Algorithmen und Datenstrukturen

Aufgabenblatt (Ausgesuchte Aufgaben & Lösungen)

Im Folgenden finden Sie für einige ausgewählte Aufgaben der Übungsblätter aus diesem Semester Lösungsvorschläge. Teilweise sind die Lösungen mit zusätzlichen Bemerkungen oder besonders ausführlichen Rechnungen versehen, um den Lösungsweg zu verdeutlichen. Sollten Sie Fragen zu den Lösungsvorschlägen bzw. Kommentaren haben oder einen möglichen Fehler finden, melden Sie sich bitte im AD-Forum.

Übung 1

Betrachten Sie das Problem *Zweit-Kleinstes-Element*:

- *Eingabe:* Ein Array $A[1, \ldots, n]$ von $n > 1$ Zahlen.
- *Ausgabe:* Ein Index i, sodass es einen Index $j \neq i$ gibt mit $A[j] \leq A[i]$ und für alle Indizes $k \in \{1, 2, \ldots, n\} \setminus \{j\}$ gilt $A[k] \geq A[i]$.

(a) Beschreiben Sie in Pseudocode einen Algorithmus der das Problem Zweit-Kleinstes-Element löst.

(b) Beweisen Sie die Korrektheit Ihres Algorithmus mit Hilfe einer geeigneten Schleifeninvariante.

(c) Analysieren Sie die worst-case Laufzeit des formulierten Algorithmus.

Lösung 1

(a) Der folgende Algorithmus löst das Problem Zweit-Kleinstes-Element.

Algorithmus 1: Zweit-Kleinstes-Element(A)

```
1  min ← 1
2  zmin ← 2
3  if A[2] < A[1]
4      min ↔ zmin
5  for i ← 3 to length(A)
6      if A[i] < A[min]
7          zmin ← min
8          min ← i
9      else if A[i] < A[zmin]
10         zmin ← i
11 return zmin
```

(b) Die Schleifeninvariante, die wir im Folgenden beweisen werden, lautet:

$$I(i, min, zmin) :=(A[min] = \min\{A[j] \mid j \in \{1, 2, \ldots, i-1\}\}) \wedge \\ (A[zmin] = \min\{A[j] \mid j \in \{1, 2, \ldots, i-1\} \setminus \{min\}\})$$

Initialisierung. Falls $A[2] \geq A[1]$, gilt $I(3, 1, 2)$: $A[min] = \min\{A[1], A[2]\}$ gilt, da min mit 1 initialisiert ist und $A[1]$ ein kleinstes Element aus $\{A[1], A[2]\}$ ist. Ansonsten gilt $I(3, 2, 1)$: $min = 2$ und $A[2] < A[1]$.

$A[zmin] = \min\{A[1], A[2]\} \setminus \{A[min]\}$ gilt in beiden Fällen, da $zmin$ so initialisiert wird, dass es auf dasjenige der ersten beiden Elemente zeigt, auf das min nicht zeigt.

Erhaltung. Annahme: $I(i-1, min, zmin)$.
Wir betrachten die drei Fälle:

- $A[i] < A[min]$: In Zeile 7 und 8 gilt $A[i] < A[min]$ und damit ist $A[i] = \min\{A[j] \mid j \in \{1, 2, \ldots, i\}\}$ und $A[min] = \min\{A[j] \mid j \in \{1, 2, \ldots, i\} \setminus \{i\}\}$. Nach Zeile 8 zeigt $zmin$ also wieder auf das zweit-kleinste und min auf das kleinste Element.
- $\neg(A[i] < A[min]) \wedge (A[i] < A[zmin])$: Da $A[i] > A[min]$ zeigt min auf das kleinste Element der Menge $\{A[j] \mid j \in \{1, 2, \ldots, i-1\}\} \cup \{A[i]\}$. Außerdem gilt $A[i] = \min\{A[j] \mid j \in \{1, 2, \ldots, i\} \setminus \{min\}\}$. Nach Zeile 10 zeigt $zmin$ also wieder auf das zweit-kleinste Element.
- $\neg(A[i] < A[min]) \wedge \neg(A[i] < A[zmin])$: Analog zu min in Fall 2.

In jedem Fall gilt nach Durchlauf der Schleife $I(i, min, zmin)$.

Terminierung. Es gilt am Ende i =length$(A)+1$ und damit $I(\text{length}(A)+1, min, zmin)$ und damit $A[zmin] = \min\{A[j] \mid j \in \{1,2,\ldots,i\} \setminus \{\min\{A[j] \mid j \in \{1,2,\ldots,i\}\}\}\}$.

(c) Sei $n = \text{length}(A)$ die Problemgröße. Die Ausführung jeder einzelnen Zeile verursacht konstante Kosten, sei c die höchste dieser Konstanten. Wir führen Zeile 4 höchstens einmal aus, Zeilen 6-10 jeweils höchstens $n-2$ mal, Zeile 5 höchstens $n-1$ mal. Dann ist $T(n) \leq 4 \cdot c + (n-1) + (n-2) \cdot 5 \cdot c + c = (n-1) \cdot 5 \cdot c = \mathrm{O}(n)$.

Anmerkung: Es ist wichtig, zwischen den Kosten pro Durchführung einer Zeile und der Anzahl der Durchführungen der Zeile zu unterscheiden.

Übung 2

Bestimmen Sie die Größenordnung der Funktionen wenn möglich mittels Mastertheorem. Falls das Mastertheorem nicht anwendbar sein sollte, begründen Sie dies und verwenden stattdessen die Substitutionsmethode (mit Beweis der Korrektheit ihrer „geratenen“ Lösung).

