Práctica 1: Eficiencia

Por: Sara Bellarabi El Fazazi, Manuel Villatoro Guevara , Arturo Sánchez Cortés, Sergio Vargas Martin

Índice

Burbuja: eficiencia teórica

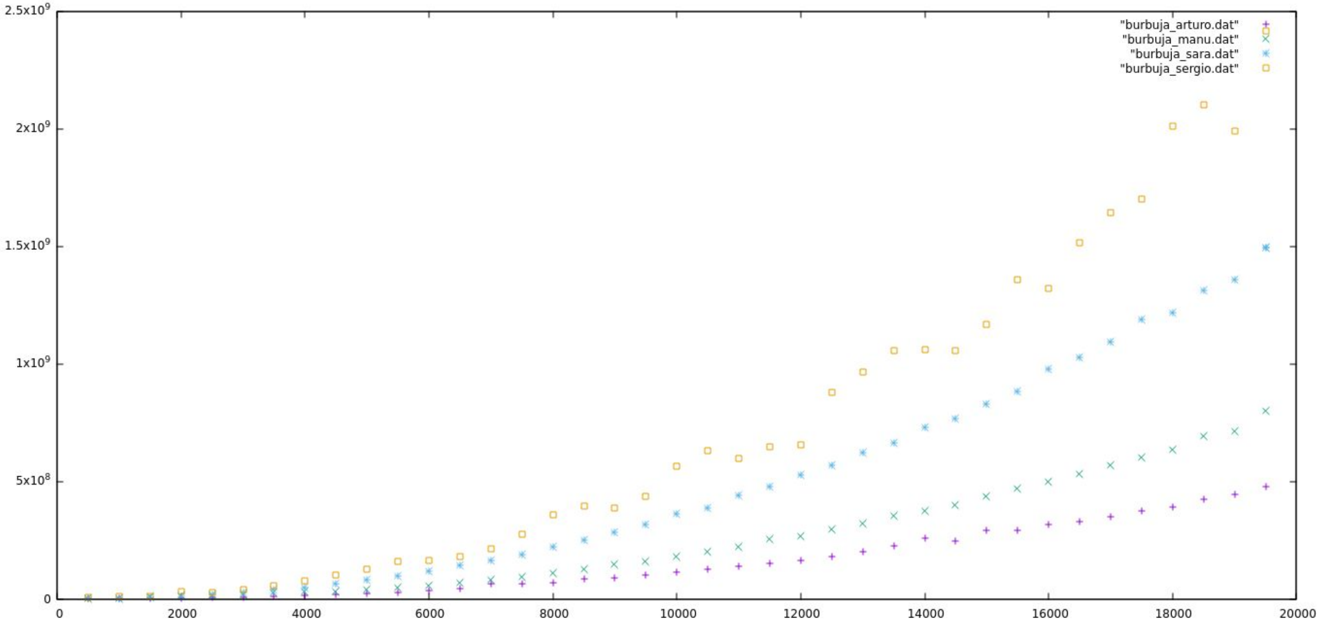
```
void OrdenaBurbuja(int *v, int n) {
 int i, j, aux;
 bool haycambios = true;

 i = 0;
 while (haycambios) {
   haycambios = false;
   for (j = n - 1; j > i; j--) {
     if (v[j - 1] > v[j]) {
       aux = v[j];
       v[j] = v[j - 1];
       v[j - 1] = aux;
       haycambios = true;
     }
   }
 }
}
```

Dentro del while, el algoritmo recorre el vector con un bucle for y compara el elemento anterior con el actual. Si es mayor se hace un cambio de variable y pone a true la variable que regula el bucle while exterior.

Mejor caso: En caso de tener un vector ya ordenado solo se ejecutará el bucle for interno por lo que el algoritmo es de orden Ω (n)

Peor caso: el bucle for interno se ejecutará tantas veces como elementos mal posicionados haya en el array, por tanto el algoritmo será de orden $O(n^2)$.

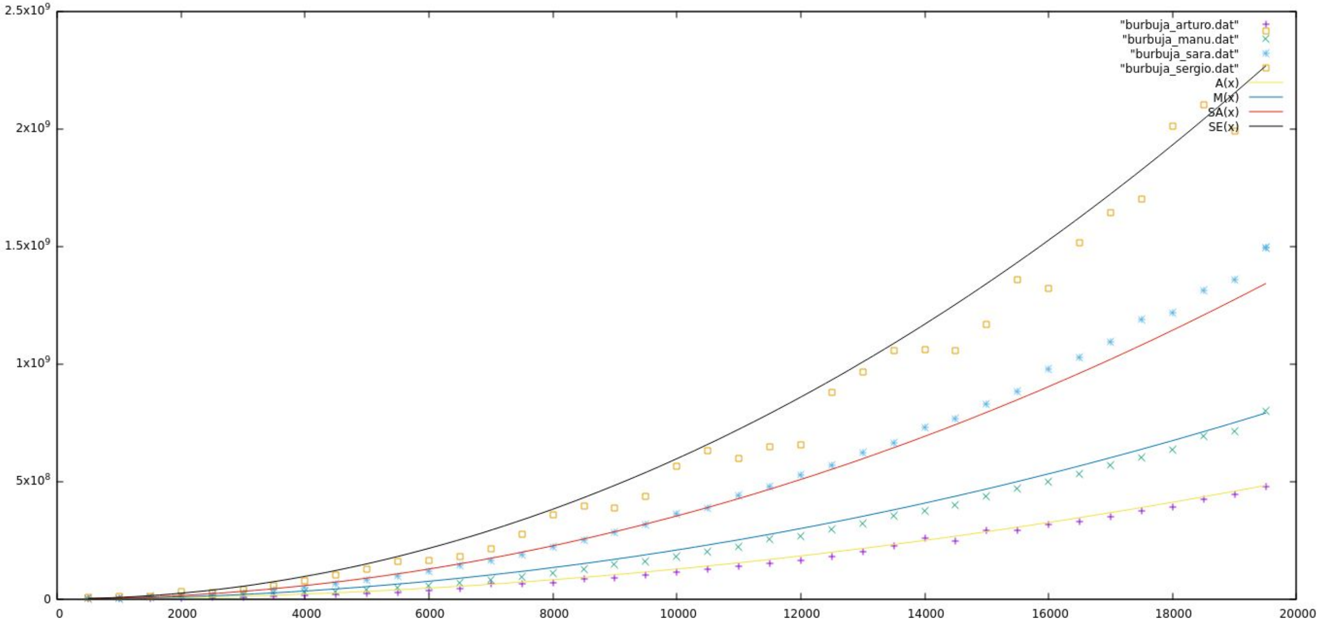
Burbuja: eficiencia empírica



Cálculo de las constantes ocultas

```cpp
#include <cmath>
#include <fstream>
#include <iostream>
#include <vector>

using namespace std;

template <class T> double media_vector(vector<T> v){
 double k = 0;
 for (auto i : v)
   k += i;
 return k / v.size();
}

double orden(double f) {
 return f*f; // Orden del algoritmo
}

int main(int argc, char *argv[]) {
  fstream file;
 for (int i = 1; i <= argc; i++) {
   vector<double> vec;
   double fx, tx;
   file.open(argv[i]);
   while (file) {
     file >> fx >> tx;
     vec.push_back(tx / orden(fx));
   }
   cout << fixed << argv[i] <<" K: " << media_vector(vec) << endl;
   file.close();
 }
}
```

Burbuja: eficiencia híbrida



burbuja_arturo.dat K: 12148.971079, burbuja_manu.dat K: 19702.317007, burbuja_sara.dat K: 38313.102503, burbuja_sergio.dat K: 56162.473696

Pivotar: eficiencia teórica

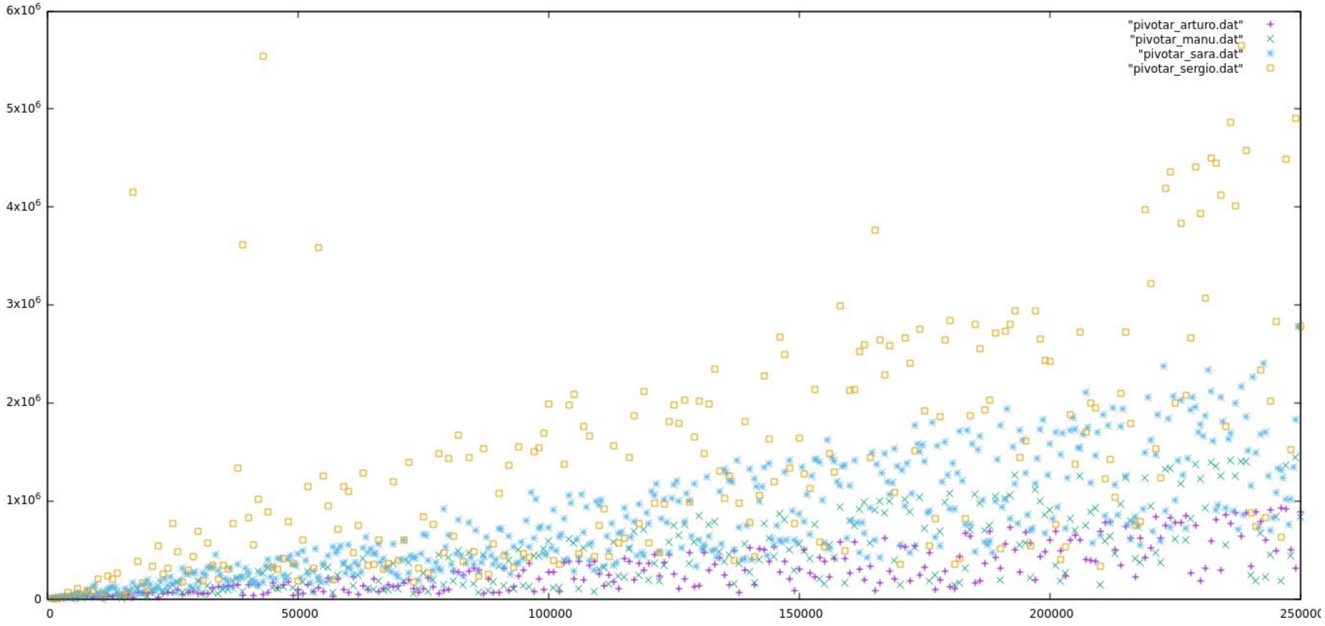
```
int pivotar(double *v, const int ini, const int fin){
 double pivote = v[ini], aux;
 int i = ini + 1, j = fin;
 while (i <= j) {   while (v[i] < pivote && i <= j) {

     i++;
   }
   while (v[j] >= pivote && j >= i) {
     j--;
   }
   if (i < j) {
     aux = v[i];
     v[i] = v[j];
     v[j] = aux;
   }
 }
 if (j > ini) {
   v[ini] = v[j];
   v[j] = pivote;
 }

 return j;
}
```

Estamos ante un algoritmo lineal ya que los bucles internos suman un recorrido del array y el externo finaliza cuando los internos han acabado el recorrido.

