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

DAP2 – Präsenzübung 4

Besprechung: 17.05.2017 — 19.05.2017

Präsenzaufgabe 4.1: (Teile & Herrsche)

Betrachten Sie das Problem der *längsten absteigenden, zusammenhängenden Teilfolge*.

Gegeben sei ein Array $A[1..n]$ ganzer Zahlen. Gesucht ist die Länge eines maximalen Teilarrays $A[i..j]$ mit $1 \leq i \leq j \leq n$ und der Eigenschaft $A[i] > A[i+1] > \ldots > A[j]$.

a) Bestimmen Sie für das folgende Array eine längste absteigende, zusammenhängende Teilfolge.

$$1 \quad 3 \quad 1 \quad -1 \quad -2 \quad 1 \quad 3 \quad 2$$

b) Geben Sie unter Verwendung der *Teile-und-Herrsche*-Methode einen rekursiven Algorithmus (in Pseudocode) an, mit dem die Länge des längsten absteigenden, zusammenhängenden Teilarrays berechnet werden kann.

c) Zeigen Sie die Korrektheit der rekursiven Formulierung des Algorithmus und bestimmen Sie die asymptotische Worst-Case-Komplexität Ihres Algorithmus.

Lösung:

a) Die längste absteigende, zusammenhängende Teilfolge ist $3 \quad 1 \quad -1 \quad -2$ mit Länge 4.

b) Wir verwenden eine rekursive Hilfsfunktion, die zu einem Array $A = [a_p, a_{p+1}, \ldots, a_q]$ ein Tripel (l_A, m_A, r_A) von Teilarrays von A liefert, wobei

- l_A die Länge der längsten absteigenden, zsh. Teilfolge von A ist, die in a_p beginnt,
- m_A die Länge einer längsten absteigenden, zsh. Teilfolge von A ist, und
- r_A die Länge der längsten absteigenden, zsh. Teilfolge von A ist, die in a_q endet.

Wird ein Array $A = [a_p, a_{p+1}, \ldots, a_q]$ zerlegt in die beiden Teilarrays $A' = [a_p, a_{p+1}, \ldots, a_k]$ und $A'' = [a_{k+1}, \ldots, a_q]$, so gilt, da eine längeres absteigendes und zusammenhängendes neues Teilarray nur dann entstehen kann, wenn $a_k > a_{k+1}$:

$$m_A = \begin{cases} \max\{m_{A'}, m_{A''}, r_{A'} + l_{A''}\} & \text{falls } a_k > a_{k+1} \\ \max\{m_{A'}, m_{A''}\} & \text{sonst} \end{cases}$$

$$l_A = \begin{cases} l_{A'} + l_{A''} & \text{falls } a_k > a_{k+1} \text{ und } l_{A'} = length\,(A') \\ l_{A'} & \text{sonst} \end{cases}$$

$$r_A = \begin{cases} r_{A'} + r_{A''} & \text{falls } a_k > a_{k+1} \text{ und } r_{A''} = length\,(A'') \\ r_{A''} & \text{sonst} \end{cases}$$

Mit diesen Vorüberlegungen können wir den Teile-und Herrsche Algorithmus angeben, der in einer rekursiv definierten Hilfsfunktion `LLhilf` zum Tragen kommt:

`LLdsa` (A)
Input: Ein Array $A[1..n]$, $n \geq 1$, von ganzen Zahlen
Output: Länge eines längsten absteigenden, zusammenhängenden Teilarrays
1. $(l_A, m_A, r_A) \leftarrow$ `LLhilf` $(A, 1, n)$ ➢ `LLhilf (`$A, 1, n$`) liefert das Tripel` (l_A, m_A, r_A) `als Ergebnis`
2. **return** m_A

`LLhilf` (A, p, q);
Input: Ein Array $A[1..n]$ von ganzen Zahlen, p, q
Output: Tripel (l_A, m_A, r_A) der Längen längster absteigender, zusammenhängender Teilarrays in $A[p..q]$
1. **if** $p = q$ **then**
2. **return** $(1, 1, 1)$
3. **else**
4. $k \leftarrow \lfloor \frac{p+q}{2} \rfloor$
5. $(l_1, m_1, r_1) \leftarrow$ `LLhilf` (A, p, k)
6. $(l_2, m_2, r_2) \leftarrow$ `LLhilf` $(A, k+1, q)$
7. **if** $A[k] > A[k+1]$ **then**
8. $m \leftarrow \max\{m_1, m_2, r_1 + l_2\}$
9. **else**
10. $m \leftarrow \max\{m_1, m_2\}$
11. **if** $A[k] > A[k+1]$ *und* $l_1 = k - p + 1$ **then**
12. $l \leftarrow l_1 + l_2$
13. **else**
14. $l \leftarrow l_1$
15. **if** $A[k] > A[k+1]$ *und* $r_2 = q - k$ **then**
16. $r \leftarrow r_1 + r_2$
17. **else**
18. $r \leftarrow r_2$
19. **return** (l, m, r)

c) Wir zeigen, dass `LLhilf` angewandt auf (A, p, q) ein Tripel (l, m, r) berechnet, sodass l, m und r die oben beschriebenen Längen für $A[p..q]$ sind, mittels erweiterter vollständiger Induktion über die Arraylänge $n = q - p + 1$.

Die Behauptung gilt im Fall $p = q$, da hier jeweils die Längen 1 als Ergebnis geliefert wird (Zeile 2).

