Solución de un problema mediante el algoritmo Divide y Vencerás

Universidad de Murcia
2º Grado en Ingeniería Informática
Asignatura: Algoritmos y estructuras de datos II
Curso 2017/2018

Proyecto realizado por:
Nicolás Enrique Linares La Barba
Miguel Ángel Lucas Barceló

Índice

No se encontraron entradas de tabla de contenido.

1. Problema

7) Dada una cadena C con n caracteres y un conjunto S de 5 caracteres, se trata de encontrar las subcadenas de C formadas por 3 elementos de S. Habrá que obtener el número de subcadenas y su posición en la cadena C. Por ejemplo, si

C = abbfabcddfcbbade , n=16

si consideramos un conjunto de cinco caracteres S={a,b,c,d,e}

la solución es 2, en las posiciones abbfabcddfcbbade

la última cadena, ade, no se tiene en cuenta, pues no se consideran repeticiones de caracteres en las cadenas, y de las cadenas que solapan elementos se considera únicamente la primera en la cadena C.

Se va a trabajar con una estructura enlazada, en este caso una lista, debido a que no conocemos cuántas subcadenas vamos a encontrar. Iremos guardando aquellas posiciones de las subcadenas válidas que vamos encontrando en cada mitad establecida por la función *divideyvenceras*, que se irán actualizando según el caso. Finalmente, la solución constará por todos los elementos de dicha lista.

2. Solución directa

En este caso se llama solución directa a aquella solución que propone un resultado sin emplear el algoritmo de Divide y Vencerás.

En primer lugar, se inicializa un contador a 0, se pone a false todos los elementos de letrasUsadas y entra en el bucle principal, desde el índice p hasta num.

En cada iteración se comprueba (con un bucle while) la letra actual con todas las del conjunto, si coincide en algún caso con alguna de las letras del conjunto, se usa encontrado para dejar de comprobar. Si se encuentra en el subconjunto, se compara si es una letra repetida o no, en el caso de no serlo se marca en letrasUsadas a true, se concatena con subcadena y se establece encontrado a true. Si es una letra repetida, mediante un bucle se busca en subcadena la última aparición de esta, para así cortar la cadena desde ella. Esto se hace porque hay varios casos que pueden darse, por ejemplo:

- subcadena: ab, letra actual: a, resultado: ba
- subcadena: ab, letra actual: b, resultado: b

Como puede apreciarse dependiendo de la posible subcadena y la letra entrante, se debe quitar una o dos letras para actualizar la variable subcadena.

A su vez, en letrasUsadas, se marca a false las letras que ahora no formarán parte de subcadena, incluida aquella que se repite para a continuación volverla a marcar como usada. También se tiene que ajustar el nuevo tamaño de contador por el número de letras que hay en subcadena y se marca a true encontrado para salir del bucle.

Se sale del bucle de comprobación de letras y se comprueba si hay un acierto (letra que coincide con el subconjunto), de ser así se suma uno al contador, sino se vuelve a inicializar las variables contador, subcadena y letrasUsadas. Si se alcanza al valor 3 en contador significa que hay una subcadena válida, por lo que se inserta su posición en la lista de soluciones, y se resetea al igual que en el caso de arriba, las variables contador, subcadena y letrasUsadas.

Finalmente se devuelve la lista de posiciones, contando cuantos aciertos hay.

3. Solución mediante Divide y Vencerás

En primer lugar, se calcula el número de elementos proporcionados, si es menor a 150, se utiliza la solucionDirecta y devuelve su resultado. Sino se calcula n y llama recursivamente a divideyvenceras de las dos mitades de la cadena (desde p a n, y desde n+1 hasta q). Finalmente, gracias a la función combinar, se puede devolver un resultado.

3.1. Función Combinar

Lo primero que hay que comprobar son las listas de soluciones recibidas por *s1* y *s2.* Si están vacías, solo hay que comprobar la frontera, es decir las dos últimas letras antes de hacer la división y las dos siguientes. Para ello se llama a *soluciónDirecta* en dicha franja.

