

# DIVIDE Y VENCERAS: TRASPUESTA DE MATRIZ

## ANALISIS DE LA EFICIENCIA

### 1. EFICIENCIA TEÓRICA

La parte recursiva a estudiar es la siguiente:

```
void traspuesta(int **matriz, int dimension, int primera_fila, int ultima_fila, int primera_columna, int
ultima_columna){

    if( primera_fila < ultima_fila ) {

        int fila_del_medio, columna_del_medio

        fila_del_medio = (primera_fila+ultima_fila)/2;
        columna_del_medio = (primera_columna+ultima_columna)/2;

        traspuesta(matriz, dimension, primera_fila, fila_del_medio, primera_columna,
columna_del_medio);
        traspuesta(matriz, dimension, primera_fila, fila_del_medio, columna_del_medio+1,
ultima_columna);
        traspuesta(matriz, dimension, fila_del_medio+1, ultima_fila, primera_columna,
columna_del_medio);
        traspuesta(matriz, dimension, fila_del_medio+1, ultima_fila, columna_del_medio+1,
ultima_columna);

        intercambiar(matriz, fila_del_medio+1, primera_columna, primera_fila,
columna_del_medio+1, ultima_fila-fila_del_medio); //se hace la traspuesta de cada cuarta parte

    }

}
```

Y de forma auxiliar se adjunta la función “intercambiar”:

```
void intercambiar(int **matriz, int fila_partida, int columna_partida, int fila_llegada, int
columna_llegada, int tamano){

    for (int i=0; i<tamano; i++){

        for (int j=0; j<tamano; j++) {

            int aux = matriz[fila_partida+i][columna_partida+j];
            matriz[fila_partida+i][columna_partida+j] = matriz[fila_llegada+i][columna_llegada+j];
            matriz[fila_llegada+i][columna_llegada+j] = aux;

        }

    }

}
```

Empezamos analizando “intercambiar” para cuando recursivamente lleguemos a esta parte, la eficiencia de esa función esté calculada:

El bucle interno cuenta con una eficiencia:

ultima\_fila-fila\_del\_medio-1

$$\sum_{i=0}^{ultima\_fila-fila\_del\_medio-1} 1 = ultima\_fila-fila\_del\_medio-1-0+1 = ultima\_fila-fila\_del\_medio = n$$

El bucle externo:

n-1

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

Ahora analizamos la función recursiva completa:

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

$$T(2^k) = (2^k)^2 + 4T(2^{k-1})$$

$$T(2^k) - 4T(2^{k-1}) = 4^k$$

$$(x-4)^2 = 0$$

$$T_k = a \cdot 4^k + b \cdot k \cdot 4^k$$

$$T_n = 2a \cdot n^2 + 2b \cdot n^2 \cdot \log(n)$$

$$T(n) \in (n^2 \log(n))$$

## 2. EFICIENCIA HIBRIDA

Tras conocer que la eficiencia es  $O(n^2 \log(n))$ , sabemos que la función que define el comportamiento de nuestro algoritmo para distintos tamaños es  $T(n) = a \cdot n^2 \log(n) + b$  y necesitamos conocer el valor de las constantes para poder determinar el comportamiento en un punto concreto de la función. Para ello se ajusta la función a nuestros resultados y obtenemos las constantes ocultas, permitiendo conocer el tiempo aproximado que usará el algoritmo para un tamaño  $n$ . Tras toda la información que se encuentra en la captura siguiente, destacamos la parte de *Final set of parameters* que nos da los valores deseados.

```
gnuplot> f(x)=a*x*x*log(x)+b
gnuplot> fit f(x) 'tiempos_optimizado.dat' via a,b
iter   chisq      delta/lim  lambda  a          b
0 7.1359837392e+18  0.00e+00  4.89e+08  1.000000e+00  1.000000e+00
1 7.4255841825e+15 -9.60e+07  4.89e+07  3.467793e-02  1.000000e+00
2 3.5955569294e+09 -2.07e+11  4.89e+06  2.511255e-03  1.000000e+00
3 2.7710420972e+09 -2.98e+04  4.89e+05  2.500532e-03  1.000000e+00
4 2.7710420971e+09 -1.71e-06  4.89e+04  2.500532e-03  1.000000e+00
iter   chisq      delta/lim  lambda  a          b

After 4 iterations the fit converged.
final sum of squares of residuals : 2.77104e+09
rel. change during last iteration : -1.7081e-11

degrees of freedom (FIT_NDF) : 13
rms of residuals (FIT_STDFIT) = sqrt(WSSR/ndf) : 14599.9
variance of residuals (reduced chisquare) = WSSR/ndf : 2.13157e+08

Final set of parameters
=====
a = 0.00250053
b = 1
Asymptotic Standard Error
=====
+/- 5.769e-06 (0.2307%)
+/- 3989 (3.989e+05%)

correlation matrix of the fit parameters:
      a      b
a      1.000
b     -0.327  1.000
gnuplot>
```

