Algoritmos y Estructuras de Datos II

Práctica DyV

Curso 2023-2024

Grupo de prácticas: [2.*3*]
Profesor: Jesús Sánchez Cuadrado
N.º de problema: 3

Autores:
Ortiz García, Juan Jesús | 24420563Z
Vicente Pérez, Gonzalo | 24420789X

Índice

1. Pseudocódigo y explicación del algoritmo

1.1. Tipos y estructuras de datos

Definimos un tipo Resultado, el cual está compuesto de dos variables de tipo entero: inicio y tamaño, que contienen la posición de inicio de la subcadena solución y el tamaño de esta, respectivamente. A la hora de devolver la solución del problema, usaremos este tipo de datos.

```
struct Resultado {
    int inicio, int tam;
};
```

1.2. Método DivideVenceras

Este método contiene la base del código, que hemos realizado conforme al algoritmo de Divide y Vencerás visto en clase de teoría:

```
DivideVenceras (p, q, m: int; A: string) : Resultado {
var mid: int
var S1, S2: Resultado
        si Pequeño (p, q, m) entonces
                return SolucionDirecta (A, p, q);
        sino:
                mid := Dividir (p, q)
                S1 := DivideVenceras (A, p, mid, m);
                S2 := DivideVenceras (A, mid+1, q, m);
                return Combinar (S1, S2, A, p, q, m);
}
```

El método recibe la cadena junto con los índices que delimitan a la misma (“p” y “q”) y un entero m correspondiente al tamaño máximo que queremos que tenga la subcadena ascendente que buscamos en A.

A continuación, definimos una variable “mid” que albergará el índice del punto medio de la cadena pasada; y un par de variables “S1” y “S2” de tipo Resultado.

En caso de que la solución sea lo suficientemente pequeña (función Pequeño), devolvemos la solución directa del problema (función SoluciónDirecta).

En caso contrario, llamamos a la función Dividir para establecer el punto medio de la cadena y llamamos recursivamente al método principal con dos trozos: S1 (la mitad izquierda) y S2 (la mitad derecha). Finalmente, con la función Combinar, devolvemos el resultado de la combinación de S1 y S2.

A continuación se explican con detalles la implementación de cada uno de los métodos mencionados previamente.

1.3. Método Pequeño

Este método se va a encargar de dictaminar si el trozo de cadena que se pasa es lo suficientemente pequeño como para devolver la solución directamente. Es de tipo booleano, devolviendo true en caso de ser el trozo de cadena lo suficientemente pequeño.

```
Pequeño (p, q, m: int) : bool {
        si ((q - p + 1) <= m) entonces
                return true;
        sino:
                return false;
}
```

Como podemos observar, se reciben p y q (que indican los índices donde empieza y termina la cadena/subcadena) y m (tamaño máximo permitido de la subcadena ascendente buscada).

Nuestro criterio para determinar si el trozo a estudiar es lo suficientemente pequeño es que dicho trozo sea más pequeño que el máximo permitido de la subcadena (es decir, m). Para calcularlo, simplemente restamos los índices y comparamos con el valor m.

1.4. Método SolucionDirecta

El método SolucionDirecta va a proporcionarnos la subcadena ascendente más larga de la cadena que le pasemos como argumento. Como se ha mencionado previamente, accederemos a este método únicamente cuando el trozo a estudiar sea lo suficientemente pequeño.

```
SolucionDirecta(p, q: int; A: string) : Resultado {
    var result := {p, 1} : Resultado;
    var longitudAct := 1 : int;

    para (int i = p + 1) hasta (i <= q)
        si (A[i] >= A[i - 1])
            longitudAct++
```

```
        sino:
            si (longitudAct > result.tam)
                result = {i - longitudAct, longitudAct};
            longitudAct = 1;
            finsi;
        finsi;

    finpara;

    si (longitudAct > result.tam)
        result = {q - longitudAct + 1, longitudAct};

    return result;
}
```

Este método recibe como parámetros los índices de la cadena/subcadena y la propia cadena. A continuación, se define la variable “result”, de tipo Resultado, donde se almacenará el resultado que devolveremos; y la variable “longitudAct”, de tipo entero, en la que almacenaremos la longitud que vaya teniendo la subcadena según vaya avanzando el algoritmo. Estas variables están inicializadas a {p, 1} y 1 respectivamente. Esto se hace para que en caso de no encontrar ninguna subcadena ascendente, el método devuelva por defecto la primera letra de la cadena pasada (que al ser un solo carácter, es por tanto de tamaño 1).

Posteriormente, se inicia un bucle que recorre letra a letra la cadena pasada (teniendo en cuenta los índices que la delimitan), y en caso de que la letra anterior a la estudiada sea menor, se aumenta la longitud. En caso contrario, si la longitud actual es mayor al tamaño almacenado en la variable “resultado”, almacenaremos la longitud actual sustituyendo a la que hubiese previamente en “resultado.tam”. Una vez hecho esto, reiniciamos la longitud actual (la reiniciaremos porque esta acción se efectúa cuando se ha roto la racha de caracteres ascendentes consecutivos).

Finalmente, ya fuera del bucle e inmediatamente antes de devolver la solución, puede darse el caso de que la subcadena más larga termine en el último carácter. Para cubrirnos de posibles fallos, comprobamos si la longitud actual es mayor al tamaño almacenado en resultado, y en dicho caso, actualizamos resultado con los valores correctos. Ahora sí, devolvemos “result”.

