Eidgenössische
Technische Hochschule
Zürich

Ecole polytechnique fédérale de Zurich
Politecnico federale di Zurigo
Federal Institute of Technology at Zurich

Institut für Theoretische Informatik
Peter Widmayer
Beat Gfeller

Prüfung
Datenstrukturen und Algorithmen
D-INFK
Musterlösung

verfasst von Beat Gfeller (allfällige Korrekturen bitte an gfeller@inf.ethz.ch)

19. August 2009

Aufgabe 1:

a) 11, 10, 6, 8, 17, 13, 12, 16, 18, 20

b)

5	9	7	11	42	10	8	12	18

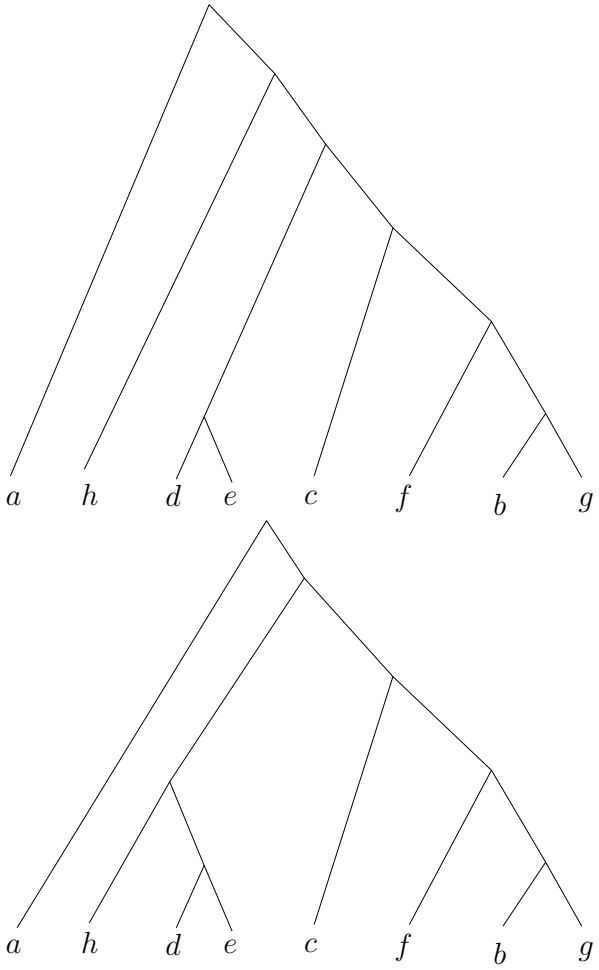
c) Es gibt zwei akzeptierte Lösungen:

8	13	16	9	5	4	19	23	33	81	69	42	80
8	13	16	9	5	4	19	23	33	80	81	69	42

d) Eine mögliche Tour ist A, F, E, D, C, B, A, es gibt aber auch andere Lösungen.

e) Eine Lösung ist $A, B, D, C, E, F, H, G, I$, es gibt aber einige weitere.

f) Es gibt genau zwei richtige Lösungen:

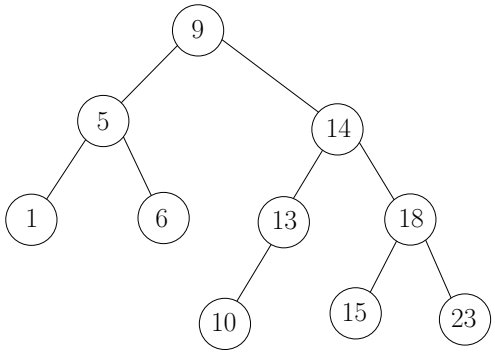


g) B (garantierter Wert: 42)

h) Es gibt nur eine gültige Lösung (egal, ob man nach links oder nach rechts sondiert):

11	1	5	17	4	12	89
0	1	2	3	4	5	6

i)



