modo, podemos establecer la siguiente relación de recurrencia:

$$T(m) = T\left(\frac{m}{2}\right) + 1$$

Resolveremos esta recurrencia realizando la sustitución de m por 2^k, y despejando:

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

Aplicamos las técnicas de resolución de recurrencias:

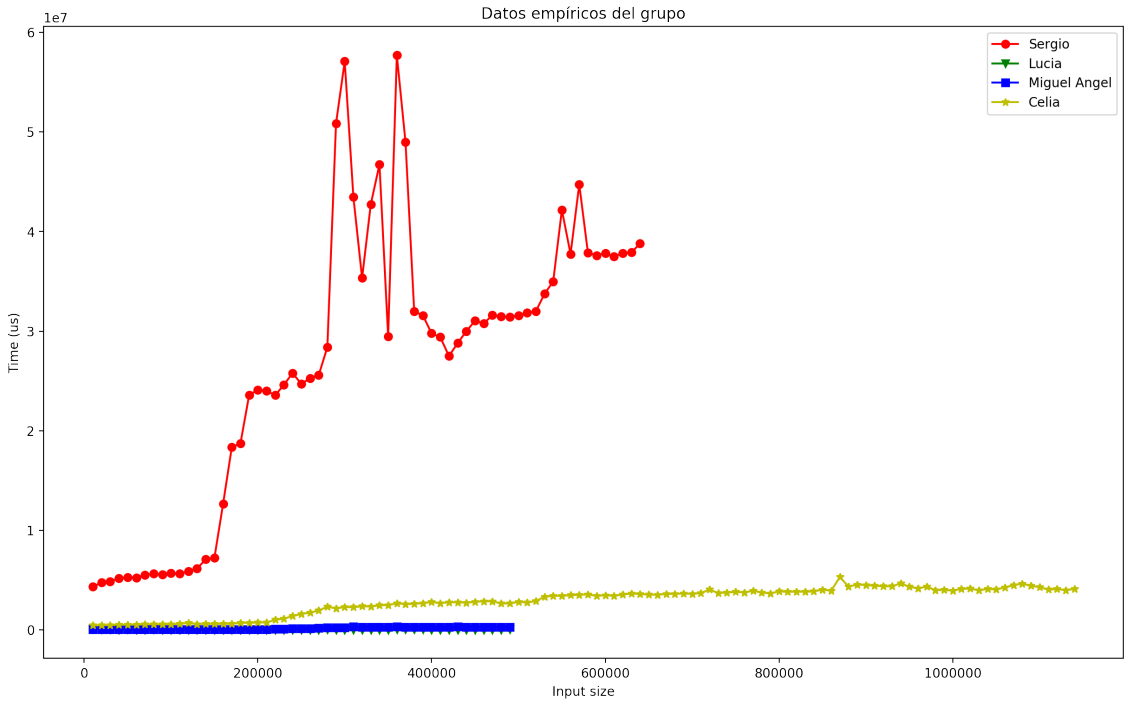
$$\begin{aligned}(x-1)^2 &= 0\\ t_k &= c_1 * 1^k + c_2 * k * 1^k\\ t_k &= c_1 + c_2 * k\\ t_n &= c_1 + c_2 * \log_2(n)\end{aligned}$$

En definitiva, obtenemos que nuestro algoritmo es $\boldsymbol{O(\log n)}$.

4.2 Eficiencia empírica

Tras ejecutar el algoritmo con variando el tamaño del vector `v` desde 10000 hasta 100000 en tramos de 10000, y de ahí hasta 500000 en tramos de 50000; los resultados de ejecución para cada uno de los miembros del grupo fueron los siguientes:

Gráfico 4.1. Datos de todas las ejecuciones

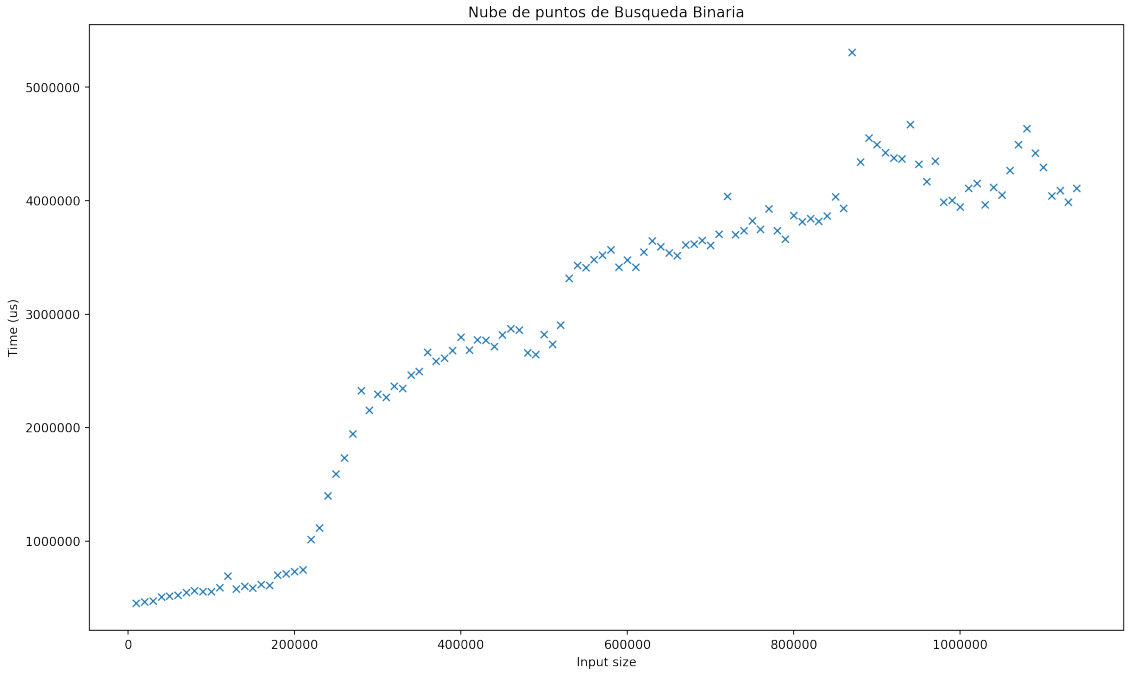


4.3 Eficiencia híbrida

A través de la eficiencia híbrida, comprobaremos que el ajuste teórico realizado es correcto.

Para realizar el análisis de la eficiencia híbrida, tomamos las ejecuciones de uno de los integrantes.