## 3. EFICIENCIA EMPIRICA

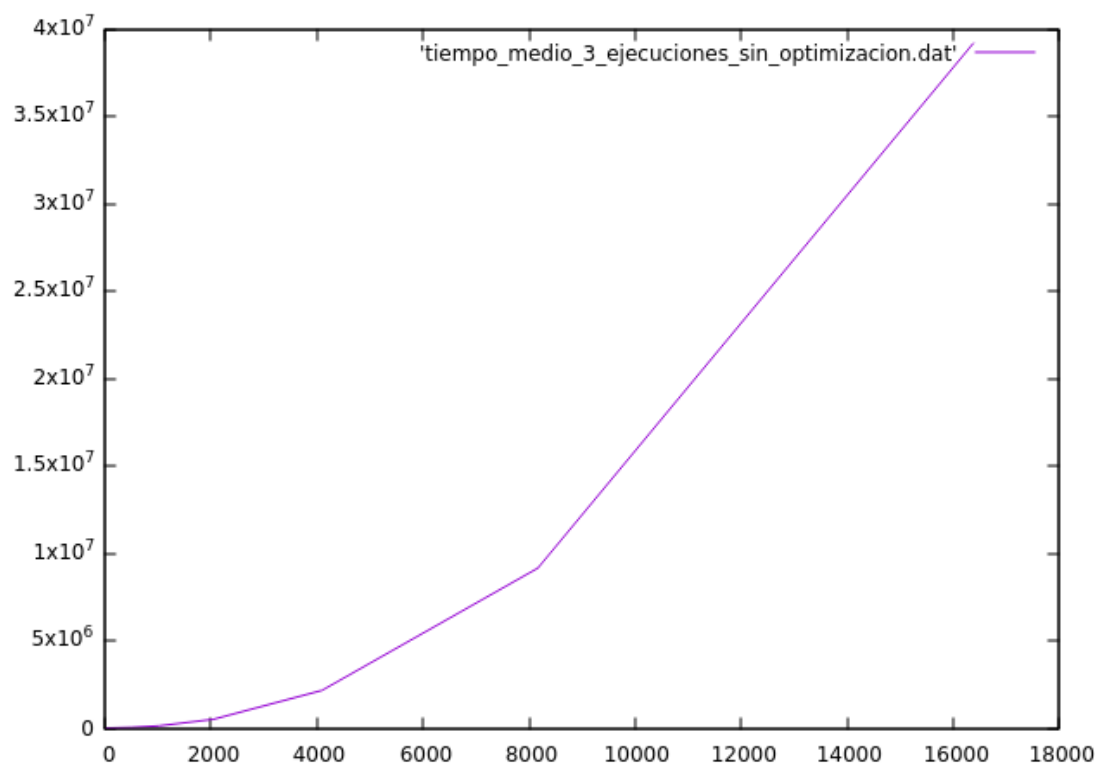
Para el cálculo de la eficiencia empírica en este caso he usado una macro como la siguiente:

```
#!/bin/bash

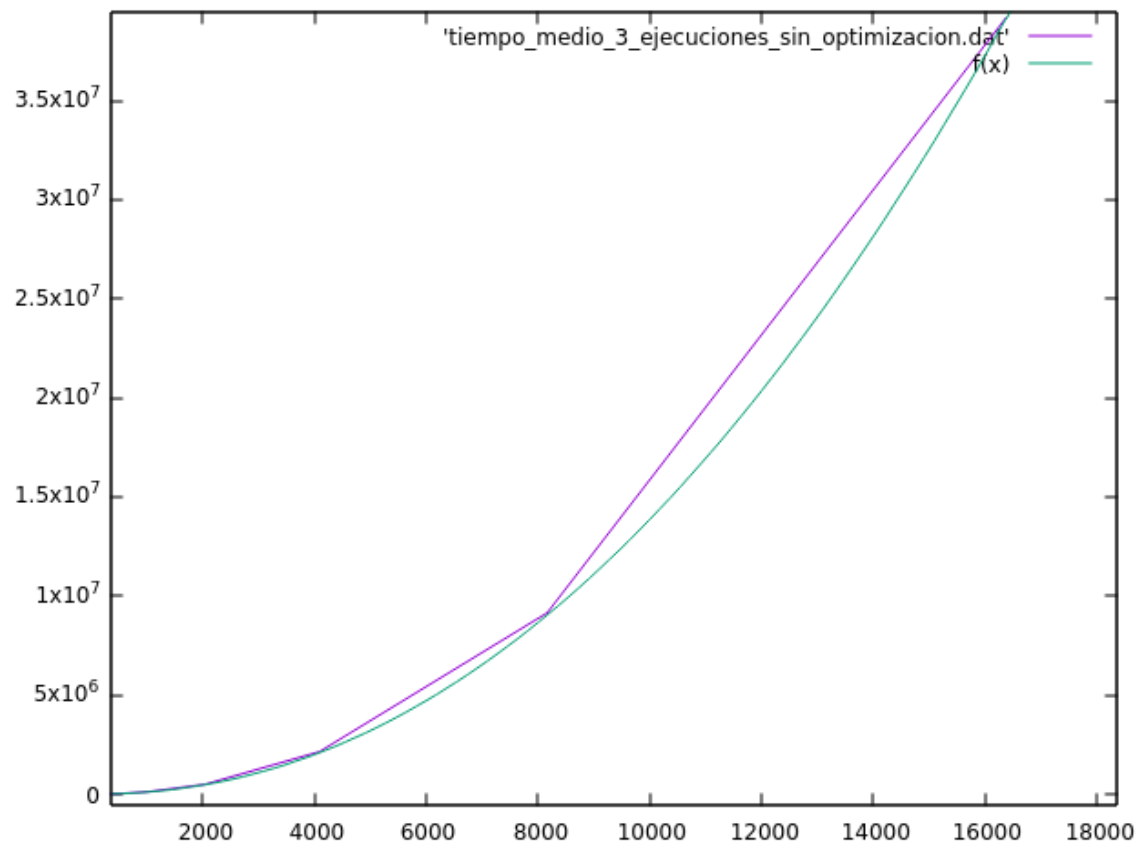
let i=1
let j=1
while [ $i -le 15 ];do
    echo $i $j
    ./divide_y_venceras $j $j >> tiempos_.dat
    let i=$i+1
    let j=$j+$j
done
```

Con el que se han obtenido 3 tiempos de ejecución diferentes para posteriormente hallar un tiempo medio de ejecución y usar ese como tiempo para dibujar la gráfica (datos recogidos en el Excel llamado “TIEMPOS\_EJECUCION\_TABLAS”). El tamaño de matriz mínimo era de 1x1 y el máximo 16384x16384. Estos son los comportamientos recogidos:

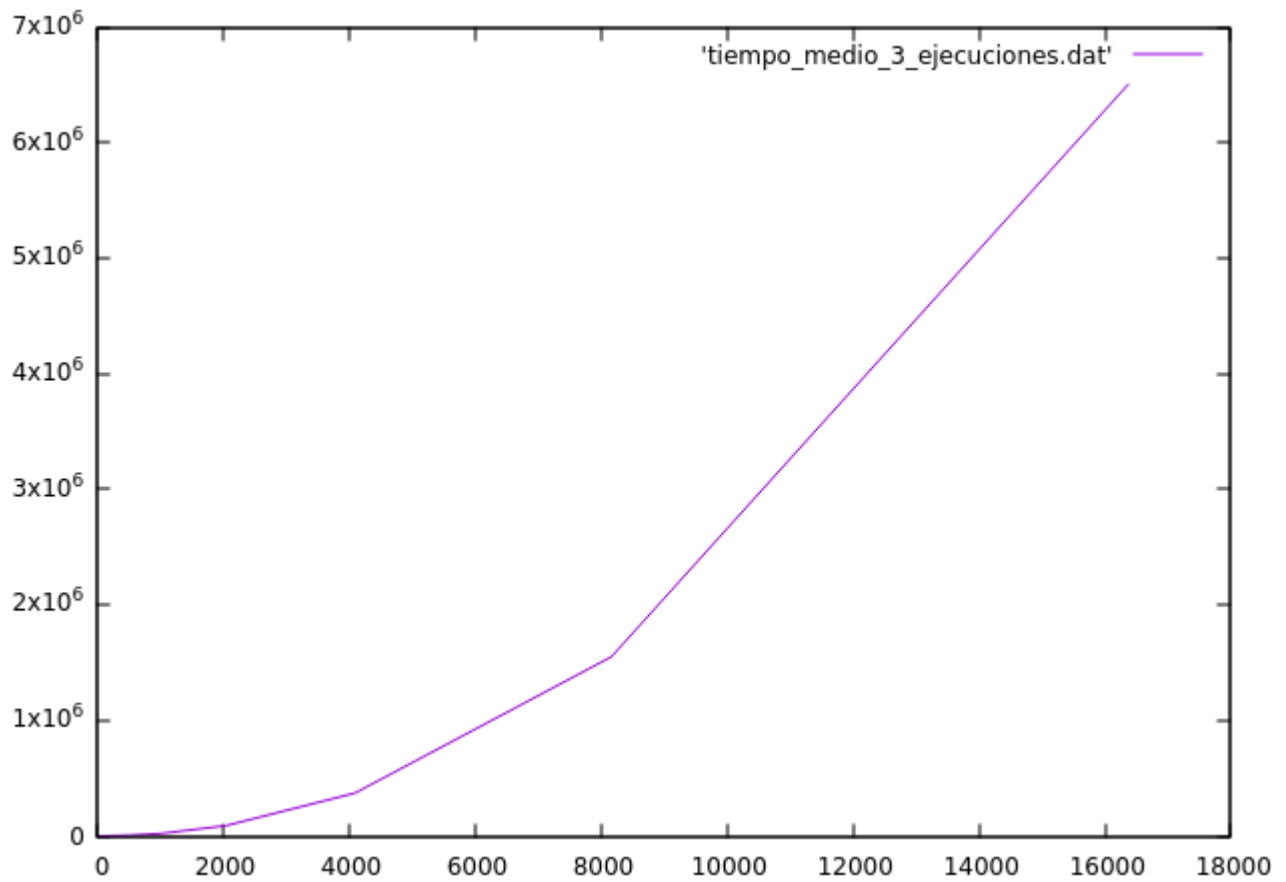
## Sin optimizar



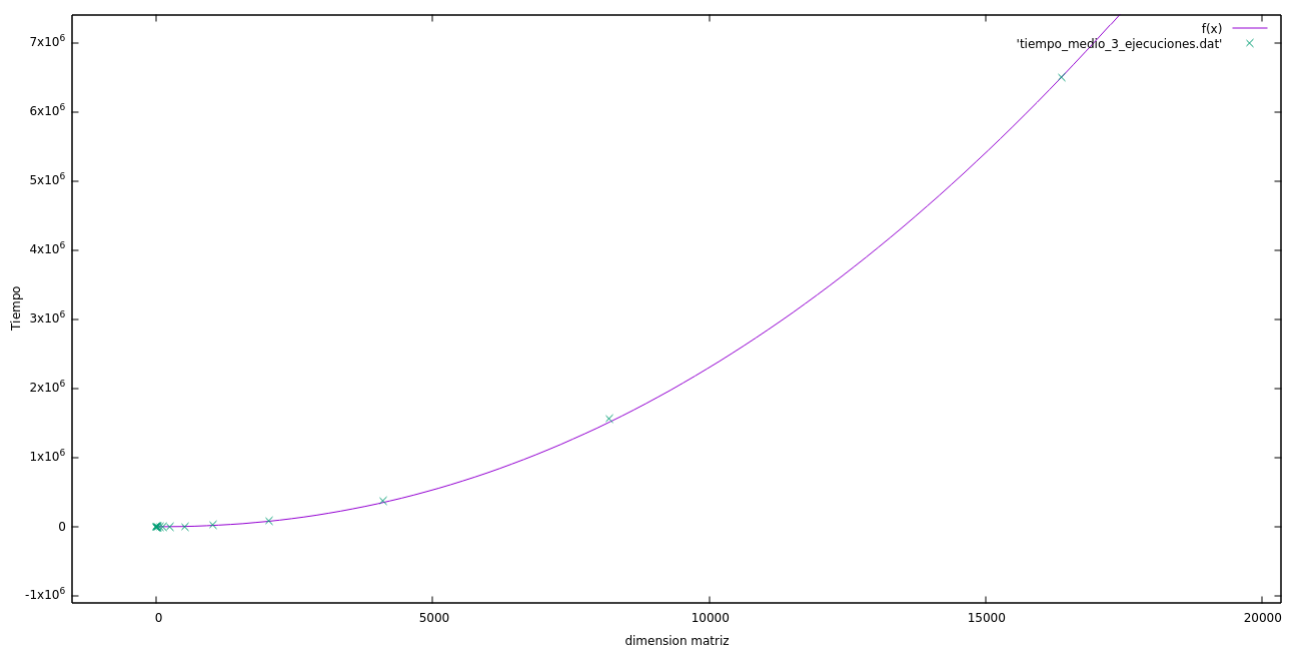
## Sin optimizar ajustada



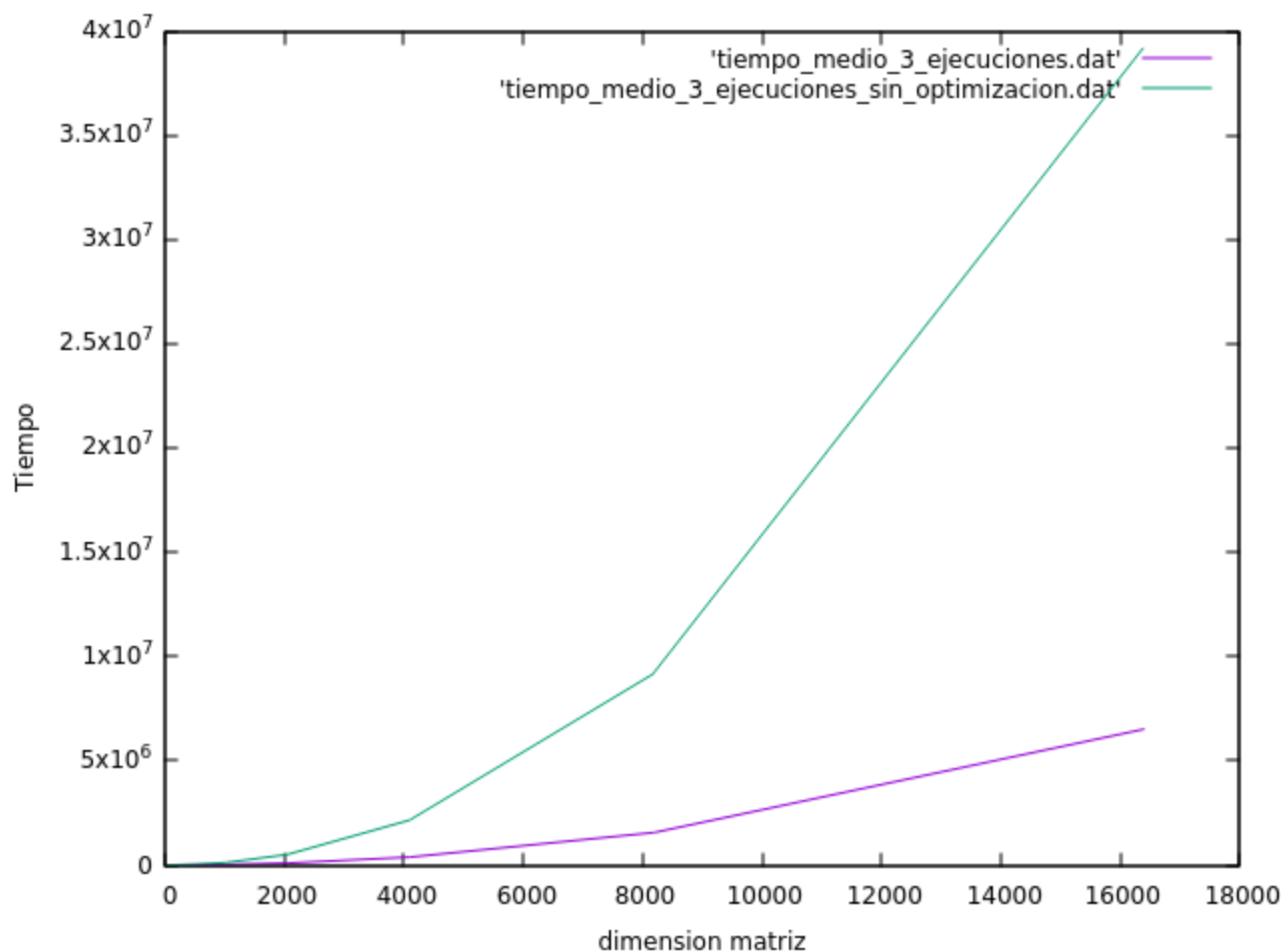
## Optimizada



## Optimizada ajustada



## Comparación de ambas



Como conclusión, se ve perfectamente que el algoritmo optimizado es mucho más rápido que el compilado sin ejecución.

## EJEMPLO DE CASO DE EJECUCION: MATRIZ 4X4

En este ejemplo se ve parte por parte lo que realiza el algoritmo de divide y vencerás. Se rellena con números aleatorios quedando una matriz original con este contenido:

3	6	7	5
3	5	6	2
9	1	2	7
6	9	3	6


Según mi programa, el procedimiento que realiza es partir la matriz en 4 porciones iguales de forma recursiva hasta llegar a la unidad mínima que es 2x2 y empezar a intercambiar la porción superior derecha por la inferior izquierda.

3	6	7	5
3	5	6	2
9	1	2	7
6	9	3	6

Las porciones entran en la recursividad una a una y como, en este caso concreto, son las unidades mínimas, ya empiezan a realizar traspuestas.

En un primer cambio, se trasponen estos valores:

3	6	7	5
3	5	6	2
9	1	2	7
6	9	3	6




3	3	7	5
6	5	6	2
9	1	2	7
6	9	3	6

Tras todos los intercambios de números dentro de matrices de 2x2, empiezan a cambiarse estas matrices completas de una posición a otra. En un principio, tras los cambios básicos, la matriz se encuentra así:

3	3	7	6
6	5	5	2
9	6	2	3
