UNIVERSIDAD
DE GRANADA

Programación dinámica

Algorítmica. Práctica 4

Celia Arias Martínez
Miguel Ángel Fernández Gutiérrez
Sergio Quijano Rey
Lucía Salamanca López

segfault

Contenidos

Introducción

Objetivo

Apreciar la utilidad de la *programación dinámica para* resolver problemas que involucran recursividad, de forma eficiente.

- **Problema escogido**: Subsecuencia de caracteres más larga (*LCS* o *Longest Common Subsequence*).

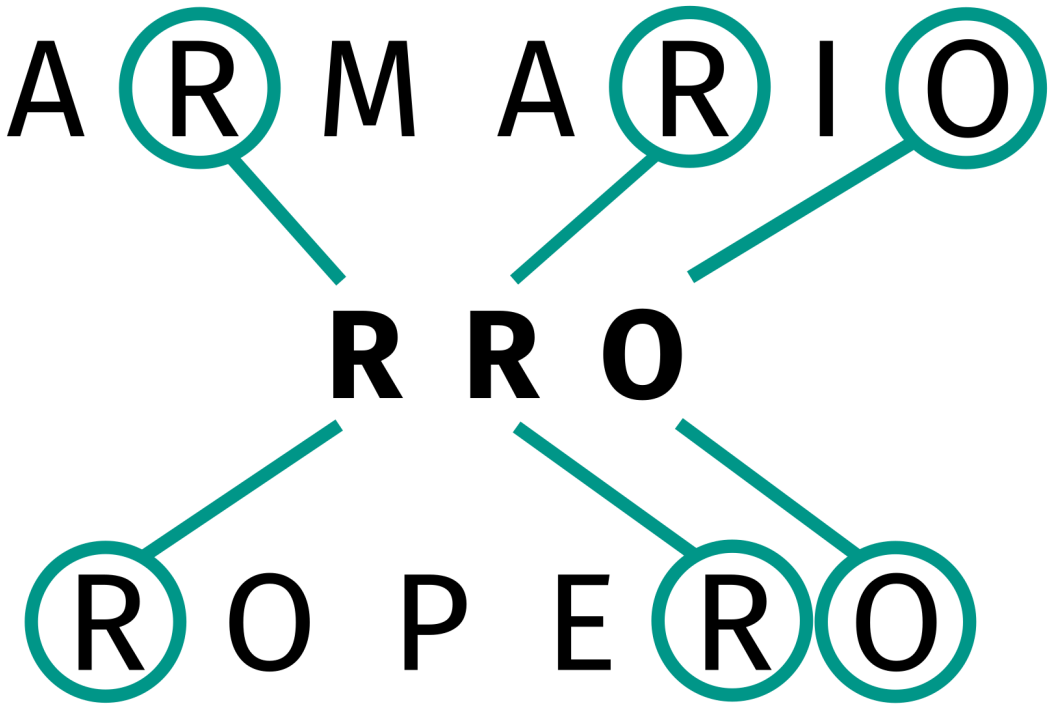
Problema: LCS (subsecuencia más larga)

Descripción del problema

Problema de la subsecuencia más larga (LCS)

Dadas dos secuencias de caracteres, encontrar la máxima subsecuencia de caracteres común que aparecen en ambas cadenas de izquierda a derecha (no necesariamente de forma contigua).

Descripción del problema



Diseño de la solución

Aplicamos programación dinámica en cuatro fases:

1. Verificación de la naturaleza n-etápica del problema.
2. Verificación del principio de optimalidad de Bellman.
3. Planteamiento de una recurrencia.
4. Cálculo de la solución.

Diseño de la solución. Naturaleza *n*-etápica del problema

El resultado es consecuencia de una sucesión de decisiones: en la etapa *n* debemos elegir qué cadena de caracteres dejamos fija y qué puntero movemos hacia la derecha.

Una **solución optimal** es aquella que proporcione la subsecuencia de caracteres más larga.

Diseño de la solución. Principio de optimalidad de Bellman

Función a maximizar: número de caracteres en común en las dos subsecuencias. Tendremos, por tanto, en la etapa n:

$$f_n(i,j) = \text{máx}\{f(i-1,j), f(i,j-1)\} + \delta_{ij}$$

- δ_{ij}: función que dados dos índices (de distintas subcadenas), devuelve 1 si coinciden y 0 en otro caso.

