

Datenstrukturen & Algorithmen

Peppo Brambilla
Universität Bern
Frühling 2018

Übersicht

Dynamische Programmierung

- Einführung
- Schneiden von Eisenstangen
- Längste gemeinsame Teilsequenz
- Optimale binäre Suchbäume
- Eigenschaften der dynamischen Programmierung

Dynamische Programmierung

- Entwurfsstrategie, wie divide-and-conquer (teile-und-beherrsche)
 - „Programmierung“ bedeutet „Tabellieren“ (historischer Hintergrund)
- Unterschied zu Teile-und-beherrsche:
 - Teile-und-beherrsche: Teilprobleme sind **unabhängig**
 - Dynamische Programmierung: Teilprobleme bestehen ihrerseits aus **gemeinsamen** Teilproblemen

Dynamische Programmierung

- Lösung von Optimierungsproblemen
 - Finde eine Lösung mit optimalen Eigenschaften
 - Kleinster oder grösster Wert (Minimierung oder Maximierung)

Dynamische Programmierung

Vier Schritte, um Algorithmus zu entwickeln

1. Beschreibe Struktur einer optimalen Lösung
 - Zeige optimale Teilstruktur
2. Definiere Wert der optimalen Lösung rekursiv
3. Berechne Wert der optimalen Lösung „bottom-up“
 - Alternative: Memoisierung
4. Konstruiere optimale Lösung basierend auf den vorherigen Berechnungen

Dynamische Programmierung

Begriffe

- Optimale Teilstruktur
 - Überlappende Teilprobleme
 - Memoisierung
-
- Zusammenfassende Diskussion am Schluss nach Beispielen

Übersicht

Dynamische Programmierung

- Einführung
- Schneiden von Eisenstangen
- Längste gemeinsame Teilsequenz
- Optimale binäre Suchbäume
- Eigenschaften der dynamischen Programmierung

Schneiden von Eisenstangen

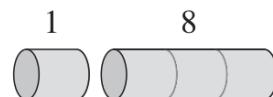
- Firma kauft lange Eisenstangen
 - Schneidet Eisenstangen in kurze Stäbe
 - Verkauft Stab der Länge i zum Preis p_i
 - Schnitt ist kostenlos
- **Problem:** Finde optimale Zerteilung, d.h. maximiere Gesamtpreis der Teilstücke
 - Bestimme anhand der Preise p_1, p_2, \dots, p_n für Stücke der Länge $i = 1, \dots, n$, wie eine Stange der Länge n zerteilt werden muss, um den maximalen Gesamtpreis r_n zu erzielen

Beispiel: Stab der Länge 4

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30



(a)



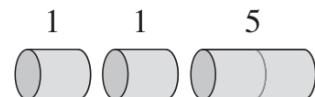
(b)



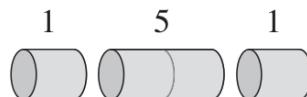
(c)



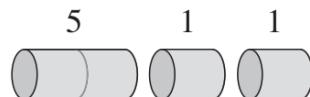
(d)



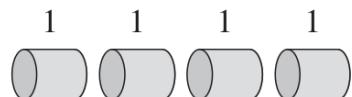
(e)



(f)



(g)



(h)

Optimum: Variante c, $r_4 = p_2 + p_2 = 10$

1. Struktur der optimalen Lösung

- Können optimale Lösung durch optimale Lösung von kürzeren Stäben ausdrücken:

$$r_n = \max(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \dots, r_{n-1} + r_1)$$

- p_n : kein Schnitt
 - $r_i + r_{n-i}$: schneide an Position i , zerteile in 2 Stäbe der Länge i und $n - i$, dann beide Stäbe optimal zerteilen.
- Noch einfacher: Lasse 1. Teilstück ganz

$$r_n = \max_{1 \leq i \leq n} (p_i + r_{n-i}) \quad \text{wobei } r_0 = 0$$

1. Struktur der optimalen Lösung

- Optimale Teilstruktur:
Optimale Lösung für ein Problem
(Zerteilung einer Stange der Länge n)
enthält optimale Lösung von
Teilproblemen (Zerteilung einer Stange
der Länge $n - i$)
- Benütze optimale Teilstruktur um optimale
Lösung zu einem Problem aus optimalen
Lösungen von Unterproblemen zu finden

