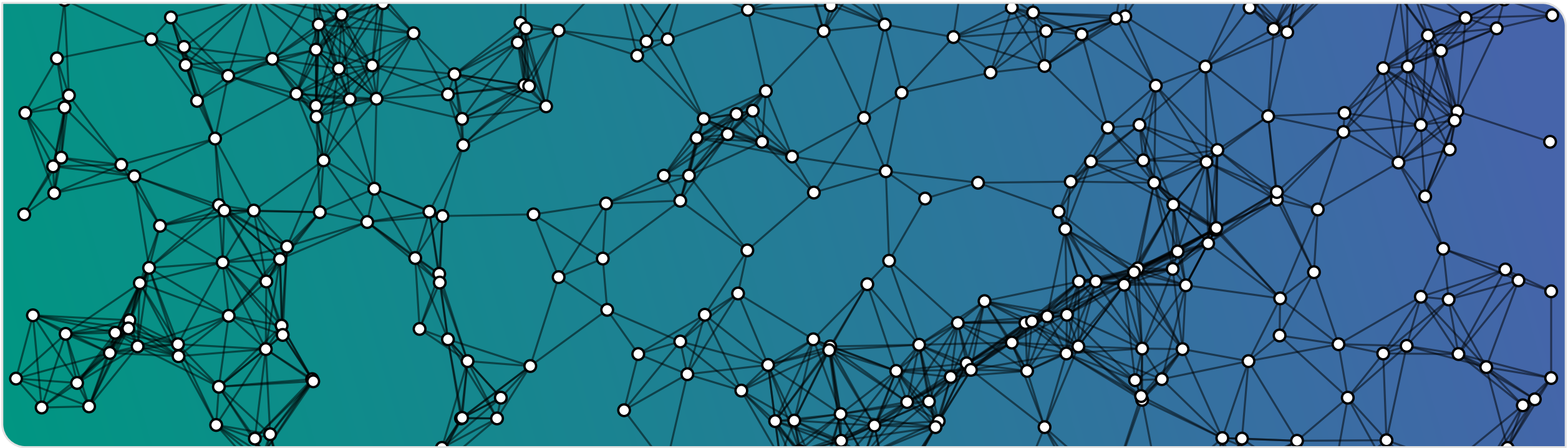
Karlsruher Institut für Technologie

Algorithmen 1

Dynamische Programmierung



Fibonacci-Zahlen

Definition: $F(1) = F(2) = 1,\ F(n) = F(n-1) + F(n-2)$ für $n \geq 3$ $\quad (1, 1, 2, 3, 5, 8, 13, 21, \ldots)$

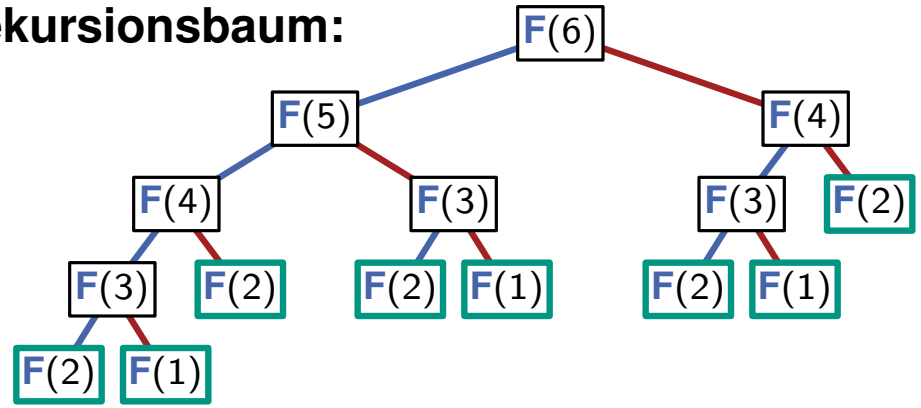
Fib(*Number* n)

```
F := Array of size n + 1
F[1] := 1
F[2] := 1
for i ∈ {3, ..., n} do
    F[i] := F[i − 1] + F[i − 2]
return F[n]
```

F(*Number* n)

```
if n = 1 or n = 2 then
    return 1
return F(n − 1) + F(n − 2)
```

Rekursionsbaum:



Laufzeit

- iterativ: $\Theta(n)$
- rekursiv: $\Theta(\varphi^n)$ mit $\varphi = \frac{1+\sqrt{5}}{2} \approx 1{,}6181$

(Rekurrenz lösen (nicht hier): $T(n) = T(n-1) + T(n-2) + \Theta(1)$)

Was ist hier passiert?

