Práctica Divide y Vencerás

Carlos Enríquez López (celopez@correo.ugr.es)
Sergio Cruz Pérez (sergiocruz@correo.ugr.es)
Manuel Ariza Ortiz (manuariza95@correo.ugr.es)
Javier Bueno López (javibl8@correo.ugr.es)

PRACTICA1

La práctica 1 trata sobre el desarrollo de algoritmos siguiendo el paradigma divide y vencerás.
Para cada problema se sigue la siguiente estructura:

- Enunciado del problema.
- Resolución teórica del problema.
- Análisis empírico. Análisis de la eficiencia híbrida.

En este último apartado se proporcionan gráficas con los resultados de los algoritmos y un análisis de la
eficiencia híbrida para los mismos.
Los algoritmos se han ejecutado sobre un ordenador con las siguientes características:

- **Marca:** Asus
- **RAM:** 8 GB
- **Procesador:** Intel® Core™ i5-5200U CPU @ 2.20GHz × 4

EJERCICIO 1

Enunciado del problema

Muchos sitios web intentan comparar las preferencias de dos usuarios para realizar sugerencias a partir de las preferencias de usuarios con gustos similares a los nuestros. Dado un ranking de n productos (p.ej. películas) mediante el cual los usuarios indicamos nuestras preferencias, un algoritmo puede medir la similitud de nuestras preferencias contando el número de inversiones: dos productos i y j están "invertidos"" en las preferencias de A y B si el usuario A prefiere el producto i antes que el j, mientras que el usuario B prefiere el producto j antes que el i. Esto es, cuantas menos inversiones existan entre dos rankings, más similares serán las preferencias de los usuarios representados por esos rankings.
Por simplicidad podemos suponer que los productos se pueden identificar mediante enteros **1, . . . , n**, y que uno de los rankings siempre es **1, . . . , n** (si no fuese así bastaría reenumerarlos) y el otro es a 1 , a 2 , . . . , a n , de forma que dos productos **i y j** están **invertidos si i < j pero a i > a j** . De esta forma nuestra representación del problema será un vector de enteros v de tamaño n, de forma que v[i] = ai, ∀i = 1, . . . , n. El objetivo es diseñar, analizar la eficiencia e implementar un algoritmo "divide y vencerás" para medir la similitud entre dos rankings. Compararlo con el algoritmo de "fuerza bruta" obvio. Realizar también un estudio empírico e híbrido de la eficiencia de ambos algoritmos.

Resolución teórica del problema

Como se ha ido indicando en el enunciado del problema, podemos abstraernos en el problema propuesto para reducirlo a un problema equivalente, que es el de contar inversiones en un vector. Decimos que hay una inversión en un vector v, si para un par de índices i, j del vector verificando i < j, se tiene que v[i] > v[j].
Se puede pensar fácilmente en una forma sencilla de resolver este problema "a la fuerza bruta", sin más que recorrer todas las posibles parejas (i, j) : i < j e incrementar el número de inversiones cada vez que se nos presente la condición v[i] > v[j]. Analizaremos esta versión junto con otras basadas en ciertos algoritmos de ordenación, que nos permiten aprovechar las mismas iteraciones del algoritmo para incrementar las inversiones sin un exceso computacional apreciable. Finalmente, concluimos de esta forma con un algoritmo divide y vencerás basado en el algoritmo mergesort.

Solución trivial

Ya hemos mencionado en el apartado anterior la idea para resolver trivialmente nuestro problema. Dado un vector v de n elementos con índices 0, ..., n − 1, la idea trivial es la siguiente:
• Para i desde 0 hasta n − 1
• Para j desde i + 1 hasta n − 1
• Si v[i] > v[j]
• inversiones++
• Devolver inversiones.
Claramente, estamos hablando de un algoritmo cuadrático. Concretamente, la expresión condicional del interior de los bucles es de orden constante, y el número de iteraciones en función del tamaño n del vector es:

$$\sum_{i=0}^{n-1}\sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-1}(n-i-1) = \frac{n(n-1)}{2}$$

Solución final. Mergesort modificado.

Para usar el paradigma
divide y vencerás partimos nuestro vector por la mitad, obteniendo los subvectores izquierdo y derecho. Ahora partimos de que conocemos la solución de este subproblema, es decir, conocemos las inversiones de cada subvector. Ahora tenemos que juntar de nuevo los vectores y a partir de sus respectivas soluciones concluir con la solución final. ¿Cómo hacemos esto?
Supongamos que hubiéramos recuperado los subvectores ordenados tras la división del problema. Esta ordenación no nos afectaría al resultado puesto que estamos partiendo de que ya hemos obtenido las inversiones de cada subvector, y como permanecen las mismas componentes, las inversiones

existentes entre ambos subvectores van a seguir siendo las mismas. Una vez razonado esto, veamos cómo podemos obtener de forma sencilla las inversiones entre ambos vectores.
Además, como hemos usado en la hipótesis de división la ordenación, deberíamos juntar de forma ordenada los subvectores para no romper con la técnica divide y vencerás. De esta forma, podemos seguir un razonamiento muy simple: creamos un nuevo vector donde almacenaremos el ordenado de los subvectores,recorremos nuestros dos subvectores, con un índice al inicio de cada subvector y realizamos los siguientes pasos:

• Si el elemento que marca el índice del subvector izquierdo es menor o igual que el del subvector derecho, no tenemos inversión. De hecho, la condición de que los subvectores estén ordenados nos permite afirmar que no tenemos ninguna inversión del elemento izquierdo con todo el restante subvector derecho, por lo que podemos incrementar el índice izquierdo sin aumentar las inversiones con total tranquilidad. El elemento izquierdo, obviamente, es el que añadimos al vector ordenado.
• Si el elemento que marca el índice del subvector izquierdo es mayor que el del subvector derecho, vamos a tener inversiones. La ordenación de los subvectores nos permite afirmar que todo el subvector restante izquierdo restante por recorrer va a tener una inversión con el elemento del subvector derecho. De esta forma, añadimos tantas inversiones como elementos queden por recorrer en el subvector izquierdo.
Añadiremos en este caso el elemento derecho al array ordenado e incrementaremos el índice derecho.
Esto lo hacemos para cada índice, hasta haber recorrido los subvectores completos. Hay que tener en cuenta que cuando un vector se haya recorrido completo, los elementos restantes del otro se añadirán directamente sin inversiones adicionales. Nótese que tal y como se ha descrito, esta parte de nuestro algoritmo es lineal respecto a la suma de los tamaños de los subvectores, que es justamente el tamaño del vector inicial. También debemos percatarnos ahora que lo que acabamos de proponer no es ni más ni menos que el algoritmo de ordenación Merge Sort (ordenación por mezcla) ya conocido, al que se añade simplemente una pequeña modificación para contar las inversiones. Tras el algoritmo se nos devolverá el vector ordenado mediante las mezclas de los subvectores correspondientes. Como caso base para las divisiones
tendremos, como en el propio Merge Sort, los vectores de un elemento.

