

*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*

*PRÁCTICA 1 ALGORÍTMICA*

*DIVIDE Y VENCERÁS*

*2017*

*DIVIDE Y VENCERÁS*

Contenido

[Introducción 4](#_Toc478504182)

[1. Comparación de preferencias 4](#_Toc478504183)

[1.1. Introducción al problema 4](#_Toc478504184)

[1.2. Descripción de la solución 4](#_Toc478504185)

[1.3. Código 5](#_Toc478504186)

[Función mezcla: 5](#_Toc478504187)

[Función combinar: 6](#_Toc478504188)

[1.4. Algoritmo de fuerza bruta 6](#_Toc478504189)

[1.5. Análisis de eficiencia 7](#_Toc478504190)

[Análisis de eficiencia teórica: 7](#_Toc478504191)

[Análisis eficiencia empírica/híbrida: 8](#_Toc478504192)

[2. Eliminar elementos repetidos 10](#_Toc478504193)

[2.1. Introducción al problema 10](#_Toc478504194)

[2.2. Descripción de la solución 10](#_Toc478504195)

[2.3. Código 10](#_Toc478504196)

[Función eliminar repetidos: 10](#_Toc478504197)

[Combinar: 11](#_Toc478504199)

[2.4. Algoritmo de fuerza bruta 11](#_Toc478504200)

[2.5. Análisis de eficiencia 12](#_Toc478504201)

[Análisis de eficiencia teórica 12](#_Toc478504202)

[Análisis eficiencia empírica/híbrida 13](#_Toc478504203)

[3. Cada elemento en su posición 15](#_Toc478504204)

[3.1. Introducción al problema 15](#_Toc478504205)

[3.2. Descripción de la solución 15](#_Toc478504206)

[3.3. Código 16](#_Toc478504207)

[3.4. Algoritmo de fuerza bruta 16](#_Toc478504208)

[3.5. Análisis de eficiencia 17](#_Toc478504209)

[Análisis de eficiencia teórica 17](#_Toc478504210)

[Análisis eficiencia empírica/híbrida 18](#_Toc478504211)

[4. Mezcla de k vectores ordenados. 22](#_Toc478504212)

[4.1. Introducción al problema 22](#_Toc478504213)

[4.2. Descripción de la solución 22](#_Toc478504214)

[4.3. Código 22](#_Toc478504215)

[Función mezclarKvectores 22](#_Toc478504216)

[Función mezclar2vectores 23](#_Toc478504217)

[4.4. Algoritmo de fuerza bruta 23](#_Toc478504218)

[4.5. Análisis de eficiencia 24](#_Toc478504219)

[Análisis de eficiencia teórica 24](#_Toc478504220)

[Análisis de eficiencia empírica/hibrida 25](#_Toc478504221)

[5. Serie unimodal de números 28](#_Toc478504222)

[5.1. Introducción al problema 28](#_Toc478504223)

[5.2. Descripción de la solución 28](#_Toc478504224)

[5.3. Código **¡Error! Marcador no definido.**](#_Toc478504225)

[29](#_Toc478504226)

[5.4. Algoritmo de fuerza bruta 29](#_Toc478504227)

[5.5. Análisis de eficiencia 30](#_Toc478504228)

[Análisis de eficiencia teórica 30](#_Toc478504229)

[Análisis de eficiencia empírica/híbrida 31](#_Toc478504230)

# 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.

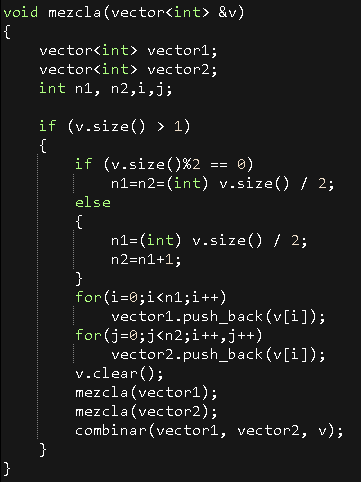
Teniendo claros estos dos aspectos, podemos entender que si incrementamos en una unidad el contador de inversiones en cualquier caso en el que encontremos un elemento en el hijo izquierdo que sea mayor que otro elemento del hijo derecho, la inversión entre ambos se contabilizará y el elemento del hijo 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

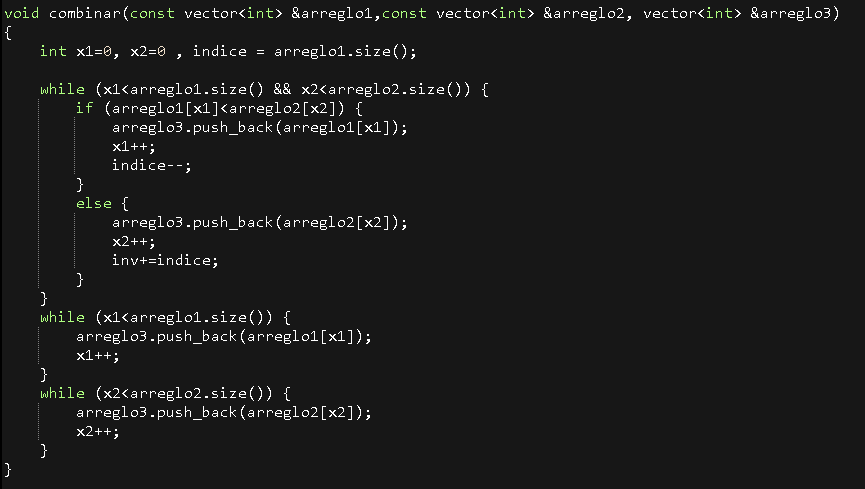
### Función mezcla:

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

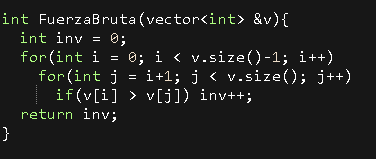


### Función combinar:

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



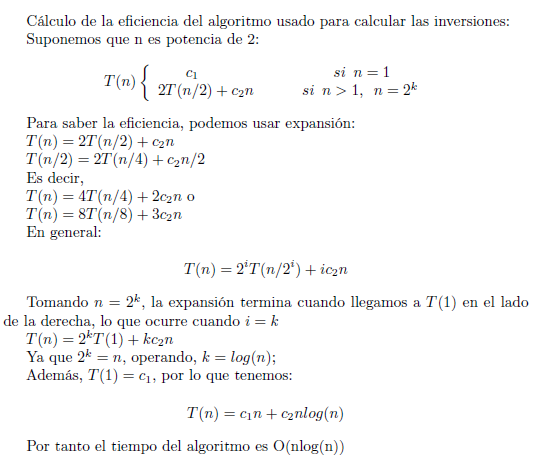
## 1.4. Algoritmo de fuerza bruta



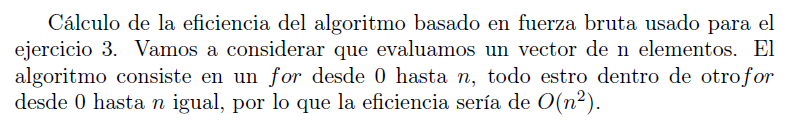
## 1.5. Análisis de eficiencia

### Análisis de eficiencia teórica:

#### Divide y vencerás:



#### Fuerza bruta:



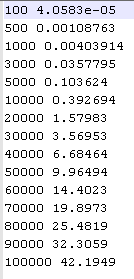
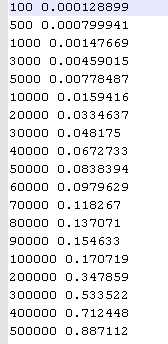
### Análisis eficiencia empírica/híbrida:

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)