1.5. Método Dividir

```
Dividir(p, q: int) : int {
    return (p + q) / 2;
}
```

Es un método muy sencillo, encargado de dividir el problema en trozos más pequeños para ser tratado más fácilmente. En este caso, tomamos los índices que se le pasen al método, los sumamos y los dividimos entre dos. De este modo, obtendremos el punto medio de la cadena. Cabe destacar que nos quedamos con la parte entera de la división.

1.6. Método Combinar

El método Combinar es el más complejo de todos, y trata con dos variables de tipo Resultado, analizando las posibles combinaciones de subcadenas ascendentes más largas que puedan darse del junte de los dos trozos de cadenas pasados al método.

```
Combinar(S1, S2: Resultado; p, q, m: int; A: string) : Resultado {

si (S1.tam == m || S2.tam == m)
    si (S1.tam >= S2.tam)
        return S1
    sino
        return S2
    finsi
finsi

si (A[mid] > A[mid+1])
    si (S1.tam >= S2.tam)
        return S1
    sino
        return S2
    finsi

sino
    si (S1.inicio + S1.tam - 1 == mid)
        si (S2.inicio == mid+1)
            var r: Resultado;
            Resultado r = {S1.inicio, (S1.tam + S2.tam)}
            si (r.tam > m)
                r.tam = m
            finsi
            return r
        finsi
```

```
        sino
            var r: Resultado;
            Resultado r = extenderDerecha(A, S1, q, m)
            si (r.tam >= S2.tam)
                return r
            sino
                return S2
            finsi
        finsi
    sino
        si (S2.inicio == mid+1)
            var r: Resultado;
            Resultado r = extenderIzquierda(A, S2, p, m)
            si (r.tam >= S1.tam)
                return r
            sino
                return S1
            finsi
        finsi
        sino
            var r, r1: Resultado;
            Resultado r = cadFrontera(A, p, q, mid, m)
              Resultado r1 = si (S1.tam >= S2.tam) entonces S1 sino S2
finsi
            return si (r.tam >= r1.tam) entonces r sino r1 finsi
        finsi
    finsi
finsi
```

El método recibe dos variables de tipo Resultado, “S1” y “S2” (las que va a combinar), la cadena en sí y sus índices “p” y “q”. Además, también se pasa la variable “mid” calculada con el método Dividir y la variable m que indica la longitud máxima de la subcadena ascendente.

En primer lugar, se comprueba si “S1” o “S2” ya tiene tamaño “m”. En dicho caso, devolvemos la variable correspondiente. Si se cumple esta condición, no es necesario buscar ninguna combinación más: el problema (o subproblema) ya está resuelto.

A continuación comprobamos si hay continuidad en la frontera, es decir, si se cumple que el último carácter de la primera parte de la cadena (“mid”) es menor que el primero de la segunda mitad (“mid+1”). En caso de no haberla, devolveremos la variable de tipo Resultado que contenga el “resultado.tam” más grande, pues si no hay continuidad no vamos a encontrar ninguna subcadena más grande que las ya presentes en sendas variables.

Ahora estudiaremos los posibles casos que pueden surgir de la existencia de continuidad en la frontera. Partiremos de dos casos base: que “S1” toque la frontera y que “S1” no la toque.

En caso de que “S1” toque la frontera, es decir, que la subcadena ascendente más larga acabe en la posición “mid”, podemos encontrarnos dos subcasos: que “S2” toque la frontera (que empiece en “mid”) o que no lo haga.

En el primer subcaso, la subcadena ascendente más larga será la resultante de concatenar “S1” y “S2”, por lo que en la solución devolveremos la posición de inicio de “S1” y la suma del tamaño de “S1” y “S2”. Podría existir la posibilidad de que la concatenación de estas subcadenas devuelva una cadena más larga que la permitida por la restricción “m”. En dicho caso, simplemente limitamos el tamaño a “m”.

En el segundo subcaso, deberemos comprobar si al extender “S1” a la derecha (con la función extenderDerecha, explicada más abajo) obtenemos una subcadena más larga que S2. Para ello, simplemente llamamos a la función y hacemos una comprobación de dicha condición.

Volviendo al otro caso base, en el que “S1” no toca la frontera, se nos vuelven a presentar dos subcasos: que “S2” toque la frontera y que “S2” no lo haga (y que, por tanto, ninguna de las variables toque la frontera).

En este nuevo primer subcaso, similar al segundo subcaso anterior, extenderemos la cadena, pero esta vez hacia la izquierda, con una función análoga a extenderDerecha llamada extenderIzquierda.

Por último, si ni “S1” ni “S2” tocan la frontera, debemos tener en cuenta que como hemos detectado continuidad en la frontera existe la posibilidad de que haya una subcadena más larga entre ambos trozos. Para comprobarlo, llamamos a la función cadFrontera, que explicaremos más abajo. Finalmente, devolvemos la cadena más larga de entre el retorno de cadFrontera, “S1” y “S2”.

