MEMORIA

DyV AED II

Nombre Completo	Subgrupo	DNI
Juan Pedreño García	2.4	
Patricia Cuenca Guardiola	2.4	

Índice

1. Introducción

El algoritmo divide y vencerás (DYV), separa un problema en subproblemas que se parecen al problema original, de manera recursiva resuelve los subproblemas y, por último, combina las soluciones de los subproblemas para resolver el problema original.

El problema que hemos resuelto aplicando el algoritmo divide y vencerás es:

6) Dada una cadena A de longitud n, un natural m, n y un carácter C, hay que encontrar B, la subcadena de A de tamaño m con más apariciones consecutivas del carácter C. Devolver el índice p de comienzo de la solución B y el número de veces que aparece el C consecutivamente en B. En caso de empate, será válida cualquiera de las soluciones óptimas.

Ejemplo: n=10, m=5, C=c

A= c d d a b c d a c c

Solución: B, posición de inicio igual a 6, y número de apariciones consecutivas igual a 2.

2.Diseño de una solución divide y vencerás

Pseudocódigo

var maximoGlobal;

var posicionGlobal;

var vectorGlobal;

operacion SecuenciaMaxima(c: caracter; l, r: entero)

si l == r **entonces**

    devolver vectorGlobal[l] == c

sino

    mid := (l + r) / 2

    l_max := SecuenciaMaxima(l, mid, c)

    r_max := SecuenciaMaxima(mid+1, r, c)

```
cruce_max := 0,  conteo_izquierda := 0, conteo_derecha := 0,  posicion_inicial := mid
si vectorGlobal[mid] == c and vectorGlobal[mid+1] == c entonces
    i := mid
    mientras i >= l and vectorGlobal[i] == c entonces
        conteo_izquierda++
        posicion_inicial := i
        i- -
    finmientras
    j := mid +1
    mientras j <= r and vectorGlobal[j] == c entonces
        conteo_derecha++
        j++
    finmientras
finsi
cruce_max := conteo_derecha + conteo_izquierda
maximo := max(cruce_max, max(l_max, r_max))
si cruce_max == maximo and cruce_max > maximoGlobal entonces
    posicionGlobal := posicionInicial
    maximoGlobal := cruce_max
finsi
si r_max == maximo and r_max > maximoGlobal entonces
    maximoGlobal := r_max
    posicionGlobal := mid +1
finsi
```

si l_max == maximo **and** l_max > maximoGlobal **entonces**

maximoGlobal := l_max

posicionGlobal := l

finsi

devolver maximo

Explicación del algoritmo divide y vencerás y decisiones de diseño

Para la resolución de este problema hemos decidido usar tres variables globales (maximoGlobal, posicionGlobal, vectorGlobal) para almacenar la mejor solución encontrada hasta el momento en cualquier parte del vector. Además, permite rastrear la longitud máxima de la subsecuencia encontrada y su posición inicial. VectorGlobal es el vector de caracteres sobre el que estamos trabajando.

Primero, definimos el caso base, se cumple si el vector solo tiene un elemento. Si l == r, se comprueba si vectorGlobal[l] es igual al carácter "c" y se devuelve 1 si es cierto, o 0 en caso contrario.

Después, dividimos el problema en dos subproblemas. Dividimos el vector en dos mitades hasta el caso base: mid = (l + r) / 2. Además, llamamos recursivamente a SecuenciaMaxima en la parte izquierda (l, mid) y derecha (mid + 1, r).

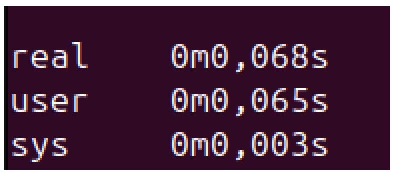
Seguidamente, combinamos resultados. Calculamos l_max y r_max con los resultados obtenidos de las llamadas recursivas. Seguidamente, analizamos si existe una secuencia que cruce mid. Si vectorGlobal[mid] == c y vectorGlobal[mid+1] == c, se cuentan los caracteres "c" consecutivos hacia la izquierda (conteo_izquierda) y hacia la derecha (conteo_derecha). Por último, se calcula el cruce: cruce_max = conteo_izquierda + conteo_derecha.