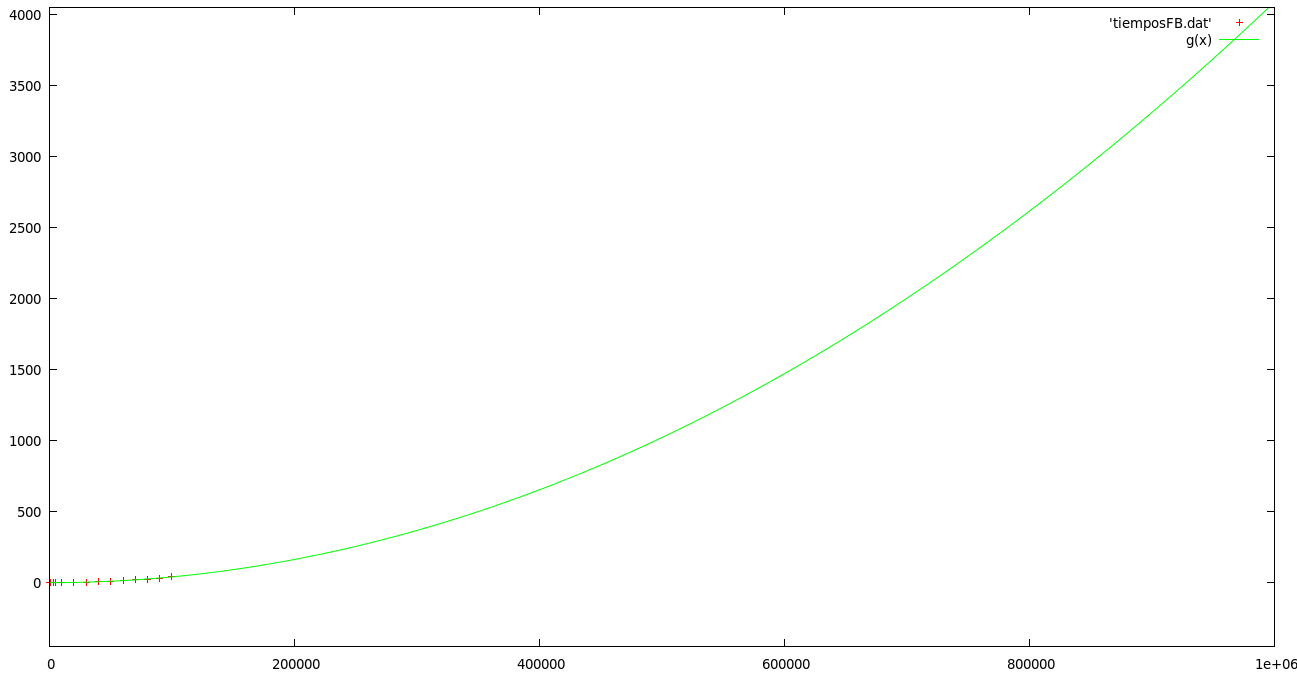
#### Datos

[Fuerza bruta:](file:///C:\Users\Adrián\Desktop\pdfALG\1.ContarInversiones\salidas\salidafbruta.dat) [Divide y vencerás:](file:///C:\Users\Adrián\Desktop\pdfALG\1.ContarInversiones\salidas\salidaDyV.dat)

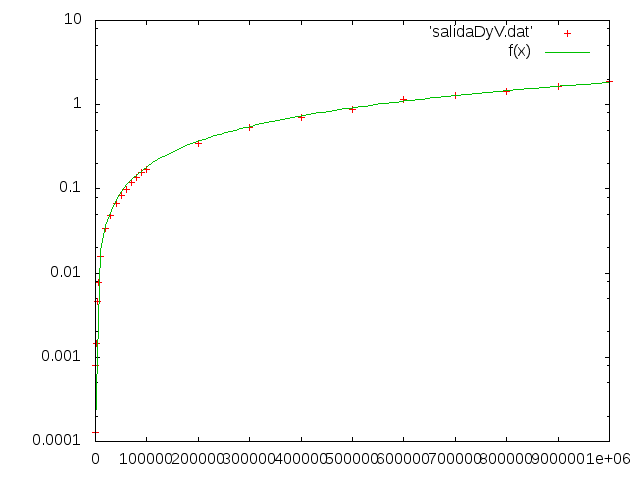


#### Gráficas:

* Fuerza bruta:

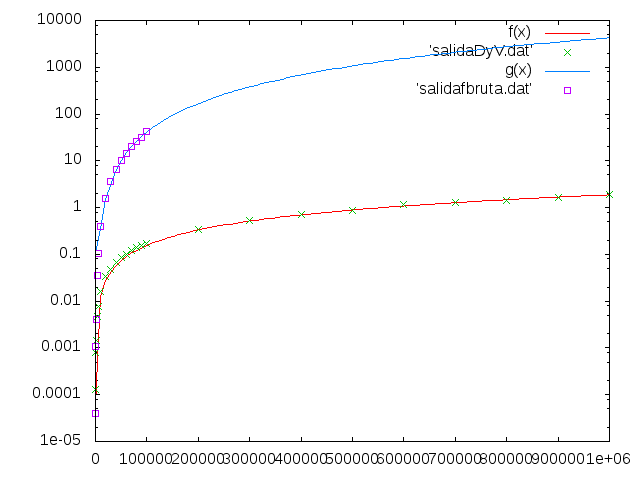
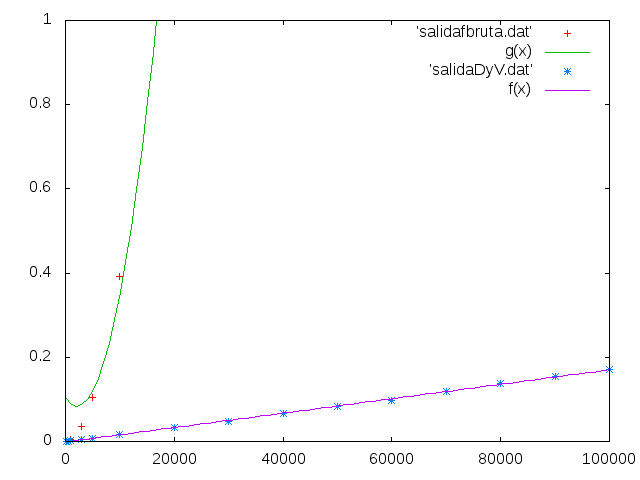


* Divide y vencerás:



* 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.

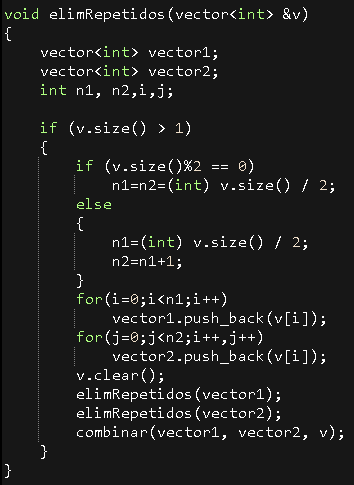
La solución consiste en dar como salida otro vector de enteros en el cual aparecerán todos los elementos que aparecían en el inicial, pero eliminando todas las repeticiones de elementos.

## 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.

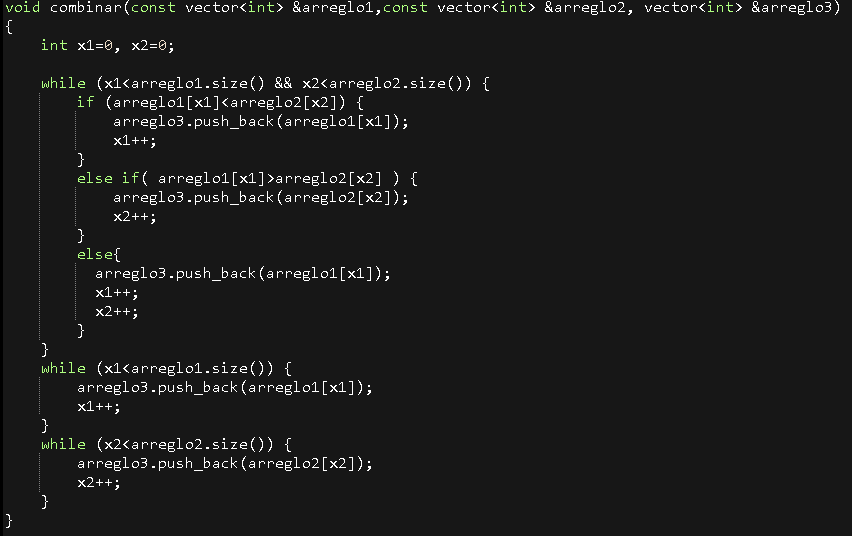
La clave para que la solución sea correcta está en esta función, la función combinar. Esta función va recorriendo y comparando ambos vectores hijos para id introduciendo en el vector resultado el elemento menor de cada comparación. En este punto, nuestro algoritmo añade una diferencia respento al algoritmo 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



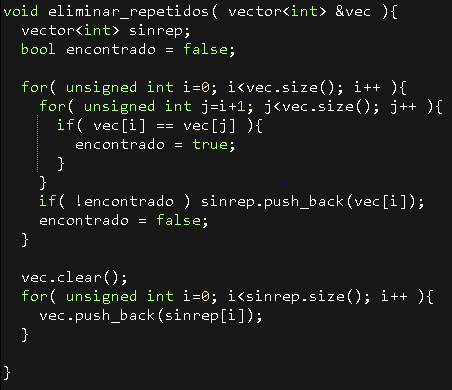
### Función eliminar repetidos:

### 



### Combinar:

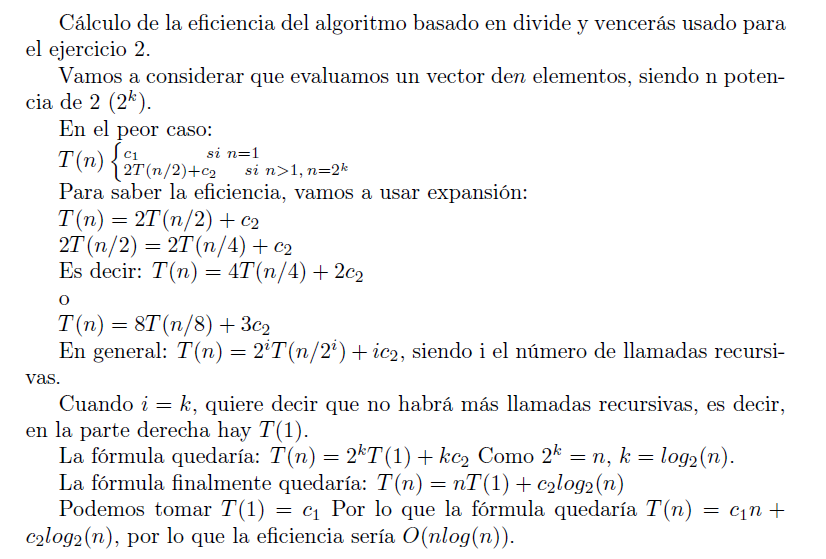
## 2.4. Algoritmo de fuerza bruta



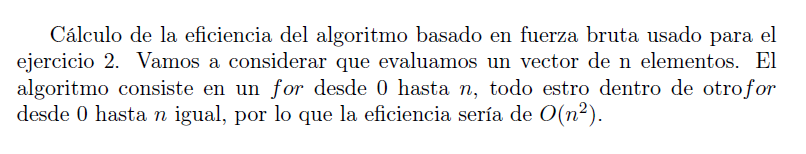
## 2.5. Análisis de eficiencia

### Análisis de eficiencia teórica

#### Divide y vencerás



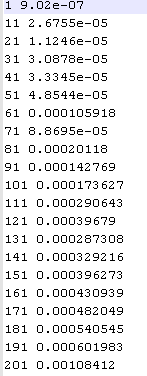
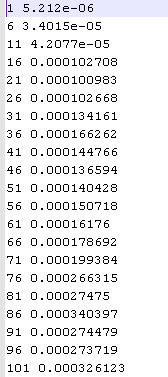
#### Fuerza bruta



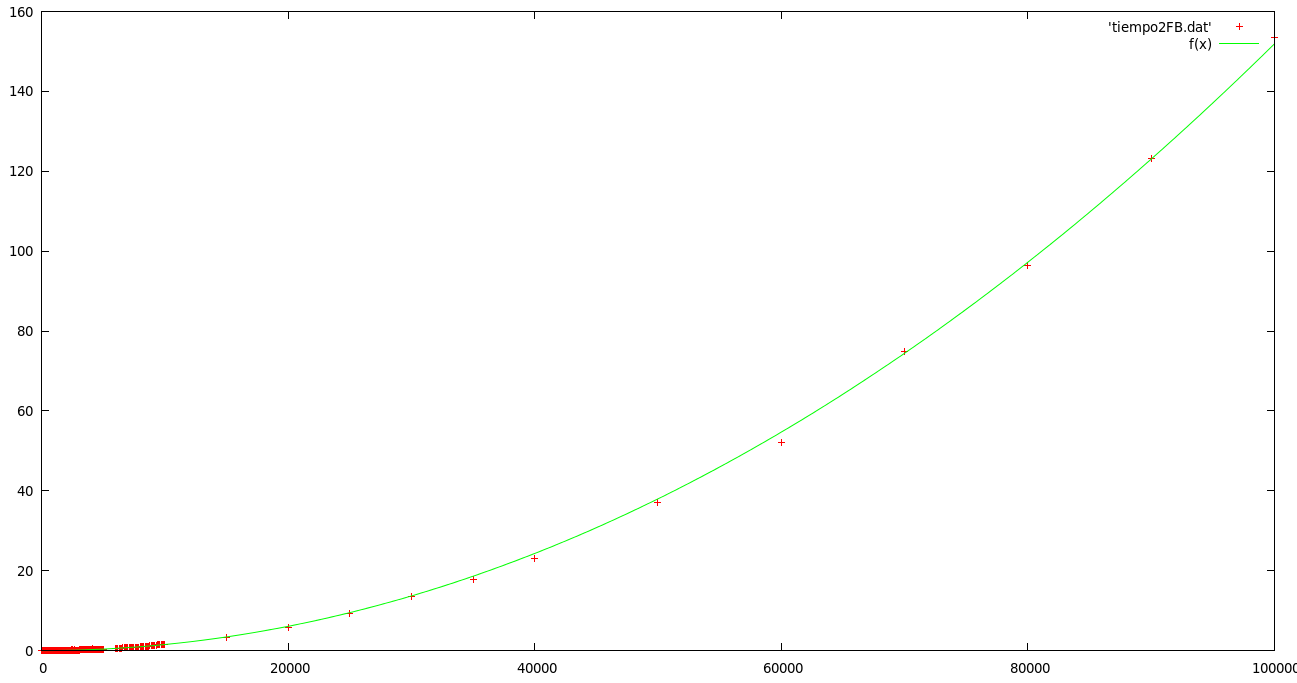
### Análisis eficiencia empírica/híbrida

#### Datos

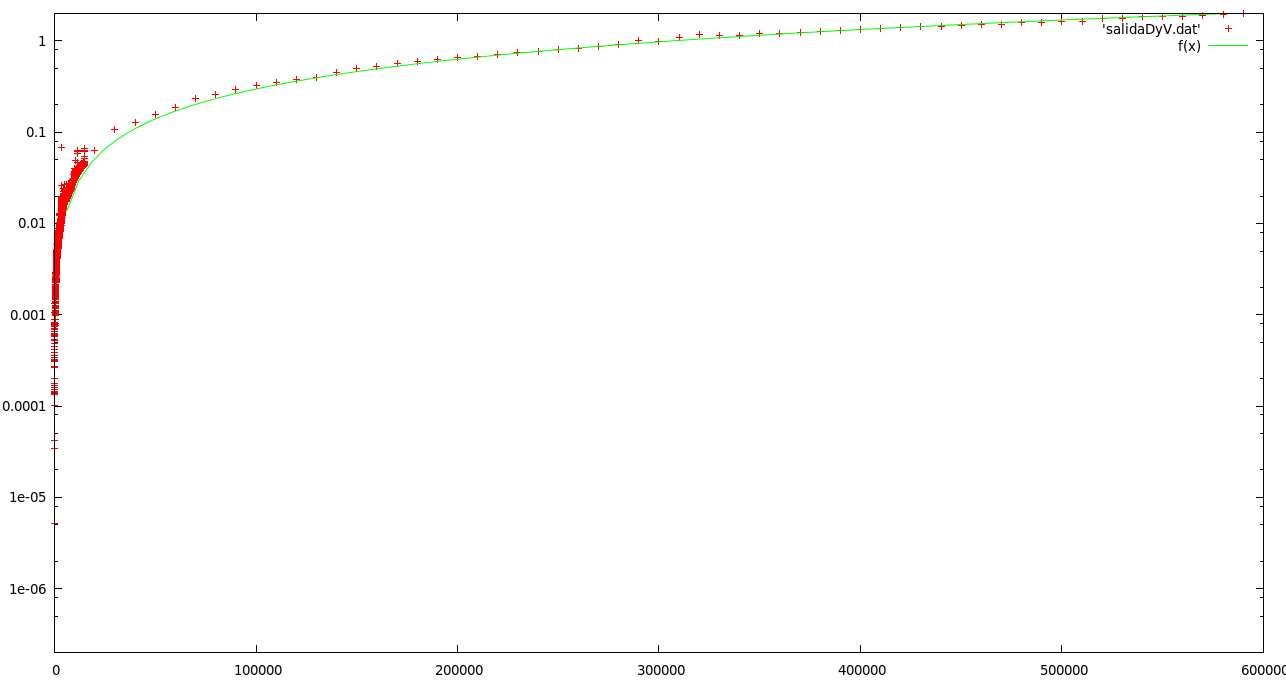
[Fuerza bruta:](file:///C:\Users\Adrián\Desktop\pdfALG\2.EliminarRepetidos\datos\salidafb.dat) [Divide y vencerás](file:///C:\Users\Adrián\Desktop\pdfALG\2.EliminarRepetidos\datos\salidaDyV.dat):



