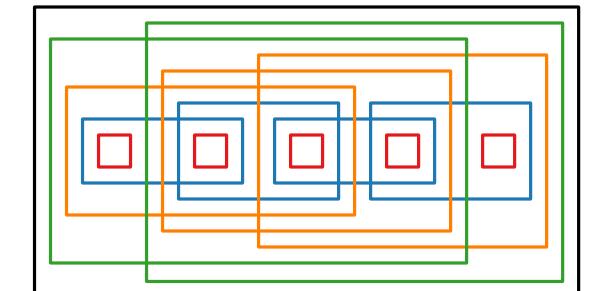
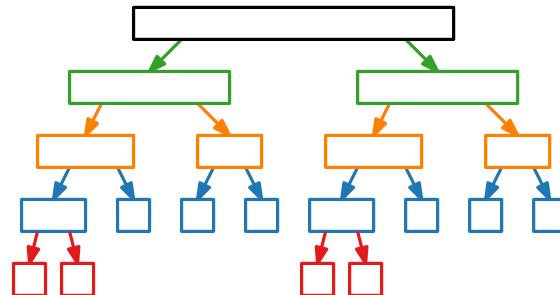




Algorithmen und Datenstrukturen

Vorlesung 22: Dynamisches Programmieren



Entwurfstechniken

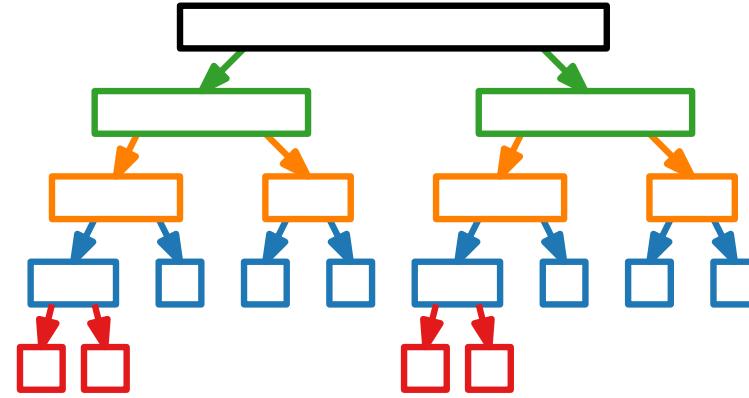
- Inkrementell
- Rekursiv
- Teile und Herrsche
- Randomisiert

Heute:

meint hier das Arbeiten mit einer Tabelle,
nicht das Schreiben eines Computerprogramms.

- Dynamisches Programmieren

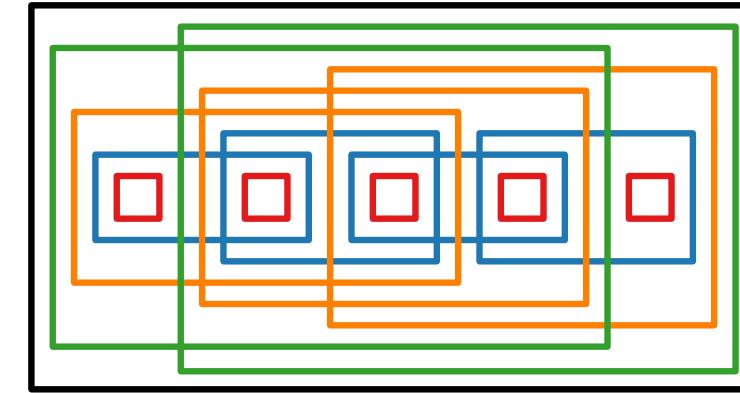
Vergleich



Teile und Herrsche

- zerlegt Instanz rekursiv in **disjunkte** Teilinstanzen

- top-down
- eher für Entscheidungs- oder Berechnungsprobleme



Dynamisches Programmieren

- zerlegt Instanz in **überlappende** Teilinstanzen, d.h. Teilinstanzen haben oft dieselben Teilteilinstanzen.
Lösungen von Teilinstanzen werden zwischengespeichert, nicht neu berechnet.