Pseudocódigo y eficiencia.
Una vez descrita nuestra solución, planteamos el algoritmo y su

eficiencia. Tenemos dos partes claramente diferenciadas en el algoritmo: la parte que se encarga de dividir y la parte que se encarga de mezclar. Sus pseudocódigos se muestran a continuación:

Mezclar:

```
int mergeAndCount(int *v, int begin, int middle, int end){
    int *sorted = new int[(unsigned)(end - begin)];

    int invs = 0, j = middle, i = begin, k = 0;

    while(i < middle){
        if(j < end){

            if(v[i] <= v[j]){
                sorted[k] = v[i];
                k++; i++;
            }
            else{
                sorted[k] = v[j];
                k++;j++;
                invs += (middle -i);
            }
        }
        else{
            sorted[k] = v[i];
            k++; i++;
        }
    }

    while(j < end){
        sorted[k] = v[j];
        k++; j++;
    }

    for(int r = 0; r < end - begin; r++){
        v[r+begin] = sorted[r];
    }

    delete [] sorted;
    return invs;
}
```

Dividir:

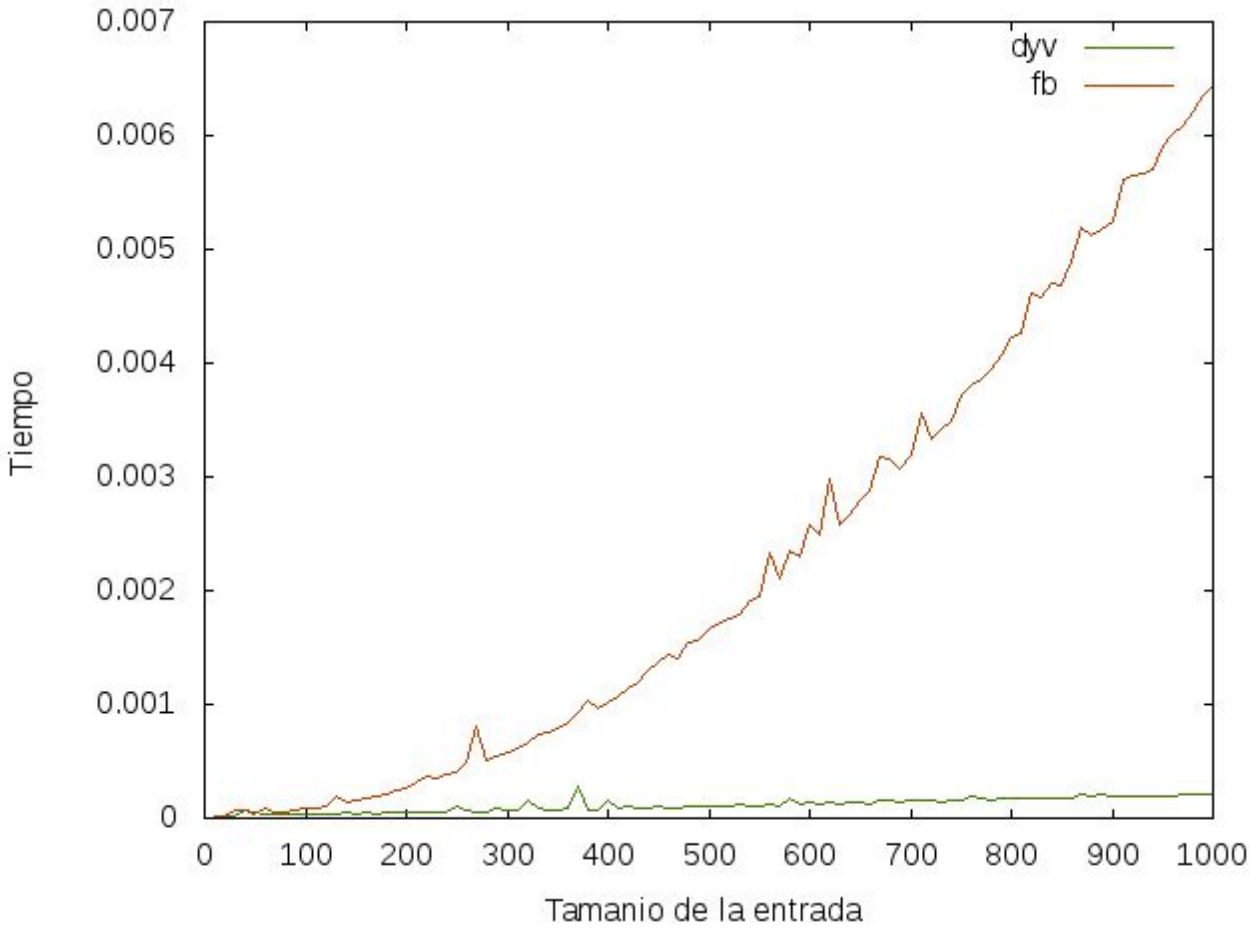
```
int sortAndCount(int *v, int begin, int end){
    int middle = (begin + end) / 2;
    int invs = 0;

    if(middle - begin > 1)
        invs += sortAndCount(v,begin,middle);

    if(end - middle > 1)
        invs += sortAndCount(v,middle,end);

    if(begin < middle && middle < end)
        invs += mergeAndCount(v,begin,middle,end);

    return invs;
}
```

