Musterlösung Übungszettel 1
Algorithmen & Datenstrukturen für Bioinformatiker

Adrian Haß

5. November 2002

1 O-Notation

Lösung:

Wir definieren folgende Bezeichnungsweise:

$$f(n) \precsim g(n) :\Leftrightarrow f(n) = O(g(n))$$
$$f(n) \sim g(n) :\Leftrightarrow f(n) = \Theta(g(n))$$
$$f(n) \succsim g(n) :\Leftrightarrow f(n) = \Omega(g(n))$$

Zuerst nehmen wir einige elementare Umformungen vor, die es uns ermöglichen die Terme in eine dann offensichtliche Ordung zu bringen:

$$\sqrt{\log n} = \log^{1/2} n \tag{1}$$
$$\ln n = \frac{1}{\log e} \cdot \log n \tag{2}$$
$$4^{\log n} = (2^2)^{\log n} = 2^{2 \log n} = 2^{\log n^2} = n^2 \tag{3}$$
$$\sqrt{3}^{\log n^3} < 2^{\log n^3} = n^3 \tag{4}$$
$$(\frac{6}{3})^n = 2^n \tag{5}$$

Nun fällt es nicht schwer folgende Ordung einzusehen:

$$2^{2^n} \succsim (n+1)! \succsim n! \succsim n \cdot 2^n \succsim 2^n \overset{(5)}{\succsim} (\frac{4}{3})^n \succsim n^3 \overset{(4)}{\succsim} \sqrt{3}^{\log n^3}$$
$$\ldots \succsim n^2 \overset{(3)}{=} 4^{\log n} \succsim n \succsim \log^2 n \overset{(2)}{\succsim} \ln n \overset{(1)}{\succsim} \sqrt{\log n} \succsim \log^* n.$$

Bewertung:

Für mind. $\frac{i}{3}$ richtige Ordnungen gab es i Punkte. Bei falschen Begründungen $-\frac{1}{2}$ Punkte bis maximal $-\frac{3}{2}$.

2 Amortisierte Analyse

Lösung

Wir überlegen uns zunächst die Kosten für die "teueren" und die "billigen" Operationen getrennt:

- Es gibt in n Operationen offensichtlich $k = \lfloor \log_2 n \rfloor$ Zweierpotenzen, somit werden insgesamt $n - k$ "billige" Operationen ausgeführt.
- Die Zweierpotenzen Kosten jeweils $2^i, i \in \{1, \ldots, k\}$, also zusammen $\sum_{i=1}^{k} 2^i = 2^{k+1} - 2$.
- Amortisierte Kosten $a(n)$ sind jetzt also die Summe der "billigen" und "teuren" Operationen *geteilt durch* n:

$$c(n) = n - k + 2^{k+1} - 2 \leq n - \log_2 n + 2^{\log_2 n + 1} - 2 = 3n - \log_2 n - 2$$
$$\Rightarrow a(n) \approx \frac{3n - \log_2 n - 2}{n} \overset{n \to \infty}{\longrightarrow} 3$$

Bewertung:

Für die richtige Formel für "teure" Operationen gab es 1 Punkt, für die exakte Gesamtformel 1 Punkt und für die Amortisierung (durch n teilen) auch noch 1 Punkt.

3 AVL-Suchbäume

Lösung:

Zuerst macht man sich klar, dass es aufgrund der *Height-Balance Property* möglich ist, das oberste k (eigentlich das erste Element mit Schlüssel k) in $\log(n)$-Zeit zu finden. Nennen wir dieses jetzt erstmal k_1. Jetzt muss man nur noch sinnvoll in linearer Zeit die weiteren Auftreten detektieren. Erklären wir das einmal am Beispiel aller k's im linken Teilbaum unter k_1:

Wir besuchen das linke Kind von k_1 und sehen uns den Schlüssel an. Dabei können zwei Fälle auftreten:

$k_2 = k_1$: Dann wissen wir, dass im ganzen rechten Teilbaum unter k_2 nur k's stehen können, wenn man sich daran erinnert, dass eine *InOrder-Traversierung* eines Suchbaums die Elemente nach Schlüsseln aufsteigend geordnet ausgibt. Diesen können wir jetzt linear (mit einer der drei möglichen Traversierungen) durchsuchen.

$k_2 < k_1$: Wir wissen jetzt, dass wir nicht mehr links von k_2 suchen müssen, da dort nur noch Elemente mit Schlüsseln stehen, die erst recht kleiner als k sind. Also gehen wir einen Schritt nach rechts und beginnen von vorn.

