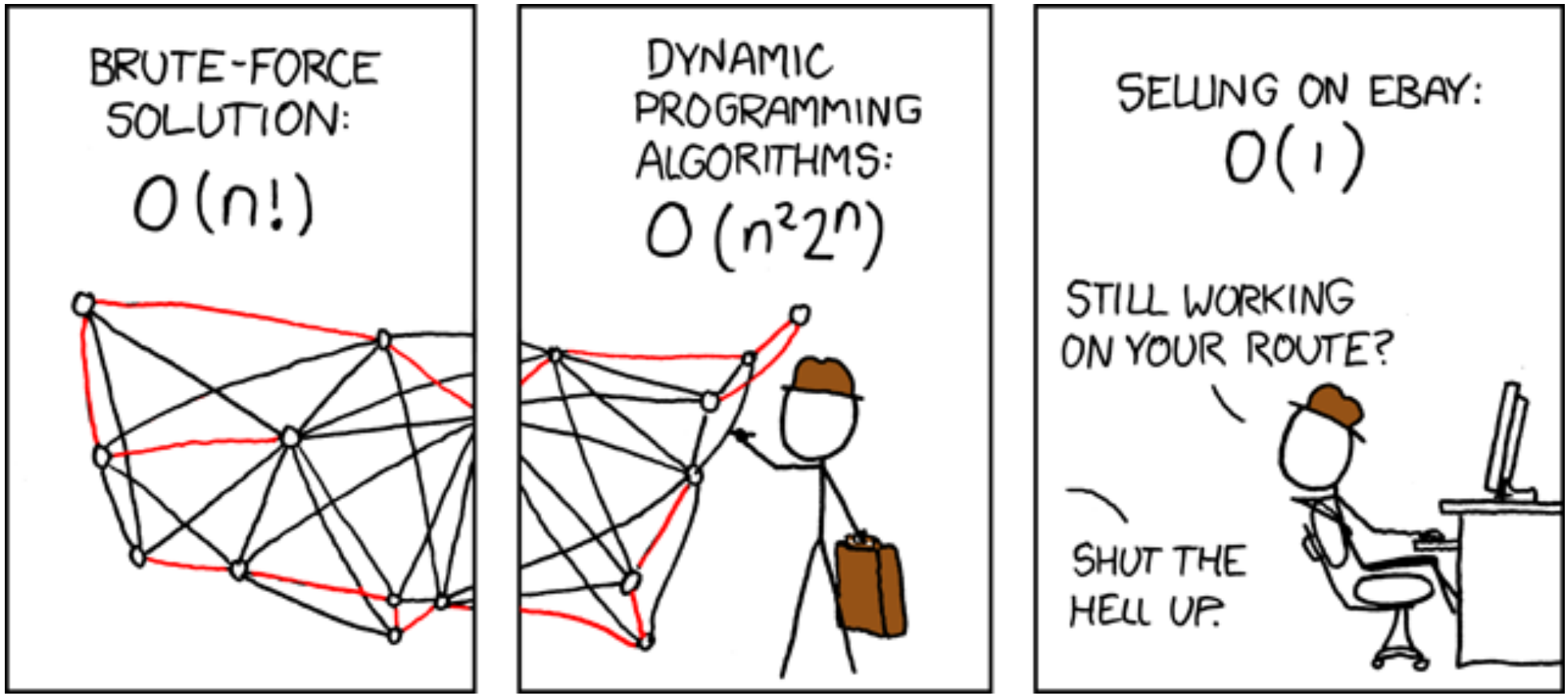
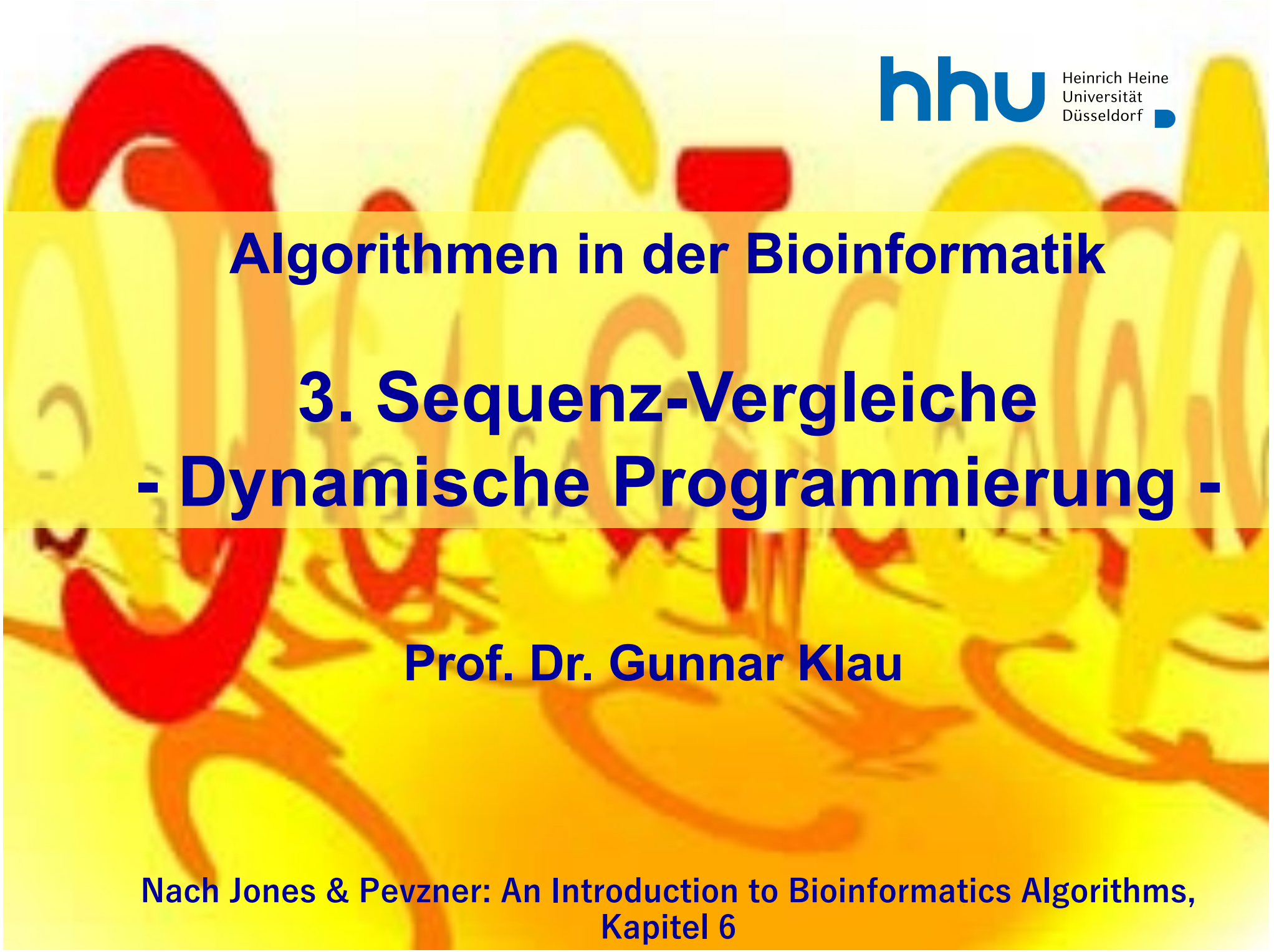

BRUTE-FORCE SOLUTION:
O(n!)
DYNAMIC PROGRAMMING ALGORITHMS:
O(n²2ⁿ)
SELLING ON EBAY:
O(1)
STILL WORKING ON YOUR ROUTE?
SHUT THE HELL UP.



Heinrich Heine Universität Düsseldorf

Algorithmen in der Bioinformatik

3. Sequenz-Vergleiche - Dynamische Programmierung -

Prof. Dr. Gunnar Klau

Nach Jones & Pevzner: An Introduction to Bioinformatics Algorithms, Kapitel 6

Biologie

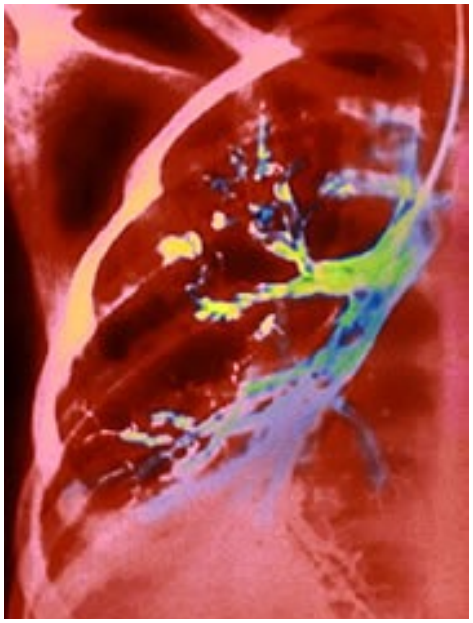
MANHATTAN
RU Campus
Queens
East River
Queensboro Bridge
FDR Drive
Yorkville
Central Park
Metropolitan Museum of Art
American Museum of Natural History
Empire State Bldg.
Bryant Park
Madison Square
Gramercy Park
Union Square
Washington Square Park
Greenwich Village
SoHo
Lower East Side
East River Park
Broadway
West Street
Lincoln Tunnel
Hudson River
NEW YORK
NEW JERSEY

DNA-Sequenz-Vergleiche

Die Suche nach Ähnlichkeit einer Sequenz mit bekannten Genen ist **die** Methode zur Funktionsbestimmung

1984 fanden Russell Doolittle & Co. Ähnlichkeit zwischen einem krebsverursachenden Gen und einem normalem Wachstumsfaktor