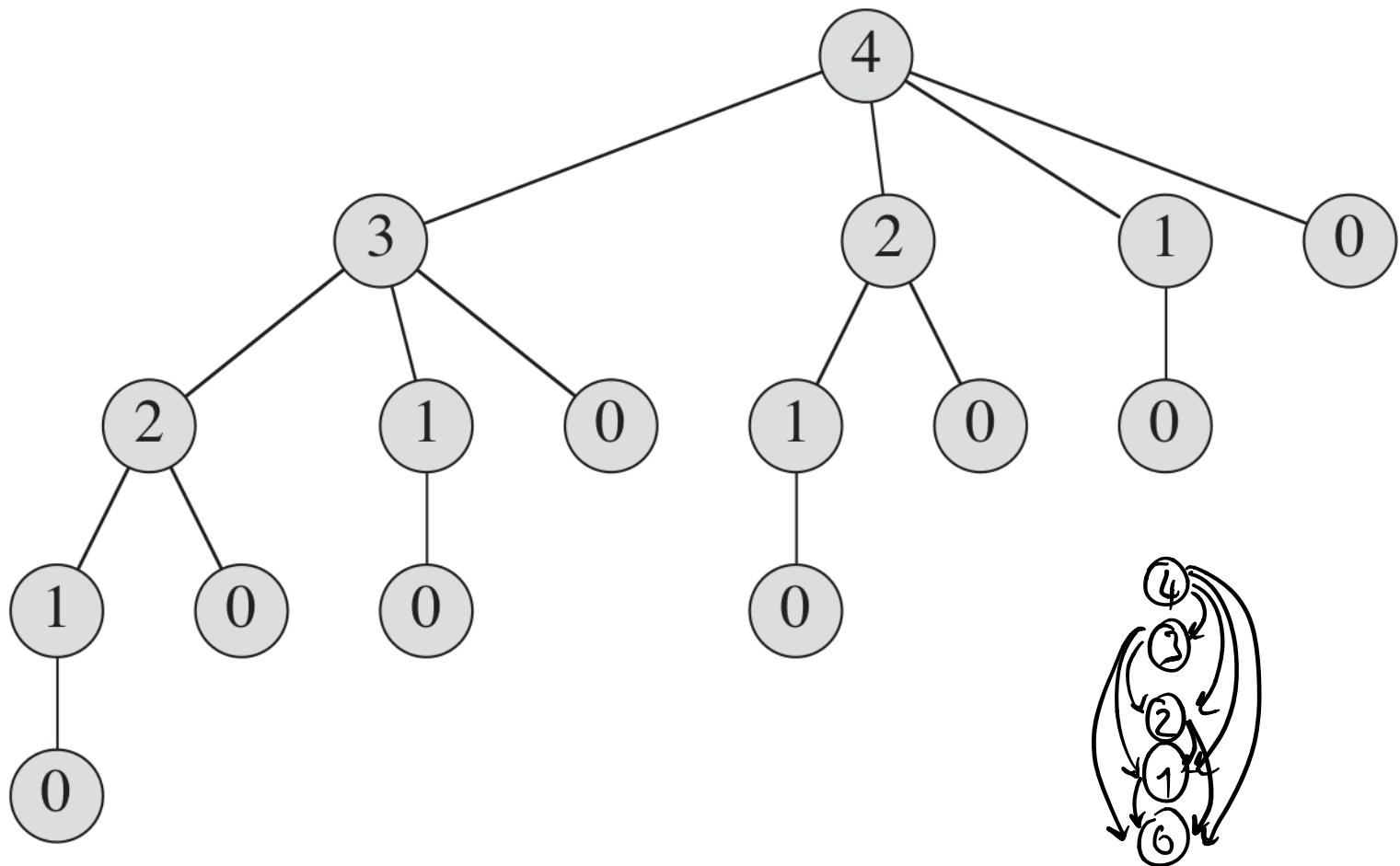
2. Rekursive Lösung

CUT-ROD(p, n)

```
1  if  $n == 0$ 
2      return 0
3   $q = -\infty$ 
4  for  $i = 1$  to  $n$ 
5       $q = \max(q, p[i] + \text{CUT-ROD}(p, n - i))$ 
6  return  $q$ 
```

- Zeigt exponentielles Laufzeitverhalten
- Grund: Teilprobleme werden mehrfach gelöst → Rekursionsbaum

Rekursionsbaum



Analyse der rekursiven Lösung

- Sei $T(n)$ Anzahl Aufrufe von CUT-ROD, die erfolgen beim Aufruf von $\text{CUT-ROD}(p, n)$.
Entspricht Anzahl Knoten im Unterbaum des Knotens n .

$$T(0) = 1$$

$$T(n) = 1 + \sum_{j=0}^{n-1} T(j)$$

- Es gilt: $T(n) = 2^n$ (siehe Übung)
- Exponentielle Laufzeit!