En el caso de que no estén vacías, se deben encontrar los índices izquierdo y derecho, que pueden producir conflictos, es decir, zonas donde podrían encontrase subcadenas válidas entre las 2 partes de la cadena. Para ello no se puede, como en el caso anterior, coger simplemente las dos letras anteriores y siguientes, ya que podría romperse una subcadena válida.

La idea planteada consiste en buscar antes de la división, la última letra que no pertenece al conjunto (en la variable *izq*) y, después la primera que no pertenece en la parte derecha (varible *der*), para ello se utiliza una función auxiliar llamada *pertenece*. Una vez encontrados los índices, se borran en la primera lista de soluciones (*s1*) todas aquellas posiciones de subcadenas mayores que *izq* y, en la segunda lista (*s2*) todas aquellas que sean menores que *der.*

A continuación, hay que llamar a *solucionDirecta* con los valores *izq* y *der,* que proporcionan las subcadenas de la franja conflictiva y se guardan en *unir.*

El resultado de *combinar* es, al igual que las anteriores, una lista enlazada de enteros con las nuevas posiciones encontradas y aquellas nuevas posiciones que se han actualizado a consecuencia de haber encontrado una subcadena entre las dos partes, *s1* concatenado

con *unir* y concatenado también con *s2.*

3.2. Función Main

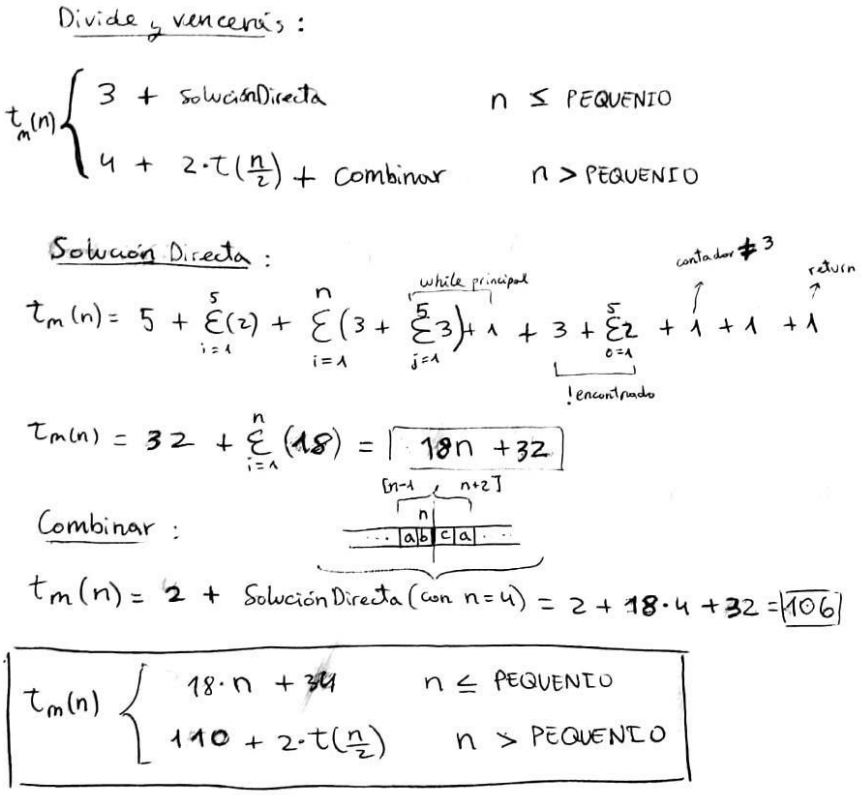
Un bucle permite resolver el número de cadenas que queramos, cuyo valor es leído junto a cadena, conjuntoS, y numElemento. Lo que hace a continuación es llamar a divideyvenceras, que se ocupará de dividir la cadena y resolverla recursivamente, y devuelve una lista que se guarda en resultado. Por último, un bucle recorre la lista imprimiendo las posiciones de las cadenas. Se aprovecha ese bucle para crear un contador (aciertos) que cuenta los elementos y muestra también el número de aciertos.