Zystische Fibrose/Mukoviszidose

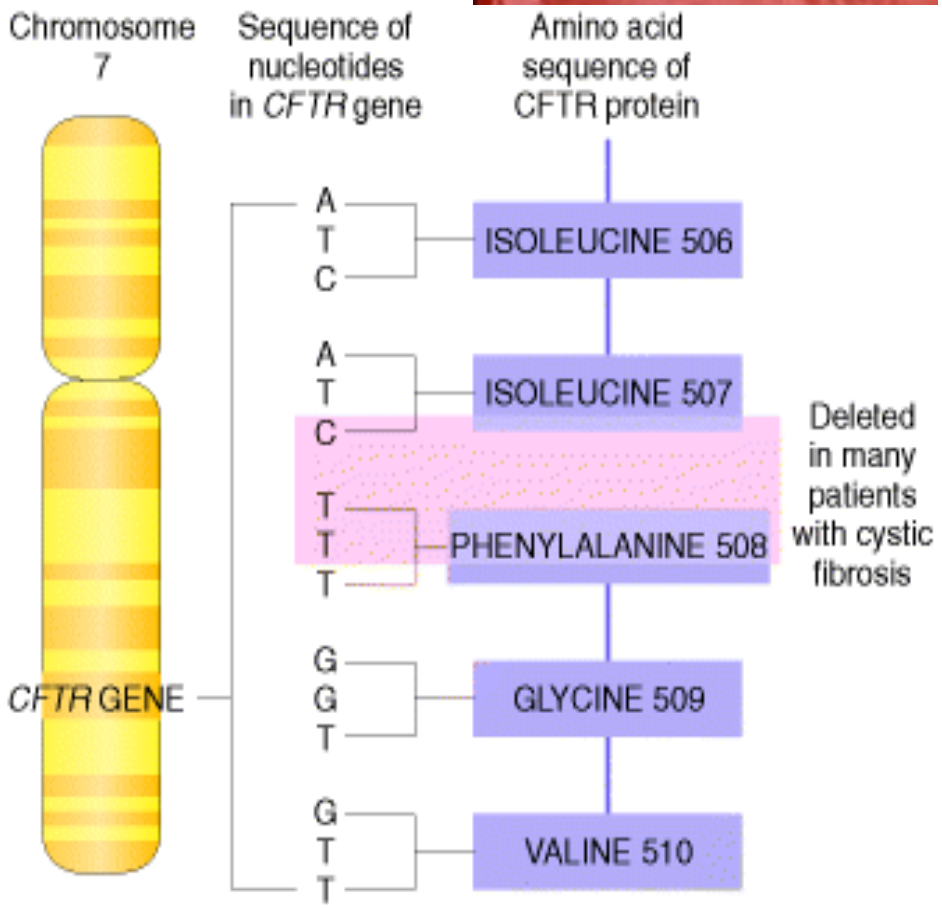


Zystische Fibrose: chronische, häufig tödliche Krankheit der Schleimdrüsen

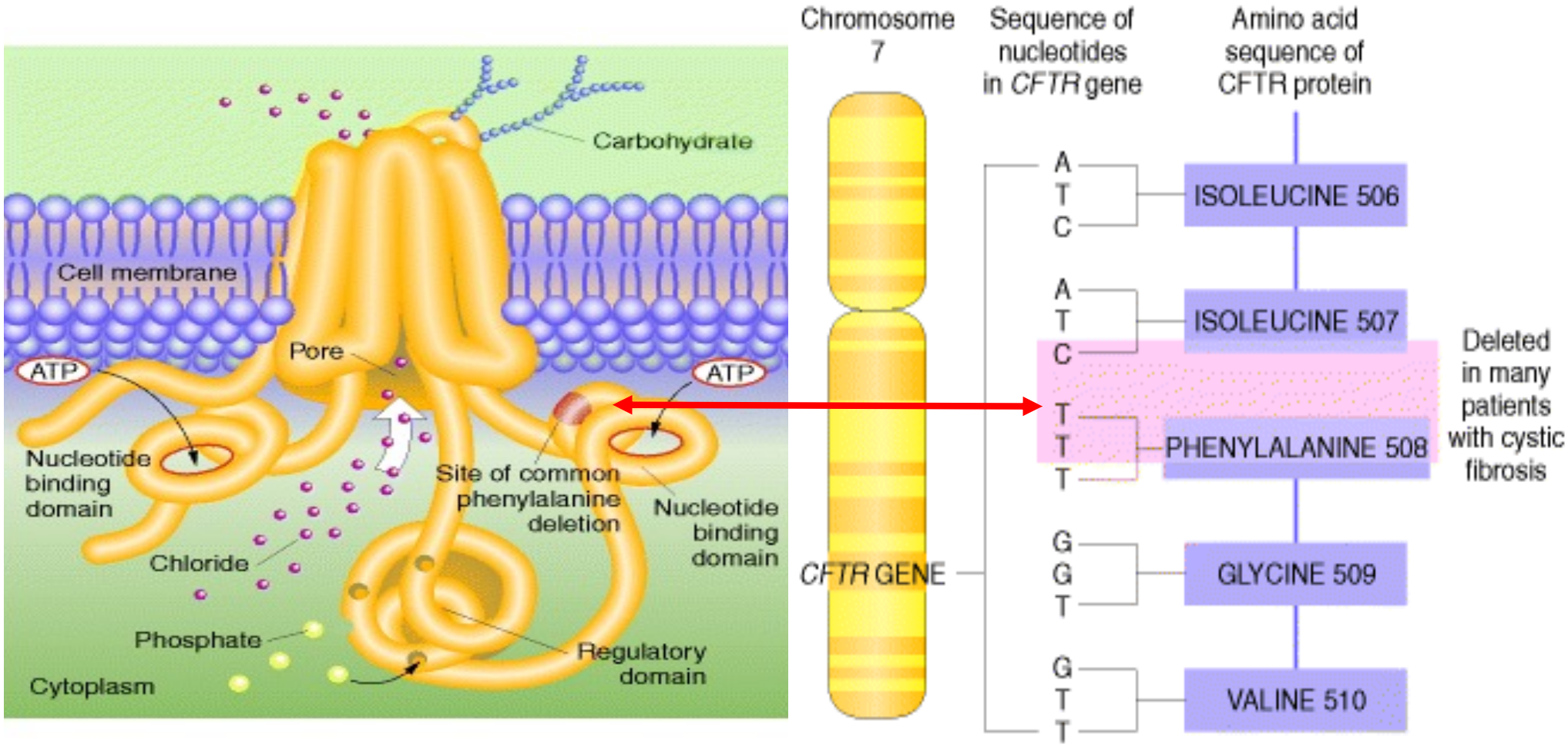
- Führt zu Verstopfung der Bronchien → Atemprobleme
- autosomal rezessiv (**++, +-,** --)
- Verantwortliches Gen unbekannt bis 1989

Bioinformatik:

- 70% der Patienten haben dieselbe Mutation des CFTR-Gens
- Ähnlichkeit mit Genen, die als Transport-Kanäle in der Zellmembran fungieren
- Funktion: Cl^--Kanal (justiert Viskosität des Sekrets)



Zystische Fibrose



Sequenzvergleich: wozu und wie?

- Ähnlichkeit eines unbekannten Gens zu einem bekannten Gen gibt Hinweise auf Funktion

- Ähnlichkeits-Score erlaubt Aussagen über die Wahrscheinlichkeit gleicher bzw. ähnlicher Funktion

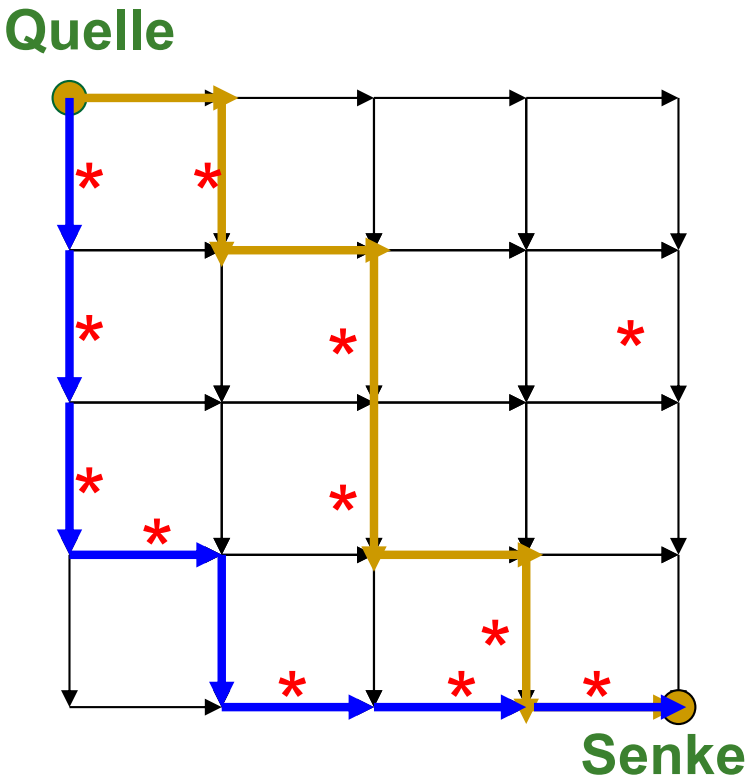
- Dynamische Programmierung ist eine Technik u. a. zum Aufdecken von Ähnlichkeiten zwischen Genen


Manhattan-Tourismus
N
RU Campus
Queens
East River
Queensboro Bridge
FDR
York
Yorkville
Lower East Side
East River Park
Grand Central
Park
Metropolitan Museum of Art
American Museum of Natural History
Central Park S.
Bryant Park
Empire State Bldg.
Madison Square Park
Greenwich Village
SoHo
Washington Square Park
Avenue of the Americas
West Street
Hudson River
Lincoln Tunnel
MANHATTAN
NEW YORK
NEW JERSEY

Tourismus in Manhattan: Problem

Finde den Weg durch Manhattan (nur ost- und südwärts!), der an den meisten Attraktionen (∗) vorbeikommt.

Wenn wir Kanten Gewichte geben (∗=1, sonst 0), dann suchen wir den ‘längsten’ Pfad.



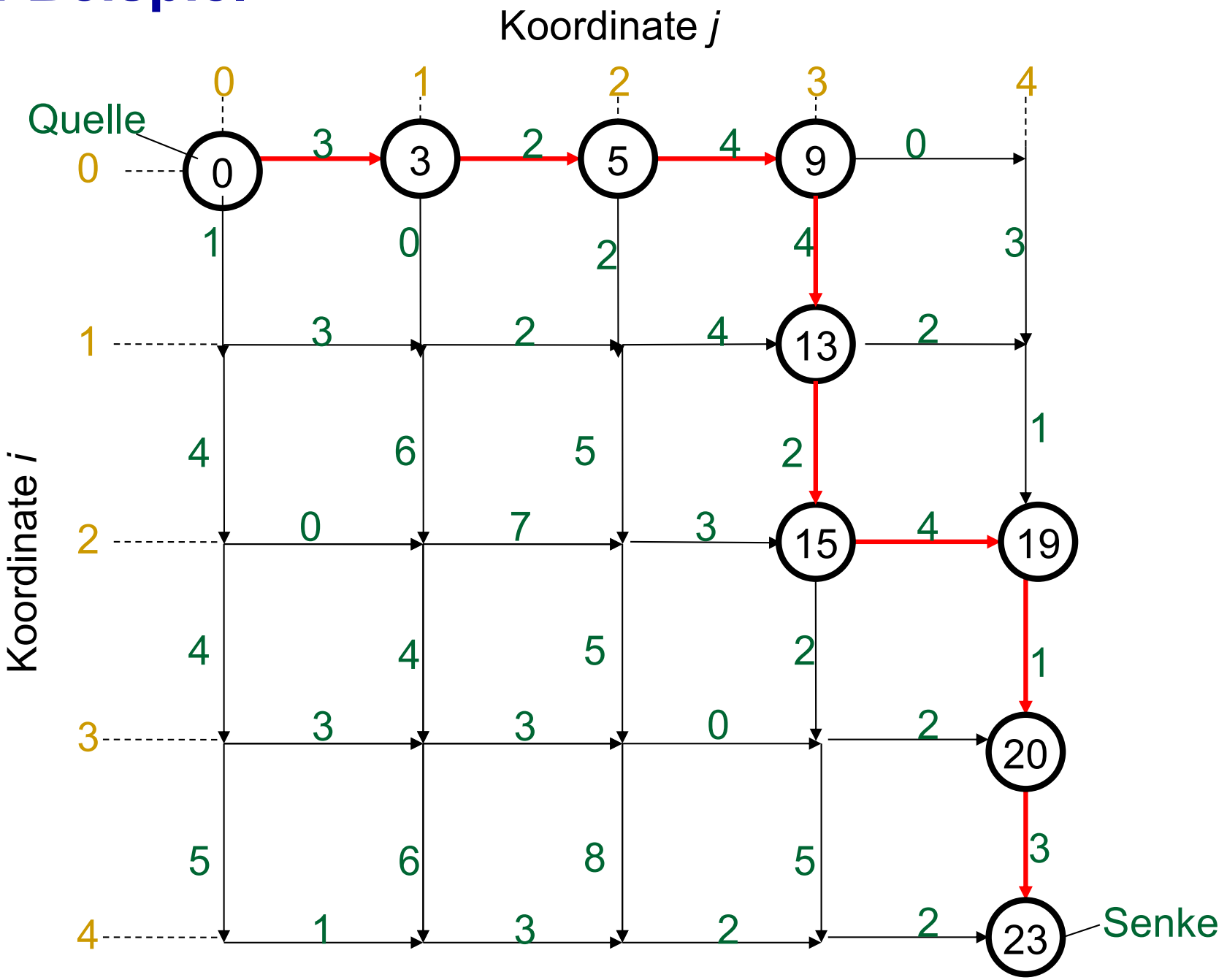
Tourismus in Manhattan: Formulierung

Ziel: Finde den längsten Pfad in einem gewichteten “Süd-Ost”-Gitter.

