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
Holger Flier

Musterlösung
Datenstrukturen und Algorithmen

26. August 2010

verfasst von Holger Flier (allfällige Korrekturen bitte an hflier@inf.ethz.ch)

Aufgabe 1:

1 P a)

5	10	14	18	0	3	12	15	4	6	9

1 P b) H,R,O,M,A,L,G

1 P c) Mit Sondieren nach links:

21	15	23	8	11	16	
0	1	2	3	4	5	6 (Index)

Mit Sondieren nach rechts:

21	15	23		11	8	16
0	1	2	3	4	5	6 (Index)

1 P d) Das untenstehende Array enthält die Elemente eines Min-Heaps in der üblichen Form gespeichert. Wie sieht das Array aus, nachdem das Minimum entfernt wurde und die Heap-Bedingung wieder hergestellt wurde?

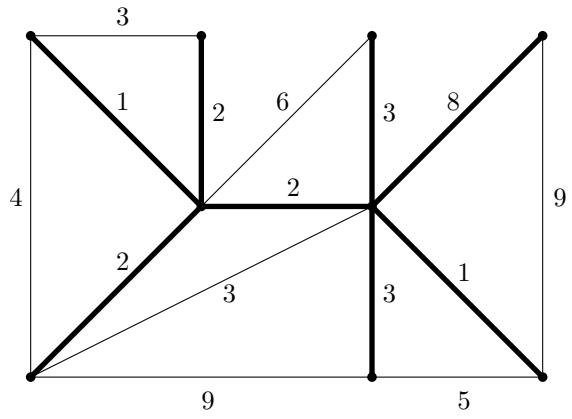
2	6	9	14	11	12	16	15	18
1	2	3	4	5	6	7	8	9

6	11	9	14	18	12	16	15

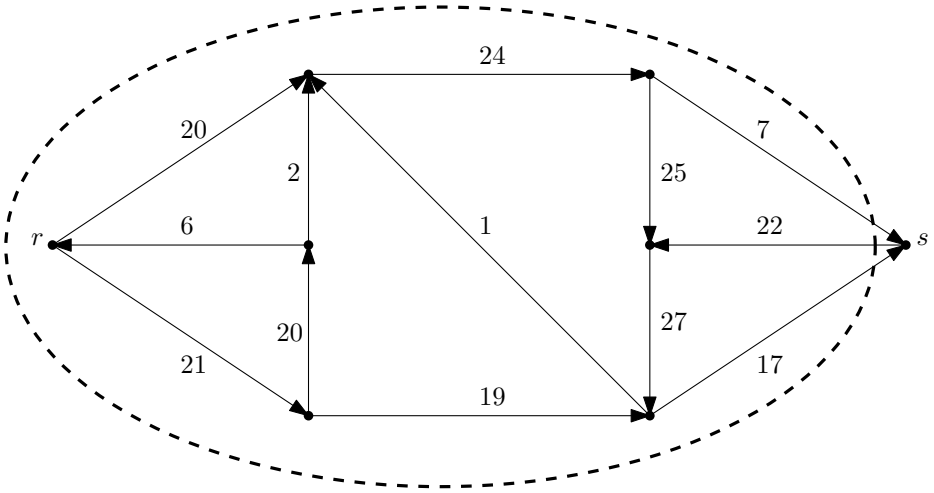
1 P e) Geben Sie für den folgenden azyklischen Graphen eine topologische Sortierung an.

C,D,A,F,B,E

1 P f) Markieren Sie im untenstehenden gewichteten Graphen die Kanten eines minimalen Spannbaums.



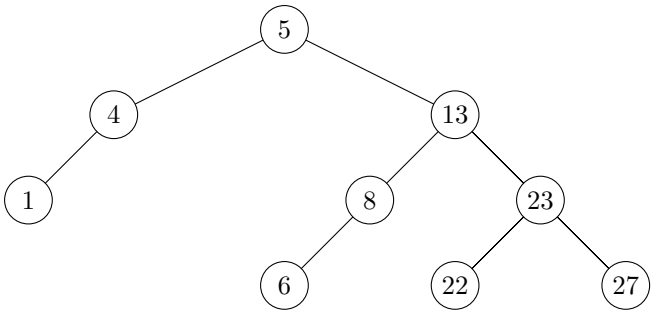
1 P g) Geben Sie den Wert eines maximalen (r, s)-Flusses in folgendem Netzwerk an und zeichnen Sie den entsprechenden minimalen Schnitt ein. Die Zahlen neben den Kanten geben die jeweilige Kantenkapazität an.