- meist bottom-up
- meist für Optimierungsprobleme

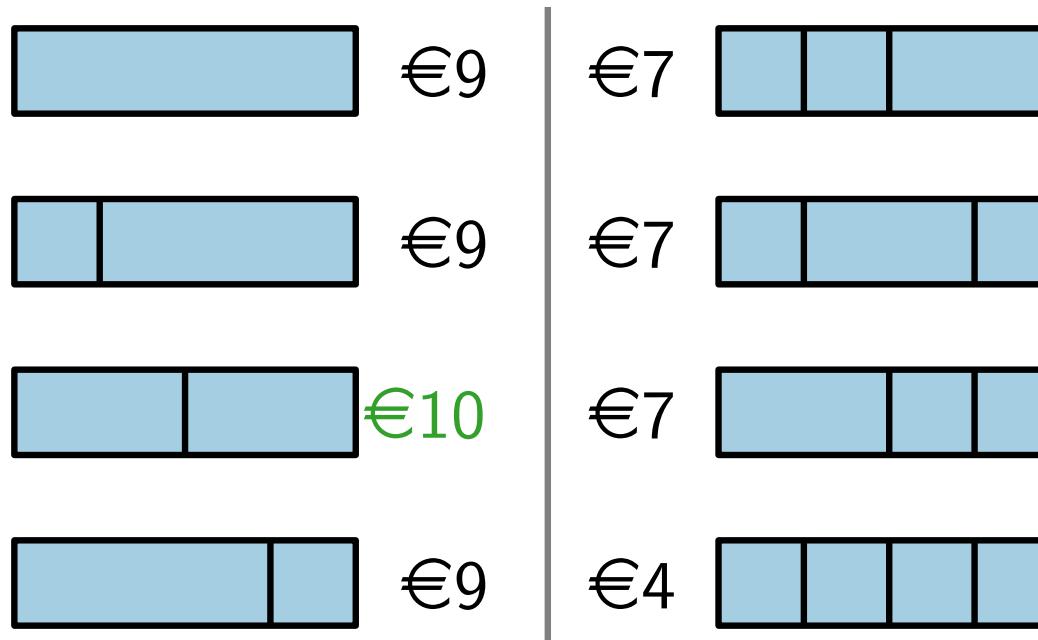
Fahrplan

1. Struktur einer optimalen Lösung charakterisieren
2. Wert einer optimalen Lösung rekursiv definieren
3. Wert einer optimalen Lösung berechnen (meist bottom-up)
4. Optimale Lösung aus berechneter Information konstruieren

Das Zerlegungsproblem

Wir haben einen Stab der Länge n und kennen die Preise p_1, p_2, \dots, p_n für Stäbe der Längen $1, 2, \dots, n$.

Durch welche Zerlegung unseres Stabs können wir unseren Ertrag maximieren?

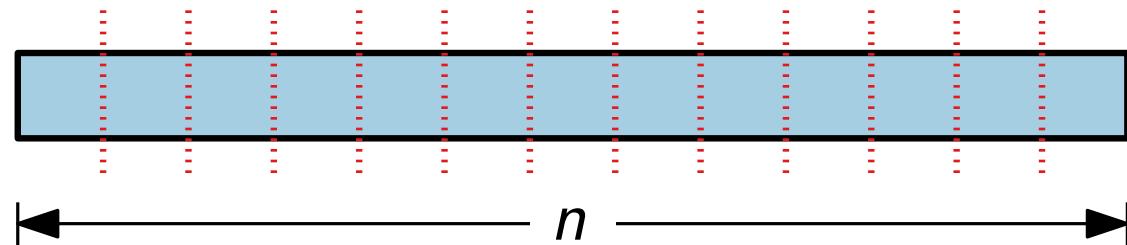


Länge i	1	2	3	4
Preis p_i [in €]	1	5	8	9



Rohe Gewalt

Frage. Wie viele Möglichkeiten gibt es, einen Stab der Länge n zu zerlegen?



Antwort. Können $n - 1$ mal entscheiden: *schneiden* oder *nicht*.

$\Rightarrow 2^{n-1}$ verschiedene Zerlegungen

Also können wir es uns nicht leisten alle Zerlegungen durchzugehen und für jede ihren Ertrag zu berechnen.

Oh, mein Gott!
Das ist ja **exponentiell!**



^{*}) Genauer: die gesuchte Zahl ist die Anzahl $p(n)$ der *Partitionen* der Zahl n , die angibt, auf wie viele Arten man n als Summe von natürlichen Zahlen schreiben kann.

$$\text{Es gilt } p(n) \approx e^{\pi\sqrt{2n/3}} / (4n\sqrt{3}) \in \Theta^*((13,00195...)^{\sqrt{n}}).$$

Ein erster Versuch

Wir haben einen Stab der Länge n und kennen die Preise p_1, p_2, \dots, p_n für Stäbe der Längen $1, 2, \dots, n$.