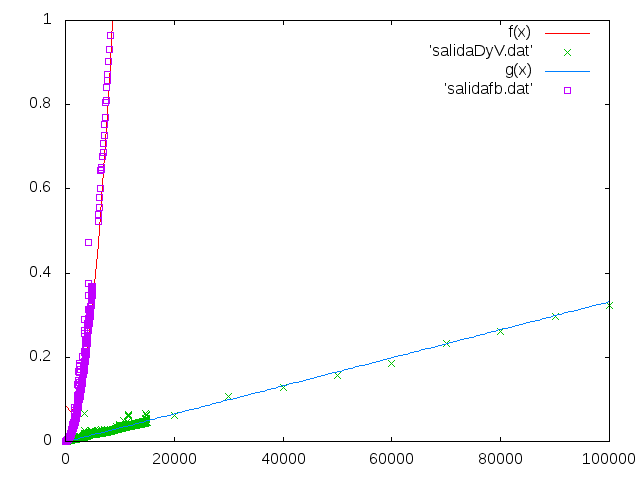
#### Gráficas:



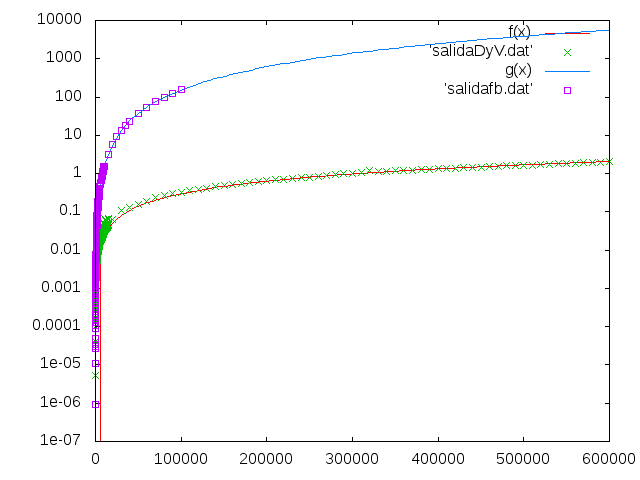
* Fuerza bruta
* Divide y vencerás:



* 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:

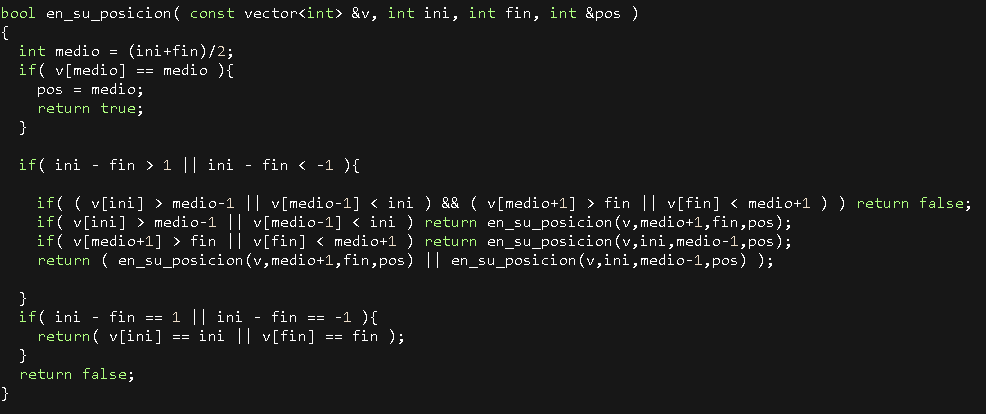
Si el entero de la posición de inicio de la mitad es mayor que el índice del elemento final, es imposible que algún entero coincida con su índice. Esto ocurre porque los elementos están ordenados en orden no decreciente. Pasa igual si el elemento de la posición final es menor que el índice de la posición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

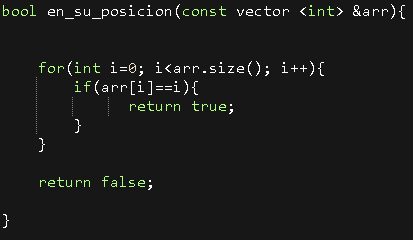
Cuando recursivamente se llega a mitades de vectores de sólo dos índices (0,1; 2,3; etc) el algoritmo no vuelve a dividir. Es más eficiente comprobar si alguno de los dos enteros coincide con su índice.

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



## 3.4. Algoritmo de fuerza bruta

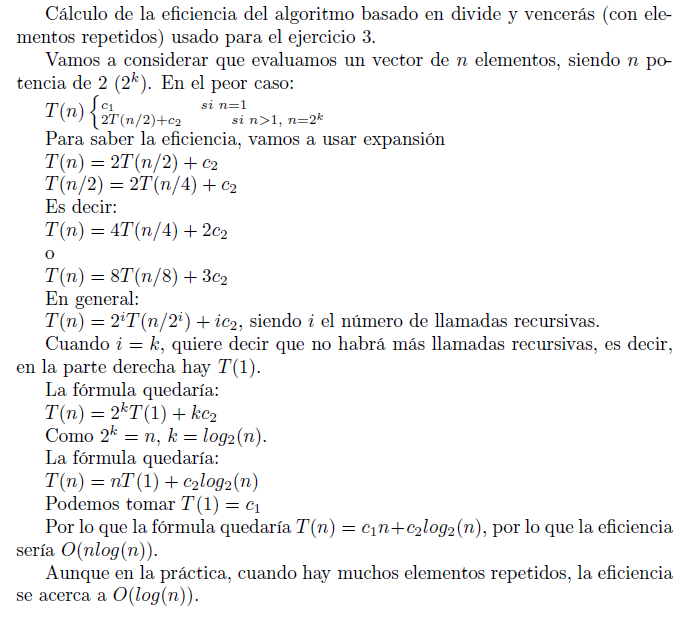
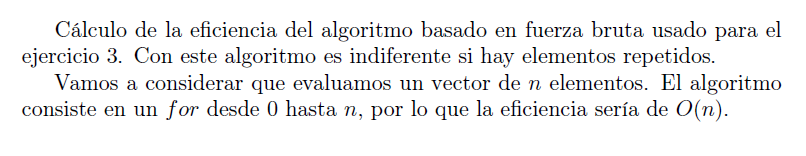


## 3.5. Análisis de eficiencia

### Análisis de eficiencia teórica

#### Divide y vencerás:

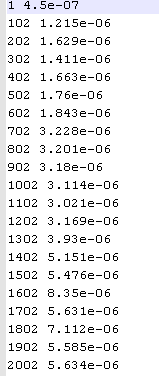
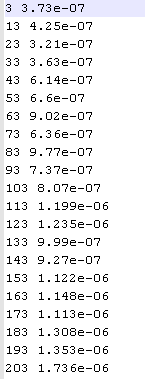
#### Fuerza bruta



### Análisis eficiencia empírica/híbrida

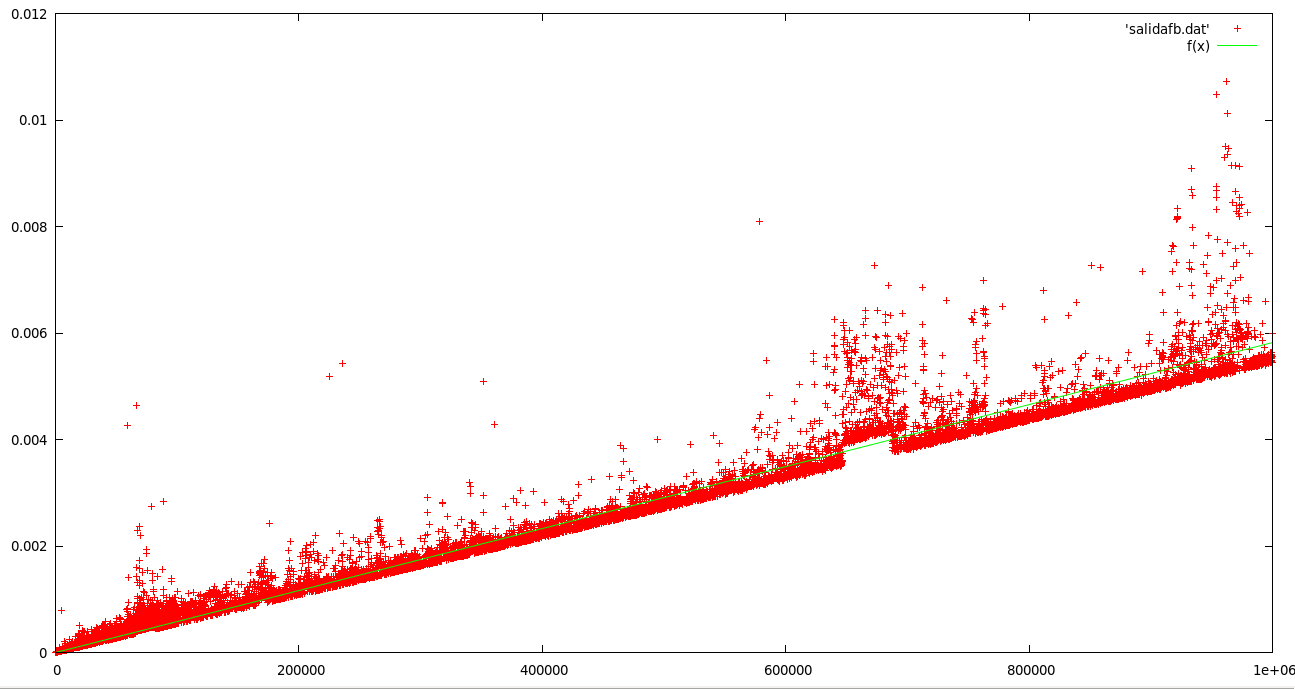
#### Datos:

[Fuerza bruta:](file:///C:\Users\Adrián\Desktop\pdfALG\3.CadaElementoEnSuPosicion\datos\salidafb.dat) [Divide y vencerás](file:///C:\Users\Adrián\Desktop\pdfALG\3.CadaElementoEnSuPosicion\datos\salida_repetidos.dat):

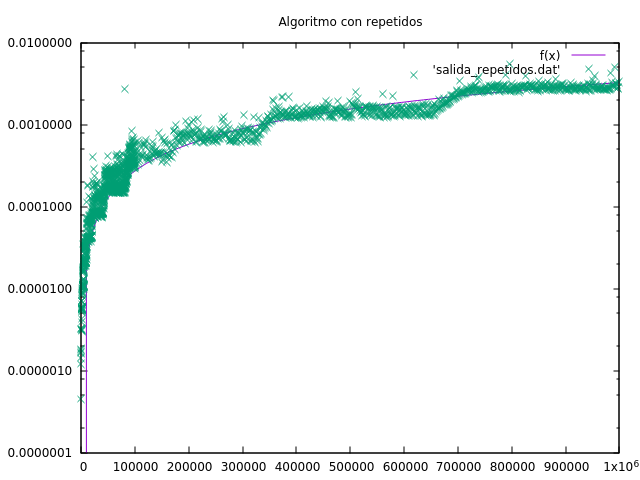


#### Gráficas

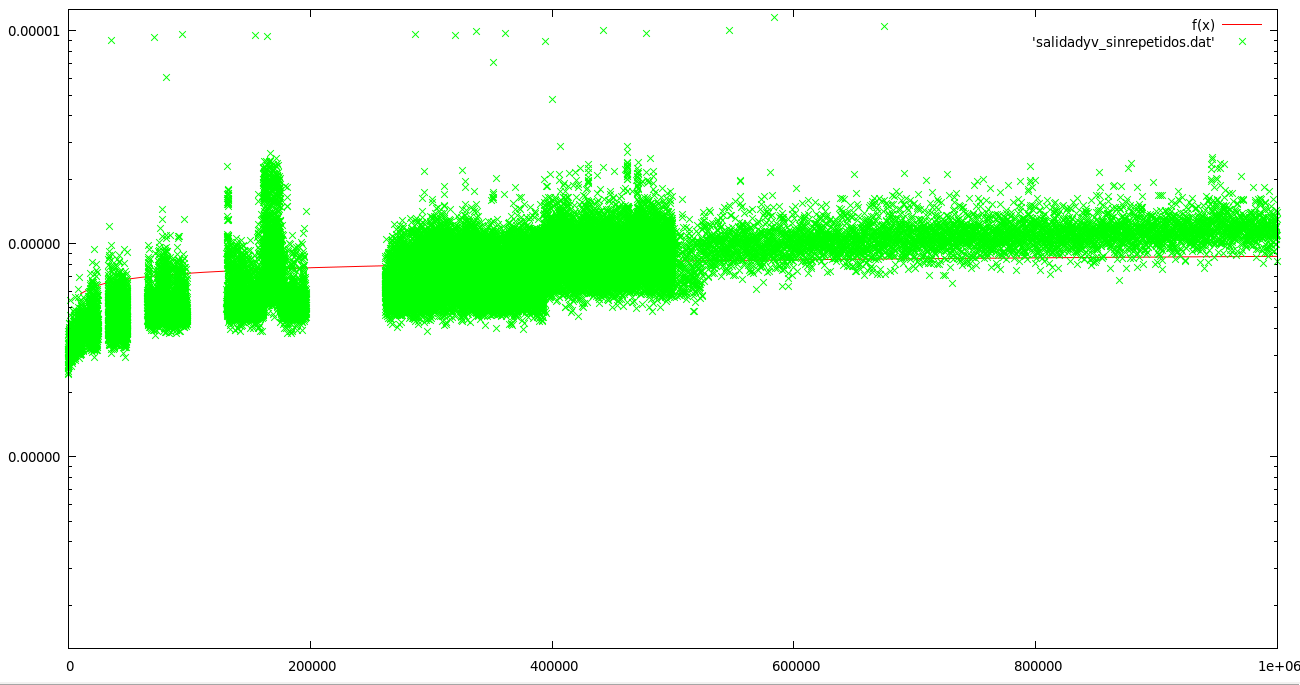
* Fuerza bruta:



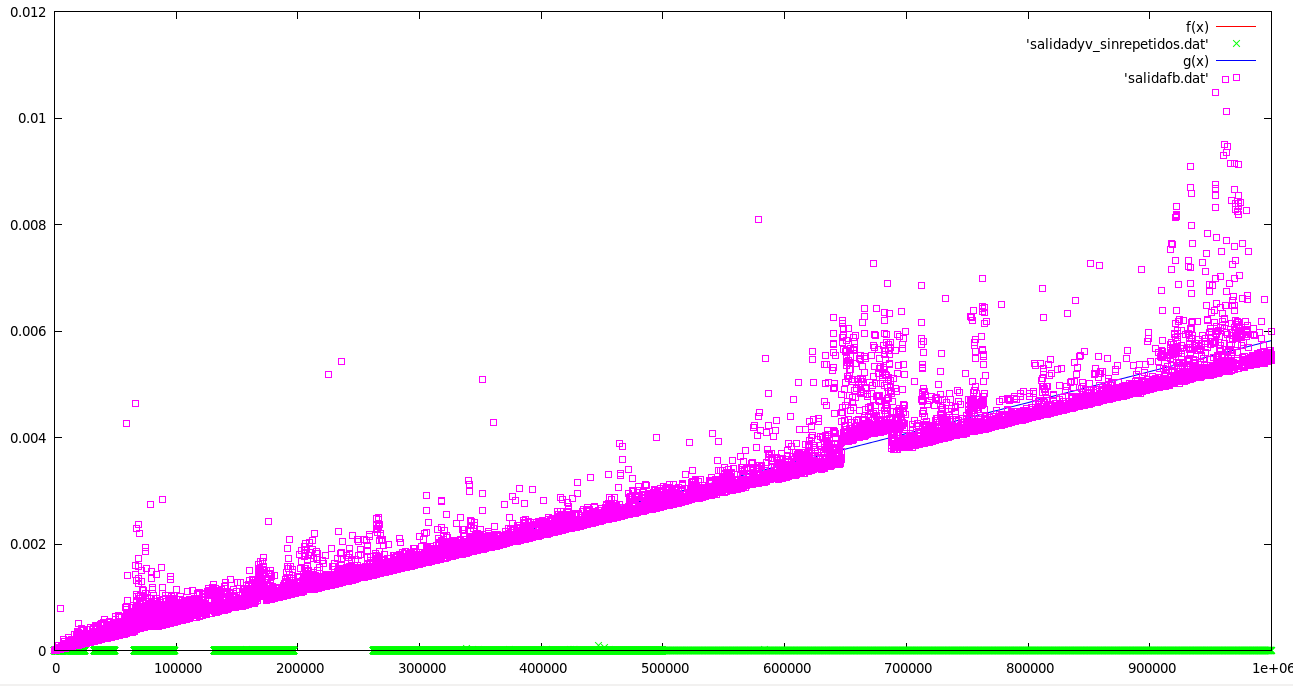
* Divide y vencerás(con repetidos):



* Divide y vencerás (sin repetidos):