1	9	7	6

Y se intercambian los bloques de arriba a la derecha con el de abajo a la izquierda:

3	3	7	6
6	5	5	2
9	6	2	3
1	9	7	6



3	3	9	6
6	5	1	9
7	6	2	3
5	2	7	6

En las siguientes capturas, se ve paso por paso cómo el programa realiza estos cambios:

ORIGINAL:

3	6	7	5
3	5	6	2
9	1	2	7
0	9	3	6

Porcion de la matriz a intercambiar:

3	6
3	5

Porcion de la matriz intercambiada:

3	3
6	5

Asi queda la original por ahora:

3	3	7	5
6	5	6	2
9	1	2	7
0	9	3	6

Porcion de la matriz a intercambiar:

7	5
6	2

Porcion de la matriz intercambiada:

7	6
5	2

Porcion de la matriz intercambiada:

7	6
5	2

Asi queda la original por ahora:

3	3	7	6
6	5	5	2
9	1	2	7
0	9	3	6

Porcion de la matriz a intercambiar:

9	1
0	9

Porcion de la matriz intercambiada:

9	0
1	9

Asi queda la original por ahora:

3	3	7	6
6	5	5	2
9	0	2	7
1	9	3	6

Porcion de la matriz a intercambiar:

2	7
3	6

Porcion de la matriz intercambiada:

2	3
---	---

Porcion de la matriz a intercambiar:

2	7
3	6

Porcion de la matriz intercambiada:

2	3
7	6

Asi queda la original por ahora:

3	3	7	6
6	5	5	2
9	0	2	3
1	9	7	6

Porcion de la matriz a intercambiar:

3	3	7	6
6	5	5	2
9	0	2	3
1	9	7	6

Porcion de la matriz intercambiada:

3	3	9	0
6	5	1	9
7	6	2	3
5	2	7	6

Asi queda la original por ahora:

3	3	9	0
6	5	1	9
7	6	2	3

Asi queda la original por ahora:

3	3	9	0
6	5	1	9
7	6	2	3
5	2	7	6

Una vez traspuesta:

