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

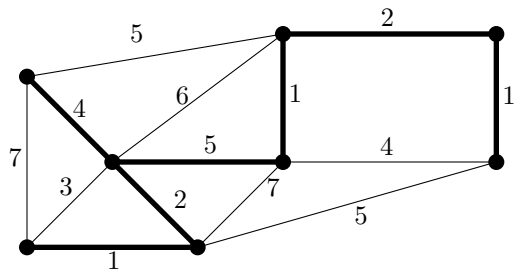
D-INFK

Musterlösung

6. Februar 2008

Aufgabe 1:

1 P a) Zeichnen Sie in den folgenden Graphen einen minimalen Spannbaum ein:



Hinweis: statt der im Bild gewählten Kante mit Gewicht 5 könnte auch eine der beiden anderen gewählt werden.

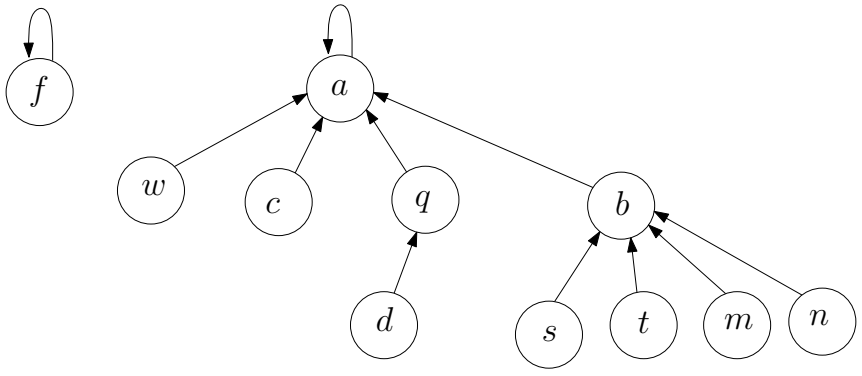
1 P b) Das untenstehende Array enthält die Elemente eines Min-Heaps in der üblichen Form gespeichert. Wie sieht das Array aus, nachdem das Minimum entfernt wurde und die Heap-Bedingung wieder hergestellt wurde?

3	5	9	22	11	14	13	32	50	20

5	11	9	22	20	14	13	32	50	

1 P c) 17, 10, 7, 1, 12, 15, 20, 18, 32

1 P d)



1 P e)

29	23	17	3	41	49	5	18	19	31	20
0	1	2	3	4	5	6	7	8	9	10

1 P f) Die kürzeste Folge von Zugriffen ist: C,B,A.

1 P g) Führen Sie auf dem gegebenen Array einen Aufteilungsschritt des Sortieralgorithmus Quicksort durch. Benutzen Sie als Pivot das am rechten Ende stehende Element im Array.

45	89	17	59	3	5	18	22	90	13	72	6	39

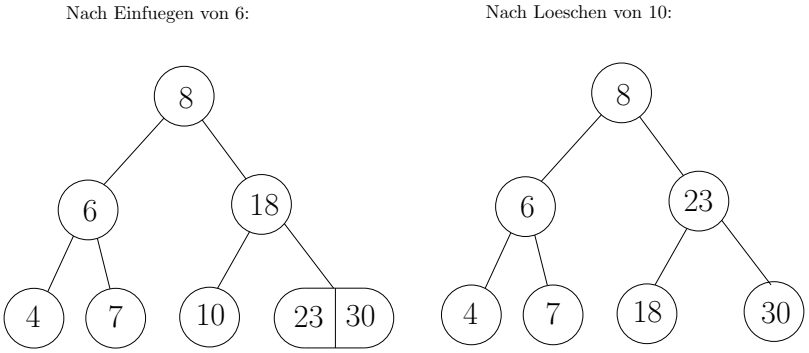
Wenn man das Vertauschen in-situ ausführt, erhält man folgendes Array:

6	13	17	22	3	5	18	39	90	89	72	45	59

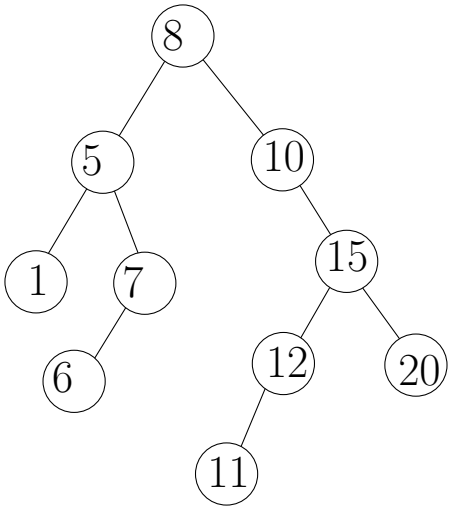
Alternative (nicht in-situ):

