Dynamische Programmierung

- Fibonacci-Folge
- Partition in Worte
- Edit Distance
- Teilmengensumme
- Dynamische Programmierung auf Bäumen

Kapitel 3 in Erickson (2019) (ohne 3.6 und 3.9)

Holger Dell

Dynamische Programmierung

- Fibonacci-Folge
- Partition in Worte
- Edit Distance
- Teilmengensumme
- Dynamische Programmierung auf Bäumen

Mātrāvṛtta

- Mātrāvṛtta. Versmaß in Gedichten und Liedern
- Piṅgala (~600-200 v.Chr.). Es gibt genau fünf 4-Schlag Versmaße:

  — —, — • •, • — •, • • —, und • • • •
- Lange Silbe —, Kurze Silbe •
- Virahāṇka (~700 n.Chr.). Anzahl M(n) der n-Schlag Versmaße:

$$M(n) = M(n-2) + M(n-1).$$

- Leonardo of Pisa a.k.a. Fibonacci (1202).

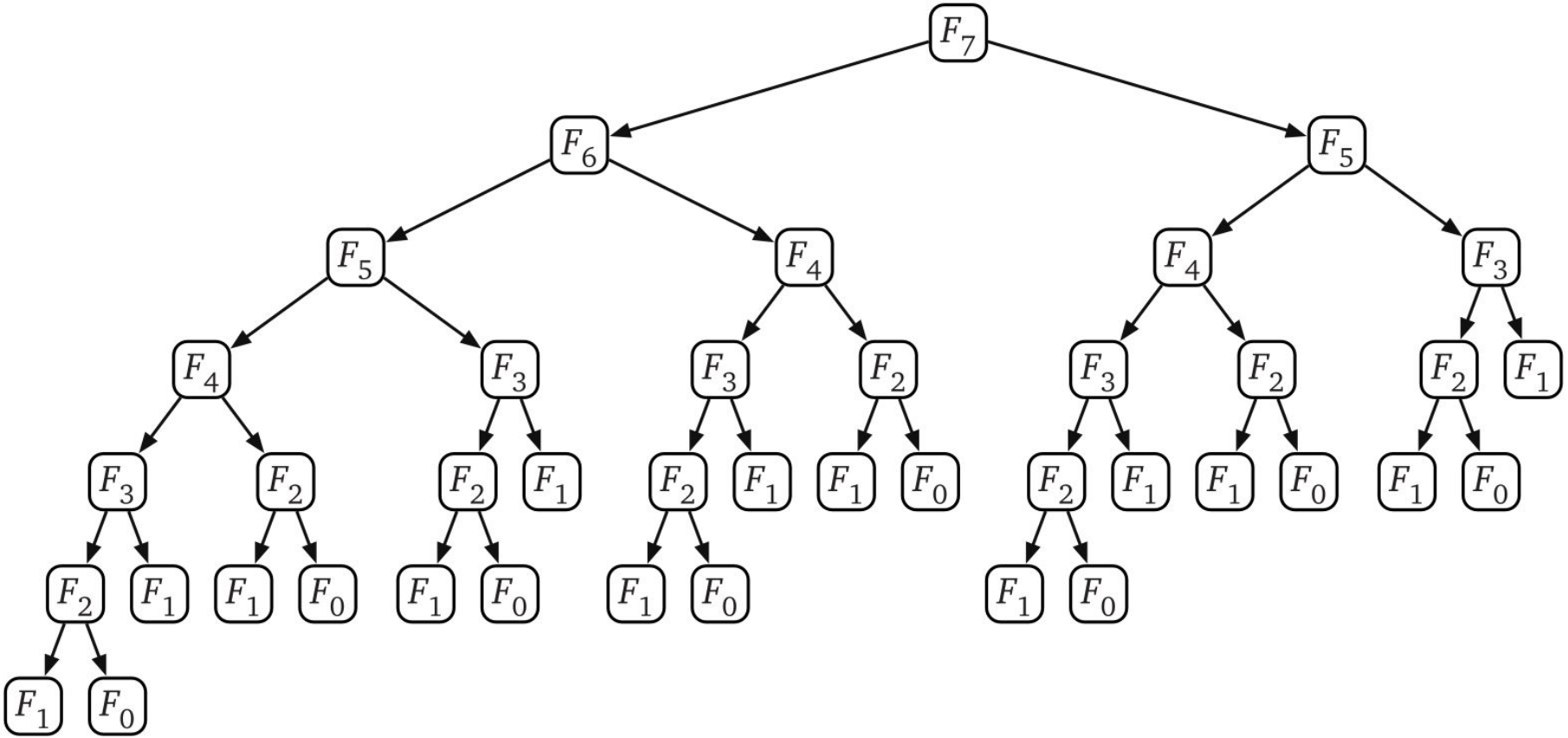
$$F_0 = 0 \qquad F_1 = 1 \qquad F_n = F_{n-1} + F_{n-2}$$