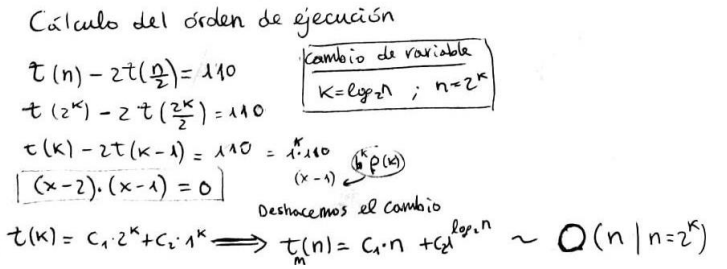
4. Estudio teórico del tiempo de ejecución

4.1. Mejor caso

El mejor caso consiste en las cadenas que no poseen elementos del conjunto. De esta manera nunca se encuentra una subcadena válida y por lo tanto la operación combinar es muy poco costosa. Esto se debe a que como combinar recibe dos listas vacías, únicamente hay que realizar *solucionDirecta* de la frontera dónde dividimos la cadena.

Por otro lado, *solucionDirecta*, tiene una eficiencia similar al de peor caso o el promedio. Como no hay ninguna letra del conjunto, cada carácter de la cadena es comparado con los 5 elementos, pero no hace nada más en el bucle, y al final de cada comprobación inicializa las variables *contador*, *subcadena* y *letrasUsadas*.



