18. Dynamisches Programmieren

- Dynamische Programmierung wie gierige Algorithmen eine Algorithmenmethode, um *Optimierungsprobleme* zu lösen.
- Wie Divide&Conquer berechnet Dynamische Programmierung Lösung eines Problems aus Lösungen zu Teilproblemen.
- Lösungen zu Teilproblemen werden *nicht* rekursiv gelöst.
- Lösungen zu Teilproblemen werden iterativ beginnend mit den Lösungen der kleinsten Teilprobleme berechnet (*bottom-up*).

Dynamisches Programmieren (2)

- Wird häufig angewandt, wenn rekursiver Ansatz Lösungen zu immer wieder denselben Teilproblemen lösen wird.
- Lernen drei Beispiele kennen: *Längste gemeinsame Teilfolge, optimale Suchbäume, Rucksackproblem.*

Längste gemeinsame Teilfolge (1)

- Seien $X = (x_1, \ldots, x_m)$ und $Y = (y_1, \ldots, y_n)$ zwei Teilfolgen, wobei $x_i, y_j \in A$ für ein endliches Alphabet A. Dann heisst Y *Teilfolge* von X, wenn es aufsteigend sortierte Indizes $i_1, \ldots, i_n$ gibt mit $x_{i_j} = y_j$ für $j = 1, \ldots, n$.

- *Beispiel:* Y=(B,C,A,C) ist Teilfolge von X=(A,B,A,C,A,B,C). Wähle $(i_1, i_2, i_3, i_4) = (2,4,5,7)$.

- Sind X,Y,Z Folgen über A, so heisst Z *gemeinsame Teilfolge* von X und Y, wenn Z Teilfolge sowohl von X als auch von Y ist.

- *Beispiel:* Z=(B,C,A,C) ist gemeinsame Teilfolge von X=(A,B,A,C,A,B,C) und Y=(B,A,C,C,A,B,B,C).

Längste gemeinsame Teilfolge (2)

- Z heisst längste gemeinsame Teilfolge von X und Y, wenn Z gemeinsame Teilfolge von X und Y ist und es keine andere gemeinsame Teilfolge von X und Y gibt, die größere Länge als Z besitzt.

- *Beispiel:* Z=(B,C,A,C) ist nicht längste gemeinsame Teilfolge von X=(A,B,A,C,A,B,C) und Y=(B,A,C,C,A,B,B,C). Denn (B,A,C,A,C) ist eine längere gemeinsame Teilfolge von X und Y.

- Beim Problem Längste-Gemeinsame-Teilfolge (LCS) sind als Eingabe zwei Teilfolgen $X = (x_1, \dots, x_m)$ und $Y = (y_1, \dots, y_n)$ gegeben. Gesucht ist dann eine längste gemeinsame Teilfolge von X und Y.

Beispiel längste gemeinsame Teilfolge

Folge *X*

A	*B*	*C*	*B*	*D*	*A*	*B*

Folge *Y*

B	*D*	*C*	*A*	*B*	*A*

Folge *X*

A	*B*	*C*	*B*	*D*	*A*	*B*

Folge *Y*

B	*D*	*C*	*A*	*B*	*A*

Struktur von LCS

Definition 18.1: Sei $X=(x_1,\ldots,x_m)$ eine beliebige Folge. Für $i=0,1,\ldots,m$ ist der i-te Präfix von X definiert als $X_i=(x_1,\ldots,x_i)$. Der i-te Präfix von i besteht also aus den ersten i Symbolen von X. Der 0-te Präfix ist die leere Folge.

Satz 18.2: Seien $X=(x_1,\ldots,x_m)$ und $Y=(y_1,\ldots,y_n)$ beliebige Folgen und sei $Z=(z_1,\ldots,z_k)$ eine längste gemeinsame Teilfolge von X und Y. Dann gilt

1. Ist $x_m=y_n$, dann ist $z_k=x_m=y_n$ und Z_{k-1} ist eine längste gemeinsame Teilfolge von X_{m-1} und Y_{n-1}.
2. Ist $x_m\neq y_n$ und $z_k\neq x_m$, dann ist Z eine längste gemeinsame Teilfolge von X_{m-1} und Y.
3. Ist $x_m\neq y_n$ und $z_k\neq y_n$, dann ist Z eine längste gemeinsame Teilfolge von X und Y_{n-1}.