17	3	5	18	22	13	6	39	45	89	59	90	72

1 P h) Lösung:



1 P i) Lösung:



Aufgabe 2:

1 P a)

$$2^{1024}, \quad \log(n^{17}) \quad [= \Theta(\log n)], \quad \frac{n}{\log n}, \quad n\sqrt{n}, \quad \sum_{i=1}^{n} \frac{i}{2} \quad [= \Theta(n^2)],$$

$$2^{3\log_2 n} \quad [= (2^{\log_2 n})^3 = \Theta(n^3)], \quad 3^{\sqrt{n}}$$

3 P b)

$$T(n) = n + \frac{n}{2} \cdot \log_2(n)$$

Beweis durch Induktion:
1) Verankerung: $T(1) = 1 + \frac{1}{2} \cdot \log_2(1) = 1 + 0 = 1$. OK.
2) Induktionsschritt: Die Annahme stimme für $n/2$. Dann folgt:

$$T(n) = 2T(n/2) + n/2 = 2\Big(n/2 + \frac{n/2}{2} \cdot \log_2(n/2)\Big) + n/2$$

$$= n + n/2 \cdot \log_2(n/2) + n/2 = n + n/2 \cdot (\log_2(n) - \log_2(2)) + n/2 = n + n/2 \cdot \log_2(n).$$

1 P c) Äusserer Loop: $i = n, n-1, n-2, \ldots, n/2$
Innerer Loop: jeweils i Schritte
Also (angenommen, n sei gerade):

$$\sum_{i=n/2}^{n} i = (\sum_{i=1}^{n} i) - (\sum_{j=1}^{n/2-1} j) = n(n-1)/2 - (n/2-1)(n/2-2)/2 = \Theta(n^2).$$

1 P d) Äusserer Loop: $k = 1, 2, 4, 8, 16, \ldots, n$
Innerer Loop: jeweils $\log k$ Schritte, also $1, 2, 3, 4, 5, \ldots, \log n$. Also:

$$\sum_{j=1}^{\log n} j = (\log(n))(\log(n) - 1)/2 = \Theta(\log^2(n)).$$

1 P e) Innerer Loop: unabhängig von i, dauert $\log n$ Schritte, danach ist $j \cdot j > n$ also $j \approx 2\sqrt{n}$
Äusserer Loop: $i \approx 0, 2\sqrt{n}, 4\sqrt{n}, 6\sqrt{n}, \ldots \sqrt{n}\sqrt{n}$. ($\Theta(\sqrt{n})$ Durchgänge)
Also: Laufzeit $\Theta(\sqrt{n}\log n)$.

Aufgabe 3:

3 P a) Im Vorverarbeitungs-Schritt erstellt man ein Array `summe: ARRAY[INTEGER]`, welches an Position i die Distanz einer Wanderung vom westlichsten Ort nach Osten bis zum i-ten Ort wandert. Dieses Array kann man in einem Durchlauf des Arrays `orte` erzeugen, also in $O($`orte.count`$)$.

3 P b) Mit dieser Datenstruktur (dem Array `summe`) kann man Anfragen nun in $O(\log n)$ beantworten, wobei $n = orte.count$: dazu führt man im wesentlichen eine binäre Suche nach den Werten `summe`$[start] + D_{\max}]$ und `summe`$[start] - D_{\max}$ durch. Ausgehend von der gefundenen Position muss man dann noch einen Schritt nach Westen bzw. Osten gehen, um zu prüfen ob dieser Ort auch noch gültig wäre.

In Eiffel: (dist enthält die Teilsummen wie in a) beschrieben) (die Suche nach links/rechts wurde hier parametrisiert; -1 heisst nach links suchen, +1 nach rechts.) Benutzte Invariante: das Intervall [l,r] enthält stets einen gültigen Zielort, d.h. einer der nicht zu weit weg liegt.

```
finde_ort_simple(start: INTEGER; maxdist: INTEGER; dist: ARRAY[INTEGER]): INTEGER is
  local
      mp: INTEGER
  do
      Result := finde_ort(start, start, dist.upper, maxdist, dist, -1)
      tmp := finde_ort(start, dist.lower, start, maxdist, dist, +1)
      if (dist[start]-dist[tmp]).abs > (dist[start]-dist[Result]).abs then
          Result := tmp
      end
  end

finde_ort(start: INTEGER; ll: INTEGER; rr: INTEGER; maxdist: INTEGER;
          dist: ARRAY[INTEGER]; i: INTEGER): INTEGER is
  local
      m, l, r : INTEGER
  do
      from l := ll; r := rr until
          l = r  or r = l + 1
      loop
          m := (l + r ) // 2
          if (dist[start]-dist[m]).abs > maxdist then   -- m is too far away
              if i = -1 then
                  r := m+i
              else
                  l := m+i
              end
          else  -- m is still OK
             if i = -1 then
                 l := m
             else
                 r := m
             end
          end
```

```
      end
      if (dist[start]-dist[l]).abs <= maxdist then
          Result := l
      end
      if (dist[start]-dist[r]).abs <= maxdist and
         (dist[start]-dist[r]).abs > (dist[start]-dist[l]).abs then
          Result := r
      end
  end -- finde_ort
```