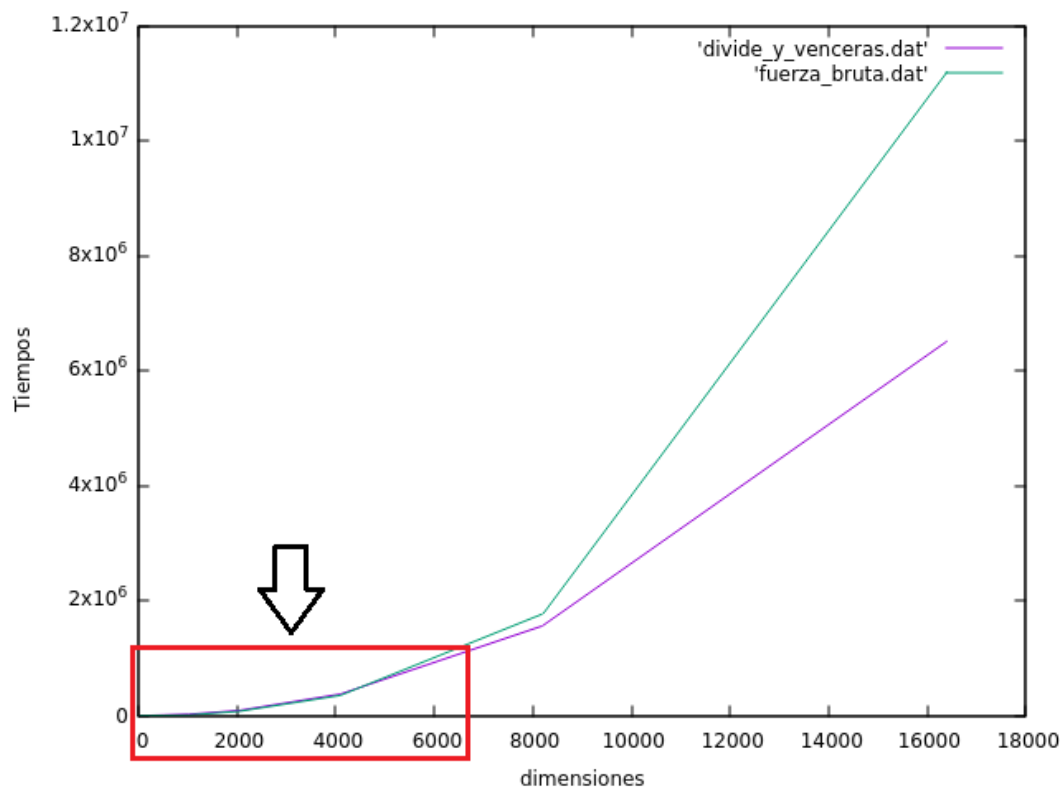
3	3	9	0
6	5	1	9
7	6	2	3
5	2	7	6

La ejecucion tarda 728 us

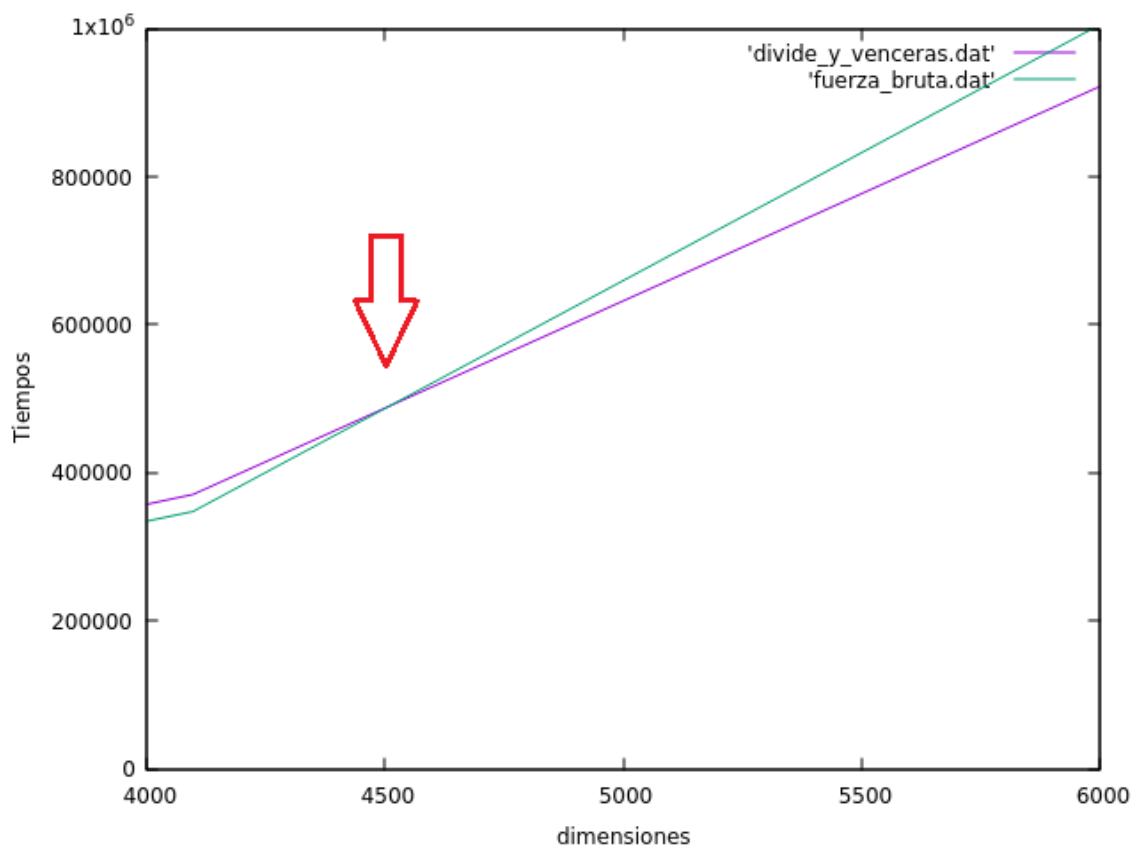


## CÁLCULO DEL VALOR UMBRAL CON EL ALGORITMO DE LA TRASPUESTA DE UNA MATRIZ

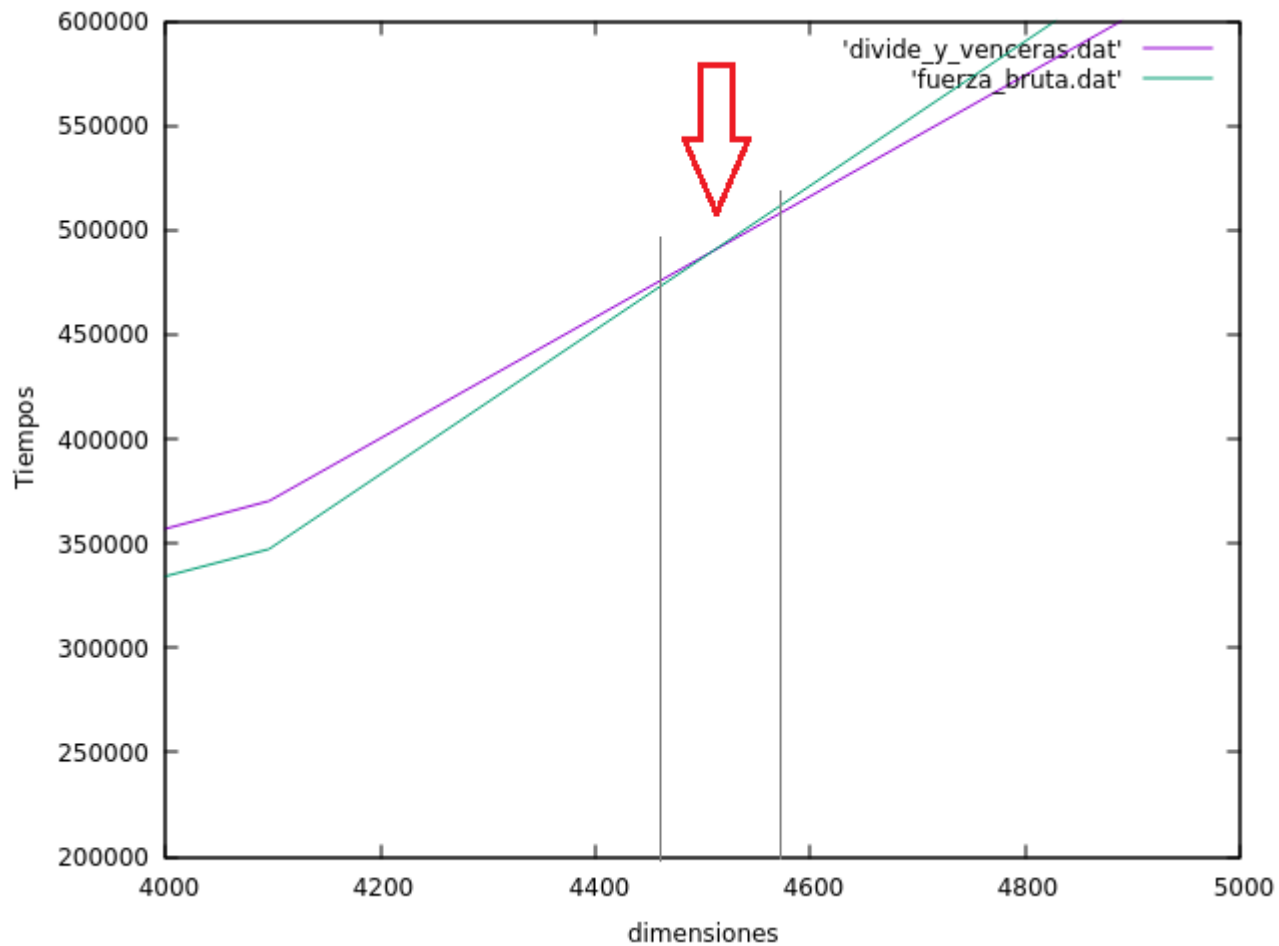
Elegir el valor umbral no es fácil porque no sólo depende de los algoritmos utilizados, sino de la implementación, lo cual hace poco posible encontrar el umbral de forma puramente teórica. Por tanto, se usa un enfoque teórico-híbrido, determinando teóricamente la forma de las ecuaciones de recurrencia y entonces encontrar empíricamente los valores de las constantes usadas en estas ecuaciones