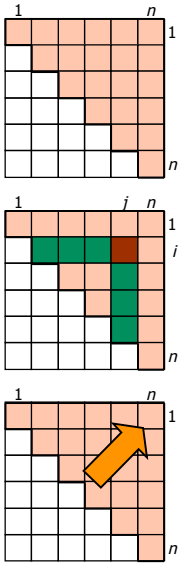
Formulierung mittels Dynamischer Programmierung

- Beobachtung: die Anzahl der Teilprobleme $A_{i...j}$ mit $1 \le i \le j \le n$ ist nur $\frac{n(n+1)}{2} \in \Theta(n^2)$
- Folgerung: der naïve rekursive Algo berechnet viele Teilprobleme mehrfach!
- Idee: Bottom-up-Berechnung der optimalen Lösung, speichere Teillösungen in Tabelle (→ daher "dynamische Programmierung")
- Welche Tabelleneinträge werden für m[i,j] benötigt?

$$m[i,j] = \min_{i \le k < j} \{m[i,k] + m[k+1,j] + p_{i-1} p_k p_j\}$$

- Hier: Bottom-up = von der Diagonale nach "rechts oben"



Berechnungsbeispiel

A_1: 30 × 35
A_2: 35 × 15
A_3: 15 × 5
A_4: 5 × 10
A_5: 10 × 20
A_6: 20 × 25

p = (30, 35, 15, 5, 10, 20, 25)

m	j=1	2	3	4	5	6
i=1	0	15750	7875	9375		
2		0	2625	4375		
3			0	750	2000	
4				0	1000	3500
5					0	5000
6						0

$$m[2,5] = \min_{2 \le k < 5} \{m[2,k] + m[k+1,5] + p_1 p_k p_5\}$$

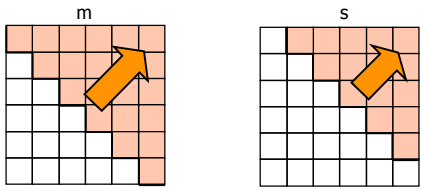
$$= \min \begin{Bmatrix} m[2,2] + m[3,5] + p_1 p_2 p_5, \\ m[2,3] + m[4,5] + p_1 p_3 p_5, \\ m[2,4] + m[5,5] + p_1 p_4 p_5 \end{Bmatrix}$$

$$= \min \begin{Bmatrix} 0 + 2500 + 35 \cdot 15 \cdot 20 = 13000 \\ 2625 + 1000 + 35 \cdot 5 \cdot 20 = 7125 \\ 4375 + 0 + 35 \cdot 10 \cdot 20 = 11375 \end{Bmatrix}$$

$$= 7125$$

Gewinnung der optimalen Reihenfolge

- Speichere die Position für die beste Trennung, d.h., denjenigen Wert *k*, der zum minimalen Wert von *m*[*i,j*] führt
- Speichere dazu in einem zweiten Array *s*[*i,j*] dieses optimale *k:*
  - *s*[*i,j*] wird nur für Folgen mit mindestens 2 Matrizen und $j > i$ benötigt
  - *s*[*i,j*] gibt an, welche Multiplikation zuletzt ausgeführt werden soll
  - Für *s*[*i,j*] = *k* und die Teilfolge $A_{i...j}$ ist es optimal, zuerst $A_{i...k}$, danach $A_{k+1...j}$ und zum Schluss die beiden Teilergebnisse zu multiplizieren



$$A_{i...j} = A_i \cdots A_j = (A_i \cdots A_{s[i,j]})(A_{s[i,j]+1} \cdots A_j)$$