Gráfico 4.2. Datos que serán usados para el análisis híbrido

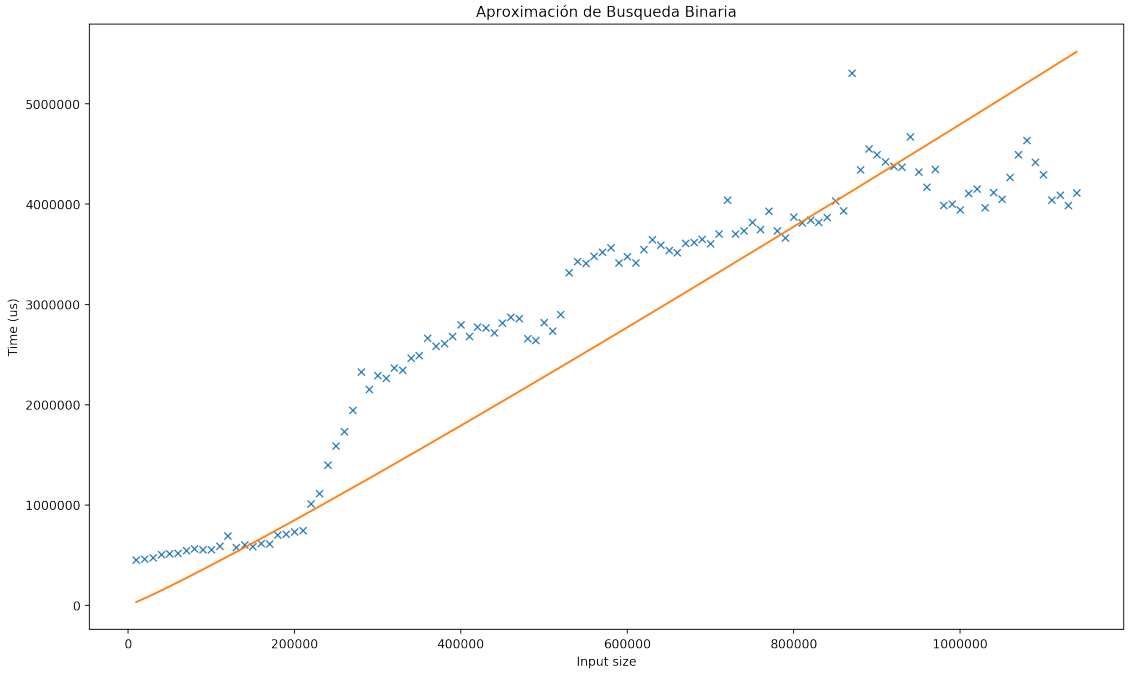


4.3.1 Ajuste de constante oculta

Tras realizar el ajuste de la constante oculta, el valor obtenido es de $K = 0.8630419355569797$.

4.3.2 Ajuste por regresión

Gráfico 4.3. Ajuste por regresión



Realizamos con *Jupyter* el ajuste de regresión para obtener la siguiente función:

$$T(n) = 0.28166261 \cdot n \log n$$

Es importante notar que, en este algoritmo, los datos han salido más dispersos, gracias a su gran velocidad. Debido a esto, hemos tenido que realizar varias pruebas para obtener datos con cierta validez.

En este caso, el mejor error posible lo tiene la recta, esto puede ser así debido a la perturbación de los datos, o al hecho de que $n \log n$ es muy similar a n. A continuación, aparece la relación entre los errores encontrados entre el resto de funciones aproximables (el que tiene mayor error tiene el 100 %, y de ahí los porcentajes restantes son los errores relativos a éste):

- **Error para recta (kn): 7.227564149512647 %**
- Error para cuadrática (kn^2): 56.14209694786215 %
- Error para cúbica (kn^3): 74.09603399513483 %
- Error para logarítmica ($k \log n$): 100.0 %
- **Error para n-logarítmica ($kn \log n$): 13.766671701121327 %**

5 Algoritmo Heap Sort

```
void heapsort(int T[], int num_elem) {
    for ( int i = num_elem/2; i >= 0; i-- ) {
        reajustar(T, num_elem, i);
    }
    for ( int i = num_elem-1; i >= 1; i-- ) {
        int aux = T[0];
        T[0] = T[i];
        T[i] = aux;
        reajustar(T, i, 0);
    }
}
```

```
void reajustar(int T[], int num_elem, int k) {
    int j, v = T[k];
    bool esAPO = false;

    while ( ( k < num_elem/2 ) && !esAPO ) {
        j = 2*k + 1;
        if ( ( j < ( num_elem - 1 ) ) && ( T[j] < T[j+1] ) )
            j++;
        if ( v >= T[j] )
            esAPO = true;
            T[k] = T[j];
            k = j;
    }

    T[k] = v;
}
```

5.1 Eficiencia teórica

Podemos observar que en la función principal `heapsort` se hace una llamada a la función `reajustar`, por lo que, primeramente, vamos a estudiar la eficiencia de esta función.

Estudio de la función `reajustar`

En la función `reajustar` llamaremos n al tamaño del vector, en este caso es la variable `num_elem`. Las cuatro primeras operaciones son $O(1)$, lo que nos lleva a estudiar el bucle `while` para el cálculo de la eficiencia del algoritmo. Como podemos observar en la *línea 2*, la variable `k` es un índice del vector, lo que implica que $0 \leq \texttt{k} \leq n-1$. El peor caso de ejecución será en el que el bucle haga el mayor número de iteraciones, para ello $\texttt{k} < \frac{n}{2}$ el mayor número de veces posible, para que esto se cumpla `k` ha de incrementar lo mínimo posible en cada iteración. En el código vemos que al final del bucle `k` se iguala a `j`, como hemos dicho anteriormente queremos que `k` incremente de forma mínima, por lo que suponemos que el primer `if` (con eficiencia $O(1)$) no se cumple ya que incrementa la `j` y al final repercute en el aumento de la `k`. También, para que el número de iteraciones sea máxima, el segundo `if` no ha de cumplirse porque al cambiar `esAPO`, saldría del bucle.

A partir de todo esto podemos observar que $k_i = 2k_{i-1} + 1$, siendo i el número de la iteración. Obtenemos así una recurrencia que procedemos a resolver:

$$k_i = 2^i \cdot c$$

Con la solución particular:

$$x = 2x + 1$$
$$x = -1$$

Obteniendo como solución de la recurrencia:

$$k_i = 2^i \cdot c - 1$$

Como queremos que haga el número máximo de iteraciones y $0 \leq \texttt{k} \leq n-1$, necesitamos que `k` esté "lo más alejado posible" de $\frac{n}{2}$ por lo que suponemos que la `k` inicial vale 0, es decir, $\texttt{k}_0 = 0$, por lo que

$$k_0 = 2^0 \cdot c - 1$$
$$0 = 1 \cdot c - 1$$
$$c = 1$$

Obtenemos así la solución de la recurrencia:

$$k_i = 2^i - 1$$

Para ver el número máximo de iteraciones tenemos que ver en qué i de la iteración es $k = \frac{n}{2}$.

$$\begin{aligned} 2^i - 1 &= \frac{n}{2} \\ 2^i &= \frac{n-2}{2} \\ i &= \log_2 \frac{n-2}{2} \\ i &= \log_2(n-2) - \log_2 2 \\ i &= \log_2(n-2) - 1 \end{aligned}$$

Con todo lo dicho, el cuerpo del `while` tiene una eficiencia $O(log_2 n)$, y al ejecutar el bucle n veces, tenemos que:

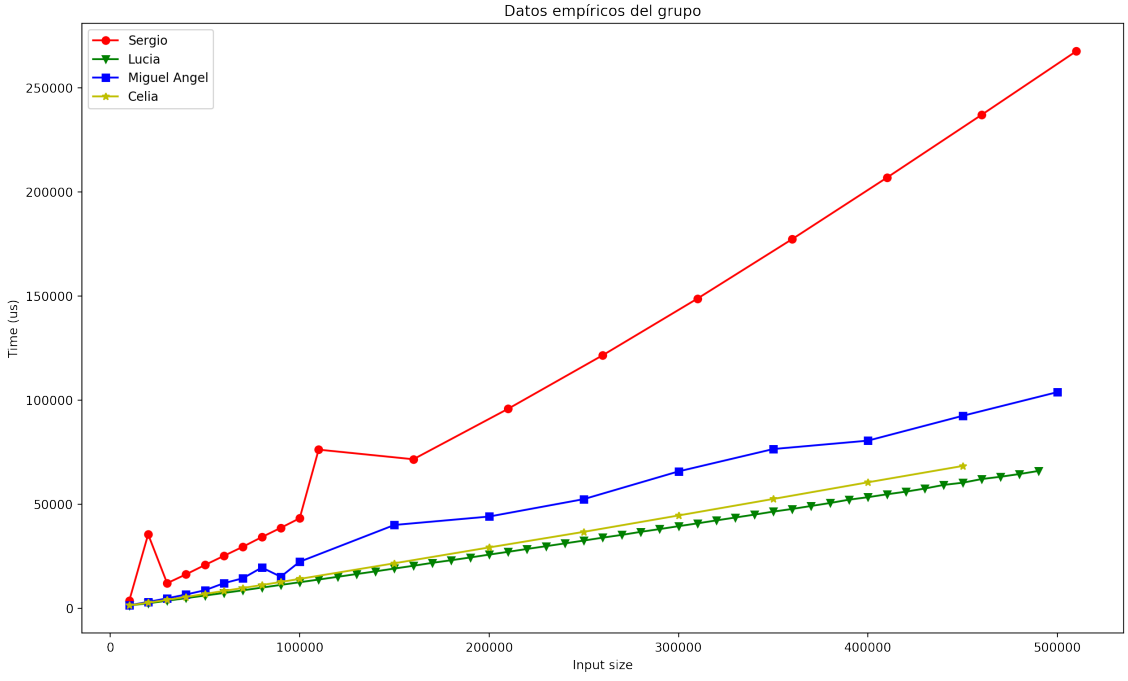
$$T(n) \in O(\log n)$$

Habiendo estudiado la eficiencia de reajustar pasamos a estudiar la eficiencia del `heapsort`. Lo más relevante del código son los dos `for` , en el interior de ambos la eficiencia es la de la función `reajustar`, ya que el resto de operaciones son $O(1)$, el primer `for` se ejecuta $\frac{n}{2}$ veces, siendo n el tamaño del vector, y el segundo `for` se ejecuta $n-1$ veces, por lo que la eficiencia del algoritmo es $O((n-1)\log_2 n) \equiv \boldsymbol{O(n \log n)}$.