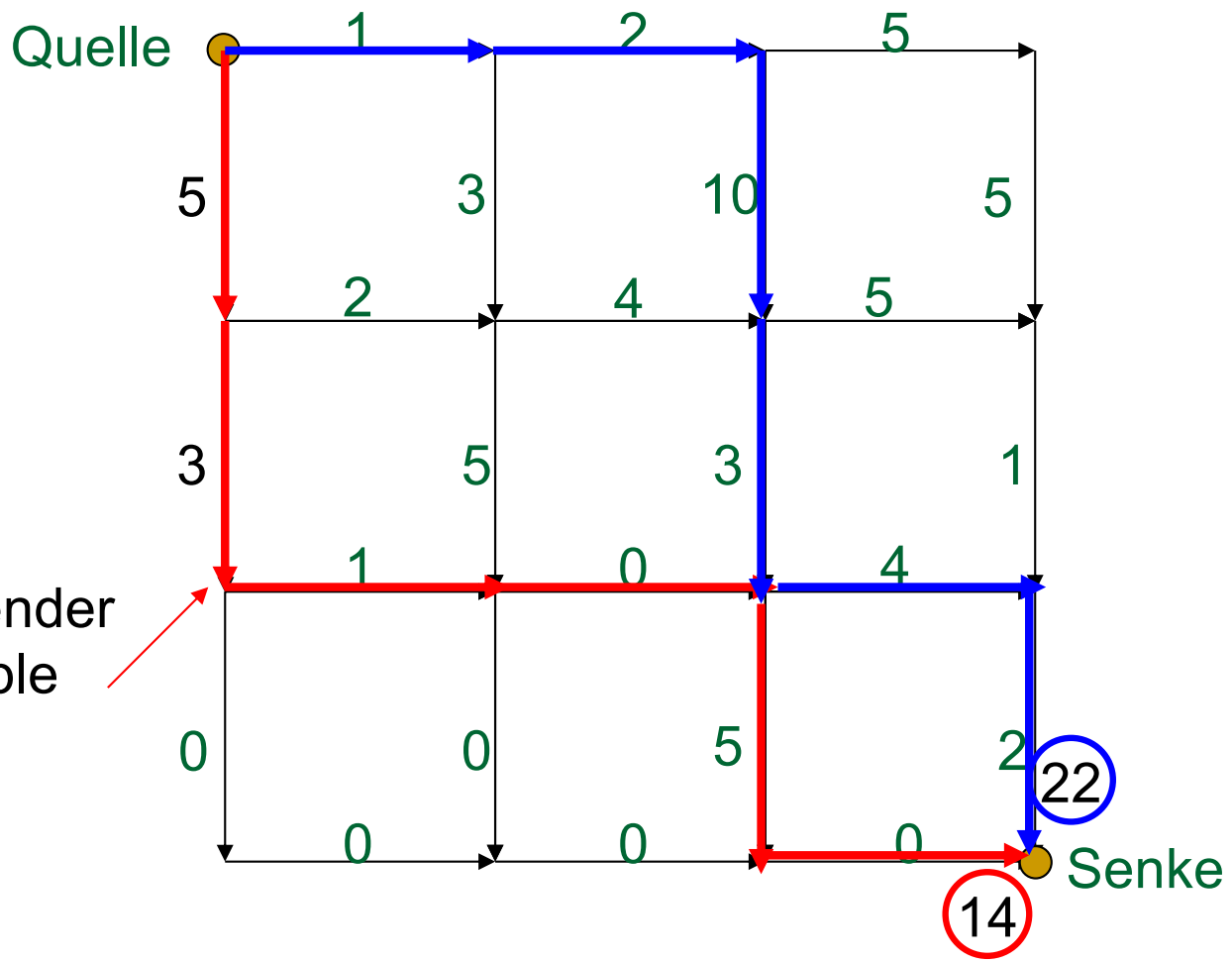
Eingabe: Ein gewichtetes Gitter *G* mit zwei speziellen Knoten (Quelle und Senke)

Ausgabe: Ein längster Pfad in *G* von Quelle zu Senke

TIM: Beispiel



TIM: Gieriger Algorithmus ist nicht exakt



Vielversprechender Start führt in üble Gegend!

Warum nur ein Problem gleichzeitig lösen?

- TIM:
  Suche den längsten Pfad von (0,0) nach (*m, n*)
- Verallgemeinertes Problem:
  Suche den längsten Pfad von (0,0) nach (*i, j*) für alle $i = 1, ..., m$, $j = 1, ..., n$.

Das allgemeine Problem sieht zwar schwieriger aus, ist aber keine große zusätzliche Arbeit:

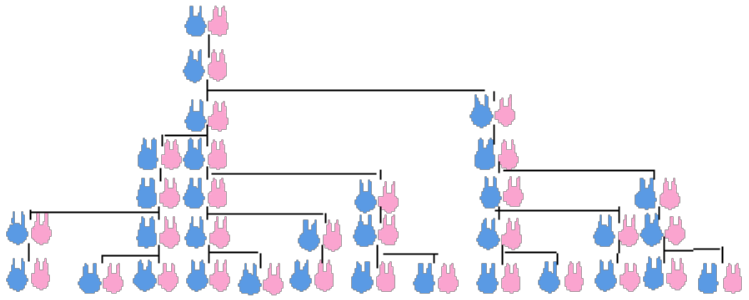
Grundidee der dynamischen Programmierung!

Dynamische Programmierung (DP)

- Idee
  - **Zerteile** Problem in kleinere Teilprobleme
  - Baue Lösungen größerer Probleme aus Lösungen kleinerer Probleme zusammen
  - **Effiziente** Realisierung eines rekursiven Algorithmus durch **Speicherung** von Teilresultaten
    - Speichere diese in **DP-Tabelle**
    - Kaufe dir Rechenzeit mit Speicherplatz

Richard E. Bellman

- Wichtige Überlegungen
  - Wie teile ich auf? Definiere Wert der Optimallösung rekursiv
  - Bestimme diesen Wert **“bottom-up”**



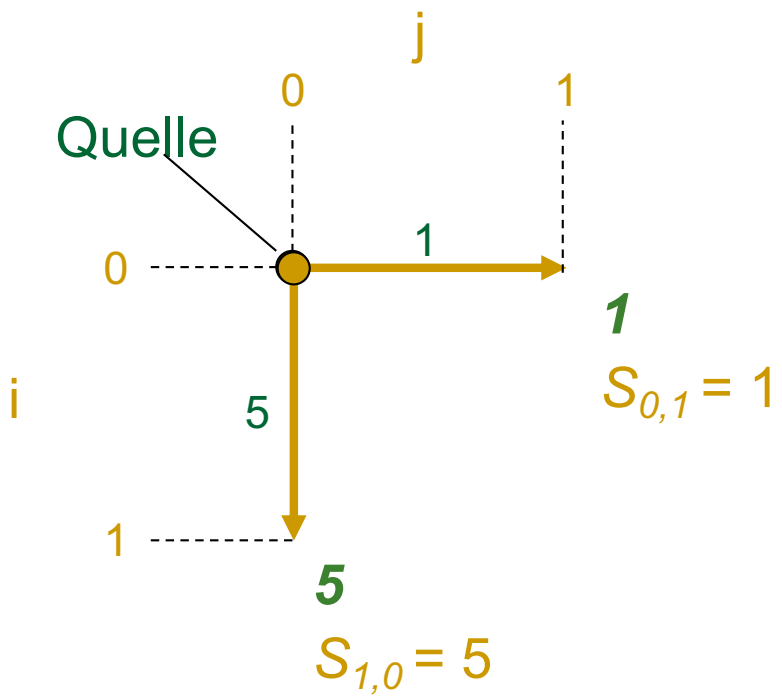
DP: Fibonacci numbers

$F_n = F_{n-1} + F_{n-2}, \quad F_0 = 0, \quad F_1 = 1$

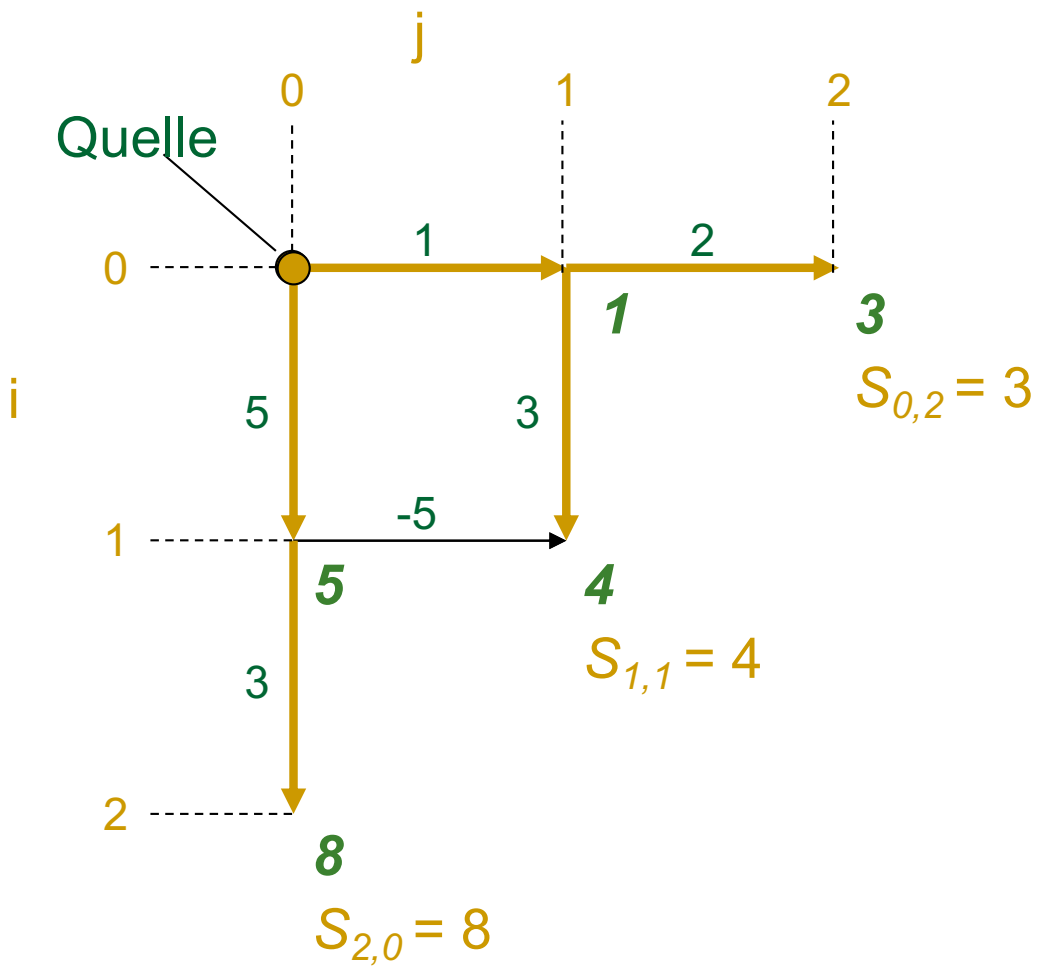
```
fib(n)
    if n = 0 : return 0;
    if n = 1 : return 1;
    else return fib(n-1) + fib(n-2)
```

```
fib_DP(n)
    f[0] = 0; f[1] = 1;
    for i = 2, ..., n : f[i] = f[i-1] + f[i-2]
    return f[n];
```