Rekursiver Algorithmus für die Fibonaccifolge

```
RecFibo(n):
if n = 0: return 0
else if n = 1: return 1
else: return RecFibo(n–1) + RecFibo(n–2)
```

- **Laufzeit?** $T(n)$ = Anzahl der **rekursiven Aufrufe**.
- $T(0) = T(1) = 1$ und $T(n) = T(n-1) + T(n-2) + 1$
- Induktion $\Rightarrow T(n) = 2\,F_{n+1} - 1$
- $F_n = \Theta(\phi^n)$, wobei goldener Schnitt $\phi = (\sqrt{5} + 1)/2 \approx 1{,}61803$
- $\Rightarrow T(n) = \boldsymbol{\Theta(\phi^n)}$

Rekursionsbaum für RecFibo



Memoisierung (*Memoization*)

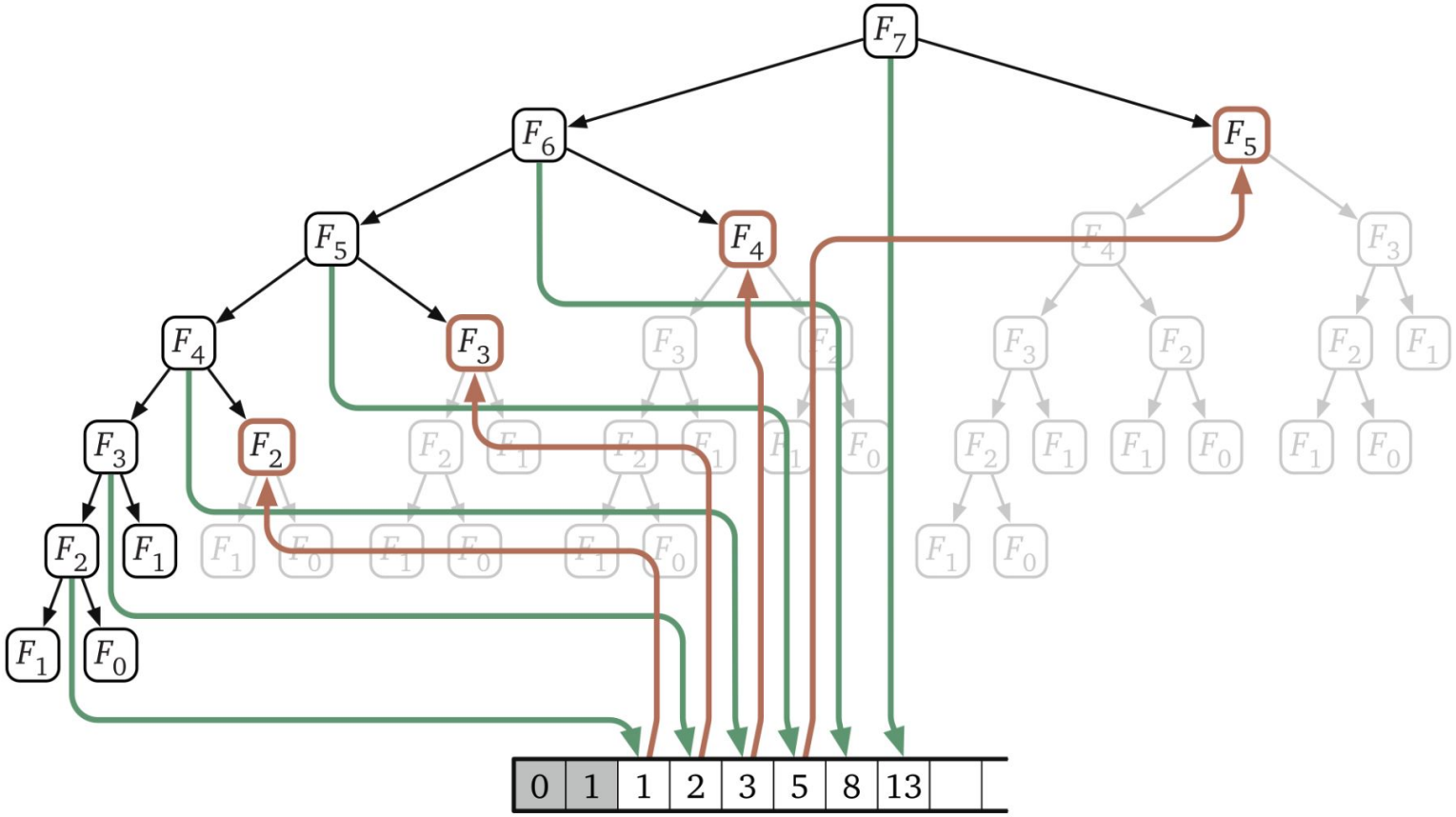
“Merk dir alles!”

```
MemFibo(n):
if n = 0:
   return 0
else if n = 1:
   return 1
else:
   if F[n] is undefined:
      F[n] = MemFibo(n−1) + MemFibo(n−2)
   return F[n]
```