1.7. Métodos extenderDerecha y extenderIzquierda

Los métodos que encontramos abajo se van a encargar de expandir la cadena en una dirección (izquierda o derecha), siempre teniendo en cuenta que la cadena que se está formando sea ascendente.

```
extenderIzquierda(A: string; S: Resultado; limite, m: int) : Resultado
{
    var inicio, cuentaTam: int
    inicio := S.inicio
    cuentaTam := S.tam
```

```
    mientras (inicio > limite Y A[inicio - 1] <= A[inicio] Y cuentaTam
<= m)    hacer
        inicio := inicio - 1
        cuentaTam := cuentaTam + 1
    finmientras
    var r: Resultado
    r := {inicio, cuentaTam}
    return r
}
```

```
extenderDerecha(A: string; S: Resultado; limite, m: int) : Resultado {
    var inicio, cuentaTam: int
    inicio := S.inicio
    cuentaTam := 1
    mientras (inicio < limite Y A[inicio] <= A[inicio + 1] Y cuentaTam
<= m) hacer
        inicio := inicio + 1
        cuentaTam := cuentaTam + 1
    finmientras
    var r: Resultado
    r := {S.inicio, cuentaTam}
    return r
}
```

Estos métodos reciben una posición a partir de la cual se quiere expandir una cadena, el límite de expansión, la restricción de tamaño “m” y la propia cadena.

En el caso de extenderIzquierda, se inicializan enteros “inicio” y “cuentaTam” con los valores inicio y tamaño, respectivamente, de la variable “S” de tipo Resultado que le pasamos al método. “cuentaTam” es una variable que usaremos para llevar cuenta del tamaño que surge en la cadena extendida. Mientras que el carácter inmediatamente anterior al indexado sea menor que este y la restricción de tamaño “m” se cumpla, decrementamos inicio y aumentamos el tamaño, devolviendo una variable “r” de tipo Resultado formada por “inicio” y “cuentaTam”.

La lógica es la misma para extenderDerecha, solo que “cuentaTam” estará ahora inicializado a 1, pues cuando extendemos a la derecha “S2” no toca la frontera, por lo que la manera más simple de encontrar el tamaño total de la cadena extendida es situarnos en el comienzo de la variable “S” que le pasamos e ir incrementando “cuentaTam” siempre que el carácter inmediatamente posterior sea mayor.

1.8. Método cadFrontera

Este método formará la cadena más larga posible a partir de "mid" y "mid+1" expandiéndola a derecha e izquierda.

```
cadFrontera(A: string; p, q, mid, m: int) : Resultado {
    var r, r1, r2: Resultado
    r := {mid, 2}
    r1 := extenderIzquierda(A, r, p, m)
    r2 := extenderDerecha(A, r1, q, m)
    return r2
}
```

Este método es bastante sencillo, basado en declarar 3 variables de tipo resultado: "r", inicializada con "mid" y "2" (porque la cadena tiene tamaño 2, "mid" y "mid+1"); "r1", que será el resultado de extender "r" a la izquierda; y "r2", que será el resultado de extender "r1" a la derecha. Finalmente, se devuelve "r2".

2. Análisis teórico del tiempo de ejecución

Estudiaremos el tiempo de ejecución de cada uno de los métodos descritos previamente hasta llegar al método DivideVenceras, que es el principal y el primero en ser llamado para resolver el problema.

2.1. Tiempo de ejecución de los métodos Pequeño y Dividir

Ambos métodos tienen un orden constante, sin mejor ni peor caso.

$$t_{\text{peq}}(p, q, m) \in \mathcal{O}(1)$$

$$t_{\text{div}}(p, q) \in \mathcal{O}(1)$$

2.2. Tiempo de ejecución del método SolucionDirecta

El mejor caso en este método se dará con una cadena totalmente ascendente, de modo que únicamente se va a efectuar la comprobación del if y la actualización de la variable "longitudAct".

$$t_{m_{\text{SolD}}}(p, q) = 3 + \sum_{i=1}^{q-p} 3 = 3 + 3(q - p) \in \mathcal{O}(q - p)$$

Teóricamente, el peor caso sería una cadena en la que siempre se acceda al segundo if (es decir, el que nace del else del primer if), la cual contiene una actualización de la variable “result”, equivalente a 2 instrucciones. Sin embargo, esto es imposible, pues no existe ninguna cadena en la que todo carácter posterior a otro (y que no siga la restricción de ascendencia) forme una cadena ascendente mayor que la previa. Por tanto, el peor caso será el que mayor número de veces acceda a esta actualización de “result”, que se daría en una cadena de caracteres alternos ascendentes (por ejemplo, cdcdcdcd).

$$t_{M_{\text{SolD}}}(p,q) = 3 + \tfrac{1}{2}\sum_{i=1}^{q-p} 3 + \frac{1}{2}\sum_{i=1}^{q-p} 6 = 3 + \frac{9}{2}(q-p) \in \mathcal{O}(q-p)$$

Como podemos ver, tanto el mejor como el peor caso pertenecen al mismo orden, por lo que podemos concluir que:

$$t_{\text{SolD}}(p,q) \in \mathcal{O}(q-p)$$

2.3. Tiempo de ejecución de los métodos extenderDerecha y extenderIzquierda

Estudiaremos ambos métodos en conjunto, puesto que son métodos casi idénticos que apenas tienen diferencias relevantes. Cabe destacar que interpretaremos “S.inicio” como “p” y “limite” como “q”, pues en la mayoría de los casos estos valores van a ser idénticos.