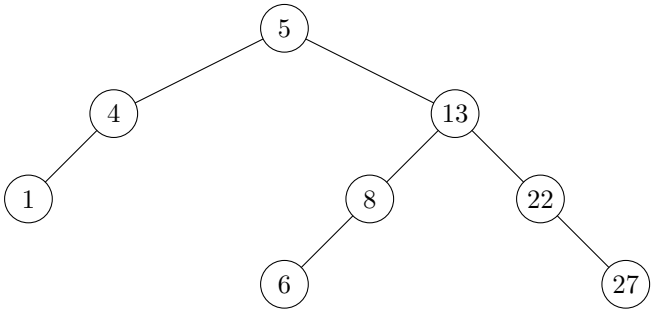
Der Flusswert beträgt 24.

1 P

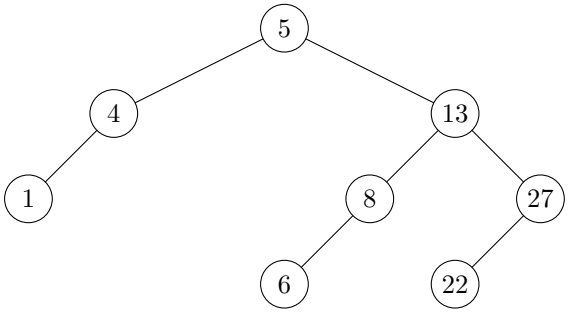
h) Nach Einfügen von 6:



Nach Löschen von 23 — mit symmetrischem Vorgänger:

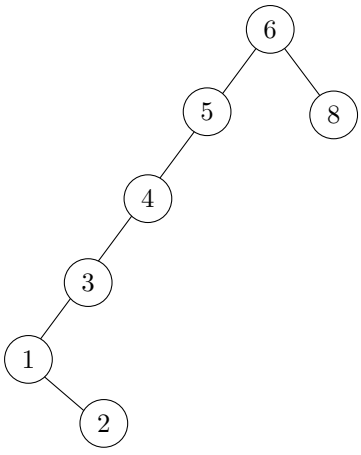


Nach Löschen von 23 — mit symmetrischem Nachfolger:



1 P

i)



Aufgabe 2:

1 P a) $\log(n)$, $\sqrt{n}$, $\log(n^n)$, $n^{\frac{3}{2}}$, $n^{\sqrt{n}}$, $n!$

3 P b)

$$\begin{aligned} T(n) &= 2T\left(\frac{n}{8}\right) + 1 \\ &= 2\left(2T\left(\frac{n}{8^2}\right) + 1\right) + 1 \\ &= 2\left(2T\left[2T\left(\frac{n}{8^3}\right) + 1\right] + 1\right) + 1 \\ &= 2^i \cdot T\left(\frac{n}{8^i}\right) + \sum_{j=0}^{i-1} 2^j \end{aligned}$$

Mit $i = \log_8(n)$ erhalten wir

$$\begin{aligned} &= 2^i + 2^i - 1 \\ T(n) &= 2\sqrt[3]{n} - 1. \end{aligned}$$

Beweis per Induktion:

Verankerung:

$$T(1) = 2 - 1 = 1 \qquad .$$

Induktionsschritt: von $n/8$ nach n:

$$T(n) = 2T\left(\frac{n}{8}\right) + 1 = 2(2 \cdot \sqrt[3]{\frac{n}{8}} - 1) + 1 = 2(2 \cdot \frac{\sqrt[3]{n}}{2}) - 2 + 1 = 2\sqrt[3]{n} - 1 \qquad .$$

1 P c) $\Theta(\log n)$, weil $\Theta(\log n)$ Iterationen über i und je genau eine Iteration über j ausgeführt werden.

1 P d) $\Theta(n^3 \log n)$, weil $\Theta(n)$ Iterationen über i, je $\Theta(n^2)$ Iterationen über j und je $\Theta(\log n)$ Iterationen über k ausgeführt werden.

1 P e) $\Theta(n \log n)$, da $\sum_{i=1}^{n} n/i = n \sum_{i=1}^{n} 1/i = \Theta(n \log n)$.

Aufgabe 3:

4 P a) Sei $W := \sum_{i=1}^{n} w_i$ die Summe der Werte der Münzen. Wir wollen nun berechnen, ob es eine Teilmenge der Münzen gibt, die genau den Wert $W/2$ hat. Offensichtlich gibt es nur in diesem Fall eine gerechte Aufteilung der Münzen auf zwei Piraten. Wir können im Folgenden annehmen, dass W gerade ist, da es sonst keine gerechte Aufteilung geben kann.

Wir erstellen eine Tabelle T der Grösse $n \times (W/2 + 1)$. Ein Eintrag $T[i, w]$ der Tabelle gibt an, ob das Gewicht w mit einer Teilmenge der Münzen $1, \ldots, i$ erreichbar ist ($true$) oder nicht ($false$). Die Rekursionsgleichung lautet:

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

Wir setzen $T[1, w] := true$ für $w \in \{0, w_1\}$ und $T[1, w] := false$ andernfalls. Danach füllt man die Tabelle, indem man für alle $2 \leq i \leq n$ jeweils alle $T[i, w]$ für $0 \leq w \leq W/2$ berechnet.