4 P c) Ein möglicher Algorithmus geht wie folgt vor: es gibt zwei Positionszeiger `l` und `r`, welche jeweils einen Index im Array `orte` angeben. Anfangs ist `l=orte.lower` und `r=orte.lower`. Nun wird r solange schrittweise erhöht, bis die Distanz (welche laufend aufsummiert wird) grösser als $D_{\max}$ ist. Der vorletzte Wert von r ist jetzt ein möglicher Kandidat für eine längstmögliche Wanderung. Nun wird r zurück auf diesen Wert gesetzt, und l um eins erhöht. Da die Wanderung l, r jetzt kürzer ist als die vorherige, ist dies auch eine gültige Wanderung. Jetzt wird wieder r schrittweise erhöht, bis die Wanderung l, r zu gross ist, dann wird wieder l erhöht, usw.

Als Pseudocode:

```
l := orte.lower
r := orte.lower
d := 0
best_d := 0
while (l < orte.upper) {
  // gehe nach rechts, solange die Wanderung nicht zu lang ist:
  while (r < orte.upper &&  d + orte[r] <= D_max) {
    d += orte[r]
    r += 1
  }
  // r ist jetzt der grösste Index, so dass l..r eine gültige Wanderung ist.
  if (d > best_d) {
    best_d := d
    best_l := l
    best_r := r
  }
  d -= orte[l]
  l += 1  // bewege Startort eins nach rechts
          // -> Wanderung l..r immer noch OK (aber ev. nicht mehr grösstmöglich).
}
```

Die Laufzeit dieses Algorithmus' ist $O(n)$, wobei n die Anzahl Orte ist. Dies sieht man wie folgt: die äussere Schleife wird $O(n)$ mal durchlaufen. Die innere while-Schleife wird höchstens n mal durchlaufen (und zwar insgesamt!), weil in jedem dieser Durchläufe r um 1 erhöht wird, und der Wert von r von einem äusseren Schleifendurchgang zum nächsten gleich bleibt. Daher ist die Laufzeit nicht etwa $O(n^2)$, sondern nur $O(n)$. Beachte: wenn man die Lösung von a),b) für jeden Startwert ausführt, kommt man auf eine Laufzeit von $O(n \log n)$.

Aufgabe 4:

2 P a) Man sortiert einfach alle Zahlen, und geht dann einmal linear durch. Wann immer eine Zahl in der sortierten Folge zum erstenmal auftritt, setzt man einen entsprechenden Zähler auf 0, und erhöht ihn mit jedem weiteren Vorkommen der Zahl in der sortierten Folge. Sobald eine höhere Zahl auftritt, gibt man die vorherige Zahl aus, falls sie mindestens dreimal vorkam. Laufzeit: $O(n \log n + n) = O(n \log n)$. (Sortieren z.B. mit Mergesort)

4 P b) Datenstruktur: ein balancierter Binärbaum, z.B. ein AVL-Baum. In diesem speichert man die Zahlen, sowie in jedem Knoten einen Zähler, der die Anzahl vorkommen "seiner" Zahl zählt. Implementation der Operationen: Zum Beantworten von MindDreimal sucht man im Binärbaum nach der Zahl, und liest deren Zähler ab. Falls der mindestens 3 ist, wird TRUE zurückgegeben, sonst FALSE (auch dann, wenn die Zahl gar nicht vorkommt).

Zum Einfügen sucht man zuerst nach der Zahl. Es gibt zwei Fälle: wenn die Zahl schon vorkommt, wird deren Zähler um 1 erhöht. Falls sie noch nicht vorkommt, wird ein neuer Knoten erstellt, und deren Zähler auf 1 gesetzt.

Zum Entfernen führt man ebenfalls eine binäre Suche nach der Zahl durch. Falls die Zahl vorkommt, dekrementiert man deren Zähler um 1. Falls der Zähler dabei auf 0 fällt, entfernt man diesen Schlüssel wie üblich, und führt die nötigen Rebalancier-Operationen durch.