Rekursion für Länge von LCS

Lemma 18.3: Sei $c[i,j]$ die Länge einer längsten gemeinsamen Teilfolge des i-ten Präfix X_i von X und des j-ten Präfix Y_j von Y. Dann gilt:

$$c[i,j] = \begin{cases} 0, & \text{falls } i = 0 \text{ oder } j = 0 \\ c[i-1, j-1] + 1, & \text{falls } i, j > 0 \text{ und } x_i = y_j \\ \max\{c[i-1, j], c[i, j-1]\}, & \text{falls } i, j > 0 \text{ und } x_i \neq y_j \end{cases}.$$

Beobachtung: Rekursive Berechnung der $c[i,j]$ würde zu Berechnung immer wieder derselben Werte führen. Dieses ist ineffizient. Berechnen daher die Werte $c[i,j]$ iterativ, nämlich zeilenweise.

Berechnung der Werte *c[i,j]*

```
LCS-Length(X,Y)
 1  m ← length(X)
 2  n ← length(Y)
 3  for i ← 0 to m
 4       do c[i,0] ← 0
 5  for j ← 0 to n
 6       do c[0,j] ← 0
 7  for i ← 1 to m
 8       do for j ← 1 to n
 9              do if x_i = y_j
10                    then c[i,j] ← c[i-1,j-1]+1
11                         b[i,j] ← "↖"
12                    else if c[i-1,j] ≥ c[i,j-1]
13                         then c[i,j] ← c[i-1,j]
14                              b[i,j] ← "↑"
15                         else c[i,j] ← c[i,j-1]
16                              b[i,j] ← "←"
17 return b,c
```

b[i, j] speichert Informationen zur späteren Berechnung einer längsten gemeinsamen Teilfolge.

Bespieltabellen $c[i,j]$ und $b[i,j]$

i	j	0	1	2	3	4	5	6
	x_i \ y_j		*B*	*D*	*C*	*A*	*B*	*A*
0		**0**	**0**	**0**	**0**	**0**	**0**	**0**
1	*A*	**0**	↑ **0**	↑ **0**	↑ **0**	↖ **1**	← **1**	↖ **1**
2	*B*	**0**	↖ **1**	← **1**	← **1**	↑ **1**	↖ **2**	← **2**
3	*C*	**0**	↑ **1**	↑ **1**	↖ **2**	← **2**	↑ **2**	↑ **2**
4	*B*	**0**	↖ **1**	↑ **1**	↑ **2**	↑ **2**	↖ **3**	← **3**
5	*D*	**0**	↑ **1**	↖ **2**	↑ **2**	↑ **2**	↑ **3**	↑ **3**
6	*A*	**0**	↑ **1**	↑ **2**	↑ **2**	↖ **3**	↑ **3**	↖ **4**
7	*B*	**0**	↖ **1**	↑ **2**	↑ **2**	↑ **3**	↖ **4**	↑ **4**

Berechnung von LCS aus *c* und *b*

$\text{Print-LCS}(b, X, i, j)$

1 **if** $i = 0 \vee j = 0$
2 **then return**
3 **if** $b[i, j] = "\nwarrow"$
4 **then** $\text{Print-LCS}(b, X, i-1, j-1)$
5 **print** x_i
6 **else if** $b[i, j] = "\uparrow"$
7 **then** $\text{Print-LCS}(b, X, i-1, j)$
8 **else** $\text{Print-LCS}(b, X, i, j-1)$

Laufzeit von LCS-Length und LCS-Print

Lemma 18.4: Algorithmus LCS-Length hat Laufzeit **O**(nm), wenn die Folgen X,Y Länge n und m haben.