(a)
$$T_1(n) := \begin{cases} 1, & \text{für } n = 1 \\ 4 \cdot T(\lceil n/4 \rceil) + 8n, & \text{sonst} \end{cases}$$

(b)
$$T_2(n) := \begin{cases} 1, & \text{für } n = 0 \\ 2 \cdot T(n-1) + 4, & \text{sonst} \end{cases}$$

(c)
$$T_3(n) := \begin{cases} 1, & \text{für } n = 1 \\ 3 \cdot T(\lfloor n/3 \rfloor) + 2n \log n, & \text{sonst} \end{cases}$$

(d)
$$T_4(n) := \begin{cases} 1, & \text{für } n = 1 \\ 4 \cdot T(\lfloor n/3 \rfloor) + 2n \log n, & \text{sonst} \end{cases}$$

Lösung 2

(a) Das Mastertheorem ist auf $T_1(n)$ anwendbar mit: $a = 4$, $b = 4$, $\log_b(a) = \log_4(4) = 1$ und $f(n) = 8n$. Aufgrund von $8n \in \Theta(n)$ liegt der 2. Fall des Mastertheorems vor ($f(n) \in \Theta(n^{\log_b(a)})$). Daher gilt: $T_1(n) \in \Theta(n \log(n))$.

(b) Das Mastertheorem ist nicht auf $T_2(n)$ anwendbar, da die Laufzeit in jedem Rekursionsschritt nicht um einen konstanten Faktor geringer wird. Das Lösen per Substitutionsmethode erfordert initial das Erraten einer möglichen Lösung:

$$T_2(n) = 2 \cdot T_2(n-1) + 4$$

$$
\begin{aligned}
&= 2 \cdot 2 \cdot T_2(n-2) + 2 \cdot 4 + 4 \\
&= 2 \cdot 2 \cdot 2 \cdot T_2(n-3) + 2 \cdot 2 \cdot 4 + 2 \cdot 4 + 4 \\
&= \dots \\
&= 2^k \cdot T_2(n-k) + 4 \sum_{i=0}^{k-1} 2^i \\
&= 2^k \cdot T_2(n-k) + 4 \cdot 2^k - 4
\end{aligned}
$$

für ein $k \in \mathbb{N}_0$ und $k \leq n$. Mit $k = n$ folgt:

$$
\begin{aligned}
T_2(n) &= 2^n \cdot T_2(n-n) + 4 \cdot 2^n - 4 \\
&= 2^n \cdot 1 + 4 \cdot 2^n - 4 \\
&= 5 \cdot 2^n - 4
\end{aligned}
$$

Zu zeigen: $T_2(n) = 5 \cdot 2^n - 4$.

Beweis der geratenen Struktur mittels Induktion:

- Induktionsanfang: Offensichtlich wahr $n = 0$, da $5 \cdot 2^0 - 4 = 1$
- Induktionsschritt:

$$
\begin{aligned}
T_2(n+1) &= 2 \cdot T(n+1-1) + 4 \\
&= 2 \cdot T(n) + 4 \\
&= 2 \cdot (5 \cdot 2^n - 4) + 4 \\
&= 5 \cdot 2^{n+1} - 4
\end{aligned}
$$

Also ist die geratene Struktur korrekt. Somit gilt: $T_2(n) \in \Theta(2^n)$.

(c) Das Mastertheorem ist nicht auf $T_3(n)$ anwendbar, da $f(n) \notin \mathcal{O}(n^{\log_b(a)})$ mit $\log_b(a) = \log_3(3) = 1$, sodass $2n \log n \notin \mathcal{O}(n^1)$. Fall 1 und 2 sind damit ausgeschlossen. Fall 3 ist ebenfalls nicht anwendbar, da $2n \log n$ zwar größer als $\Omega(n)$ aber nicht polynomiell größer (sondern nur logarithmisch größer).

Lösen per Substitutionsmethode (indem zuerst eine mögliche Lösung geraten wird):

$$
\begin{aligned}
T_3(n) &= 3 \cdot T_3(\lfloor n/3 \rfloor) + 2n \log n \\
&= 3 \cdot 3 \cdot T_3(\lfloor n/3^2 \rfloor) + 3 \cdot (2\frac{n}{3} \log \frac{n}{3}) + 2n \log n \\
&= 3 \cdot 3 \cdot 3 \cdot T_3(\lfloor n/3^3 \rfloor) + 3^2 \cdot (2\frac{n}{3^2} \log \frac{n}{3^2}) + 3 \cdot (2\frac{n}{3} \log \frac{n}{3}) + 2n \log n \\
&= 3^3 \cdot T_3(\lfloor n/3^3 \rfloor) + 2n \log \frac{n}{3^2} + 2n \log \frac{n}{3} + 2n \log n \\
&= 3^3 \cdot T_3(\lfloor n/3^3 \rfloor) + 2n \log \frac{n^3}{3^3} \\
&= 3^3 \cdot (3 \cdot T_3(\lfloor n/3^4 \rfloor) + 2\frac{n}{3^3} \log \frac{n}{3^3}) + 2n \log \frac{n^3}{3^3}
\end{aligned}
$$

$$
\begin{aligned}
&= 3^4 \cdot T_3(\lfloor n/3^4 \rfloor) + 2n \log \frac{n}{3^3} + 2n \log \frac{n^3}{3^3} \\
&= 3^4 \cdot T_3(\lfloor n/3^4 \rfloor) + 2n \log \frac{n^4}{3^6} \\
&= \dots \\
&= 3^k \cdot T_3(\lfloor n/3^k \rfloor) + 2n \log \frac{n^k}{3^{\sum_{i=0}^{k-1} i}} \\
&= 3^k \cdot T_3(\lfloor n/3^k \rfloor) + 2n \log \frac{n^k}{3^{0.5(k-1)k}} \\
&= 3^k \cdot T_3(\lfloor n/3^k \rfloor) + 2kn \log \frac{n}{3^{0.5(k-1)}}
\end{aligned}
$$

für ein $k \in \mathbb{N}$ und $k \leq \log_3 n$. Mit $k = \log_3 n$ folgt:

$$
\begin{aligned}
T_3(n) &= n \cdot 1 + 2n \log_3(n) \log(\frac{n}{3^{0.5(\log_3(n)-1)}}) \\
&= n + 2n \log_3(n) \log(\frac{n}{(3^{\log_3(n/3)})^{0.5}}) \\
&= n + 2n \log_3(n) \log(\frac{n}{(n/3)^{0.5}}) \\
&= n + 2n \log_3(n) \log((3n)^{0.5}) \\
&= n + n \log_3(n) \log(3n)
\end{aligned}
$$

Zu zeigen: $T_3(n) = n + n \log_3(n) \log(3n)$.