Zeit?

Jedes `F[i]` höchstens einmal beschrieben ⇒ nur O(n) Additionen.

Rekursionsbaum für MemFibo



Dynamische Programmierung

“Mach Memoisierung iterativ!”

```
IterFibo(n):
F[0]=0
F[1]=1
for i = 2..n:
   F[i] = F[i-1] + F[i-2]
return F[n]
```

- Zeit. O(n) Additionen
- Platz. O(n) Zahlen
- Frage. Kommen wir mit weniger Platz aus?

Platzsparende Dynamische Programmierung

“Merk dir nur das, was du wirklich noch brauchst!”

```
IterFibo2(n):
prev = 1
curr = 0
for i = 1..n:
    next = curr + prev
    prev = curr
    curr = next
return curr
```

- Zeit. O(n) Additionen
- Platz. O(1) Zahlen

Zusammenfassung: Fibonacci ausrechnen

	Zeit	Platz
Rekursiv (RecFibo)	$\Theta(\phi^n)$	$\Theta(n)$
Memoisierung (MemFibo)	$\Theta(n)$	$\Theta(n)$
Dynamische Programmierung (IterFibo)	$\Theta(n)$	$\Theta(n)$
Platzsparende Dynamische Programmierung (IterFibo2)	$\Theta(n)$	$\Theta(1)$

- Es gibt Algorithmen mit nur **O(log n)** arithmetischen Operationen.
- **N.B.** Zeit/Platz sind eigentlich größer, weil die Zahlen sehr groß werden und nicht mehr in ein Register passen!

Dynamische Programmierung

- Fibonacci-Folge
- Partition in Worte
- Edit Distance
- Teilmengensumme
- Dynamische Programmierung auf Bäumen

In Worte aufteilen

- Gegeben:
  - Zeichenkette A[1..n].
  - Funktion <u>IsWord</u>(i,j) stellt fest, ob A[i..j] ein Wort ist.
- Frage:
  - Kann A in Worte partitioniert werden?

BLAUBAUMEINHEITROBOTERHERZUNDSATURNDREHT

BLAU BAUM EINHEIT ROBOTER HERZ UND SATURN DREHT

Rekursiver Algorithmus

Prädikat Splittable(i) stellt fest, ob A[i..n] in Worte partitionierbar ist:

$$Splittable(i) = \begin{cases} \textsc{True} & \text{if } i > n \\ \bigvee_{j=i}^{n} \big(IsWord(i,j) \ \wedge \ Splittable(j+1) \big) & \text{otherwise} \end{cases}$$

EINHEITROBOTERHERZUND → EIN HEITROBOTERHERZUND

EINHEITROBOTERHERZUND → EINHEIT ROBOTERHERZUND

- **Zeit.** $T(n) \leq n\, T(n-1) = n\,(n-1)\, T(n-2) = \ldots = \Theta(n!)$

Memoisierung

$$Splittable(i) = \begin{cases} \text{TRUE} & \text{if } i > n \\ \bigvee_{j=i}^{n} \big(IsWord(i,j) \ \wedge \ Splittable(j+1) \big) & \text{otherwise} \end{cases}$$

“Wannimmer Splittable(i) berechnet wurde, speichere es in SplitTable[1…n+1] ab.”