Beispiel: $n = 4$

Länge i [in m]	1	2	3	4
Preis p_i [in €]	1	5	8	9
Quotient q_i [€/m]	1	$2\frac{1}{2}$	$2\frac{2}{3}$	$2\frac{1}{4}$

Welche Stabzerlegung maximiert den Ertrag?

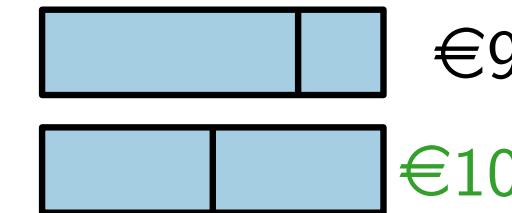
Greedy:

- Berechne für $i = 1, \dots, n$ den Preis pro Meter $q_i = p_i/i$.
- Zerlege Stab in möglichst viele Stücke der Länge i mit q_i max.
- Streiche alle Stablängen $\geq i$ aus der Tabelle und wiederhole den Prozess mit dem Stabrest (falls > 0).

Liefert dieser Greedy-Algorithmus immer das Optimum?

Ja? Beweisen!

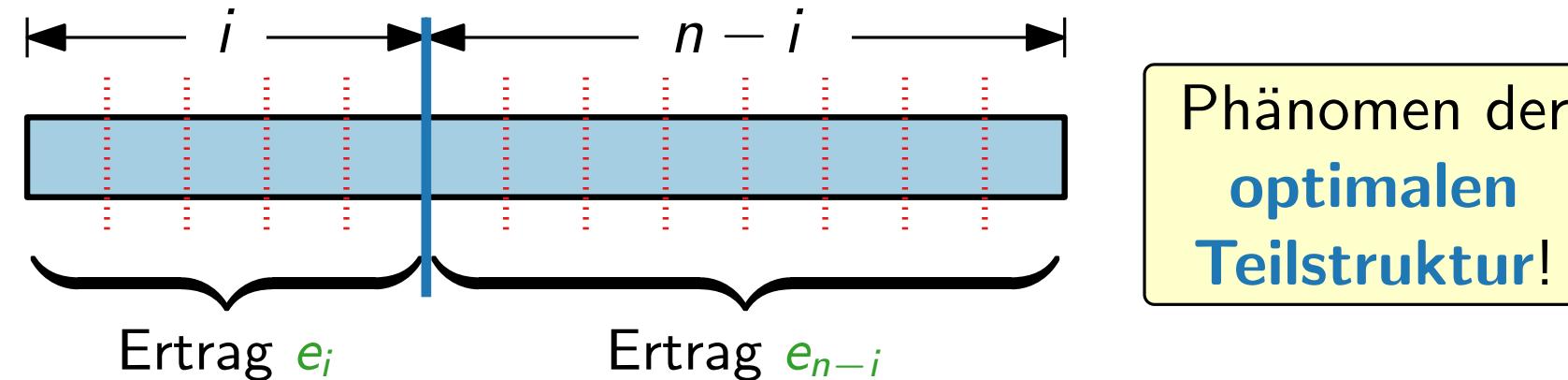
Nein? Gegenbeispiel!



1. Struktur einer optimalen Lösung charakterisieren

Def. Für $i = 1, \dots, n$

sei e_i der maximale Ertrag für einen Stab der Länge i .



Beob. Ein Schnitt zerlegt das Problem in **unabhängige** Teilprobleme.

2. Wert einer optimalen Lösung rekursiv definieren

Wissen nicht, *welcher* Schnitt in einer opt. Lösung vorkommt.

