Práctica 2 - Algoritmos Divide y Vencerás

Memoria de práctica

Germán Castilla López
Jorge Gangoso Klöck
Pedro Morales Leyva
Clara Mª Romero Lara

Índice de Contenidos

1. Descripción general de la práctica

Esta segunda práctica se centra en la técnica de diseño “Divide y Vencerás”. Los problemas a considerar son:

> Un problema común, la **traspuesta de una matriz.** Dada una matriz cuadrada de tamaño $n = 2k$, diseñar el algoritmo que devuelva la traspuesta de dicha matriz.
>
> Un problema elegido aleatoriamente, el 3.2: la **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).

Ambos problemas deben resolverse de dos maneras: por fuerza bruta, y por "divide y vencerás”. Se comparará la eficiencia teórica, empírica e híbrida de ambas versiones, y obtendremos el caso umbral para determinar en qué casos una versión es superior a la otra.

2. Análisis de la eficiencia

2.1. Fuerza bruta

2.1.1. Problema común: matriz traspuesta

La solución dada a este problema consiste en dos bucles for anidados que recorren la matriz intercambiando los valores que no pertenecen a la diagonal principal

$$\begin{pmatrix} n11 & n12 & n13 \\ n21 & n22 & n23 \\ n31 & n32 & n33 \end{pmatrix} \quad \begin{pmatrix} n11 & n21 & n31 \\ n12 & n22 & n32 \\ n13 & n23 & n33 \end{pmatrix}$$

Cada posición *i, j* se cambia por la posición *j, i*; con lo cual cuando termina el recorrido de los dos bucles queda la matriz traspuesta.

Usaremos el método `TrasponeFB(M, tam):` este método realiza el cambio de posición de los elementos de la matriz mediante dos bucles anidados. Recibe dos parámetros: la matriz a trasponer y su tamaño

Eficiencia teórica

Estudiando el código de la función `TransponeFB(M, tam)` vemos que la eficiencia de dicha función es: *x*log(x)* ya que el primer bucle se ejecuta *x* veces y el segundo se ejecuta *log(x)* veces.

No hay mejor ni peor caso ya que en cada iteración se cambia la posición de dos elementos de la matriz independientemente del tamaño de la matriz o el valor de los elementos.

Como el tamaño de la matriz se duplica en cada iteración, la eficiencia teórica de la función es:

T(n) = $(x*log(x))^n$

Eficiencia empírica

Se realizaron ejecuciones desde *k=2* hasta *k=30*. Dado que las matrices debían ser cuadradas, sólo se cumple la precondición cuando *k* es par.

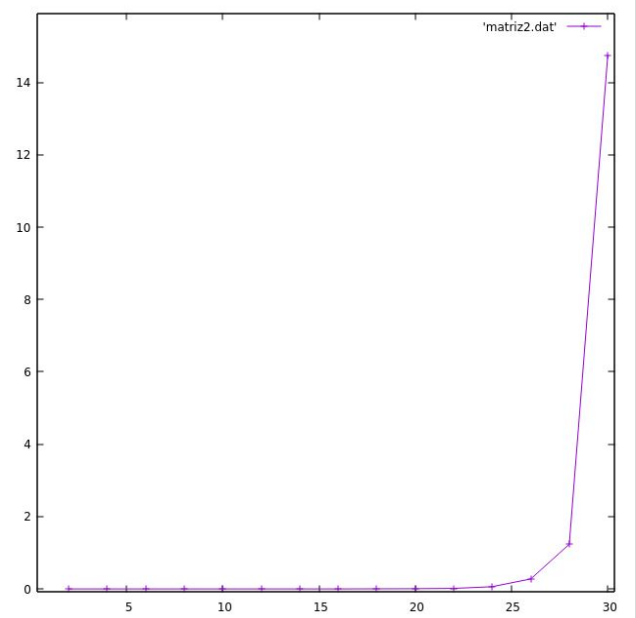
Este estudio se ha hecho en una máquina con las siguientes características:

- *Intel(R) Core(TM) i7-7700K CPU @ 4.20GHz*
- *RAM: 16GB*
- *SO: Ubuntu 18.04.2 LTS*

Los tiempos obtenidos fueron los siguientes:

K	Tiempo	K	Tiempo
2	0.000005	18	0.003212
4	0.000004	20	0.007431
6	0.000004	22	0.013671
8	0.000005	24	0.062384
10	0.000013	26	0.275385
12	0.000043	28	1.235537
14	0.000164	30	14.751908
16	0.000624		

Y la gráfica obtenida en gnuplot, siendo el eje *x* la *k;* y el eje *y* el tiempo:

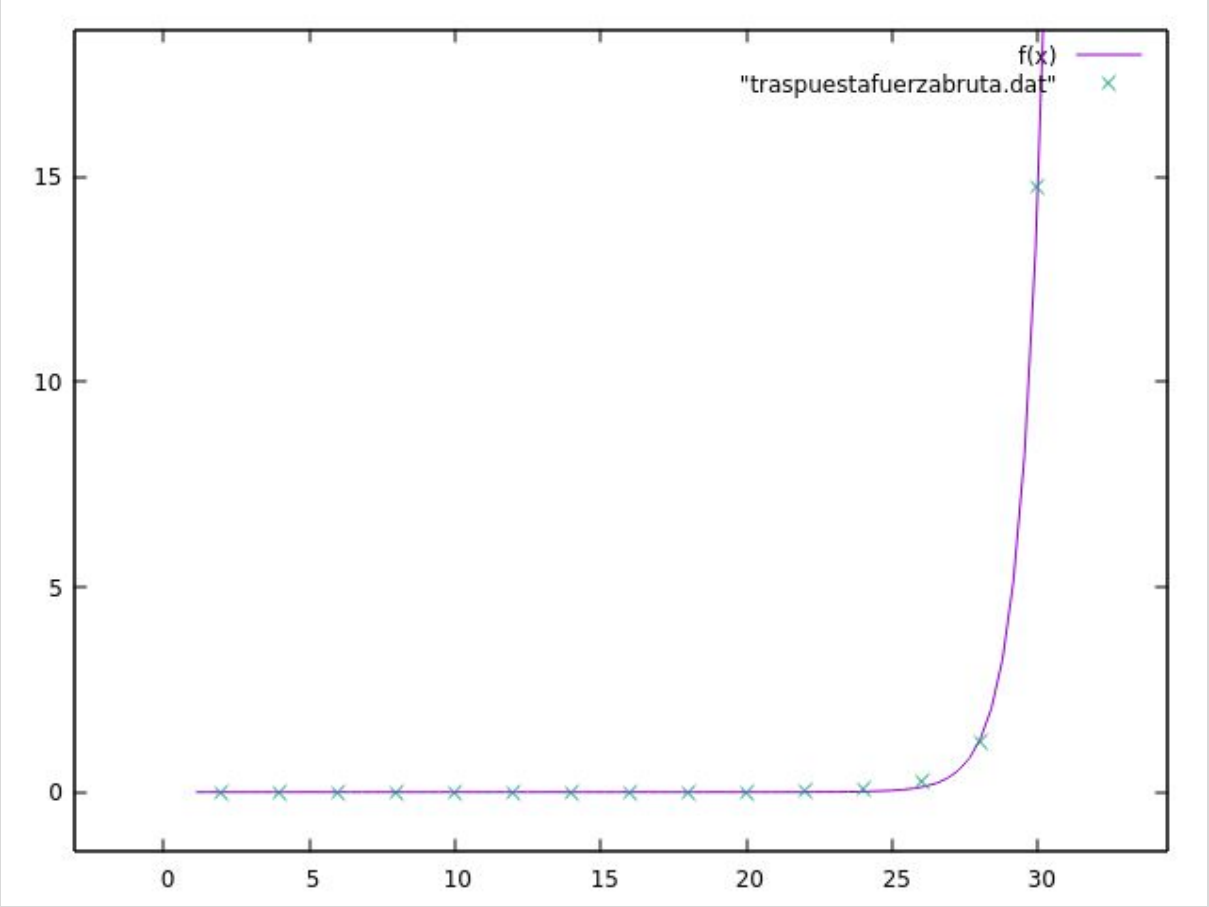


Eficiencia híbrida

Tras hacer el ajuste de $T(n) = (a*log(x))^{b*x}$, las constantes ocultas que obtenemos son $a = 0.300739$, con un margen de error del 0.03321% y b = 3.96663, con un margen de error del 1.561% . Por tanto, la ecuación resultante es:

$$T(n) = (0.300739*log(x))^{(3,96663*x)}$$

El resultado de la gráfica con el ajuste es:



2.1.2. Problema individual: mezcla de *k* vectores ordenados

La solución por fuerza bruta implementada consiste en un bucle for que es recorrido tantas veces como vectores haya que juntar.