Dynamische Programmierung

```
FastSplittable(A[1..n]):
SplitTable[n+1] = True
for i = n..1:
   SplitTable[i] = False
   for j = i..n:
      if IsWord(i,j) and SplitTable[j+1]:
         SplitTable[i] = True
return SplitTable[1]
```

- Zeit. $\Theta(n^2)$
- Platz. $O(n)$

Vorgehen bei Dynamischer Programmierung

1. Formuliere das Problem rekursiv!
   a. Spezifiziere präzise, **was** das rekursiv formulierte algorithmische Problem ist.
   b. Formalisiere eine rekursive Formel oder einen rekursiven Algorithmus, der das Problem auf kleinere Instanzen des exakt identischen Problems reduziert.

2. Baue die Lösung des Gesamtproblems aus den Lösungen der Teilprobleme.

3. Analysiere Platzverbrauch und Laufzeit.

Dynamische Programmierung

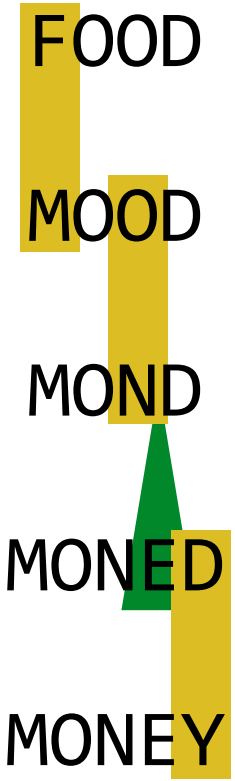
- Fibonacci-Folge
- Partition in Worte
- Edit Distance
- Teilmengensumme
- Dynamische Programmierung auf Bäumen

Edit Distance

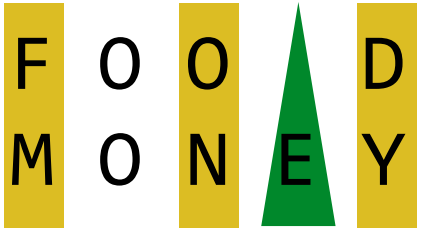
- Abstand zwischen Zeichenketten.

- Erlaubt sind:
  - **Einfügen** eines Buchstaben.
  - **Löschen** eines Buchstaben.
  - **Substituieren** eines Buchstaben.

Beispiel: FOOD → MONEY



Kompakte Visualisierung der Editiersequenz:

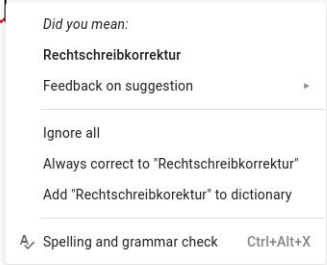


Abstand = Anzahl Spalten mit verschiedenen Einträgen.

⇒ Abstand von FOOD zu MONEY ist ≤4.

Anwendungen

- Bioinformatik. DNA-Sequenzalignment, Phylogenetische Bäume, RNA-Faltung, …
- Computerlinguistik. Spracherkennung, OCR, …
- Signalverarbeitung. Fehlerkorrektur, Kompression, ...
- Textverarbeitung. Versionskontrolle, Rechtschreibkorektur



Beispiel 2:

```
A L G O R   I   T H M
A L   T R U I S T I C
```

⇒ Abstand zwischen ALGORITHM und ALTRUISTIC ist höchstens 6.

Frage. Geht es besser?

