DIVIDE Y VENCERÁS

PRÁCTICA 1 ALGORÍTMICA

2017

Jose Antonio Ruiz Millán — jantonioruiz@correo.ugr.es
Alejandro Rodríguez Muñoz — aromu@correo.ugr.es
Julio Antonio Fresneda García — juliofresnedag@correo.ugr.es
Adrián Peláez Vegas — adrianpelaez@correo.ugr.es

GRUPO B3

DIVIDE Y VENCERÁS

Contenido

Introducción

En este documento se redactará la solución formulada para cada uno de los problemas que se planteaban en la práctica 1 de la asignatura algorítmica. Se explicará el funcionamiento del algoritmo formulado así como el estudio de su eficiencia. También se aportarán las gráficas de las pruebas de tiempo realizadas con dicho algoritmo. Por último mostraremos la codificación de los algoritmos en c++.

1. Comparación de preferencias

1.1. Introducción al problema

Este problema tiene como idea principal la comparación de dos rankings y dar como salida lo parecidos que son ambos. Para representar un ranking usaremos un vector de enteros, donde la posición 0 representa el primer elemento del ranking y la posición n el último. Para simplificar el problema asumiremos que uno de los rankings estará ordenado en orden creciente.

Trataremos de encontrar el número de inversiones que tendríamos que hacer con los elementos del vector para que este quedara ordenado en orden creciente. Como el vector con el que lo comparamos está ordenado en orden creciente, el número de inversiones calculado coincide con el número de inversiones respecto al vector con el que comparamos.

1.2. Descripción de la solución

Nuestra solución tiene como base el algoritmo mergesort. El vector inicial se dividirá en dos vectores por la mitad de forma recursiva. Tendremos que tener en cuenta si el vector es par o impar en cada llamada a la función recursiva para que no se produzca un error a la hora de dividir los vectores. Esta división se hará hasta que los vectores resultantes tengan tamaño 1, tendremos entonces una estructura en forma de árbol con todas las subdivisiones del vector inicial. Una vez llegados a este punto, la función no vuelve a llamarse a sí misma de forma recurrente, y pasa a ejecutarse la siguiente función, la función mezclar vectores. Esta función recorrerá el árbol desde abajo hacia arriba, dando como resultado el vector producto de la mezcla y ordenación de sus vectores descendientes. Hasta este punto el algoritmo es un algoritmo de ordenación de vectores mergesort, pero aquí es donde tenemos que contar las inversiones que hay en los vectores que mezclamos. Para ello, la función mezclar sumará una determinada cantidad a la variable que contabiliza las inversiones, en el caso de que las haya. En primer lugar, hay que tener en cuenta cuando hay una inversión, y cuando no la hay, la habrá cuando, a la hora de mezclar y ordenar, haya un elemento en el vector izquierdo mayor que otro elemento en el vector derecho con el que lo comparamos, en este caso habría que introducir en el vector ordenado primero el elemento del vector derecho, y tendríamos una inversión, por lo tanto tenemos que sumarle una cantidad al contador de inversiones, pero… ¿qué cantidad habrá que sumarle?

Intuitivamente parece lógico pensar en incrementar en una unidad el contador de inversiones, pero esto es un fallo, ya que de esta forma no contabiliza correctamente el total de inversiones. Para entender la causa del fallo hay que tener en cuenta varias cosas:

1.- Tanto el vector hijo de la derecha como el de la izquierda se encuentran ordenados de forma creciente a la hora de contar las inversiones.

2.- Cuando comparamos un elemento del vector izquierdo con otro del derecho, el menor de ambos es introducido al vector resultado y eliminado del vector hijo.

Teniendo claros estos dos aspectos, podemos entender que si incrementamos en una unidad el contador de inversiones en cualquier caso en el que encontremos un elemento en el hijo izquierdo que sea mayor que otro elemento del hijo derecho, la inversión entre ambos se contabilizará y el elemento del hijo derecho se añadirá al vector solución y será eliminado del vector hijo, ya que este es menor y debe ir antes para que el resultado se encuentre ordenado de forma creciente. De este modo el elemento eliminado no volverá a ser comparado con el resto de elementos del vector izquierdo, y como sabemos que los vectores están ordenados de forma creciente, todos los elementos siguientes al comparado en el vector izquierdo también van a ser mayores que el comparado en el vector derecho, provocando así una inversión en cada uno de ellos, pero estas inversiones pasarán desapercibidas al haber añadido ya el elemento menor al vector solución, ya que no se volverá a comparar este elemento con ningún otro, por lo tanto el resultado será erróneo.

La solución optada para resolver este inconveniente es tener un índice asociado a cada elemento del vector izquierdo irá en orden decreciente desde el primero al último, teniendo de esta forma el primer elemento del vector como índice el tamaño del mismo, y el último elemento un 1 como índice. A la hora de sumar una cantidad al contador de inversiones, se le sumará el índice del elemento del vector izquierdo que provoca la inversión. De esta forma contabilizamos todas las inversiones y la solución es correcta.

1.3. Código

Función mezcla:

Encargada de la subdivisión de los vectores. Será la función que se llame así misma de forma recursiva.

```
void mezcla(vector<int> &v)
{
    vector<int> vector1;
    vector<int> vector2;
    int n1, n2,i,j;

    if (v.size() > 1)
    {
        if (v.size()%2 == 0)
            n1=n2=(int) v.size() / 2;
        else
        {
            n1=(int) v.size() / 2;
            n2=n1+1;
        }
        for(i=0;i<n1;i++)
            vector1.push_back(v[i]);
        for(j=0;j<n2;i++,j++)
            vector2.push_back(v[i]);
        v.clear();
        mezcla(vector1);
        mezcla(vector2);
        combinar(vector1, vector2, v);
    }
}
```

Función combinar:

Encargada de copiar los elementos de los vectores hijos en el vector resultado de forma ordenada, y de contabilizar las inversiones existentes.

```
void combinar(const vector<int> &arreglo1,const vector<int> &arreglo2, vector<int> &arreglo3)
{
    int x1=0, x2=0 , indice = arreglo1.size();

    while (x1<arreglo1.size() && x2<arreglo2.size()) {
        if (arreglo1[x1]<arreglo2[x2]) {
            arreglo3.push_back(arreglo1[x1]);
            x1++;
            indice--;
        }
        else {
            arreglo3.push_back(arreglo2[x2]);
            x2++;
            inv+=indice;
        }
    }
    while (x1<arreglo1.size()) {
        arreglo3.push_back(arreglo1[x1]);
        x1++;
    }
    while (x2<arreglo2.size()) {
        arreglo3.push_back(arreglo2[x2]);
        x2++;
    }
}
```

1.4. Algoritmo de fuerza bruta

```
int FuerzaBruta(vector<int> &v){
  int inv = 0;
  for(int i = 0; i < v.size()-1; i++)
    for(int j = i+1; j < v.size(); j++)
      if(v[i] > v[j]) inv++;
  return inv;
}
```

1.5. Análisis de eficiencia

Análisis de eficiencia teórica:

➔ *Divide y vencerás:*

Cálculo de la eficiencia del algoritmo usado para calcular las inversiones:
Suponemos que n es potencia de 2:

$$T(n) \begin{cases} c_1 & si \;\; n = 1 \\ 2T(n/2) + c_2 n & si \;\; n > 1, \;\; n = 2^k \end{cases}$$

Para saber la eficiencia, podemos usar expansión:
$T(n) = 2T(n/2) + c_2 n$
$T(n/2) = 2T(n/4) + c_2 n/2$
Es decir,
$T(n) = 4T(n/4) + 2c_2 n$ o
$T(n) = 8T(n/8) + 3c_2 n$
En general:

$$T(n) = 2^i T(n/2^i) + ic_2 n$$

Tomando $n = 2^k$, la expansión termina cuando llegamos a $T(1)$ en el lado de la derecha, lo que ocurre cuando $i = k$
$T(n) = 2^k T(1) + kc_2 n$
Ya que $2^k = n$, operando, $k = log(n)$;
Además, $T(1) = c_1$, por lo que tenemos:

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

Por tanto el tiempo del algoritmo es O(nlog(n))

➔ *Fuerza bruta:*

Cálculo de la eficiencia del algoritmo basado en fuerza bruta usado para el ejercicio 3. Vamos a considerar que evaluamos un vector de n elementos. El algoritmo consiste en un for desde 0 hasta n, todo estro dentro de otro for desde 0 hasta n igual, por lo que la eficiencia sería de $O(n^2)$.

Análisis eficiencia empírica/híbrida:

Para analizar la eficiencia de forma empírica hemos recopilado un conjunto de datos del tiempo de ejecución de ambos algoritmos, los hemos representado gráficamente y hemos ajustado una función que defina su eficiencia.

(Para acceder al fichero completo de los datos tomados hacer click en el nombre de la técnica usada)

➔ *Datos*

Fuerza bruta:

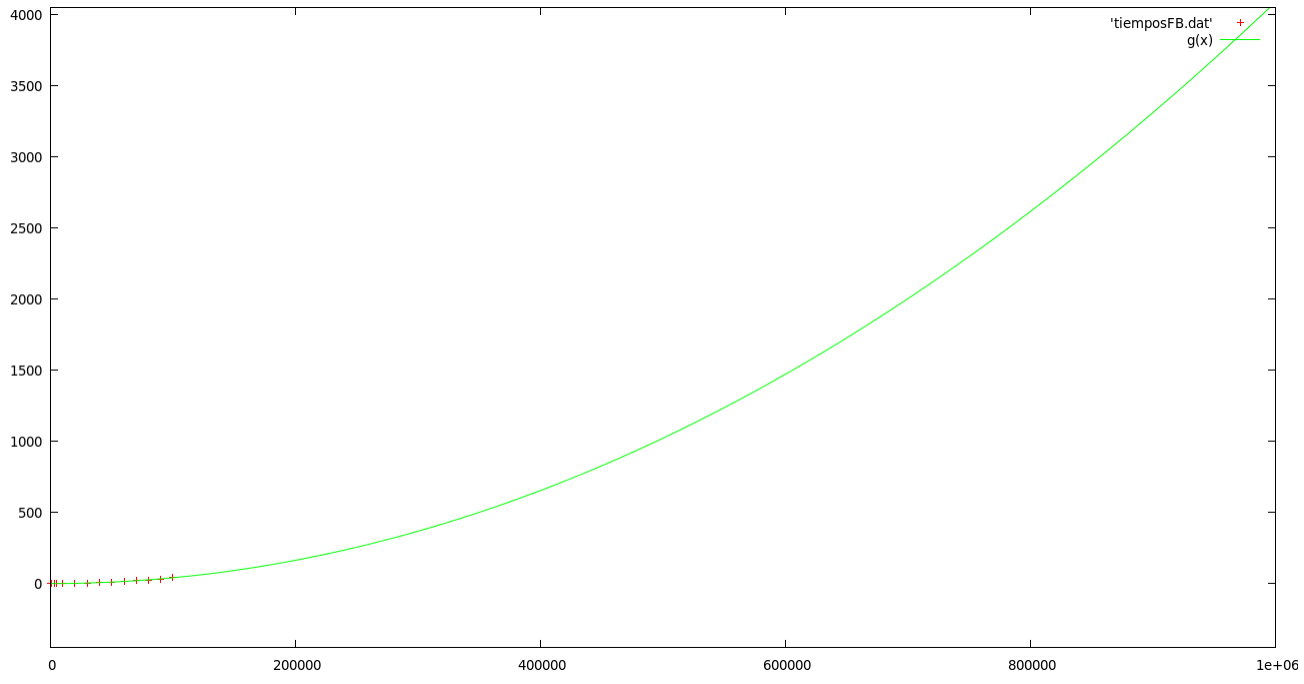
```
100 4.0583e-05
500 0.00108763
1000 0.00403914
3000 0.0357795
5000 0.103624
10000 0.392694
20000 1.57983
30000 3.56953
40000 6.68464
50000 9.96494
60000 14.4023
70000 19.8973
80000 25.4819
90000 32.3059
100000 42.1949
```

