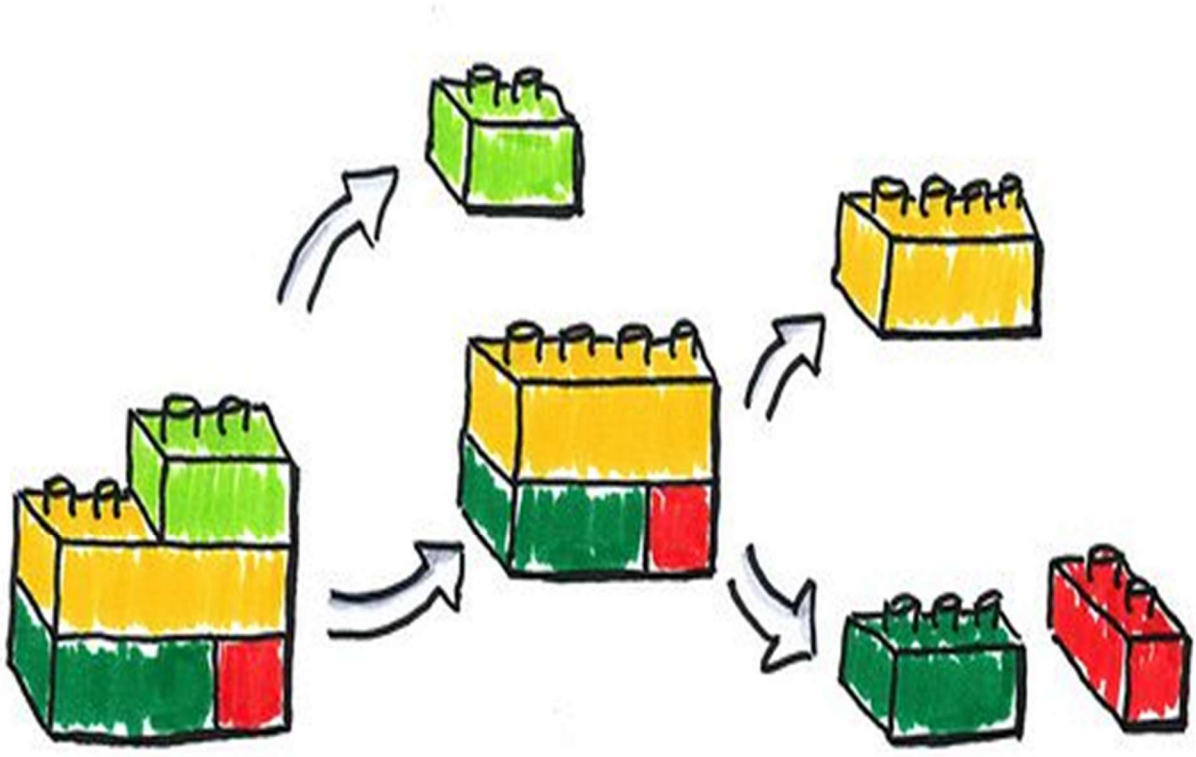
PRÁCTICA 2: DIVIDE Y VENCERÁS

- Ahmed El Moukhtari Koubaa
- Damián Marín Fernández
- Eduardo Segura Richart
- Jesús Martín Zorrilla



ÍNDICE

1. Descripción del problema.

Para esta práctica lo que se nos pide es resolver 2 problemas. Uno de estos problemas es común a todos los grupos, es decir, debe ser resuelto por todo grupo que realice la práctica, mientras que el otro es uno de los 4 problemas descritos en el guion de la práctica y asignado por sorteo. Así pues, el primero de los problemas es la traspuesta de una matriz, es decir, dada una matriz cuadrada con 2^n elementos debemos convertirla en su traspuesta. Para resolver este problema se dan dos algoritmos, una versión por fuerza bruta (fb) y otra por divide y vencerás (dv). La versión por fuerza bruta es simple pero ineficiente y la divide y vencerás es algo más complicada de programar, pero más eficiente. Respecto al segundo problema, a nuestro grupo se le ha asignado por sorteo el problema 3.4 del guion de prácticas, es decir, encontrar los elementos repetidos de un vector. Al igual que en el problema anterior se han realizado dos versiones, una por fuerza bruta y otra por divide y vencerás y al igual que en el caso previo, la versión por fuerza bruta es simple pero ineficiente y la alternativa divide y vencerás es más complicada de programar, pero más eficiente.

Ya descritos los problemas a resolver, describamos que más se nos pedía hacer con respecto a estos problemas. Se nos pide hallar la eficiencia teórica, es decir, mediante el análisis del código, también se nos pide la eficiencia empírica consistente en realizar ejecuciones con diversos tamaños (los tamaños van a depender de cada algoritmo) y la eficiencia hibrida, la cual se basa en las previas eficiencias. También se nos pide dar un valor umbral a partir del cual resulte más eficiente usar una versión que otra. Además de lo citado previamente también se nos pide describir un caso de ejecución y demostrar así que se ha dado una solución correcta.

Aunque no es parte del problema, en el guion de la práctica también se nos pide realizar esta memoria y preparar una presentación resumida para su exposición en clase.

2. Matriz traspuesta por fuerza bruta.

Descripción detallada del algoritmo.

El objetivo es realizar un algoritmo que calcule la traspuesta de una matriz cuadrada con n=2^k. En este primer caso se resolverá por el método de fuerza bruta. Como ya hemos mencionado, la idea del algoritmo es calcular la traspuesta de una matriz. En el caso de calcularlo por la fuerza bruta, consiste en recorrer las posiciones que quedan por debajo de la diagonal e intercambiarlas con su simétrica con respecto de la diagonal.

Cálculo de la eficiencia teórica

La parte del código que se encarga de calcula la traspuesta es la siguiente:

```
// Modifica la matriz pasada convirtiendola en la traspuesta
void TrasponerMatriz(int ** m, const int dim){
  for (int i = 0; i < dim; ++i){
    for (int j = i+1; j < dim; ++j){
      int intercambia = m[i][j];
      m[i][j] = m[j][i];
      m[j][i] = intercambia;
    }
  }
}
```