Rekursive Struktur

	Präfix	Suffix
A[1..m]	ALGOR	I T H M
B[1..n]	ALTR	U I S T I C

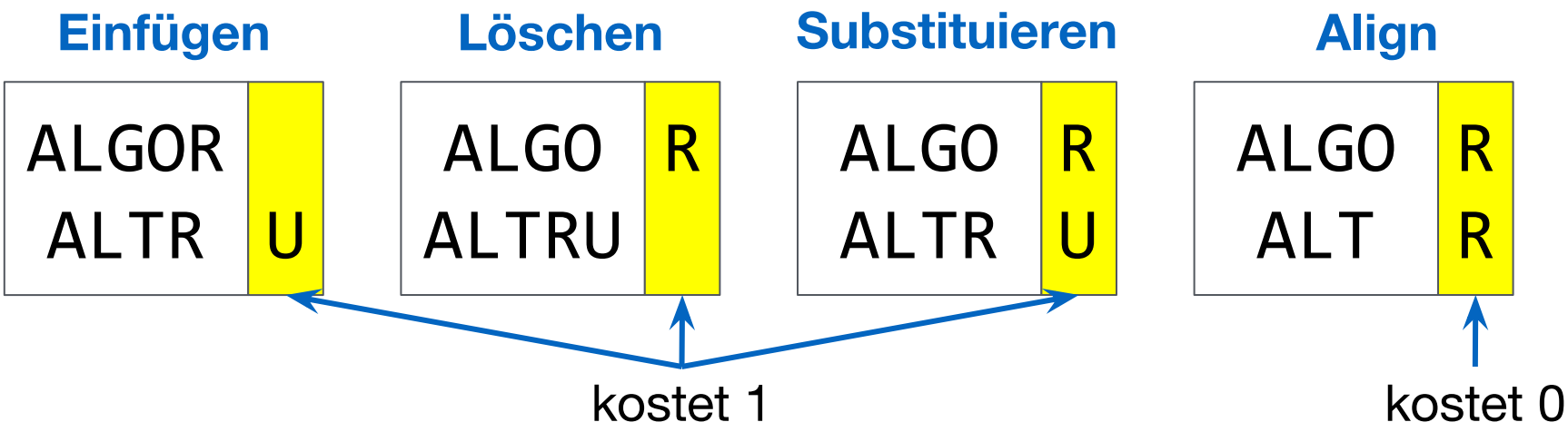
Beobachtung. Editiersequenzen von Präfix und Suffix sind unabhängig!

Das wollen wir rekursiv berechnen:

Edit(i,j) = edit distance von `A[1..i]` zu `B[1..j]`

Ziel: Berechne Edit(n,m)

Rekursive Formel



Rekursive Formel

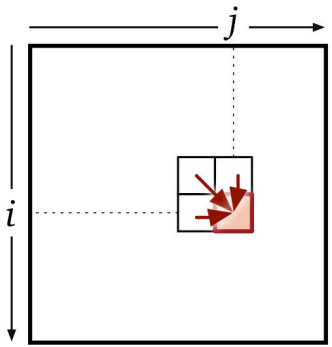
Einfügen		Löschen		Substituieren		Align	
ALGOR		ALGO	R	ALGO	R	ALGO	R
ALTR	U	ALTRU		ALTR	U	ALT	R

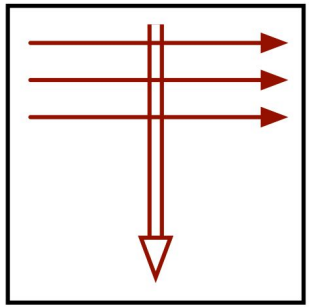
1. Basisfälle: Edit(0,j) = j und Edit(i,0) = i
2. Wenn A[i] = B[j]: Edit(i,j) = Edit(i–1, j–1)
3. Wenn A[i] ≠ B[j]:

$$\text{Edit}(i,j) = \min\{\text{Edit}(i,j-1), \text{Edit}(i-1,j), \text{Edit}(i-1,j-1)\} + 1$$

Dynamische Programmierung

- Teilprobleme. Das Paar (i,j) identifiziert ein Teilproblem.
- Memoisierung. Wir speichern bereits berechnete Werte Edit(i,j) in 2D-Feld Edit[1..m, 1..n].
- Abhängigkeiten. Eintrag Edit[i,j] hängt nur von drei Nachbarn ab.
- Ausführungsreihenfolge.





- Platz und Zeit.
  - Jeder Eintrag braucht O(1) Platz.
  - und ist aus den drei Vorgängern in O(1) Zeit berechnet.
  - Gesamt: **O(nm)** Platz und Zeit.