Caso base:

$$f_1(i,j) = \text{máx}\{\delta_{ij}\} = \delta_{ij}$$

Es obvio que la solución proporcionada en la etapa n es optimal, pues estamos calculando el máximo de dos funciones optimales.

Diseño de la solución. Planteamiento de recurrencia

Recurrencia planteada

$$LCS[i,j] = \begin{cases} 1 + LCS[i-1, j-1] \\ \text{máx}\{LCS[i-1, j], LCS[i, j-1]\} \end{cases}$$

Conseguimos asegurar el principio de optimalidad: si estamos en la etapa n, las decisiones tomadas hasta la etapa $n-1$ son óptimas.

Diseño de la solución. Cálculo de solución

constructLCSMat

```
1  vector<vector<int> > constructLCSMat(string str1, string str2) {
2      vector<vector<int> > lcs_mat(str1.size() + 1,
3          vector<int>(str2.size() + 1, 0));
4
5      for ( int i = 1; i <= str1.size(); i++ ) {
6          for ( int j = 1; j <= str2.size(); j++ ) {
7              if ( str1[i-1] == str2[j-1] )
8                  lcs_mat[i][j] = 1 + lcs_mat[i-1][j-1];
9              else
10                 lcs_mat[i][j]
11                     = max(lcs_mat[i-1][j], lcs_mat[i][j-1]);
12         }
13     }
14
15     return lcs_mat;
16 }
```

Diseño de la solución. Cálculo de solución

`constructLCSMat`

Creamos la matriz usada para encontrar el número de caracteres y las letras que conforman la máxima subsecuencia en común.

$$
\begin{array}{cc}
 & \begin{array}{cccccccc} a & r & m & a & r & i & o \end{array} \\
\begin{array}{c} \\ r \\ o \\ p \\ e \\ r \\ o \end{array} &
\begin{pmatrix}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 \\
0 & 0 & 1 & 1 & 1 & 1 & 1 & 2 \\
0 & 0 & 1 & 1 & 1 & 1 & 1 & 2 \\
0 & 0 & 1 & 1 & 1 & 1 & 1 & 2 \\
0 & 0 & 1 & 1 & 1 & 2 & 2 & 2 \\
0 & 0 & 1 & 1 & 1 & 2 & 2 & 3
\end{pmatrix}
\end{array}
$$

Diseño de la solución. Cálculo de solución

getLCS

```
1  string getLCS(string str1, string str2) {
2      vector<vector<int> > lcs_mat = constructLCSMat(str1, str2);
3      string word;
4
5      int i = str1.size(), j = str2.size();
6
7      while ( j > 0 ) {
8          j--;
9          if ( lcs_mat[i][j] != lcs_mat[i][j+1] ) {
10             word = str2[j] + word;
11             i--;
12         }
13     }
14
15     return word;
16 }
```

Diseño de la solución. Cálculo de solución

`getLCS`

Llamamos a la función `constructLCSMat` para construir la matriz, luego la recorremos para encontrar las letras de la LCS.

		a	r	m	a	r	i	o
	0	0	0	0	0	0	0	0
r	0	0	**1**	1	1	1	1	1
o	0	0	**1**	1	1	1	1	2
p	0	0	**1**	1	1	1	1	2
e	0	0	**1**	**1**	**1**	1	1	2
r	0	0	1	1	1	**2**	**2**	2
o	0	0	1	1	1	2	2	**3**