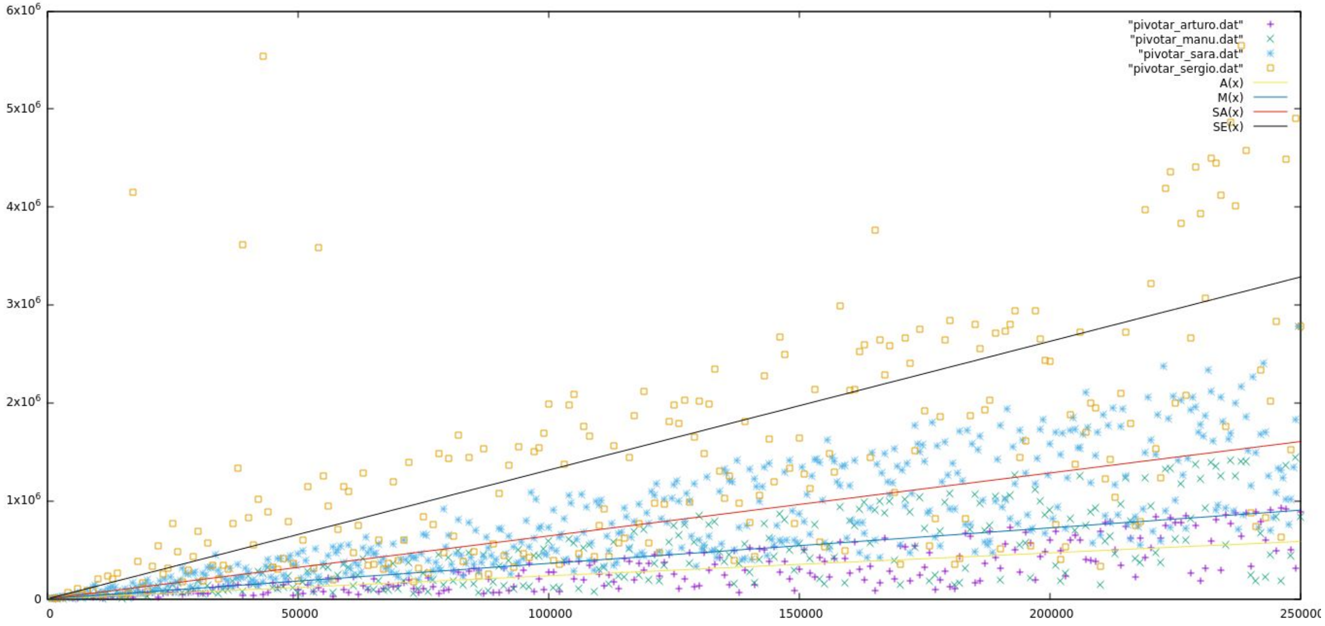
Peor caso: $O(n)$

Mejor caso: $\Omega(n)$

Pivotar: eficiencia empírica



Pivotar: eficiencia híbrida



pivotar_arturo.dat K: 2.345623, pivotar_manu.dat K: 3.621301, pivotar_sara.dat K: 6.426688, pivotar_sergio.dat K: 13.148911

Búsqueda: eficiencia teórica

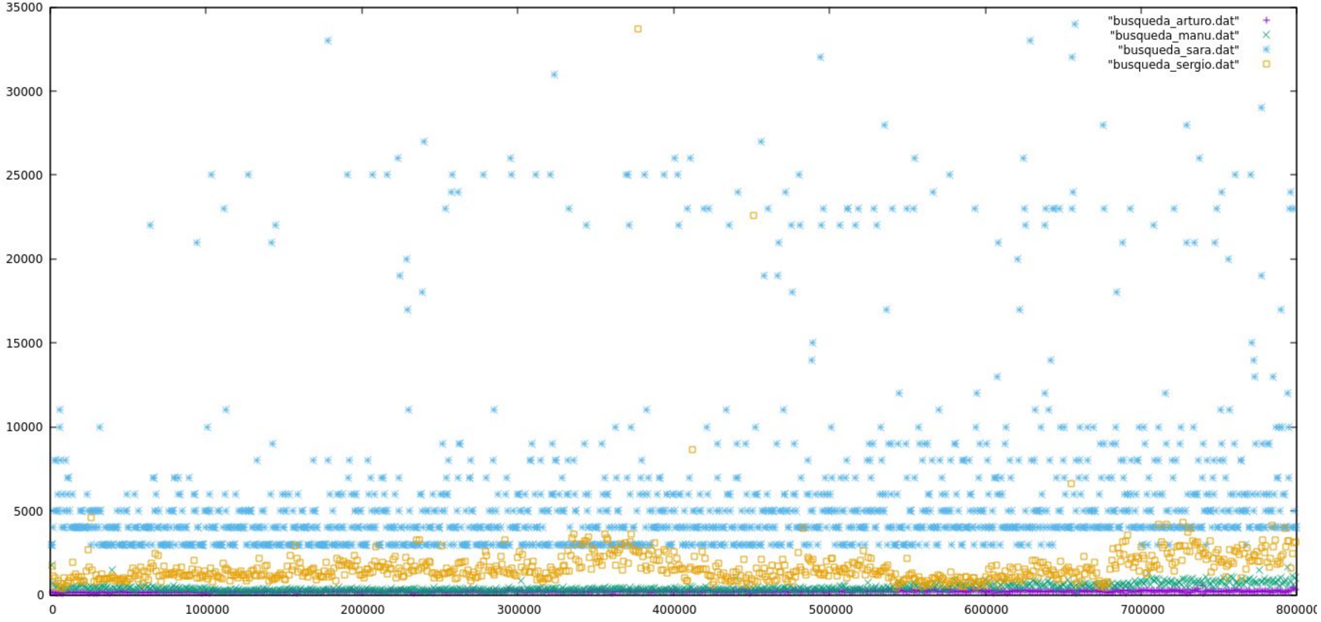
```
int Busqueda(int *v, int n, int elem) {
 int inicio, fin, centro;
 inicio = 0;
 fin = n - 1;
 centro = (inicio + fin) / 2;

 while ((inicio <= fin) && (v[centro] != elem))
{
   if (elem < v[centro])
     fin = centro - 1;
   else
     inicio = centro + 1;

   centro = (inicio + fin) / 2;
 }

 if (inicio > fin)
   return -1;

 return centro;
}
```

Para un vector de enteros ordenados del 1 al 16 el máximo número de divisiones que hará será 4 cuando busque el 16. Para un vector del 1 al 32 serán 5 divisiones y así sucesivamente. Por tanto tenemos que para un vector de tamaño 2^i necesitará como máximo i divisiones. Así que $f(2^i)=i \rightarrow f(i)=\log_2(i)$.

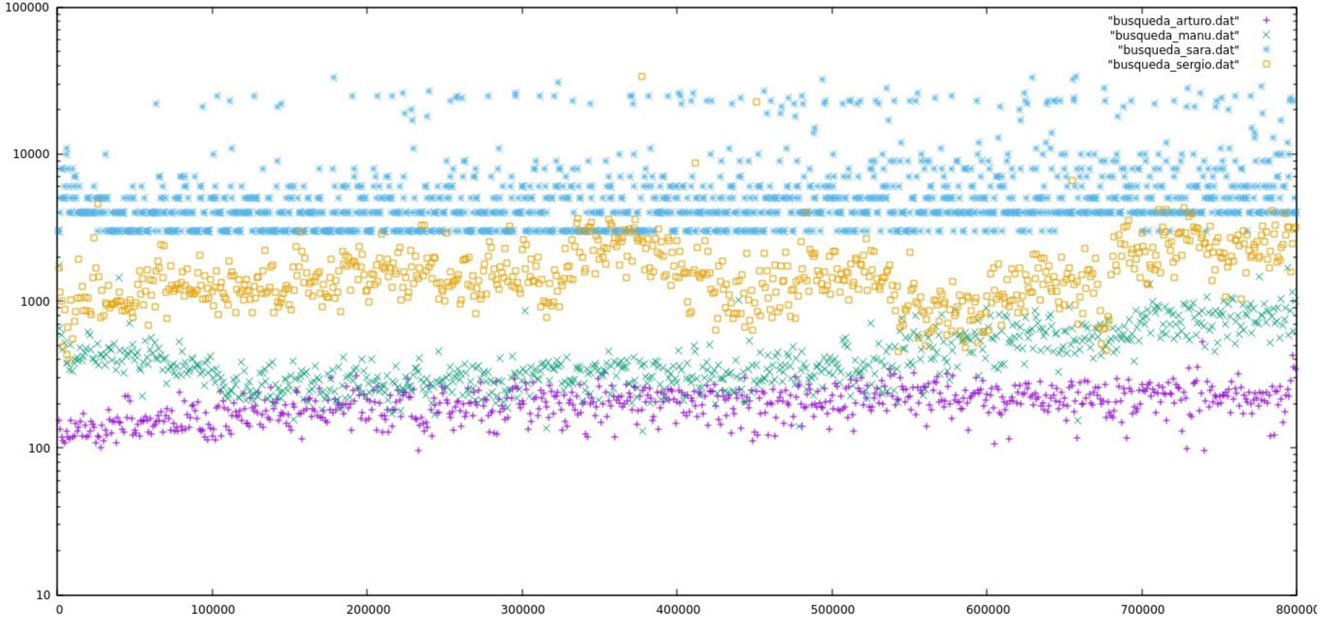
En consecuencia el algoritmo es logarítmico, por tanto, de orden $O(\log(n))$.

