technische universität dortmund

SoSe 2017

C. Sohler
A. Krivošija, A. Rey, H. Sandvoß

DAP2 – Präsenzübung 6

Besprechung: 31.05.2017 — 02.06.2017

Präsenzaufgabe 6.1: (Dynamische Programmierung)

In der Vorlesung wurde ein dynamisches Programm zur Maximumssuche besprochen. Des Weiteren wurde eine Idee angegeben, wie man den Index eines größten Elements in einem Array A der Länge n dynamisch bestimmen kann (siehe Foliensatz 8, Folie 58):

Sei

$$\mathrm{Max}(i) = \begin{cases} -\infty & \text{falls } i = 0 \\ A[1] & \text{falls } i = 1 \\ \max\{\mathrm{Max}(i-1), A[i]\} & \text{sonst} \end{cases}$$

die rekursive Funktion, die für ein i, $1 \leq i \leq n$, das Maximum $\max_{i \leq j \leq i}\{A[j]\}$ des Arrays bis zur Stelle i berechnet.

Der Index j des maximalen Elements ist der größte Wert, für den $\mathrm{Max}(j) > \mathrm{Max}(j-1)$ gilt.

Untersuchen Sie dies wie folgt.

a) Geben Sie ein dynamisches Programm in Pseudocode an, das basierend auf diesen Überlegungen einen Index eines größten Elements in einem Array ausgibt.

b) Analysieren Sie die Laufzeit Ihres Algorithmus.

c) Beweisen Sie die Aussage, dass der größte Wert j mit $\mathrm{Max}(j) > \mathrm{Max}(j-1)$ der gesuchte Index des Maximums ist.

Lösung:

a) Der folgende Algorithmus berechnet zunächst (wie in der Vorlesung) die aktuellen Maxima und daraus den gesuchten Index.

MaxIndex(Array A):

$n \leftarrow \mathrm{length}(A)$
new array Max[1..n]
$\mathrm{Max}[1] \leftarrow A[1]$ ➢ `vgl. MaxDynamic aus VL`
for $i \leftarrow 2$ **to** n **do**
    $\mathrm{Max}[i] \leftarrow \max\{\mathrm{Max}[i-1], A[i]\}$
$j = n$
while $j > 1$ *and* $\mathrm{Max}[j] = \mathrm{Max}[j-1]$ **do**
    $j \leftarrow j - 1$
return j

b) Die Laufzeit des Algorithmus hängt von der Eingabegröße $n = \text{length}(A)$ ab. Zeilen 1 bis 5 benötigen wie in der Vorlesung gesehen eine Laufzeit in $\mathcal{O}(n)$. Zeilen 6 und 9 benötigen konstante Laufzeit. Die Schleife in Zeilen 7 und 8 wird im Worst Case n mal mit jeweils einer konstanten Anzahl an Schritten durchlaufen. Es ergibt sich also insgesamt eine Laufzeit in $\mathcal{O}(n)$.

c) Zu zeigen: Der größte Wert j mit $\text{Max}(j) > \text{Max}(j-1)$ ist der gesuchte Index.

Wir beobachten zunächst, dass Max schwach monoton steigend ist:

$$\text{Max}(i_1) \leq \text{Max}(i_2) \quad \text{für alle } i_1 < i_2. \qquad (*)$$

Das können wir induktiv über i_2 zeigen: Für $i_2 = 1$ haben wir nach der Konvention $\text{Max} = -\infty < \text{Max}(1) = A[1]$. Gelte nun die Aussage für ein festes i_2 und alle $i_1 < i_2$, dann gilt $\text{Max}(i_2+1) = \max\{\text{Max}(i_2), A[i_2+1]\} \geq \text{Max}(i_2)$, das wiederum nach Voraussetzung größer oder gleich $\text{Max}(i_1)$ für alle i_1, $1 \leq i_1 < i_2$, ist.

Sei j nun der größte Index mit $\text{Max}(j) > \text{Max}(j-1)$. Der gesuchte Index j_0 kann nicht kleiner sein als j, denn

$$\text{Max}(j_0) \underset{(*)}{\leq} \text{Max}(j-1) < \text{Max}(j) \underset{\text{Def.}}{=} \max\{\text{Max}(j-1), A[j]\} = A[j].$$

Es muss j_0 also mindestens so groß wie j sein.