3. Bottom-up Algorithmus

- Problem abhängig von kleineren Teilproblemen
 - Löse Teilprobleme der Grösse nach
 - Speichere Ergebnis (\rightarrow Laufzeit-Speicher-Tradeoff)
 - Statt Rekursion: benutze gespeicherte Ergebnisse

BOTTOM-UP-CUT-ROD(p, n)

```
1 let  $r[0..n]$  be a new array
2  $r[0] = 0$ 
3 for  $j = 1$  to  $n$ 
4      $q = -\infty$ 
5     for  $i = 1$  to  $j$ 
6          $q = \max(q, p[i] + r[j - i])$ 
7          $r[j] = q$ 
8 return  $r[n]$ 
```

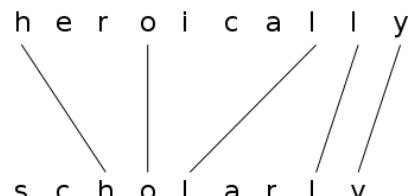
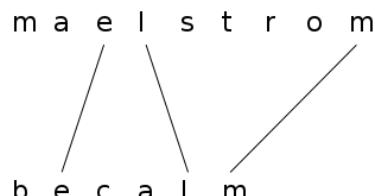
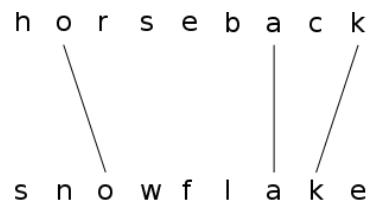
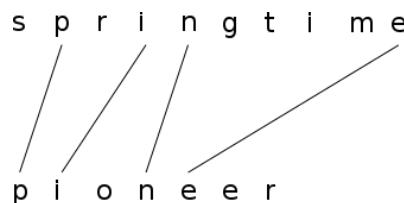
Übersicht

Dynamische Programmierung

- Einführung
- Schneiden von Eisenstangen
- Längste gemeinsame Teilsequenz
- Optimale binäre Suchbäume
- Eigenschaften der dynamischen Programmierung

Längste gemeinsame Teilsequenz

- Gegeben zwei Sequenzen $X = \langle x_1, \dots, x_m \rangle$ und $Y = \langle y_1, \dots, y_n \rangle$. Finde eine Sequenz von maximaler Länge, die in X und Y vorkommt
 - Elemente der Sequenz müssen nicht aufeinanderfolgend, jedoch in korrekter Reihenfolge in X und Y vorkommen
 - Longest common subsequence (LCS)
- Beispiele



Naive Lösung

- Für jede Teilsequenz von X , prüfe ob sie in Y vorkommt
- 2^m Teilsequenzen in X
 - Jeder Bitvektor der Länge m definiert eine Teilsequenz
- Überprüfen jeder Teilsequenz in $\Theta(n)$
 - Gesamte Sequenz Y muss durchlaufen werden
- Aufwand $\theta(n \cdot 2^m)$
- Exponentiell!

1. Optimale Teilstruktur

- Notation
 - X_i bezeichnet Präfix $\langle x_1, \dots, x_i \rangle$
 - Y_j bezeichnet Präfix $\langle y_1, \dots, y_j \rangle$
- Theorem

Sei $Z = \langle z_1, \dots, z_k \rangle$ LCS von X_m und Y_n

 1. Falls $x_m = y_n$, dann gilt $z_k = x_m = y_n$ und Z_{k-1} ist LCS von X_{m-1} und Y_{n-1}
 2. Falls $x_m \neq y_n$, dann bedeutet $z_k \neq x_m$, dass Z LCS von X_{m-1} und Y_n ist
 3. Falls $x_m \neq y_n$, dann bedeutet $z_k \neq y_n$, dass Z LCS von X_m und Y_{n-1} ist

Beweis Theorem

Sei $Z = \langle z_1, \dots, z_k \rangle$ LCS von X_m und Y_n (1)

1. Sei $x_m = y_n$

a. Zeige $z_k = x_m = y_n$

GgA: $z_k \neq x_m = y_n$

Dann: Z gem. Teilsequenz von X_{m-1} und Y_{n-1}

Also: $Z' = \langle z_1, \dots, z_k, x_m \rangle$ gem. Teilsequenz von X_m und Y_n . Widerspruch zu (1).

b. Zeige Z_{k-1} ist LCS von X_{m-1} und Y_{n-1}

- Gemeinsame Teilsequenz: klar
- Längste Teilsequenz: durch Gegenannahme

Sei W gem. Teilsequenz von X_{m-1} und Y_{n-1} länger als Z_{k-1} , d.h. Länge $W \geq k$

Dann ist $W' = \langle W, x_m \rangle$ gem. Teilsequenz von X_m und Y_n . Widerspruch zu (1), da Länge $W' \geq k + 1$

Beweis Theorem

Sei $Z = \langle z_1, \dots, z_k \rangle$ LCS von X_m und Y_n (1)

2. Sei $z_k \neq x_m$

Zeige Z ist LCS von X_{m-1} und Y_n

- Gemeinsame Teilsequenz: klar
- Längste Teilsequenz: durch Gegenannahme

Sei W gemeinsame Teilsequenz von X_{m-1} und Y_n länger als Z

Dann ist W auch gemeinsame Teilsequenz von X_m und Y_n . Widerspruch zu (1).