El caso mejor para este algoritmo es que el elemento a buscar esté en el centro, por tanto es de orden $\Omega(1)$.

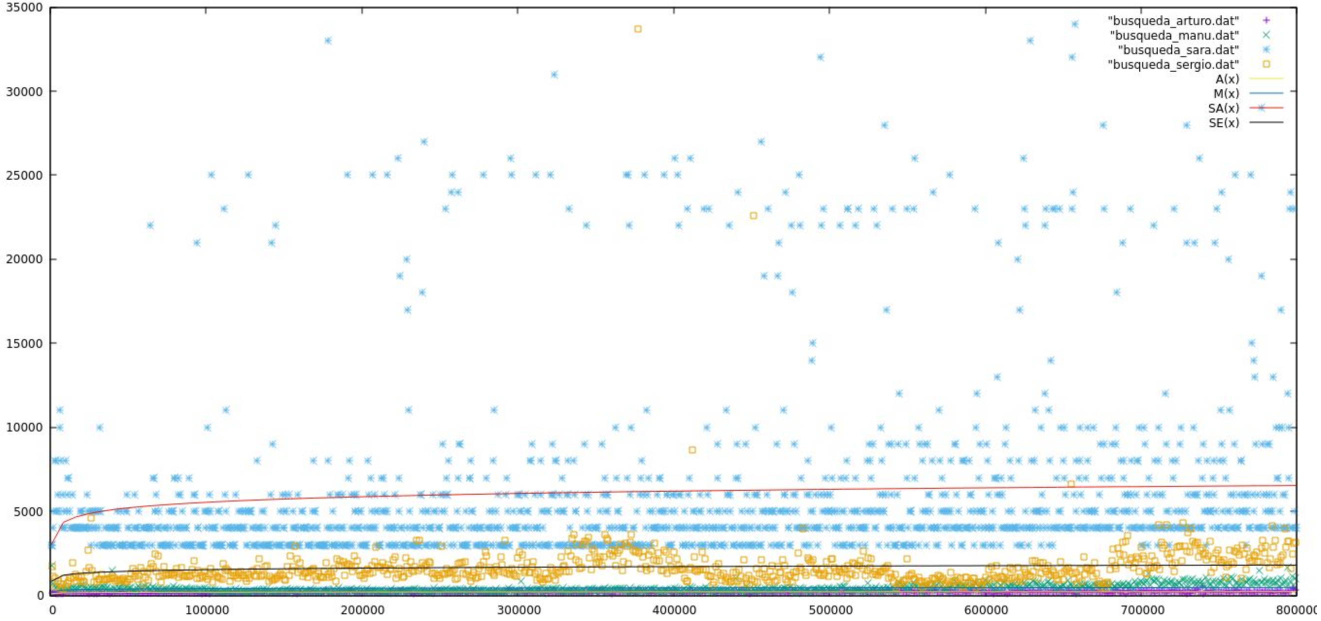
Búsqueda: eficiencia empírica



Búsqueda: eficiencia empírica en escala logarítmica

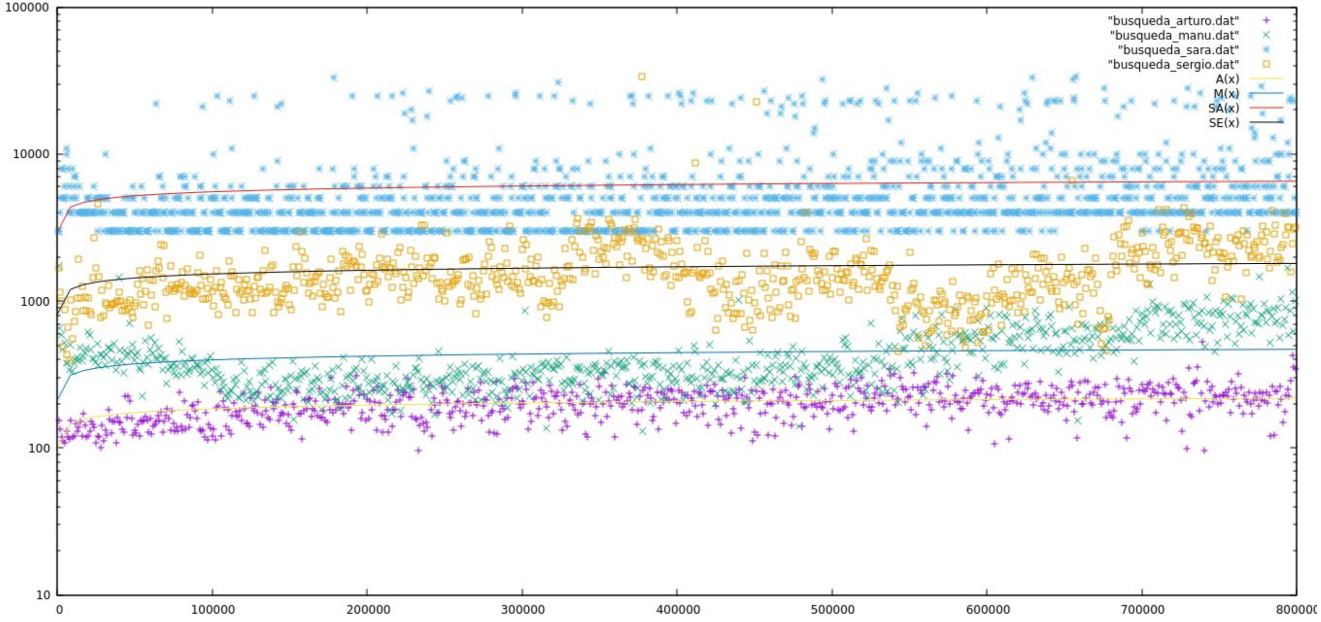


Búsqueda: eficiencia híbrida



busqueda_arturo.dat K: 16.032304, busqueda_manu.dat K: 34.512091, busqueda_sara.dat K: 480.698753, busqueda_sergio.dat K: 132.538248

Búsqueda: eficiencia híbrida en escala logarítmica



busqueda_arturo.dat K: 16.032304, busqueda_manu.dat K: 34.512091, busqueda_sara.dat K: 480.698753, busqueda_sergio.dat K: 132.538248

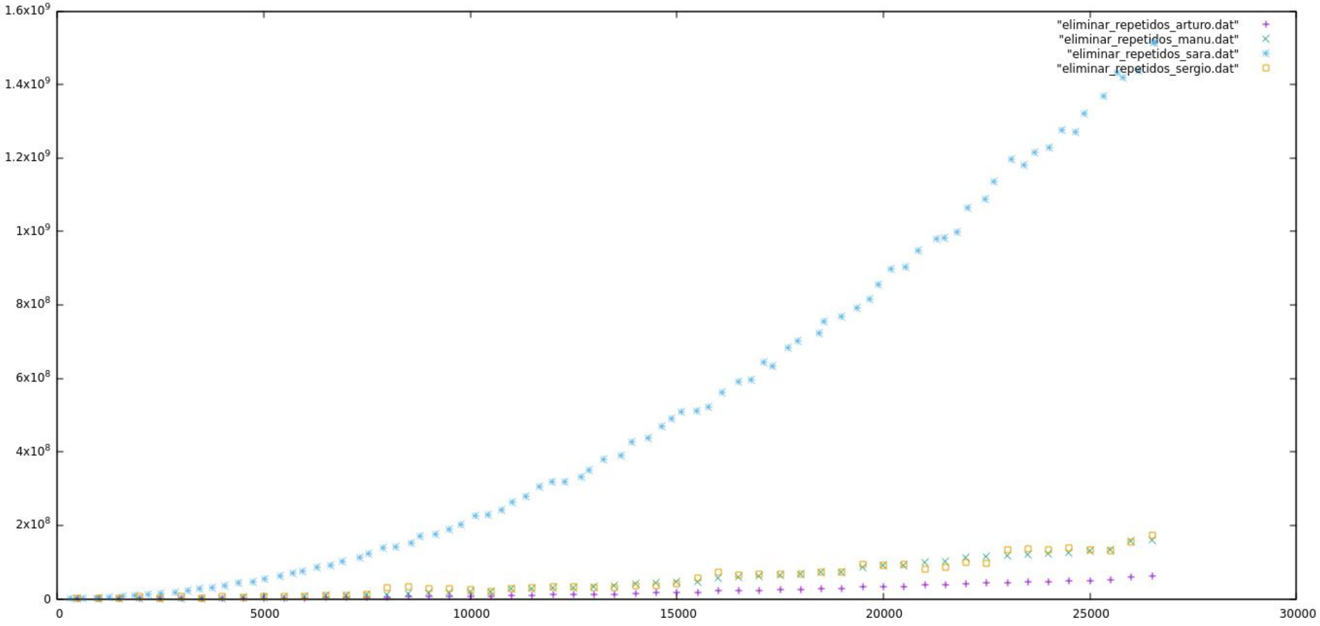
Eliminar Repetidos: eficiencia teórica

```
1   void EliminaRepetidos(double original[], int &nOriginal){
2    int i, j, k;
3    for (i = 0; i < nOriginal; i++) {
4      j = i + 1;
5      do {
6        if (original[j] == original[i]) {
7          for (k = j + 1; k < nOriginal; k++)
8            original[k - 1] = original[k];
9
10         nOriginal--;
11       } else
12         j++;
13     } while (j < nOriginal);
14   }
15  }
```

