DIVIDE y VENCERÁS

EJERCICIO 11

INTEGRANTES :

Pablo Gómez Alcaraz
Grupo 3.1

Alberto Zapata Mira
Grupo 3.1

Profesor : Norberto Marin Perez

Abril 2024. Curso 2023/2024

ÍNDICE

ÍNDICE **2**

PSEUDOCÓDIGO **3**

EXPLICACIÓN DEL ALGORITMO **5**

ESTUDIO TEÓRICO **5**

ALGORITMO **7**

VALIDACIÓN **11**

ESTUDIO EXPERIMENTAL **12**
- Pruebas caso Mejor 12
- Prueba caso Peor 12
- Prueba caso Promedio 13
- Tabla y Gráfico 14

CONTRASTE DE ESTUDIOS **15**

CONCLUSIÓN **16**

PSEUDOCÓDIGO

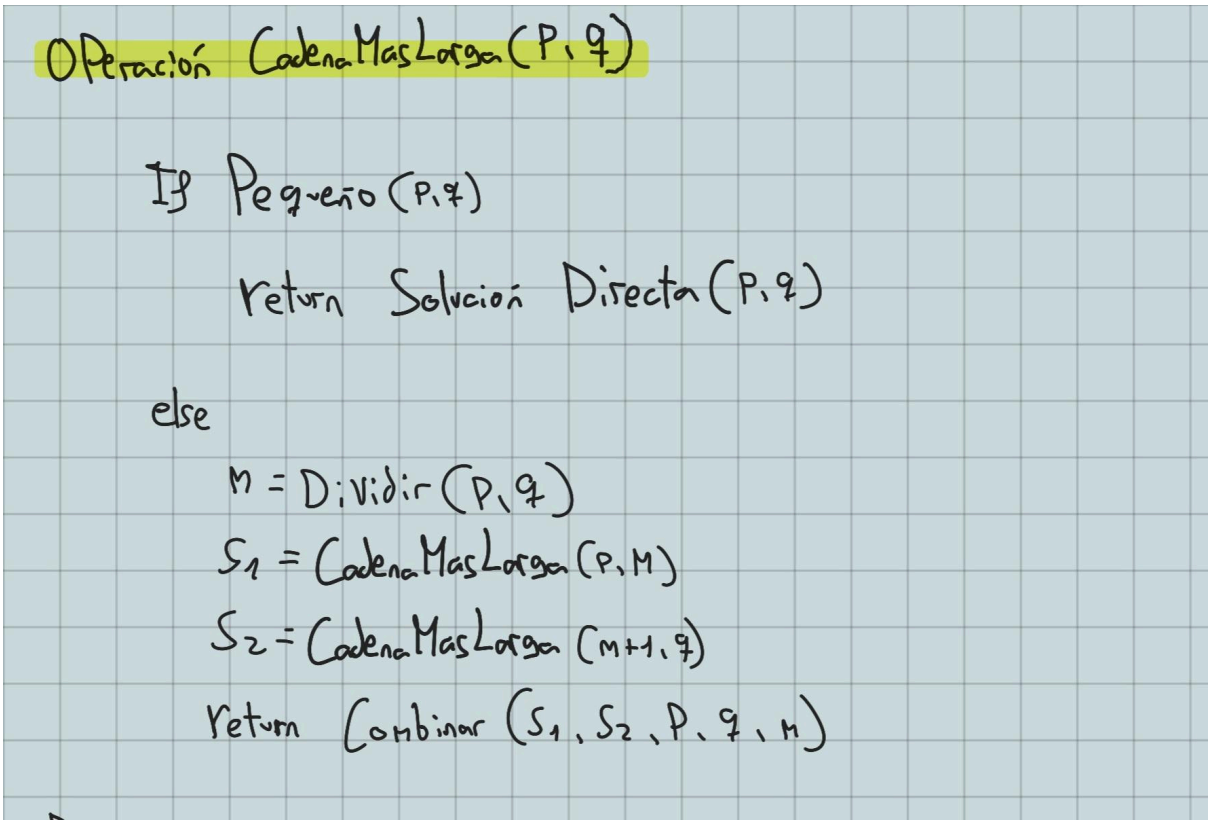
```
Estructura Solucion
     entero p:= 0
     entero num_caracteres:= 0
FinEstructura


booleano Pequeño (inicio,fin:entero)
    devolver(fin-inicio+1)<=1000



Solucion SolucionDirecta (c:cadena,inicio,fin:entero)
    entero mejorInicio,mejorLongitud,inicioActual
    mejorInicio:=inicio
    mejorLongitud:=0
    inicioActual:=inicio

    para i desde inicio hasta fin hacer
        si i+1<=fin y abs(c[i]-c[i+1])<= 3 entonces
            continuar
        sino
            longitudActual:=i-inicioActual+1
            si longitudActual>mejorLongitud entonces
                mejorInicio:=inicioActual
                mejorLongitud:=longitudActual
            finsi
            inicioActual:=i+1
        finsi
    finpara

    solucion.p:=mejorInicio
    solucion.num_caracteres:=mejorLongitud
    devolver solucion



Solucion BusquedaCentral (c:cadena,inicio,fin,m:entero)
    entero mejorInicio,longitud,indiceActual
    booleano sigue
    mejorInicio:=m