5.2 Eficiencia empírica

Tras ejecutar el algoritmo con `num_elem` desde 10000 hasta 100000 en tramos de 10000, y de ahí hasta 500000 en tramos de 50000; los resultados de ejecución para cada uno de los miembros del grupo fueron los siguientes:

Gráfico 5.1. Datos de todas las ejecuciones

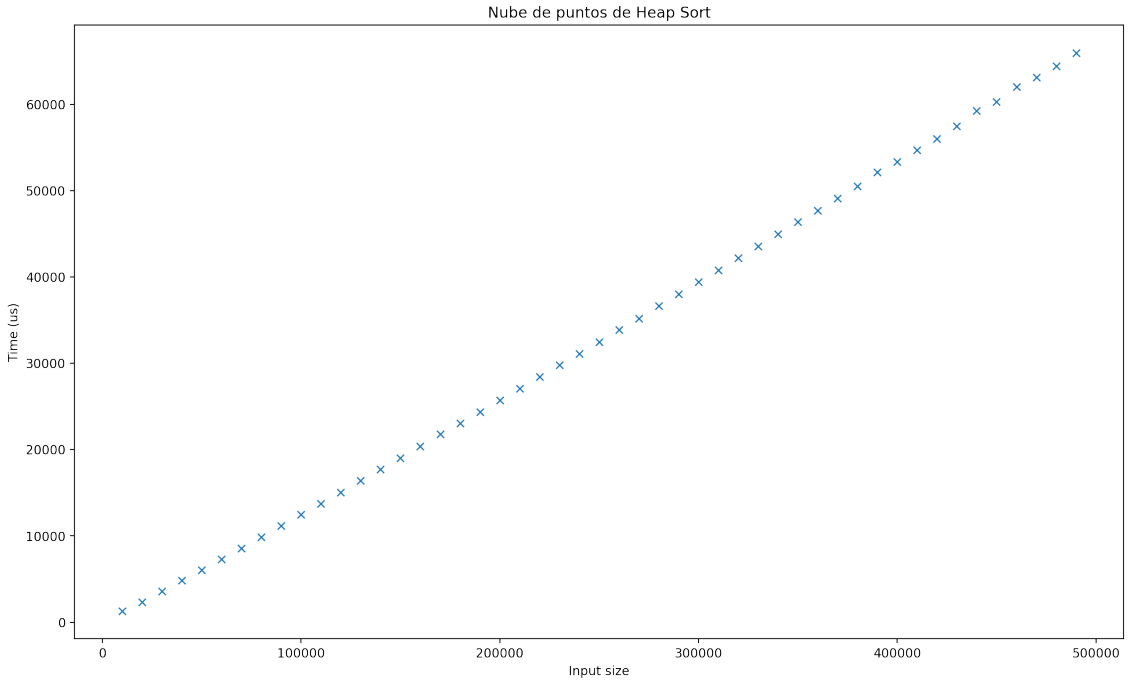


5.3 Eficiencia híbrida

A través de la eficiencia híbrida, comprobaremos que el ajuste teórico realizado es correcto.

Para realizar el análisis de la eficiencia híbrida, tomamos las ejecuciones de uno de los integrantes.

Gráfico 5.2. Datos que serán usados para el análisis híbrido

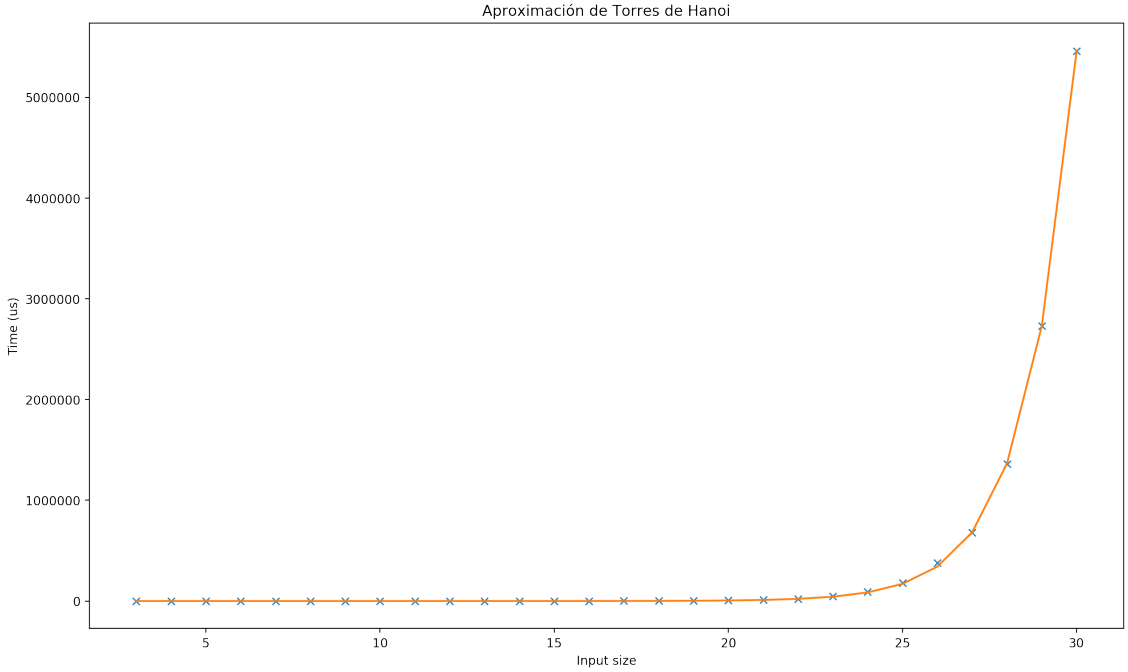


5.3.1 Ajuste de constante oculta

Tras realizar el ajuste de la constante oculta, el valor obtenido es de $K = 0.9753044058661043$.

5.3.2 Ajuste por regresión

Gráfico 5.3. Ajuste por regresión



Realizamos con *Jupyter* el ajuste de regresión para obtener la siguiente función:

$$T(n) = 0.00718669 \cdot n \log n$$

En este caso, el mejor error posible lo tiene la recta, esto puede ser así debido a la perturbación de los datos, o al hecho de que $n \log n$ es muy similar a n. A continuación, aparece la relación entre los errores encontrados entre el resto de funciones aproximables (el que tiene mayor error tiene el 100 %, y de ahí los porcentajes restantes son los errores relativos a éste):

- **Error para recta (kn): 0.04033354197487824 %**
- Error para cuadrática (kn^2): 27.33530020306984 %
- Error para cúbica (kn^3): 74.09603399513483 %
- Error para logarítmica ($k \log n$): 100.0 %
- **Error para n-logarítmica ($kn \log n$): 0.04894682223025979 %**

6 Algoritmo Bubble Sort

```
1  void burbuja(int v[], int n) {
2      int aux;
3
4      for ( int i = inicial, i < final-1; i++ ) {
5          for ( j = final-1; j > i; j-- ) {
6              if ( v[j] < v[j-1] ) {
7                  aux = v[j];
8                  v[j] = v[j-1];
9                  v[j-1] = aux;
10             }
11         }
12     }
13 }
```

6.1 Eficiencia teórica

Este algoritmo es iterativo, por lo que nos centraremos en el estudio de los bucles `while` y `for` anidados que se encuentran en el código.