3. Sei $z_k \neq y_n$: analog zu 2.

Optimale Teilstruktur

- Eine LCS von zwei Sequenzen hat ein Präfix, welches eine LCS von Präfixen dieser Sequenzen ist
 - Jedes Präfix einer LCS von zwei Sequenzen ist eine LCS von Präfixen dieser Sequenzen
- Das Problem erfüllt das Kriterium der optimalen Teilstruktur

2. Rekursive Formulierung

- Idee
 - Berechne zuerst nur Länge der LCS
 - Rekonstruiere nachher die LCS selbst
- Sei $c[i, j]$ Länge der LCS von X_i und Y_j
- Gesucht $c[m, n]$

$$c[i, j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ c[i - 1, j - 1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j, \\ \max(c[i - 1, j], c[i, j - 1]) & \text{if } i, j > 0 \text{ and } x_i \neq y_j. \end{cases}$$

Rekursive Formulierung

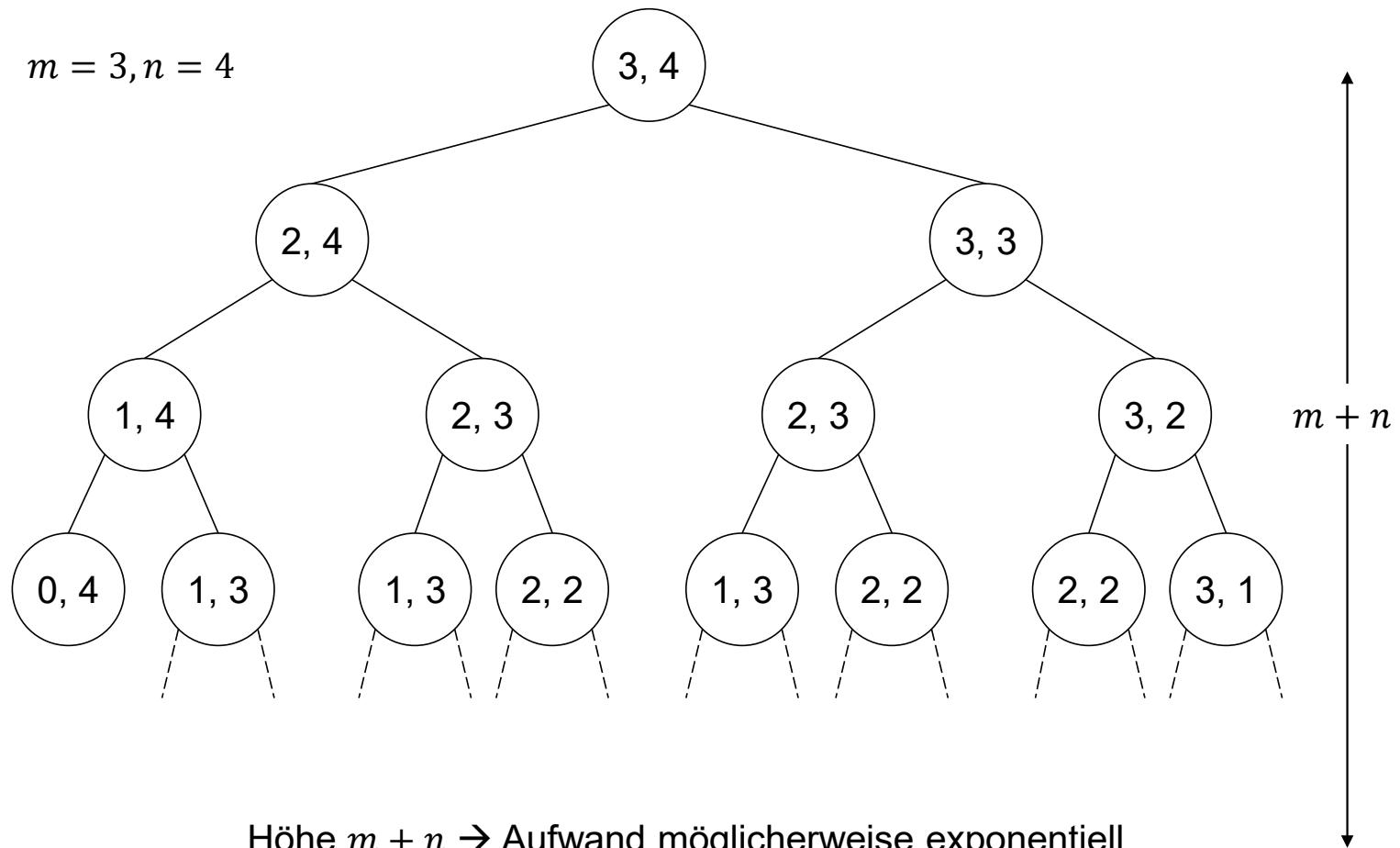
$\text{LCS}(x, y, i, j)$

```
1  if  $i == 0$  or  $j == 0$ 
2      return 0
3  elseif  $x[i] == y[j]$ 
4       $c[i, j] = \text{LCS}(x, y, i - 1, j - 1) + 1$ 
5  else  $c[i, j] = \max(\text{LCS}(x, y, i - 1, j),$ 
          $\text{LCS}(x, y, i, j - 1))$ 
```

