Proyecto de prácticas AED-2.

Algoritmo: Divide y Vencerás.

Problema nº 6 (*)

Grupo: 3.2
Profesor de prácticas: Carlos Hoyos Barceló.

Sergio Sánchez López - 49245005G
Dario García Usero - 26937651M

Enlace al proyecto en GitHub: Proyecto-DyV

(*)En la primera clase yo, Sergio, hable con Carlos Hoyos para ver que problema escogía ya que no tenía pareja en ese momento por lo que se me asignó el problema según mi DNI sin tener en cuenta el de mi compañero ya que fue al cabo de una semana y media cuando decidimos hacer el trabajo juntos.

ÍNDICE

1. Introducción

En este trabajo hemos llevado a cabo el estudio diseño e implementación de un algoritmo capaz de resolver un problema.

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.

Para ello hemos implementado un algoritmo en C++ aplicando la metodología Divide y Vencerás (DyV).
A parte de nuestro algoritmo hemos creado un generador de casos de prueba capaz de generar entradas para nuestro algoritmo y así poder testearlo.

Tras su prueba y ver que todo funcionaba mediante un script en bash hemos automatizado la ejecución de nuestro algoritmo para un n creciente desde 500 hasta 1.000.000 y la extracción de su tiempo de ejecución para volcarlo todo a un archivos.
Después mediante otro programa en python hemos dibujado todas las gráficas comparando tiempos para mejor, peor y caso promedio para su posterior estudio.

Todos estos programas y archivos están disponibles en el repositorio de GitHub (Proyecto-DyV)

En el fichero README.md del repositorio encontrarás una explicación de que es cada fichero.

2. Diseño de una solución al problema aplicando Divide y Vencerás.

El pseudo-código que diseñamos para resolver nuestro problema es el siguiente:

```
contarCcons(string A, char C) {

    resultado

    cont = 0
    index = -1

    for(char x : A) {
        if(x = C) {
            if(cont==0) {
                index = posicion_actual
            }
            cont++
        }
        else {
            if(cont > resultado) {
                actualizar resultado
            }
            cont = 0
        }
    }

    return resultado
}


combinar (string A, int div resultado_Izq, resultado_Der) {

    if (resultado_Izq mejor que resultado_Der) {
        firstAttempt = resultado_Izq
    }
    else {
        firstAttempt = resultado_Der
    }

    if(A[div-1]!=C || A[div]!=C) {
        return firstAttempt
    }
    else {
        iterador = div-1
```

```
        contadorSecondAttempt = 0
        while(iterador>=0 && A[iterador]==C) {
            contadorSecondAttempt++
            iterador--
        }

        indexIniSecondAttempt=iterador+1

        if (necesaryC > resultado_Der) {
            return firstAttempt
        }
        else {
            for(int i =div;i<A.length;i++) {
                secondAttemptCont++
            }

            if(secondAttemptCont mejor que firstAttempt ){
                return secondAttempt
            }
            return firstAttempt
        }
    }
}



DyV_algorithm (string A, int i, int n, int m, char C) {

    if(m<1) {
        salir
    }
    if(n<=m) {
        contarCcons(A,C)
    }
    else {
        combinar(A,n,C,DyV_algorithm(A/2,i,m,C),DyV_algorithm(A/2,C))
    }
}
```

Nuestro algoritmo consta de 2 casos, caso base, en el cual la cadena a analizar es del mismo tamaño que el de la cadena resultado, ya que pensamos que si tenemos ya una cadena del mismo o menor tamaño que la del resultado no merece la pena seguir dividiendo.
En el otro caso entramos en la parte de dividir resolver y combinar, y por tanto en una llamada recursiva. Para resolver con divide y vencerás hemos decidido dividir la cadena en dos mitades y resolver ambas por separado y luego combinar.

Lo primero es el funcionamiento de contarCcons, el cual es la base de nuestro algoritmo ya que busca una subcadena dentro de la cadena que se le pasa en la cual haya el mayor número de char C consecutivos, esta recibe como parámetros el string A a analizar y el char C que debe buscar de manera consecutiva.
Es muy sencillo, la variable resultado, la cual es de un tipo de dato que no especificamos en el pseudo ya que más tarde en la parte de implementación se explica exactamente como es ya que es más tema de implementación que de algoritmia, almacena el índice de inicio y en número de ocurrencias de la mejor solución de la cadena pasada como parámetro.

