ALGORÍTMICA

Práctica 1: Análisis de eficiencia de algoritmos

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

Índice

Modelo ordenador Sara: Intel(R) Core(TM) i7-6700HQ CPU @ 2.60GHz
Compilador: gcc (Debian 8.1.0-12) 8.1.0
Flags de compilación: ninguna

Modelo ordenador Manu: Intel(R) Core(TM) i5-4210U CPU @ 1.70GHz
Compilador: gcc (Ubuntu 4.8.4-2ubuntu1~14.04.4) 4.8.4
Flags de compilación: -O2

Modelo ordenador Arturo: Intel(R) Core(TM) i7-4790K CPU @ 4.00GHz
Compilador: gcc (GCC) 8.2.1 20181127
Flags de compilación: -O3

Modelo ordenador Sergio: Intel(R) Core(TM) i7-6700HQ CPU @ 2.60GHz en VMware
Compilador: gcc (Ubuntu 7.3.0-27ubuntu1~18.04) 7.3.0
Flags de compilación: -O3

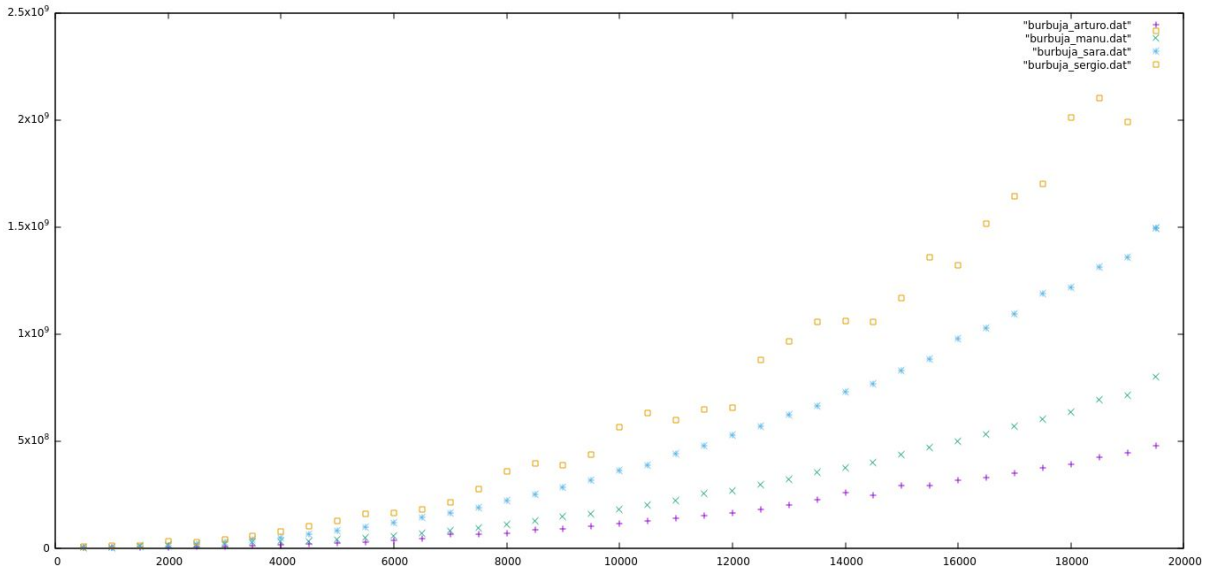
1. Burbuja

Teórica:

El algoritmo OrdenaBurbuja se encarga de ordenar un vector. Para ello, existe un bucle while que funcionará hasta que ya no haya más cambios, es decir, cuando el vector esté ordenado. Dentro del while, recorre el vector con un bucle for en orden descendente y compara el elemento anterior con el actual, si es mayor, hace un cambio de variable, además pone a verdadero la variable que regula el bucle while exterior. Por tanto en caso de tener un array ya ordenado solo se ejecutará el bucle for interno y estaremos ante un algoritmo de orden $\Omega(n)$, pero en cualquier otro caso el bucle for interno se ejecutará tantas veces como elementos mal posicionados haya en el array, por tanto el algoritmo será de orden $O(n^2)$.

```
1   void OrdenaBurbuja(int *v, int n) {
2    int i, j, aux;
3    bool haycambios = true;
4
5    i = 0;
6    while (haycambios) {
7      haycambios = false;
8      for (j = n - 1; j > i; j--) {   //O(n-1)
9        if (v[j - 1] > v[j]) {
10         aux = v[j];
11         v[j] = v[j - 1];
12         v[j - 1] = aux;
13         haycambios = true;
14       }
15     }
16   }
17  }
```

Empírica:

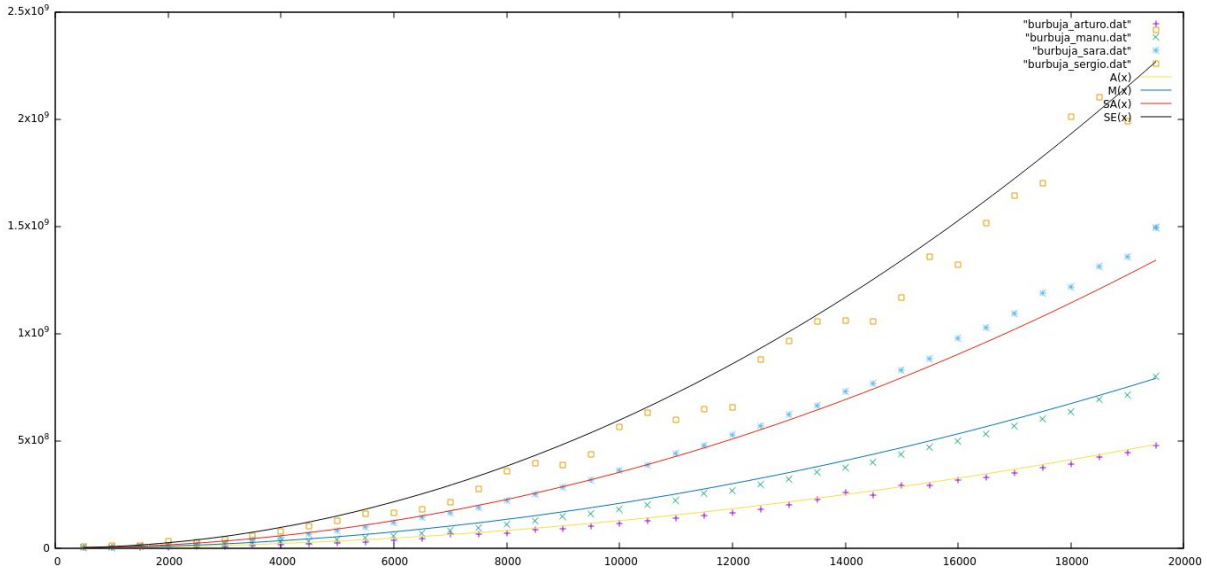


Híbrida:

Para calcular la constante oculta y poder hacer la eficiencia híbrida hemos diseñado el siguiente programa, el cual recibe como argumento los ficheros con los tiempos y devuelve por pantalla la constante K de cada fichero.

```
1   #include <cmath>
2   #include <fstream>
3   #include <iostream>
4   #include <vector>
5
6   using namespace std;
7
8   template <class T> double media_vector(vector<T> v) {
9    double k = 0;
10   for (auto i : v)
11     k += i;
12   return k / v.size();
13  }
14
15  double orden(double f) {
16   return f*f; // Orden del algoritmo
17  }
18
```

```
19  int main(int argc, char *argv[]) {
20    fstream file;
21   for (int i = 1; i <= argc; i++) {
22     vector<double> vec;
23     double fx, tx;
24     file.open(argv[i]);
25     while (file) {
26       file >> fx >> tx;
27       vec.push_back(tx / orden(fx));
28     }
29      cout << fixed << argv[i] <<" K: " << media_vector(vec) <<
    endl;
30     file.close();
31   }
32  }
```



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

2. Pivotar

Teórica:

