Algorithmen und Datenstrukturen

Prof. Dr. Ralf Möller

Universität zu Lübeck

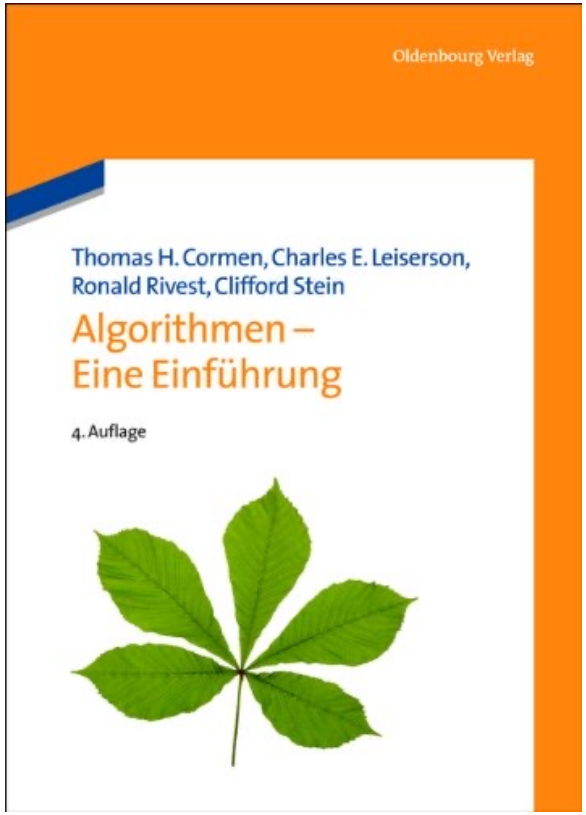
Institut für Informationssysteme

Magnus Bender (Übungen)

sowie viele Tutoren

Danksagung

- Animationen wurden übernommen aus dem Kurs: CS 3343 Analysis of Algorithms von Jianhua Ruan
- Inhalte sind angelehnt an
- Wer ein “Skript“ sucht, ist hier richtig



Algorithmen-Entwurfsmuster

Beispielproblemklasse: Optimierungsprobleme

- Beladungsprobleme
- Anordnungsprobleme
- Planungsprobleme

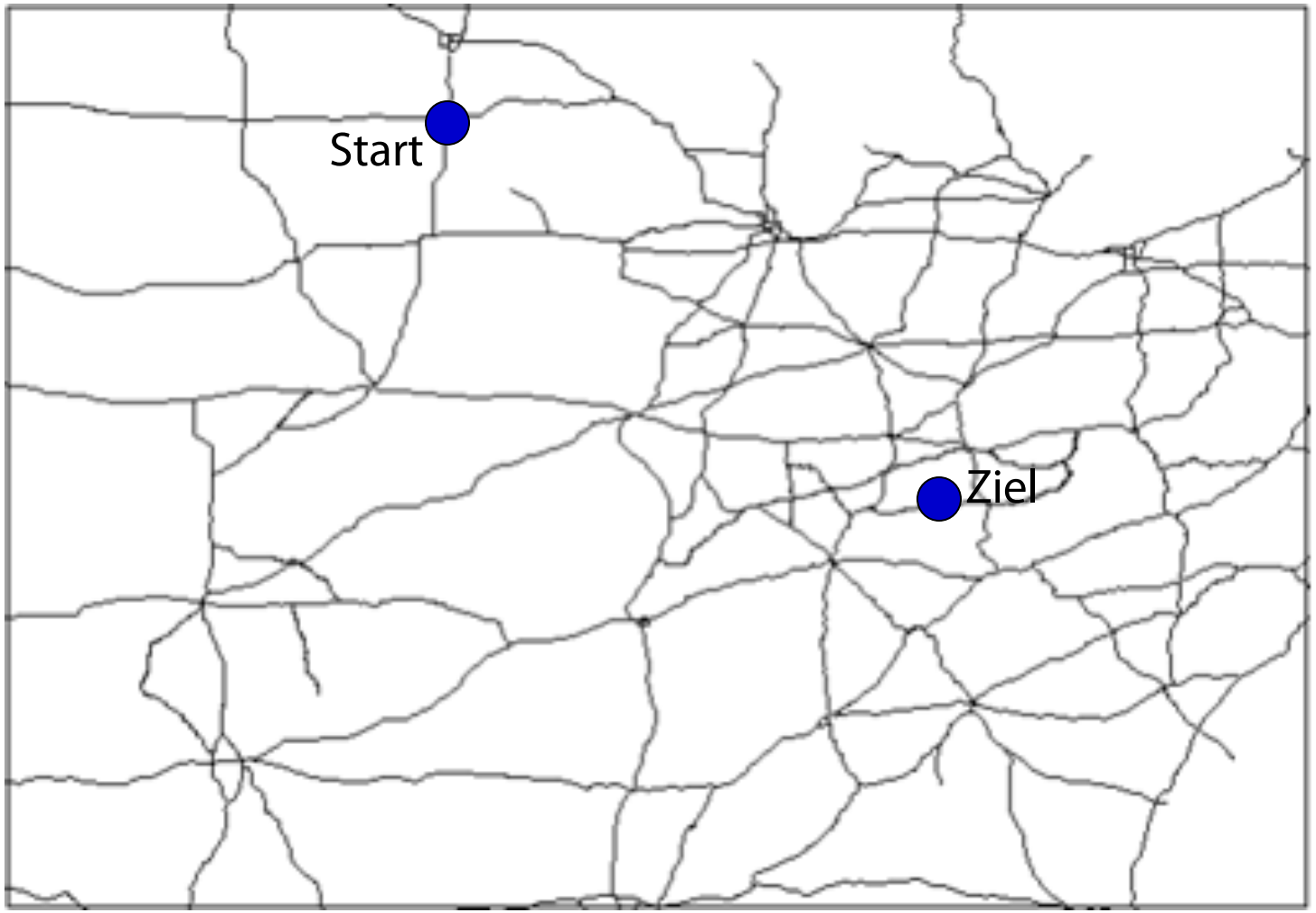
Naiver Ansatz (Brute-Force) ist fast immer kombinatorisch oder die optimale Lösung wird nicht gefunden (Unvollständigkeit)

Entwurfsziel: Vermeidung von Kombinatorik
unter Beibehaltung der Vollständigkeit

Überblick

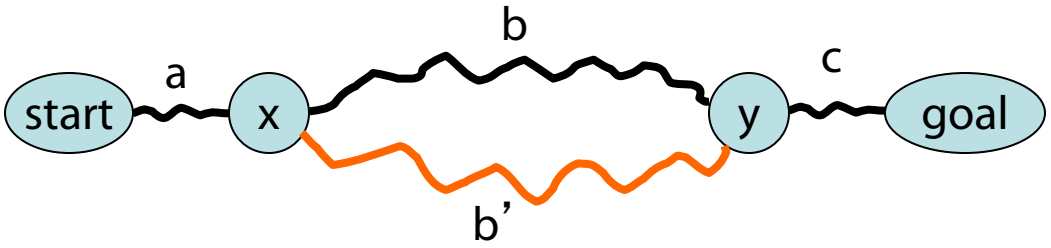
- Dynamische Programmierung
  - Name historisch begründet (sollte gut klingen, kein Bezug zum heutigen Begriff der Programmierung)
  - Bellmans Optimalitätsprinzip
  - Fragestellung: Wie können Problemlösungen aus Lösungen für Teilprobleme hergeleitet werden, so dass Vollständigkeit erreicht wird
- Gierige Algorithmen (Greedy Algorithms)
  - Verfolgung nur des augenscheinlich aktuell günstigsten Wegs zum Ziel
  - Fragestellung: Wann sind gierige Algorithmen vollständig?

Kürzeste Wege als Optimierungsproblem betrachtet



Bellmans Optimalitätsprinzip

Behauptung: Falls ein Weg start⟶goal optimal ist, muss jeder Teilweg x⟶y, der auf dem optimalen Pfad liegt, optimal sein



a + b + c ist kürzester Weg

b' < b

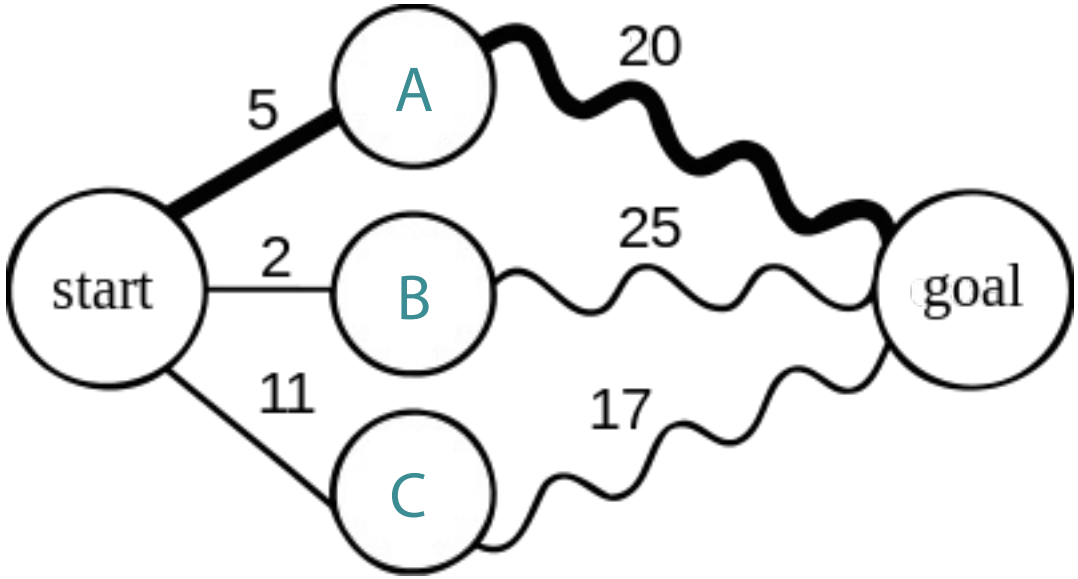
a + b' + c < a + b + c

Beweis durch Widerspruch:

- Falls Teilweg b zwischen x und y nicht kürzester Weg, können wir ihn durch kürzeren Weg b' ersetzen
- Dadurch wird der Gesamtweg kürzer und der betrachtete Weg start⟶goal ist nicht optimal: Widerspruch
- Also muss b der kürzeste Weg von x nach y sein

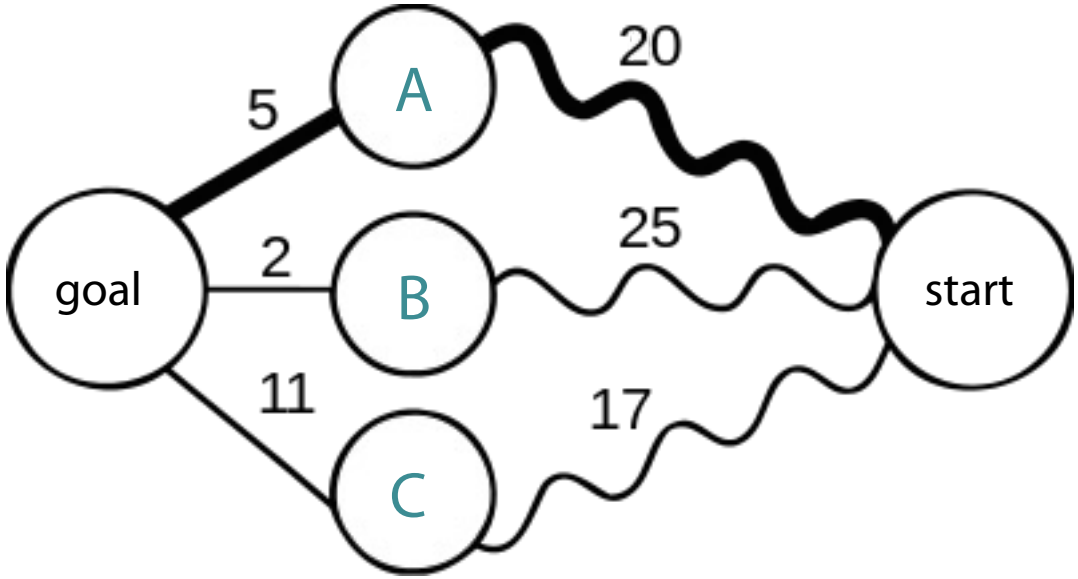
R. Bellman, *On the Theory of Dynamic Programming*, Proceedings of the National Academy of Sciences, **1952**

Problem des kürzesten Pfads



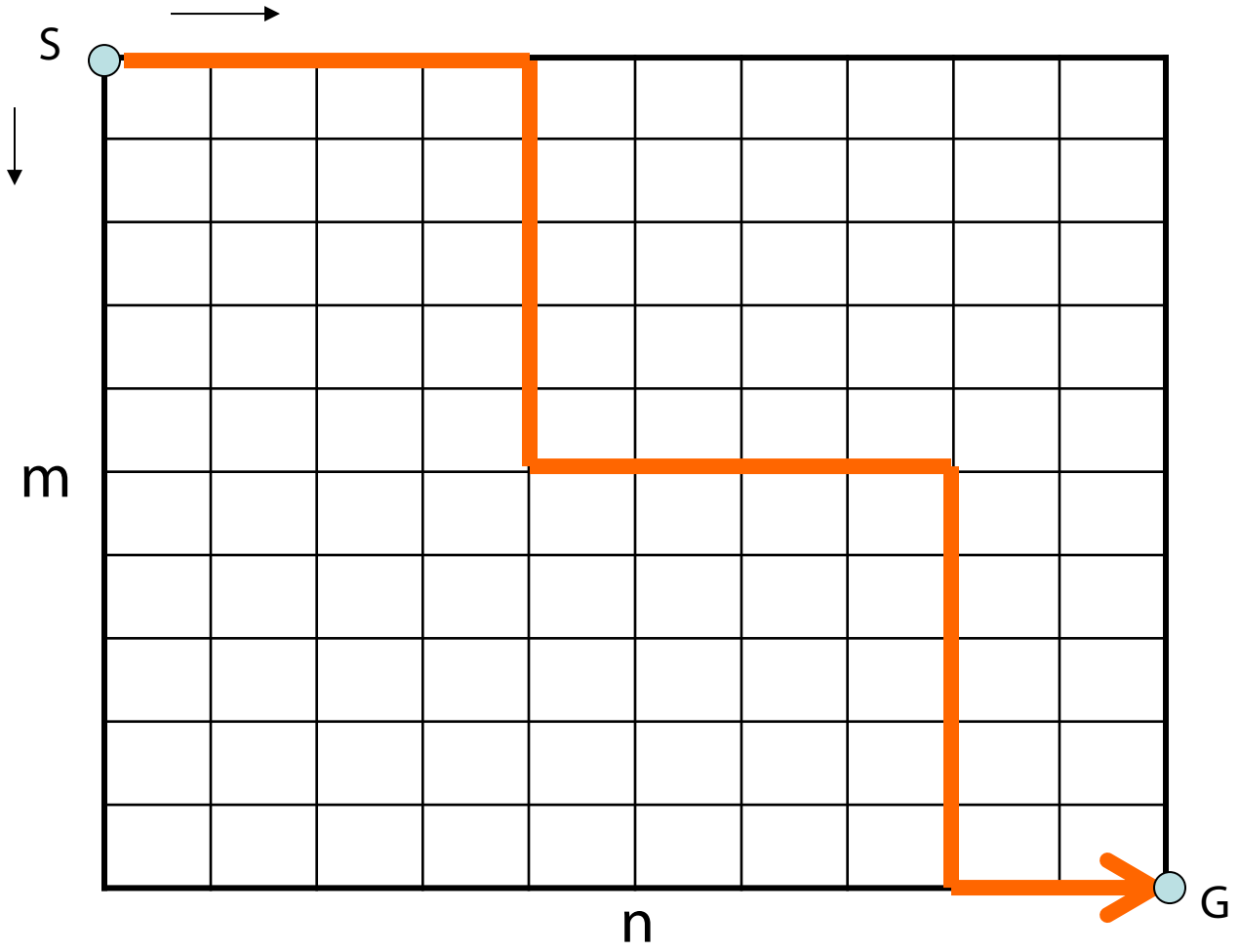
$$SP(start, goal) = \min \begin{cases} dist(start, A) + SP(A, goal) \\ dist(start, B) + SP(B, goal) \\ dist(start, C) + SP(C, goal) \end{cases}$$

Problem des kürzesten Pfads



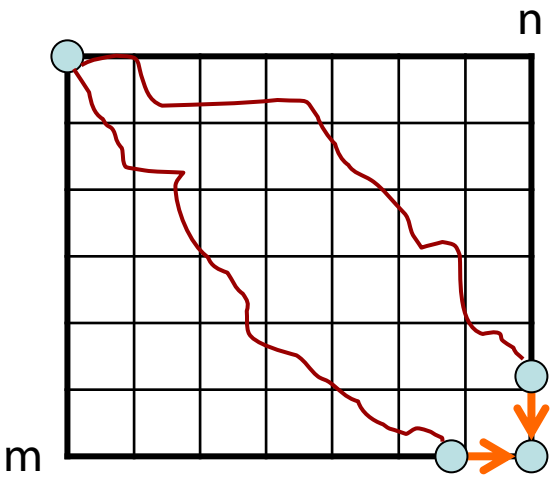
$$SP(start, goal) = \min \begin{cases} dist(start, A) + SP(A, goal) \\ dist(start, B) + SP(B, goal) \\ dist(start, C) + SP(C, goal) \end{cases}$$

Ein spezielles Szenario für den kürzesten Pfad



Jeder Kante sind (verschiedene) Kosten zugeordnet. Schritte nur nach rechts oder nach unten möglich. Ziel: Kürzester Weg von S nach G.