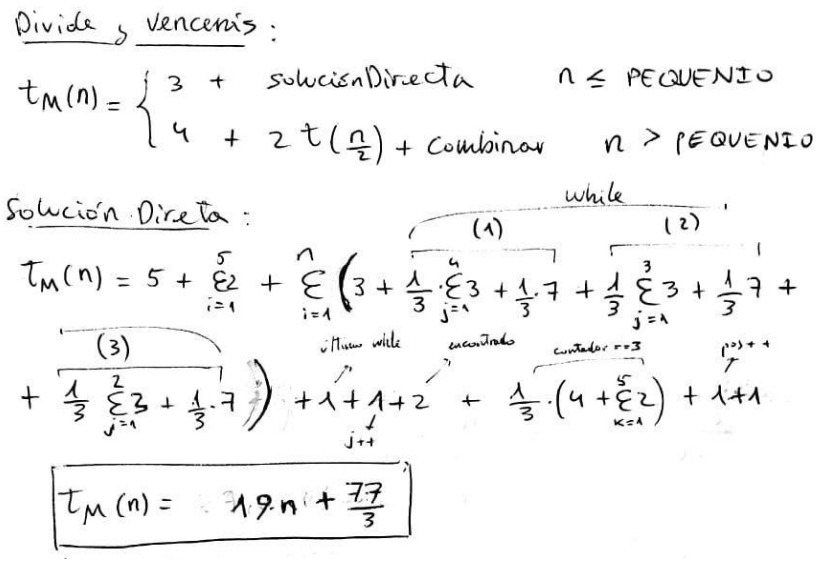


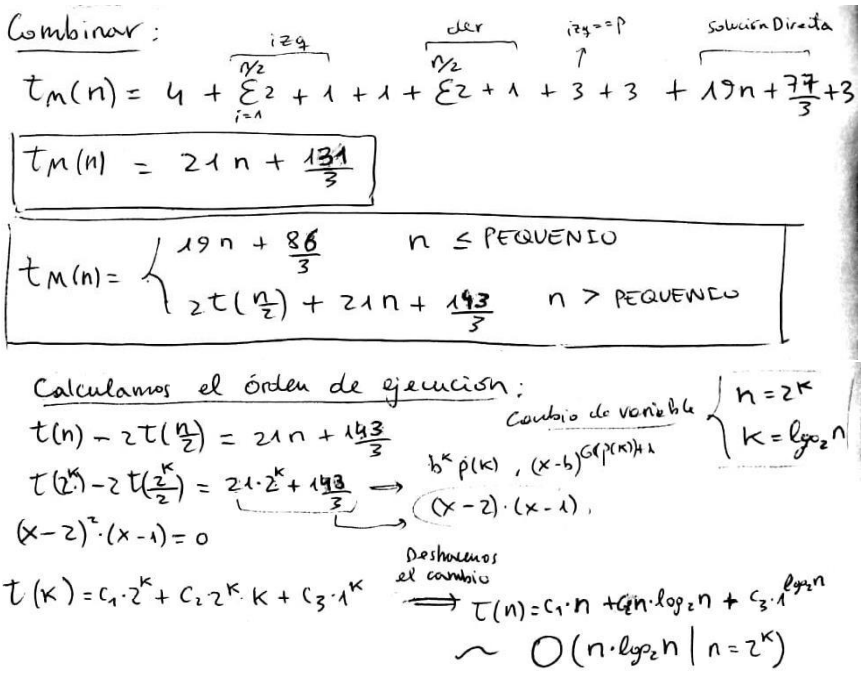
4.2. Peor caso

En el peor caso cada cadena está formada por las tres últimas letras de su conjunto repetidas constantemente, de esta manera el bucle *while* de *solucionDirecta*, realiza las mayores iteraciones posibles. Por ejemplo: el conjuntoS (xycba) y la cadena (abcabcabcabc).

No entraría nunca en la posibilidad de carácter repetido, y únicamente resetearía las variables contador, *letrasUsadas* y subcadena, cada 3 iteraciones, cuando encuentra la subcadena válida. El problema de la ineficiencia se encuentra en la función Combinar porque al recibir las listas llenas, tenemos que encontrar los índices izq y der, es decir, que se encontraría al principio y final de la cadena, por lo que se recorrería entera. Después como la zona conflictiva es toda la cadena habría que llamar a solución directa de toda la cadena (como si no se hubiera utilizado el *divideyvenceras*).

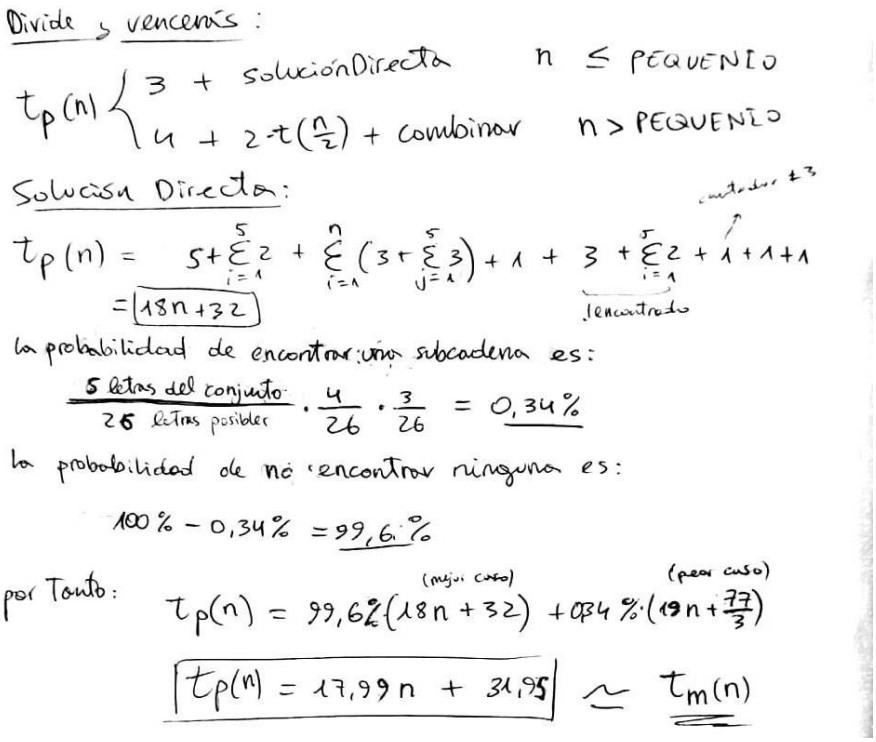
El algoritmo es muy poco eficaz en este caso ya que, al ser muy poco probable, se ha optado por intentar ahorrar instrucciones en todos los demás casos, aunque perjudique a este en concreto.