Rekursiv definiertes Problem

- reduziert Gesamtproblem auf kleinere Teilprobleme
  - Gesamtproblem: berechne $F(n)$; Teilprobleme: berechne $F(n-1)$ und $F(n-2)$
- Teilprobleme überlappen sich
  - für $F(n-1)$ muss man nochmal $F(n-2)$ berechnen
- rekursive Lösung: betrachtet selbes Teilproblem mehrfach

Iterative Lösung

- baue schrittweise größere Teillösungen auf
- Teillösungen speichern $\rightarrow$ keine mehrfache Berechnung

} meist recht einfach

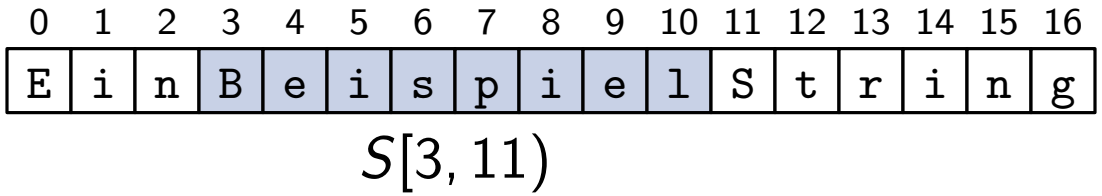
Und wenn das Problem gar nicht rekursiv definiert ist?

- finde eine äquivalente rekursive Definition

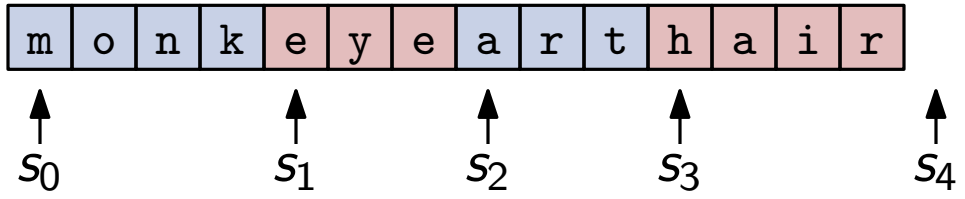
} das ist der schwierige Teil

Problemstellung: Splittability

Definition
Ein **String** S der Länge n ist eine Folge von Buchstaben $\langle S[0], \ldots, S[n-1] \rangle$. Die Teilfolge $\langle S[a], \ldots, S[b] \rangle$ bezeichnen wir auch mit $S[a, b] = S[a, b+1) = S(a-1, b] = S(a-1, b+1)$.



Problem: Splittability
Gegeben ein String S, gibt es Trennstellen $0 = s_0 < \cdots < s_k = n$, sodass $S[s_{i-1}, s_i)$ für alle $i \in [1, k]$ ein gültiges Wort ist? Wir sagen dann, dass S **splittable** ist.

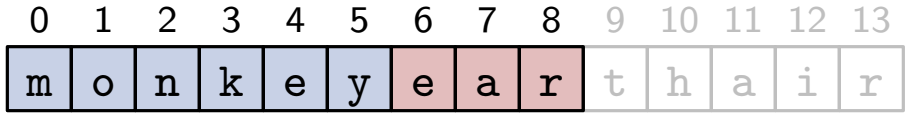


Teilprobleme und rekursive Definition

Problem: Splittability
Gegeben ein String S, gibt es Trennstellen $0 = s_0 < \cdots < s_k = n$, sodass $S[s_{i-1}, s_i)$ für alle $i \in [1, k]$ ein gültiges Wort ist? Wir sagen dann, dass S **splittable** ist.

Teilprobleme

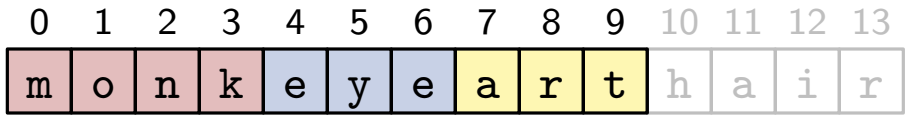
- für $i \in [0, n]$: Ist $S[0, i)$ splittable?
- beachte: Gesamtproblem fragt ob $S[0, n)$ splittable ist



Beispiel: $S[0, 9)$ ist splittable

Rekursive Definition auf Basis der Teilprobleme

- **splittable**$(0) = \texttt{true}$
- **splittable**$(i) = \exists j \in [0, i)$ **splittable**$(j) \wedge$ **isWord**$(S[j, i))$



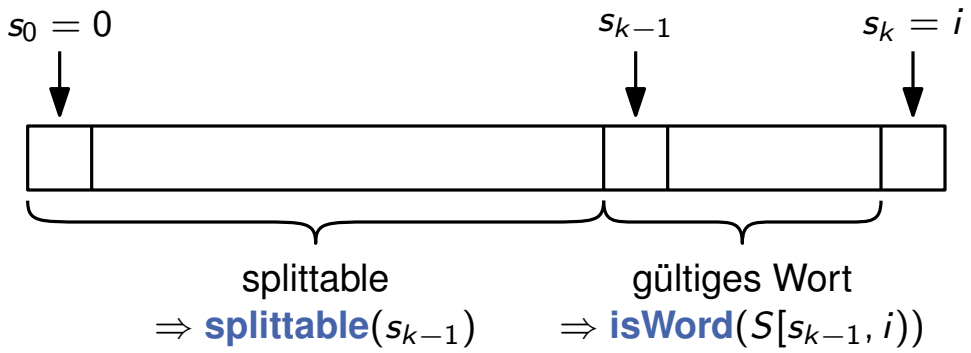
Beispiel:
splittable$(7) \wedge$ **isWord**$(S[7, 10)) = \texttt{true}$
$\Rightarrow$ **splittable**$(10) = \texttt{true}$