En las *líneas 6-10*, vemos que las operaciones del bloque condicional son $O(1)$. Comenzaremos estudiando los bucles de dentro hacia afuera, para ello, llamaremos n al tamaño del vector.

Estudiaremos el bucle `for` de la *línea 5*. Este bucle se ejecuta de acuerdo a un `i` fijado por el bucle `for` más externo *línea 4*: desde `fin` -1 hasta `i`, en decrementos de 1. Esto se ejecutará tantas veces como dicta el `for` más externo: desde 0 hasta `final` -2. En definitiva, tenemos que:

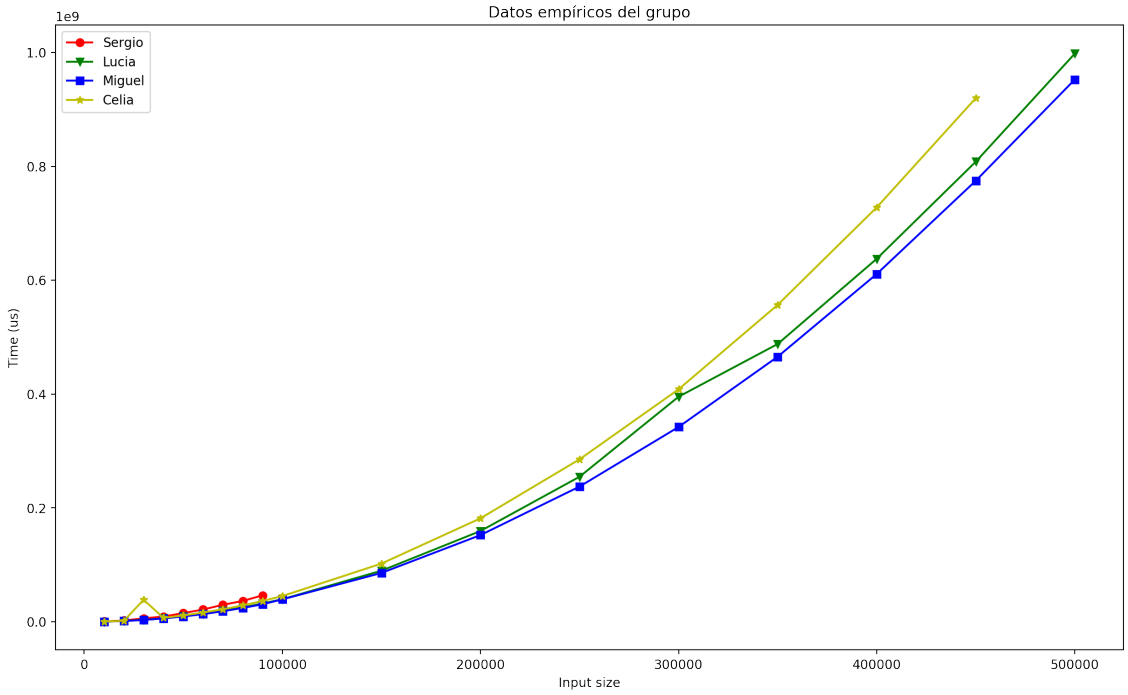
$$T(n) = \sum_{i=0}^{n-1} (n-i-1) = n^2 - \frac{(n-2)(n-1)}{2} - 5 \in O(n^2)$$

Por tanto, este algoritmo es $\boldsymbol{O(n^2)}$.

6.2 Eficiencia empírica

Tras ejecutar el algoritmo con `n` desde 10000 hasta 100000 en tramos de 10000, y de ahí hasta 500000 en tramos de 50000; los resultados de ejecución para cada uno de los miembros del grupo fueron los siguientes:

Gráfico 6.1. Datos de todas las ejecuciones

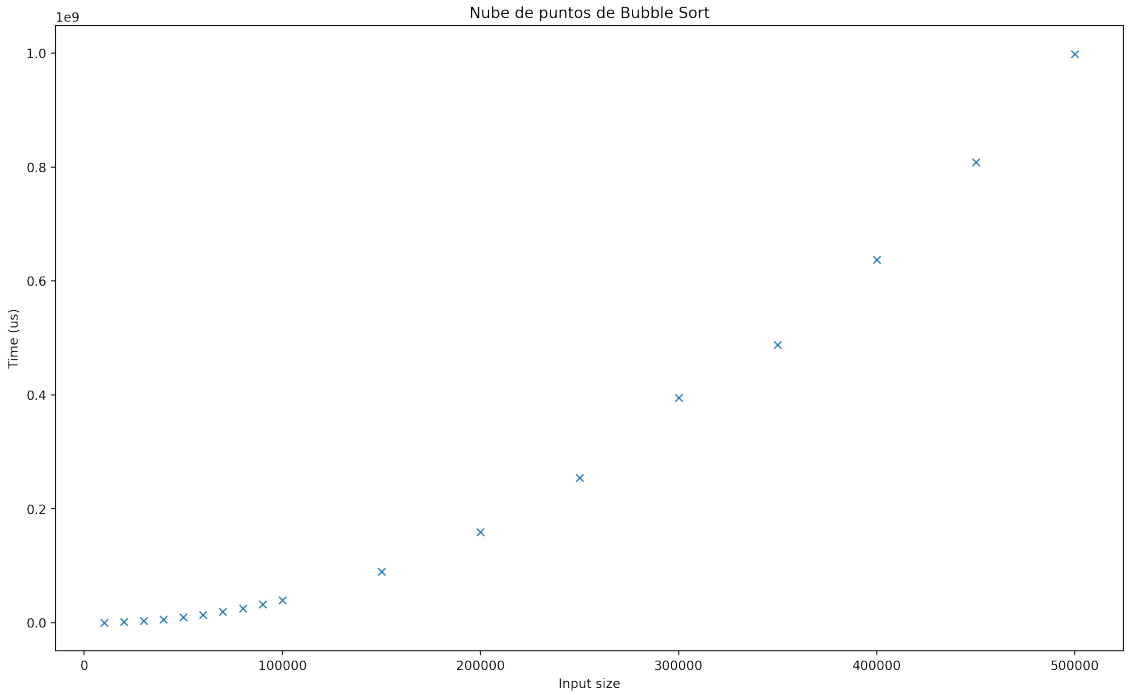


6.3 Eficiencia híbrida

A través de la eficiencia híbrida, comprobaremos que el ajuste teórico realizado es correcto.

Para realizar el análisis de la eficiencia híbrida, tomamos las ejecuciones de uno de los integrantes.

