Eidgenössische
Technische Hochschule
Zürich

Ecole polytechnique fédérale de Zurich
Politecnico federale di Zurigo
Federal Institute of Technology at Zurich

Institut für Theoretische Informatik
Peter Widmayer
Thomas Tschager
Antonis Thomas

20. April 2016

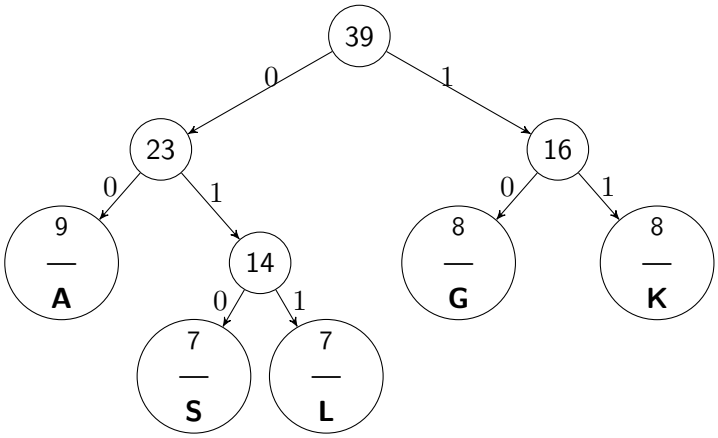
Datenstrukturen & Algorithmen Lösungen zu Blatt 7 FS 16

Lösung 7.1 *Häufigkeiten.*

Achtung: In der ursprünglichen Aufgabenstellung sind die Schlüssel K und L fälschlicherweise vertauscht.

a) Wir berechnen einen optimalen Kodierungsbaum mit dem Algorithmus von Huffman: Zu Beginn erstellen wir für jeden Schlüssel und seine Zugriffshäufigkeit einen trivialen Baum, der nur aus einem einzelnen Knoten besteht. Nun werden die beiden Bäume mit den geringsten, in der Wurzel gespeicherten, Zugriffshäufigkeiten zusammengefasst, indem ein neuer Knoten erstellt wird, der mit den beiden Wurzeln der Bäume verbunden wird. Die Zugriffshäufigkeit dieser neuen Wurzel ist die Summe der Zugriffshäufigkeiten der beiden zusammengefassten Bäume. Dieser Vorgang wird solange wiederholt, bis nur mehr ein Baum übrig bleibt.

Dadurch entsteht der folgende Kodierungsbaum. Jede Verzweigung nach links wird mit 0 markiert und jede Verzweigung nach rechts mit 1. Der Code eines Buchstaben entspricht den Markierungen entlang des Pfades von der Wurzel bis zum entsprechenden Blatt.



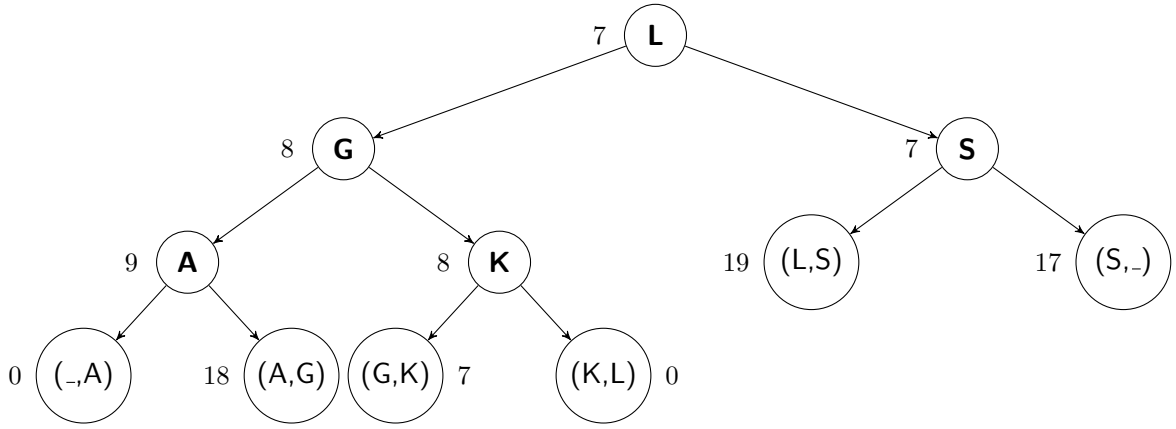
Das Wort `AGGLGSK` wird also mit 00 10 10 011 10 010 11 kodiert.

