Práctica 2
Algoritmos Divide y Vencerás
UGR - ETSIIT - ALGORÍTMICA - B1

Universidad de Granada

Componentes del Grupo "Oliva"

Jose Luis Pedraza Román

Pedro Checa Salmerón

Antonio Carlos Perea Parras

Raúl Del Pozo Moreno

Índice:

1. Descripción del problema

El objetivo de esta práctica es apreciar la utilidad de la técnica “divide y vencerás” (DyV) para resolver problemas de forma diferente a otras alternativas más sencillas o directas, como los métodos de fuerza bruta.

Deberemos realizar los siguientes problemas:

- Traspuesta de una matriz: Dada una matriz cuadrada de tamaño n = 2^k, diseñar el algoritmo que devuelva la traspuesta de dicha matriz.
- Mezcla de k vectores ordenados: Se tienen k vectores ordenados (de menor a mayor), cada uno con n elementos, y queremos combinarlos en un único vector ordenado (con k*n elementos).

También se mostrarán los ajustes teóricos, empíricos e híbridos realizados para cada algoritmo mediante Gnuplot.

Matriz traspuesta ($2^k * 2^k$)*			
	Inicio	Fin	Intervalo
k	2	14	1
2^k	$2^2 = 4$	$2^{14} = 16384$	2^{k+1}

Mezcla K vectores ordenados con N enteros			
Tamaño de cada vector (N)	K inicial	K final	Intervalo
30	100	2700	104
60			
120			

Con un total de 13 muestras para la matriz traspuesta y 26 para la mezcla de vectores.
*Se usan 13 muestras debido a que al ser potencias de 2 la matriz a trasponer tarda mucho tiempo en generarse.

2. Hardware y software

```
$lscpu
Arquitectura:                    x86_64
modo(s) de operación de las CPUs:    32-bit, 64-bit
Orden de los bytes:              Little Endian
Tamaños de las direcciones:      39 bits physical, 48 bits virtual
CPU(s):                          4
Lista de la(s) CPU(s) en línea:  0-3
Hilo(s) de procesamiento por núcleo: 2
Núcleo(s) por «socket»:          2
«Socket(s)»                      1
Modo(s) NUMA:                    1
ID de fabricante:                GenuineIntel
Familia de CPU:                  6
Modelo:                          69
Nombre del modelo:               Intel(R) Core(TM) i7-4510U CPU @ 2.00GHz
```

Arch Linux x64 con i7-4510U 2GHz

```
$uname -rms
Linux 5.5.8-arch1-1 x86_64
```

A continuación, se van a exponer una serie de gráficas comparativas de las diferentes ejecuciones de los problemas propuestos, en las que veremos cómo varía el tiempo de ejecución de cada uno de ellos según los tamaños y el número de vectores en cada caso.

3. Traspuesta de una matriz

Debemos diseñar un algoritmo que devuelva la traspuesta de una matriz cuadrada de tamaño 2^k.

- Primero lo haremos por fuerza bruta, donde recorreremos la matriz inferior y realizará el intercambio de valores cambiando las coordenadas.
- También haremos este problema usando Divide y Vencerás, para lo cual iremos dividiendo el número de filas y columnas entre dos de forma recurrente hasta llegar al caso base, el cual colocaremos en el sitio correcto de la matriz traspuesta.

3.1. Ejemplo de ejecución

FUERZA BRUTA

```
$./bin/matrix_bruta 2
Matriz original
1       2       3       4
5       6       7       8
9       10      11      12
13      14      15      16

Cambiando F1C0 (2) con F0C1 (5)
1       5       3       4
2       6       7       8
9       10      11      12
13      14      15      16

Cambiando F2C0 (3) con F0C2 (9)
1       5       9       4
2       6       7       8
3       10      11      12
13      14      15      16

Cambiando F3C0 (4) con F0C3 (13)
1       5       9       13
2       6       7       8
3       10      11      12
4       14      15      16

Cambiando F2C1 (7) con F1C2 (10)
1       5       9       13
2       6       10      8
3       7       11      12
4       14      15      16

Cambiando F3C1 (8) con F1C3 (14)
1       5       9       13
2       6       10      14
3       7       11      12
4       8       15      16

Cambiando F3C2 (12) con F2C3 (15)
1       5       9       13
2       6       10      14
3       7       11      15
4       8       12      16
```

En la imagen izquierda se ve una ejecución para una matriz de tamaño 2^2, es decir, una matriz 4x4 con 16 elementos.

En cada paso se puede ver la posición que intercambia indicando la fila y columna de cada valor intercambiado.

Se vé como en cada iteración, va bajando por la columna 0 intercambiando sus valores con la fila 0, al terminar, pasa a la columna 1 intercambiando sus valores con la fila 1, este recorrido corresponde con la matriz inferior, que se intercambia con la matriz superior, dejando la diagonal intacta.

DIVIDE VENCERAS

```
g++ -std=gnu++0x src/matrix_DyV.cpp -o bin/matrix_DyV

Matriz origen
1       2       3       4
5       6       7       8
9       10      11      12
13      14      15      16

Trasponiendo F0C0(1) en F0C0 (matriz traspuesta)
Trasponiendo F0C1(2) en F1C0 (matriz traspuesta)
Trasponiendo F1C0(5) en F0C1 (matriz traspuesta)
Trasponiendo F1C1(6) en F1C1 (matriz traspuesta)

Trasponiendo F0C2(3) en F2C0 (matriz traspuesta)
Trasponiendo F0C3(4) en F3C0 (matriz traspuesta)
Trasponiendo F1C2(7) en F2C1 (matriz traspuesta)
Trasponiendo F1C3(8) en F3C1 (matriz traspuesta)

Trasponiendo F2C0(9) en F0C2 (matriz traspuesta)
Trasponiendo F2C1(10) en F1C2 (matriz traspuesta)
Trasponiendo F3C0(13) en F0C3 (matriz traspuesta)
Trasponiendo F3C1(14) en F1C3 (matriz traspuesta)

Trasponiendo F2C2(11) en F2C2 (matriz traspuesta)
Trasponiendo F2C3(12) en F3C2 (matriz traspuesta)
Trasponiendo F3C2(15) en F2C3 (matriz traspuesta)
Trasponiendo F3C3(16) en F3C3 (matriz traspuesta)


Matriz traspuesta
1       5       9       13
2       6       10      14
3       7       11      15
4       8       12      16
```

En la imagen superior se puede observar una ejecución del algoritmo divide y vencerás para calcular la traspuesta de una matriz, se ve que la ejecución está dividida en 4 bloques separados por una línea en blanco, cada bloque corresponde con una llamada recursiva del algoritmo.

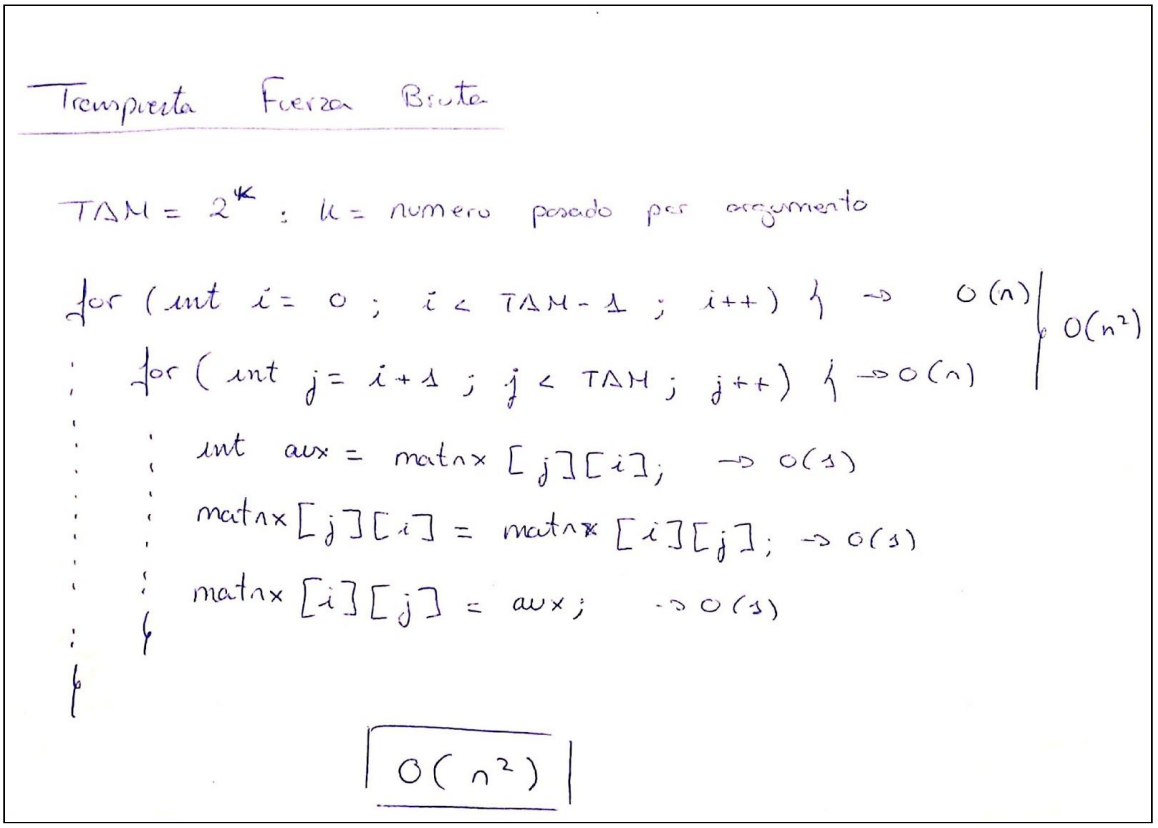
Como se tiene una matriz de 4x4, la primera llamada se encarga de una parte de la matriz de 4 elementos, por lo que se estarían realizado 4 llamadas recursivas en las que cada llamada se encarga de 4 elementos de la matriz.