Divide y vencerás:

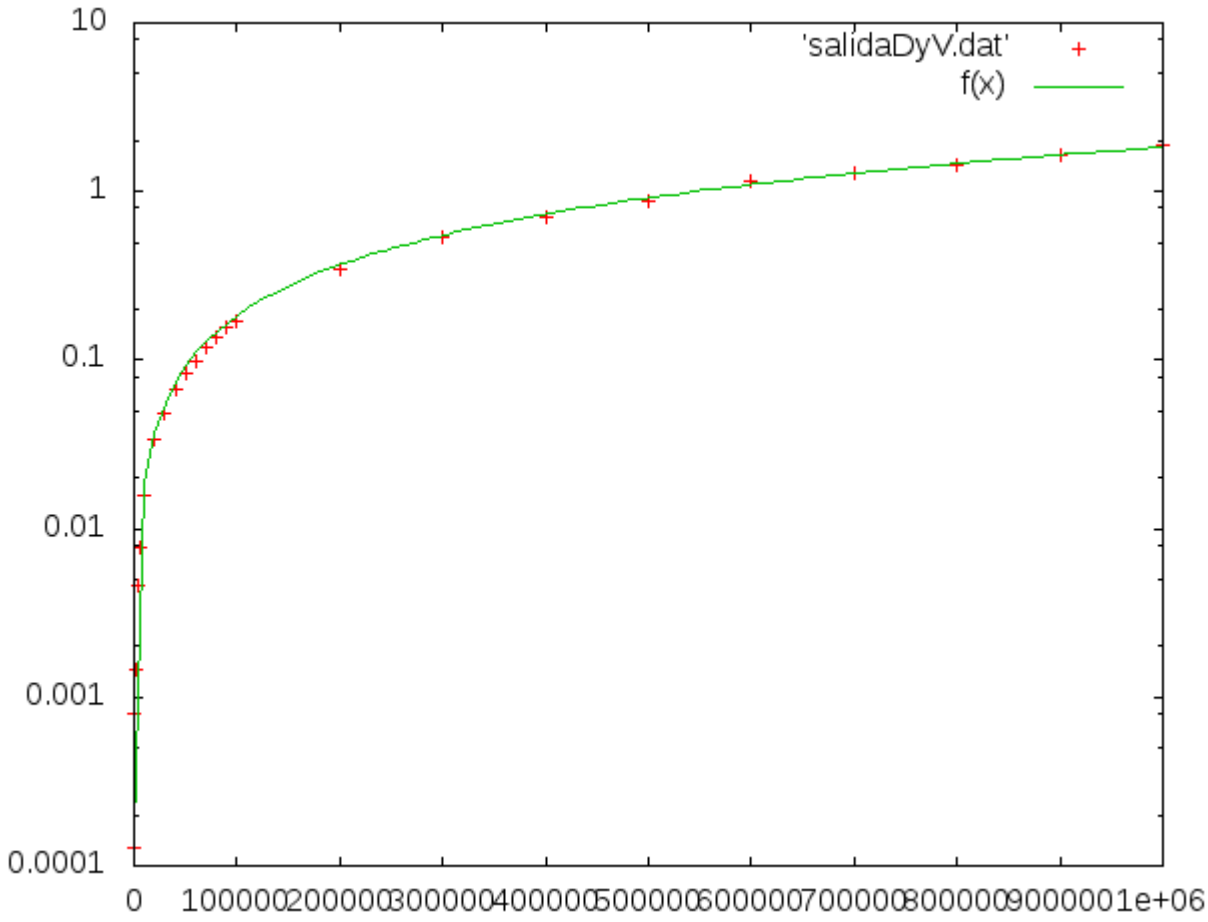
```
100 0.000128899
500 0.000799941
1000 0.00147669
3000 0.00459015
5000 0.00778487
10000 0.0159416
20000 0.0334637
30000 0.048175
40000 0.0672733
50000 0.0838394
60000 0.0979629
70000 0.118267
80000 0.137071
90000 0.154633
100000 0.170719
200000 0.347859
300000 0.533522
400000 0.712448
500000 0.887112
```

➔ *Gráficas:*

- Fuerza bruta:

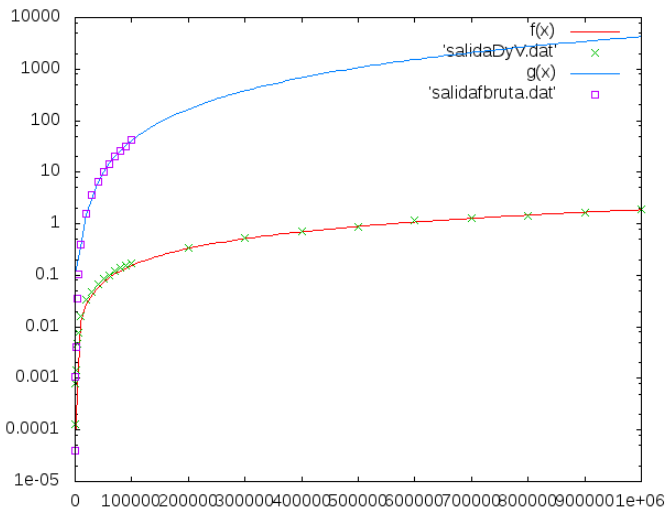


- Divide y vencerás:

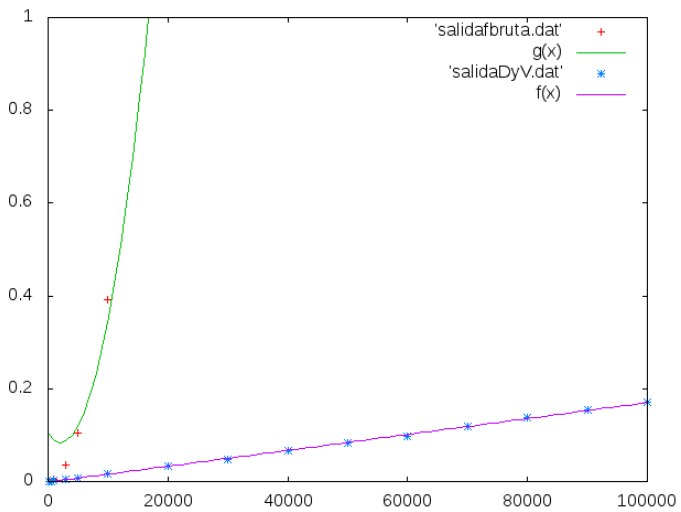


- Comparativa:

Con escala logarítmica:



Sin escala logarítmica:



Como podemos observar, gracias al análisis empírico de la eficiencia de ambos algoritmos, podemos reafirmar lo ya demostrado en el análisis teórico. El algoritmo divide y vencerás, el cual tiene una eficiencia de O(nlog(n)), es bastante más eficiente que el algoritmo de fuerza bruta, cuya eficiencia es O(n2).

2. Eliminar elementos repetidos

2.1. Introducción al problema

En este problema tenemos un vector inicial de enteros cuyos elementos pueden estar repetidos.

La solución consiste en dar como salida otro vector de enteros en el cual aparecerán todos los elementos que aparecían en el inicial, pero eliminando todas las repeticiones de elementos.

2.2. Descripción de la solución

Al igual que en el primer problema, la solución aportada tiene como base un algoritmo de ordenación por divide y vencerás mergesort. Este dividirá el vector inicial llamándose así mismo recursivamente hasta que el tamaño del vector sea 1. Una vez llegados a este punto, el vector deja de dividirse y comienza a ejecutar la función combinar, la cual es la encargada de unir los dos vectores hijos en un solo vector de forma ordenada crecientemente.

La clave para que la solución sea correcta está en esta función, la función combinar. Esta función va recorriendo y comparando ambos vectores hijos para id introduciendo en el vector resultado el elemento menor de cada comparación. En este punto, nuestro algoritmo añade una diferencia respento al algoritmo clásico mergesort, y es una condicional que se cumpla cunado ambos elementos comparados sean iguales, en cuyo caso se avanzará en el recorrido de ambos vectores hijos pero solo se añadirá uno de los elementos comparados al vector resultado. De esta forma el vector final quedará con todos los elementos que tenía el vector inicial, pero sin elementos repetidos.

2.3. Código

Función eliminar repetidos:

```
void elimRepetidos(vector<int> &v)
{
    vector<int> vector1;
    vector<int> vector2;
    int n1, n2,i,j;

    if (v.size() > 1)
    {
        if (v.size()%2 == 0)
            n1=n2=(int) v.size() / 2;
        else
        {
            n1=(int) v.size() / 2;
            n2=n1+1;
        }
        for(i=0;i<n1;i++)
            vector1.push_back(v[i]);
        for(j=0;j<n2;i++,j++)
            vector2.push_back(v[i]);
        v.clear();
        elimRepetidos(vector1);
        elimRepetidos(vector2);
        combinar(vector1, vector2, v);
    }
}
```

Combinar:

```
void combinar(const vector<int> &arreglo1,const vector<int> &arreglo2, vector<int> &arreglo3)
{
    int x1=0, x2=0;

    while (x1<arreglo1.size() && x2<arreglo2.size()) {
        if (arreglo1[x1]<arreglo2[x2]) {
            arreglo3.push_back(arreglo1[x1]);
            x1++;
        }
        else if( arreglo1[x1]>arreglo2[x2] ) {
            arreglo3.push_back(arreglo2[x2]);
            x2++;
        }
        else{
          arreglo3.push_back(arreglo1[x1]);
          x1++;
          x2++;
        }
    }
    while (x1<arreglo1.size()) {
        arreglo3.push_back(arreglo1[x1]);
        x1++;
    }
    while (x2<arreglo2.size()) {
        arreglo3.push_back(arreglo2[x2]);
        x2++;
    }
}
```

2.4. Algoritmo de fuerza bruta

```
void eliminar_repetidos( vector<int> &vec ){
  vector<int> sinrep;
  bool encontrado = false;

  for( unsigned int i=0; i<vec.size(); i++ ){
    for( unsigned int j=i+1; j<vec.size(); j++ ){
      if( vec[i] == vec[j] ){
        encontrado = true;
      }
    }
    if( !encontrado ) sinrep.push_back(vec[i]);
    encontrado = false;
  }

  vec.clear();
  for( unsigned int i=0; i<sinrep.size(); i++ ){
    vec.push_back(sinrep[i]);
  }

}
```

2.5. Análisis de eficiencia

Análisis de eficiencia teórica

➔ *Divide y vencerás*

Cálculo de la eficiencia del algoritmo basado en divide y vencerás usado para el ejercicio 2.

Vamos a considerar que evaluamos un vector de$n$ elementos, siendo n potencia de 2 (2^k).