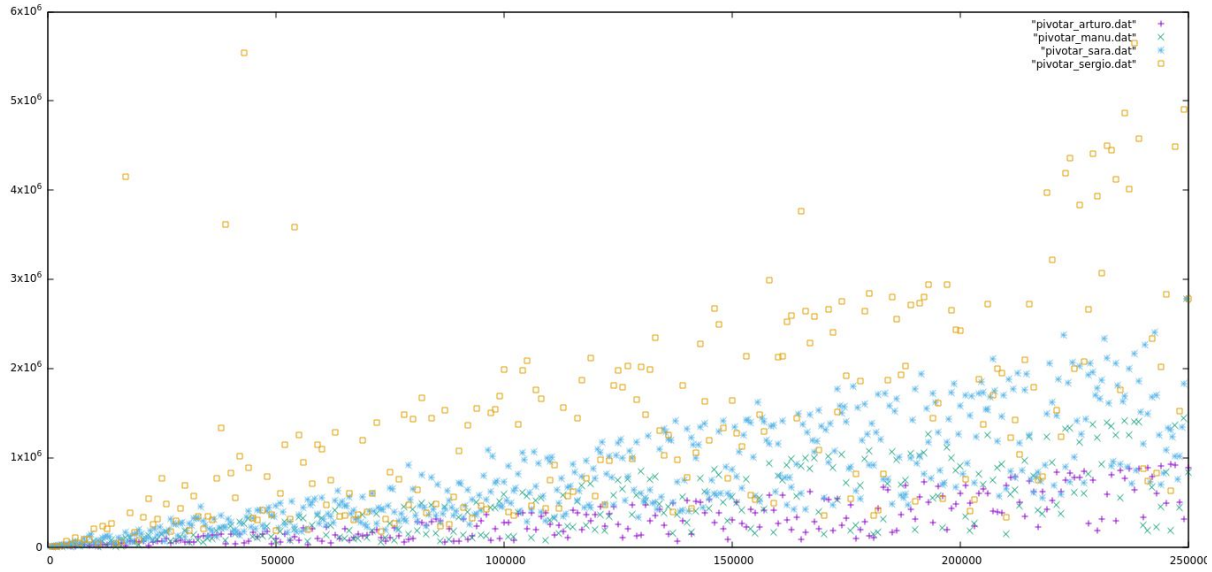
```
1   int pivotar(double *v, const int ini, const int fin) {
2    double pivote = v[ini], aux;
3    int i = ini + 1, j = fin;
4    while (i <= j) {
5      while (v[i] < pivote && i <= j) {
6        i++;
7      }
8      while (v[j] >= pivote && j >= i) {
9        j--;
10     }
11     if (i < j) {
12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

Si analizamos el algoritmo vemos que recibe un array y dos enteros, uno que contiene la posición de inicio del array, y otro con la posición final. Como podemos observar en el trozo de código la mayor parte del tiempo de ejecución se emplea en el cuerpo del bucle while. Podemos observar que conforme avanzan los bucles interiores, “reducen” la ejecución del bucle exterior. Esto se debe a lo siguiente: El bucle de la línea 5 va aumentando i mientras se cumpla que los elementos del array por los que itera sean inferiores al pivote. El bucle de la línea 6 hace algo similar, va decrementando j mientras los elementos sobre los que itere sean superiores o iguales al pivote. Ambos bucles paran si en algún momento i es superior a j.

Hasta aquí vemos que estos bucles simplemente verifican una condición del array, que del pivote a la izquierda todos los elementos sean inferiores al pivote,

y del pivote a la derecha todos los elementos sean superiores o iguales. Y como sus índices no se pueden “cruzar”, el primer y el segundo bucle en conjunto solo realizan un recorrido al array, no dos como cabría esperar en un primer vistazo. Cuando los bucles de las líneas 5 y 6 finalizan, vemos que se realiza un intercambio entre v[i] y v[j] siempre que i sea inferior a j, de nuevo esto son operaciones elementales.
Pasamos a analizar ahora el bucle externo de la línea 4. Este bucle depende de que i sea menor o igual que j, y como hemos visto los bucles internos aumentan y decrementan i y j respectivamente. Por lo que este bucle simplemente se encarga de realizar el intercambio entre v[i] y v[j] cuando las condiciones de los bucles internos fallan. Y cuando este bucle acaba significa que ambos bucles internos han recorrido el array una sola vez.
Por todo esto podemos concluir que 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. Por tanto estamos ante una eficiencia O(n) y Ω(n).

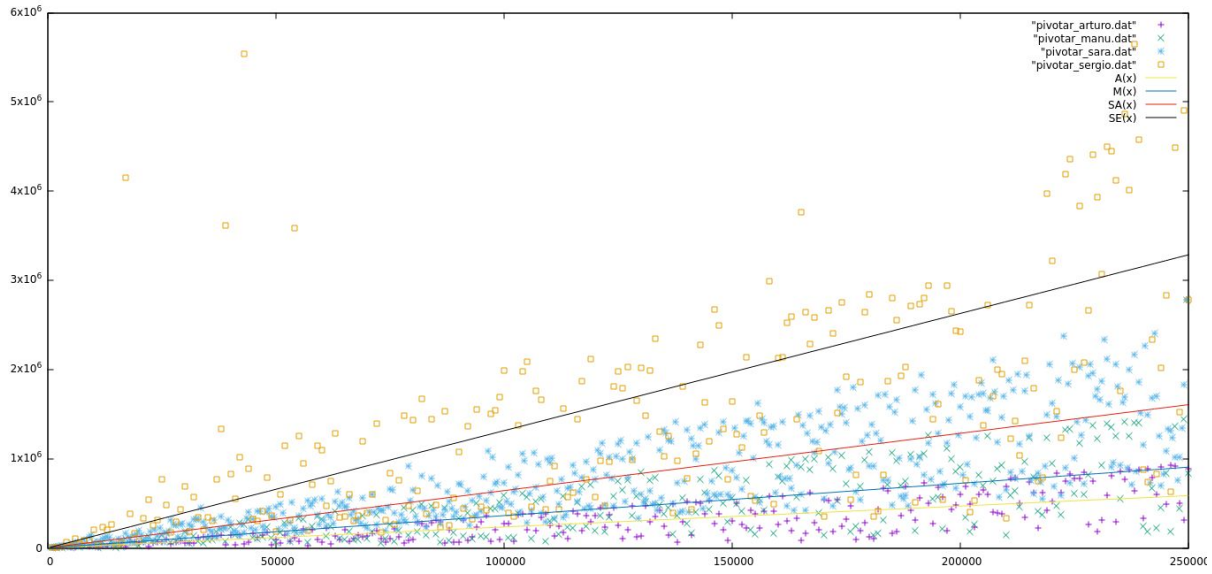
Empírica:



Híbrida:

Para este algoritmo hemos cambiado la función orden() del programa de calcular constantes ocultas por la siguiente función:

```
33   double orden(double f) {
34    return f;
35   }
```



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

3. Búsqueda

Teórica:

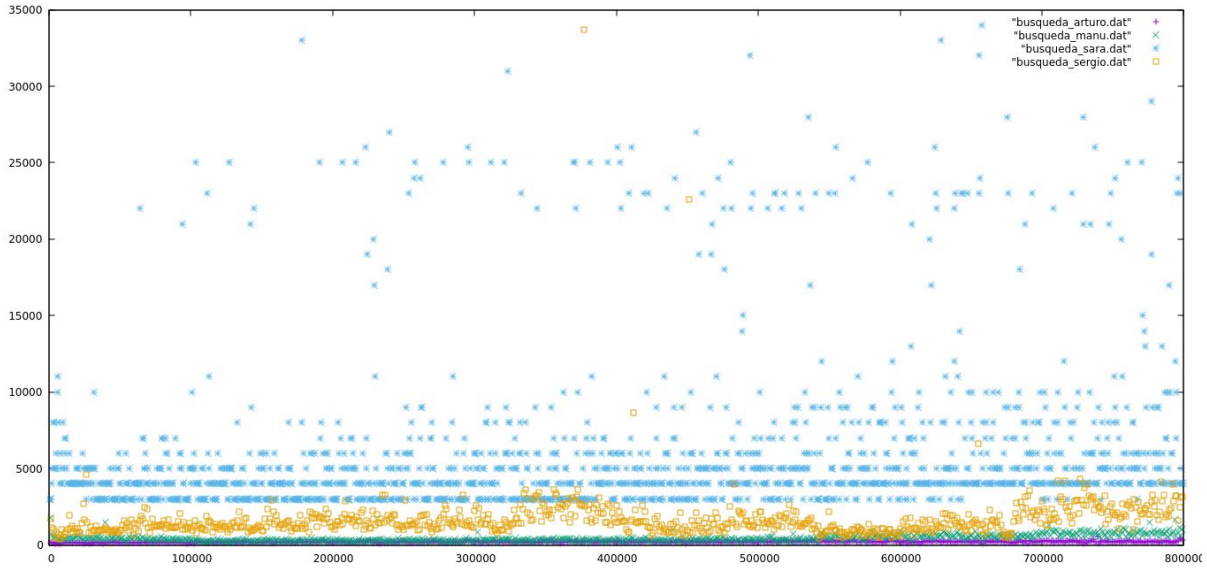
```
1   int Busqueda(int *v, int n, int elem) {
2    int inicio, fin, centro;
3    inicio = 0;
4    fin = n - 1;
5    centro = (inicio + fin) / 2;
6
7    while ((inicio <= fin) && (v[centro] != elem)) {
8      if (elem < v[centro])
```