Betrachtung eines Lösungsansatzes

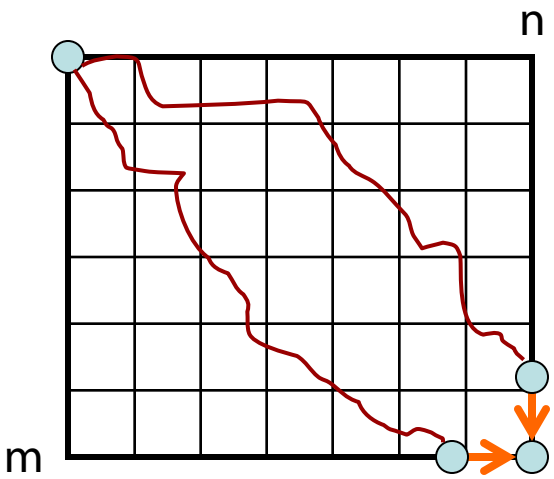


Annahme: Kürzester Pfad gefunden

- Folgende Kanten kommen vor
  - (m, n-1) nach (m, n) Fall 1
  - (m-1, n) nach (m, n) Fall 2
- Fall 1:
  - Suche kürzesten Pfad von (0, 0) nach (m, n-1)
  - Length(SP(0, 0, m, n-1)) + dist(m, n-1, m, n) ist die kürzeste Pfadlänge
- Fall 2:
  - Suche kürzesten Pfad von (0, 0) nach (m-1, n)
  - Length(SP(0, 0, m-1, n)) + dist(m-1, n, m, n) ist die kürzeste Pfadlänge
- Wir wissen nicht, welcher Fall eintreten wird
  - Falls wir die zwei Pfade haben, können wir aber vergleichen
  - Dann lässt sich der kürzeste Pfad leicht bestimmen

Rekursiver Lösungsansatz

Sei F(i, j) = length(SP(0, 0, i, j)). => F(m, n) die Länge des SP von (0, 0) nach (m, n)

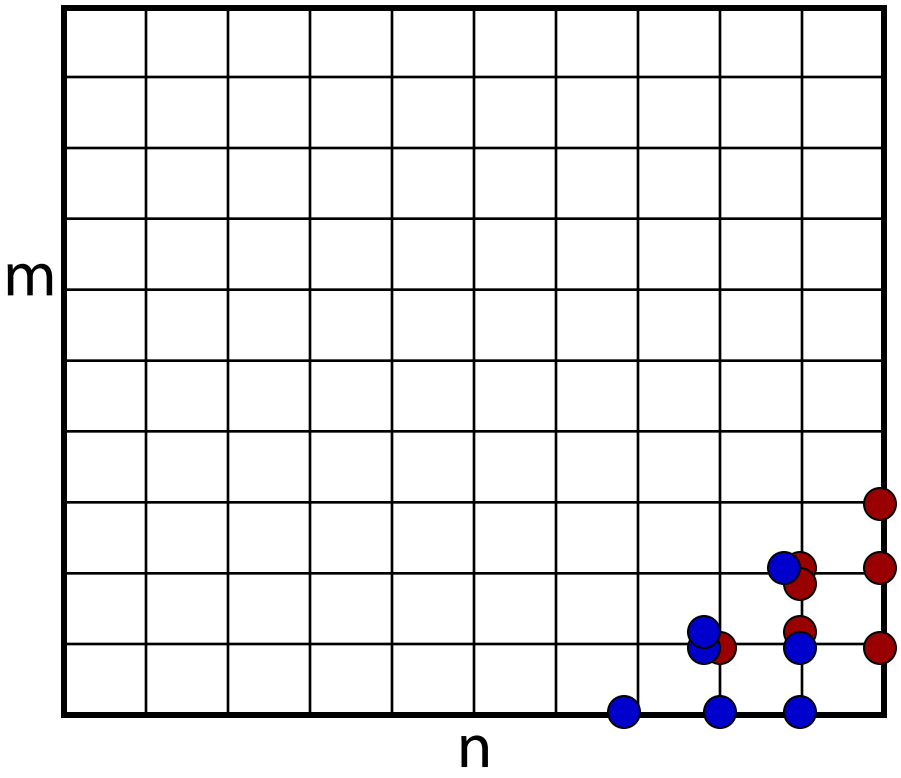


$$F(m, n) = \min \begin{cases} F(m-1, n) + dist(m-1, n, m, n) \\ F(m, n-1) + dist(m, n-1, m, n) \end{cases}$$

Generalisierung

$$F(i, j) = \min \begin{cases} F(i-1, j) + dist(i-1, j, i, j) \\ F(i, j-1) + dist(i, j-1, i, j) \end{cases}$$

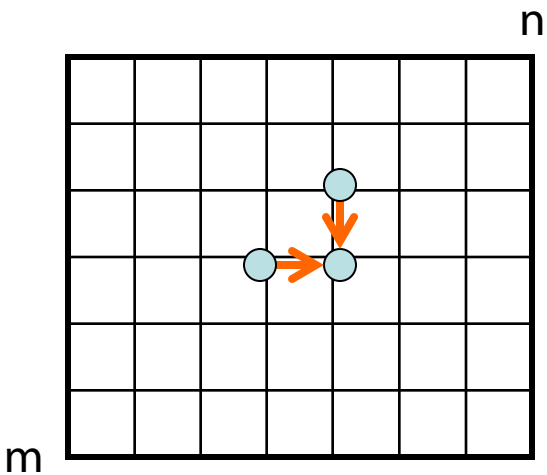
Vermeide Neuberechnungen



- Um den kürzesten Pfad von (0, 0) nach (m, n) zu finden, benötigen wir die kürzesten Pfade nach (m-1, n) und (m, n-1)
- Bei rekursivem Aufruf werden Lösungen immer wieder neu berechnet
- Strategie:
  Löse Teilprobleme in richtiger Reihenfolge, so dass redundante Berechnungen vermieden werden

Prinzip der Dynamischen Programmierung

Sei F(i, j) = SP(0, 0, i, j). => F(m, n) die Länge von SP von (0, 0) nach (m, n)

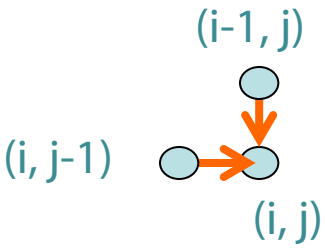


$$F(i, j) = \min \begin{cases} F(i-1, j) + dist(i-1, j, i, j) \\ F(i, j-1) + dist(i, j-1, i, j) \end{cases}$$

Grenzfall: i = 0 oder j = 0
Leicht zu identifizieren

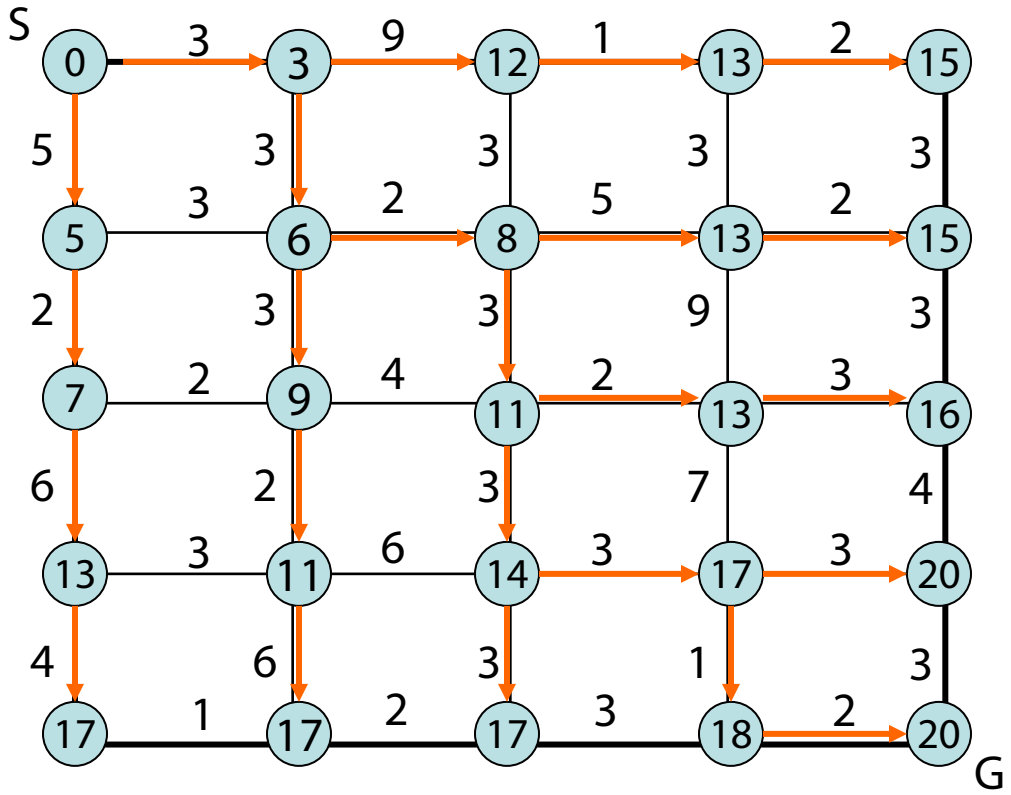
i = 1 .. m, j = 1 .. n

Anzahl der Unterprobleme: m · n



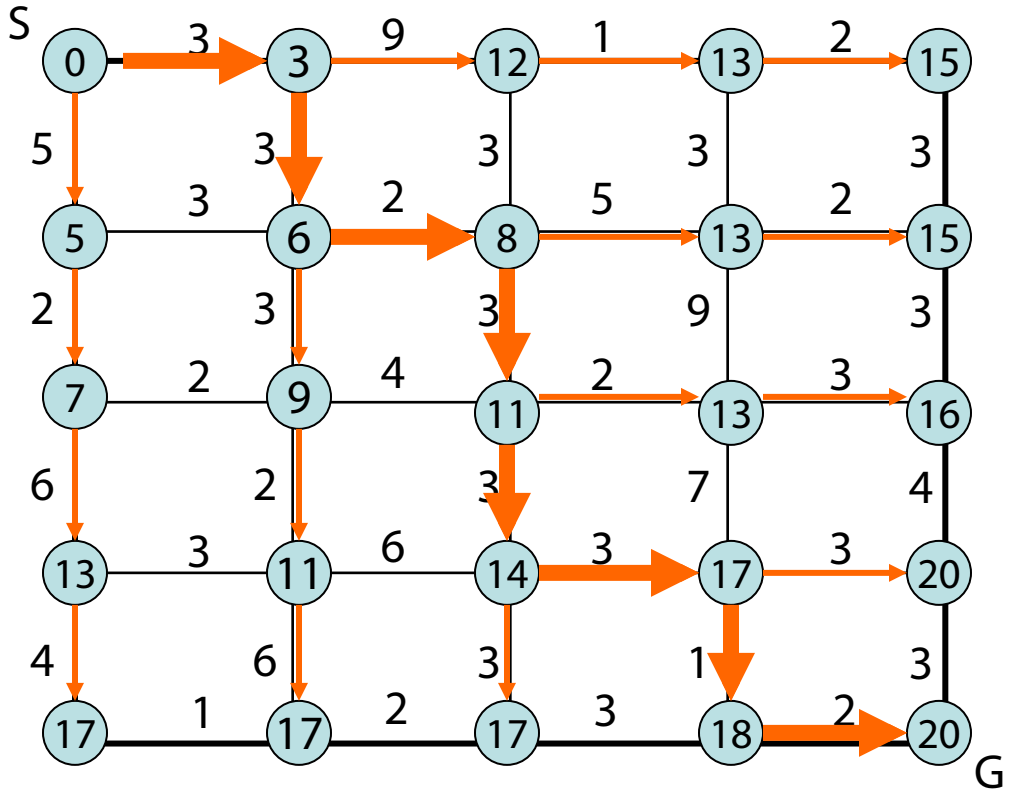
Datenabhängigkeit definiert Anordnung der Teilprobleme; hier von links nach rechts, von oben nach unten

Dynamische Programmierung: Illustration



$$F(i, j) = \min \begin{cases} F(i-1, j) + dist(i-1, j, i, j) \\ F(i, j-1) + dist(i, j-1, i, j) \end{cases}$$

Rückverfolgung zur Bestimmung der Lösung



Kürzester Pfad hat die Länge 20

Elemente der Dynamischen Programmierung

- Optimale Unterstrukturen
  - Optimale Lösungen des ursprünglichen Problems enthält optimale Lösungen von Teilproblemen
- Überlappung von Teilproblemen
  - Einige Teilprobleme kommen in vielen Lösungen vor
  - Formulierung von Lösungen als Rekurrenz auf Teillösungen
- Speicherung und Wiederverwendung
  - Wähle Ordnung der Teilprobleme sorgfältig aus, so dass jedes Teilproblem eines höheren Problems vorher gelöst wurde und die Lösung wiederverwendet werden kann

Überblick über betrachtete Probleme

- Sequenzausrichtungsprobleme (mehrere Varianten)
  - Vergleich zweier Zeichenketten auf Ähnlichkeit
    - Definition von Ähnlichkeit nötig
    - Nützlich in vielen Anwendungen
      - Vergleich von DNA-Sequenzen, Aufsätzen, Quellcode
  - Beispielproblem: Longest-Common-Subsequence (LCS)
- Anordnungsprobleme
- Planungsprobleme (Scheduling)
- Rucksackproblem (mehrere Varianten)

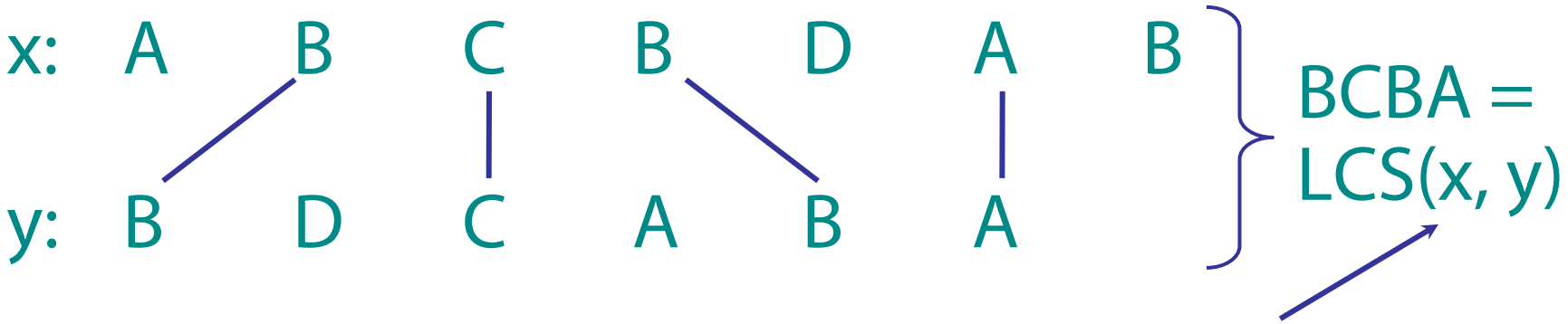
Gemeinsame Teilsequenz (Common Subsequence)

- Teilsequenz einer Zeichenkette: Zeichenkette mit 0 oder mehr ausgelassenen Zeichen
- Gemeinsame Teilsequenz von zwei Zeichenketten
  - Teilsequenz von beiden Zeichenketten
- Beispiel:
  - x = <A B C B D A B>, y = <B D C A B A>
  - <B C> und <A A> sind gemeinsame Teilsequenzen von x und y

Längste gemeinsame Teilsequenz

Gegeben sei ein Alphabet Σ und zwei Sequenzen x[1..m] und y[1..n] in denen jeder Buchstabe aus Σ vorkommt.
Aufgabe: Bestimme eine längste gemeinsame Teilsequenz (longest common subsequence, LCS)

- NB: „eine“ längste, nicht die längste



Brute-Force-Algorithmus?