Korrektheit

Theorem
Der Teilstring $S[0, i)$ ist genau dann splittable, wenn **splitable**$(i) = \texttt{true}$.

Beweis: Induktion über i

- Induktionsanfang: der leere String ist splittable und **splitable**$(0) = \texttt{true}$
- Hinrichtung: $S[0, i)$ splittable $\Rightarrow$ **splittable**$(i) = \texttt{true}$
  - betrachte Lösung $0 = s_0 < \cdots < s_k = i$
  - dann ist $S[0, s_{k-1})$ splittable und $S[s_{k-1}, i)$ ein Wort
  - nach I.V. gilt dann **splittable**$(s_{k-1}) = \texttt{true}$ (und es gilt **isWord**$(s_{k-1}, i) = \texttt{true}$)



Erinnerung: Rekursive Definition von splittable

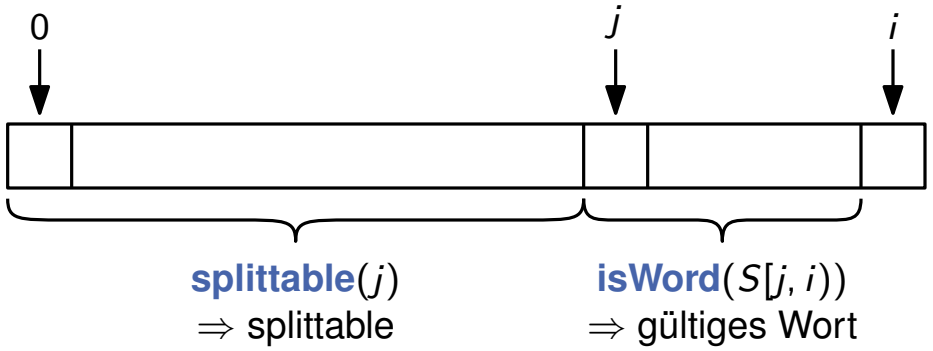
- **splittable**$(0) = \texttt{true}$
- **splittable**$(i) = \exists j \in [0, i)$ **splittable**$(j) \wedge$ **isWord**$(S[j, i))$

Korrektheit

Theorem
Der Teilstring $S[0, i)$ ist genau dann splittable, wenn **splitable**$(i) = \texttt{true}$.

Beweis: Induktion über i

- Induktionsanfang: der leere String ist splittable und **splitable**$(0) = \texttt{true}$
- Rückrichtung: $S[0, i)$ splittable $\Leftarrow$ **splittable**$(i) = \texttt{true}$
  - es gibt j, sodass **splittable**(j) und **isWord**$(S[j, i)$
  - nach I.V.: $S[0, j)$ ist splittable (außerdem ist $S[j, i)$ ein Wort)
  - damit ist auch $S[0, i)$ splittable



Erinnerung: Rekursive Definition von splittable

- **splittable**$(0) = \texttt{true}$
- **splittable**$(i) = \exists j \in [0, i)$ **splittable**$(j) \wedge$ **isWord**$(S[j, i))$

Iterative Implementierung

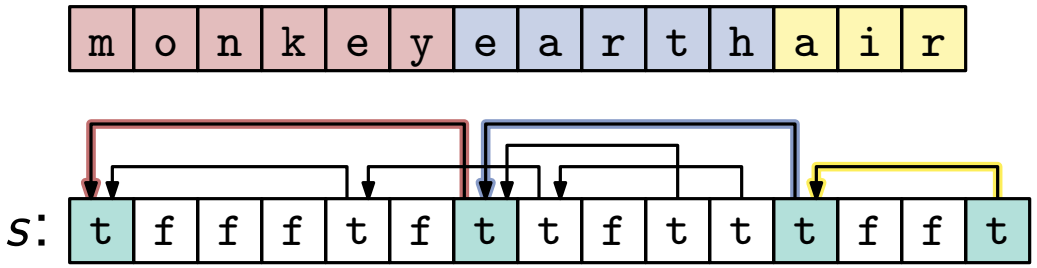
```
isSplittable(String S)
    s := Array of size n + 1
    s[0] := true
    for i ∈ {1, ..., n} do
        s[i] = false
        for j ∈ {0, ..., i − 1} do
            if s[j] and isWord(S[j, i)) then
                s[i] := true
    return s[n]
```

Laufzeit

- $\Theta(n^2)$ Aufrufe von **isWord** (also: polynomiell)

Bekommen wir auch eine Lösung?