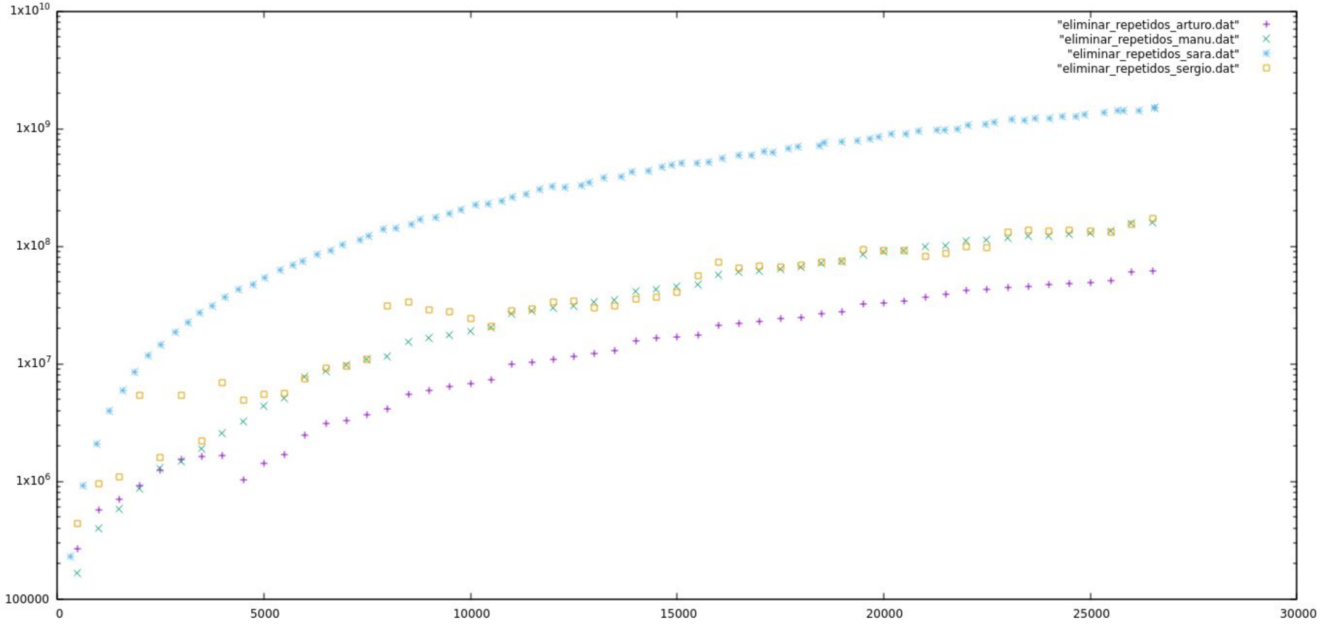
Caso peor: Se entra en el if de la línea 6 tantas veces como elementos del vector hay - 1 , por lo que nOriginal acaba siendo 1. Así que solo se ejecutan dos de los 3 bucles, en consecuencia es de orden $O(n^2)$

Caso mejor:
Nunca entra en el if así que solo se ejecutan dos de los tres bucles, en consecuencia es de orden $\Omega(n^2)$

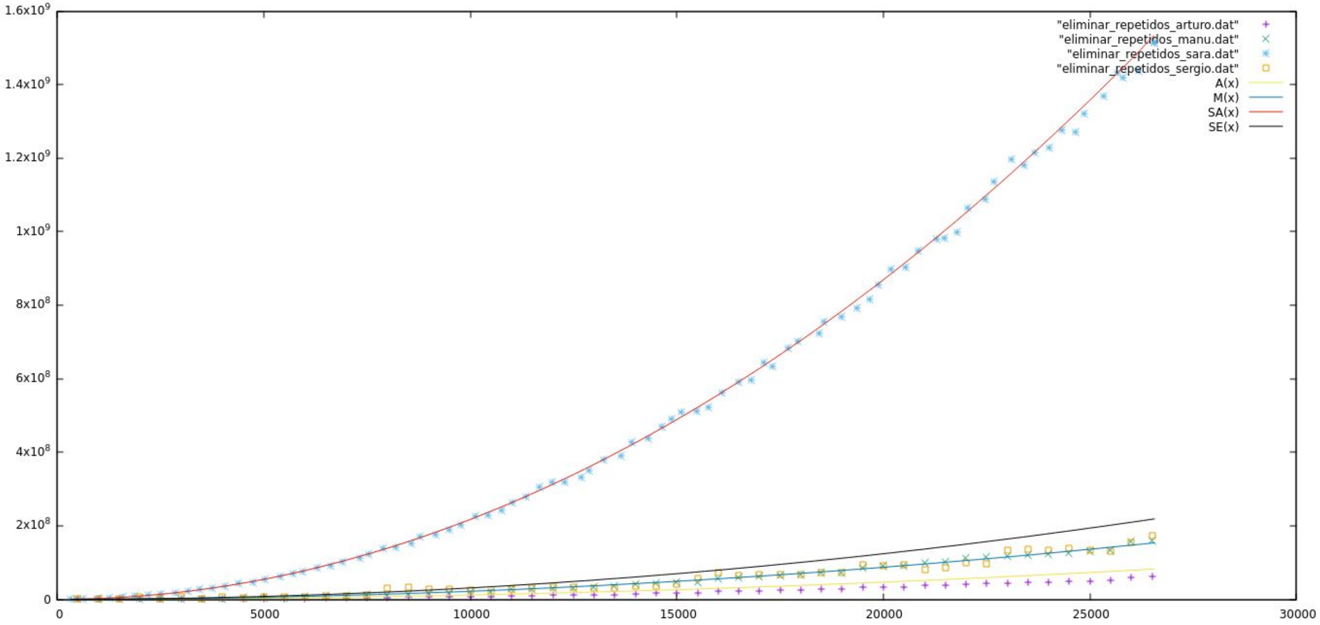
Eliminar Repetidos: eficiencia empírica



Eliminar Repetidos: eficiencia empírica en escala logarítmica

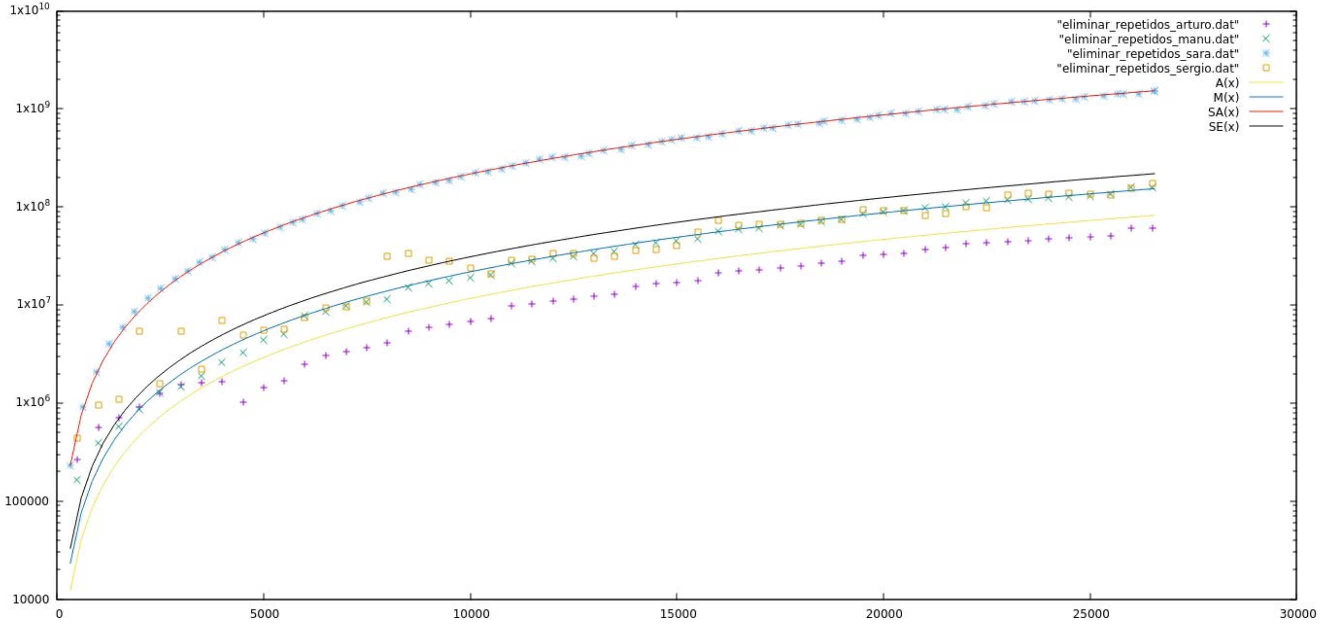


Eliminar Repetidos: eficiencia híbrida



eliminar_repetidos_arturo.dat K:0.116618, eliminar_repetidos_manu.dat K:0.217906, eliminar_repetidos_sara.dat K:2.172988, eliminar_repetidos_sergio.dat K:0.309992

Eliminar Repetidos: eficiencia híbrida en escala logarítmica



eliminar_repetidos_arturo.dat K:0.116618, eliminar_repetidos_manu.dat K:0.217906, eliminar_repetidos_sara.dat K:2.172988, eliminar_repetidos_sergio.dat K:0.309992

Busqueda Binaria Recursiva: eficiencia teórica

```
int BuscarBinario(double *v, const int ini, const int fin, const double x) {
 int centro;                                //O(1)
 if (ini > fin)                             //O(1)
   return -1;

 centro = (ini + fin) / 2;                  //O(log(n))
 if (v[centro] == x)                        //O(1)
   return centro;
 if (v[centro] > x)
   return BuscarBinario(v, ini, centro - 1, x); //O(log(n))
 return BuscarBinario(v, centro + 1, fin, x); //O(log(n))
}
```