Asimismo, cada llamada recursiva se repite hasta que la llamada tiene nada más que 1 elemento de la matriz, en ese caso entra al caso base y aparece el mensaje de Trasponiendo la fila y columna origen a la fila y columna (invertida del origen) destino.

3.2. Análisis teórico

En la imagen siguiente se puede ver el análisis teórico de la eficiencia para fuerza bruta de la matriz traspuesta, consiste en dos bucles for anidados que recorren la matriz inferior, ya que la matriz superior es la opuesta a esta solo hace falta acceder alternando [i][j] por [j][i], además la diagonal es inmutable ya que no se intercambia con ella misma. Como cada bucle tiene una eficiencia de O(n), al estar anidada seria O(n^2).

El código de esta solución se puede ver en el fichero "**matrix_bruta.cpp**".

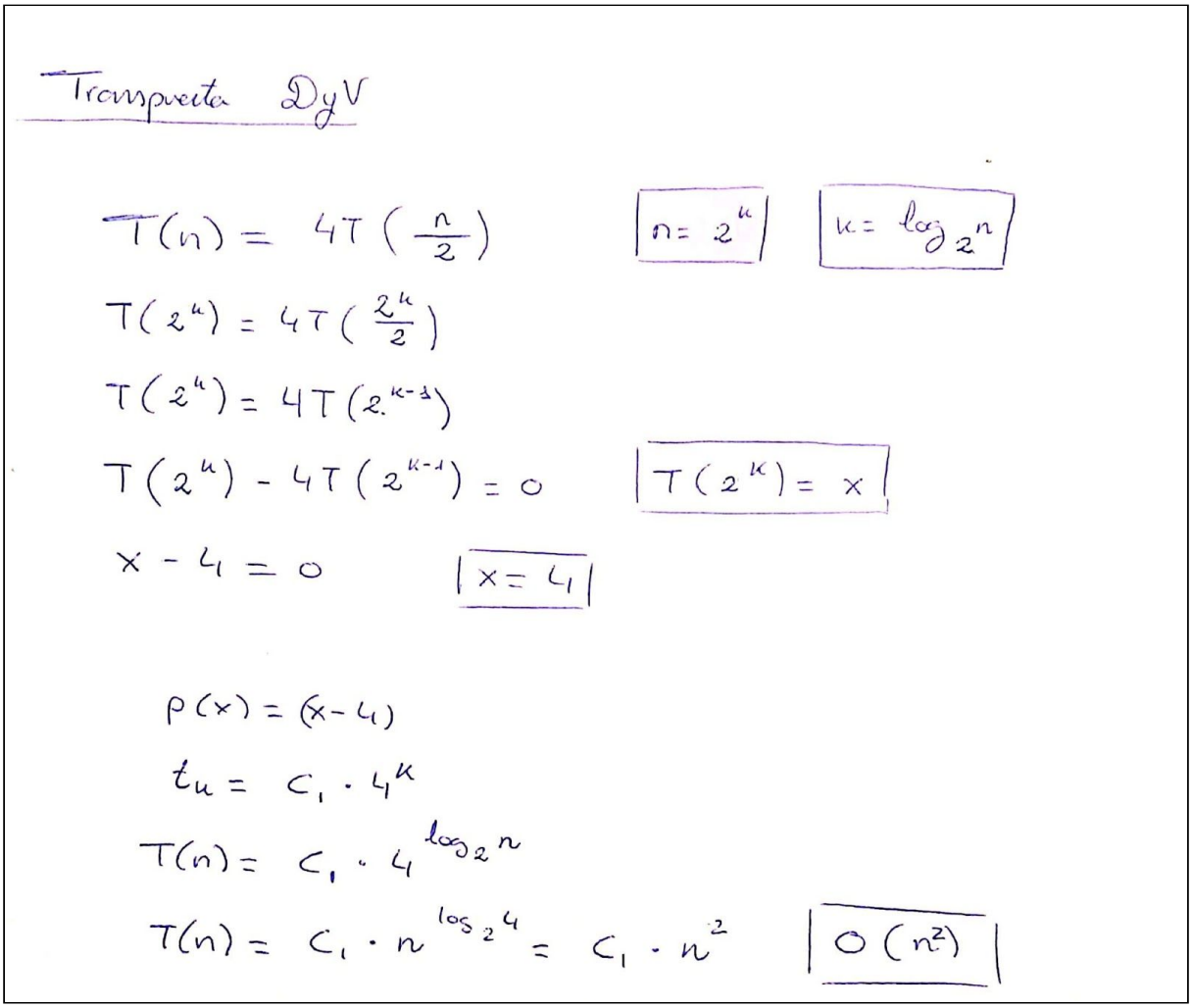


A continuación se puede ver el análisis teórico de la solución por el algoritmo de Divide y Vencerás, el cual se obtiene desarrollando la recurrencia obtenida del código, el cual se puede ver en el fichero "**matrix_DyV.cpp**".

Este algoritmo se ha desarrollado haciendo una llamada recursiva en la cual se distinguen dos opciones:

- Caso base: la submatriz es de 1x1 elementos (los índices de rangos son los mismos)
- Caso recursivo: La submatriz es diferente a 1x1 elementos.

Dicha llamada recursiva se llama 4 veces a sí misma, dividiendo el tamaño del problema por la mitad ya que divide el número de filas y columnas a la mitad. Resolviendo la parte homogénea de la recurrencia se obtiene una eficiencia O(n^2).