Lemma 18.5: Algorithmus LCS-Print hat Laufzeit **O**(n+m), wenn die Folgen X,Y Länge n und m haben.

Berechnung von LCS aus *c,b* - Illustration

i	x_i \ y_j	*j* = 0	1: *B*	2: *D*	3: *C*	4: *A*	5: *B*	6: *A*
0		**0**	**0**	**0**	**0**	**0**	**0**	**0**
1	*A*	**0**	↑ **0**	↑ **0**	↑ **0**	↖ **1**	← **1**	↖ **1**
2	*B*	**0**	↖ **1**	← **1**	← **1**	↑ **1**	↖ **2**	← **2**
3	*C*	**0**	↑ **1**	↑ **1**	↖ **2**	← **2**	↑ **2**	↑ **2**
4	*B*	**0**	↖ **1**	↑ **1**	↑ **2**	↑ **2**	↖ **3**	← **3**
5	*D*	**0**	↑ **1**	↖ **2**	↑ **2**	↑ **2**	↑ **3**	↑ **3**
6	*A*	**0**	↑ **1**	↑ **2**	↑ **2**	↖ **3**	↑ **3**	↖ **4**
7	*B*	**0**	↖ **1**	↑ **2**	↑ **2**	↑ **3**	↖ **4**	↑ **4**

Dynamisches Programmieren - Struktur

1. Bestimme rekursive Struktur einer optimalen Lösung.

2. Entwerfe rekursive Methode zur Bestimmung des Werts einer optimalen Lösung.

3. Transformiere rekursiv Methode in eine iterative (bottom-up) Methode zur Bestimmung des Werts einer optimalen Lösung.

4. Bestimmen aus dem Wert einer optimalen Lösung und in 3. ebenfalls berechneten Zusatzinformationen eine optimale Lösung.

Optimale binäre Suchbäume (1)

- Gegeben ist Folge $K=(k_1,\ldots,k_n)$ mit $k_1<k_1<\cdots<k_n$, sowie Werte $p_1,\ldots,p_n; q_0,q_1,\ldots,q_n \geq 0$ mit

$$\sum_{i=1}^{n} p_i + \sum_{j=0}^{n} q_j = 1.$$

- Bedeutung der p_i, q_j: Nach Schlüssel k_i wird mit Wahrscheinlichkeit p_i gesucht. Nach einem nicht existierenden Schlüssel im offenen Intervall (k_i,k_{i+1}) wird mit Wahrscheinlichkeit q_i gesucht. Dabei sei $k_0=-\infty$ und $k_{n+1}=\infty$.

- Repräsentieren nicht existierende Schlüssel im Intervall (k_i,k_{i+1}) durch dummy Schlüssel $d_i, i=0,1,\ldots,n$.

Optimale binäre Suchbäume (2)

- Gesucht ist binärer Suchbaum T_i, der erwarteten Aufwand pro Suche minimiert.

- Hat k_i oder d_i Tiefe t im Suchbaum T, so ist der Aufwand für eine Suche nach k_i bzw. d_i genau t+1.

- Sei $\text{depth}_T(k_i), \text{depth}_T(d_i)$ Tiefe von Schlüssel k_i bzw. d_i in einem Suchbaum T. Dann ist erwarteter Aufwand einer Suche in T gegeben durch

$$E[\text{Suchaufwand}] = \sum_{i=1}^{n} (\text{depth}_T(k_i)+1) \cdot p_i + \sum_{i=0}^{n} (\text{depth}_T(d_i)+1) \cdot q_i$$

$$= 1 + \sum_{i=1}^{n} \text{depth}_T(k_i) \cdot p_i + \sum_{i=0}^{n} \text{depth}_T(d_i) \cdot q_i$$

Nützliche Verallgemeinerung

- Verlangen nicht mehr, dass die Werte $p_1,\ldots,p_n; q_0,q_1,\ldots,q_n \geq 0$ eine Wahrscheinlichkeitsverteilung bilden. Erlauben also $\sum_{i=1}^{n} p_i + \sum_{j=0}^{n} q_j \neq 1$.

