Ludwig-Maximilians-Universität München
Institut für Informatik
Prof. Dr. F. Kröger, Dr. M. Hölzl, F. Hacklinger

Übungen zu Informatik I (Lösungsvorschlag)

Aufgabe 11-1 Permutationen

Die Funktionen *member* und *remove* haben Komplexität $O(n)$ (wobei n die Länge der Eingabeliste ist). Sei $T(n)$ die Zeitkomplexität von *member* für Eingabelisten der Länge n. Zu zeigen ist: Es gibt Konstanten $n_0, c \in \mathbb{N}$, so dass für alle $n \geq n_0$ gilt $T(n) \leq cn$.

Wir zeigen durch Induktion nach n, dass für alle $n \in \mathbb{N}$ gilt: $T(n) \leq 3n+1$, daraus folgt dann, dass für alle $n \geq 1$ gilt $T(n) \leq 4n$. Seien $A = \mathbb{N}$, $M = \mathbb{N}$, $h : A \to M, h(n) = n$, $\prec = <$.

Induktionsannahme: Für alle Listen mit Länge n' mit $n' < n$ gilt $T(n') \leq 3n' + 1$.

Induktionsschluss: 1. Fall: $n = 0$. In diesem Fall wird nur der Test, ob die Eingabeliste leer ist, ausgeführt, somit ist $T(0) = 1$.

2. Fall: Sei $n > 0$. Zur Berechnung von $member(xy :: ys)$ für eine Liste $y :: ys$ der Länge n sind folgende Berechnungsschritte notwendig: (1) Test ob $y :: ys$ die leere Liste ist. (2) Test ob $x = y$ ist. (3) Falls das nicht zutrifft: Berechung von $member(x, ys)$. Nach Induktionsvoraussetzung benötigt für (3) höchstens $T(n-1) \leq 3(n-1)+1$ Berechnungsschritte. Im schlechtesten Fall benötigt man also insgesamt höchstens $3(n-1)+1+3 = 3n+1$ Schritte.

Somit hat *member* die Komplexität $O(n)$. Der Beweis für *remove* ist ähnlich.

Die Funktion *perm* hat im schlechtesten Fall Komplexität $O(n^2)$: Seien l, m Listen der Länge n. Falls m eine Permutation von l ist, wird jedes Element von l während der Abarbeitung betrachtet. Dabei muss für das i-te Element von l eine Liste der Länge $n-i$ mit *member* durchsucht werden; ebenso muss aus einer Liste der Länge $n-i$ ein Element mit *remove* gelöscht werden. Damit benötigt man für das i-te Element $c_{member}(n-i) + c_{remove}(n-i) = c(n-i)$ Schritte, insgesamt also:

$$
\begin{aligned}
\sum_{i=0}^{n} c(n-i) &= c \sum_{i=0}^{n} (n-i) \\
&= c \sum_{i=0}^{n} i \\
&= c \frac{n(n+1)}{2} \\
&= O(n^2) \quad .
\end{aligned}
$$

Eine andere Möglichkeit die geforderte Funktionalität zu implementieren ist, die Listen erst zu sortieren und dann die sortierten Listen auf Gleichheit zu vergleichen. Die Lösung mit allen benötigten Hilfsfunktionen ist:

```
fun insertel x [] = [x]
  | insertel x (y::ys) = if x <= y
 then x::y::ys
 else y::(insertel x ys);
```

```
fun insertlist [] a = a
  | insertlist (x::xs) a = insertlist xs (ins x a);

fun inssort xs = insertlist xs [];

fun listequal nil nil = true
  | listequal (x::xs) (y::ys) = x = y andalso listequal xs ys
  | listequal _ _ = false

fun perm2 l m =
    let val l' = inssort l
        val m' = inssort m
    in  listequal l' m' end
```

Da die Zeitkomplexität von *inssort* die Ordnung $O(n^2)$ hat, ist *perm2* nicht effizienter als die Funktion *perm* Wenn man Sortierfunktionen mit geringerer Zeitkomplexität verwendet, lässt sich aber auch die Effizienz von *perm2* verbessern.

Aufgabe 11-2 **Kleinstes gemeinsames Vielfaches**

a) Der einzige Funktionsaufruf in *kgv* ist der Aufruf von *loop*(m). Allerdings wird im rekursiven Aufruf von *loop* das Argument größer, so dass wir keine Induktion bezüglich der $<$-Relation über den natürlichen Zahlen durchführen können.

   Wir zeigen daher folgende Aussage: Für alle $m, n \in \mathbb{N}$ mit $m, n > 0$ gilt: *loop*(i) terminiert für alle i mit $0 \leq i \leq mn$. Seien $A = [1, mn]$, $M = \mathbb{N}$, $h : A \to M$ die Funktion $h(i) = mn - i$ und $\prec = <$.