Prüfe jede Teilsequenz von x[1..m] und prüfe, ob es sich auch um eine Teilsequenz von y[1..n] handelt

Analyse:

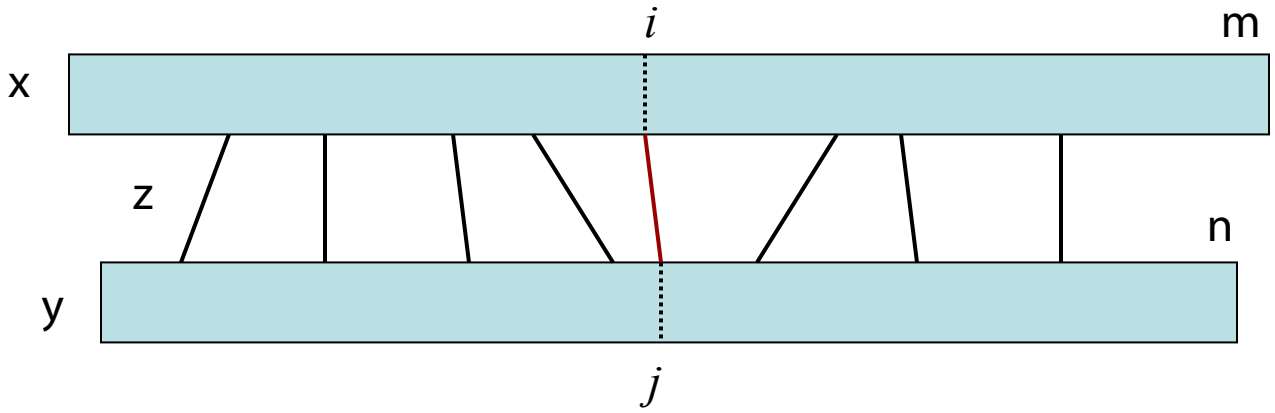
- 2^m Teilsequenzen in x vorhanden (jeder Bitvektor der Länge m bestimmt unterschiedliche Teilsequenz)
- Die Zeitfunktion dieses Algorithmus wäre in $\Theta(2^m)$, der Algorithmus also exponentiell ($\longrightarrow$ nicht praxistauglich)

Auf dem Weg zu einer besseren Strategie:

- Ansatz der dynamischen Programmierung
  - Bestimme opt. Substruktur, überlappende Teilprobleme
- Zunächst: Bestimmung der Länge eines LCS, dann Bestimmung eines LCS

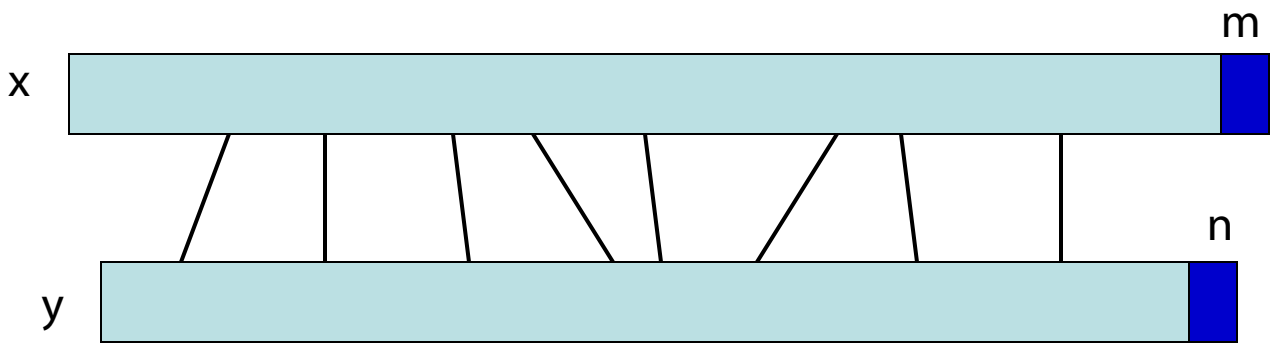
Optimale Substruktur

- Falls z=LCS(x, y), dann gilt für jeden Präfix u:
  uz ist ein LCS von ux und uy



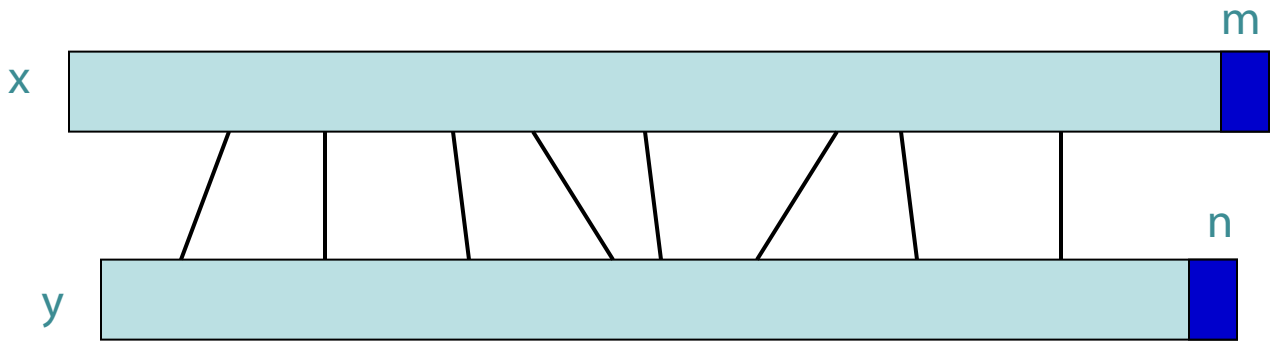
- Teilprobleme: Finde LCS von Präfixen von x und y

Rekursiver Ansatz



- Fall 1: x[m]=y[n]: Es gibt **einen** optimalen LCS in dem x[m] mit y[n] abgeglichen wird → Finde LCS (x[1..m-1], y[1..n-1])
- Fall 2: x[m]≠ y[n]: Einer könnte in LCS sein
  - Fall 2.1: x[m] nicht in LCS → Finde LCS (x[1..m-1], y[1..n])
  - Fall 2.2: y[n] nicht in LCS → Finde LCS (x[1..m], y[1..n-1])

Rekursiver Ansatz



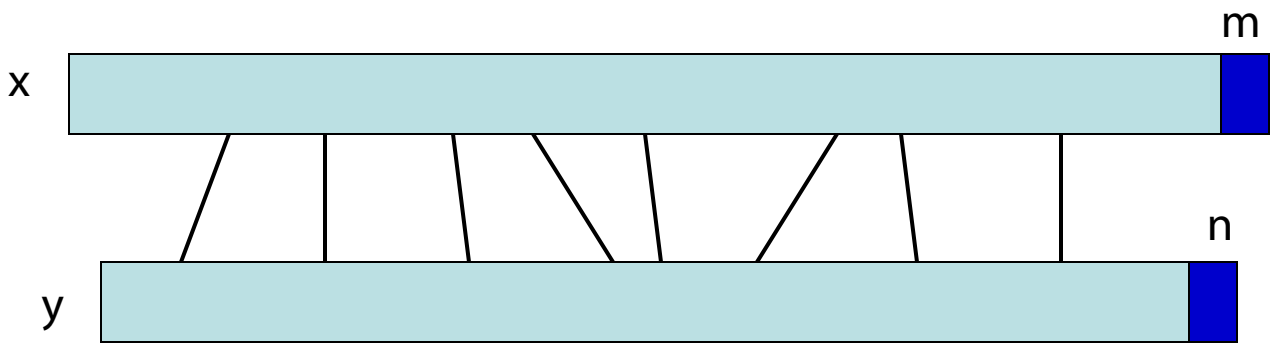
- Fall 1: x[m]=y[n]
  - LCS(x, y) = LCS(x[1..m-1], y[1..n-1]) || x[m]
- Fall 2: x[m] ≠ y[n]
  - LCS(x, y) = LCS(x[1..m-1], y[1..n]) oder
    LCS(x[1..m], y[1..n-1])

Reduziere beide Sequenzen um 1 Zeichen

Konkatenierung

Reduziere eine der Sequenzen um 1 Zeichen

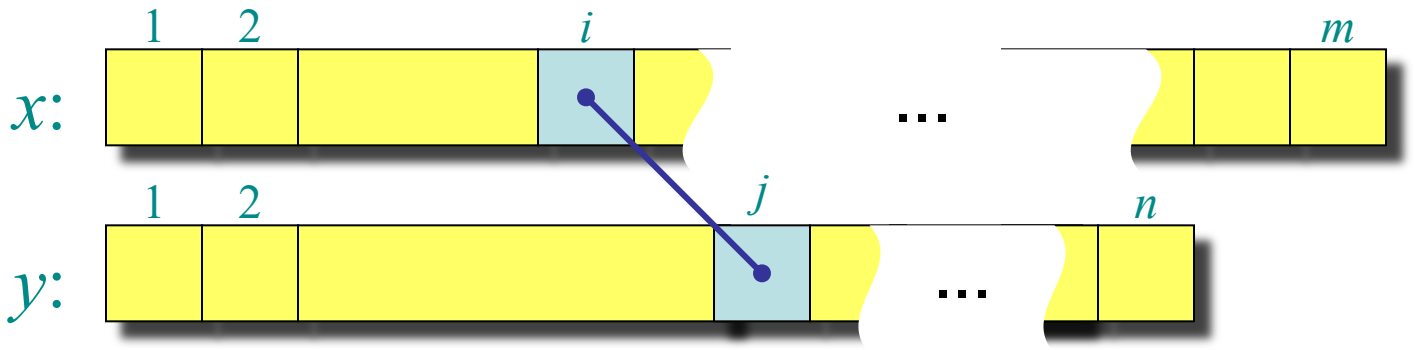
Finden der Länge eines LCS



- Sei c[i, j] die Länge von LCS(x[1..i], y[1..j])
  dann ist c[m, n] die Länge von LCS(x, y)
- Falls x[m] = y[n] dann
  c[m, n] = c[m-1, n-1] + 1
- Falls x[m] ≠ y[n] dann
  c[m, n] = max({ c[m-1, n], c[m, n-1] })

Generalisierung: Rekursive Formulierung

$$c[i, j] = \begin{cases} c[i-1, j-1] + 1 & \text{falls } x[i] = y[j], \\ \max\{c[i-1, j], c[i, j-1]\} & \text{sonst} \end{cases}$$



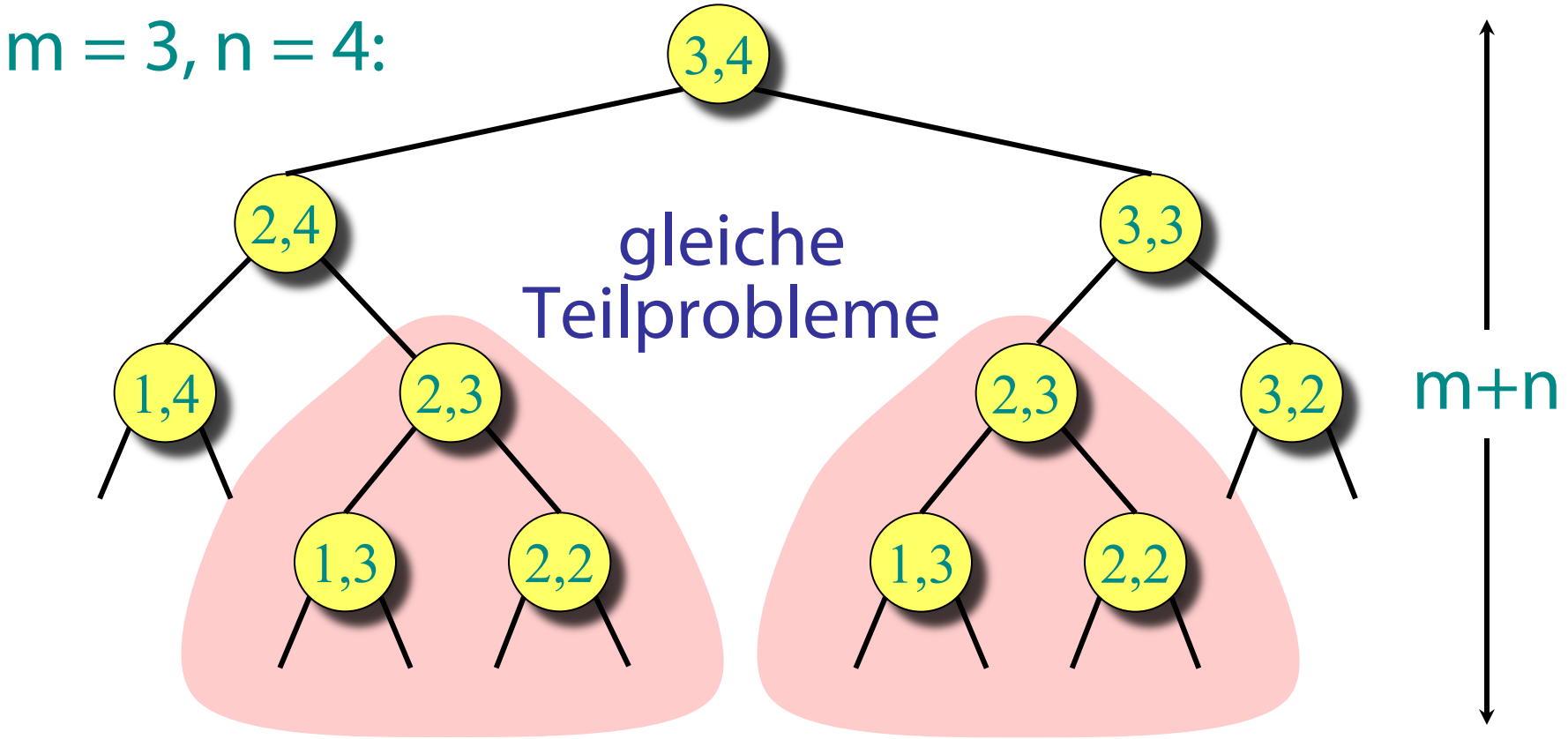
Rekursiver Algorithmus für LCS

```
function LCS(x, y, i, j)
  if x[i] == y[j]
    c[i, j] = LCS(x, y, i-1, j-1) + 1
  else
    c[i, j] = maximum([LCS(x, y, i-1, j),
                       LCS(x, y, i, j-1)])
  end
end
```

Schlimmster Fall: x[i] ≠ y[j]

dann zwei Subprobleme, jedes mit nur einer Dekrementierung (um 1)

Rekursionsbaum

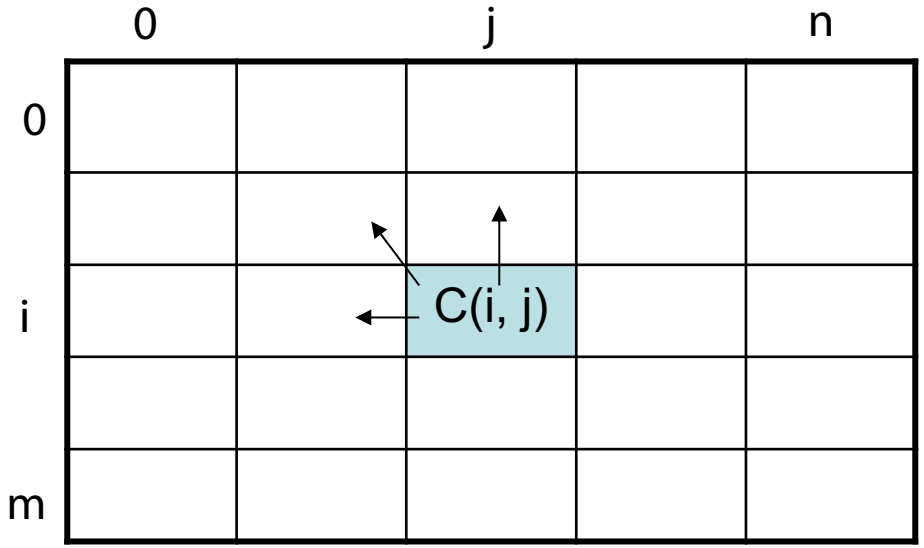


Höhe = m + n ⇒ potentiell exponentieller Aufwand
mit wiederholter Lösung gleicher Teilprobleme!

Dynamische Programmierung

- Finde richtige Anordnung der Teilprobleme
- Gesamtanzahl der Teilprobleme: $m \cdot n$

$$c[i, j] = \begin{cases} c[i-1, j-1] + 1 & \text{falls } x[i] = y[j], \\ \max(\{c[i-1, j], c[i, j-1]\}) & \text{sonst.} \end{cases}$$



Algorithmus LCS

Dieses Array beginnt mit Index 0, man könnte den Code auch für Arrays beginnend mit Index 1 umschreiben.

```
function lcs_length(X, Y)
  m = length(X); n = length(Y)
  c = ZeroIndexArray{Int}(undef, m+1, n+1)
  for i = 0:m  c[i, 0] = 0 end # Sonderfall: Y[0]
  for j = 0:n  c[0, j] = 0 end # Sonderfall: X[0]
  for i = 1:m                  # für alle X[i]
    for j = 1:n                # für alle Y[j]
      if X[i] == Y[j]
        c[i, j] = c[i-1, j-1] + 1
      else
        c[i, j] = maximum([c[i-1, j], c[i, j-1]])
  end end end
  return c
end
```

LCS Anwendungsbeispiel

- X = ABCB
- Y = BDCAB

Was ist der LCS von X und Y?