- Wollen aber weiterhin Suchbaum für die Schlüssel $k_1,\ldots,k_n$, so dass die Summe

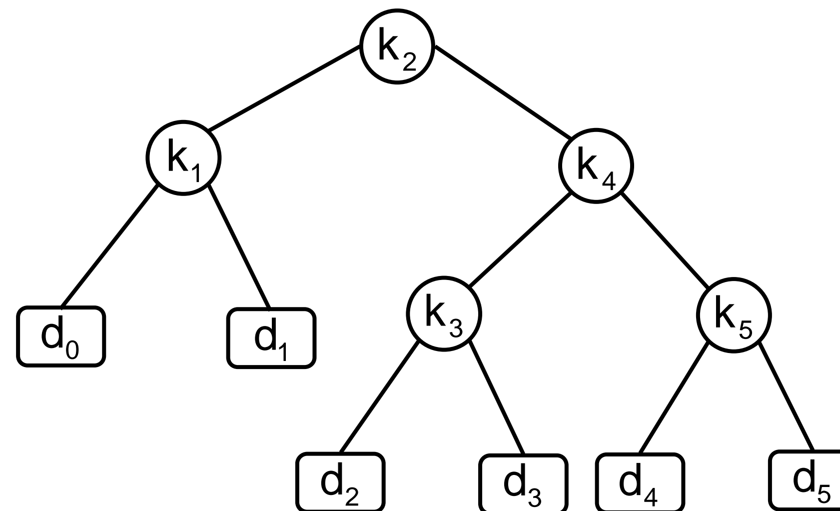
$$\sum_{i=1}^{n} (\text{depth}_T(k_i)+1)\cdot p_i + \sum_{i=0}^{n} (\text{depth}_T(d_i)+1)\cdot q_i$$

minimiert wird.

- Nennen diese Summe trotzdem erwarteten Suchaufwand.

Erwarteter Suchaufwand – Beispiele (1)

i	0	1	2	3	4	5
p_i		0,15	0,10	0,05	0,10	0,20
q_i	0,05	0,10	0,05	0,05	0,05	0,10



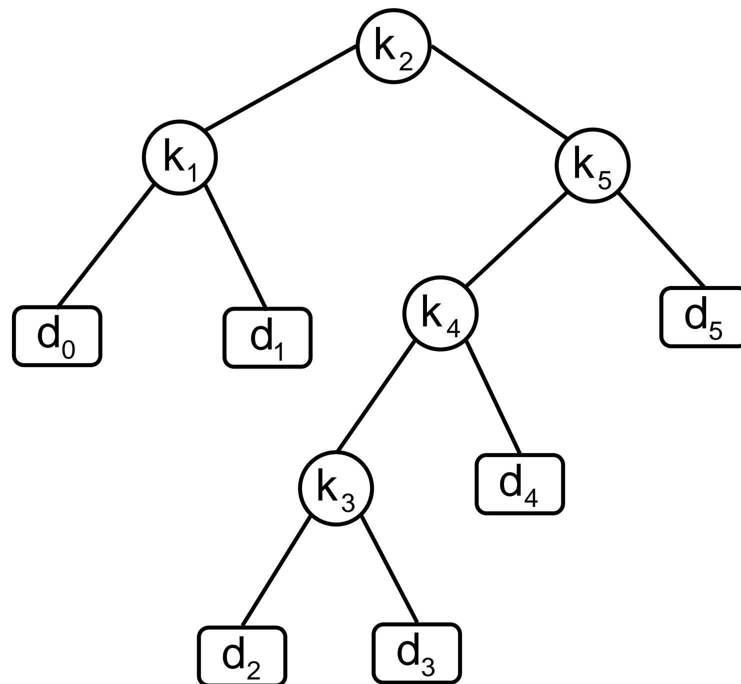
Erwarteter Suchaufwand – Beispiele (2)