Una vez tenemos la mejor solución de cada una de las mitades de la cadena original se combina esa solución para buscar la solución final.
La función combinar es sin duda la más compleja ya que tiene que comprobar que cuando combinas ambas subcadenas que en el centro no se forme una solución la cual sea mejor que las que teníamos, esta recibe como parámetros el string A original, el punto de división de ambas mitades, y la mejor solución de la parte izquierda y derecha.
Para combinar primero obtiene la mejor solución de ambas partes para que en caso de que la unión de ambas sub-mitades no generen una solución devolver esta primera (firstAttempt).
De ahora en adelante llamaremos parte A a la mitad izquierda y parte B a la otra mitad derecha.
Cuando se comprueba que en la unión se puede generar otra solución, lo que se hace es avanzar hacia atrás en A desde el punto de división de las dos mitades, e ir contando el número de ocurrencias que se van encontrando, cuando ya no hay más ocurrencias consecutivas se calcula cuál sería el número de char C consecutivos en la derecha necesarios para tener una mejor solución que la que ya teníamos.
Si el número de ocurrencias necesarios en la parte B es mayor que el número de ocurrencias de la mejor solución de la parte B no se sigue y se devuelve la primera solución ya que no vamos a encontrar más ocurrencias en B que las que hay en su mejor solución.
En caso de que las C necesarias sea menor, entonces desde el punto de división contamos hacia adelante en la parte B y vamos sumandolas a el número de ocurrencias que teníamos ya de contar hacia atrás de A. Una vez con la solución de el centro la comparamos con firstAttempt y devolvemos la mejor.

3. Análisis teórico.

Para analizar el tiempo de ejecución primero hemos calculado cuánto es el tiempo y orden de contarCcons.
Para contarCcons tenemos que recorrer todos los caracteres del string pasado como parámetro por lo que su orden es O(n).

Con esto ya podemos calcular cual seria el tiempo para el mejor caso (t_m) el cual seria cuando la cadena original es menor que o igual que que la cadena resultado, en este caso solo se pasaría por el caso base (contarCcons) por lo que tendríamos un tiempo de:

$$t\_n(n) = n \rightarrow O(n)$$

Para el peor caso es un poco más complicado ya que tuvimos que pensar cómo sería una cadena la cual maximiza el tiempo del algoritmo y llegamos a la conclusión de que una cadena la cual es mayor que m (longitud cadena solución) y la cual es toda el carácter que se desea buscar maximiza el tiempo de ejecución, esta cadena haria log(n) llamadas recursivas como mucho y luego tendría que procesar todos los caracteres en contarCcons y luego para combinar se vuelven a procesar todos los caracteres por segunda vez. Con esto podemos decir que el tiempo del peor caso (t_M) es:

$$t\_M = 2t(n/2) + O(n) = 2n * \log(n) \rightarrow O(n \log(n))$$

4. Implementación del algoritmo.

A Continuación todo el código de la implementación, hay ciertos cambios respecto al pseudo pero solo detalles de implementación, el esquema del algoritmo y el funcionamiento es el mismo.