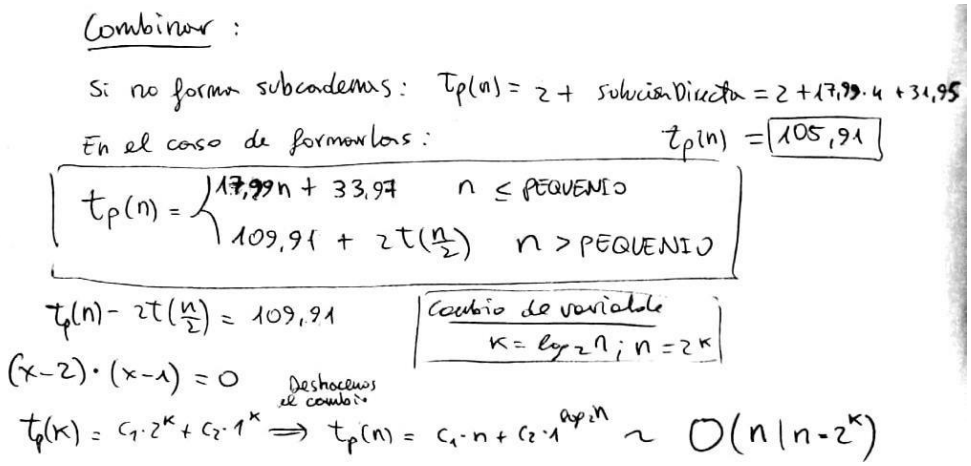




4.3. Caso promedio

En el caso promedio hemos determinado que es similar al mejor caso, debido a que es muy poco probable que todas las letras de la cadena formen parte del conjunto y menos aún que estén dispuestas de manera que formen todas las subcadenas válidas de tres posibles. Según lo calculado, salen una media de (0,0034 * n) subcadenas pues el 0,34 % es la probabilidad de que una letra forme una subcadena. Así que tanto el orden como la eficiencia del algoritmo es idéntico al mejor caso.





5. Explicación de las variables utilizadas

5.1. Variables usadas en la solución directa

La variable *posicion*, se va actualizando por iteración, y como su nombre indica sirve para guardar la posición de la letra leída en la cadena (comenzando por 0).

La variable *subcadena* es la encargada de guardar la posible subcadena que estamos analizando. Su uso es importante, ya que al no poder formar subcadenas con repeticiones, si se repite alguna letra, deberíamos saber con qué parte de la subcadena quedarnos.

LetrasUsadas no es más que un array de booleanos, que se utiliza para saber las letras que se encuentran en una posible subcadena y no repetirlas.

Las variables *i, j, o, k, t*, son solo índices para utilizarlos en los bucles.

La variable *contador*, sirve para saber el número de letras en una subcadena, si llega a 3, significa que tenemos una subcadena válida.

Cont, es una variable que se utiliza para saber con qué parte de la variable *subcadena* quedarnos en caso de repetición.

El booleano *encontrado* es utilizado como condición de terminación del bucle while, cuando nos encontramos una letra del conjunto.

5.2. Variables usadas en la solución de Divide y Vencerás

5.2.1. Variables de la función Divide y Vencerás

Las variables que usamos como parámetros son *p* y *q*. Son los índices de la parte de la cadena a la que aplicamos el divide y vencerás (en la primera llamada es la toda la cadena). Por otro lado, la solución la devolvemos en una lista enlazada que contiene las posiciones de las subcadenas encontradas.

La constante *PEQUENIO* es el valor que hemos considerado para llamar a *soluciónDirecta* sin tener que dividir la cadena.

La variable *tam*, contiene el número de letras que hay entre los índices que ha recibido la función, y la variable *n*, tiene el índice de la posición central de cadena.