b) Wir berechnen einen optimalen Suchbaum mit dem in der Vorlesung vorgestellten dynamischen Programm. Wir bezeichnen die Schlüssel in aufsteigender Reihenfolge mit $k_1, \ldots, k_N$ und die Intervalle (ebenfalls in aufsteigender Reihenfolge) mit (k_j, k_{j+1}) für $0 \leq j \leq N$. Sei $T[i,j]$ der optimale Suchbaum für die Schlüssel $(k_i, k_{i+1}), k_{i+1}, \ldots, k_j, (k_j, k_{j+1})$. Wir bezeichnen mit $W[i,j]$ das Gewicht des Baumes $T[i,j]$ und mit $P[i,j]$ die gewichtete Pfadlänge von $T[i,j]$. Der Eintrag $r[i,j]$ enthält den Index der Wurzel des Baumes $T[i,j]$. Im Folgenden sind die Berechnungsergebnisse aller Einträge $W[i,j]$, $P[i,j]$ und $r[i,j]$ gegeben. Wir erhalten den unten dargestellten optimalen Suchbaum mit gewichteter Pfadlänge 235.

W	0	1	2	3	4	5
0	0	27	42	50	76	100
1		18	33	41	67	91
2			7	15	41	65
3				0	26	50
4					19	43
5						17

P	0	1	2	3	4	5
0	0	27	69	92	159	235
1		0	33	56	123	190
2			0	15	56	121
3				0	26	76
4					0	43
5						0

r	0	1	2	3	4	5
0	-	1	2	2	2	4
1		-	2	2	4	4
2			-	3	4	5
3				-	4	5
4					-	5
5						-



Lösung 7.2 *Palindrome aufzählen.*

a) *Definition der DP-Tabelle:* Wir verwenden eine $n \times n$-Tabelle T mit Einträgen, die entweder 0 oder 1 sind. Für $1 \leq i \leq j \leq n$ sei genau dann $T[i,j] = 1$, wenn $\langle A[i], \ldots, A[j] \rangle$ ein Palindrom ist.

Berechnung eines Eintrags: Wir unterscheiden drei Fälle.

1. Fall: $1 \leq i = j \leq n$. $A[i]$ ist ein Palindrom der Länge 1, also setzen wir $T[i,i] = 1$ für alle i, $1 \leq i \leq n$.

2. Fall: $1 \leq i \leq n$, $j = i + 1 \leq n$. Dann betrachten wir Palindrome der Länge 2, und wir setzen $T[i, i+1] = 1$ genau dann, wenn $A[i] = A[i+1]$ gilt.

3. Fall: $1 \leq i \leq n$, $i + 1 < j \leq n$. Sei $\langle A[i], \ldots, A[j] \rangle$ die betrachtete Zeichenkette. Diese ist nach Definition genau dann ein Palindrom, wenn $A[i] = A[j]$ gilt und zusätzlich $\langle A[i+1], \ldots, A[j-1] \rangle$ ein Palindrom ist. Wir setzen also genau dann $T[i,j] = 1$, wenn $A[i] = A[j]$ gilt und $T[i+1, j-1] = 1$ ist.

Berechnungsreihenfolge: Wir berechnen die Einträge $T[i,j]$ mit wachsender Differenz von $j - i$, starten also bei $T[i,i]$ für $1 \leq i \leq n$. Danach fahren wir fort mit der Berechnung von $T[i, i+1]$ für $1 \leq i \leq n-1$, danach mit $T[i, i+2]$ für $1 \leq i \leq n-2$, usw. Schlussendlich wird $T[1,n]$ berechnet.

Auslesen der Lösung: Wir iterieren über alle Einträge $T[i,j]$, $1 \leq i \leq j \leq n$, und geben genau dann (i,j) aus, wenn $T[i,j] = 1$ ist.