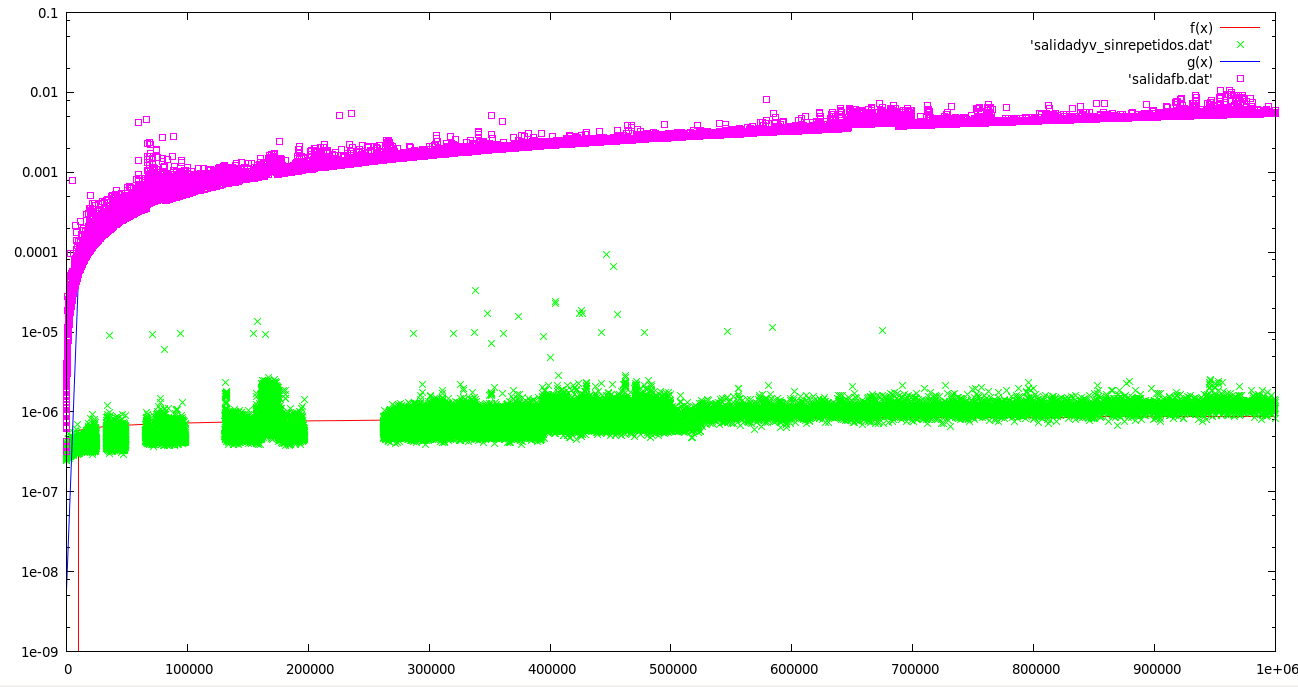


* Comparativas sin repetidos:



<- Sin escala logarítmica

Con escala logarítmica:

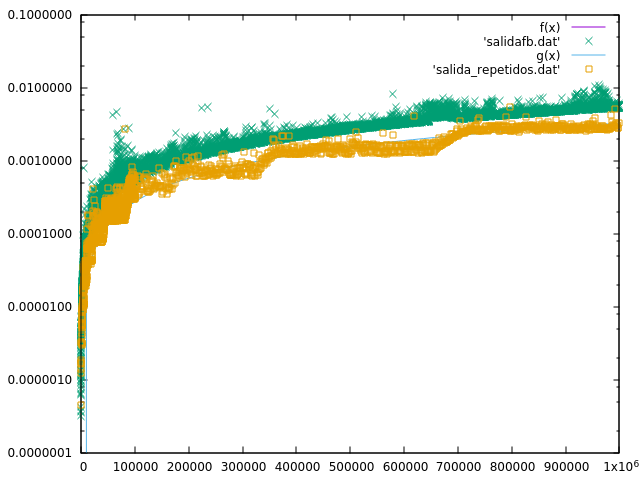


En este ejercicio, dependiendo del apartado abordado (con o sin elementos repetidos), obtendremos una eficiencia u otra:

* 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

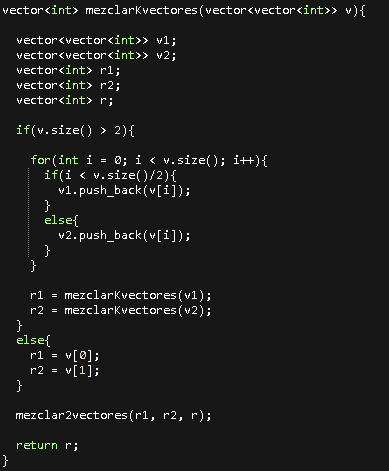
En este caso no disponemos inicialmente de un vector de enteros, sino de un conjunto de vectores de enteros, ordenados todos ellos de menos a mayor, que lo representaremos como un vector de vectores. El problema a resolver en este ejercicio es, a partir del conjunto de vectores dado, proporcionar como 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

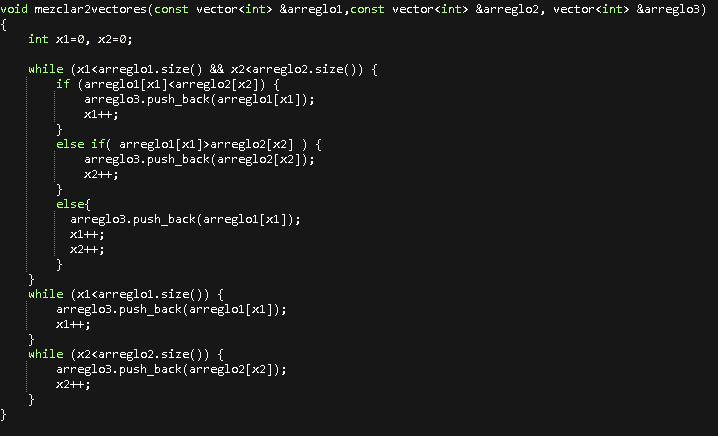
En esta solución volvemos a hacer uso como base de esta un algoritmo mergesort, pero aquí no va a subdividir un vector en otros dos vectores, sino que dividirá el conjunto de vectores proporcionando así otros dos subconjuntos de vectores. Esta subdivisión se llevará a cabo mediante la llamada recursiva 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

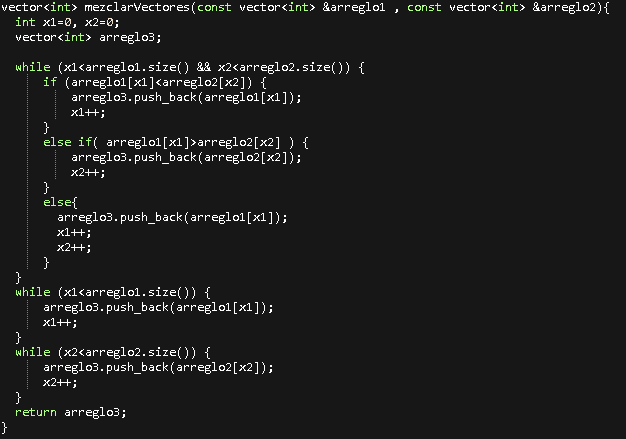
### Función mezclarKvectores



### Función mezclar2vectores



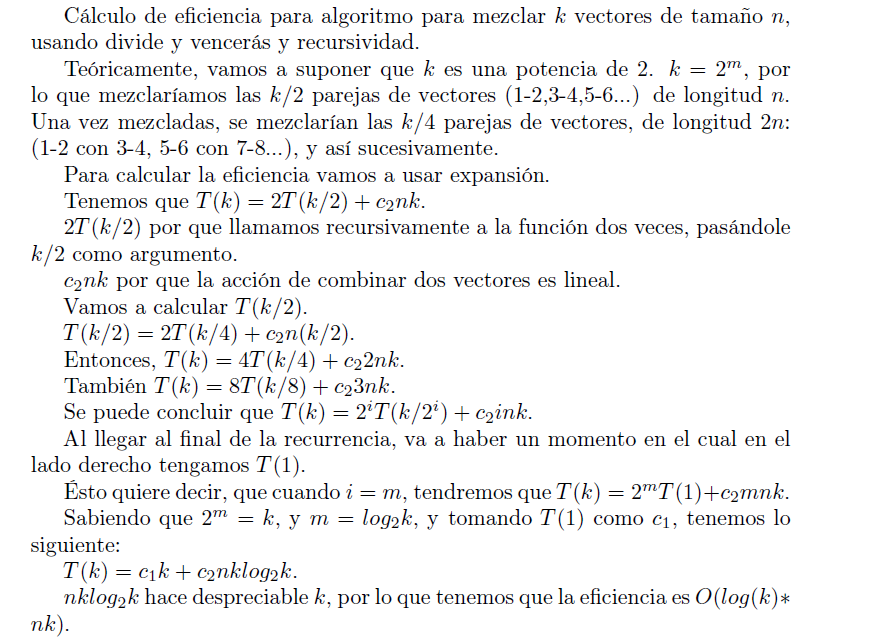
## 4.4. Algoritmo de fuerza bruta



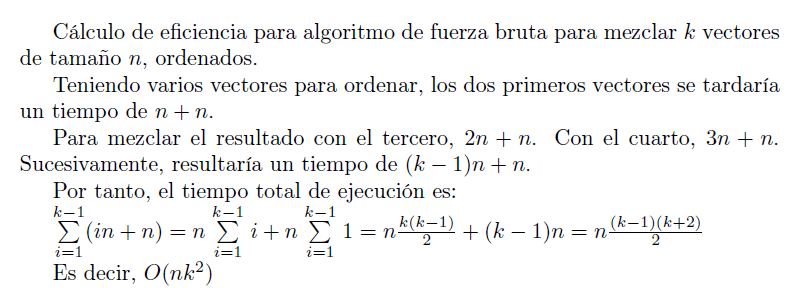
## 4.5. Análisis de eficiencia

### Análisis de eficiencia teórica

#### Divide y vencerás:



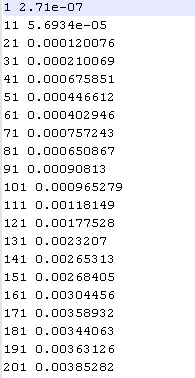
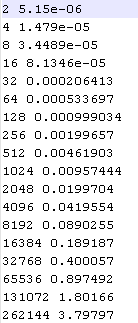
#### Fuerza bruta