LCS(X, Y) = BCB

X = A **B** **C** **B**

Y = **B** D **C** A **B**

LCS Beispiel (0)

ABCB
BDCAB

i	j	0	1	2	3	4	5
		Y[j]	**B**	**D**	**C**	**A**	**B**
0	X[i]						
1	**A**						
2	**B**						
3	**C**						
4	**B**						

X = ABCB; m = |X| = 4
Y = BDCAB; n = |Y| = 5
Alloziere Array c[5,6]

LCS Beispiel (1)

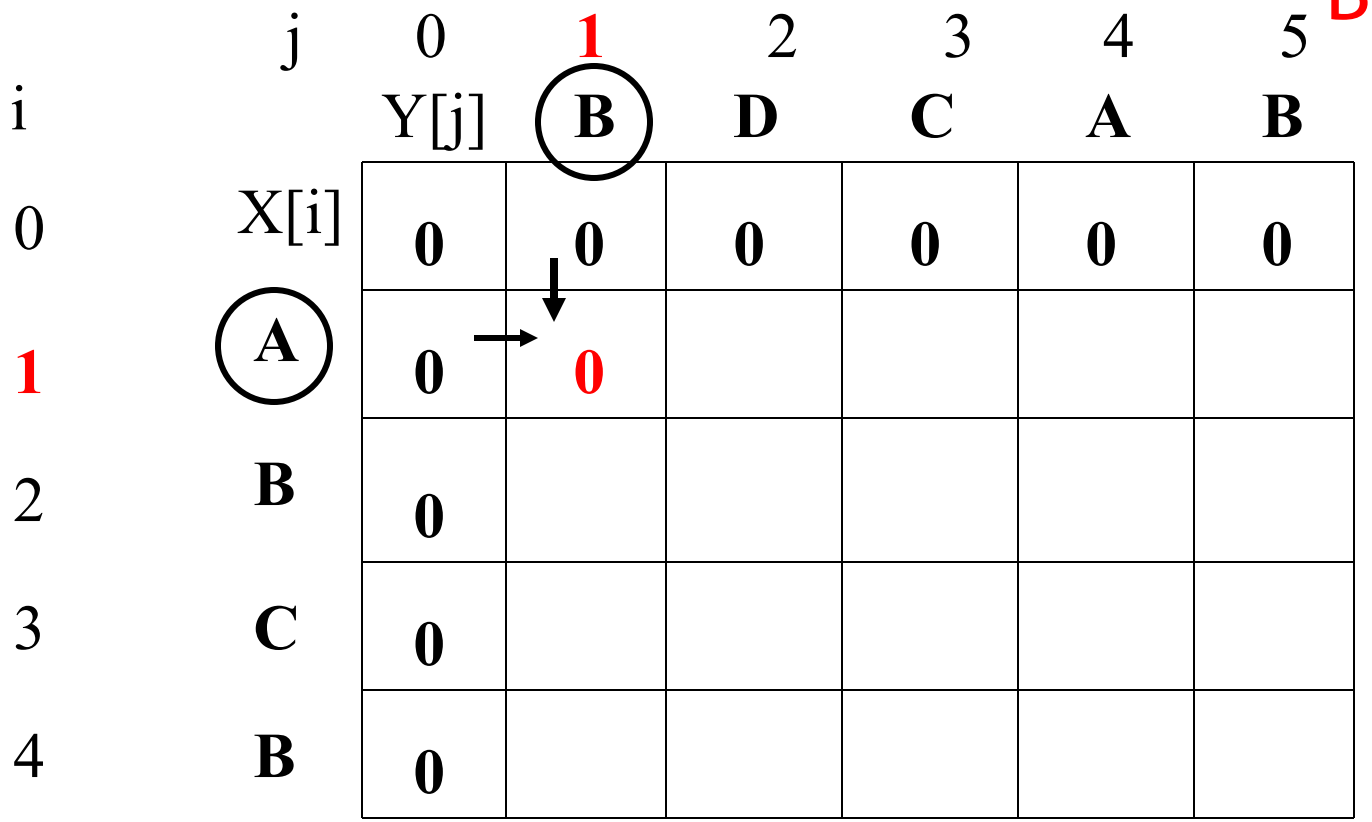
ABCB
BDCAB

i	j	0	1	2	3	4	5
		Y[j]	**B**	**D**	**C**	**A**	**B**
0	X[i]	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**					
2	**B**	**0**					
3	**C**	**0**					
4	**B**	**0**					

```
for i = 0:m   c[i, 0] = 0 end
for j = 0:n   c[0, j] = 0 end
```

LCS Beispiel (2)

ABCB
BDCAB




```
if X[i] == Y[j]
  c[i, j] = c[i-1, j-1] + 1
else
  c[i, j] = maximum([c[i-1, j], c[i, j-1]])
```

LCS Beispiel (3)

ABCB

BDCAB

i	j	0	1	2	3	4	5
		Y[j]	**B**	**D**	**C**	**A**	**B**
0	X[i]	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**		
2	**B**	**0**					
3	**C**	**0**					
4	**B**	**0**					

```
if X[i] == Y[j]
  c[i, j] = c[i-1, j-1] + 1
else
  c[i, j] = maximum([c[i-1, j], c[i, j-1]])
```

LCS Beispiel (4)

ABCB

BDCAB

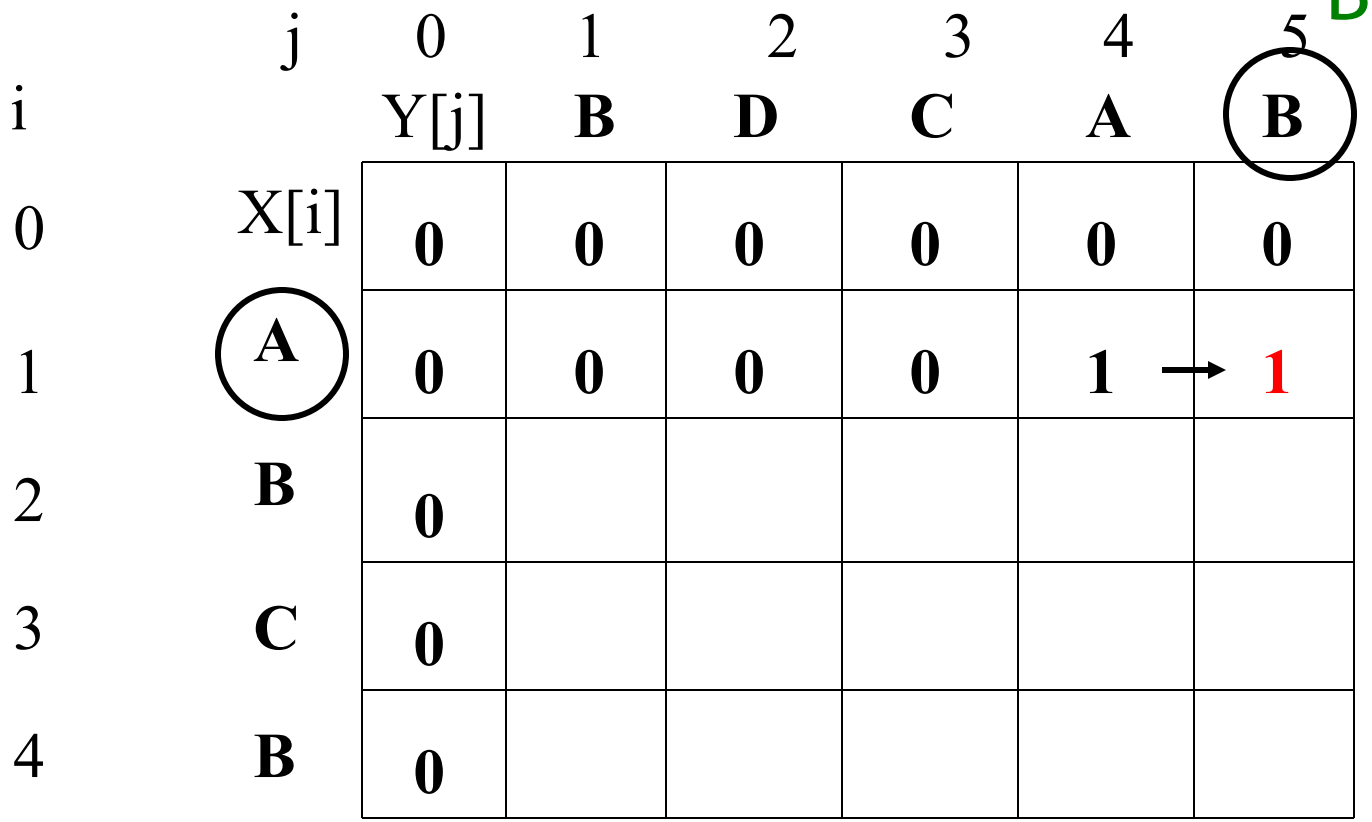
i	j	0	1	2	3	4	5
		Y[j]	B	D	C	A	B
0	X[i]	0	0	0	0	0	0
1	A	0	0	0	0	1	
2	B	0					
3	C	0					
4	B	0					

```
if X[i] == Y[j]
  c[i, j] = c[i-1, j-1] + 1
else
  c[i, j] = maximum([c[i-1, j], c[i, j-1]])
```

LCS Beispiel (5)

ABCB

BDCAB

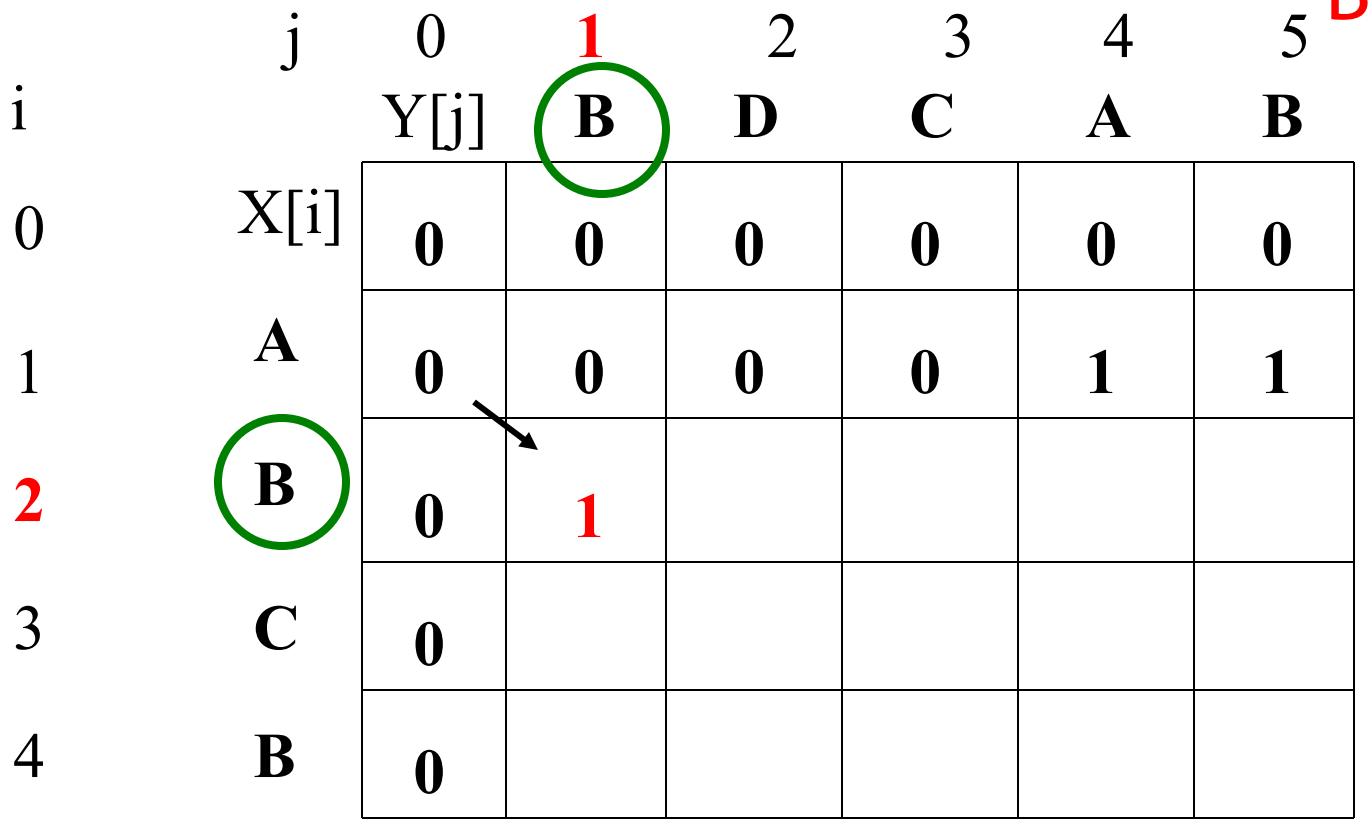


i		j	0	1	2	3	4	5
			Y[j]	B	D	C	A	B
0	X[i]		0	0	0	0	0	0
1	A		0	0	0	0	1	1
2	B		0					
3	C		0					
4	B		0					

```
if X[i] == Y[j]
  c[i, j] = c[i-1, j-1] + 1
else
  c[i, j] = maximum([c[i-1, j], c[i, j-1]])
```

LCS Beispiel (6)

ABCB
BDCAB

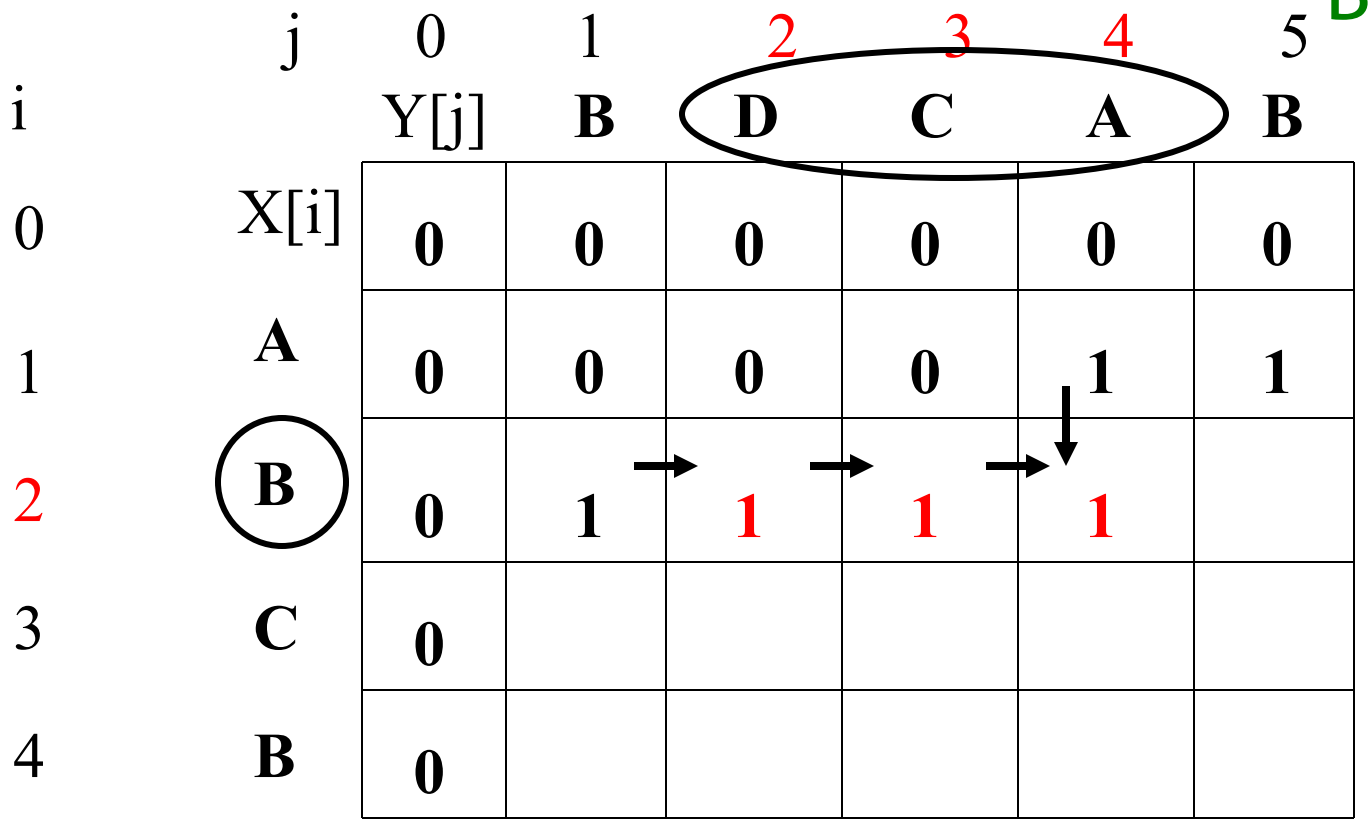


i		j	0	1	2	3	4	5
			Y[j]	B	D	C	A	B
0	X[i]		0	0	0	0	0	0
1	A		0	0	0	0	1	1
2	B		0	1				
3	C		0					
4	B		0					

```
if X[i] == Y[j]
  c[i, j] = c[i-1, j-1] + 1
else
  c[i, j] = maximum([c[i-1, j], c[i, j-1]])
```

LCS Beispiel (7)

ABCB
BDCAB




```
if X[i] == Y[j]
  c[i, j] = c[i-1, j-1] + 1
else
  c[i, j] = maximum([c[i-1, j], c[i, j-1]])
```

