Práctica 4: Programación dinámica

Francisco Javier Bolívar Expósito
Manuel Jesús Núñez Ruiz
Miguel Pedregosa Pérez

Índice

1. Subsecuencia más larga

1.1. Definiciones

Sean S y R dos cadenas de caracteres:

$S = \{s1, s2 \dots sn\};\ R = \{r1, r2 \dots rm\}$
n, m *tamaños de las secuencias*

Definimos una subsecuencia común de caracteres de S y R como dos secuencias de índices de caracteres de S y R, I_S e I_R, las cuales cumplen:

$Is = \{a1, a2 \dots ak\};\ Ir = \{b1, b2 \dots bk\}$ *tal que* :

- $ai < aj \Leftrightarrow i < j$
- $bi < bj \Leftrightarrow i < j$
- $S[ai] = R[bi]\ \forall i,\ 0 < i \leq k$

Definimos la operación ‘ (apóstrofe) para cadenas de caracteres de la siguiente manera:

$S = \{s1, s2 \dots sn\} \Rightarrow S' = \{s2 \dots sn\}$

1.2. Algoritmo básico

Así, estamos en posición de enunciar el algoritmo de generación de la subsecuencia máxima (como secuencia de caracteres):

1. *Si S o R tienen cardinalidad cero, la solución es la cadena vacía. En caso contrario, continuar.*
2. *Comparar los primeros caracteres de S y R. Si son iguales ir al paso* 3, *si son diferentes ir al paso* 4
3. *La solución será la concatenación del primer carácter de S con la solución del algoritmo para S' y R'*
4. *Sean dos soluciones parciales A, B definidas de la siguiente manera* :
   A es la solución del algoritmo para S, R'
   B es la solución del algoritmo para S', R
   La solución será la solución parcial con mayor cardinalidad

El algoritmo funciona porque recorre las combinaciones posibles de subsecuencias (en forma de secuencias de índices) mediante la realización de la operación definida previamente. Podemos definir el algoritmo de forma equivalente, pero tratando los índices de manera explícita, de la siguiente manera:

Sean i y j índices de S y R respectivamente, con valor inicial de 0
1. *Si i es el último índice de S o j de R, retornar la cadena vacía. En caso contrario, continuar.*
2. *Comparar S*[*i*] *y R*[*j*]. *Si son iguales ir al paso* 3, *si son diferentes ir al paso* 4
3. *++ i, ++ j, retornar S*[*i* − 1] *concatenado con el resultado de volver al paso* 1
4. *Dividimos el flujo del algoritmo en dos* :
    4.1 *++ i, sea A la solución de volver al paso* 1
    4.2 *++ j, sea B la solución de volver al paso* 1
Reunir el flujo del algoritmo, retornar el máximo entre A y B

En esta segunda definición vemos claramente cómo se construye la subsecuencia manteniendo las propiedades previamente mencionadas.

Aunque este algoritmo es una solución al problema, es muy ineficiente. En el caso peor, en el que en todas las iteraciones ejecutamos el paso 4 y no el 3, es fácil imaginarse un árbol binario como representación de las llamadas recursivas del algoritmo, un árbol cuya hoja menos profunda estará a profundidad $min(n, m)$ y cuya hoja más profunda estará a profundidad $m + n - 1$. Por esto, aunque podríamos mejorar un poco la estimación debido a la irregularidad del árbol, vemos que la complejidad del algoritmo es de **O($2^{max(n,m)}$)**.

1.2.1 Pseudocódigo

```
FUNCION SML(A : Cadena de caracteres, B: Cadena de caracteres) {
        IF A o B están vacías THEN
                RETURN Cadena vacía
        ELSE IF A[0] = B[0] THEN
                RETURN A[0] concatenado con SML(A sin primer carácter, B sin primer
carácter)
        ELSE
                RETURN max(SML(A sin primer carácter, B), SML(A, B sin primer carácter))
```