Alle drei Operationen sind so in $O(\log n)$ Zeit möglich, wobei n die Anzahl *verschiedener* Zahlen in der Kollektion ist. Der Platzbedarf dieser Datenstruktur ist $O(n)$.

Hinweis: Wenn man keine Zähler verwendet, sondern mehrfache Zahlen auch mehrfach speichert, benötigen die Operationen mehr Zeit und die Datenstruktur braucht mehr Platz.

3 P c) Eine mögliche Lösung ist, einen Heap zu verwenden, indem ein Element pro verschiedene Zahl in der Kollektion gespeichert ist. Die Zahlen sind dabei im Heap nach ihrer Häufigkeit geordnet: Es muss im ganzen Heap gelten, dass ein Element mindestens so häufig vorkommt wie seine beiden Söhne. Auf dieser Datenstruktur kann Mal(k) in $O(g)$ ausgeführt werden, wobei g gleich der Anzahl ausgegebener Elemente ist: Der Heap wird von der Wurzel her per Tiefensuche traversiert, wobei man immer soweit in die Tiefe geht, bis man auf eine Zahl stösst, die weniger als k mal vorkommt (weil die Söhne von diesem Element dann auch nicht k-mal vorkommen können, "erwischt" man so alle Elemente).

Zusätzlich zum Heap benötigt man einen balancierten binären Suchbaum, in welchem man alle Zahlen speichert, und für jede Zahl einen Pointer auf das entsprechende Element im Heap speichert. Beim Einfügen bzw. Entfernen stellt man zuerst im Baum fest, ob die Zahl überhaupt vorkommt. Einfügen: falls die Zahl noch nicht vorkommt, erstellt man ein neues Heap-Element (x,1) und fügt es am Ende des Heaps ein (die Ordnung stimmt dann, weil ja alle Elemente mindestens einmal vorkommen). Falls die Zahl schon vorkommt, erhöht man den Zähler im entsprechenden Heap-Element. Danach muss die Heap-Bedingung u.U. wieder hergestellt werden, indem man dieses Element "hochblubbert".

Entfernen: Falls das Element nicht vorkommt (kann man im Baum prüfen), muss man nichts tun. Sonst dekrementiert man den entsprechenden Zähler im Heap, und muss dann dieses Element gegebenenfalls "versichern" lassen (hier ist wichtig, dass man das Element dabei immer mit dem häufigeren der beiden Sohn-Elemente vertauscht).

Einfügen und Entferenen sind so in $O(\log n)$ möglich, wobei n die Anzahl verschiedener Zahlen in der Kollektion sind (zum Zeitpunkt der Operation).
Hinweis: Es gibt auch ganz andere Möglichkeiten. Hier ist eine Datenstruktur, die für Mal(k) $O(\log(n) \cdot g)$ Zeit benötigt, auch OK.

Aufgabe 5:

3 P a) Pseudocode:

```
anzahl(M: ARRAY[INTEGER]; i: INTEGER; B: INTEGER): INTEGER is
do
  if (B = 0) return 1
  if (i < M.lower) return 0
  return anzahl(M, i, B-M[i]) + anzahl(M, i-1,B)
end
```

Aufruf mit

```
anzahl(M, M.upper, B)
```

Die asymptotische Laufzeit ist gleich der Anzahl Möglichkeiten, also $O(|M|^B)$, weil 1 der kleinstmögliche Münzwert ist (denn `M: ARRAY[INTEGER]`).

4 P b) Man benutzt ein zweidimensionales Array `A`, wobei `A[i,b] =` Anzahl Möglichkeiten, den Betrag b nur mit den ersten i Münzwerten (von 1 an indiziert) im Array M zu erreichen. Die Randbedingungen sind `A[i,0] = 1` für alle i und `A[0,b] = 0` für $b > 0$. Die Rekursionsgleichung ist `A[i,b] = A[i-1,b] + A[i,b-M[i]]`. Somit kann man die Tabelle Feld für Feld ausfüllen, mit konstanter Zeit pro Feld, also ist die Laufzeit insgesamt $O(|M| \cdot B)$.

1 P c) Man startet in `A[|M|,B]` ($i = |M|, b = B$). Falls die Zahl dort 0 ist, ist der Betrag nicht darstellbar. Ansonsten geht man entweder nach `A[i-1,b]` oder nach `A[i,b-M[i]]`, je nachdem welcher davon > 0 ist (einer ist es sicher, es können auch beide sein). So fährt man fort bis $b = 0$ erreicht wurde, und jedesmal, wenn man nach `A[i,b-M[i]]` geht (die 2.Variante), gibt man `M[i]` als eine Münze der Lösung aus. Dieser Prozess benötigt $O(|M| + B)$ Zeit.