    longitud:=1
    indiceActual:=m
    sigue:=verdadero
    para i desde m hasta fin y sigue hacer
        si i+1<=fin y abs(c[i]-c[i+1])<=3 entonces
```

```
            longitud:=longitud + 1
        sino
            sigue:=falso
        finsi
    finpara

    sigue:=verdadero

    para i desde m hasta inicio y sigue hacer
        si i-1 >=inicio y abs(c[i]-c[i-1])<=3 entonces
            longitud:=longitud+1
            mejorInicio:=mejorInicio-1
        sino
            sigue:=falso
        finsi
    finpara

    solucion.p:=mejorInicio
    solucion.num_caracteres:=longitud
    devolver solucion



Solucion MaxCadenas (s1, s2, central: Solucion)
    si s1.num_caracteres>=s2.num_caracteres entonces
        si s1.num_caracteres>=central.num_caracteres entonces
            devolver s1
        sino
            devolver central
        finsi
    sino
        si s2.num_caracteres>=central.num_caracteres entonces
            devolver s2
        sino
            devolver central
        finsi
    finsi



Solucion Combinar (s1, s2: Solucion, c: cadena, m: entero)
    Solucion central
    central:=BusquedaCentral(c,s1.p,s2.p+s2.num_caracteres-1,m)
    devolver MaxCadenas(s1,s2,central)



Solucion CadenaMasLarga (c:cadena,inicio,fin:entero)
```

```
si Pequeño(inicio,fin) entonces
    devolver SolucionDirecta(c, inicio, fin)
sino
    m:=(inicio+fin)/2
    s1:=CadenaMasLarga(c,inicio,m)
    s2:=CadenaMasLarga(c,m+1,fin)
    devolver Combinar(s1,s2,c,m)
finsi
```