Dentro del for se crean dos vectores auxiliares: uno con el vector final, resultado de los vectores que ya han sido juntados y ordenados (o, si es la primera vuelta, el primer vector dado); y otro con el vector a añadir al vector final. Se elimina el contenido del vector final. Finalmente, se llama a una función encargada de juntar los dos vectores, cuyo resultado se guarda en el vector final que acabamos de limpiar.

Dentro de la función el procedimiento es sencillo: se crea un vector resultado, comparamos los dos vectores auxiliares (pasados como parámetros) y vamos añadiendo los elementos ordenadamente, avanzando elemento por elemento y viendo cual es el menor. Con esto conseguimos un algoritmo poco eficiente pero funcional.

Eficiencia teórica

Si nos fijamos en el código podremos observar que no habrá un mejor y peor caso, ya que el número de acciones serán las mismas a igual tamaño de vector y número de vectores. El cómo estén distribuidos los elementos no importa, ya que los vectores están ordenados de menor a mayor y nosotros lo único que hacemos es comparar y volcar los datos de dos vectores en uno solo. Esto da que el tiempo que tarda el algoritmo en ejecutarse es n*k (número de vectores por el tamaño total de todos los vectores) lo que da un $O(n^2)$.

Eficiencia empírica

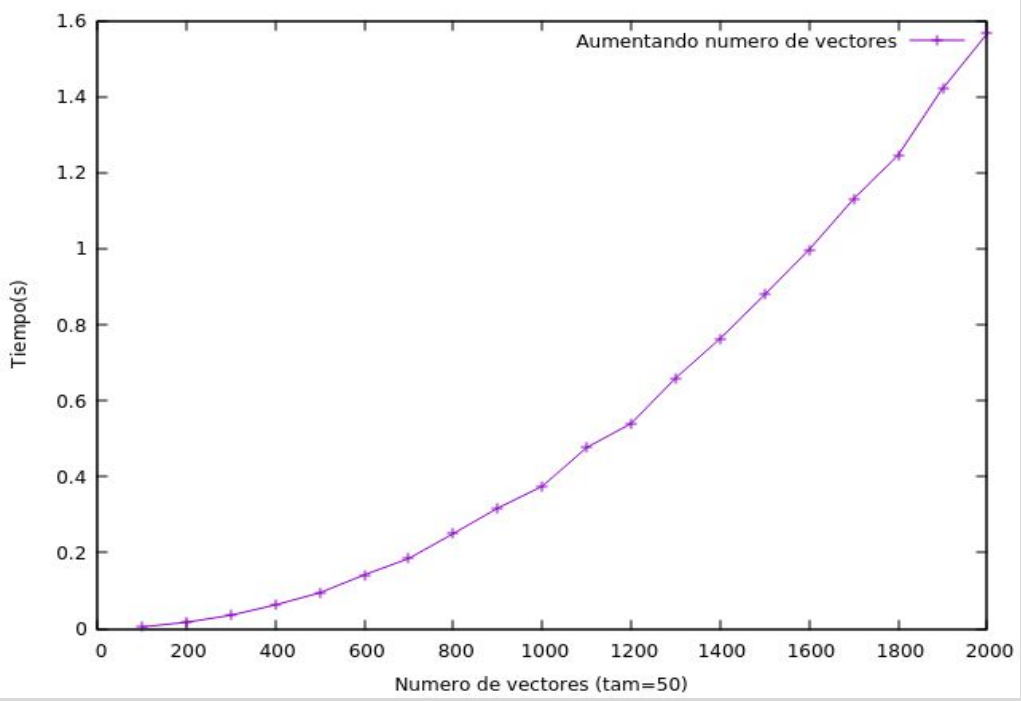
Para la eficiencia empírica del algoritmo hemos estudiado el aumento de vectores, ya que es la variable más significativa y la que realmente nos va a mostrar la eficiencia empírica del algoritmo. Para ello hemos asignado un valor constante al número de elementos de los vectores (50) y un aumento de 100 en 100 en el número de vectores a ordenar, partiendo de 100 y llegando a 2000.

Este estudio se ha hecho en una máquina con las siguientes características:

- *Intel® Core™ i7-7700HQ CPU @ 2.80GHz × 8*
- *RAM: 16GB*
- *SO: Ubuntu 18.04.2 LTS*

Aumento de número de vectores y tamaño constante de 50 elementos:

N vectores	Tiempo (s)	N vectores	Tiempo (s)
100	0.013789	1100	0.461993
200	0.016603	1200	0.560253
300	0.036989	1300	0.641803
400	0.064941	1400	0.731497
500	0.10086	1500	0.848798
600	0.146674	1600	1.00347
700	0.189528	1700	1.13655
800	0.247516	1800	1.31312
900	0.307981	1900	1.49912
1000	0.391103	2000	1.52599



Como podemos observar, es un algoritmo bastante poco eficiente, ya que con una cantidad de 2000 vectores ya tarda un segundo y medio. La curva nos puede ir anunciando que efectivamente se trata de un algoritmo de eficiencia cuadrática, pero esto lo veremos en e estudio de eficiencia híbrida.

Eficiencia híbrida

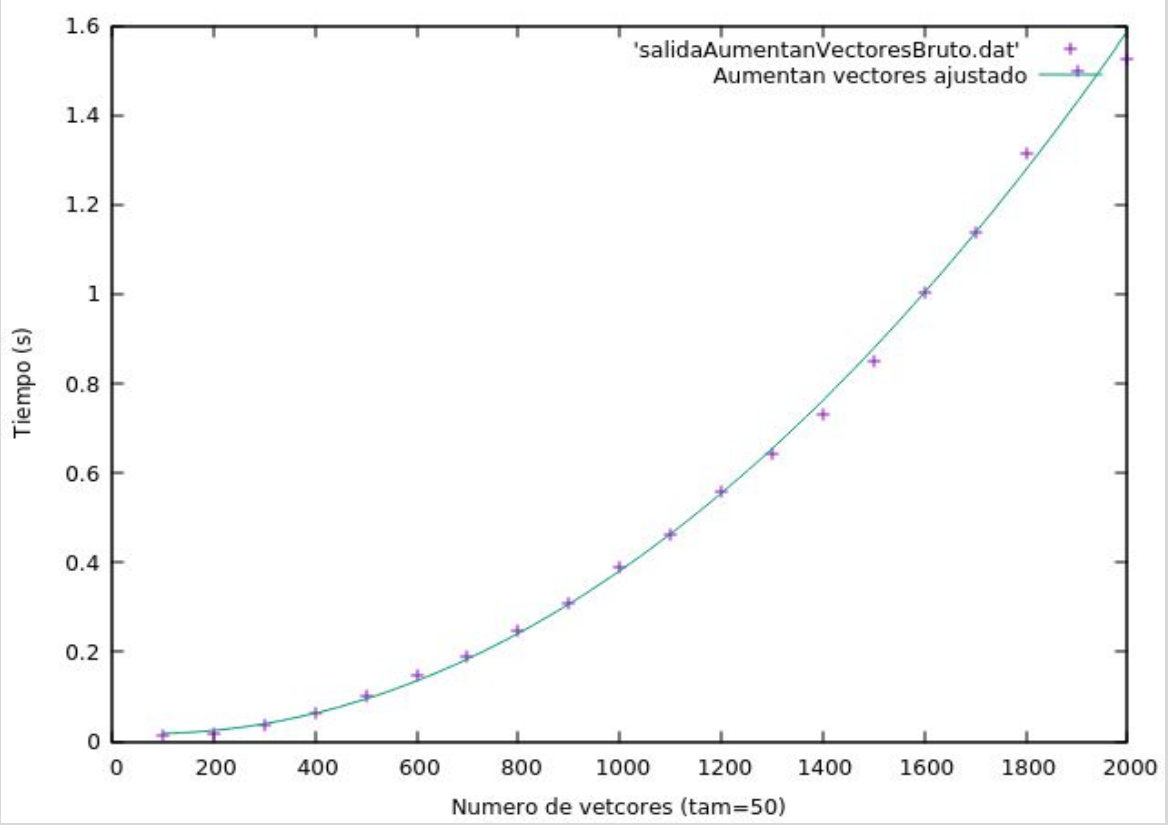
Para la eficiencia híbrida usaremos la función $f(x) = a*x^2+b*x+c$.
Con esto nos da unos valores:

a = 4.21918e-07
b = -6.01002e-05
c = 0.0195824

Con unos errores de:

a = 4.825%
b = 73.23%
c = 102.5%

Los errores del resultado son decentes, lo cual nos indica que efectivamente el algoritmo es de orden cuadrático. Esto nos lo vuelve a confirmar la gráfica, donde se ve más claramente:



2.2. Divide y vencerás

2.2.1. Problema común: matriz traspuesta

La solución dada a este problema se basa en llamadas recursivas a un método que divide la matriz en cuatro submatrices. Observemos cómo se traspone una matriz cuadrada:

$$\begin{pmatrix} n11 & n12 & n13 \\ n21 & n22 & n23 \\ n31 & n32 & n33 \end{pmatrix} \quad \begin{pmatrix} n11 & n21 & n31 \\ n12 & n22 & n32 \\ n13 & n23 & n33 \end{pmatrix}$$

Las posiciones cuya fila y columna coinciden, $i=j$, se mantienen en su lugar, mientras que las posiciones que cumplen que $i_1=j_2$, $j_1=i_2$ intercambian puestos.

Ahora, si tenemos en cuenta que vamos a trabajar con matrices cuadradas tal que $n=2k$, sabemos que las matrices que obtengamos tendrán un tamaño de fila/columna divisible entre 2. Así que, si tomamos un ejemplo nuevo que se ajuste a nuestras precondiciones...

$$\begin{pmatrix} n11 & n12 & n13 & n14 \\ n21 & n22 & n23 & n24 \\ n31 & n32 & n33 & n34 \\ n41 & n42 & n43 & n44 \end{pmatrix} \quad \begin{pmatrix} n11 & n21 & n31 & n41 \\ n12 & n22 & n32 & n42 \\ n13 & n23 & n33 & n43 \\ n14 & n24 & n34 & n44 \end{pmatrix}$$

Vemos que, además de cumplirse lo dicho sobre la trasposición de matrices cuadradas, podemos hacer una división en 4 submatrices 2*2:

$$\begin{pmatrix} n11 & n12 & n13 & n14 \\ n21 & n22 & n23 & n24 \\ n31 & n32 & n33 & n34 \\ n41 & n42 & n43 & n44 \end{pmatrix}$$

$$\downarrow$$

$$\begin{pmatrix} n11 & n12 \\ n21 & n22 \end{pmatrix} \begin{pmatrix} n13 & n14 \\ n23 & n24 \end{pmatrix}$$

$$\begin{pmatrix} n31 & n32 \\ n41 & n42 \end{pmatrix} \begin{pmatrix} n33 & n34 \\ n43 & n44 \end{pmatrix}$$

Si trasponemos las submatrices obtenidas de la matriz original...

$$\begin{pmatrix} n11 & n21 \\ n12 & n22 \end{pmatrix}\begin{pmatrix} n13 & n23 \\ n14 & n24 \end{pmatrix}$$
$$\begin{pmatrix} n31 & n41 \\ n32 & n42 \end{pmatrix}\begin{pmatrix} n33 & n43 \\ n34 & n44 \end{pmatrix}$$

Y luego trasponemos las posiciones de estas submatrices dentro de lo que sería la matriz original, **obtenemos la matriz traspuesta**

$$\begin{pmatrix} n11 & n21 \\ n12 & n22 \end{pmatrix}\begin{pmatrix} n31 & n41 \\ n32 & n42 \end{pmatrix}$$
$$\begin{pmatrix} n13 & n23 \\ n14 & n24 \end{pmatrix}\begin{pmatrix} n33 & n43 \\ n34 & n44 \end{pmatrix}$$
$$\downarrow$$
$$\begin{pmatrix} n11 & n21 & n31 & n41 \\ n12 & n22 & n32 & n42 \\ n13 & n23 & n33 & n43 \\ n14 & n24 & n34 & n44 \end{pmatrix}$$

Siendo entonces fácil de implementar de manera recursiva: tomamos la matriz, la dividimos hasta que las submatrices sean de dimensión 2*2 y trasponemos los resultados recursivamente.

Usaremos tres métodos:

- `Traspone(M, tam)` - Este método es el que recopila la llamada al algoritmo divide y vencerás, nos redirige a DivideMatriz. Recibe la matriz completa.

- `DivideMatriz(M, f_ini, f_fin, c_ini, c_fin)` - Este método recibe una matriz que cumple las precondiciones dadas y unas medidas: el inicio y el fin de la fila y la columna que queremos tomar: por ejemplo, si queremos introducir una matriz cuadrada tamaño *n,* completa, escribiremos:

  `DivideMatriz(M, 0, n-1,0, n-1)`

  Si la dimensión de la matriz es mayor de 2*2, se ejecuta recursivamente dividiéndola por la mitad en 4 submatrices cuadradas.

- `CambiaPosicion(M, f_iniA, c_iniA, f_iniB, c_iniB, tam)` - Este método intercambia las posiciones (A y B) de una matriz que no pertenecen a la diagonal principal.

Eficiencia teórica

Para obtener la eficiencia teórica, nos centraremos principalmente en el método `DivideMatriz`, ya que es en el que se da la recursividad.

El método DivideMatriz es llamado un total de k veces.
El caso final (es decir, cuando se alcanza una matriz 2*2) no lo contamos puesto que su orden de eficiencia es constante (asignación de variables) y, por tanto, trivial.

El tamaño de la matriz se divide entre 4 cada vez que se llama a la función.
Por lo tanto, determinamos que la eficiencia de este programa es del orden de $T(n)=2^n$

Eficiencia empírica

Se realizaron ejecuciones desde $k=2$ hasta $k=30$. Dado que las matrices debían ser cuadradas, sólo se cumple la precondición cuando k es par.

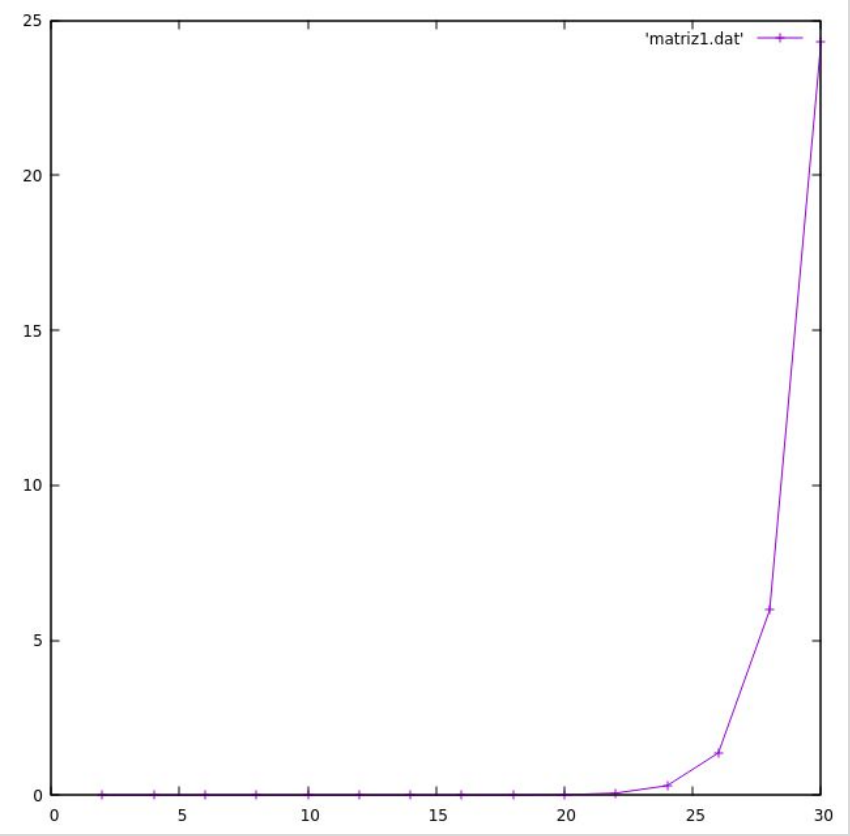
Este estudio se ha hecho en una máquina con las siguientes características:

- *Intel® Core™ i5-8250U CPU @1.60GHz × 8*
- *RAM: 12GB*
- *SO: Arch Linux 5.5.13-arch2-1*

Los tiempos obtenidos fueron los siguientes:

K	Tiempo	K	Tiempo
2	0.000003	18	0.004131
4	0.000004	20	0.016743
6	0.000004	22	0.072233
8	0.000011	24	0.310197
10	0.000040	26	1.365992
12	0.000182	28	5.978411
14	0.000551	30	24.288269
16	0.000881		

Y la gráfica obtenida en gnuplot, siendo el eje *x* la *k;* y el eje *y* el tiempo:

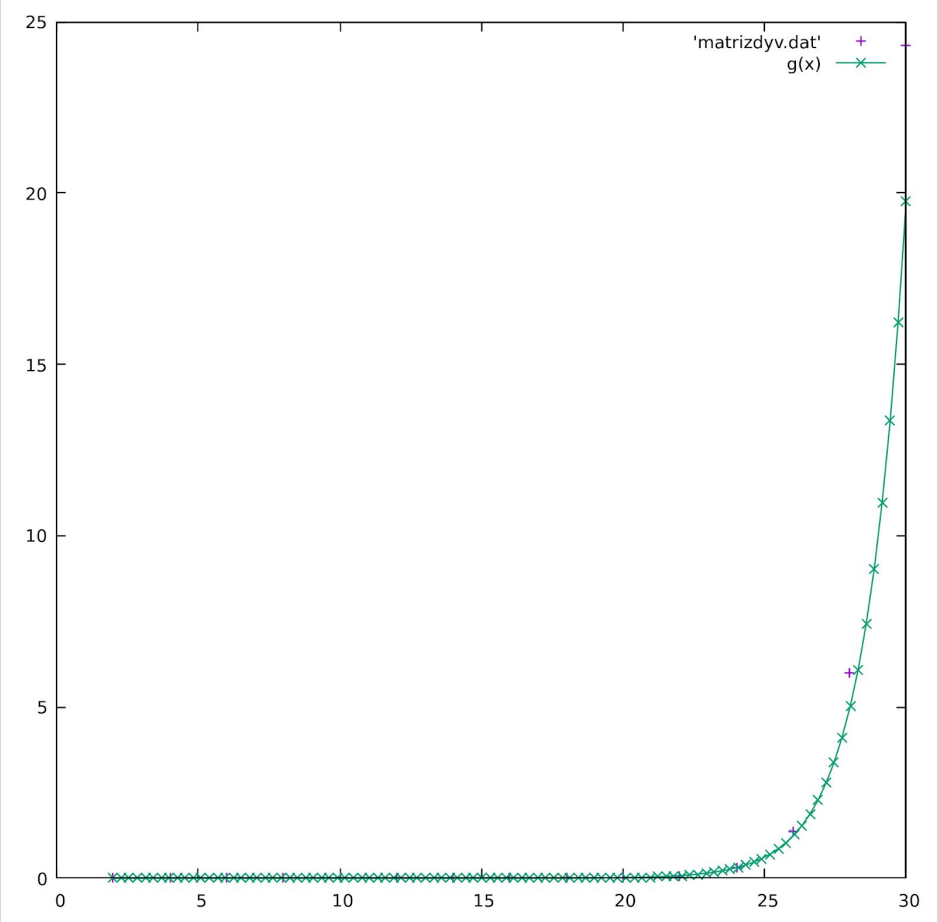


Eficiencia híbrida

Tras hacer el ajuste de $T(n) = a*2^n$, la constante oculta que obtenemos es $a = 1.8405e-08$, con un margen de error del 0.5498%. Por tanto, la ecuación resultante es:

$$T(n) = 1.8405e-08*2^n$$

La gráfica con ajuste resultante es la siguiente:



2.2.2. Problema individual: mezcla de *k* vectores ordenados

La solución a este problema se ha planteado desde el punto de vista de ejecutar la segunda fase de mergesort usando una estrategia de Divide y Vencerás, por esto, hemos llamado a la función que soluciona el problema `merge`.

Además de `merge,` se ha empleado una función auxiliar llamada `recombine`, que mezcla los subvectores que ya han sido mezclados.

Desarrollamos las funciones utilizadas:

- `Merge(int v_inicial, int v_final, int* parcial, int** T, int n)`:

1. **Parámetros:**
   La función recibe en este orden:
   - El índice del primer vector a partir del cual mezclar.
   - El índice del último vector que se debe mezclar. Con esto, delimitamos la cantidad de vectores que va a mezclar la función merge.
   - Un vector (parcial), el cual se utilizará para ir enviando y recibiendo las soluciones parciales y será donde finalmente se reciba la solución. Se envía como puntero para poder modificarlo en cada llamada recursiva.
   - La estructura en la cual están almacenados los *k* vectores de tamaño *n* que deberemos mezclar.
   - El tamaño *n* de cada vector necesario para un resultado correcto y evitar fallos de acceso a memoria restringida.
2. **Funcionamiento:**
   Al ser merge la base de la estructura Divide y Vencerás encontramos dos grandes bloques:
   El caso base: en el que `v_inicial == v_final,` es decir, sólo tenemos un vector que mezclar, no hay que realizar ninguna llamada recursiva. Simplemente colocamos en la posición adecuada el subvector ordenado, que posteriormente será reorganizado en la función `recombine`.

   El segundo bloque realiza la recursividad. Separa la tarea de mezclar los vectores comprendidos entre `v_inicial` y `v_final` en dos subtareas (mediante dos llamadas a merge), cada una con la mitad de vectores para mezclar, y, una vez realizada cada una de las subtareas, recombinar la solución parcial obtenida para generar una solución mayor (mediante una llamada a recombine).

- `Recombine(int v_inicial, int v_final, int* vector, int n)`:

1. **Parámetros:**
   La función recibe en este orden:
   - El índice del primer vector a partir del cual mezclar.
   - El índice del último vector que se debe mezclar. La función recombine siempre considerará que tiene dos vectores que mezclar, el primero desde v_inicial a (v_final+v_inicial)/2 y el segundo desde el siguiente elemento a ese hasta el último que estaría representado por: inicio_vfin+tamaño_vfin.
   - El vector en el que previamente se han guardado las soluciones parciales de las llamadas a merge
   - El tamaño n de cada vector necesario para un resultado correcto y evitar fallos de acceso a memoria restringida.
2. **Funcionamiento:**
   Consiste en inicializar las variables necesarias para actualizar el vector solución sin perder datos en el proceso (principalmente, copia del tramo del vector con el que se va a trabajar). Finalmente, itera mediante una técnica de doble puntero o doble índice para colocar ordenadamente en la solución el elemento de un subvector u otro según se deba para mantener el orden.

Por comentar brevemente algunos obstáculos encontrados durante la práctica, al realizar la medición de tiempo empírica con `clock()`, el tiempo de ejecución del programa alcanzaba los 5 segundos, pero devolvía un resultado de 0.001 segundos de ejecución.

Se tomaron dos medidas:

Primero, cambiar el reloj al de la librería <*chrono*>. Parecía que necesitábamos algo más de precisión ya que los tiempos eran realmente bajos

Y segundo, colocar un clock también en las funciones que generan y ordenan los subvectores para comenzar el problema.

Gracias a estas dos medidas pudimos comprobar que el 95% del tiempo empleado de ejecución era **consumido por estas tareas de inicialización** y que los tiempos dados por `clock()`, aunque imprecisos, no eran erróneos.