En el peor caso:

$T(n)\begin{cases} c_1 & si\ n=1 \\ 2T(n/2)+c_2 & si\ n>1, n=2^k \end{cases}$

Para saber la eficiencia, vamos a usar expansión:

$T(n) = 2T(n/2) + c_2$

$2T(n/2) = 2T(n/4) + c_2$

Es decir: $T(n) = 4T(n/4) + 2c_2$

o

$T(n) = 8T(n/8) + 3c_2$

En general: $T(n) = 2^iT(n/2^i) + ic_2$, siendo i el número de llamadas recursivas.

Cuando $i = k$, quiere decir que no habrá más llamadas recursivas, es decir, en la parte derecha hay $T(1)$.

La fórmula quedaría: $T(n) = 2^kT(1) + kc_2$ Como $2^k = n$, $k = log_2(n)$.

La fórmula finalmente quedaría: $T(n) = nT(1) + c_2log_2(n)$

Podemos tomar $T(1) = c_1$ Por lo que la fórmula quedaría $T(n) = c_1n + c_2log_2(n)$, por lo que la eficiencia sería $O(nlog(n))$.

➔ *Fuerza bruta*

Cálculo de la eficiencia del algoritmo basado en fuerza bruta usado para el ejercicio 2. Vamos a considerar que evaluamos un vector de n elementos. El algoritmo consiste en un for desde 0 hasta n, todo estro dentro de otrofor desde 0 hasta n igual, por lo que la eficiencia sería de $O(n^2)$.

Análisis eficiencia empírica/híbrida

➔ *Datos*

Fuerza bruta:

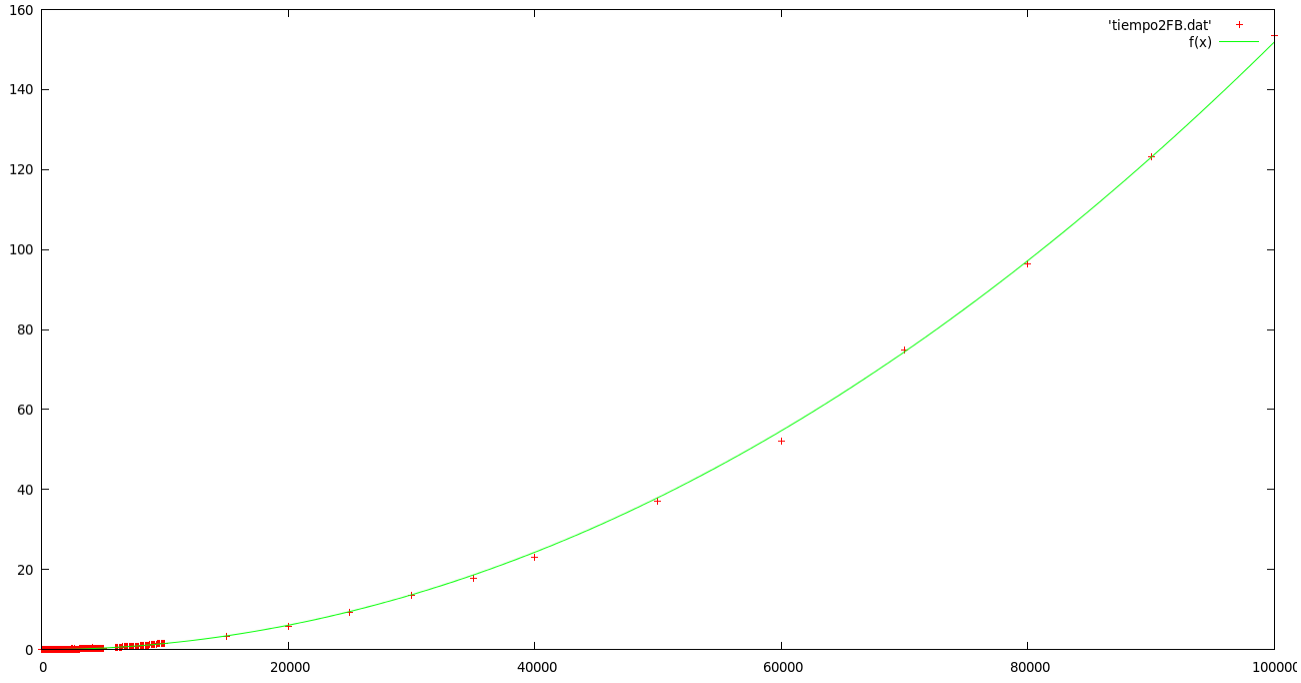
```
1 9.02e-07
11 2.6755e-05
21 1.1246e-05
31 3.0878e-05
41 3.3345e-05
51 4.8544e-05
61 0.000105918
71 8.8695e-05
81 0.00020118
91 0.000142769
101 0.000173627
111 0.000290643
121 0.00039679
131 0.000287308
141 0.000329216
151 0.000396273
161 0.000430939
171 0.000482049
181 0.000540545
191 0.000601983
201 0.00108412
```

Divide y vencerás:

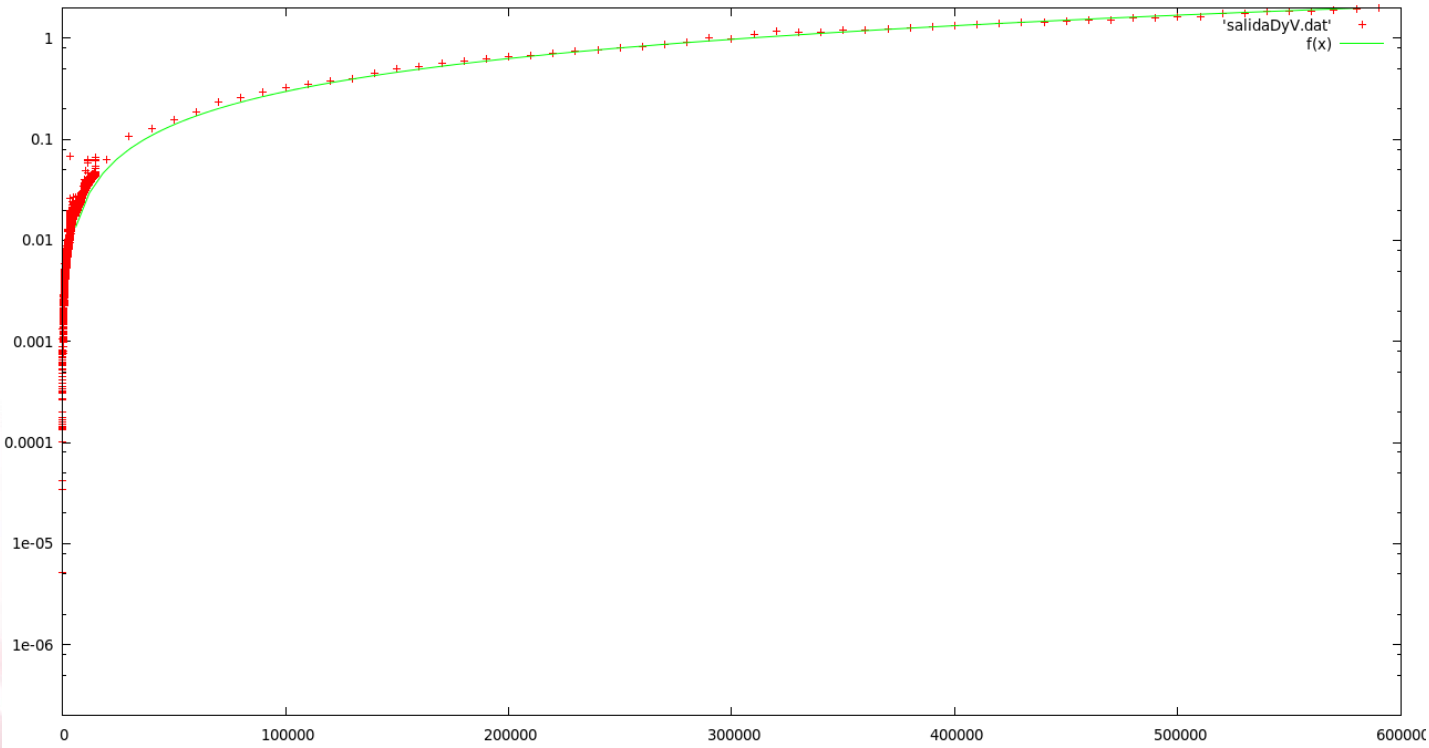
```
1 5.212e-06
6 3.4015e-05
11 4.2077e-05
16 0.000102708
21 0.000100983
26 0.000102668
31 0.000134161
36 0.000166262
41 0.000144766
46 0.000136594
51 0.000140428
56 0.000150718
61 0.00016176
66 0.000178692
71 0.000199384
76 0.000266315
81 0.00027475
86 0.000340397
91 0.000274479
96 0.000273719
101 0.000326123
```

➔ *Gráficas:*

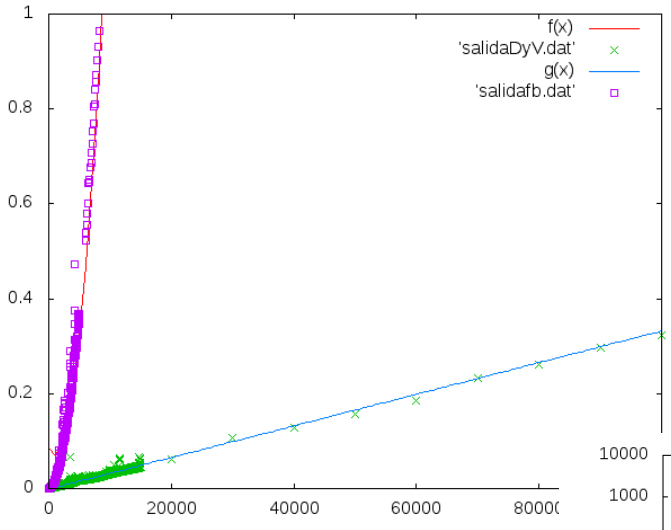
- Fuerza bruta



- Divide y vencerás:

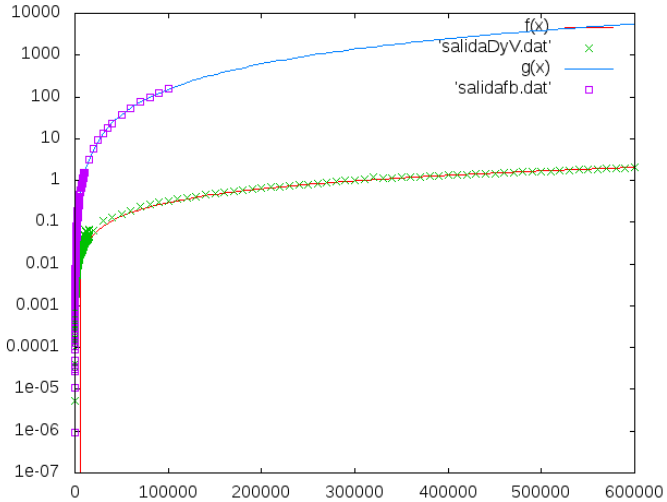


- Comparativa:



<- Sin escala logarítmica

Con escala logarítmica ->



Como podemos observar, gracias al análisis empírico de la eficiencia de ambos algoritmos, podemos reafirmar lo ya demostrado en el análisis teórico. El algoritmo divide y vencerás, el cual tiene una eficiencia de O(nlog(n)), es bastante más eficiente que el algoritmo de fuerza bruta, cuya eficiencia es O(n2).