Beweis der geratenen Struktur mittels Induktion:

- Induktionsanfang: Offensichtlich wahr für $n = 1$, da $1 + 1 \cdot \log_3(1) \log(3 \cdot 1) = 1$
- Induktionsschritt: Betrachtung erfolgt o.B.d.A. für $3 \cdot n$ statt $n + 1$

$$
\begin{aligned}
T_3(3 \cdot n) &= 3 \cdot T(\lfloor (3 \cdot n)/3 \rfloor) + 2(3 \cdot n) \log(3 \cdot n) \\
&= 3 \cdot T(n) + 2(3 \cdot n) \log(3 \cdot n) \\
&= 3 \cdot (n + n \log_3(n) \log(3n)) + 2(3 \cdot n) \log(3 \cdot n) \\
&= 3n + 3n \log_3(n) \log(3n) + 3n \cdot 2 \log(3n) \\
&= 3n + 3n \log(3n)(\log_3(n) + 2) \\
&= 3n + 3n \log(3n) \log_3(3 \cdot 3n) \\
&= 3n + 3n \frac{\log_3(3n)}{\log_3(e)} \frac{\log(3 \cdot 3n)}{\log(3)} \\
&= 3n + 3n \log_3(3n) \log(3 \cdot 3n) \frac{\log(3)}{\log(e)} \frac{1}{\log(3)} \\
&= 3n + 3n \log_3(3n) \log(3 \cdot 3n)
\end{aligned}
$$

Also ist die gerate Struktur korrekt. Somit gilt: $T_3(n) \in \mathcal{O}(n \log^2 n)$

(d) Das Mastertheorem ist auf $T_4(n)$ anwendbar mit $a = 4$, $b = 3$, $\log_b(a) = \log_3(4) = 1,26\ldots$ und $f(n) = 2n \log n$. Da $2n \log n \in \mathcal{O}(n^{\log_3 4})$, liegt der 1. Fall des Mastertheorems vor. Entsprechend gilt: $T_4(n) \in \Theta(n^{\log_3 4})$

Übung 3

In Algorithmus 2 ist der Pseudocode des Algorithmus StoogeSort zu sehen.

Algorithmus 2: StoogeSort(A, i, j)

```
1  if A[i] > A[j]
2      A[i] ↔ A[j]
3  if i + 1 ≥ j
4      return
5  k ← ⌊(j − i + 1)/3⌋
6  StoogeSort(A, i, j − k)        // sortiere ersten beiden Drittel
7  StoogeSort(A, i + k, j)        // sortiere letzten beiden Drittel
8  StoogeSort(A, i, j − k)        // sortiere ersten beiden Drittel
```

(a) Beweisen Sie die Korrektheit von StoogeSort. Das heißt, beweisen Sie dass der Aufruf StoogeSort$(A, 1, \text{length}(A))$ das Array A korrekt sortiert. Führen Sie dazu einen Induktionsbeweis über die Länge des Teilarrays von A im Intervall $[i, j]$.

(b) Analysieren Sie die worst-case Laufzeit von StoogeSort im O-Kalkül. Nutzen Sie dazu eine geeignete Rekursionsgleichung.

Lösung 3

In der folgenden Lösung sind *grau geschriebene* Abschnitte als zusätzliche Hilfestellung gedacht, und nur die nicht grau geschriebenen Abschnitte sind für eine Lösung erforderlich.

(a) *Wir beweisen die Korrektheit von StoogeSort, d.h. wir zeigen, dass der Algorithmus das Array A im Intervall $[i, j]$ $(j \geq i)$ korrekt sortiert. Die Länge dieses Intervalls ist $j - i + 1$, nennen wir $l = j - i + 1$. Wie in der Aufgabenstellung gegeben, führen wir einen Induktionsbeweis über die Länge von A.*
Wir führen einen Induktionsbeweis über die Länge des Teilarrays A im Intervall $[i, j]$. Sei $l = j - i + 1 \geq 1$ dessen Länge.
Für eine rekursive Funktion macht es Sinn, wenn der Induktionsanfang genau die Aufrufe abdeckt, wo keine weiteren Rekursionen stattfinden. In diesem Fall returned die Funktion ohne weitere Rekursion, falls $i + 1 \geq j$. Wir nehmen also dies als Induktionsanfang an. Umgestellt erhalten wir $j - i + 1 \leq 2$ oder $l \leq 2$. Der Induktionsanfang sollte also Arrays der Länge 1 und 2 abdecken.
Induktionsanfang: Sei $1 \leq l \leq 2$. Für $l = 1$ gilt $i = j$, daher ist die Bedingung $A[i] > A[j]$ falsch und wir fahren mit Zeile 3 fort. Außerdem ist daher $j - i + 1 = 1$ oder umgestellt $i + 1 = j + 1 \geq j$, also wird in Zeile 4 returned und das Array

wurde korrekt sortiert (da es nur aus einem Element bestand). Für $l = 2$ stellen Zeile 1 und 2 sicher, dass $A[i] \leq A[j]$ gilt (das zweielementige Array ist also damit korrekt sortiert). Außerdem gilt für die Bedingung in Zeile 4 wieder $j - i + 1 = 2$, also $i + 1 = j$ und wir returnen mit dem sortierten Array.

In Zeile 5 berechnet StoogeSort $\lfloor (j - i + 1)/3 \rfloor = \lfloor l/3 \rfloor$*, also bis auf Abrundung ein Drittel der Arraylänge. Danach werden jeweils (bis auf Rundung)* $2/3$ *des Arrays sortiert. Wir werden benötigen, dass diese rekursiven Aufrufe korrekt sortieren. Zeilen 1-4 fangen Arraylängen* ≤ 2 *ab. Damit ist sichergestellt, dass alle drei* StoogeSort*-Aufrufe mindestens ein Element zum Sortieren haben (sonst müsste unser Induktionsanfang den Fall* $l = 0$ *abdecken!) Durch* $k = \lfloor l/3 \rfloor < l$ *ist sichergestellt, dass die folgenden rekursiven Aufrufe sich auf kleinere Längen* l *beziehen. Unsere Induktionsannahme nimmt also an, dass alle kleineren Teilarrays korrekt durch* StoogeSort *sortiert werden. Wir nehmen uns also eine Variable* L *her, die uns beschreibt, bis zu welchen Arraylängen* l *wir bereits wissen, dass* StoogeSort *korrekt sortiert und zeigen die Aussage für* $l = L + 1$*.*

Induktionsannahme: Für alle $l = j - i + 1 \leq L$ sortiert StoogeSort korrekt.

