PRÁTICA 2:

ALGORITMOS DIVIDE Y VENCERÁS

Grupo hibisco:
Quintin Mesa Romero
Noura Lachhab Bouhmadi

ÍNDICE

1.INTRODUCCIÓN 1

2. LISTADO DEL HARDWARE UTILIZADO EN LA PRÁCTICA 3
Ordenador de Quintín Mesa Romero: 3
Ordenador de Noura Lachhab Bouhmadi: 3

3. SOFTWARE UTILIZADO 4

4. EJERCICIO 1 5
4.1 Versión fuerza bruta 5
4.1.1 Análisis teórico 5
4.1.2 Ejemplo de ejecución 5
4.1.3 Análisis empírico 6
4.1.4 Análisis híbrido 6
4.2 Versión divide y vencerás I 7
4.1.2 Ejemplo de ejecución 7
4.1.3 Análisis empírico 8
4.1.4 Análisis híbrido 8
4.3 Comparativa entre los algoritmos 9

5. EJERCICIO 2 14
5.1 Versión fuerza bruta 14
5.1.1 Análisis teórico 14
5.1.2 Ejemplo de ejecución 15
5.1.3 Análisis empírico 16
5.1.4 Análisis híbrido 18
5.2 Versión divide y vencerás 19
5.2.1 Análisis teórico 20
5.2.2 Ejemplo de ejecución 21
5.2.3 Análisis empírico 21
5.2.4 Análisis híbrido 24
5.3 Comparativa entre los algoritmos 26
5.3.1 Comparativa entre los algoritmos cuando tenemos N elementos 26
5.3.2 Comparativa entre los algoritmos cuando tenemos K vectores 27
5.4 Conclusiones del Ejercicio 2 28

1. INTRODUCCIÓN

A menudo, en nuestro día a día se nos presentan problemas difíciles de distinta índole que tenemos que resolver. Muchas veces abordamos el problema en todo su grueso, intentando resolverlo de una vez. Esto normalmente nos lleva a desistir en la búsqueda de la solución, dada su complejidad o a emplear más tiempo del que deberíamos. Quizás la solución a esto está en dividir el problema inicial en partes más simples tantas veces como sea necesario (**algoritmo divide y vencerás**), hasta que la resolución de las partes se torne obvia, de tal forma que la solución del problema principal se construya con las soluciones encontradas. Aunque pudiera ser que, dependiendo del problema al que nos enfrentemos, interese más resolverlo de "forma bruta".

De esto precisamente es lo que trata la segunda práctica de la asignatura: **Algoritmos Divide y Vencerás**. El objetivo de la misma es comprender la utilidad de la técnica de "divide y vencerás", para resolver problemas de forma más eficiente que otras alternativas más directas o sencillas. La práctica se divide en dos partes, que se corresponden con la resolución de los dos ejercicios propuestos.

En una **primera parte**, se da solución al **Ejercicio 1:**

> *Dado un vector ordenado (de forma no decreciente) de números enteros v, todos distintos, el objetivo es determinar si existe un índice i tal que v[i] = i y encontrarlo en ese caso.*

Se presentan dos algoritmos que resuelven el problema que plantea el ejercicio, uno "obvio" o directo, y otro en el que se aplica la técnica "divide y vencerás". Se ha hecho un estudio empírico e híbrido de la eficiencia de ambos algoritmos.
Posteriormente, se analiza cuál de los dos algoritmos es mejor, en el caso de que los enteros del vector no tengan que ser necesariamente distintos entre sí.

La **segunda parte** de la práctica, aborda el **Ejercicio 2:**

> *Se tienen k vectores ordenados (de menor a mayor), cada uno con n elementos, y queremos combinarlos en un único vector ordenado (con kn elementos).*

En respuesta a las preguntas del ejercicio, se presentan dos algoritmos que dan solución al problema, un algoritmo clásico (se analiza el tiempo de ejecución del mismo) y otro más eficiente que el anterior.
Posteriormente, se ha hecho un análisis empírico e híbrido de la eficiencia de ambos algoritmos.

Para la realización de los experimentos con los algoritmos se han empleado los generadores de datos de entrada para cada uno de los problemas.

2. LISTADO DEL HARDWARE UTILIZADO EN LA PRÁCTICA

A continuación se presenta un listado de los dispositivos, junto con sus características, con los que se ha llevado a cabo esta práctica.

- Ordenador de Quintín Mesa Romero:
  - **Nombre del dispositivo**: Lenovo-Yoga-S740-14IIL
  - **CPU:** Intel® Core™ i7-1065G7 CPU @ 1.30GHz × 8
  - **Memoria RAM:** 16 GB
  - **Tarjeta gráfica:** Intel® Iris(R) Plus Graphics (ICL GT2)
  - **Sistema operativo:** Ubuntu 20.04.3 LTS

- Ordenador de Noura Lachhab Bouhmadi:
  - **Nombre del dispositivo**: GF63-Thin-10SCXR
  - **CPU:** Intel® Core™ i7-10750H CPU @ 2.60GHz × 12
  - **Memoria RAM:** 16 GB
  - **Tarjeta gráfica:** Intel® UHD Graphics (CML GT2)
  - **Sistema operativo:** Ubuntu 20.04.3 LTS

3. SOFTWARE UTILIZADO

A continuación se especifica todo lo relativo al software utilizado en la práctica.

- Para la **edición del código de los algoritmos** (programas escritos en el lenguaje de programación C++), se ha empleado el editor de código que por defecto viene instalado en Linux:

- Para la **compilación de los programas** se ha utilizado el compilador de *línea de órdenes* que compila y enlaza programas en C++, **g++**. En cuanto a la ejecución de los programas, se ha elaborado un script de bash que tiene la labor de ejecutar un algoritmo tantas veces como se especifique.
  En la siguiente imagen se aprecia el código del script, preparado para ejecutar el algoritmo de selección, para un número de componentes del vector que varía entre 100 y 200000, de 7996 en 7996 (para conseguir 25 tiempos de ejecución para tamaños de vector equiespaciados):

```
#!/bin/bash

echo "" >> salida_Ej1DYV.dat
i=100
a=200000

while [ $i -lt $a ]
do
  ./ejercicio1_DYV $i >> salida_Ej1DYV.dat
  let i+=7996
done
```

- La **generación de gráficas** se ha llevado a cabo mediante **gnuplot**, usando las correspondientes órdenes especificadas en el guión de la práctica.
- La **generación de tablas** con los tiempos de ejecución se han obtenido con **google spreadsheets.**
- Para la elaboración de la memoria de la práctica se ha utilizado la herramienta **google docs** y **google presentations** para la presentación.

4. EJERCICIO 1

Dado un vector ordenado (de forma no decreciente) de números enteros v, todos distintos, el objetivo es determinar si existe un índice i tal que v[i] = i y encontrarlo en ese caso.

4.1 Versión fuerza bruta

```
int buscaIgualIndice (int v[], int n)
{
    bool continua = true;
    int index = -1;

    for (int i = 0; i < n && continua; i++){
      if (v[i] == i){
          index = i;
          continua = false;
        }
    }

    return index;
}
```

Va **comparando** cada elemento del vector con su posición. Si el elemento en la posición i, v[i], coincide con i, se fuerza la salida del bucle y se devuelve dicho elemento.

4.1.1 Análisis teórico

Si observamos el código del algoritmo vemos que consta en primer lugar de dos sentencias que suponen tiempos constantes y, a continuación un bucle for que se ejecuta n veces (n≡tamaño del vector), cuyo cuerpo consta de sentencias simples que suponen tiempos constantes. Con lo cual, podemos afirmar que la eficiencia del algoritmo, en su versión obvia, es O(n).

4.1.2 Ejemplo de ejecución

A continuación, se ofrece un ejemplo de la ejecución de este algoritmo de fuerza bruta, para un valor de n=10. Téngase en cuenta que el vector sobre el que se va aplicar el algoritmo de fuerza bruta se ha creado mediante el generador 1, aportado por la profesora.

```
quintin@quintin-Lenovo-Yoga-S740-14IIL:~/Documentos/Documentos/DGIIM/2ºDGIIM/Informátic
a/Segundo_Cuatrimestre/Algorítmica/Prácticas/Practica2/Ejercicio1/Fuerza_Bruta$ ./Ejercicio1_Obvio 10
 -9 -8 -6 -4 -3 -2 1 4 8 9
Indice: 8 elemento:8
```

4.1.3 Análisis empírico

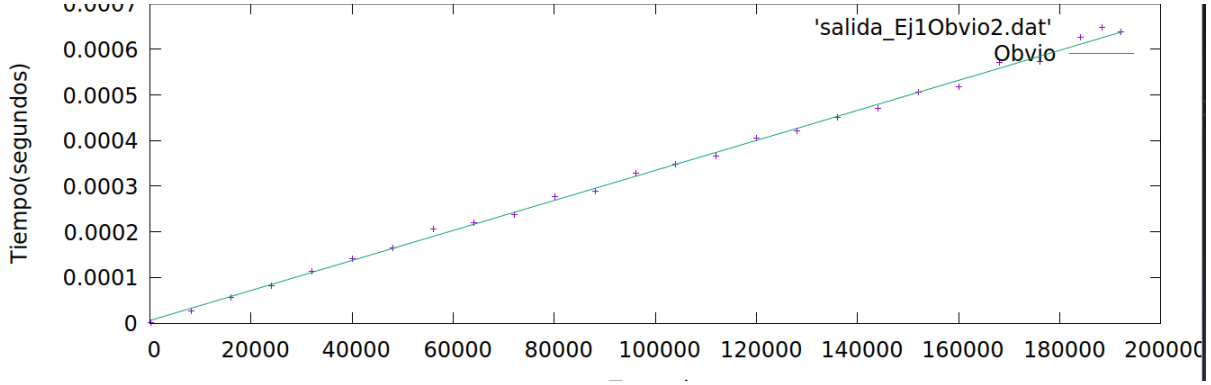
Para el análisis híbrido hemos utilizado el ordenador de Quintin cuyas características aparecen al principio.
Para el análisis empírico hemos utilizado el ordenador de Quintín cuyas prestaciones aparecen listadas en el punto 2.
Para distintos valores de n (especificados en la parte de software utilizado), se ha ido ejecutando el algoritmo mediante un script de bash, y se ha obtenido un tiempo promedio para cada uno de esos tamaños, los cuales se han recopilado en la tabla que se muestra a continuación. Los tiempos se han obtenido de la siguiente forma, para que fueran suficientemente representativos: para cada n de los que hemos tomado, se ha probado el algoritmo 15 veces; se han generado 15 vectores distintos (15 para la versión en la que no se pueden repetir los elementos y 15 para la que sí admite repeticiones), se han medido los tiempos de ejecución del algoritmo para cada uno de dichos vectores y se ha calculado el tiempo promedio, que es el que aparece en la tabla que aparece aquí abajo.