Eficiencia teórica

Si nos centramos en la eficiencia teórica de la función merge hallamos una ecuación de recurrencia de la forma:

$$T(k) = 2T(k/2) + k*n \quad si\ k>1$$

$$T(k) = n$$

Donde *k* es la cantidad de vectores a mezclar, *n* el tamaño de cada vector y *T(k)* el tiempo de la función merge para un número de vectores *k*.

Esta ecuación no representa más que el hecho de que el caso base consiste en un bucle que itera *n* veces realizando una operación sencilla, y que el caso en que *k>1*, se realizan dos llamadas a `merge`, pero esta vez con tamaño *k/2*, es decir *2*T(k/2)*, y una llamada a `recombine` que realiza la comparación del doble puntero *n*k* veces.

Se considera que las demás operaciones consumen un tiempo constante y, por tanto, no son relevantes para calcular el Orden de Eficiencia teórico.

Resolver la Ecuación de Recurrencia:

La ecuación mostrada anteriormente es del tipo: recurrente, no homogénea y lineal (aunque presenta una falta de linealidad en el argumento *[k/2]*).

Por esto, debemos eliminar esa falta de linealidad del argumento antes de poder resolver la ecuación. Para ello, realizamos un cambio de variable $k = 2^m$. La ecuación resulta así:

$$T(2^m) = 2T(2^{m-1}) + n(2^m)$$

Podemos observar como la *n* va perdiendo relevancia ya que se trata de una constante y no afectará más que las constantes ocultas del resultado de la eficiencia.

Ordenamos pues la función para ver con claridad la ecuación característica que la forma y vemos:

$$Tm - 2T(m-1) = n(2^m)$$

Ecuación característica: *(x-2)(x-b) -> b=2 -> (x-2)(x-2)*

Si lo expresamos como sumatorio de constantes multiplicando las raíces obtenemos:

$$Tm = C1 * 2^m + C2 * m * 2^m$$

(El segundo sumando tiene una m adicional ya que es (x-2) raíz doble)

Deshaciendo el cambio de variable $2^m = k$

$$Tm = C1 * k + C2 * log(k) * k$$

Y puesto que O(k) está contenido en O(k*log(k)), podemos afirmar que nuestro algoritmo es de orden **O(k*log(k))**.

Eficiencia empírica

Para el cálculo de la eficiencia empírica hemos hecho uso de la librería <*chrono*> y la librería <*ctime*>.

Hemos tenido en cuenta dos enfoques: variar el número de vectores; y variar el tamaño de cada vector. La cantidad total de elementos a mezclar es la misma teniendo 5 vectores de 10 elementos, que 10 vectores de 5 elementos.

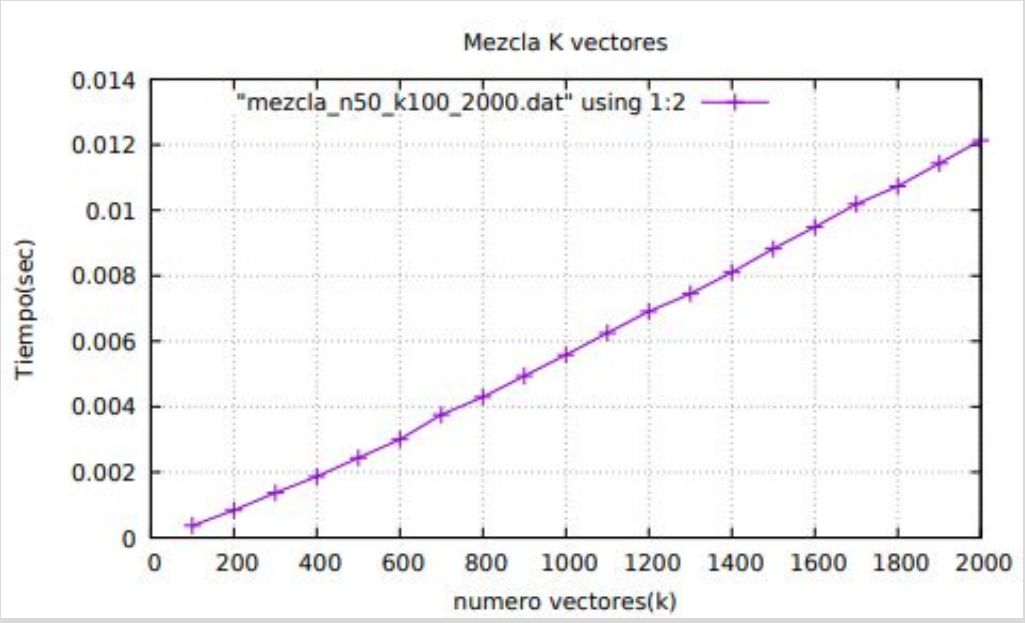
Este estudio se ha hecho en una máquina con las siguientes características:

- *AMD Ryzen 5 1600X Six-Core Processor @3.60GHz*
- *RAM: 16GB*
- *SO: Windows 10, 64 bits*

En primer lugar hemos mantenido constante el número de elementos por vector en 50 elementos y hemos ido variando mediante un script de bash el número de vectores a mezclar.

Número de vectores	Tiempo	Número de vectores	Tiempo
100	0.0003703	1100	0.0062481
200	0.0008362	1200	0.0069094
300	0.0013677	1300	0.0074482
400	0.0018679	1400	0.0081037
500	0.0024371	1500	0.0088217
600	0.0030007	1600	0.0094864
700	0.003753	1700	0.0101869
800	0.004295	1800	0.0107328
900	0.0049378	1900	0.01143
1000	0.0055845	2000	0.0121343

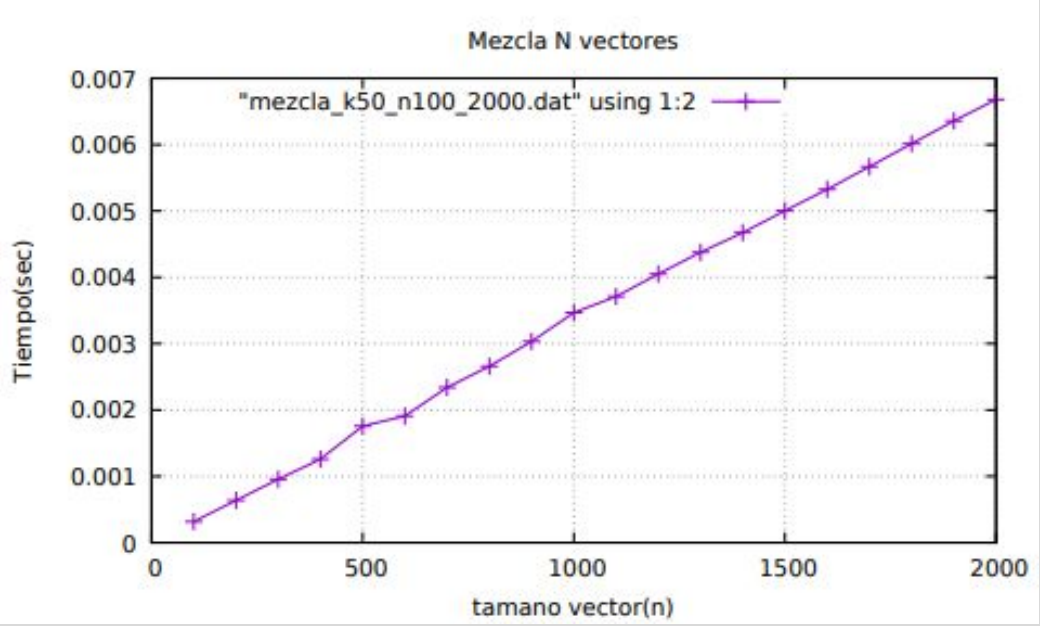
La gráfica asociada a dichos datos es:



A continuación mantenemos el número de vectores constante a 50 y variamos la cantidad de elementos por vector