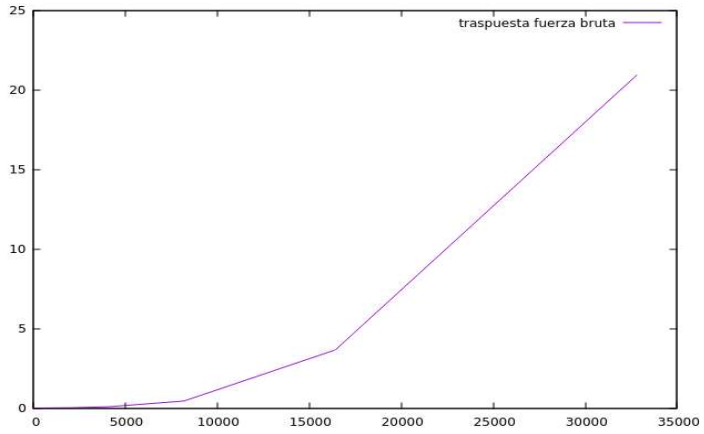
A continuación, vamos a calcular la eficiencia teórica del algoritmo para una matriz de tamaño n, con $n=2^k$. Para ello, nos centramos en el bucle de mayor profundidad, y podemos ver que en él se hacen 3 operaciones, siendo cada una de ella de O(1), y notamos que dicho bucle recorre desde j=i+1 hasta j=dim-1. Por consecuencia, se harían un total de 3_ioperaciones. Por tanto, este bucle se repetirá desde i=0 hasta i=n-1. De esta forma, se obtendría la siguiente fórmula:

$$\sum_{i=0}^{n-1} 3i = \frac{3n \times n}{2} = \frac{3}{2}n^2 \in O(n^2)$$

Tamaño	tiempo	Opt
4	1.6e-06	1.6e-06
8	1.7e-06	1.7e-06
16	2.2e-06	1.8e-06
32	4.1e-06	2.1e-06
64	1.03e-05	3.8e-06
128	3.7e-05	9.4e-06
256	0.0001402	3.13e-05
512	0.0006555	0.0001807
1024	0.0038022	0.001129
2048	0.0168626	0.0083153
4096	0.0799885	0.0562152
8192	0.437094	0.31406
16384	3.6398	2.1247
32768	20.9105	16.0097

Cálculo de la eficiencia empírica

Para calcular la eficiencia empírica, hemos ejecutado el algoritmo para 14 tamaños distintos, siendo el primero de ellos 4 y aumentado de forma geométrica multiplicando por 2, siendo el último de ellos 32768. Los resultados son los siguientes:



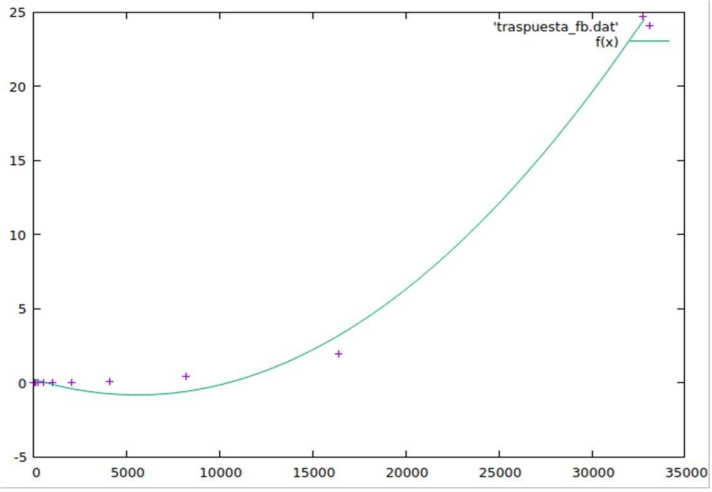
Cálculo de la eficiencia híbrida

Para calcular la eficiencia híbrida, necesitamos ajustar los datos obtenidos en la eficiencia empírica para $f(x)=ax^2 + bx + c$, haciendo uso de la herramienta Gnuplot, y se obtienen los siguientes datos:

a = 3,41959e-08

b = - 0,000380884

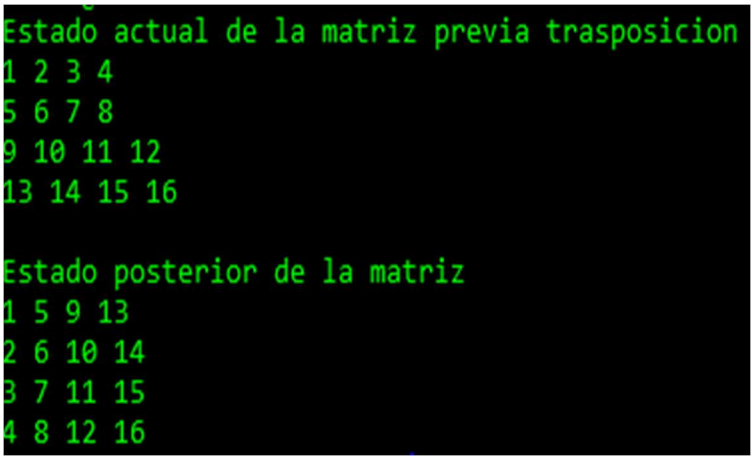
c = 0,225173



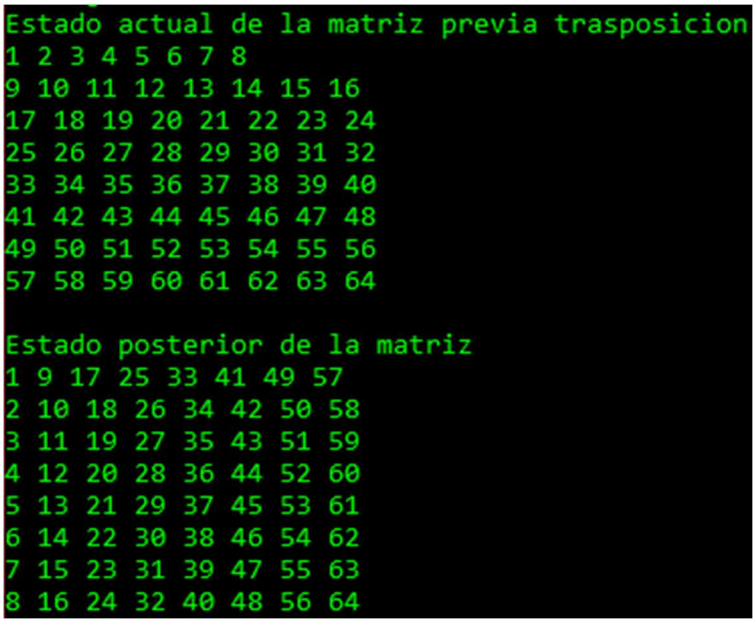
Por último, se adjunta una foto de la gráfica obtenida:

Caso de ejecución

Para 2 tamaños, 4 y 8:



Los elementos se encuentran enumerados de izquierda a derecha y de arriba abajo, para tamaño 4 es evidente que la trasposición fue correcta.