Busqueda Binaria Recursiva: eficiencia teórica

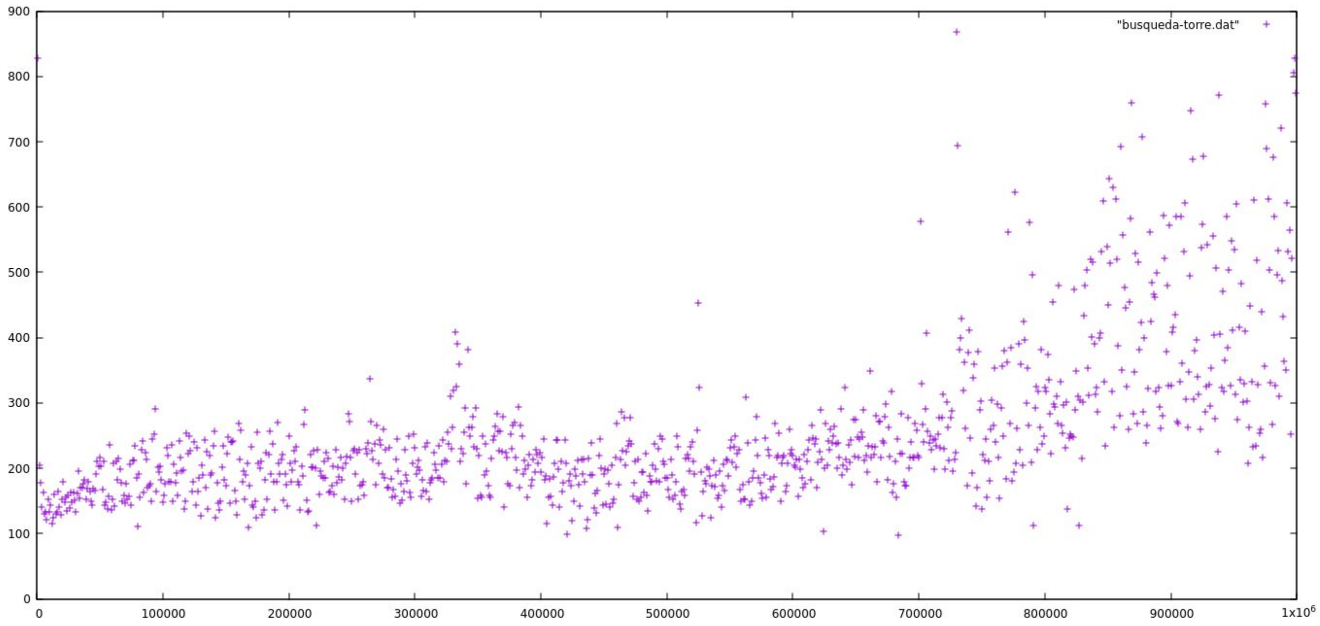
Para el caso peor extraemos la ecuación de la recurrencia:

$$\mathrm{T(n)} \rightarrow \mathrm{T(n/2)} \rightarrow \mathrm{T(n/4)} \rightarrow \mathrm{T(n/8)} \rightarrow T(n/2^{i})$$
$$T(n/2^{log(n)} + log_2(n)) \rightarrow T(1) + log_2(n)$$

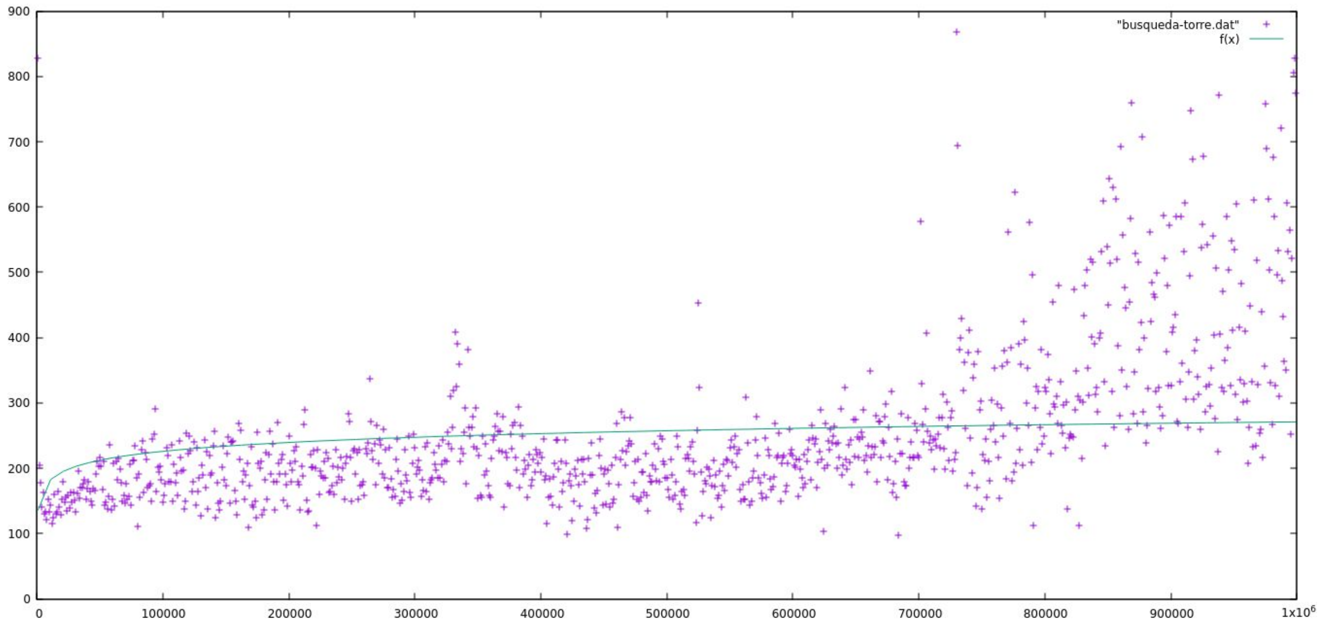
Vemos que es de orden O(log(n))

El caso mejor se da cuando el elemento buscado está en el centro del array y lo devuelve inmediatamente, por tanto es de orden $\Omega(1)$.

Busqueda Binaria Recursiva: eficiencia empírica



Busqueda Binaria Recursiva: eficiencia híbrida



K: 19.617718

Reajustar: eficiencia teórica

```
1   void reajustar(int T[], int num_elem, int k) {
2    int j;
3    int v;
4    v = T[k];
5    bool esAPO = false;
6    while ((k < num_elem / 2) && !esAPO) {
7      j = k + k + 1;
8      if ((j < (num_elem - 1)) && (T[j] < T[j + 1]))
9        j++;
10     if (v >= T[j])
11       esAPO = true;
12     T[k] = T[j];
13     k = j;
14   }
15   T[k] = v;
16  }
```

Caso peor:
La función le da estructura de árbol binario al array, por tanto es de orden O(log (n))

Caso mejor:
Al empezar entra en el if de la línea 10 y el bucle acaba. Por tanto es $\Omega(1)$.

Heapsort: eficiencia teórica

```
1   void Heapsort(int T[], int num_elem) {
2    int i;
3    for (i = num_elem / 2; i >= 0; i--)
4      reajustar(T, num_elem, i);
5    for (i = num_elem - 1; i >= 1; i--) {
6      int aux = T[0];
7      T[0] = T[i];
8      T[i] = aux;
9      reajustar(T, i, 0);
10   }
11  }
```