Fijándonos en el algoritmo podemos observar con facilidad la relación de recurrencia que se satisface. Si
denotamos T (n) a la función con la eficiencia del algoritmo, y teniendo en cuenta como ya se dijo la linealidad del algoritmo de mezcla, tenemos que en el peor caso (cuando se entra en todas las sentencias condicionales)
T (n) verifica:

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

Tenemos una ecuación en diferencias, que tras el cambio de variable n = 2^k podemos resolver con facilidad:
T (n) = T (2^k) = 2T (2^k−1) + 2^k ⇐⇒ T (2^k) = a2^k + bk2^k ⇐⇒ T (n) = an + bn log n ∈ O(n log n)
De esta forma, gracias a la estrategia divide y vencerás hemos conseguido mejorar el orden de eficiencia cuadrático de la solución más inmediata a este problema hasta un orden n log n, lo cual es un gran avance respecto al coste computacional.

La eficiencia híbrida que se obtiene se ve representada en la siguiente gráfica:



Como vemos el algoritmo de divide y vencerás es mucho más rápido comparado al de fuerza bruta cuyo código es:

```
int fuerzabruta(vector<int> v){
int n_inversiones = 0;

    for (int i = 0; i < v.size(); i++){
        for(int j = i+1; j < v.size(); j++){
            if(v[i]>v[j])
            n_inversiones++;
        }
    }
return n_inversiones;
}
```

EJERCICIO 2

Enunciado del problema

Dado un vector de n elementos, de los cuales algunos pueden estar duplicados, el problema es obtener otro vector donde todos los elementos duplicados hayan sido eliminados. Diseñar, analizar la eficiencia e implementar un algoritmo sencillo para esta tarea, y luego hacer lo mismo con un algoritmo más eficiente, basado en Divide y Vencerás, con eficiencia O(n log n). Realizar también un estudio empírico e híbrido de la eficiencia de ambos algoritmos.

Resolución trivial

Un posible algoritmo consistiría en ir comparando cada elemento del vector v con los demás, y si son iguales eliminar uno de ellos. Los elementos no duplicados se van guardando en otro vector x. El algoritmo sería:

```
x[1]=v[1];
k=1;
for i=2 to n {
    j=1;
    found = false;
    while (j<=k && !found) {
        if (v[i]==x[j]) then found=true;
        else j=j+1;
    }
    if (!found) {
        k=k+1;
        x[k]=v[i];
    }
}
```

Pero este proceso tiene un tiempo de ejecución O(n^2) que es mayor que el exigido.

Resolución del problema

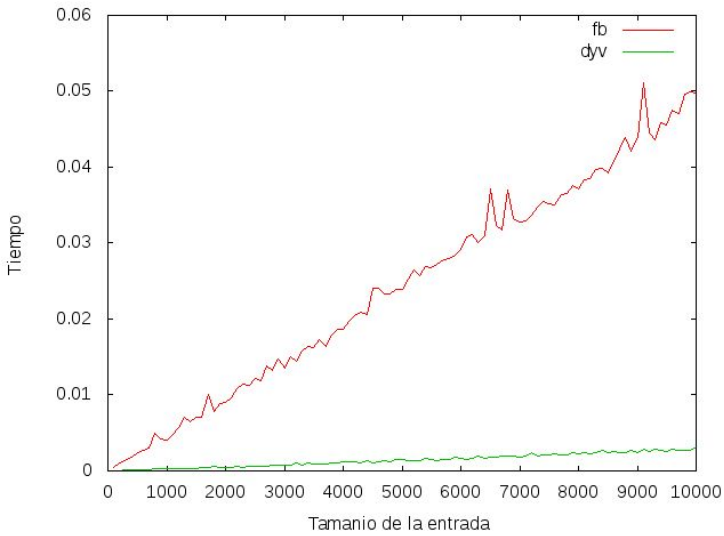
La mejor alternativa es emplear un enfoque DV directamente (no a través de un algoritmo de ordenación basado en DV): Dividimos el problema en 2 subproblemas de tamaño mitad aproximadamente, eliminamos los duplicados en cada una de las mitades, y luego eliminamos los elementos duplicados en las dos mitades. Si suponemos que los subproblemas se

resuelven eliminado duplicados y ordenándolos de menor a mayor, entonces es posible diseñar un método eficiente para componer las soluciones de los dos subproblemas (que también ordena el resultado de menor a mayor):

```
elimina_duplicados(v) {

n=length.v;
 if (n==1)
    return (v);
 dividir el vector v en dos mitades, a y b;
 a=elimina_duplicados(a);
 b=elimina_duplicados(b);
 v=mezcla_y_elimina(a,b);
 return (v);
}
```

Eficiencia empírica, estudio híbrido

Para analizar la eficiencia empírica hemos realizado un breve script en bash que nos sirve para lanzar el programa con distintos tamaños y crear así un fichero de datos que posteriormente interpretamos con gnuplot. Como consecuencia hemos obtenido las siguientes gráficas para cada algoritmo.



EJERCICIO 3

Enunciado del problema

Dado un vector V de números enteros, todos distintos, ordenado de forma no decreciente, se quiere determinar si existe un índice i tal que V[i] = i y encontrarlo en ese caso.

Diseñar e implementar un algoritmo Divide y Vencerá que permita resolver el problema. ¿Qué complejidad tiene ese algoritmo? ¿Y el algoritmo “obvio” para realizar esa tarea?
Realizar también un estudio empírico e híbrido de la eficiencia de ambos algoritmos.
Supóngase ahora que los enteros no tienen porqué ser todos distintos (pueden repetirse). Determinar si el algoritmo anterior sigue siendo válido, y en caso negativo proponer uno que sí lo sea. ¿Sigue siendo preferible al algoritmo obvio?

Resolución teórica del problema

En principio el problema se puede resolver de manera trivial, es decir, recorriendo el vector desde su inicio hasta el final y para solo en caso de que v[m] = m.
Otra forma de poder resolver el problema es mediante la técnica divide y vencerás. De esta manera actuaremos sobre el vector como lo hace la búsqueda binaria reduciendo el problema a la mitad en cada iteración lo que en principio debería ser más eficiente.