5.2.2. Variables de la función Combinar

Los parámetros recibidos son: *p, q, n, s1* y *s2.* El primero *p* es el índice de la primera letra de la cadena que queremos combinar, *q* es el último índice. La variable *n* contiene la posición por dónde la cadena se separó a la hora de resolverse. Por último, *s1* y *s2,* son las listas enlazadas que se obtuvieron como resultados de resolver la primera parte y la segunda parte de la cadena.

La función *pertenece* es una función auxiliar que es utilizada para comprobar si la letra de la cadena se encuentra en el conjunto, para ello le pasamos su índice en la cadena, devolviendo verdadero en ese caso.

Las variables *izq*, y *der*, que se tratan de los índices en la cadena, dónde pueden existir conflictos a la hora de combinar los resultados.

La variable *unir,* que es una lista de enteros con las posiciones de las subcadenas válidas en la franja conflictiva.

5.2.3. Variables de la Main

Tenemos las variables globales *cadena* y *conjuntoS,* que guardan la cadena que queremos resolver y el conjunto de letras sobre las que queremos formar subcadenas. Son globales ya que al usarlas en casi todas las funciones sería un malgasto de memoria pasarlas como parámetros, además que puede darse el caso de que sean muy largas, de esta manera es más fácil trabajar con ellas.

La variable *numElementos,* contiene el tamaño de *cadena, y niteraciones* el número de cadenas.

Por último, el resultado constaría de la variable *aciertos*, que guarda el número de subcadenas válidas encontradas y *resultado* que es la lista de enteros que contiene todas las posiciones.

6. Validación del algoritmo

La validación del algoritmo ha pasado por diferentes fases. Mediante el programa generador, creamos un fichero con el número de cadenas que le introducimos, además del rango de tamaño que queremos que sean, y eligiendo entre tres casos posibles:

1. Mejor caso: cadenas formadas por letras que no pertenecen a conjuntoS.
2. Peor caso: cadenas compuestas solo por subcadenas válidas.
3. Aleatorio: cadenas generadas utilizando la función random.

Nos centramos primero en la función soluciónDirecta donde fuimos comprobando con el fichero generado, en nuestro programa generador, con cadenas de tamaños pequeños, como pueden ser de entre 3 y 20 letras para su fácil comprobación, que todo funcionaba correctamente. Con cada prueba que hacíamos iban surgiendo casos que no se resolvían correctamente, por lo que creamos un fichero aparte llamado CasosEspeciales.txt para añadir aquellos casos que nos diesen problemas.

Una vez que sabemos que esos casos conflictivos funcionaban correctamente, pasamos a realizar el divide y vencerás, el cual resuelve de la misma manera todos los casos del fichero CasosEspeciales.txt pero poniendo el valor de la constante PEQUENIO en un valor de 8 para poder dividir esas cadenas.

Por último, cualquier fichero generado que es analizado por divideyvenceras nos proporciona el mismo resultado que soluciónDirecta.