$$t_{\text{extIzq}} = t_{\text{extDer}} = t_{\text{ext}}$$

No tenemos mejor ni peor caso en cuanto a flujo de ejecución, aunque sí dependiendo del contenido de la entrada. En particular, el mejor caso se dará con cadenas que al extenderlas a la izquierda tengan un carácter inmediatamente anterior no descendente y con cadenas que al extenderlas a la derecha tengan un carácter no ascendente:

$$t_{m_{\text{ext}}}(p,q) = 5 \in \mathcal{O}(\mathbf{1})$$

El peor caso se dará cuando la cadena pueda extenderse hasta su límite pasado como argumento, o bien cuando hemos logrado formar una cadena de tamaño “m”. Cualquiera de estas dos condiciones rompe el bucle while, pero usaremos la segunda porque normalmente estaremos trabajando con cadenas grandes, y

normalmente la subdivisión de la cadena siempre va a ser más grande que m. Por lo tanto, queda:

$$t_{M_{\text{ext}}}(p, q, m) = 4 + \sum_{i=1}^{m} 2 = 4 + 2m \in \mathcal{O}(m)$$

Como podemos ver, los tiempos mejor y peor son de distinto orden, por lo que debemos intentar aproximar el tiempo promedio. La posibilidad de que un carácter cualquiera esté precedido por un carácter anterior o seguido de uno posterior, para extenderIzquierda y extenderDerecha respectivamente, es de aproximadamente un 50%. De esta forma, podemos asumir que en promedio podemos modelar el problema como una suma de una progresión geométrica, de modo que cada término tiene una probabilidad $\frac{1}{2}^n$ de ocurrir, siendo un medio la razón de la progresión. De este modo, la suma de la progresión geométrica, que podemos calcular como $S=\frac{a}{1-r}$, siendo "a" el primer término de la progresión (1) y "r" la razón ($\frac{1}{2}$. Sustituyendo, llegamos a que $S=2$, y podemos concluir que, en promedio, cada vez que se llame a extenderIzquierda o extenderDerecha se va a extender dos veces. Por tanto, el tiempo promedio:

$$t_{p_{\text{ext}}}(p, q) = 5 + 2 \times 2 = 9 \in \mathcal{O}(\mathbf{1})$$

2.4. Tiempo de ejecución del método cadFrontera

El mejor, peor y caso promedio de este método depende de que la cadena pasada sea un mejor o peor caso de los métodos extenderDerecha y extenderIzquierda:

$$t_{M_{\text{cadF}}} = 6 + t_{M_{\text{extIzq}}} + t_{M_{\text{extDer}}} = 6 + 2\,(4 + 2m) \in \mathcal{O}(m)$$

$$t_{p_{\text{cadF}}} = 6 + t_{p_{\text{extIzq}}} + t_{p_{\text{extDer}}} = 6 + 2 \times 9 \in \mathcal{O}(\mathbf{1})$$

2.5. Tiempo de ejecución del método Combinar

Encontraremos el mejor caso cuando alguna de las variables de tipo Resultado "S1" o "S2" que recibe contiene una subcadena ascendente de tamaño "m" (contaremos la condición del if como una sola instrucción):

$$t_{m_{\text{Comb}}}(p, q) = 3 \in \mathcal{O}(\mathbf{1})$$

El peor caso lo encontramos cuando ni S1 ni S2 tocan la frontera. En ese caso, tendremos que hacer dos asignaciones de variables de tipo Resultado y además llamar al método cadFrontera, el cual llama a su vez a los métodos extenderIzquierda y extenderDerecha. Además, nos pondremos en el peor caso del método cadFrontera.

$$t_{M_{\text{Comb}}}(p,q,m) = 10 + t_{M_{\text{cadF}}} = 16 + 2 * 6m \in \mathcal{O}(m)$$

La exactitud del tiempo promedio es algo muy complejo de calcular, pues existen muchos tipos de cadenas las cuales van a producir determinadas entradas en el flujo de instrucciones. En un generador aleatorio, la probabilidad de generar cada uno de los casos previstos en Combinar no es la misma para cada uno de los casos. Por ello, asumiremos que la probabilidad combinada de caer en uno de los dos primeros casos (sin continuidad en la frontera y cadena ascendente de tamaño m en el trozo analizado) es de un 50%, y la del resto de los casos (los que nacen a partir de la continuidad de la frontera) es también de un 50%.

De este modo, vamos a calcular un número de instrucciones promedio. Las instrucciones ejecutadas en los casos sin continuidad son de 3 y 4; mientras que en los casos con continuidad vamos a agrupar los casos en 4 posibilidades, aunque podrían sacarse más pero complicaría bastante el problema. Estos serían los siguientes: S1 y S2 tocan la frontera, S1 la toca y S2 no, S1 no la toca y S2 sí, y ni S1 ni S2 tocan la frontera, con número de instrucciones de 7, 8, 16 y 34 respectivamente. Haciendo el cálculo con las probabilidades, obtenemos que:

$$t_{p_{\text{Comb}}} = \frac{1}{4}\times 3 + \frac{1}{4}\times 4 + \frac{1}{8}\times 7 + \frac{1}{8}\times 8 + \frac{1}{8}\times 16 + \frac{1}{8}\times 34 = 9.875 \in \mathcal{O}(\mathbf{1})$$