Aufgabe 2:

a) $\log\left(\frac{n^3}{2}\right)$, $\log^2 n$, $\sqrt{n}$, $\frac{n}{\log n}$, $\sum_{i=1}^{n} i$, $2^{\sqrt{n}}$

b) Durch Teleskopieren erhält man:

$$\begin{aligned}
T(n) =& 2T(n/4) + 3n + 5 \\
=& 2(2T(n/4^2) + 3n/4 + 5) + 3n + 5 \\
=& 2(2[2T(n/4^3) + 3n/4^2 + 5] + 3n/4 + 5) + 3n + 5 \\
=& 2^i T(n/4^i) + 3n(1 + 2/4 + 2^2/4^2 + \ldots + 2^{i-1}/4^{i-1}) + 5(1 + 2 + 2^2 + \ldots + 2^{i-1}) \\
=& 2^{\log_4 n} \cdot 2 + 3n\frac{1/2^{\log_4 n} - 1}{-1/2} + 5\frac{2^{\log_4 n} - 1}{1} \\
=& 2\sqrt{n} - 6n(1/\sqrt{n} - 1) + 5\sqrt{n} - 5 \\
=& 6n + \sqrt{n} - 5.
\end{aligned}$$

Beweis durch vollständige Induktion:

Verankerung: $n = 1$

$6 \cdot 1 + \sqrt{1} - 5 = 2 = T(1)$. $\rightarrow$ OK.

Induktionsschritt: $n/4 \rightarrow n$

$T(n) = 2T(\frac{n}{4}) + 3n + 5 = 2(6 \cdot \frac{n}{4} + \sqrt{n/4} - 5) + 3n + 5$ (nach Induktionsannahme)

$= 6n + \sqrt{n} - 5 = T(n)$. $\rightarrow$ OK.

c) $\Theta(n^2)$

Begründung (für Punkte nicht nötig): grob gesagt ist die Laufzeit $\sum_{i=1}^{n} i$. Alle $i > n/2$ aufsummiert ergeben bereits $\Omega(n^2)$. Dass die Laufzeit in $O(n^2)$ liegt, ist sofort klar.

d) $\Theta(\log n)$

Begründung (für Punkte nicht nötig): Im Prinzip ist dies genau der Vorgang, mit dem man als Betrunkener sein Hotel finden kann (wie in der Vorlesung vorgestellt), d.h. man sucht abwechselnd in beide Richtungen und verdoppelt jeweils die Schrittweite. Damit $i < 1$ oder $i > 2n$, muss die Schrittweite mindestens $\Omega(n)$ betragen, daher muss die Laufzeit mindestens $\Omega(\log n)$ sein. Spätenstens wenn die Schrittweite grösser als $2n$ ist, terminiert die Schleife, daher ist die Laufzeit auch höchstens $O(\log n)$.

e) $\Theta(n^2)$

Begründung (für Punkte nicht nötig): Anfangs ist $k \leq n$, daher wird zuerst k $\Theta(\log n)$ mal verdoppelt. Ab dann schwankt k bei jedem Durchgang um n herum, bleibt also stets $\Theta(n)$ für die restlichen $\Theta(n - \log n) = \Theta(n)$ Durchgänge. Da jeder Durchgang (wegen der j-Schleife) k Schritte benötigt, erhält man die Laufzeit $\Theta(n^2)$ (die Laufzeit der ersten $\log n$ Durchgänge ist kleiner als die nächsten, und haben keinen Einfluss auf die Laufzeit).

Aufgabe 3:

3 P a)

- Als Datenstruktur wird ein Array verwendet, in dem die n Buchungen nach Ankunftstag sortiert sind.
- Um das Array aufzubauen, muss man einfach die Buchungen sortieren. Z.B. mit Mergesort kostet dies $O(n \log n)$ Zeit.
- Um zu prüfen, ob eine gewünschte Buchung (x', y') möglich ist, sucht man im Array mit binärer Suche nach x'. Dort, wo die Suche endet, findet man zwei Buchungen: (i) Die Buchung $(\bar{x}, \bar{y})$ mit grösstem $\bar{x}$, für die gilt $\bar{x} \leq x'$; (ii) Die Buchung $(\hat{x}, \hat{y})$ mit kleinstem $\hat{x}$, für die gilt $x' \leq \hat{x}$. Die Buchung (x', y') ist genau dann noch möglich, wenn $\bar{y} \leq x'$ und $y' \leq \hat{y}$. (Es kann sein, dass es keine Buchung mit Eigenschaft (i) oder (ii) hat. Dann entfällt die entsprechende Bedingung.) Die Laufzeit beträgt $O(\log n)$.