EXPLICACIÓN DEL ALGORITMO

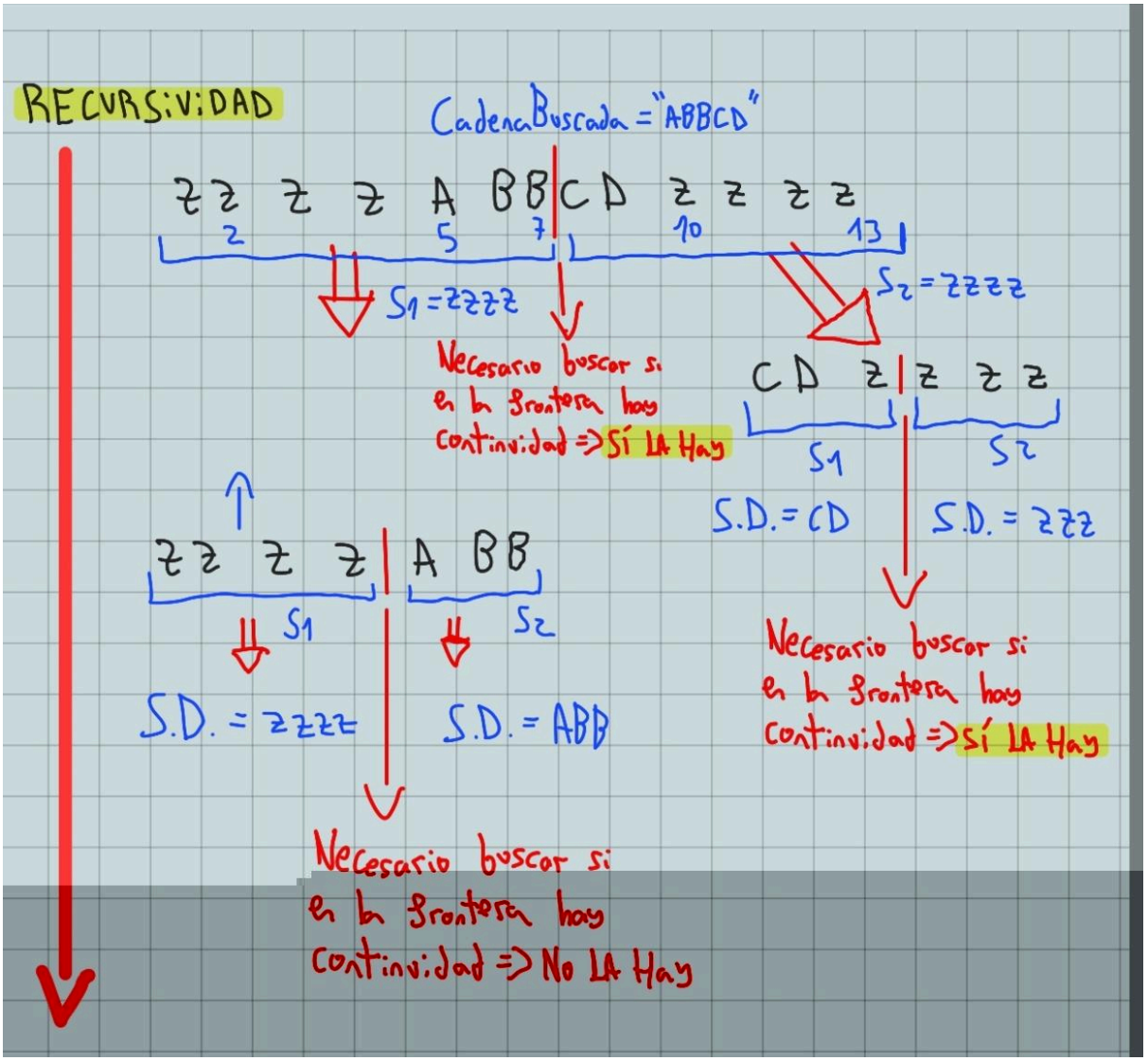
Desde un inicio, planteamos el problema mediante el esquema básico de divide y vencerás que nos proporcionaban los apuntes:



Siguiendo este esquema, y el desarrollo por separado de cada función, nuestro primer error fue un error en la forma de plantear el problema. Por cómo es nuestro enunciado, pensábamos que lo óptimo sería que el programa dividiese la cadena en dos subcadenas, estas subcadenas pasarlas cada una por separado a la misma función CadenaMasLarga, hasta que llegase al tamaño considerado pequeño. Este primer planteamiento no estaba mal del todo, pero tenía el error de que trabajabamos con trozos de la cadena principal, lo que nos dificultó el desarrollo del programa a tal nivel de no saber continuarlo. Revisando los apuntes y hablando con otros compañeros, nos comentaron que ellos trabajaban con índices, por lo que reestructuramos el programa para trabajar con índices. En un principio usamos dos variables globales, una para el inicio de la subcadena solución y otra para la longitud de la misma. Pero finalmente nos decantamos por usar un struct conformado por estas dos variables, lo cual nos

facilitó mucho el continuar con el programa, y plantearlo de una mejor manera. Desde ahí, pensamos que lo mejor sería hacer todas las funciones del tipo struct, que en nuestro caso lo llamamos Solucion, para que así operasen con los índices y la longitud de estos, y cuando la función terminase que lo devuelva en forma de un struct Solucion.

