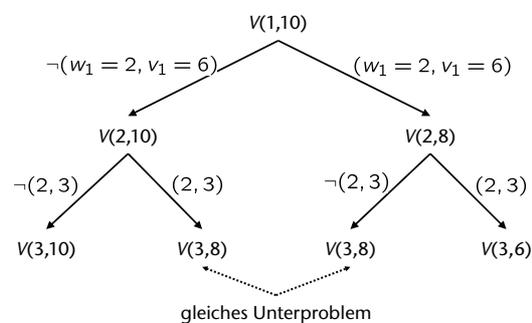


- Sei $V(i,k)$ der maximal mögliche Wert für die Gegenstände $i, i+1, \dots, n$ bei gegebener max. Kapazität k
- $V(i,k)$ kann dann für $i \leq n$ geschrieben werden als

$$V(i, k) = \begin{cases} 0 & i = n \wedge w_n > k \\ v_n & i = n \wedge w_n \leq k \\ V(i+1, k) & i < n \wedge w_i > k \\ \max \left\{ V(i+1, k), v_i + V(i+1, k - w_i) \right\} & i < n \wedge w_i \leq k \end{cases}$$

- Algorithmus, basierend auf diesen 4 Fällen, hat Laufzeit von $O(2^n)$
- Ist ineffizient, denn $V(i,k)$ wird für die gleichen i und k mehrmals berechnet
- Beispiel: $n = 5, c = 10, w = (2, 2, 6, 5, 4), v = (6, 3, 5, 4, 6)$



Lösung mittels Dynamischer Programmierung

- Ineffizienz kann vermieden werden, indem alle $V(i,k)$, einmal berechnet, in einer Tabelle gespeichert werden
- Die Tabelle wird in der Reihenfolge $i = n, n-1, \dots, 2, 1$ für $1 \leq k \leq c$ gefüllt