```
 9         fin = centro - 1;
10       else
11         inicio = centro + 1;
12
13       centro = (inicio + fin) / 2;
14     }
15
16     if (inicio > fin)
17       return -1;
18
19     return centro;
20   }
```

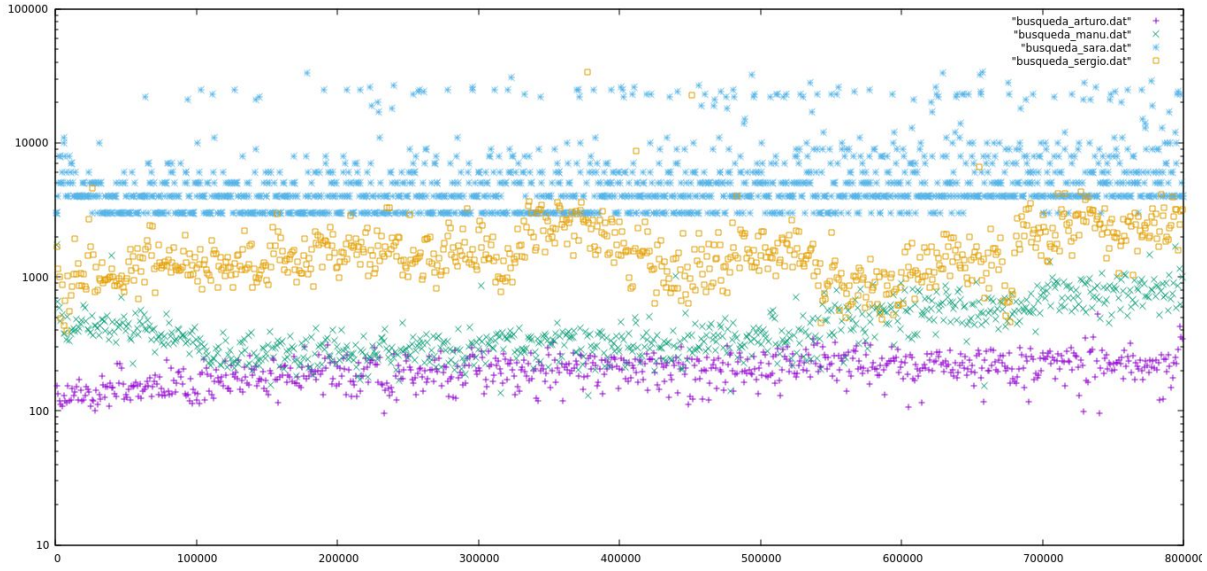
El algoritmo de Búsqueda trata de encontrar un elemento en un vector ordenado. Primero coge el elemento que se encuentra en el centro del vector y comprueba si es el elemento que estamos buscando, si lo es, devuelve la posición del elemento. En el caso contrario, divide el vector justo por la mitad en dos partes y comprueba si el elemento está en una mitad o en otra. Repite el proceso hasta que se encuentre el elemento o el valor de inicio sea mayor que fin por lo que no existe ese elemento en el vector y devuelve -1.

Sabiendo como funciona podemos ver que 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, para uno de tamaño 64 serán 6 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 $\Omega(1)$.

Empírica:



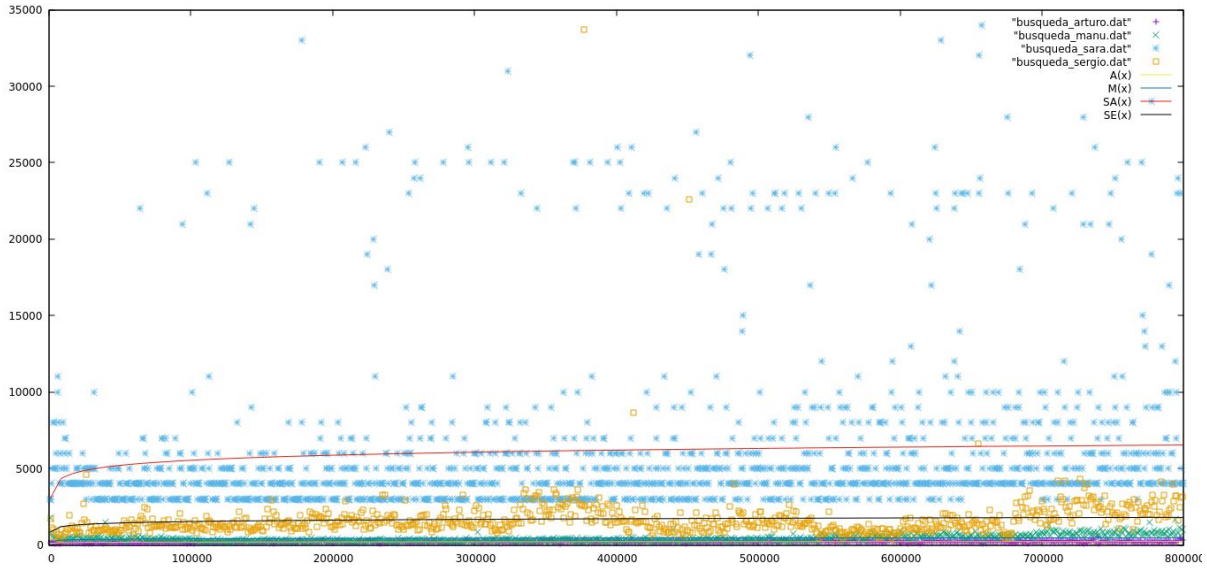
Dado que no se aprecian bien las diferencias pasamos a visualizarlo con escala logarítmica:

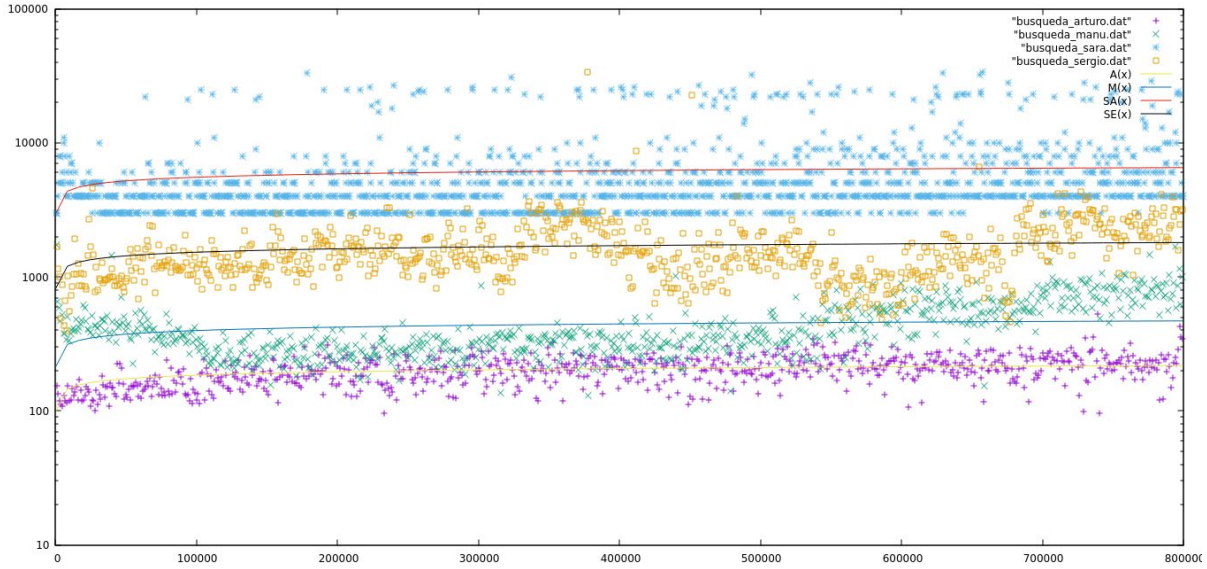


Híbrida:

Para este algoritmo hemos cambiado la función orden() del programa de calcular constantes ocultas por por la siguiente función:

```
double orden(double f) {
 return log(f); }
```





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

4. Eliminar Repetidos

Teórica:

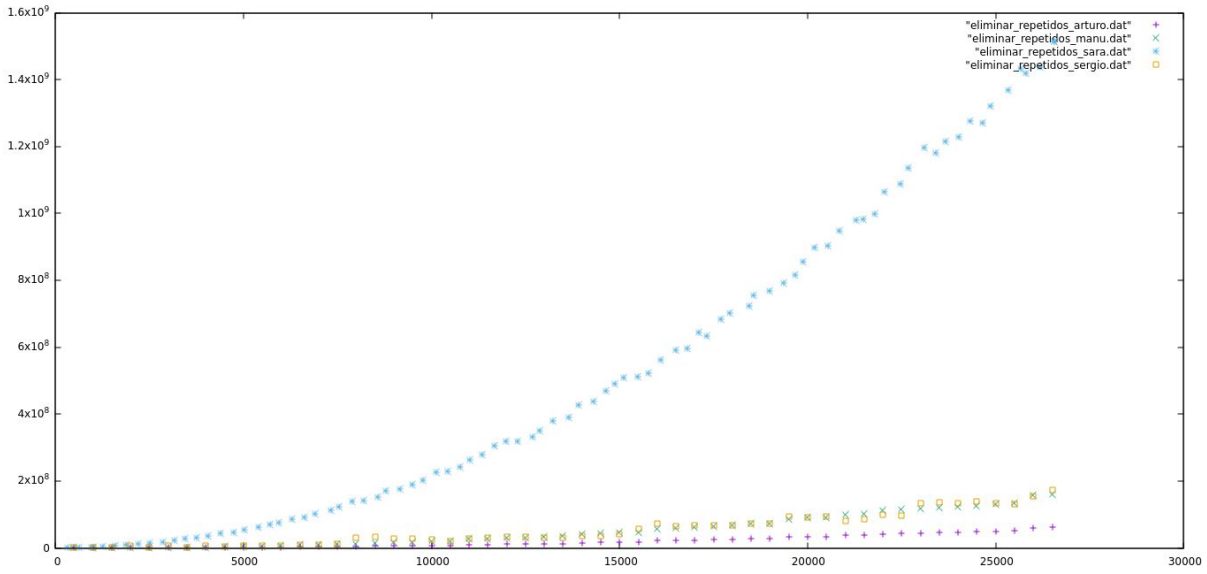
```
1   void EliminaRepetidos(double original[], int &nOriginal) {
2    int i, j, k;
3    for (i = 0; i < nOriginal; i++) {
4      j = i + 1;
5      do {
6        if (original[j] == original[i]) {
7          for (k = j + 1; k < nOriginal; k++)
8            original[k - 1] = original[k];
9
10        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

El algoritmo recibe un vector por referencia y el número de elementos que lo componen. Su objetivo es el de devolver un array en el cual han sido eliminados los elementos repetidos de su predecesor. Al fijarnos en el algoritmo vemos que entre las líneas 3 y 14 tenemos tres bucles anidados, un bucle for que realiza nOriginal iteraciones y un bucle do while y otro bucle for. Empezamos analizando este último, vemos que va desde j+1 hasta nOriginal, (nOriginal – (j+1) +1), el bucle do while va desde j hasta nOriginal (nOriginal - j + 1), por último sabemos que el bucle for mas externo va desde 0 hasta nOriginal (nOriginal – 0 + 1).