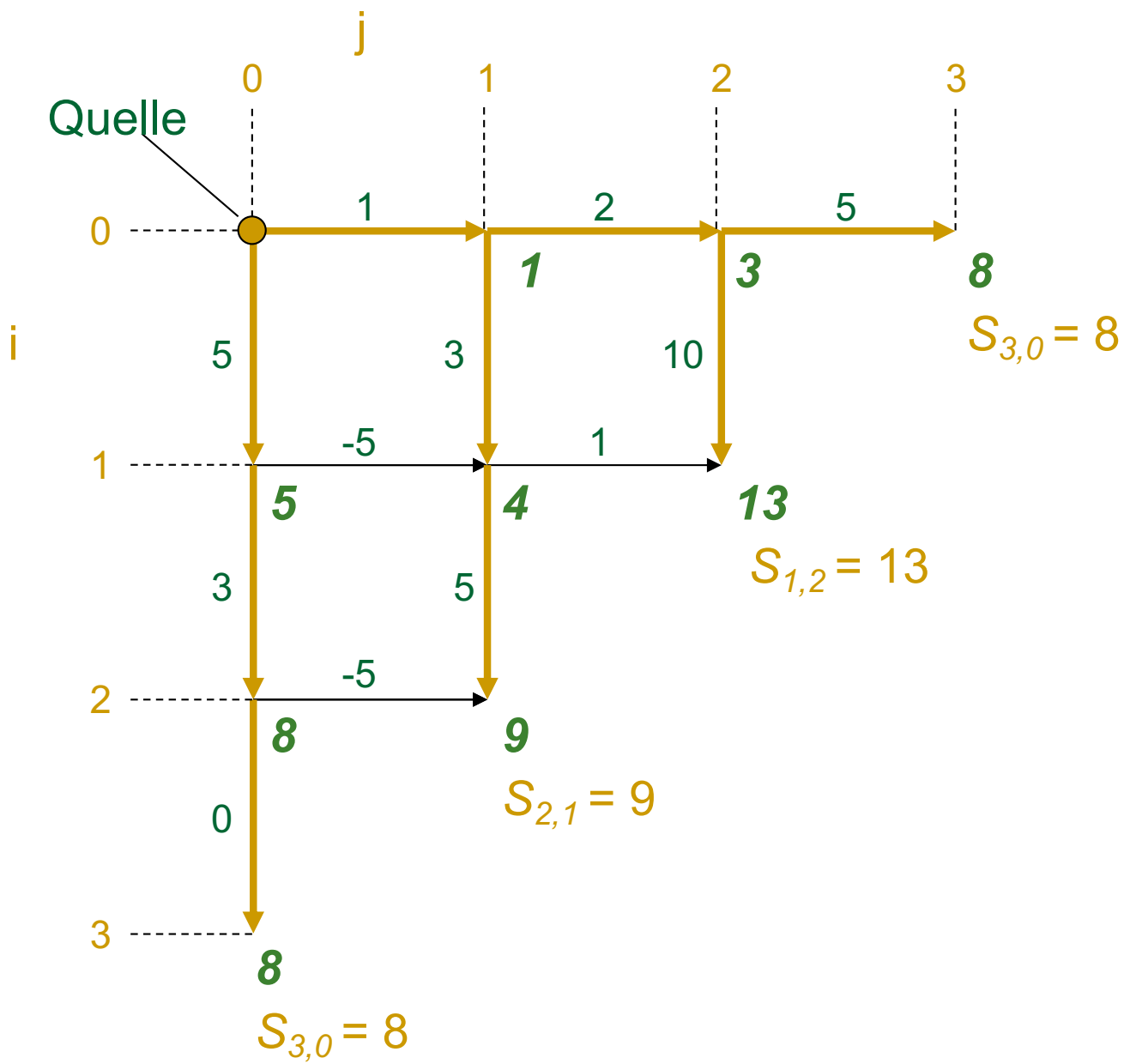
TIM: Dynamische Programmierung



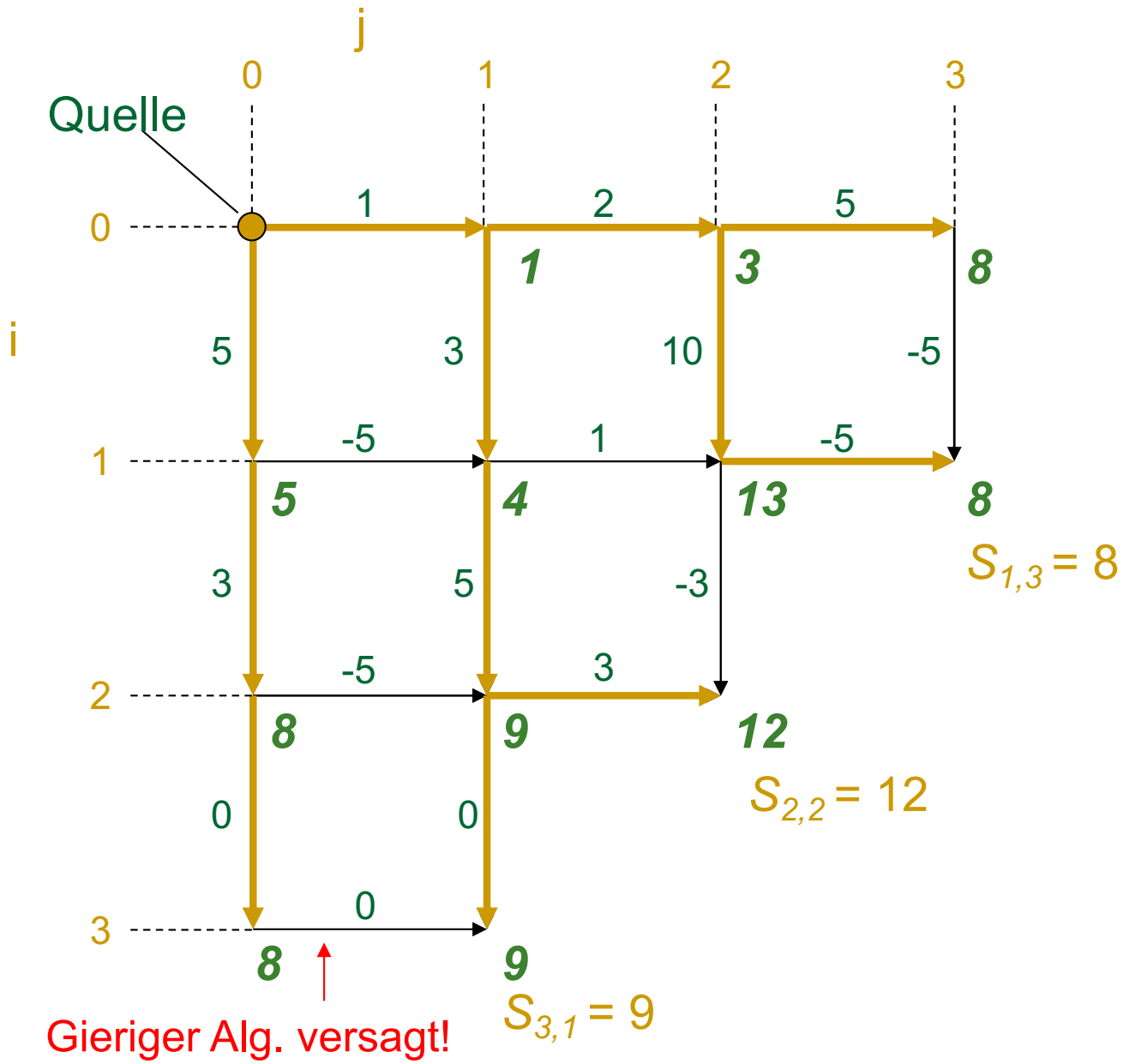
TIM: Dynamische Programmierung



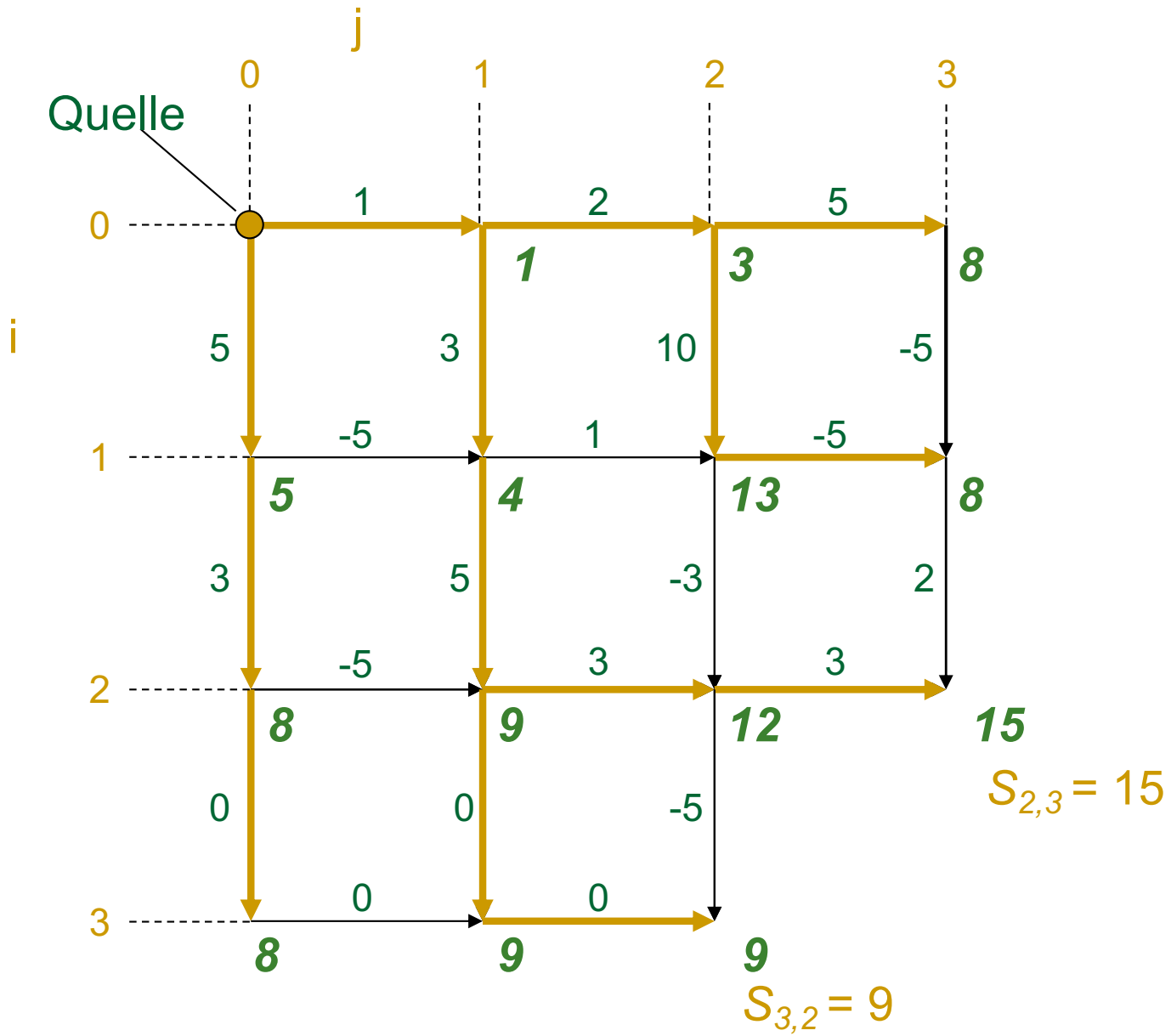
TIM: Dynamische Programmierung



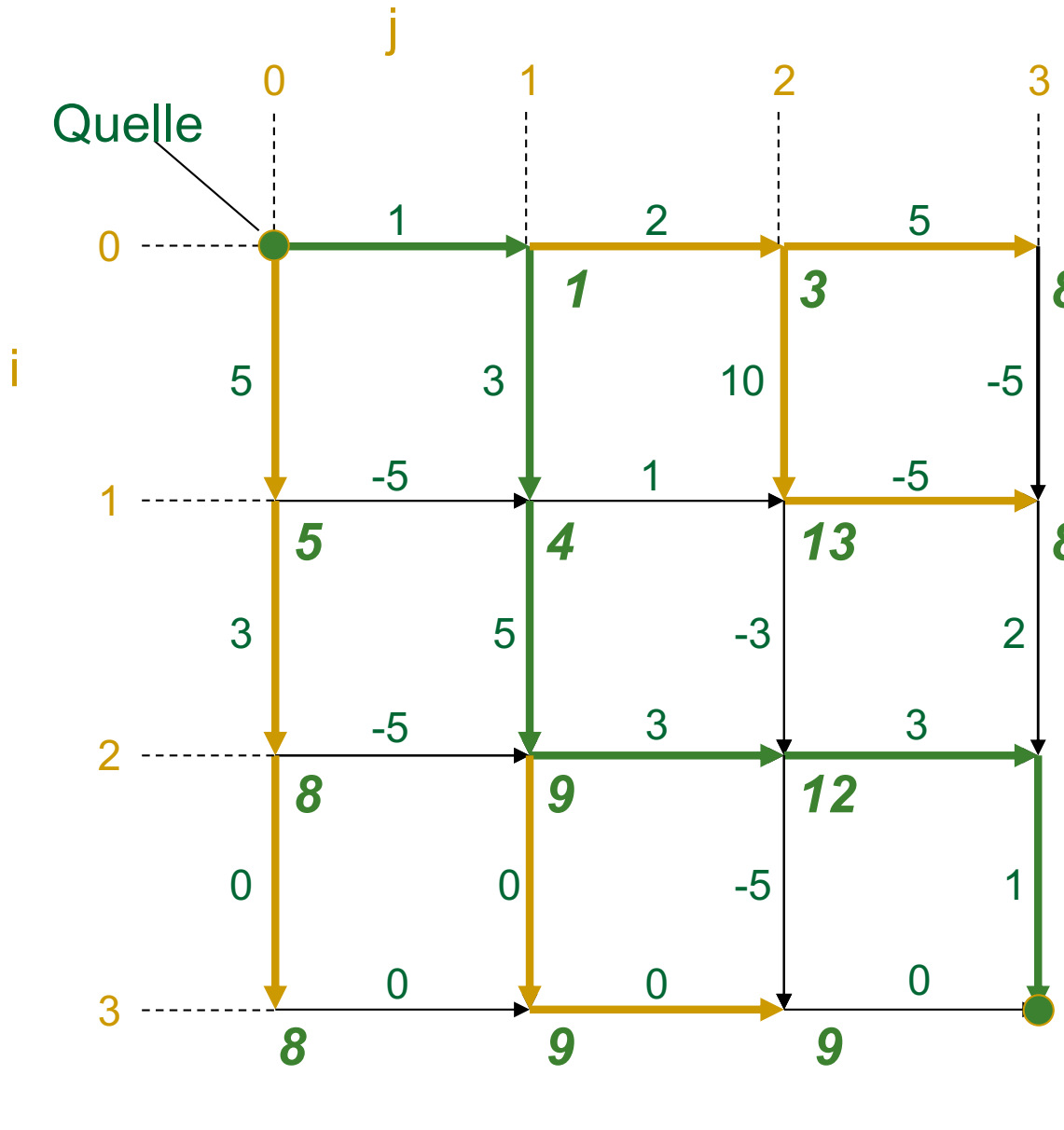
TIM: Dynamische Programmierung



TIM: Dynamische Programmierung



TIM: Dynamische Programmierung



Fertig!

(mit Rückverfolgungen)

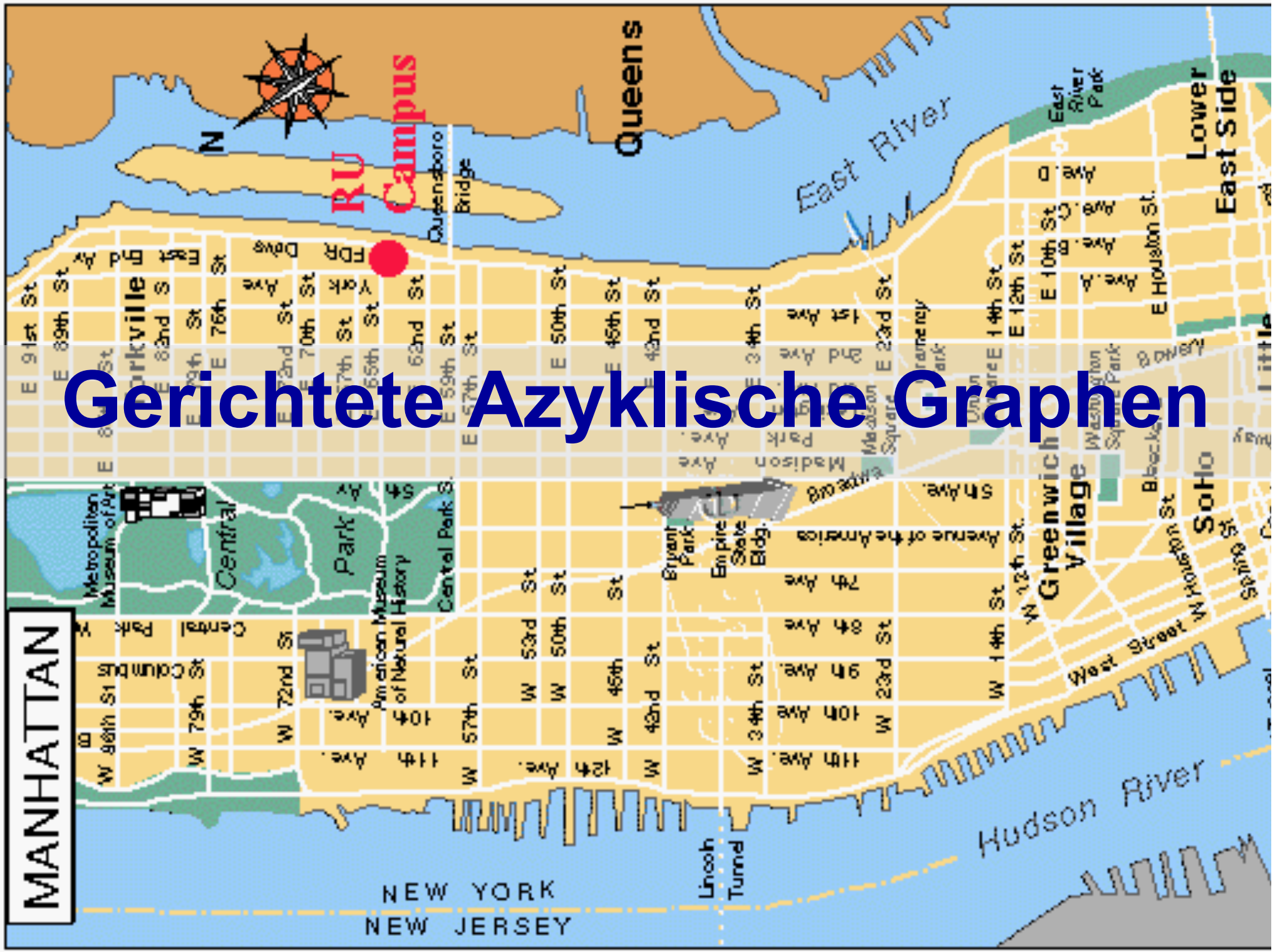
$S_{3,3} = 16$

TIM: Rekurrenz

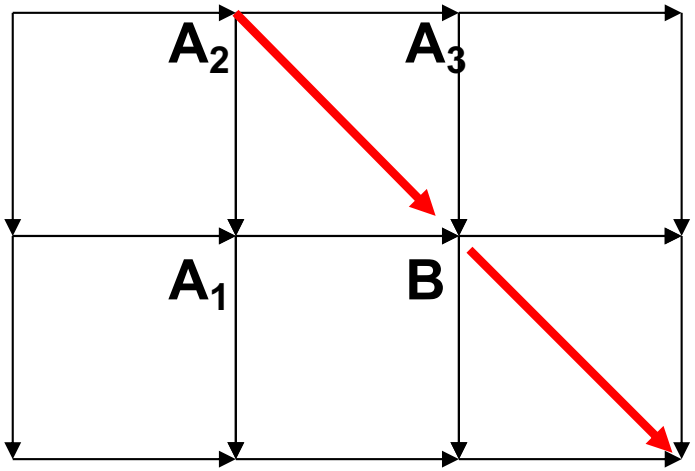
Berechne *Score*(*i,j*) durch die Rekurrenz-Relation:

$s_{i,\,j}$ =

Laufzeit: *n x m* (*n* = # Spalten, *m* = # Zeilen)


Gerichtete Azyklische Graphen
RU Campus
Queens
East River
Queensboro Bridge
FDR
York
Ave