- speichere für jedes i wegen welchem j wir $s[i] := \texttt{true}$ gesetzt haben
- verfolge diese Pointer rückwärts von n aus



Erinnerung: Rekursive Definition von splittable

- **splittable**$(0) = \texttt{true}$
- **splittable**$(i) = \exists j \in [0, i)$ **splittable**$(j) \wedge$ **isWord**$(S[j, i))$

In drei Schritten zum dynamischen Programm

Schritt 1: Spezifikation der Teilprobleme
- Welche Zwischenergebnisse wollen wir berechnen?
- Worüber geht die Induktion?
- Beispiel: Ist $S[0, i)$ splittable?

schwer
erfordert Kreativität
ggf. richtige Idee nötig
interagiert mit Schritt 2

Schritt 2: Aufstellung der Rekurrenz für Teillösungen
- Wie berechnen wir neue aus alten Zwischenergebnissen?
- Korrektheit: Induktion
- Beispiel: Lösung für i dank Lösungen für $0, \ldots, i-1$

nicht superschwer

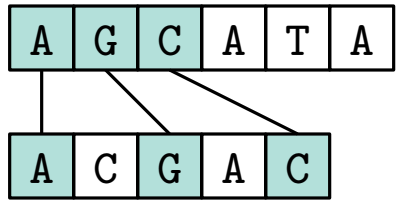
Schritt 3: Iterative Berechnung der Teillösungen
- Wie verwalten wir die Zwischenergebnisse?
- Wie hängen die Zwischenergebnisse voneinander ab?
- In welcher Reihenfolge berechnen wir Zwischenergebnisse?
- Wie bekommen wir die tatsächliche Lösung?

leicht
(mit etwas Übung)

Longest Common Subsequence

Problem: LCS
Gegeben Folgen $\langle a_1, \ldots, a_n \rangle$ und $\langle b_1, \ldots, b_m \rangle$, was ist die längste gemeinsame Teilfolge?

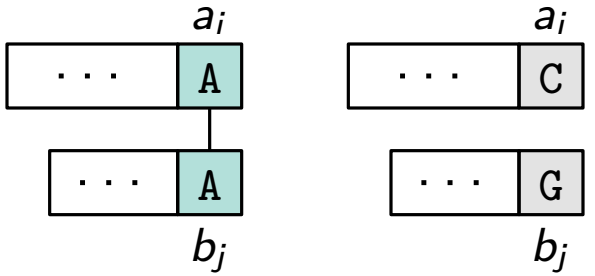
Schritt 1: Spezifikation der Teilprobleme



- Wie lang ist die LCS von $\langle a_1, \ldots, a_i \rangle$ und $\langle b_1, \ldots, b_j \rangle$?
- $i = n$ und $j = m$ liefert Gesamtlösung
- vereinfachte Sichtweise: nur Größe der Lösung (Berechnung der Lösung selbst heben wir uns für Schritt 3 auf)

Schritt 2: Aufstellung der Rekurrenz

- Ziel: **LCS**(i, j) bezeichnet Länge der LCS von $\langle a_1, \ldots, a_i \rangle$ und $\langle b_1, \ldots, b_j \rangle$
- **LCS**$(i, 0) =$ **LCS**$(0, j) = 0$
- $$\textbf{LCS}(i,j) = \begin{cases} \textbf{LCS}(i-1, j-1) + 1 & \text{wenn } a_i = b_j \\ \max\{\textbf{LCS}(i, j-1), \textbf{LCS}(i-1, j)\} & \text{wenn } a_i \neq b_j \end{cases}$$
- Korrektheit: folgt induktiv



Longest Common Subsequence: Schritt 3

Rekurrenz

- $\mathbf{LCS}(i,0) = \mathbf{LCS}(0,j) = 0$
- $\mathbf{LCS}(i,j) = \begin{cases} \mathbf{LCS}(i-1,j-1)+1 & \text{wenn } a_i = b_i \\ \max\{\mathbf{LCS}(i,j-1), \mathbf{LCS}(i-1,j)\} & \text{wenn } a_i \neq b_i \end{cases}$

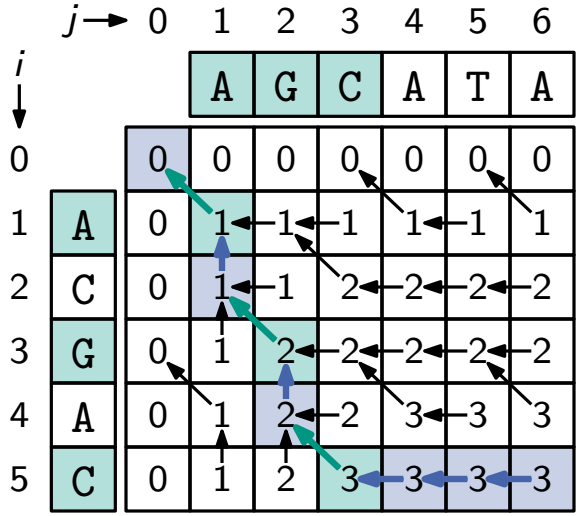
Schritt 3: Iterative Berechnung der Teillösungen

- Wie verwalten wir die Zwischenergebnisse?
- Wie hängen die Zwischenergebnisse voneinander ab?
- In welcher Reihenfolge berechnen wir Zwischenergebnisse?
- Wie bekommen wir die tatsächliche Lösung?