La parte más importante a la hora de plantear y solucionar el problema, fue a la hora de tratar los casos en los que la subcadena solución tiene caracteres a la izquierda y a la derecha de la cadena original, es decir, tiene continuidad en la frontera. No fuimos conscientes de este problema hasta que probamos de manera práctica con nuestro código primigenio. Pensamos, que una manera de solucionar este problema, era que la función combinar comprobase si el último carácter de la subcadena de la izquierda tenía continuidad con el primer carácter de la subcadena de la derecha. Planteamiento que no era del todo erróneo, pero que seguía dando errores. Después de probar con la cadena : "zzzzabbcdzzzz", cadena escogida a propósito, dado que tiene la cadena más larga en la frontera, llegamos a una conclusión:



Aclaración: en este esquema, tomamos como pequeño, tamaños menores o iguales a 4 (en el programa final pusimos 1000 dado que era un tamaño que daba tiempos pequeños para cadenas con 10^4<=n<=n^6).

Dado el esquema de la imagen, concluimos que era necesario que la función combinar trabajase con tres cadenas: la subcadena más larga de la izquierda, la subcadena más larga de la derecha, y la subcadena más larga en la frontera. La función combinar, entonces, devolvería la subcadena con un *solucion.num_caracteres* más grande. Para poder realizar esto, la función combinar se tenía que ayudar en: la propia solución directa, y una función que llamamos *busqueda_central*, que, partiendo desde la mitad de la cadena, avanza de derecha a izquierda hasta encontrar un caracter que no cumpla la condición del ejercicio, es decir, comprobar si la subcadena más larga que cumple la condición pedida está en la frontera. Con este último planteamiento del ejercicio fue todo ponerse a programar e ir comprobando que el programa realizaba correctamente lo que debía.

ESTUDIO TEÓRICO

Función pequeño.
Cuenta solo una instrucción,O(1).

Función SolucionDirecta.
Depende de la entrada, ya que tenemos un bucle que va desde el principio de la cadena hasta el final de esta. Podemos concluir que el mejor caso es cuando la cadena tiene continuidad, ya que haría un menor número de instrucciones, y por lo tanto el peor caso sería cuando la cadena no tiene continuidad y además es la subcadena más larga hasta el momento, ya que entraría en el *else* del *for* , que te obliga a hacer un *if* , el cual daría verdadero, y estaría obligado a hacer 2 instrucciones más.

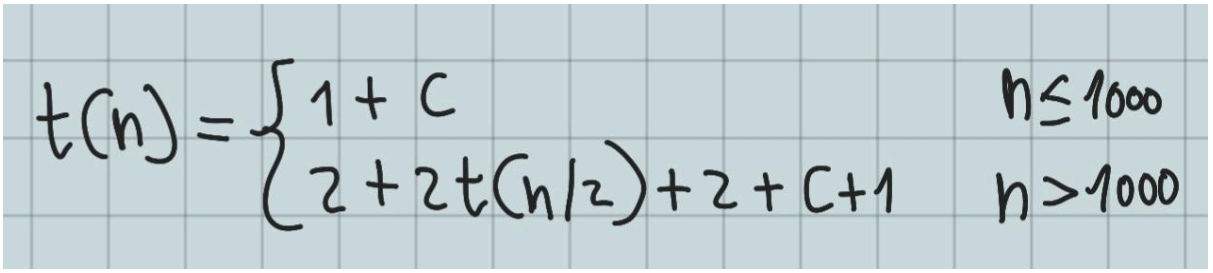
Funcion busqueda_central.
Esta función tiene como mejor caso que no tenga que extender hacia ningún lado (izquieda o derecha), ya que así el booleano sigue sería falso, y saldría inmediatamente de los dos bucles. El peor caso es que la cadena de entrada tenga continuidad tanto por la izquierda, tanto como por la derecha, como por ejemplo, la cadena “abcdefghi”. Por lo que podemos concluir que esta función depende de la cadena de entrada. El orden de magnitud es O(n), siendo n la longitud de la cadena de entrada, y Omega(l), dado que en el mejor caso, los dos bucles harían una sola iteración.