1.3. Algoritmo dinámico

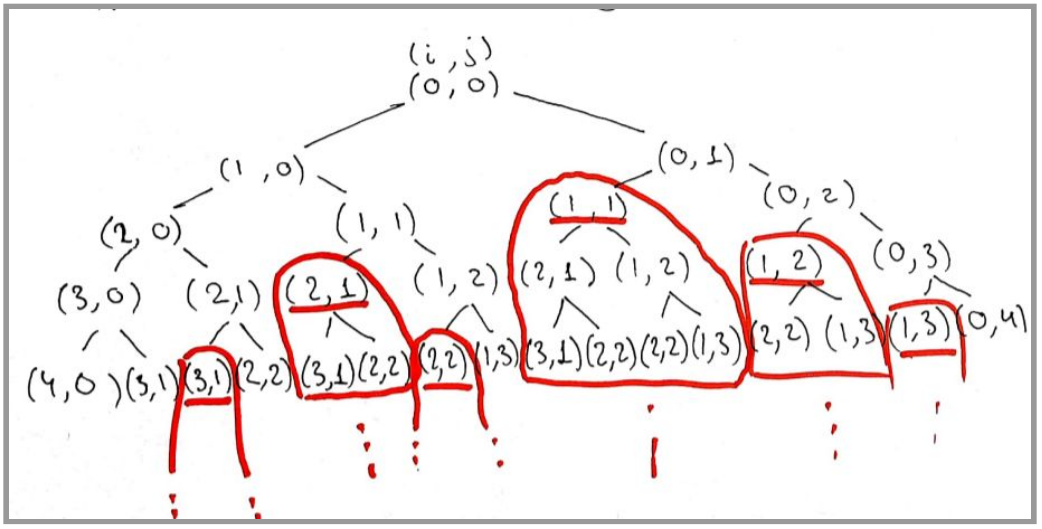
Esta sería la función recurrente del algoritmo:

$$SML(X_i,Y_j) \begin{cases} Ø & \text{si } i = 0 \text{ o } j = 0 \\ SML(X_{i-1}, Y_{j-1})\hat{}x_i & \text{si } i,j > 0 \text{ and } x_i = y_j \\ \max\{SML(X_i,Y_{j-1}), SML(X_{i-1}, Y_j)\} & \text{si } i,j > 0 \text{ y } x_i \text{ != } y_j \end{cases}$$

1.3.1. Algoritmo recursivo

1.3.1.1. Implementación

Esto lo podemos solucionar fácilmente mediante la utilización de programación dinámica. Pongamos un ejemplo: la iteración del algoritmo con los índices i=3 y j=4 será llamada por la iteración (2, 4) y por la (3, 3). Podemos pensar que si almacenásemos el resultado del algoritmo para cada entrada, podaríamos así todas las ramas repetidas del árbol. De hecho, a profundidad k del árbol, solo tendremos iteraciones con entradas (i, j) tal que i+j=k. Así, para profundidad n tenemos $\sum_{1}^{n} n$ iteraciones no repetidas. Así, si almacenamos los resultados de las iteraciones, podemos pasar de una eficiencia exponencial a una eficiencia **O(n^2)**.



Aquí podemos ver una representación del árbol mencionado, subrayadas en rojo las iteraciones repetidas que se pueden podar.

Así, definimos nuestro algoritmo utilizando almacenamiento de la información dinámico:

1. *Si tenemos solución almacenada para S y R, retornarla. Si no, seguir.*
2. *Si S o R tienen cardinalidad cero, la solución es la cadena vacía. En caso contrario, continuar.*
3. *Comparar los primeros caracteres de S y R. Si son iguales ir al paso* 4, *si son diferentes ir al paso* 5
4. *La solución será la concatenación del primer carácter de S con la solución del algoritmo para S' y R'.*
   Almacenamos la solución y la retornamos
5. *Sean dos soluciones parciales A, B definidas de la siguiente manera* :
   A es la solución del algoritmo para S, R'
   B es la solución del algoritmo para S', R
   La solución será la solución parcial con mayor cardinalidad
   Almacenamos la solución y la retornamos

1.3.1.2. Pseudocódigo

```
FUNCION SML(A: Cadena de caracteres, B: Cadena de caracteres, index_a, index_b)
        SOLUCIONES : Matriz con las soluciones encontradas

        SI A o b están vacíos ENTONCES
                DEVOLVER cadena vacía
        SI SOLUCIONES[index_a][index_b] != “\0” ENTONCES
                DEVOLVER SOLUCIONES[index_a][index_b]

        SI A[0] = B[0] ENTONCES
                DEVOLVER A[0] concatenado con SML(A sin primer carácter, B sin primer carácter, index_a+1, index_b+1)
        SI NO
                RETURN max(SML(A sin primer carácter, B, index_a+1, index_b), SML(A, B sin primer carácter, index_a, index_b +1))
```

1.3.2. Algoritmo iterativo

1.3.2.1. Implementación

Se han realizado tres implementaciones. Una para el algoritmo sin la optimización dinámica utilizando recursividad, y dos para el algoritmo dinámico, correspondientes a las dos definiciones del algoritmo utilizando índices y cadenas de caracteres. De estas tres la más interesante y de la cual explicaremos las particularidades es la implementación optimizada utilizando índices.

