Grundlagen der Informatik 1
WS 2008/09

Prof. Mühlhäuser, Dr. Rößling, Melanie Hartmann, Daniel Schreiber
http://proffs.tk.informatik.tu-darmstadt.de/gdi1

Übung 6 Version: 1.2 23.11.2008

---

1 Mini Quiz

Die abstrakte Laufzeit eines rekursiven Algorithmus...

- ☐ lässt sich mit einer Kostenfunktion $T(n)$ abschätzen.
- ☐ kann durch mehrere Testeingaben exakt ermittelt werden.
- ☐ kann nie einer Komplexitätsklasse zugeordnet werden.
- ☐ ist der Quotient aus Eingabegröße und der Zahl der Rekursionsschritte.
- ☐ kann (ungefähr) durch die Zeilenanzahl des Codes bestimmen werden.

Komplexität:

- ☐ Die O-Notation ist generell nur für große Eingaben aussagekräftig.
- ☐ Algorithmus $A \in O(n)$ terminiert für alle Eingaben schneller als Algorithmus $B \in O(n^2)$.
- ☐ QuickSort liegt in $O(n\,log(n))$.
- ☐ $n^3 \in O(n^n)$
- ☐ $O(log(n^{1000000})) \subset O(n)$
- ☐ $O(n) = O(0,0123456789n)$
- ☐ $10000000000000000000001 \in O(1)$

Akkumulatoren:

- ☐ Funktionen im Akkumulator-Stil sind immer schneller als Funktionen ohne Akkumulator.
- ☐ Manche Funktionen lassen sich nur mit Akkumulatoren implementieren.
- ☐ Ein Akkumulator wird nur nur bei Problemen, die generative Rekursion nutzen, benötigt.
- ☐ Akkumulator-Stil führt unter Umständen zu einem Effizienzgewinn.

2 Fragen

1. Welche Schritte sind nötig, um eine Funktion im Akkumulator-Stil zu implementieren?
2. Was versteht man unter der Akkumulator-Invarianten?

3. Zeigen oder widerlegen Sie folgende Behauptung: Man kann einen Sortieralgorithmus, dessen Mechanismus auf dem Vergleich zweier Zahlen beruht, in der Komplexitätsklasse $O(log(n))$ implementieren. **Hinweis**: Dieses Problem soll Ihnen zeigen, an welche Grenzen Algorithmen manchmal stossen. Den Beweis für die Aufgabe können Sie informell (also: mathematisch, zeichnerisch oder mit eigenen Worten) beschreiben. Stellen Sie sich eine Liste von Zahlen vor, die sortiert werden müssen. Wie gross ist der Aufwand, wenn jede Zahl in der Liste wenigstens einmal "angefasst" werden muss?

4. Angenommen, wir haben eine Liste mit 30 identischen Elementen. Wie verhält sich die Laufzeit von QuickSort in diesem Fall? Wäre es ratsamer, hier ausnahmsweise einen Algorithmus mit schlechterer Komplexität zu wählen?

5. Fassen Sie bitte in Ihren eigenen Worten, wofür $\Theta(..)$, $\Omega(..)$ und $O(..)$ stehen.

6. Welche der folgenden Aussagen gelten?
   a) $O(3\,n\,log_4(n)) \subseteq \Theta(9\,n\,log_{10}(n))$
   b) $O(3\,n\,log_4(n)) \subseteq O(9\,n\,log_{10}(n))$
   c) $\Theta(3\,n\,log_4(n)) \subseteq O(9\,n\,log_{10}(n))$?