Función maxCadenas.
Esta función está compuesta por distintos condicionales, por lo que sería de orden O(l) y Omega(l).
Función combinar.
Esta función tiene una llamada a maxCadenas, lo cual nos da igual, dado que seguiría manteniendo un orden O(l) y Omega(l), es decir, orden constante.

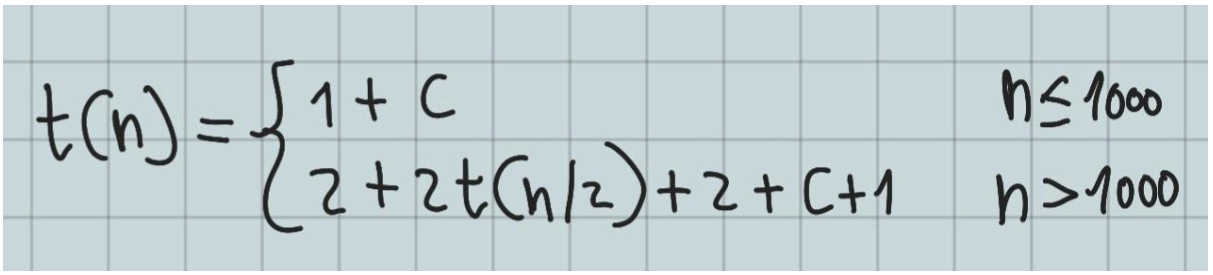
Función CadenaMasLarga.

El caso mejor y peor depende de la cadena de entrada. Si la cadena de entrada cumple con la condición para ser considerada como pequeña, entonces sería el mejor caso. En el peor caso, la función entra al else, ya que no es considerada como pequeña, y se encuentra con dos llamadas recursivas dividiendo el problema por la mitad (divide y vencerás). Por lo tanto tenemos dos instrucciones de orden constante, más dos llamadas recursivas del estilo t(n/2). Las funciones del tiempo de ejecución serían:

Tiempo del mejor caso:



Tiempo del peor caso:



Por lo que concluimos que, dado que la función CadenaMasLarga divide la cadena de entrada a la mitad, la complejidad es logarítmica en base 2, es decir, el algoritmo hace aproximadamente log2(n) llamadas recursivas, donde n es la longitud de la cadena de entrada. Cada llamada recursiva tarda O(l) tiempo en dividir la cadena, y O(n) tiempo para encontrar y combinar las soluciones. Por lo que el orden de complejidad sería O(n*log n), y como las funciones hemos comprobado manualmente que son iguales para mejor y peor caso, tendríamos O(n*log n) y Omega(n*log n).