Gráfico 6.2. Datos que serán usados para el análisis híbrido

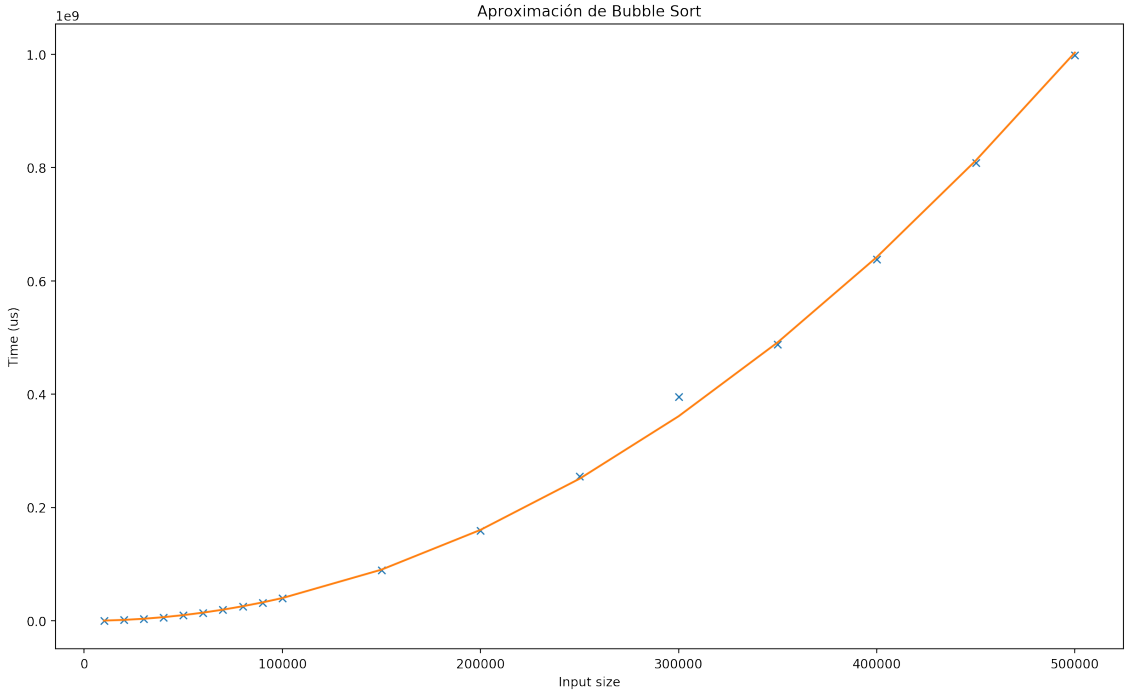


6.3.1 Ajuste de constante oculta

Tras realizar el ajuste de la constante oculta, el valor obtenido es de $K = 1.019659179330859$.

6.3.2 Ajuste por regresión

Gráfico 6.3. Ajuste por regresión



Realizamos con *Jupyter* el ajuste de regresión para obtener la siguiente función:

$$T(n) = 0.00401246 \cdot n^2$$

Que es la que mejor aproxima a nuestros datos. Nótese que es quien tiene el mejor error posible de entre el resto de funciones aproximables (el que tiene mayor error tiene el 100 %, y de ahí los porcentajes restantes son los errores relativos a éste):

- Error para recta (kn): 6.15853318137504 %
- **Error para cuadrática (kn^2): 0.08527050249560174 %**
- Error para cúbica (kn^3): 5.41027219494451 %
- Error para logarítmica ($k \log n$): 100.0 %
- Error para n-logarítmica ($kn \log n$): 10.096552530276018 %

7 Algoritmo Merge Sort

```
1  static void mergesort_lims(int T[], int inicial, int final) {
2      if ( final-inicial < UMBRAL_MS ) {
3          insercion_lims(T, inicial, final);
4      } else {
5          int k = (final - inicial)/2;
6
7          int * U = new int [k - inicial + 1];
8          assert(U);
9          int l, l2;
10         for (l = 0, l2 = inicial; l < k; l++, l2++)
11             U[l] = T[l2];
12         U[l] = INT_MAX;
13
14         int * V = new int [final - k + 1];
15         assert(V);
16         for (l = 0, l2 = k; l < final - k; l++, l2++)
17             V[l] = T[l2];
18         V[l] = INT_MAX;
19
20         mergesort_lims(U, 0, k);
21         mergesort_lims(V, 0, final - k);
22         fusion(T, inicial, final, U, V);
23         delete [] U;
24         delete [] V;
25     }
26 }
```

```
1  static void fusion(int T[], int inicial, int final, int U[], int V[]) {
2      int j = 0, k = 0;
3      for ( int i = inicial; i < final; i++ ) {
4          if (U[j] < V[k]) {
5              T[i] = U[j];
6              j++;
7          } else {
8              T[i] = V[k];
9              k++;
10         }
11     }
12 }
```

7.1 Eficiencia teórica

Sabemos que el algoritmo de inserción es $O(n^2)$, pero solo se ejecutará si el tamaño del vector es suficientemente pequeño. Estudiaremos el caso contrario.

Lo primero que hace es dividir el vector en dos partes iguales. Las instrucciones de las *líneas 5-9, 12-15 y 18* son $O(1)$ y los dos bucles `for` se repiten cada uno $\frac{k}{2}$, por tanto llegado a este punto tenemos eficiencia $O(n)$. En las *líneas 20 y 21* hacemos dos llamadas recursivas a `mergesort_lims` y en la *línea 22* llamamos a `fusion`. Estudiaremos estos dos hechos por separado.

Estudio de la función `fusion`

La función `fusion`, por un lado, tiene un bucle `for` que se ejecuta n veces. Las demás instrucciones son $O(1)$, por tanto, podemos decir que su eficiencia es $O(n)$.

Estudio de la función `mergesort_lims`

Las dos llamadas recursivas a `mergesort_lims` se hacen sobre vectores que constituyen cada uno la mitad del vector original. Construimos la ecuación de recurrencia teniendo en cuenta todo lo anteriormente especificado, y sólo el caso $n \geq$ `UMBRALMS`.

$$T(m) = 2T(\frac{m}{2}) + n$$

Sustituyendo m por 2^k y despejando obtenemos:

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

Resolvemos la recurrencia:

$$(x-2) * (x-2) = 0$$
$$(x-2)^2 = 0$$

$$t_k = c_1 * 2^k + c_2 * k * 2^k$$
$$t_n = c_1 * n + c_2 * \log_2(n) * n$$

Tenemos que este algoritmo es $O(\log n)$.

Conclusión

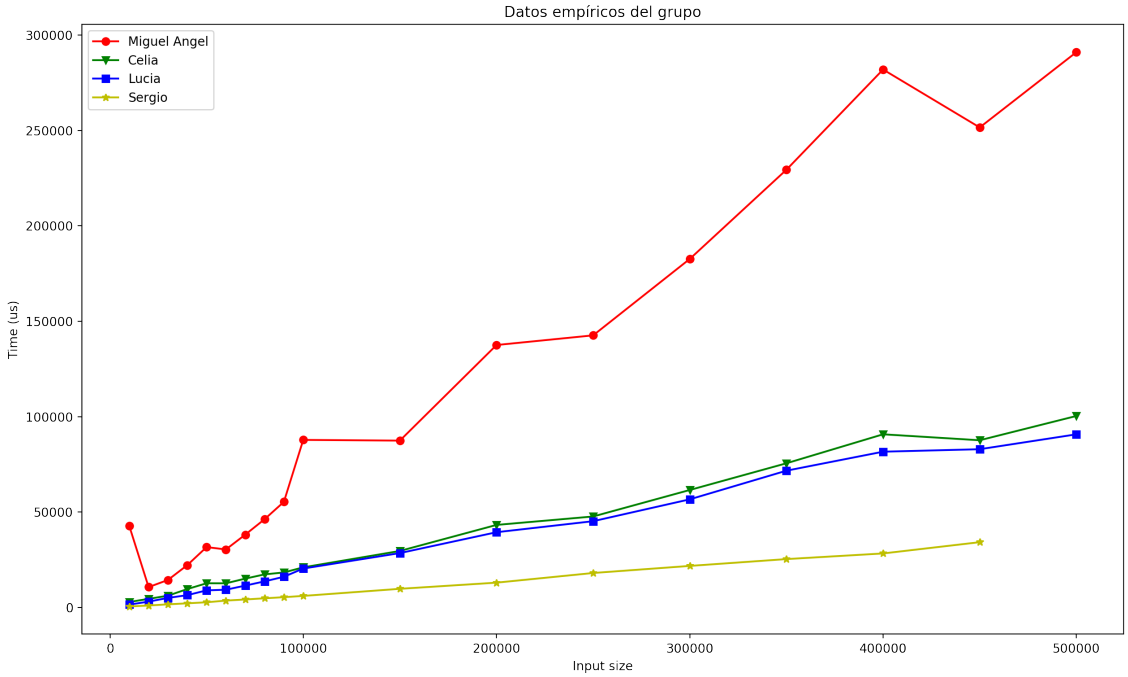
Por tanto deducimos que el algoritmo es $\boldsymbol{O(n \log_2(n))}$.

