War schon: Wie ähnlich sind zwei Zeichenketten?

(diff, fc)

- Formalisierung: $s = s_1 s_2 \dots s_k$, $t = t_1 t_2 \dots t_l$ Strings

  (i) Definiere elementare Operationen: ins, del, swap und bestimme min # Operationen, um s nach t zu überführen (Levenshtein Abstand, Editierabstand)

  (ii) Finde die längste gemeinsame Teilfolge (LCS)

  $s_{i_1} s_{i_2} \dots s_{i_z}$ $\quad i_1 < i_2 < \dots < i_z$

  $t_{j_1} t_{j_2} \dots t_{j_z}$ $\quad j_1 < j_2 < \dots < j_z$

z.B. WEIHNACHTEN
NEUJAHR

LCS: NAH (oder EAH)

Wie bestimmt man die LCS? Zuerst: bestimme nur die Länge der LCS:

Rekursion von hinten

- Wenn beide Strings mit dem gleichen Symbol aufhören, tue es in die LCS, sonst streiche eins der Zeichen und rekursiere.

Länge

$$LLCS(s_1 s_2 s_3 \dots s_k, t_1 t_2 \dots t_l) = \begin{cases} 0 & \text{, wenn } k=0 \text{ oder } l=0 \\ 1 + LLCS(s_1 \dots s_{k-1}, t_1 \dots t_{l-1}) & \text{wenn } s_k = t_l \\ \max\{LLCS(s_1 \dots s_k, t_1 \dots t_{l-1}), LLCS(s_1 \dots s_{k-1}, t_1 \dots t_l)\} & \text{sonst} \end{cases}$$

⇒ direkte Implementierung der Rekursion ist ineffizient!

Lösung: Dynamisches Programmieren

- verwende Tabelle LLCS ($k+1 \times l+1$ Array)
- fülle Tabelle sukzessive auf

```
for i:=0 to k
    LLCS[0,i] ← 0
for j:=0 to l
    LLCS[j,0] ← 0
for i=1 to k
    for j=1 to l
        if s_i = t_j
            LLCS[j,i] ← 1+LLCS[j-1,i-1]
        else
            LLCS[j,i] ← max{LLCS[j-1,i], LLCS[j,i-1]}
```

Laufzeit $O(k \cdot l)$

Beispiel: $s =$ HUND

$t =$ FUNKE

	j\i	0	1 (H)	2 (U)	3 (N)	4 (D)
	0	0	0	0	0	0
F	1	0	0	0	0	0
U	2	0	0	1	1	1
N	3	0	0	1	2	2
K	4	0	0	1	2	2
E	5	0	0	1	2	2

Das Ergebnis steht in LLCS[5,4]

Um die LCS selbst zu bestimmen, folge den Pfeilen von LLCS[5,4] nach LLCS[0,0] (die Pfeile geben an welche Vorgängerwerte gewählt wurden)

Gib das Zeichen bei jedem ↖ Pfeil aus.

Bemerkungen:

- In der Implementierung kann man entweder die Pfeile mitspeichern (in einem zweiten Array) oder hinterher rekonstruieren
- Laufzeit $O(\overset{k}{n} \cdot \overset{l}{m})$ nicht so gut, aber mit ein paar Heuristiken und geschickten umdefinieren des Problems kann man diff implementieren.

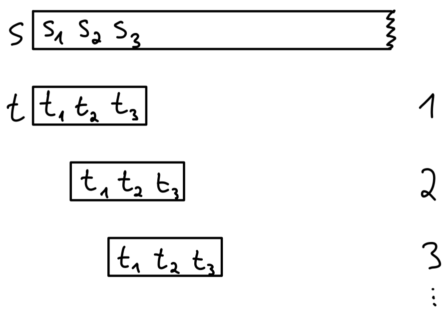
- Mit dyn. Programmieren kann man viel mehr Probleme lösen: DNA-Sequenzierung, Editierabstand, Fibonacci-Zahlen, Viterbi-Algorithmus, CYK

String-Suche

gegeben: zwei Strings $s = s_1 s_2 s_3 \dots s_n$
$t = t_1 t_2 \dots t_m$ $\quad m \leq n$

Kommt t in s vor?

Naiver Algorithmus:




```
for i = 1 to n-m+1
    j ← 1
    while (j ≤ m and s_{i+j-1} = t_j)
        j++;
    if (j = m+1)
        return i
return -1
```

Laufzeit $O(m \cdot n)$ nicht so toll :(

Geht es besser? - JA!

Eine einfache Idee: Schreibe den Code um:

```
for i := 1 to n-m+1
    if h(s[i...i+m-1]) = h(t)
        return i
```

Hashfunktion

kann man diesen Test beschleunigen?

Trick: Verwende Hashfunktionen!

Idee: Durch die Hashfunktion reduziert sich die Zeit für den Vergleich auf $O(1)$. $h(t)$ muss nur einmal berechnet werden

Aber, neues Problem: wie findet man $h(s[i \dots i+m-1])$ schnell?

$\to$ hängt von h ab, gute Wahl:

$$h(s[i \dots i+m-1]) = \sum_{j=0}^{m-1} (|\Sigma|^{m-1-j} s_{i+j}) \bmod p$$

Interpretiere $s[\dots]$ als Zahl, rechne modulo Primzahl p.

Tolle Eigenschaft: wie kommt man von $h(s[i \dots i+m-1])$ zu $h(s[i+1 \dots i+m])$?

$h(s[i+1 \dots i+m]) = (|\Sigma| h(s[i \dots i+m-1]) - |\Sigma|^m s_i + s_{i+m}) \bmod p$ $O(1)$ Operationen!

Brauchen also nun $O(m+n)$ Zeit! Aber: Kollisionen

Aus $h(s[\dots]) = h(t)$ folgt nicht $s[\dots] = t$!!!

Daher: Müssen trotzdem noch $s[\dots]$ und t vergleichen, in $O(m)$ Zeit. Wenn h oft kollidiert bringt das nichts. Worst case Laufzeit immer noch $O(mn)$. Aber: in der Praxis wesentlich schneller. Wenn p zufällig gewählt ist, kann man zeigen, dass die erwartete Laufzeit $O(m+n)$ ist. wähle $p \in \{1, \dots, n^2 m \log n^2 m\}$