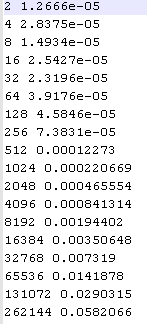
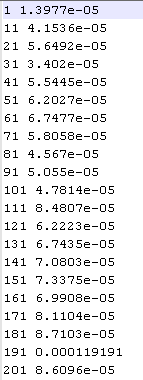
### Análisis de eficiencia empírica/hibrida

#### Datos:

[Fuerza bruta(elementos constantes)](file:///C:\Users\Adrián\Desktop\pdfALG\4.MezclarVectoresOrdenados\datos\salidafbruta5elem.dat): [Divide y vencerás(elementos constantes):](file:///C:\Users\Adrián\Desktop\pdfALG\4.MezclarVectoresOrdenados\datos\salidaDyV5tam.dat)

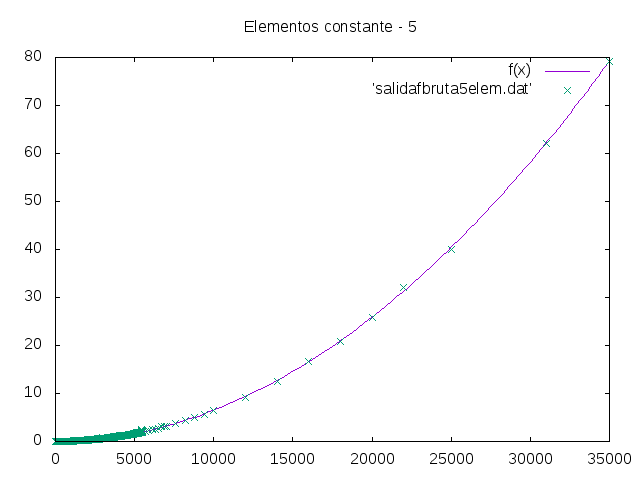


[Fuerza bruta(vectores constantes):](file:///C:\Users\Adrián\Desktop\pdfALG\4.MezclarVectoresOrdenados\datos\salidafbruta5vect.dat) [Divide y vencerás(vectores constantes):](file:///C:\Users\Adrián\Desktop\pdfALG\4.MezclarVectoresOrdenados\datos\salidaDyV4vect.dat)

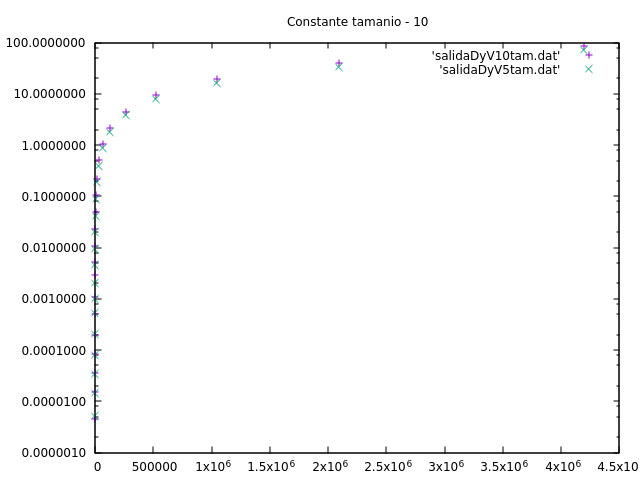


#### Gráficas:

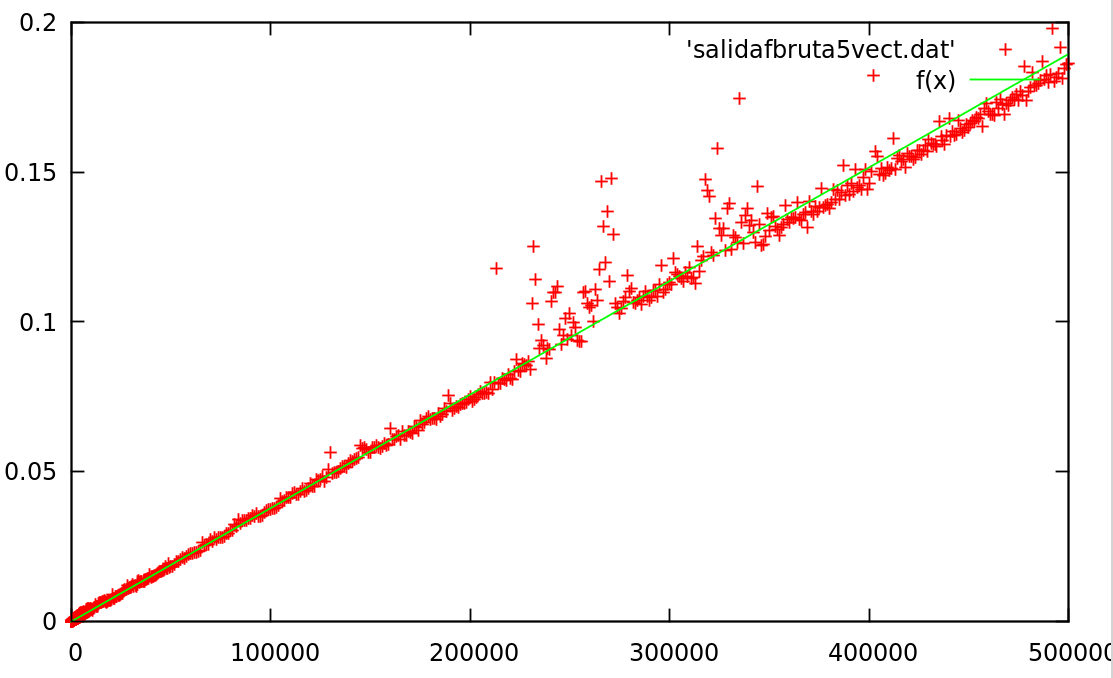
Fuerza bruta(elementos constantes):



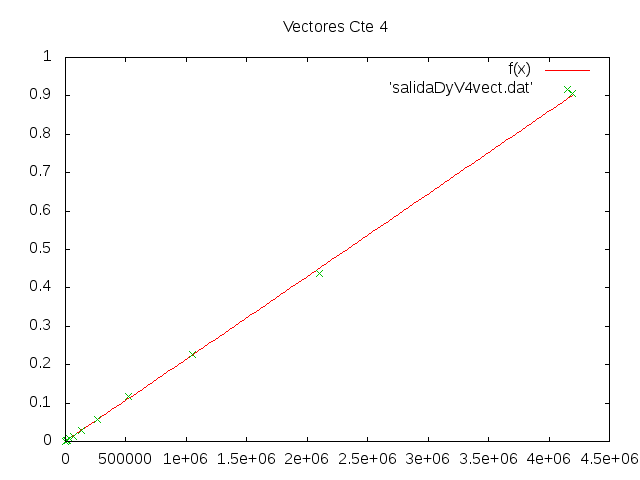
Divide y vencerás(elementos constantes):



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

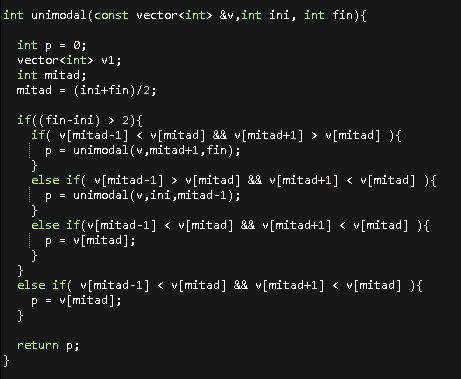
La solución aportada tiene como base un algoritmo de búsqueda binaria utilizando la técnica de divide y vencerás. El vector inicial será subdividido de forma recursiva hasta que este tenga un tamaño igual a 3, pero en este caso, a diferencia del algoritmo mergesort, no haremos dos llamadas recursivas 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.