Lo único que si explicaremos ahora será el struct Data, este struct es lo que devuelven las funciones como solución ya que en el enunciado se nos pide que digamos tanto la posición de donde empieza dicha solución y en número de ocurrencias, además para el tema de comparaciones de cual es mejor solución y sus índices es útil.
Si se desea ver de una manera más cómoda en la portada hay un enlace al proyecto en GitHub.

```cpp
#include <stdlib.h>
#include <iostream>
#include <string>
#include "generardorString.cpp"

using namespace std;


// Struct usado para devolver los datos necesarios para comparar soluciones y resolver el problema
typedef struct dataStruct
{
   int index;
   int nOcu;
}Data;



// Funcion que resuelve el problema de manera directa
// Cuenta las ocurrencias de C consecutivas y devuelve el indice donde esta ocurrencia comienza y cuantas ocurrencias hay
/*
//  string A -> string de entrada para anlizar
//  i -> indice inicio de analisis de A
//  n -> longuid del string a analizar
//  C -> caracter que se desea buscar
*/
Data contarCcons(string A,int i, int n, char C){

   Data res ={-1,0};   // Variable que se ira actualizando para almacenar la posible solucion

   int cont=0;         // Contador temporal de ocurrencias
   int index=-1;       // Indice de donde comienza la posible solucion

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

       if(A[j]==C){
           if(cont==0){    // Primera iteracion sobre una nueva ocurrencia de C, se acutaliza index
               index=j;
           }
           cont++;
       }
```

```
       else{
           if(cont>res.nOcu){  // Si la solucion actual es mejor que la que teniamos anteriormete se
actualiza la mejor solucion
               res.nOcu=cont;
               res.index=index;
           }
           cont=0;
       }

   }
   if (cont>0) {               // En caso de que la subcadena analizada termine con C y estemos ante
la mejor solucion, res no se actualizario
       res.nOcu=cont;          // por eso hacer falta actualizarlo aqui
       res.index=index;
   }
   return res;

}

// Función para combinar las soluciones y buscar una posible mejor solucion en la combinacion de
ambas subpartes
/*
//  string A -> string de entrada para anlizar
//  n -> longuid del string a analizar
//  m -> longuitud de la subcadena a buscar
//  C -> caracter que se desea buscar
//  X -> mejor solucion de la primera subparte
//  finX -> division de las dos subpartes
//  Z -> mejor solucion de la segunda subparte
*/
Data combinar(string A, int n, int m, char C, Data X, int finX, Data Z){

   // Seleccion de la mejor solucion de ambas partes, primer intento de mejor solucion
   Data firstAttempt;
   if (X.nOcu > Z.nOcu) {
       firstAttempt = X;
   } else {
       firstAttempt = Z;
   }



   // Busqueda de una posible mejor solucion en el centro de ambas partes

   // Si donde temina la primera parte o donde empieza la segunda parte no son C, o que estemos ya
ante la ultima combinacion, la mejor solucion es firstAttempt
   if (finX >= A.length() || A[finX-1]!=C || A[finX]!=C) {
       return firstAttempt;
   }
   else {          // Si la primera parte termina y la segunda parte empiezan en C se busca una
posible solucion

       // Avanzamos hacia atras hasta dejar de encontrar C
       // Se utiliza j como iterador que ira retrocediendo en la primera subparte
       int j=finX-1;

       // Contador de C consecutivas hacia atras en la primera parte
       int secondAttemptCont = 0;
       while(j>=0&&A[j]==C) {
           secondAttemptCont++;
           j--;
```

```
        }

        // Indice de inicio de la posible solucion
        int indexIniSecondAttempt=j+1;

        // Valoracion del resultado
        int necesaryC = firstAttempt.nOcu-secondAttemptCont;

        // En el caso de que las C consecutivas necesarias para una mejor solucion sea mayor que el
numero de ocurrencias de la mejor solucion
        // de dicha parte, no existiria una mejor solucion.
        if (necesaryC > Z.nOcu) {
            // En este punto no habria una mejor solucion que la que ya tenemos
            return firstAttempt;
        } else {    // En este punto puede haber una mejor solucion en la combinacion de las dos
subpartes
            // K como iterador para contar las C consecutivas desde la division de las dos subpartes
hacia delante
            int k = finX;
            while (k < A.length() && A[k] == C) {
                secondAttemptCont++;
                k++;
            }

            // Si la nueva cantidad de C consecutivas es mayor, actualizamos el resultado
            if (secondAttemptCont > firstAttempt.nOcu) {
                return {indexIniSecondAttempt,min(m,secondAttemptCont)};
            }
            return firstAttempt;
        }
    }


}


/*
//  string A -> string de entrada para anlizar
//  i -> indice inicio de analisis de A
//  n -> longuid del string a analizar
//  m -> longuitud de la subcadena a buscar
//  C -> caracter que se desea buscar
*/
Data DyV_algorithm(string A, int i, int n, int m, char C){

    if(m<1){
        exit(1);
    }
    if(n<=m){       // Solucion directa
        return contarCcons(A,i,n,C);
    }
    else{           // Recursividad
        int div = n/2;
        return combinar(A,n,m,C,DyV_algorithm(A,i,div,m,C),i+div,DyV_algorithm(A,i+div,n-div,m,C));
    }

}
```

```
// Este programa busca en la cadena A una subcadena de tamaño m donde aparzezcan en mayor numero de
caracteres C consecutivos
int main(int argc, char* argv[]){
   const int i = 0;                              // Indice que marca el inicio del String A
   const int m = 100;                             // Tamaño de las subcadenas solucion
   const char C = 'a';                           // Caracter que deber aparecer consecutivo

   long n = strtol(argv[1], nullptr, 10);

   string A = generarStringAleatorioConCaracteresLimitados(n,"abcdef");

   string B = "hooola";

                     // Longuitud de la cadena A

   Data result = DyV_algorithm(A,i,n,m,C);

   //printf("Inicio=%d\nN° apariciones=%d\n",result.index+1,result.nOcu);

}
```