Also probieren wir einfach *alle* möglichen Schnitte aus:

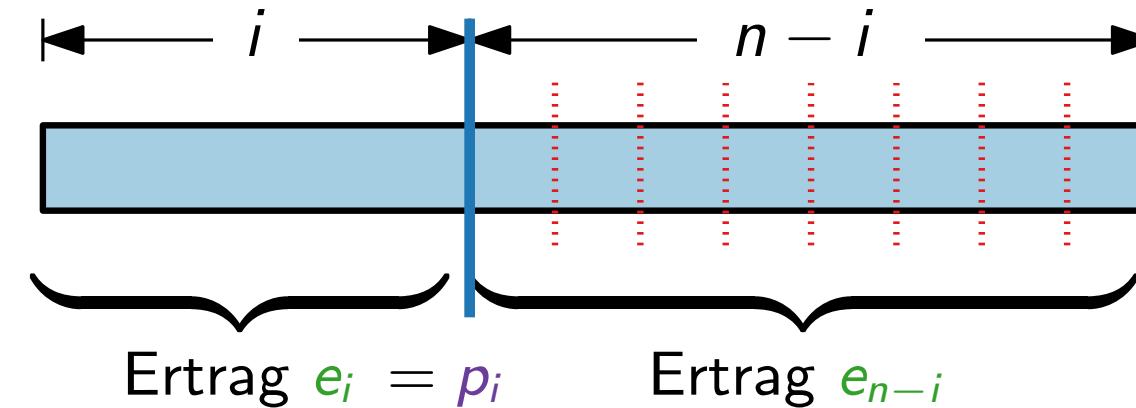
$$e_n = \max\{ p_n, e_1 + e_{n-1}, e_2 + e_{n-2}, \dots, e_{n-1} + e_1 \}$$

2. Wert einer optimalen Lösung rekursiv definieren

$$e_n = \max\{ p_n, e_1 + e_{n-1}, e_2 + e_{n-2}, \dots, e_{n-1} + e_1 \}$$

Kleine Verbesserung:

Verbiete weitere Schnitte im linken Teilstück!



Also gilt:

$$\begin{aligned} e_n &= \max\{ p_n, p_1 + e_{n-1}, p_2 + e_{n-2}, \dots, p_{n-1} + e_1 \} \\ &= \max_{1 \leq i \leq n} \{ p_i + e_{n-i} \}, \quad \text{wobei } e_0 := 0. \end{aligned}$$

Vorteil: Wert einer optimalen Lösung ist Summe aus einer Zahl der Eingabe und *einem* Wert einer optimalen Teillösung.

3. Wert einer optimalen Lösung berechnen: top-down

Wir wissen: $e_n = \max_{1 \leq i \leq n} \{ p_i + e_{n-i} \}$, wobei $e_0 := 0$.

STANGENZERLEGUNG(int[] p , int $n = p.length$)

```

if  $n == 0$  then return 0
 $q = -\infty$ 
for  $i = 1$  to  $n$  do
     $q = \max\{q, p[i] + \text{STANGENZERLEGUNG}(p, n - i)\}$ 
return  $q$ 
```

Laufzeit. Sei $A(n)$ die Gesamtzahl von Aufrufen von $\text{STANGENZERLEGUNG}(p, \cdot)$ beim Ausführen von $\text{STANGENZERLEGUNG}(p, n)$.

$$\Rightarrow A(0) = 1$$

$$\text{und } A(n) = 1 + \sum_{i=1}^n A(n-i) = 1 + \underbrace{\sum_{j=0}^{n-1} A(j)}_{1 + 2 + 4 + 8 + \dots + 2^{n-1}} = 2^n$$

Beweis?!



3. Wert einer optimalen Lösung berechnen: mit Tabelle

Zeit-Speicher-Tausch (engl. *time-memory trade-off*)

MEMOSTANGENZERLEGUNG(int[] p , int $n = p.length$)

```
e = new int[0...n]
e[0] = 0
for i = 1 to n do
    e[i] = -∞
```

return HAUPTSTANGENZERLEGUNG(p, n, e)

Laufzeit?

- Wie letzte Folie?
- Asymptotisch schneller?

HAUPTSTANGENZERLEGUNG(int[] p , int n , int[] e)

```
if e[n] > -∞ then return e[n]
q = -∞
for i = 1 to n do
    q = max{q,  $p[i] + \text{HAUPTSTANGENZERLEGUNG}(p, n-i, e)$ }
e[n] = q; return q
```