3. Cada elemento en su posición

3.1. Introducción al problema

Este ejercicio tiene 2 apartados:

1. Tenemos un vector inicial de n enteros, todos distintos y ordenados de forma no decreciente.
2. Al igual que en el primer apartado, tenemos un vector de enteros ordenado de forma no decreciente, pero en este caso el vector puede tener elementos repetidos.

Para ambos casos, el problema es determinar si existe un índice i tal que V[i] = i, y en caso de que exista, encontrarlo.

3.2. Descripción de la solución

El algoritmo utilizado es una versión modificada de la búsqueda binaria. Sirve para encontrar, en un vector de enteros no decrecientes, el entero que sea igual a su índice. Consiste en lo siguiente:

El algoritmo elige el elemento central (como en la búsqueda binaria), y comprueba si coincide con su índice. Si coincide, el proceso ha finalizado. En caso contrario, en el peor de los casos, se llama recursivamente dos veces, pasando como parámetros la parte izquierda del elemento central la primera vez, y la parte derecha del elemento central la segunda. Sin embargo, cuando hay elementos repetidos, puede ocurrir que no necesitemos comprobar una de las dos (o las dos) partes.

Las comprobaciones son las siguientes. Por cada mitad:

> Si el entero de la posición de inicio de la mitad es mayor que el índice del elemento final, es imposible que algún entero coincida con su índice. Esto ocurre porque los elementos están ordenados en orden no decreciente. Pasa igual si el elemento de la posición final es menor que el índice de la posición inicial.

Si alguna mitad cumple las características mencionadas, el algoritmo omite esa parte, como en una búsqueda binaria, ya que nunca va a encontrar un elemento que coincida con su índice ahí.

Cuando recursivamente se llega a mitades de vectores de sólo dos índices (0,1; 2,3; etc) el algoritmo no vuelve a dividir. Es más eficiente comprobar si alguno de los dos enteros coincide con su índice.

Cuantos más elementos repetidos haya, más eficiente es el algoritmo, puesto que más se acerca a una búsqueda binaria en cuanto funcionamiento y, por tanto, eficiencia.

3.3. Código

```
bool en_su_posicion( const vector<int> &v, int ini, int fin, int &pos )
{
  int medio = (ini+fin)/2;
  if( v[medio] == medio ){
    pos = medio;
    return true;
  }

  if( ini - fin > 1 || ini - fin < -1 ){

    if( ( v[ini] > medio-1 || v[medio-1] < ini ) && ( v[medio+1] > fin || v[fin] < medio+1 ) ) return false;
    if( v[ini] > medio-1 || v[medio-1] < ini ) return en_su_posicion(v,medio+1,fin,pos);
    if( v[medio+1] > fin || v[fin] < medio+1 ) return en_su_posicion(v,ini,medio-1,pos);
    return ( en_su_posicion(v,medio+1,fin,pos) || en_su_posicion(v,ini,medio-1,pos) );

  }
  if( ini - fin == 1 || ini - fin == -1 ){
    return( v[ini] == ini || v[fin] == fin );
  }
  return false;
}
```

3.4. Algoritmo de fuerza bruta

```
bool en_su_posicion(const vector <int> &arr){


    for(int i=0; i<arr.size(); i++){
        if(arr[i]==i){
            return true;
        }
    }

    return false;

}
```

3.5. Análisis de eficiencia

Análisis de eficiencia teórica

➔ *Divide y vencerás:*

Cálculo de la eficiencia del algoritmo basado en divide y vencerás (con elementos repetidos) usado para el ejercicio 3.

Vamos a considerar que evaluamos un vector de n elementos, siendo n potencia de 2 (2^k). En el peor caso:

$$T(n) \begin{cases} c_1 & si\ n=1 \\ 2T(n/2)+c_2 & si\ n>1,\ n=2^k \end{cases}$$

Para saber la eficiencia, vamos a usar expansión

$T(n) = 2T(n/2) + c_2$

$T(n/2) = 2T(n/4) + c_2$

Es decir:

$T(n) = 4T(n/4) + 2c_2$

o

$T(n) = 8T(n/8) + 3c_2$

En general:

$T(n) = 2^iT(n/2^i) + ic_2$, siendo i el número de llamadas recursivas.

Cuando $i = k$, quiere decir que no habrá más llamadas recursivas, es decir, en la parte derecha hay $T(1)$.

La fórmula quedaría:

$T(n) = 2^kT(1) + kc_2$

Como $2^k = n$, $k = log_2(n)$.

La fórmula quedaría:

$T(n) = nT(1) + c_2log_2(n)$

Podemos tomar $T(1) = c_1$

Por lo que la fórmula quedaría $T(n) = c_1n+c_2log_2(n)$, por lo que la eficiencia sería $O(nlog(n))$.

Aunque en la práctica, cuando hay muchos elementos repetidos, la eficiencia se acerca a $O(log(n))$.

➔ *Fuerza* bruta

Cálculo de la eficiencia del algoritmo basado en fuerza bruta usado para el ejercicio 3. Con este algoritmo es indiferente si hay elementos repetidos.

Vamos a considerar que evaluamos un vector de n elementos. El algoritmo consiste en un for desde 0 hasta n, por lo que la eficiencia sería de $O(n)$.

Análisis eficiencia empírica/híbrida

➔ *Datos:*

Fuerza bruta:

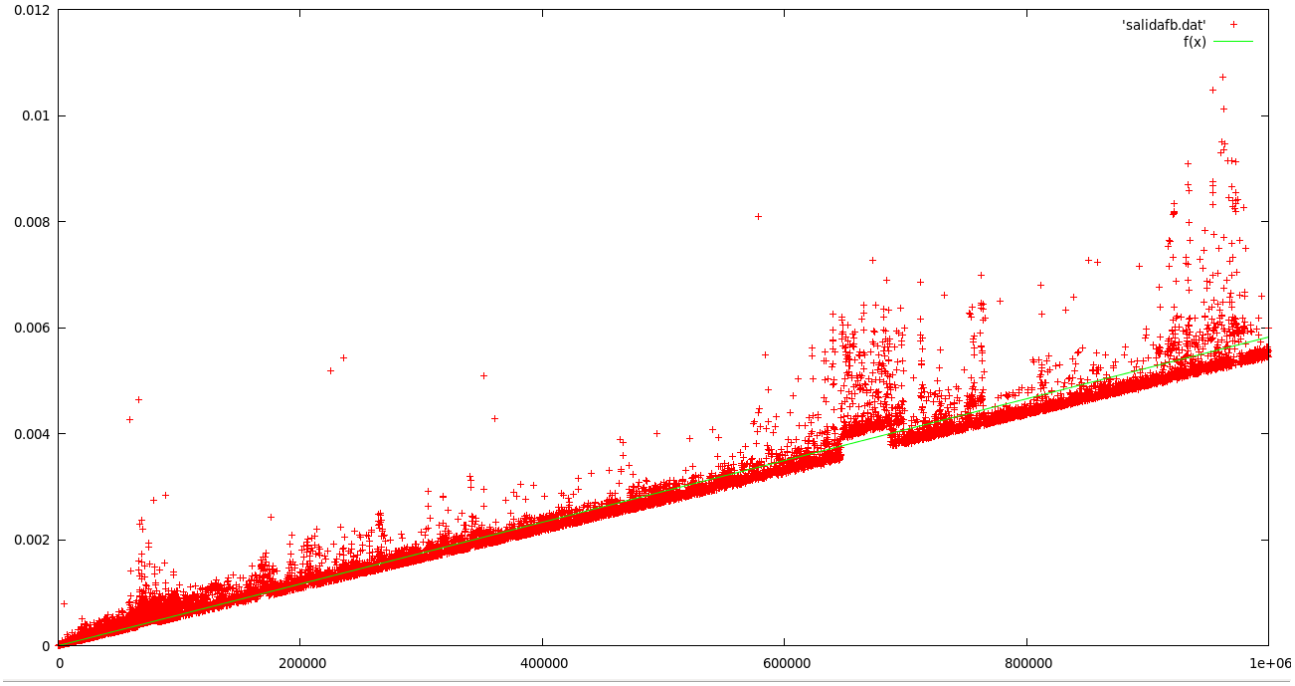
```
3 3.73e-07
13 4.25e-07
23 3.21e-07
33 3.63e-07
43 6.14e-07
53 6.6e-07
63 9.02e-07
73 6.36e-07
83 9.77e-07
93 7.37e-07
103 8.07e-07
113 1.199e-06
123 1.235e-06
133 9.99e-07
143 9.27e-07
153 1.122e-06
163 1.148e-06
173 1.113e-06
183 1.308e-06
193 1.353e-06
203 1.736e-06
```

Divide y vencerás:

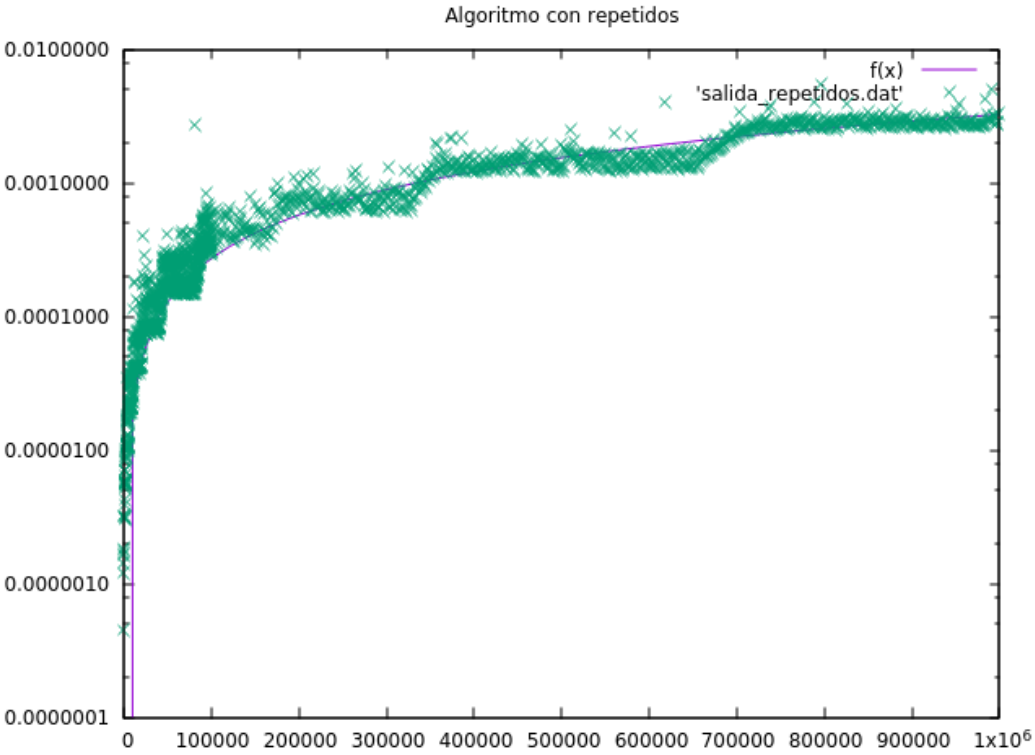
```
1 4.5e-07
102 1.215e-06
202 1.629e-06
302 1.411e-06
402 1.663e-06
502 1.76e-06
602 1.843e-06
702 3.228e-06
802 3.201e-06
902 3.18e-06
1002 3.114e-06
1102 3.021e-06
1202 3.169e-06
1302 3.93e-06
1402 5.151e-06
1502 5.476e-06
1602 8.35e-06
1702 5.631e-06
1802 7.112e-06
1902 5.585e-06
2002 5.634e-06
```