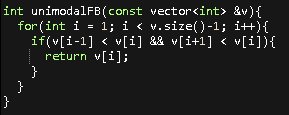
Estas comprobaciones se realizarán antes de hacer cada subdivisión, hasta que el vector sea de tamaño 3, en cuyo caso se cumpliría siempre la tercera condición y la solución siempre sería la correcta(no tiene por que llegar a tamaño 3, puede darse el caso en el que en una de las comprobaciones se de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

## 5.3. Código



## 

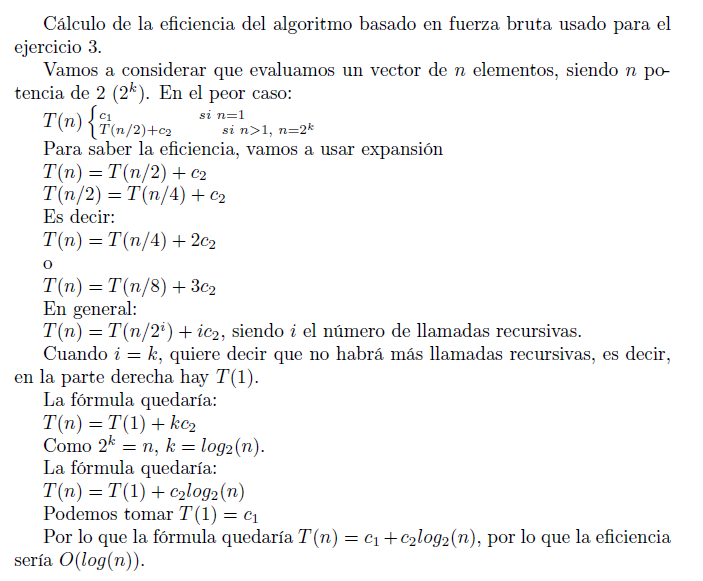
## 5.4. Algoritmo de fuerza bruta



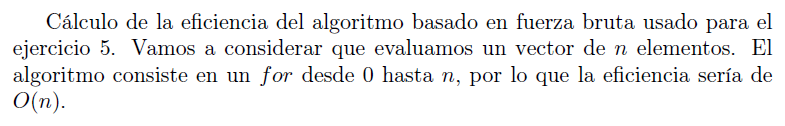
## 5.5. Análisis de eficiencia

### Análisis de eficiencia teórica

#### Divide y vencerás



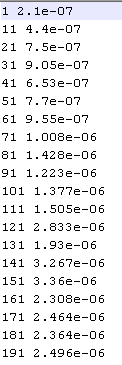
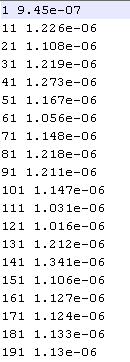
#### Fuerza bruta



### Análisis de eficiencia empírica/híbrida

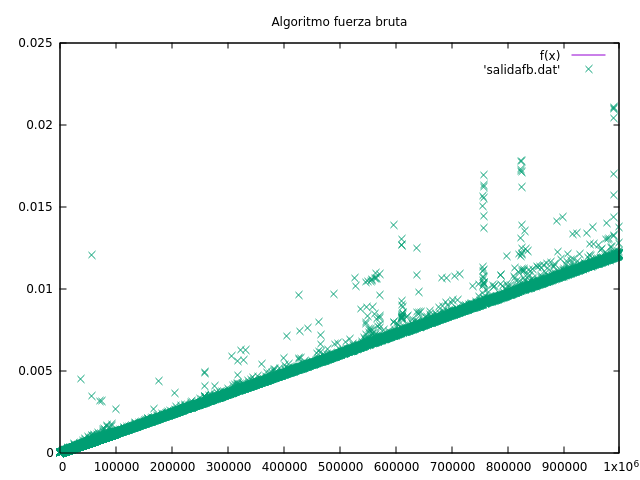
#### Datos:

[Fuerza bruta:](file:///C:\Users\Adrián\Desktop\pdfALG\5.Unimodal\datos\salidafb.dat) [Divide y vencerás:](file:///C:\Users\Adrián\Desktop\pdfALG\5.Unimodal\datos\salidadyv.dat)

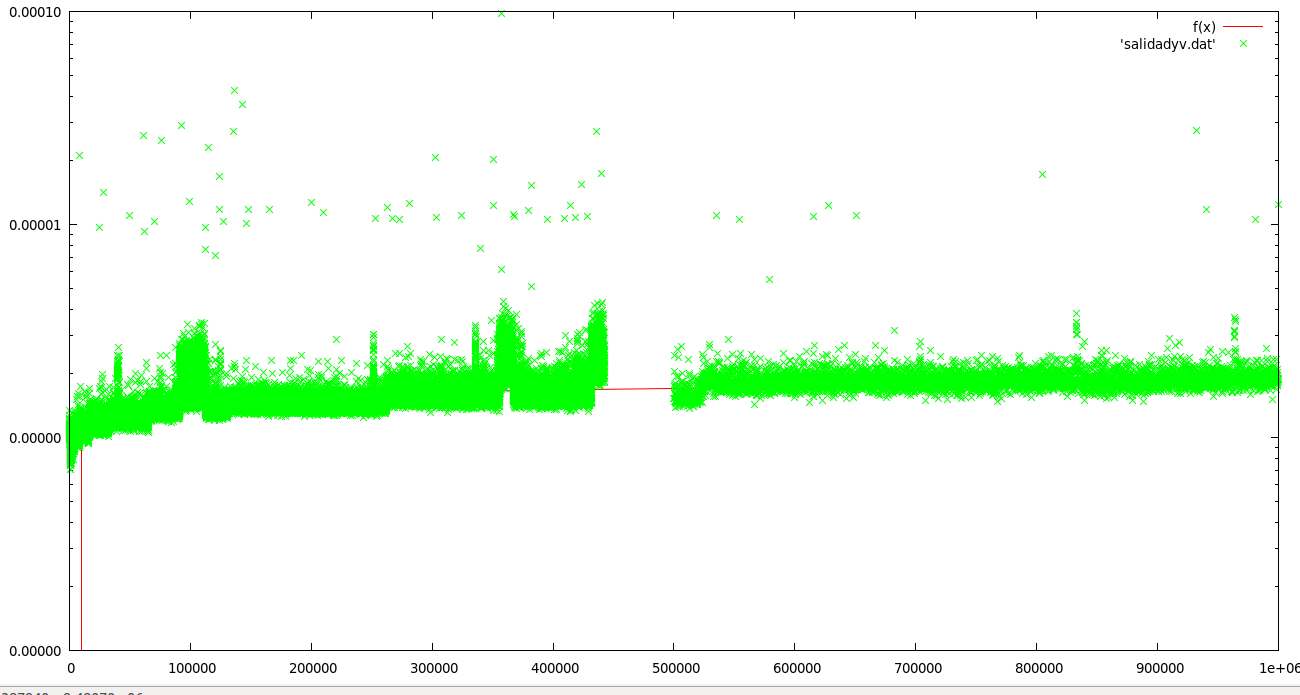


#### Gráficas

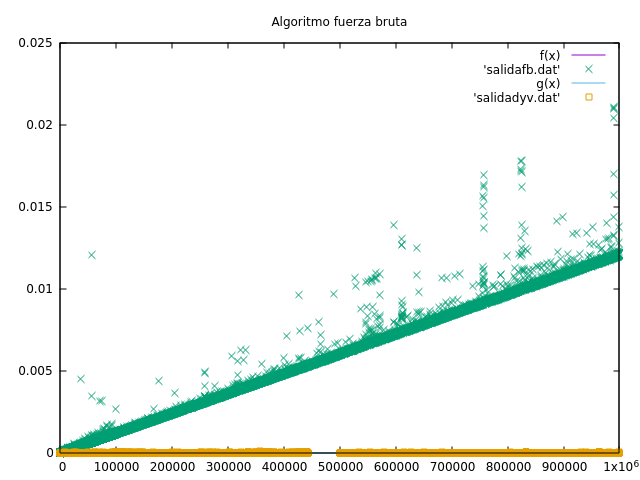
Fuerza bruta:



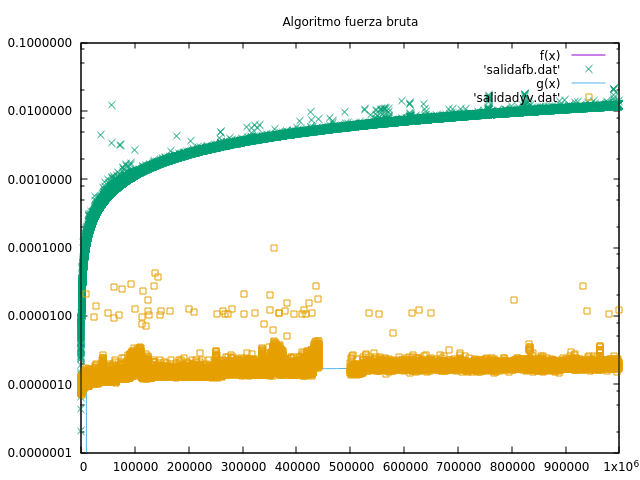
Divide y vencerás:



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.