Induktionsbehauptung: Auch für $l = L + 1$ sortiert StoogeSort korrekt.

Für den Induktionsschritt ist es wichtig, die Logik hinter der Sortierung von StoogeSort *nachzuvollziehen: Warum folgt aus den drei Teilsortierungen die Sortierung des gesamten Teilarrays im Intervall* $[i, j]$*? Zunächst müssen wir aber zeigen, dass für die Teilsortierungen tatsächlich die Induktionsannahme angewendet werden kann (die Teilarrays also kürzer sind).*

Induktionsschritt: Wir zeigen, dass StoogeSort für $l = L+1$ korrekt sortiert. In Zeile 5 wird $k = \lfloor (j-i+1)/3 \rfloor = \lfloor l/3 \rfloor$ berechnet. Wir haben $1 \leq \lfloor l/3 \rfloor \leq l-1$. Die drei Teilsortierungen sortieren auf den Intervallen $[i, j-k]$, $[i+k, j]$. Deren Längen sind $(j-k)-i+1 = l-k$ und $j-(i+k)+1 = l-k$. Da $k \geq 1$, ist die Länge beider Teilarrays höchstens $l - k \leq l - 1 = L$, daher können wir für diese Sortierungen die Induktionsannahme anwenden, womit die drei StoogeSort-Aufrufe korrekt sortieren.

Jetzt fehlt nur noch, wie aus den Sortierungen der Teilarrays die Gesamtsortierung folgt. Der Algorithmus sortiert auf den Dritteln $[i, i + k - 1]$*,* $[i + k, j - k]$ *und* $[j - k + 1, j]$*.*

Wir fassen das Teilarray $[i, j]$ als drei Drittel auf, jeweils in den Intervallen $D_1 = [i, i + k - 1]$, $D_2 = [i + k, j - k]$ und $D_3 = [j - k + 1, j]$. Zeilen 6 und 8 sortieren D_1, D_2 und Zeile 7 sortiert D_2, D_3. Nach dem Sortierschritt in Zeile 6 sind alle Elemente in D_2 größer als die Elemente in D_1. Durch den Sortierschritt in Zeile 7 sind alle Elemente in D_3 größer als die Elemente in D_1 und D_2 und die Elemente in D_3 sind aufsteigend sortiert. Durch Zeile 8 werden D_1 und D_2 aufsteigend sortiert. Aus diesen drei Fakten (D_3 aufsteigend sortiert, D_3 enthält größte Elemente und D_1, D_2 aufsteigend sortiert) folgt, dass StoogeSort für $l = L + 1$ korrekt sortiert.

Der nachfolgende Induktionsschluss wird weitestgehend als optional gehandhabt.

Induktionsschluss: Damit sortiert StoogeSort korrekt für alle Längen $l = j - i + 1$. Hieraus folgt auch, dass StoogeSort$(A, 1, \mathit{length}(A))$ das Array A

korrekt sortiert.

(b) *StoogeSort zerlegt das Problem rekursiv in kleinere Teilprobleme. Daher sollte man überprüfen, ob das Master-Theorem anwendbar ist. Anhand der Länge $l = j - i + 1$ können wir wieder wie in Punkt (a) den Basisfall und den Rekurrenzfall unterscheiden.*
Sei $l = j - i + 1$ die Länge des Teilarrays $[i, j]$ in A. Falls $l \leq 2$ (oder äquivalent $i + 1 \geq j$), returned StoogeSort in Zeile 4, womit die Laufzeit durch eine Konstante c_1 beschränkt ist. Die drei Teilaufrufe operieren auf Teilarrays mit den Längen $l - k$ (siehe Punkt (a)). Dies ist höchstens $l - k \leq l - \lfloor l/3 \rfloor \leq \lceil 2/3 \cdot l \rceil$. Hat also StoogeSort die Laufzeit $T(l)$, so benötigen diese drei Teilaufrufe jeweils $T(\lceil 2/3 \cdot l \rceil)$ Zeit (zuzüglich einer Konstante c_2 für die konstanten Laufzeiten für Zeilen 1-5. Die Komplexität von StoogeSort lässt sich also als folgende Rekurrenz darstellen:

$$T(l) = \begin{cases} c_1 & \text{, falls } l \leq 2 \\ 3T(\lceil \frac{2}{3} l \rceil) + c_2 & \text{, sonst} \end{cases}$$

*Aus dem Cormen wissen wir, dass die Rundungsklammern **für die Anwendung des Mastertheorems** weggelassen werden dürfen.*
Es lässt sich das Mastertheorem anwenden mit $a = 3$, $b = 1/(2/3) = 3/2$ und $f(l) = c_2$ (c_1 ist eine Konstante). Damit ist $f(l) = c_2 = O(1) = \mathcal{O}(l^{\log_{3/2} 3}) = \mathcal{O}(l^{\log_b a})$, also tritt der erste Fall des Mastertheorems ein. Wir erhalten die Zeitkomplexität $T(l) = \mathcal{O}(l^{\log_{3/2} 3}) \approx \mathcal{O}(l^{2.71})$.

Übung 4

(a) Ist ein aufsteigend sortiertes Array ein min-Heap oder ein max-Heap? Begründen Sie Ihre Antwort.

(b) Zeigen Sie, dass ein Heap mit n Elementen eine Höhe $\Theta(\log n)$ hat.

(c) Zeigen Sie, dass ein Heap der Größe n maximal $\lceil \frac{n}{2^{h+1}} \rceil$ Knoten der Höhe h hat.

(d) Zeigen Sie, dass in einem Heap die Anzahl der inneren Knoten um eins kleiner oder gleich der Anzahl an Blättern ist.