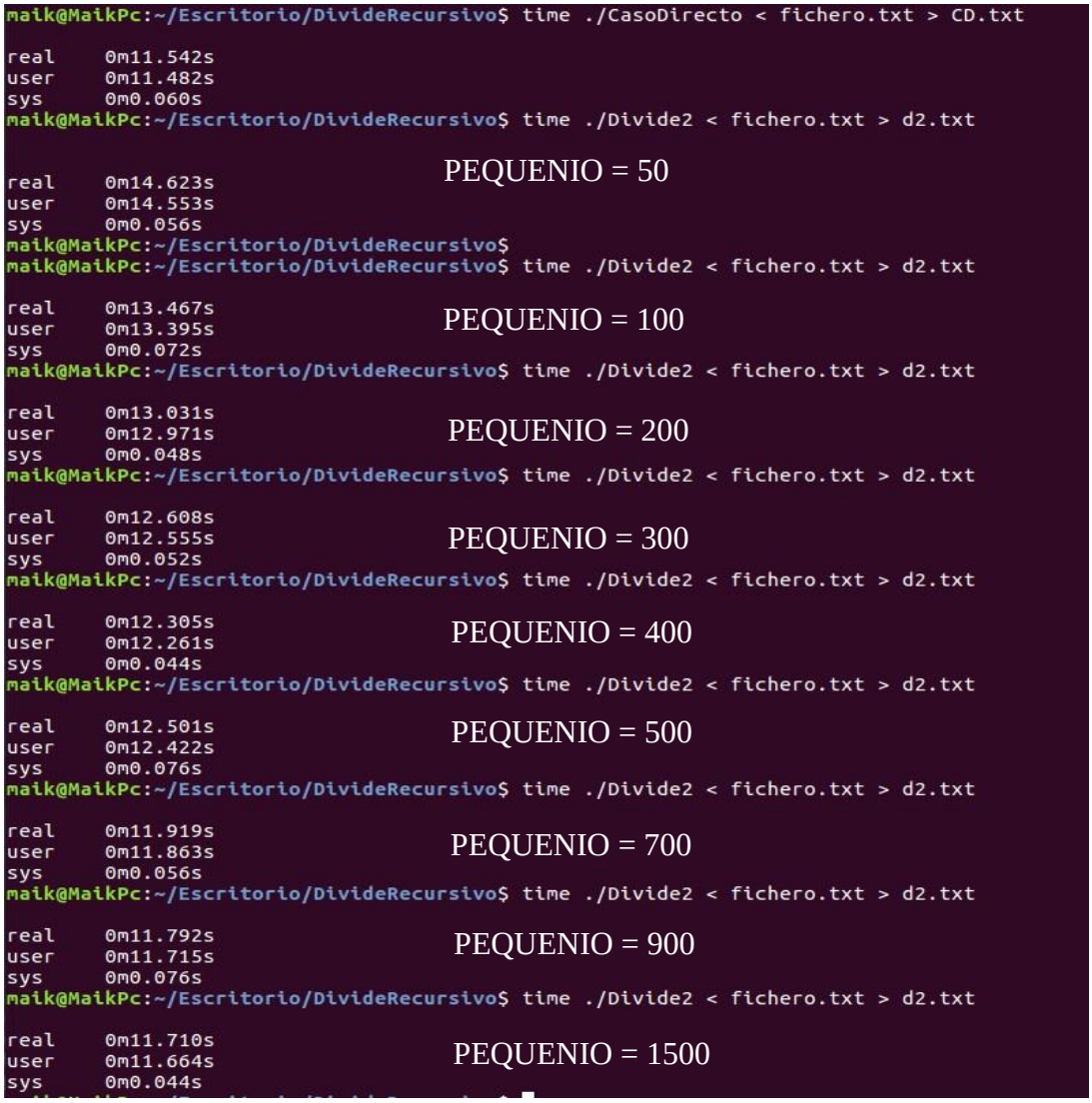
7. Estudio experimental

El valor de *PEQUENIO* ha sido calculando de forma experimental según los resultados obtenidos con el comando *time.* Para obtener un valor más real hemos quitado la salida de los datos, es decir, las operaciones que muestran por pantalla los resultados (cout).

Como podemos observar en el Caso Directo resuelve el fichero en 11.5 s de media y a continuación ejecutamos el divide y vencerás para distintos valores de *PEQUENIO*. Cuando este valor es muy bajo, el tiempo aumenta pues se hacen más llamadas a la función *divideyvenceras* por lo que los resultados de cada división se tienen que combinar más veces, lo que supone más trabajo. A medida que aumentamos el valor de *PEQUENIO* podemos ver que se reduce el tiempo, pero a partir de 400 el tiempo que mejora es cada vez menor que tenemos que aumentar.

Por ello para cadenas de 30.000 caracteres proporcionalmente podemos escoger 400 como tamaño o si queremos reducir el tiempo lo máximo posible deberemos escoger 1500 o más como tamaño de *PEQUENIO*.