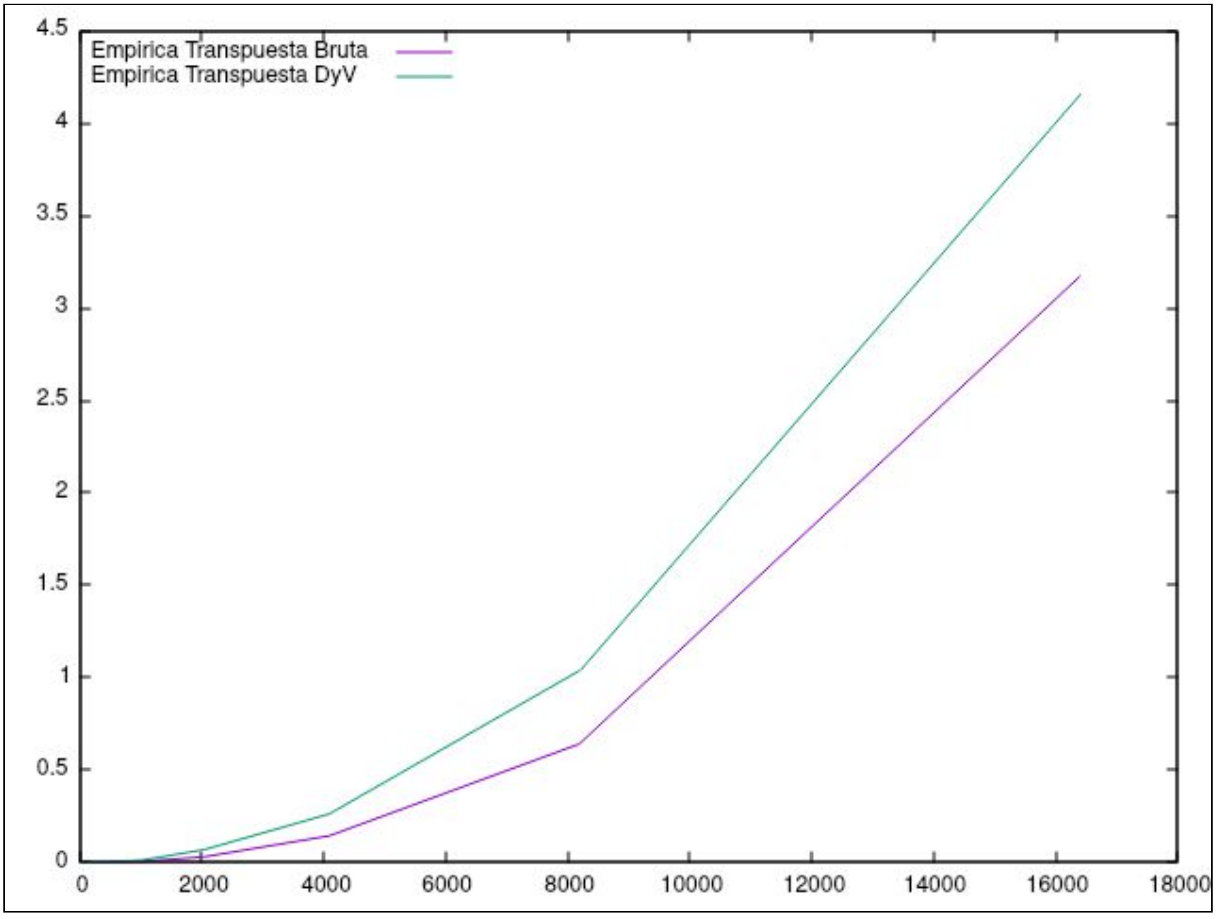


3.3. Análisis empírico

Para el análisis empírico se ha utilizado un .sh en el que se ejecuta para el rango de tamaños indicados en la tabla del punto 1 (página 3).

Como se puede, las curvas tienen mucha similitud, la cual parece que corresponde con su eficiencia teórica, ya que hace la curva de una función cuadrática. Se puede ver que los tiempos obtenidos mediante el algoritmo Divide y Vencerás son peores que los tiempos obtenidos por Fuerza Bruta, esto tiene sentido ya que el problema es simple, y es más eficiente accediendo a posiciones directamente que haciendo llamadas recursivas y disminuyendo el tamaño del problema, ya al final, va a realizar el mismo cambio que por fuerza bruta.

Esto tiene sentido debido a que es más eficiente ejecutar iteraciones de un bucle “for” en el que solo tiene que aumentar la variable contadora; mientras que mediante “Divide y Vencerás” tiene que realizar una llamada a un método y dividirse recursivamente hasta llegar al caso base, que realiza la misma operación que el contenido de los bucles “for” anidados de fuerza bruta.



3.4. Análisis híbrido

Para realizar el análisis híbrido, se ha usado una función cuadrática para ambos algoritmos para comprobar si el análisis teórico se ajusta a su curva empírica.

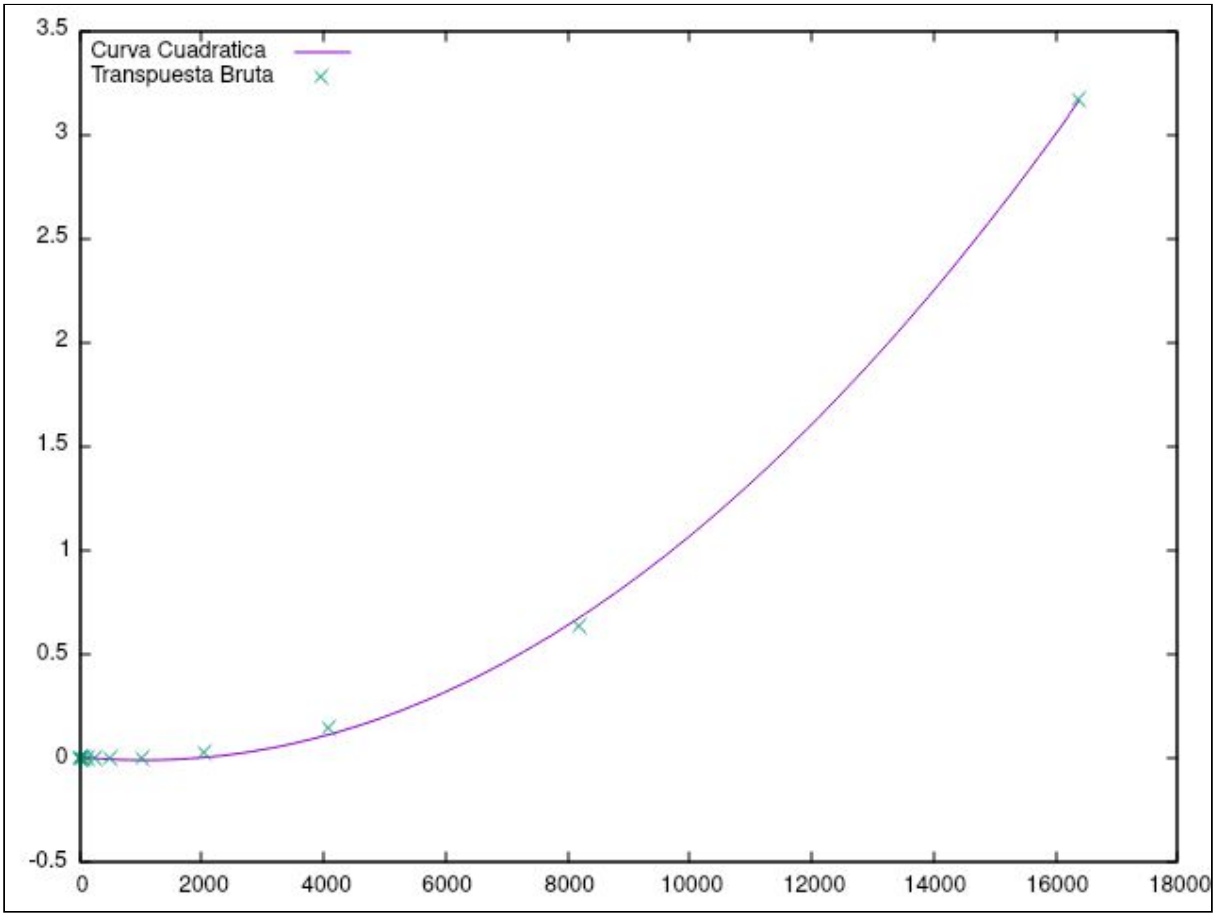
$$f(x) = a0*x*x+a1*x+a2$$
$$g(x) = b0*x*x+b1*x+b2$$

En las imágenes de los ajustes obtenidos por gnuplot aparece en ambas 'a0' porque se han realizado por separado

Como se puede ver, los coeficientes de correlación obtenidos son muy cercanos a 1 y los errores bajos.

Siguiendo los pasos realizados en la práctica 1, se ha establecido en gnuplot una función cuadrática, se le ha realizado el "fit" a cada algoritmo obteniendo y se han graficado.