para la implementación que utilizamos. Existe otra forma de calcular este valor, una empírica. En nuestro caso, esa será la forma más útil: requiere el dibujo de ambas gráficas y observar en qué punto se cruzan.



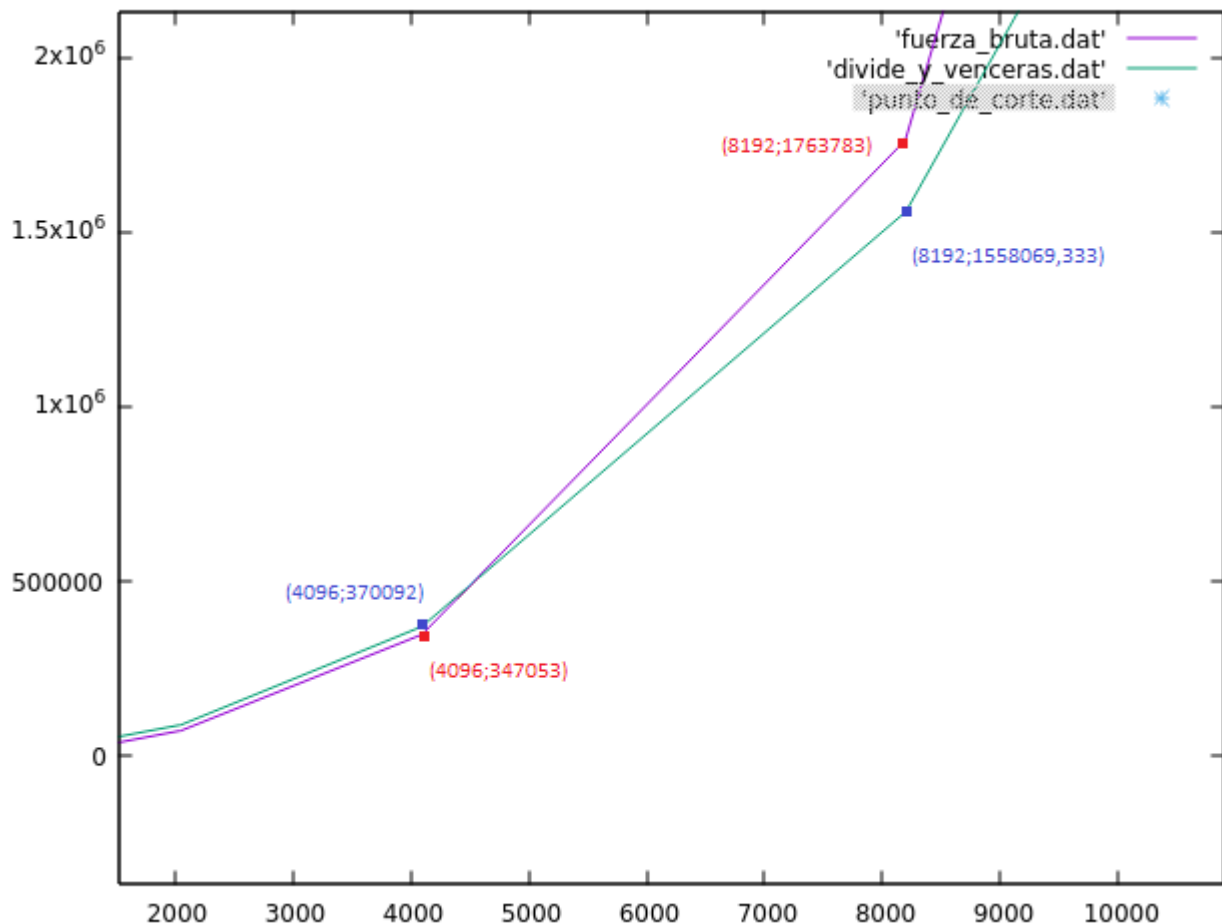
En el gráfico anterior se comparan las gráficas de la fuerza bruta y DV, observando claramente lo rápido que es el segundo algoritmo sobre el primero. En el recuadro rojo se observa una parte en la que las líneas avanzan demasiado juntas para distinguir quien va por encima o debajo, pero suficiente para saber que, de existir un cruce, es ahí. Por ello, centramos el estudio en esa parte:



Y aún podemos acotar un poco más el intervalo:



Se observa que efectivamente existe un cruce y se produce entre los valores aproximados de 4450 y 4550. Las limitaciones de zoom de GNUPLOT me impiden sacar visualmente un valor más preciso, pero se puede calcular. Para ello, se coge un punto directamente previo al cruce y otro directamente posterior en cada uno de los algoritmos. En este caso, las potencias de 2 que están antes y después del rango que estimamos son 4096 y 8192, cuyos valores en tiempo están en los archivos “divide\_y\_venceras.dat” y “fuerza\_bruta.dat”. Con esa información, se sacan los siguientes puntos:



Sacamos la ecuación de la recta que une cada par de puntos respectivo.

Ecuación de la recta:  $y = m \cdot x + n$ , donde  $m$  es la pendiente y  $n$  es el punto de intercepción en la ordenada.

La fórmula de  $m$ :  $m = (y_2 - y_1) / (x_2 - x_1)$

Para la fuerza bruta (puntos rojos):

$$m = (1763783 - 347053) / (8192 - 4096) = 1416730 / 4096 = 345,88$$

Sustituyo la  $x$  y la  $y$  en la ecuación por valores conocidos para hallar  $n$  (meto el punto (4096 ; 347053)):

$$\begin{aligned} 347053 &= 345,88 \cdot 4096 + n \\ 347053 &= 1416724,48 + n \\ n &= - 1069671,48 \end{aligned}$$

Quedando la ecuación de la recta para Fuerza Bruta tal que:

$$y = 345,88x - 1069671,48$$

Para DV (puntos azules):

$$m = (1558069,333 - 370092) / (8192 - 4096) = 1187977,333 / 4096 = 290,034$$

Sustituyo la x y la y en la ecuación por valores conocidos para hallar n (meto el punto (4096 ; 370092)):

$$\begin{aligned} 370092 &= 290,034 * 4096 + n \\ 370092 &= 1187979,264 + n \\ n &= - 817887,264 \end{aligned}$$

Quedando la ecuación de la recta para DV tal que:

$$y = 290,034x - 817887,264$$

Con ambas calculadas, puedo ver en qué punto se cortan y comprobar si coincide con el visto en la gráfica:

$$290,034x - 817887,264 = 345,88x - 1069671,48$$

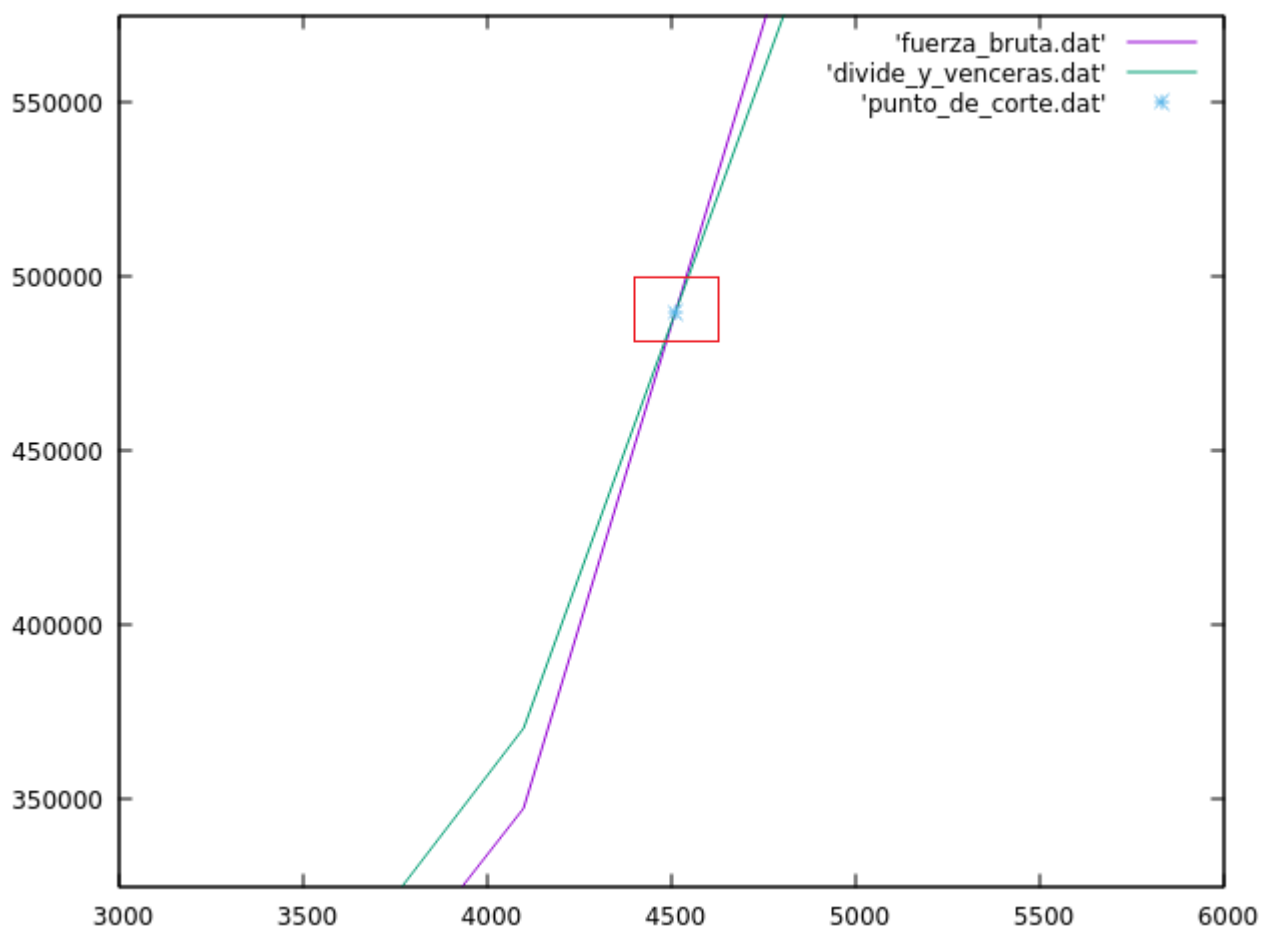
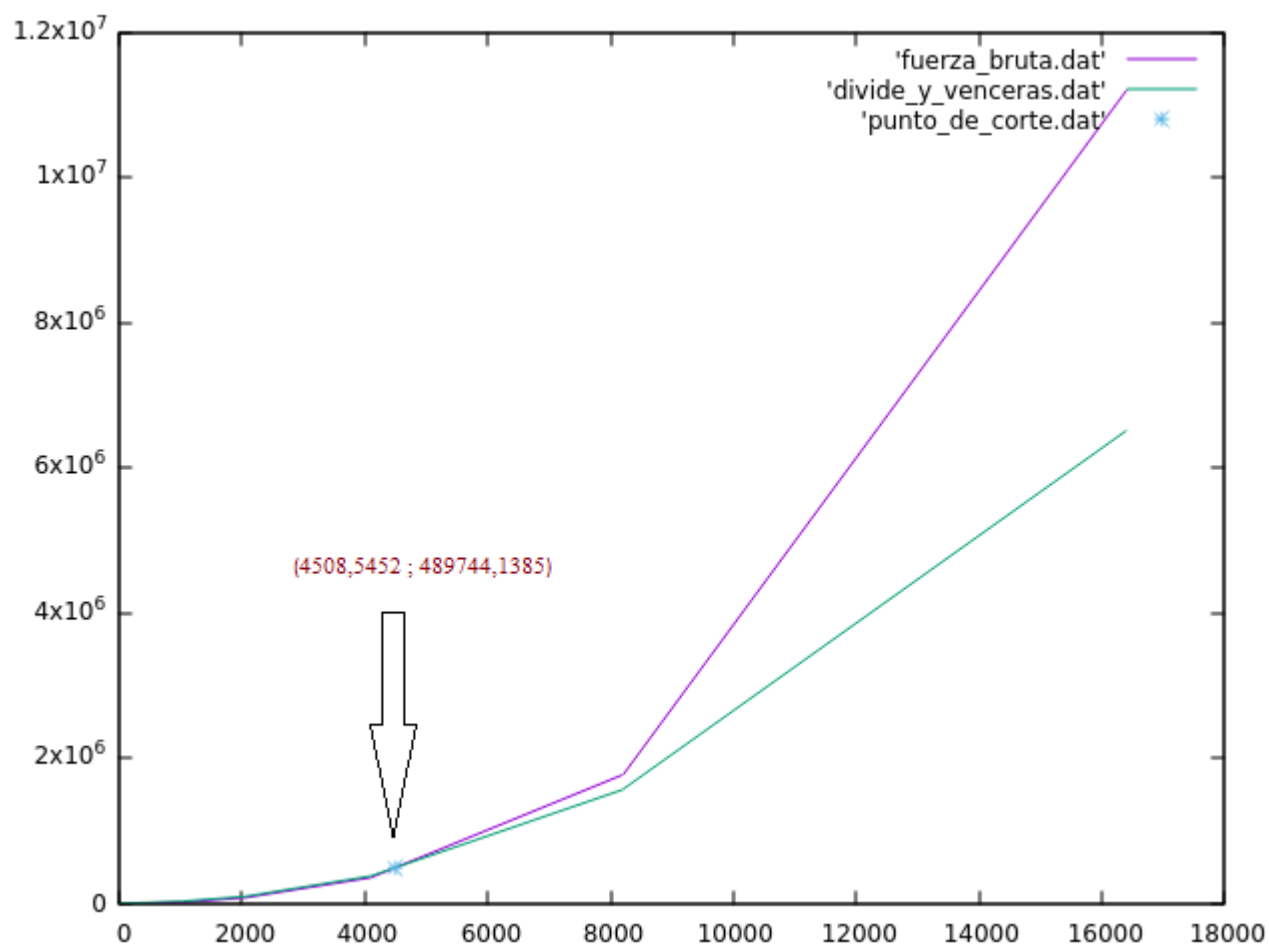
$$55,846x = 251784,216$$

$$x = 4508,5452$$

Y como es un punto en el que se cruzan ambas, ese valor de X tendrá el mismo valor de Y en las dos, por lo que sustituyo en una cualquiera (DV):

$$y = 290,034 * 4508,5452 - 817887,264 = 489744,1385$$

Por lo que aproximadamente sabemos que se cruza en el punto (4508,5452 ; 489744,1385).  
Pasamos a pintar ese punto en la gráfica para comprobar si coincide con lo observado previamente.



Como puede observarse, el punto calculado queda realmente próximo al punto en el que se ve la intersección entre las rectas, por lo que lo consideramos acertados. Ese punto marca el tamaño concreto en el que un algoritmo es más rápido que otro. Desde el principio hasta ahí, el algoritmo de fuerza bruta realizaba los cálculos en menor tiempo, pero de ese punto en adelante, el algoritmo de DV es más rápido. Como las matrices son de tamaño  $2^k$ , ese valor no es realmente un valor en el que ambos algoritmos tardan lo mismo porque no es un caso evaluado ni evaluable por ninguno de los dos. De esa forma, el tamaño justo anterior, en este caso 4096, es el último en el que fuerza bruta es mejor. A raíz de 8192, este inclusive, DV obtiene mejores resultados. Aun así, recuerdo que este valor umbral depende de la implementación de unas constantes ocultas que varían según el algoritmo y los tiempos obtenidos.