Lösung 4

(a) Sei A das betrachtete aufsteigend sortierte Array der Länge n. In einem Heap gilt die Funktion $Parent(i) = \lfloor \frac{i}{2} \rfloor$ für alle $i \in \{1, 2, ..., n\}$. Für ein min-Heap gilt $key(A[Parent(i)]) \leq key(A[i])$ für alle $i \in \{2, 3, ..., n\}$. Somit muss ein min-Heap $key(A[\lfloor \frac{i}{2} \rfloor]) \leq key(A[i])$ für alle $i \in \{2, 3, ..., n\}$ erfüllen. In einem aufsteigend sortiertem Array gilt $A[k] \leq A[j]$ für alle $k < j$ mit $k, j \in \{1, 2, ..., n\}$. Folglich ist ein Heap über einem aufsteigend sortiertem Array ein min-Heap.

(b) Sei h die Höhe eines Heap mit n Elementen. Dann gilt für die Anzahl an Knoten, die nicht in der untersten Ebene liegen $\sum_{i=0}^{h-1} 2^i = 2^h - 1$. Die Anzahl an Knoten die in der untersten Ebene liegen ist zwischen 1 und 2^h.
Enthält die unterste Ebene des Heaps nur einen Konten, dann gilt für die Anzahl an Knoten im Heap $n = 2^h - 1 + 1 = 2^h$.
Sind in der untersten Ebene 2^h Knoten, so gilt $n = 2^h - 1 + 2^h = 2^{h+1} - 1 < 2^{h+1}$.
Folglich ist die Anzahl an Knoten eines Heaps der Höhe h eingeschränkt durch $2^h \leq n < 2^{h+1} \Rightarrow h \leq \log_2 n < h + 1$. Da die Höhe h eine natürlich Zahl ist, folgt $h = \lfloor \log_2 n \rfloor = \Theta(\lfloor \log_2 n \rfloor) = \Theta(\log n)$.

(c) **Behauptung**: Ein Heap der Größe n hat maximal $\lceil \frac{n}{2^{h+1}} \rceil$ Knoten der Höhe h.
Im folgenden wird die Behauptung per Induktion bewiesen:
Induktionsvorraussetzung (IV): Ein Heap der Größe n hat maximal $k_h(n) = \lceil \frac{n}{2^{h+1}} \rceil$ Knoten der Höhe h.
Induktionsanfang: Für h = 0 gilt $k_0(n) = \lceil \frac{n}{2^1} \rceil = \lceil \frac{n}{2} \rceil$. Dies gilt, da die Knoten mit Höhe 0 die Blätter sind.
Induktionsschritt: Sei H ein Heap mit n Knoten. Sei H' der Heap der sich ergibt, wenn alle Blätter aus H entfernt werden. H' besteht dann aus $n' = n - \lceil \frac{n}{2} \rceil = \lfloor \frac{n}{2} \rfloor$ Knoten. Sei I_H die Menge der inneren Knoten aus H. Für alle Knoten i in I_H ist die Höhe als der längste Pfad von i zu einem Blatt im Teilbaum von i definiert. Durch das entfernen der Blätter wird jeder dieser Pfade um eins reduziert. Somit folgt, dass Knoten der Höhe $h + 1$ in Heap H eine Höhe h in Heap H' haben und es gilt $k_{h+1}(n) = k_h(n')$. Daraus folgt

$$k_{h+1}(n) = k_h(n') \overset{IV}{=} \lceil \frac{n'}{2^{h+1}} \rceil = \lceil \frac{\lfloor \frac{n}{2} \rfloor}{2^{h+1}} \rceil \leq \lceil \frac{\frac{n}{2}}{2^{h+1}} \rceil = \lceil \frac{n}{2^{h+2}} \rceil.$$

Somit gilt die Behauptung für alle h.

(d) **Behauptung**: In einem Heap sind die Anzahl der inneren Knoten um eins kleiner oder gleich der Anzahl an Blättern.

Zunächst werden Definitionen eingeführt. Für einen beliebigen Heap H bezeichnet n die gesamte Anzahl an Knoten, n_i die Anzahl an inneren Knoten und n_b die Anzahl an Blättern. Der innere Knoten mit den höchsten Index wird als i_{max} bezeichnet. Die Funktion $children(i_{max})$ gibt die Anzahl an Kindern von i_{max} zurück.
Induktionsanfang:

- $n = 0$: Es gibt weder innere Knoten noch Blätter. Folglich gilt $n_b = n_i = 0$ und die Behauptung für $n = 0$.
- $n = 1$: Die Wurzel ist ein Blatt und es gibt keine innere Knoten. Folglich gilt $n_b - 1 = n_i = 0$ und die Behauptung für $n = 1$.

Induktionsvoraussetzung (IV): Für einen Heap der Größe n gilt $IV_a(n) \oplus IV_b(n)$. Dabei sind $IV_a(n)$ und $IV_b(n)$ definiert als:

- $IV_a(n) : n_b = n_i \wedge children(i_{max}) = 1$

- $IV_b(n) : n_b - 1 = n_i \wedge children(i_{max}) = 2$.

Anmerkung: Es gilt entweder $IV_a(n)$ oder $IV_b(n)$. Für $IV_a(n)$ gilt, dass die Anzahl der inneren Knoten gleich der Anzahl der Blätter ist und das der innere Knoten mit dem höchsten Index (i_{max}) genau ein Kind besitzt. Für $IV_b(n)$ gilt, dass die Anzahl der inneren Knoten um eins kleiner als die Anzahl der Blätter ist und das der innere Knoten mit dem höchsten Index (i_{max}) genau zwei Kinder besitzt. Die Betrachtung der Kinder in der IV ist nicht notwendig, erleichter aber die Argumentation.

Induktionsschritt: Die IV gilt für einen Heap H der Größe n. Es können zwei Fälle auftreten:

- Fall 1: $IV_a(n)$ gilt und demnach auch $n_b = n_i \wedge children(i_{max}) = 1$. Da i_{max} nur ein Kind hat, wird beim Hinzufügen eines weiteren Knotens an H, dieser an i_{max} angehängt. Dadurch steigt n_b um eins, während n_i gleich bleibt. Folglich gilt dann $n_b - 1 = n_i$. i_{max} besitzt nun zwei Kinder ($children(i_{max}) = 2$). Daraus folgt, dass $IV_b(n+1)$ für $n+1$ gilt.