5. Proceso de validación

Para el problema que hemos realizado, resulta ser mucho más eficiente la solución directa. Siendo así planteamos desde el inicio el problema de forma más simple, eficiente y directa y después lo convertimos en un algoritmo DyV. En consecuencia ha sido sencillo diseñar un algoritmo de solución sin DyV ya que solo ha sido necesario modificar un poco el código. Gracias a esto hemos podido comprobar que el algoritmo devuelve la solución correcta, aunque en caso de empate, elegirá la opción que más cerca del final de la cadena se encuentre.

Para poder validar que nuestro algoritmo funciona correctamente lo que hemos hecho ha sido apoyarnos en

```
Data solucionDirecta(string A, char C){
   Data res = {-1,0};

   int cont = 0;
   int index = -1;

   for(int j=0;j<A.length();j++){
       if(A[j]==C){
           if(cont==0){
               index=j;
           }
           cont++;
       }
       else{
           if(cont>res.nOcu){
               res.nOcu=cont;
               res.index=index;
           }
           cont=0;
       }

   }
   if (cont>0) {
       res.nOcu=cont;
       res.index=index;
   }
   return res;
}
```

6. Estudio experimental

Para llevar a cabo este estudio, hemos desarrollado una clase ajena a la principal en la que definimos dos funciones: “generarStringAleatorioConRistra” y “generarStringAleatorioConCaracteresLimitados” que generan un string de tamaño n, el primero fuerza una ristra de caracteres iguales en una posición dada y el segundo, en lugar de contar con todo el abecedario, cuenta con caracteres que el usuario desee. Además, ya que las funciones devuelven strings, estos se pueden concatenar para formar casos de prueba más complejos.

Además, con un script en python (que llama a la función de generar string aleatorios) y otro en bash (que ejecuta el programa sobre el archivo creado), hemos podido generar y dar a probar al algoritmo cientos de casos de prueba de manera automática.