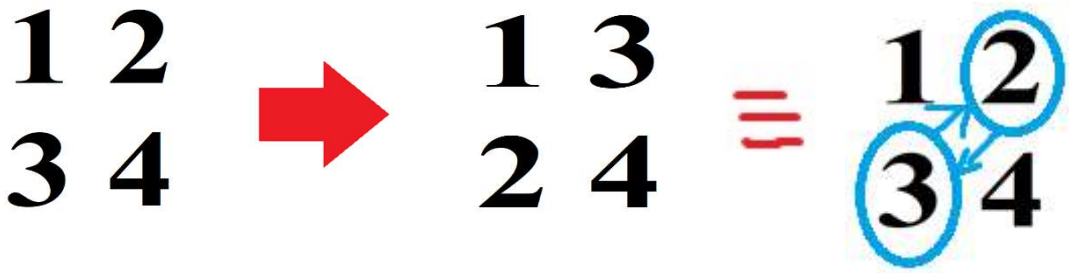


En este caso al igual que el anterior se consigue que las filas previas sean las actuales columnas y viceversa, también es una trasposición correcta como era de esperar.

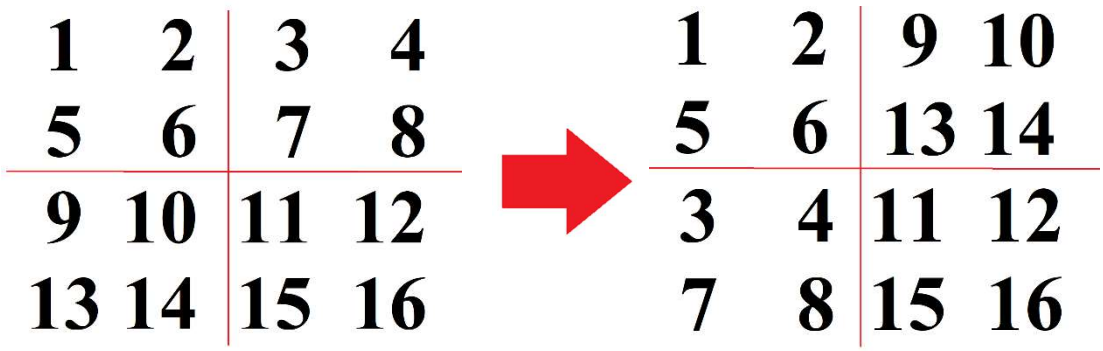
3. Matriz traspuesta por Dyv.

Descripción detallada del algoritmo.

El algoritmo que se ha creado para llevar a cabo esta tarea mediante DyV es simple. Consiste en dividir la matriz original en 4 submatrices, realmente la división que se intenta hacer es dividir la matriz en solamente 2 subprocesos, el problema es que una matriz es una estructura bidimensional y por ello no se divide en 2 y tampoco se divide en 8 particiones, por ejemplo, ya que resultaría más ineficiente y complicado de unir a posteriori. Así pues, partiendo del caso básico que es una matriz de dimensión 2, es decir, 2*2 tenemos que para conseguir la traspuesta de esta basta con intercambiar los elementos de la diagonal.



Con este simple movimiento tenemos hecha la traspuesta, probemos ahora con una matriz de mayor tamaño, de dimensión 4, por ejemplo. Dividiendo la matriz en 4 submatrices haciendo corresponder cada una de estas a cada uno de los elementos de la matriz básica previamente descrita e intercambiándolos, nos quedaría la siguiente matriz.



Al hacer el intercambio nos queda la siguiente matriz, que es parecida pero no igual a la matriz traspuesta, para ello debemos tomar las dos matrices diagonales por separado y hacer un intercambio de los elementos en sus diagonales. Quedándonos así la siguiente matriz. Esta matriz es la traspuesta de la matriz original.

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

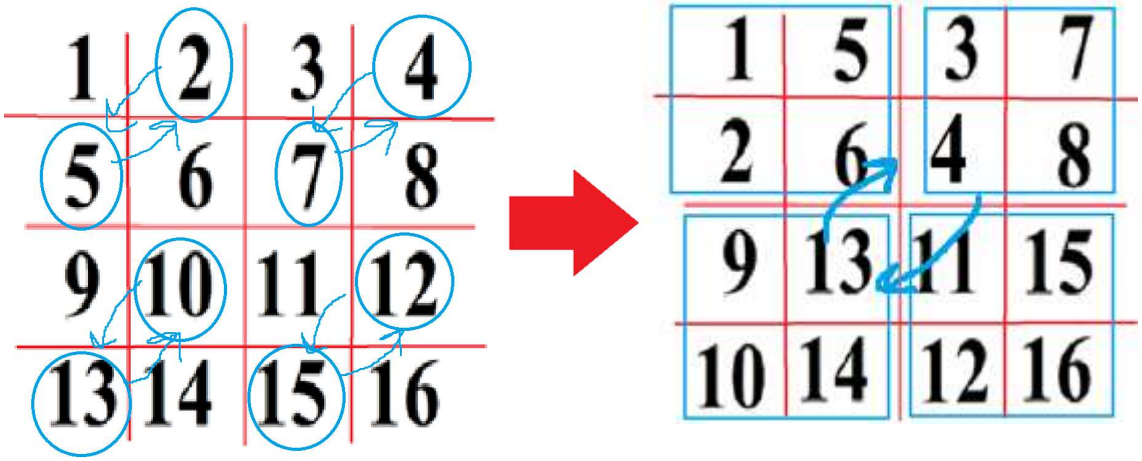
Es fácil darse cuenta de que lo que estamos haciendo en el anterior caso es:

1.- Dividir en 4 submatrices la matriz original.

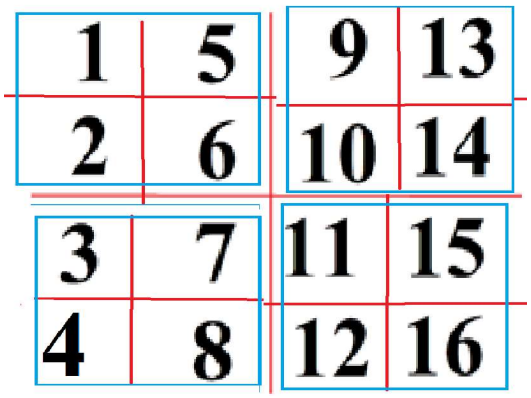
2.- Intercambiar las submatrices diagonales (2 y 3).

3.- Repetir desde el paso 1 hasta llegar al caso base (matriz de dimensión 2).

Pues el algoritmo implementado funciona de manera similar, solo que se altera el orden los pasos, es decir, divide la matriz en 4 submatrices hasta el caso base y en cuanto llegue empieza a intercambiarlas de "abajo" a "arriba" describiendo el funcionamiento típico de un algoritmo DyV.