6 P b) Als Datenstruktur kann man hier einen AVL-Baum verwenden, in welchem zudem jeder Schlüssel einen Verweis auf seinen Vorgänger (nächstkleinerer Schlüssel) und einen Verweis auf seinen Nachfolger (nächstgrösserer Schlüssel) im Baum hat (falls vorhanden). In diesem AVL-Baum speichert man alle aktuellen Buchungen, wobei als Schlüssel der Ankunftstag verwendet wird (man könnte übrigens auch den Abfahrtstag verwenden). Einfügen und löschen sind dann einfach die entsprechenden Operationen auf dem AVL-Baum. Für ein Query (x, y) muss man im Baum nach x suchen, und findet so den Vorgänger und Nachfolger von x im Baum, indem man bei dem Knoten, in dem die Suche endet, dem entsprechenden Verweis folgt (ähnlich wie bei a)).

Jeder Operation ist so in $O(\log n)$ Zeit ausführbar, wobei n die Anzahl aktuell vorhandener Buchungen ist.

Pseudocode (Java-ähnliche Syntax): Wir benutzen eine Klasse AVLTree, deren Knoten zudem per "previous" und "next" Referenzen in Inorder verkettet sind.

```
AVLTree tree = new AVLTree();
Insert(x,y) {
  if Query(x,y) { // nur einfügen, falls Buchung noch möglich
    tree.insert(x, Tuple(x,y));   // x ist der Schlüssel,
                                  // unter dem das Tupel(x,y) gespeichert wird
  } else {
    // melden: Buchung kann nicht eingefügt werden
  }
}
Delete(x,y) {
  if tree.contains(x) and tree.get(x).y == y {  // suche die Buchung
    tree.delete(x);  // es kann nur eine Buchung mit Startzeit x geben.
  } else {
    // melden: Buchung nicht vorhanden
  }
}
Query(x,y) {
  o = tree.search(x);
   // Annahme: search liefert das Element mit Schlüssel x,
   // falls es vorkommt, und sonst entweder den nächstgrösseren
   // oder nächstkleineren Schlüssel (wir ignorieren den Fall,
```

```
   // dass der Baum leer ist).
  if o.x == x return false;
  if o.x < x {
    return o.y <= x and (o.next == null or o.next.x <= y)
  } else { // o.x > x
    return (o.previous == null or o.previous.y <= x) and o.x <= y
  }
}
```

Aufgabe 4:

5 P a) Sei $W := \sum_{i=1}^{n} w_i$ die Summe der Gewichte aller Würfel. Die Grundidee ist, zu versuchen mit einer Teilmenge der Würfel das Gewicht $W/2$ zu erreichen. Wenn dies möglich ist, dann kann offensichtlich ein Gleichgewicht erreicht werden. Wenn W ungerade ist, kann ein Gleichgewicht sicher nicht erreicht werden, daher nehmen wir im folgenden an, dass W gerade ist.

Man benutzt ein $n \times (\frac{W}{2}+1)$-Tableau, in dem man an Position $E[i,w]$ abspeichert, ob das Gewicht w erreichbar ist, wenn man nur die Würfel $1, 2, \ldots, i$ benutzt. Die Rekursionsgleichung ist

$$E[i,w] = E[i-1,w] \vee (w_i \leq w \wedge E[i-1, w-w_i]),$$

mit den Startbedingungen $E[1,0] = true$, und $\forall w \neq 0 : E[1,w] = (w == w_1)$.

Man füllt das Tableau aus, indem man nach der Initialisierung für alle i von 2 bis n jeweils alle w von 0 bis $W/2$ durchgeht.

