Semana 2 - Ejercicios de grupo

Métodos algorítmicos en resolución de problemas II
Facultad de Informática - UCM

	Nombres y apellidos de los componentes del grupo que participan	**ID Juez**
1	Borja Aday Guadalupe Luis	MAR241
2	Óscar Morujo Fernández	MAR262
3	Roberto Plaza Hermira	MAR267
4	Pablo Martínez	MAR258

Instrucciones:

1. Para editar este documento, es necesario hacer una copia de él. Para ello:
    - Alguien del grupo inicia sesión con la cuenta de correo de la UCM (si no la ha iniciado ya) y accede a este documento.
    - Mediante la opción *Archivo* → *Hacer una copia*, hace una copia del documento en su propia unidad de *Google Drive*.
    - Abre esta copia y, mediante el botón *Compartir* (esquina superior derecha), introduce los correos de los demás miembros del grupo para que puedan participar en la edición de la copia.

2. La entrega se realiza a través del Campus Virtual. Para ello:
    - Alguien del grupo convierte este documento a PDF (dándole como nombre el número del grupo, 1.pdf, 2.pdf, etc…). Desde *Google Docs*, puede hacerse mediante la opción *Archivo* → *Descargar* → *Documento PDF*.
    - Esta misma persona sube el fichero PDF a la tarea correspondiente del *Campus Virtual*. Solo es necesario que uno de los componentes del grupo entregue el PDF.

Subsecuencia común más larga

El objetivo de hoy es resolver el **problema 7 Subsecuencia común más larga**, del juez automático. Ahí podéis encontrar el enunciado completo del problema.

Las subsecuencias de una secuencia son todas aquellas que se obtienen a base de seleccionar algunos caracteres de la secuencia (una cantidad entre 0 y la longitud de la secuencia) en el mismo orden en que se encuentran en la secuencia original. Dadas dos secuencias de caracteres se pretende encontrar una subsecuencia de ambas que sea lo más larga posible. Fijaos en que si hay varias subsecuencias comunes de igual longitud se puede devolver como resultado cualquiera de ellas.

Solución: (Plantead aquí la recurrencia que resuelve el problema y explicad las razones por las que es mejor resolver el problema utilizando programación dinámica que una implementación recursiva sin memoria.)

Definición : maxSubsecuencia(i,j) = longitud de la mayor subsecuencia teniendo en cuenta los i primeros caracteres de la primera palabra y los j primeros caracteres de la segunda, donde donde 0 <= i < N1 y donde 0 <= j < N2 (siendo N1 la longitud de la primera palabra y N2 la longitud de la segunda)

Casos base:
Sea N1 la longitud de la primera palabra y N2 la longitud de la segunda

maxSubsecuencia(0, j) = 0 donde 0 <= j < N2

maxSubsecuencia(i, 0) = 0 donde 0 <= i < N1

Casos recursivos:
Con i, j >= 0
maxSubsecuencia(i , j) = maxSubsecuencia(i-1 ,j - 1) + 1 si cadena1[i] == cadena2[j]
maxSubsecuencia(i,j) = max(maxSubsecuencia(i - 1, j), maxSubsecuencia(i, j - 1)) si cadena1[i] != cadena2[j]

Llamada inicial :

maxSubsecuencia(N1, N2), donde 0 <= i < N1 y donde 0 <= j < N2 (siendo N1 la longitud de la primera palabra y N2 la longitud de la segunda).

Como en el resto de problemas que hemos realizado con programación dinámica, el objetivo está en evitar el cálculo repetido de subproblemas.

Solución: (Escribid aquí las explicaciones necesarias para contar de manera comprensible la implementación del algoritmo de programación dinámica. Explicad con detalle cómo se construye una de las subsecuencias comunes más largas y en qué parte de ese código se está eligiendo una de las posibles subsecuencias en caso de existir varias. Dad un ejemplo en que existan varias posibles soluciones e indicad cuál elegiría vuestro algoritmo.
Incluid el código y el coste justificado de las funciones que resuelven el problema. Extended el espacio al que haga falta.)

