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

11. August 2011

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

Aufgabe 1:

Hinweise:

1. In dieser Aufgabe sollen Sie **nur die Ergebnisse** angeben. Diese können Sie direkt bei den Aufgaben notieren.
2. Sofern Sie die Notationen, Algorithmen und Datenstrukturen aus der Vorlesung "Datenstrukturen & Algorithmen" verwenden, sind Erklärungen oder Begründungen nicht notwendig. Falls Sie jedoch andere Methoden benutzen, müssen Sie diese **kurz** soweit erklären, dass Ihre Ergebnisse verständlich und nachvollziehbar sind.
3. Als Ordnung verwenden wir für Buchstaben die alphabetische Reihenfolge, für Zahlen die aufsteigende Anordnung gemäss ihrer Grösse.

1 P a) Das Verfahren von Karatsuba/Ofman zur Multiplikation ganzer Zahlen berechnet das Produkt zweier Zahlen rekursiv anhand einer Formel, die bis auf Addition und Multiplikation mit der Basis (hier: 10) drei Produkte enthält. Geben Sie zwei Zahlen x und y an, für welche diese drei Produkte $(74 \cdot 93)$, $(51 \cdot 80)$ und $(74 \pm 80) \cdot (51 \pm 93)$ sind.

$x = 7480$, $y = 9351$ oder $x = 5193$, $y = 8074$

1 P b) Geben Sie eine Folge von 5 Zahlen an, bei denen Bubblesort zum Sortieren genau 10 Vertauschungen von Schlüsseln durchführt.

Folge: 5,4,3,2,1

1 P c) 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?

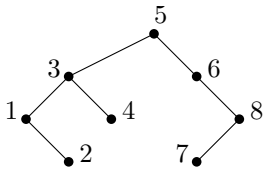
25	60	32	61	62	52	57	80	86
1	2	3	4	5	6	7	8	9

32	60	52	61	62	86	57	80

1 P d) Nehmen Sie an, die Schlüssel $7, 40, 13, 3, 24, 17, 1, 10, 14, 31, 15, 4, 23, 28, 11$ sollen in dieser Reihenfolge mittels Cuckoo-Hashing in Hashtabellen t_1 und t_2 eingefügt werden. Die Hashfunktion für Tabelle t_1 lautet $h_1(k) = k \mod 7$, die für Tabelle t_2 lautet $h_2(k) = 3k \mod 7$. Nehmen Sie ferner an, dass t_1 und t_2 jeweils die Grösse 7 haben. Wie lautet der erste Schlüssel in der oben genannten Folge, der nicht mehr eingefügt werden kann, ohne die Tabellen zu vergrössern (also eine `rehash` Operation auszuführen)?

Schlüssel: 17

1 P e) Zeichnen Sie den binären Suchbaum für die Schlüssel $1, 2, 3, 4, 5, 6, 7, 8$, dessen Preorder-Traversierung mit der Folge $5, 3, 1, 2$ beginnt und dessen Postorder-Traversierung mit der Folge $7, 8, 6, 5$ endet.

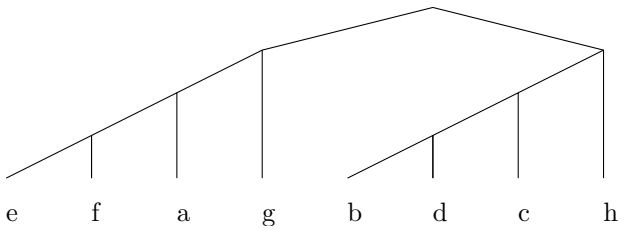


1 P f) Geben Sie eine Folge von höchstens 5 Zugriffen auf die Liste $A \to B \to C \to D \to E$ an, welche unter Benutzung der Move-To-Front-Regel genau 17 Vergleichsoperationen benötigt.