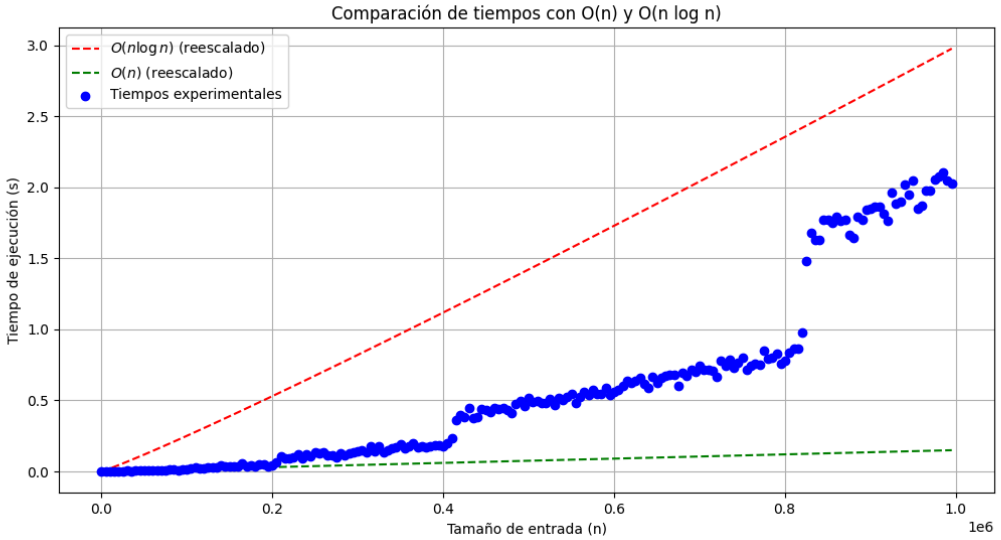
Concatenando los string devueltos por esta función y gracias a los scripts de automatización, hemos podido generar suficientes casos de prueba como para probar el satisfactorio funcionamiento de nuestro algoritmo y darle a probar casos más y menos favorables con los siguientes resultados:

```
string generarStringAleatorioConRistra(int longitud, char caracter, int
posicion, int numeroDeRepeticiones) {

   const string caracteres =
"ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789"; // se puede
lograr lo mismo con char c = (char)random(123); pero asi queda mas claro
   srand(time(nullptr));

   string resultado;
   resultado.reserve(longitud);

   for (int i = 0; i < longitud; ++i) {
       if (i >= posicion && i < (posicion + numeroDeRepeticiones)){
           resultado += caracter;
       }
       else {
           resultado += caracteres[rand() % caracteres.size()];
       }
   }

   return resultado;
}
```

```
string generarStringAleatorioConCaracteresLimitados(int longitud, string
caracteres) {

   srand(time(nullptr));
```

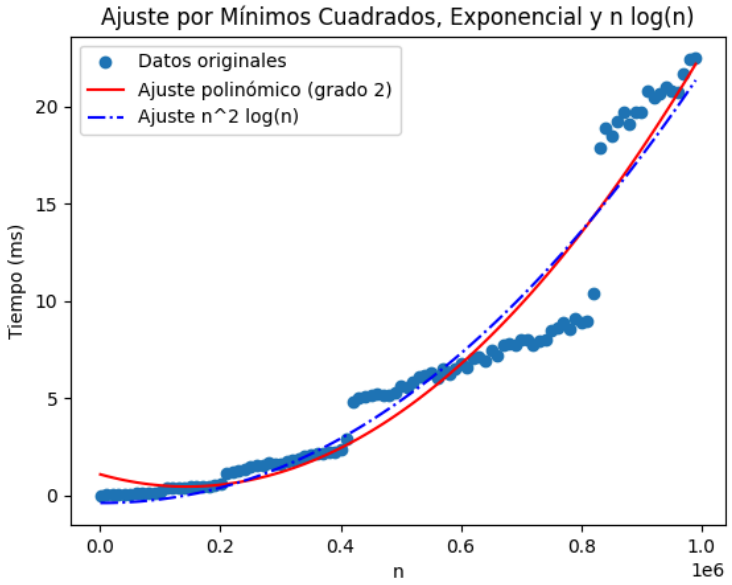
```
    string resultado;
    resultado.reserve(longitud);

    for (int i = 0; i < longitud; ++i) {
        resultado += caracteres[rand() % caracteres.size()];
    }

    return resultado;
}
```

La siguiente tabla de caso promedio para tamaño de entrada de 10e6 compara un caso promedio con los órdenes O(n) y O(nlogn). Al plantear el problema teóricamente, el peor caso tendría un orden de **nlog n**, y en esta primera prueba parecíamos estar en lo correcto. Además podemos observar que el caso promedio tiene un par de escalones pronunciados, en concreto en los tamaños de entrada de 0.2, 0.4 y 0.8. Esto se debe al número de llamadas recursivas necesarias, que aumenta exponencialmente para poder resolver el algoritmo.