LCS Beispiel (8)

ABCB

BDCAB

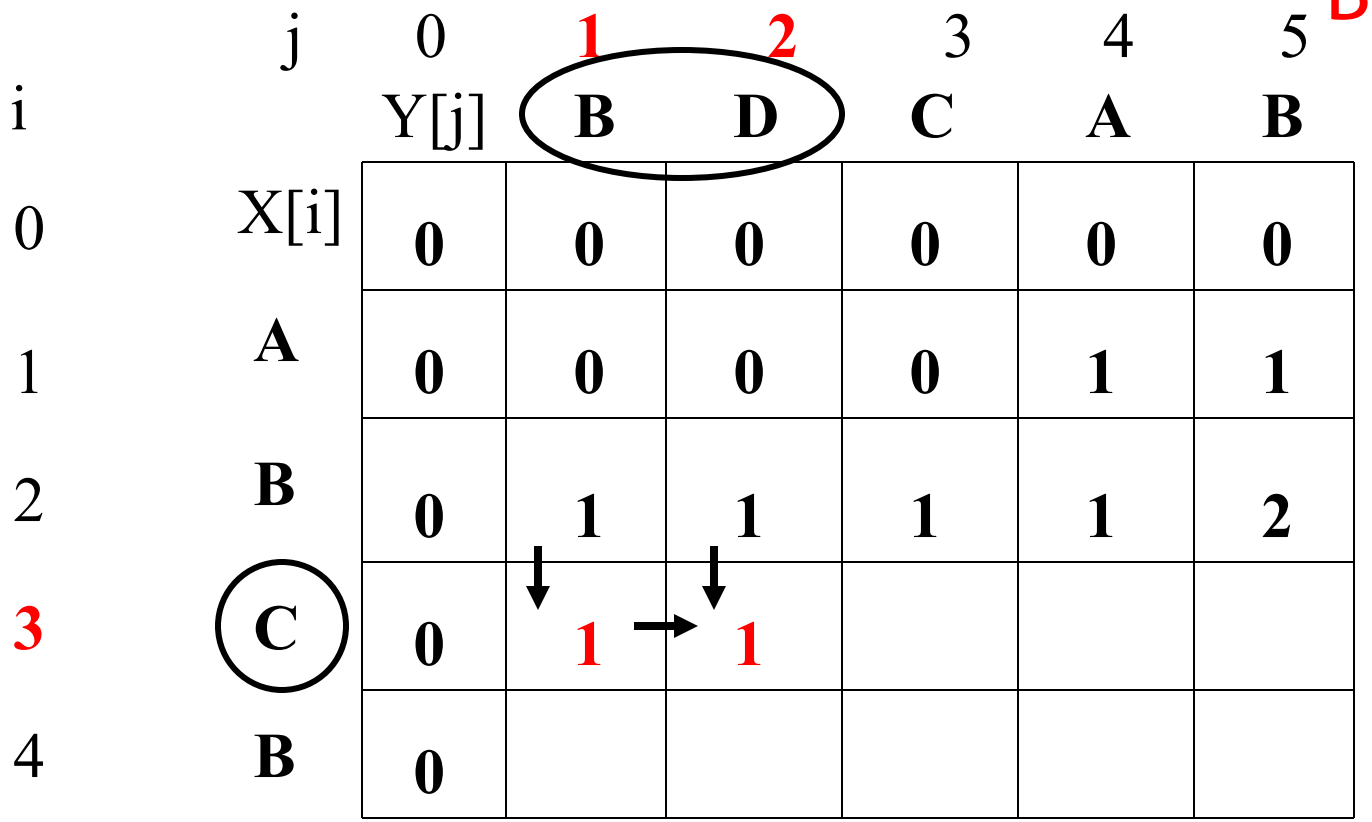
i		j	0	1	2	3	4	5
			Y[j]	B	D	C	A	B
0	X[i]		0	0	0	0	0	0
1	A		0	0	0	0	1	1
2	B		0	1	1	1	1	2
3	C		0					
4	B		0					

```
if X[i] == Y[j]
  c[i, j] = c[i-1, j-1] + 1
else
  c[i, j] = maximum([c[i-1, j], c[i, j-1]])
```

LCS Beispiel (9)

ABCB

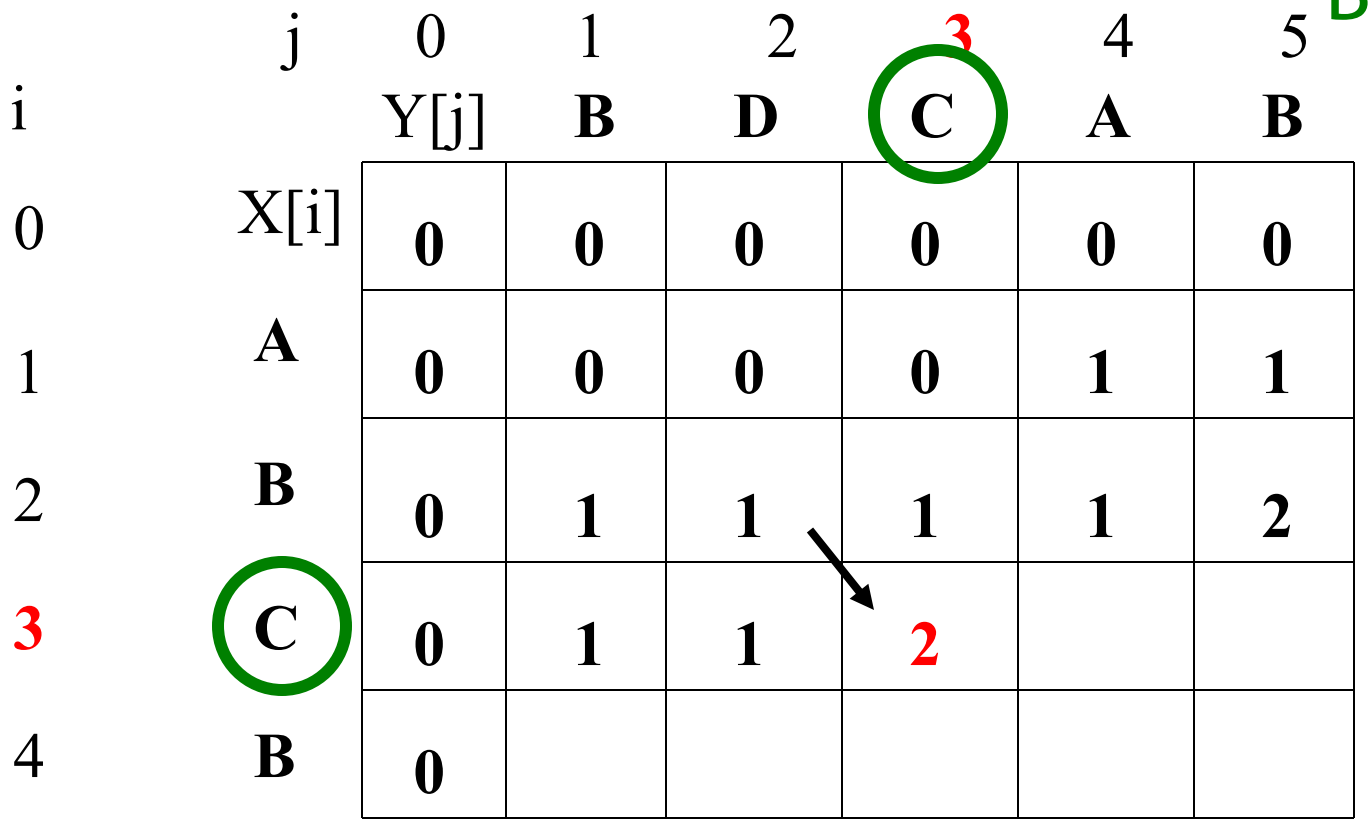
BDCAB




```
if X[i] == Y[j]
  c[i, j] = c[i-1, j-1] + 1
else
  c[i, j] = maximum([c[i-1, j], c[i, j-1]])
```

LCS Beispiel (10)

ABCB
BDCAB



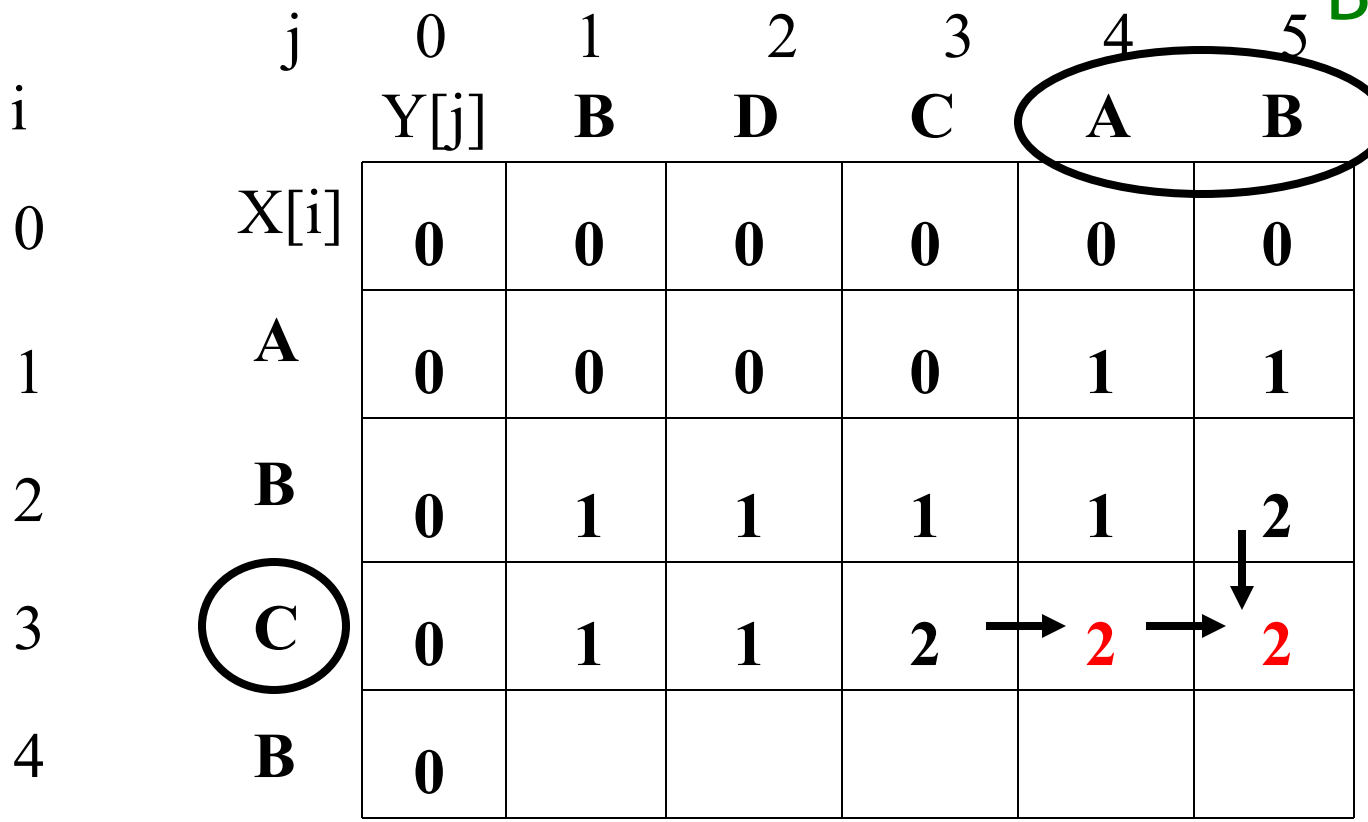
i		j	0	1	2	3	4	5
			Y[j]	B	D	C	A	B
0	X[i]		0	0	0	0	0	0
1	A		0	0	0	0	1	1
2	B		0	1	1	1	1	2
3	C		0	1	1	2		
4	B		0					

```
if X[i] == Y[j]
  c[i, j] = c[i-1, j-1] + 1
else
  c[i, j] = maximum([c[i-1, j], c[i, j-1]])
```

LCS Beispiel (11)

ABCB

BDCAB




```
if X[i] == Y[j]
  c[i, j] = c[i-1, j-1] + 1
else
  c[i, j] = maximum([c[i-1, j], c[i, j-1]])
```

LCS Beispiel (12)

ABCB

BDCAB

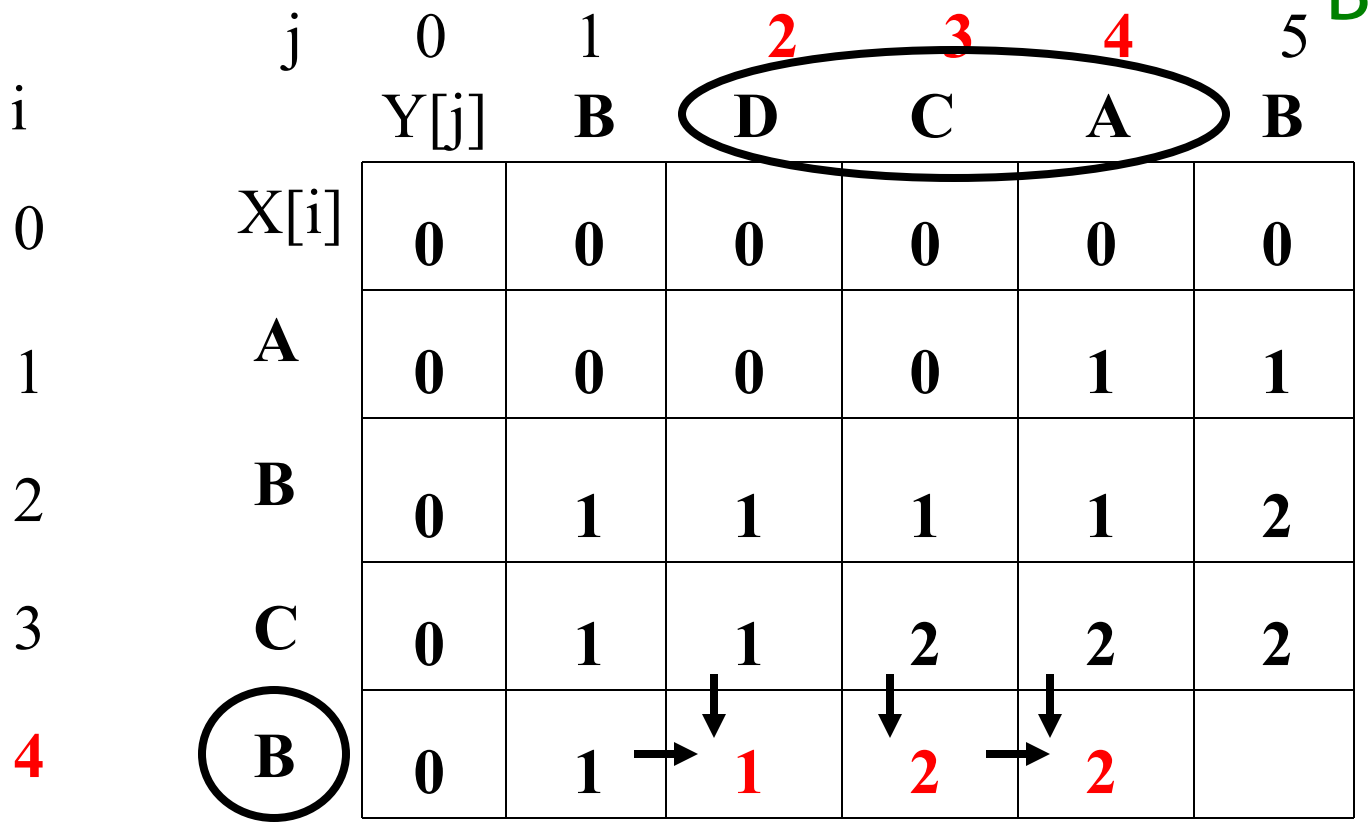
i	j	0	1	2	3	4	5
		Y[j]	B	D	C	A	B
0	X[i]	0	0	0	0	0	0
1	A	0	0	0	0	1	1
2	B	0	1	1	1	1	2
3	C	0	1	1	2	2	2
4	B	0	1				

```
if X[i] == Y[j]
  c[i, j] = c[i-1, j-1] + 1
else
  c[i, j] = maximum([c[i-1, j], c[i, j-1]])
```

LCS Beispiel (13)