La eficiencia del for de la sentencia 3 del bloque es de

$O((n/2)\log(n))$

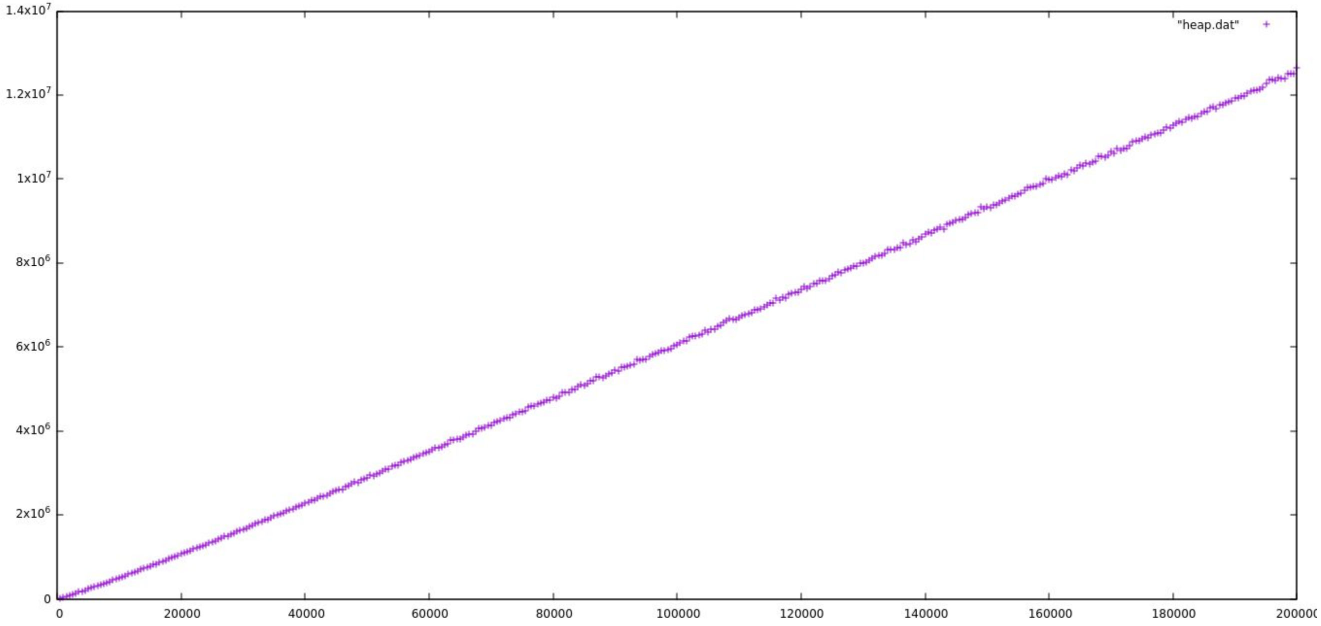
y la del for de la sentencia 5 es de

$O(n-1)\log(n))=O(n\log(n))$

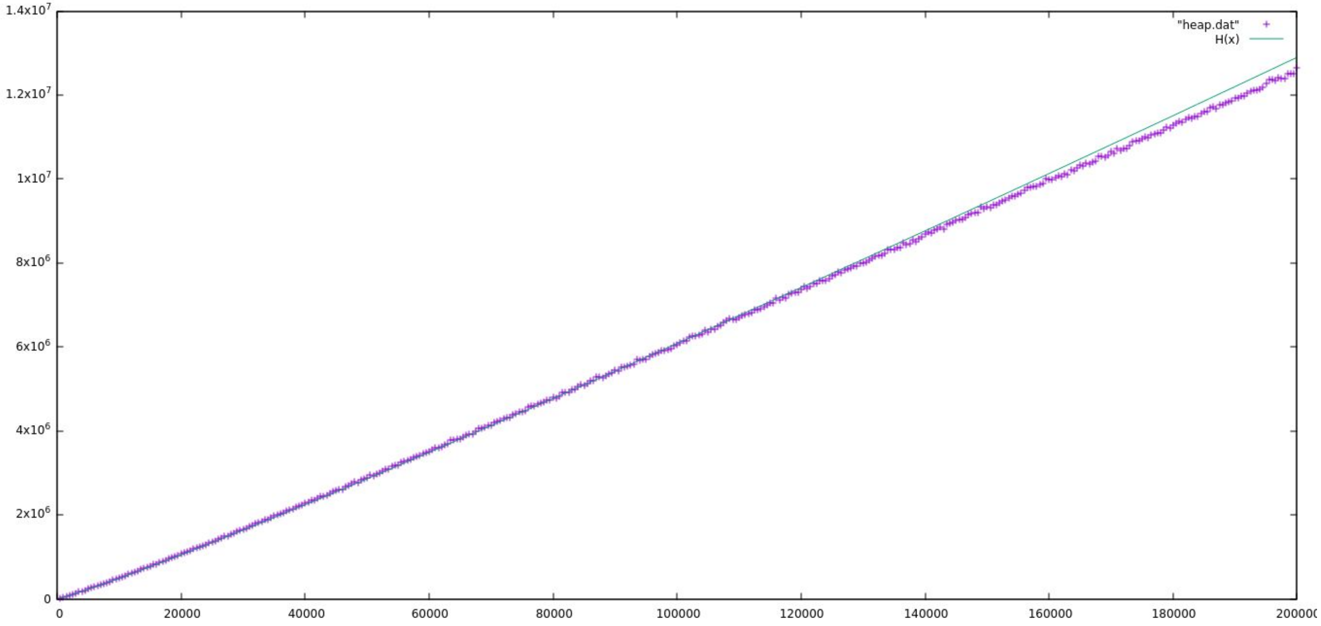
Máximo de los órdenes de eficiencia:

$O(n \log(n))$

Heapsort: eficiencia empírica



Heapsort: eficiencia híbrida



K: 5.283937

Mergesort: eficiencia teórica

```
static void mergesort_lims(int T[], int inicial, int final) {
  if (final - inicial < UMBRAL_MS) {
   insercion_lims(T, inicial, final);
 } else {
   int k = (final - inicial) / 2;
   int *U = new int[k - inicial + 1];
   assert(U);
   int l, l2;
   for (l = 0, l2 = inicial; l < k; l++, l2++)
     U[l] = T[l2];
   U[l] = INT_MAX;
   int *V = new int[final - k + 1];
   assert(V);
   for (l = 0, l2 = k; l < final - k; l++, l2++)
     V[l] = T[l2];
   V[l] = INT_MAX;
   mergesort_lims(U, 0, k);
   mergesort_lims(V, 0, final - k);
   fusion(T, inicial, final, U, V);
   delete[] U;
   delete[] V;
 };
}
```

```
static void fusion(int T[], int inicial, int final, int U[], int V[]) {
 int j = 0;
 int k = 0;
 for (int i = inicial; i < final; i++){
   if (U[j] < V[k]) {
     T[i] = U[j];
     j++;
   } else {
     T[i] = V[k];
     k++;
   };
 };
}
```

Mergesort: eficiencia teórica

$$T(n) \begin{cases} 1, & n=1; \\ 2T(n/2) + n & n>1 \end{cases}$$

$$t_i = T(2^i)$$

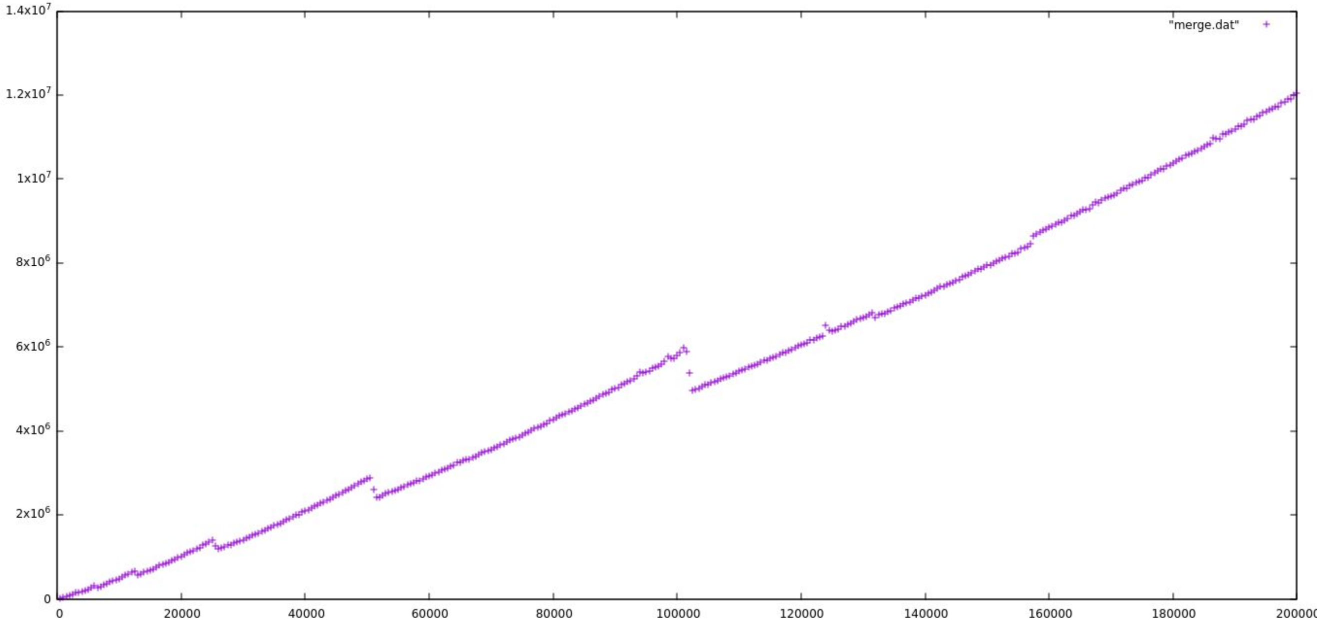
$$t_i = 2t_{i-1} + 2^i \rightarrow c_1 2^i + c_2 i\ 2^i$$

$$T(n) = c_1 2^{log_2(n)} + c_2 log_2(n) 2^{log_2(n)} = c_1 n + c_2 n log_2(n)$$

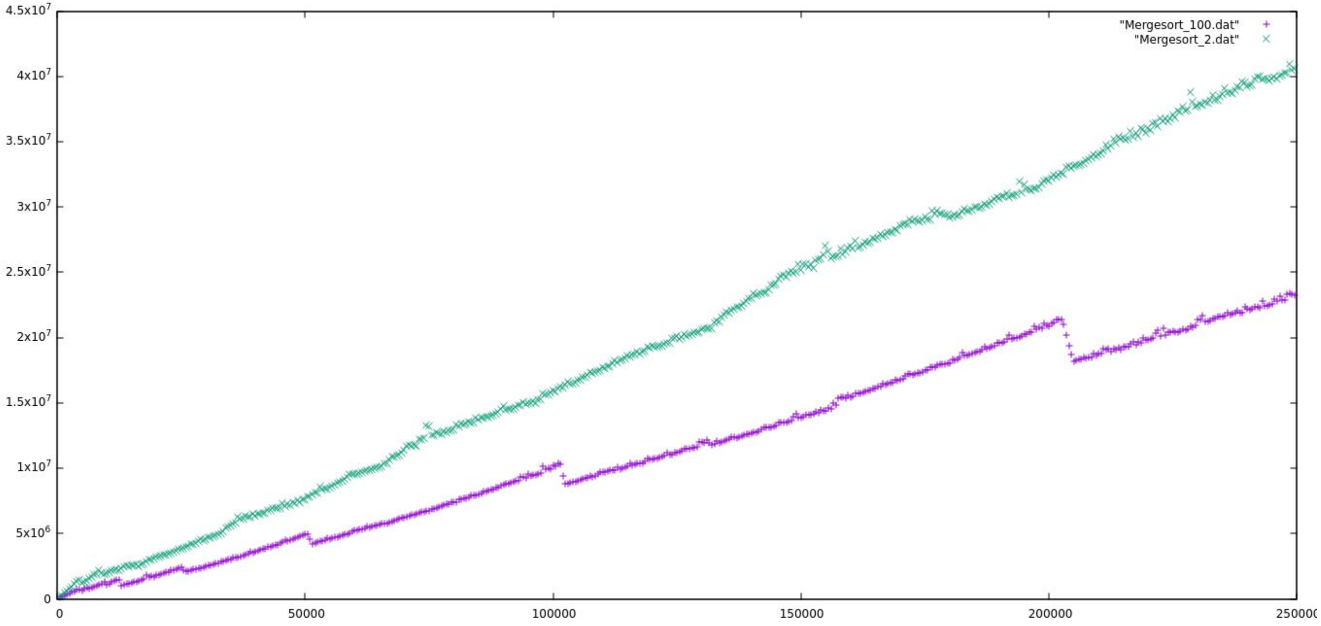
De aquí podemos concluir que T(n) es O(n log(n))