- Fall 2: $IV_b(n)$ gilt und demnach auch $n_b - 1 = n_i \wedge children(i_{max}) = 2$. Da i_{max} zwei Kinder besitzt, wird der nächste Knoten an das Blatt b_{min} mit dem kleinsten Index angehängt. Dadurch wird b_{min} zu dem inneren Knoten i_{max} mit dem höchsten Index und besitzt genau ein Kind ($children(i_{max}) = 1$). Ein neues Blatt wurde zum Heap hinzugefügt und ein anderes Blatt ist zu einem inneren Knoten geworden. Damit steigt die Zahl n_i um eins, während n_b gleich bleibt. Dann gilt $n_b = n_i$. Folglich gilt $IV_a(n+1)$ für $n+1$.

Dadurch ist gezeigt, dass die IV für alle n gilt. Die Behauptung ist in der IV enthalten und daher bewiesen.

Übung 5

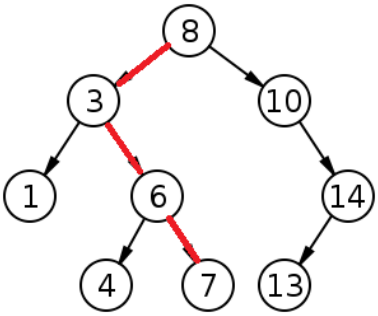
(a) Nehmen Sie an, die Suche nach einem Schlüssel k in einem binären Suchbaum endet in einem Blatt. Dann gibt es drei Mengen: A, die Schlüssel links des Suchpfads; B, die Schlüssel auf dem Suchpfad; sowie C, die Schlüssel rechts des Suchpfads. Beweisen oder widerlegen Sie die folgende Behauptung: Für beliebige drei Schlüssel $a \in A$, $b \in B$ und $c \in C$ gilt $a \leq b \leq c$.

(b) Aus der Vorlesung ist Ihnen bekannt, dass zum Suchen in einem vollständigen Binärbaum $O(\log_2 n)$ Zeit benötigt wird. Wie ist die asymptotische Laufzeit, wenn stattdessen jeder innere Knoten bis zu 3 Kinder haben darf? Wie, bei bis zu $\log_2 n$ Kindern?

Lösung 5

(a) Wir betrachten den eingezeichneten Suchpfad vom unten dargestellten Binärbaum. Dann gilt $A = \{1, 4\}$, $B = \{3, 6, 7, 8\}$ und $C = \{10, 13, 14\}$. (Beachten Sie das

Beispiele mit $A = \varnothing$ keine Gegenbeispiele darstellen, da die Aussage implizit für alle $a \in A$ gelten muss. Da keine Elemente in A existieren ist die Aussage somit dann standardmäßig erfüllt.



(b) Betrachten wir zunächst den Fall mit 3 Kinder pro innerem Knoten. Da der Baum vollständig ist, ist die Tiefe des Baums $O(\log_3 n) = O(\log n)$. Damit wird wie beim Binärbaum $O(\log_2 n) = O(\log n)$ Zeit zum Suchen benötigt.

Wenn jeder Knoten bis zu $\log_2 n$ Kinder haben darf, ist die Höhe des vollständigen Baums $O\left(\log_{\log_2 n} n\right)$. Mithilfe der Logarithmengesetze bekommen wir $O\left(\log_{\log_2 n} n\right) = O\left(\frac{\ln n}{\ln \log_2 n}\right)$. In jedem der Suchbaumknoten, den wir während einer Suche durchlaufen, müssen wir entscheiden, in welchen der $O(\log n)$ Teilbäume wir absteigen müssen. Selbst, wenn wir eine Binärsuche durchführen, wird für diesen Schritt $O(\log_2 \log_2 n) = O(\ln \log_2 n)$ Zeit benötigt.

Insgesamt benötigt man also für die Suche $O\left(\frac{\ln n}{\ln \log_2 n}\right) \cdot O(\ln \log_2 n) = O(\ln n)$ Zeit.

In beiden Fällen erhalten wir also keine Laufzeitverbesserung.

Übung 6

Beweisen oder widerlegen Sie folgende Aussagen zu ungerichteten Graphen:

(a) Es gilt für $G = (V, E)$:

$$\sum_{v \in V} deg(v) = 2|E|$$

(b) Seien v, w die einzigen beiden Knoten in $G = (V, E)$ mit ungeradem Grad, so sind v und w über einen Pfad in G verbunden.
Hinweis: Nutzen Sie die Aussage aus Aufgabe a).

(c) $G = (V, E)$ ist zusammenhängend, wenn für alle Knoten $v \in V$ gilt:

$$deg(v) \geq \lceil (|V| - 1)/2 \rceil$$

Beachten Sie: Die Aussage ist **nicht**: "*Wenn $G = (V, E)$ zusammenhängend ist, dann gilt für alle Knoten $v \in V$: $deg(v) \geq \lceil (|V| - 1)/2 \rceil$*"

Lösung 6

(a) Sei m die Anzahl der Kanten.
Induktionsvoraussetzung: Für Graphen mit höchstens m Kanten gilt:

$$\sum_{v \in V} deg(v) = 2m.$$

Induktionsanfang: Ein Graph ohne Kanten ($m = 0$) hat nur Knoten mit Grad 0.
Induktionsschluss: Angenommen der Graph G hat $m + 1$ Kanten. Entferne eine zufällige Kante. Der daraus resultierende Graph $G' = (V, E')$ hat m Kanten. Es gilt die IV für G': $\sum_{v \in V} deg(v) = 2m$. G hat genau die entfernte Kante zusätzlich. Diese verbindet genau zwei Knoten in G, wodurch deren Grad jeweils um 1 erhöht wird. Daher gilt für G: $\sum_{v \in V} deg(v) = 2m + 2 = 2(m + 1)$.