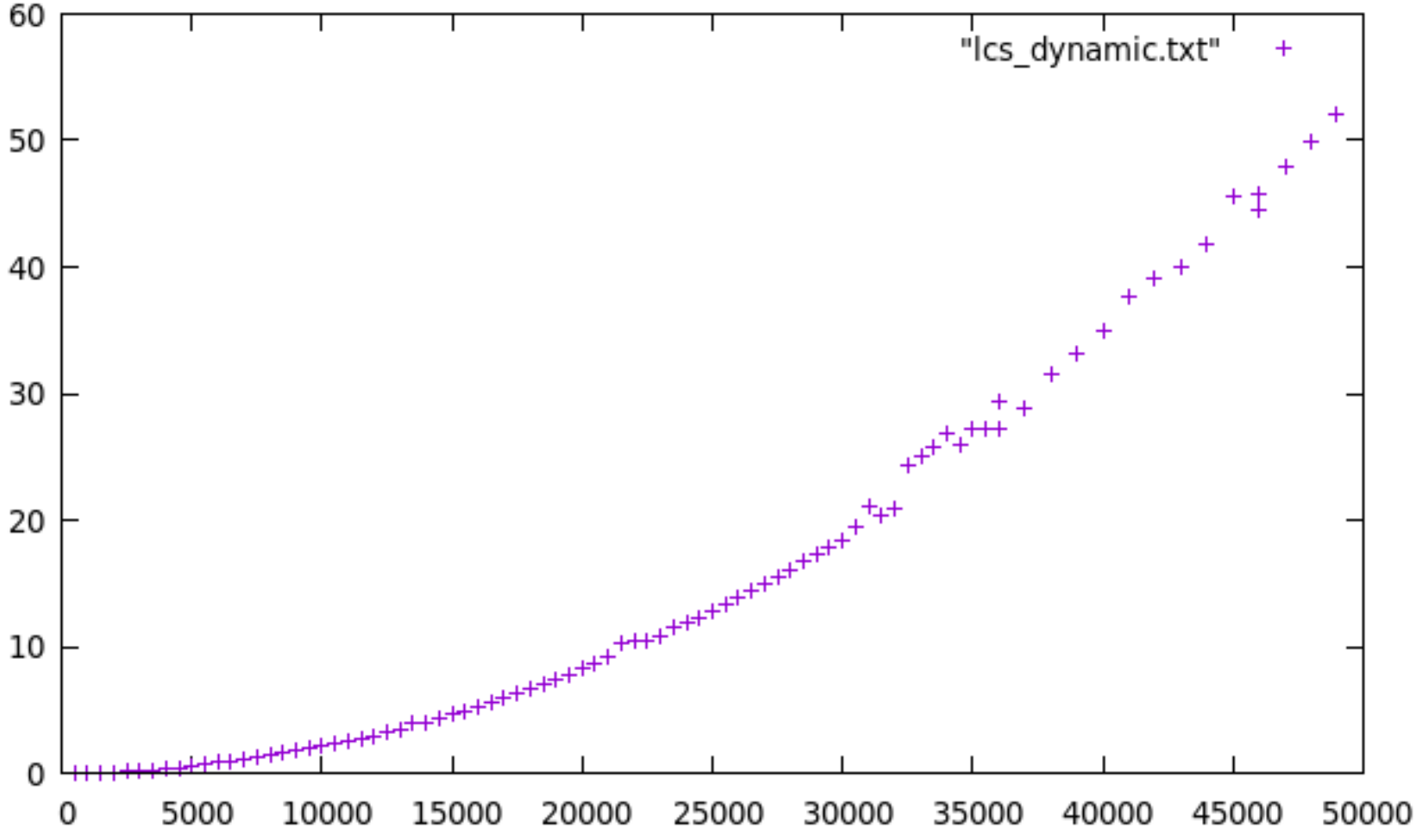
Diseño de la solución. Cálculo de solución

		a	r	m	a	r	i	o
	0	0	0	0	0	0	0	0
r	0	0	**1**	1	1	1	1	1
o	0	0	**1**	1	1	1	1	2
p	0	0	**1**	1	1	1	1	2
e	0	0	**1**	**1**	**1**	1	1	2
r	0	0	1	1	1	**2**	**2**	2
o	0	0	1	1	1	2	2	**3**

“rro”