Y al intercambiar las submatrices 2 y 3 nos quedaría la traspuesta, que es la siguiente matriz:



Cálculo de la eficiencia teórica

Para hallar la eficiencia teórica haremos un análisis (aunque no muy en detalle) del código que implementa este algoritmo. Vemos así que:

Para hallar la eficiencia teórica hay que resolver la siguiente recurrencia:

T(n) = 4*T(n/4) + n

- Hacemos el siguiente cambio de variable para ello → n = 4^k
- Nos queda → T(4^k) = T(4^(k-1)) + 4^k
- Expresamos en función de k → tk = tk-1 + 4^k
- Resolvemos con la ecuación característica → (x-4)*(x-4)
- Nos queda → c1*4^k + c2*k*4^k
- Deshacemos el cambio → c1*n + c2*n*log(n)
- Luego este algoritmo es del orden → O(n*log(n)+n)
- Expresamos la eficiencia sin valores despreciables → **O(n*log(n))**

Nos queda así que esta versión DyV es O(nlogn) que no es la mejor posible, pero sí que es bastante mejor comparada con la versión por fuerza bruta

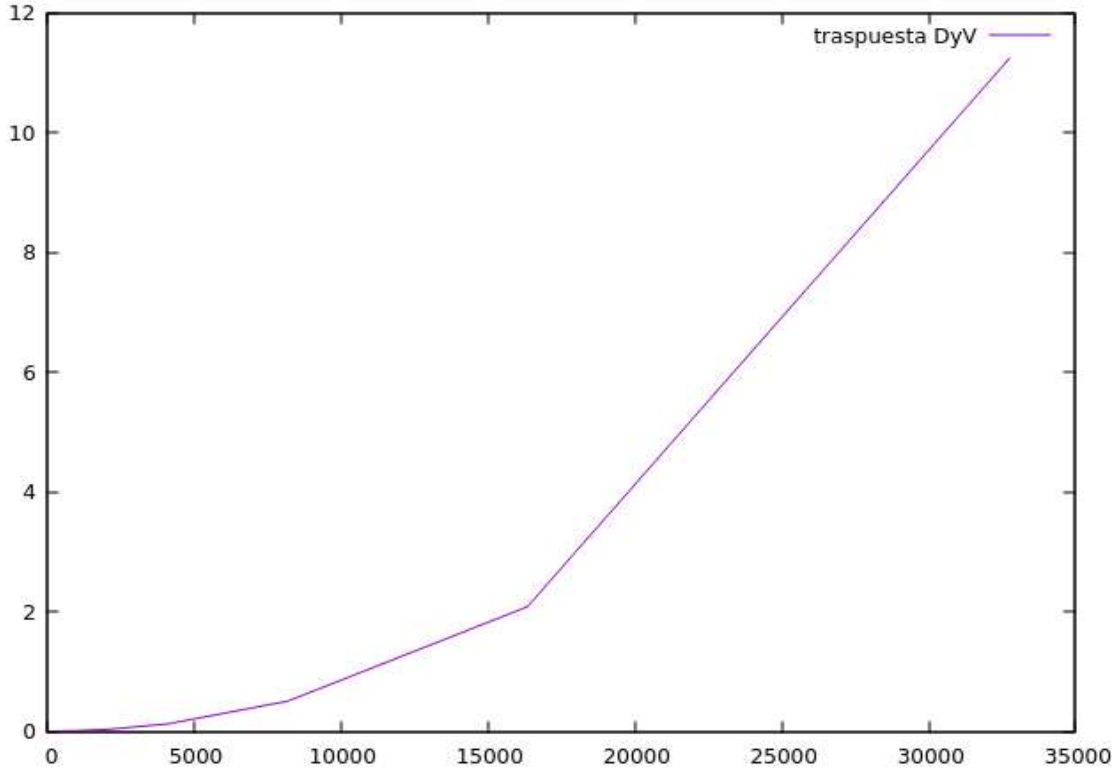
Cálculo de la eficiencia empírica

Para llevar a cabo estos cálculos vamos a hacer múltiples ejecuciones del algoritmo desde el valor más pequeño, hasta el valor más grande posible, es decir, desde 2 hasta 2^15 duplicando en cada ejecución el tamaño. No se pueden asignar mayores valores de entrada ya que para 2^16, por ejemplo, tenemos un consumo de memoria de 2^16*2^16*2^2, es decir, 2^34 Bytes (16 GB) de memoria lo cual es insostenible para los equipos con los cuales estamos realizando estas ejecuciones.

Tamaño	tiempo	Opt
4	3.1e-06	3.3e-06
8	3.4e-06	3.3e-06
16	4.9e-06	6.2e-06
32	1e-05	1.05e-05
64	2.93e-05	2.05e-05
128	0.0001078	7.21e-05
256	0.0004066	0.0002814
512	0.0017361	0.0013057
1024	0.0073086	0.0063488
2048	0.0293867	0.0239823
4096	0.122589	0.0976504
8192	0.502631	0.42082
16384	2.0873	1.73995
32768	11.2342	9.70815

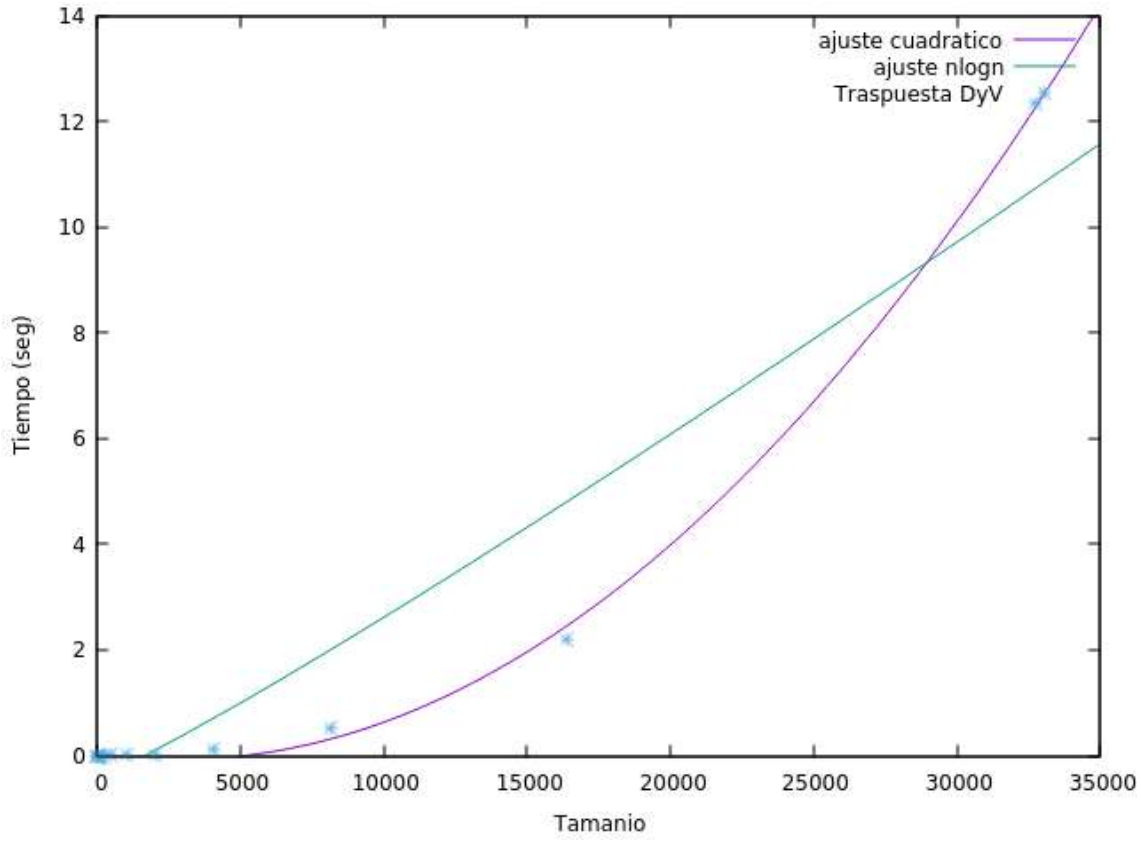
Los resultados obtenidos son los siguientes:

Entre el programado compilado con optimización y el programa no hay una gran diferencia, al menos no para estos tamaños. La diferencia se apreciaría bastante más si pudiésemos tamaños más grandes, pero esto no es posible por las razones mencionadas previamente.



Cálculo de la eficiencia híbrida

Partiendo del cálculo de las eficiencias anteriores y usando las herramientas que nos proporciona la asignatura como es ente caso Gnuplot, hallamos con un comando simple la eficiencia hibrida, además de representarla gráficamente. La eficiencia es:



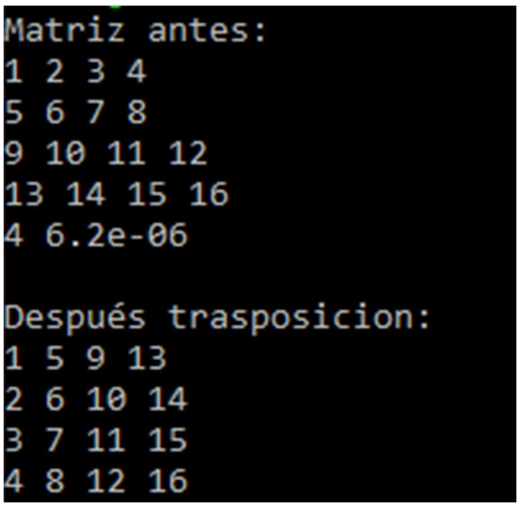
Siendo la fórmula de la eficiencia híbrida la siguiente:

f(n) = 0.000227427*log(n)*n - 0.000227427*n + 0.000227427

Los resultados obtenidos no se ajustan totalmente a la curva O(n*log(n)) calculada empíricamente, esto es debido a que a eficiencia empírica no coincide con la teórica y es O(n^2) ya que la unión de las submatrices es de ese orden y no se apreciaba realmente así en el análisis del código. También recalcar que no es posible hacer mediciones para tamaños mayores de 2^15 pues la memoria se agota.

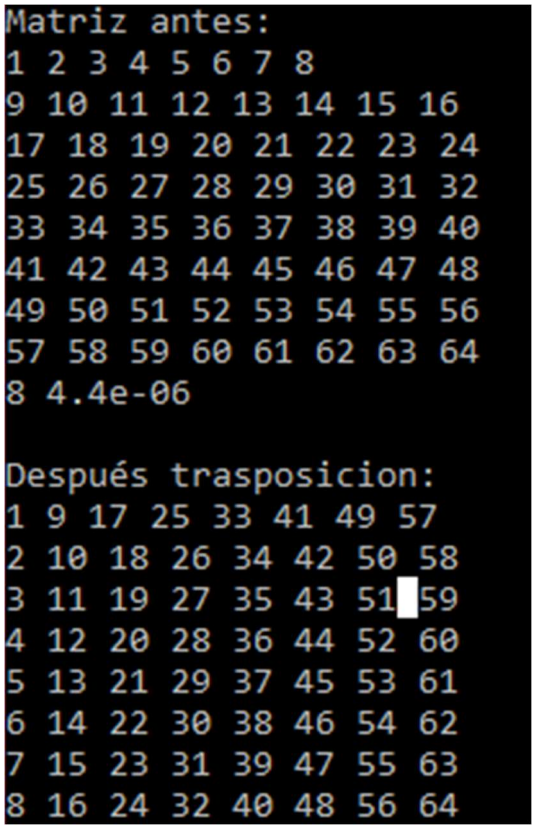
Caso de ejecución

Se van a mostrar dos ejecuciones, para tamaños 4 y 8.



Tenemos elementos de la matriz enumerados desde izquierda a derecha y desde arriba abajo para poder identificar de forma más sencilla el resultado.

Observamos que la matriz ha sido traspuesta con éxito pues se ha obtenido el resultado esperado, las anteriores filas son las actuales columnas y viceversa.



En este caso al igual que en al anterior también están ordenados los elementos siguiendo el mismo criterio. La trasposición ha sido correcta y podemos ver esto fácilmente gracias a la enumeración de los elementos.

4. Eliminar repetidos por fuerza bruta.

Descripción detallada del algoritmo.

El algoritmo de búsqueda de repetidos tiene como objetivo buscar aquellos elementos los cuales se encuentren más de una vez en el vector y eliminarlos dejando solo una aparición. Para ello se recorren los elementos y para cada elemento en la posición se hace una comparación con los n-k elementos siguientes.

```
//para quitar repetidos en el vector generado
int aux;
    for(int i=0; i<n; i++){
        for(int j = i+1; j < n; j++){
            if(T[i] == T[j]){
                aux = j;
                while(aux < n){
                    T[aux] = T[aux+1];
                    ++aux;
                }
                --n;
                --j;
            }
        }
    }
```

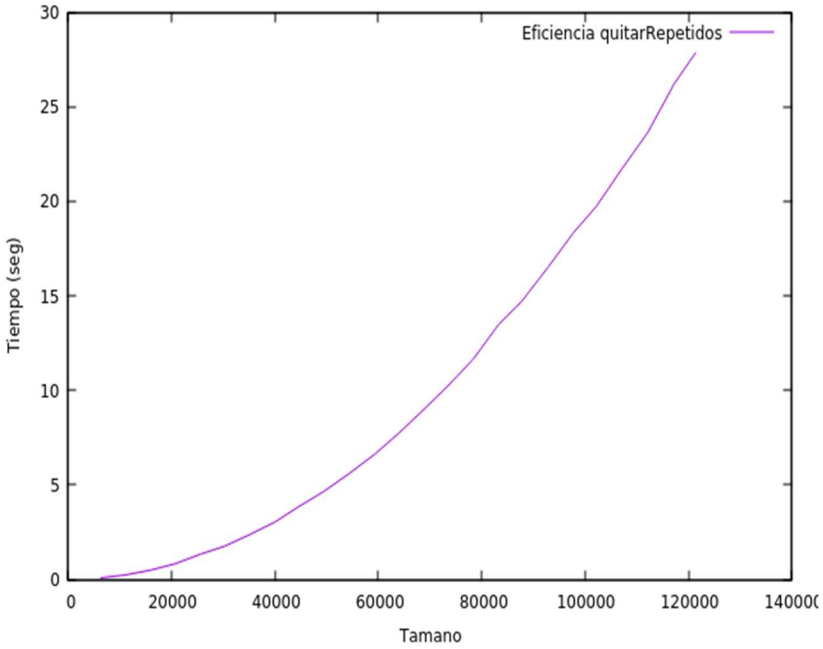
Cálculo de la eficiencia teórica

Como podemos observar en la captura del código este algoritmo usa dos bucles anidados, en cada iteración del bucle interno hace las correspondientes comparaciones y de ser desplaza los n-k elementos restantes hacia la izquierda. Por estas razones es más que evidente que la eficiencia teórica del algoritmo es O(n^2) para el peor caso y O(1) para el caso de un vector vacío.