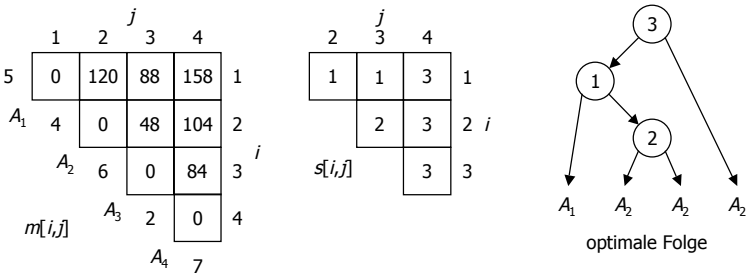
Implementierung mittels dynamischer Programmierung

```
n = len(p)
for i in range( 1,n+1 ):  # assume m has dim (n+1)·(n+1)
    m[i,i] = 0
for l in range( 2,n+1 ):  # consider chains of length l
    for i in range( 1,n-l ):
        j = i+l-1         # len l → j-i = l-1
        m[i,j] = ∝
        for k in range( i,j ):
            q = m[i,k] + m[k+1,j] + p[i-1]*p[k]*p[j]
            if q < m[i,j]:
                m[i,j] = q
                s[i,j] = k
```

- Komplexität: es gibt 3 geschachtelte Schleifen, die jeweils höchstens *n*-mal durchlaufen werden, die Laufzeit beträgt also $O(n^3)$

Beispiel

- Gegeben: Folge von Dimensionen (5, 4, 6, 2, 7)
- Multiplikation von A_1 (5×4), A_2 (4×6), A_3 (6×2) und A_4 (2×7)
- Optimale Folge ist $((A_1(A_2A_3))A_4)$



Technik der dynamischen Programmierung

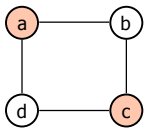
- Rekursiver Ansatz: Lösen eines Problems durch Lösen mehrerer kleinerer Teilprobleme, aus denen sich die Lösung für das Ausgangsproblem zusammensetzt
  - Phänomen: Mehrfachberechnungen von Lösungen
  - Bottom-up-Berechnung: fügt Lösungen kleinerer Unterprobleme zusammen, um größere Unterprobleme zu lösen und liefert so eine Lösung für das gesamte Problem
  - Methode: iterative Erstellung einer Tabelle
- Optimalitätsprinzip: eine optimale Lösung für das Ausgangsproblem setzt sich aus optimalen Lösungen für kleinere Probleme zusammen

Wichtige Begriffe

- Optimale Unterstruktur (Prinzip der Optimalität):
  - Ein Problem besitzt die (Eigenschaft der) optimalen Substruktur, bzw. gehorcht dem Prinzip der Optimalität :⇔:

1. Die Lösung eines Problems setzt sich aus den Lösungen von Teilproblemen zusammen
   - Bsp. MCMP: gesuchte Klammerung von $A_1 \ldots A_n$ setzt sich zusammen aus der Klammerung einer (bestimmten) Teilkette $A_1 \ldots A_k$ und einer Teilkette $A_{k+1} \ldots A_n$
2. Wenn die Lösung optimal ist, dann müssen auch die Teillösungen optimal sein!
   - Bsp. MCMP: wir haben folgende Behauptung bewiesen:
     Falls Klammerung zu $A_1 \ldots A_k$ nicht optimal ⇒
     Klammerung zu $A_1 \ldots A_n$ (die gemäß Ann. Teillsg zu $A_1 \ldots A_k$ enthält) kann nicht optimal sein

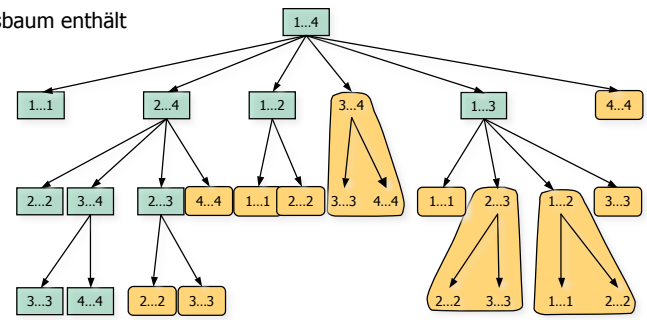
- Achtung: Zweite Bedingung (Teillösungen müssen optimal sein) ist manchmal nicht erfüllt:
  - Bsp.: längster Pfad durch einen Graphen
  - Aufgabe im Bsp.: bestimme längsten Pfad von a nach c