- **Worst-case:** $x[i] \neq y[i]$
 - Evaluiere zwei Teilprobleme, je nur ein Parameter dekrementiert

Rekursionsbaum (Worst Case)

$m = 3, n = 4$



Höhe $m + n \rightarrow$ Aufwand möglicherweise exponentiell

Überlappende Unterprobleme

- Rekursive Lösung enthält eine „kleine“ Anzahl verschiedener Unterprobleme, die wiederholt gelöst werden („Überlappung“)
 - Anzahl verschiedener Unterprobleme für das LCS Problem mit Reihen der Länge m und n ist nur $m \cdot n$
 - X hat m verschiedene Präfixe, Y hat n : es gibt $m \cdot n$ Kombinationen von Präfixen
- Dynamisches Programmieren stützt sich auf optimale Teilstruktur und überlappende Unterprobleme

3. Memoisierung

- Speichere Lösung von Teilproblemen in einer Tabelle $c[i, j]$
- Initialisiere: $c[0, j] = c[i, 0] = 0$, sonst NIL
- Bei wiederholten Aufrufen des Teilproblems, lese Lösung aus Tabelle

MEM-LCS(x, y, i, j)

```
1  if  $c[i, j] == \text{NIL}$ 
2      if  $x[i] == y[j]$ 
3           $c[i, j] = \text{MEM-LCS}(x, y, i - 1, j - 1) + 1$ 
4      else  $c[i, j] = \max(\text{MEM-LCS}(x, y, i - 1, j),$ 
            $\text{MEM-LCS}(x, y, i, j - 1))$ 
```

- Aufwand $\Theta(mn)$ (Zeit und Speicherplatz)

Beispiel

	a	m	p	u	t	a	t	i	o	n
s	0	0	0	0	0	0	0	0	0	0
p	0	0	0	0	1	1	1	1	1	1
a	0	1	1	1	1	1	2	2	2	2
n	0	1	1	1	1	2	2	2	2	3
k	0	1	1	1	1	2	2	2	2	23
i	0	1	1	1	1	2	2	3	3	3
n	0	1	1	1	1	2	2	2	2	4
g	0	1	1	1	1	2	2	3	3	4
			P		a		:			n

4. Rekonstruktion der LCS

- Jeder Eintrag $c[i, j]$ hängt nur von drei anderen Einträgen ab:

$$c[i - 1, j - 1], c[i - 1, j], c[i, j - 1]$$

$$c[i, j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ c[i - 1, j - 1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j, \\ \max(c[i - 1, j], c[i, j - 1]) & \text{if } i, j > 0 \text{ and } x_i \neq y_j. \end{cases}$$

- Gegeben $c[i, j]$, können in konstanter Zeit bestimmen, welcher vorhergehende Eintrag zur Berechnung verwendet wurde

Bottom-up (statt Memoisierung)

LCS-LENGTH(X, Y)

```
1   $m = X.length$ 
2   $n = Y.length$ 
3  let  $b[1..m, 1..n]$  and  $c[0..m, 0..n]$  be new tables
4  for  $i = 1$  to  $m$ 
5     $c[i, 0] = 0$ 
6  for  $j = 0$  to  $n$ 
7     $c[0, j] = 0$ 
8  for  $i = 1$  to  $m$ 
9    for  $j = 1$  to  $n$ 
10      if  $x_i == y_j$ 
11         $c[i, j] = c[i - 1, j - 1] + 1$ 
12         $b[i, j] = “↖”$ 
13      elseif  $c[i - 1, j] \geq c[i, j - 1]$ 
14         $c[i, j] = c[i - 1, j]$ 
15         $b[i, j] = “↑”$ 
16      else  $c[i, j] = c[i, j - 1]$ 
17         $b[i, j] = “←”$ 
18  return  $c$  and  $b$ 
```

Übersicht

Dynamische Programmierung

- Einführung
- Schneiden von Eisenstangen
- Längste gemeinsame Teilsequenz
- Optimale binäre Suchbäume
- Eigenschaften der dynamischen Programmierung

Optimale binäre Suchbäume

- Gegeben Sequenz $K = \langle k_1, k_2, \dots, k_n \rangle$ von Schlüsseln, sortiert nach $k_1 < k_2 < \dots$
- Wahrscheinlichkeit p_i für Suche nach k_i
- **Problem:** konstruiere binären Suchbaum mit minimalem erwartetem Aufwand für Suche
- Anwendung: Wörterbuch für automatische Übersetzungen
 - Worthäufigkeiten bekannt