Cálculo de la eficiencia empírica

Tamaño	tiempo
6314	0.077351
11154	0.237305
15938	0.484241
20776	0.828906
25648	1.3297
30302	1.7567
35197	2.37045
40079	3.04086
44623	3.84492
49434	4.63525
54266	5.57575
59328	6.61627
63970	7.72621
68845	8.98567
73551	10.2458
78363	11.6415
83296	13.487
87713	14.6903
92846	16.5195
97765	18.3429
102268	19.7659
107122	21.7133
112128	23.6367
116916	26.0533
121436	27.8372

Los resultados de las mediciones de tiempo para cada ejecución son las siguientes:



La gráfica y los datos nos confirman que la eficiencia es cuadrática pues el ritmo al que avanza lo verifica.

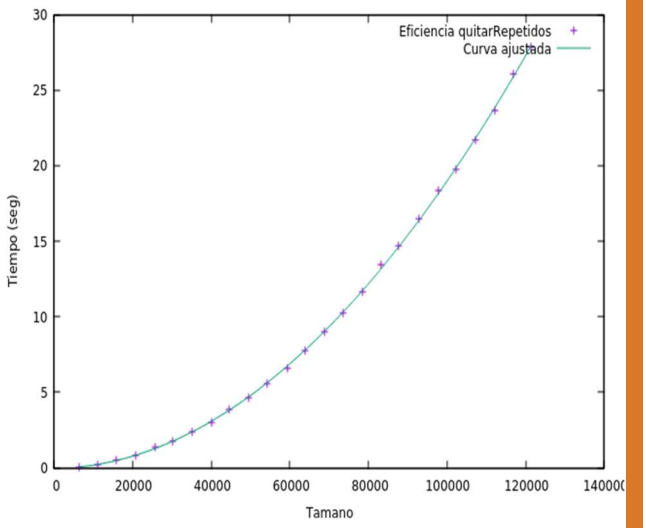
Cálculo de la eficiencia híbrida

Al ser la eficiencia Teórica: **O(n²) → T (n) = a0×n² + a1×n + a2**

Tras unas pocas iteraciones, nos sale los resultados del proceso de regresión de la función a ajustar, que son:

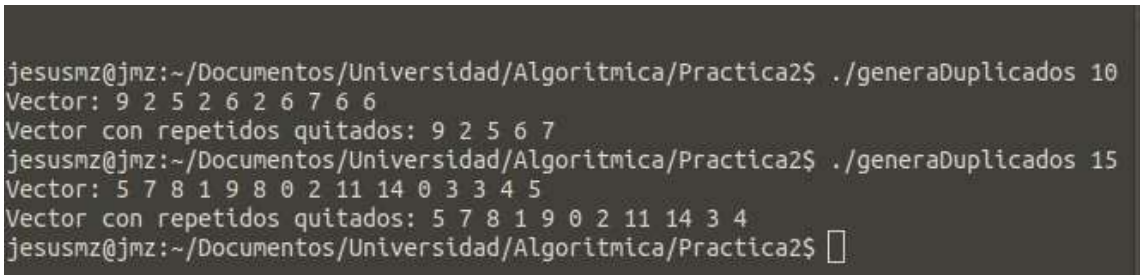
```
Final set of parameters            Asymptotic Sta
======================             ==============
a0              = 1.87415e-09      +/- 2.064e-11
a1              = 2.8429e-06       +/- 2.716e-06
a2              = -0.0410974       +/- 0.07559

correlation matrix of the fit parameters:
                a0     a1     a2
a0              1.000
a1             -0.972  1.000
a2              0.790 -0.894  1.000
```



Caso de ejecución

Vamos a comprobar que el programa que implementa este algoritmo funciona correctamente con 2 ejemplos, uno de tamaño 10 y otro de tamaño 15:



Vemos como se genera un vector con elementos aleatorios y después de la ejecución del algoritmo los valores repetidos desaparecen, mejor dicho, han sido eliminados. Esto se verifica tanto para 10 como para 15 elementos y también se da para un caso de n elementos, aunque verificar esos ejemplos es más complicado para un tamaño grande y por eso se han realizado estos casos y no otros de mayor tamaño.

5. Eliminar repetidos por DyV.

Descripción detallada del algoritmo

Se nos pide implementar un algoritmo que elimine los elementos repetidos en el vector y que sea del tipo Divide y vencerás. La implementación de este algoritmo no es del todo posible, pues para eliminar los elementos repetidos hay que tratar con todo el vector en cada iteración, es decir, al dividir los datos y tratar cada subvector como un subproblema se están excluyendo posibles valores repetidos y para garantizar el correcto funcionamiento del problema se deberían "comunicar" los subproblemas, en cuyo caso estaríamos resolviendo el problema original, pero de forma más ineficiente.

Por las razones citadas previamente se ha llegado a la conclusión de que el algoritmo DyV de eliminación de repetidos no es posible, sin embargo, podemos aplicar divide y vencerás para hacer más eficiente esta tarea.

La solución dada al problema consiste pues en la ordenación de los datos para facilitar la búsqueda de elementos repetidos. Para la ordenación, se divide el vector en 2 particiones (excluyentes) sucesivamente hasta que queden vectores 2 o menos elementos y tras esto ordenar los elementos, después ir mezclando las particiones en orden (lo cual tiene un costo lineal). Este método de ordenación coincide con uno de los vistos en clase, el mergesort. Teniendo el vector completamente ordenado, se recorre este eliminando los valores repetidos (costo lineal).