El mejor caso es Ω(n log(n)) porque sigue teniendo que hacer las mismas llamadas recursivas

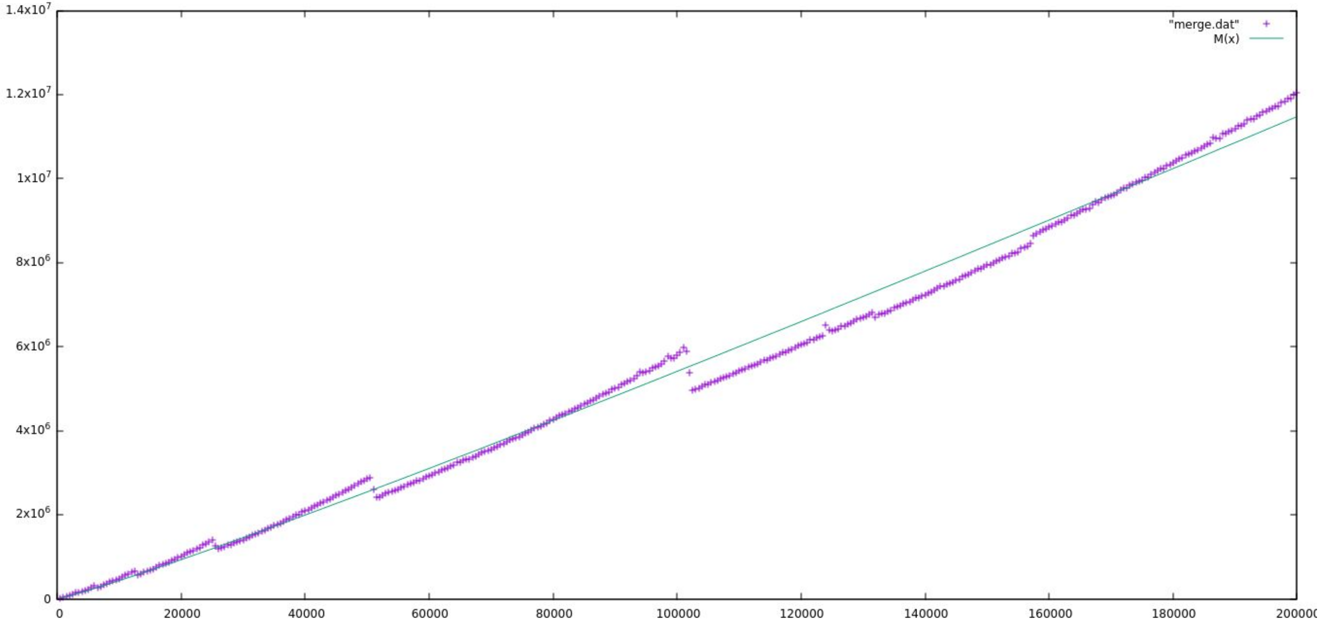
Mergesort: eficiencia empírica



Mergesort: comparación de umbrales



Mergesort: eficiencia híbrida



K: 4.701676

Hanoi: eficiencia teórica

$$T(n)\ 0 \quad \begin{cases} n=0 \\ 2T(n-1)+1 & \text{otro caso} \end{cases}$$

$$tn - 2tn - 1 = 1$$

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

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

Ecuación general:

$$tn = c_1 + 2^n + c_2 1^n \rightarrow T(n) = c_1 2^n + c_2$$

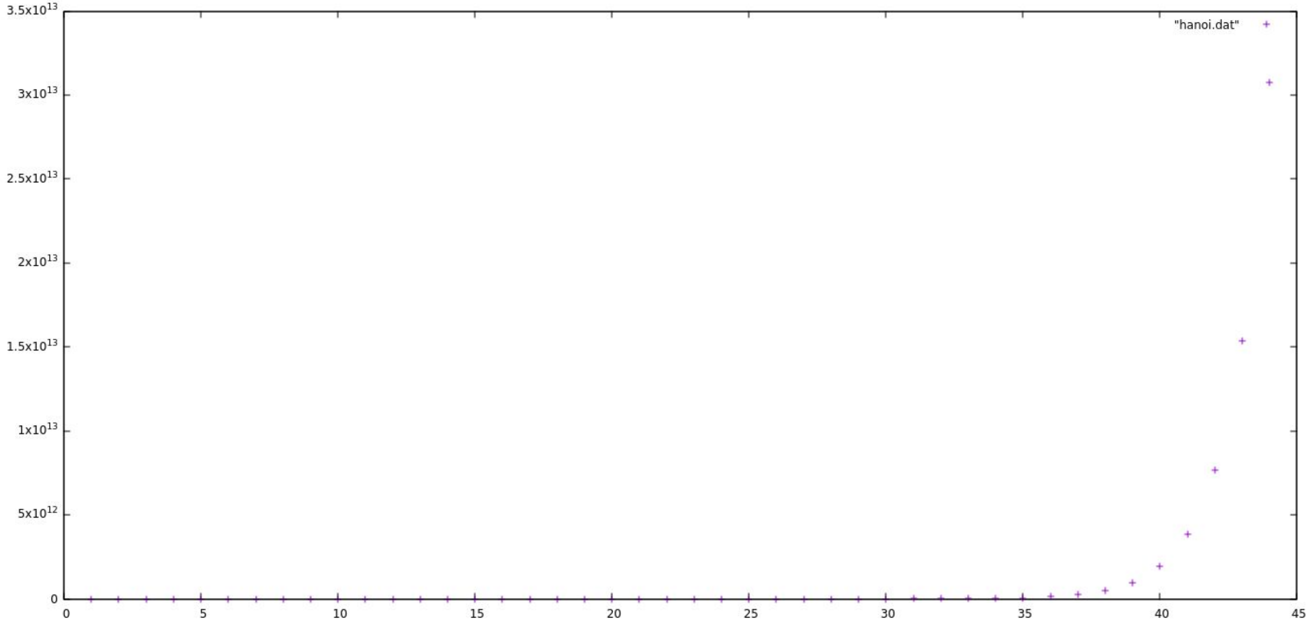
$$T(0) = 0 \quad \rightarrow \quad c_1 + c_2 = 0 \quad \rightarrow \quad c_1 = 1 \quad \rightarrow \quad T(n) = 2^n - 1$$

$$T(1) = 1 \quad \rightarrow \quad 2c_1 + c_2 = 1 \quad \rightarrow \quad c_2 = -1 \quad \text{(sustituyendo)}$$

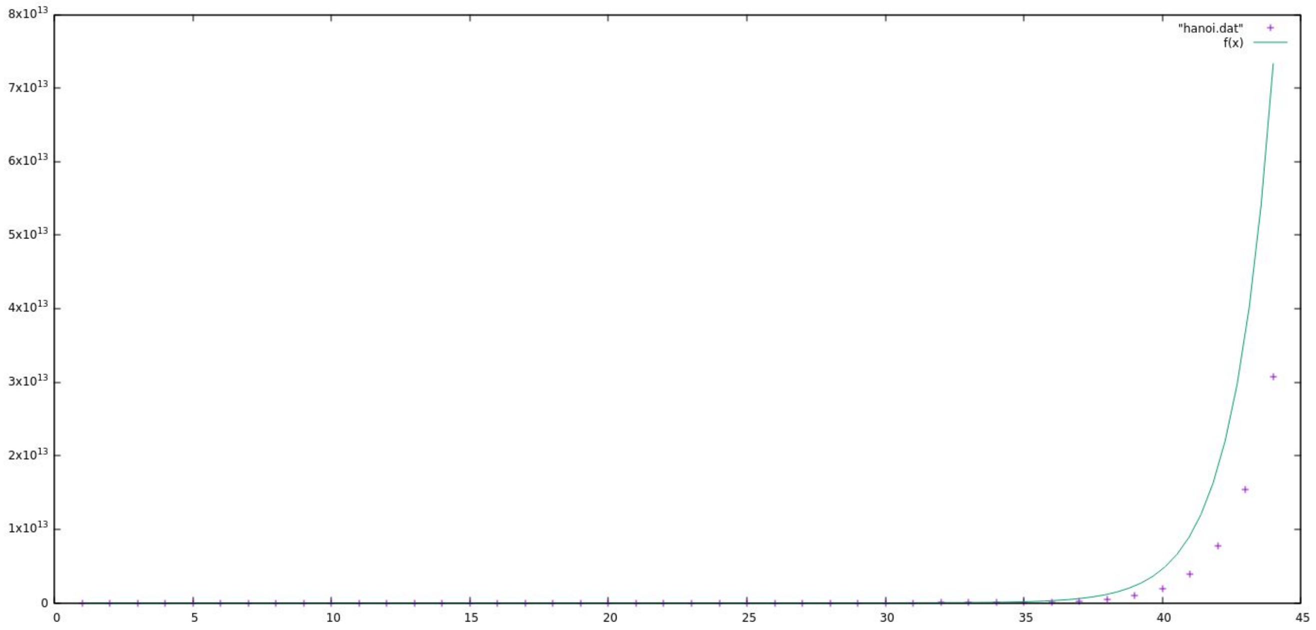
Por tanto la eficiencia del algoritmo es de $O(2^n - 1)$ que es igual a $O(2^n)$.