FUERZA BRUTA	
TAMAÑO	TIEMPO DE EJECUCIÓN EN SEGUNDOS
100	5,33E-04
8096	2.77505e-05
16092	5.56081e-05
24088	8.21521e-05
32084	0.000113871
40080	0.000140849
48076	0.000165736
56072	0.000206813
64068	0.000220565
72064	0.000237331
80060	0.000276449
88056	0.000289482
96052	0.000329335
104048	0.000347544
112044	0.000366818
120040	0.000405891
128036	0.000421047
136032	0.000451158
144028	0.000470098
152024	0.000504965
160020	0.000518047
168016	0.00057008
176012	0.000572705
184008	0.000625479
192004	0.000638166

4.1.4 Análisis híbrido

Para el análisis híbrido hemos utilizado el ordenador de Quintin cuyas características aparecen al principio.
En este apartado se va a llevar a cabo un estudio de la eficiencia híbrida del algoritmo de fuerza bruta. Esto es, el ajuste de la nube de puntos obtenida (los datos de la tabla anterior), mediante una función de ajuste. De acuerdo con lo expuesto en el apartado de eficiencia teórica, dicha función ha de ser lineal, es decir, de la forma:

$$f(x) = ax + b$$



La función lineal que mejor ajusta a la nube de puntos es la siguiente:

$$f(x) = 5.83677 \cdot 10^{-6} + 3.28678 \cdot 10^{-9} \cdot x$$

4.2 Versión divide y vencerás I

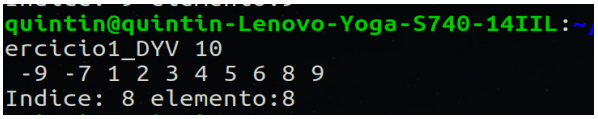
```
int buscaIgualIndiceDYV (int v[], int n) {  // Versión en la que no puede hab
                                            // elementos repetidos
    int izda = 0;
    int dcha = n-1;
    int half = 0;


    while (izda <= dcha){
      half = (izda+dcha)/2;
      if (v[half] > half) dcha = half-1;
      else if (v[half] < half) izda = half+1;
      else return half;
    }

    return -1;
}
```

4.1.1 Análisis teórico

Si observamos el código del algoritmo vemos que consta en primer lugar de tres sentencias que suponen tiempos constantes y, a continuación un ciclo while. En él, aparte de que consta de sentencias que suponen tiempos constantes (O(1)), en cada iteración, se va reduciendo el tamaño del vector con el que se trabaja, a la mitad, pues, al seleccionar la mitad del vector y hacer las comprobaciones pertinentes, se desprecia una de las mitades del mismo. De esta forma, dado que el tamaño del problema se va reduciendo a la mitad, y teniendo en cuenta que el resto de operaciones son O(1), concluimos que la eficiencia de este algoritmo es O(log n).

4.1.2 Ejemplo de ejecución

Para el análisis híbrido hemos utilizado el ordenador de Quintin cuyas características aparecen al principio.
A continuación, se ofrece un ejemplo de la ejecución de este algoritmo de divide y vencerás (con vectores sin elementos repetidos), para un valor de n=10. Téngase en cuenta que el vector sobre el que se va aplicar el algoritmo de divide y vencerás se ha creado mediante el generador 1, aportado por la profesora.



4.1.3 Análisis empírico

Para el análisis híbrido hemos utilizado el ordenador de Quintin cuyas características aparecen al principio.
Para el análisis empírico hemos utilizado el ordenador de Quintín. Para distintos los distintos valores de n especificados anteriormente, se ha ido ejecutando el algoritmo mediante el script de bash, y se ha obtenido un tiempo promedio para cada uno de esos tamaños, los cuales se han recopilado en la tabla que se muestra a continuación(los pares (tamaño, tiempo en segundos) son los puntos que aparecerán representados en las gráficas de más abajo). Los tiempos se han obtenido de la misma forma que en el apartado anterior.

DIVIDE Y VENCERÁS (SIN REPETICIONES)	
TAMAÑO	TIEMPO DE EJECUCIÓN EN SEGUNDOS
100	3.69467e-07
8096	1.55907e-06
16092	1.9632e-06
24088	5.34987e-06
32084	8.1754e-06
40080	9.29387e-06
48076	1.00005e-05
56072	1.21519e-05
64068	1.27416e-05
72064	1.82634e-05
80060	1.6684e-05
88056	1.86288e-05
96052	2.14082e-05
104048	2.30803e-05
112044	2.38357e-05
120040	2.73881e-05
128036	2.75237e-05
136032	3.06091e-05
144028	3.39657e-05
152024	3.82245e-05
160020	3.88711e-05
168016	3.99393e-05
176012	3.81673e-05
184008	4.51063e-05
192004	4.44051e-05

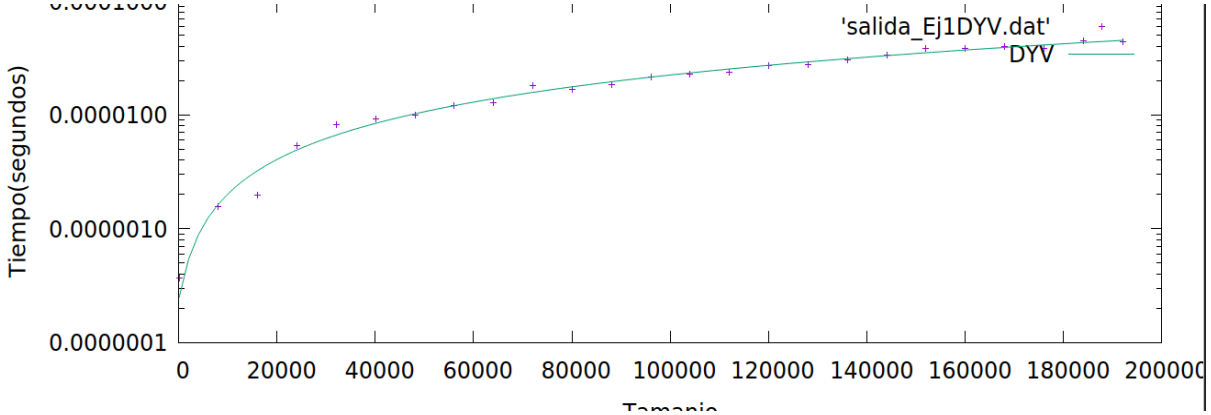
4.1.4 Análisis híbrido

En este apartado se va a llevar a cabo un estudio de la eficiencia híbrida del algoritmo divide y vencerás (caso en que los elementos del vector son distintos dos a dos). De acuerdo con lo expuesto en el apartado de eficiencia teórica, la función que mejor ajusta la nube de puntos obtenida tras las ejecuciones del algoritmo para los distintos valores de n considerados, ha de ser de tipo logarítmica, es decir, de la forma:

$$f(x) = a \cdot x \cdot log(x) + b$$

A continuación de ofrece la representación gráfica que corrobora lo obtenido en la eficiencia teórica:

(Representación en escala logarítmica)

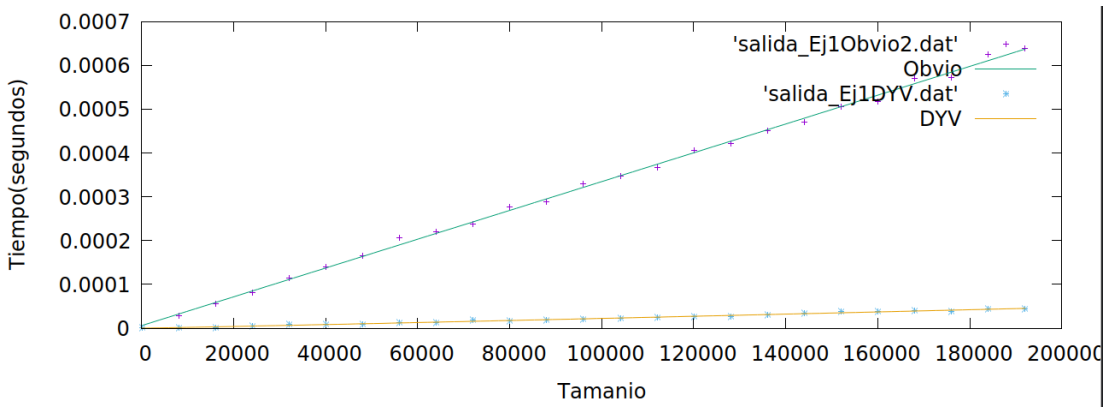


La función logarítmica que mejor ajusta a la nube de puntos es la siguiente:

$$f(x) = 4.4451 \cdot 10^{-11} \cdot x \cdot log(x) + 2.37879 \cdot 10^{-7}$$

4.3 Comparativa entre los algoritmos

En este apartado se va a llevar a cabo una comparativa entre los dos algoritmos para discernir cuál de ellos es más eficiente a la hora de resolver el problema planteado.
La forma más directa de comprobarlo es mediante la representación conjunta de las nubes de puntos:



Como podemos observar en la gráfica comparativa, la velocidad de ejecución del algoritmo divide y vencerás es más de un 90% mayor que el algoritmo de fuerza bruta; los tiempos de ejecución del DYV son prácticamente despreciables en comparación con los del obvio. Luego, podemos concluir que, en el caso de tener un vector de enteros ordenado de forma creciente y con elementos distintos dos a dos, el algoritmo más eficiente a la hora de hallar el i tal que v[i] == i, es el divide y vencerás.
Llegados a este punto, es lógico preguntarse si, en el caso de que ahora los elementos del vector no tuvieran que ser necesariamente distintos dos a dos, el algoritmo que hemos propuesto, como divide y vencerás sigue siendo válido.