Antworten auf die Fragen von Schritt 3

- speichere für jedes $(i,j) \in [0,n] \times [0,m]$ eine Zahl
- $\rightarrow$ Array von Arrays
- Eintrag bei (i,j) hängt ab von Einträgen bei: $(i-1,j-1)$, $(i,j-1)$ und $(i-1,j)$
- von oben links nach unten rechts ausfüllen (Zeilenweise)
- zur Rekonstruktion der Lösung:
  - merke für jeden Eintrag, von wo er kommt
  - Rückverfolgung dieser Pointer $\rightarrow$ Lösung

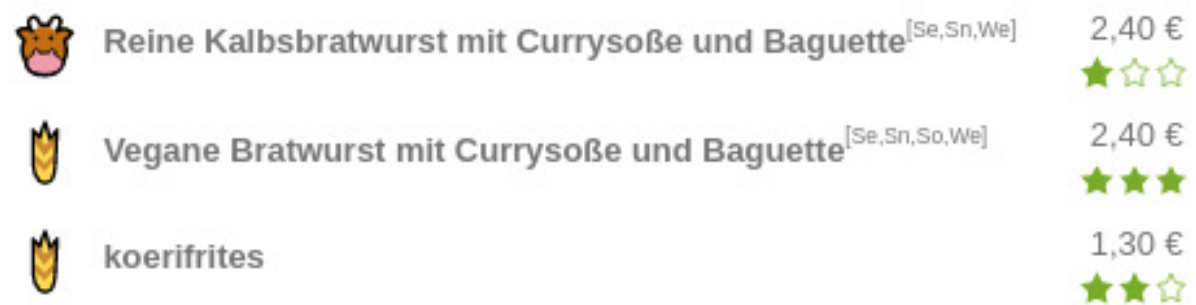
(jeder diagonale Schritt ist gemeinsames Zeichen)



Subset Sum

Situation

- ihr braucht kœri + Pommes (3,70€)
- ihr wollt gerne passend bezahlen



Problem: Subset Sum
Gegeben eine Multimenge $A = \{a_1, \ldots, a_n\}$ von natürlichen Zahlen und eine natürliche Zahl S. Gibt es eine Teilmenge $A' \subseteq A$, sodass $\sum_{a \in A'} a = S$?

Anmerkungen

- das ist in gewissem Sinne ein schweres Problem
- mehr dazu in TGI nächstes Semester
- aber: S nicht zu groß $\rightarrow$ effiziente Lösung mit DP

Subset Sum: Teilprobleme und Rekurrenz

Problem: Subset Sum Problem
Gegeben eine Multimenge $A = \{a_1, \ldots, a_n\}$ von natürlichen Zahlen und eine natürliche Zahl S. Gibt es eine Teilmenge $A' \subseteq A$, sodass $\sum_{a \in A'} a = S$?

Schritt 1: Spezifikation der Teilprobleme

- betrachte ein Paar (i, s) mit $i \leq n$ und $s \leq S$
- Gibt es Teilmenge $A' \subseteq \{a_1, \ldots, a_i\}$, sodass $\sum_{a \in A'} a = s$?
- $i = n$ und $s = S$ liefert Gesamtlösung

Schritt 2: Aufstellung der Rekurrenz

- Ziel: $\mathbf{SSP}(i, s) = \texttt{true}$ genau dann wenn es für das Paar (i, s) eine Lösung gibt
- $\mathbf{SSP}(0, s) = \texttt{true} \Leftrightarrow s = 0$ ($\texttt{false}$ sonst) und $\mathbf{SSP}(i, s) = \texttt{false}$ für $s < 0$
- $\mathbf{SSP}(i, s) = \underbrace{\mathbf{SSP}(i-1, s)}_{a_i \text{ nicht gewählt}} \vee \underbrace{\mathbf{SSP}(i-1, s-a_i)}_{a_i \text{ gewählt}}$

(Korrektheit folgt induktiv)

Subset Sum: Schritt 3

Rekurrenz

- $\mathrm{SSP}(0, s) = \texttt{true} \Leftrightarrow s = 0$
- $\mathrm{SSP}(i, s) = \texttt{false}$ für $s < 0$
- $\mathrm{SSP}(i, s) = \mathrm{SSP}(i-1, s) \vee \mathrm{SSP}(i-1, s-a_i)$