ALGORITMO

```
#include <iostream>
#include <string>
using namespace std;

struct Solucion{
    int p=0;
    int num_caracteres=0;
}

bool pequeno(int inicio,int fin){
    // Acordamos que un tamaño pequeño en este caso se refiere a cadenas de 4 o menos caracteres
    return ((fin-inicio+1)<=1000); //m = n/1000, ir cambiando
}

Solucion SolucionDirecta(const string c, int inicio, int fin) {
//Esta función se corresponde a la función básica del esquema algorítmico como la solución directa. Lo que hace es, dada una cadena, devuelve la subcadena más larga.

    int mejor_inicio = inicio;  //Empezamos desde el índice inicial dado
    int mejor_longitud = 0;     //En un principio, la mejor longitud es de 0
    int inicio_actual = inicio; //Conforme avance el bucle irá cambiando su valor
    Solucion solucion;

    for (int i = inicio; i <= fin; ++i) {

        if (i + 1 <= fin && abs(static_cast<int>(c[i]) - static_cast<int>(c[i + 1])) <= 3) {
            // Continua hasta que encuentre un carácter que no cumpla la condición
        } else {
            int longitud_actual = i - inicio_actual + 1; //Guardamos la longitud
            if (longitud_actual > mejor_longitud) {
                mejor_inicio = inicio_actual;
                mejor_longitud = longitud_actual;
            }

            inicio_actual = i + 1;  //Dado que hemos registrado la longitud de una subcadena candidata, reiniciamos el inicio
```

```
        }

    }

    solucion.p = mejor_inicio; // Actualizamos los valores del
struct solución
    solucion.num_caracteres = mejor_longitud;

    return solucion; // Devolvemos el índice inicial y la
longitud de la subcadena más larga de la cadena c
}

Solucion busqueda_central(const string c, int inicio, int fin,int
m){
    int mejor_inicio=m;  //Índice del inicio de la subcadena
central
    int longitud=1; //Longitud de la subcadena central
    int indice_actual=m;
    bool sigue=true; //Variable booleana, representa la
continuidad en la frontera
    Solucion solucion;

    for(int i=m;i<=fin&&sigue;i++){   //Se expande hacia la
derecha y cuando la condición es falsa para de comprobar
        if(i+1<=fin && abs(static_cast<int>(c[i]) -
static_cast<int>(c[i+1])) <= 3){
            longitud++;
        }else{
            sigue=false;
        }
    }
    sigue=true;
    for(int i=m;i>=inicio&&sigue;i--){//Se expande hacia la
izquierda y cuando la condición es falsa para de comprobar
        if(i-1 >=inicio && abs(static_cast<int>(c[i]) -
static_cast<int>(c[i-1])) <= 3){
            longitud++;
            mejor_inicio--;
        }else{
            sigue=false;
        }

    }

    solucion.p=mejor_inicio;
    solucion.num_caracteres=longitud;
    return solucion;
```

```
}


Solucion maxCadenas(Solucion s1,Solucion s2, Solucion central){
```
//Esta función recibe la solucion derecha, la solucion izquierda y la central (obtenida de busqueda_central), y devuelve la cadena más larga de las tres
```
    if(s1.num_caracteres>=s2.num_caracteres){
        if(s1.num_caracteres>=central.num_caracteres){
            return s1;
        }
        return central;
    }else{
        if(s2.num_caracteres>=central.num_caracteres){
            return s2;
        }else{
            return central;
        }
    }
}

Solucion combinar(Solucion s1,Solucion s2,const string& c,int m)
```
//Esta función se corresponde a la función básica del esquema algorítmico como combinar. En este caso, no es un combinar "al uso" , su funcionamiento es simple, obtiene la subcadena del centro de la cadena, y devuelve la mayor.
```
{
Solucion central=busqueda_central(c,s1.p,s2.p+s2.num_caracteres-1,m);
```
//Subcadena central
```

    return(maxCadenas(s1,s2,central));
```
//La llamo maxCadenas por si acaso se confunde con la función max que ya tiene c++
```


}

Solucion CadenaMasLarga(const string c,int inicio,int fin)
```
//Esta función responde a la función básica del esquema algorítmico como pequeño y dividir. Determina cuando el problema es suficientemente pequeño como para poder utilizar la solución directa y cuando hay que continuar dividiendo el problema

//Trabajamos con índices
```
{
    if (pequeno(inicio,fin)){
```

```
        return SolucionDirecta(c,inicio,fin);
    }
    else{

        int m = (inicio+fin)/2; // Punto medio de la cadena

        Solucion s1 = CadenaMasLarga(c,inicio,m); //Subcadena
izquierda
        Solucion s2 = CadenaMasLarga(c,m+1,fin); //Subcadena
derecha
        return (combinar(s1,s2,c,m));
    }
}

int main(){

    string c;
    cout << "Introduce una cadena de caracteres:";
    cin >> c;
    Solucion cadenaBuscada = CadenaMasLarga(c,0,c.length()-1);
    cout << "La solucion es: "
<<c.substr(cadenaBuscada.p,cadenaBuscada.num_caracteres)<< endl;
    cout << "La posicion es " <<cadenaBuscada.p+1<< " y la
longitud es " <<cadenaBuscada.num_caracteres<< "." << endl;
    return 0;
}
```

VALIDACIÓN

Para validar el algoritmo, se sometió a distintas comprobaciones.