3. Wert einer optimalen Lösung berechnen: bottom-up

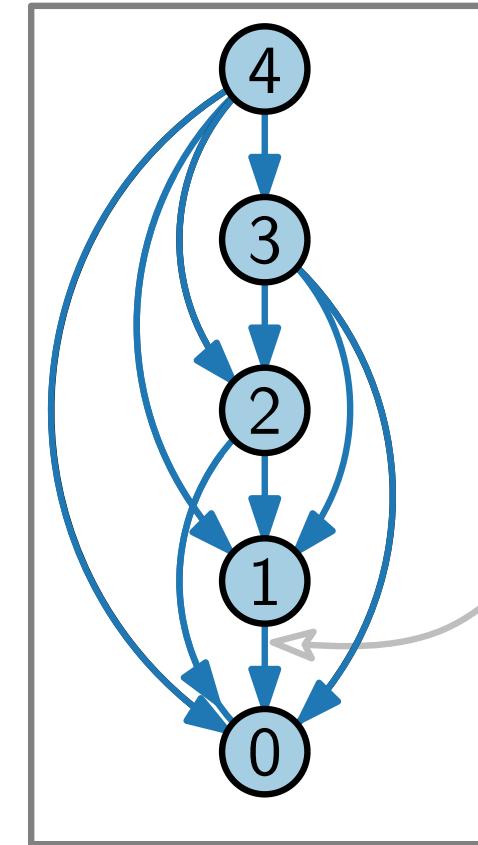
BOTTOMUPSTANGENZERLEGUNG(int[] p , int n)

```

e = new int[0...n]
e[0] = 0
for j = 1 to n do
    q = -∞
    for i = 1 to j do
        q = max{q, p[i] + e[j - i]}
    e[j] = q
return q

```

Neu: *kein*
rekursiver
Aufruf!



Kante (j, i) bedeutet:
Teilinstanz j benutzt
Wert einer opt. Lösung
von Teilinstanz i .

Graph der Teilinstanzen

Beob. Die Anzahl der Kanten im Graphen ist proportional zur Laufzeit des DP (Anzahl Additionen).

Satz. BOTTUPSZERL() und MEMOSZERL() laufen in $\mathcal{O}(n^2)$ Zeit.

4. Optimale Lösung aus berechneten Informationen konstruieren

ERWEITERTEBOTTOMUPZERLEGUNG(int[] p , int[] e , int[] ℓ , int n)

$e[0] = 0$

for $j = 1$ **to** n **do**

$q = -\infty$

for $i = 1$ **to** j **do**

if $q < p[i] + e[j-i]$ **then**

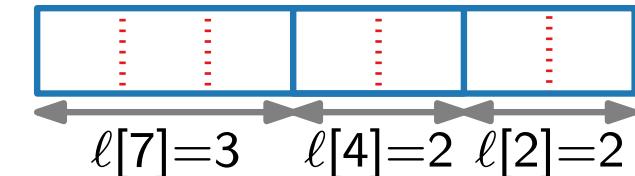
$q = p[i] + e[j-i]$

$\ell[j] = i$

 } $q = \max\{q, p[i] + e[j-i]\}$

$e[j] = q$

// merke Länge des linkesten Teilstücks



GIBZERLEGUNGAUS(int[] p , int n)

$\ell = \text{new int}[0 \dots n]; e = \text{new int}[0 \dots n]$

ERWEITERTEBOTTOMUPZERLEGUNG(p, e, ℓ, n)

while $n > 0$ **do**

 print $\ell[n]; n = n - \ell[n]$ // gib wiederholt Länge des linkesten
// Teilstücks des Reststabs aus

Längste Wege

Gegeben: ungewichteter gerichteter Graph G
mit $s, t \in V(G)$, $s \neq t$, aber t von s erreichbar.

Gesucht: ein längster einfacher $s-t$ -Weg,
d.h. eine Folge $\langle s = v_0, v_1, \dots, v_k = t \rangle$ mit
 $v_0 v_1, \dots, v_{k-1} v_k \in E$, $v_i \neq v_j$ (für $i \neq j$) und k maximal.

$\langle s, u, t \rangle$ ist ein längster einfacher $s-t$ -Weg.

Aber:

$\langle s, u \rangle$ ist *kein* längster einfacher $s-u$ -Weg;

$\langle s, v, t, u \rangle$ ist ein längster einfacher $s-u$ -Weg!

Fahrplan

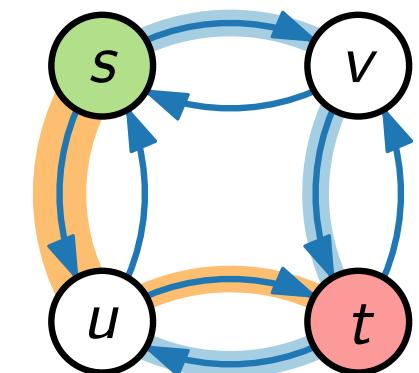
1. Struktur einer optimalen Lösung charakterisieren



2. Wert einer optimalen Lösung rekursiv definieren

3. Wert einer optimalen Lösung berechnen (meist bottom-up)

*) Es ist NP-schwer für (G, s, t, k) zu entscheiden, ob G einen einfachen $s-t$ -Weg der Länge k enthält. (Vgl. Hamilton-Weg!)