Lower East Side
East River Park
E Houston St
Greenwich Village
SoHo
Washington Square Park
Madison Square Park
Bryant Park
Empire State Bldg.
Avenue of the Americas
5th Ave
Broadway
West Street
Hudson River
Lincoln Tunnel
Central Park
Metropolitan Museum of Art
American Museum of Natural History
Central Park S.
MANHATTAN
NEW YORK
NEW JERSEY

Manhattan ist kein perfektes Gitter



Score(*B*) :

$$s_B = \max \begin{cases} s_{A1} + \text{Gewicht der Kante } (A_1, B) \\ s_{A2} + \text{Gewicht der Kante } (A_2, B) \\ s_{A3} + \text{Gewicht der Kante } (A_3, B) \end{cases}$$

Allgemeine Rekurrenz-Relation:

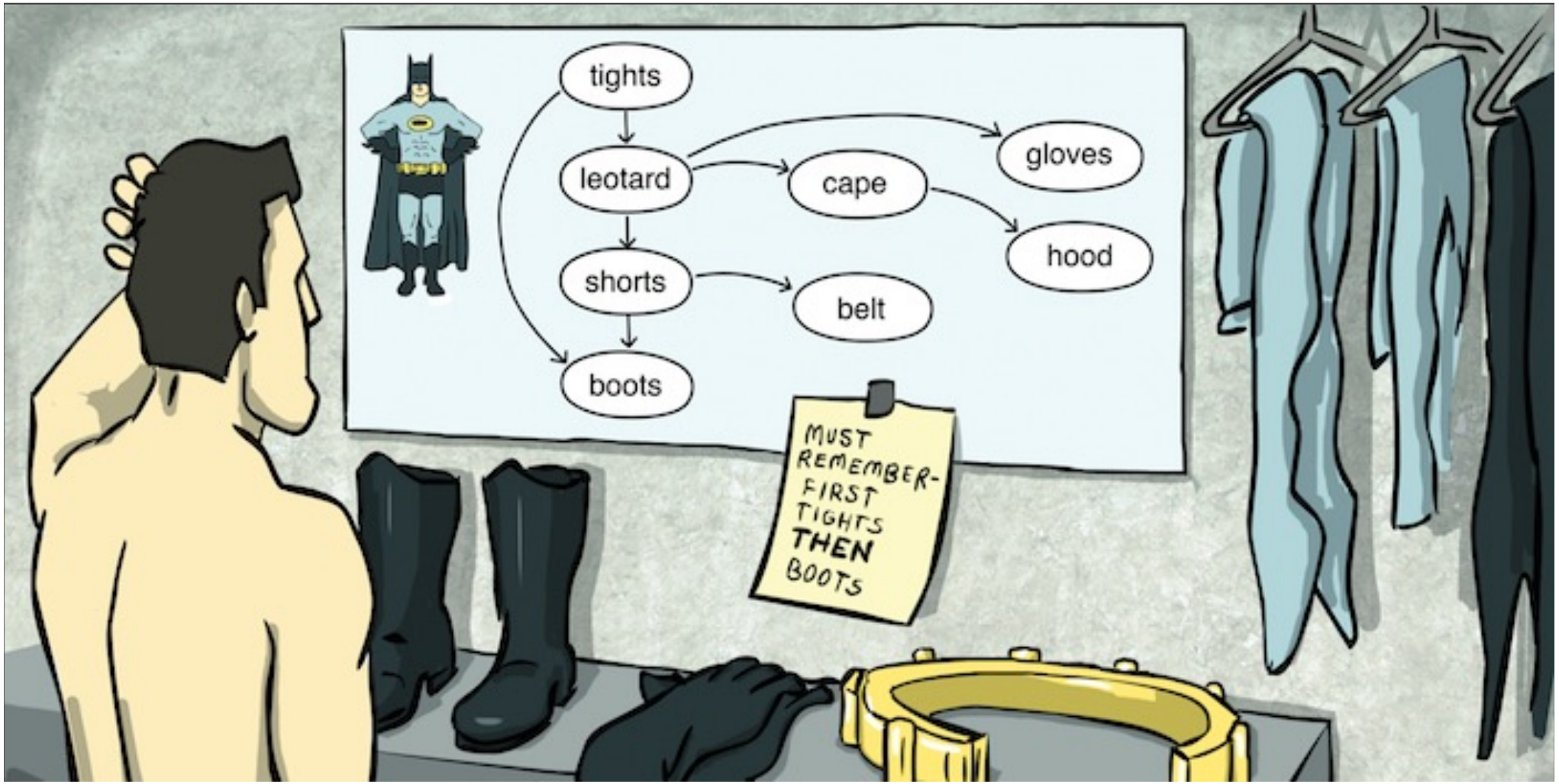
$$s_x = \max_{Y \in \text{Vorgänger}(X)} \{s_Y + \text{Gewicht der Kante } (Y, X)\}$$