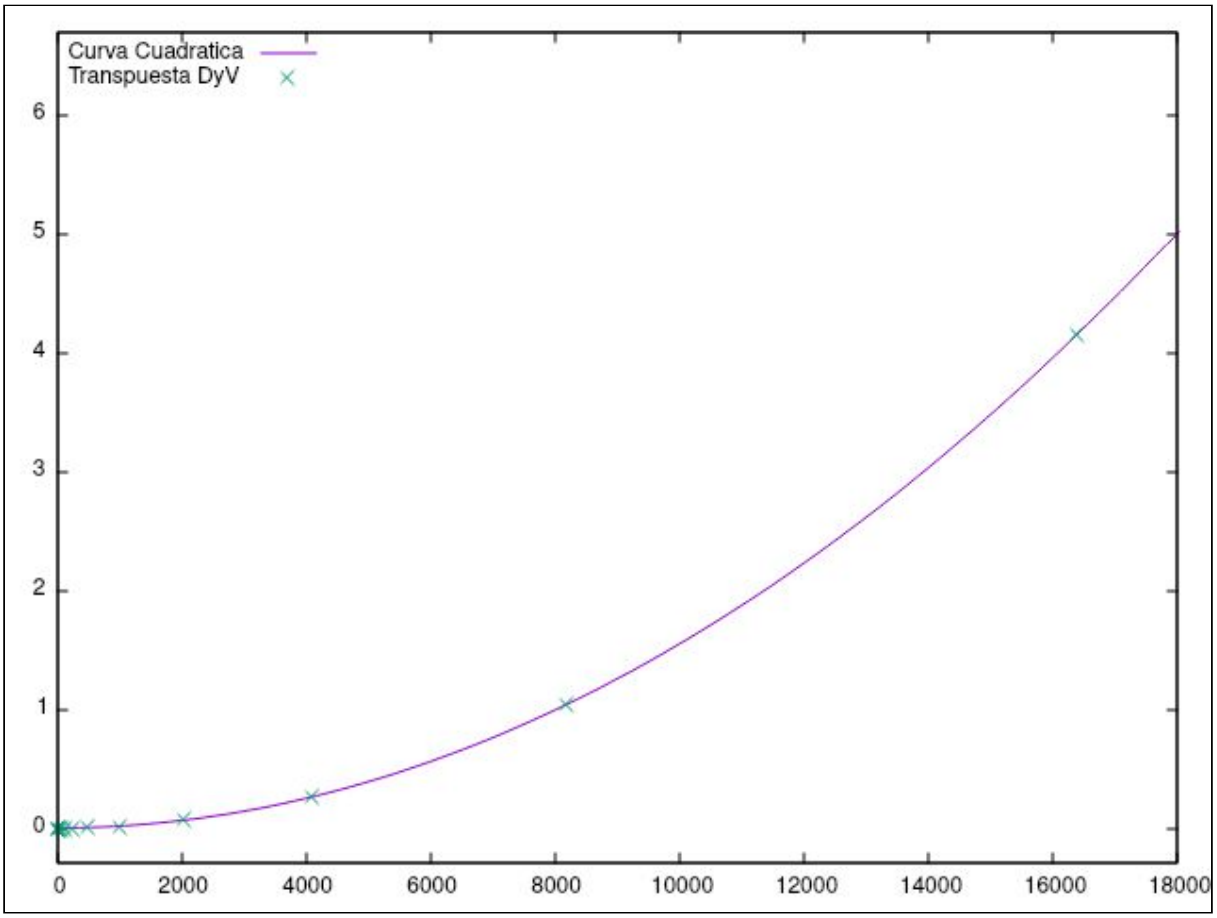
A continuación se puede ver el ajuste realizado para el algoritmo por fuerza bruta, como se observa, la línea de la función cuadrática pasa por los puntos de tiempos obtenidos por la ejecución empírica, con un error del 1'848%



```
Final set of parameters            Asymptotic Standard Error
=======================            ==========================
a0              = 1.35038e-08      +/- 2.496e-10    (1.848%)
a1              = -2.88691e-05     +/- 3.894e-06    (13.49%)
a2              = 0.00807891       +/- 0.006252     (77.39%)

correlation matrix of the fit parameters:
                a0     a1     a2
a0              1.000
a1             -0.960  1.000
a2              0.408 -0.514  1.000
```

Respecto al algoritmo Divide y Vencerás, también se puede ver que la función cuadrática pasa por los puntos obtenidos por la ejecución empírica, con un error del 0’1986%.



```
Final set of parameters            Asymptotic Standard Error
=======================            ==========================
a0              = 1.5432e-08       +/- 3.064e-11    (0.1986%)
a1              = 5.72143e-07      +/- 4.78e-07     (83.55%)
a2              = 7.15507e-05      +/- 0.0007675    (1073%)

correlation matrix of the fit parameters:
                a0     a1     a2
a0              1.000
a1             -0.960  1.000
a2              0.408 -0.514  1.000
```

3.5. Umbral

Calculamos el umbral mediante la igualdad de las dos ecuaciones híbridas obtenidas en gnuplot al realizar los ajustes:

DyV	$f(x) = 1'5432 * 10^{-8}x^2 + 5'72143 * 10^{-7}x + 7'15507 * 10^{-5}$
F. Bruta	$g(x) = 1'35038 * 10^{-8}x^2 - 2'88691 * 10^{-5}x + 0.00807891$

Al realizar la igualdad de f(x) = g(x) y resolver la ecuación de segundo grado resultante, obtenemos dos valores:

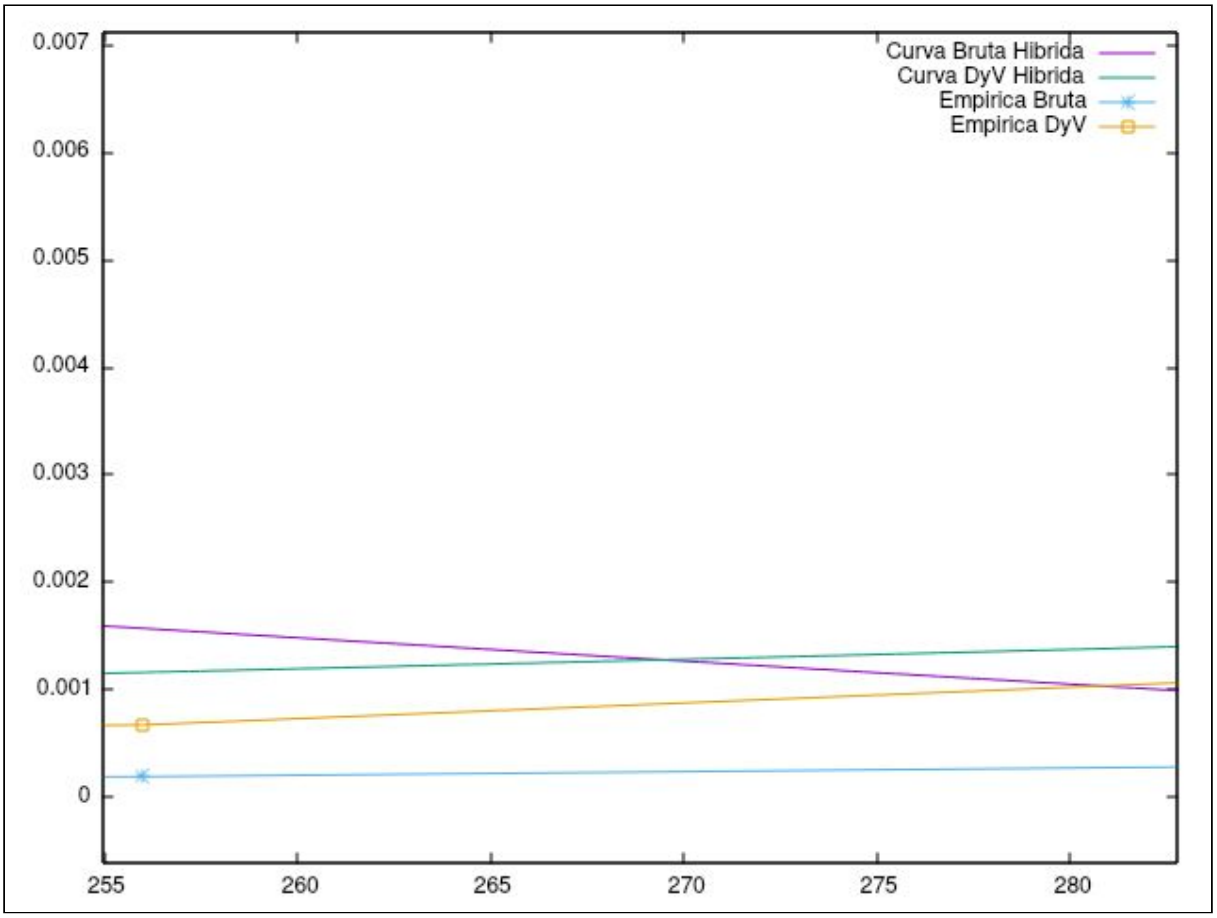
x1	267’298
x2	-15536.1

En el enlace siguiente se puede observar la solución de la igualdad f(x) = g(x):

https://www.wolframalpha.com/input/?i=%281.5432*10%5E%28-8%29%29*x%5E2%2B%285.72143*10%5E%28-7%29%29*x%2B%287.15507*10%5E%28-5%29%29%3D%281.35038*10%5E%28-8%29%29*x%5E2-%282.88691*10%5E%28-5%29%29*x%2B%280.00807891%29

De los dos resultados obtenidos, el valor negativo se rechaza al estar trabajando con valores positivos, quedando una única solución que es x1, al graficar las eficiencias híbridas, se puede observar que existe un corte en dicho tamaño en las curvas híbridas, pero que no en las empíricas. Esto se puede deber al error existente en el ajuste realizado ya que existen errores del 1.88% para la eficiencia bruta y 0.2% para Divide y Vencerás.