Angenommen, j_0 ist größer als j, d. h., an Stelle j_0 steht ein größerer Wert $A[j_0] > \text{Max}(j)$. Dann folgt

$$\text{Max}(j_0) \underset{\text{Def.}}{=} \max\{\text{Max}(j_0-1), A[j_0]\} \geq A[j_0] > \text{Max}(j).$$

D. h., es gibt einen Index $j' > j$ mit $\text{Max}(j') > \text{Max}(j'-1)$, was ein Widerspruch dazu ist, dass j der größte solche Index ist. Also ist j selbst der gesuchte Index. $\square$

Präsenzaufgabe 6.2: (Dynamische Programmierung)

Betrachten Sie das folgende Partitionierungs-Problem: Gegeben eine Menge $A = \{a_1, \ldots, a_m\}$ bestehend aus natürlichen Zahlen, gibt es eine Aufteilung der Zahlen in drei Teilmengen, sodass die Summe der Elemente jeweils gleich groß ist?

a) Formulieren Sie eine geeignete rekursive Form für dieses Problem.

b) Geben Sie einen Algorithmus in Pseudocode an, der das Gesamtproblem basierend auf der rekursiven Beziehung aus Teilaufgabe a) mit dynamischer Programmierung löst.

c) Analysieren Sie die Laufzeit Ihres Algorithmus.

d) Beweisen Sie die Korrektheit der in Teilaufgabe a) angegebenen rekursiven Form.

Lösung:

a) Wie beim Problem PARTITION, lässt sich auch hier die Frage äquivalent umformen zu: Gibt es zwei disjunkte Teilmengen $B, C \subseteq A$ mit $\sum_{x\in B} x = \sum_{x\in C} x = W/3$, $W = \sum_{x\in A} x$? Wir können also hier ähnlich vorgehen mit dem Unterschied, dass es für jedes Element drei Alternativen gibt: Es gehört zu B, es gehört zu C oder zum Rest. Es sei $E(i, j_1, j_2) = 1$, falls man die Zahl j_1 als Summe aus einer Teilmenge S aus den Zahlen in $\{a_1, \dots, a_i\}$ und j_2 als Summe aus den Zahlen in $\{a_1, \dots, a_i\} \setminus S$ darstellen kann; es sei $E(i, j_1, j_2) = 0$ sonst. Es ergibt sich folgende rekursive Form:

$$E(i,j_1,j_2) = \begin{cases} 1 & \text{falls } j_1 = j_2 = 0 \\ 0 & \text{falls } i = 0, j_1 > 0 \text{ oder } j_2 > 0 \\ E(i-1, j_1, j_2) & \text{falls } i > 0, j_1 < a_i \text{ und } j_2 < a_i \\ \max\{E(i-1,j_1,j_2), E(i-1, j_1 - a_i, j_2)\} & \text{falls } i > 0, j_1 \geq a_i \text{ und } j_2 < a_i \\ \max\{E(i-1,j_1,j_2), E(i-1, j_1, j_2 - a_i)\} & \text{falls } i > 0, j_1 < a_i \text{ und } j_2 \geq a_i \\ \max\{E(i-1,j_1,j_2), E(i-1, j_1 - a_i, j_2), & \\ \qquad\qquad E(i-1, j_1, j_2 - a_i)\} & \text{falls } i > 0, j_1 \geq a_i \text{ und } j_2 \geq a_i. \end{cases}$$

d) Wir wollen nun zeigen, dass die in Aufgabenteil a) angegebene rekursive Form korrekt ist, also für alle i, $0 \leq i \leq n$, und alle j_k, $0 \leq j_k \leq W/3$, $k \in \{1, 2\}$, $E(i, j_1, j_2)$ genau dann 1 ist, wenn sich die Zahl j_1 als Summe aus einer Teilmenge S aus den Zahlen in $\{a_1, \dots, a_i\}$ und j_2 als Summe aus den Zahlen in $\{a_1, \dots, a_i\} \setminus S$ darstellen lässt.

Wir zeigen dies mittels Induktion über i.

Induktionsanfang Falls $i = 0$ und $j_1 = j_2 = 0$ sind, können beide Zahlen als Summe über die leere Menge dargestellt werden. Sobald aber eine der beiden Zahlen j_1 oder j_2 nicht mehr dargestellt werden.

Induktionsvoraussetzung Für ein festes $i_0 \geq 0$ und für alle $j_1, j_2 \geq 0$ sei $E(i_0, j_1, j_2)$ korrekt.