Y como la única variación posible es el tamaño del problema, entonces $\Omega(2^n)$

Hanoi: eficiencia empírica

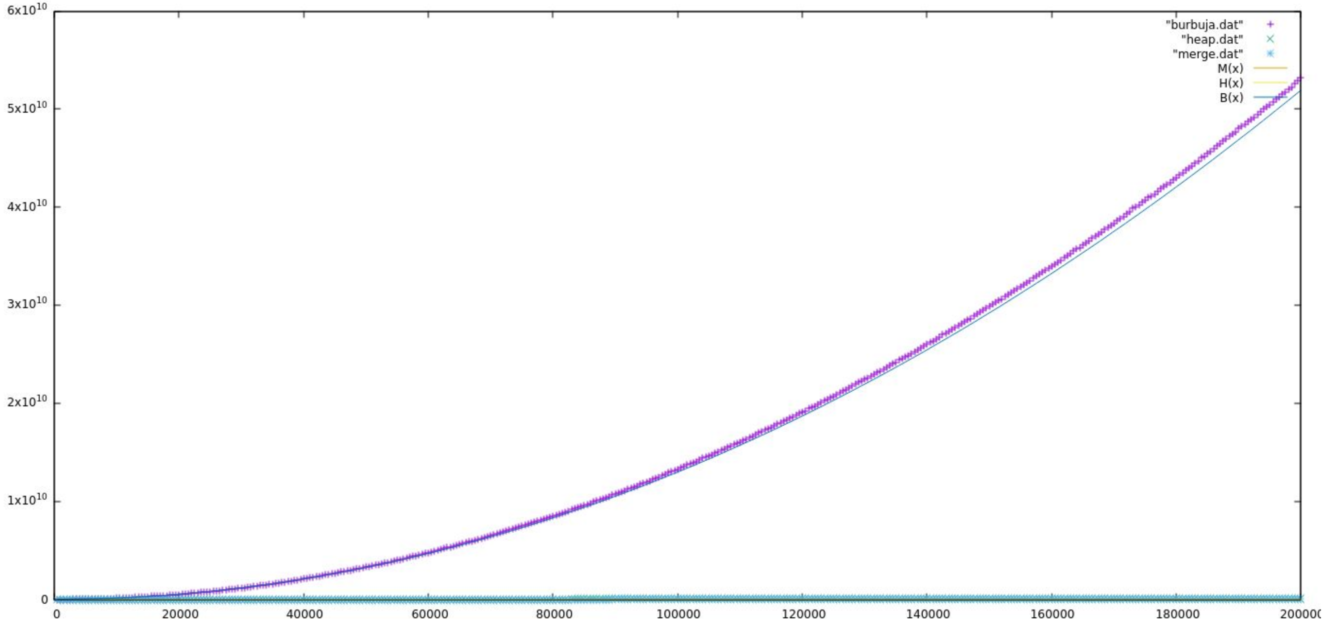


Hanoi: eficiencia híbrida



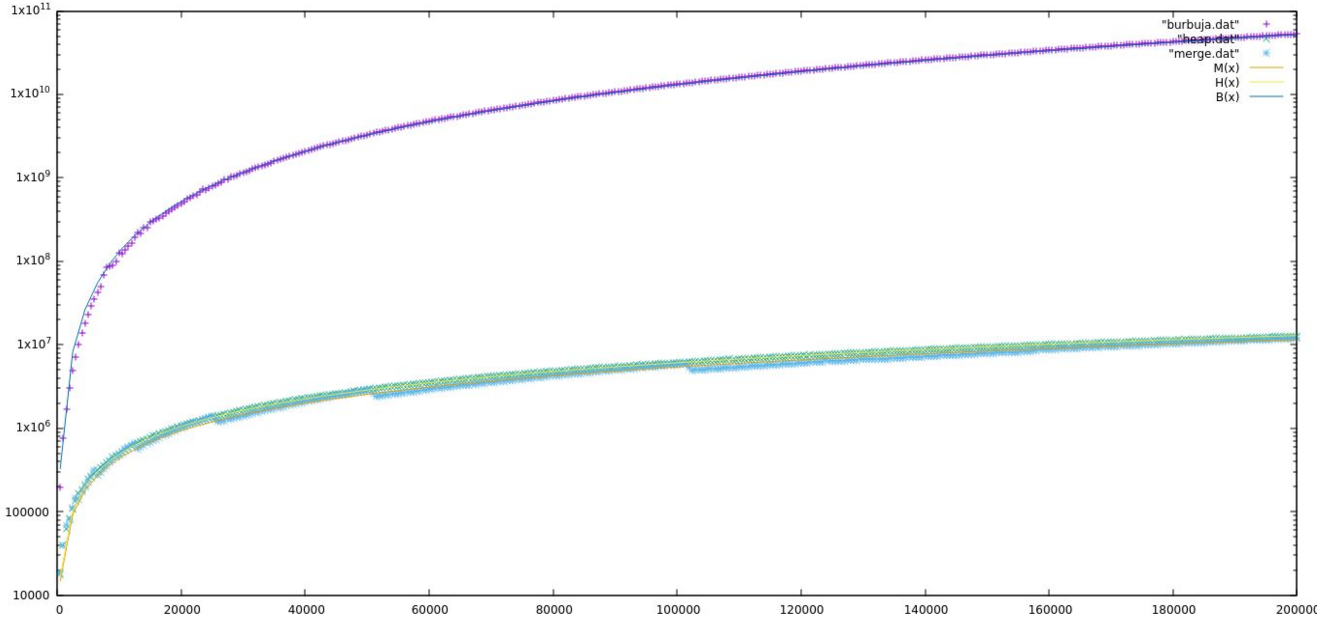
K: 4.168112

Comparación de Algoritmos de búsqueda



mergesort.dat K: 4.701676, heapsort.dat K: 5.283937, burbuja.dat K: 1.298619

Comparación de Algoritmos de búsqueda en escala logarítmica



mergesort.dat K: 4.701676, heapsort.dat K: 5.283937, burbuja.dat K: 1.298619

Comparación de Distintas Flags de Optimización: Flags usadas

-O0: Sin optimizaciones.

-O1: Optimizaciones que no incrementan demasiado el tiempo de compilación.

-O2: Todas las optimizaciones que no intercambian espacio por velocidad.

-O3: Todas las optimizaciones, incluyendo aquellas que intercambian espacio por velocidad.

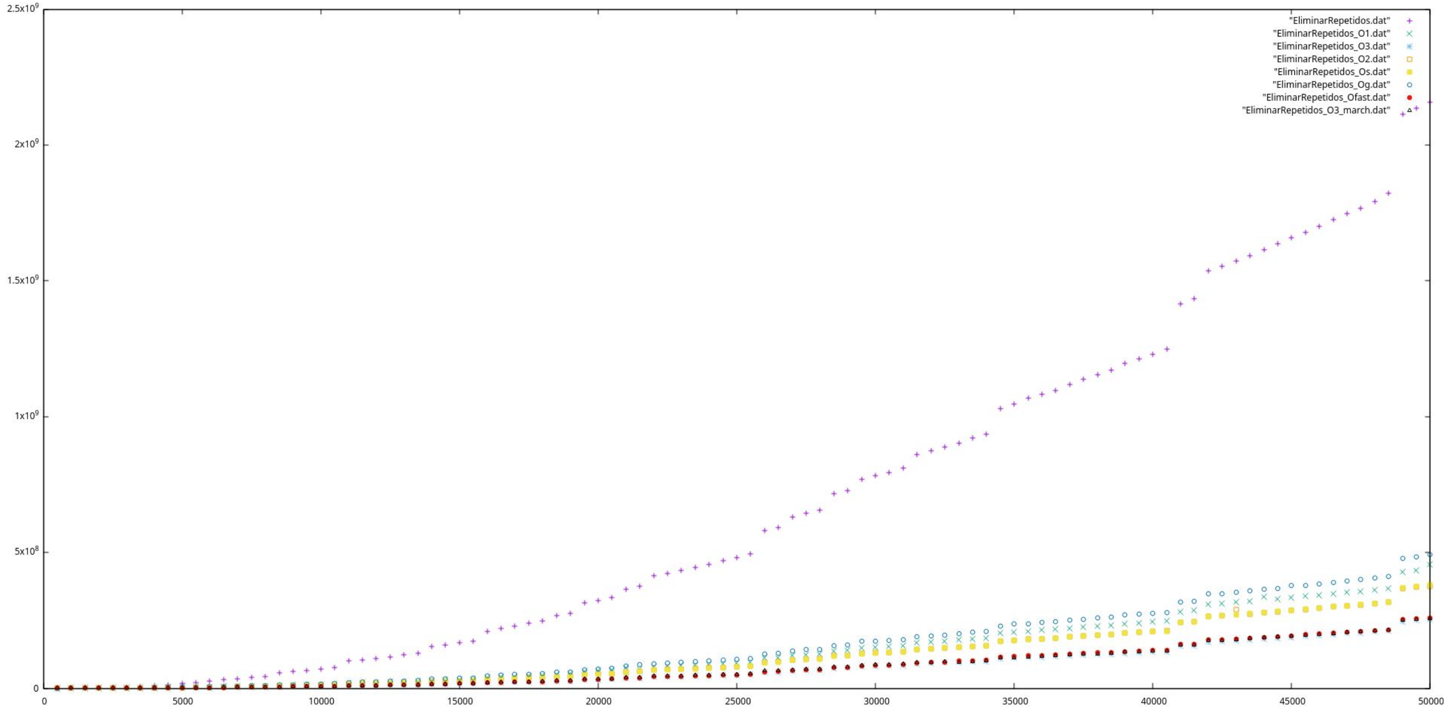
-Os: Optimiza buscando reducir el tamaño del ejecutable.

-Og: Optimiza buscando facilitar la depuración del programa.

-Ofast: Todas las optimizaciones de -O3 más algunas que ignoran ciertas normas de compilación.

-O3 -march=haswell: Además de -O3 el código generado está optimizado para procesadores Intel Core de cuarta generación.

Comparación de Distintas Flags de Optimización: Eliminar Repetidos



Comparación de Distintas Flags de Optimización: Mergesort