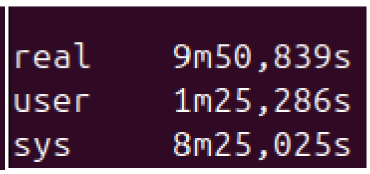
Determinamos la mejor solución, para ello obtenemos el máximo entre l_max, r_max y cruce_max. Si cruce_max es el nuevo máximo global, se actualizan maximoGlobal y posicionGlobal. Se repite el mismo proceso para r_max y l_max. De esta forma, obtiene la mejor solución de la parte izquierda, la mejor de la derecha y la mejor secuencia que cruza la división.

Por último, se devuelve la longitud máxima encontrada en el subvector actual.

Estructuras de Datos Utilizadas

Respecto a las estructuras de datos empleadas, hemos decidido usar un vector global (vectorGlobal), que contiene los elementos de la secuencia en la cual se busca la subsecuencia más larga de un carácter específico. Usamos un vector global ya que cuando probamos a pasarlo por referencia, con tamaños muy grandes, como 800000 caracteres, teníamos que pasarle un vector de tamaño 800000 en cada llamada recursiva, lo que incrementó considerablemente el tiempo de ejecución del algoritmo. De la misma forma, para no desperdiciar memoria, le asignamos el tamaño en el main para que el tamaño del vector dependa de la entrada.





Comparación del tiempo de ejecución con una cadena del mismo tamaño pero con un vector global(imagen de la izquierda) versus pasarlo como parámetro a la función (imagen de la derecha).

Además, hemos utilizado variables enteras (l, r, mid). Por un lado, l y r representan los índices del inicio y fin del subvector en el que se está operando. Por otro lado, mid representa el punto medio del subvector en cada paso de la recursión. Por último, hemos manejado variables auxiliares (l_max, r_max, cruce_max, conteo_izquierda, conteo_derecha, posicion_inicial). Las variables l_max y r_max guardan las longitudes máximas de las subsecuencias en la parte izquierda y derecha. La variable cruce_max representa la mejor secuencia que cruza mid. Además, conteo_izquierda y conteo_derecha cuentan los caracteres consecutivos c hacia la izquierda y derecha de mid, respectivamente.

3.Análisis teórico del tiempo de ejecución

$$\begin{cases} t_m(n) = 2 \\ t_M(n) = 2t_M(n/2) + n + 20 \end{cases}$$

El tiempo en el mejor caso es O(1) ya que solo tiene que comprobar la condición del primer if y devuelve el resultado.

Por otro lado, para obtener el peor tiempo hay que resolver la ecuación de recurrencia.

La parte homogénea es:

$t_M(n) = 2t_M(n/2) \rightarrow t'(k) = 2t'(k-1) \rightarrow x-2$

La parte no homogénea es:

$t_M = n+20$

aplicamos cambio de variable: $k = \log_2 n$; $n = 2^k t_M(n) = t(2^k) = t(k)$

$t(k) = \log_2 n + 20 \rightarrow t(2^k) = \log_2 2^k + 20 \rightarrow t(2^k) = (k+20) \times 1^k \rightarrow (x-1)^2$

Soluciones = 2, 1, 1

Ecuación característica = $t(k) = c_1 \times 2^k + c_2 + c_3 \times k \rightarrow t(n) = c_1 \times 2^{\log_2 n} + c_2 + c_3 \times \log_2 n \rightarrow O(2^{\log_2 n}) = O(n)$

Según el teorema maestro: $\Theta(n^{\log 2\ 2} \times \log n) = \Theta(n \times \log n)$

