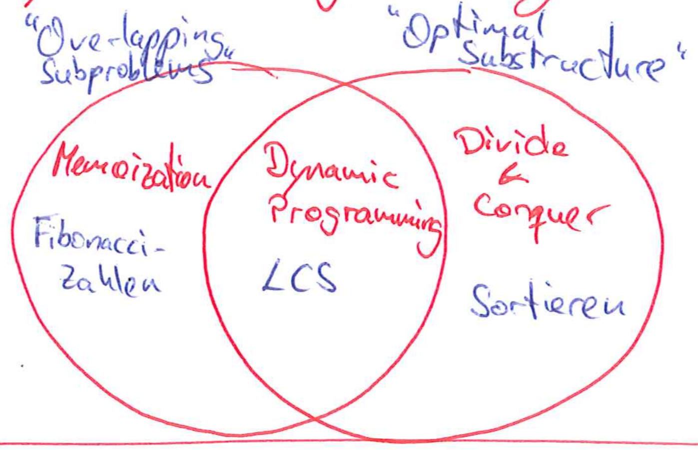
Übung Nr. 3 17.05.17

(1)

Heute:

1. Dynamic Programming
   - Wann ist es anwendbar?
2. Matrix Chain Multiplication
3. Longest Common Subsequence (LCS)

1. Dynamic Programming



Overlapping Subproblems:

Subprobleme tauchen mehrfach auf.

Beispiel: Fibonacci-Zahlen

$$F_n = F_{n-1} + F_{n-2} \ , \ F_0 = 0, \ F_1 = 1$$

Rekursiver Ansatz ist schlecht! → ~~Rekursive~~ Exponentielle Laufzeit.

Besser: "Memoization"

Merke Teilergebnisse!

zB so:

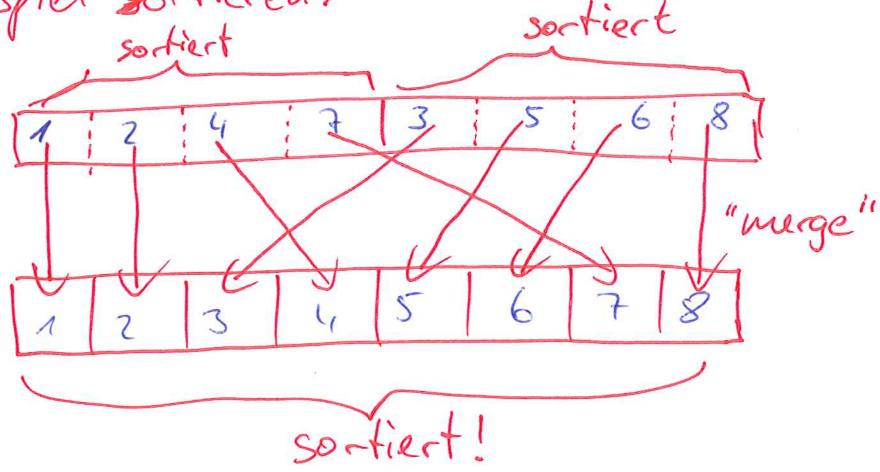
```
function Fib(i)
    F[0] = 0
    F[1] = 1
    for i=2 to n do
        F[i] = F[i-1] + F[i-2]
    endfor
end function
```

optimal substructure: ②

Eine optimale Lösung kann effizient von optimalen ~~Lösung~~ Teillösungen konstruiert werden.

Stichwort: "Divide & Conquer"

Beispiel Sortieren:



Besitzt ein Problem beide Eigenschaften, kann man Dynamic Programming benutzen.

Welche Probleme gibt es?

Schon gesehen: Knapsack, Subset Sum

Auf dem Aufgabenblatt:

2. Matrix Chain Multiplication

Eine Matrix M besteht aus n Zeilen und m Spalten und ~~mit~~ n·m Einträgen oder kurz M ist eine ~~n×m-Matrix~~ n×m-Matrix

Bei Multiplikation von zwei Matrizen M_0 ~~und~~ und M_1 bekommen wir eine Matrix M mit Dimension

$d_0 \times d_1$ $\quad$ $d_1 \times d_2$ $\quad$ $d_0 \times d_2$.

~~Vorraussetzung: $m_0 = n_1$!~~

Wie funktioniert eine solche Multiplikation?

Am Beispiel:

$5\cdot 1+3\cdot 0+2\cdot 1 \rightarrow$ 3 Multiplikationen!

$$\underbrace{\begin{pmatrix} 5 & 3 & 2 \\ 7 & 1 & 2 \\ 1 & 3 & 4 \\ 2 & 5 & 6 \\ 4 & 2 & 1 \end{pmatrix}}_{5\times 3} \cdot \underbrace{\begin{pmatrix} 1 & 0 & 2 & 3 \\ 0 & 2 & 1 & 5 \\ 1 & 4 & 0 & 1 \end{pmatrix}}_{3\times 4} = \underbrace{\begin{pmatrix} 7 & 14 & 13 & 32 \\ 9 & 10 & 15 & 28 \\ 5 & 22 & 5 & 22 \\ 8 & 34 & 9 & 37 \\ 5 & 12 & 10 & 23 \end{pmatrix}}_{5\times 4}$$

$5\times 4 \rightarrow 5\cdot 3\cdot 4$ Multiplikationen!