Induktionsschritt Wir betrachten $i = i_0 + 1 > 0$. Falls $j_1 < a_i$ ist, kann a_i nicht in der ersten Teilsumme enthalten sein. Anderenfalls ist es möglich, dass a_i in der ersten Teilsumme enthalten ist oder nicht. Falls $j_2 < a_i$ ist, kann a_i nicht in der zweiten Teilsumme entalten sein. Anderenfalls ist es möglich, dass a_i in der zweiten Teilsumme enthalten ist oder nicht. Daraus ergeben sich die vier oben angegebenen Fälle, wobei die Zahl a_i entweder in keiner oder einer der beiden Teilsummen enthalten sein kann. Um zu entscheiden, ob das Problem insgesamt lösbar ist, kann man also auf den vorherigen Wert i_0 und $j_1 - a_i$ bzw. $j_2 - a_i$ zugreifen. Genauer:

- Falls a_i größer als j_1 und j_2 ist, muss a_i im Rest enthalten sein, um eine gültige Partitionierung zu erlauben. Dies ist also genau dann möglich, wenn die Aufteilung für a_1 bis a_{i-1} und den Grenzen j_1 und j_2 möglich ist.
- Falls a_i nicht größer als eins der beiden, etwa j_1, ist, aber größer als das andere, j_2, gibt es nur zwei Möglichkeiten: a_i ist Teil der ersten Teilsumme, dann ist die Entscheidung äquivalent dazu, ob es eine Aufteilung von $a_1, \dots, a_{i-1}$ für j_2 und $j_1 - a_i$ gibt; oder a_i ist nicht Teil der ersten Teilsumme, sondern des Rests, dann ist die Entscheidung wieder äquivalent zu Entscheidung $E(i-1, j_1, j_2)$.
- Für $a_i > j_1$ und $a_i \leq j_2$ gelten analoge Argumente.

– Falls a_i kleiner oder gleich j_1 und j_2 ist, gibt es drei Möglichkeiten, a_i unterzubringen. Also gibt es eine gültige Aufteilung, falls sich $a_1, \ldots, a_{i-1}$ in j_1 und j_2, in $j_1 - a_i$ und j_2 oder in j_1 und $j_2 - a_i$ und jeweils einen Rest aufteilen lassen.

Diese Entscheidungen $E(i-1, j_1, j_2)$, $E(i-1, j_1 - a_i, j_2)$ oder $E(i-1, j_1, j_2 - a_i)$ sind nach Induktionsvoraussetzung korrekt. Falls eine der in Frage kommenden Entscheidungen 1 ergibt, ist das Maximum 1 und somit der aktuelle Wert korrekt. □

b) Folgender Algorithmus entscheidet die Lösung des Gesamtproblems bei einer Eingabe der Menge A in Form eines Arrays, indem er $E[n][W/3][W/3]$ zurückgibt.

```
ThreePartition(Array A):
1  n ← length(A)
2  W ← ∑_{i=1}^{n} A[i]
3  if W nicht durch 3 teilbar then
4      return 0
5  else
6      E ← new Array [0..n][0..W/3][0..W/3]
7      for i ← 0 to n do
8          E[i][0][0] ← 1
9      for j1 ← 1 to W/3 do
10         for j2 ← 1 to W/3 do
11             E[0][j1][j2] ← 0
12     for i ← 1 to n do
13         for j1 ← 0 to W/3 do
14             for j2 ← 1 to W/3 do
15                 if j1 < A[i] and j2 < A[i] then
16                     E[i][j1][j2] ← E[i − 1][j1][j2]
17                 else if j1 ≥ A[i] and j2 < A[i] then
18                     E[i][j1][j2] ← max{E[i − 1][j1][j2], E[i − 1][j1 − A[i]][j2]}
19                 else if j1 < A[i] and j2 ≥ A[i] then
20                     E[i][j1][j2] ← max{E[i − 1][j1][j2], E[i − 1][j1][j2 − A[i]]}
21                 else
22                     E[i][j1][j2] ← max{E[i − 1][j1][j2], E[i − 1][j1 − A[i]][j2 − A[i]]}
23 return E[n][W/3][W/3]
```

c) Die Laufzeit des Algorithmus hängt von den Eingabegrößen $n = \text{length}(A)$ und der Summe W der Elemente in A ab. Zeilen 1 und 23 benötigt konstante Laufzeit. In Zeile 2 werden n Werte aufsummiert, benötigt also eine Laufzeit in $\mathcal{O}(n)$. Zeilen 3 und 4 benötigen maximal konstante Laufzeit. Im Worst Case wird der Fall ab Zeile 5 aufgerufen. In Zeile 6 werden $n \cdot W/3 \cdot W/3$ Felder initialisiert, dies benötigt also eine Laufzeit in $\mathcal{O}(n \cdot W^2)$. Die Schleife in Zeilen 7 und 8 läuft in $\mathcal{O}(n)$; die verschachtelte Schleife von Zeile 9 bis 11 läuft in $\mathcal{O}(W^2)$; die verschachtelte Schleife von Zeile 12 bis 22 läuft in $\mathcal{O}(n \cdot W^2)$. Insgesamt hat der Algorithmus also eine Laufzeit in $\mathcal{O}(n \cdot W^2)$.