Por tanto hasta matrices de 267 elementos, es mejor una eficiencia por Divide y Vencerás mientras que a partir de dicho tamaño es mejor fuerza bruta.



4. Mezclando K vectores ordenados

Debemos diseñar un algoritmo tal que, dados k vectores de n enteros ordenados, nos permite mezclarlos en un vector de tamaño k*n de forma ordenada.

- Para el algoritmo de fuerza se inicializa un vector solución con el primer vector de la matriz, y se va actualizando mediante la comparación del vector solución con el siguiente vector de la matriz, código en fichero "**mezcla_bruta.cpp**"

- Por otro lado, con el algoritmo Divide y Vencerás, se divide el número de vectores por la mitad (no el tamaño de cada vector, sino vectores en su totalidad) hasta que se llega al caso base de tener 1 solo vector por llamada recursiva, en este caso se añade al vector solución y cuando las llamadas recursivas van terminando, se reordenan los vectores en el rango de vectores añadidos al vector solución hasta ese momento, de forma que va ordenando los vectores a la izquierda, luego a la derecha, en este punto la parte izquierda está ordenada y la parte derecha también, por lo que termina ordenando dichas partes, código en fichero "**mezcla_DyV.cpp**".

4.1. Ejemplo de ejecución

FUERZA BRUTA

```
g++ -std=gnu++0x src/mezcla_bruta.cpp -o bin/mezcla_bruta
Matriz a ordenar
3       5       6
4       6       8
2       4       5


Vector solucion:        3 5 6
Vector a comparar:      4 6 8

Compara: 3 con 4  -> Añade el 3
Compara: 5 con 4  -> Añade el 4
Compara: 5 con 6  -> Añade el 5
Compara: 6 con 6  -> Añade el 6
Reañade el 6
Añade elementos restantes vector comparado: 8


Vector solucion:        3 4 5 6 6 8
Vector a comparar:      2 4 5

Compara: 3 con 2  -> Añade el 2
Compara: 3 con 4  -> Añade el 3
Compara: 4 con 4  -> Añade el 4
Reañade el 4
Compara: 5 con 5  -> Añade el 5
Reañade el 5
Añade elementos restantes vector solucion previo: 6 6 8

Vector solucion final ordenado
2 3 4 4 5 5 6 6 8
```

En la imagen superior se puede observar un caso práctico de ejecución por fuerza bruta para el algoritmo de mezcla K-vectores.

Al principio se muestra la matriz a ordenar, acto seguido inicializa un vector solución con el primer vector de la matriz (fila 0) y lo compara con el segundo vector de la matriz (fila 1).

A continuación se compara elemento a elemento, cada elemento de la fila 0, con los demás elementos de la fila 1, en el caso de que el elemento de la fila 0 sea menor, se añade y pasa al siguiente elemento, en el caso de que el elemento de la fila 1 sea menor, se añade este y se continúa con el siguiente. En el caso de que dichos elementos comparados sean iguales, se añaden los dos valores.

En cualquier caso, las comparaciones se hacen respecto al último elemento comparado, para no volver a comparar con elementos ya comparados.

Finalmente, al terminar de añadir todos los elementos de un vector, añade el resto de elementos del vector con elementos no añadidos.

Este proceso se repite pero en cada repetición, el vector solución que se compara, contiene N*i elementos, donde N es el tamaño del vector a comparar e “i” el número de vectores añadidos a la solución.

Al final se obtiene un vector solución de K*N con todos los elementos de la matriz ordenados de menor a mayor.

DIVIDE Y VENCERÁS

```
g++ -std=gnu++0x src/mezcla_DyV.cpp -o bin/mezcla_DyV
Matriz a ordenar
1       5       7
3       4       8
3       4       6

Caso base, añade añade array: 0 { 1, 5, 7 }
        Array solucion tiene:   { 1, 5, 7, -1, -1, -1, -1, -1, -1 }

Caso base, añade añade array: 1 { 3, 4, 8 }
        Array solucion tiene:   { 1, 5, 7, 3, 4, 8, -1, -1, -1 }

Ordena array izquierdo: { 1, 5, 7 }
Ordena array derecho:   { 3, 4, 8 }

Compara 1 con 3 -> Añade 1
Compara 5 con 3 -> Añade 3
Compara 5 con 4 -> Añade 4
Compara 5 con 8 -> Añade 5
Compara 7 con 8 -> Añade 7
        Añade resto der 8

Caso base, añade añade array: 2 { 3, 4, 6 }
        Array solucion tiene:   { 1, 3, 4, 5, 7, 8, 3, 4, 6 }

Ordena array izquierdo: { 1, 3, 4, 5, 7, 8 }
Ordena array derecho:   { 3, 4, 6 }

Compara 1 con 3 -> Añade 1
Compara 3 con 3 -> Añade 3
Compara 4 con 3 -> Añade 3
Compara 4 con 4 -> Añade 4
Compara 5 con 4 -> Añade 4
Compara 5 con 6 -> Añade 5
Compara 7 con 6 -> Añade 6
        Añade resto izq 7
        Añade resto izq 8


Array solucion final ordenado
1 3 3 4 4 5 6 7 8
```

En la imagen superior se puede observar un caso práctico de ejecución por divide y vencerás para el algoritmo de mezcla K-vectores.

Este algoritmo realiza dos llamadas recursivas repartiéndose los vectores de la matriz hasta que llegan al caso base. Este caso base se cumple cuando la llamada recursiva se encargue solamente de un vector, en dicho caso, añade al vector solución el vector dado.

Se inicializa el vector solución con valores “-1” para que se vea el proceso de relleno del vector solución.

Así, en esta matriz de 3*3, la primera llamada recursiva desde el main contiene el vector solución vacío.

En esta primera llamada recursiva, se llama dos veces recursivamente:

1. Primera llamada: Desde la fila 0 a la fila 1 incluida
2. Segunda llamada: Solamente la fila 2 (ésta se ejecuta cuando la primera termina)

Así, la primera llamada recursiva (que contiene 2 filas de vectores) se vuelve a llamar recursivamente dos veces y, en este caso, cada llamada recursiva se encarga de una sola fila, por lo que cada una entra al caso base e inserta en el vector solución dichos vectores.

En este punto, se tiene que el vector solución está relleno ⅔ del total de forma no ordenada, como se puede en la imagen con los valores: { 1, 5, 7, 3, 4, 8, -1,-1, -1}

Al terminar estas dos llamadas recursivas del punto, ordena los valores almacenados de la misma forma que hacia con la fuerza bruta (ya que es ordenación simple).

Una vez que termina la primera llamada recursiva (fila 0 y fila 1) se llama la segunda llamada recursiva que contiene solamente la fila 2 de la matriz. Esta llamada entra al caso base, añade la fila 2 a la parte de la solución.solución que aún no se ha rellenado, quedando un vector solución, con una parte izquierda ordenada (fila 0 y fila 1) y una parte derecha también ordenada (fila 2).

Una vez que termina de añadir la fila 2, termina la ejecución de la llamada recursiva y ordena las dos partes ordenadas del vector solución.

Finalmente se obtiene un vector solución ordenado con los valores de la matriz.