Erwartete Suchkosten

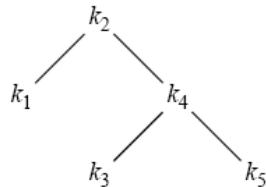
- Tatsächliche Kosten = Anzahl untersuchter Knoten
 - Für k_i , Kosten = $depth_T(k_i) + 1$
- Erwartete Kosten

$$\begin{aligned} E[\text{Suchkosten in } T] &= \sum_{i=1}^n (depth_T(k_i) + 1)p_i \\ &= \sum_{i=1}^n depth_T(k_i)p_i + \underbrace{\sum_{i=1}^n p_i}_{=1} \end{aligned}$$

Beispiel

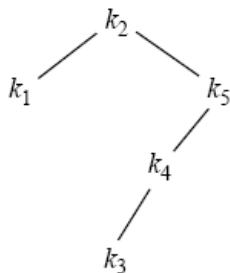
i	1	2	3	4	5
p_i	.25	.2	.05	.2	.3

Example:



i	$\text{depth}_T(k_i)$	$\text{depth}_T(k_i) \cdot p_i$
1	1	.25
2	0	0
3	2	.1
4	1	.2
5	2	.6
		1.15

Therefore, $E[\text{search cost}] = 2.15$.



i	$\text{depth}_T(k_i)$	$\text{depth}_T(k_i) \cdot p_i$
1	1	.25
2	0	0
3	3	.15
4	2	.4
5	1	.3
		1.10

Therefore, $E[\text{search cost}] = 2.10$, which turns out to be optimal.

Beobachtungen

- Optimaler BST (binary search tree)
 - hat nicht unbedingt kleinste Höhe
 - hat nicht unbedingt Schlüssel mit grösster Suchwahrscheinlichkeit an der Wurzel
- Naive Konstruktion
 - Teste alle Möglichkeiten
 - Aufwand exponentiell

1. Optimale Teilstruktur

- Gegeben: optimaler BST T mit Teilbaum T'
- **Behauptung:** Falls T ein optimaler BST ist, muss T' auch optimal sein
- Beweis durch Widerspruch
 - Falls T' nicht optimal wäre, könnten wir ihn durch einen optimalen Teilbaum T'' ersetzen
 - Dies würde die Gesamtkosten von T reduzieren, und somit wäre T nicht optimal

Teilprobleme

- „Problem“: gegeben Schlüssel k_i, \dots, k_j
 - Sei k_r Wurzel von k_i, \dots, k_j
- „Teilprobleme“
 - Linker Teilbaum $k_i, \dots, k_{r-1} < k_r$
 - Rechter Teilbaum $k_{r+1}, \dots, k_j > k_r$
- Mögliche Lösung
 - Für jedes Problem k_i, \dots, k_j , untersuche alle möglichen Wurzeln
 - Bestimme jeweils optimale BST für linke und rechte Teilbäume

2. Rekursive Lösung

- Sei $e[i, j] = \text{erwartete Suchkosten für } k_i, \dots, k_j$
 - Falls $j = i - 1$, dann $e[i, j] = 0$
- Sei k_r Wurzel für optimalen BST von k_i, \dots, k_j
$$e[i, j] = p_r + (e[i, r - 1] + w(i, r - 1)) + (e[r + 1, j] + w(r + 1, j))$$
 - wobei $w(i, j) = \sum_{l=i}^j p_l$
- Beachte $w(i, j) = w(i, r - 1) + p_r + w(r + 1, j)$
- Somit $e[i, j] = e[i, r - 1] + e[r + 1, j] + w(i, j)$
- Rekursionsgleichung

$$e[i, j] = \begin{cases} 0 & \text{falls } j = i - 1 \\ \min_{i \leq r \leq j} \{e[i, r - 1] + e[r + 1, j] + w(i, j)\} & \text{falls } i \leq j \end{cases}$$

3. Bottom-up Lösung

- Berechne Tabellen
 - $e[i, j]$ erwartete Kosten
 - $w[i, j]$ Wahrscheinlichkeit für Suche
 - $root[i, j]$ Wurzel
- Bottom-up Strategie, drei Schleifen
 - Über alle Größen l von Teilbäumen;
d.h. Teilbäume mit $l = j - i + 1$ Schlüsseln;
starte bei $l = 1$
 - Über alle Teilbäume $[i, j]$ der Grösse l
 - Über alle möglichen Wurzeln jedes Teilbaums
 - » Evaluiere Kosten, berechne Tabelleneinträge