2.6. Tiempo de ejecución del método DivideVenceras

El método DivideVenceras es el que nos devuelve el tiempo total de ejecución, y podemos definirlo mediante una ecuación de recurrencia. Respecto a los casos mejores y peores, estos van a estar determinados por el tiempo de ejecución del método Combinar. Es decir, que el método tardará más o menos dependiendo de lo fácil que sean de combinar sus subcadenas. También tiene algo de implicación los casos en SoluciónDirecta, aunque con un impacto mucho menor pues tanto mejor como peor caso son del mismo orden.

$$t(p,q) = t_{\text{DV}}(p,q,m) = \begin{cases} 6 + t_{\text{peq}}(p,q,m) + t_{\text{SolD}}(p,q), & \text{if } q-p+1 \leq m \\ 6 + t_{\text{peq}}(p,q,m) + t_{\text{div}}(p,q) + 2t_{\text{DV}}\left(p, \frac{p+q}{2}, m\right) + t_{\text{comb}}(p,q), & \text{if } q-p+1 > m \end{cases} \quad (1)$$

Sustituimos $q - p + 1 = n$ (el +1 es cuestión de índices e irrelevante) y la ecuación de recurrencia queda, de forma simplificada, para el caso promedio (de mismo orden que el mejor caso):

$$t_{p\text{DV}}(n) = 2 \cdot t_{p\text{DV}}\left(\tfrac{n}{2}\right) + \mathcal{O}(1)$$

Y usando el teorema maestro $T(n) = aT\left(\frac{n}{b}\right) + f(n)$, llegamos a que:

$$t_{pDV}(n) = \Theta(n \log_2 2) = \Theta(n).$$

En el peor caso del algoritmo, el orden ya no es el mismo y la ecuación de recurrencia cambia, quedando de la siguiente forma:

$$t_{M_{DV}}(n, m) = 2 \cdot t_{M_{DV}}\left(\tfrac{n}{2}\right) + \mathcal{O}(m)$$

No obstante, sabemos que m = n/1000, por lo que en realidad O(m) = O(n). Aplicando el teorema maestro, obtenemos:

$$t_{M_{DV}}(n) = \Theta\left(n^{\log_b a} \log^{k+1} n\right) = T(n) = \Theta\left(n \log^{0+1} n\right) = \Theta(n \log n)$$

De esta forma, concluimos que en el mejor caso y el promedio obtenemos un orden lineal, mientras que en el peor caso tendremos un orden $n \log n$.

Cabe destacar que debido a las particularidades de nuestro problema, la solución directa es más rápida para los peores casos, pues la extensión a la derecha e izquierda supone recorrer más de una vez la cadena que le pasamos.

3. Programación del algoritmo

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

using namespace std;

// Estructura para guardar los índices
struct Resultado {
```

```
      int inicio, tam; // inicio es el índice donde comienza la
subcadena, y tam el tamaño
};

// Función para comprobar si el problema es lo suficientemente pequeño
bool Pequeno(int p, int q, int m) {
    return (q - p + 1) <= m;
}
// Función que retorna la solución directa al problema
Resultado SolucionDirecta(const string& A, int p, int q) {
    Resultado result = {p, 1};
     int longitudAct = 1;     // La inicializo a 1 porque inicialmente
cada subsolucion es de 1 caracter

     // Iterar sobre la cadena para encontrar la subcadena ascendente
más larga
    for (int i = p + 1; i <= q; ++i) {
        if (A[i] >= A[i - 1]) {
            longitudAct++;
        } else {
            if (longitudAct > result.tam) {
                result = {i - longitudAct, longitudAct};
            }
            longitudAct = 1;
        }
    }
      // Esto es por si la subcadena más larga termina en el último
caracter
    if (longitudAct > result.tam) {
        result = {q - longitudAct + 1, longitudAct};
    }
    return result;
}

// Función que divide el problema en subproblemas más pequeños
int Dividir(int p, int q) {

    return (p + q) / 2;
}


// Función que extiende hacia la izquierda en caso de haber una
posible subcadena más larga
Resultado extenderIzquierda(const string& A, const Resultado& S, int
limite, int m) {
    int inicio = S.inicio;
    int cuentaTam = S.tam;
     while (inicio > limite && A[inicio - 1] <= A[inicio] && cuentaTam
<= m) {
        --inicio;
```

```
        ++cuentaTam;
    }
    Resultado r = {inicio, cuentaTam};
    return r;
}


// Función que extiende hacia la derecha en caso de haber una posible
subcadena más larga
Resultado extenderDerecha(const string& A, const Resultado& S, int
limite, int m) {
    int inicio = S.inicio;
    int cuentaTam=1;
     while (inicio < limite && A[inicio] <= A[inicio + 1] && cuentaTam
<= m) {
        ++inicio;
        ++cuentaTam;
    }
    Resultado r = {S.inicio, cuentaTam};
    return r;
}

// Función que crea la subcadena ascendente más larga a partir del
punto medio de la cadena
Resultado cadFrontera(const string& A, int p, int q, int mid, int m) {
    Resultado r = {mid, 2};
    Resultado r1 = extenderIzquierda(A, r, p, m);
    Resultado r2 = extenderDerecha(A, r1, q, m);
    return r2;
}


// Función que combina dos resultados distintos
Resultado Combinar(const Resultado& S1, const Resultado& S2, const
string& A, int p, int q, int m, int mid) {

    if (S1.tam == m || S2.tam == m) {
        return (S1.tam >= S2.tam) ? S1 : S2;
    }