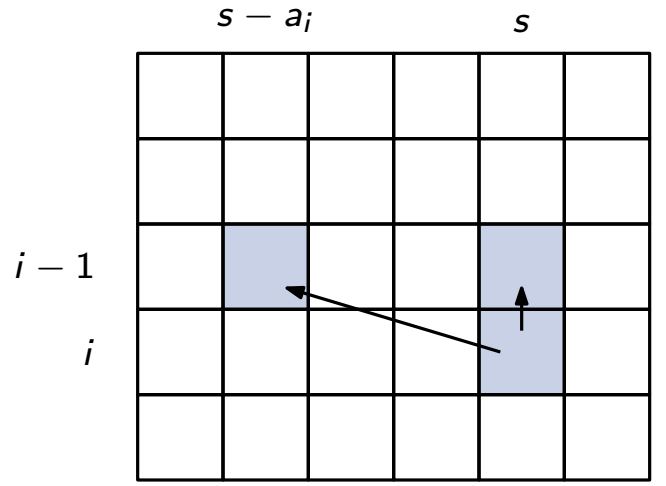
Schritt 3: Iterative Berechnung der Teillösungen

- Wie verwalten wir die Zwischenergebnisse?
- Wie hängen die Zwischenergebnisse voneinander ab?
- In welcher Reihenfolge berechnen wir Zwischenergebnisse?
- Wie bekommen wir die tatsächliche Lösung?

Antworten auf die Fragen von Schritt 3

- $\mathrm{SSP}(i, s) = \texttt{false}$ für $s < 0$ brauchen wir nicht explizit speichern
- für jedes $(i, s) \in [0, n] \times [0, S]$ ein boolescher Wert $\rightarrow$ Array von Arrays
- Einträge in Zeile i hängen von Einträgen in Zeile $i-1$ ab
- Reihenfolge: Tabelle zeilenweise ausfüllen
- zur Rekonstruktion der Lösung:
  - merke für jeden Eintrag, von wo er kommt
  - Rückverfolgung dieser Pointer $\rightarrow$ Lösung

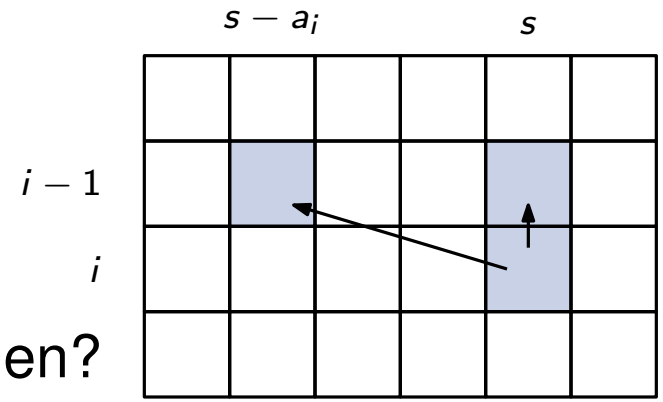
(jeder diagonale Schritt entspricht ausgewähltem Element)



Alternativer Sichtweise für Schritt 3: Pfadfindung

Rekurrenz

- $\mathrm{SSP}(0, s) = \texttt{true} \Leftrightarrow s = 0$
- $\mathrm{SSP}(i, s) = \texttt{false}$ für $s < 0$
- $\mathrm{SSP}(i, s) = \mathrm{SSP}(i-1, s) \vee \mathrm{SSP}(i-1, s - a_i)$

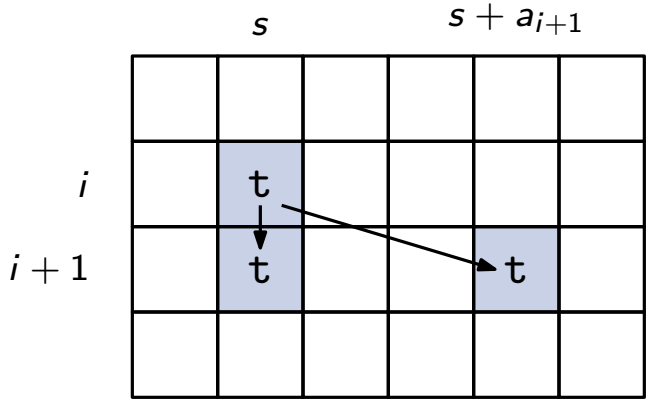


Bisher: Welche Einträge muss ich anschauen, um (i, s) auszufüllen?

Jetzt: Welche Einträge kann ich auf `true` setzen, wenn ich weiß, dass $\mathrm{SSP}(i, s) = \texttt{true}$?