Die Laufzeit beträgt $O(nW)$ (oder etwas ungenauer $O(n^2 w_{\max})$, wobei $w_{\max} = \max_{i \in \{1,\ldots,n\}} w_i$).

1 P b) Man kann eine Lösung rekonstruieren, indem man beim Feld $E[n, W/2]$ startet: Wenn da *true* steht, existiert eine Lösung. Startend bei $i = n$ geht man in diesem Fall wie folgt vor: Falls $E[i-1,w] = true$ ist, kommt Würfel i nach links, und man rekonstruiert die Lösung weiter von $E[i-1,w]$ aus. Falls $E[i-1,w] = false$ ist, kommt Würfel i nach rechts, und man rekonstruiert die Lösung weiter von $E[i-1, w-w_i]$ aus.

3 P c) Die Grundidee bleibt die selbe: Sei $W := \sum_{i=1}^{n} w_i$ die Summe der Gewichte aller Würfel. Um ein Gleichgewicht zu erreichen, muss die Menge C das Gesamtgewicht $W/2$ haben, und die Mengen A und B je Gewicht $W/4$. Wenn W nicht durch 4 teilbar ist, ist sicher keine Lösung möglich. Falls W durch 4 teilbar ist, füllt man ein dreidimensionales Tableau aus, mit Dimensionen $n \times (W/2+1) \times (W/4+1)$: In $E[i,w,w']$ steht, ob man aus den Würfeln $1, 2, \ldots, i$ zwei Mengen bilden kann (ohne unbedingt alle zu verwenden), so dass die eine Menge Gesamtgewicht w hat und die andere Menge Gesamtgewicht w'. Nachdem dieses Tableau ausgefüllt wurde, kann die Lösung in $E[n, W/2, W/4]$ abgelesen werden (da das Gesamtgewicht aller Würfel gleich W ist).

Die Rekursionsgleichung ist

$$E[i,w,w'] = E[i-1,w,w'] \vee (w_i \leq w \wedge E[i-1, w-w_i, w']) \vee (w_i \leq w' \wedge E[i-1, w, w'-w_i]),$$

mit der Initialisierung $E[1,0,0] = true$, $\forall w > 0 : E[1,w,0] = (w == w_1)$, $\forall w' > 0 : E[1,0,w'] = (w' == w_1)$.

Das Tableau füllt man nach der Initialisierung wie folgt aus:

```
from
  i := 2 until i > n loop
  from w := 0 until w > W//4 loop
    from w' := 0 until w' > W//2 loop
       E[i,w,w'] = E[i-1,w,w'] or (w_i <= w  and  E[i-1, w-w_i, w'])
                               or (w_i <= w' and  E[i-1, w, w'-w_i])
      w' := w' + 1
    end
    w := w + 1
  end
  i := i + 1
end
```

Die Laufzeit beträgt hierfür $O(nW^2)$.

Aufgabe 5:

5 P a) Die Grundidee ist, einen Scanline-Ansatz zu verwenden. Man kann entweder eine horizontale oder eine vertikale Scanline verwenden. Wir betrachten im Folgenden eine vertikale Scanline, die von links nach rechts läuft. In der Scanline-Datenstruktur muss man die Menge der Segmente verwalten können, welche an der aktuellen Scanline-Position innerhalb des Zauns liegen. Als Haltepunkte benutzt man alle Anfangs- und Endpunkte der *vertikalen* Polygonkanten, sowie alle Kuh-Positionen. An den Haltepunkten, welche für eine Kuh-Position stehen, prüft man, ob der entsprechende Punkt (die Kuh) im Polygon liegt. An den anderen Haltenpunkten muss man die Scanline entsprechend updaten.