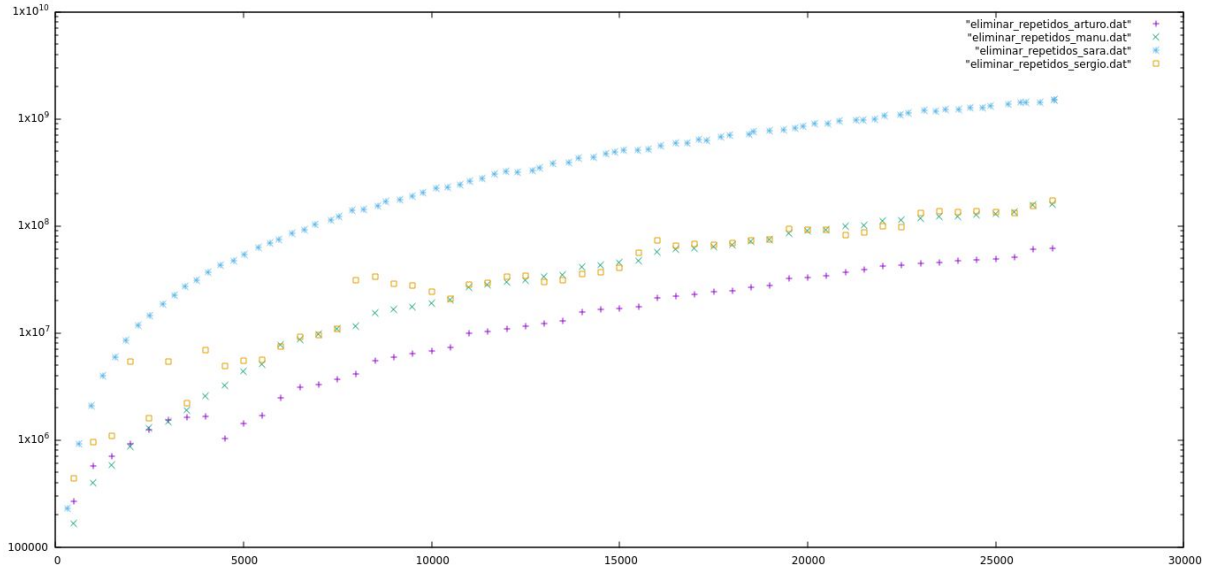
En el if de la línea 6 se altera nOriginal, por tanto no podemos asumir directamente que al haber tres bucles la eficiencia es de orden $O(n^3)$, de hecho en el caso peor, un vector de un solo número repetido n veces, siempre se entra dentro del if, y en consecuencia nOriginal es decrementado hasta que vale 1. Por tanto el bucle for de la línea 3 solo se ejecuta una vez, resultando en un algoritmo de orden $O(n^2)$. Para el caso mejor hay que tener en cuenta que el bucle for de la línea 7 solo se ejecuta en caso de encontrar un elemento repetido,

así que si no hubiera elementos repetidos sólo tendríamos dos bucles y por tanto un orden $\Omega(n^2)$

Empírica:



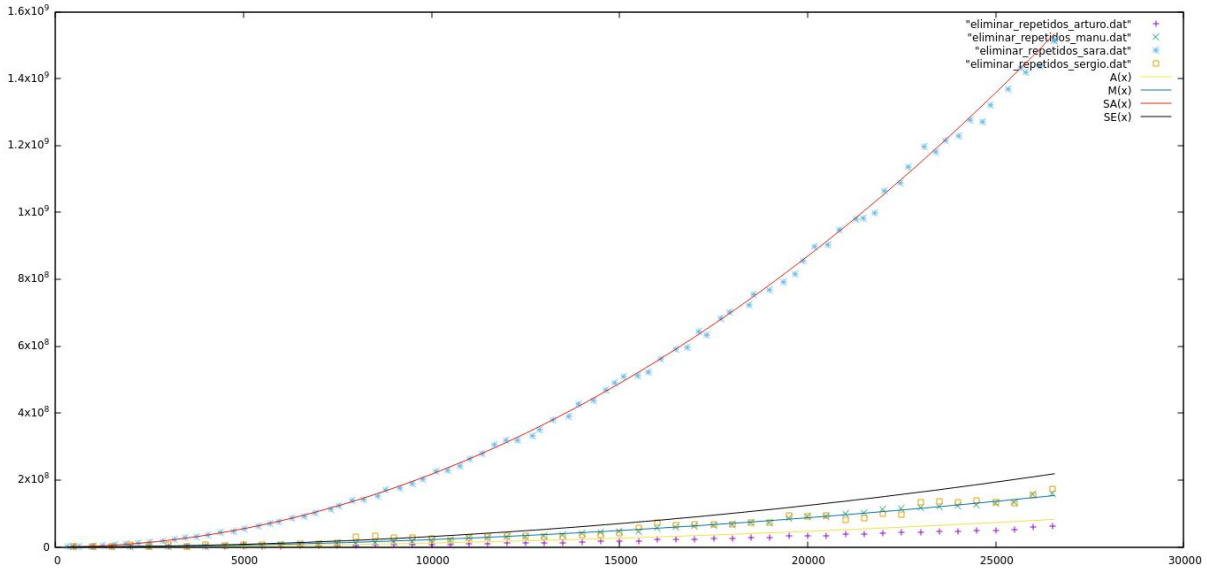
Para visualizar mejor las diferencias usamos una escala logarítmica:

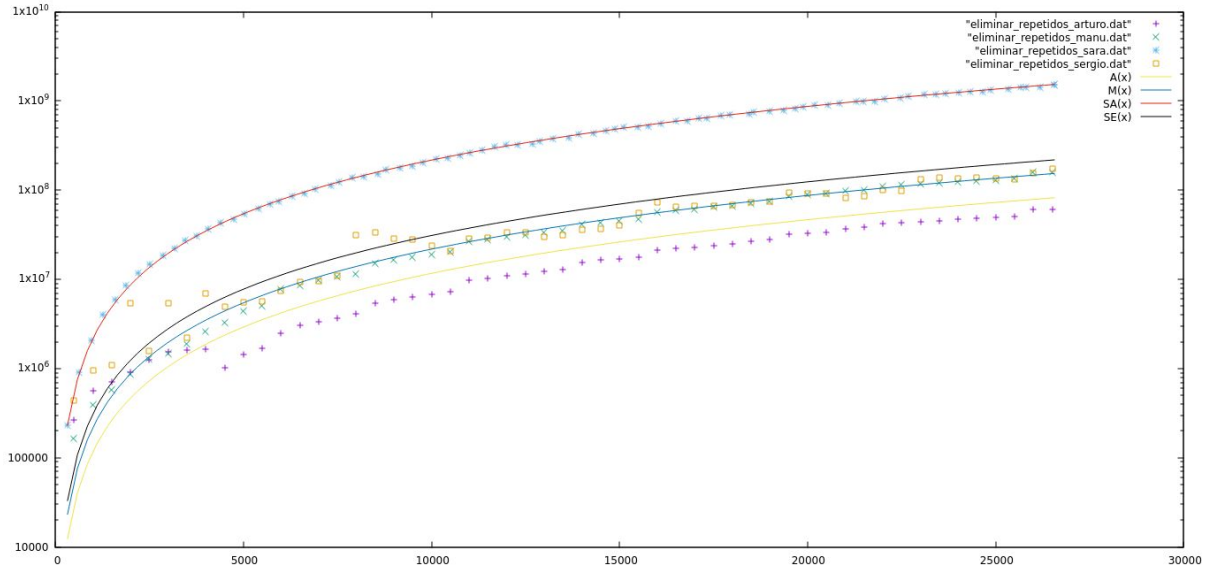


Hibrida:

Para este algoritmo hemos cambiado la función orden() del programa de calcular constantes ocultas por por la siguiente función:

```
double orden(double f) {
 return f*f;
}
```





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

5. Busqueda Binaria Recursiva

Teórica:

Peor caso (O(n)):
Vamos a considerar que cualquier sentencia simple (lectura, escritura, asignación, etc...) va a consumir un tiempo constante O(1), excepto cuando contenga una llamada a una función.
Primero calcularemos el orden de eficiencia de cada una de las sentencias del bloque. El orden de eficiencia del algoritmo será el máximo de los órdenes de eficiencia de cada una de las sentencias del bloque.

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

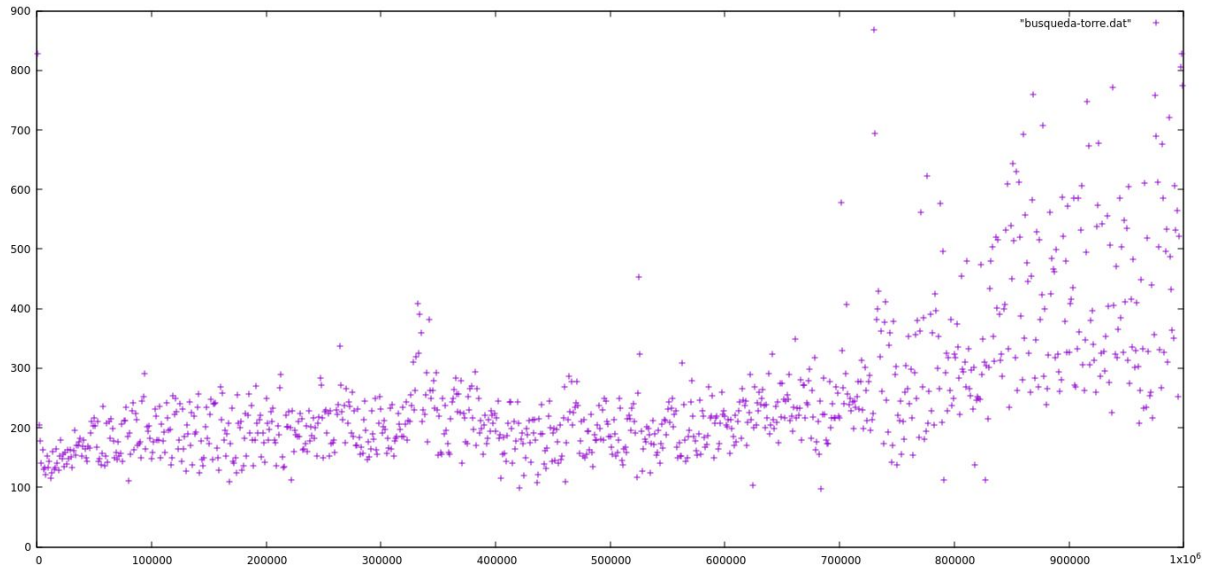
Máximo de los órdenes de eficiencia = O(log(n)).
También podemos llegar a la misma conclusión calculando el tiempo de ejecución:

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

T(n) es O(log(n))

Mejor caso (Ω(1)):
En el mejor caso el número coincidiría con el centro y el bloque if nos devolvería el centro:
if (v[centro] ==x) return centro;
Como vimos antes esta sentencia del bloque tiene de eficiencia O(1). Concluimos entonces que en el mejor caso la eficiencia es de Ω(1).