Hay que señalar que, si el tamaño del vector es suficientemente pequeño, el algoritmo sería $O(n^2)$ –al usar el algoritmo de inserción en ese caso– pero no significará un retraso en los tiempos de ejecución ya que este algoritmo funciona bien para casos no demasiado grandes.

7.2 Eficiencia empírica

Tras ejecutar el algoritmo con un tamaño del vector variando desde 10000 hasta 100000 en tramos de 10000, y de ahí hasta 500000 en tramos de 50000; los resultados de ejecución para cada uno de los miembros del grupo fueron los siguientes:

Gráfico 7.1. Datos de todas las ejecuciones

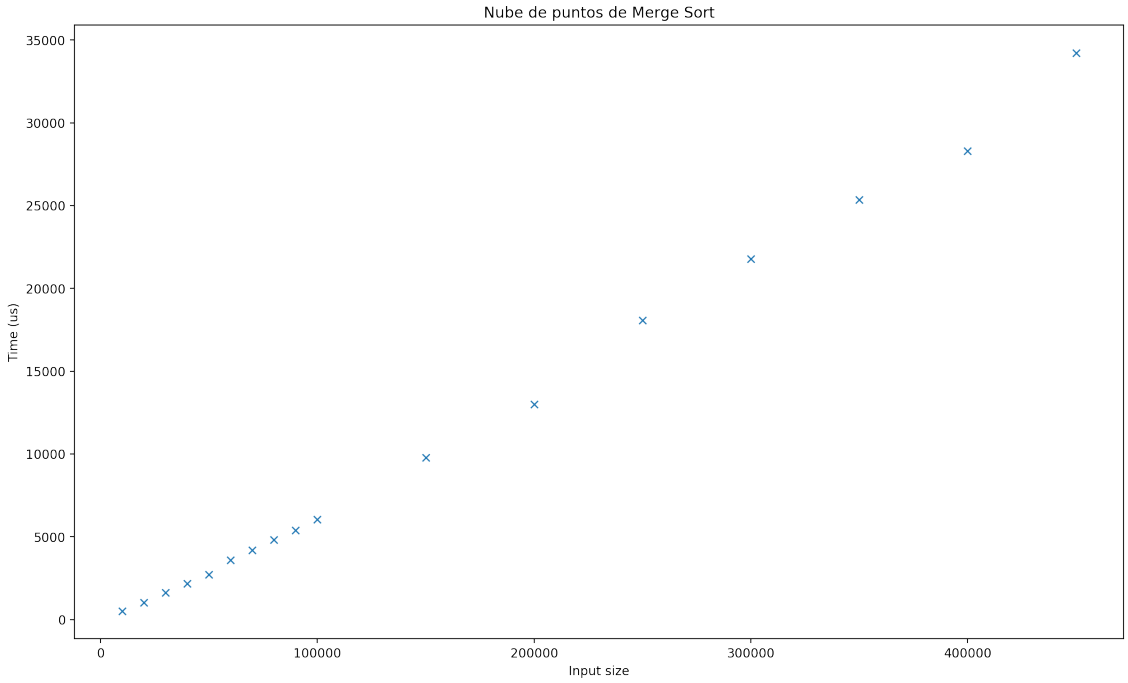


7.3 Eficiencia híbrida

A través de la eficiencia híbrida, comprobaremos que el ajuste teórico realizado es correcto.

Para realizar el análisis de la eficiencia híbrida, tomamos las ejecuciones de uno de los integrantes.

Gráfico 7.2. Datos que serán usados para el análisis híbrido

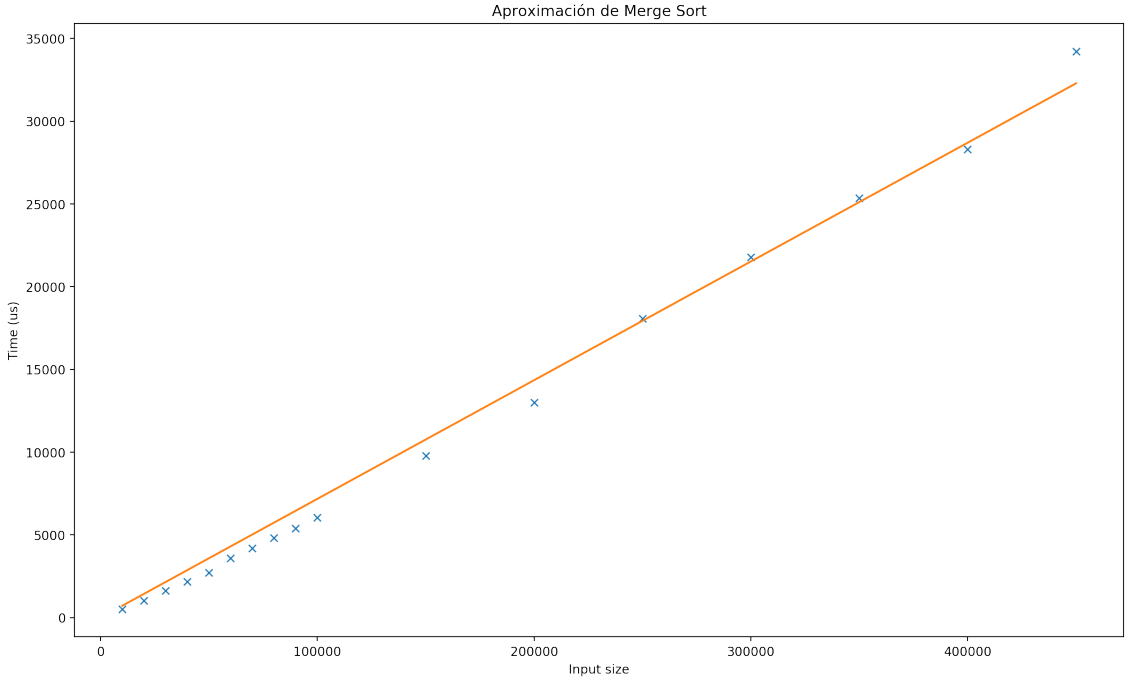


7.3.1 Ajuste de constante oculta

Tras realizar el ajuste de la constante oculta, el valor obtenido es de $K = 1.1670955046281213$.

7.3.2 Ajuste por regresión

Gráfico 7.3. Ajuste por regresión



Realizamos con *Jupyter* el ajuste de regresión para obtener la siguiente función:

$$T(n) = 0.07175341 \cdot n \log n$$

Que es la que mejor aproxima a nuestros datos. Nótese que es quien tiene el mejor error posible de entre el resto de funciones aproximables (el que tiene mayor error tiene el 100 %, y de ahí los porcentajes restantes son los errores relativos a éste):

- Error para recta (kn): 0.3473246336299704 %
- Error para cuadrática (kn^2): 14.334551135315282 %
- Error para cúbica (kn^3): 37.228355082448616 %
- Error para logarítmica ($k \log n$): 100.0 %
- **Error para n-logarítmica ($kn \log n$): 0.246638483904476 %**

8 Algoritmo de las torres de Hanoi

```
void hanoi(int M, int i, int j) {
    if ( M > 0 ) {
        hanoi(M-1, i, 6-i-j);
        hanoi(M-1, 6-i-j, j);
    }
}
```