Wir müssen nun noch spezifizieren, welche Datenstruktur für die Scanline benutzt wird, und wie man darin Abfragen für Punkte durchführen kann. Man könnte hier einen Segmentbaum oder Intervallbaum verwenden. Ein Intervallbaum ist vorzuziehen, weil dieser zum Speichern der n Segmente nur $O(n)$ Platz braucht (im Gegensatz zum Segmentbaum, der $O(n \log n)$ benötigt). Um zu prüfen, ob ein Punkt auf der Scanline im Polygon liegt, führt man einfach eine Aufspiessanfrage aus. Diese findet höchstens ein aufgespiesstes Segment (wenn der Punkt im Polygon liegt), oder keines (wenn er ausserhalb liegt). An den Haltepunkten, die einer Polygon-Kante entsprechen, muss man unterscheiden:

- Wenn die Kante so liegt, dass rechts davon das Polygoninnere ist, dann muss man das entsprechende Segment einfügen. Falls es jedoch oberhalb und/oder unterhalb des Segments direkt benachbarte Segmente hat, dann entfernt man diese einzelnen Segmente, und fügt stattdessen ein Segment ein, welches all diese Segmente umfasst. (Dies erhält die Invariante, dass sich in der Scanline keine zwei separat gespeicherten Segmente berühren.)
- Wenn die Kante so liegt, dass links davon das Polygoninnere ist, dann mus man das entsprechende Segment entfernen. Dies geht aber nicht direkt, weil das entsprechenden Segment vielleicht Teil eines grösseren Segments in der Scanline ist. Daher sucht man zuerst mittels eine Aufspiess-Anfrage das entsprechende Segment in der Scanline, entfernt es, und fügt die bis zu zwei übrigbleibenden Segmente wieder ein.

Um den Scanline-Algorithmus durchzuführen, muss man die vertikalen Polygon-Kanten sammeln und nach x-Koordinate sortieren.

Einfachere Alternative: Da die in der Scanline gespeicherten Kanten sich nie berühren, kann man statt einem Segment oder Intervallbaum auch einen AVL-Baum benutzen, in dem die Segmente anhand ihrer beiden y-Koordinaten repräsentiert sind (und zwar so, dass man einer Koordinate ansieht, ob dies eine obere oder eine untere ist), und in dem zudem alle Knoten in Inorder-Reihenfolge doppelt verkettet sind. Für eine Aufspiessanfrage muss man dann einfach nach der y-Koordinate der Kuh-Position suchen: Wenn diese so liegt, dass die nächstuntere Koordinate eine "untere" ist, und die nächstobere eine "obere", dann liegt der Punkt im Polygon, andernfalls ausserhalb. Dies liefert eine einfachere Lösung, ist aber asymptotisch nicht effizienter. Wir beschreiben Sie hier vor allem, weil sie für Aufgabe c) sehr nützlich ist.

1 P b) Das Sortieren der Polygon-Kanten kostet $O(n \log n)$, das Sortieren der Kuh-Positionen kostet $O(m \log m)$, es gibt $m+n$ Haltepunkte, und an jedem wird $O(\log n)$ Zeit benötigt. Die Laufzeit ist somit $O(n \log n + m \log m)$.

3 P c) Wir erweitern die "Einfachere Alternative" aus Aufgabe a). Prinzipiell kann man dann genauso vorgehen wie bei orthogonalen Kanten, mit einem wichtigen Unterschied: Statt die unteren und oberen Enden der gespeicherten Segmente als feste Koordinaten abzuspeichern, benutzt man eine Geradengleichung (wie dies in der Vorlesung z.B. beim Schnittproblem für allgemeine Liniensegmente gemacht wurde). Dazu nehmen wir an, dass kein Polygonsegment genau vertikal liegt (falls dies der Fall ist, drehen wir einfach den ganzen Input leicht). Als Haltepunkte benutzt man wieder die Positionen der Kühe und alle Eckpunkte des Polygons. An jedem Eckpunkt muss man genau zwei Punkte (repräsentiert als Geraden) je entweder einfügen oder löschen. Die Aufspiessanfragen funktionieren dann wie in a). Die Laufzeit bleibt $O(m \log m + n \log n)$.

Bemerkung: Es scheint deutlich schwieriger, die Aufgabe c) mit einem Segmentbaum oder Intervallbaum zu lösen.