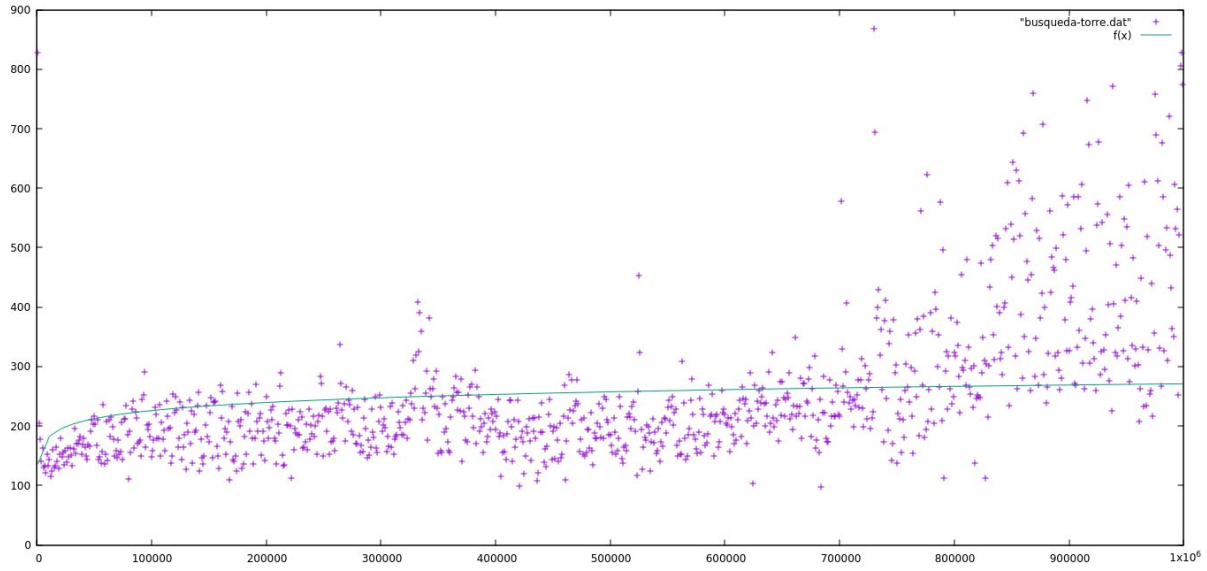
Empírica:



Híbrida:

Para este algoritmo hemos cambiado la función orden() del programa de calcular constantes ocultas por por la siguiente función:

```
double orden(double f) {
 return log(f);
}
```



K: 19.617718

6. Heapsort

Teórica:

Se trata de un algoritmo de ordenamiento por montículos. En este caso tenemos dos funciones: *void heapsort(...) y void reajustar(...)*, donde la primera llama iterativamente a reajustar.

Peor caso (O(n)):
Empezaremos analizando las eficiencias de las sentencias de bloque de la función *reajustar* ya que es a la que se llama recursivamente desde *heapsort*:

```
1   void reajustar(int T[], int num_elem, int k) {
2    int j;
3    int v;
4    v = T[k];
5    bool esAPO = false;                                  //O(1)
6    while ((k < num_elem / 2) && !esAPO) {               //O(log(n))
7      j = k + k + 1;
8      if ((j < (num_elem - 1)) && (T[j] < T[j + 1]))
9        j++;                                             //O(1)
10     if (v >= T[j])
```

```
11        esAPO = true;                                    //O(1)
12      T[k] = T[j];
13      k = j;
14    }
15    T[k] = v;
16  }
```

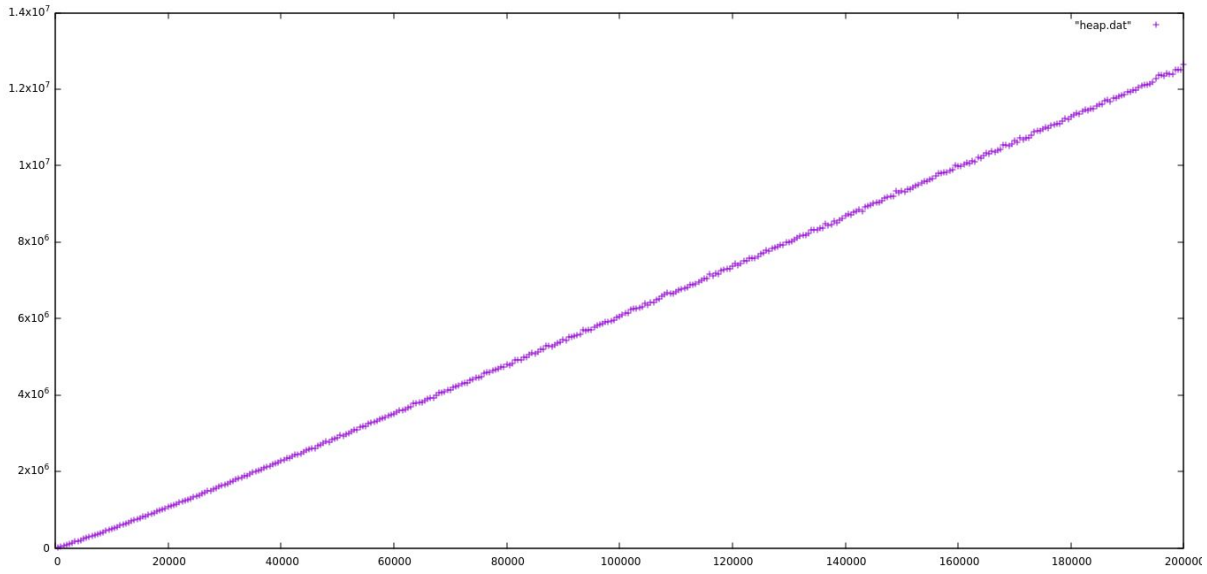
Máximo de los órdenes de eficiencia: O(log(n)).

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

Máximo de los órdenes de eficiencia: O(n log(n)).

Para el caso es de orden Ω(n) ya que podria darse el caso de que en la llamada a reajustar el vector ya sea un APO, por lo que reajustar() pasa a ser O(1).

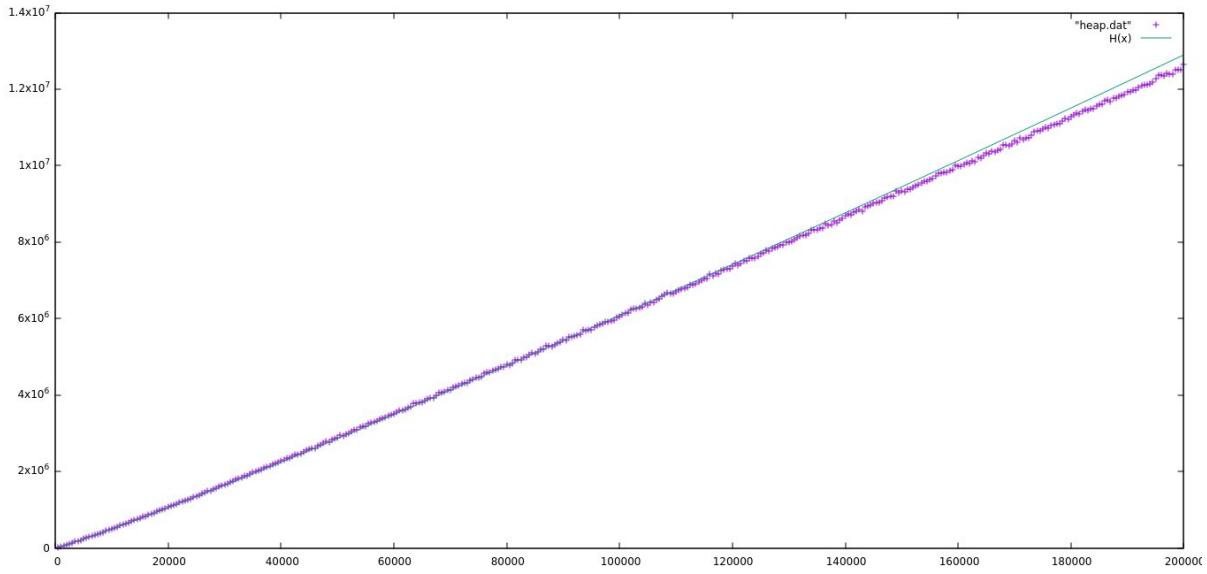
Empírica:



Híbrida:

Para este algoritmo hemos cambiado la función orden() del programa de calcular constantes ocultas por por la siguiente función:

```
double orden(double f) {
 return f*log(f);
}
```



K: 5.283937

7. Mergesort

Teórica:

```
1   static void insercion_lims(int T[], int inicial, int final) {
2    int i, j;
3    int aux;
4    for (i = inicial + 1; i < final; i++) {
5      j = i;
6      while ((T[j] < T[j - 1]) && (j > 0)) {
7        aux = T[j];
8        T[j] = T[j - 1];
9        T[j - 1] = aux;
10       j--;
11     };
12   };
13  }
14
15  inline static void insercion(int T[], int num_elem) {
16   insercion_lims(T, 0, num_elem);
17  }
```

```
1   static void fusion(int T[], int inicial, int final, int U[], int
    V[]) {
2    int j = 0;
3    int k = 0;
4    for (int i = inicial; i < final; i++) {
5      if (U[j] < V[k]) {
6        T[i] = U[j];
7        j++;
8      } else {
9        T[i] = V[k];
10       k++;
11     };
12   };
13  }
```

```
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  }
27
28  void mergesort(int T[], int num_elem) { mergesort_lims(T, 0,
    num_elem); }
```

Estamos ante un algoritmo cuya funcionalidad es la de ordenar un vector, para ello sigue los siguientes pasos:

Primero, divide el vector en vectores más pequeños hasta llegar a vectores de tamaño UMBRAL_MS (por defecto 100), los cuales se ordenan por inserción.