k	1	2	...	$j-1$	j	$j+1$...	c
$V(n, k)$	0	0	...	0	v_n	v_n	...	v_n

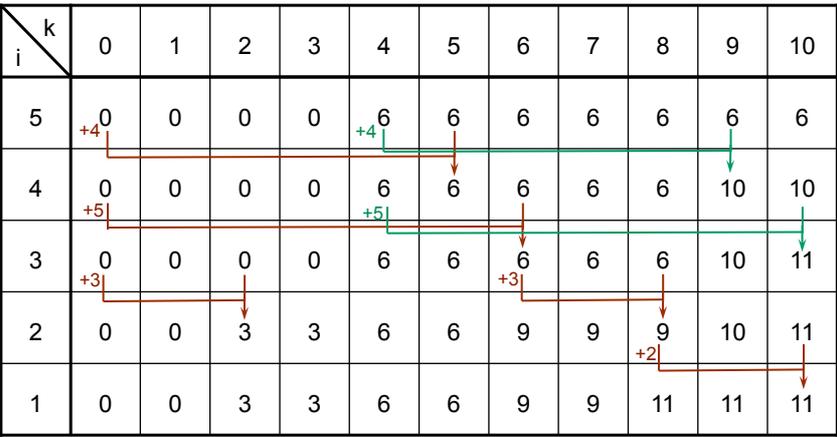


 j ist das erste k mit $w_n \leq k$

G. Zachmann Informatik 2 — SS 10 Dynamische Programmierung 45

Beispiel

$n = 5, c = 10, w = (2, 2, 6, 5, 4), v = (2, 3, 5, 4, 6)$

<table border="1" style="display: inline-table; text-align: left;"> <tr><td style="border: none;">\</td><td style="border: none;">k</td><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>10</td></tr> <tr><td style="border: none;">i</td><td style="border: none;">5</td><td>0</td><td>0</td><td>0</td><td>0</td><td>6</td><td>6</td><td>6</td><td>6</td><td>6</td><td>6</td><td>6</td></tr> <tr><td style="border: none;">4</td><td style="border: none;">0</td><td>0</td><td>0</td><td>0</td><td>6</td><td>6</td><td>6</td><td>6</td><td>6</td><td>10</td><td>10</td><td></td></tr> <tr><td style="border: none;">3</td><td style="border: none;">0</td><td>0</td><td>0</td><td>0</td><td>6</td><td>6</td><td>6</td><td>6</td><td>6</td><td>10</td><td>11</td><td></td></tr> <tr><td style="border: none;">2</td><td style="border: none;">0</td><td>0</td><td>3</td><td>3</td><td>6</td><td>6</td><td>9</td><td>9</td><td>9</td><td>10</td><td>11</td><td></td></tr> <tr><td style="border: none;">1</td><td style="border: none;">0</td><td>0</td><td>3</td><td>3</td><td>6</td><td>6</td><td>9</td><td>9</td><td>11</td><td>11</td><td>11</td><td></td></tr> </table>	\	k	0	1	2	3	4	5	6	7	8	9	10	i	5	0	0	0	0	6	6	6	6	6	6	6	4	0	0	0	0	6	6	6	6	6	10	10		3	0	0	0	0	6	6	6	6	6	10	11		2	0	0	3	3	6	6	9	9	9	10	11		1	0	0	3	3	6	6	9	9	11	11	11		
\	k	0	1	2	3	4	5	6	7	8	9	10																																																																			
i	5	0	0	0	0	6	6	6	6	6	6	6																																																																			
4	0	0	0	0	6	6	6	6	6	10	10																																																																				
3	0	0	0	0	6	6	6	6	6	10	11																																																																				
2	0	0	3	3	6	6	9	9	9	10	11																																																																				
1	0	0	3	3	6	6	9	9	11	11	11																																																																				

G. Zachmann Informatik 2 — SS 10 Dynamische Programmierung 46

(Re-)konstruktion der Lösung

$n = 5$, $c = 10$, $w = (2, 2, 6, 5, 4)$, $v = (2, 3, 5, 4, 6)$
 $x = [0, 0, 1, 0, 1]$ oder $x = [1, 1, 0, 0, 1]$

$i \backslash k$	0	1	2	3	4	5	6	7	8	9	10
5	0	0	0	0	6	6	6	6	6	6	6
4	0	0	0	0	6	6	6	6	6	10	10
3	0	0	0	0	6	6	6	6	6	10	11
2	0	0	3	3	6	6	9	9	9	10	11
1	0	0	3	3	6	6	9	9	11	11	11

G. Zachmann Informatik 2 — SS 10 Dynamische Programmierung 47

Bemerkungen

- Aufwand: $O(n \cdot c)$, c = Kapazität des Rucksacks
- Solche Komplexitäten nennt man auch **pseudo-polynomial time**
 - Dabei erscheint nicht (nur) die Länge des Inputs, sondern (auch) dessen *Wert* in der Komplexitätsfunktion!
- Achtung: dieser Algorithmus klappt nur, wenn c und die w_i Integers sind!
- Falls c oder die w_i keine Integers sind, dann ist das Problem "NP-vollständig", und es gibt (wahrscheinlich) keinen polynomiellen Algorithmus

G. Zachmann Informatik 2 — SS 10 Dynamische Programmierung 48

Die längste gemeinsame Teilfolge

- Seien $X = (x_1, \dots, x_m)$ und $Y = (y_1, \dots, y_n)$ zwei Folgen, 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, \dots, i_n gibt, mit $x_{i_j} = y_j$ für $j = 1, \dots, n$
- Beispiel: $Y = BCAC$ ist Teilfolge von $X = ABACABC$;
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 Y ist
- Beispiel: $Z = BCAC$ ist gemeinsame Teilfolge von $X = ABACABC$ und $Y = BACCABBC$