ABCB

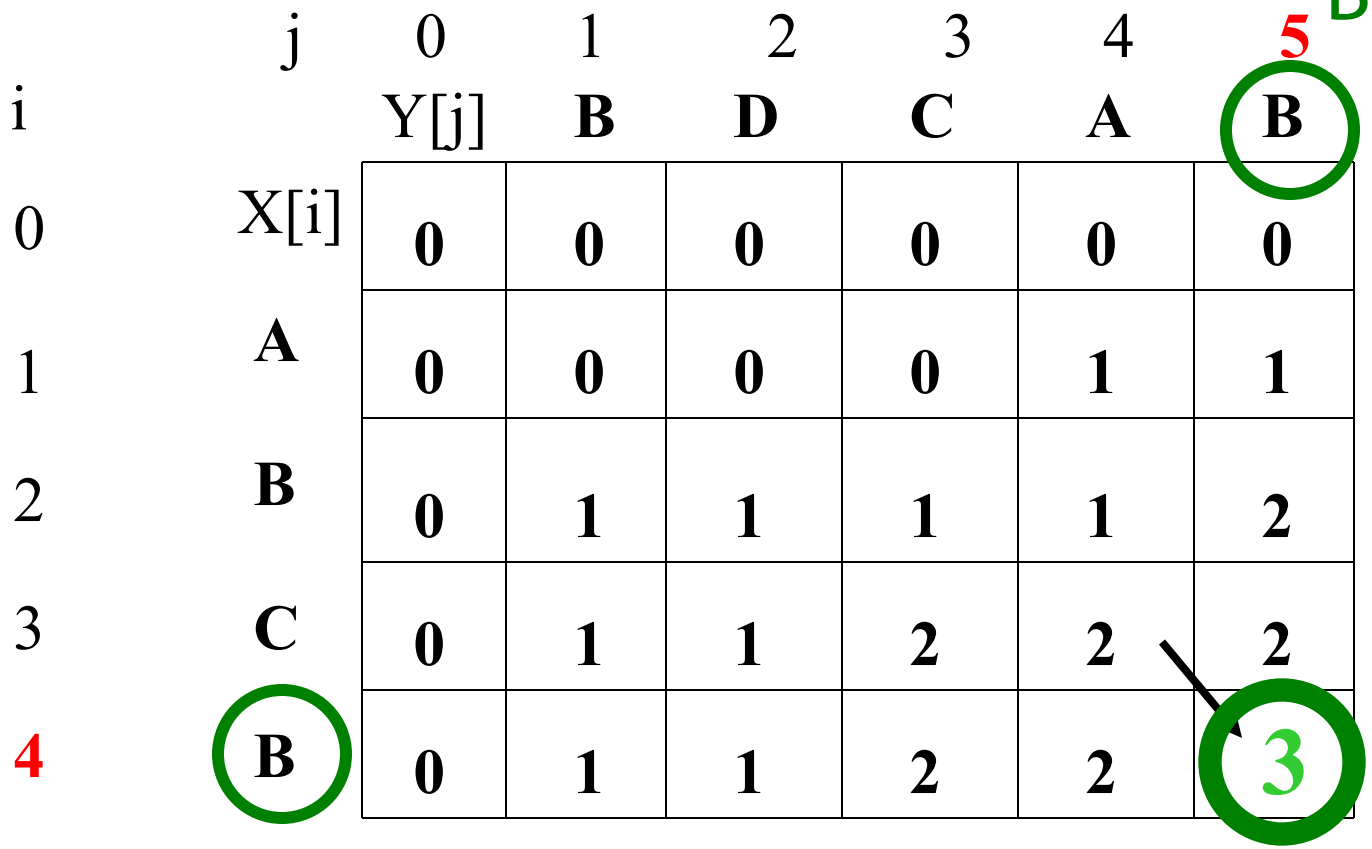
BDCAB




```
if X[i] == Y[j]
   c[i, j] = c[i-1, j-1] + 1
else
   c[i, j] = maximum([c[i-1, j], c[i, j-1]])
```

LCS Beispiel (14)

ABCB
BDCAB



i	j	0	1	2	3	4	5
		Y[j]	B	D	C	A	B
0	X[i]	0	0	0	0	0	0
1	A	0	0	0	0	1	1
2	B	0	1	1	1	1	2
3	C	0	1	1	2	2	2
4	B	0	1	1	2	2	3

```
if X[i] == Y[j]
  c[i, j] = c[i-1, j-1] + 1
else
  c[i, j] = maximum([c[i-1, j], c[i, j-1]])
```

LCS-Algorithmus: Analyse

- Der LCS-Algorithmus bestimmt die Werte des Feldes c[m,n]
- Laufzeit?

O(m·n)

Jedes c[i,j] wird in konstanter Zeit berechnet, und es gibt m·n Elemente in dem Feld

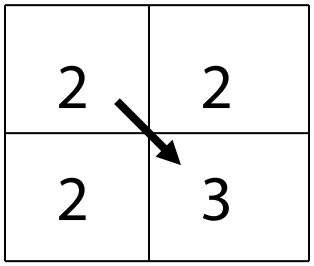
Wie findet man den tatsächlichen LCS?

- Für c[i, j] ist bekannt wie es hergeleitet wurde:

$$c[i,j] = \begin{cases} c[i-1,j-1]+1 & \text{falls } x[i] = y[j], \\ \max(c[i,j-1], c[i-1,j]) & \text{sonst} \end{cases}$$

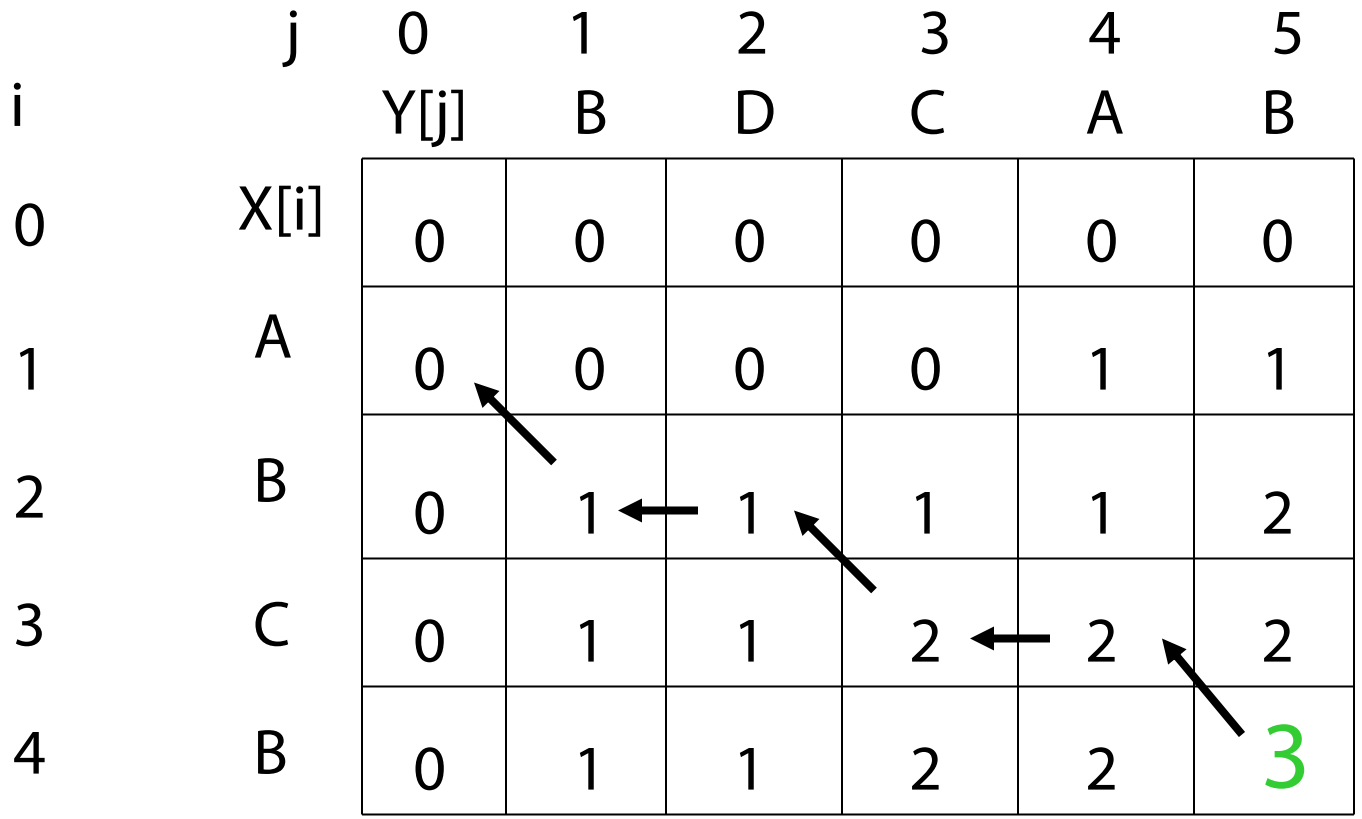
- Match liegt nur vor, wenn erste Gleichung verwendet
- Beginnend von c[m,n] und rückwärtslaufend, speichere x[i] wenn c[i,j] = c[i-1, j-1]+1.

Beispielimplementierung in Julia vorhanden.



Zum Beispiel hier
c[i,j] = c[i-1,j-1] +1 = 2+1=3

Finde LCS Zeit für Rückverfolgung: O(m+n)



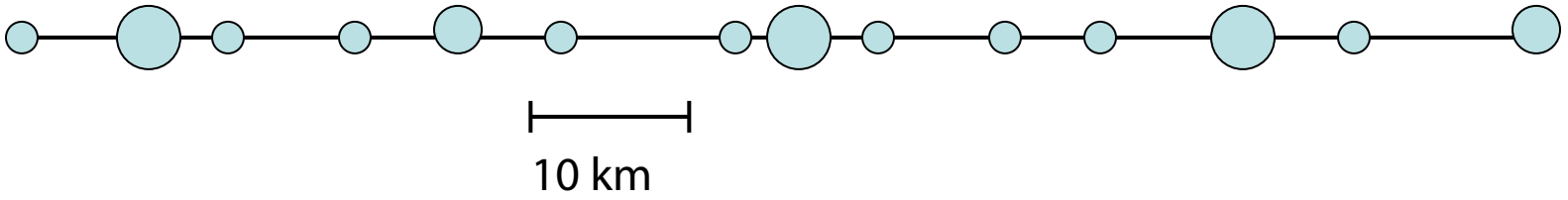
LCS (umgekehrt): B C B

LCS (richtig dargestellt): **B C B**

(ein Palindrom)

Dynamische Programmierung: Restaurant-Platzierung

- Städte $t_1, t_2, \ldots, t_n$ an der Autobahn
- Restaurants in t_i haben von der Größe der Stadt abhängigen geschätzten jährlichen Profit p_i
- Restaurants mit Mindestabstand von 10 km aufgrund von Vorgaben
- Ziel: Maximierung des Profits – großer Bonus

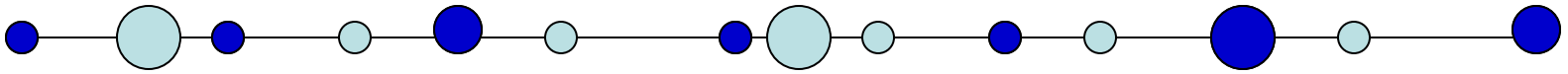


Brute-Force-Ansatz

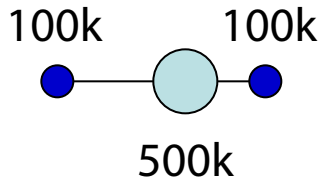
- Jede Stadt wird entweder gewählt oder nicht
- Testen der Bedingungen für 2^n Teilmengen
- Eliminiere Teilmengen, die Einschränkungen nicht erfüllen
- Berechne Gesamtprofit für jede übrigbleibende Teilmenge
- Wähle Teilmenge von Städten mit größtem Profit

- $\Theta(n \cdot 2^n)$

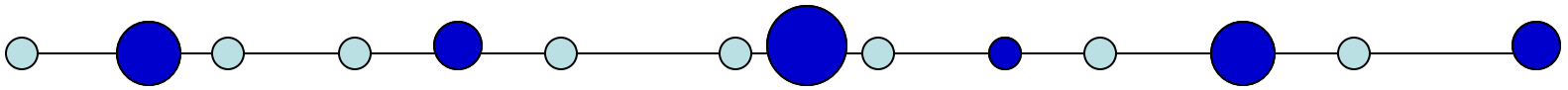
Natürlich-gierige Strategie 1



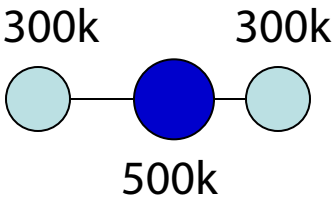
- Nehme erste Stadt. Dann nächste Stadt mit Entfernung >= 10 km
- Können Sie ein Beispiel angeben, bei dem nicht die richtige (beste) Lösung bestimmt wird?



Natürlich-gierige Strategie 2

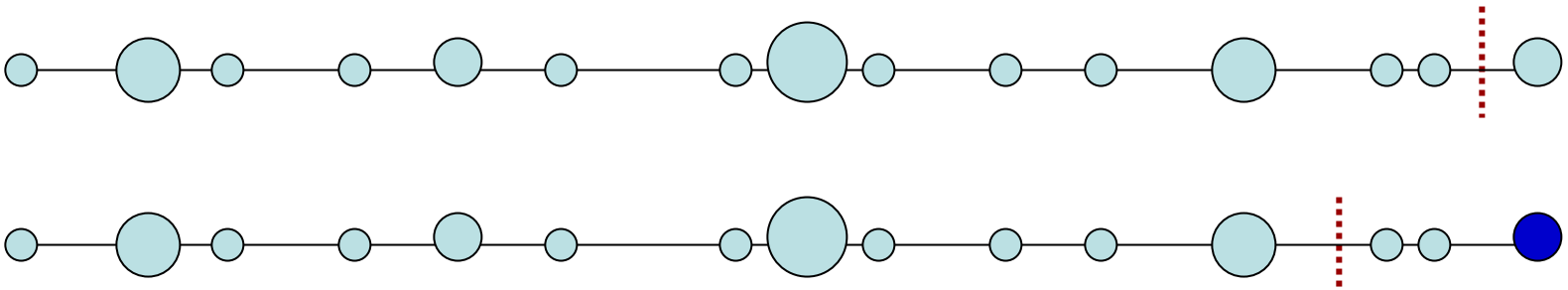


- Nehme Stadt mit höchstem Profit und dann die nächsten, die nicht <10 km von der vorher gewählten Stadt liegen
- Können Sie ein Beispiel angeben, bei dem nicht die richtige (beste) Lösung bestimmt wird?



Formulierung über dynamische Programmierung

- Nehmen wir an, die optimale Lösung sei gefunden
- Entweder enthält sie t_n oder nicht
- Fall 1: t_n nicht enthalten
  - Beste Lösung identisch zur besten Lösung von $t_1, \ldots, t_{n-1}$
- Fall 2: t_n enthalten
  - Beste Lösung ist p_n + beste Lösung für $t_1, \ldots, t_j$, wobei $j < n$ der größte Index ist, so dass $dist(t_j, t_n) \geq 10$



Formulierung als Rekurrenz

- Sei S(i) der Gesamtprofit der optimalen Lösung, wenn die ersten i Städte betrachtet, aber nicht notwendigerweise ausgewählt wurden
  - S(n) ist die optimale Lösung für das Gesamtproblem

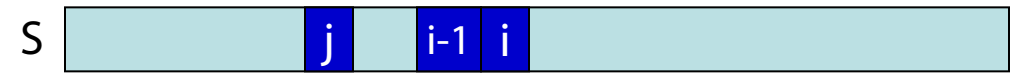
$$S(n) = \max \begin{cases} S(n-1) & \\ S(j) + p_n & j < n \;\&\; dist(t_j, t_n) \geq 10,\ j \text{ maximal} \end{cases}$$

↓ Generalisiere

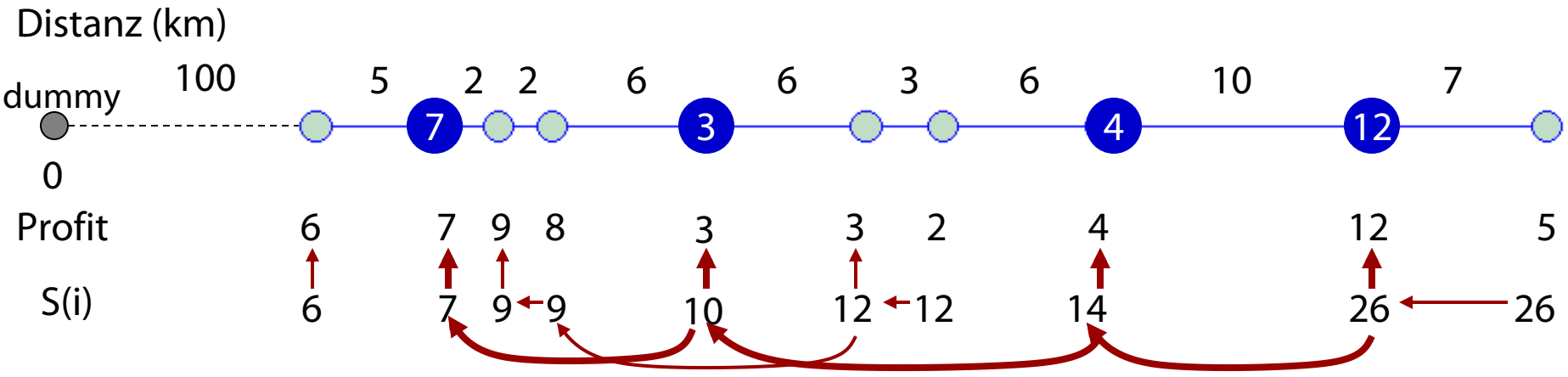
$$S(i) = \max \begin{cases} S(i-1) & \\ S(j) + p_i & j < i \;\&\; dist(t_j, t_i) \geq 10,\ j \text{ maximal} \end{cases}$$

Anzahl der Teilprobleme: n. Grenzfall: S(0) = 0.