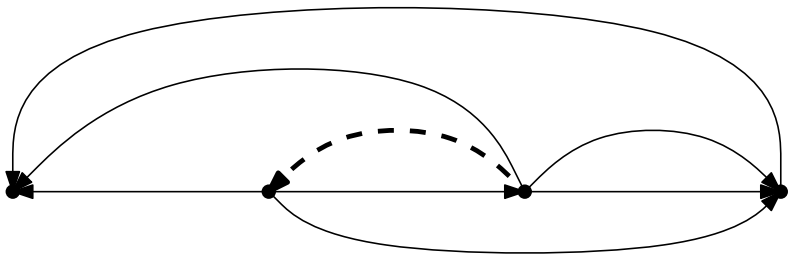
Folge von Zugriffen: E,D,C,C,C oder E,D,C,D

1 P g) Gegeben sind acht Schlüssel mit der relativen Anzahl der Zugriffe. Erstellen Sie mit Hilfe des Huffman-Algorithmus einen Codierungsbaum (Trie).

Schlüssel	a	b	c	d	e	f	g	h
Anzahl Zugriffe	41	33	70	39	23	25	78	98

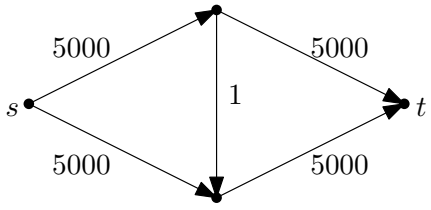


1 P h) Markieren Sie in folgendem Graphen $G = (V, E)$ eine kleinstmögliche Menge S an Kanten, so dass der Graph $G' := (V, E \setminus S)$ eine topologische Sortierung hat :

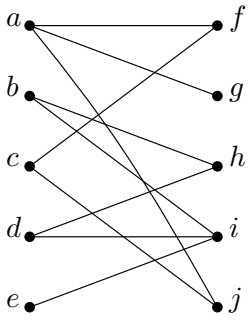


z.B.

1 P i) Geben Sie ein Beispiel für einen gerichteten Graphen $G = (V, E)$ an, der höchstens 6 Kanten hat und für den der Zunehmende-Wege-Algorithmus (Ford-Fulkerson) bei unglücklicher Wahl von zunehmenden Wegen bis zu 10000 Flusserhöhungen vornimmt, um einen maximalen Fluss zu berechnen. Geben Sie für jede Kante die Kapazität an. Kennzeichnen Sie auch den Startknoten s und Zielknoten t.



1 P j) Geben Sie eine Teilmenge der Knoten des folgendem bipartiten Graphen an, die mit dem Satz von Hall beweist, dass der Graph kein perfektes Matching hat.



b, d, e, h, i oder a, c, f, g, j

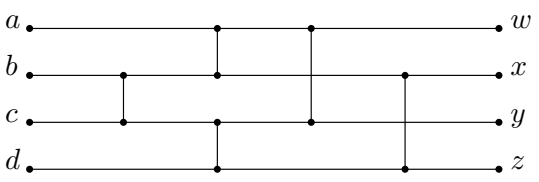
1 P k) Vervollständigen Sie die lückenhaften Zeilen des folgenden Pseudocodes, so dass der resultierende Algorithmus eine 2-Approximation für Minimum-Vertex-Cover liefert.

Input: Graph $G = (V, E)$
$S = \emptyset$;
$U = E$;
while $U \neq \emptyset$ **do**
  wähle eine Kante $e = (u, v) \in U$;
  $S = S \cup \{u, v\}$;
  $U = U \setminus \{e = (x, y) \in U \mid \{x, y\} \cap \{u, v\} \neq \emptyset\}$;
end
return S;

---

1 P l) Folgendes Netzwerk aus Sortierbausteinen ist kein Sortiernetzwerk. Geben Sie Werte für die Inputs a, b, c, d des Netzwerks an, welche durch das Netzwerk nicht korrekt sortiert werden. (Ein korrekt sortierter Output hat dabei die Form $w \leq x \leq y \leq z$.)