A continuación, se adjunta una imagen con las funciones que implementan las tareas previamente descritas. Estas funciones son usadas por otra función llamada "EliminaRepetidosDV" la cual se encuentra en el fichero con el mismo nombre.

```
void EliminaRepetidosDV(int * v, int & n){
  Ordena(v, 0, n);                  // Ordenar elementos
  EliminaRepetidosOrdenado(v,n);    // Eliminar elemenos repetidos ya ordenados
}
```

```cpp
void EliminaRepetidosOrdenado(int * v, int & n){
  int esc, lec;                     // Para leer y escribir
  lec = esc = 0;                    // Posicion inicial es 0

  while (lec < n){                  // Recorrer todos los elementos
    ++lec;                          // Incrementar indice de lectura
    if (v[esc] != v[lec]){          // Si los elementos actuales son diferentes
      esc++; v[esc] = v[lec];}      // escribirlos, en otro caso pasar al siguiente
  }

  n = esc;                          // Tamaño vector es num elementos escritos
}

/*****************************************************************************/
// Elimina los elementos repetidos del vector que empieza en
// inicio y acaba en final
void Ordena(int * v, int inicio, int final){
  int num_elem = final - inicio;                      // calcular numero total de elementos
  if (num_elem <= 2){                                 // Si el vector tiene 2 o menos elementos
    if (num_elem == 2 && v[inicio] > v[final-1]){     // si tiene 2 elementos
      int intercambia = v[inicio];                    // intercambiar
      v[inicio] = v[final-1];
      v[final-1] = intercambia;
    }
  }
  else {                                              // Si tiene mas de 2 elementos aplicar recursividad
    int medio = (final+inicio)/2;                     // Calcular el punto medio del vector
    Ordena(v, inicio, medio);                         // Aplicar sobre la parte izda
    Ordena(v, medio, final);                          // Aplicar sobre la parte dcha

    int izda = inicio, dcha = medio;                  // Apuntar a primer elemento de cada vector
    int * res = new int[num_elem];                    // Reservar espacio para vector auxialiar
    int anterior, i = 0;                              // i es el indice que recorrera el vector

    while (izda < medio && dcha < final){             // Recorrer hasta que algun vector acabe
      if (v[izda] < v[dcha]) res[i] = v[izda++];      // Si izda es menor que dcha escribir izda
      else                   res[i] = v[dcha++];      // En otro caso escribir dcha
      ++i;                                            // Avanzar posicion de escritura
    }

    while (izda < medio) res[i++] = v[izda++];        // Escribir posibles elementos de izda
    while (dcha < final) res[i++] = v[dcha++];        // Escribir posibles elementos de dcha

    memcpy(v+inicio, res, num_elem*sizeof(int));      // Copiar los datos ordenados al vector original
    delete [] res;                                    // Liberar memoria reservada
  }
}
```

Cálculo de la eficiencia teórica

La función que implementa el algoritmo usa las dos mencionadas previamente, para analizar la eficiencia teórica del algoritmo completo analizaremos la de esos dos por separado.

El primero es ordena, este algoritmo divide el vector en 2 hasta encontrar 2 o menos elementos, luego los mezcla. Sabemos que este algoritmo coincide con el mergesort, el cual fue analizado la semana pasada y cuya eficiencia es O(n*log(n)).

El segundo algoritmo elimina los repetidos ordenados y es trivial ver que es lineal, pues se recorre el vector con índices de lectura y escritura, si los elementos referenciados en cada iteración son diferentes se escribe, en caso contrario se avanza en lectura. Así pues, nos quedaría que es O(n).

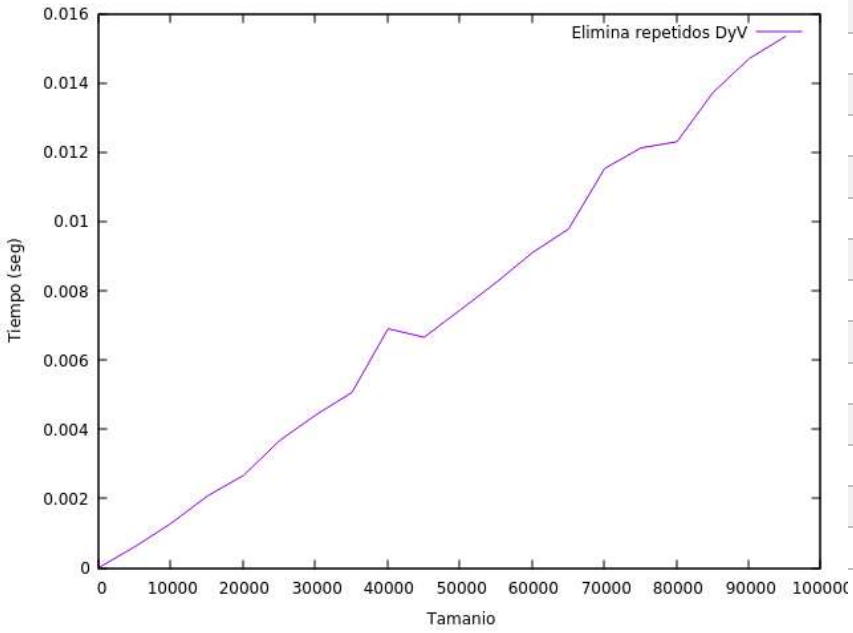
La eficiencia total de nuestro algoritmo es la suma de estas dos quedándonos pues O(n*log(n)+n), que es simplemente O(n*log(n)).

Cálculo de la eficiencia empírica

Para esta eficiencia se han realizados las siguientes mediciones obteniendo los siguientes resultados:

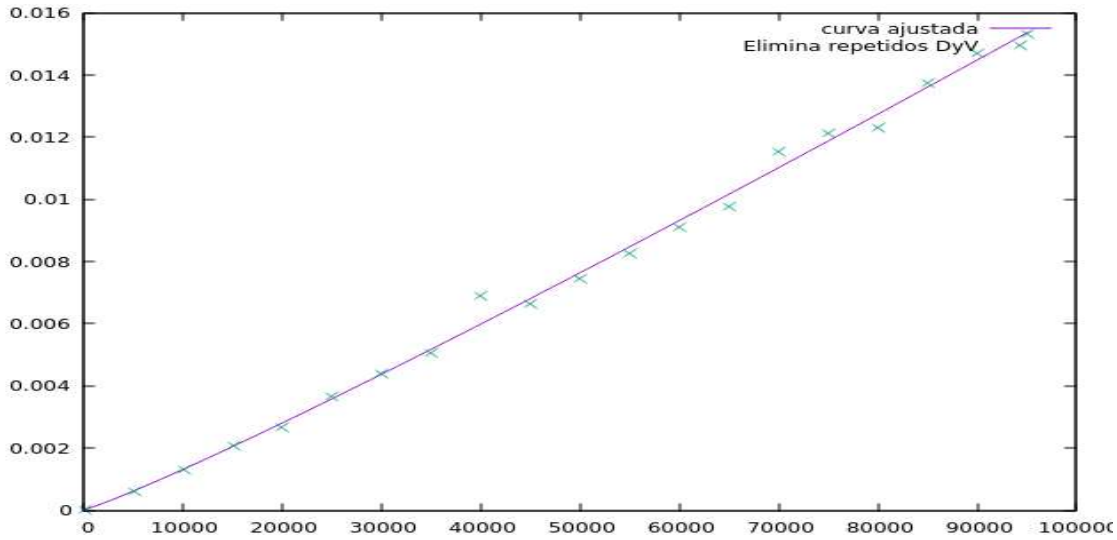
Podemos ver que el tiempo es bastante bajo en comparación con la versión por fuerza bruta. Como dato curioso la implementación del algoritmo de ordenación por mezcla usado es más eficiente que la usada en la anterior práctica.