Pfadfindung in einem Graphen

- fasse jedes Paar (i, s) als Knoten auf
- Kanten: $(i, s) \to (i+1, s)$ und $(i, s) \to (i+1, s + a_{i+1})$
- Ja-Instanz $\Leftrightarrow$ es gibt einen Pfad von $(0, 0)$ zu (n, S)
- beachte: Graph nur implizit repräsentiert
- Vorteil: wir besuchen in der Tabelle nur Zellen, die `true` sind



(ggf. deutlich weniger als alle Zellen)

In drei Schritten zum dynamischen Programm

Schritt 1: Spezifikation der Teilprobleme
- Welche Zwischenergebnisse wollen wir berechnen?
- Worüber geht die Induktion?
- Induktion über Parameter oder eine Baumstruktur

schwer
erfordert Kreativität
ggf. richtige Idee nötig
interagiert mit Schritt 2

Schritt 2: Aufstellung der Rekurrenz für Teillösungen
- Wie berechnen wir neue aus alten Zwischenergebnissen?
- Korrektheit: vollständige oder strukturelle Induktion

nicht superschwer
(vorausgesetzt ihr habt eine gute Spezifikation der Teilergebnisse)

Schritt 3: Iterative Berechnung der Teillösungen
- Wie verwalten wir die Zwischenergebnisse?
- Wie hängen die Zwischenergebnisse voneinander ab?
- In welcher Reihenfolge berechnen wir Zwischenergebnisse?
- Wie bekommen wir die tatsächliche Lösung?

leicht
(mit etwas Übung)

Dynamische Programmierung

DPs haben wir schon häufiger gesehen (ohne es zu wissen)

- Bellman–Ford
  - DP über die Anzahl Kanten aus denen die Pfade bestehen
  - Iteration i: berechne kürzeste Pfade unter allen Pfaden, die aus $\leq i$ Kanten bestehen
- Blatt 10, Aufgabe 1 – Dijkstras Angst-Gegner
  - berechne kürzesten Pfad in einem DAG
  - DP über topologische Sortierung der Knoten

Lernziel

- ihr habt verstanden, wie DPs funktionieren $\rightarrow$ schnelleres Verständnis unbekannter DPs
- ihr könnt zumindest einfache DPs selber bauen

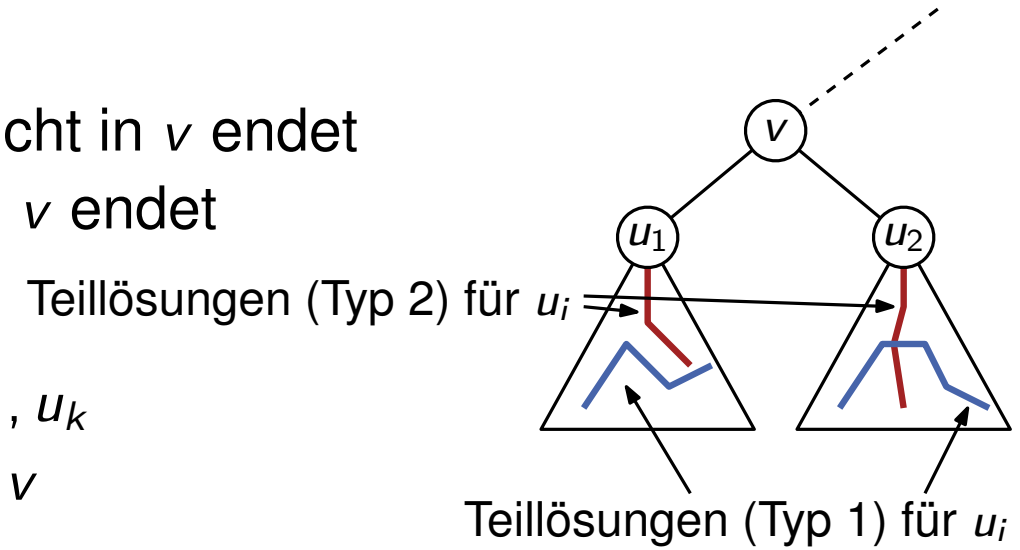
(insbesondere, wenn die Teillösungen gegeben sind)

Bonus: Längster Pfad im Baum

Problem: Longest Path
Gegeben einen (gewurzelten) Baum T. Wie lang ist der längste Pfad in T?