Esta implementación consiste en una simplificación estructural de las estructuras de datos utilizadas. En primer lugar creamos la matriz que servirá, esta vez, para almacenar la cardinalidad de las soluciones parciales. Esto es, si la solución parcial para los índices (i, j) es la cadena “ab” concatenado con la solución de la ejecución para otros índices, almacenaremos 2 en la posición (i, j) de la matriz. Para ejemplificar gráficamente, aquí tenemos la matriz resultante para la instancia del problema donde S = “MZJAWXU” y R = “XMJYAUZ”:

		0	1	2	3	4	5	6	7
		Ø	M	Z	J	A	W	X	U
0	Ø	0	0	0	0	0	0	0	0
1	X	0	0	0	0	0	0	1	1
2	M	0	1	1	1	1	1	1	1
3	J	0	1	1	2	2	2	2	2
4	Y	0	1	1	2	2	2	2	2
5	A	0	1	1	2	3	3	3	3
6	U	0	1	1	2	3	3	3	4
7	Z	0	1	2	2	3	3	3	4

Tras la generación de esta matriz, la solución al problema la obtendremos mediante un análisis de esta (señalado en amarillo en la imagen). Comenzando por la esquina más alejada del comienzo de las cadenas (en cuya posición estará almacenada la cardinalidad de la solución) podemos trazar el camino mediante el cual se ha llegado a ella de la manera:

- Si [i-1, j] = [i , j] entonces nos movemos a [i-1, j]
- Si no, hacemos lo mismo comparando con [i , j-1]
- Si no ocurre ninguna de las dos, nos movemos a [i-1 , j-1] e incluimos S[i] = R [j] en la solución

Así, tenemos un algoritmo dividido en dos partes, siendo estas la generación de la matriz y la posterior generación de la solución. Además, por trabajar meramente con índices, las cadenas y la matriz, esta implementación del algoritmo es fácilmente implementable con herramientas de programación de bajo nivel. No es necesario disponer de la capacidad de programar recursivamente ni de estructuras de datos de alto nivel. Por esto, es el algoritmo que menos memoria consume, ya que no hay varios hilos de ejecución cada uno con su versión parcial de los datos.

1.3.2.2. Pseudocódigo

```
FUNCION SML(A: Cadena de caracteres, B; Cadena de caracteres)
    SOLUCION: Matriz de números

    PARA i = 0 HASTA B.size() HACER:
        SOLUCION[0][i] = 0
    PARA i = 0 HASTA A.size() HACER:
        SOLUCION[i][0] = 0

    PARA i = 1 HASTA A.size() HACER:
        PARA j = 1 HASTA B.size() HACER:
            SI A[i-1] = B[j-1] ENTONCES
                SOLUCION[i][j] = SOLUCION[i-1][j-1] + 1
            SI NO ENTONCES
                SOLUCION[i][j] = MAX(SOLUCION[i-1][j], SOLUCION[i][j-1])

    x, y, actual : intergers
    x = A.size()
    y = B.size()
    actual = SOLUCION[x][y]
    SML: Cadena de caracteres

    MIENTRAS ACTUAL != 0 HACER
        SI SOLUCION[x][y-1] = actual ENTONCES
            y = y-1
        SI NO ENTONCES SI SOLUCION[x-1][y] = actual ENTONCES
            x = x-1
        SI NO ENTONCES
            Añadir al final de SML A[x-1]
            x = x-1
            y = y-1
            actual = SOLUCION[x][y]

    DEVOLVER SML
```

1.3.2.3. Complejidad del algoritmo

Como podemos observar en el pseudocódigo hay un bucle for dentro de otro para recorrer la matriz entera por lo que esa parte es $O(n^2)$ y por último hay un bucle while cuya complejidad es O(n).
Max{$O(n^2)$, O(n)} = **$O(n^2)$**.

1.4. Casos de ejecución

```
miguel@miguel-UX305FA:~/Escritorio/ALG/practica4/codigo$ ./sml_iterativo 20 20
String A (tam 20): DYZQSRLKSSOEUGGBXKEK
String B (tam 20): YHTZNIFBGLKPWSKNJHPK
Solución (tam 7): YZLKSKK
miguel@miguel-UX305FA:~/Escritorio/ALG/practica4/codigo$ ./sml_recursivo 20 20
String A (tam 20): HILXTGRBHEYVAUMWYKOF
String B (tam 20): LTIOUCIMPQTVWNERCPNA
Solución (tam 5): LTUMW
```

```
miguel@miguel-UX305FA:~/Escritorio/ALG/practica4/codigo$ ./sml_recursivo -manual
Introduce el string a: acbaed
Introduce el string b: abcadf
String A (tam 6): acbaed
String B (tam 6): abcadf
Solución (tam 4): abad
```

(Tiempo medido en microsegundos)

Tamaño	sml-recursivo	sml-recursivo-dyn	sml-iterativo
15x15	7042337	200	30
100x100	-	12946	426
500x500	-	197290	7283
1000x1000	-	567437	13946