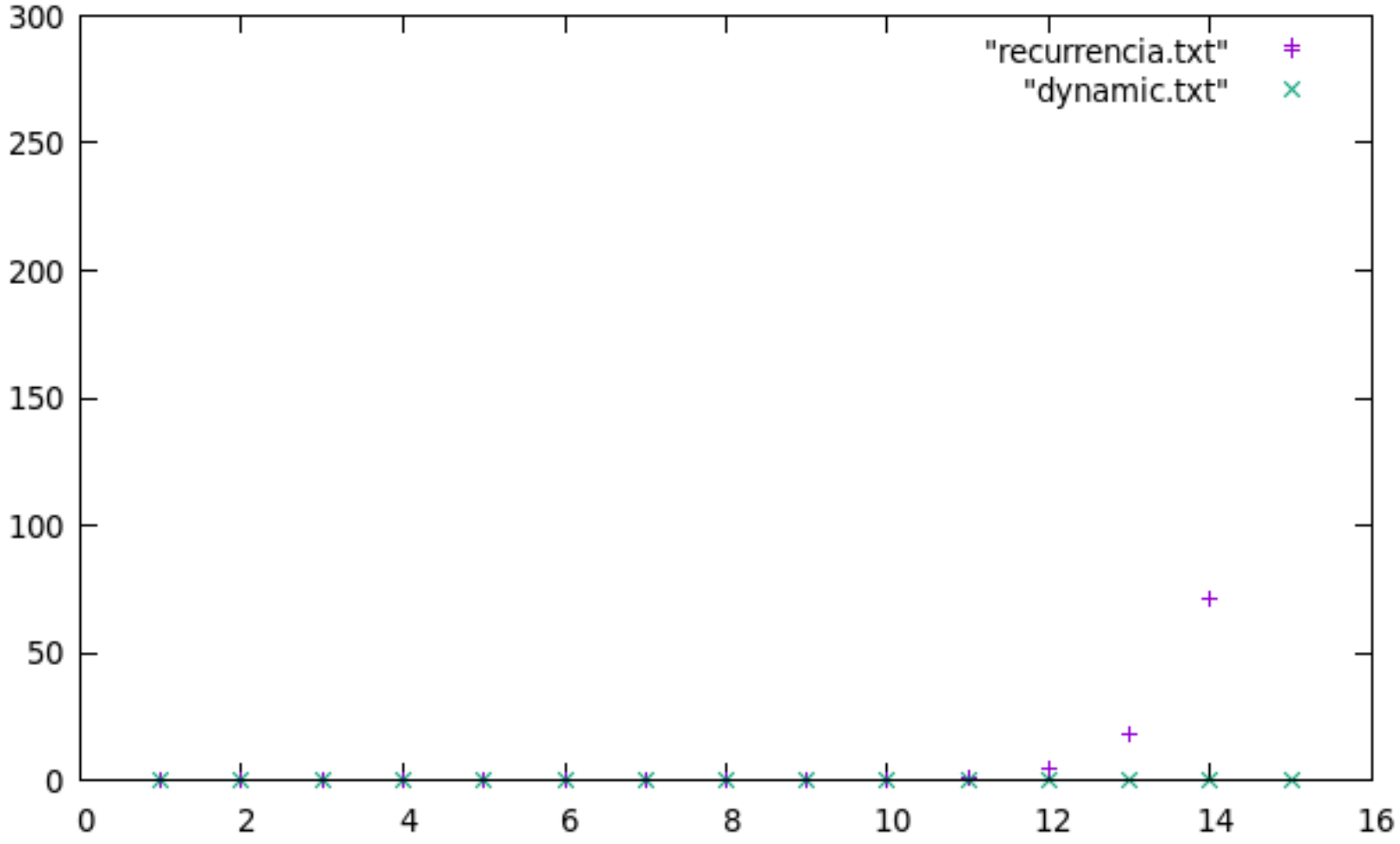
(rope**ro**, ar**ma**ri**o**)

Análisis empírico



Datos empíricos: algoritmo dinámico

Comparación: dinámico vs. fuerza bruta



Comparación: algoritmo dinámico vs fuerza bruta

Conclusiones

Conclusiones

Con esta práctica, hemos aprendido a crear algoritmos de programación dinámica para resolver problemas encadenados que podemos optimizar, comprobando las condiciones que han de verificar.

Al contrario de algoritmos simples de recurrencia en cada iteración no hay que volver a comprobar todas las soluciones posibles, ya que hemos guardado las mejores soluciones anteriores.

Especial relevancia para encontrar solución **óptima**.