Beispiel für den Pseudocode:
`AVLTree T` sei der gegebene Suchbaum, `Object k` sei der Schlüssel.

```
public Vector isEqual(Object k) {
   startK = T.findElement(k); //erstes Element
   Vector keys = new Vector();//Zielvector mit allen Elementen
   keys.add(startK);
   nextK = startK.leftChild();
   while(nextK!=null) {
      if (nextK.key() = k) { //erster Fall von oben
         keys.add(nextK);
         subTree = T.inOrder(nextK.rightChild());
         /* Wir nehmen an, dass dabei ein Iterator mit Knoten
         in der inOrder Reihenfolge des Teilbaums beginnend mit
         nextK.rightChild() ausgegeben wird.
         Ein Iterator ist ein Objekt, dass viele Elemente enth\"lt,
         durch die man mit *.nextElement() ``durchscrollen'' kann.*/
         while (subTree.hasMoreElements())
            keys.add(subTree.nextElement());
         } //durchwandern des Iterators
         nextK = nextK.leftChild();
         } else {
         //dass das kleiner als k bedeutet ist hoffentlich klar..
         nextKey = nextKey.rightChild(); //der Schritt nach rechts
   }}
   nextK = startK.rightChild();
   // das ganze f\"ur die andere Seite..
   while(nextK!=null) {
            if (nextK.key() = k) {
         keys.add(nextK);
         subTree = T.inOrder(nextK.leftChild());
         //jetzt ist der ganze linke Teilbaum mit k's voll.
         while (subTree.hasMoreElements())
            keys.add(subTree.nextElement());
         }
         nextKey = nextKey.rightChild();
         \}else {
         nextKey = nextKey.leftChild();
         // und wir gehen auch hier nach rechts.
   }}
   return keys;
}
```

Da insbesondere die inOrder-Traversierung eine Baumes in Linearzeit läuft, ist also gesichert, dass wir wenn der Teilbaum nur k's enthält auch die gewünschte Lauftzeit von $O(s)$ nicht überschreiten. Alles in Allem haben wir dann also $O(\log n + s)$ als Größenordnung.

Bei beliebigen Suchbäumen würde uns ein "entarteter" Suchbaum (z.B.: quasi-linear) beim Auffinden des obersten k's einen Strich durch die Rechnung machen.

Bewertung:

Suche bis oberstem k in $\log n$ (AVL-Eigenschaft) 1 Punkt; Wo befinden sich die gesuchten k's relativ zum ersten im Baum 1 Pkt.; Absuchen des Weges nach links bzw. rechts mit den richtigen Fällen 1 Pkt.; Verständlicher Pseudocode 1 Pkt.

4 Zirkuläre Strings

Lösung:

Betrachte String $S = \alpha\$\beta\beta$. Darüber lasse man den Z-Algorithmus laufen und suche nun ab der Position $|\alpha|+2$ bis zu $|\alpha|+|\beta|+1$ nach einer Z-Box der Länge $|\beta|$. Hat man mind. eine gefunden, dann ist β eine zirkuläre Verschiebung von α. Laufzeit ist aufgrund des Z-Algorithmus $O(|S|)$. Da $|S| = 3n$ ist, gilt lineare Laufzeit auch für diese Modifikation.

Bewertung:

Mögliches S 1 Pkt.; Linearzeitalgorithmus 1 Pkt.; Analyse 1 Pkt.

5 Anzahl der Matches

Lösung:

Wir überlegen uns zunächst mal für einen Text der Form $T = aaaa\ldots a$, was für Pattern man sich am besten erzeugen sollte. *O.B.d.A* sei die Länge von T eine Zweierpotenz: $|T| = 2^k$.

Nun erzeugen wir uns Pattern P_i der Länge 2^{i-1} als $P_1 = a$, $P_2 = aa$, $P_3 = aaaa$ usw. bis hin zu P_k. Deren Gesamtlänge ist offensichtlich:

$$n = \sum_{i=0}^{k-1} = 2^k - 1, \text{ somit } m + n = 2m - 1 = O(n).$$

Sehen wir uns nun die Matches an. Offensichtlich matcht P_1 2^k-mal. P_2 $(2^k - 2)$-mal und allgemein P_i $(2^k - 2^i)$-mal. Die Summe s der Matches ist also:

$$s = \sum_{i=0}^{k-1} 2^k - 2^i = k \cdot 2^k - \sum_{i=1}^{k-1} 2^i = (k-1) \cdot 2^k - 1 = O(n \cdot \log n).$$

Damit liegt die Anzahl der Matches "größenordnungsmäßig" (sprich: in o-Notation) über der der Gesamtlänge von Pattern und Text.

Leicht kommt man auch auf eine Verallgemeinerung in der a jetzt ein fest vorgegebenes Motiv ist. Dann können wir unsere Patternmenge zusätzlich um Teilstrings des Motivs erweitern, was im allgemeinen jedoch nur wenig bessere Ergebnisse bringt (Addition eines Linearfaktors).

Bewertung

Da hier viele nicht die richtige Lösung gefunden haben hat es ausgereicht mehr als $m+n$ Matches zu produzieren (1P). Dazu noch sinnvolle Erläuterungen (1P) und ein Beispiel (1P).