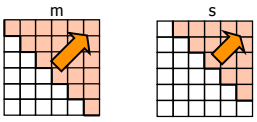
  - Im Bsp rechts: Lösung besteht aus Teilpfaden a→b und b→c
  - Aber diese sind nicht optimale(!) Lösungen der entspr. Teilprobleme
    - Optimale (d.h., längste) Lösung für a→b = a→d→c→b

- Unabhängigkeit der Teillösungen:
  - Die Teilprobleme heißen (im Sinne der Dyn. Progr.) unabhängig :⇔: die Optimierung des einen Teilproblems beeinflußt nicht die Optimierung des anderen (z.B. bei der Wahl der Unterteilung)
    - Bsp. MCMP: die Wahl der Klammerung für $A_1 \ldots A_k$ ist völlig unabhängig von der Klammerung für $A_{k+1} \ldots A_n$
    - Gegenbsp. "längster Pfad": die optimale Lsg für a→b (nämlich a→d→c→b) nimmt der optimalen Lsg für b→c Elemente weg

- Überlappende Teilprobleme:
  - Problem wird zerlegt in Unterprobleme, diese wieder in Unter-Unterprobleme, usw.
  - Ab irgendeinem Grad müssen dieselben Unter-Unterprobleme mehrfach vorkommen, sonst ergibt das DP wahrscheinlich keine effiziente Lösung
    - Bsp. MCMP: Rekursionsbaum enthält viele überlappende Teilprobleme



- Rekonstruktion der optimalen Lösung:
  - Optimale Lösung für Gesamtproblem beinhaltet 3 Schritte:
    1. Entscheidung treffen zur Zerlegung des Problems in Teile
    2. Optimalen Wert für Teilprobleme berechnen
    3. Optimalen Wert für Gesamtproblem "zusammensetzen"
  - Dynamische Programmierung berechnet zunächst oft nur den "Weg" zur optimalen Lösung, aber
    - im zweiten Schritt wird dann die optimale Lösung mittels diese Weges berechnet
    - dazu Entscheidungen einfach in Phase 1 speichern und in Phase 2 dann "abspielen"
    - Beispiel: MCMP
      Speichere Index k, der zum optimalen Wert führt in zweitem Array s
      $A_{i\ldots j} = (A_i \cdot \cdots \cdot A_{s[i,j]})(A_{s[i,j]+1} \cdot \cdots \cdot A_j)$



Schritte bei der dynamischen Programmierung

1. Charakterisiere die (rekursive) Struktur der optimalen Lösung
2. Definiere den Wert einer optimalen Lösung rekursiv
3. Transformiere die rekursive Methode in eine iterative bottom-up Methode, bei der alle Zwischenergebnisse in einer Tabelle gespeichert werden
4. Erstelle eine optimale Lösung aus dem in (3) berechneten optimalen Wert, zusammen mit der ebenfalls in (3) gespeicherten Zusatzinformationen

Optimale Suchbäume

- Beispiel: Wörterbuch Englisch → Französisch
- Mit AVL-Bäumen oder perfekt balancierten Bäumen bekommt man $O(n \log n)$ worst-case Lookup-Zeit
- Bsp.: Übersetzung eines englischen Textes
- Folge: manche Wörter werden wesentlich häufiger als andere nachgesehen
- Ziel: Gesamtzeit zum Übersetzen eines Textes möglichst klein, d.h., Gesamtzeit für Lookup (oft mehrfach) aller Wörter des Textes soll klein sein
- M.a.W.: durchschnittliche Lookup-Zeit pro Wort soll klein sein
- Folge: häufige Wörter müssen "eher weiter oben" an der Wurzel stehen