La primera comprobación fue la propia del ejercicio :

C =abcehfeksrtzyx

```
Introduce una cadena de caracteres:abcehfeksrtzyx
La solucion es: abcehfe
La posicion es 1 y la longitud es 7.
```

Luego se ha ido probando con casos que podrían hacer fallar al algoritmo

En este caso, aunque pueda parecer no tener mucho sentido, esta prueba pretende probar que el algoritmo devolvería la cadena completa si se le daba ordenada en orden alfabético :

C=abcdefg

```
Introduce una cadena de caracteres:abcdefg
La solucion es: abcdefg
La posicion es 1 y la longitud es 7.
```

Este comprueba que en caso de que todos los caracteres presenten una diferencia mayor a tres, devuelva la primera subcadena de mayor longitud, es decir, la primera letra :

C=zgkahpwblcre

```
Introduce una cadena de caracteres:zgkahpwblcre
La solucion es: z
La posicion es 1 y la longitud es 1.
```

Este caso comprueba que trate bien la continuidad en la frontera :

C=zzzzabbcdzzzz

```
Introduce una cadena de caracteres:zzzzabbcdzzzz
La solucion es: abbcd
La posicion es 5 y la longitud es 5.
```

ESTUDIO EXPERIMENTAL

Pruebas caso Mejor

En este caso, generamos una cadena que consistiera un letras aleatorias, que tuvieran una diferencia de valor de más de tres, seguidas por la cadena abcd, es decir, que la solución esté es un extremo de la cadena :

```
#include <iostream>
#include <cstdlib>
#include <ctime>

using namespace std;

char generarLetra(){
    return 'a' + rand() % 26;
}

int main(){
    srand(time(0));

    char letra_anterior = generarLetra();
    cout << letra_anterior;

    for (int i = 1; i < 100000; ++i){ //cambiar aquí el tamaño de la prueba
        char letra_nueva;
        do{
            letra_nueva=generarLetra();
        } while(abs(letra_nueva-letra_anterior)<=3);

        cout <<letra_nueva;
        letra_anterior=letra_nueva;
    }
    cout <<"abcd";
    cout<<endl;
    return 0;
}
```

Prueba caso Peor

En este caso decidimos concatenar la cadena abcd, para hacer un generador de cadenas que tengan continuidad del tamaño de la propia cadena,es decir, que la función frontera se extienda hasta ambos extremos:

```
#include <iostream>
#include <cstdlib>
```

```
#include <ctime>

using namespace std;

int main() {

    string letra_anterior="abcd";
    int lDeseada=10000; //cambiar aquí el tamaño de la prueba
    int lActual=0;

   while(lActual<lDeseada){
    cout<<letra_anterior;
    lActual+=letra_anterior.length();
   }

    cout<<endl;
    return 0;
}
```