➔ *Gráficas*

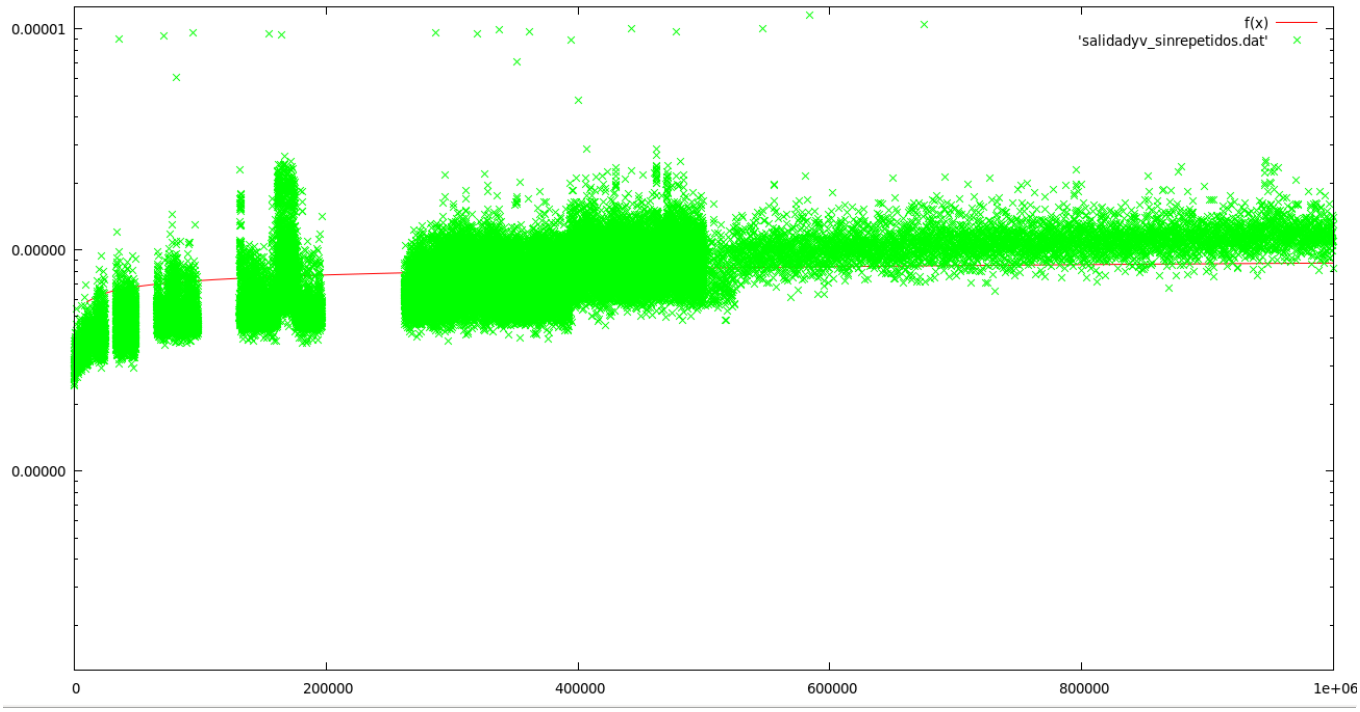
- Fuerza bruta:
-



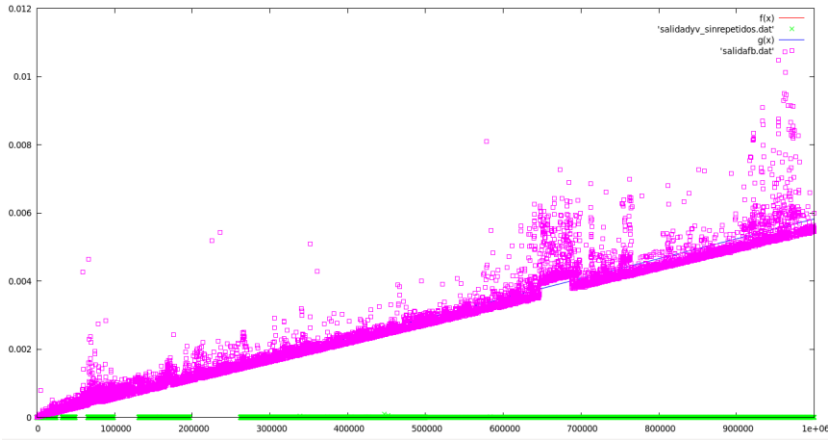
- Divide y vencerás(con repetidos):



- Divide y vencerás (sin repetidos):

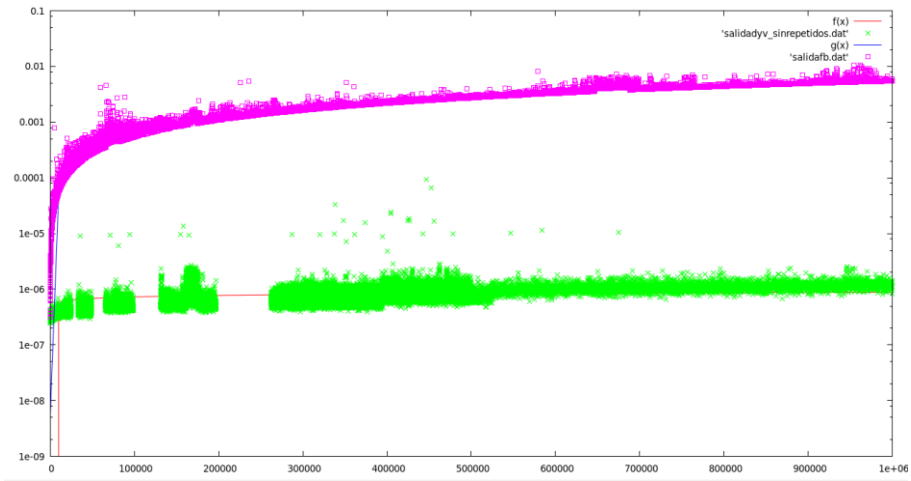


- Comparativas sin repetidos:



<- Sin escala logarítmica

Con escala logarítmica:

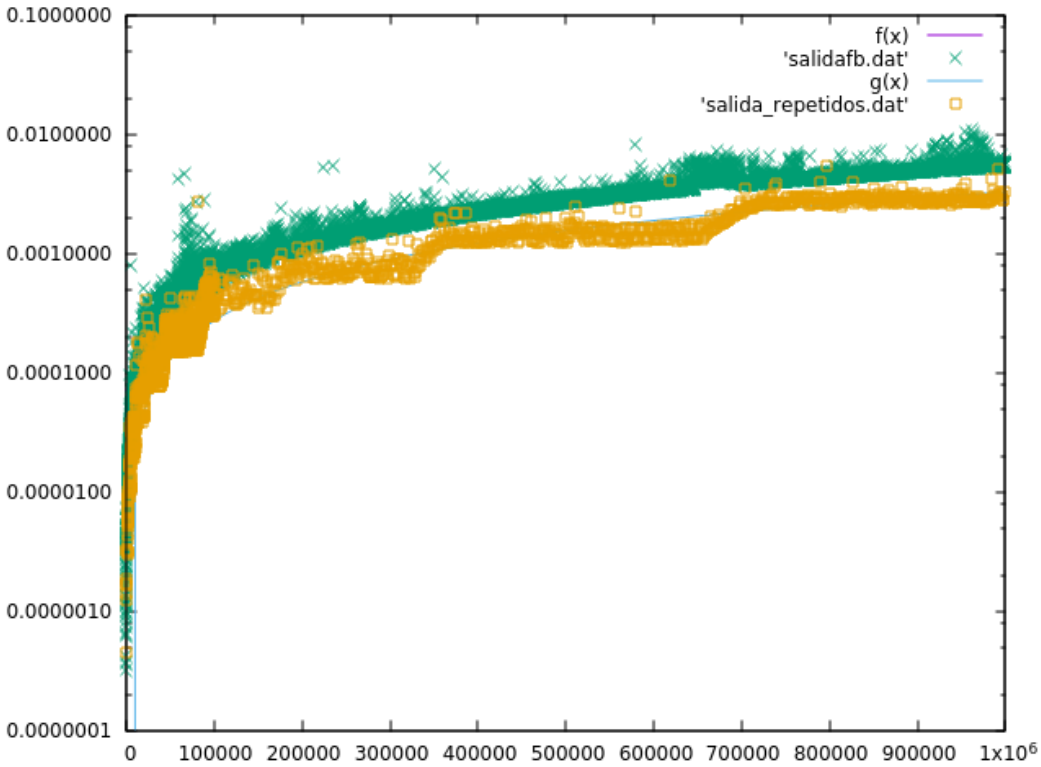


En este ejercicio, dependiendo del apartado abordado (con o sin elementos repetidos), obtendremos una eficiencia u otra:

- Si no tenemos elementos repetidos, la eficiencia del algoritmo divide y vencerás es de O(log(n))-
- Si tenemos elementos repetidos, la eficiencia del divide y vencerás teóricamente empeora a O(nlog(n)).
- La eficiencia del algoritmo de fuerza bruta, tengamos o no elementos repetidos es O(n).

Por lo tanto, el algoritmo divide y vencerás en este caso solo será más eficiente (teóricamente) que el algoritmo de fuerza bruta en el caso de que no tengamos elementos repetidos.

Pero si observamos la siguiente gráfica en la que se muestran los datos recogidos de los algoritmos fuerza bruta y divide y vencerás con elementos repetidos:



Podemos ver que nuestro algoritmo divide y vencerás consigue ser un poco más rápido que el de fuerza bruta, que tiene una eficiencia de O(n), es decir pasamos de una eficiencia de O(nlog(n)) a una cercana a O(log(n)), lo que es una ganancia de eficiencia a tener en cuenta. Esto se debe a que nuestro algoritmo combina el algoritmo mergesort y el de búsqueda binaria, de manera que en ocasiones solo hará una llamada recursiva(búsqueda binaria) en vez de dos(mergesort) descartando de esta manera la mitad del vector si detecta que es despreciable para hallar la solución. De esta manera mientras mas elementos repetidos haya, mas se acercará a O(log(n)).

4. Mezcla de k vectores ordenados.

4.1. Introducción al problema

En este caso no disponemos inicialmente de un vector de enteros, sino de un conjunto de vectores de enteros, ordenados todos ellos de menos a mayor, que lo representaremos como un vector de vectores. El problema a resolver en este ejercicio es, a partir del conjunto de vectores dado, proporcionar como salida un único vector en el que se encuentren todos los elementos de los vectores del conjunto inicialmente dados, y también ordenados de menor a mayor.