En segundo lugar, se juntan los vectores obtenidos en vectores de cada vez más tamaño, de forma ordenada. Al final de este proceso, tras haber fusionado todas

las subdivisiones de nuestro vector inicial, obtenemos de nuevo un único vector del mismo tamaño con los datos que lo integran ordenados.

Peor caso O(n log (n)): La parte en la que se dividen los vectores tiene un tiempo constante. La eficiencia de este algoritmo viene determinada por las fases de ordenación y mezcla de la cual podemos determinar que cada vez que se realizan tienen una eficiencia O(n). A medida que avanzamos en el problema, se debe realizar sucesivamente este paso con vectores de diferentes medidas y diferente cantidad de vectores, que son de manera explicativa: 1 vector de N elementos, 2 vectores * n/2 elementos, 4 vectores * n/4 elementos, 8 vectores * n/8 elementos… y así sucesivamente determinado por el valor de n.

De esta forma la eficiencia de este paso podemos decir que era n para cada una de las operaciones, las cuales se realizan un total de log n veces, dando un resultado de eficiencia O(n log (n)).

Podemos llegar a la misma conclusión calculando la función del tiempo de ejecución del algoritmo:

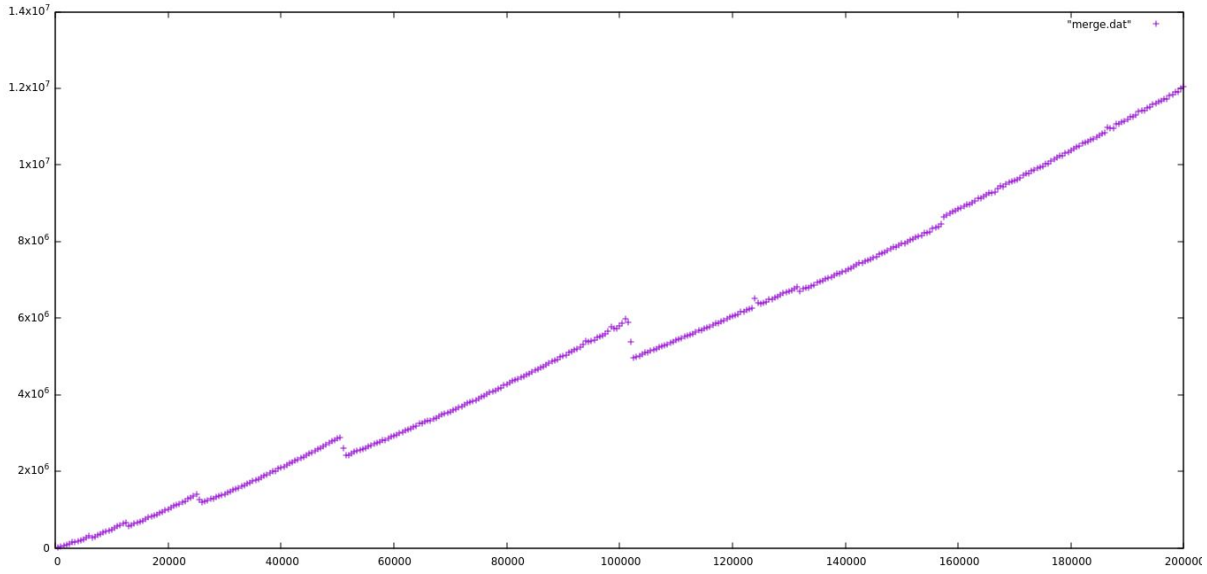
$$T(n) \begin{cases} 1, & n=1; \\ 2T(n/2)+n & n>1 \end{cases}$$

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

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

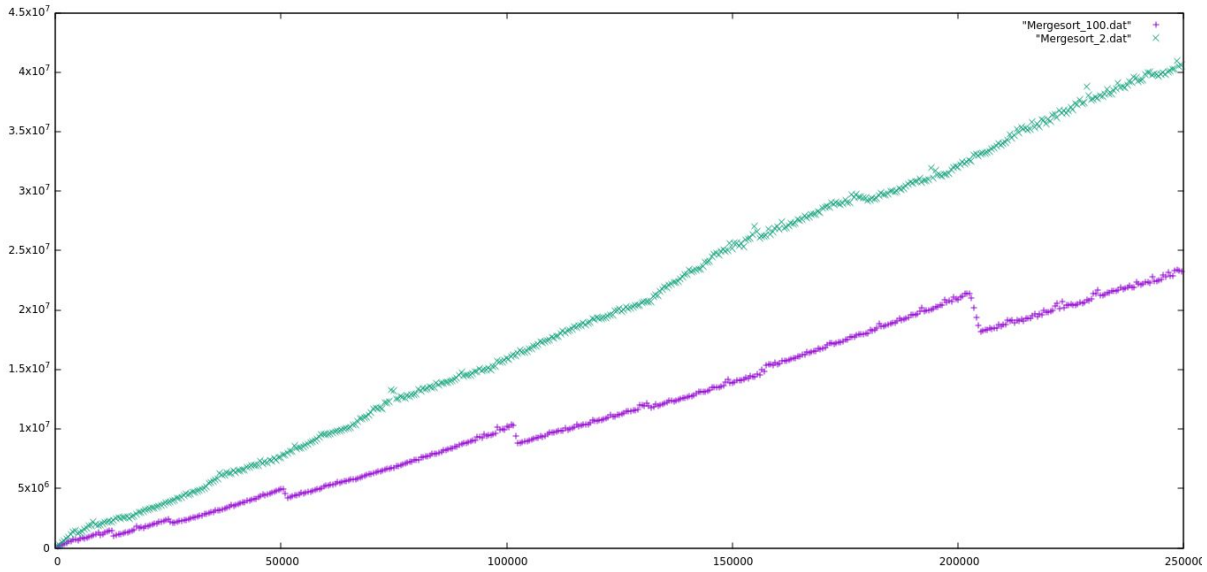
De aquí podemos concluir que $T(n)\ es\ O(nlog_2(n))$.
El mejor caso del algoritmo sería que los elementos del vector ya estuviesen ordenados aunque no presentaría grandes cambios a la hora de ejecutarse el código ya que la única parte que se saltaría es la de ordenación de elementos, pero seguiría avanzando en el resto. Por ello, también $T(n)\ es\ \Omega(nlog_2(n))$.
Cabe destacar que este algoritmo Mergesort en concreto se "beneficia" a la hora de que el vector esté ordenado, ya que lleva a cabo la ordenación por inserción.

Empírica:



Comparación de umbrales:

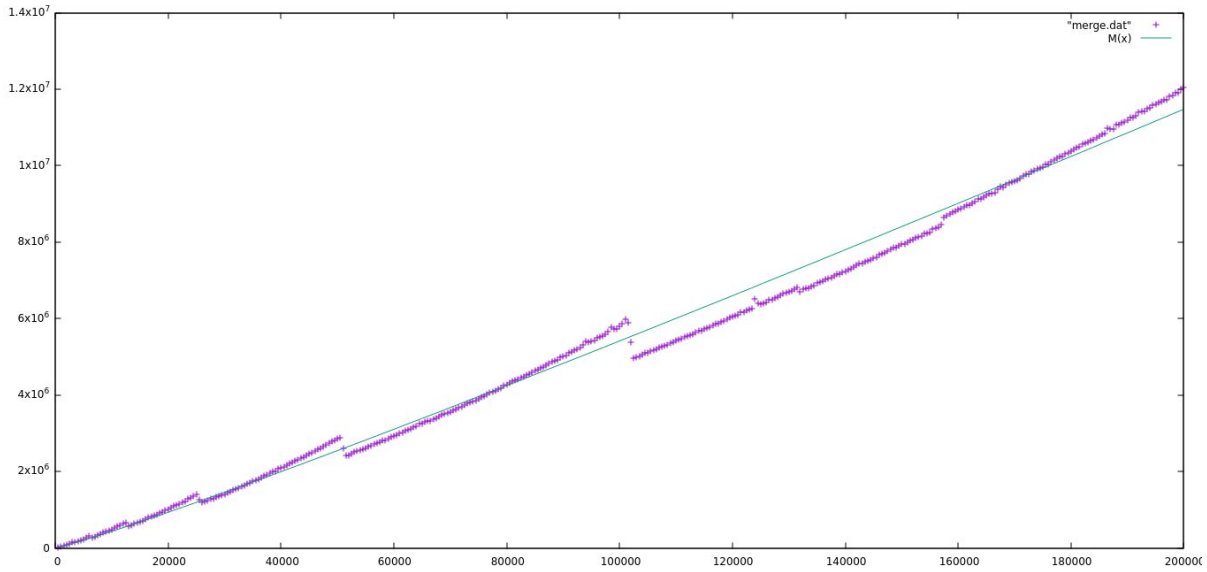
En la gráfica anterior vemos que hay unos saltos, esto es debido a que al llegar un tamaño de vector 100, pasa a usarse el algoritmo de ordenación por inserción. Si cambiamos la constante UMBRALS_MS por un número menor, veremos como estos saltos disminuyen. De hecho si la ponemos a 2 (el menor valor posible), los saltos desaparecen completamente, pero a cambio las constantes ocultas aumentan considerablemente como podemos ver en la siguiente gráfica.



Híbrida:

Para este algoritmo hemos cambiado la función orden() del programa de calcular constantes ocultas por por la siguiente función:

```
double orden(double f) {
 return f*log(f);
}
```



K: 4.701676

8. Hanoi

Teórica:

```
1   void hanoi (int M, int i, int j)
2   {
3    if (M > 0)
4      {
5        hanoi(M-1, i, 6-i-j);
6        //cout << i << " -> " << j << endl;
7        hanoi (M-1, 6-i-j, j);
8    }
9   }
```

- Peor caso:

Para calcular el orden de eficiencia del algoritmo, vamos a calcular la función del tiempo de ejecución del algoritmo T(n).