    // Si no hay continuidad en la frontera
    if (A[mid] > A[mid+1]) {
        return (S1.tam >= S2.tam) ? S1 : S2;
    }

    else {
        // Si S1 llega a la frontera
        if (S1.inicio + S1.tam - 1 == mid) {
            // Si S2 llega a la frontera
```

```
            if (S2.inicio == mid+1) {
                 Resultado r = {S1.inicio, (S1.tam + S2.tam)};
                      // En caso de que la concatenación devuelva una
subcadena mayor que m, devolver una de tamaño máximo m
                 if (r.tam > m) {
                    r.tam = m;
                 }
                return r;
            }
              // Si S2 no toca la frontera, extender S1 y comprobar si
la cadena resultante es más larga que S2
            else {
                Resultado r = extenderDerecha(A, S1, q, m);
                return (r.tam >= S2.tam) ? r : S2;
            }
        }
        // Si S1 no llega a la frontera
        else {
             // Si S2 toca la frontera, extender S2 y comprobar si la
cadena resultante es más larga que S1
            if (S2.inicio == mid+1) {
                Resultado r = extenderIzquierda(A, S2, p, m);
                return (r.tam >= S1.tam) ? r : S1;
            }
               // Si S1 y S2 no tocan frontera, extender a derecha e
izquierda la cadena formada por mid, mid+1 y comprobar si esta es más
larga que S1 y S2
            else {
                Resultado r = cadFrontera(A, p, q, mid, m);
                Resultado r1 = (S1.tam >= S2.tam) ? S1 : S2;
                return (r.tam >= r1.tam) ? r : r1;

            }
        }
    }
}


// Función que resuelve el problema y representa al algoritmo
Resultado DivideVenceras(const string& A, int p, int q, int m) {
    if (Pequeno(p, q, m)) {
        return SolucionDirecta(A, p, q);
    } else {
        int mid = Dividir(p, q);
        Resultado S1 = DivideVenceras(A, p, mid, m);
        Resultado S2 = DivideVenceras(A, mid + 1, q, m);
        return Combinar(S1, S2, A, p, q, m, mid);
    }
}
```

4. Validación del algoritmo

En este apartado vamos a comprobar que el algoritmo funciona para todos los casos posibles y de todo tipo.

En primer lugar, comprobaremos los 7 casos posibles de subcadenas ascendentes entre trozos, con cadenas simples y cortas y donde la solución puede verse de un solo vistazo.

En el siguiente cuadro de código, se muestran pruebas para 6 de esos casos, con palabras de 6 caracteres y valor “m” de 3:

```
    // Cadena sin subcadenas ascendentes
    string cadena = "xgdcba";

    // Cadena con la subcadena ascendente más larga íntegra en S1
    string cadena1 = "xyzgba";

    // Cadena con la subcadena ascendente más larga íntegra en S2
    string cadena2 = "vqmabc";

    // Cadena con S1 y S2 llegando a la frontera
    string cadena3 = "qabcab";

    // Cadena en la que S1 no llega a la frontera pero S2 sí
    string cadena4 = "uvhija";

    // Cadena en la que S2 no llega a la frontera pero S1 sí
    string cadena5 = "qabcda";



       vector<string> cadenas = {cadena, cadena1, cadena2, cadena3,
cadena4, cadena5};

    for (const auto& cadena : cadenas) {
         auto resultado = DivideVenceras(cadena, 0, cadena.length()-1,
3);
           string subcadenaDetectada = cadena.substr(resultado.inicio,
resultado.tam);

        cout << "Cadena: " << cadena << "\n";
        cout << "Posición de inicio: " << resultado.inicio << ", ";
              cout << "Número de caracteres en la mayor subcadena
ascendente: " << resultado.tam << "\n";
                  cout << "Subcadena ascendente detectada: " <<
subcadenaDetectada << "\n\n";
    }
```

```
------------------------------------------------------------------------
-------

Cadena: xgdcba
Posición de inicio: 0, Número de caracteres en la mayor subcadena
ascendente: 1
Subcadena ascendente detectada: x

Cadena: xyzgba
Posición de inicio: 0, Número de caracteres en la mayor subcadena
ascendente: 3
Subcadena ascendente detectada: xyz

Cadena: vqmabc
Posición de inicio: 3, Número de caracteres en la mayor subcadena
ascendente: 3
Subcadena ascendente detectada: abc

Cadena: qabcab
Posición de inicio: 1, Número de caracteres en la mayor subcadena
ascendente: 3
Subcadena ascendente detectada: abc

Cadena: uvhija
Posición de inicio: 2, Número de caracteres en la mayor subcadena
ascendente: 3
Subcadena ascendente detectada: hij

Cadena: qabcda
Posición de inicio: 1, Número de caracteres en la mayor subcadena
ascendente: 3
Subcadena ascendente detectada: abc
```

En este cuadro de texto se estudia el caso restante, en el que para una interpretación más sencilla usamos una palabra de 7 caracteres:

```
    string cadena = "ijkefghabc";
    auto resultado = DivideVenceras(cadena, 0, cadena.length()-1, 4);
        string  subcadenaDetectada  =  cadena.substr(resultado.inicio,
resultado.tam);
    cout << "Cadena: " << cadena << "\n";
    cout << "Posición de inicio: " << resultado.inicio << ", ";
     cout << "Número de caracteres en la mayor subcadena ascendente: "
<< resultado.tam << "\n";
     cout << "Subcadena ascendente detectada: " << subcadenaDetectada
<< "\n\n";
```