(b) Angenommen v und w sind nicht verbunden. Dann lässt sich V in zwei Mengen V_1 und V_2 partitionieren, sodass kein Knoten aus V_1 mit einem Knoten aus V_2 verbunden ist. Das heißt, $G = (V_1 \uplus V_2, E_1 \uplus E_2)$[1]. Dann ist $G_1 = (V_1, E_1)$ ein Graph.
Sei o.B.d.A. $v \in V_1, w \notin V_1$. Dann gilt $\forall x \in V_1 \setminus \{v\} : deg(x)$ ist gerade. Damit ist $\sum_{x \in V_1} deg(x)$ ungerade und nach Aufgabe 1.a) G_1 kein Graph.
Durch diesen Widerspruch schließen wir, dass v und w verbunden sind.

(c) Angenommen $G = (V, E)$ ist nicht zusammenhängend und $\forall v \in V : deg(v) \geq \lceil (|V| - 1)/2 \rceil$. Dann existieren zwei disjunkte Mengen V_1 und V_2, sodass $V = V_1 \uplus V_2$ und zwei disjunkte Mengen E_1 und E_2, sodass $E = E_1 \uplus E_2$ und $G_1 = (V_1, E_1)$ und $G_2 = (V_2, E_2)$ Graphen sind.
Sei o.B.d.A. $|V_1| \leq |V_2|$. Dann hat V_1 höchstens $\lfloor |V|/2 \rfloor$ Knoten. Damit hat jeder Knoten in V_1 höchstens Grad $\lfloor |V|/2 - 1 \rfloor = \lfloor (|V| - 2)/2 \rfloor$. Das ist ein Widerspruch zur Annahme und G ist entweder zusammenhängend oder es existiert ein Knoten mit kleinerem Grad.
Alternative: Angenommen, zwischen u und v existiert kein Pfad. Beide Knoten sind mit mindestens der Hälfte der Knotenmenge verbunden. Nach dem Schubfachprinzip existiert ein Knoten w, welcher jeweils mit u und v verbunden ist. Widerspruch zu der Annahme, dass G nicht zusammenhängend ist.

Übung 7

(a) Beschreiben Sie wie das folgende Problem als Graph modelliert werden kann. Sie befinden sich in einem quadratischen Gitter mit der Kantenlänge m. Felder können belegt sein (mit einem roten „X“ markiert) oder frei (leer). Sie starten an der Position $(1, 1)$ und möchten zur Position (m, m) gelangen. Der Output ist die Anzahl an Schritten, die man benötigt, von um zur Position (m, m) gelangen. In einem Schritt können Sie sich vertikal oder horizontal von einem freien Platz zu einem anderen freien Platz bewegen. Unten finden Sie ein Beispiel mit $m = 4$.

[1] $A \uplus B$ bezeichnet die disjunkte Vereinigung, d.h. $A \cap B = \emptyset$.

(b) Nehmen Sie an, dass man die Anzahl der Schritte minimieren möchte. Beschreiben Sie einen Algorithmus, der berechnet wie viele Schritte man braucht um von (1,1) zur Position (m, m) in $O(m^2)$ zu gelangen. Begründen Sie **kurz** die Korrektheit des Algorithmus', sowie die Einhaltung der asymptotischen Laufzeitschranke. **Hinweis**: Es kann auch sein, dass es keine Lösung gibt! In diesem Fall geben sollte der Algorithmus ∞ zurückgeben.

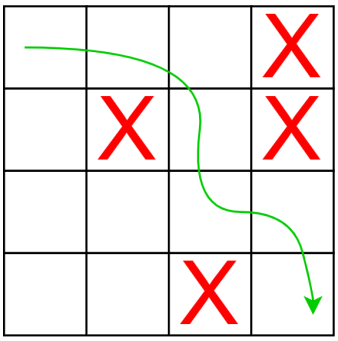


Abbildung 1: Ein möglicher Input mit $m = 4$ Der grüne Pfeil stellt den optimalen Pfad dar. Die Antwort für diese Instanz ist der Integer '6'.

Lösung 7

(a) Man kann das Gitter als einen ungerichteten (oder gerichteten!) Graphen auffassen, indem man alle freie Plätze als Knoten modelliert. Kanten zwischen Knoten gibt es genau dann, wenn die Position der codierten Felder sich in genau einer Koordinate um 1 unterscheidet. Belegte Felder werden nicht als Knoten modelliert.

(b) Bevor wir anfangen, analysieren wir die Größe des Graphen $G = (V, E)$, der durch die oben beschriebene Konstruktion entsteht. Im Worst-case, wo alle Felder frei sind, haben wir $O(|V|) = O(m^2)$ und $O(|E|) = O(m^2) = O(m^2)$, da $|E| \leq 8m^2$ gilt (jeder Knoten kann nicht mehr als vier ausgehende oder eingehende Kanten haben; eine deutlich schärfere Schranke ist möglich, aber nicht notwendig). Der folgende Algorithmus löst das Problem in $O(|V| + |E|)$:

(i) Initialisiere die Distanz von allen Knoten mit unendlich ($O(|V|)$).

(ii) Führe eine Breitensuche mit dem Startknoten $(1, 1)$ durch und gebe die Distanz von (m, m) als Lösung zurück ($O(|V| + |E|)$).

Da die Breitensuche immer den kürzesten Pfad findet, arbeitet der Algorithmus korrekt. Falls es keinen Pfad zur Position (m, m) gibt, wird ∞ zurückgegeben. Die Laufzeitschranke wird eingehalten, da:

$$O((|V| + |E|) + |V|) = O((m^2 + m^2) + m^2) = O(m^2). \tag{1}$$

Freiwillig, nicht notwendig:
Außerdem, wissen wir dass die Modellierung in $O(|V| + |E|)$ möglich ist. Falls man also einen Algorithmus entwickeln möchte, der zusätzlich auch die Modellierung übernimmt (also eine einfache, kodierte Eingabe der Probleminstanz in eine Graphrepresentation anwendet), die in (a) beschrieben wird, dann verändert sich die Laufzeit nicht:

$$O(m^2) + O(|V| + |E|) = O(m^2 + m^2) + O(m^2) = O(m^2). \tag{2}$$

Übung 8

Im *maximalen Teilsummenproblem* besteht die Eingabe aus einer Folge $A = (a_1, a_2, \ldots, a_n)$ von n ganzen Zahlen $a_i \in \mathbb{Z}$. Gesucht ist die maximale Summe einer zusammenhängenden Teilfolge $(a_l, a_{l+1}, \ldots, a_r)$ von A, also der Wert

$$\max\left\{ \sum_{i=l}^{r} a_i \;\middle|\; 1 \leq l \leq r \leq n \right\}. \tag{3}$$

(a) Bezeichne $R(k) \coloneqq \max\{ \sum_{i=l}^{k} a_i \mid 1 \leq l \leq k \}$ die maximale *rechte Randsumme* in $(a_1, a_2, \ldots, a_k)$. Das heißt die Teilsumme, die genau in a_k endet. Finden Sie eine rekursive Formulierung für $R(k)$.