La respuesta es **no**, por la siguiente razón:
En el algoritmo divide y vencerás anteriormente expuesto, lo que hacemos es:

1. Calculamos la mitad (m) del vector(v) y el elemento que ocupa dicha posición.
2. Si v[m] > m , dado que el vector está ordenado y los elementos del mismo no pueden repetirse, $\nexists\, i \in \{m+1,m+2,...,n-1\}$ tal que v[i] == i. La única forma de que sí existieran es que haya elementos repetidos, y no es el caso. Así que podemos tajantemente despreciar la mitad derecha del vector.
3. Análogamente se procede cuando v[m] < m.

Luego, si el vector tuviera elementos repetidos, los pasos 2 y 3, no se podrían llevar a cabo; no se podrían descartar mitades, porque pudiera ser que el elemento que buscamos se encuentre en una mitad.

Es por esto que el algoritmo divide y vencerás expuesto anteriormente no nos sirve para el problema que ahora intentamos resolver. Por lo tanto, hay que idear otro algoritmo divide y vencerás que sea capaz de encontrar tal i, en un vector cuyos elementos pueden repetirse.

4.2 Versión divide y vencerás II (los elementos pueden repetirse)

```
int buscaIgualIndiceDYV_REP (int v[], int l, int r)
{
  if (l <= r){

     if (l - r == 0){
         if (v[l] == l) return l;
         else return -1;
     }
     else if (v[l] == l) return l;
     else if (v[r] == r) return r;
     else{
          int half = (l+r)/2;
          if (v[half] == half) return half;
          else{
            l++;
            r--;
            int i= buscaIgualIndiceDYV_REP(v,l,half-1);
            if(i < 0){
                return buscaIgualIndiceDYV_REP(v,half+1,r);
            }else
              return i;
            }
     }
  }

  else return -1;
}
```

4.2.1 Análisis teórico

Esta versión alternativa al algoritmo divide y vencerás anterior, enfocada al caso en que el vector con el que trabajamos pueda contener elementos repetidos, se basa en una serie de estructuras condicionales que de cumplirse devuelven un elemento y de no cumplirse, otro (en el contexto del problema, se devuelve el elemento que verifica que v[i] == i, o -1 en caso de no encontrarse). Todas estas estructuras condicionales en las que se llevan a cabo operaciones simples, suponen tiempos constantes; O(1), salvo el último else, en el que se encuentra la parte recursiva del algoritmo, en la cual se va partiendo el vector en subvectores de igual dimensión y se vuelve a aplicar el algoritmo sobre esos subvectores, y así sucesivamente, hasta encontrar un elemento del vector que verifique la condición del problema, o no, en cuyo caso, se devuelve -1. Luego, claramente, esta parte supone un tiempo logarítmico. Por lo tanto, la eficiencia del algoritmo es O(log(n)).

4.2.2 Análisis empírico

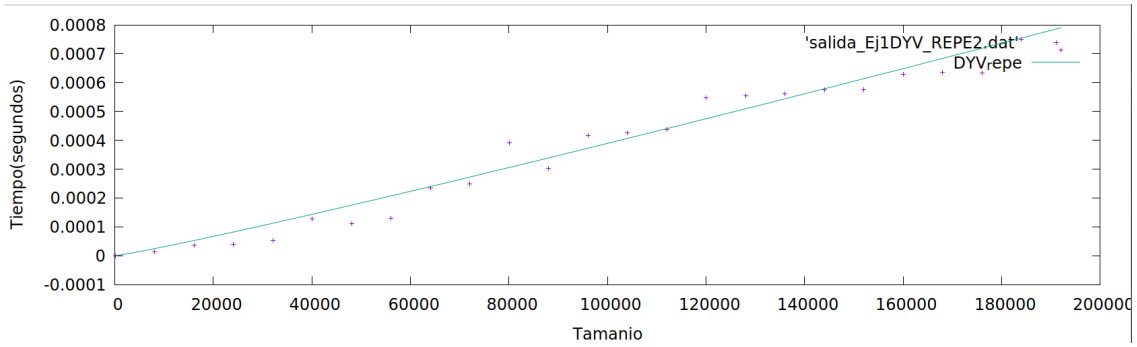
Para el análisis empírico hemos utilizado el ordenador de Quintín. Para distintos los distintos valores de n especificados anteriormente, se ha ido ejecutando el algoritmo mediante el script de bash, y se ha obtenido un tiempo promedio para cada uno de esos tamaños, los cuales se han recopilado en la tabla que se muestra a continuación Los tiempos se han obtenido de la misma forma que en el apartado anterior:

DIVIDE Y VENCERÁS CON REPETICIONES)	
TAMAÑO	TIEMPO DE EJECUCIÓN EN SEGUNDOS
100	4.14067e-07
8096	1.23172e-05
16092	3.706977e-05
24088	3.912655e-05
32084	5.3868e-05
40080	0.000128044
48076	0.000110946
56072	0.00012946
64068	0.000235981
72064	0.000248381
80060	0.000392104
88056	0.00030201
96052	0.000415648
104048	0.00042642
112044	0.000438016
120040	0.000546999
128036	0.000555258
136032	0.00056006
144028	0.000574978
152024	0.000574993
160020	0.000629421
168016	0.000634981
176012	0.00063243
184008	0.000750756
192004	0.00071303

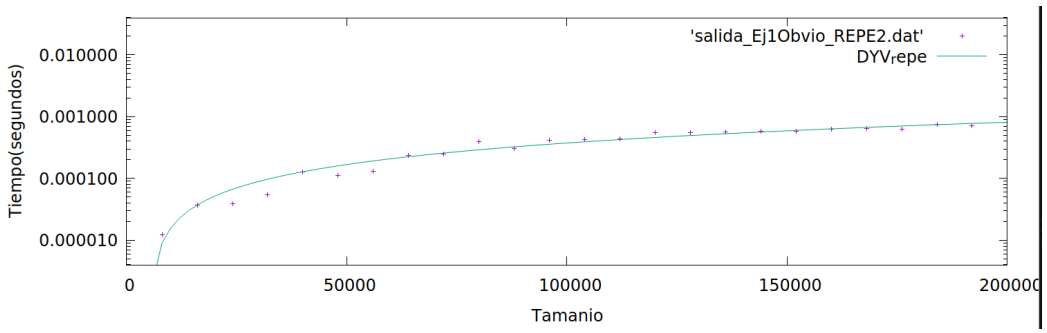
4.2.3 Análisis híbrido

De acuerdo con lo expuesto en el apartado de eficiencia teórica, la función que mejor ajusta la nube de puntos obtenida tras las ejecuciones del algoritmo para los distintos valores de n considerados ha de ser de tipo logarítmica, es decir, de la forma:

$$f(x) = a \cdot x \cdot log(x) + b$$



Escala logarítmica:

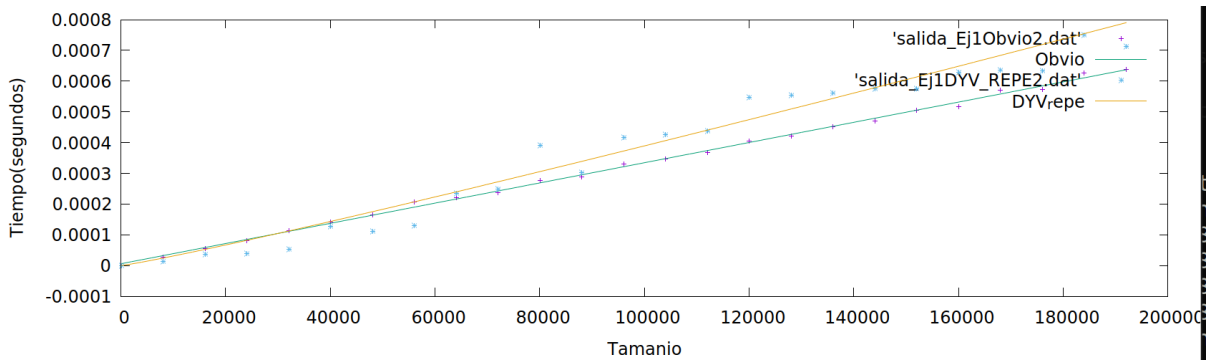


La función logarítmica que mejor ajusta a la nube de puntos es la siguiente:

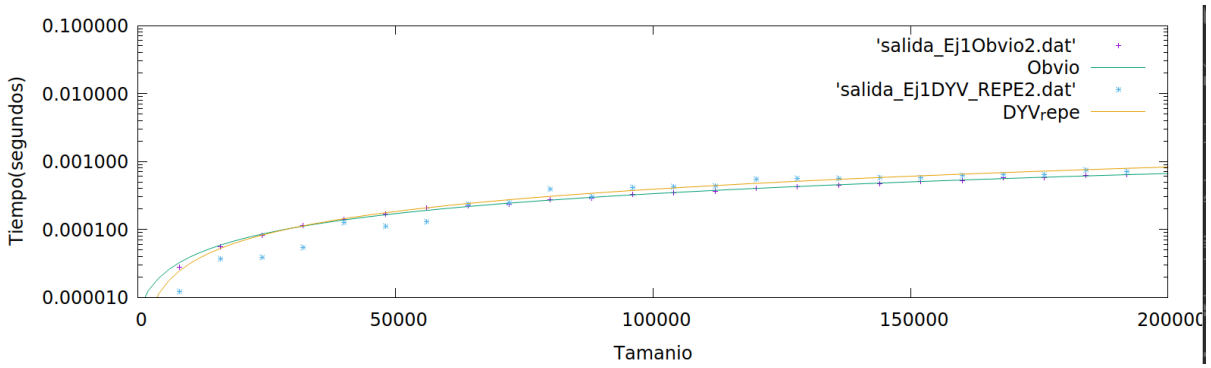
$$f(x) = 7.78941 \cdot 10^{-10} \cdot x \cdot log(x) - 1.56589 \cdot 10^{-7}$$

4.2.4 Comparativa con el algoritmo de fuerza bruta

Para dar respuesta a la pregunta que nos planteamos anteriormente acerca de si seguía siendo preferible en cuanto a eficiencia el algoritmo divide y vencerás frente al algoritmo de fuerza bruta, en caso de que se pudieran repetir los enteros del vector, ofrecemos a continuación una gráfica en la que aparecerán las dos nubes de puntos correspondientes a los dos algoritmos, con sus funciones de ajuste representadas, y a partir de las cuales podremos sacar conclusiones y responder a la pregunta:



Escala logarítmica:



Si observamos claramente las gráficas, apreciamos una diferencia en cuanto a velocidad de ejecución, en favor de la versión de fuerza bruta del algoritmo; se obtienen menores tiempos de ejecución en esta versión frente a la divide y vencerás, aunque la diferencia es mínima.

Por lo tanto, podemos concluir que si los elementos del vector se repiten es preferible y más eficiente usar el algoritmo de fuerza bruta.

4.3 Conclusiones del Ejercicio 1

- Para el caso en que tengamos un vector ordenado de forma creciente de números enteros, todos distintos, el algoritmo más eficiente a la hora de resolver el problema que plantea el ejercicio es el **divide y vencerás**, hecho que se corrobora en las gráficas y tablas de datos vistas anteriormente, donde la diferencia de tiempos entre el fuerza bruta y el dyv es de más de un 90%.
- Por otra parte, en el caso de que tengamos un **vector** ordenado de forma creciente de números **enteros** los cuales no tienen por qué ser todos distintos, es decir, se pueden **repetir**, el algoritmo **divide y vencerás** usado para vectores de enteros, todos distintos, es **inválido** para la resolución del problema.
- Por último, el **algoritmo alternativo** que hemos planteado para la resolución del problema del vector con **elementos repetidos**, es menos eficiente que el de fuerza bruta; su velocidad de ejecución es menor; sus tiempos son mayores, aunque no con mucha diferencia respecto al algoritmo de fuerza bruta. En cualquier caso, para este problema, por todo lo especificado y justificado en los apartados anteriores, concluimos que es **preferible** utilizar el algoritmo de **fuerza bruta** frente al divide y vencerás.

5. EJERCICIO 2

Se tienen k vectores ordenados (de menor a mayor), cada uno con n elementos, y queremos combinarlos en un único vector ordenado (con kn elementos).

5.1 Versión fuerza bruta

```
int * mezcla_FB(int numVectores, int numElementos, int ** m){
  int * fusion = nullptr;
  fusion = new int [numVectores * numElementos];

  bool found;
  int k;
  for(int i = 0; i < numElementos; i++){
    fusion[i] = m[0][i];
  }
  for(int i = 1; i < numVectores; i++){ // Iteracion por vector
    for(int j = 0; j < numElementos; j++){ // Iteracion por cada elemento del nuevo
      found = false;
      k = 0;
      while(!found && k < numElementos * i + j){
        if(fusion[k] > m[i][j])
          found = true;
        else
          k++;
      }
      if(found){
        for(int p = numElementos*i+j-1; p >= k; p--){ // Traslacion (uno a la derec
          fusion[p+1] = fusion[p];
        }
        fusion[k] = m[i][j];
      }
      if(!found){
        fusion[numElementos * i + j ] = m[i][j];
      }
    }
  }

  return fusion;
}
```

El método que hemos implementado para resolverlo por fuerza bruta consiste en crear un vector de tamaño numVectores * numElementos e introducir en él los elementos que se encuentran en la primera fila de la matriz que le hemos pasado como argumento. Luego con el uso de 2 bucles for hemos ido insertando los demás elementos de la matriz. A la vez que íbamos insertando los elementos que se encontraban en la matriz en el vector los íbamos comparando con los elementos que ya tenemos almacenados en el vector ordenado y si el elemento que tenemos que insertar se encuentra entre otros valores ya almacenados en el vector lo que hacemos es introducirlo en la posición correcta y desplazamos a la derecha el restos de los elementos a partir de dicha posición.

5.1.1 Análisis teórico

En primer lugar, hagamos una aclaración acerca de la notación que vamos a seguir:

- n ≡ dimensión del vector; número de elementos.
- m ≡ número de vectores ordenados.

  Una vez aclarada la cuestión de notación, analicemos la eficiencia teórica del algoritmo de fuerza bruta.

  Si observamos el código, en las líneas 6,7 y 8 se ejecuta un primer bucle en el cúal se inserta el primer vector ordenado en el vector fusión. Dicho bucle se ejecuta en un tiempo cn, con lo cual, la eficiencia es O(n).

  Si avanzamos en el código, nos encontramos con el bloque de código que inserta el resto de los vectores ordenados en el vector fusión cuya implementación tiene dos bucles for y dentro del segundo bucle for nos encontramos con un bucle while y otro bucle for, al mismo nivel. Dado que estos últimos bucles están al mismo nivel, hemos de analizar la eficiencia de los mismos en el peor de los casos:

- Peor caso para el bucle for:
  - El primer bucle for del anidamiento se ejecuta m-1 veces, el segundo, n veces y el bucle for del que estamos hablando t_1*(n*i+j-1) donde t_1 es el tiempo de ejecución de la línea del bucle en la que se aplica el método fusión. Luego, claramente, la eficiencia del algoritmo en el caso de que el bucle que sale perjudicado es el for y en el peor caso, es O(n³). En concreto, esto se corrobora si lo analizamos matemáticamente. Nos queda entonces:

    $$\sum_{i=1}^{m}\sum_{j=0}^{n}\sum_{k=0}^{n*i+j-1} t_1 = t_1 n\frac{m(m+1)}{2} + t_1 m(\frac{(n+1)(n+2)}{2} - 1)$$

    Con lo cual, efectivamente, la eficiencia es O(nm²+mn²).

- Peor caso para el bucle while:
  - El primer bucle for del anidamiento se ejecuta m-1 veces, el segundo, n veces y el bucle for del que estamos hablando t_2*(n*i+j+1) donde t_2 es el tiempo de ejecución del cuerpo del bucle en el cual se llama al método fusión. Luego, claramente, la eficiencia del algoritmo en el caso de que el bucle que sale perjudicado es el while, es, en el peor caso, O(n³). Más formalmente, quedaría de la siguiente forma:

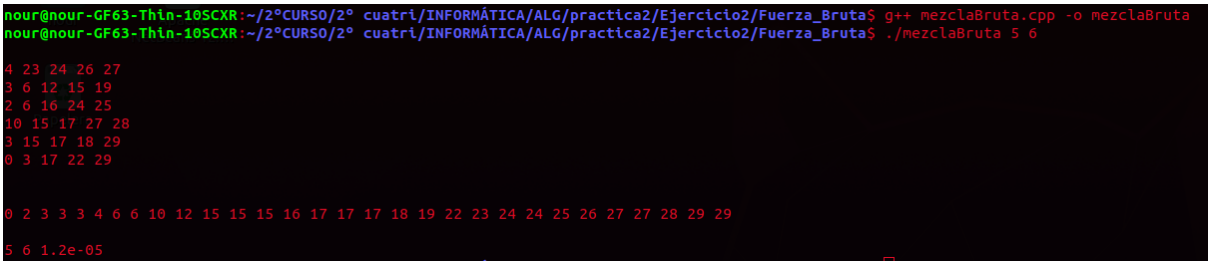
    $$\sum_{i=1}^{m}\sum_{j=0}^{n}\sum_{k=0}^{n*i+j} t_2 = t_2 n\frac{m(m+1)}{2} + t_2 m(\frac{(n+1)(n+2)}{2})$$

    Luego, la eficiencia es de O(nm²+mn²), también.

Por lo tanto, por la regla del producto y de la suma, concluimos que la eficiencia del algoritmo de fuerza bruta es O(nm²+mn²).

5.1.2 Ejemplo de ejecución

En este apartado vamos a mostrar lo que nos aparece al ejecutar el algoritmo.