Los resultados comentados corresponden a la imagen siguiente para un fichero de 10.000 cadenas de entre 30.000 caracteres generadas por el caso *Aleatorio.*



8. Contraste del estudio teórico y experimental

En el peor caso con cadenas suficientemente grandes podemos apreciar claramente la diferencia de tiempos con el mejor caso y el promedio, siendo el primero mucho mayor. En el análisis teórico podemos decir que se debe a la diferencia de órdenes, en el promedio y el mejor caso son de O(n) y en el peor caso de $O(n*\log_2 n)$.

En cuanto al análisis experimental, en este ejercicio comprobamos que el divide y vencerás no cumple correctamente su función de reducir el tiempo en todos los casos, sino que depende del número de caracteres de las cadenas y del valor de PEQUENIO.

El tiempo se reduce en el caso donde la constante es aproximadamente 2.000 veces más pequeña, lo podemos ver en el siguiente ejemplo con una cadena de tamaño 10.000.000 y un valor de *PEQUENIO* de 5.000, donde este último valor es el límite entre que salga un tiempo mayor o menor para este caso, ya que si ponemos 2.500 saldrá mayor tiempo y si aumentamos a partir de 5.000 se mantiene en un tiempo por debajo de la Solución Directa, a excepción del peor caso que como hemos visto en todos los casos mostrados sale mucho mayor.

```
maik@MaikPc:~/Escritorio/DivideRecursivo$ ./generador
Numero de cadenas: 1
Rango de caracteres en las cadenas N M: 10000000 10000000
1 Mejor caso, 2 Peor caso, 3 Aleatorio: 1
maik@MaikPc:~/Escritorio/DivideRecursivo$ time ./CasoDirecto < fichero.txt > CD.txt

real    0m0.519s
user    0m0.511s
sys     0m0.008s
maik@MaikPc:~/Escritorio/DivideRecursivo$ time ./Divide2 < fichero.txt > d2.txt

real    0m0.492s
user    0m0.487s
sys     0m0.004s
maik@MaikPc:~/Escritorio/DivideRecursivo$ ./generador
Numero de cadenas: 1
Rango de caracteres en las cadenas N M: 10000000 10000000
1 Mejor caso, 2 Peor caso, 3 Aleatorio: 2
maik@MaikPc:~/Escritorio/DivideRecursivo$ time ./CasoDirecto < fichero.txt > CD.txt

real    0m0.786s
user    0m0.769s
sys     0m0.016s
maik@MaikPc:~/Escritorio/DivideRecursivo$ time ./Divide2 < fichero.txt > d2.txt

real    0m11.338s
user    0m11.220s
sys     0m0.060s
maik@MaikPc:~/Escritorio/DivideRecursivo$ ./generador
Numero de cadenas: 1
Rango de caracteres en las cadenas N M: 10000000 10000000
1 Mejor caso, 2 Peor caso, 3 Aleatorio: 3
maik@MaikPc:~/Escritorio/DivideRecursivo$ time ./CasoDirecto < fichero.txt > CD.txt

real    0m0.534s
user    0m0.513s
sys     0m0.012s
maik@MaikPc:~/Escritorio/DivideRecursivo$ time ./Divide2 < fichero.txt > d2.txt

real    0m0.528s
user    0m0.512s
sys     0m0.016s
```

Como conclusión diríamos que cuantas más divisiones hagamos de la cadena más lento sería el programa.

9. Conclusión

Para terminar, debido a los resultados obtenidos podemos concluir que la técnica de divde y vencerás puede resultar útil en determinados casos, pero en otros puede conducir a un tiempo peor que la solución directa, además de complicar bastante el trabajo de la resolución del problema como ha sido en este caso la función *combinar*, ya que en nuestro problema surgían muchas situaciones a tener en cuenta cuando dividíamos la cadena.

En aspectos generales, hemos comprendido correctamente el funcionamiento y esquema de divide y vencerás a la hora de solucionar ciertos problemas, y a tener en cuenta que no siempre es la mejor técnica a aplicar, sino que debe haber un análisis detrás de lo respalde.