$a = 1$
$b = 3$
$c = 4$
$d = 2$



Aufgabe 2:

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

3 P b)

$$T(n) := \begin{cases} 2 + 2T(\frac{n}{4}) & n > 1 \\ 1 & n = 1 \end{cases}$$

Induktionsvermutung durch Teleskopieren: $T = 3\sqrt{n} - 2$

Induktionsverankerung: $T(1) = 3\sqrt{1} - 2 = 1$

Induktionsschritt: $T(n) = 2 + 2T(n/4) = 2 + 2(3\sqrt{n/4} - 2)) = 2 + 3\sqrt{n} - 4 = 3\sqrt{n} - 2$

1 P c)

```
from j := 1 until j > n loop
  from k := 1 until k > n loop
    k := k + j
  end
  j := j + 1
end
```

$O(n \log n)$

1 P d)

```
from j := 1 until j > n*n loop
  from k := 2 until k > n*n*n loop
    k := k * 2
  end
  j := j + 1
end
```

$O(n^2 \log n)$

1 P e)

```
from h := 1 until h > n loop
  from j := 1 until j*j > n*n loop
    j := j + 1
  end
  from k := 2 until k > n loop
    k := k * k
  end
  h := h + 10
end
```

$O(n^2)$

Aufgabe 3:

2 P a) Der Zustand, in welchem sich der Springer auf Feld (x, y) befindet und mit der Sprungweite (sx, sy) dorthin vom vorherigen Feld gelangt ist (ursprünglich (0,0) auf Feld s) kann als 4-Tupel (x, y, sx, sy) beschrieben werden. Der Zustandsraum ist dann der Graph $G = (V, E)$, wobei jeder Knoten aus V einem möglichen Zustand entspricht und jede Kante einem zulässigen Sprung zwischen zwei Feldern.

Von einem gegebenen Knoten (Zustand) $(x, y, sx, sy) \in V$ aus sind also genau die Knoten $(x', y', sx', sy') \in V$ erreichbar, für die gilt:

- $x' = x + sx'$, $y' = y + sy'$
- $(x', y') \in [1, \ldots, n] \times [1, \ldots, n]$ ist frei
- $sx' \in \{sx - 1, sx, sx + 1\} \cap \{-3, \ldots, 3\}$ und $sy' \in \{sy - 1, sy, sy + 1\} \cap \{-3, \ldots, 3\}$,

4 P b) Der folgende Algorithmus führt eine Breitensuche auf G ausgehend von s aus, wobei G nicht explizit erzeugt wird, sondern nur die besuchten Knoten. Da jeder Sprung genau einer Kante in E entspricht, findet der Algorithmus somit einen kürzesten Weg von s nach z. Die Koordinaten eines Feldes s notieren wir mit $(s.x, s.y)$, die eines Zustands p mit $(p.x, p.y)$. Ferner benutzen wir ein Array A, wobei Feld $A[p]$ für jeden Zustand p angibt, ob er schon besucht wurde. Wir bezeichnen mit $N(p)$ die Menge der von p aus erreichbaren Zustände, so wie unter a) beschrieben.

```
   Input: Graph G = (V, E), Startfeld s, Zielfeld z
 1 P = {(s.x, s.y, 0, 0)};
 2 Q = ∅;
 3 A[·, ·, ·, ·]=false // Initialisiere alle Felder von A.
 4 ℓ = 0;
 5 while P ≠ ∅ do
 6 |   for p ∈ P do
 7 |   |   if (p.x, p.y) = (z.x, z.y) then
   |   |   |   // Ziel erreicht, gib den Länge des kürzesten Pfads aus
 8 |   |   |   return "Kürzester Weg hat Länge ", ℓ
 9 |   |   end
10 |   |   for q ∈ N(p) do
11 |   |   |   if A[q] = false then
12 |   |   |   |   Q ← Q ∪ {q};
13 |   |   |   |   A[q] ← true;
14 |   |   |   end
15 |   |   end
16 |   end
17 |   P ← Q ;
18 |   Q ← ∅ ;
19 |   ℓ ← ℓ + 1 ;
20 end
21 return "Es gibt keinen Weg von s nach z";
```

1 P c) Der Algorithmus entspricht einer Breitensuche in G und läuft somit in $O(|V| + |E|)$. Da $|V| = O(ab)$ und $|E| = O(|V|)$, läuft der Algorithmus in $O(ab)$.