Como podemos observar, nos aparece al principio una matriz que tiene almacenado los K vectores ordenados, y a continuación nos aparece el vector fusión que mezcla todos los vectores ordenados y los ordena.
En la última línea nos aparece el número de elementos, el número de vectores y el tiempo que ha tardado en mezclar los vectores.

5.1.3 Análisis empírico

Para el análisis empírico hemos utilizado el ordenador de Noura cuyas características aparecen al principio.

Tanto para la mezcla con fuerza bruta como para la mezcla con divide y vencerás, haremos el estudio de la eficiencia empírica fijando una de las dos variables que le pasamos al programa como argumento. Fijamos el número de elementos con N=8, y el número de vectores con N=5.

En esta tabla, tenemos el resultado tras ejecutar el algoritmo de mezcla con fuerza bruta fijando el número de vectores en el intervalo [100,10000], con saltos de 396 unidades por ejecución.

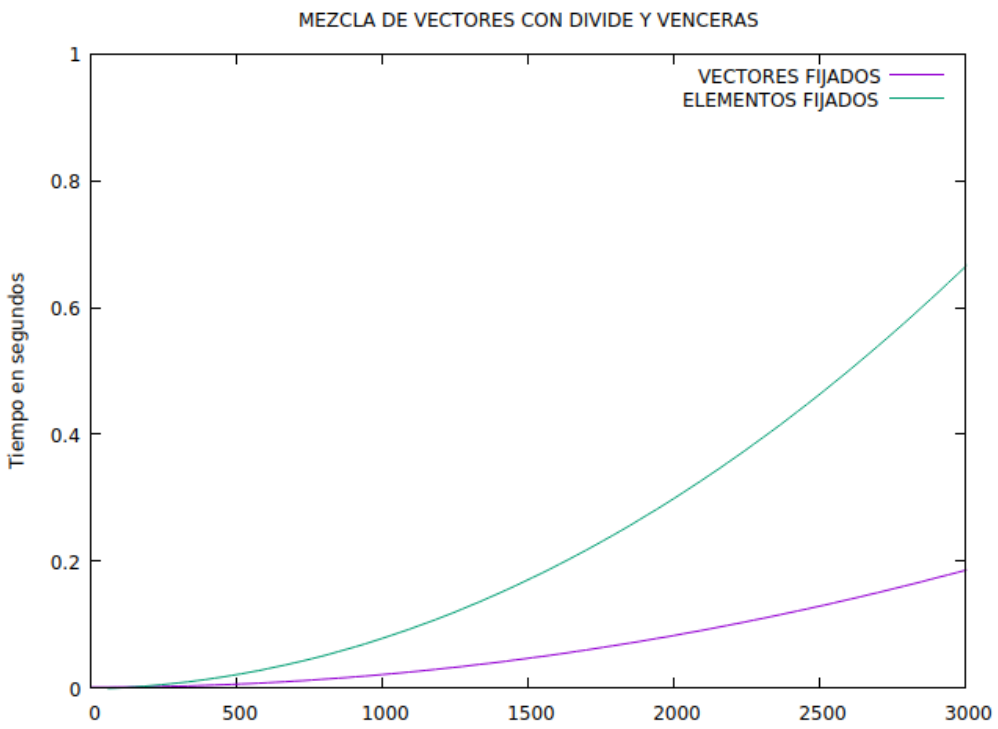
NUMERO DE ELEMENTOS (VECTORES N=5)	TIEMPOS DE EJECUCIÓN EN SEGUNDOS
100	0.001172
496	0.006097
892	0.015509
1288	0.032932
1684	0.053301
2080	0.080623
2476	0.113845
2872	0.152556
3268	0.197894
3664	0.249331
4060	0.30896
4456	0.367212
4852	0.438958
5248	0.518436
5644	0.609944
6040	0.674232
6436	0.762268
6832	0.868134
7228	0.964117
7624	1.07277
8020	1.19122
8416	1.30782
8812	1.43861
9208	1.62673
9604	1.69641

En la siguiente tabla, tenemos el resultado tras ejecutar el algoritmo de mezcla con fuerza bruta fijando esta vez el número de elementos en el intervalo [100,10000], con saltos de 396 unidades por ejecución.

NUMERO DE VECTORES (ELEMENTOS N=8)	TIEMPOS DE EJECUCIÓN EN SEGUNDOS
100	0.00299
496	0.016775
892	0.040056
1288	0.08454
1684	0.151455
2080	0.206555
2476	0.292292
2872	0.394914
3268	0.510728
3664	0.641054
4060	0.785313
4456	0.94878
4852	1.14668
5248	1.3385
5644	1.53266
6040	1.76466
6436	2.00264
6832	2.26571
7228	2.53527
7624	2.82307
8020	3.12794
8416	3.45626
8812	3.84793
9208	4.12697
9604	4.47861

Como podemos observar en las tablas anteriores, al variar el número de vectores tarda más tiempo en ejecutarse el algoritmo.

A continuación mostraremos una gráfica comparativa de los tiempos de ejecución del algoritmo de mezcla fijando el número de elementos y el número de vectores:



Como bien podemos observar en la gráfica, el algoritmo tarda menos al fijar el número de vectores, es decir, tarda menos tiempo en ordenar 5 vectores de N elementos que si tratamos de ordenar de N vectores de 8 elementos.

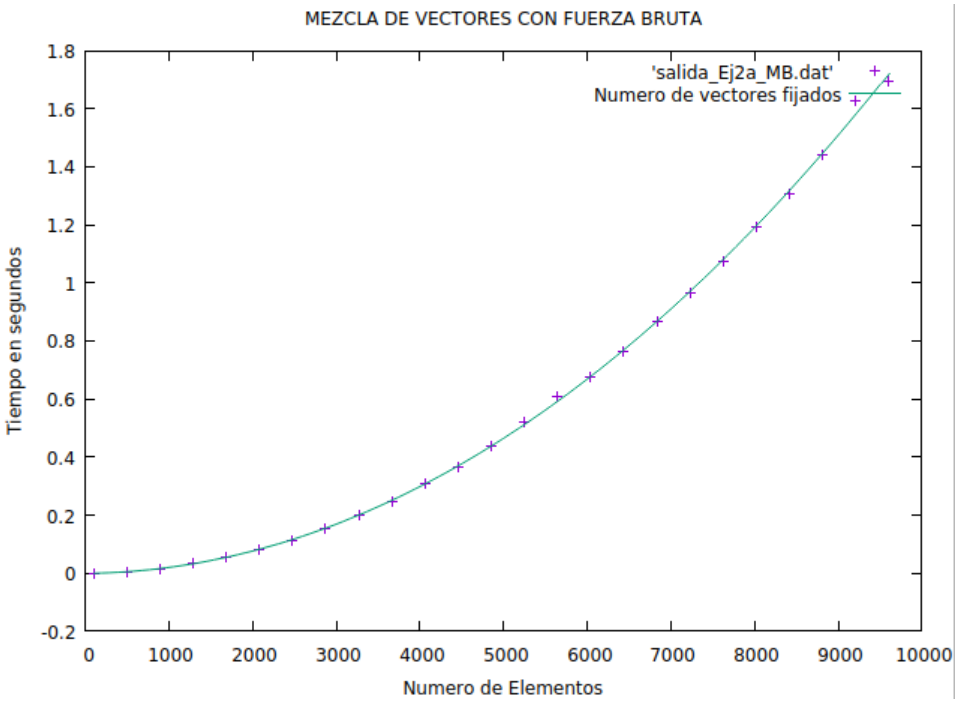
5.1.4 Análisis híbrido

Para el análisis híbrido hemos utilizado el ordenador de Noura cuyas características aparecen al principio.

En este apartado se va a llevar a cabo el ajuste de las nubes de puntos obtenidos, mediante una función de ajuste. Dicha función, teniendo en cuenta lo expuesto en el apartado de eficiencia teórica, ha de ser de tipo cúbica. Esto es, de la forma:

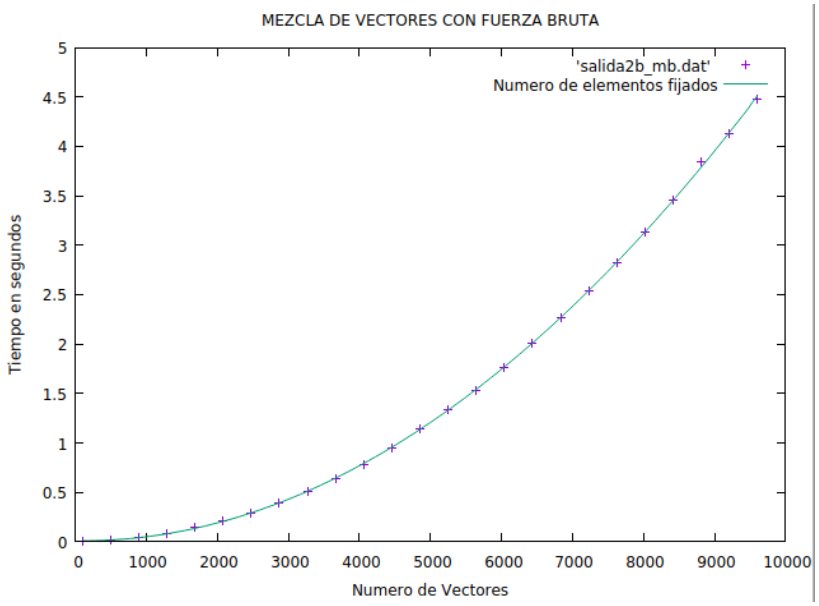
$$f(x) = ax^3 + bx^2 + cx + d$$

A continuación se muestran dos gráficas: una en la que se representan los tiempos asociados al caso en que se fija el número de vectores, junto con su función de ajuste, y por otro lado, la gráfica en la que se representan los tiempos asociados al caso en el que se fija el número de elementos, junto con la función que mejor ajusta a la nube de puntos.