7. Zeigen Sie, dass aus $f(n) \in O(log_a(n))$ folgt $f(n) \in O(log_b(n))$ für beliebige a, b.

3 Rekursion vs. Iteration: Fakultät (K)

Die Fakultät von n (geschrieben $n!$) ist wie folgt definiert:

$$n! = \begin{cases} 1 & falls\, n\, =\, 0 \\ n \cdot (n-1)! & sonst \end{cases}$$

1. Implementieren Sie die Funktion `fak-rec`, die eine Zahl n konsumiert und $n!$ zurückliefert. Verwenden Sie Rekursion! Markieren Sie die Stelle in Ihrer Lösung, in der die Rekursion terminiert (Rekursionsanker) und die Stelle an der die Funktion sich selbst rekursiv aufruft.

2. Veranschaulichen Sie nun Ihre Lösung (genauso wie in der Folie T7.7) indem Sie als Parameter für die Fakultät die Zahl 6 nehmen.

3. Implementieren Sie nun die iterative Variante `fak-iter` der Fakultät unter Verwendung eines Akkumulators. Der Akkumulator soll das bisher aufmultiplizierte Produkt enthalten. Initialisieren Sie den Akkumulatorwert mit 1, multiplizieren Sie mit n, das Ergebnis multiplizieren Sie mit $n-1$, das neue Ergebnis mit $n-2$, solange bis Sie bei $n - n + 1 = 1$ angelangt sind.

4. Veranschaulichen Sie Ihre Lösung wieder mit der Zahl 6.

5. Vergleichen Sie beide Varianten hinsichtlich der Länge des Codes, der Komplexität der Berechnung und der Anzahl der im Speicher zu haltenenden Elemente pro Rechenschritt.

4 Rekursion vs. Iteration: Fibonacci (K)

In dieser Aufgabe sollen Sie, wie in Aufgabe 3, die Fibonacci Funktion erst rekursiv, dann iterativ implementieren. Die Fibonacci Funktion ist wie folgt definiert:

$$fib(n) = \begin{cases} fib(n-1) + fib(n-2) & n \geq 2 \\ 1 & sonst \end{cases}$$

1. Implementieren Sie die Funktion `fib-rec: number -> number`, die die n-te Fibonaccizahl rekursiv berechnet. Zeichnen Sie einen Rekursionsbaum für die 6. Fibonacci Zahl und beurteilen Sie die Effizienz Ihrer Lösung. Zur Erinnerung, die ersten acht Fibonacci Zahlen lauten (beginnend mit der 0ten): 1, 1, 2, 3, 5, 8, 13 und 21.

2. Implementieren Sie nun die Fibonacci-Funktion iterativ und beurteilen Sie einen Vorteil der iterativen Lösung gegenüber der rekursiven Lösung. Verwenden Sie Akkumulatoren in Ihrer Lösung.

5 Akkumulatoren

Lesen Sie in T8.24 über die Funktion `invert` mit Akkumulator.

1. Schreiben Sie jetzt eine Funktion `invert2`, die *ohne* Akkumulator auskommt. Vervollständigen Sie dazu folgenden Code, ohne `append` oder eine ähnliche bereits verfügbare Funktion zu verwenden:

```
1  ;; invert2 : (listof X) -> (listof X)
2  ;; construct the reverse of list lst
3  ;; example: (invert2 (list 'A 'B 'C)) should return (list 'C 'B 'A)
4  (define (invert2 lst)
5      ;; rcons : (listof X) X -> (listof X)
6      ;; appends el to the end of list lst.
7      ;; example: (rcons (list 'A 'B) 'C) should be (list 'A 'B 'C)
8    (local (
9      (define (rcons lst el)
10               (...) ))
11     (...) ))
```

2. Vergleichen Sie Ihre Implementierung `invert2` mit `invert`. Welcher Algorithmus liegt in welcher Komplexitätsklasse?

6 Rekursion und Komplexität

Die Ackermannfunktion ist eine 1926 von Wilhelm Ackermann gefundene mathematische Funktion, die in der theoretischen Informatik eine wichtige Rolle bezüglich Komplexitätsgrenzen von Algorithmen spielt. Sie wird als Benchmark zur Überprüfung rekursiver Prozeduraufrufe verwendet, wenn man die Leistungsfähigkeit von Programmiersprachen oder Compilern überprüfen will. Implementieren Sie die vereinfachte Ackermann Funktion nach Péter rekursiv anhand folgender Definition:

$$\begin{aligned} Ack(0,m) &= m+1 \\ Ack(n,0) &= Ack(n-1,1) \\ Ack(n,m) &= Ack(n-1,Ack(n,m-1)) \end{aligned}$$