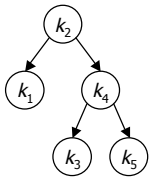
Problemstellung

- Gegeben sei Folge $K = k_1 < k_1 < ... < k_n$ von n sortieren Keys, mit einer Suchwahrscheinlichkeit p_i für jeden Key k_i
- Es soll ein binärer Suchbaum (BST) mit minimal zu erwartenden Suchkosten erstellt werden
- Kosten eines Lookup = Anzahl der besuchten Knoten im Baum
- Für jeden Key k_i sind die Kosten = $d_T(k_i)+1$, mit $d_T(k_i)$ = Tiefe von k_i im BST T

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

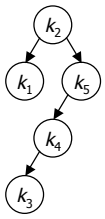
Beispiel

- Gegeben seien 5 Keys mit den Suchwahrscheinlichkeiten:
  $p_1 = 0.25,\ p_2 = 0.2,\ p_3 = 0.05,\ p_4 = 0.2,\ p_5 = 0.3$



i	$d_T(k_i)$	$d_T(k_i)\cdot p_i$
1	1	0.25
2	0	0
3	2	0.1
4	1	0.2
5	2	0.6
		1.15

→ E[Suchkosten] = 2.15



i	$d_T(k_i)$	$d_T(k_i)\cdot p_i$
1	1	0.25
2	0	0
3	3	0.15
4	2	0.4
5	1	0.3
		1.1

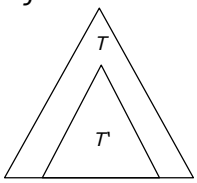
→ E[Suchkosten] = 2.1

Dies ist der optimale Baum für diese Key-Menge

- Beobachtungen:
  - optimaler BST muß nicht die kleinste Höhe haben
  - optimaler BST muß nicht die größte Wahrscheinlichkeit an der Wurzel haben
- Erstellen durch erschöpfendes Testen? (*exhaustive enumeration*)
  - erstelle alle möglichen BST mit n Knoten
  - für jeden BST: verteile die Schlüssel und berechne die zu erwartenden Suchkosten
  - es gibt aber $C_n \in \Omega\left(\frac{4^n}{n^{\frac{3}{2}}}\right)$ verschiedene BST mit n Knoten

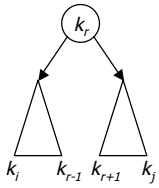
Optimale Unterstruktur

1. Jeder Unterbaum eines BST beinhaltet Schlüssel in einem zusammenhängenen Bereich $k_i, \ldots, k_j$ für $1 \le i \le j \le n$
2. Wenn T ein optimaler BST ist und den Unterbaum T' enthält, dann ist auch T' ein optimaler BST für die Keys $k_i, \ldots, k_j$



- Beweis: "Cut and paste", d.h., Beweis durch Widerspruch:
  - Annahme: T' nicht optimal für $k_i, \ldots, k_j$
  - Es ex. T'' der besser als T' ist
  - Sei $\hat{T}$ der BST, der aus T entsteht, indem T' durch T'' ersetzt wird
  - $\hat{T}$ ist korrekter BST und hat außerdem kleinere mittlere Suchkosten als T $\Rightarrow$ W!

1. Einer der Schlüssel $k_i, \ldots, k_j$, z.B. k_r mit $i \le r \le j$, muß die Wurzel eines optimalen Unterbaumes für diese Schlüssel sein
   - Der linke Unterbaum von k_r enthält $k_i, \ldots, k_{r-1}$
   - Der rechte Unterbaum von k_r enthält $k_{r+1}, \ldots, k_j$



2. Zum Finden eines optimalen BST:
   - betrachte alle Knoten k_r ($i \le r \le j$), die als Wurzel in Frage kommen
   - bestimme alle optimalen BSTs für $k_i, \ldots, k_{r-1}$ und $k_{r+1}, \ldots, k_j$