8.1 Eficiencia teórica

Estamos ante un algoritmo recursivo, así que buscaremos la relación de recurrencia. Si nos situamos en la iteración n-ésima, el algoritmo efectuará una comprobación en el `if` de eficiencia $O(1)$, y volverá a llamarse a sí misma dos veces. Por tanto, la ecuación de recurrencia que caracteriza a este algoritmo es:

$$T_n = 2T_{n-1} + 1$$

Despejando y operando, resolvemos esta recurrencia:

$$(x-2)(x-1) = 0$$

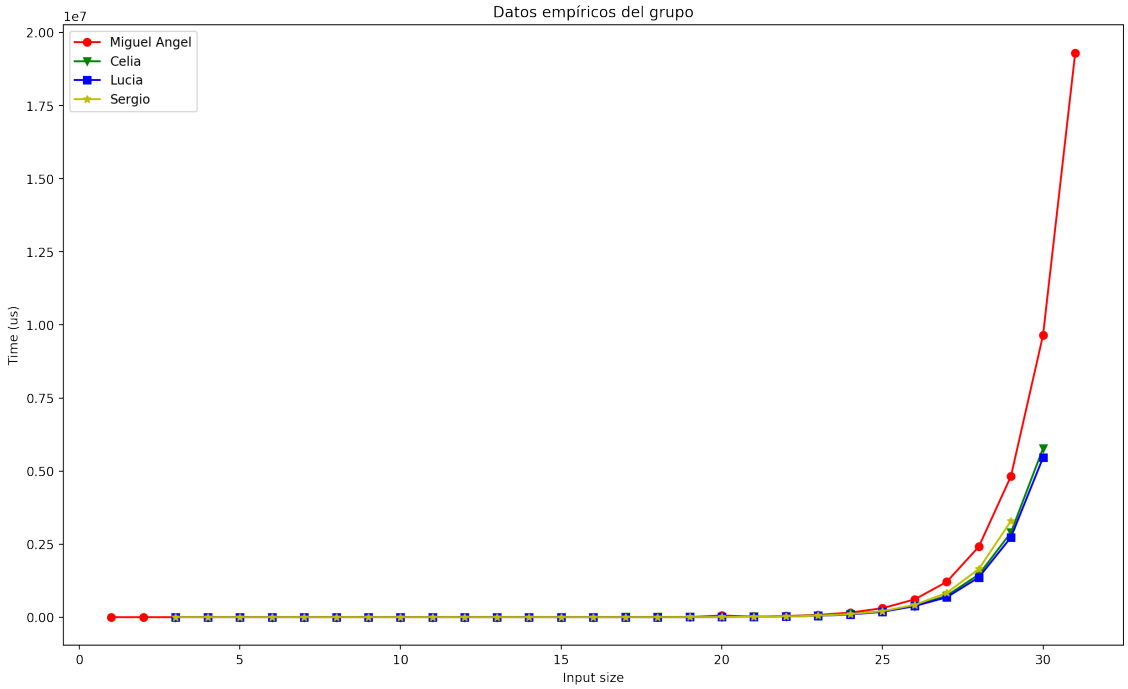
$$t_n = c_1 \cdot 2^n + c_2 \cdot 1^n = c_1 \cdot 2^n + c_2$$

En definitiva, nuestro algoritmo tiene una eficiencia $\boldsymbol{O(2^n)}$.

8.2 Eficiencia empírica

Tras ejecutar el algoritmo con `M` desde 2 hasta 30, los resultados de ejecución para cada uno de los miembros del grupo fueron los siguientes:

Gráfico 8.1. Datos de todas las ejecuciones

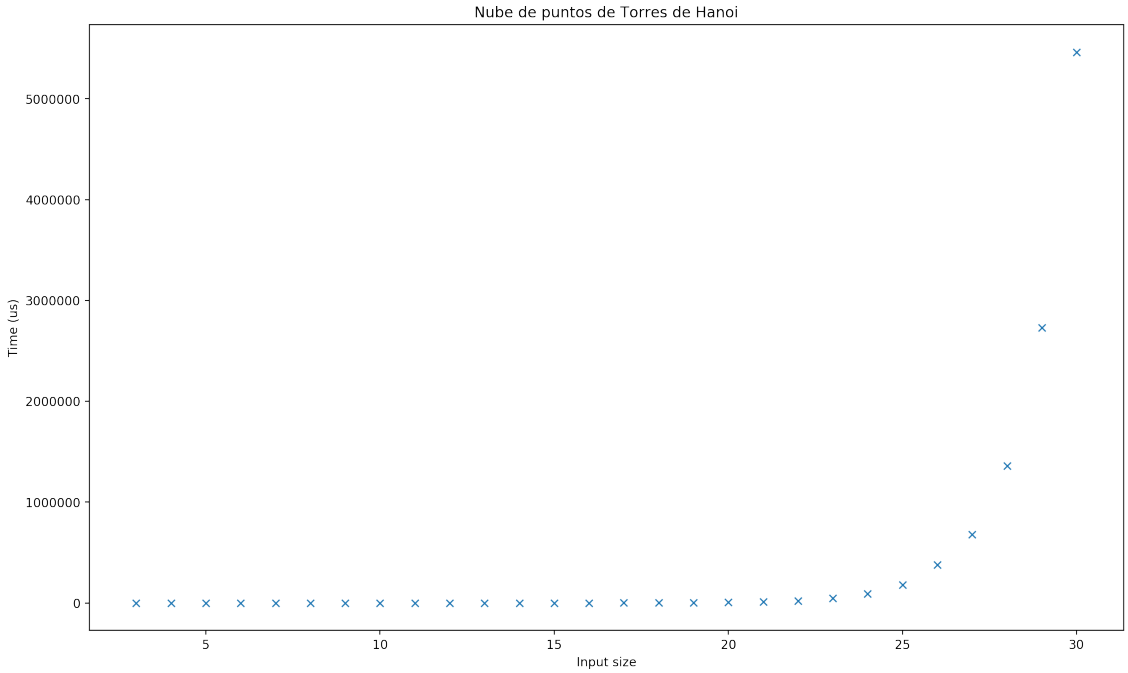


8.3 Eficiencia híbrida

A través de la eficiencia híbrida, comprobaremos que el ajuste teórico realizado es correcto.

Para realizar el análisis de la eficiencia híbrida, tomamos las ejecuciones de uno de los integrantes.

Gráfico 8.2. Datos que serán usados para el análisis híbrido

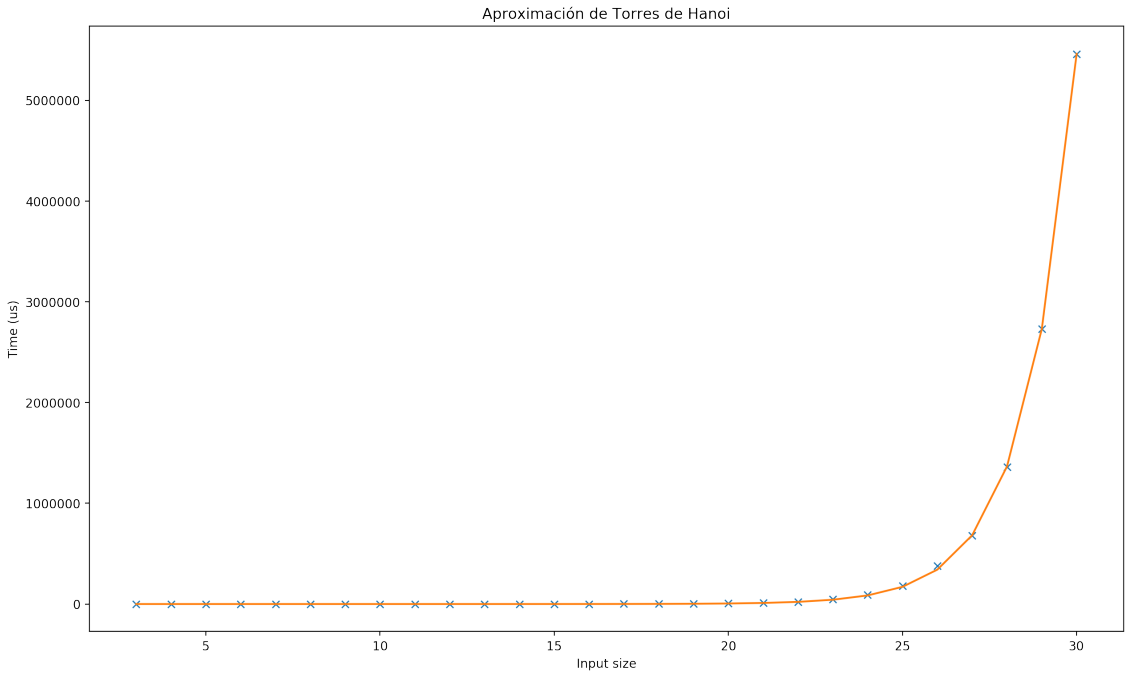


8.3.1 Ajuste de constante oculta

Tras realizar el ajuste de la constante oculta, el valor obtenido es de $K = 0.7263436302553232$.

