Algo-Tutorium 13

Infos

- DynProg: Gutes Video vom MIT: 19. Dynamic Programming I: Fibonacci, Shortest Paths – YouTube
- Klausur: Altklausuren FSI

Dynamische Programmierung – Idee

- häufig: rekursive Funktionen bzw. Lösungsverfahren
- Bsp.: Fibonacci-Zahlen:

$$f(n) = f(n-1) + f(n-2), f(0) = 0, f(1) = 1$$

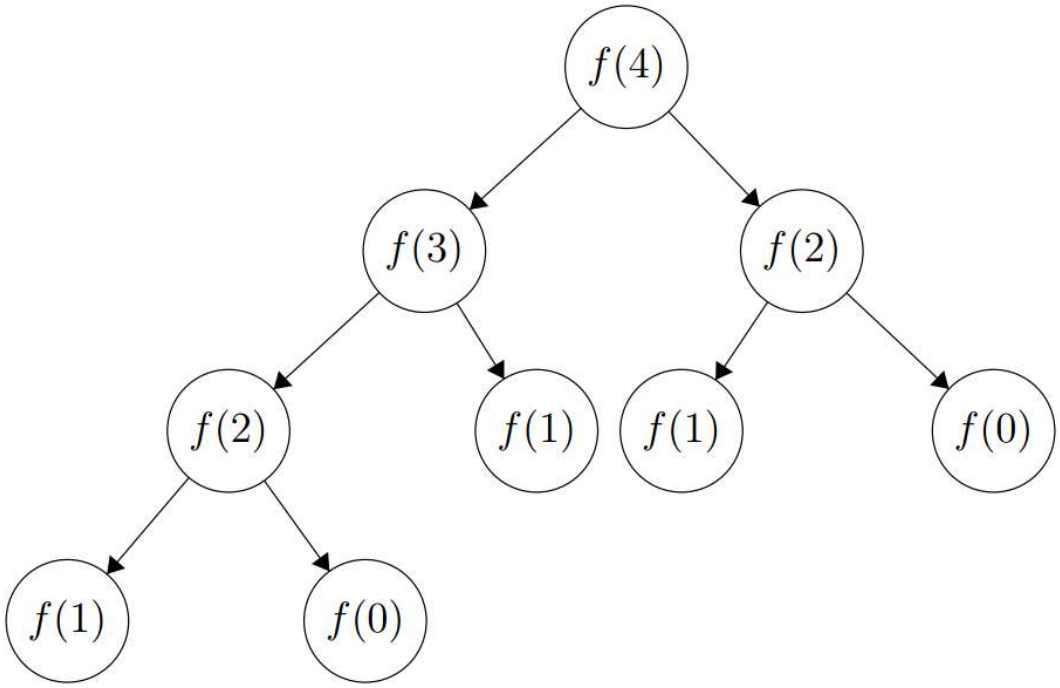
- berechenbar mit rekursivem Algorithmus:

Data: n
Result: $fib(n)$

if $n = 0 \mid n = 1$ **then**
 | **return** n
else
 | $fib(n) = fib(n-1) + f(n-2)$
end

Dynamische Programmierung – Idee

- aber: wir berechnen viele Ergebnisse doppelt, bspw. für f(4):



Dynamische Programmierung – Idee

- Mit dynamische Programmierung wollen wir die Zwischenergebnisse nutzen.
- Bottom-up bspw.:

Data: n
Result: $fib(n)$
initialization:
$A = newint \ \ [n+1]$
$A[0] = 0$
$A[1] = 1$
for $i = 2, \ldots, n$ **do**
$\quad A[i] = A[i-1] + A[i-2]$
end
return $A[n]$

DP – Allgemein

- **Konzept**: Vermeide mehrfache Berechnung von Zwischenergebnissen durch Speichern in geeigneter Datenstruktur. (oft: Matrizen/ Arrays)
- **Lösung** dann durch Zusammenfügen der Zwischenergebnisse
- Wichtig: **optimale Substruktur**. Diese beantwortet:
  - Wie sieht die Lösung der nächsten „Rekursionstiefe" aus?
  - Wie fügt man Teilergebnisse zusammen?
- **Design**: top-down oder bottom-up