Allgemein:

Einzelmultiplikation

$$M[i][j] = \sum_{k=1}^{d_1} M_0[i][k]\cdot M_1[k][j]\text{, für alle } 1\le i\le d_0 \text{ und } 1\le j\le d_2$$

Man ~~Ich~~ benötigt also $d_0\cdot d_1\cdot d_2$ viele Einzelmultiplikationen.

Multiplikation von Matrizen ist assoziativ, also:

$$(M_0\cdot M_1)\cdot M_2 = M_0\cdot(M_1\cdot M_2)$$

Sei nun $\langle M_1,\dots,M_n\rangle$ Folge von Matrizen mit

M_i ist $\underbrace{d_i\times d_{i+1}}_{\text{Dadurch funktioniert die Multiplikation!}}$-Matrix.

Frage: In welcher Reihenfolge muss man ~~ich~~ die Matrizen multiplizieren, damit man ~~ich~~ möglichst wenige Einzelmult. hat?

Also: Wie klammern?

~~Antwort:~~ über Dynamic Programming bestimmen!

Wie baut ~~ich~~ man ein DP auf?

3. Longest Common Subsequence

- Ähnlichkeit von Strings (Wörtern), z.B. in der DNA

Gegeben: • Alphabet Σ
• Sequenzen
$X = x_1 \dots x_n$ mit $x_i \in \Sigma$ für $i \in \{1, \dots, n\}$
$Y = y_1 \dots y_m$ mit $y_i \in \Sigma$ für $i \in \{1, \dots, m\}$

Gesucht: Längste Teilsequenz T, die in X und Y vorkommen.

Eine Teilsequenz eines Wortes/Strings entsteht durch weglassen von Buchstaben. z.B. ist ACTG eine TS von <u>A</u><u>C</u>CTA<u>T</u>AT<u>G</u>TT

Definition: $LCS(X,Y) :=$ längste TS von X und Y

Bsp $X = ACCTATATGTT$
$Y = CATGACATTGA$ $\Big\}\overset{LCS}{=}$ CATATG

Vorschläge?

Wie löst man LCS mit DP?

Überlegung: Irgendwas über die Länge der Wörter.
→ Betrachte Teilwörter bis zum i-ten/j-ten Buchstaben!

$X^i = x_1 \dots x_i$; $Y^j = y_1 \dots y_j$

Zunächst: Initialisierung. Was passiert, wenn ein Wort leer ist?
→ keine LCS möglich!
→ $LCS(X^i, Y^j) = 0$, falls $i = 0$ oder $j = 0$

Und weiter? Welche Möglichkeiten existieren?

1. Ignoriere den letzten Buchstaben von X
2. Ignoriere den letzten Buchstaben von Y
3. Ignoriere von X und Y den letzten Buchstaben, und falls ~~letzten~~ ignorierte Buchstaben übereinstimmen erhöhe Zähler um 1!

Also:

$$LCS(x^i, y^j) = \max\{LCS(x^{i-1}, y^j), LCS(x^i, y^{j-1}), LCS(x^{i-1}, y^{j-1}) + M(x_i, y_j)\}$$

wobei $M(x_i, y_j) = \begin{cases} 1, & \text{falls } x_i = y_j \\ 0, & \text{sonst} \end{cases}$

Das liefert den Algorithmus:

```
LCS(X,Y) {
    n := |X|    } Länge von X und Y
    m := |Y|    }
    init LCS[n+1][m+1]    -> Erstelle 2-D Array!
(1) for i=0 to n do                     }
        LCS[i][0] = 0                   } Init
(2) for j=0 to m do                     }
        LCS[0][j] = 0                   }
(3) for i=1 to n do
        for j=1 to m do
            max = LCS[i-1][j]                              Fall 1
            if (max < LCS[i][j-1]) then max = LCS[i][j-1]  Fall 2
            if (max < LCS[i-1][j-1]+1 und x_i = y_i) then
                max = LCS[i-1][j-1] + 1                    Fall 3
            LCS[i][j] = max
```

Laufzeit: $\underbrace{\Theta(n)}_{1} + \underbrace{\Theta(m)}_{2} + \underbrace{\Theta(nm)}_{3} = \Theta(nm)$

Am Beispiel:

y \ x	∅	A	C	C	T	A	T	A	T	G	T	T
∅	0	0	0	0	0	0	0	0	0	0	0	0
C	0	0	1	1	1	1	1	1	1	1	1	1
A	0	1	1	1	1	2	2	2	2	2	2	2
T	0	1	1	1	2	2	3	3	3	3	3	3
G	0	1	1	1	2	2	3	3	3	4	4	4
A	0	1	1	1	2	3	3	4	4	4	4	4
C	0	1	2	2	2	3	3	4	4	4	4	4
A	0	1	2	2	2	3	3	4	4	4	4	4
T	0	1	2	2	3	3	4	4	5	5	5	5
T	0	1	2	2	3	3	4	4	5	5	6	6
G	0	1	2	2	3	3	4	4	5	6	6	6
A	0	1	2	2	3	4	4	5	5	6	6	6

← init

↑ init

Jetzt kennen wir die Länge von ~~der~~ LCS(x,y).

Man kann den Algorithmus leicht ändern, um auch die LCS selbst zu bekommen!

Lösung vom Beispiel: CATATG