Vorgänger(X) := {Knoten Y mit Kante Y $\rightarrow$ X}

DAG: Directed Acyclic Graphs

:= Graphen mit gerichteten Kanten ohne Zyklen
(d.h.: Kein Weg von einem Knoten zu sich selbst)

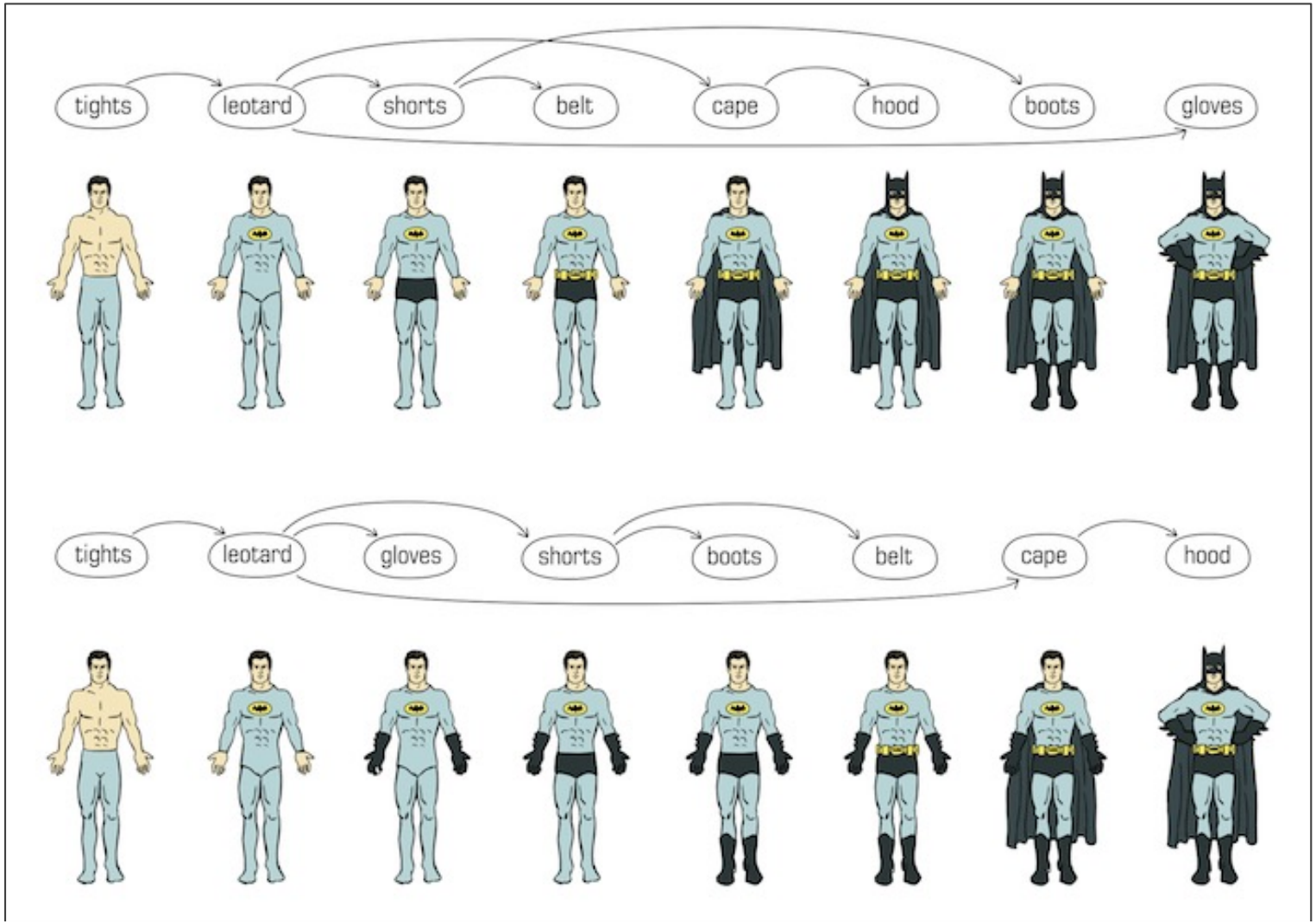
Wir brauchen eine Ordnung: Wenn wir Knoten X analysieren, müssen Werte für alle Vorgänger Y bereits vorliegen.



Topologische Ordnungen

Eine Nummerierung von Knoten heißt ***topologische Ordnung*** des DAG

:⇔ jede Kante läuft von einem Knoten mit kleinerer Nummer
zu einem Knoten mit größerer Nummer



Der längste Pfad in einem DAG: Problem

Ziel: Finde den längsten (d.h. am stärksten gewichteten) Pfad zwischen zwei Knoten in einem gewichteten DAG

Eingabe: Ein gewichteter DAG *G* mit zwei ausgezeichneten Knoten (Quelle und Senke)

Ausgabe: Ein längster Pfad in *G* von Quelle zu Senke

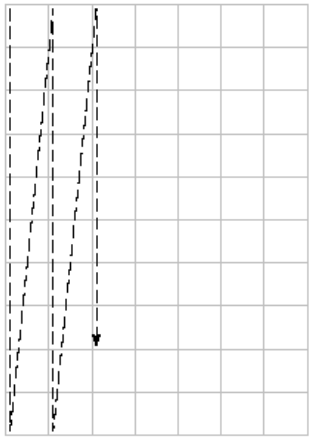
Der längste Pfad: Dynamische Programmierung

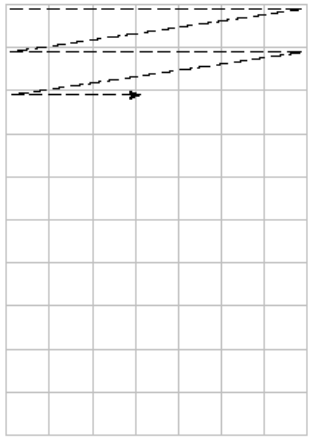
Rekurrenz-Relation:

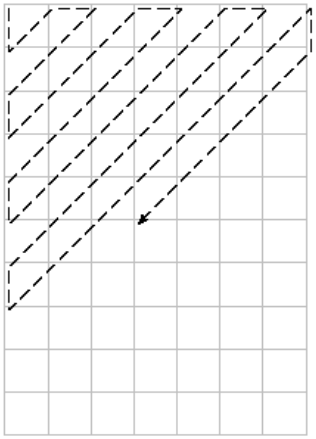
$$s_X = \max_{Y \in \text{Vorgänger}(X)} \{s_Y + \text{Gewicht der Kante } (Y, X)\}$$

Laufzeit ist O(# Kanten)

Verschiedene Strategien:







Bsp: Münzenkette

Spielregeln:

- Auf dem Tisch liegt eine Kette von Münzen mit verschiedenem Wert.
- Ich kann entweder die erste oder die letzte Münze nehmen und behalten.
- Dann ist mein Gegenspieler dran…
- …bis alle Münzen weg sind.

Ziel: möglichst viel gewinnen.

Frage: Kann ich vor Beginn des Spiels angeben, wieviel ich mindestens gewinne wenn ich bestmöglich spiele (egal, welche Züge mein Gegenspieler macht)?

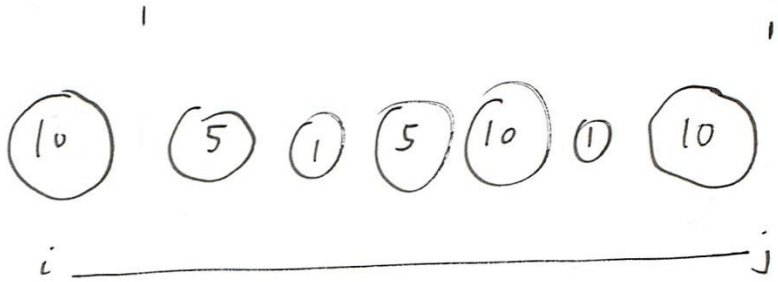
Antwort: Dynamische Programmierung!

Rekurrenz für eine beliebige Teilsequenz von i…j:

$G(i,j) = \max\{ v_i + \min\{G(i+1,j-1), G(i+2,j)\},\ v_j + \min\{G(i+1,j-1), G(i,j-2)\} \}$

M

Münzen mit Wert $v_i \in \mathbb{N}$ für $i \in M$

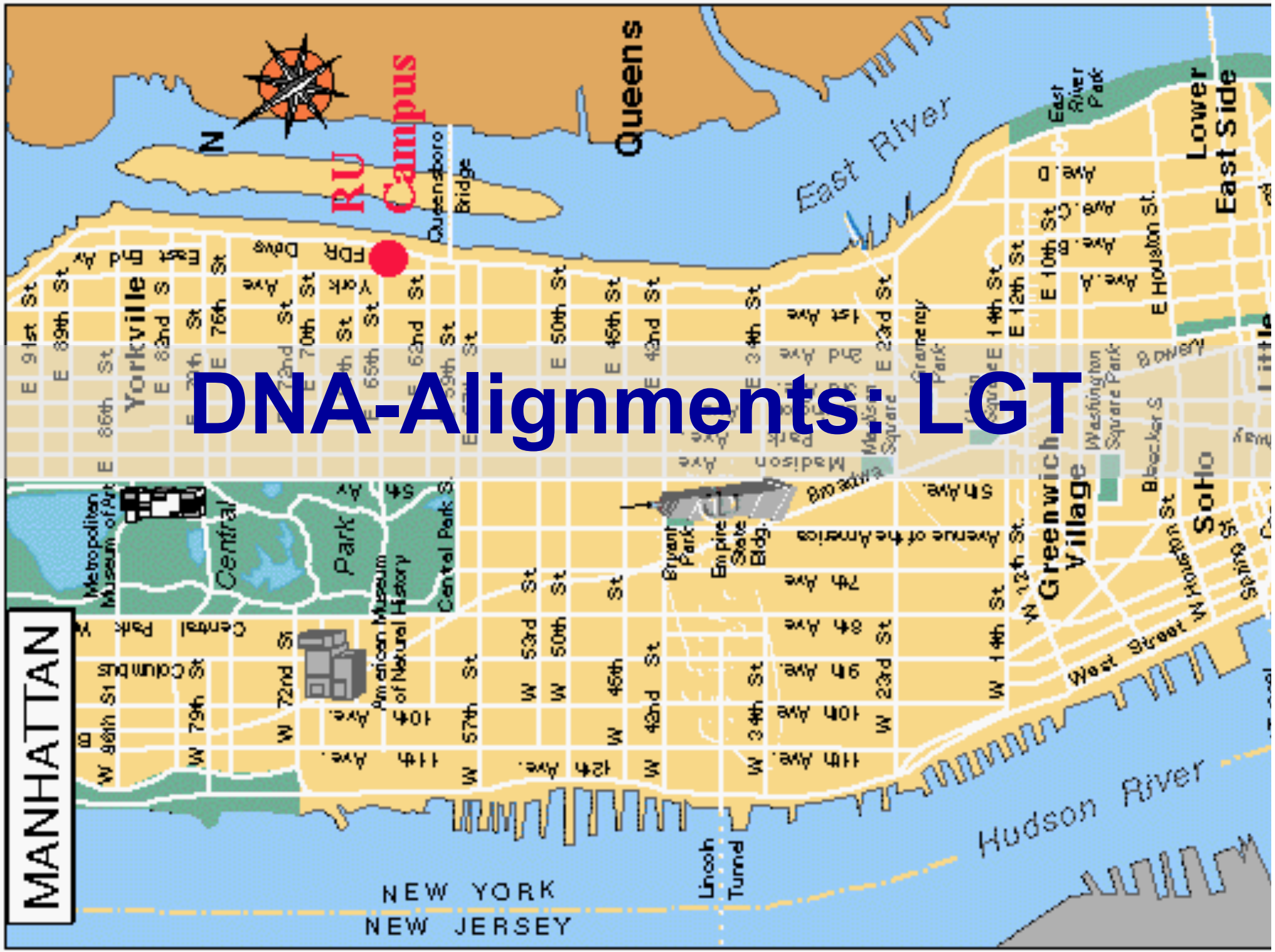


$$G(i,j) = \max \{ v_i + \min \{ G(i+2, j), G(i+1, j-1) \}, v_j + \min \{ G(i+1, j-1), G(i, j-2) \} \}$$