G. Zachmann Informatik 2 — SS 10 Dynamische Programmierung 49

- 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 länger ist
- Beispiel: $Z = BCAC$ ist nicht längste gemeinsame Teilfolge von $X = ABACABC$ und $Y = BACCABBC$, denn $BACAC$ ist eine längere gemeinsame Teilfolge von X und Y
- Beim Problem **Längste-Gemeinsame-Teilfolge** (*longest-common-subsequence problem*, LCSP) sind als Eingabe zwei Folgen $X = (x_1, \dots, x_m)$ und $Y = (y_1, \dots, y_n)$ gegeben, gesucht ist eine längste gemeinsame Teilfolge X und Y
- Anwendung: "Distanz" zwischen Strings messen (**Edit-Distanz**)

G. Zachmann Informatik 2 — SS 10 Dynamische Programmierung 50

Ein naiver Algorithmus

- Für jede mögliche Unterfolge von X :
 - prüfe ob es eine Unterfolge von Y ist
- Laufzeit: $\Theta(n 2^m)$
 - Es gibt 2^m mögliche Unterfolgen von X
 - Für jede Unterfolge wird Zeit $\Theta(n)$ benötigt, um zu überprüfen, ob diese Unterfolge von Y ist:
 - Scanne Y , "verbrauche" jeweils den nächsten Buchstaben von X , falls er passt
 - X ist Unterfolge von Y , wenn am Ende von Y kein Zeichen von X mehr übrig ist

G. Zachmann Informatik 2 — SS 10 Dynamische Programmierung 51

Die Struktur des LCSP

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

G. Zachmann Informatik 2 — SS 10 Dynamische Programmierung 52




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

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}

G. Zachmann Informatik 2 — SS 10 Dynamische Programmierung 53




Beweis

- Fall 1 ($x_m = y_n$):
 - Jede gemeinsame Teilfolge Z' , die **nicht** mit $z'_l = x_m = y_n$ endet, kann verlängert werden, indem $x_m = y_n$ angefügt wird \Rightarrow
 - die LCS Z muß mit $x_m = y_n$ enden;
 - Z_{k-1} ist längste gemeinsame Teilfolge von X_{m-1} und Y_{n-1} , denn
 - es gibt keine längere gemeinsame Teilfolge von X_{m-1} und Y_{n-1} , oder Z wäre keine längste gemeinsame Teilfolge.
- Fall 2 ($x_m \neq y_n$ und $z_k \neq x_m$):
 - Da Z nicht mit x_m endet \Rightarrow
 - Z ist gemeinsame Teilfolge von X_{m-1} und Y ;
 - daher gibt es keine längere gemeinsame Teilfolge von X_{m-1} und Y , oder Z wäre keine **längste** gemeinsame Teilfolge

G. Zachmann Informatik 2 — SS 10 Dynamische Programmierung 54

Eine Rekursion für die Länge der LCS

Lemma:

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

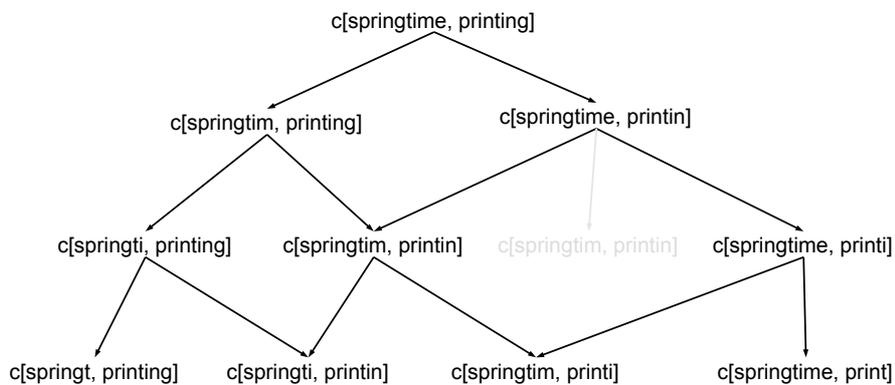
$$c[i,j] = \begin{cases} 0 & \text{falls } i = 0 \vee j = 0 \\ c[i-1, j-1] + 1 & \text{falls } i, j > 0 \wedge x_i = y_j \\ \max\{c[i-1, j], c[i, j-1]\} & \text{falls } i, j > 0 \wedge x_i \neq y_j \end{cases}$$