Solución trivial

Como ya hemos mencionado la solución trivial consiste en:
-Recorrer el vector desde 0 hasta n-1
-Si v[m] = m entonces devolvemos ‘m’ y acabamos la ejecución.

Ya que en el peor caso recorremos el vector al completo no cabe duda de que este algoritmo es O(n).

Algoritmo de fuerza bruta utilizado para la solución trivial:

```
int fuerza_bruta(vector<int> &v,int TAM){
    for (int i = 1; i < TAM; ++i)
    {
        if(v[i]==i){
            return i;
        }
    }

    return 0;
}
```

Solución mediante divide y vencerás

Como el vector está ordenado de menor a mayor, podemos actuar como con la búsqueda binaria, examinando el elemento que se encuentra en la posición de la mitad, m = (n + 1)/2 (la mediana en este caso).

Si coincide para ese elemento su valor con el índice (v[m] = m), ya hemos encontrado el índice buscado. En caso contrario, si el valor de ese entero es mayor que el índice (v[m] > m), al estar los elementos ordenados y no repetirse, sabemos que para todos los índices mayores que m, los valores en esas posiciones serán siempre mayores que los propios índices, es decir v[j] > j, $\forall j > m$ (esto se demuestra más adelante).
Por tanto sabemos que en el lado derecho del vector, a partir de la posición m, no se puede producir la situación buscada.
Basta entonces comprobar si en la parte izquierda del vector se puede producir tal situación.

Reducimos la búsqueda entonces al subvector desde la posición inicial a la posición m − 1. Si lo que ocurre es que v[m] < m, entonces el razonamiento es el mismo pero al revés: no se puede producir la situación buscada en la parte izquierda del vector (desde el inicio hasta m), y solo tenemos que comprobar si ocurre en la parte derecha, desde la posición m + 1 hasta la final.
El hecho de que si v[m] > m entonces v[j] > j, $\forall j > m$, se puede demostrar fácilmente por inducción:
Para el caso base, al ser v[m + 1] > v[m] (por estar ordenado el vector y no repetirse los enteros), entonces $v[m + 1] \geq v[m] + 1 > m + 1$
Para el paso de inducción, si v[j] > j, entonces (por la misma razón de antes) $v[j + 1] \geq v[j] + 1 > j + 1$.

Sabiendo esto es sencillo implementar el algoritmo divide y vencerás asociado a la solución:

```
int localiza(vector<int> &v, int primero, int ultimo){

  if (primero==ultimo)
  {
    if (v[primero]==primero)
    {
        return primero;
    }
    else
        return 0;
  }
  else{
    int i=(primero+ultimo)/2;
    if (v[i]==i)
    {
        return i;
    }
    else if (v[i]>i)
    {
        return localiza(v,primero,i-1);
    }
    else
    {
        return localiza(v,i+1,ultimo);
    }
  }
}
```

Cuando los enteros se pueden repetir el algoritmo anterior puede no funcionar correctamente (nótese que en la demostración de la propiedad clave era preciso suponer que $v[j+1] > v[j]$, no bastaba con $v[j+1] \geq v[j]$).
El algoritmo anterior puede fallar porque no podemos asegurar, cuando por ejemplo $v[m] > m$, que podamos descartar totalmente la parte derecha del vector a partir de m.

A primera vista se puede ver que la modificación que usa divide y vencerás para un vector con elementos repetidos el claramente peor que la anterior, no obstante la implementación a partir del segundo algoritmo es sencilla por lo que se ha implementado para un único estudio empírico de su eficiencia.

Eficiencia teórica.
Como este algoritmo divide y vencerás está basado en la búsqueda binaria lo más probable es que sea O(logn), no obstante habrá que demostrarlo.

En cada llamada recursiva del algoritmo se hace un trabajo constante para

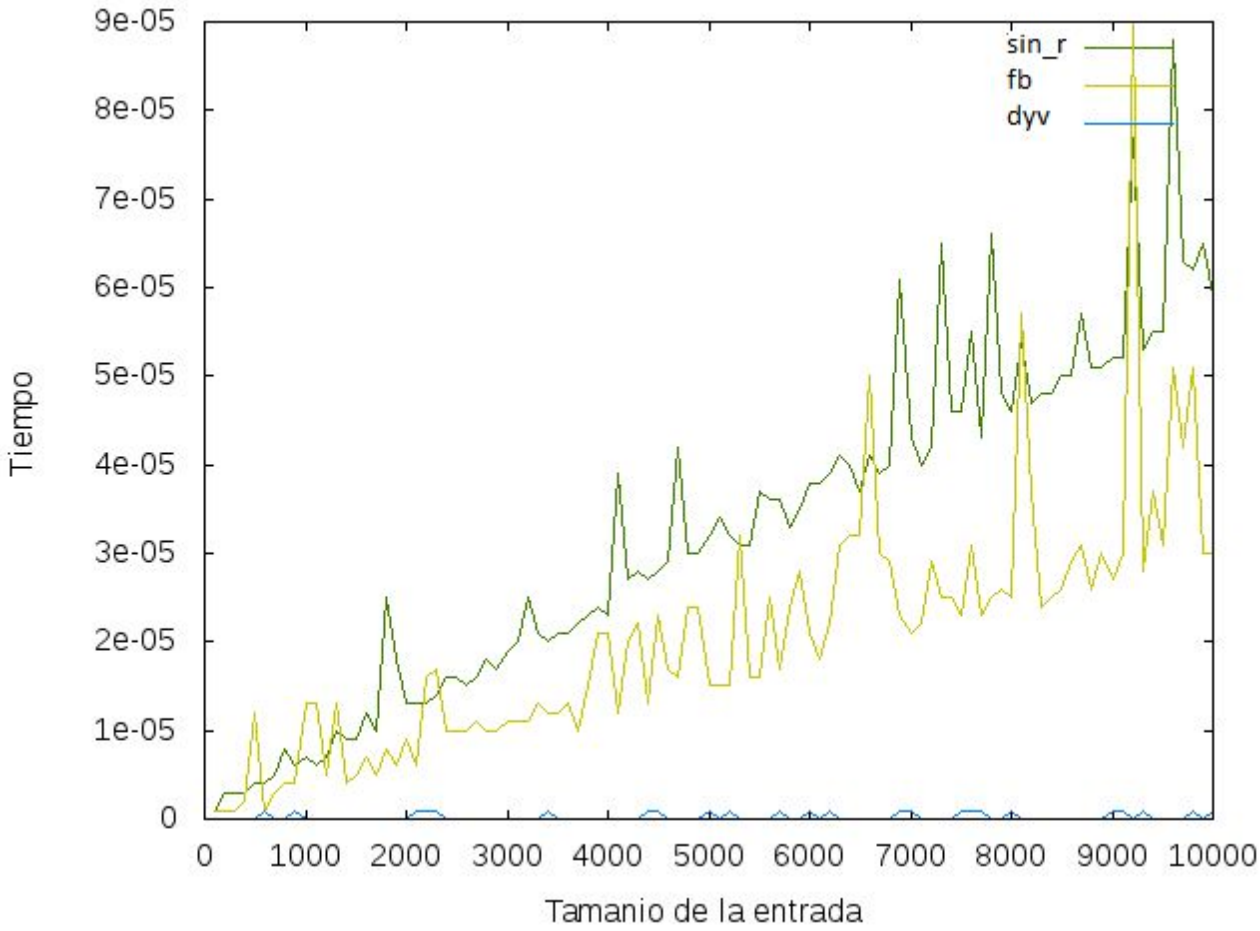
calcular el punto medio y decidir en cuál de los tres posibles casos estamos. En el peor de ellos, nunca se encuentra p y siempre se debe hacer una llamada recursiva a un vector de tamaño mitad. Si denotamos T(n) a función con la eficiencia del algoritmo, en el peor caso esta verifica:

$T(n) = 2T(n/2) + O(1)$

Tenemos una ecuación diferencial que podemos resolver con cambio de variable $n = 2^k$:
$T(n) = T(2^k) = 2T(2^{k-1}) + O(1) \Leftarrow\Rightarrow T(1) + (k-1)O(1) \in O(k) = O(\log n)$

Eficiencia empírica, estudio híbrido

Para analizar la eficiencia empírica hemos realizado un breve script en bash que nos sirve para lanzar el programa con distintos tamaños y crear así un fichero de datos que posteriormente interpretamos con gnuplot. Como consecuencia hemos obtenido las siguientes gráficas para cada algoritmo.



En la gráfica podemos apreciar, como ya sabíamos, que el algoritmo divide y vencerás es bastante más rápido que el de fuerza fuerza. Sin embargo lo

que destaca de la gráfica es que algoritmo utilizado para el vector con elementos repetidos es mucho más lento que el de fuerza bruta. Esto ocurre porque dicho algoritmo lanza llamadas recursivas a ambos lados con la consecuencia de que recorre el vector al completo igual que el algoritmo de fuerza bruta, sin embargo su constante oculta es mucho mayor por la cantillas de operaciones y llamadas utilizadas.

EJERCICIO 4

Enunciado del problema

Se tienen k vectores ordenados (de menor a mayor), cada uno con n elementos, y queremos combinarlos en un único vector ordenado (con kn elementos). Una alternativa directa, utilizando un algoritmo clásico, es mezclar los dos primeros vectores, posteriormente mezclar el resultado con el tercero, y así sucesivamente. – ¿Cuál sería la eficiencia de este algoritmo? – Diseñar, analizar la eficiencia e implementar un algoritmo de mezcla más eficiente, basado en Divide y Vencerás. – Realizar también un estudio empírico e híbrido de la eficiencia de ambos algoritmos.

Resolución teórica del problema

Como podemos observar el problema se puede resolver de dos formas, una de forma trivial y otra mediante la técnica divide y vencerás.
De la forma trivial, es solamente limitarse a coger dos vectores ordenados y mezclarlos entre sí, obteniendo un vector ordenado también que será mezclado con el siguiente y así sucesivamente hasta mezclar el vector resultante, que contiene la mezcla de todos los vectores, con el último vector.

Mediante la técnica de divide y vencerás, la única condición que tenemos que tener es que el número de vectores sea potencia de 2. Y lo que hacemos es ir seleccionando vectores en parejas de dos, esto lo que hace es ir obteniendo un vector resultante, que será seleccionado con otro resultante, y así sucesivamente hasta que el vector resultante no se pueda mezclar con otro, ésto significa que estará todo mezclado.

Solución trivial

El algoritmo inicialmente propuesto haría uso de un método para mezclar dos vectores ordenados cuyo tiempo de ejecución es proporcional a la suma de los tamaños de los vectores que se mezclan. Por tanto, en mezclar los dos primeros vectores tardaría un tiempo n + n. Para mezclar ese vector con el 3º tardaría 2n+n, para mezclar el resultado con el 4º tardaría 3n + n, y así sucesivamente, de modo que para mezclar el vector resultante con el

k-ésimo (el último) tardaría (k−1)n+n. Por tanto el tiempo total de ejecución es cuadrático en el número de vectores a mezclar: O(nk2).

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

Lo único que hacemos es pasarle como argumento, un vector de vectores, que contiene todos los vectores que tenemos que mezclar entre ellos, cada vez que mezclamos dos vectores con la función MezclarVectoresOrdenados(), obtenemos un auxiliar que posteriormente será pasado como argumento a la función mencionada anteriormente y así hasta llegar al último vector de la lista de vectores.

```
std::vector<int> FuerzaBruta( vector < vector<int> > v )
{
    std::vector<int> aux=v[0];
    for(int i=1; i<v.size();i++)
    {
        aux=MezclaVectoresOrdenados(aux,v[i]);
    }

    return aux;
}

int main(int argc, char *argv[])
{
```

Solución mediante divide y vencerás

Otra forma de proceder para resolver este problema sería pensando en descomponer el problema. Por ejemplo, si suponemos por un momento que k es una potencia de 2, k = 2m , entonces podríamos mezclar las k/2 = 2m−1 parejas de vectores (de longitud n) (el vector 1 con el 2, el vector 3 con el 4, hasta el vector k − 1 con el k), luego mezclar también las k/4 = 2m−2 parejas de vectores (de longitud 2n) (el vector 1-2 con el 3-4, el 5-6 con el 7-8,...), y así sucesivamente hasta mezclar la última pareja resultante de vectores (de longitud 2m−1n = nk/2).