Längste Wege in azyklischen Graphen

Gegeben: gerichteter **kreisfreier** Graph G mit Kantengewichten $w: G(E) \rightarrow \mathbb{R}_{\geq 0}$ und $s, t \in V(G)$, $s \neq t$, aber t von s erreichbar.

Gesucht: ein längster s - t -Weg.

Beobachtung 1. In kreisfreien Graphen sind alle Wege einfach.

Beobachtung 2. Dieses Problem hat optimale Teilstruktur, denn:

Angenommen, ein längster s - t -Weg π geht durch u , d.h.

$$\pi = s \xrightarrow{\pi_{su}} u \xrightarrow{\pi_{ut}} t.$$

Dann gilt:

π_{su} ist längster s - u -Weg; π_{ut} ist längster u - t -Weg – sonst wäre π kein längster s - t -Weg.

Außerdem gilt $V(\pi_{su}) \cap V(\pi_{ut}) = \{u\}$;
sonst gäbe es einen Kreis!

Algorithmus nach Fahrplan

- Struktur einer optimalen Lösung charakterisieren



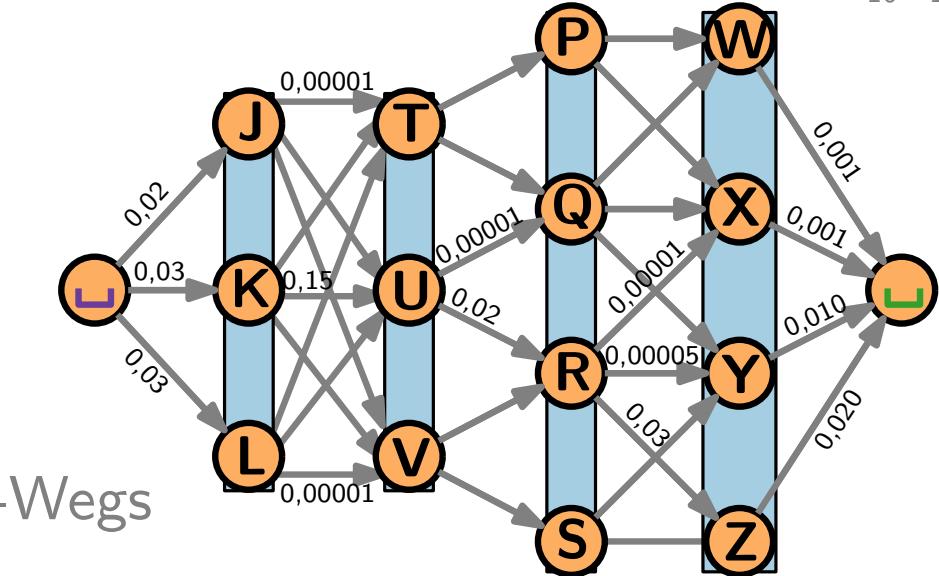
- Wert einer optimalen Lösung rekursiv definieren

$d_s = 0$ // Länge eines längsten s - v -Wegs

$$d_v = \max_{u: uv \in E(G)} d_u + w(u, v) \quad \text{so!}$$

- Wert einer optimalen Lösung berechnen (hier bottom-up)

- G topologisch sortieren
- d -Werte initialisieren: $d_s = 0$ und $d_v = -\infty$ für alle $v \neq s$
- **for**-Schleife durch Knoten v.l.n.r. d -Werte berechnen



Übrigens: Kürzeste Wege in kreisfreien Graphen kann man genauso berechnen (mit min statt max und $+\infty$ statt $-\infty$).

Genauso kann man auch das „T9-Problem“ lösen (mit \cdot statt $+$).

Und jetzt?

Im Buch [CLRS] werden weitere, praxisrelevante Probleme mit dynamischem Programmieren gelöst:

- Ketten von Matrixmultiplikationen
- Längste gemeinsame Teilfolge (in Zeichenketten)
- Optimale binäre Suchbäume