(b) Entwerfen sie mittels dynamischer Programmierung einen effizienten Algorithmus zur Berechnung von $R(k)$.

(c) Bezeichne $S(k)$ den Wert der maximalen Teilsumme in $(a_1, a_2, \ldots, a_k)$. Finden Sie eine rekursive Formulierung für $S(k)$. Verwenden Sie dabei gegebenenfalls R.

(d) Entwerfen sie mittels dynamischer Programmierung einen effizienten Algorithmus zur Berechnung der maximalen Teilsumme von A.

Lösung 8

(a) Wir betrachten zunächst den Randfall $k = 1$. Hier gibt es nur eine einzige mögliche rechte Randsumme (nämlich a_1), so dass sich für die maximale rechte Randsumme $R(1) = \max\{ \sum_{i=l}^{1} a_i \mid 1 \leq l \leq 1 \} = \sum_{i=1}^{1} a_i = a_i$ ergibt.

Nun sei $1 < k \leq n$. Beachte zunächst, dass *jede* rechte Randsumme von $(a_1, a_2, \ldots, a_k)$ entweder

- nur aus a_k oder
- aus einer rechten Randsumme von $(a_1, a_2, \ldots, a_{k-1})$ zuzüglich a_k

besteht. Wir können $R(k)$ also durch das Maximum dieser beiden Fälle berechnen, sprich:

$$R(k) = \max\{ R(k-1) + a_k, a_k \}. \tag{4}$$

Zusammenfassend erhalten wir die Rekursion

$$R(k) = \begin{cases} a_1 & \text{, falls } k = 1 \\ \max\{\, R(k-1) + a_k, a_k \,\} & \text{, sonst.} \end{cases} \tag{5}$$

(b) Mit Gleichung (5) können wir leicht einen entsprechenden Algorithmus formulieren, der die Werte $R(i)$ für alle $1 \leq i \leq k$ mittels dynamischer Programmierung berechnet. Der gesuchte Wert ist dann $R(k)$, die maximale rechte Randsumme der Folge $(a_1, a_2, \ldots, a_k)$. Ein entsprechender Pseudocode ist in Algorithmus 3 zu finden.

Algorithmus 3: MaxRechteRandsumme(A, k)

```
1  R ← new Array(k)
2  R[1] ← A[1]
3  for i ← 2 to k
4      R[i] ← max { R[i − 1] + A[i], A[i] }
5  return R[k]
```

Die Laufzeit von Algorithmus 3 beträgt $\Theta(n)$, zum einen für die Allokation des Arrays, zum anderen für die for-Schleife, welche $\Theta(n)$-mal durchlaufen wird, wobei jeder Durchlauf konstante Zeit benötigt.

(c) Ähnlich wie im Fall der rechten Randsumme betrachten wir zunächst den Randfall $k = 1$. Wieder gibt es nur eine Teilsumme, so dass $S(1) = a_1$ gilt.

Für den Fall $1 < k \leq n$ stellen wir fest, dass *jede* Teilsumme von $(a_1, a_2, \ldots, a_k)$ in eine der folgenden beiden Kategorien fällt:

- Die Teilsumme enthält den Wert a_k. Insbesondere ist die Teilsumme dann eine rechte Randsumme von $(a_1, a_2, \ldots, a_k)$.
- Die Teilsumme enthält den Wert a_k nicht. Dann ist die Teilsumme auch eine Teilsumme von $(a_1, a_2, \ldots, a_{k-1})$.

Wir fassen dies wieder in einer Rekursion zusammen, indem wir den Fall $k = 1$ gesondert behandeln und aus den beiden Möglichkeiten oben die bessere auswählen:

$$S(k) = \begin{cases} a_1 & \text{, falls } k = 1 \\ \max\{\, S(k-1), R(k) \,\} & \text{, sonst.} \end{cases} \tag{6}$$

(d) Mit Gleichung (6) können wir leicht einen entsprechenden Algorithmus formulieren, der die Werte $S(i)$ für alle $1 \leq i \leq n$ mittels dynamischer Programmierung berechnet. Der gesuchte Wert ist dann $S(n)$, die maximale Teilsumme der Folge $(a_1, a_2, \ldots, a_n)$. Eine Besonderheit hierbei ist, dass wir (da unsere Rekursion für S auch auf R zurück greift) parallel zu S auch die Werte für R per dynamischer Programmierung berechnen. Ein entsprechender Pseudocode ist in Algorithmus 4 zu finden.

Algorithmus 4: MaxTeilsumme(A)

```
1  n ← length(A)
2  R ← new Array(n)
3  S ← new Array(n)
4  R[1] ← A[1]
5  S[1] ← A[1]
6  for i ← 2 to n
7      R[i] ← max { R[i − 1] + A[i], A[i] }
8      S[i] ← max { S[i − 1], R[i] }
9  return S[n]
```

Die Laufzeit von Algorithmus 4 beträgt $\Theta(n)$, zum einen für die Allokation der Arrays, zum anderen für die for-Schleife, welche $\Theta(n)$-mal durchlaufen wird, wobei jeder Durchlauf konstante Zeit benötigt.

Bemerkung. Ein paar Hintergrundinformationen und eine nette Anekdote zu diesem Problem bzw. zu diesem Algorithmus finden sich zum Beispiel auf der englischen Wikipedia unter dem Eintrag Maximum Subarray Problem. Der dort vorgestellte Algorithmus (Kadanes Algorithmus) ist in der Tat im Wesentlichen äquivalent zu Algorithmus 4 (obwohl man es ihm vielleicht nicht direkt ansieht).