Abhängigkeiten:



Beispiel



Optimal: 26

$$S(i) = \max \begin{cases} S(i-1) \\ S(j) + p_i \end{cases} \qquad j < i \;\&\; \text{dist}(t_j, t_i) \geq 10$$

- Natürlich-gierig 1: 6 + 3 + 4 + 12 = 25
- Natürlich-gierig 2: 12 + 9 + 4 = 25

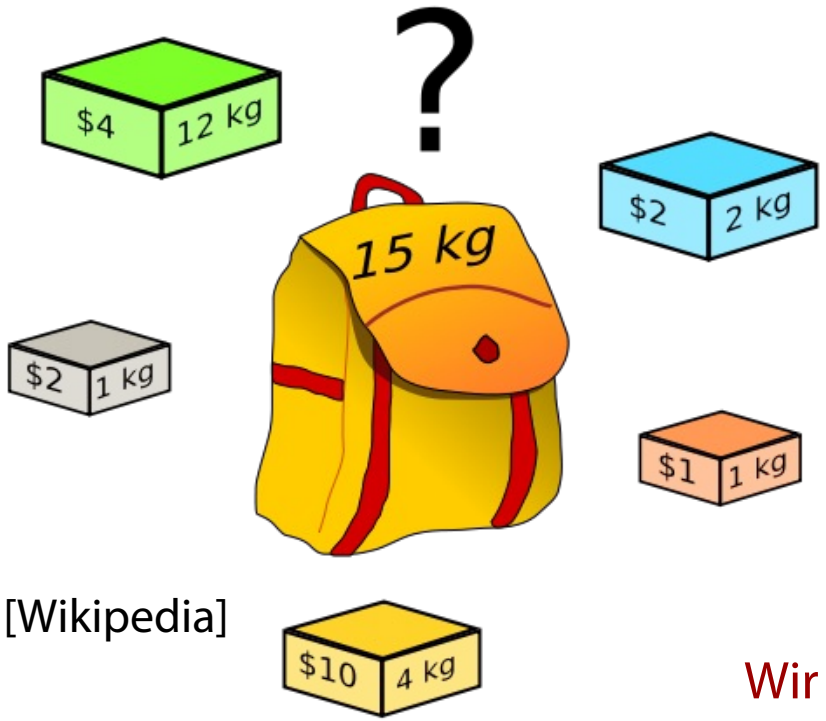
Aufwandsanalyse

- Zeit: $\Theta(nk)$, wobei k die maximale Anzahl der Städte innerhalb von 10km nach links zu jeder Stadt ist
  - Im schlimmsten Fall $\Theta(n^2)$
  - Kann durch Vorverarbeitung verbessert werden zu $\Theta(n)$
- Speicher: $\Theta(n)$

Rucksackproblem

Gegeben sei eine Menge von Gegenständen

- Jeder Gegenstand hat einen Wert und ein Gewicht
- Ziel: Maximiere Wert der Dinge im Rucksack
- Einschränkung: Rucksack hat Gewichtsgrenze



[Wikipedia]

Drei Versionen:

0-1-Rucksackproblem:
Wähle Gegenstand oder nicht

Stückelbares Rucksackproblem:
Gegenstände sind teilbar

Unbegrenztes Rucksackproblem:
Beliebige Anzahlen verfügbar

Welches ist das leichteste Problem?

Wir beginnen mit dem 0-1-Problem

0-1-Problem

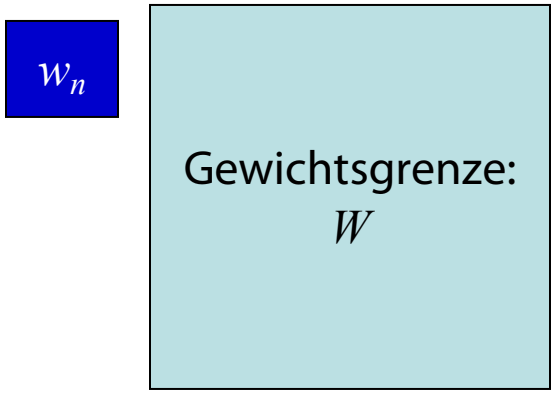
- Rucksack hat Gewichtseinschränkung W
- Gegenstände $1, 2, \ldots, n$ (beliebig)
- Gegenstände haben Gewichte $w_1, w_2, \ldots, w_n$
  - Gewichte sind Ganzzahlen und $w_i < W$
- Gegenstände haben Werte $v_1, v_2, \ldots, v_n$
- Ziel: Finde eine Teilmenge $S \subseteq \{1, 2, ..., n\}$ von Gegenständen, so dass $\sum_{i \in S} w_i \leq W$ und $\sum_{i \in S} v_i$ maximal bezüglich aller möglicher Teilmengen

Naive Algorithmen

- Betrachte alle Untermengen $S \subseteq \{1, 2, ..., n\}$
  - Optimale Lösung wird gefunden, aber exponentiell
- Gierig 1: Nimm jeweils den nächsten passenden Gegenstand mit dem größten Wert
  - Optimale Lösung wird nicht gefunden
  - Wer kann ein Beispiel geben?
- Gierig 2: Nehme jeweils nächsten Gegenstand mit dem besten Wert/Gewichts-Verhältnis
  - Optimale Lösung wird nicht gefunden
  - Wer kann ein Beispiel geben?

Ansatz mit dynamischer Programmierung

- Angenommen, die optimale Lösung S ist bekannt
- Fall 1: Gegenstand n ist im Rucksack
- Fall 2: Gegenstand n ist nicht im Rucksack



Finde optimale Lösung unter Verwendung von Gegenständen 1, 2, …, n-1 mit Gewichtsgrenze $W - w_n$

Finde optimale Lösung unter Verwendung von Gegenständen 1, 2, …, n-1 mit Gewichtsgrenze W

Rekursive Formulierung

Sei V[i, w] der optimale Gesamtwert wenn Gegenstände 1, 2, …, i betrachtet werden für aktuelle Gewichtsgrenze w

=> V[n, W] ist die optimale Lösung

Beispielimplementierung in Julia vorhanden.

$$V[n, W] = \max \begin{cases} V[n-1, W-w_n] + v_n \\ V[n-1, W] \end{cases}$$

↓ Generalisierung

$$V[i, w] = \begin{cases} \max \begin{cases} V[i-1, w-w_i] + v_i & \text{Ggst i gewählt} \\ V[i-1, w] & \text{Ggst i nicht gewählt} \end{cases} \\ V[i-1, w] \text{ if } w_i > w & \text{Ggst i nicht gewählt} \end{cases}$$

Grenzfall: V[i, 0] = 0, V[0, w] = 0. Wieviele Teilprobleme?

Beispiel

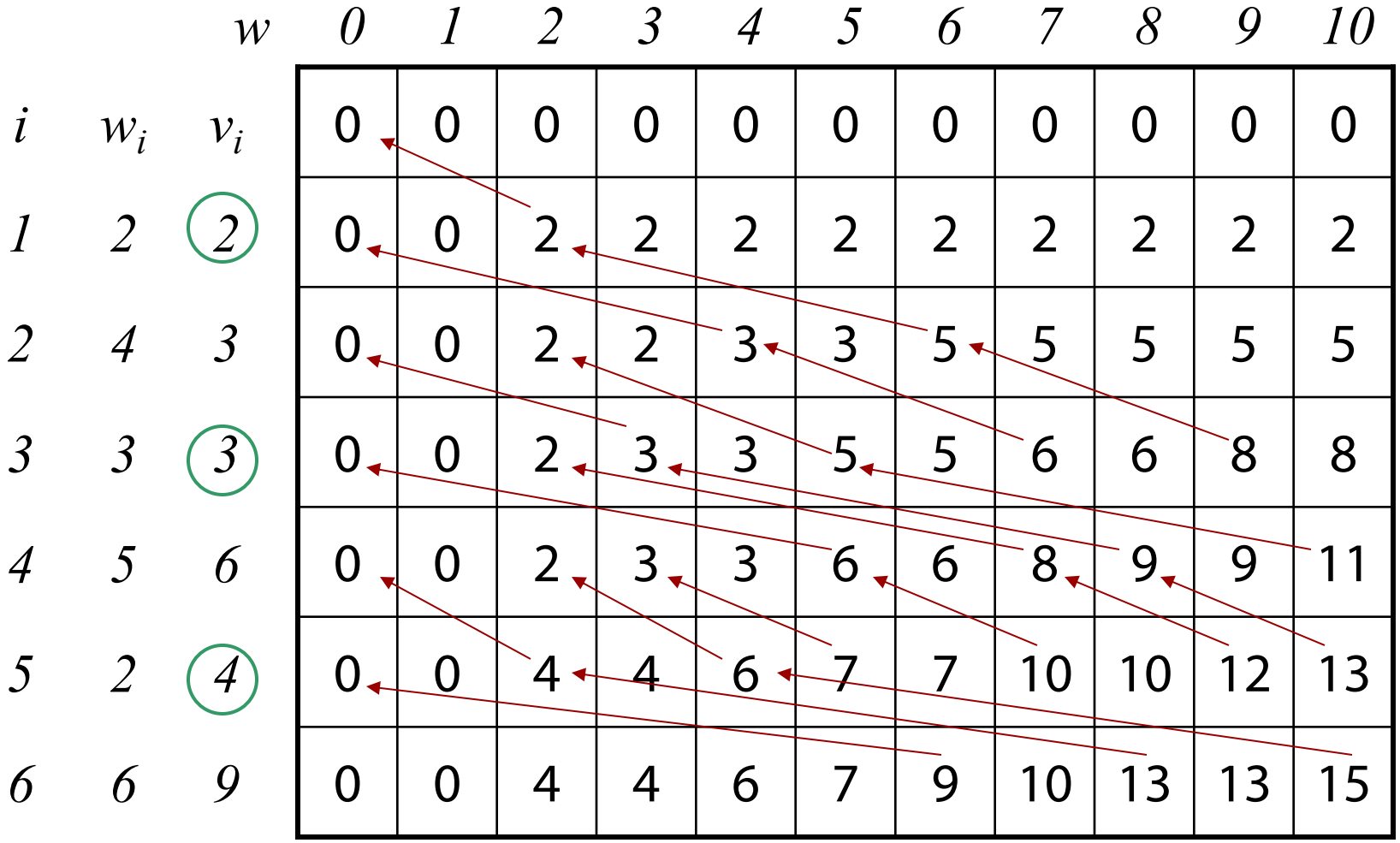
- n = 6 (# der Gegenstände)
- W = 10 (Gewichtsgrenze)
- Gegenstände (Gewicht und Wert):

	w_i	v_i
1	2	2
2	4	3
3	3	3
4	5	6
5	2	4
6	6	9

	w_i	v_i
1	2	2
2	4	3
3	3	3
4	5	6
5	2	4
6	6	9

			w	0	1	2	3	4	5	6	7	8	9	10
i	w_i	v_i												
				0	0	0	0	0	0	0	0	0	0	0
1	2	2		0										
2	4	3		0				w_i						
3	3	3		0		V[i-1,	w-w_i]				V[i-1,	w]		
4	5	6		0							V[i, w]			
5	2	4		0										
6	6	9		0										

$$V[i, w] = \begin{cases} \max \begin{cases} V[i-1, w-w_i] + v_i & \text{falls Gegenstand } i \text{ verwendet} \\ V[i-1, w] & \text{falls Gegenstand } i \text{ nicht verwendet} \end{cases} \\ V[i-1, w] \quad \text{falls } w_i > w \quad \text{Ggst. } i \text{ nicht verwendet} \end{cases}$$



$$V[i, w] = \begin{cases} \max \begin{cases} V[i-1, w-w_i] + v_i & \text{falls Gegenstand } i \text{ verwendet} \\ V[i-1, w] & \text{falls Gegenstand } i \text{ nicht verwendet} \end{cases} \\ V[i-1, w] \quad \text{falls } w_i > w \quad \text{Ggst. } i \text{ nicht verwendet} \end{cases}$$

		w	0	1	2	3	4	5	6	7	8	9	10
i	w_i	v_i											
			0	0	0	0	0	0	0	0	0	0	0
1	2	2		0	0	2	2	2	2	2	2	2	2
2	4	3		0	0	2	2	3	3	5	5	5	5
3	3	3		0	0	2	3	3	5	5	6	6	8	8
4	5	6		0	0	2	3	3	6	6	8	9	9	11
5	2	4		0	0	4	4	6	7	7	10	10	12	13
6	6	9		0	0	4	4	6	7	9	10	13	13	15

Optimaler Wert: 15

Gegenstand: 6, 5, 1

Gewichte: 6 + 2 + 2 = 10

Wert: 9 + 4 + 2 = 15

Analyse

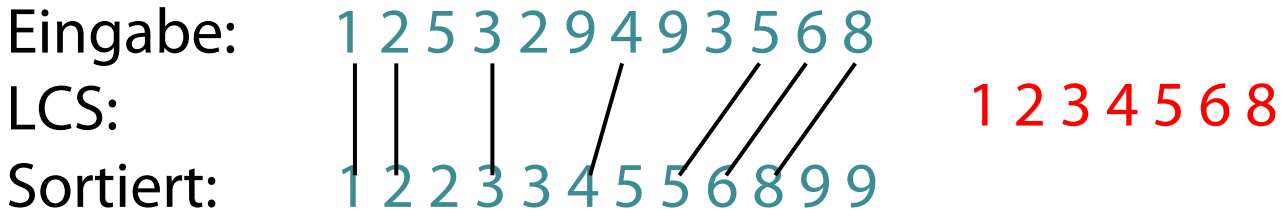
- $\Theta(nW)$
- Polynomiell?
  - Pseudo-polynomiell
    - Laufzeit hängt vom numerischen Wert W ab
    - Numerische Werte von der Länge klein (log w)
    - Verdopplung der Länge heißt dann aber exponentiell höhere Werte
  - Funktioniert trotzdem gut, wenn W klein
- Betrachte folgende Gegenstände (Gewicht, Wert):

  (10, 5), (15, 6), (20, 5), (18, 6)
- Gewichtsgrenze 35
  - Optimale Lösung: Ggst. 2, 4 (Wert = 12). Iterationen: $2^4 = 16$ Mengen
  - Dynamische Programmierung: Fülle Tabelle 4 x 35 = 140 Einträge
- Was ist das Problem?
  - Viele Einträge nicht verwendet: Gewichtskombination nicht möglich
  - Brute-Force-Ansatz kann besser sein

Longest-Increasing-Subsequence-Problem

- Gegeben sei eine Liste von Zahlen

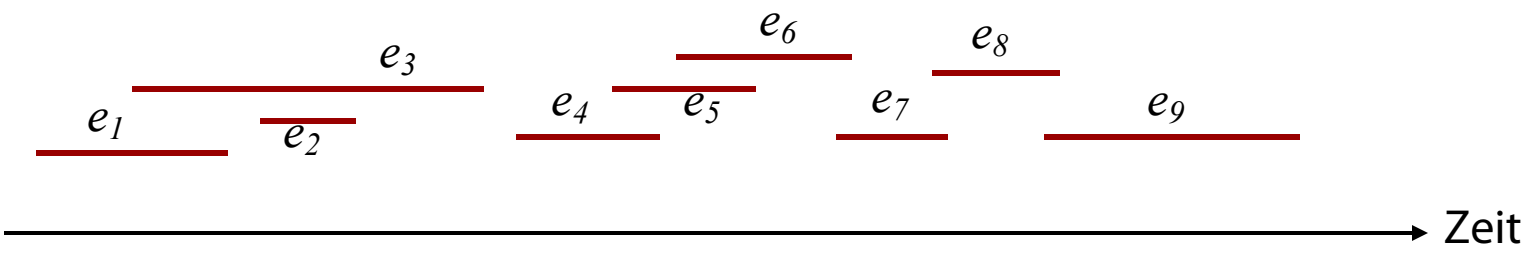
  1 2 5 3 2 9 4 9 3 5 6 8
- Finde ***längste*** Teilsequenz, in der keine Zahl kleiner ist als die vorige
  - Beispiel: 1 2 5 9
  - Teilsequenz der originalen Liste, aber nicht die längste
  - Die Lösung ist auch Teilsequenz der sortierten Liste



- Beispiel zeigt Rückführung auf bekanntes Problem mit Vorverarbeitung der Originaleingabe in O(n log n)

  bei kleinen Zahlen in O(n)

