Algorithmen & Datenstrukturen

Herbst 2019

Vorlesung 6

Dynamisches Programmieren (DP)

DP ist nichts anderes als Induktion

DP besteht aus 2 wesentlichen Komponenten:

1. Bottom-Up Berechnung von Rekurrenzen

Beispiel: Fibonacci-Zahlen

$$F_1 = F_2 = 1, \quad F_n = F_{n-1} + F_{n-2} \text{ für } n \geq 3$$

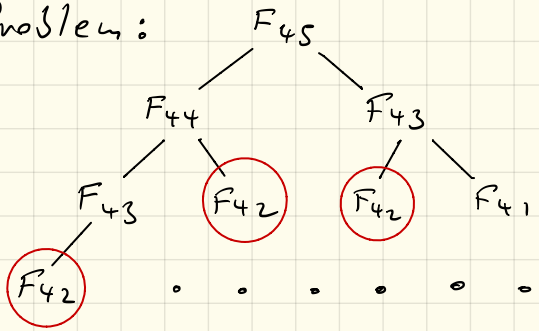
```
Fib(n)
    if n ≤ 2 : return 1
    f = Fib(n-1) + Fib(n-2)
    return f
```

Laufzeit: $T(n) = T(n-1) + T(n-2) + c$ (c: konstant)

$$\geq 2\,T(n-2)$$

Also $T(n) \geq \Omega(2^{n/2}) = \Omega(\sqrt{2}^n)$ sehr!

Problem:



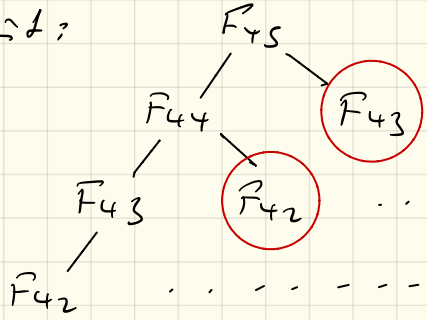
mehrfach berechnet

Idee: merken! (Memoization)

```
FibM(n)
  if n gespeichert: return memo[n]
  if n ≤ 2: f = 1
  else f = FibM(n-1) + FibM(n-2)
  memo[n] = f
  return f
```

Top-down Berechnung mit Memoization

Laufzeit:



schon berechnet!

$\Rightarrow$ Laufzeit $O(n)$!

Alternative: bottom-up Tabelle bauen

```
F[1] = 1
F[2] = 1
for i = 3..n: F[i] = F[i-1] + F[i-2]
```

Laufzeit: $O(n)$, Speicher: $O(n)$

Es geht auch mit Speicher $O(1)$
(nur letzte 2 merken)

Was hat das mit Algorithmen zu tun?
Es gibt Probleme die durch eine geeignete Rekurrenz induktiv gelöst werden

2. Design der Rekurrenz (Induktion)

Maximum Subarray noch mal

```
randmax = 0
max = 0
für i = 1..n
    randmax = max(0, randmax + A[i])
    max = max(randmax, max)
```

max(i): max der ersten i

randmax(i): randmax der ersten i

Algorithmus

Rekurrenz (Induktion): randmax(i-1) → randmax(i)
max(i-1) → max(i)

Der Induktionsschritt max(i-1) → max(i) braucht also randmax(i), was nicht sofort offensichtlich ist.

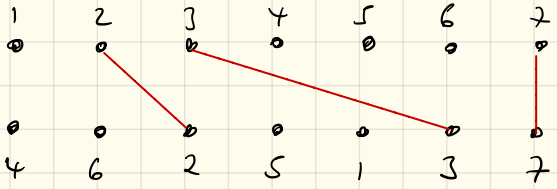
Bottom-up Tabelle:

i	0	1	2	3	4	5	...	n
randmax	0	→	→				....	
max	0	↓	↓					

○ Lösung

Aufgrund der Form der Rekurrenz muss man sich immer nur die aktuelle Spalte der Tabelle merken.

Längste aufsteigende Teilfolge



(Permutation)

Verbinde ohne Kreuzen
Maximiere Anzahl Verbindungen

Äquivalent: Finde längste aufsteigende Teilfolge in einem Array von n Zahlen

A[1] A[n]

z.B. 2 9 13 11 17 4 78 28 13 105

Wir nennen Lat(i) = längste aufsteigende TF in den ersten i Elementen

Designe Induktion!

Grundidee: Lat(i) = Lat(i-1) + A[n] anhängen falls möglich

1. Invariante: Wir haben Lat(i-1)

Fall 1: A[i] passt → Lat(i) ✓
Fall 2: A[i] passt nicht

Problem: Lat ist nicht eindeutig

Lat = 125

1	10	2	5	3	4

also brauchen wir mehr als Lat(i-1)

2. Invariante: Wir haben alle Lat(i-1)

Fall 1: A[i] passt an mindestens eine
→ alle Lat(i) durch Anhängen wo passt
Fall 2: A[i] passt an keine

Problem: es kann neue Lats geben also muss man sich auch Kürzere merken

1	2	8	5	3

Lats: 128 neues Lat 123
125

Damit müsste man sich alle aufsteigenden TFs merken → zu teuer

3. Invariante: Wir haben die Lat(i-1) die mit dem kleinsten Element aufhört

Fall 1: A[i] passt → Lat(i) (und erhält Bedingung)
Fall 2: passt nicht

1	2	8	7	6

Lat = 127 Lat wird besser: 126
→ Austausch notwendig

Also brauchen
wir auch die Kürzeren die mit dem kleinsten Element aufhören

4. Invariante: Wir haben für jede Länge die aufsteigende TF die mit dem kleinsten Element aufhört

Wenn man am Ende nur die Länge will genügt es sich die letzten Elemente zu merken:

gemerkt: (nach i-1 Iterationen)	Längen	1	2	3	4	5	....
	Endwerte	3	7	11	21	37	

ist aufsteigend sortiert (warum?)