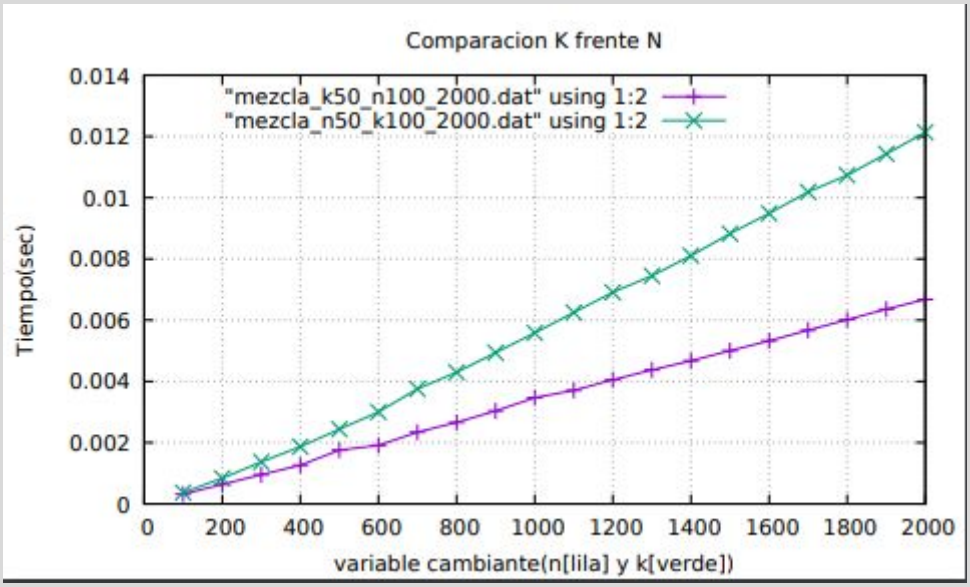
Número de elementos	Tiempo	Número de elementos	Tiempo
100	0.0003169	1100	0.0037086
200	0.0006369	1200	0.00405
300	0.0009541	1300	0.0043761
400	0.001259	1400	0.0046722
500	0.001757	1500	0.0050015
600	0.0019083	1600	0.0053257
700	0.0023406	1700	0.0056678
800	0.0026586	1800	0.0060152
900	0.0030336	1900	0.0063576
1000	0.0034676	2000	0.0066791

La gráfica asociada a dichos datos es:



Al haber usado los mismos valores tanto en el caso 1 como en el caso 2, ahora podemos representar ambas gráficas sobre una sola y comprobar empíricamente qué variable tiene más peso sobre la eficiencia. Como se comentó anteriormente, la cantidad total de elementos, es, en cada valor del eje x, el mismo para ambas gráficas.

El resultado es el siguiente:



Como podemos observar, al aumentar los valores de k (gráfica verde) el tiempo aumenta considerablemente más que al aumentar n (gráfica lila). Por esto, podemos afirmar que, empíricamente, la variable k tiene mayor peso en la eficiencia del algoritmo, lo cual se ve respaldado por los cálculos teóricos realizados anteriormente.

Eficiencia híbrida

Si tomamos ahora los valores obtenidos empíricamente, podemos tratar de ajustar esos valores a una gráfica que se corresponda con el orden de eficiencia calculado teóricamente.

Sin embargo, para comprobar mejor el ajuste, se han tomado nuevas medidas de datos exclusivamente para dicha tarea de un tamaño mayor.

El tiempo total de ejecución para calcular dichos datos fue alrededor de una hora por el tiempo necesario para inicializar los datos para el ejercicio como se comentó anteriormente también.

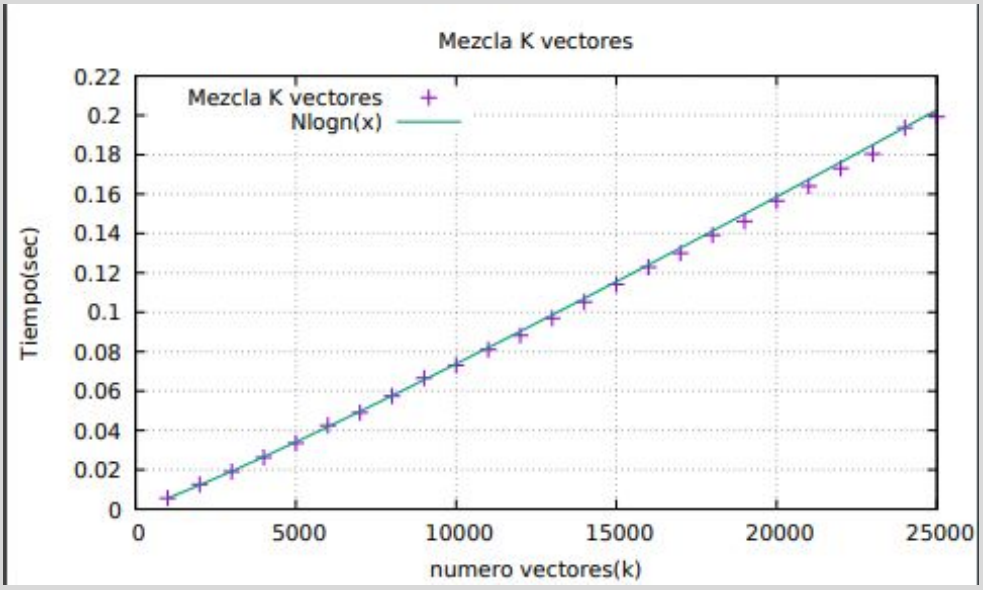
Podemos ver como la gráfica se ajusta bastante bien a los puntos dados por el fichero de datos y los valores de error de las variables a y b son los siguientes:

$$a = 8.00792e\text{-}07$$
$$b = 4.90826e\text{-}06$$

Con un porcentaje de error de 0.33% en la variable de mayor peso, se trata de un ajuste excelente que verifica por tanto la validez del estudio teórico y empírico.

Número de elementos	Tiempo	Número de elementos	Tiempo
1000	0.0055515	14000	0.105196
2000	0.0125723	15000	0.11414
3000	0.019034	16000	0.122869
4000	0.0262588	17000	0.130022
5000	0.0335156	18000	0.139148
6000	0.0424774	19000	0.146157
7000	0.0489811	20000	0.156544
8000	0.0574302	21000	0.16402
9000	0.0665629	22000	0.173024
10000	0.0730637	23000	0.180396
11000	0.080987	24000	0.193653
12000	0.0882166	25000	0.199263
13000	0.0969151		

Así pues, con esta nueva tabla de datos realizamos el ajuste a una gráfica de orden *O(n*log(n)) = a*x*log(x) + b*