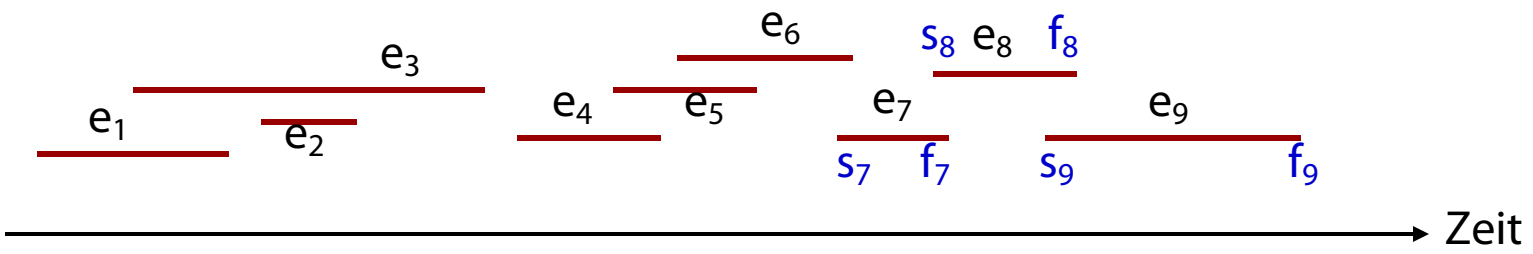
Ereignisplanungs-Problem



- Eingabe: Ein Liste von Ereignissen (Intervallen) zur Berücksichtigung
  - e_i hat Anfangszeit s_i und Endezeit f_i
  - wobei gilt, dass $f_i < f_j$ falls $i < j$
- Jedes Ereignis hat einen Wert v_i
- Gesucht: Plan, so dass Gesamtwert maximiert
  - Nur ein Ereignis kann pro Zeitpunkt stattfinden
- Vorgehen ähnlich zur Restaurant-Platzierung
  - Sortiere Ereignisse nach Endezeit
  - Betrachte ob letztes Ereignis enthalten ist oder nicht

Ereignisplanungs-Problem

Rückführung auf Optimierung durch Dynamische Programmierung



- V(i) ist der optimale Wert, der erreicht werden kann, wenn die ersten i Ereignisse betrachtet werden

- $V(n) = \max \{ \begin{array}{ll} V(n-1) & e_n \text{ nicht gewählt} \\ V(j) + v_n & e_n \text{ gewählt} \end{array}$

$j < n$, j maximal und $f_j < s_n$

Münzwechselproblem

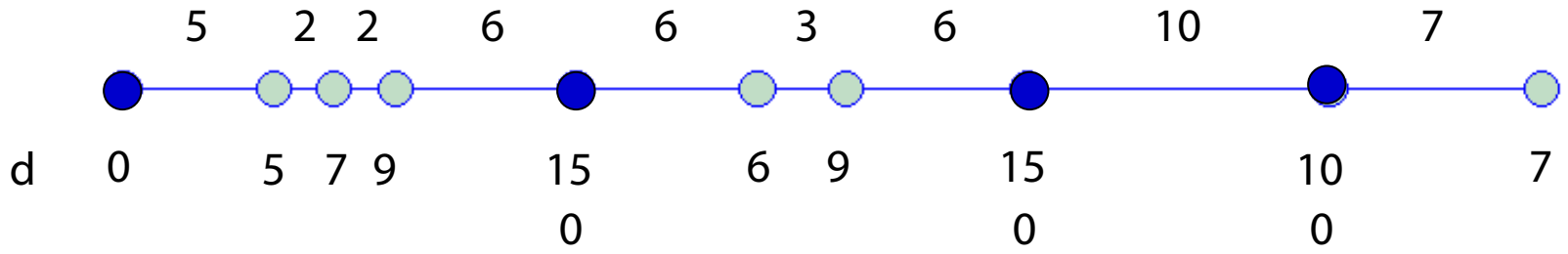
- Gegeben eine Menge von Münzwerten (z.B. 2, 5, 7, 10), entscheide, ob es möglich ist, Wechselgeld für einen gegebenen Wert (z.B. 13) zurück zu geben, oder minimiere die Anzahl der Münzen
- Version 1: Unbegrenzte Anzahl von Münzen mit entsprechenden Werten
  - Rückführung auf Unbegrenztes Rucksackproblem
- Version 2: Verwende jeden Münztyp nur einmal
  - Rückführung auf 0-1-Rucksackproblem
  - Und damit auf Dynamische Programmierung

Gierige Algorithmen

- Für einige Probleme ist dynamische Programmierung des Guten zuviel
  - Gierige Algorithmen können u.U. die optimale Lösung garantieren…
  - … und sind dabei effizienter

- Beispiel: Restaurant-Platzierung
  - Sonderfall: uniformer Profit

Gierige Restaurant-Platzierung

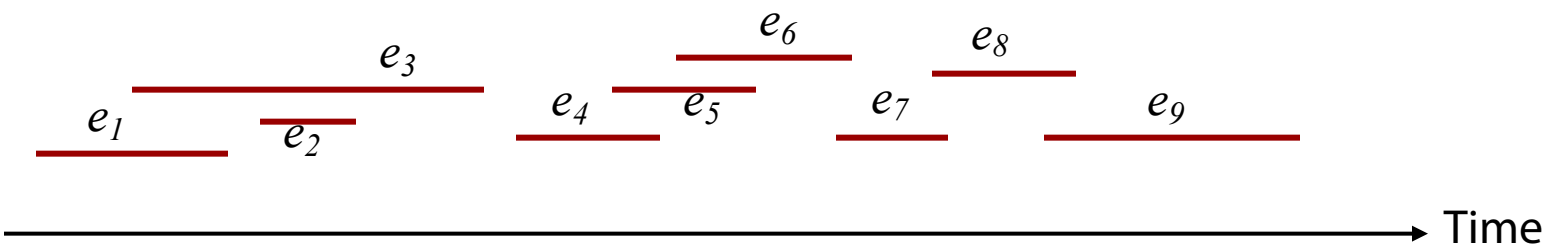



```
function greedy_locate(towns, min_dist)
  selected_towns = [towns[1]]
  n = length(towns); d = 0
  for i = 2:n
    d = d + dist(towns[i-1], towns[i])
    if d >= min_dist
      push!(selected_towns, towns[i])
      d = 0
  end end
  return selected_towns
end
```

Analyse

- Zeitaufwand: $\Theta(n)$
- Speicheraufwand:
  - $\Theta(n)$ um die Eingabe zu speichern
  - $\Theta(n)$ für die gierige Auswahl

Ereignisauswahl-Problem (Uniformer Wert)

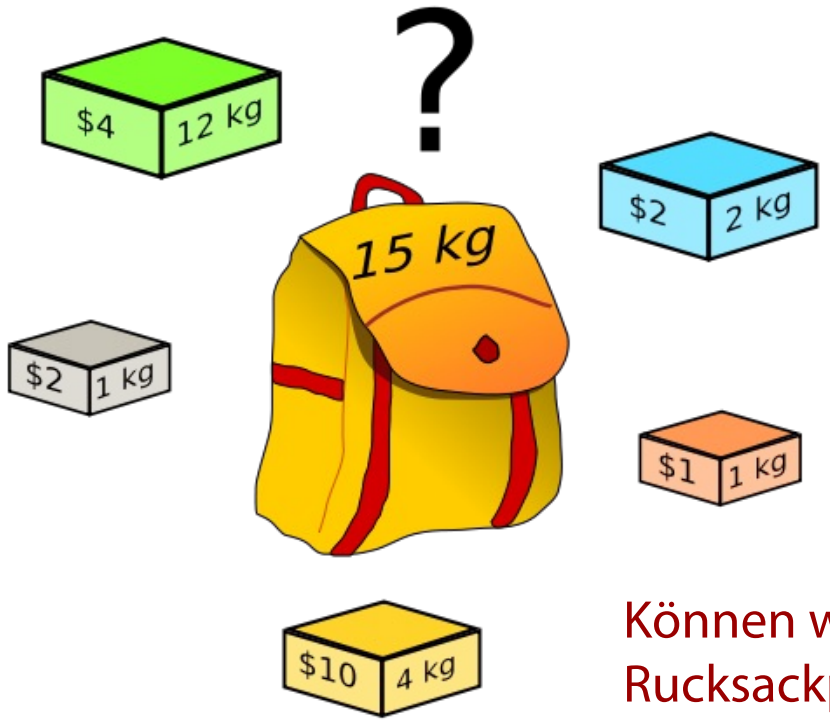


- Ziel: maximiere die **Anzahl** der ausgewählten Ereignisintervalle
- Setze $v_i = 1$ für alle i und löse Problem aufwendig mit dynamischer Programmierung
- Gierige Strategie: Wähle das nächste Ereignisintervall kompatibel mit voriger Wahl
- Liefert (e_1, e_2, e_4, e_6, e_8) für das obige Beispiel
- Warum funktioniert das hier?
  - Behauptung 1: optimale Substruktur
  - Behauptung 2: Es gibt optimale Lösung, die e_1 beinhaltet
  - Beweis durch Widerspruch: Nehme an, keine optimale Lösung enthält e_1
  - Sagen wir, das erste gewählte Ereignis ist e_i => andere gewählte Ereignisse starten, nachdem e_i endet
  - Ersetzte e_i durch e_1 ergibt eine andere optimale Lösung (e_1 endet früher als e_i)
  - Widerspruch
- Einfache Idee: Wähle das nächste Ereignis, mit dem die maximale Zeit verbleibt

Rucksackproblem

Gegeben sei eine Menge von Gegenständen

- Jeder Gegenstand hat einen Wert und ein Gewicht
- Ziel: maximiere Wert der Dinge im Rucksack
- Einschränkung: Rucksack hat Gewichtsgrenze



Drei Versionen:

0-1-Rucksackproblem:
Wähle Gegenstand oder nicht

Stückelbares Rucksackproblem:
Gegenstände sind teilbar

Unbegrenztes Rucksackproblem:
Beliebige Anzahlen verfügbar

Welches ist das leichteste Problem?

Können wir das stückelbare Rucksackproblem mit einem gierigen Verfahren lösen?

Gieriger Algorithmus für Stückelbares-Rucksackproblem

- Berechne Wert/Gewichts-Verhältnis für jeden Gegenstand
- Sortiere Gegenstände bzgl. ihres Wert/Gewichts-Verhältnisses
  - Verwende Gegenstand mit höchstem Verhältnis als wertvollstes Element (most valuable item, MVI)
- Iterativ:
  - Falls die Gewichtsgrenze nicht durch Addieren von MVI überschritten
    - Wähle MVI
  - Sonst wähle MVI partiell bis Gewichtsgrenze erreicht

Beispiel

Ggst	Gewicht (kg)	Wert ($)	$ / kg
1	2	2	1
2	4	3	0.75
3	3	3	1
4	5	6	1.2
5	2	4	2
6	6	9	1.5

- Gewichtsgrenze: 10

Beispiel: Sortierung

Ggst	Gewicht (kg)	Wert ($)	$ / kg
5	2	4	2
6	6	9	1.5
4	5	6	1.2
1	2	2	1
3	3	3	1
2	4	3	0.75

- Gewichtsgrenze: 10
- Wähle Ggst 5
  - 2 kg, $4
- Wähle Ggst 6
  - 8 kg, $13
- Wähle 2 kg von Ggst 4
  - 10 kg, 15.4

Wann funktioniert der gierige Ansatz?

1. Optimale Substruktur
2. Lokal optimale Entscheidung führt zur global optimalen Lösung

- Vergleiche auch die Bestimmung des minimalen Spannbaums

- Für die meisten Optimierungsprobleme gilt das nicht

Danksagung

Die nachfolgenden 5 Präsentationen wurden mit einigen Änderungen übernommen aus:

- „Algorithmen und Datenstrukturen“
  gehalten von Sven Groppe an der UzL

Gierige Strategie zur Wechselgeldberechnung

- **Beispiel:** Wechselgeldberechnung
- **Problem:** Finde eine Münzkombination für ein beliebiges Wechselgeld, die aus möglichst wenig Münzen besteht

- Quasi unbegrenztes Rucksackproblem durch Schaffnertasche

Schaffnertasche mit Münzmagazin

http://commons.wikimedia.org/wiki/File:Schaffnertasche_mit_galoppwechsler.jpeg

Gierige Strategie zur Wechselgeldberechnung

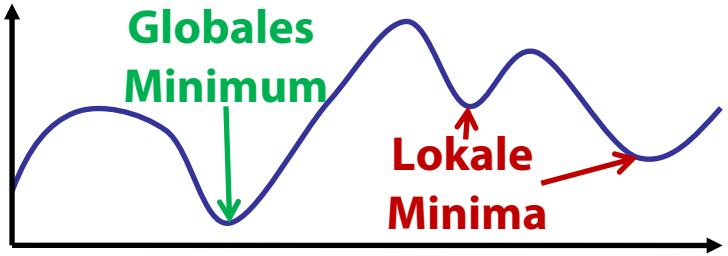
- Nimm im nächsten Schritt die größte Münze, deren Wert kleiner oder gleich dem noch verbleibenden Wechselgeld ist
- **Beispiel:**
  - Münzwerte

  - Wechselgeld sei 63 Cent:
    - 1-mal 50-Cent-Stück — Rest: 13 Cent
    - 1-mal 10-Cent-Stück — Rest: 3 Cent
    - 1-mal 2-Cent-Stück — Rest: 1 Cent
    - 1-mal 1-Cent-Stück

http://de.wikipedia.org/wiki/Eurom%C3%BCnzen

Gierige Strategie zur Wechselgeldberechnung

- Im Fall der Euro-Münzen ist die obige Strategie optimal
- Für die Münzwerte 1, 5 und 11 jedoch nicht:
  - Wechselgeld sei 15
  - Gieriges Verfahren liefert 11 1 1 1 1
  - Optimum wäre: 5 5 5
- **Damit:** Manche gierige Strategien sind anfällig dafür, ein lokales anstatt eines globalen Minimums (*bei Maximierungsproblemen: Maximums*) zu ermitteln



Gierige Strategie zur Wechselgeldberechnung

- Wie schlecht kann die Lösung der gierigen Strategie werden?
  - Münzwerte seien $n_1=1$, n_2 und n_3 mit Bedingung $n_3 = 2n_2 +1$ (vorheriges Bsp.: $n_2 = 5$, $n_3 = 11$)
  - Wechselgeld sei $N = 3n_2$ (vorheriges Bsp.: N=15)
  - **Für N sind drei Münzen optimal (3-mal n_2)**
  - **Greedy-Strategie**
    - **1-mal n_3 Rest: n_2-1**
    - **0-mal n_2 Rest: n_2-1**
    - **(n_2 -1)–mal n_1**
    - **Insgesamt n_2 Münzen**
    - **Beliebig schlecht gegenüber der optimalen Lösung (abhängig von den Münzwerten)**

Gierige Strategie zur Wechselgeldberechnung

- Es geht noch schlimmer:
  - Münzwerte seien 25 Cent, 10 Cent und 4 Cent
  - Wechselgeld sei 41 Cent
  - Lösbar ist dieses mit
    - 1-mal 25-Cent-Stück und
    - 4-mal 4-Cent-Stück
  - Greedy-Strategie
    - 1-mal 25-Cent-Stück Rest: 16 Cent
    - 1-mal 10-Cent-Stück Rest: 6 Cent
    - 1-mal 4-Cent-Stück Rest: 2 Cent
    - ??? Keine Lösung!

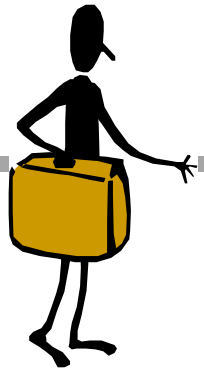
Entwurfsmuster / Entwurfsverfahren

- Im Laufe der Zeit haben sich in der immensen Vielfalt von Algorithmen nützliche Entwurfsmuster herauskristallisiert
- Gründe für das Studium von Entwurfsmustern
  - Verständnis für einzelne Algorithmen wird gesteigert, wenn man ihr jeweiliges Grundmuster verstanden hat
  - Bei der Entwicklung neuer Algorithmen kann man sich an den Grundmustern orientieren
  - Das Erkennen des zugrundeliegenden Musters hilft bei der Komplexitätsanalyse des Algorithmus
  - Warnung: Hilft **nicht** bei der Analyse der Komplexität eines **Problems**

Welches Rucksackproblem ist das leichteste?

- 0-1-Rucksackproblem: Wähle Gegenstand oder nicht
  - Schwierig
- Stückelbares Rucksackproblem:
  Gegenstände sind teilbar
  - Leicht
- Unbegrenztes Rucksackproblem:
  Beliebige Anzahlen verfügbar
  - Es kommt auf die konkrete Probleminstanz an
  - Einige Instanzen sind leicht, andere schwierig

Zusammenfassung Entwurfsmuster



- Am Anfang des Kurses behandelt:
  - Ein-Schritt-Berechnung
  - Verkleinerungsprinzip
  - Teile und Herrsche
- Jetzt haben wir dazu noch verstanden:
  - Vollständige Suchverfahren
    - Verzweigen und Begrenzen auch genannt Branch and Bound: Dijkstra, A*,...
    - Pruning (α-β-Prinzip)
  - Suche mit richtiger Problemformulierung und Subproblemanordnung
    - Dynamisches Programmieren, Bellmans Optimalitätsprinzip (Berechnung von Teilen und deren Kombination, Wiederverwendung von Zwischenergebnissen)
  - Manchmal vollständig: Gierige Suche, Optimale Substrukturen