Dynamische Programmierung

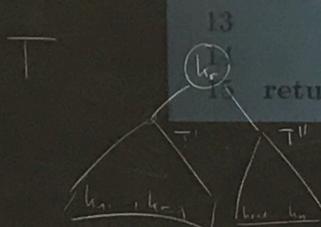
OPTIMAL-BST(p, n)

```
1 let  $e[1..n + 1, 0..n]$ ,  $w[1..n + 1, 0..n]$ , and  
     $root[1..n, 1..n]$  be new tables  
2 for  $i = 1$  to  $n + 1$           // leere Bereiche initialisieren  
3      $e[i, i - 1] = 0$   
4      $w[i, i - 1] = 0$   
5 for  $l = 1$  to  $n$           // Schleife: Teilbaumgrösse 1 ... n  
6     for  $i = 1$  to  $n - l + 1$   // Schleife: Teilbäume der Grösse  $l$   
7          $j = i + l - 1$   
8          $e[i, j] = \infty$   
9          $w[i, j] = w[i, j - 1] + p_j$   
10        for  $r = i$  to  $j$       // Schleife: Wurzel  $r$   
11             $t = e[i, r - 1] + e[r + 1, j] + w[i, j]$   
12            if  $t < e[i, j]$       // temporäres Optimum  
13                 $e[i, j] = t$   
14                 $root[i, j] = r$   
15 return  $e$  and  $root$ 
```

```

2 for i = 1 to n+1           // leere Bereiche initialisieren
3   e[i, i - 1] = 0
4   w[i, i - 1] = 0
5 for l = 1 to n             // Schleife: Teilbaumgrösse 1 ... n
6   for i = 1 to n - l + 1    // Schleife: Teilbäume der Grösse l
7     j = i + l - 1
8     e[i, j] = ∞
9     w[i, j] = w[i, j - 1] + p_j
10    for r = i to j          // Schleife: Wurzel r
11      t = e[i, r - 1] + e[r + 1, j] + w[i, j]
12      if t < e[i, j]         // temporäres Optimum

```



return e and $root$

$$e[i, j] \stackrel{0.75}{=} \min_{\text{root}[i, j]} \left(\sum_{l=1}^{j-i} \text{depth}_T(h_e) + 1 \right) p_e$$

$$= p_e + \sum_{l=1}^{j-i} \text{depth}_T(h_e) + 1 p_e$$

$$+ \sum_{l=i+1}^n \text{depth}_T(h_e) + 1 p_e$$

$$e[2, 5] = \min_{\text{root}[2, 5]} (e[3, 5] + \overbrace{w[2, 5]}^{0.75})$$

$$\rightarrow e[2, 5] = 4$$

$$e[2, 5] = \min_{\text{root}[2, 5]} (e[2, 5] + \overbrace{w[2, 5]}^{0.75})$$

$$e[2, 5] = \min_{\text{root}[2, 5]} (e[2, 4] + \overbrace{w[2, 5]}^{0.75})$$

$$e[2, 5] = \min_{\text{root}[2, 5]} (e[2, 3] + \overbrace{w[2, 5]}^{0.75})$$

$$e[2, 5] = \min_{\text{root}[2, 5]} (e[2, 2] + \overbrace{w[2, 5]}^{0.75})$$

$$e[2, 5] = \min_{\text{root}[2, 5]} (e[2, 1] + \overbrace{w[2, 5]}^{0.75})$$

$$e[2, 5] = \min_{\text{root}[2, 5]} (e[1, 1] + \overbrace{w[2, 5]}^{0.75})$$

$$e[2, 5] = \min_{\text{root}[2, 5]} (0 + \overbrace{w[2, 5]}^{0.75})$$

$$e[2, 5] = \min_{\text{root}[2, 5]} (0 + 4)$$

$$e[2, 5] = 4$$

$$\text{root}[2, 5] = 4$$

- 1 Beschreibe Struktur einer opt. Lösung
- Zeige opt. Teil-Struktur
- 2 Definiere Wert der Lösung rekursiv
- 3 Berechne Wert der opt. Lösung Bottom-Up
- 4 Konstruiere optimale Lösung

Es gilt für $i \leq l \leq r$

$$\text{depth}_T(h_e) = \text{depth}_{T'}(h_e) + 1$$

$$\boxed{\quad} = \sum_{l=1}^{r-i} (\text{depth}_{T'}(h_e) + 1 + 1) p_e + \overbrace{w(i, r-1)}^{0.75}$$

$$= \sum_{l=1}^{r-i} (\text{depth}_{T'}(h_e) + 1) p_e + \sum_{l=i+1}^{r-i} p_e$$

$$\text{root}[2, 5] = 4$$

$$e[2, 5] = \min_{\text{root}[2, 5]} (e[3, 5] + \overbrace{w[2, 5]}^{0.75})$$

$$\rightarrow e[2, 5] = \min_{\text{root}[2, 5]} (e[2, 5] + \overbrace{w[2, 5]}^{0.75})$$