! Oft berechnen wir numerische Ergebnisse. Die „Lösungs-Indizes" können wir aber durch „rückwärts" nachverfolgen unseres Rechenwegs rekonstruieren. (**Backtracking**)

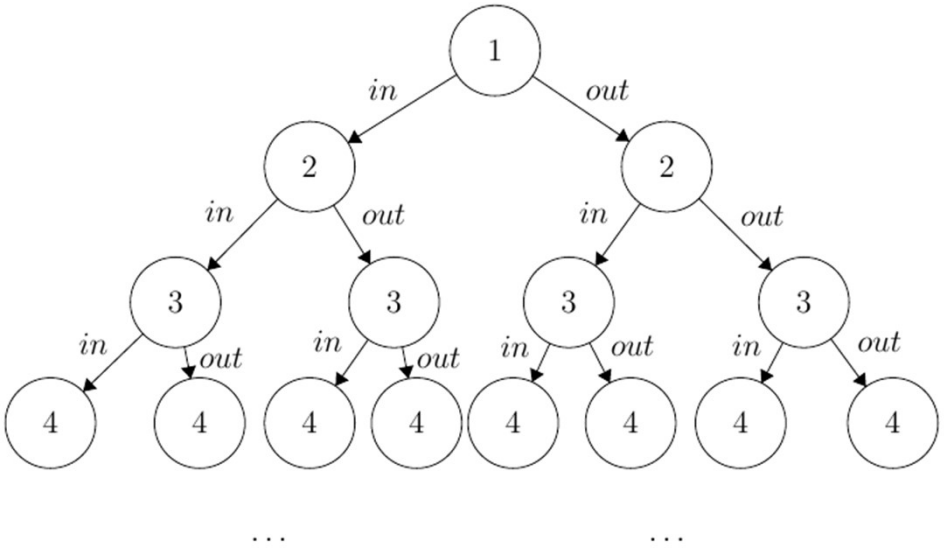
xx2

DP – Beispiel 1

Das Rucksackproblem:

Geg.: n Gegenstände $1 \leq i \leq n$ mit jeweils Wert v_i und Gewicht w_i; Rucksack R mit „Größe" (Maximalgewicht W)
Ges.: $max(\sum_{i \in R} v_i)$ unter Nebenbedingung $\sum_{i \in R} w_i \leq W$

Wo sind hier nun die überlappenden Teilprobleme?



Am Pfad entlang für Knoten j:
$\sum_{i=1}^{j} v_i$ berechnen
$\sum_{i=1}^{j} w_i \leq W$ prüfen

Ergebnis in v_n-Blatt mit max. Wert

Folie 7

xx2 xy xy; 10.02.2021

DP – Beispiel 1

- idR kann man die Problemstellungen binär erfassen und so die optimale Substruktur finden:

1. Gegenstand passt noch rein:
   a) Packe Gegenstand ein. Aktualisiere Restplatz in R.
   b) Packe Gegenstand nicht ein. Wir hoffen auf bessere Optionen ☺
2. Gegenstand passt nicht rein. Versuche den nächsten.

➢ über die Alternativen 1a), 1b) und 2. maximieren wir, da wir ein Max-Problem haben.

- es fehlt noch: die Datenstruktur
- Merken uns immer bisher optimales Ergebnis für Kombi aus Restplatz und Anzahl der bereits betrachteten Gegenstände:
  $T[i, w]$ = Wert für w-Restplatz und i-ten Gegenstand

DP – Beispiel 1

Input: $W \in \mathbb{N}$, v_i, w_i mit $1 \leq i \leq n$
Output: $max(\sum_{i \in R} v_i)$ s.t. $\sum_{i \in R} w_i \leq W$ (optimaler Wert)

Anlegen der Datenstruktur für Zwischenergebnisse:
$T[i,j] = int[n+1][W+1]$
Initialisierung für base case:
// für 0 Gegenstände kann der Wert von R nur 0 sein!
for $w = 0, \ldots, W$ **do**
  $T[0,w] = 0$
end
Berechnung:
for $i = 1, \ldots, n$ **do**
  for $w = 0, \ldots, W$ **do**
    if $w_i > w$ **then**
      $T[i,w] = T[i-1,w]$
    else
      $T[i,w] = max(T[i-1,w], T[i-1, w-w_i] + v_i)$
    end
  end
end