// Aufgabe: progr., rek. / DP

$$G(i,i) = v_i$$

$$G(i, i+1) = \max \{ v_i, v_{i+1} \}$$


DNA-Alignments: LGT
RU Campus
Queens
East River
Queensboro Bridge
Yorkville
Lower East Side
Greenwich Village
SoHo
Central Park
Metropolitan Museum of Art
American Museum of Natural History
Empire State Bldg.
Bryant Park
Washington Square Park
Madison Square Park
Gramercy Park
Hudson River
MANHATTAN
NEW YORK
NEW JERSEY
Lincoln Tunnel
FDR Drive
West Street

Alignments: 2-zeilige Darstellung

Für 2 DNA-Sequenzen:

v: **ATGTTAT** $m = 7$

w: **ATCGTA** $n = 6$

Alignment: 2 $\times$ k **-Matrix mit** $k \geq \max(m,n)$

v	A	T	--	G	T	T	A	T
w	A	T	C	G	T	--	A	--

5 Matches | 1 Einfügung | 2 Löschungen

Indels (= INsertions & DELetionS)

Längste gemeinsame Teilsequenz: Problem

Ziel: Finde die längsten identischen, geordneten Teilsequenzen

Eingabe: 2 Sequenzen

$\boldsymbol{v} = v_1\, v_2 \ldots v_m$

$\boldsymbol{w} = w_1\, w_2 \ldots w_n$

Ausgabe: Eine Abfolge von Positionen **i** für **v** und **j** in **w** mit

$$1 \leq i_1 < i_2 < \ldots < i_t \leq m$$

$$1 \leq j_1 < j_2 < \ldots < j_t \leq n$$

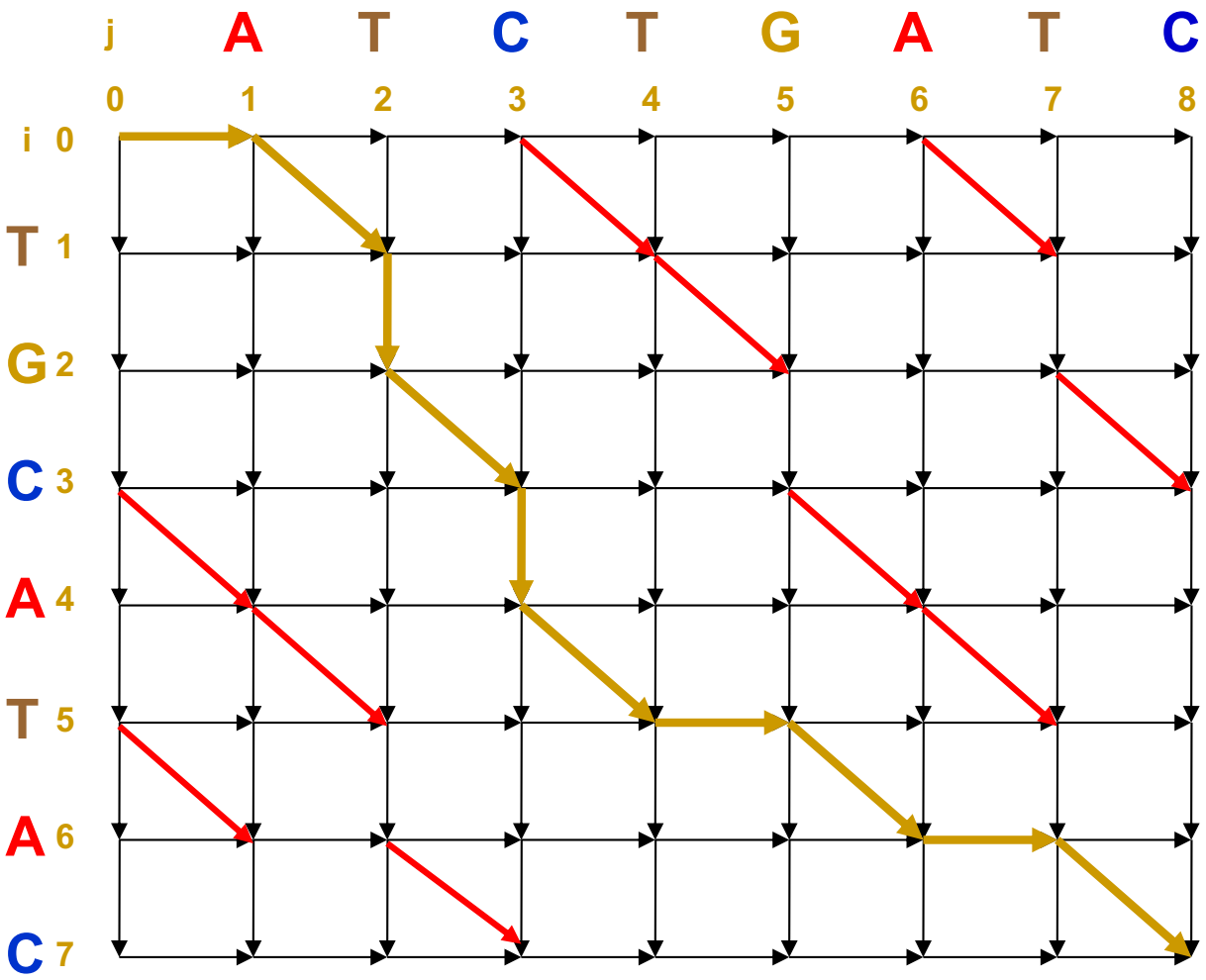
so dass t maximal ist und $v_{i_s} = w_{j_s}$ für alle s

Koordinaten *i*:	0	1	2	2	3	3	4	5	6	7	8
Elemente von *v*:		A	T	--	C	--	T	G	A	T	C
Elemente von *w*:		--	T	G	C	A	T	--	A	--	C
Koordinaten *j*:	0	0	1	2	3	4	5	5	6	6	7

(0,0)→(1,0)→(2,1)→(2,2)→(3,3)→(3,4)→(4,5)→(5,5)→(6,6)→(7,6)→(8,7)


Edit-Graphen
RU Campus
Queens
East River
Queensboro Bridge
N
MANHATTAN
NEW YORK
NEW JERSEY
Hudson River
Yorkville
Lower East Side
Greenwich Village
SoHo
Central Park
Metropolitan Museum of Art
American Museum of Natural History
Empire State Bldg.
Bryant Park
Union Square
Washington Square Park
Lincoln Tunnel
FDR Drive
East River Park
Avenue of the Americas
West Street

LGT & Manhattan-Tourismus



Jeder Pfad ist eine gemeinsame Teilsequenz

Jede diagonale Kante gibt 1 zusätzliches Element

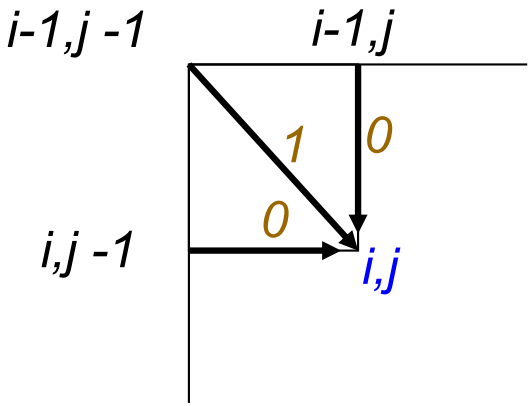
***LGT Problem:** Finde den Pfad mit den meisten diagonalen Kanten*

LGT: Rekurrenz

Länge der LGT bis (i, j)

:

$$s_{i,j} = \max \begin{cases} s_{i-1,j} \\ s_{i,j-1} \\ s_{i-1,j-1} + 1 \text{ wenn } v_i = w_j \end{cases}$$



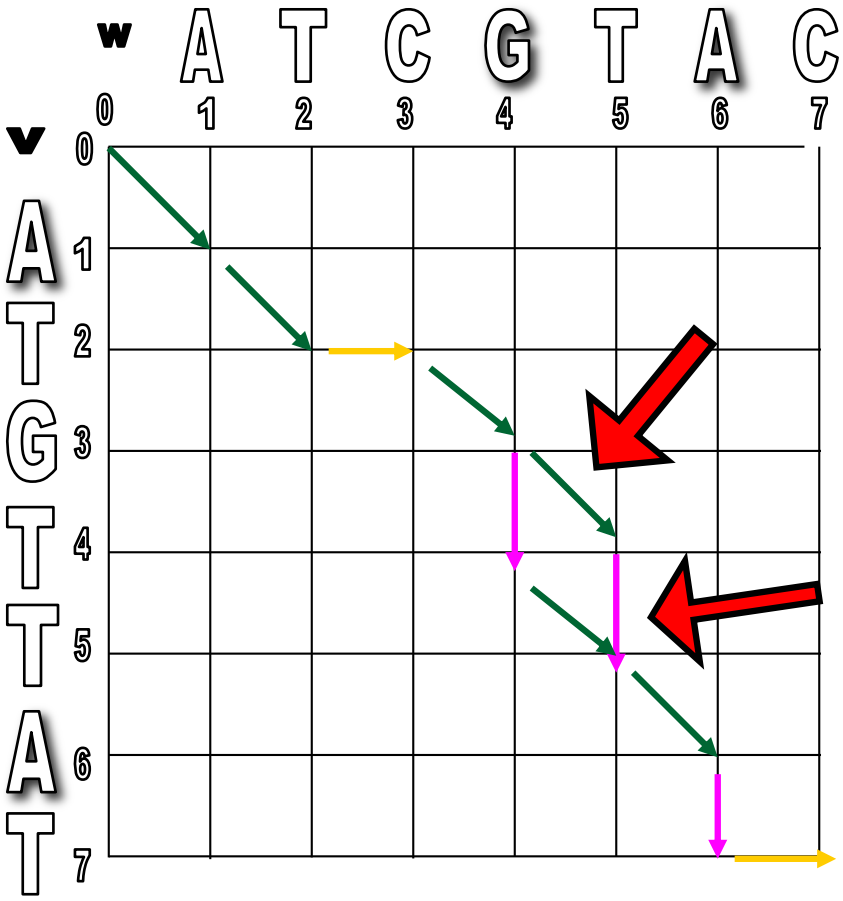
Edit-Abstand

:= Minimale Anzahl elementarer Operationen (Einfügungen, Löschungen, Substitutionen), um String **v** in String **w** zu überführen
:= $d_E(\mathbf{v},\mathbf{w})$