Laufzeit: Die Tabelle hat n^2 Einträge. Die Berechnung jedes Eintrags geht in Zeit $\mathcal{O}(1)$. Zum Auslesen der Lösung wird jeder Eintrag der Tabelle erneut betrachtet, und auch dort fällt nur Zeit $\mathcal{O}(1)$ pro Eintrag an. Die Gesamtlaufzeit beträgt daher $\mathcal{O}(n^2)$.

c) Wir betrachten die Einträge der Tabelle in umgekehrter Berechnungsreihenfolge, d.h. wir starten mit $T[1,n]$, betrachten danach $T[1,n-1]$ und $T[2,n]$, usw. Sobald wir einen Eintrag $T[i,j]$ mit Wert 1 finden, terminiert unser Verfahren, und wir geben $\langle A[i],\ldots,A[j]\rangle$ aus.

Wie vorher fällt bei der Betrachtung eines Eintrags nur Zeit $\mathcal{O}(1)$ an. Da es n^2 Einträge gibt und das ausgegebene Palindrom maximal n Zeichen lang ist, ergibt sich wie vorher eine Gesamtlaufzeit von $\mathcal{O}(n^2)$.

Lösung 7.3 *Längste gemeinsame Teilfolge (**Programmieraufgabe**).*

Der Algorithmus benutzt eine Tabelle $A[\cdot,\cdot]$ mit $n+1$ Zeilen und $m+1$ Spalten. Für $0 \le i \le n$ und $0 \le j \le m$ enthält der Eintrag $A[i,j]$ die Länge einer längsten gemeinsamen Teilfolge der Teilzeichenketten $a_1 \cdots a_i$ sowie $b_1 \cdots b_j$. Einträge mit $i = 0$ und $j = 0$ repräsentieren die leere Zeichenkette. Daher werden $A[i,0]$ und $A[0,j]$ für alle i, $0 \le i \le n$ und für alle j, $0 \le j \le m$, auf 0 gesetzt. Die verbleibenden Einträge können wie folgt berechnet werden:

$$A[i,j] = \begin{cases} A[i-1,j-1]+1 & \text{falls } a_i = b_j \\ \max\{A[i-1,j], A[i,j-1]\} & \text{ansonsten.} \end{cases} \tag{1}$$

Die Einträge können nach aufsteigenden Werten von i und für gleiche i nach aufsteigenden Werten von j berechnet werden. Nachdem die Tabelle ausgefüllt wurde, enthält der Eintrag $A[n,m]$ genau die Länge k einer längsten gemeinsamen Teilsequenz.

Untenstehend findet sich das Beispiel der Tabelle $A[i,j]$ für die Eingaben $A =$ "ROCK" und $B =$ "ROLL".

$A[i,j]$	∅	R	O	L	L
∅	0	0	0	0	0
R	0	1	1	1	1
O	0	1	2	2	2
C	0	1	2	2	2
K	0	1	2	2	2

```
import java.util.Scanner;

class Main {

    //takes as input two strings and computes the table
    //needed for the dynamic programming
    static int[][] computeTable(String a, String b) {
        int i, j;
        int n = a.length();
        int m = b.length();
        int [][] A = new int[n+1][m+1];

        // Initialization
        for(i = 0; i <= n; ++i)
            A[i][0] = 0;
        for(j = 0; j <= m; ++j)
            A[0][j] = 0;

        // Compute entries of the table
        for(i = 1; i <= n; ++i)
            for(j = 1; j <= m; ++j)
                if(a.charAt(i-1) == b.charAt(j-1))
                    A[i][j] = A[i-1][j-1] + 1;
                else
                    A[i][j] = Math.max(A[i-1][j], A[i][j-1]);

        return A;
```

```
    }

    //returns the length of the longest common subsequence
    static int longestCommonSubseqLen(int[][] table, int n, int m) {
        return table[n][m];
    }

    public static void main(String[] args) {
        int test, ntest;
        String s1, s2;
        Scanner sc = new Scanner(System.in);
        ntest = sc.nextInt();
        for(test = 0; test < ntest; ++test)
        {
            // Read two input strings.
            s1 = sc.next();
            s2 = sc.next();

            int [][]A = computeTable(s1, s2);
            int maxvalue = longestCommonSubseqLen(A, s1.length(), s2.length());

            System.out.println(maxvalue);
        }
    }
}
```