4.Implementación: Programación del algoritmo

```
#include<iostream>

#include<vector>

using namespace std;


//La variable maximoGlobal sirve para actualizar la posición donde empieza la
secuencia que buscamos y la variable posicionGlobal para almacenarla

int maximoGlobal;

int posicionGlobal;

vector<char> vectorGlobal; // Cadena con la que estamos trabajando

int max_seq( int l, int r, char c) // Función SecuenciaMaxima

{

   //Si l es igual a r devolvemos 1 o 0 según sea el caracter que buscamos o
no

   if(l == r) return vectorGlobal[l] == c;

   //Calculamos la mitad y hacemos una llamada recursiva a ambas mitades

   int mid = (l+r)/2;

   int l_max = max_seq( l, mid, c);

   int r_max = max_seq( mid+1, r, c);


   //Ahora lo que vamos a comprobar es si alguna secuencia pasa entre las dos
mitades y por tanto no es reconocida completamente por las llamadas anteriores
```

```
    int cruce_max = 0;          //El tamaño de la subsecuencia que pasa por en
medio

    int conteo_izq = 0;         //Los elementos que hay hasta mid

    int conteo_der = 0;         //Los elementos que hay desde mid+1

    int posInicial = mid;       //La posición inicial de esta hipotética
subsecuencia

    if(vectorGlobal[mid] == c && vectorGlobal[mid+1] == c) {    //Esto solo se
da en el caso de que el caracter mid y el mid+1 sean el caracter que buscamos

        //Contamos los elementos que hay desde mid hasta l

        int i = mid;

        while(i>=l && vectorGlobal[i] == c){

            conteo_izq++;

            posInicial = i;

            i--;

        }

        //Contamos los elementos que hay desde mid+1 hasta r

        int j = mid+1;

        while(j<=r && vectorGlobal[j] == c){

            conteo_der++;

            j++;

        }

    }

    //El tamaño de esta subsecuencia es la suma de los dos conteos

    cruce_max = conteo_der + conteo_izq;

    //El maximo de las 3 subsecuencias calculadas es la que buscamos
```

```
    int maximo = max(cruce_max, max(l_max, r_max));

    //Según la subsecuencia mayor de las 3. Cambiamos la variable global maximo
y la posicion inicial de la tesecuencia

    if(cruce_max == maximo && cruce_max > maximoGlobal){

        posicionGlobal = posInicial;

        maximoGlobal = cruce_max;

    }

    if(r_max == maximo && r_max > maximoGlobal){

        maximoGlobal = r_max;

        posicionGlobal = mid+1;

    }

    if(l_max == maximo && l_max > maximoGlobal){

        maximoGlobal = l_max;

        posicionGlobal = l;

    }

    //Devolvemos la subsecuencia más grande.

    return maximo;

}

int main(){

    //Número de casos

    int b;

    cin >> b;

    for(int a = 0; a<b; a++){

        int n, m;
```

```
        char c;

        //Leemos los datos de entrada

        cin >> n >> m >> c;

        vectorGlobal.resize(n); //Asignar tamaño al vector según la entrada

        for(int i = 0; i<n; i++){

            cin >> vectorGlobal[i];

        }

        //Variables globales de donde se encuentra la mayor secuencia de
caracteres consecutivos y su posición de inicio

        maximoGlobal = 0;

        posicionGlobal = 0;

        //Calculamos la máxima secuencia

        int maximo = max_seq( 0, n, c);

        //Si el máximo es mayor al tamaño de la subsecuencia que nos piden,
hacemos que sea igual a m

        if(maximo > m) maximo = m;

        //Si la posición de inicio no es coherente por el tamaño de m, la
ajustamos

        if(posicionGlobal + m > n){

            posicionGlobal = n - m;

        }

       //Imprimimos la salida

        cout << "DyV: ";

        cout << "MAXIMO: " << maximo;

        if(maximoGlobal > 0) cout << " POSICIÓN: " << posicionGlobal+1;
```

```
        cout << endl;

    }

    return 0;

}
```

Respecto a la asignación del tamaño del vector, conocíamos la función memset para arrays pero al usar un vector le preguntamos al ChatGPT una alternativa y nos aconsejó usar la función resize().

5.Diseño y aplicación de un proceso de validación del algoritmo dyv

El algoritmo que hemos empleado para la validación es el siguiente:

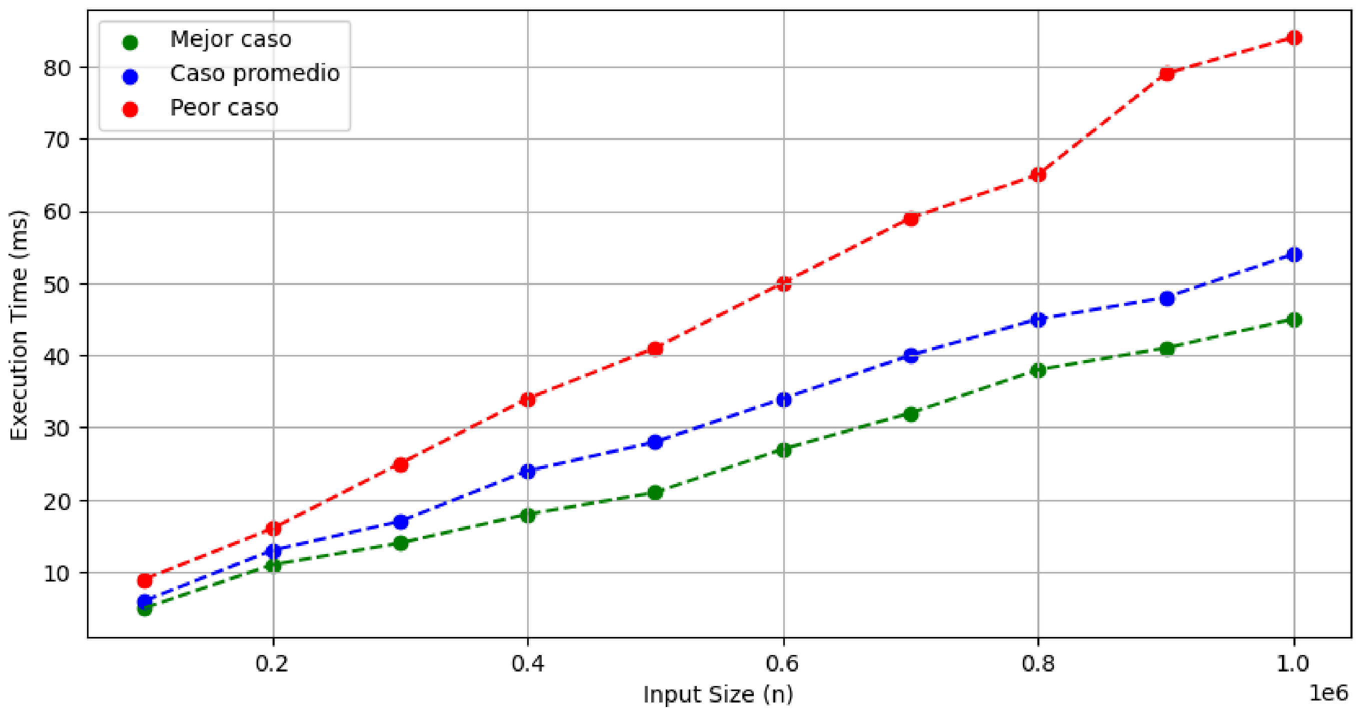
```
void directo(vector<char> v, int n, int m, char c){
    //Variables de maximo y posicion
    int posicion = 0;
    int maximo = 0;
    //Recorremos el vector de m en m elementos
    for(int i = 0; i<=n-m; i++){
        //Hallamos la mayor subsecuencia
        int actualsubsecuencia = 0;
        for(int j = i; j<i+m; j++){
            if(v[j] == c){
                actualsubsecuencia++;
            }else{
                actualsubsecuencia=0;
            }
            if(actualsubsecuencia>maximo){
                maximo = actualsubsecuencia;
                posicion = i;
            }
        }
```

```
    }
    //Imprimimos la salida
    cout << "MAXIMO: " << maximo;
    if(maximo>0) cout << " POSICION: " << posicion+1;
}
```

Para comprobar que el método de divide y vencerás funcionaba comparamos la salida que generaba el algoritmo anterior y el directo con casos de prueba de distintos tamaños. El algoritmo directo tiene un orden de O(n*m). Por tanto, este algoritmo es bueno cuando tiene un tamaño de m pequeño. Además, hemos utilizado los ficheros de prueba de la carpeta “casos”. Sin embargo, hay casos en los que devuelve una posición distinta el directo y el algoritmo DyV (los casos en los que m es mayor a la mayor subsecuencia de caracteres repetidos consecutivamente). Esto se debe a que el directo siempre saca la primera posición que contiene toda la subsecuencia que buscamos, mientras que en el DyV depende de cómo se hayan ido haciendo las llamadas recursivas.