Es gelte nun für ein festes $m > 0$, dass das Hilfsprogramm für jedes Teilarray der Länge $n \leq m$ korrekt arbeitet.

Im Induktionsschritt betrachten wir nun ein $n > m$. Da $n > 0$ ist, wird der Else-Zweig ab Zeile 3 ausgeführt. Da $k - p + 1$ und $q - k$ je kleiner als m sind, haben wir nach Induktionsvoraussetzung, dass für die Eingaben (A, p, k) und $(A, k+1, q)$ die Werte l, m und r richtig berechnet werden. Mit den Vorüberlegungen folgt dann, dass die Behauptung auch für (A, p, q) gilt.

Für die Bestimmung des Berechnungsaufwandes in Abhängigkeit von $n = q-p+1$ erhalten

wir für `LLhilf` die folgende Rekursionsgleichung (wir nehmen n als eine Zweierpotenz an):

$$T(n) = \begin{cases} 2 & \text{falls } n = 1 \\ 2 \cdot T(\frac{n}{2}) + 10 & \text{sonst,} \end{cases}$$

so dass sich $T(n) \in \mathcal{O}(n)$ ergibt. Damit wird auch die Länge der längsten absteigenden, zusammenhängenden Teilfolge in Zeit $\mathcal{O}(n)$ berechnet.

Präsenzaufgabe 4.2: (Ein Ladeproblem)

Alice zieht um. Sie besitzt fast keine schweren Möbel, dafür n Bücherkartons der Größe $40cm \times 40cm \times 30cm$, wobei Bücherkarton i Gewicht w_i habe. Für den Umzug hat sie einen kleinen Transporter mit einem Laderaum von $2,8m \times 1,2m \times 1,2m$ gemietet. Sie hat sich überlegt, da am Anfang noch die meisten Umzugshelfer Zeit und Energie haben, dass es am praktikabelsten ist, bei der ersten Fahrt nur Bücherkartons mit einem möglichst großen Gesamtgewicht mitzunehmen.

a) Entwerfen Sie einen gierigen Algorithmus zur Berechnung einer optimalen (und zulässigen) Beladung des Transporters $B \subseteq \{1, \ldots, n\}$, die das zu tragende Gesamtgewicht $\sum_{i \in B} w_i$ maximiert, und beschreiben Sie ihn in eigenen Worten. Geben Sie Ihren Algorithmus außerdem in Pseudocode an.

b) Geben Sie eine möglichst kleine Worst-Case-Laufzeitschranke in $\mathcal{O}$-Notation an.

c) Beweisen Sie, dass Ihr Algorithmus eine optimale Lösung berechnet.

Lösung:

a) Wir bemerken, dass unter den gegebenen Maßen des Transporters und der Kantenlängen der Kartons $7 \cdot 3 \cdot 4 = 84$ Kartons verladen werden können. D. h., das Problem lässt sich dahingehend reduzieren, dass wir die $k = 84$ wertvollsten Kartons aus n Kartons wählen müssen. Wir können also das aus der Vorlesung bekannte Prinzip der *Greedy-Algorithmen* zu Nutze machen.

Wir können also die Kartons nach absteigendem Gewicht w_i sortieren und in der sortierten Permutation die k ersten Kartons wählen. Aus der Vorlesung wissen wir, dass der Algorithmus eine optimale Lösung für unser Problem berechnet.

Der folgende Algorithmus gibt bei Eingabe einer ganzen Zahl k und eines Arrays die k grösten Werte des Array aus.

`GierigeAuswahl` (int k, Array *Kartons*):
1 Sortiere das Array *Kartons* absteigend bzgl. ihrer Gewichte w_i
2 *Auswahl*[1..k] $\leftarrow$ *Kartons*[1..k]
3 **return** *Auswahl*

c) Zeile 1 benötigt $\mathcal{O}(n \log n)$ Rechenschritte (z. B. mit MergeSort). Zeile 2 benötigt $k+1 = \mathcal{O}(n)$ Rechenschritte und die Rückgabe in Zeile 3 benötigt $\mathcal{O}(1)$ Rechenschritte. Insgesamt können wir also $\mathcal{O}(n \log n)$ als asymptotische obere Schranke angeben.

d) *Behauptung:* Algorithmus `GierigeAuswahl` berechnet eine optimale Beladung a, welche die Zielfunktion $\sum_{i \in B} w_i$ maximiert.

Beweis: Für einen Widerspruchsbeweis nehmen wir an, die von unserem Algorithmus berechnete Lösung $a = (a_1, \ldots, a_k)$ sei nicht optimal. Dann gibt es eine optimale Lösung $b = (b_1, \ldots, b_k)$ mit $b \neq a$. Wir nehmen o.B.d.A. an, es gelte $(a_1 \geq a_2 \ldots \geq a_k)$ und $(b_1 \geq b_2 \ldots \geq b_k)$.

Sei j der kleinste Index für den $a_j \neq b_j$ gilt. Aufgrund unserer gierigen Strategie gilt $a_j > b_j$ und da die b_i absteigend sortiert sind, kann a_j nicht in der Lösung b vorkommen. Wenn wir nun die Kiste mit Gewicht b_j durch die Kiste mit Gewicht a_j ersetzen, dann erhalten wir eine Lösung $b' = (b_1, \ldots, b_{j-1}, a_j, b_{j+1}, \ldots, b_k)$ mit höherem Gesamtgewicht als b. Dies ist ein Widerspruch, denn wir hatten b als optimal angenommen. □