DP – Beispiel 2

Werbeeinnahmen maximieren

- Besitzer/in eines TV-Senders will Werbeeinnahmen maximieren
- Film hat $\geq n$ Minuten Länge
- Mögliche Einblendung bei Minute m_i erzielt Einnahme r_i
- ABER: Einblendungen müssen mind. 10min auseinander liegen

Aufgabe: Bestimme für $(m_1, \dots, m_n), (r_1, \dots, r_n)$

a) die maximal mögliche Einnahme

b) die Platzierungen für diese bestmögliche Einnahme

DP – Beispiel 2

Beispiel-Werte:

$m_i = (20, 25, 40, 42)$, $r_i = (5,6,5,1)$

- bestes Ergebnis?
- $\sum_{i \in m_{opt}} r_i = 11$ mit $m_{opt} = \{m_2, m_3\}$
- „greedy" funktionert offenbar nicht: immer beste Einnahme kollidiert manchmal mit Kriterium von 10min Pause
- sollten also immer wissen, welche Werbepunkte nacheinander kommen können (zusätzlich zur optimalen Substruktur)

DP – Beispiel 2

- Datenstruktur: $G[i]$ für Zwischenergebnis = optimale Einnahme bei Minute i
- Optimale Substruktur wieder binär:

1. Keine Werbung bei Minute m_i: $G[i] = G[i-1]$
2. Werbung bei Minute m_i: $G[i] = G[i'] + r_i$

(i': letzter erlaubter Werbeplatz)

- Zusammenfügen der Ergebnisse: $\max(G[i-1], G[i'] + r_i)$

DP – Beispiel 2

- Zuerst: müssen i' für ein gegebenes i wissen
- Letzten Index i' mit $m_{i'} \leq m_i - 10$ berechnen:

Input: $m_i, 1 \leq i \leq n$
Output: Array mit je größtem Index i' mit $m_{i'} \leq m_i - 10$

Anlegen der Datenstruktur für Indizes i' bzw. k:
$I[n] = int[n]$
for $i = 1, \ldots, n$ **do**
  $k = 1$;
  while $m_k \leq m_i - 10$ **do**
    $k++$
  end
  $I[i] = k - 1$
end

- Bemerkung: Das hat nicht unbedingt was mit DP zu tun. Manchmal müssen wir „normale" Zwischenberechnungen machen.

DP – Beispiel 2

können nun optimales Einnahmen-Ergebnis berechnen:

Input: $(m_1, \ldots, m_n), (r_1, \ldots, r_n), I[\,]$
Output: Optimale Werbeeinnahmen

Anlegen der Datenstruktur für Zwischenergebnisse:
$G[\,] = int[n+1]$ // *für 0, . . . ,n*
Basis-Fälle:
$G[0] = 0$
$G[1] = r_1$
for $i = 2, \ldots, n$ **do**
| $G[i] = max(G[i-1], G[I[i]] + r_i)$
end
return $G[n]$

DP – Beispiel 2

Nun noch Zeitpunkte berechnen: Wann wurde $G[i]$ verändert?

Input: $M[\,], I[\,]$
Output: Werbezeitpunkte P
$i = n$
$P = \emptyset$
while $i > 0$ **do**
  if $G[i] = G[i-1]$ **then**
    $i--$
  else
    $P = P \cup \{m_i\}$
    $i = I[i]$
  end
end
return P

PB

Aufgabe 1: Dynamisches Programmieren: Palindromlänge berechnen

— Vorbereitung auf Aufgabe 1 und 2 des Übungsblattes —

Ein *Palindrom* ist eine Zeichenfolge, die vorwärts und rückwärts gelesen gleich ist. Beispiele für Palindrome sind alle Zeichenfolgen der Länge 1, und die Zeichenfolgen `KAJAK`, `REITTIER`, `SEIFIES` und `DREHMALAMHERD`.