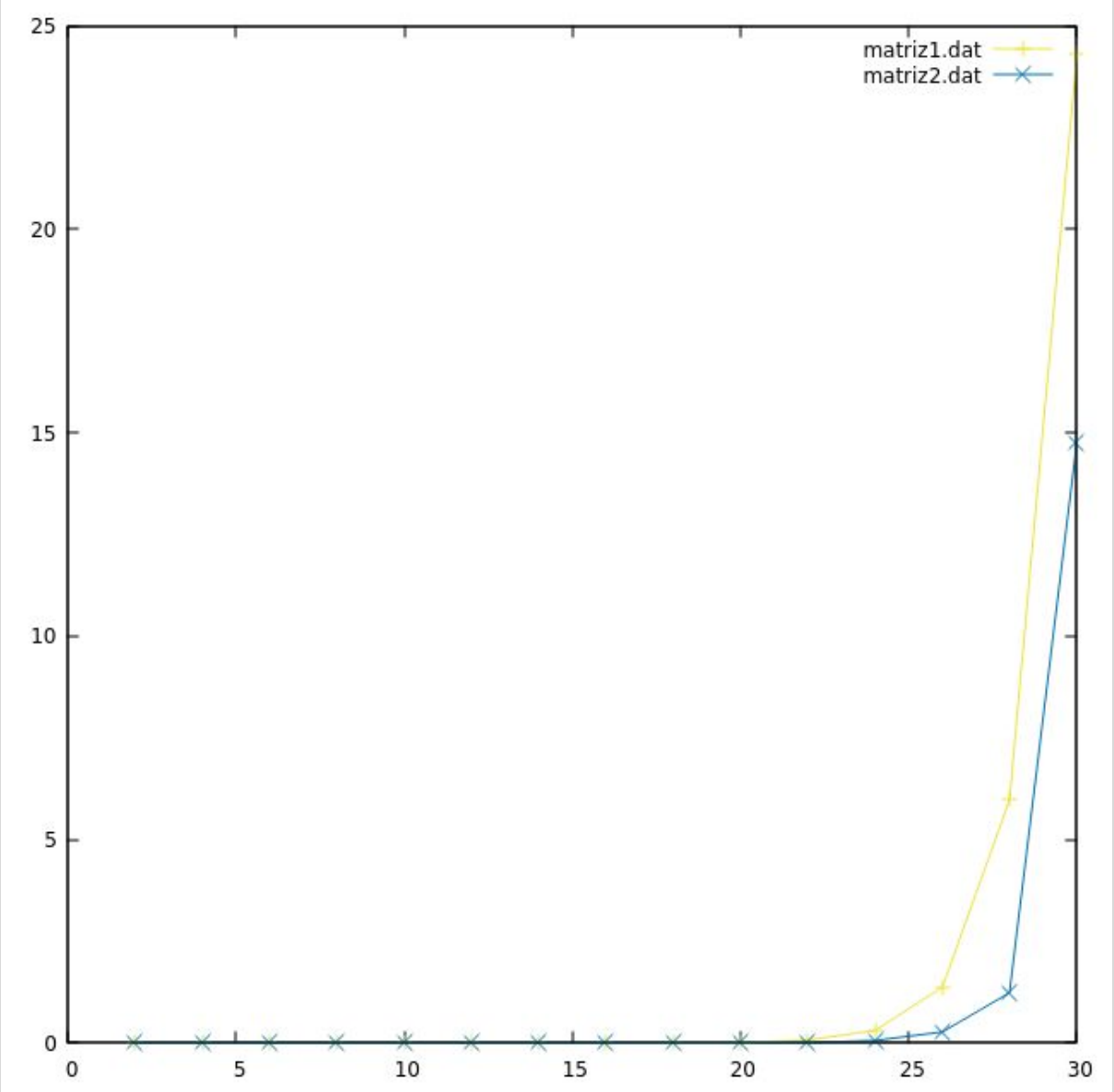
2.3. Estudio del umbral

2.3.1. Problema común: matriz traspuesta

Si comprobamos las gráficas obtenidas de ambas soluciones, podemos afirmar lo desarrollado en los apartados de eficiencia teórica: la versión por fuerza bruta es más eficiente que la versión con divide y vencerás.

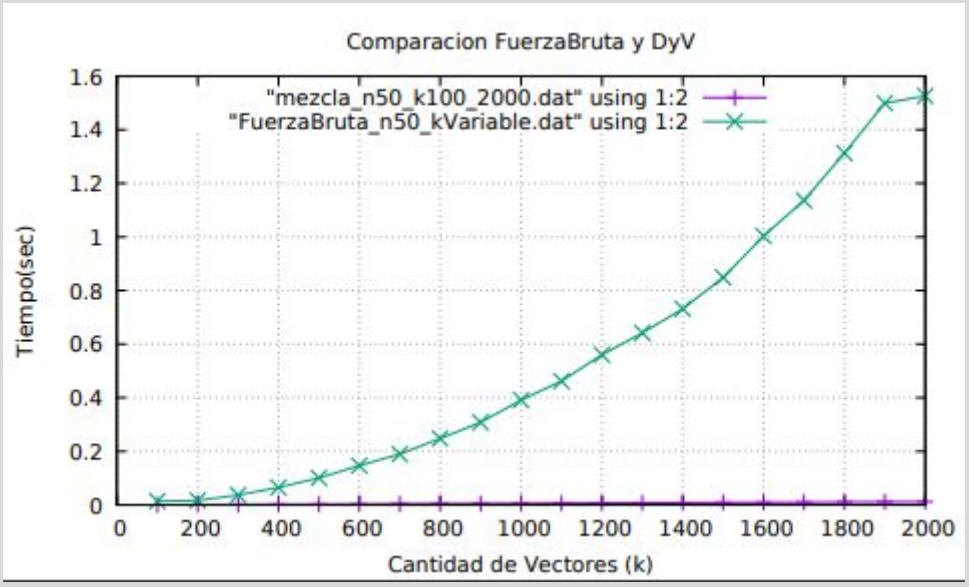
En la gráfica siguiente vemos que la versión DyV (matriz1.dat, amarilla) a partir del caso *k=24* crece mucho más rápidamente que la versión por fuerza bruta (matriz2.dat, azul).

Dicho esto, no consideramos que *k=24* sea un caso umbral, ya que las diferencias en los tiempos de las primeras iteraciones son triviales. Consideramos pues, que el caso de fuerza bruta es siempre mejor opción directamente.

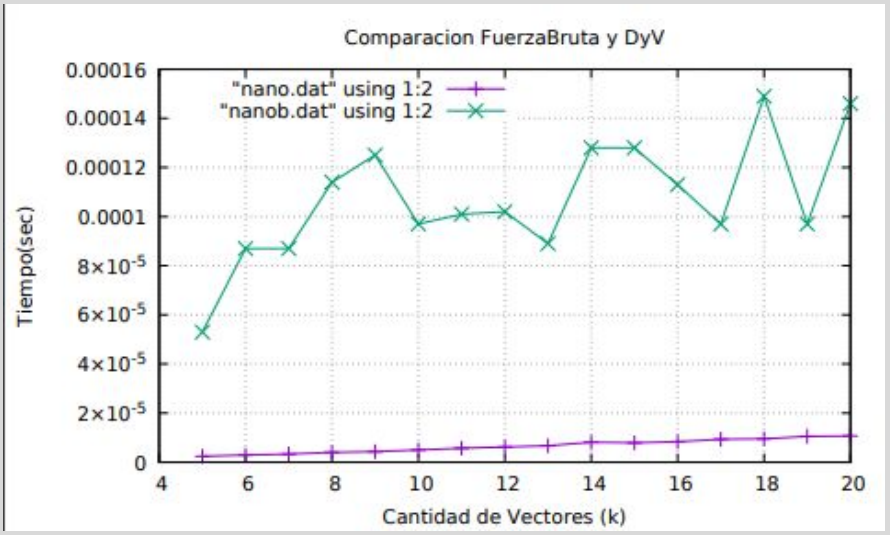


2.3.2. Problema individual: mezcla de k vectores ordenados

Para hallar el valor umbral, comparamos los tiempos de ejecución de las versiones por fuerza bruta y Divide y Vencerás para encontrar el valor a partir del cual uno es más eficiente que el otro.
Tras observar las siguientes gráficas llegamos a la conclusión de que nuestra versión Divide y Vencerás siempre es más eficiente que la versión por fuerza bruta y, por tanto, deberíamos emplearla para cualquier cantidad de datos.



Y si intentamos observar la parte de menor tamaño vemos que aún sigue habiendo diferencia.



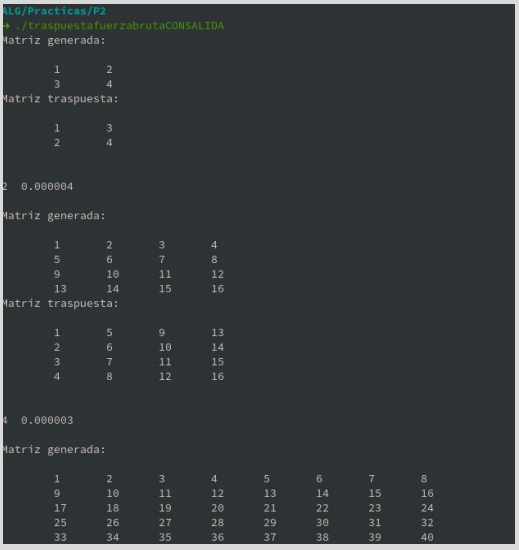
3. Caso de ejecución

3.1. Matriz traspuesta

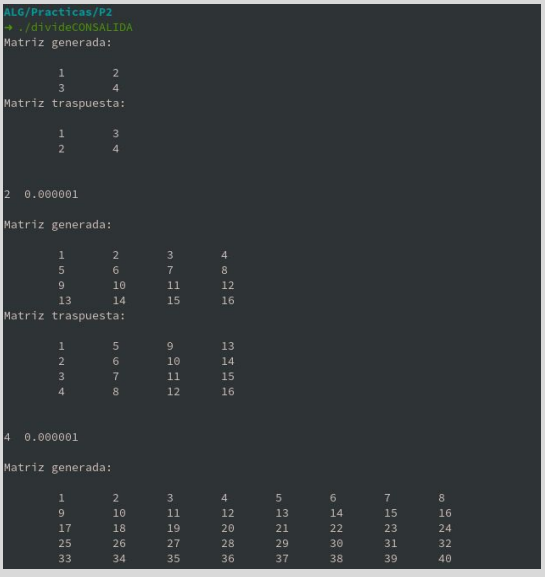
1. Compilamos el programa de fuerza bruta
   `g++ traspuestafuerzabruta.cpp -o traspuestafuerzabruta`