La función cuadrática que mejor ajusta a la nube de puntos es:

$$f(x) = 1.76692 \cdot 10^{-8} \cdot x^3 - 2.94532 \cdot 10^{-6} \cdot x^2 - 0.000842 \cdot x + 7.06951 \cdot 10^{-14}$$



La función cuadrática que mejor ajusta a la nube de puntos es:

$$f(x) = -8.44243 \cdot 10^{-15} x^3 + 4.9798 \cdot 10^{-8} x^2 - 9.50898 \cdot 10^{-8} x - 0.0105839$$

5.2 Versión divide y vencerás

Para la resolución del ejercicio con el algoritmo de divide y vencerás hemos definido 2 métodos. El primer método nos mezcla 2 vectores y los almacena en un vector resultado.

El segundo método se encarga de hacer los cálculos recursivos.

El método mezcla_DYV_recursivo lo que que hace es dividir la matriz dónde están almacenados los vectores en 2 vectores, y mediante recursividad estos 2 vectores a su vez se dividen en 2 vectores y así sucesivamente, hasta que llega a vectores de una sola componentes y vuelve hacia atrás mezclando cada uno de los subvectores obtenidos hasta llegar a mezclar los K vectores.

```
void mezclaVectores(int * vec1, int * vec2, int dim1, int dim2, int * &mezcla){

  for(int i=0; i<dim1; i++)
    mezcla[i] = vec1[i];

  bool found;
  int k;

  for(int j=0; j<dim2; j++){
    k=0; found = false;
    while(!found && k<(dim1+dim2)){
      if(mezcla[k] >= vec2[j])
        found = true;
      else
        k++;
    }

    if(found){
      for(int p = dim1+j-1; p >= k; p--){
        mezcla[p+1] = mezcla[p];
      }
      mezcla[k] = vec2[j];
    }
    if(!found){
      mezcla[dim1+j] = vec2[j];
    }
  }
}
```

5.2.1 Análisis teórico

Empezaremos analizando la eficiencia del primer método que hemos implementado, que, como se ha especificado, lleva a cabo la mezcla ordenada de dos vectores ordenados:

Si nos fijamos bien en el algoritmo, es fácil ver que se trata de un caso particular del algoritmo de mezcla fuerza bruta, para un número de vectores igual a 2. Con lo cual, la eficiencia de este algoritmo no es otra que $O(2n^2+4n) = O(n^2)$.

Analicemos ahora la eficiencia del algoritmo divide y vencerás recursivo:

```
void mezcla_DYV_recursivo(int vec_inicial, int vec_final, int numElementos,
 int ** m, int * &mezcla){


 if(vec_inicial < vec_final){

  int numVectores = vec_final-vec_inicial+1;
  int vec_medio = (vec_final+vec_inicial)/2;

  int dim2 = numVectores/2;
  int dim1 = numVectores - dim2;

  int * mezcla1 = nullptr; int * mezcla2 = nullptr;
  mezcla1 = new int [numElementos*(dim1)];
  mezcla2 = new int [numElementos*(dim2)];


  mezcla_DYV_recursivo(vec_inicial,vec_medio,numElementos,m,mezcla1);

  mezcla_DYV_recursivo(vec_medio+1,vec_final,numElementos,m,mezcla2);

  mezclaVectores(mezcla1,mezcla2,dim1*numElementos,dim2*numElementos,mezcla);

  delete [] mezcla1;
  mezcla1 = nullptr;

  delete [] mezcla2;
  mezcla2 = nullptr;

 }
 //EN CASO CONTRARIO AL PASO1: CASO BASE (SOLO MEZCLAMOS 1 VECTOR)
 else{
  for(int i=0; i<numElementos;i++)
  mezcla[i] = m[vec_inicial][i];
  }
```

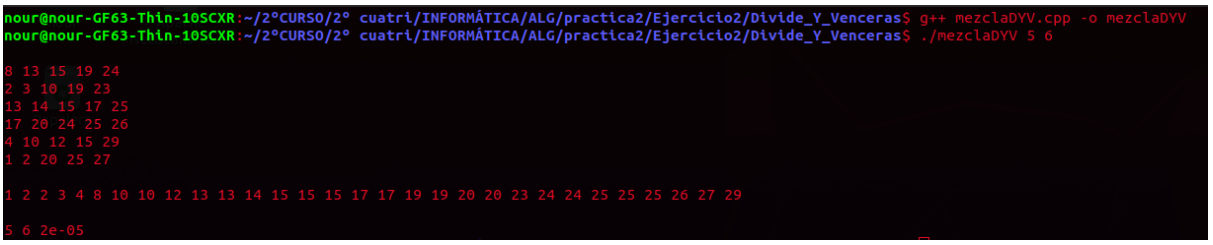
En este algoritmo, se parte de una matriz en la que se encuentran los vectores ordenados que se van a mezclar. La mezcla se hace de forma recursiva:
Se van dividiendo sucesivamente los vectores en dos grupos, hasta que la división ya no es posible, y de vuelta, se van ordenando todos los subvectores que se habían creado y guardado sucesivamente en la pila, hasta conseguir un único vector, mezcla de todos los anteriores, y ordenado.
Si analizamos el caso en que tenemos un único vector (m = 1), como el vector ya está ordenado, solo hay que insertar el vector inicial en el vector resultado. Luego, en este caso, el tiempo es lineal.
Por otra parte, si hay más de un vector, (m > 1), y unimos todos los tiempos constantes que conllevan las sentencias simples que hay en el algoritmo en una constante t, teniendo en cuenta que en la recurrencia se dividen los vectores en dos subconjuntos de m/2 vectores, habrá que sumar a la constante antes mencionada, dos veces el tiempo de ejecución para m/2 vectores de n componentes, más el tiempo de la mezcla, que, como antes habíamos analizado, es n^2. Es decir, la eficiencia del algoritmo, al fin y al cabo vendrá dada por una recurrencia del estilo:

$$X_m = \begin{cases} n^2 + X_{m/2} + t & si \quad m > 1 \\ n & si \quad m = 1 \end{cases}$$

Si resolvemos la recurrencia, nos queda X_{m} = mn². Por lo tanto, la eficiencia de la recurrencia no es otra que O(n²m)

5.2.2 Ejemplo de ejecución

En este apartado vamos a mostrar lo que nos aparece al ejecutar el algoritmo.



Al igual que el algoritmo con fuerza bruta, nos aparece al principio una matriz que tiene almacenado los K vectores ordenados, y a continuación nos aparece el vector fusión que mezcla todos los vectores ordenados y los ordena.

En la última línea nos aparece el número de elementos, el número de vectores y el tiempo que ha tardado en mezclar los vectores.

5.2.3 Análisis empírico

Para el análisis empírico hemos utilizado el ordenador de Noura cuyas características aparecen al principio.

Al igual que con el algoritmo de mezcla de fuerza bruta, haremos el estudio de la eficiencia empírica del algoritmo de divide y vencerás fijando una de las dos variables que le pasamos al programa como argumento. Fijamos el número de elementos con N=8, y el número de vectores con N=5.

En esta tabla, tenemos el resultado tras ejecutar el algoritmo de mezcla con divide y vencerás fijando el número de vectores en el intervalo [100,10000], con saltos de 396 unidades por ejecución.

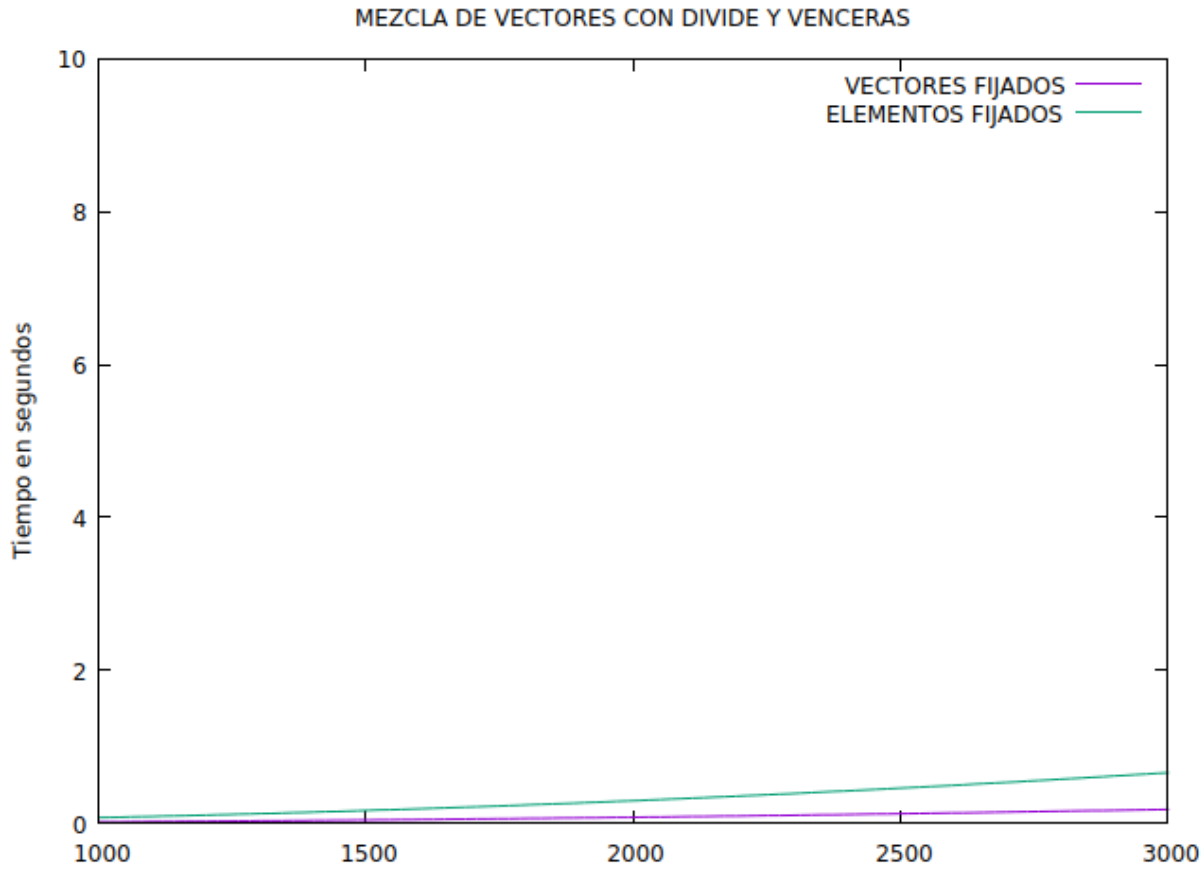
NUMERO DE ELEMENTOS (VECTORES N=5)	TIEMPOS DE EJECUCIÓN EN SEGUNDOS
100	0.001329
496	0.006933
892	0.017853
1288	0.035513
1684	0.060093
2080	0.091773
2476	0.125806
2872	0.169719
3268	0.220153
3664	0.275771
4060	0.339713
4456	0.406758
4852	0.485479
5248	0.574904
5644	0.657258
6040	0.750443
6436	0.853993
6832	0.962448
7228	1.07375
7624	1.18986
8020	1.31601
8416	1.45522
8812	1.59453
9208	1.73925
9604	1.88978