Beispiel

		A	L	G	O	R	I	T	H	M
	0	1	2	3	4	5	6	7	8	9
A	1	**0**	1	2	3	4	5	6	7	8
L	2	1	**0**	1	2	3	4	5	6	7
T	3	2	1	1	2	3	4	**4**	5	6
R	4	3	2	2	2	**2**	3	4	5	6
U	5	4	3	3	3	3	3	4	5	6
I	6	5	4	4	4	4	**3**	4	5	6
S	7	6	5	5	5	5	4	4	5	6
T	8	7	6	6	6	6	5	**4**	5	6
I	9	8	7	7	7	7	**6**	5	5	6
C	10	9	8	8	8	8	7	6	6	6

→ Einfügen

↓ Löschen

↘ Align

↘ Substitute

- Pfeil nur dann, wenn das Minimum durch diesen Fall erreicht wird.
- Wege von oben links nach unten rechts = **optimale Editiersequenz**.
- rote Pfeile kosten 0.

Beispiel

A L T R U I S T I C
A L G O R I T H M

		A	L	G	O	R	I	T	H	M
	0	1	2	3	4	5	6	7	8	9
A	1	0	1	2	3	4	5	6	7	8
L	2	1	0	1	2	3	4	5	6	7
T	3	2	1	1	2	3	4	4	5	6
R	4	3	2	2	2	2	3	4	5	6
U	5	4	3	3	3	3	3	4	5	6
I	6	5	4	4	4	4	3	4	5	6
S	7	6	5	5	5	5	4	4	5	6
T	8	7	6	6	6	6	5	4	5	6
I	9	8	7	7	7	7	6	5	5	6
C	10	9	8	8	8	8	7	6	6	6

→ Einfügen

↓ Löschen

↘ Align

↘ Substitute

- Pfeil nur dann, wenn das Minimum durch diesen Fall erreicht wird.
- Wege von oben links nach unten rechts = **optimale Editiersequenz**.
- rote Pfeile kosten 0.

Pseudocode

```
EditDistance(A[1..m], B[1..n]):
Setze Edit[i,0] = i und Edit[0,j] = j für alle i,j
for i = 1..m:
  for j = 1..n:
    if A[i] = B[j]:
      Edit[i,j] = Edit[i−1,j−1]  // Align
    else:
      Edit[i,j] =
        min{Edit[i,   j−1] + 1,      // Insert
            Edit[i−1, j  ] + 1,      // Delete
            Edit[i−1, j−1] + 1}      // Substitute
return Edit[m,n]
```

Dynamische Programmierung

- Fibonacci-Folge
- Partition in Worte
- Edit Distance
- Teilmengensumme
- Dynamische Programmierung auf Bäumen

Teilmengensumme (*Subset Sum*)

- Gegeben: Feld X[1..n] von natürlichen Zahlen und eine Zahl T.
- Problem: Finde eine Teilmenge der Zahlen in X mit Summe T.

T = 15

X = {8, **6**, 7, 5, 3, 10, **9**}

6+9=15

⇒Antwort ist `True`

T = 15

X = {11, 6, 5, 1, 7, 13, 12}

keine Teilmengensumme ist 15

⇒Antwort ist `False`

Teilmengensumme: Erschöpfende Suche

Für alle Teilmengen $S \subseteq \{1, \ldots, n\}$:

Teste ob $\sum_{i \in S} X[i] = T$ gilt.

- Laufzeit. 2^n Teilmengen, $O(n)$ arithmetische Operationen pro Teilmenge.
- Gesamtzeit: $O(2^n n)$ arithmetische Operationen
- Platz. $O(n)$

Dynamische Programmierung

- Hilfsfunktion:

  `S(i,t) = True` genau dann, wenn irgendeine Teilmenge von X[i..n] zu t summiert.

- rekursive Formel:

  `S(i,0) = True`

  `S(i,t) = False`, wenn t<0 oder i>n

  `S(i,t) = S(i+1,t) ∨ S(i+1, t-X[i])`, sonst

- Zeit und Platz: O(nT) Zeit und Platz.

Teilmengensumme: Übersicht

- Erschöpfende Suche: $O(2^n n)$ Zeit, $O(n)$ Platz
- Dynamische Programmierung: $O(nT)$ Zeit und Platz
- Wenn $T \leq 2^n$, dann ist dynamische Programmierung schneller, sonst nicht.
- Noch bessere Algorithmen: Es gibt randomisierte Algorithmen mit Laufzeit $O(n+T)$ (siehe Bringmann 2017)

Dynamische Programmierung

- Fibonacci-Folge
- Partition in Worte
- Edit Distance
- Teilmengensumme
- Dynamische Programmierung auf Bäumen

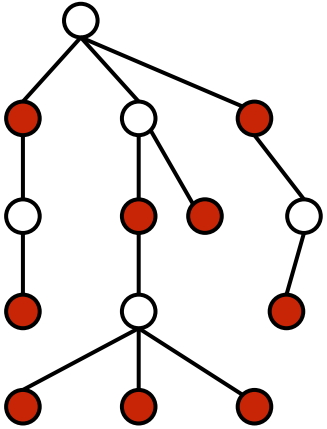
Größte unabhängige Menge (largest independent set)

- Unabhängige Menge. G ungerichteter Graph. Menge S von Knoten ist unabhängig, wenn S keine Kante enthält.