4.2. Descripción de la solución

En esta solución volvemos a hacer uso como base de esta un algoritmo mergesort, pero aquí no va a subdividir un vector en otros dos vectores, sino que dividirá el conjunto de vectores proporcionando así otros dos subconjuntos de vectores. Esta subdivisión se llevará a cabo mediante la llamada recursiva a la función mezclarKvectores, que se llamará a íi misma una y otra vez hasta que el tamaño del conjunto sea de dos vectores. Se creará un árbol de subconjuntos de vectores. Una vez llegados al punto en el que los conjuntos son de 2 vectores, se dejará de llamar recursivamente a la anterior función y pasará a ejecutarse la función mezclar2vectores, a la que se le pasarán como argumentos dos vectores y esta devolverá la mezcla de forma ordenada de ambos vectores en un solo vector. La primera llamada a esta función se hará cunado los subconjuntos de vectores sean de tamaño dos, por lo tanto en esta llamada se les pasará estos dos vectores como argumentos, mientras que en las siguientes llamadas se les pasará los retornos de las dos funciones recursivas que la preceden, las cuales dan como salida los vectores ordenados de cada lado del árbol, hasta que tengamos el resultado final, que será un vector ordenado con todos los elementos de los vectores del conjunto de vectores inicial.

4.3. Código

Función mezclarKvectores

```
vector<int> mezclarKvectores(vector<vector<int>> v){

  vector<vector<int>> v1;
  vector<vector<int>> v2;
  vector<int> r1;
  vector<int> r2;
  vector<int> r;

  if(v.size() > 2){

    for(int i = 0; i < v.size(); i++){
      if(i < v.size()/2){
        v1.push_back(v[i]);
      }
      else{
        v2.push_back(v[i]);
      }
    }

    r1 = mezclarKvectores(v1);
    r2 = mezclarKvectores(v2);
  }
  else{
    r1 = v[0];
    r2 = v[1];
  }

  mezclar2vectores(r1, r2, r);

  return r;
}
```

Función mezclar2vectores

```
void mezclar2vectores(const vector<int> &arreglo1,const vector<int> &arreglo2, vector<int> &arreglo3)
{
    int x1=0, x2=0;

    while (x1<arreglo1.size() && x2<arreglo2.size()) {
        if (arreglo1[x1]<arreglo2[x2]) {
            arreglo3.push_back(arreglo1[x1]);
            x1++;
        }
        else if( arreglo1[x1]>arreglo2[x2] ) {
            arreglo3.push_back(arreglo2[x2]);
            x2++;
        }
        else{
          arreglo3.push_back(arreglo1[x1]);
          x1++;
          x2++;
        }
    }
    while (x1<arreglo1.size()) {
        arreglo3.push_back(arreglo1[x1]);
        x1++;
    }
    while (x2<arreglo2.size()) {
        arreglo3.push_back(arreglo2[x2]);
        x2++;
    }
}
```

4.4. Algoritmo de fuerza bruta

```
vector<int> mezclarVectores(const vector<int> &arreglo1 , const vector<int> &arreglo2){
  int x1=0, x2=0;
  vector<int> arreglo3;

  while (x1<arreglo1.size() && x2<arreglo2.size()) {
      if (arreglo1[x1]<arreglo2[x2]) {
          arreglo3.push_back(arreglo1[x1]);
          x1++;
      }
      else if( arreglo1[x1]>arreglo2[x2] ) {
          arreglo3.push_back(arreglo2[x2]);
          x2++;
      }
      else{
        arreglo3.push_back(arreglo1[x1]);
        x1++;
        x2++;
      }
  }
  while (x1<arreglo1.size()) {
      arreglo3.push_back(arreglo1[x1]);
      x1++;
  }
  while (x2<arreglo2.size()) {
      arreglo3.push_back(arreglo2[x2]);
      x2++;
  }
  return arreglo3;
}
```

4.5. Análisis de eficiencia

Análisis de eficiencia teórica

➔ *Divide y vencerás:*

Cálculo de eficiencia para algoritmo para mezclar k vectores de tamaño n, usando divide y vencerás y recursividad.

Teóricamente, vamos a suponer que k es una potencia de 2. $k = 2^m$, por lo que mezclaríamos las $k/2$ parejas de vectores (1-2,3-4,5-6...) de longitud n. Una vez mezcladas, se mezclarían las $k/4$ parejas de vectores, de longitud $2n$: (1-2 con 3-4, 5-6 con 7-8...), y así sucesivamente.

Para calcular la eficiencia vamos a usar expansión.

Tenemos que $T(k) = 2T(k/2) + c_2nk$.

$2T(k/2)$ por que llamamos recursivamente a la función dos veces, pasándole $k/2$ como argumento.

c_2nk por que la acción de combinar dos vectores es lineal.

Vamos a calcular $T(k/2)$.

$T(k/2) = 2T(k/4) + c_2n(k/2)$.

Entonces, $T(k) = 4T(k/4) + c_22nk$.

También $T(k) = 8T(k/8) + c_23nk$.

Se puede concluir que $T(k) = 2^iT(k/2^i) + c_2ink$.

Al llegar al final de la recurrencia, va a haber un momento en el cual en el lado derecho tengamos $T(1)$.

Ésto quiere decir, que cuando $i = m$, tendremos que $T(k) = 2^mT(1)+c_2mnk$.

Sabiendo que $2^m = k$, y $m = log_2k$, y tomando $T(1)$ como c_1, tenemos lo siguiente:

$T(k) = c_1k + c_2nklog_2k$.

$nklog_2k$ hace despreciable k, por lo que tenemos que la eficiencia es $O(log(k)*nk)$.

➔ *Fuerza* bruta

Cálculo de eficiencia para algoritmo de fuerza bruta para mezclar k vectores de tamaño n, ordenados.

Teniendo varios vectores para ordenar, los dos primeros vectores se tardaría un tiempo de $n + n$.

Para mezclar el resultado con el tercero, $2n + n$. Con el cuarto, $3n + n$. Sucesivamente, resultaría un tiempo de $(k - 1)n + n$.

Por tanto, el tiempo total de ejecución es:

$$\sum_{i=1}^{k-1}(in+n) = n\sum_{i=1}^{k-1} i + n\sum_{i=1}^{k-1} 1 = n\frac{k(k-1)}{2} + (k-1)n = n\frac{(k-1)(k+2)}{2}$$

Es decir, $O(nk^2)$

Análisis de eficiencia empírica/hibrida

➔ *Datos:*

Fuerza bruta(elementos constantes):

```
1 2.71e-07
11 5.6934e-05
21 0.000120076
31 0.000210069
41 0.000675851
51 0.000446612
61 0.000402946
71 0.000757243
81 0.000650867
91 0.00090813
101 0.000965279
111 0.00118149
121 0.00177528
131 0.0023207
141 0.00265313
151 0.00268405
161 0.00304456
171 0.00358932
181 0.00344063
191 0.00363126
201 0.00385282
```

Divide y vencerás(elementos constantes):

```
2 5.15e-06
4 1.479e-05
8 3.4489e-05
16 8.1346e-05
32 0.000206413
64 0.000533697
128 0.000999034
256 0.00199657
512 0.00461903
1024 0.00957444
2048 0.0199704
4096 0.0419554
8192 0.0890255
16384 0.189187
32768 0.400057
65536 0.897492
131072 1.80166
262144 3.79797
```

Fuerza bruta(vectores constantes):

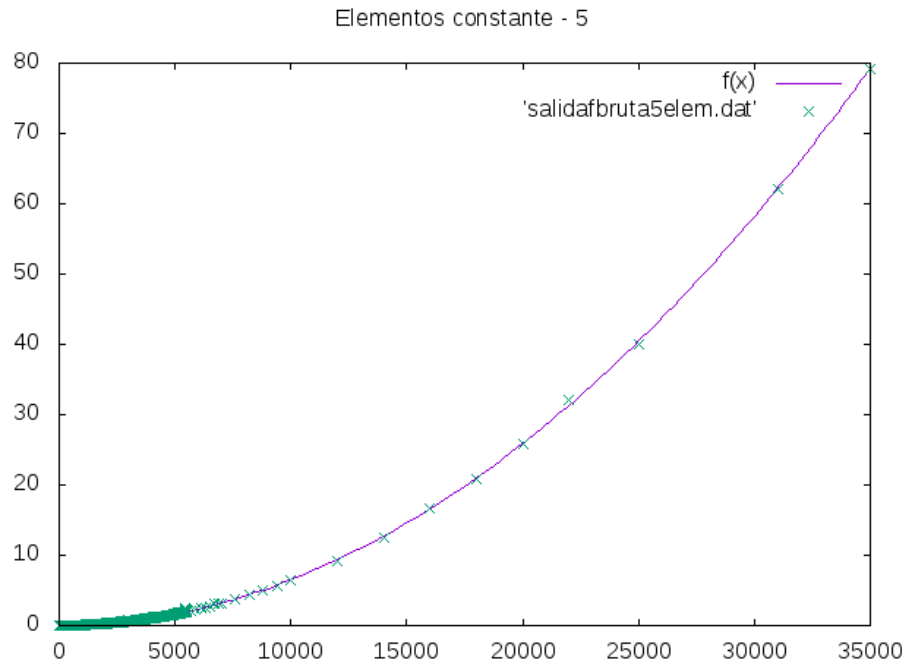
```
1 1.3977e-05
11 4.1536e-05
21 5.6492e-05
31 3.402e-05
41 5.5445e-05
51 6.2027e-05
61 6.7477e-05
71 5.8058e-05
81 4.567e-05
91 5.055e-05
101 4.7814e-05
111 8.4807e-05
121 6.2223e-05
131 6.7435e-05
141 7.0803e-05
151 7.3375e-05
161 6.9908e-05
171 8.1104e-05
181 8.7103e-05
191 0.000119191
201 8.6096e-05
```

Divide y vencerás(vectores constantes):

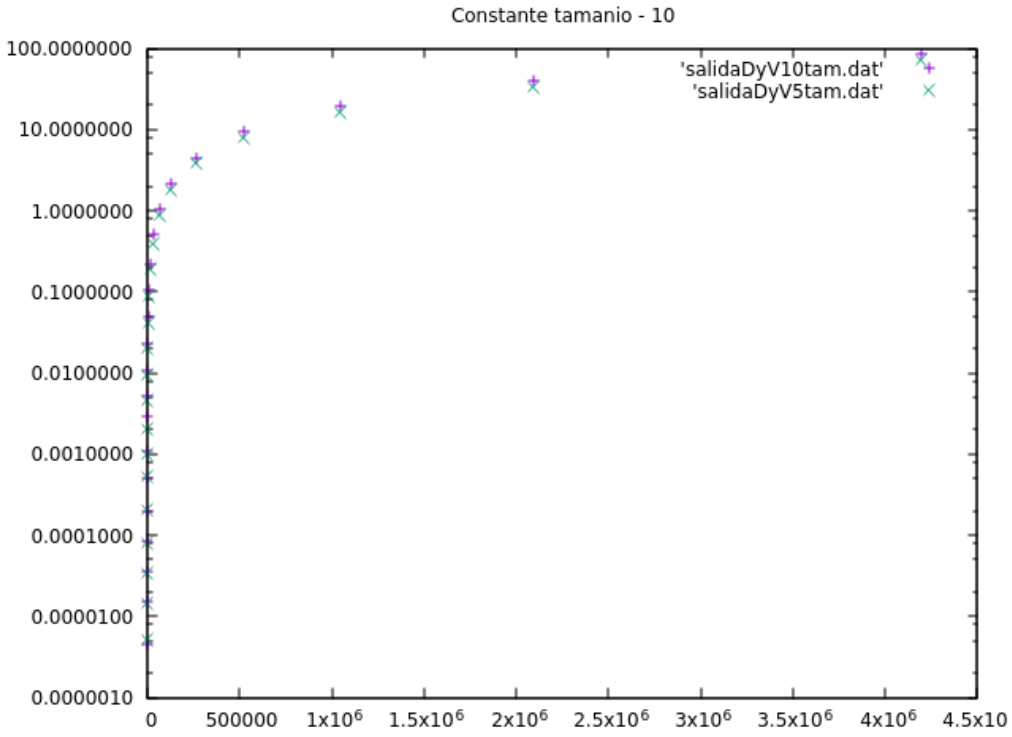
```
2 1.2666e-05
4 2.8375e-05
8 1.4934e-05
16 2.5427e-05
32 2.3196e-05
64 3.9176e-05
128 4.5846e-05
256 7.3831e-05
512 0.00012273
1024 0.000220669
2048 0.000465554
4096 0.000841314
8192 0.00194402
16384 0.00350648
32768 0.007319
65536 0.0141878
131072 0.0290315
262144 0.0582066
```