Beobachtung:

- Die rekursive Berechnung der $c[m,n]$ würde immer wieder zur Berechnung derselben Werte führen!
- Also dynamische Programmierung: berechne die Werte $c[i,j]$ iterativ bottom-up, z.B. zeilenweise
- Zusätzlich: speichere in $b[i,j]$ Informationen zur späteren (Re-)Konstruktion einer längsten gemeinsamen Teilfolge

Beispiel

$$c[\text{präfix}_1\alpha, \text{präfix}_2\beta] = \begin{cases} 0 & \text{falls } \alpha \text{ leer oder } \beta \text{ leer} \\ c[\text{präfix}_1, \text{präfix}_2] + 1 & \text{falls } \text{end}(\alpha) = \text{end}(\beta) \\ \max\{c[\text{präfix}_1\alpha, \text{präfix}_2], c[\text{präfix}_1, \text{präfix}_2\beta]\} & \text{falls } \text{end}(\alpha) \neq \text{end}(\beta) \end{cases}$$



Berechnung der Werte $c[i,j]$

```
def lcs_length( x, y ):
    for i in range( 0, len(x) ):
        c[i,0] = 0
    for j in range( 0, len(y) ):
        c[0,j] = 0
    for i in range( 1, len(x) ):
        for j in range( 1, len(y) ):
            if x[i] == y[j]:
                c[i,j] = c[i-1,j-1] + 1
                b[i,j] = "NW"
            else:
                if c[i-1,j] >= c[i,j-1]:
                    c[i,j] = c[i-1,j]
                    b[i,j] = "N"
                else:
                    c[i,j] = c[i,j-1]
                    b[i,j] = "W"
    return b, c
```

Beispiele für die Tabellen $c[i,j]$ und $b[i,j]$

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

b c

Laufzeit

- **Lemma:**
Der Algorithmus `lcs_length` hat die Laufzeit $O(nm)$, wenn die Folgen X und Y die Längen n bzw. m haben.

G. Zachmann Informatik 2 — SS 10 Dynamische Programmierung 59

Verwandte Probleme

- Es gibt viele Probleme, die sehr ähnlich zum LCSP sind
- Z.B. die **Edit-Distanz** (ein anderes Maß für den Abstand/Distanz 2-er Strings):
 - Gegeben 2 Strings A, B
 - Aufgabe: welches ist die minimale Folge von elementaren Editieroperationen, um A in B zu überführen?
 - Zugelassene Operationen: Zeichen löschen, einfügen, ersetzen
- Beispiele:

springtime

printing

ncaa tournament

north carolina

basketball

krzyzewski

G. Zachmann Informatik 2 — SS 10 Dynamische Programmierung 60




- Approximative Stringsuche:
 - Gegeben Text T und String S
 - Finde dasjenige Teilstück $T[i:j]$, das am ähnlichsten zu S ist (von allen anderen Teilstücken $T[i':j']$)
- Anwendungen: DNA-Sequence-Alignment, Google, Spell-Checker

G. Zachmann Informatik 2 — SS 10 Dynamische Programmierung 61




Memoisierung (Top-down-Ansatz)