Knoten	Tiefe	Wahrscheinlichkeit	Beitrag
k_1	1	0,15	0,30
k_2	0	0,10	0,10
k_3	2	0,05	0,15
k_4	1	0,10	0,20
k_5	2	0,20	0,60
d_0	2	0,05	0,15
d_1	2	0,10	0,30
d_2	3	0,05	0,20
d_3	3	0,05	0,20
d_4	3	0,05	0,20
d_5	3	0,10	0,40

Erwarteter Suchaufwand: 2,80

Erwarteter Suchaufwand – Beispiele (3)

i	0	1	2	3	4	5
p_i		0,15	0,10	0,05	0,10	0,20
q_i	0,05	0,10	0,05	0,05	0,05	0,10



Erwarteter Suchaufwand: 2,75

Struktur optimaler Lösung

Satz 18.5: Sei T ein optimaler Suchbaum für die Schlüsselfolge $K = (k_1,\ldots,k_n)$ mit Wahrscheinlichkeiten $p_1,\ldots,p_n; q_0, q_1,\ldots,q_n$. Enthält T den Teilbaum T' mit den Schlüsseln $k_i,\ldots,k_j$, dann ist T' in optimaler Suchbaum für die Schlüssel $k_i,\ldots,k_j$.

Rekursive optimale Lösung (1)

- Sei $e[i,j]$ der erwartete Suchaufwand für einen optimalen Suchbaum mit Schlüsseln $k_i,\ldots,k_j$.

- Ist k_r Wurzel eines optimalen Suchbaums für die Schlüssel $k_i,\ldots,k_j$, so gilt

$$e[i,j] = p_r + e[i,r-1] + w[i,r-1] + e[r+1,j] + w[r+1,j].$$

Dabei ist

$$w[i,j] := \sum_{k=i}^{j} p_k + \sum_{k=i-1}^{j} q_k .$$

Rekursive optimale Lösung (2)

- Es gilt :

$$w[i,j] = p_r + w[i,r-1] + w[r+1,j]$$

$$w[i,j] = p_j + q_j + w[i,j-1].$$

- Damit dann

$$e[i,j] = e[i,r-1] + e[r+1,j] + w[i,j].$$

- Schließlich folgt

$$e[i,j] = \begin{cases} q_{i-1}, & \text{falls } j = i-1 \\ \min\limits_{i \le r \le j}\{e[i,r-1] + e[r+1,j] + w[i,j]\}, & \text{falls } i \le j. \end{cases}$$

Berechnung der Werte e[i,j]

```
Optimal-BST(p,q,n)
 1  for i ← 1 to n+1
 2      do e[i,i-1] ← q_{i-1}
 3         w[i,i-1] ← q_{i-1}
 4  for l ← 1 to n
 5      do for i ← 1 to n-l+1
 6             do j ← i+l-1
 7                e[i,j] ← ∞
 8                w[i,j] ← w[i,j-1] + p_j + q_j
 9                for r ← i to j
10                    do t ← e[i,r-1] + e[r+1,j] + w[i,j]
11                       if t < e[i,j]
12                          then e[i,j] ← t
13                               root[i,j] ← r
14  return e, root
```

root[i, j] speichert Informationen zur späteren Berechnung eines opt-malen Suchbaums

Laufzeit von Optimal-BST

Satz 18.7: Optimal-BST hat bei einer Folge von n Schlüsseln Laufzeit $O(n^3)$.

Beispiel für Optimal-BST

i	0	1	2	3	4	5
p_i		0,15	0,10	0,05	0,10	0,20
q_i	0,05	0,10	0,05	0,05	0,05	0,10

e

i \ j	0	1	2	3	4	5
1	0,05	0,45	0,90	1,25	1,75	2,75
2		0,10	0,40	0,70	1,20	2,00
3			0,05	0,25	0,60	1,30
4				0,05	0,30	0,90
5					0,05	0,50
6						0,10

w

i \ j	0	1	2	3	4	5
1	0,05	0,30	0,45	0,55	0,70	1,00
2		0,10	0,25	0,35	0,50	0,80
3			0,05	0,15	0,30	0,60
4				0,05	0,20	0,50
5					0,05	0,35
6						0,10