```
-------------------------------------------------------------------------
-------

Cadena: ijkefghabc
Posición de inicio: 3, Número de caracteres en la mayor subcadena
ascendente: 4
Subcadena ascendente detectada: efgh
```

Una vez vistos los casos básicos, vamos a generar con un generador de casos 500 palabras aleatorias de longitud n = 100000. Esta es la implementación del generador de casos:

```
vector<pair<string, int>> generador(int numCasos, int longCadena, int
valorM) {
    vector<pair<string, int>> casosDePrueba;
    srand(static_cast<unsigned>(time(nullptr)));.

    for (int i = 0; i < numCasos; ++i) {
        string randomCadena;
        for (int j = 0; j < longCadena; ++j) {
            char randomChar = 'a' + rand() % 26;
            randomCadena += randomChar;
        }

        casosDePrueba.emplace_back(randomCadena, valorM);

    return casosDePrueba;
}
```

Para comprobar que se está encontrando la subcadena ascendente más larga, generaremos 50 casos de 10000 caracteres de longitud, con m = 10, y comprobaremos que obtenemos la misma respuesta (o al menos una respuesta de igual longitud, puesto que si aparecen cadenas de la misma longitud máxima cualquiera de estas es válida).

En los archivos **casos.txt** y **comprobación.txt** se almacena un ejemplo de casos generados aleatoriamente y la comprobación de que el resultado obtenido es el mismo tanto por DivideVenceras como por SolucionDirecta. Son casos generados aleatoriamente pero ya prefijados. Si se quiere probar con otros casos de prueba, bastaría con generar un nuevo archivo casos.txt con el generador proporcionado.

5. Estudio experimental del tiempo de ejecución

Para estudiar el tiempo de ejecución experimentalmente, usaremos generadores de casos promedio, mejor y peor. Tal y como se indica en el enunciado, los valores de n (el tamaño de la cadena) se encuentran entre 10000 y 1000000, así que tomaremos una serie de valores entre estos dos números y para cada longitud n, generaremos 100 casos, cuyo tiempo de ejecución plasmaremos en una gráfica.

5.1. Estudio de los casos peor

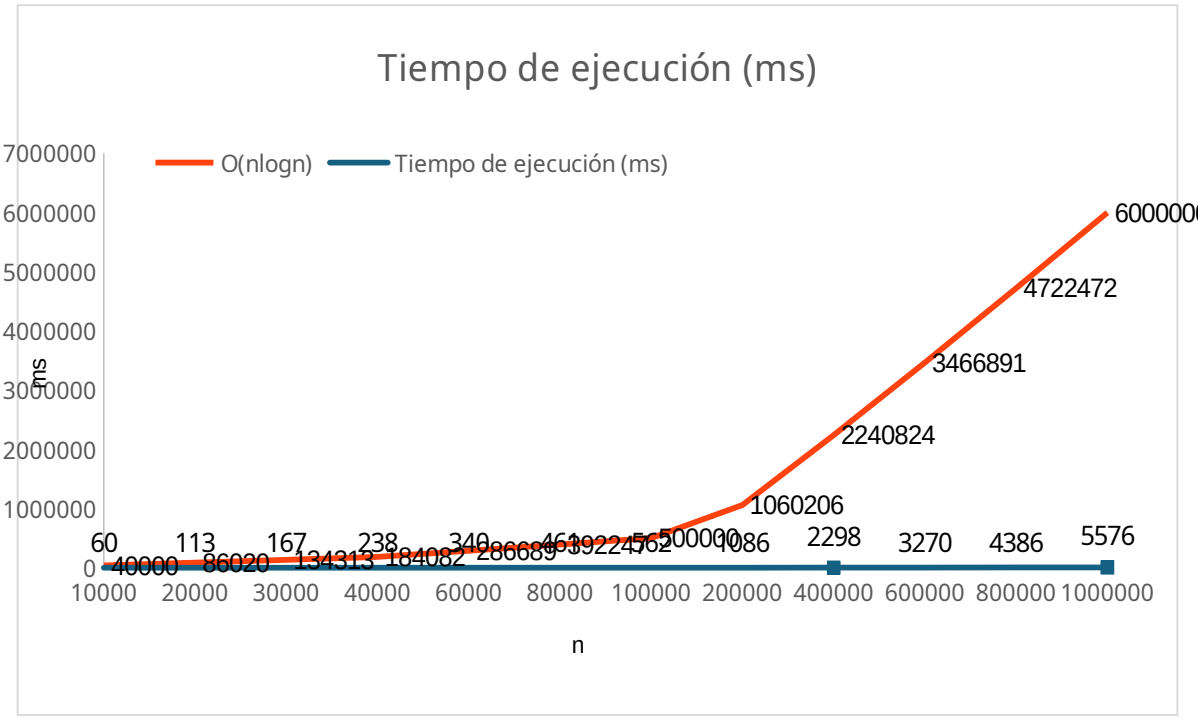
Para la generación de casos peores, hemos creado un generadorPeor (véase el fichero **generadorPeor.cpp**), el cual genera cadenas con un formato específico de tal forma que se tenga que llamar el máximo número de veces al método cadenaFrontera y a los métodos extenderIzquierda y extenderDerecha, consiguiendo así llegar al peor caso del algoritmo.

Para generar las cadenas de caso peor, dividiremos los caracteres en tripletas descendentes (desde zyx, wvu... hasta edc. Llamaremos c1, c2, y c3 a los caracteres de cada una de las tripletas). Cada una de estas tripletas se formará a partir de dos trozos de tamaño “m”, el tamaño máximo de las subcadenas ascendentes que le metemos al programa. Así, con dos trozos de tamaño m, formamos un patrón que consistirá en (m/2 + 1) c1 caracteres de la tripleta, seguidos de 2 * (m/2 - 1) c2 caracteres y seguidos finalmente de (m/2 + 1) c3 caracteres. Este patrón se repite constantemente hasta llegar a la tripleta edc, cuando vuelve a comenzar de nuevo. La primera “z” de las tripletas zyx se sustituye por una “a” para que no se forme una cadena ascendente de “c” y “z” cada vez que se reinicie el ciclo. Para ilustrarlo, tomaremos de ejemplo una cadena con m=8. En color rojo tenemos los caracteres c1, en negrita los c2 y en verde los c3. Además, subrayamos los trozos de tamaño m. El comienzo de la cadena sería el siguiente:

Azzzz**yyyyyy**xxxxx**wwwww****vvvvvv**uuuuu**ttttt****ssssss**rrrrr...