Este proceso, en cada iteración mezcla k/2i = 2m−i parejas de vectores de tamaño 2i−1n, tardando pues un tiempo proporcional a 2m−i 2 2i−1n = 2mn = kn.

Como se realizan m = log k iteraciones, el tiempo total es proporcional a kn log k.

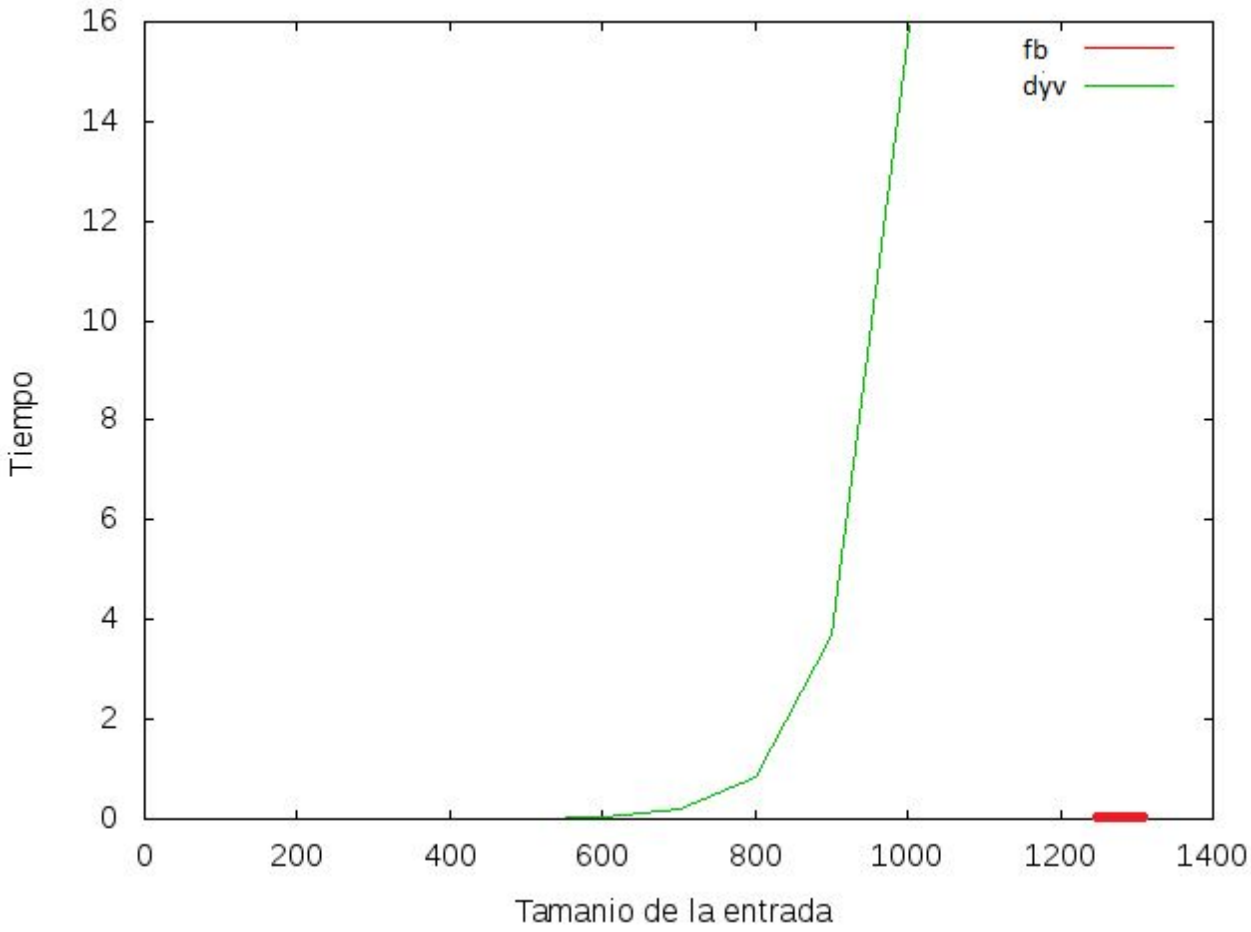
Realmente lo que hacemos mediante está solución es ir recorriendo un vector que contiene los vectores de enteros ordenados, e ir emparejándolos, así hasta llegar al final del vector que se le pasa a la función MezclaVector() como argumento.Dentro del bucle se llama a la función MezclaVectoresOrdenados, que da como resultado el vector mezclado al que ha dado lugar la mezcla de los dos vectores pasados como argumentos. La llamada a la función MezclaVector() se llama de forma recursiva, tantas veces como sea divisible, el tamaño del vector de vectores original, por 2.

```
vector< vector<int> > MezclaVectores( vector < vector<int> > v )
{
    vector< vector<int> > a_devolver;
    vector < vector<int> > aux;
    int j=0;

    if(v.size()==1)
        return v;

    for( int i=v.size();i>1;i/=2)
    {
        a_devolver.push_back(MezclaVectoresOrdenados(v[j],v[j+1]));
        j=j+2;
    }

    v=MezclaVectores(a_devolver);

    return v;
}
```

```
std::vector<int> MezclaVectoresOrdenados(std::vector<int> a,std::vector<int> b){
    std::vector<int> a_devolver;
    int final_A=a.size();
    int final_B=b.size();
    int indice_a=0;
    int indice_b=0;

    while(indice_a<final_A && indice_b<final_B){
        if(a[indice_a] < b[indice_b]){
            a_devolver.push_back(a[indice_a]);
            indice_a++;
        }
        else if(a[indice_a] > b[indice_b]){
            a_devolver.push_back(b[indice_b]);
            indice_b++;
        }
        else if( a[indice_a] == b[indice_b] ){
            a_devolver.push_back(b[indice_b]);
            a_devolver.push_back(a[indice_a]);
            indice_a++;
            indice_b++;
        }
    }

    if(indice_a==final_A){
        while(indice_b<final_B){
            a_devolver.push_back(b[indice_b]);
            indice_b++;
        }
    }

    if(indice_b==final_B){
        while(indice_a<final_A){
            a_devolver.push_back(a[indice_a]);
            indice_a++;
        }
    }
    return a_devolver;
}
```

Eficiencia empírica, estudio híbrido.