1 P d) Wir führen ein Feld $B[p]$ ein, das den Vorgänger von Zustand p auf einem kürzesten Weg von s nach p angibt. Der vollständige Algorithmus lautet nun:

```
   Input: Graph G = (V, E), Startfeld s, Zielfeld z
 1 P = {(s.x, s.y, 0, 0)};
 2 Q = ∅;
 3 A[·,·,·,·]=false // Initialisiere alle Felder von A.
 4 B[·,·,·,·] = ∅ // Initialisiere alle Felder von B.
 5 ℓ = 0;
 6 while P ≠ ∅ do
 7     for p ∈ P do
 8         if (p.x, p.y) = (z.x, z.y) then
               // Ziel erreicht, gib den kürzesten Pfad aus
 9             o[·] = ∅ // Array für kürzesten Weg
10             i ← ℓ ;
11             while p ≠ ∅ do
12                 o[i] ← p;
13                 p ← B[p] ;
14                 i ← i − 1;
15             end
16             return "Kürzester Weg der Länge ", ℓ, " ist ", o
17         end
18         for q ∈ N(p) do
19             if A[q] = false then
20                 Q ← Q ∪ {q};
21                 A[q] ← true;
22             end
23         end
24     end
25     P ← Q ;
26     Q ← ∅ ;
27     ℓ ← ℓ + 1 ;
28 end
29 return "Es gibt keinen Weg von s nach z";
```

1 P e) G hat im Allgemeinen keine topologische Sortierung, da Springen im Kreis (in Bezug auf Zustände) möglich ist, und G somit nicht azyklisch sein kann.

Aufgabe 4:

1 P a) Die Haltepunkte der Sweepline sind die Punkte $p_3, \ldots, p_n \in S$. Sie werden aufsteigend nach x-Koordinate bearbeitet.

1 P b) Sei p_i der aktuelle Haltepunkt der Sweepline, und δ der kleinste bisher berechnete Mindestabstand zwischen zwei Punkten. In der mit der Sweepline assoziierten Datenstruktur D werden nur solche Punkte $p_\ell \in S$ (mit $\ell < i$) gespeichert, für die $x_i - x_\ell \leq \delta$ gilt.

1 P c) Als Datenstruktur bietet sich für D z.B. ein AVL-Baum an, damit eine Bereichsabfrage in y-Richtung effizient möglich ist. Die Punkte verlassen D in der gleichen Reihenfolge, in der sie in D eingefügt werden. Daher bietet es sich an, die Knoten des Baums zusätzlich zu einer Liste zu verketten.

4 P d) Der Algorithmus lautet in Pseudocode wie folgt:

```
   Input: Menge S = {p_1, ..., p_n} von Punkten in der Ebene
 1 a = p_1;
 2 b = p_2;
 3 δ = dist(a, b);
 4 D ← {p_1, p_2} ;
 5 for p_i ∈ {p_3, ..., p_n} do
 6 |   entferne aus D alle Punkte p_ℓ mit x_ℓ < x_i − δ ;
 7 |   R ← {p_j ∈ D | |y_j − y_i| ≤ δ} // Bereichsanfrage
 8 |   for p_j ∈ R do
 9 |   |   if dist(p_i, p_j) < δ then
10 |   |   |   a = p_i ;
11 |   |   |   b = p_j ;
12 |   |   |   δ ← dist(p_i, p_j) ;
13 |   |   end
14 |   end
15 |   D ← D ∪ {p_i} ;
16 end
17 return "Das dichteste Punktepaar lautet:", (a,b);
```