root

i \ j	1	2	3	4	5
1	1	1	2	2	2
2		2	2	2	4
3			3	4	5
4				4	5
5					5

Dynamisches Programmieren - Struktur

1. Bestimme rekursive Struktur einer optimalen Lösung.
2. Entwerfe rekursive Methode zur Bestimmung des Werts einer optimalen Lösung.
3. Transformiere rekursiv Methode in eine iterative (bottom-up) Methode zur Bestimmung des Werts einer optimalen Lösung.
4. Bestimmen aus dem Wert einer optimalen Lösung und in 3. ebenfalls berechneten Zusatzinformationen eine optimale Lösung.

0/1-Rucksackproblem und dynamisches Programmieren

- Gegeben sind n Gegenstände $g_1,\ldots,g_n$ mit Werten $v_1,\ldots,v_n$ und Gewichten $w_1,\ldots,w_n$. Außerdem ist eine Gewichtsschranke G gegeben.
- Zulässige Lösungen sind Zahlen $a_1,\ldots,a_n \in \{0,1\}$ mit

$$\sum_{i=1}^{n} a_i w_i \leq G .$$

D.h. jeder Gegenstand muss entweder vollständig oder überhaupt nicht genommen werden.

- Gesucht ist eine zulässige Lösung mit möglichst großem Gesamtwert $\sum_{i=1}^{n} a_i v_i$.

Struktur einer optimalen Lösung

Definieren für i=1,…,n und $x \in Z$ den Wert $g[i,x]$ als das minimale Gesamtgewicht einer Teilmenge der ersten i Gegenstände mit Gesamtwert mindestens x.

Lemma 18.8: Der optimale Wert opt für eine Instanz des 0/1-Rucksackproblems ist gegeben durch

$$\max\{x : g[n,x] \leq G\}$$

Rekursive optimale Lösung

Lemma 18.9: Für alle i=0,…,n und alle ganzzahligen x gilt:

$$g[i,x] := \begin{cases} 0, & \text{falls } x \leq 0 \\ \infty, & \text{falls } x > 0, i = 0 \\ \min\{g[i-1,x], w_i + g[i-1,x-v_i]\}, & \text{falls } i,x > 0 \end{cases}$$

Berechnung der Werte *g*[*i,v*] bis zum Optimum

$Optimal - Knapsack(w, v, G)$

```
1  x ← 0
2  for i ← 0 to n
3        do g[i, x] ← 0
4  while g[n, x] ≤ G
5           do x ← x + 1
6               g[0, x] ← ∞
7               for i ← 1 to n
8                   do g[i, x] ← g[i − 1, x]
9                       if w_i + g[i − 1, max{x - v_i, 0}] < g[i, x]
10                          then g[i, x] ← w_i + g[i − 1, max{x - v_i, 0}]
11 return x - 1
```

Beispiel für Optimal-Knapsack

i	v_i	w_i
1	1	2
2	2	3
3	1	1
4	3	1

i \ *x*	0	1	2	3	4	5	6
0	**0**	∞	∞	∞	∞	∞	∞
1	**0**	**2**	∞	∞	∞	∞	∞
2	**0**	**2**	**3**	**5**	∞	∞	∞
3	**0**	**1**	**3**	**4**	**6**	∞	∞
4	**0**	**1**	**1**	**1**	**2**	**4**	**5**

Laufzeit von Optimal-Knapsack

Lemma 18.9: Bei n Gegenständen hat Optimal-Knapsack Laufzeit $O(n \cdot opt)$, wobei opt der Wert einer optimalen Lösung ist.

Dynamisches Programmieren - Struktur

1. Bestimme rekursive Struktur einer optimalen Lösung.
2. Entwerfe rekursive Methode zur Bestimmung des Werts einer optimalen Lösung.
3. Transformiere rekursiv Methode in eine iterative (bottom-up) Methode zur Bestimmung des Werts einer optimalen Lösung.
4. Bestimmen aus dem Wert einer optimalen Lösung und in 3. ebenfalls berechneten Zusatzinformationen eine optimale Lösung.