Berechnen Sie dann den Aufruf: Ack(3,4). Berechnen Sie nun die ersten Schritte des Aufrufs Ack(4,2). Was fällt Ihnen hierbei sehr schnell auf? Ist diese Funktion bezüglich der Komplexität von praktischem Nutzen?

Hausübung

Die Vorlagen für die Bearbeitung werden im GdI1-Portal bereitgestellt. Kommentieren Sie Ihren selbst erstellten Code. Die Hausübung muss bis zum Abgabedatum im GdI1-Portal abgegeben werden. Der Fachbereich Informatik misst der Einhaltung der Grundregeln der wissenschaftlichen Ethik großen Wert bei. Zu diesen gehört auch die strikte Verfolgung von Plagiarismus. Mit der Abgabe Ihrer Hausübung bestätigen Sie, dass Sie bzw. Ihre Gruppe alleiniger Autor des gesamten Materials sind. Falls Ihnen die Verwendung von Fremdmaterial gestattet war, so müssen Sie dessen Quellen deutlich zitieren.

Abgabedatum: Fr, 05.12.08, 16:00 Uhr.

Denken sie daran ihren Code mindestens mit Verträgen und Beschreibungen zu kommentieren, sowie für jede Prozedur 2 Testfälle anzugeben. Wählen sie für Hilfsfunktionen und Parameter sinnvolle Namen. Benutzen sie als Sprachlevel "Zwischenstufe mit Lambda".

7 Zum Aufwärmen: Palindrome (K) (4 P)

Implementieren Sie eine Funktion `palindrome`, die eine Liste konsumiert und sie zu einem Palindrom erweitert. Geben Sie mindestens 2 Testfälle an. Ein Palindrom ist eine Liste, die von vorn und von hinten gelesen gleich bleibt. Der Vertrag, Beschreibung und Beispiele sehen wie folgt aus:

```
1 ;; Contract: palindrome: (listof x) -> (listof x)
2 ;; Purpose:  to build a list which transforms the given
3 ;;           list into a palindrome list..
4 ;; Examples 1.) (palindrome (list 'a 'b 'c))
5 ;;                  should produce: (list 'a 'b 'c 'b 'a)
6 ;;           2.) (palindrome (list 1 2 3 4))
7 ;;                  should produce: (list 1 2 3 4 3 2 1)
```

8 Sinus-Approximation (K) (4P)

Im Folgenden soll eine Approximation der Sinus-Funktion $sin(x)$ implementiert werden. x soll dabei im Bogenmaß angegeben werden. Für kleine x kann man $sin(x)$ durch x abschätzen. Als "klein" sollen im Rahmen dieser Aufgabe Werte von $|x| \leq 0.1$ angesehen werden. Der Sinus für größere Werte lässt sich dann nach folgender Gleichung abschätzen:

$$sin(x) = \begin{cases} x & |x| \leq 0.1 \\ 3sin(\frac{x}{3}) - 4(sin(\frac{x}{3}))^3 & sonst \end{cases}$$

1. Implementieren Sie die Abschätzungsfunktion für den Sinus in Scheme. Vermeiden Sie, dass $sin(\frac{x}{3})$ mehrfach berechnet wird, indem Sie die lambda-Notation verwenden. Vergessen Sie nicht den Vertrag anzugeben, Beschreibung, Beispiel und Testfälle können Sie vom Übungsblatt übernehmen.