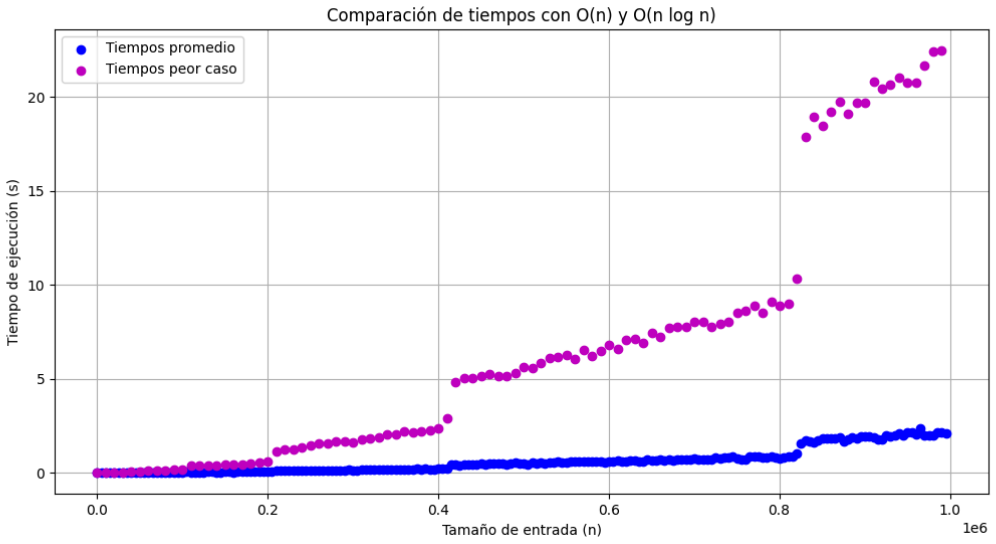
En la tabla de comparación de peor con caso promedio podemos comprobar que, aunque habíamos planteado y calculado el peor caso como **n * log n**, al llevarlo a la práctica es mucho más cercano a **n^2 * log n.**

(tabla de la aproximación mediante mínimos cuadrados, de los tiempos para peor caso a f(n)= n^2 log(n))



Además, podemos observar que cuando aumenta el número de llamadas recursivas hasta cierto punto, los tiempos dan unos saltos exponenciales muy notorios, por la misma razón que en la tabla previa.

(tabla de comparación del peor caso con caso promedio)



7. Contraste entre estudio teorico y practico

Como se ha comentado en apartados anteriores, en teoría nuestro algoritmo tiene un mejor caso casi constante, ya que en caso de ser m = n se ejecuta directamente sin recurrencia.
En lo referente al peor caso, al plantearlo teóricamente los cálculos indican que es de O(nlog n), sin embargo la realidad es que es mucho más cercano al O(n^2 log n).
Esto se puede deber al hecho de que, al aumentar los casos de prueba, también se va aumentando el número de veces que el algoritmo hace uso de la recurrencia exponencialmente.
En los primeros casos es difícil por no decir imposible darse cuenta de esto debido a la pequeña varianza de los casos de prueba fruto de la aleatoriedad de la entrada, pero conforme más se agranda el tamaño de la entrada mayor se vuelve la inclinación de la curva.

8. Conclusiones

Como conclusión a la hora del análisis de los algortimos creemos que es difícil clavar el orden de complejidad en una estudio teórico con luego su orden real de ejecución ya que a la hora de ejecutarse hay muchas otras variables que entran en juego, como el simple paso de parámetros en C++.
En nuestro caso hemos optado por usar referencias a string ya que si no en cada llamada recursiva se estaría copiando el string completo y eso aumentaria mucho el tiempo de ejecución y el uso de memoria, enmascarando el verdadero orden de complejidad, igual pasa con otras muchas instrucciones las cuales se consideran constantes pero en realidad cuando se ejecutan miles o millones de veces si acaban influyendo en el tiempo de ejecución.

En general ha sido un trabajo bastante agradable de realizar, las explicaciones teóricas en clase y las dudas que nos han ido surgiendo han sido muy satisfactorias y nos han ayudado a llevar el trabajo al dia.
Ha sido una práctica interesante pero consideramos que sería aún mejor realizar un problema que si se vea positivamente afectado con la aplicación del algoritmo divide y vencerás porque, por lo menos para nosotros, el problema era mucho más sencillo y eficiente de realizar sin usar el algoritmo y se nos ha hecho un poco raro estar pensando todo el rato una forma más eficaz que no podíamos aplicar.

En general hemos terminado el trabajo sintiéndonos realizados y con ganas de seguir aprendiendo.