Hamming-Abstand:	Edit-Abstand:
Vergleicht immer ***i*-ten** Buchstaben von **v** mit ***i*-tem** Buchstaben von **w**	Vergleicht (für ‘beste’ *i,j*) ***i*-ten** Buchstaben von **v** mit ***j*-tem** Buchstaben von **w**
V = ATATATATT **W** = TATATATA	1 Einfügung 1 Löschung →
V = −ATATATATT **W** = TATATATA−	
$d_H(\mathbf{v},\mathbf{w}) = \mathbf{8}$	$d_E(\mathbf{v},\mathbf{w}) = \mathbf{2}$
(trivial)	(nicht-trivial)

Beispiel: ATGTTAT → ATCGTAC

LGT im Edit-Graphen



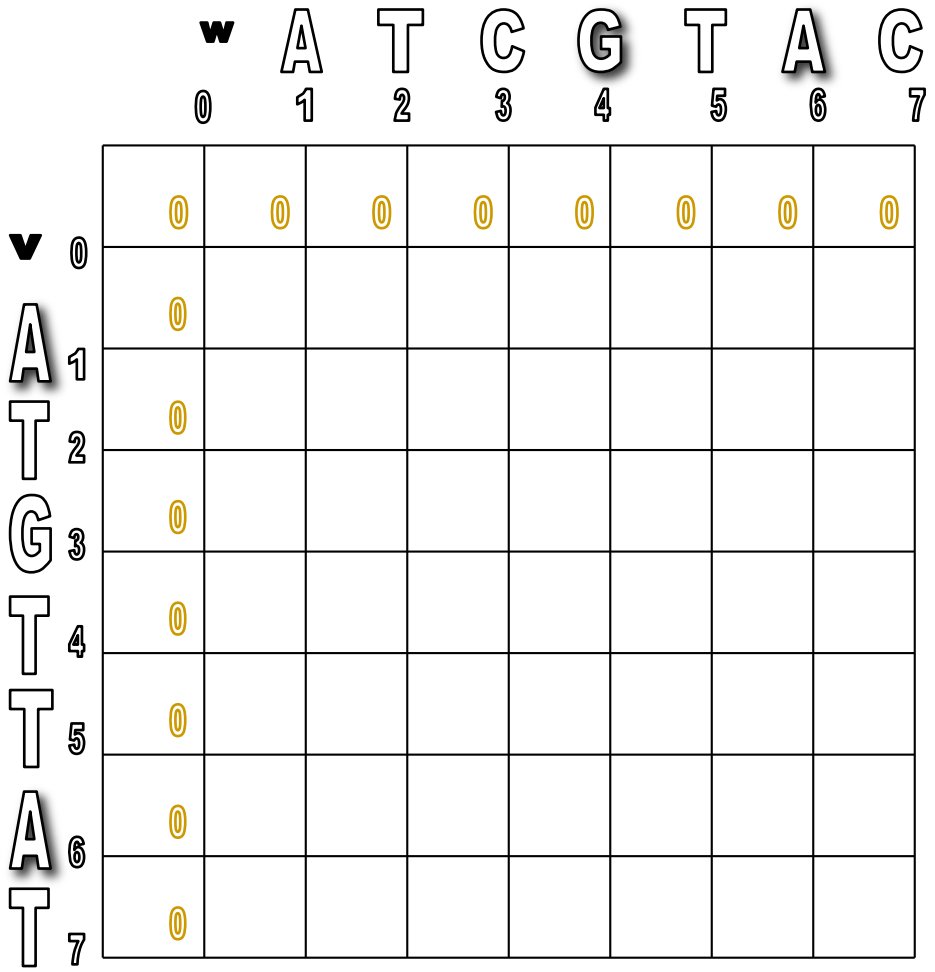
↓ und → sind Indels in **v** und **w** mit Score 0.

↘ sind Matches mit Score 1.

→ Alignment-Pfad hat Score 5

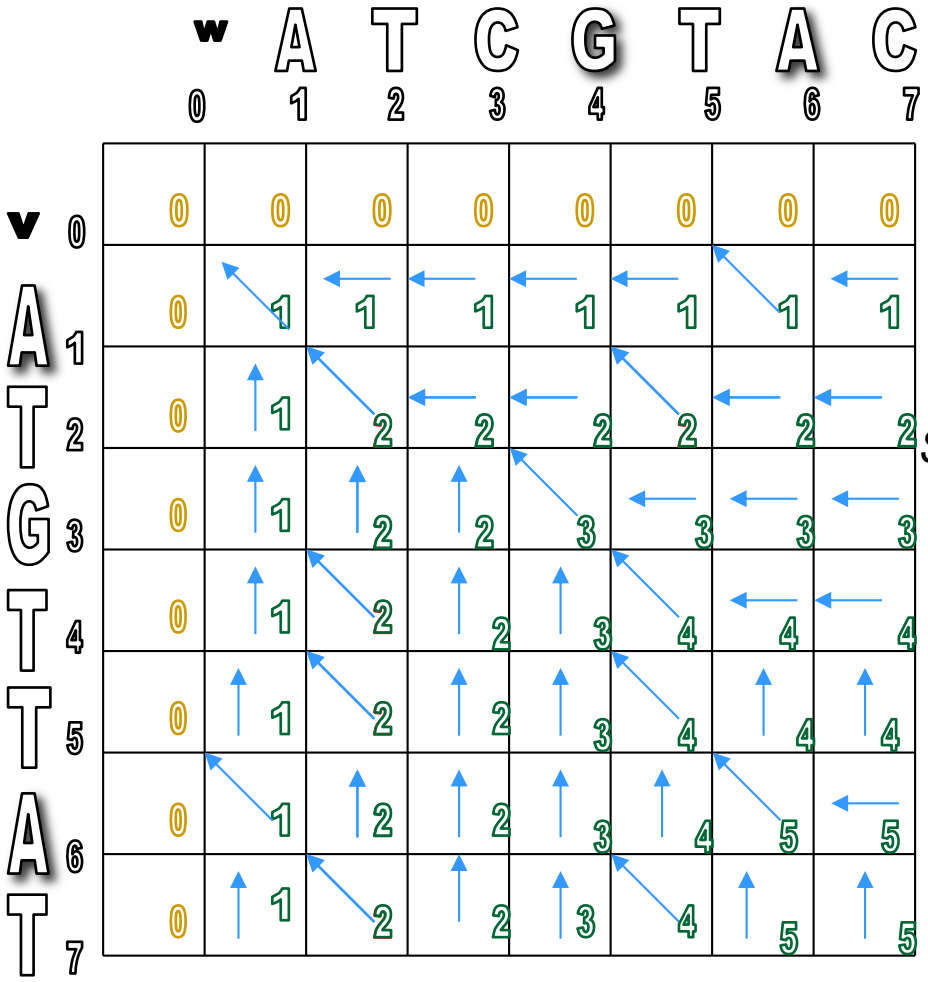
Alternative Pfade können gleichen Score haben!

LGT: Dynamische Programmierung



Initialisiere Reihe 0 und Spalte 0
mit Score 0 (Indel-Präfix)

LGT: Dynamische Programmierung



$S_{i,j}$ = **max** { $S_{i-1,\,j-1}$ + 1 wenn $v_i = w_j$ (von NW); $S_{i-1,\,j}$ (vom Norden); $S_{i,\,j-1}$ (vom Westen)

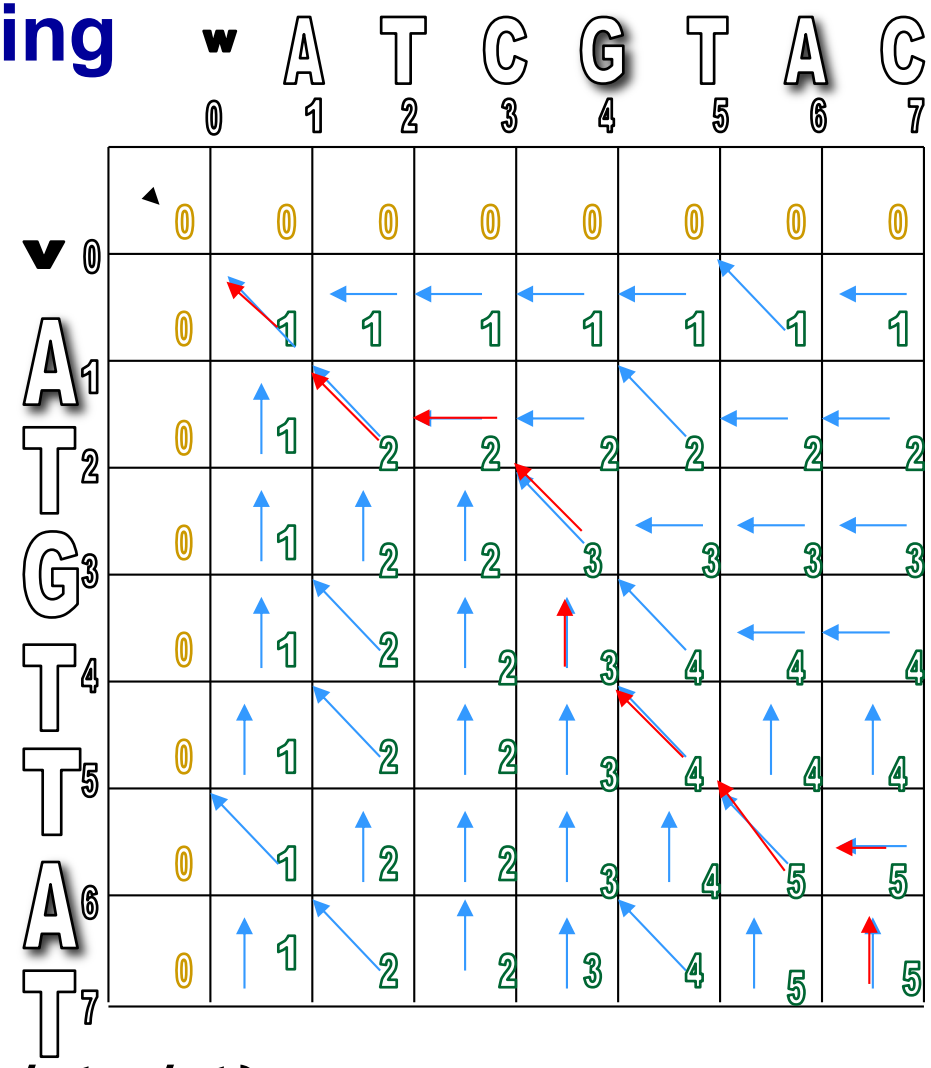
Manhattan-Touristen mit allen vertikalen und horizontalen Kanten = 0

LGT Algorithmus

```
1.  LGT(v,w)
2.     for i ← 1 to n
3.          s_{i,0} ← 0
4.     for j ← 1 to m
5.          s_{0,j} ← 0
6.     for i ← 1 to n
7.          for j ← 1 to m
8.             if (v_i = w_j )
9.                s_{i,j} ← max (s_{i-1,j}, s_{i,j-1}, s_{i-1,j-1} + 1)
10.            else
11.               s_{i,j} ← max (s_{i-1,j}, s_{i,j-1})
12.            case:
13.               s_{i,j} = s_{i-1,j}                    : b_{i,j} ← "↑"
14.               s_{i,j} = s_{i,j-1}                    : b_{i,j} ← "←"
15.               s_{i,j} = s_{i-1,j-1} + 1 and v_i = w_j : b_{i,j} ← "↖"
16.    return (s_{n,m}, b)
```

O(*nm*) um das Gitter zu füllen

Ausgabe LGT: Backtracking




```
1. PrintLGT(b,v,i,j)
2.     if  i = 0 or j = 0
3.         return
4.     case
5.         b_{i,j} = "↖" : PrintLGT(b,v, i-1,j-1)
6.                         print v_i
7.         b_{i,j} = "←" : PrintLGT(b,v, i,j-1)
8.         b_{i,j} = "↑" : PrintLGT(b,v, i-1,j)
```