- "Memo" = Gedächtnis
- Üblicherweise ist Formulierung der optimalen Lösung rekursiv, aber Algorithmus geht bottom-up vor
- *Memoization* [sic] = Technik in der dynamischen Programmierung, falls ein Bottom-up-Algorithmus nicht klar ist
- *Notizblock-Methode* zur Beschleunigung einer rekursiven Problemlösung:
 - Algo bleibt rekursiv
 - Ein Teilproblem wird nur beim **ersten Auftreten** gelöst
 - Die Lösung wird in einer Tabelle (z.B. Hash Table) gespeichert und bei jedem späteren Auftreten desselben Teilproblems (d.h., rekursiver Aufruf mit denselben Parametern) wird die Lösung (ohne erneute Rechnung!) in der Tabelle nachgeschlagen

G. Zachmann Informatik 2 — SS 10 Dynamische Programmierung 62

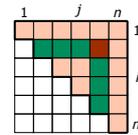
Beispiel: MCMP mittels Memoisierung

```
def mcm_mem_rek( p, i, j ):
    if i == j:
        return 0
    if m[i,j] < ∞ :
        return m[i,j] # check first, # if already computed
    for k in range( i,j ):
        q = p[i-1]*p[k]*p[j] + mcm_rek(p,i,k) + \
            mcm_rek(p, k+1, j)
        if q < m[i,j]:
            m[i,j] = q
    return m[i,j]

def mcm_mem( p ):
    for i in range( 1, len(p)+1 ):
        for j in range( 1, len(p)+1 ):
            m[i,j] = ∞ # z.B. maxint
    return mcm_mem_rek( p, 1, len(p)-1 )
```

Aufwand

- Behauptung: Zur Berechnung aller Einträge $m[i,j]$ mit Hilfe von `mcm_mem_rek` genügen insgesamt $O(n^3)$ Schritte
- Beweis:
 - $O(n^2)$ Einträge
 - jedes Element $m[i,j]$ wird einmal eingetragen
 - jeder Eintrag $m[i,j]$ wird zur Berechnung von weniger als $2n$ weiteren Einträgen $m[i',j']$ herangezogen, wobei $i = i' \wedge j < j'$ oder $i > i' \wedge j = j'$
- Bemerkungen zum MCMP
 - Es gibt einen Algorithmus mit linearer Laufzeit $O(n)$, der eine Klammerung mit Multiplikationsaufwand $\leq 1.155 M_{\text{opt}}$ findet
 - Es gibt einen Algorithmus mit Laufzeit $O(n \log n)$, der eine optimale Klammerung findet



Zusammenfassung

- Dynamische Programmierung = Algorithmenentwurfstechnik, die oft bei Optimierungsproblemen angewandt wird
 - Man muß eine Menge von Entscheidungen treffen, die Bedingungen unterliegen, um eine optimale (min. / max.) Lösung zu erlangen
 - Es kann verschiedene Lösungswege geben
- Allgemein einsetzbar bei rekursiven Verfahren, wenn Teillösungen (von Unterproblemen) mehrfach benötigt werden
- Lösungsansatz: Tabellieren von Teilergebnissen
- Vergleich mit Divide-and-Conquer:
 - Ähnlich: DP berechnet Lösung eines Problems aus Lösungen von Teilproblemen
 - Anders: Teilprobleme werden oft nicht rekursiv gelöst sondern iterativ, beginnend mit den Lösungen der kleinsten Teilprobleme (bottom-up)

G. Zachmann Informatik 2 — SS 10 Dynamische Programmierung 65

Zwei verschiedene Ansätze

- Bottom-up
 - + kontrollierte effiziente Tabellenverwaltung, spart Zeit
 - + spezielle optimierte Berechnungsreihenfolge, spart Platz
 - weitgehende Umcodierung des Originalprogramms erforderlich
 - möglicherweise Berechnung nicht benötigter Werte
- Top-down (Memoisierung, Notizblockmethode)
 - + Originalprogramm wird nur gering oder nicht verändert
 - + nur tatsächlich benötigte Werte werden berechnet
 - eventuell unnötige rekursive Aufrufe
 - Tabellengröße oft nicht optimal

G. Zachmann Informatik 2 — SS 10 Dynamische Programmierung 66