Tamaño	tiempo
100	1.13e-05
5100	0.0006161
10100	0.0012946
15100	0.0020756
20100	0.0026694
25100	0.0036772
30100	0.0044156
35100	0.0050642
40100	0.0069055
45100	0.0066585
50100	0.0074444
55100	0.0082465
60100	0.0091049
65100	0.0097859
70100	0.0115315
75100	0.0121244
80100	0.0123026
85100	0.0137282
90100	0.0147036
95100	0.0153416



Cálculo de la eficiencia híbrida

Con lo visto en los anteriores apartados sabemos que la eficiencia de este algoritmo es O(n*log(n)) así pues, la fórmula que tenemos que seguir es a0*x*log(n) + a1. Los resultados de este cálculo con la herramienta Gnuplot son:



Caso de ejecución

Se muestran a continuación 2 ejecuciones, para tamaños 15 y 30, de manera que sea fácil demostrar el correcto funcionamiento del algoritmo.

```
El tamaño del vector es: 15
El vector es: 14 13 0 6 4 8 3 7 1 8 2 4 14 0 0
El tamaño del vector es: 10
El vector es: 0 1 2 3 4 6 7 8 13 14
```

```
El tamaño del vector es: 30
El vector es: 2 5 9 18 6 16 15 22 18 1 26 23 10 0 11 2 3 14 2 28 22 11 8 11 16 23 3 20 5 0
El tamaño del vector es: 19
El vector es: 0 1 2 3 5 6 8 9 10 11 14 15 16 18 20 22 23 26 28
```

Podemos ver en los anteriores ejemplos que se procesan correctamente los datos. Se han eliminado todos los elementos repetidos y el vector ha sido devuelto ordenado (como consecuencia de la aplicación del algoritmo), podemos verificar la corrección del algoritmo haciendo las cuentas a manos pues 10 es un tamaño trivial.

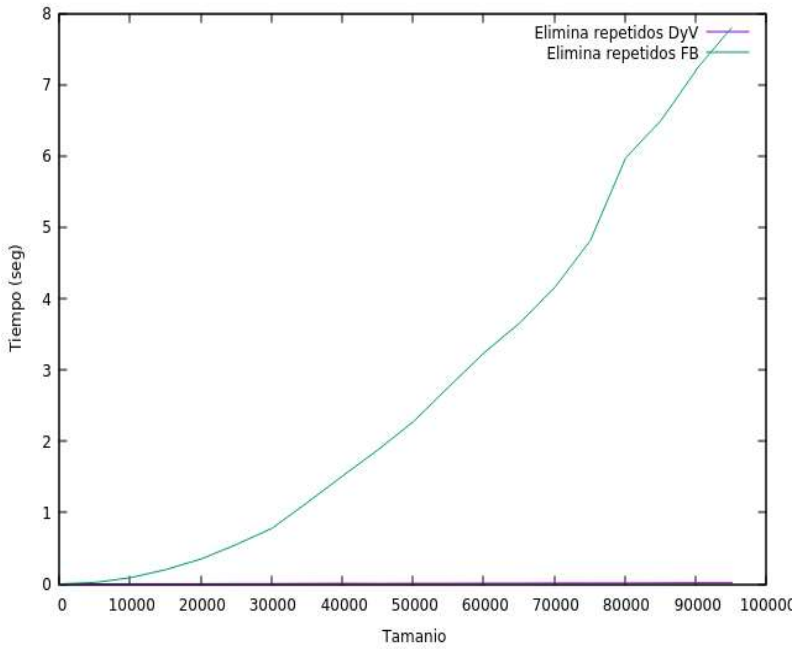
6. Comparativa traspuesta.

Vamos a comparar las dos versiones realizadas de la trasposición de una matriz. Para empezar, sabemos que las eficiencias teóricas son O(n^2) y O(n*log(n)) para la versión fuerza bruta y la versión DyV respectivamente, por lo que la versión DyV es mejor.

Empíricamente tenemos que ambas son cuadráticas, por lo que la versión DyV no es mejor que la versión por fuerza bruta, aunque sí que mejora el tiempo de ejecución a partir de un tamaño de entrada mayor a 4096, siendo hasta 2 veces más rápida. Luego nos quedaría que el mejor algoritmo en tiempo de ejecución es la versión DyV, aunque no se consigue una mejora de eficiencia.

7. Comparativa elimina repetidos.

La versión por fuerza bruta es de eficiencia teórica O(n*2) mientras que la eficiencia de la versión que usa DyV es O(n*log(n)), la cual es una mejora realmente considerable para esta tarea a partir de 0 elementos. Recalcamos que la versión DyV no es completamente DyV pues lo que se hace realmente es una ordenación previa del vector y la eliminación lineal de los elementos repetidos, ya que las particiones del vector son excluyentes y eso no permite la correcta búsqueda de elementos repetidos en el vector.



Empíricamente las eficiencias coinciden con las teóricas y la diferencia en tiempo es abismal y totalmente rotunda. En el tiempo que la versión por fuerza bruta procesa 200.000 datos, la versión divide y vencerás proceso 200.000.000, que son 1000 veces más (debido a la eficiencia logarítmica evidentemente).

8. Conclusiones.

En general las versiones que implementan divide y vencerás han mejorado todos los tiempos de ejecución, sin embargo, no siempre se mejora el orden de eficiencia y no siempre es posible hacer una implementación divide y vencerás total de un algoritmo.

En el problema de la matriz traspuesta tenemos que el orden de eficiencia no mejora, pero que el tiempo de ejecución para los mismos tamaños de entrada se reduce en hasta un 50%, la cual es una gran rebaja de tiempo, pero no tan buena como reducir el orden de eficiencia.

En el segundo problema asignado la eliminación de repetidos (problema 3.4 del guion) la eficiencia se ve tremendamente mejorada, de una cuadrática a una n*log(n), sin embargo, como ya se mencionó previamente no se realiza una implementación total de un algoritmo de eliminación de repetidos, sino que el algoritmo va ordenando los elementos para poder luego hacer una búsqueda más de repetidos más eficientes.

En conclusión, usar divide y vencerás no es siempre posible y no se garantiza que siempre se mejore el orden de eficiencia, pero al menos en nuestros 2 problemas resueltos lo que se ha mejorado bastante son los tiempos de ejecución, lo cual implica que hemos realizado una misma tarea de manera mucho más rápida.

FIN