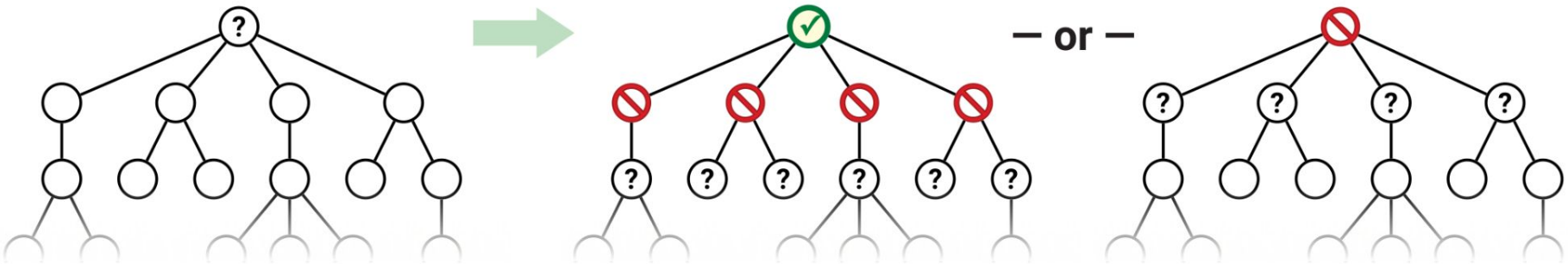
- Größte unabhängige Menge. S hat maximale Größe |S|.

- Größte unabhängige Menge auf Bäumen. Sei T ein Baum.

  MIS(v) = Größe |S| der größten unabhängige Menge S im Teilbaum mit Wurzel v.



Rekursive Idee



$$MIS(v) = \max\left\{ 1 + \sum_{w\downarrow v}\sum_{x\downarrow w} MIS(x)\,, \qquad \sum_{w\downarrow v} MIS(w) \right\}$$

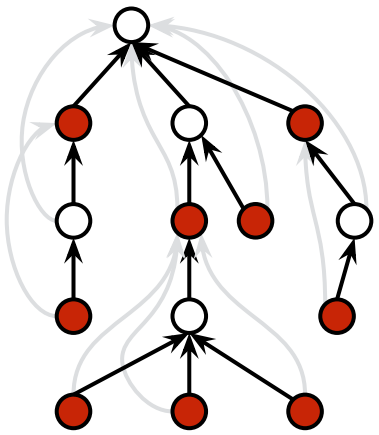
w↓v heißt hier, dass wir über alle Kinder w von v summieren.

Dynamische Programmierung

- Memoisierung. Sobald MIS(v) einmal berechnet wurde, merken wir uns den Wert `v.MIS` an jedem Knoten.

- Abhängigkeiten. `v.MIS` hängt von `u.MIS` ab für alle Kinder und Kindeskinder u von v.

- Reihenfolge. Wir füllen die Einträge `v.MIS` in post-order aus.



Pseudocode

```
TreeMIS(v):
skipv ← 0
for each child w of v
    skipv ← skipv + TreeMIS(w)

keepv ← 1
for each grandchild x of v
    keepv ← keepv + x.MIS

v.MIS ← max{keepv, skipv}
return v.MIS
```

Platz. Wir speichern O(1) Information an jedem Knoten ⇒ O(n) Gesamt.

Zeit.

- TreeMIS durchläuft den Baum in post-order.
- Jeder Knoten kommt einmal als Kind und einmal als Enkel vor.
- O(n) Laufzeit.

Dynamische Programmierung

- Fibonacci-Folge
- Partition in Worte
- Edit Distance
- Teilmengensumme
- Dynamische Programmierung auf Bäumen