Rekursive Lösung

- Finde optimalen BST für $k_i, \ldots, k_j$ mit $1 \le i \le j \le n$, für $j < i$ ist der Baum leer
- Definiere $e[i,j]$:= erwartete Suchkosten des optimalen BST für $k_i, \ldots, k_j$
- Wenn $i = j$, dann $e[i,i] = p_i$
- Zur technischen Vereinfachung setze $e[i,j] = 0$ für $j < i$
- Falls $j > i$:
  - wähle Wurzel k_r für $i \le r \le j$
  - erstelle rekursiv einen optimalen BST
    - für $k_i, \ldots, k_{r-1}$ als den linken Unterbaum und
    - für $k_{r+1}, \ldots, k_j$ als den rechten Unterbaum

- Nützliche Verallgemeinerung:
  - Es wird nicht mehr verlangt, daß $p_i, \ldots, p_j$ eine Wahrscheinlichkeitsverteilung bilden; es darf gelten

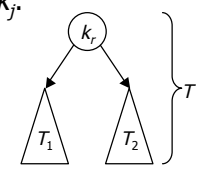
$$\sum_{l=i}^{j} p_l \neq 1$$

  - Gesucht ist aber weiterhin ein BST für $k_i, \ldots, k_j$, so daß die Summe

$$\sum_{l=i}^{j} (d_T(k_l) + 1) p_l$$

    minimiert wird
- Diese Summe wird dennoch erwarteter Suchaufwand genannt

- Wenn k_r die Wurzel eines optimalen BST für $k_i, \ldots, k_j$:



$$
\begin{aligned}
e[i,j] &= \sum_{l=i}^{j} (d_T(k_l)+1)p_l \\
&= \sum_{l=i}^{j} d_T(k_l)p_l + \sum_{l=i}^{j} p_l \\
&= \underbrace{\sum_{l=i}^{r-1} (d_{T_1}(k_l)+1)p_l}_{e[i,r-1]} + \underbrace{\sum_{l=r+1}^{j} (d_{T_2}(k_l)+1)p_l}_{e[r+1,j]} + \sum_{l=i}^{j} p_l \\
&= e[i,r-1] + e[i,r+1] + w(i,j)
\end{aligned}
$$

- Aber k_r ist nicht bekannt, daher gilt

$$e[i,j] = \min_{i \le r \le j} \{e[i,r-1] + e[i,r+1] + w(i,j)\}$$

Bottom-up-Berechnung einer optimalen Lösung

- Speichere für jedes Unterproblem (i,j):
  - zu erwartende Suchkosten in einer Tabelle e[*i,j*] mit $1 \le i \le j \le n$
  - *r*[*i,j*] = Wurzel des Unterbaums mit den Schlüsseln $k_i, \ldots, k_j$, $i \le r[i,j] \le j$
  - *w*[*i,j*] $\in [0,1]$ = Summe der Wahrscheinlichkeiten
    - *w*[*i,i*] = p_i für $1 \le i \le n$
    - *w*[*i,j*] = *w*[*i,j-1*] + p_j für $1 \le i \le j \le n$

```
def optimal_bst( p,q,n ):
    for i in range( 1,n ):
        e[i,i] = w[i,i] = p[i]
    for l in range( 2,n ):# calc all opt trees w/ l keys
        for i in range( 1, n-l ):           # n - ell
            j = i+l-1
            e[i,j] = ∝                      # z.B. 2^31-1
            w[i,j] = w[i,j-1] + p[j]
            for r in range( i, j+1 ):
                t = e[i,r-1] + e[r+1,j] + w[i,j]
                if t < e[i,j]:
                    e[i,j] = t
                    r[i,j] = r
    return e,r
```

- Laufzeit: $O(n^3)$

- Satz: Ein optimaler Suchbaum für n Keys mit gegebenen Zugriffshäufigkeiten kann in Zeit $O(n^3)$ konstruiert werden.