Schritt 1: Spezifikation der Teilprobleme

- strukturelle Induktion über Baumstruktur
- Teillösung (Typ 1) für v: längster Pfad in T_v, der nicht in v endet
- Teillösung (Typ 2) für v: längster Pfad in T_v, der in v endet



Schritt 2: Aufstellung der Rekurrenz

- erhalte Lösung für v aus Lösung für Kinder $u_1, \ldots, u_k$
- Ziel: $\mathbf{LP}_t(v)$ repräsentiert Teillösung von Typ t für v
- Falls v ein Blatt: $\mathbf{LP}_1(v) = \mathbf{LP}_2(v) = 0$
- $\mathbf{LP}_2(v) = \max_i\{\mathbf{LP}_2(u_i) + 1\}$
- $\mathbf{LP}_1(v) = \max\left[\max_i\{\max\{\mathbf{LP}_1(u_i), \mathbf{LP}_2(u_i)\}\},\ \max_{i\neq j}\{\mathbf{LP}_2(u_i) + \mathbf{LP}_2(u_j) + 2\}\right]$

Bonus: Längster Pfad im Baum: Korrektheit

Rekurrenz

- Falls v ein Blatt: $\mathbf{LP_1}(v) = \mathbf{LP_2}(v) = 0$
- $\mathbf{LP_2}(v) = \max_i\{\mathbf{LP_2}(u_i) + 1\}$
- $\mathbf{LP_1}(v) = \max\Big[\max_i\{\max\{\mathbf{LP_1}(u_i), \mathbf{LP_2}(u_i)\}\}, \max_{i\neq j}\{\mathbf{LP_2}(u_i) + \mathbf{LP_2}(u_j) + 2\}\Big]$

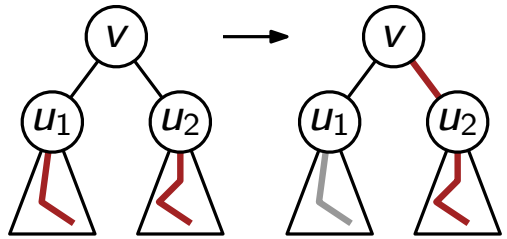
Was wollen wir zeigen?

- Ziel: $\mathbf{LP}_i(v)$ repräsentiert Teillösung von Typ i für v
- Typ 1: längster Pfad in Teilbaum T_v unter v
- Typ 2: längster Pfad in T_v, der in v endet

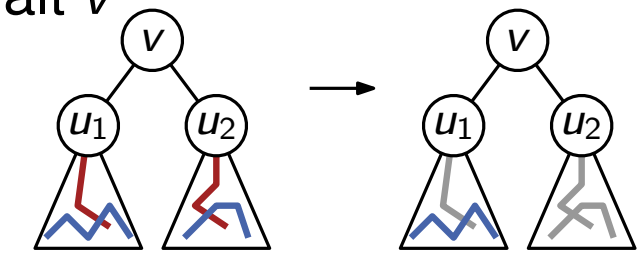
Korrektheit mittels struktureller Induktion

- korrekt für die Blätter
- korrekt für v falls korrekt für die Kinder $u_1, \ldots, u_k$
- **Typ 2:** Pfad führt über ein Kind u_i ($+1$ wegen $\{v, u_i\}$)
- **Typ 1, Fall 1:** längster Pfad in T_v enthält v nicht
- **Typ 1, Fall 2:** längster Pfad in T_v enthält v

$\mathbf{LP_2}(v) = \max_i\{\mathbf{LP_2}(u_i) + 1\}$

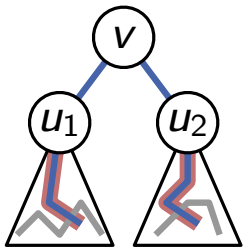


$\max_i\{\mathbf{LP_1}(u_i), \mathbf{LP_2}(u_i)\}$



$\max_{i\neq j}\{\mathbf{LP_2}(u_i) + \mathbf{LP_2}(u_j) + 2\}$

oder



Bonus: Längster Pfad im Baum: Schritt 3

Rekurrenz

- Falls v ein Blatt: $\mathbf{LP_1}(v) = \mathbf{LP_2}(v) = 0$
- $\mathbf{LP_2}(v) = \max_i\{\mathbf{LP_2}(u_i) + 1\}$
- $\mathbf{LP_1}(v) = \max\Big[\max_i\{\max\{\mathbf{LP_1}(u_i), \mathbf{LP_2}(u_i)\}\}, \max_{i\neq j}\{\mathbf{LP_2}(u_i) + \mathbf{LP_2}(u_j) + 2\}\Big]$

Schritt 3: Iterative Berechnung der Teillösungen

- Wie verwalten wir die Zwischenergebnisse?
- Wie hängen die Zwischenergebnisse voneinander ab?
- In welcher Reihenfolge berechnen wir Zwischenergebnisse?
- Wie bekommen wir die tatsächliche Lösung?

Antworten auf die Fragen von Schritt 3

- speichere zwei Werte an jedem Knoten v: $\mathbf{LP_1}(v)$ und $\mathbf{LP_2}(v)$
- Eltern hängen von Kindern ab $\rightarrow$ Reihenfolge: bottom-up (z.B. entsprechend der Lagen im BFS-Baum)
- tatsächliche Lösung berechnen: wie bei vorherigen DPs

Anmerkung

- Teillösungen überlappen nicht: Teillösung jedes Kinds u_i nur für den einen Elter v relevant
- rekursive Implementierung in dem Fall also auch ok (entspricht im Prinzip einer DFS)