   1. Fall: $i \bmod m = 0$ und $i \bmod n = 0$. Dann gibt es keinen rekursiven Aufruf von *loop*.

   2. Fall: $i \bmod m \neq 0$ oder $i \bmod n \neq 0$. Dann ist $i < mn$ (wegen $mn \bmod m = 0$ und $mn \bmod n = 0$) und somit $i + 1 \in A$. Es gibt einen rekursiven Aufruf der Form *loop*$(i + 1)$ und es gilt $h(i + 1) = mn - (i + 1) < mn = h(i)$.

b) Für voneinander verschiedene Primzahlen m, n ist mn das kleinste gemeinsame Vielfache. Da das Argument von *loop* bei jedem Aufruf um 1 erhöht wird, werden mn rekusive Aufrufe von *loop* ausgeführt. Also hat *kgv* die asymptotische Komplexität $O(mn)$.

c)
```
fun euclid m n =
    let fun loop k l = if l = 0
                       then k
                       else loop l (k mod l)
    in if m < n then loop n m else loop m n end
fun kgv2(m,n) = let val ggt = euclid m n
                    val m' = m div ggt
                    val n' = n div ggt
                in  ggt * m' * n' end
```

Aufgabe 11-3 **Revertieren von Listen**

a) Nach Voraussetzung gibt es Konstanten $n_0, c \in \mathbb{N}$, so dass zur Berechnung von $l_1 @ l_2$ weniger als cn Schritte benötigt werden, falls l_1 mehr als n_0 Elemente hat. Wir zeigen durch Induktion, dass es dann Konstanten n_1 und c' gibt, für die gilt: zum Revertieren einer Liste der Länge $n \geq n_1$ werden weniger als $c'n(n + 1)/2$ Schritte benötigt: Für $n = 0$ wird kein

Funktionsaufruf durchgeführt. Sei für alle Listen der Länge n die Anzahl der benötigten Berechnungsschritte kleiner als $c'n(n+1)/2$, und sei $x :: xs$ eine Liste der Länge $n+1$. Dann werden zur Berechnung von $rev(x :: xs)$ folgende Berechnungsschritte durchgeführt:

- ein Aufruf von :: (zur Berechnung von $[x]$), der 1 Schritt benötigt;
- ein Aufruf von @ mit erstem Argument der Länge n, dazu werden 1 Schritt für den Funktionsaufruf sowie cn Schritte zur Berechnung von @;
- ein Aufruf von *rev* mit einer Liste der Länge n, dazu wird 1 Schritt für den Funktionsaufruf und (nach Induktionsvoraussetzung) $c'n(n+1)/2$ Berechnungsschritte benötigt.

Also werden insgesamt

$$cn + c'n(n+1)/2 + 3$$

Berechnungsschritte ausgeführt. Mit $n_1 = \max(n_0, 3)$ und $c' \geq 2(c+1)$ gilt für alle $n \geq n_1$:

$$\begin{aligned}
cn + \frac{c'n(n+1)}{2} + 3 &\leq (c+1)n + \frac{c_3 n(n+1)}{2} \\
&\leq \frac{c_3 n}{2} + \frac{c_3 n(n+1)}{2} \\
&= \frac{c_3 n + c_3 n(n+1)}{2} \\
&= \frac{c_3 n(1+n+1)}{2} \\
&< \frac{c_3 (n+1)(n+2)}{2}
\end{aligned}$$

Somit gilt: die Komplexität von *rev* ist $O(n^2)$.

b) Einbettung: $reviter(x :: xs, a) = a@rev(xs)@[x]$ und $reviter(nil, a) = a$, also

```
fun reviter nil a = a
  | reviter (x::xs) a = reviter xs (x::a);
fun rev' xs = reviter xs [];
```

c) Intuitiv: Die Funktion *reviter* wird für jedes Element der Eingabeliste genau einmal aufgerufen, also hat rev' Zeitkomplexität $O(n)$.

Beweis: Sei $T(n)$ die Zeitkomplexität für *reviter* in Abhängigkeit von der Länge n der ersten Eingabeliste. Wir zeigen durch Induktion nach n, dass gilt: $T(n) \leq 2n$ für alle $n \in \mathbb{N}$. Für $n = 0$ wird keine Berechnung ausgeführt. Für $n > 0$ erfolgt ein Aufruf von *reviter* bei dem das erste Argument die Länge $n-1$ hat (dieser Aufruf benötigt nach IV höchstens $2(n-1)$ Schritte), und ein Aufruf von ::. Somit gilt: $T(n) = 1 + T(n-1) + 1 \leq 2(n-1) + 2 = 2n$.

Aufgabe 11-4 Größe einer endlichen Menge

Nach dem Schema der Vorlesung:

$$setsizerep(set, k) = k + setsize(set) = \begin{cases} k & \text{falls } set = \emptyset \\ \underbrace{k + 1 + setsize(delete(element, set))}_{setsizerep(delete(element,set),k+1)} & \text{mit } element \in set \end{cases}$$

```
use "set.sml";
```

```
fun setsizerep(set,k) = if isemptyset(set) then k
  else let val element = any(set) in
   setsizerep(delete(element, set), k + 1)
       end;

val testset = insert(1, insert(2, insert(3, insert(4, emptyset))));

setsizerep(testset, 0);
```

Aufgabe 11-5 **Suchen in binären Bäumen**

```
use "bintree.sml";

fun enthaltenrep(a, trees, result) =
    if trees = nil then result
    else if hd(trees) = emptybt then enthaltenrep(a, tl(trees), result)
    else enthaltenrep(a, left(hd(trees))::(right(hd(trees))::tl(trees)),
      result orelse root(hd(trees)) = a);


val tree = build(1,build(2,emptybt, build(3,emptybt,emptybt)),
 build(4, build(6,emptybt,emptybt), build(1,emptybt,emptybt)));

(* Aufruf *)
enthaltenrep(a, [tree], false);
```