Prueba caso Promedio

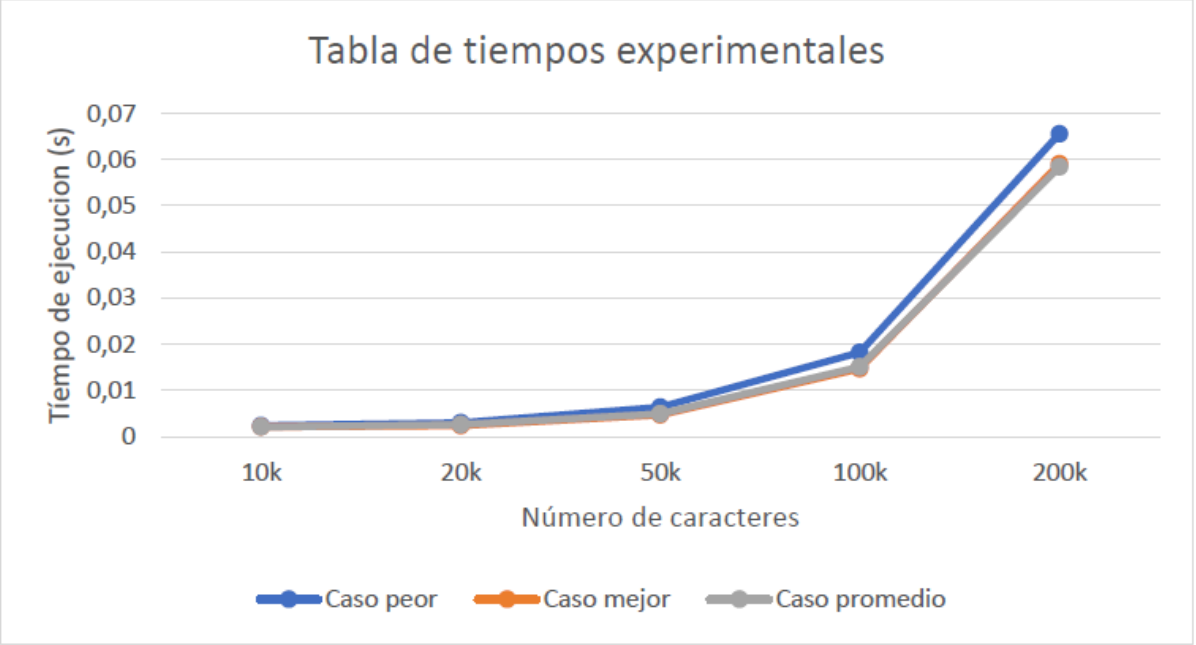
Para el caso promedio decidimos que fuera un archivo con letras completamente aleatorias de la 'a' a la 'z' :

```
#include <iostream>
#include <cstdlib>
#include <ctime>

using namespace std;

int main(){
    srand(time(0));
    const int num_letras=10000; //cambiar aquí el tamaño de la
prueba

    for(int i=0;i<num_letras;i++){
        char letra='a'+(rand()%26);
        cout<<letra;
    }

    return 0;
}
```

Tabla y Gráfico

A continuación se aporta una tabla con los tiempos de ejecución en segundos, y un gráfico representando estos tiempos :

Tamaño	Caso peor	Caso mejor	Caso promedio
10k	0,0024	0,0021	0,0021
20k	0,003	0,0024	0,0026
50k	0,0064	0,0047	0,0049
100k	0,0182	0,0147	0,0151
200k	0,0656	0,0591	0,0584



Anotacion. Con “10k” o “200k”, nos referimos a 10.000 y 200.000 caracteres de entrada, respectivamente.

CONTRASTE DE ESTUDIOS

Los estudios experimentales encajan a la perfección con los teóricos, dado que era de esperar, que tanto el mejor, peor, y caso promedio, iban a dar tiempos muy cercanos, dado que en algoritmo no hay ni mejor ni peor caso. También se puede vislumbrar que, la gráfica de cara a los 20.000 caracteres de entrada, empieza a crecer logarítmicamente, lo que confirma los estudios teóricos de que el algoritmo tiene orden exacto de complejidad O(n*log n).

CONCLUSIÓN

Esta práctica nos ha servido para comprender en profundidad la forma en que trabaja un algoritmo del tipo divide y vencerás. La mayor complicación fue la parte de la continuidad en la frontera, dado que es una parte crucial del algoritmo, y en nuestro caso nos costó mucho ver que debíamos enfocar hacia ese lado el problema. Por otra parte, hemos afianzado más nuestros conocimientos de C++, y nuestra lógica en general. Hemos tardado en realizar la tarea unas 20 horas aproximadamente. Lo más difícil fue enfocar bien el problema, realizar bien la solución directa, y sin duda, tratar los casos donde hay continuidad en la frontera.