Die Laufzeit beträgt somit $O(nW)$.

1 P b) Man bestimmt die Menge A, indem man beim Eintrag $T[n, W/2]$ die Lösung rückverfolgt. Falls dort $false$ steht, gibt es keine Lösung. Ansonsten sei anfangs $i = n$ und $w = W/2$: Falls im Feld $T[i-1, w - w_i]$ $true$ steht, füge Münze i zu A hinzu und fahre bei $T[i-1, w - w_i]$ fort; andernfalls füge die Münze zu B hinzu und fahre bei $T[i-1, w]$ fort. Breche in jedem Fall bei $i = 0$ ab.

2 P c) Man verfährt analog zu Teil a), aber mit k Mengen $A_1, \ldots, A_k$ von Münzen, die alle den gleichen Wert haben sollen. Dementsprechend erstellt man ein k-dimensionales Tableau T, wobei die erste Dimension im Bereich $1 \leq i \leq n$ definiert ist, und die übrigen $k-1$ Dimensionen jeweils im Bereich $0, \ldots, W/k$. Ein Eintrag $T[i, j_1, \ldots, j_\ell, \ldots, j_{k-1}]$ gibt dabei an, ob es eine Teilmenge der Münzen $1, \ldots, i$ gibt, die sich so aufteilen lässt, dass Pirat ℓ Münzen im Wert von insgesamt j_ℓ erhält. Dabei erhält Pirat k genau die Münzen, welche die anderen Piraten nicht erhalten.

Die Rekursionsgleichung lautet dabei:

$$T[i, j_1, \ldots, j_\ell, \ldots, j_{k-1}] := T[i-1, j_1, \ldots, j_\ell, \ldots, j_{k-1}] \vee (w_i \leq j_\ell \wedge T[i-1, j_1, \ldots, j_\ell - w_i, \ldots, j_{k-1}])$$

Man initialisiert die Felder analog zum Teil a). Danach füllt man die Tabelle nach aufsteigendem i für alle gültigen Kombinationen $j_1, \ldots, j_k$ aus. Somit beträgt die Laufzeit

$$O(n \Pi_{\ell=1}^{k-1}(W/k)) = O(nW^k) \quad .$$

2 P d) Hier ist nach dem Approximationsalgorithmus für das Knapsack-Problem gefragt, der auf dem Greedy-Algorithmus basiert, siehe Übungsaufgabe 3.3. (Sortiere die Münzen nach nichtaufsteigendem spezifischen Wert w_i/g_i, und wähle aus dieser sortierten Folge die ersten, d.h. wertvollsten, ℓ Münzen, oder aber die $\ell + 1$-ste Münze aus, falls diese wertvoller ist als die Summe der ℓ Münzen.)

Aufgabe 4:

4 P a) Hier ist ein Scanline-Ansatz gefragt. Die vertikale Scanline verläuft dabei von links nach rechts. Haltepunkte sind die x-Koordinaten der Rechtecke. In der Scanline werden die gerade aktiven Rechtecke gehalten, also alle jene, welche die Scanline an ihrer aktuellen Position schneiden. Ist ein Haltepunkt an der linken Seite eines Rechtecks (kleinere der beiden x-Koordinaten), so wird das Rechteck der Menge der aktiven Rechtecke hinzugefügt. Andernfalls wird das entsprechende Rechteck aus der Menge entfernt. In jedem Fall wird die maximale Höhe aller aktiven Rechtecke bestimmt. Falls diese sich durch Hinzufügen oder Löschen ändert, werden zwei neue Punkte zum Umriss hinzugefügt. Beide haben als x-Koordinate die aktuelle Position der Scanline, und als y-Koordinate jeweils das alte und das neue Maximum. Als Datenstruktur eignet sich z.B. ein AVL-Baum. Dann können Einfügen, Löschen, und Bestimmen des Maximums in je $O(\log n)$ Schritten erledigt werden, wobei n die Anzahl der gegebenen Rechtecke bezeichnet.

1 P b) Die Laufzeit beträgt $O(n \log n)$, sowohl für das Sortieren der Rechtecke, als auch insgesamt für die Operationen an den $O(n)$ Haltepunkten.

3 P c) Auch hier ist ein Scanline-Ansatz gefragt. Es handelt sich um eine Erweiterung des Schnittpunktaufzählungsproblems, wobei wir die gegebenen Polygonzüge als Vereinigung von Liniensegmenten ansehen. Dabei besteht ein Polygonzug mit q Punkten aus $q-1$ Liniensegmenten, die sich in $q-2$ Punkten schneiden. Statt alle Schnittpunkte auszugeben, geben wir hier einen Schnittpunkt nur dann aus, sofern dieser am aktuellen Haltepunkt maximale Höhe hat.