En este caso hemos optado por una implementación de programación dinámica descendente para resolver solo los subproblemas necesarios. Así, para encontrar la subsecuencia más larga en común,empezaremos teniendo en cuenta todos los caracteres de las dos palabras y empezando por comparar caracteres desde el final, así, si son iguales podremos sumar uno a la longitud de la subsecuencia y seguir comparando los restantes, y si son diferentes tendremos que buscar la longitud de la mayor subsecuencia descartando uno de los dos caracteres y quedándonos con el mayor que obtengamos.

- Coste en tiempo : O(N1 x N2) , en el caso peor las dos cadenas tienen el mismo tamaño y no comparten ninguna letra, por lo tanto la complejidad sería cuadrática (O(N^2)).

- Coste en memoria : O(N1 x N2) , ya que necesitamos una matriz para almacenar las longitudes.

Una vez tenemos calculada la matriz, podemos reconstruir la solución empezando desde la última fila, buscamos la longitud de la subsecuencia máxima, y cogemos el index (indica el carácter usado para construir la subsecuencia). Así, si lo encontramos restamos en uno el tamaño máximo y buscamos en la fila anterior, si no, no restamos uno al tamaño máximo y buscamos en la anterior fila.

- Coste en tiempo : O(N1x N2)
- Coste en espacio : O(1) , no se usa espacio adicional para el cálculo

Dad un ejemplo en que existan varias posibles soluciones e indicad cuál elegiría vuestro algoritmo:

En el ejemplo: AMAPOLA MATAMOSCAS
Solución del enunciado : AAOA
Nuestra solución: AMOA

Debido al orden de los ifs y la recursividad de la función de reconstrucción, se descarta primero los caracteres de la primera cadena, por lo que la salida al ejemplo es la que obtenemos. Si descartamos primero el carácter de la cadena 2, obtendremos el ejemplo del enunciado.

```
// Programacion dinamica descendente. El coste está en O(N*M) siendo N el tamano de la cadena1 y M el tamano de la cadena2.
// Esto se debe a que vamos descartando caracteres de ambas cadenas hasta agotar una o ambas.
int maxSubsecuencia(string const& cadena1, string const& cadena2, int i, int j, Matriz<int>& tabla) {
    if (i < 0 || j < 0) {
        return 0;
    }
    int& res = tabla[i][j];
    if (res == -1) {
        if (i >= 0 && j >= 0 && cadena1[i] == cadena2[j]) {
            res = maxSubsecuencia(cadena1, cadena2, i - 1, j - 1, tabla) + 1;
        }
        else if (i >= 0 && j >= 0 && cadena1[i] != cadena2[j]) {
            res = max(maxSubsecuencia(cadena1, cadena2, i - 1, j, tabla), maxSubsecuencia(cadena1, cadena2, i, j - 1, tabla));
        }
    }
    return res;
}

// El coste de esta función en el caso peor esta en O(N*M) siendo N el tamano de la cadena1 y M el tamano de la cadena2.
void reconstruir(string const& cadena1, string const& cadena2, Matriz<int> const& tabla, int i, int n2, int tam, string & sol) {
    int j = 0;
    if (tam > 0)
    {
        bool encontrado = false;
        while (j < n2 && !encontrado)
        {
            if (tabla[i][j] == tam)
                encontrado = true;
            else
                j++;
        }
        if(j == n2) reconstruir(cadena1, cadena2, tabla, i - 1, n2, tam, sol);
        else {
            reconstruir(cadena1, cadena2, tabla, i - 1, j, tam - 1, sol);
            sol.push_back(cadena2[j]);
        }
    }
}
```

Resolved el problema completo del juez, que ahora debe ser ya muy sencillo.

Número de envío: s32022