➔ *Gráficas:*

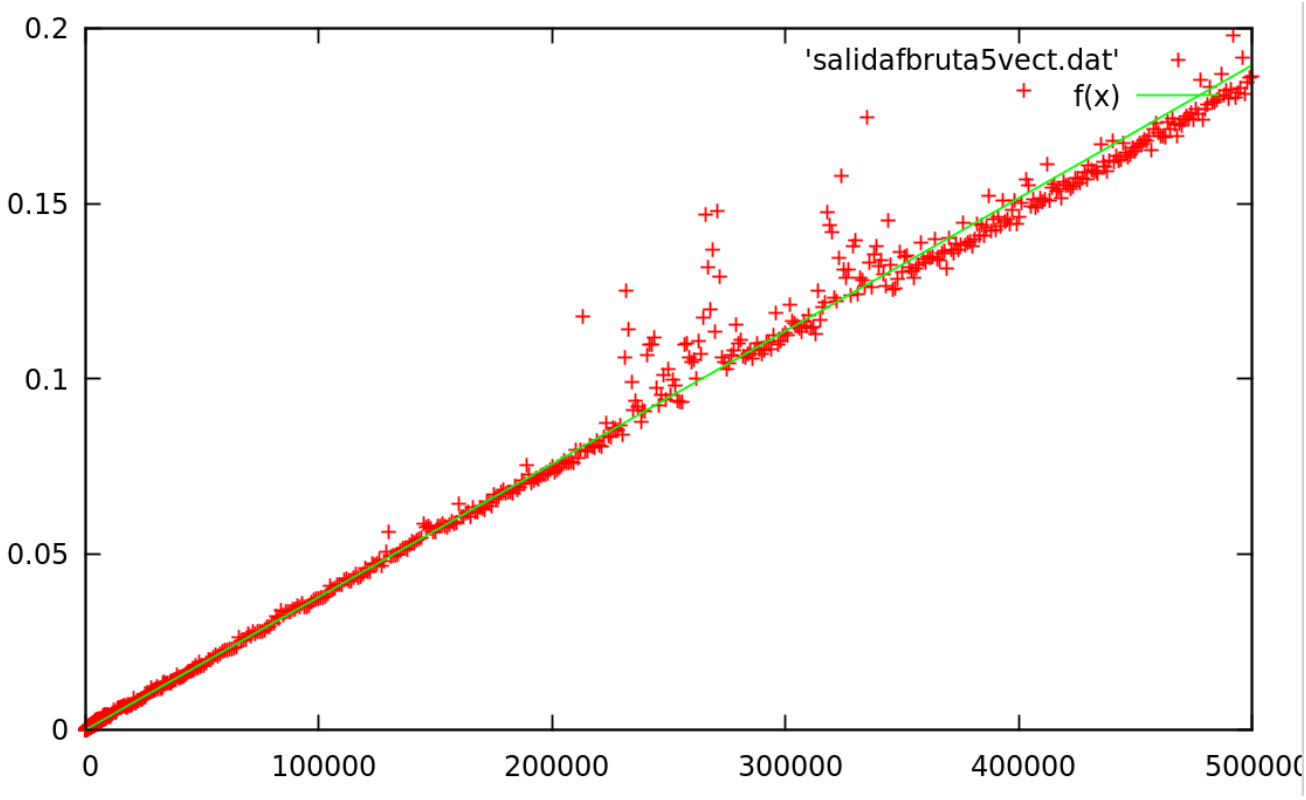
Fuerza bruta(elementos constantes):



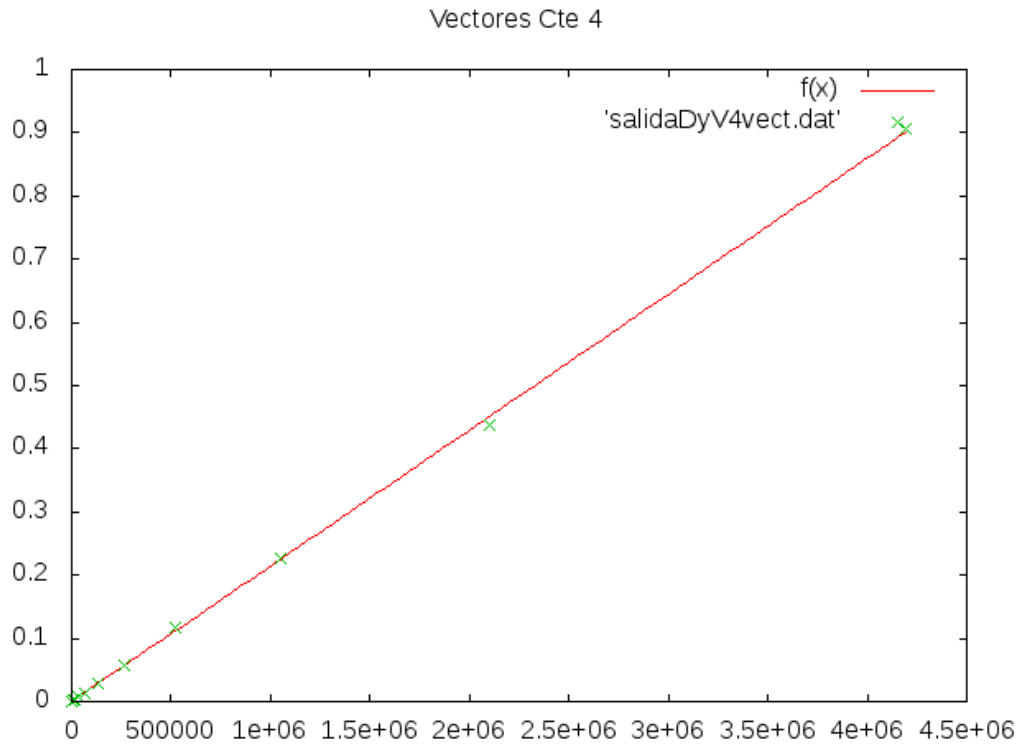
Divide y vencerás(elementos constantes):



Fuerza bruta(vectores constantes):



Divide y vencerás(vectores constantes):



Como podemos ver en este problema, tenemos dos variables cuyo tamaño puede ser variable, y de que sean constantes o no dependerá la eficiencia del algoritmo.

- Fuerza bruta: Como hemos visto con la eficiencia teórica, y hemos reafirmado con el estudio empírico, la eficiencia de este algoritmo es **O(nk^2)**, siendo n el número de elementos que tiene cada vector, y siendo k el número de vectores del conjunto inicial.
    - Si tomamos el número de elementos (n)constante, la eficiencia sería **O(k^2)**, estaríamos ante un algoritmo cuadrático como podemos ver en las gráficas.
    - Si tomamos como constante el número de vectores del conjunto inicial (k), tendríamos como eficiencia **O(n)**, con lo cual el algoritmo sería lineal.
- Divide y vencerás: Al igual que con el algoritmo de fuerza bruta, teóricamente hemos demostrado que la eficiencia de este algoritmo es **O(nklog(k))**, y empíricamente lo podemos reafirmar.
    - Si tomamos el número de elementos (n) constante, la eficiencia de este algoritmo sería **O(klog(k))**.
    - Si tomamos como constante el número de vectores (k), la eficiencia sería **O(n)**.

Como conclusión podemos decir que para este problema, será conveniente elegir una metodología de divide y vencerás para implementar una solución en el caso de que tengamos que aplicarlo a un conjunto en el que el número de vectores no es constante y el tamaño de estos si lo es. En caso contrario la eficiencia del algoritmo divide y vencerás es la misma que la del algoritmo de fuerza bruta, por lo que no tendría sentido usar divide y vencerás para implementar la solución.

5. Serie unimodal de números

5.1. Introducción al problema

En este caso tenemos un vector inicial de enteros en el que sus elementos están ordenados de forma creciente hasta un elemento p del mismo, a partir del cual estos están ordenados de forma decreciente.

Como solución a este problema nuestro algoritmo tiene que encontrar dicho punto p y devolverlo como salida.

5.2. Descripción de la solución

La solución aportada tiene como base un algoritmo de búsqueda binaria utilizando la técnica de divide y vencerás. El vector inicial será subdividido de forma recursiva hasta que este tenga un tamaño igual a 3, pero en este caso, a diferencia del algoritmo mergesort, no haremos dos llamadas recursivas a la función, sino que haremos tan solo una pero con la mitad del vector en la que sabemos que está la solución, la otra parte es despreciable ya que podemos saber que en ella no se encuentra el punto máximo.

¿Cómo sabemos en qué mitad del vector vamos a encontrar el punto que buscamos? Para saberlo, nos basta con fijarnos en el elemento anterior y en el posterior al elemento que se encuentra en la mitad. De esta forma distinguiremos 3 casos a la hora de quedarnos con una u otra mitad del vector:

- Los 3 elementos en los que nos fijamos se encuentran ordenados de forma **creciente**: Si esto se cumple, podemos afirmar que la solución se encuentra en la mitad derecha del vector.

- Estos elementos están ordenados de forma **decreciente**: Podemos afirmar que la solución se encuentra en la mitad izquierda del vector.
- De estos tres elementos, el elemento del medio es mayor que los otros dos: En este caso hemos dado con la solución y la función acabaría retornando esta.

Estas comprobaciones se realizarán antes de hacer cada subdivisión, hasta que el vector sea de tamaño 3, en cuyo caso se cumpliría siempre la tercera condición y la solución siempre sería la correcta(no tiene por que llegar a tamaño 3, puede darse el caso en el que en una de las comprobaciones se de con el elemento buscado, en cuyo caso la función retornará sin la necesidad de llegar hasta tamaño 3).

5.3. Código

```
int unimodal(const vector<int> &v,int ini, int fin){

  int p = 0;
  vector<int> v1;
  int mitad;
  mitad = (ini+fin)/2;

  if((fin-ini) > 2){
    if( v[mitad-1] < v[mitad] && v[mitad+1] > v[mitad] ){
      p = unimodal(v,mitad+1,fin);
    }
    else if( v[mitad-1] > v[mitad] && v[mitad+1] < v[mitad] ){
      p = unimodal(v,ini,mitad-1);
    }
    else if(v[mitad-1] < v[mitad] && v[mitad+1] < v[mitad] ){
      p = v[mitad];
    }
  }
  else if( v[mitad-1] < v[mitad] && v[mitad+1] < v[mitad] ){
    p = v[mitad];
  }

  return p;
}
```

5.4. Algoritmo de fuerza bruta

```
int unimodalFB(const vector<int> &v){
  for(int i = 1; i < v.size()-1; i++){
    if(v[i-1] < v[i] && v[i+1] < v[i]){
      return v[i];
    }
  }
}
```

5.5. Análisis de eficiencia

Análisis de eficiencia teórica

➔ *Divide y vencerás*

Cálculo de la eficiencia del algoritmo basado en fuerza bruta usado para el ejercicio 3.