$$e[2, 5] = \min_{\text{root}[2, 5]} (e[2, 4] + \overbrace{w[2, 5]}^{0.75})$$

$$e[2, 5] = \min_{\text{root}[2, 5]} (e[2, 3] + \overbrace{w[2, 5]}^{0.75})$$

$$e[2, 5] = \min_{\text{root}[2, 5]} (e[2, 2] + \overbrace{w[2, 5]}^{0.75})$$

$$e[2, 5] = \min_{\text{root}[2, 5]} (e[2, 1] + \overbrace{w[2, 5]}^{0.75})$$

$$e[2, 5] = \min_{\text{root}[2, 5]} (0 + \overbrace{w[2, 5]}^{0.75})$$

$$e[2, 5] = \min_{\text{root}[2, 5]} (0 + 4)$$

$$e[2, 5] = 4$$

Beispiel

	j					
w	0	1	2	3	4	5
1	0	.25	.45	.5	.7	1.0
2		0	.2	.25	.45	.75
3			0	.05	.25	.55
4				0	.2	.5
5					0	.3
6						0

	j					
e	0	1	2	3	4	5
1	0	.25	.65	.8	1.25	2.10
2		0	.2	.3	.75	1.35
3			0	.05	.3	.85
4				0	.2	.7
5					0	.3
6						0

	j				
$root$	1	2	3	4	5
1	1	1	1	2	2
2		2	2	2	4
3			3	4	5
4				4	5
5					5

4. Rekonstruktion der Lösung

- Mittels einfachem rekursivem Algorithmus aus der *root* Tabelle
- Übung

Übersicht

Dynamische Programmierung

- Einführung
- Schneiden von Eisenstangen
- Längste gemeinsame Teilsequenz
- Optimale binäre Suchbäume
- Eigenschaften der dynamischen Programmierung

Dynamische Programmierung

- Optimale Teilstruktur
- Überlappende Teilprobleme
- Memoisierung
- Rekonstruktion der optimalen Lösung

Optimale Teilstruktur

- „Optimale Lösung eines Problems beinhaltet optimale Lösungen von Teilproblemen“
- Rezept um zu zeigen, dass Problem Eigenschaft der optimalen Teilstruktur besitzt
 1. Zeige, dass Problem gelöst werden kann, indem Teilprobleme ausgewählt werden können (Annahme, dass Teilprobleme bekannt sind, welche Teil der optimalen Lösung sind)
 2. Zeige, dass Lösung der Teilprobleme optimal sein muss, um optimale Lösung des Problems zu erhalten (oft durch Widerspruch mit «cut and paste» Argumentation)

Optimale Teilstruktur

- Wieviele Teilprobleme werden in einer optimalen Lösung benutzt?
- Wieviele Kandidaten für Teilprobleme gibt es?
- Eisenstangen
 - 1 Teilproblem (Zerschneiden des Reststücks)
 - n Kandidaten (möglicher Schnitt bei $1, \dots, n$)
- LCS
 - 1 Teilproblem (LCS für Reihen, wo mindestens eine um eins kürzer ist)
 - 1 oder 2 Kandidaten (welche Reihe wird um eins gekürzt)
- Optimale Suchbäume
 - 2 Teilprobleme (linker und rechter Unterbaum einer Wurzel)
 - Anz. Kandidaten = Anz. Mögl. Positionen der Wurzel im Unterbaum der Wurzel

Optimale Teilstruktur

- Dynamisches Programmieren verwendet optimale Teilstruktur **bottom up**
 - **Zuerst**: finde Lösungen zu Teilproblemen
 - **Dann**: wähle welche Teilprobleme zur optimalen Lösung des Problems gehören
- Greedy Algorithmen arbeiten **top down**
 - Nächstes Mal
- Nicht alle Optimierungsprobleme haben Eigenschaft der optimalen Teilstruktur!

Überlappende Teilprobleme

- Treten auf, wenn rekursiver Algorithmus dasselbe Problem mehrmals löst
- Unterschied zu effizienten Teile-und-beherrsche Algorithmen
 - Generieren ein **neues** Teilproblem in jedem Aufruf (Merge Sort, Quicksort)
 - Jedes Teilproblem wird nur einmal gelöst

Memoisierung

- Variante der Reihenfolge, in welcher die Tabelle mit Lösung von Teilproblemen gefüllt wird
 - Verwende rekursiven Algorithmus
 - Speichere Lösung von Teilproblemen in Tabelle
 - Verfolge Rekursion nur dann weiter, wenn Lösung des Teilproblems in Tabelle noch nicht gespeichert
- Bevorzugter Ansatz im dynamischen Programmieren: bottom-up Berechnung der Teilprobleme ohne Rekursion, Memoisierung nicht nötig
- LCS Problem
 - Hier in Slides: mit Memoisierung
 - Buch: bottom up Algorithmus ohne Memoisierung

Nächstes Mal

- Greedy Algorithmen