Para analizar la eficiencia empírica hemos realizado un breve script en bash que nos sirve para lanzar el programa con distintos tamaños y crear así un fichero de datos que posteriormente interpretamos con gnuplot. Como consecuencia hemos obtenido las siguientes gráficas para cada algoritmo.



EJERCICIO 5

Enunciado del problema

Sea un vector v de números con n componentes, todas distintas, de forma que existe un índice p (que no es ni el primero ni el último) tal que a la izquierda de p los números están ordenados de forma creciente y a la derecha de p están ordenados de forma decreciente, es decir:
∀i , j ≤ p con **i < j ⇒ v[i] < v[j]** y **∀i, j ≥ p** con **i < j ⇒ v[i] > v[j]** (*Imagen 1*)
Lo que implica que el máximo se encuentra en p. Diseñe un algoritmo "divide y vencerás" que permita determinar p. ¿Cuál es la complejidad del algoritmo?. Compárelo con el algoritmo "trivial" para realizar esta tarea. Realizar también un estudio empírico e híbrido de la eficiencia de ambos algoritmos.

Resolución teórica del problema

Un ejemplo de vector unimodal es el dado en la Imagen 1, en la cual se señala el buscado elemento p.
Se harán referencias a la imagen a lo largo del texto.

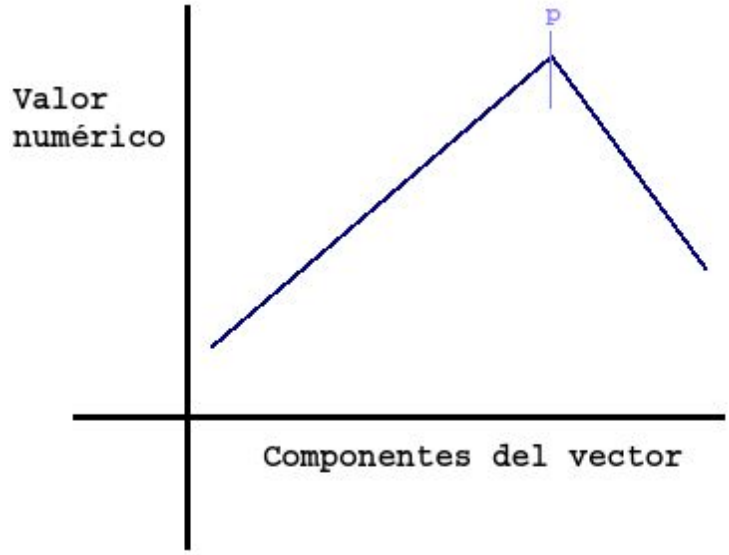


Imagen 1. Representación de un vector unimodal.

Daremos dos soluciones al problema, una trivial y otra basada en divide y vencerás. Como es de esperar, la primera tendrá un orden de eficiencia lineal mientras que con la segunda conseguiremos alcanzar el deseado orden log n.
En primer lugar, nótese que dada la forma del vector, caracterizada por (1), el elemento p es el máximo del vector. Esto implica que p verifica (2):
$v[p - 1] < v[p] > v[p + 1]$ (2)
Y, además, es el único elemento del vector que verifica (2) pues en caso caso contrario no se mantendría el invariante (1) del vector. Es por tanto el único máximo local. Utilizaremos este hecho para el desarrollo de ambos algoritmos.

Solución trivial
La idea es sencilla: recorrer el vector hasta encontrar el punto p. Recordemos que la propiedad (2) caracteriza al punto p. Por tanto, en nuestro algoritmo trivial basta recorrer todas las componentes 2, 3, ..., n − 1 y encontrar la primera componente que verifica (2), que será p por unicidad. Dado que nuestro recorrido es lineal de izquierda a derecha podemos obviar la primera comparación $v[p - 1] < v[p]$ pues si no hemos encontrado todavía p es porque $v[p - 2] < v[p - 1] < v[p - 1]$.

Solución mediante divide y vencerás

Queremos obtener una solución con un orden de eficiencia θ(log n). Para ello deberíamos librarnos de la búsqueda lineal. Un problema similar y ampliamente conocido es el de búsqueda en un vector. La búsqueda lineal sirve en vectores desordenados. Pero en el caso de ser ordenados, podemos hacerlo mucho mejor con una búsqueda binaria, que sigue las directrices del paradigma de diseño de algoritmos divide y vencerás. Nos inspiraremos en la búsqueda binaria para el diseño de nuestro algoritmo.
En primer lugar, comprobamos si el elemento en la posición mitad del vector verifica (2). En tal caso, este elemento debe ser p, devolviendo su índice como resultado del algoritmo. En el caso de no ser p recurrimos a divide y vencerás. Para ello, tal y como sucede en la búsqueda binaria, obviaremos uno de los dos vectores mitades (izquierda y derecha) llamando a nuestro algoritmo recursivo sobre el otro subvector.
La cuestión es la siguiente: ¿sobre qué subvector aplicar la recursividad y por qué?.
Consideremos el vector unimodal de la Imagen 1. Los elementos a la izquierda de p están ordenados de menor a mayor y los que se encuentran a su derecha de mayor a menor. Encontramos la componente mitad del vector, llamémosla m1. El vector se encuentra dividido tal y como muestra la Imagen 2.
Es claro que la componente p se queda en el subvector derecha. Podemos ahora abstraer la razón de este hecho. Esta es:
$v[m1 - 1] < v[m1] < v[m1 + 1]$
Como $v[m1] < v[p]$ y nos encontramos en el subvector creciente, p debe estar a la derecha de m1 tal y como sucede en la imagen.