En la siguiente tabla, tenemos el resultado tras ejecutar esta vez el algoritmo de mezcla con divide y vencerás fijando el número de elementos en el intervalo [100,10000], con saltos de 396 unidades por ejecución.

NUMERO DE VECTORES (ELEMENTOS N=8)	TIEMPOS DE EJECUCIÓN EN SEGUNDOS
100	0.000883
496	0.019304
892	0.06202
1288	0.128148
1684	0.220234
2080	0.335605
2476	0.449421
2872	0.603817
3268	0.782334
3664	0.980963
4060	1.21511
4456	1.4598
4852	1.7286
5248	2.08436
5644	2.3484
6040	2.69669
6436	3.1304
6832	3.48079
7228	3.89226
7624	4.32137
8020	4.86858
8416	5.36424
8812	5.87993
9208	6.43104
9604	7.03543

Con el algoritmo de divide y vencerás también al variar el número de vectores tarda más tiempo en ejecutarse el algoritmo.

A continuación mostraremos una gráfica comparativa de los tiempos de ejecución del algoritmo de mezcla con divide y vencerás fijando el número de elementos y el número de vectores:



El algoritmo de divide y vencerás tarda un poco menos al fijar el número de vectores, es decir, tarda menos tiempo en ordenar 5 vectores de N elementos que si tratamos de ordenar de N vectores de 8 elementos.

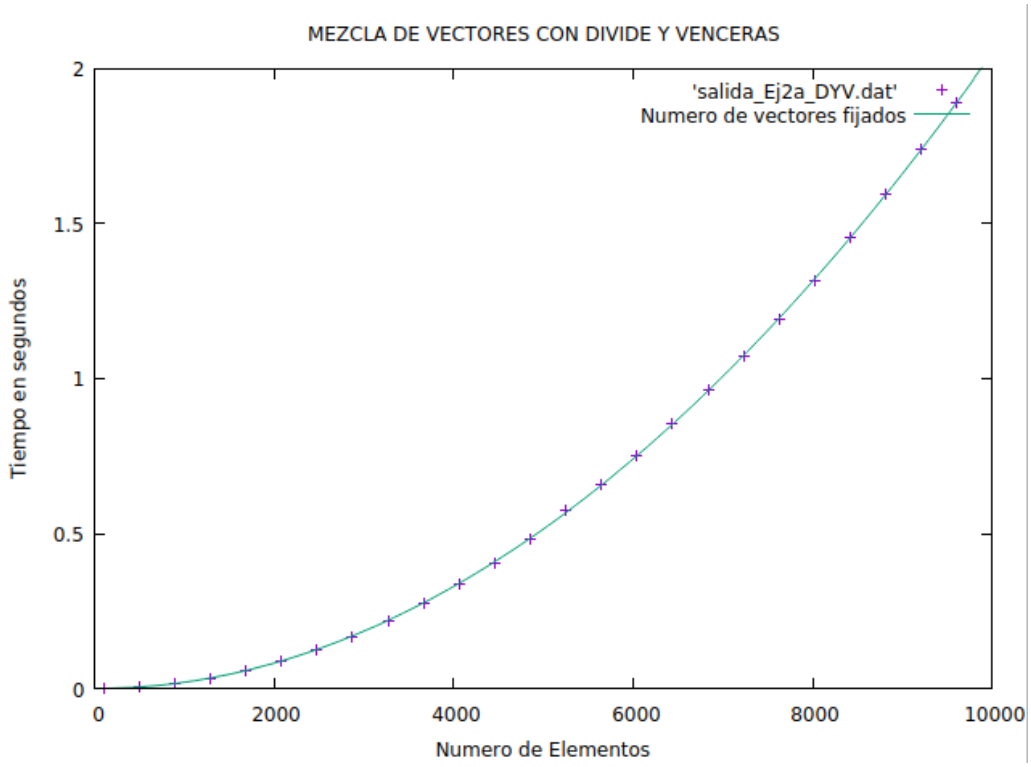
5.2.4 Análisis híbrido

Para el análisis híbrido hemos utilizado el ordenador de Noura cuyas características aparecen al principio.

En este apartado se va a llevar a cabo el ajuste de las nubes de puntos obtenidos, mediante una función de ajuste. Dicha función, teniendo en cuenta lo expuesto en el apartado de eficiencia teórica, ha de ser de tipo cúbica. Esto es, de la forma:

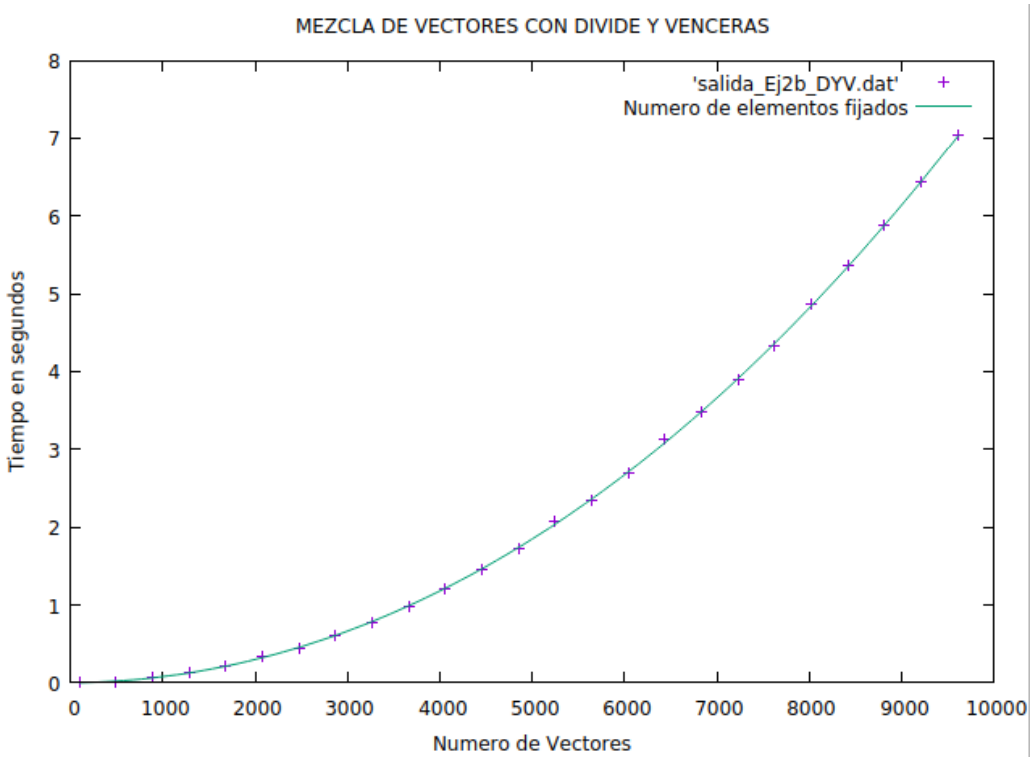
$$f(x) = ax^3 + bx^2 + cx + d$$

A continuación se muestran dos gráficas: una en la que se representan los tiempos asociados al caso en que se fija el número de vectores, junto con su función de ajuste, y por otro lado, la gráfica en la que se representan los tiempos asociados al caso en el que se fija el número de elementos, junto con la función que mejor ajusta a la nube de puntos.



