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	Daniela Alejandra Cordova	MAR220
2	Alejandro Corpas Calvo	MAR222
3	Erik Karlgren Domercq	MAR248
4	David Bugoi	MAR211

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:

- subsecuencia(palabra1, palabra2, i, j) = es el número de letras de la subsecuencia más larga común de las 2 secuencias de caracteres (palabra1 y palabra2) entre el final de la secuencia y el carácter i y j respectivamente. Recorre las secuencias de derecha a izquierda, por lo cual se obtiene la subsecuencia más larga correspondiente a este caso.
- Rango de 'i': (0<=i<=palabra1.size())
- Rango de 'j': (0<=j<=palabra2.size())

Llamada Inicial:

- subsecuencia(palabra1, palabra2, palabra1.size(), palabra2.size())

Casos Base:

- subsecuencia(palabra1, palabra2, 0, j) = 0
- subsecuencia(palabra1, palabra2, i, 0) = 0

Casos Recursivos:

- subsecuencia(palabra1, palabra2, i - 1, j - 1) + 1 → palabra1[i-1] == palabra2[j-1]
- max(subsecuencia(palabra1, palabra2, i - 1, j), subsecuencia(palabra1, palabra2, i, j - 1)) → e.o.c

Si resolviésemos este problema mediante recursión sin memoria calcularía varias veces el mismo resultado para varias subsecuencias siendo muy ineficiente. Esto es debido a que según vamos ejecutando el algoritmo, es decir, quitando caracteres de las palabras, nos van apareciendo subsecuencias ya resueltas previamente. Si guardásemos esas soluciones en memoria ahorraríamos una gran cantidad de operaciones repetidas. Por ello, para resolver este problema conviene usar programación dinámica, y por tanto, memoria adicional que guarda las soluciones a esos subproblemas para tener que calcularlas una sola vez.

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 el problema recorremos las palabras de derecha a izquierda reduciendo el número de letras a leer dependiendo del caso. Luego se va actualizando el valor de la matriz dependiendo de los valores i y j a leer. Siendo **i** la letra que estamos usando para comparar de la palabra1 y **j** la letra en la palabra2. Utilizamos programación dinámica descendente para guardar los resultados previamente calculados.
Utilizamos programación dinámica descendente ya que no podemos realizar una mejora en espacio adicional y este método es mucho más sencillo de implementar.

Ejemplo de varias posibles subsecuencias:
En la reconstrucción es dónde se decide la palabra que vamos a reconstruir y como empezamos de derecha a izquierda, tenemos el caso que explicamos en el primer caso:
Palabras iniciales: Amapola y matamoscas

1. Nuestra solución: Recorremos y reconstruimos de derecha a izquierda: MAOA
2. Solución alternativa 1: Recorrer y reconstruir de izquierda a derecha: AAOA
3. Solución alternativa 2: Recorrer y reconstruir por ambos lados a la vez: AMOA

Código:

```
int subsecuencia(string const& palabra1, string const& palabra2, int i, int j, Matriz<int>& m) {
	int& res = m[i][j];
	if (res == 0) {
		 if (i == 0 || j == 0) res = 0;
		else if (palabra1[i-1] == palabra2[j-1])
			res = subsecuencia(palabra1, palabra2, i -1, j - 1, m) + 1;
		else
			res = max(subsecuencia(palabra1, palabra2, i - 1, j, m), subsecuencia(palabra1,
palabra2, i, j - 1, m));
	}
	return res;
}

void reconstruir(string const& palabra1, string const& palabra2, Matriz<int> const& m, int i, int j, string& sol) {
	if (i != 0 && j != 0) {
		if (palabra1[i - 1] == palabra2[j - 1]) {
			reconstruir(palabra1, palabra2, m, i - 1, j - 1, sol);
			sol.push_back(palabra1[i - 1]);
		}
		else {
			if (m[i][j] == m[i][j - 1]) {
				reconstruir(palabra1, palabra2, m, i, j - 1, sol);
			}
			else {
				reconstruir(palabra1, palabra2, m, i - 1, j, sol);
			}
		}
	}
}
```

Sean L1 la longitud de palabra1 y L2 la de palabra2.

Coste en espacio

Usamos una matriz de L1+1 por L2+1 elementos para guardar los resultados, por tanto el coste en espacio adicional está en O(L1 * L2), es decir, es lineal en la multiplicación del número de caracteres de la palabra1 con el número de caracteres de la palabra2.

Como es necesario reconstruir la secuencia común más larga no es posible reducir ese coste en espacio adicional. Si nos pidiesen simplemente el número de caracteres de esa secuencia podría llegar a resolverse con coste en espacio adicional de O(max(L1, L2)), o lo que es lo mismo, con un vector del mismo tamaño que la palabra más larga.

Coste en tiempo

Tenemos que evaluar el coste de encontrar la longitud máxima posible de una subcadena común y el de reconstruir la solución por separado. Primero, para encontrar dicha longitud vamos considerando cada vez subcadenas de un carácter menos para ambas palabras hasta que una de ellas sea una cadena vacía. En el caso peor llegaremos a una subcadena de longitud 1 y otra de longitud 0 habiendo decrementado ambas de uno en uno alternadamente. Por tanto, el coste será del orden de la multiplicación de las longitudes originales de las 2 palabras, es decir, O(L1 * L2).

El coste de reconstruir la solución es de O(min(L1, L2)) ya que recorremos como mucho la menor de las palabras completamente. Además el coste de añadir a sol una letra es de coste constante (push_back).

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

Número de envío: s32005