Die Scanline verläuft wieder von links nach rechts. Haltepunkte sind die Endpunkte der Liniensegmente der Polygonzüge sowie alle Schnittpunkte dieser Liniensegmente. Da die Schnittpunkte erst im Verlauf des Verfahrens entdeckt werden, werden die Haltepunkte in eine Priority-Queue Q eingefügt. In der Scanline werden alle aktiven Liniensegmente der Polygonzüge in einer nach y-Koordinate geordneten Menge L gehalten. Dabei können L und Q je als AVL-Baum implementiert werden, so dass alle Operationen in $O(\log |L|)$ bzw. $O(\log |Q|)$ ausgeführt werden können.

An jedem Haltepunkt p wird eine der folgenden Operationen durchgeführt:

Beginnt ein Liniensegment s bei p, so wird es in die Menge der aktiven Liniensegmente L aufgenommen. Schneidet s seinen Vorgänger oder Nachfolger in L, so werden die entsprechenden Schnittpunkte als Haltepunkte in die Priority-Queue Q eingefügt.

Endet ein Liniensegment an dem Haltepunkt, so wird es aus der Menge L entfernt. Falls sich der Vorgänger und der Nachfolger von s schneiden, so füge einen neuen Haltepunkt zu Q hinzu.

Ist der Haltepunkt p der Schnittpunkt zweier Liniensegmente, so gehe wie folgt vor. Falls eines der am Schnittpunkt beteiligten Liniensegmente maximal bezüglich der y-Koordinate ist, füge einen entsprechenden Punkt zum Umriss hinzu. Vertausche nun die beiden Liniensegmente in L, und füge etwaige Schnittpunkte der beiden Liniensegmente mit dem neuen Vorgänger bzw. Nachfolger als Haltepunkte in Q ein.

Falls der erste Haltepunkt p' kein Schnittpunkt ist, füge einen entsprechenden Punkt $(p', 0)$ zum Umriss hinzu. Falls der letzte Haltepunkt p'' kein Schnittpunkt ist, füge einen entsprechenden Punkt $(p'', 0)$ zum Umriss hinzu.

1 P d) Bei k Schnittpunkten und N Liniensegmenten benötigt das Verfahren $O((N+k) \log N)$ Schritte, oder, etwas ungenauer, $O(N^2 \log N)$, da es höchstens $O(N^2)$ Schnittpunkte geben kann.

Aufgabe 5:

4 P a) Der Algorithmus lautet wie folgt, siehe auch Übungsaufgabe 4.2.

Algorithm 1: Längste absteigende Teilfolge

Input : Folge $S = (a_1, a_2, \ldots, a_n)$ mit $a_i \in \mathbb{N}$
Output: Längste absteigende Teilfolge $S' = (a_{i_1}, a_{i_2}, \ldots, a_{i_k})$

```
1  Arrays A[0..n], p[0..n]
2  for i := 1 to n do  A[i] := −∞, p[i] := null
3  k := 1  //Länge einer längsten absteigenden Teilfolge
4  A[1] := a_1
5  for i := 2 to n do
6      j := min{j : a_i > A[j], 1 ≤ j ≤ k + 1}
7      A[j] := a_i
8      p[i] := A[j − 1]
9      if j = k + 1 then k := k + 1
10 end
11 Array S'[1..k]
12 p := A[k]
13 ℓ := k
14 i := n
15 while ℓ > 0 do
16     while a_i ≠ p do i := i − 1
17     S'[ℓ] := a_i
18     p := p[i]
19     i := i − 1
20     ℓ := ℓ − 1
21 end
```

1 P b) Wenn man Zeile 6 mittels Binärer Suche implementiert, kommt der Algorithmus auf eine Laufzeit von $O(n \log n)$.

3 P c) Hier genügt schon ein Greedy-Ansatz:

Algorithm 2: Längste alternierende Teilfolge

Input : Folge $S = (a_1, a_2, \ldots, a_n)$ mit $a_i \in \mathbb{N}$
Output: Längste alternierende Teilfolge $S' = (a_{i_1}, a_{i_2}, \ldots, a_{i_k})$

```
1  Arrays A[0..n]
2  k := 1  //Länge einer längsten alternierenden Teilfolge
3  A[0] := a_1
4  if a_1 < a_2 then up:=true else up:=false
5  for i := 2 to n do
6      if up then
7          if (up ∧ A[k − 1] < a[i]) ∨ (¬up ∧ A[k − 1] < a[i]) then
8              A[k] = a[i]
9              k := k + 1
10         else
11             A[k − 1] = a[i]
12         end
13     end
14     S' := A[0..k − 1]
15 end
```

1 P d) Der Greedy-Algorithmus läuft in $O(n)$.