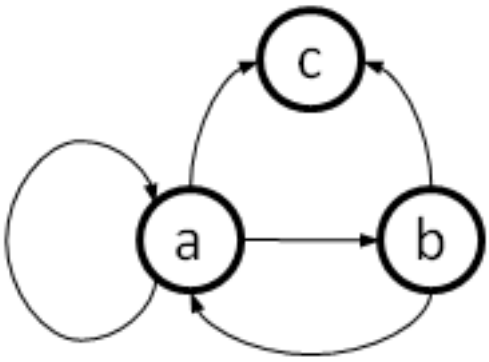
```
1  ;; Purpose: calculate sinus x using the approximation for small
     numbers
2  ;; Example: (sin-approx 3.14) should be close to 0
3
4  (define (sin-approx angle)
5    ...)
6  ;; Test
7  (sin-approx 1.2)
8  ;; should be
9  0.93218229417886703406711722434709627405353802620137...
10
11 ;; Test
12 (sin-approx 3.14)
13 ;; should be
14 0.00080567746742247618824 65...
```

2. Wie oft ruft sich Ihre Funktion rekursiv selbst auf, wenn Sie den Sinus von 12.15 berechnen? Begründen Sie Ihre Antwort!

3. Da sin periodisch ist, gilt $sin(x) = sin(x \ mod \ 2\Pi)$, $x \ mod \ 2\Pi$ ist dabei der Rest, der bei der Teilung durch 2Π entsteht. Jedes x kann so im ersten Schritt auf das Intervall $[0, 2\Pi]$ abgebildet werden, das größte x für das $sin(x)$ also rekursiv berechnet werden muss, ist $x = 2\Pi$. Was ist daher die Komplexität des Algorithmus?

9 Graphen (5 Punkte)

In der Vorlesung haben Sie bereits verschiedene Graphen kennengelernt (T6,T8). Im Folgenden möchten wir nun auf gerichteten Graphen arbeiten. Wir repräsentieren solche Graphen als `listof edge`. Anders als in der Vorlesung soll eine Kante (`edge`) **eine Struktur** aus Startknoten und Endknoten sein. Startknoten und Endknoten sollen wie in der Vorlesung Symbole sein. In der Vorlage ist die Struktur und ein kleiner Graph `samplegraph` definiert, der foldenden Graphen repräsentiert.



Beachten Sie: In dieser Übung dürfen Graphen Zyklen enthalten! Ein Zyklus ist ein Pfad in einem Graph, der beim gleichen Knoten startet und endet. Kette von mit Kanten verbundenen Knoten, die in sich geschlossen sind, z.B. die Kette a, b oder die einelementige Kette a im Beispielgraph.

- Schreiben Sie eine Prozedur `neighbors: node graph -> (listof node)`, die einen Knoten n und einen Graphen G konsumiert und die Nachbarknoten dieses Knotens als Liste zurückliefert. Nachbarknoten sind diejenigen Knoten in G die Endknoten zu einer Kante in G sind bei der n Startknoten ist. Zum Beispiel sind a,b und c Nachbarknoten von a, a und c von b und c hat keine Nachbarknoten. Beachten Sie, dass Kanten Strukturen vom Typ edge sind!

- Wir wollen nun für unsere Graphen eine Prozedur `find-route: node node graph -> (listof node)` erstellen, die zwei Knoten und einen Graphen konsumiert, und einen beliebigen zyklenfreien Pfad vom ersten zum zweiten Knoten ausgibt (vgl. T6.53ff). Falls kein Pfad existiert soll `false` zurückgegeben werden. Stellen Sie dabei mit Hilfe von Akkumulatoren sicher, dass die Prozedur auch auf Graphen, **die Zyklen enthalten**, terminiert! Sie dürfen die Funktion `contains?` aus der Vorlage verwenden. `contains?` gibt für ein Symbol s und eine Liste von Symbolen los zurück, ob s in los enthalten ist.

Vergessen Sie nicht Vertrag, Beschreibung, Beispiel und mindestens zwei Tests anzugeben.