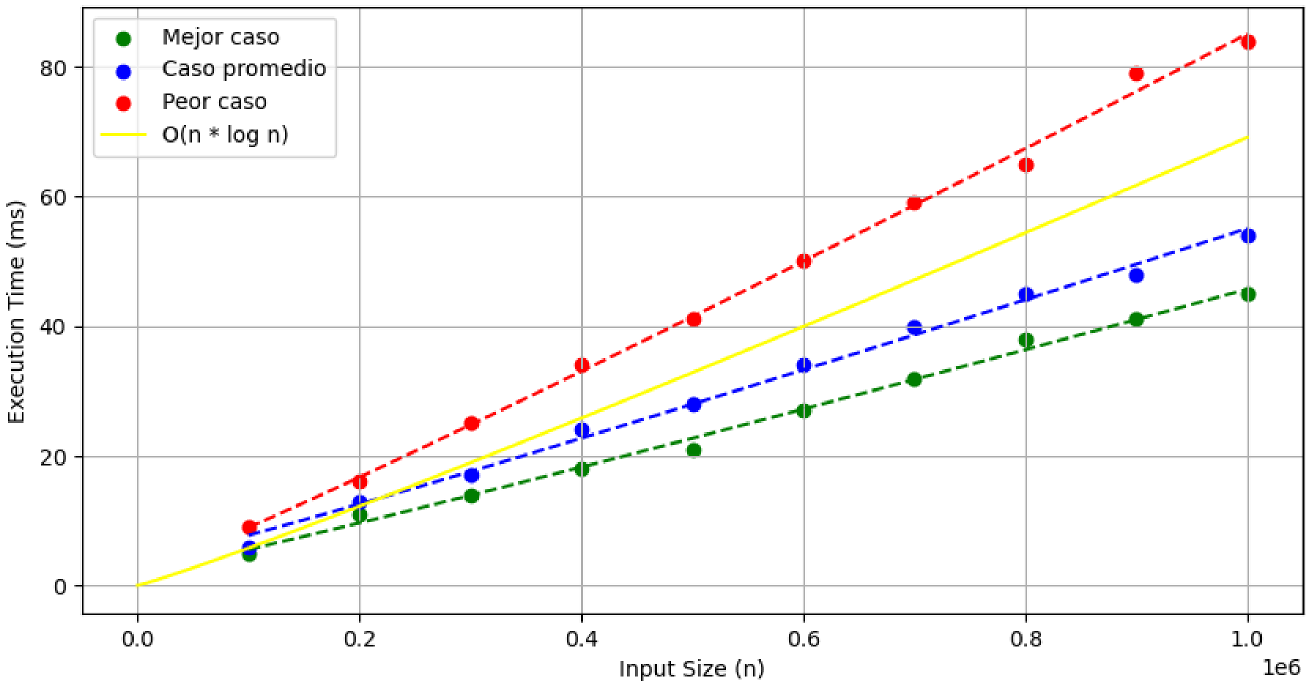
6.Estudio experimental del tiempo de ejecución

Los casos de prueba que hemos generado para comprobar el adecuado funcionamiento los hemos adjuntado en la carpeta “casos”. Hemos probado con cadenas de entrada grandes como pequeñas. Por tanto, hemos averiguado que el peor caso es cuando toda la cadena solo tiene el carácter que buscamos, es decir, si buscamos una subcadena de como mucho 100 ‘a’ en una cadena de 500 y todas las letras que aparecen son la ‘a’. Por otro lado, el mejor caso es cuando el carácter que buscamos no aparece consecutivamente, es decir, solo es 1 o 0. Ese es el mejor caso ya que no se ejecutan los bucles while que comprueban que los caracteres de en medio sean iguales y haya que seguir iterando. El caso promedio es el resto de casos, que hemos generado aleatoriamente.



7. Contraste del estudio teórico y el experimental

Tabla con los valores normalizados comparando con O(nlog(n)*C):



Como podemos ver en esta gráfica, no hay diferencia en el crecimiento entre los tiempos de ejecución del algoritmo y n*log(n) multiplicado por una constante (0.000005). Por lo que no hay discrepancias entre nuestro estudio teórico y nuestro estudio experimental.

8. Conclusiones

Esta práctica nos ha parecido muy útil para entender el algoritmo divide y vencerás, pues nos hemos tenido que enfrentar a un problema y aplicar de forma práctica lo aprendido en teoría. Para realizar el algoritmo nos ayudamos de las diapositivas de teoría, así como de los scripts de código que hemos encontrado en el repositorio. Las mayores dificultades que hemos afrontado son el análisis tanto teórico como experimental, ya que tuvimos que generar muchos ficheros de prueba para poder asegurarnos de que el código funcionaba correctamente y de la forma más eficiente posible.

Hemos calculado que aproximadamente le hemos dedicado unas 20 horas al proyecto cada miembro, contando el tiempo invertido planteando el algoritmo, refinando el código, haciendo los respectivo análisis, etc.

En definitiva, nos ha resultado un trabajo muy importante para lograr comprender la técnica divide y vencerás, aunque nos haya llevado su tiempo terminarlo.