8.3.2 Ajuste por regresión

Gráfico 8.3. Ajuste por regresión



Realizamos con *Jupyter* el ajuste de regresión para obtener la siguiente función:

$$T(n) = 0.00508635 \cdot 2^n$$

Que es la que mejor aproxima a nuestros datos. Nótese que es quien tiene el mejor error posible de entre el resto de funciones aproximables (el que tiene mayor error tiene el 100 %, y de ahí los porcentajes restantes son los errores relativos a éste):

- Error para recta (kn): 76.01343903600339 %
- Error para cuadrática (kn^2): 71.13401593179258 %
- Error para cúbica (kn^3): 57.61256154034523 %
- Error para exponencial (ke^n): 3.1920070654956563 %
- **Error para potencial ($k2^n$): 0.004591299781178793 %**
- Error para logarítmica ($k \log n$): 100.0 %

III Conclusiones

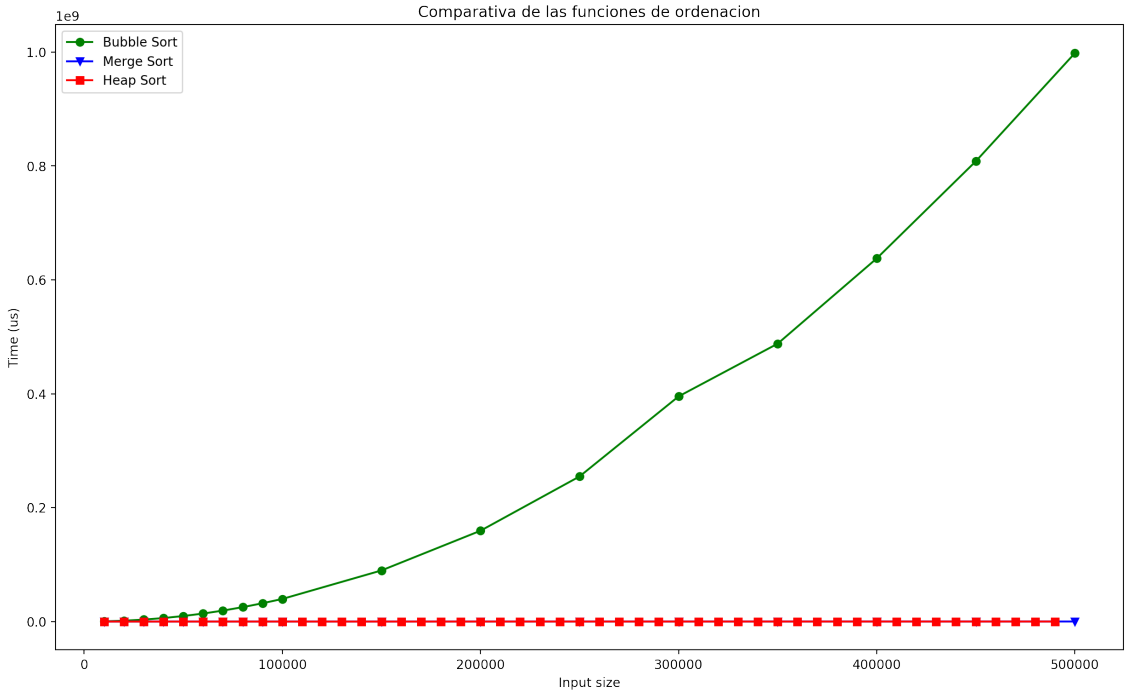
Tras realizar los análisis teóricos de los diversos algoritmos, hemos visto que con los datos obtenidos empíricamente y utilizando una curva de regresión podemos ajustar adecuadamente a la función que deberíamos obtener. Con todo esto, deducimos que nuestro análisis teórico ha sido correcto.

También hemos podido comprobar que los datos obtenidos en la práctica dependen de la arquitectura del ordenador de cada componente del grupo, aún así las funciones son las mismas, siendo esto de mayor relevancia en el análisis de algoritmos.

Comparación entre algoritmos de búsqueda

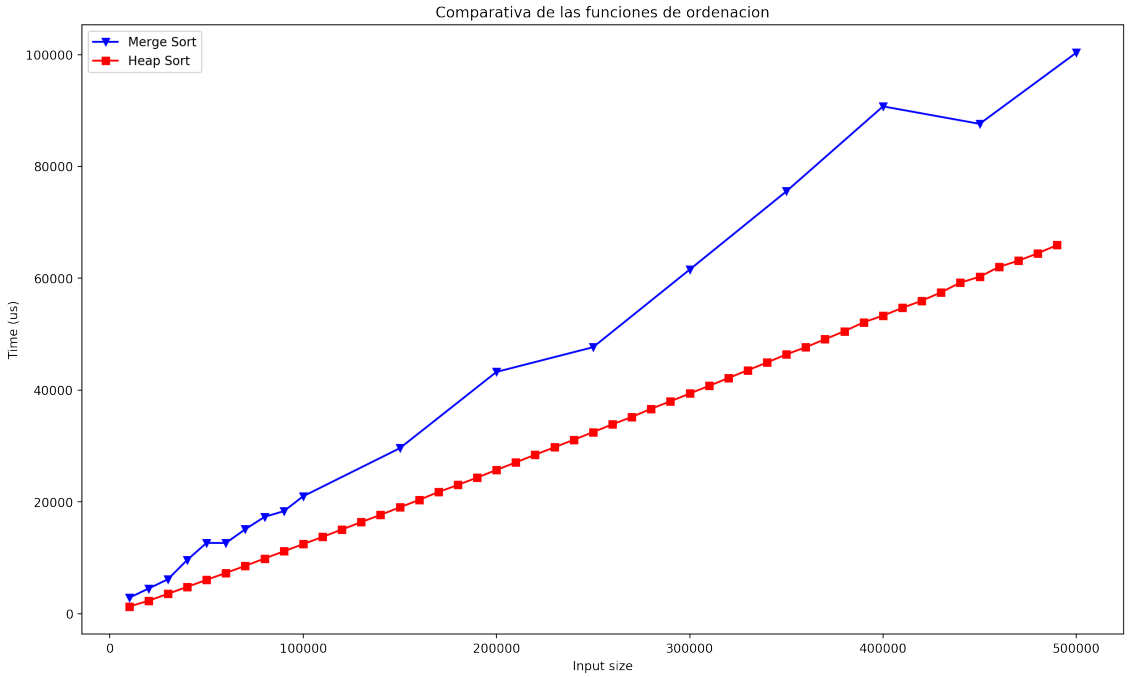
Tras haber ejecutado los códigos y haber analizado los algoritmos teóricamente hemos comprobado que *Heap Sort* y *Merge Sort* son notablemente mejores que *Bubble Sort*. Esto se debe a que el orden de eficiencia de los primeros es $n \log n$, al contrario que *Bubble Sort*, que es n^2.

Gráfico III.1. Comparación entre algoritmos de búsqueda



Ya que la diferencia es aplastante, nos centraremos en los algoritmos *Merge Sort* y *Heap Sort*.

Gráfico III.2. Comparación entre algoritmos de búsqueda: *Merge Sort* y *Heap Sort*



Entre estos dos algoritmos con orden de eficiencia mejor, podemos ver que, tras haber realizado un estudio experimental de ambos, el mejor de ellos es *Heap Sort*.