A[i] = 8 → gibt update hier (warum kann 21 nicht upgedatet werden?)

Fall 1: A[i] passt ✓ (→ neue Länge)

Fall 2: passt nicht
senke den Endwert einer TF deren Endwert > A[i] und Endwert der eins kleineren TF < A[i]

Nur eine Änderung!

Beispiel: 4 9 8 13 10 11 7 3 16

Länge	1	2	3	4	5	6
Endwert	~~4~~ 3	~~9~~ 8	~~13~~ 10	11	16	

updates

7

Lösung: 4 8 10 11 16

⇒ Länge = 5

Um die Folge zu bekommen, merke Vorgänger von jedem Endwert (= Endwert zur Länge) in Extra array ⇒ Lösung durch Rückverfolgen.

Laufzeit: In jeder Iteration wird ein Element verändert, Stelle durch sinnvolle Suche

$$\Rightarrow T(n) = \sum_{i=1}^{n} c \log(i) = O(n \log n)$$

Speicher: $O(n)$ (Tabelle)

Längste gemeinsame Teilfolge

V O R L E S U N G T I G E R A[1..n]
L A P T O P Z I E G E B[1..m]
(nicht eindeutig)

Schreibe so hin dass man es sieht: (Alignment)

T I _ G E R LGT(n,m) = Länge LGT von
Z I E G E _ A[1..n] u. B[1..m]

Betrachte Ende: 4 Fälle

1. $\begin{matrix} X \\ - \end{matrix}$ $\Rightarrow LGT(n,m) = LGT(n-1,m)$

2. $\begin{matrix} - \\ X \end{matrix}$ $\Rightarrow LGT(n,m) = LGT(n,m-1)$

3. $\begin{matrix} X \\ Y \end{matrix}$, $X \neq Y$ $\Rightarrow LGT(n,m) = LGT(n-1,m-1)$

4. $\begin{matrix} X \\ X \end{matrix}$ $\Rightarrow LGT(n,m) = LGT(n-1,m-1)+1$

↳ also $A[n] = B[m]$

Also Induktion:

$$LCT(i,j) = \max(\ LCT(i-1,j),\ LCT(i,j-1),\ LCT(i-1,j-1) + 1 \text{ falls } A[i] = B[j]\ )$$

"top-down"

Basis:

$$LCT(0,\cdot) = LCT(\cdot,0) = 0$$

irgendwas

Berechnung "bottom-up" durch Füllen von Tabelle:

LCT	-	T	I	G	E	R
-	0	0	0	0	0	0
Z	0	0	0	0	0	0
I	0	0	1	1	1	1
E	0	0	1	1	2	2
G	0	0	1	2	2	2
E	0	0	1	2	3	3

↑ Länge

Lösung wieder durch Rückverfolgen

Merke von jedem Eintrag einen Vorgänger

Jedes Feld (i,j) mit $A[i] = B[j]$ gibt einen Buchstaben

Laufzeit: $O(mn)$, Speicher: $O(mn)$

Minimale Editierdistanz

Gegeben zwei Zeichenfolgen $A[1..n]$, $B[1..m]$

Editieroperationen:

- Zeichen einfügen
- Zeichen löschen
- Zeichen ändern

	T I G E R	3 Operationen (ist minimal)
ändern →	2 I G E R	
einfügen →	2 I E G E R	
löschen →	2 I E G E	

Gesucht: Minimale Anzahl Ops $A \longrightarrow B$.

Induktion: Betrachte weder letzte Elemente

$$ED(i,j) = \min\begin{cases} ED(i-1,j)+1 & \leftarrow A[i] \text{ löschen} \\ ED(i,j-1)+1 & \leftarrow B[j] \text{ hinzufügen} \\ ED(i-1,j-1)+1 \text{ wenn } A[i] \neq B[j] & \leftarrow A[i] \text{ durch } B[j] \text{ ersetzen} \end{cases}$$

$ED(0,i) = i$

$ED(j,0) = j$

Man muss sich noch genau überlegen dass diese Regel das Minimum produziert (Widerspruchsbeweis)

$A[1..n]$

ED	-	T	I	G	E	R
-	0	1	2	3	4	5
2	1	1	2	3	4	5
I	2	2	1	2	3	4
E	3	3	2	2	2	3
G	4	4	3	2	3	3
E	5	5	4	3	2	3

$B[1..m]$

↑ Länge

Lösung kann durch Rückverfolgen rekonstruiert werden.

- : $A[i]$ löschen
- | : $B[j]$ einfügen
- \ : nichts (wenn gleich) oder $A[i]$ durch $B[j]$ ersetzen

Laufzeit / Speicher: $O(mn)$