2. Compilamos el programa de divide y vencerás con
   `g++ DYV_Matriz.cpp -o divide`

3. Ejecutamos el programa de fuerza bruta y recopilamos el .dat
   `./traspuestafuerzabruta >> traspuestafuerzabruta.dat`



4. Ejecutamos el programa de divide y vencerás y recopilamos el .dat
   `./divide >> matrizdyv.dat`



3.2. Mezcla de k vectores ordenados

1. Compilamos el programa de fuerza bruta con
   `g++ OrdenacionVectoresBruto.cpp -o ordenaVectores`

2. Compilamos el programa de Divide y Vencerás con
   `g++ -std=gnu++0x -o mezcla genera-mezclavectores.cpp`

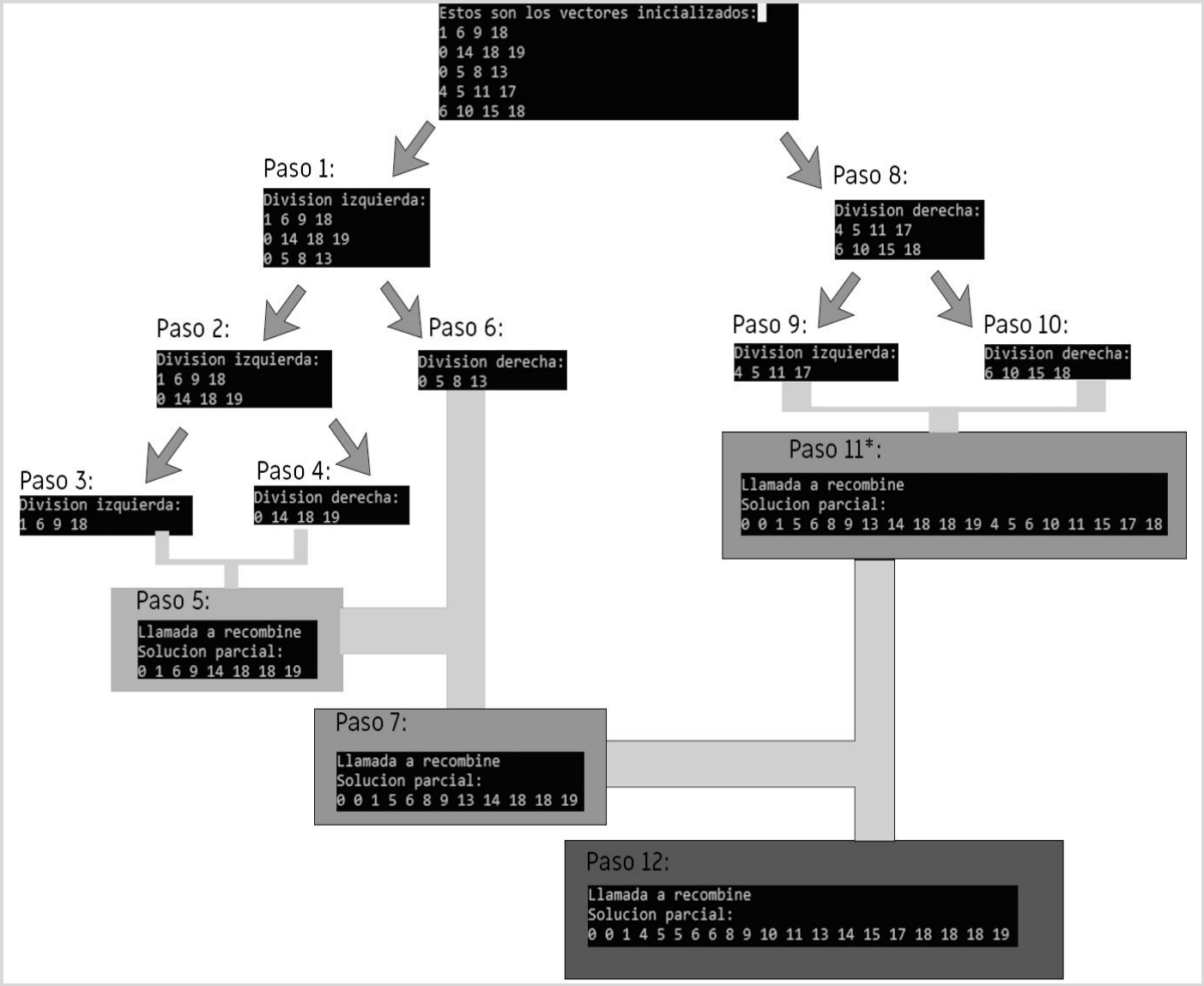
3. Ejecutamos el programa de fuerza bruta
   `./mimacro.sh`
   Esta macro básicamente lo único que hace es asignar un número de vectores y de elementos, se los pasa al programa y va aumentando el número de vectores. La salida la redirecciona a un .dat.

```
#!/bin/bash
let n=50
let k=100
while [ $k -le 2000 ];do
  echo $k
  ./ordenaVectores $n $k >> salidaAumentaPrueba.dat
  let k=$k+100
done
```

4. Ejecutamos el programa Divide y Vencerás.
   Este es un ejemplo de ejecución de la mezcla DyV con valores bajos para poder realizar una traza del funcionamiento (para sacar los ficheros de datos he usado un script de bash ./script.sh que ejecuta el programa con distintas entradas de datos en bucle y lo redirecciona a un fichero):

```
Estos son los vectores inicializados:
1 6 9 18
0 14 18 19
0 5 8 13
4 5 11 17
6 10 15 18
Division izquierda:
1 6 9 18
0 14 18 19
0 5 8 13
Division izquierda:
1 6 9 18
0 14 18 19
Division izquierda:
1 6 9 18
Division derecha:
0 14 18 19
Llamada a recombine
Solucion parcial:
0 1 6 9 14 18 18 19
Division derecha:
0 5 8 13
Llamada a recombine
Solucion parcial:
0 0 1 5 6 8 9 13 14 18 18 19
Division derecha:
4 5 11 17
6 10 15 18
Division izquierda:
4 5 11 17
Division derecha:
6 10 15 18
Llamada a recombine
Solucion parcial:
0 0 1 5 6 8 9 13 14 18 18 19 4 5 6 10 11 15 17 18
Llamada a recombine
Solucion parcial:
0 0 1 4 5 5 6 6 8 9 10 11 13 14 15 17 18 18 18 19
La solucion final es:
0 0 1 4 5 5 6 6 8 9 10 11 13 14 15 17 18 18 18 19
```

Para verlo de una forma algo más clara vamos a ramificar las llamadas que se realizan usando un árbol:



4. Conclusiones

Las conclusiones que hemos inferido de esta práctica son las siguientes:

- En esta práctica hemos podido comprobar que la técnica Divide y vencerás es útil pero no infalible: mientras que en la ordenación de vectores la eficiencia mejoraba, no es una técnica adecuada para el problema de la traspuesta de una matriz.
    - La técnica “Divide y vencerás” no siempre es equivalente a una mayor eficiencia: es por esto que a lo largo del curso veremos más técnicas de mejora de eficiencia, ya que no existe una manera universal de mejorar la eficiencia de un algoritmo.
- Reafirmamos lo estudiado en clases de teoría respecto al diseño de modelos Divide y vencerás, aplicándolos con recursividad y búsqueda de un caso base.