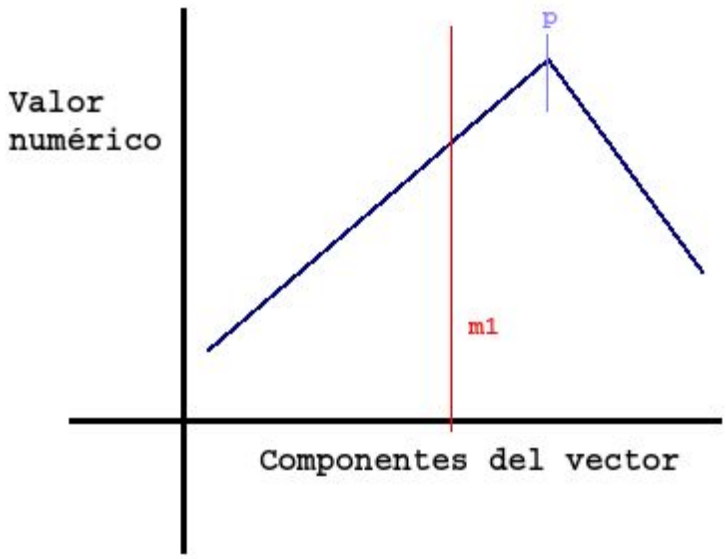


Imagen 2. Componente mitad del vector unimodal.

Utilizamos ahora recursividad sobre las componentes {m1 + 1, ..., n − 1}. La Imagen 3 muestra esta situación.
Obtenemos la componente mitad de este nuevo subvector, denotemos m2, que tampoco verifica (2) y por
ello no es p. Pero en este caso se tiene que m2 cumple:
v[m2 − 1] > v[m2] > v[m2 + 1]
Por tanto, estamos situados en la zona decreciente del vector v y por ello p se encuentra a la izquierda de m2
(v[p] > v[m2]). El subvector sobre el que se volvería a aplicar la recursividad se ha resaltado en gris. Se debe
repetir el proceso hasta dar con el elemento p.

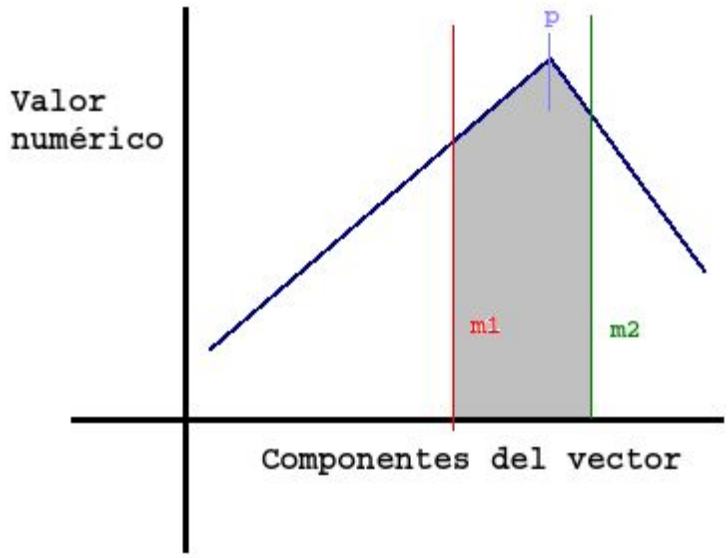


Imagen 3. Segunda iteración sobre el vector unimodal.

Eficiencia teórica.
En cada llamada recursiva del algoritmo se hace un trabajo constante para calcular el punto medio y decidir en cual de los tres posibles casos estamos. En el peor de ellos, nunca se encuentra p y siempre se debe hacer una llamada recursiva a un vector de tamaño mitad. Si denotamos T(n) a función con la eficiencia del algoritmo, en el peor caso esta verifica:

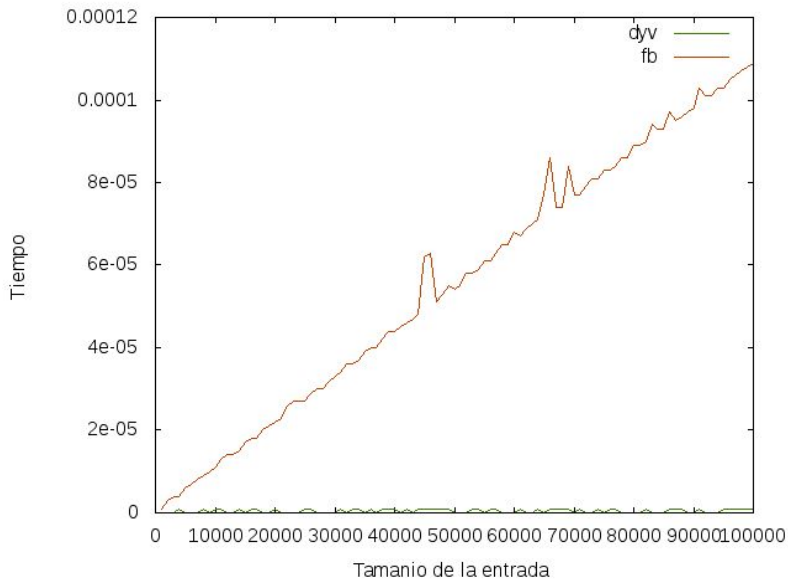
$T(n) = T(n/2) + \theta(1)$

Tenemos una ecuación diferencial lineal de fácil solución. Tomando n = 2^K

$T(n) = T(2^k) = T(2^{k-1}) + \theta(1) = \ldots = T(1) + (k-1)\theta(1) \in \theta(k) = \theta(\log n)$
Efectivamente, como ya se había adelantado, el algoritmo en el peor caso es $\theta(\log n)$.

Análisis empírico. Análisis de la eficiencia híbrida

Para analizar la eficiencia empírica hemos realizado un breve script en bash que nos sirve para lanzar el programa con distintos tamaños y crear así un fichero de datos que posteriormente interpretamos con gnuplot. Como consecuencia hemos obtenido las siguientes gráficas para cada algoritmo.



Como vemos, el algoritmo divide y vencerás es mucho más rápido que el algoritmo de fuerza bruta empleado.