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

Luego obtenemos la siguiente ecuación de recurrencia (no homogénea):
tn - 2tn - 1 = 1
P(x) = (x-2)(x-1)
Ecuación general:

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

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

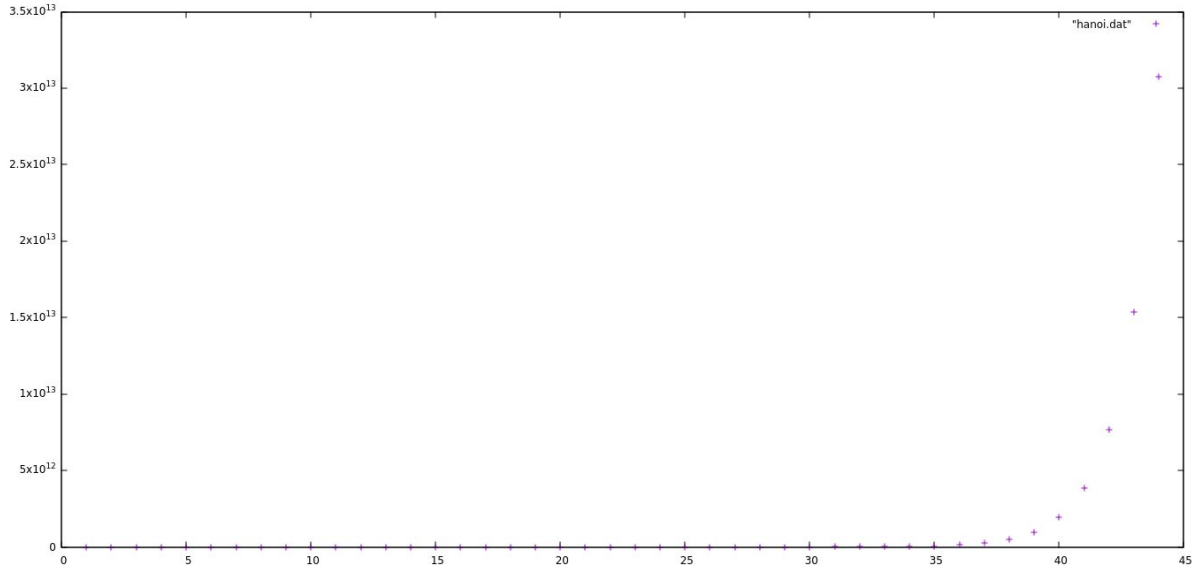
$T(1) = 1 \quad \rightarrow \quad 2c_1 + c_2 = 1 \quad \rightarrow \quad c_2 = -1$ (sustituyendo)

De aquí concluimos que la eficiencia del algoritmo es de $O(2^n - 1)$ que es igual a $O(2^n)$.

En el mejor caso el algoritmo no presentaría cambios significantes a la hora de ejecutarse las sentencias del bloque. La única variación que vemos es posible es

el tamaño del problema, M. Por ello concluimos que en el mejor caso la eficiencia sería la misma: $\Omega(2^n)$

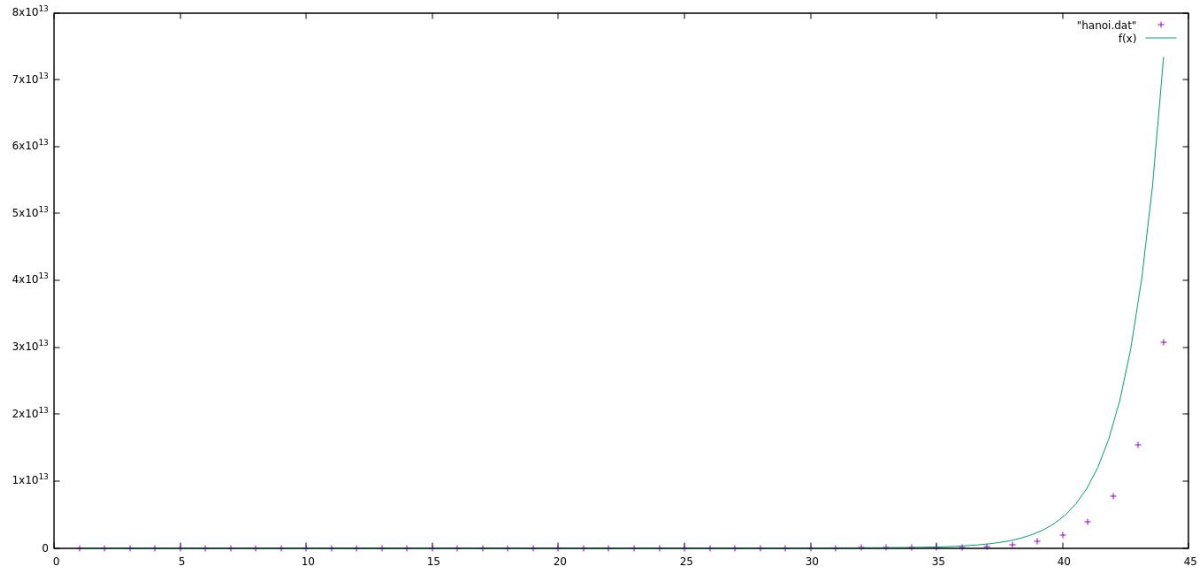
Empírica:



Híbrida:

Para este algoritmo hemos cambiado la función orden() del programa de calcular constantes ocultas por por la siguiente función:

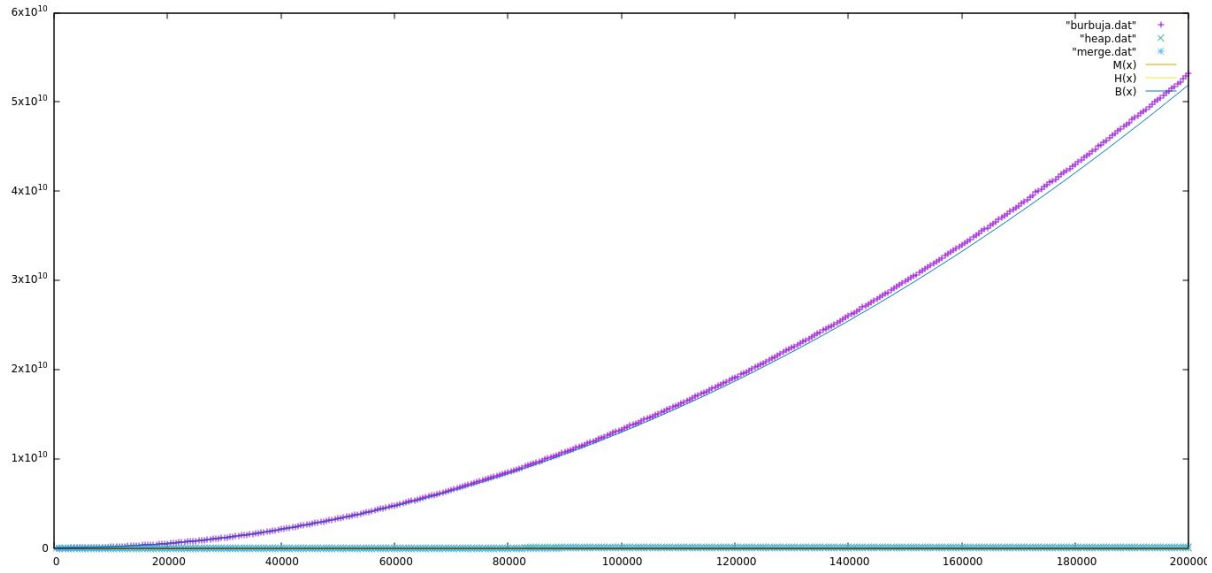
```
double orden(double f) {
 return pow(2,f);
}
```



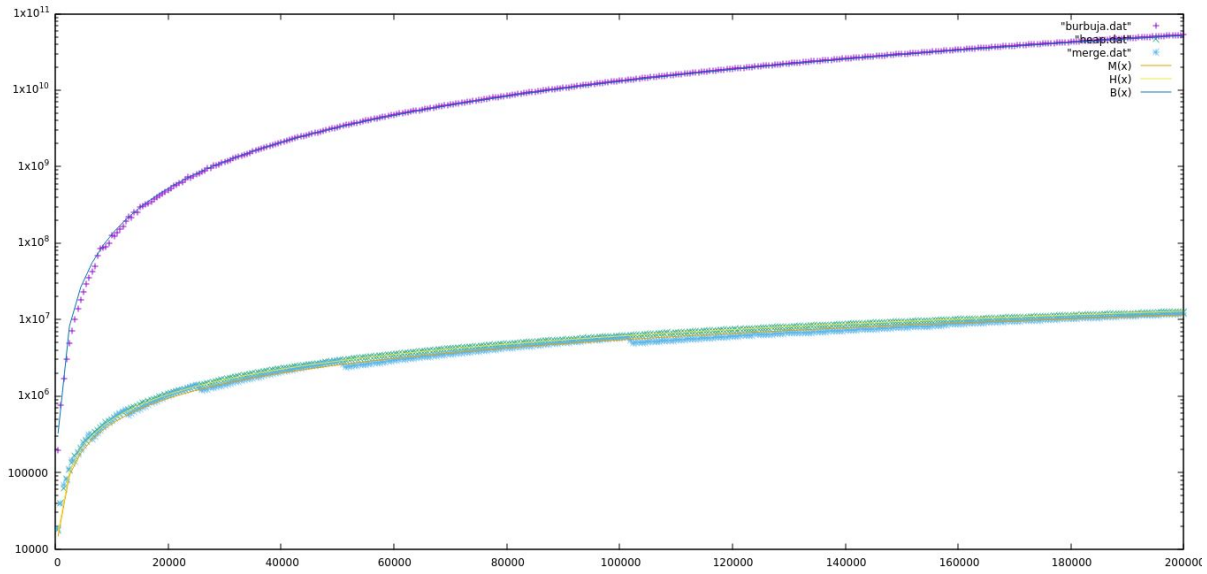
K: 4.168112

9. Comparación de Algoritmos de Búsqueda

Como hemos visto anteriormente el Heapsort y el Mergesort son de orden O(n log(n)) mientras que el burbuja es de orden cuadrático, por tanto este último es claramente inferior a los dos primeros, pero la diferencia entre Heapsort y Mergesort no está tan clara. Así que vamos a realizar un análisis híbrido.