4.2. Análisis teórico

A continuación, se puede ver la obtención de la eficiencia teórica para el algoritmo por fuerza bruta para este problema, como se ve, hay un bucle “for” principal que recorre el número de vectores menos uno con eficiencia O(n). Dentro tiene dos partes, un bucle “for” con eficiencia O(n) ya que se encarga de actualizar la solución y una llamada a una función.

Esta función “comparaVectores” se encarga de comparar dos vectores, de igual o diferente tamaño, para realizar esto tiene que comparar ambos vectores mediante dos for anidados, esta comparación se realiza intentando que cada elemento se compara con el último comparado del otro vector, de forma que evita hacer comparaciones con elementos ya añadidos. En el peor caso tiene que recorrer los dos vectores dando una eficiencia O(n^2).

De esta forma se tienen en el peor caso una eficiencia O(n^3) ya que habría 3 for anidados.

Mezcla Fuerza Bruta - " k vectores de N elementos " $O(n^3)$

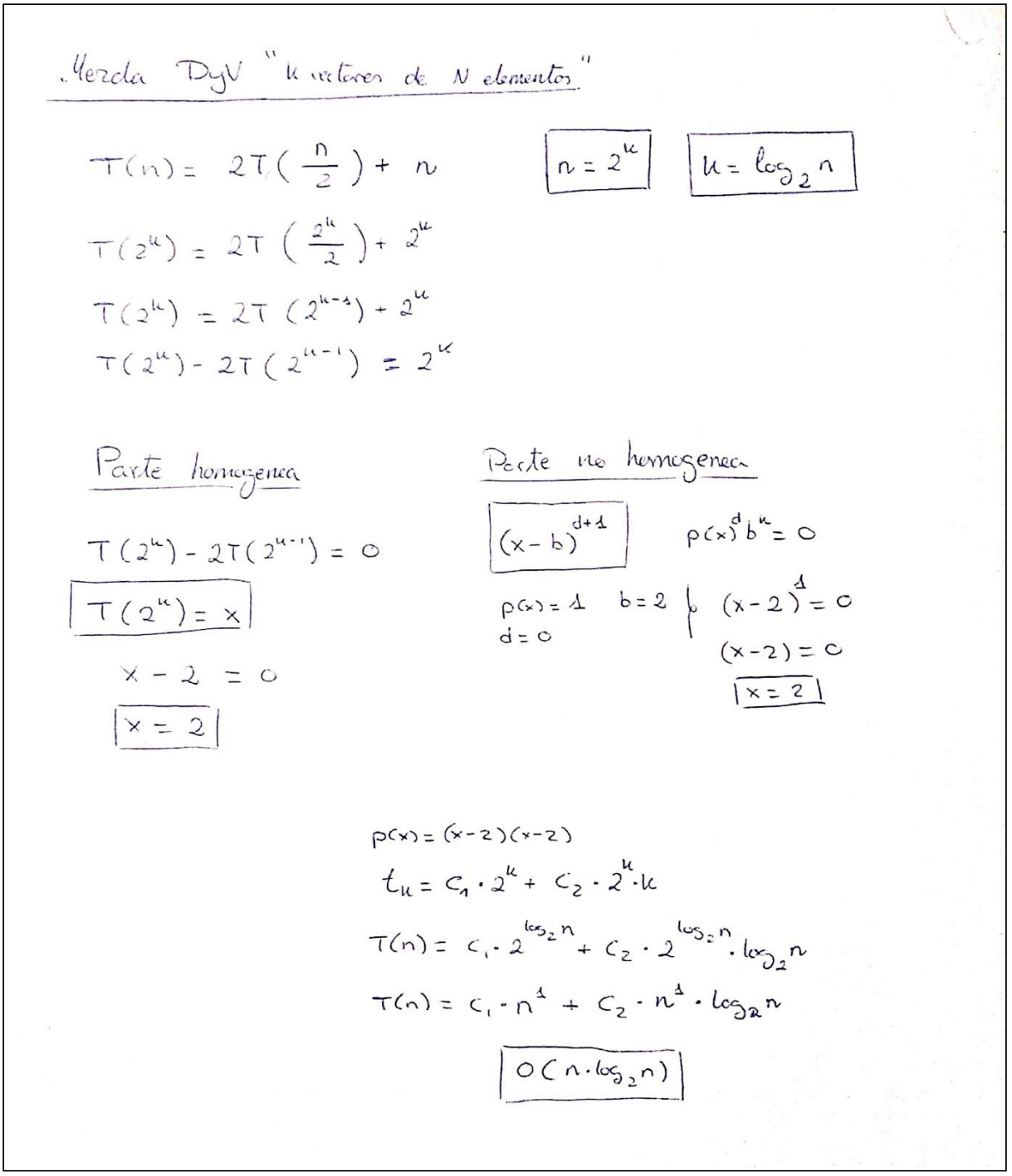
$O(n)^A$ y $O(n^2)^B$

```
for (int i = 1; i < k; i++) {                          -> O(n)^A
    int *solucion = new int [N*(i+1)];                 ->
    solucionAux = comparaVectores (T, solucion, N, i); -> O(n^2)^B
    for (int j = 0; j < N*(i+1); j++) {                -> O(n)
        solucion [j] = solucionAux[j];                 -> O(1)
    }
}
```

comparaVectores (...) { -> $O(n^2)^B$ -> $O(n)^C$ y $O(n)^D$

```
    ...
    for (int i=0; i < N*iter; i++) {                   -> O(n)^C
        if ( ) {                                       -> O(1)
            // Todo dentro O(1)
        }
        else {
            for ( j= pos_ultima; j < N && !termina; j++) { -> O(n)^D
                // Todo dentro O(1)
            }
        }
    }
    if ( pos_ultima < N) {                             -> O(1)
        for (int i = pos_ultima; i < N; i++) {         -> O(n)
            solucionA [contador++] = T[iter][i];       -> O(1)
        }
    }
    return solucionA;
```

En el caso de Divide y Vencerás, cada llamada al método recursivo se llama internamente 2 veces, dividiendo el tamaño del problema entre 2 (cada llamada divide el total de vectores entre 2) y en el caso base, N veces, por lo que quedaría la siguiente recurrencia:

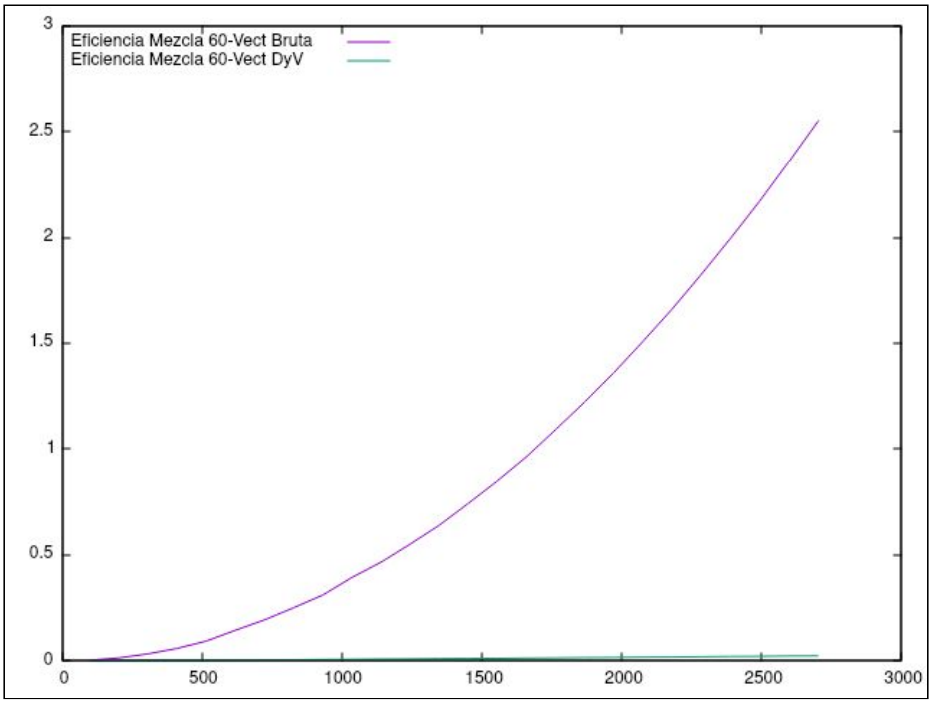


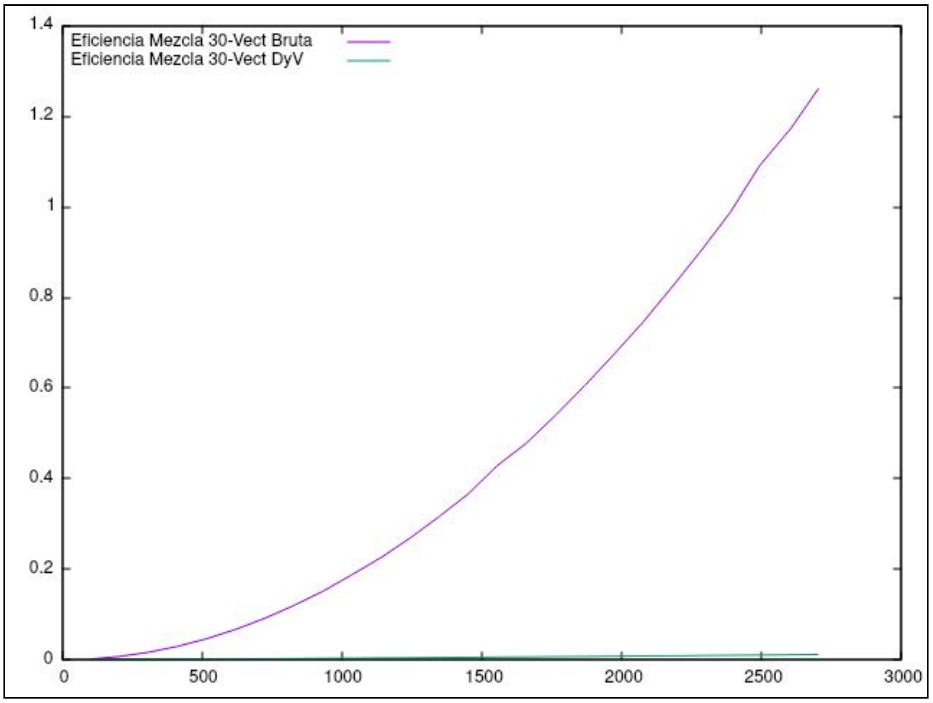
De esta forma quedaría una eficiencia de N*Log2(N).

4.3. Análisis empírico

Para realizar el análisis empírico se han utilizado un rango de tamaños que se puede observar en el punto 1 (página 3).

Como se puede ver en las gráficas empíricas (correspondiente a vectores de tamaño 60 y 30), los datos obtenidos mediante el método de fuerza bruta son mucho peores que con divide y vencerás (se ven los tiempos pegados al eje x), cosa que tendría sentido si la eficiencia teórica fuera correcta, porque O(n^3) es pero que O(n*log2(n)).





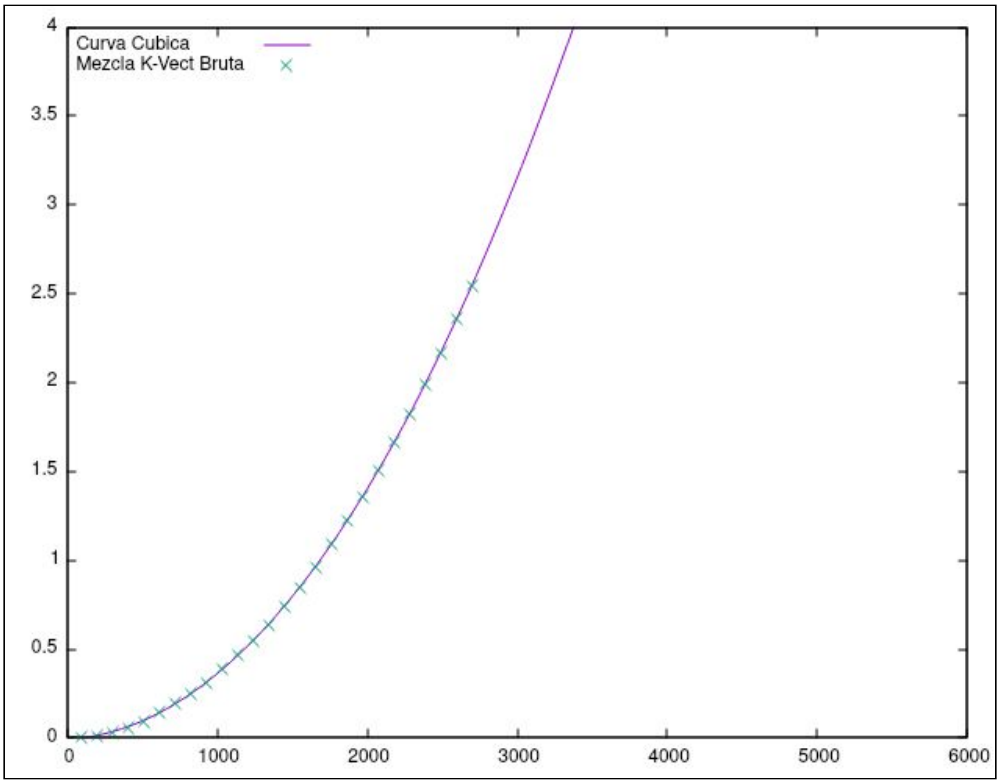
4.4. Análisis híbrido

Para el análisis híbrido se ha usado una función cúbica para la fuerza bruta y una función N*Logaritmo para el algoritmo divide y vencerás:

f(x) = a0*x*x*x+a1*x*x+a2*x+a3
g(x) = b0*x*(log10(x)/log10(2))

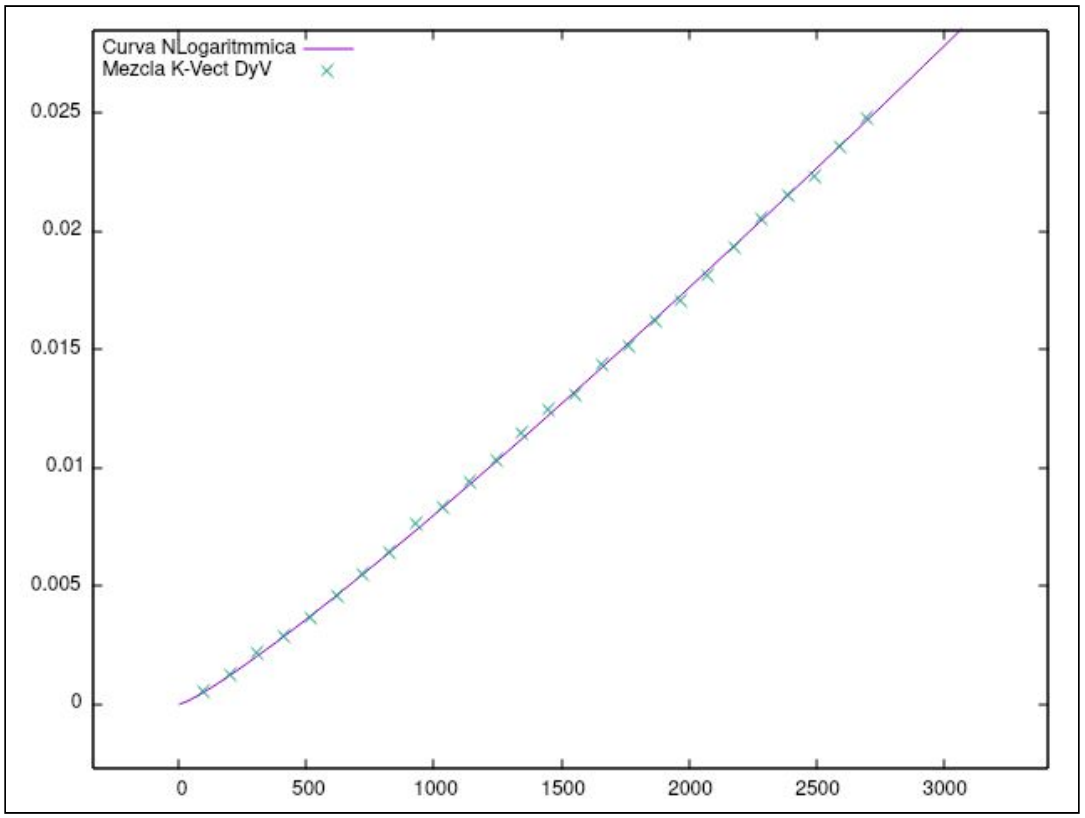
Como se puede ver, los coeficientes de correlación obtenidos son muy cercanos a 1 y los errores bajos (aunque en la cúbica el error es algo alto pero ajusta correctamente).