La lógica tras este tipo de cadenas reside en que, por la forma en la que dividimos, acabamos con trozos de m=8 caracteres. Siendo el trozo izquierdo S1 y el derecho S2, encontramos que tanto en S1 como en S2 la cadena más larga es de tamaño 5, y que esta no toca la frontera. Por tanto, como ningún trozo toca frontera, debe llamarse siempre al método cadFrontera, que es el que nos llevaba a un orden n log n. Tras finalizarse la llamada a este método, obtenemos que

hemos encontrado una cadena de tamaño 6. Esto lo hacemos así con todos los trozos y de esta manera maximizamos el orden del algoritmo.



5.2. Estudio de los casos promedio

Para el estudio de los casos promedio hemos creado un generador (**generador.cpp)** que simplemente va generando caracteres aleatorios y los introduce en la cadena hasta llegar a la longitud introducida

5.3. Estudio de los casos mejor

Para el estudio de los mejores casos, hemos creado un generador (**generadorMejor.cpp)** el cual genera cadenas que están formadas por la misma letra, consiguiendo así llegar al mejor caso posible.

5.4. Conclusiones del estudio teórico y experimental

Podemos concluir que el estudio teórico era correcto, pues en los 3 casos nos ha dado una cota superior del tiempo de ejecucción, y nuestros resultados experimentales han quedado por debajo, tal y como suponíamos nosotros. Esto nos permite hacernos una idea de los recursos que consume nuestro algoritmo, y permite prepararnos bien en caso de querer ejecutarlo.

6. Instrucciones de uso

Primero se debe ejecutar el make, que compila tanto los 3 generadores como el fichero main del algoritmo. Luego los 3 generadores funcionan igual, simplemente hay que indicar el número de casos y la longitud de la cadena, y se puede redireccionar a un fichero para mayor comodidad (con el carácter >). El código main tiene varios modos de ejecución (se puede cambiar yendo al main.cpp y descomentando y comentando el modo deseado). Por defecto está en el modo 3, que lee de la entrada estándar (es conveniente que sea un fichero redireccionado con el carácter <) y devuelve la solución por DyV y por solución directa. Aquí va un ejemplo:

juanje@juanje-Lenovo-ideapad-110-15ISK:~/2cuatri/AED2/DyV$ make

g++ main.cpp -o main

g++ generadorPeor.cpp -o generadorPeor

g++ generadorMejor.cpp -o generadorMejor

g++ generador.cpp -o generador

juanje@juanje-Lenovo-ideapad-110-15ISK:~/2cuatri/AED2/DyV$ ./generador > c.txt

Ingrese el número de casos: 10

Ingrese la longitud de las cadenas: 10000

juanje@juanje-Lenovo-ideapad-110-15ISK:~/2cuatri/AED2/DyV$ ./main < c.txt

7. Conclusiones

Ha sido un proyecto altamente complejo y tedioso. Si bien pudimos apoyarnos bastante en ejemplos explicados en la clase de teoría, la fase de implementación y diseño del algoritmo ha sido bastante compleja, pues tuvimos que dar muchos pasos en falso hasta conseguir llegar a un algoritmo correcto.

También fue complicada la creación de los generadores, pues para ello primero tuvimos que darnos cuenta de cuál era el mejor caso y el peor, antes de poder construirlos. Una vez que nos dimos cuenta de ello, el peor si que resultó más largo de terminar, pero pudimos completarlo sin problemas.

Pero sin embargo la parte más extensa de esta práctica, ha sido el estudio teórico del algoritmo, ya que hemos tenido que discernir en todo momento cuando estábamos en un mejor caso, cuando no... Es por ello que esta parte se nos hizo un poco cuesta arriba, ya que era muy fácil cometer un error y no realizar un estudio correcto.

Creemos que el desarrollo de toda la práctica, contando las sesiones de laboratorio, nos ha llevado un tiempo aproximado de 25 horas hasta poder completarla satisfactoriamente.