Al introducir en la comparación el burbuja no se puede apreciar bien la diferencia entre heapsort y mergesort, así que probamos a usar una escala logarítmica.



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

Con la escala logarítmica empezamos a distinguir ligeramente entre Heapsort y Mergesort, este último parece tener tiempos inferiores. Pero donde mejor podemos apreciar la diferencia es en las constantes ocultas, la K del mergesort es aproximadamente 0,5 menos que la del heapsort, así que podemos concluir que el mergesort es más rápido que el heapsort.

10. Comparación de Distintas Flags de Optimización

Como parte extra hemos realizado una comparación de las distintas flags de optimización sobre los algoritmos Eliminar Repetidos y Mergesort. En concreto se han comparado las siguientes flags:

-O0: No aplica ninguna optimización, es el comportamiento por defecto del compilador.

-O1: Con esta flag el compilador intenta reducir el tamaño del código y el tiempo de ejecución, sin realizar ninguna optimización que requiera una gran cantidad de tiempo de compilación.

-O2: Usando este nivel de optimización gcc realiza casi todas las optimizaciones soportadas que no requieren un intercambio de espacio por velocidad. En comparación con -O1, esta opción aumenta tanto el tiempo de compilación como el rendimiento del código generado.

-O3: Cuando se le pasa esta flag al compilador, éste activa todas las optimizaciones de -O2 más aquellas que pueden requerir un intercambio de espacio por velocidad. Como resultado al compilar un código con esta flag es probable que obtengamos un ejecutable de mayor tamaño. Dicho ejecutable no siempre va más rápido que el mismo código compilado con -O2.

-Os: Esta flag optimiza pensando en el tamaño del ejecutable final. Activa todas las optimizaciones de -O2 excepto aquellas que a menudo aumentan el tamaño del código.

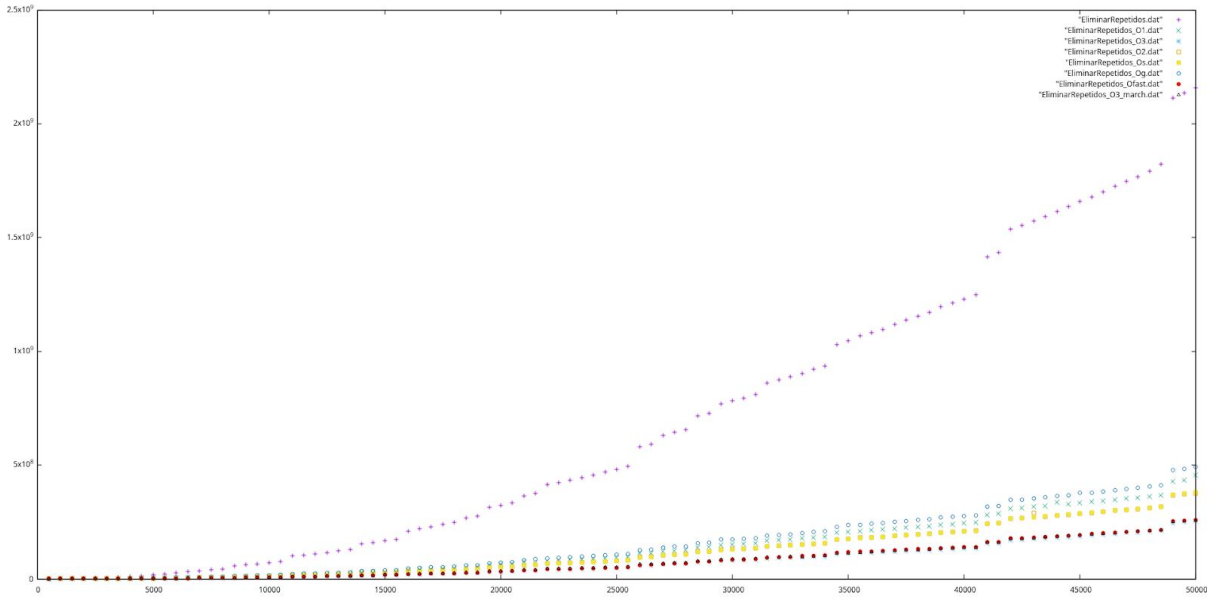
-Og: Optimiza para facilitar la depuración. Habilita todas las optimizaciones de -O1 excepto aquellas que puedan interferir con la depuración.

-Ofast: Activa todas optimizaciones de -O3 más algunas que ignora el cumplimiento estricto de las normas de compilación con el objetivo de ganar velocidad en las operaciones matemáticas. Como resultado puede que algunos programas no funcionen como se esperaba,

-O3 -march=haswell: Además de las optimizaciones de -O3, optimiza el código generado para una generación concreta de procesadores, en este caso para procesadores Intel Core de cuarta generación. Cómo resultado el compilador podrá utilizar los siguientes conjuntos de instrucciones: MOVBE, MMX, SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, POPCNT, AVX, AVX2, AES, PCLMUL, FSGSBASE, RDRND, FMA, BMI, BMI2 y F16C.

Si el procesador donde se ejecuta el binario generado no tiene soporte para estas instrucciones, la ejecución resultará en un error.

Eliminar Repetidos:



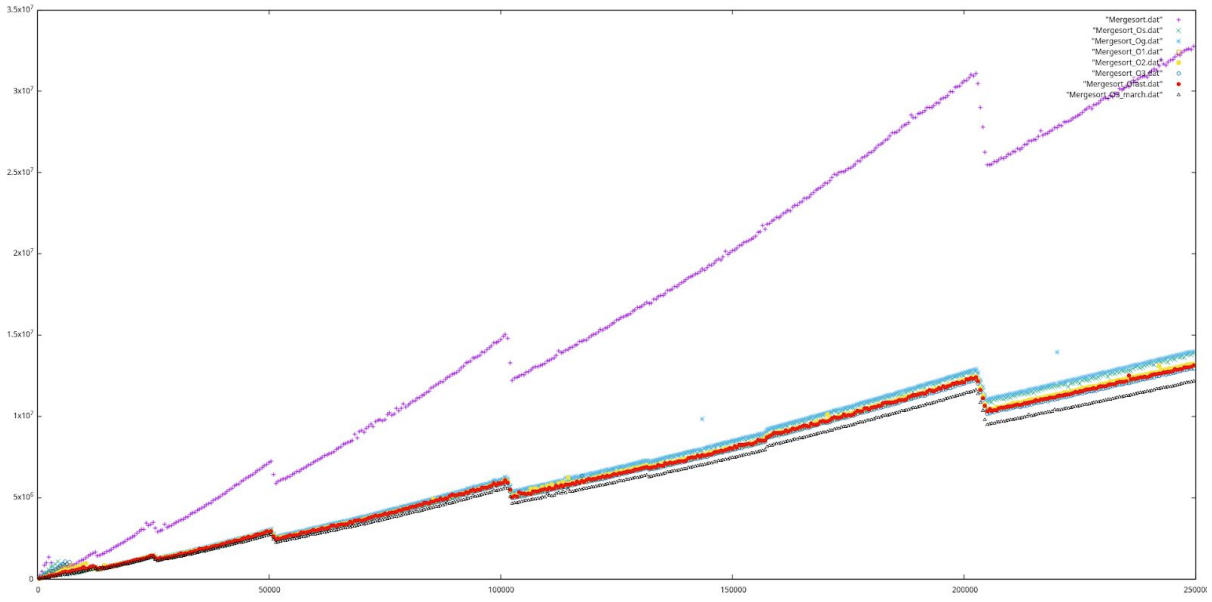
En esta gráfica podemos observar como las distintas opciones de optimización afectan a los tiempos de ejecución. La gráfica más alta, como era de esperar, se corresponde con la ausencia de flags de optimización (o -O0).

Las siguientes dos se corresponden con -Og y -O1, Og en este caso parece haber optimizado menos que -O1 buscando conservar la "depurabilidad" del binario.

Las dos siguientes son -Os y -O2, las cuales están completamente solapadas, así que podemos suponer que el compilador no ha podido aplicar ninguna optimización de reducción de tamaño. De haberlo hecho lo más probable es que la velocidad de ejecución se hubiera visto resentida.

Las tres últimas líneas corresponden a -O3, -Ofast y menos -O3 -march=haswell. Podemos ver que también están solapadas y por lo que en este algoritmo las optimizaciones extra que añaden -Ofast y -march=native no afectan a la velocidad de ejecución

Mergesort:



Al igual que en el caso anterior aquí vemos la gran diferencia entre poner y no poner flags de optimización. En este algoritmo vemos como la flag que menos mejora su rendimiento es -Og seguido de -Os, lo que nos da a entender que las optimizaciones que no se han añadido (buscando mejorar la "depurabilidad" y reducir el tamaño) han resultado en una ligera pérdida de rendimiento.
A continuación vemos como -O1, -O2, -O3 y -Ofast están prácticamente solapadas. Y por último podemos ver como en este caso la flag -march=haswell sí que ha hecho efecto, el compilador ha podido aprovechar las instrucciones del procesador para mejorar la velocidad de ejecución del programa.