Las siguientes gráficas corresponden a las ejecuciones para vectores de tamaño 60:




```
Final set of parameters            Asymptotic Standard Error
=======================            ==========================
a0              = 6.60176e-12      +/- 1.708e-12    (25.87%)
a1              = 3.16192e-07      +/- 7.268e-09    (2.299%)
a2              = 4.52309e-05      +/- 8.839e-06    (19.54%)
a3              = -0.00784749      +/- 0.002908     (37.05%)

correlation matrix of the fit parameters:
                a0     a1     a2     a3
a0              1.000
a1             -0.987  1.000
a2              0.925 -0.972  1.000
a3             -0.713  0.790 -0.895  1.000
```




```
Final set of parameters                 Asymptotic Standard Error
=======================                 ==========================
a0                  = 8.02567e-07       +/- 1.591e-09    (0.1982%)
```

4.5. Umbral

Calculamos el umbral mediante la igualdad de las dos ecuaciones híbridas obtenidas en gnuplot al realizar los ajustes:

Cúbica	$f(x) = 6.60176 * 10^{-12}x^3 + 3.16192 * 10^{-7}x^2 + 4.52309 * 10^{-5}x - 0.00784749$
DyV	$g(x) = 8.02567 * 10^{-7}x * log2(x)$

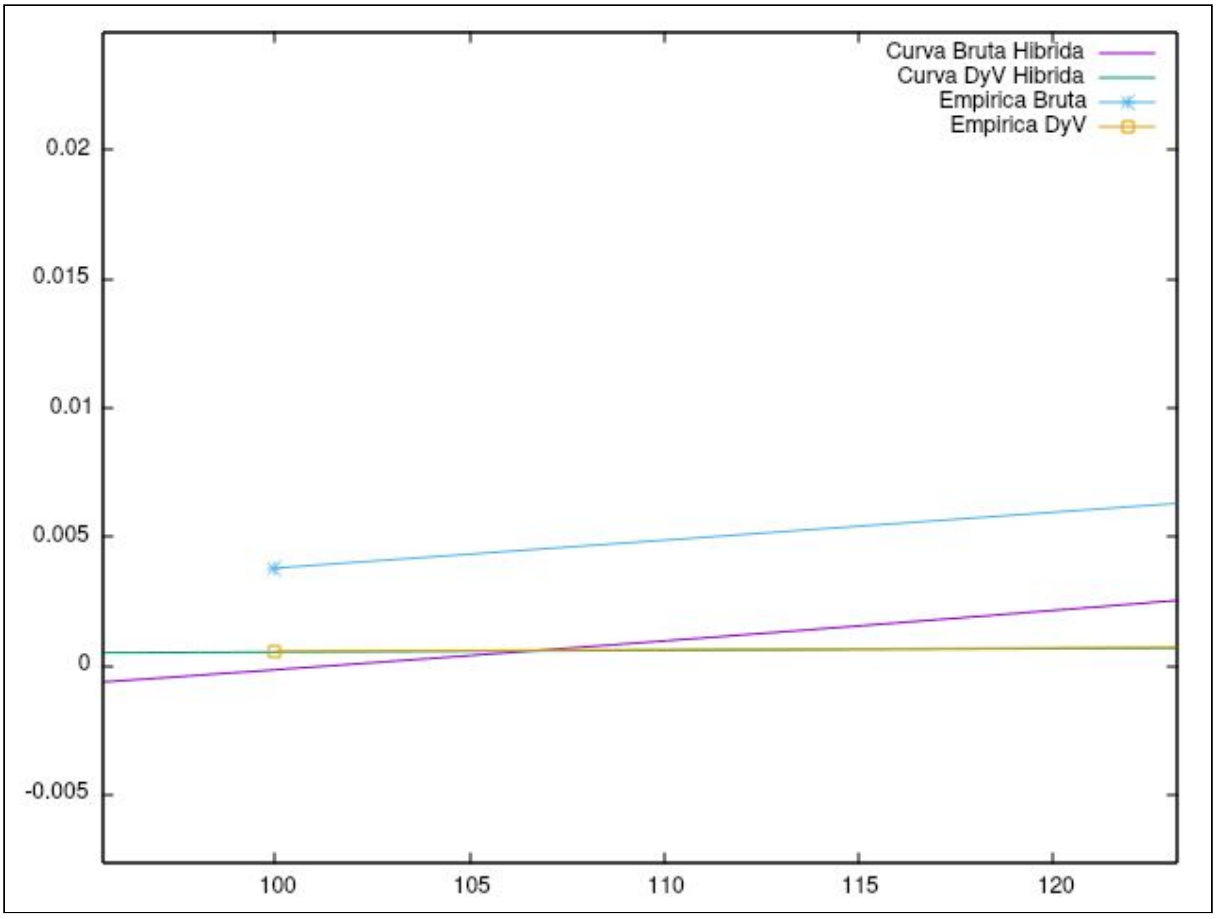
Al realizar la igualdad de f(x) = g(x) y resolver la ecuación resultante, obtenemos el siguiente valor:

x	106’61

En el siguiente enlace se puede ver la solución a la igualdad f(x) = g(x)

https://www.wolframalpha.com/input/?i=%286.60176*10%5E%28-12%29%29*x%5E3%2B%283.16192*10%5E%28-7%29%29*x%5E2%2B%284.52309*10%5E%28-5%29%29*x-%280.00784749%29%3D%288.02567*10%5E%28-7%29*x*log2%28x%29%29

Graficando en gnuplot y mirando en el valor resultante, se puede observar un cruce dichas eficiencias híbridas. A diferencia del ejercicio anterior, en este caso sí vemos que con el algoritmo de fuerza bruta adquiere un matiz ascendente mientras que el Divide y Vencerás continua lineal por el eje “x”.



5. Conclusión

Como conclusión, en esta práctica hemos visto que un algoritmo Divide y Vencerás no siempre es mejor que un algoritmo por fuerza bruta, ya que para problemas poco complejos como es realizar la traspuesta de una matriz, se realiza más eficientemente por fuerza bruta que realizando llamadas recursivas (tiene el gasto de realizar las llamadas) ya que cada caso base, realiza la misma función que realiza un recorrido matricial, esto lo hemos comprobado viendo que la eficiencia teórica es igual tanto para fuerza bruta que para divide y vencerás O(n^2).

Sin embargo para el caso del problema de mezclar K vectores ordenados hemos visto que si es más eficiente un algoritmo Divide y Vencerás que por fuerza bruta, ya que recursivamente, se logra tener un vector solución ordenado parcialmente conforme se va aumentando el número de vectores comparados, viendo que la eficiencia por fuerza bruta es menor que mediante un algoritmo divide y vencerás $O(n^3)$ mejor que O(n*log2(n)).

6. Compilación y Ejecución

Ficheros fuente:

- matrix_bruta.cpp → Algoritmo matriz traspuesta por fuerza bruta
- matrix_DyV.cpp → Algoritmo matriz traspuesta por divide y vencerás
- mezcla_bruta.cpp → Algoritmo mezcla k vectores por fuerza bruta
- mezcla_DyV.cpp → Algoritmo mezcla k vectores por divide y vencerás

Los ficheros fuente deben estar en una carpeta llamada "src" y los binarios en otra carpeta llamada "bin" y compilar desde el padre de dichas carpetas.

Para compilar:

- g++ -std=gnu++0x src/**fichero_fuente.cpp** -o bin/**fichero_binario**

Para ejecutar:

- ./bin/**fichero_binario** {parámetros}

7. Bibliografía

Lecciones de Algorítmica; Capítulo: 2; Tema: 4, 5, 6; Págs: 85-130; Verdegay Galdeano, José Luis

https://math.stackexchange.com/questions/3362825/comparing-the-time-complexity-of-two-algorithms-inequality

https://www.wolframalpha.com/

https://es.khanacademy.org/computing/computer-science/algorithms/merge-sort/a/divide-and-conquer-algorithms

8. Tabla de Notas

NOTA	Jose	Antonio	Pedro	Raul
Jose	10	10	10	10
Antonio	10	10	10	10
Pedro	10	10	10	10
Raul	10	10	10	10