3 P e) Die Laufzeit des Algorithmus beträgt $O(n \log n)$. Man beachte eine weitere Invariante, nämlich dass $\text{dist}(p_j, p_{j'}) \geq \delta$ für alle $p_j, p_{j'} \in D$. Da also auch die Punkte in $R \subseteq D$ paarweise mindestens den Abstand δ haben, folgt die konstante obere Schranke $|R| \leq \frac{2\delta^2}{\frac{1}{4}\pi\delta^2} =: c$. An jedem Haltepunkt werden nur $|R|$ Distanzberechnungen ausgeführt, also nur $O(n)$ insgesamt. Die Laufzeit der im Verlauf des Algorithmus ausgeführten Operationen auf der Datenstruktur D beträgt $O(n \log n)$ für Einfügen und Entfernen, sowie $O(\log n + |R_i|)$ für die Bereichsanfrage am i-ren Haltepunkt, wobei R_i die Menge R für den i-ten Haltepunkt bezeichnet.

Aufgabe 5:

a) Wir speichern die Punkte in einem balancierten Suchbaum, in dem die Punkte nach x-Koordinate geordnet sind. Da sich die x-Werte nicht ändern, ist dieser Baum fix. Wir können ihn z.B. wie einen 1-dimensionalen Range-Tree aufbauen, oder auch einen AVL-Baum benutzen. Nachdem dieser Baum berechnet wurde, speichern wir in jedem Knoten i eine zusätzliche Ganzzahl ab, die wir mit y_i^* bezeichnen.

   Die Invariante besagt, dass "y_i^* = die kleinste y-Koordinate eines Punktes, welcher im Teilbaum mit Wurzel i vorkommt."

b) Das Minimum kann gefunden werden, indem man ein 1-dimensionales Range-Query durchführt (wie in einem 1d-Range-Tree): D.h. man sucht im Baum nach $x_{\min}$ sowie nach $x_{\max}$, und besucht somit insgesamt $O(\log n)$ Knoten, deren Teilbäume alle Punkte mit x-Koordinaten im gesuchten Bereich einhalten.

   Genauer: die Suche geht zuerst zum "split-Punkt", wo sich die beiden Suchpfade trennen. Wenn ab da die Suche nach $x_{\min}$ in den linken Teilbaum absteigt, weiss man dass alle Punkte im rechten Teilbaum im gesuchten x-Bereich liegen. Analog für die Suche nach $x_{\max}$ (nur spiegelverkehrt). Anstatt nun sämtliche Punkte in diesen $O(\log n)$ Teilbäumen durchzugehen, berechnet man einfach das Minimum der y_i^*-Werte in all diesen Knoten. Somit erhält man den gesuchten y-Wert in Zeit $O(\log n)$.

c) Für die Operation `Update` sucht man zuerst den Knoten mit x-Koordinate x_i im Baum. Dann ändert man dessen y-Koordinate auf y_{new}^*. Dadurch ändert sich möglicherweise das Minimum in einigen Teilbäumen (aber nur in denen, welche den Punkt $(x_i, \cdot)$ enthalten. Um nun die Invariante wiederherzustellen, muss man den Suchpfad von unten nach oben zurücklaufen, und dabei alle y_i^*-Werte der Knoten auf diesem Pfad gegebenenfalls aktualisieren. Dies geht pro Knoten in konstanter Zeit, wie folgt: Am Knoten j berechnet man einfach $y_j^* = \max\{y_l^*, y_r^*, y_j^*\}$, wobei l der linke Sohn von j ist (falls vorhanden), und r der rechte Sohn von j ist (falls vorhanden, einer von beiden muss vorhanden sein). Für y_l^* und y_r^* gilt die Invariante schon, weil wir die Knoten von unten nach oben aktualisieren. y_j^* ist natürlich einfach die y-Koordinate des in Knoten j gespeicherten Punkts. Da jeder Knoten in $O(1)$ aktualisiert werden kann, benötigt das die Operation `Update` somit nur $O(\log n)$ Zeit.