Gegeben sei eine Zeichenfolge $S[1..n]$. Eine *Teilfolge* S' von S besteht aus Buchstaben, die in S' in der selben Reihenfolge auftreten wie in S. Zum Beispiel ist `ATMEN` eine Teilfolge von `ALGORITHMEN`. Das *längste Palindrom* in S ist die längste Teilfolge von S, die ein Palindrom darstellt.

a) Ein trivialer Algorithmus für die Bestimmung der Länge des längsten Palindroms wäre es, für jede Teilsequenz zu prüfen, ob diese ein Palindrom ist. Schätzen Sie die Laufzeit für solch einen Ansatz grob ab.

b) Durch dynamische Programmierung kann man eine bessere Lösung finden. Geben Sie einen Algorithmus in Pseudocode an, der die Länge des längsten Palindroms in S berechnet. Verwenden Sie dabei Dynamische Programmierung.

c) Diskutieren Sie die Korrektheit und die Laufzeit Ihres Algorithmus aus a).

Aufgabe 2: Dynamisches Programmieren: Längste Palindrome finden

— Vorbereitung auf Aufgabe 1 und 2 des Übungsblattes —

In der letzten Übungsaufgabe haben Sie einen Algorithmus entwickelt, der die Länge des längsten Palindroms bestimmen kann. Bei dieser Berechnung werden auch Informationen generiert, durch die das längste Palindrom rekonstruiert werden kann. Wir möchten uns das hier an einem Beispiel veranschaulichen.

a) Demonstrieren Sie die Funktionsweise Ihres Algorithmus von Aufgabe 1 an der Zeichenfolge `DADBBCAA`.

b) Diskutieren Sie anhand des Beispiels aus a), wie sich anhand der Berechnungen des Algorithmus aus Aufgabe 1 das längste Palindrom rekonstruieren lässt.

Besprechung

Nr. 1
a)

- Es gibt 2^n Teilfolgen ⇒ Laufzeit $\Omega(2^n)$

Nr. 1
b)

- Sei $1 \leq i \leq j \leq n$ und $L[i,j]$ die Länge des längsten Palindroms in $S[i \ldots j]$

- Jede Teilsequenz der Länge 1 ist Palindrom

$$\Rightarrow L[i,i] = 1 \; \forall i \in \{1, \ldots, n\} \qquad (*)$$

- Teilsequenz der Länge 2 ist Palindrom, falls

$$S[i] = S[i+1] \; (i \in \{1, \ldots, n-1\}).$$

  Dann gilt $L[i, i+1] = 2$.

- Sonst (also $S[i] \neq S[i+1]$) ist $L[i, i+1] = 1$ wegen (*)

Nr. 1
b)

Restliche Längen (also $S[i \dots j]$ mit $i < j - 1$) zwei Fälle:

1. Erstes und letztes Zeichen verschieden. Dann gilt:

$$L[i,j] = \max\{L[i+1,j], L[i,j-1]\}$$

2. Erstes und letztes Zeichen gleich. Dann gilt:

$$L[i,j] = \max\{L[i+1,j], L[i,j-1], 2 + L[i+1,j-1]\}$$

Nr.1
b)

Input: Eine Zeichenkette S der Länge n
Output: Die Länge des längsten Palindroms in S

```
1  Initialisiere ein Array L der Größe n × n;
2  for k = 0, ..., n − 1 do
3  |  for i = 1, ..., n − k do
4  |  |  if k = 0 then
5  |  |  |  L[i, i] ⟵ 1;
6  |  |  else if k = 1 then
7  |  |  |  if S[i] = S[i + k] then
8  |  |  |  |  L[i, i + k] ⟵ 2;
9  |  |  |  else
10 |  |  |  |  L[i, i + k] ⟵ 1;
11 |  |  |  end
12 |  |  else
13 |  |  |  if S[i] = S[i + k] then
14 |  |  |  |  L[i, i + k] ⟵ max{L[i + 1, i + k], L[i, i + k − 1], 2 + L[i + 1, i + k − 1]};
15 |  |  |  else
16 |  |  |  |  L[i, i + k] ⟵ max{L[i + 1, i + k], L[i, i + k − 1]};
17 |  |  |  end
18 |  end
19 end
20 return L[1, n];
```

Nr. 1
c)

- **Korrektheit:**
  - Beide initialen Fälle (Teilstringlänge 1 bzw. 2) offensichtlich korrekt
  - Anderen Fälle bilden Konstruktion von größeren Palindromen ab

- **Laufzeit:**
  - Von for-Schleifen bestimmt
  - $O(n^2)$

Nr. 2
a)

Alle Länge 1 Teilsequenzen:

	1(D)	2(A)	3(D)	4(B)	5(B)	6(C)	7(A)	8(A)
1(D)	1							
2(A)		1						
3(D)			1					
4(B)				1				
5(B)					1			
6(C)						1		
7(A)							1	
8(A)								1

Alle Länge 2 Teilsequenzen:

	1(D)	2(A)	3(D)	4(B)	5(B)	6(C)	7(A)	8(A)
1(D)	1	1						
2(A)		1	1					
3(D)			1	1				
4(B)				1	2			
5(B)					1	1		
6(C)						1	1	
7(A)							1	2
8(A)								1

Nr. 2
a)

Alle Länge 3 Teilsequenzen:

	1(D)	2(A)	3(D)	4(B)	5(B)	6(C)	7(A)	8(A)
1(D)	1	1	3					
2(A)		1	1	1				
3(D)			1	1	2			
4(B)				1	2	2		
5(B)					1	1	1	
6(C)						1	1	2
7(A)							1	2
8(A)								1

Alle Länge 4 Teilsequenzen:

	1(D)	2(A)	3(D)	4(B)	5(B)	6(C)	7(A)	8(A)
1(D)	1	1	3	3				
2(A)		1	1	1	2			
3(D)			1	1	2	2		
4(B)				1	2	2	2	
5(B)					1	1	1	2
6(C)						1	1	2
7(A)							1	2
8(A)								1

Nr. 2
a)

Alle Länge 5 Teilsequenzen:

	1(D)	2(A)	3(D)	4(B)	5(B)	6(C)	7(A)	8(A)
1(D)	1	1	3	3	3			
2(A)		1	1	1	2	2		
3(D)			1	1	2	2	2	
4(B)				1	2	2	2	2
5(B)					1	1	1	2
6(C)						1	1	2
7(A)							1	2
8(A)								1

Alle Länge 6 Teilsequenzen:

	1(D)	2(A)	3(D)	4(B)	5(B)	6(C)	7(A)	8(A)
1(D)	1	1	3	3	3	3		
2(A)		1	1	1	2	2	4	
3(D)			1	1	2	2	2	2
4(B)				1	2	2	2	2
5(B)					1	1	1	2
6(C)						1	1	2
7(A)							1	2
8(A)								1

Nr. 2
a)

Alle Länge 7 Teilsequenzen:

	1(D)	2(A)	3(D)	4(B)	5(B)	6(C)	7(A)	8(A)
1(D)	1	1	3	3	3	3	4	
2(A)		1	1	1	2	2	4	4
3(D)			1	1	2	2	2	2
4(B)				1	2	2	2	2
5(B)					1	1	1	2
6(C)						1	1	2
7(A)							1	2
8(A)								1

Alle Länge 8 Teilsequenzen:

	1(D)	2(A)	3(D)	4(B)	5(B)	6(C)	7(A)	8(A)
1(D)	1	1	3	3	3	3	4	4
2(A)		1	1	1	2	2	4	4
3(D)			1	1	2	2	2	2
4(B)				1	2	2	2	2
5(B)					1	1	1	2
6(C)						1	1	2
7(A)							1	2
8(A)								1

Nr. 2
b)

Backtracking in Zahlenwerten:

- Zahl an Stelle $[i, j]$
  - entweder so groß wie Zahl an Stelle $[i+1, j]$ bzw. $[i, j-1]$
  - oder zwei größer als Zahl an Stelle $[i+1, j-1]$ und $S[i] = S[j]$
- Zahl an Stelle $[1, n]$ muss so entstanden sein. Es ergibt sich eine 0:

	1(D)	2(A)	3(D)	4(B)	5(B)	6(C)	7(A)	8(A)
1(D)	1	1	3	3	3	3	4	4
2(A)		1	1	1	2	2	4	4
3(D)			1	1	2	2	2	2
4(B)				1	2	2	2	2
5(B)				0	1	1	1	2
6(C)						1	1	2
7(A)							1	2
8(A)								1

- Bei Werterhöhung werden 2 Buchstaben hinzugefügt
- Möglichkeiten für Palindrom ABBA an Stellen 2,4,5,7 und 2,4,5,8

Klausur

- Letzte Übungsstunde = Fragestunde
- Falls ihr Fragen habt, schickt mir diese bitte bis nächste Woche Donnerstag 3.2.22 16 Uhr.