La función cuadrática que mejor ajusta a la nube de puntos es:

$$f(x) = -3.61135 \cdot 10^{-14}x^3 + 2.091 \cdot 10^{-8}x^2 - 9.01735 \cdot 10^{-7}x + 0.00178979$$



La función cuadrática que mejor ajusta a la nube de puntos es:

$$f(x) = 7.11628 \cdot 10^{-13}x^3 + 6.82646 \cdot 10^{-8}x^2 + 1.07696 \cdot 10^{-5}x - 0.000955068$$

5.3 Comparativa entre los algoritmos

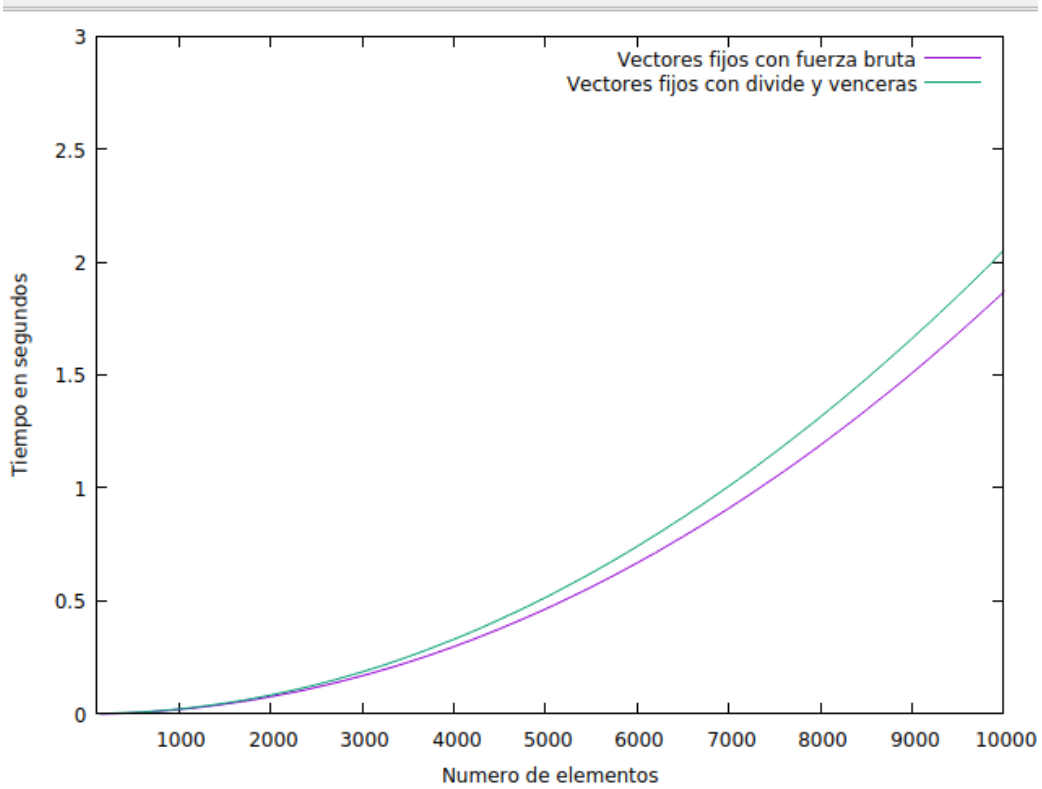
Para la comparación entre ambos algoritmos vamos a representar la tabla comparando los tiempos de ejecución obtenidos por fuerza bruta y por divide y vencerás.

5.3.1 Comparativa entre los algoritmos cuando tenemos N elementos

Para los vectores fijos, tenemos la siguiente tabla de los tiempos de ejecución de cuando tenemos 5 vectores de N elementos:

NUMERO DE ELEMENTOS (VECTORES N=5)	TIEMPOS DE EJECUCIÓN EN SEGUNDOS CON FB	TIEMPOS DE EJECUCIÓN EN SEGUNDOS CON DYV
100	0.001172	0.001329
496	0.006097	0.006933
892	0.015509	0.017853
1288	0.032932	0.035513
1684	0.053301	0.060093
2080	0.080623	0.091773
2476	0.113845	0.125806
2872	0.152556	0.169719
3268	0.197894	0.220153
3664	0.249331	0.275771
4060	0.30896	0.339713
4456	0.367212	0.406758
4852	0.438958	0.485479
5248	0.518436	0.574904
5644	0.609944	0.657258
6040	0.674232	0.750443
6436	0.762268	0.853993
6832	0.868134	0.962448
7228	0.964117	1.07375
7624	1.07277	1.18986
8020	1.19122	1.31601
8416	1.30782	1.45522
8812	1.43861	1.59453
9208	1.62673	1.73925
9604	1.69641	1.88978

Y representando gráficamente estos tiempos obtenemos la siguiente gráfica:



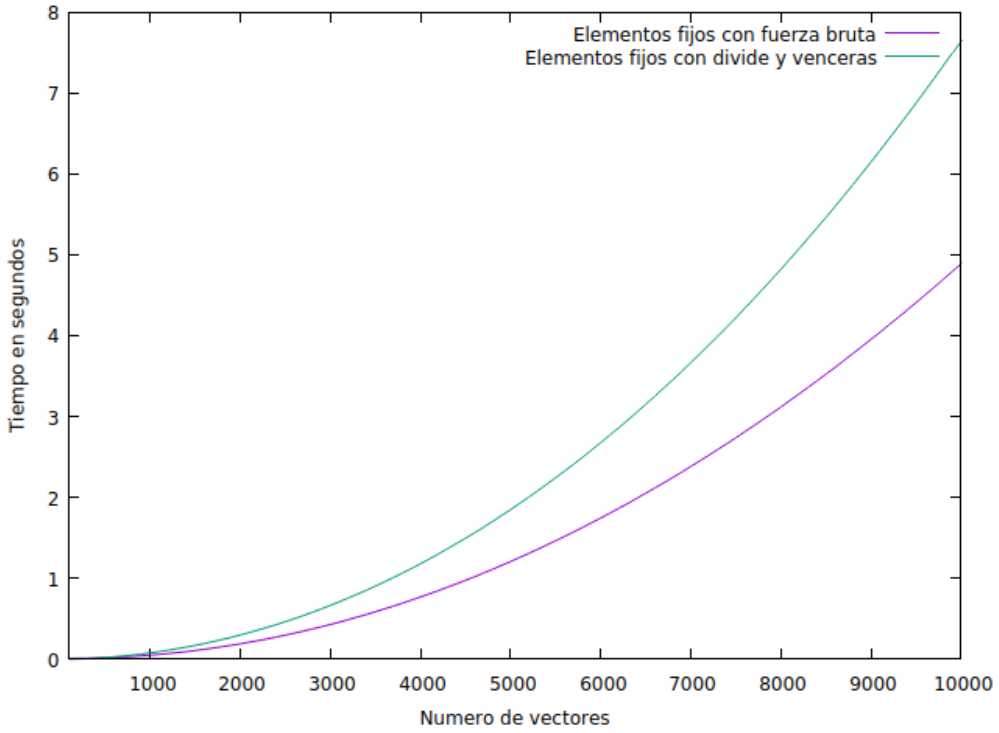
Como podemos observar tanto en la gráfica como en la tabla se obtienen mejores tiempos de ejecución por fuerza bruta que por divide y vencerás para una mezcla de 5 vectores con N elementos.

5.3.2 Comparativa entre los algoritmos cuando tenemos K vectores

Veamos ahora la tabla de los tiempos de ejecución que obtenemos en este caso cuando tenemos K vectores de 8 elementos:

NUMERO DE VECTORES (ELEMENTOS N=8)	TIEMPOS DE EJECUCIÓN EN SEGUNDOS CON FB	TIEMPOS DE EJECUCIÓN EN SEGUNDOS CON DyV
100	0.00299	0.000883
496	0.016775	0.019304
892	0.040056	0.06202
1288	0.08454	0.128148
1684	0.151455	0.220234
2080	0.206555	0.335605
2476	0.292292	0.449421
2872	0.394914	0.603817
3268	0.510728	0.782334
3664	0.641054	0.980963
4060	0.785313	1.21511
4456	0.94878	1.4598
4852	1.14668	1.7286
5248	1.3385	2.08436
5644	1.53266	2.3484
6040	1.76466	2.69669
6436	2.00264	3.1304
6832	2.26571	3.48079
7228	2.53527	3.89226
7624	2.82307	4.32137
8020	3.12794	4.86858
8416	3.45626	5.36424
8812	3.84793	5.87993
9208	4.12697	6.43104
9604	4.47861	7.03543

Y representando gráficamente estos tiempos obtenemos la siguiente gráfica:



Llegamos a las mismas conclusiones que con el número de vectores fijos,ya que en este caso también tanto en la gráfica como en la tabla se obtienen mejores tiempos de ejecución por fuerza bruta que por divide y vencerás para una mezcla de K vectores con 8 elementos.

5.4 Conclusiones del Ejercicio 2

Viendo los resultados de la comparación de los algoritmos en el apartado anterior, tenemos que el algoritmo implementado por fuerza bruta tarda menos que algoritmo implementado por divide y vencerás y esto se debe a que en el algoritmo de divide y vencerás usamos la recursividad que es una herramienta muy potente, pero en nuestro caso, que tenemos un ejercicio sencillo que no necesita recursividad renta más resolver el ejercicio por fuerza bruta ya que tardaría menos tiempo en ejecutarse.

Observando las gráficas comparativas anteriores, vemos también que cada vez la gráfica de divide y vencerás se va alejando cada vez más de la gráfica de fuerza bruta, esto quiere decir que en ningún momento el algoritmo de divide y vencerás será mejor que el de fuerza bruta para resolver este ejercicio. Esto nos lleva a deducir que usar la recursividad para ejercicios sencillos como este no es buena idea ya que no nos va reducir el tiempo de ejecución del algoritmo.

De las conclusiones anteriores, deducimos una importante conclusión y es hacer un análisis previo del problema ya que eso nos ayuda mucho a la hora de elegir el algoritmo más adecuado y eficiente para nuestro problema.