Vamos a considerar que evaluamos un vector de n elementos, siendo n potencia de 2 (2^k). En el peor caso:

$T(n)\begin{cases} c_1 & si\ n=1 \\ T(n/2)+c_2 & si\ n>1,\ n=2^k \end{cases}$

Para saber la eficiencia, vamos a usar expansión

$T(n) = T(n/2) + c_2$

$T(n/2) = T(n/4) + c_2$

Es decir:

$T(n) = T(n/4) + 2c_2$

o

$T(n) = T(n/8) + 3c_2$

En general:

$T(n) = T(n/2^i) + ic_2$, siendo i el número de llamadas recursivas.

Cuando $i = k$, quiere decir que no habrá más llamadas recursivas, es decir, en la parte derecha hay $T(1)$.

La fórmula quedaría:

$T(n) = T(1) + kc_2$

Como $2^k = n$, $k = log_2(n)$.

La fórmula quedaría:

$T(n) = T(1) + c_2 log_2(n)$

Podemos tomar $T(1) = c_1$

Por lo que la fórmula quedaría $T(n) = c_1 + c_2 log_2(n)$, por lo que la eficiencia sería $O(log(n))$.

➔ *Fuerza bruta*

Cálculo de la eficiencia del algoritmo basado en fuerza bruta usado para el ejercicio 5. Vamos a considerar que evaluamos un vector de n elementos. El algoritmo consiste en un *for* desde 0 hasta n, por lo que la eficiencia sería de $O(n)$.

Análisis de eficiencia empírica/híbrida

➔ *Datos:*

Fuerza bruta:

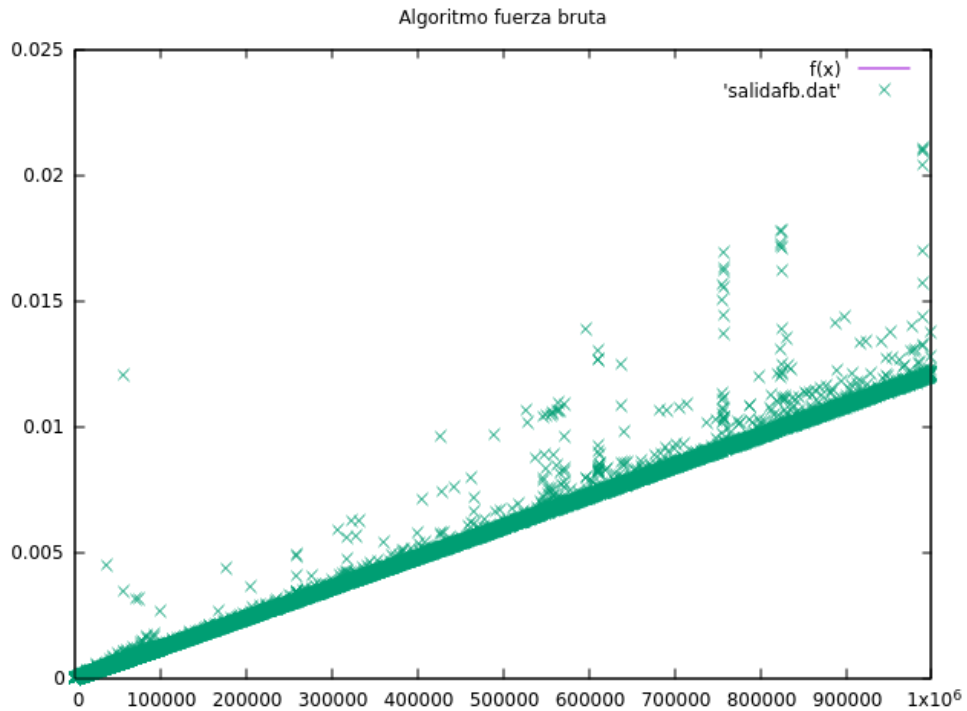
1	2.1e-07
11	4.4e-07
21	7.5e-07
31	9.05e-07
41	6.53e-07
51	7.7e-07
61	9.55e-07
71	1.008e-06
81	1.428e-06
91	1.223e-06
101	1.377e-06
111	1.505e-06
121	2.833e-06
131	1.93e-06
141	3.267e-06
151	3.36e-06
161	2.308e-06
171	2.464e-06
181	2.364e-06
191	2.496e-06

Divide y vencerás:

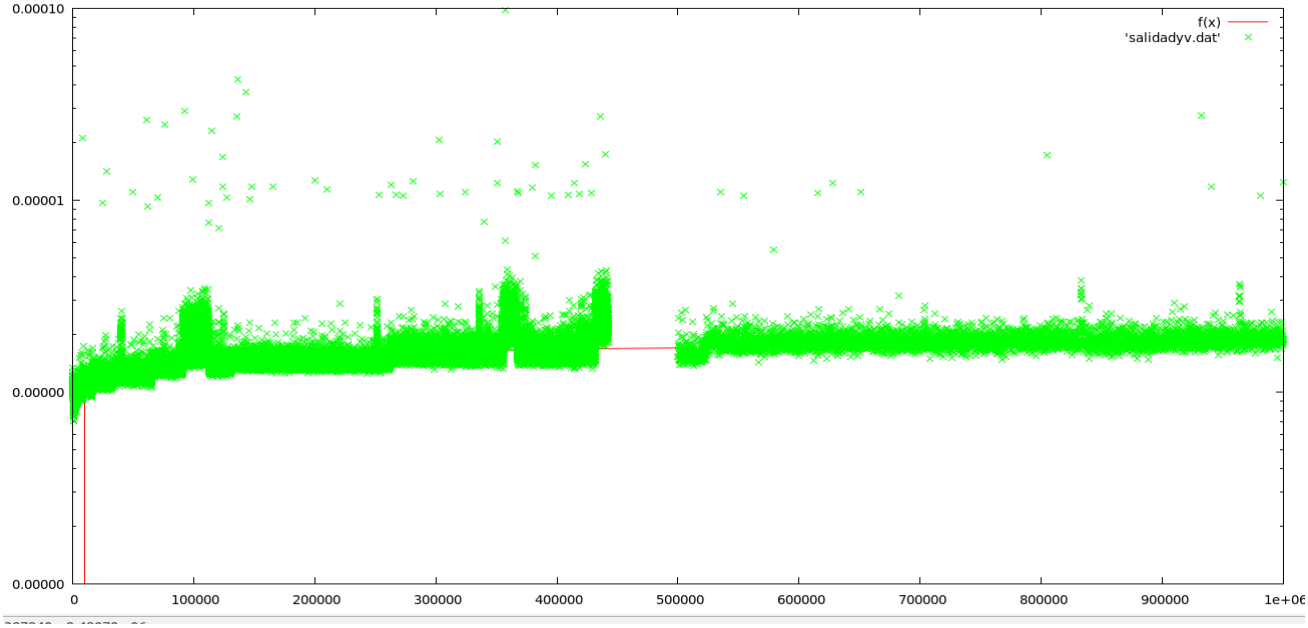
1	9.45e-07
11	1.226e-06
21	1.108e-06
31	1.219e-06
41	1.273e-06
51	1.167e-06
61	1.056e-06
71	1.148e-06
81	1.218e-06
91	1.211e-06
101	1.147e-06
111	1.031e-06
121	1.016e-06
131	1.212e-06
141	1.341e-06
151	1.106e-06
161	1.127e-06
171	1.124e-06
181	1.133e-06
191	1.13e-06

➔ *Gráficas*

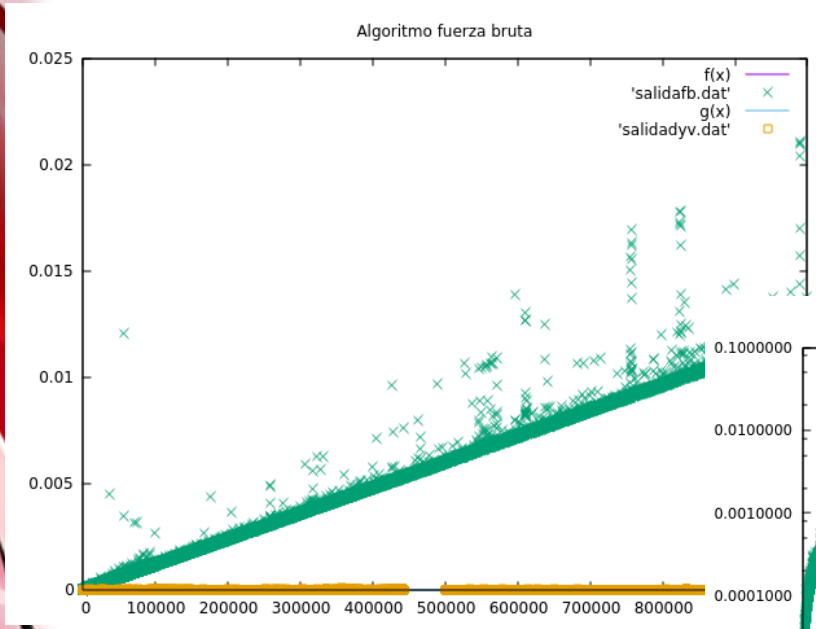
Fuerza bruta:



Divide y vencerás:

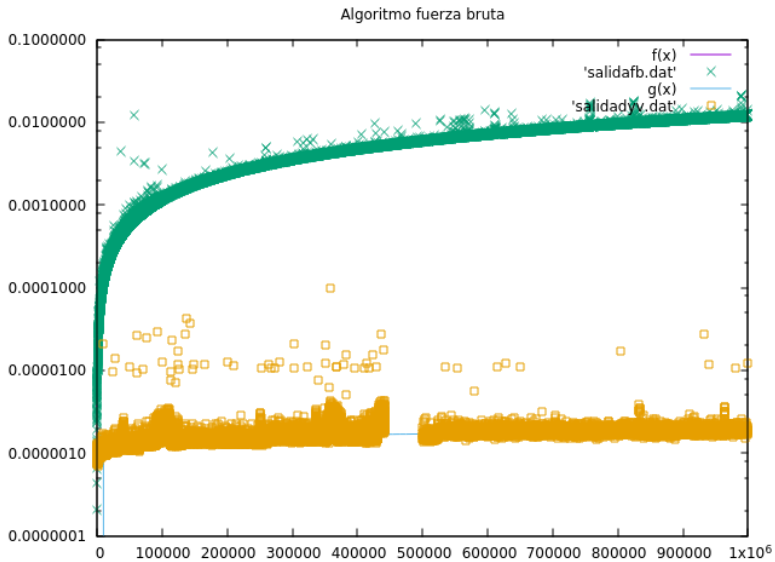


Comparativa:



<- Sin escala logarítmica

Con escala logarítmica ->



Como hemos deducido teóricamente, y demostrado empíricamente, el algoritmo de fuerza bruta para la resolución de este problema tiene una eficiencia de **O(n)**, mientras que el algoritmo divide y vencerás es **O(log(n)).** Por lo tanto, el algoritmo divide y vencerás mejora considerablemente la eficiencia de la solución.