# Grundlagen der Informatik 1

## Sommersemester 2011

Dr. Guido Rößling

https://moodle.informatik.tu-darmstadt.de/

Übung 6 Version: 1.0 23. 05. 2011

---

# 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 $O(n\,log(n))$.
- ☐ $n^3 \in O(n^n)$
- ☐ $O(\log(n^{1000000})) \subset O(n)$
- ☐ $0,0123456789n \in O(n)$
- ☐ $100.000.000.000.000.000.000.001 \in O(1)$

Akkumulatoren:

- ☐ Funktionen im Akkumulator-Stil sind immer schneller als Funktionen ohne Akkumulator.
- ☐ Manche Funktionen lassen sich nur mit Akkumulatoren implementieren.
- ☐ Ein Akkumulator wird nur benötigt für Probleme die generative Rekursion nutzen.
- ☐ Der 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 (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 „betrachtet“ 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, beispielsweise InsertionSort, zu wählen?

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

6. Welche der folgenden Aussagen gelten? Dabei sei $f(n) = 3\,n\,log_4(n)$ und $g(n) = 9\,n\,log_{10}(n)$.

   a) $f(n) \in \Theta(g(n))$
   b) $O(f(n)) \subseteq O(g(n))$
   c) $\Theta(f(n)) \subsetneq O(g(n))$

7. Zeigen Sie, dass aus $f(n) \in O(log_a(n))$ folgt $f(n) \in O(log_b(n))$ für beliebige $a, b > 1$.

# 3 Rekursion vs. Iteration: Fakultät (K)

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

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

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

2. Veranschaulichen Sie nun Ihre Lösung (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. Überlegen Sie sich dazu, wie der Akkumulator zu initialisieren ist!

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 haltenden 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:

$$fib(n) = \begin{cases} 0 & \text{für } n = 0 \\ 1 & \text{für } n = 1 \\ fib(n-1) + fib(n-2) & \text{falls } n \geq 2 \end{cases}$$

1. Implementieren Sie die Funktion fib−rec: nat −> nat, die die n-te Fibonaccizahl rekursiv berechnet[1]. Zeichnen Sie einen Rekursionsbaum für die 4. Fibonacci Zahl und beurteilen Sie die Effizienz Ihrer Lösung. Zur Erinnerung: die ersten neun Fibonacci Zahlen (beginnend ab $n = 0$) lauten *0, 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 eine Funktion `invert2`, die *ohne* Akkumulator und/oder **foldl**/**foldr** auskommt. Vervollständigen Sie dazu folgenden Code, ohne `append` oder eine ähnliche bereits verfügbare Funktion zu verwenden:

```
;; invert2: (listof X) −> (listof X)
;; construct the reverse of list lst
;; example: (invert2 (list 'A 'B 'C)) should return (list 'C 'B 'A)
(define (invert2 lst)
    ;; rcons : (listof X) X −> (listof X)
    ;; appends el to the end of list lst.
    ;; example: (rcons (list 'A 'B) 'C) should be (list 'A 'B 'C)
  (local (
    (define (rcons lst el)
            (...) ))
    (...) ))
```

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

# 6 Rekursion und Komplexität

Die Ackermann-Funktion 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:

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

[1] nat steht dabei für *nat*ürliche Zahlen

Berechnen Sie dann den Aufruf Ack(3,4). Berechnen Sie anschließend 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 Lernportal Informatik bereitgestellt. Kommentieren Sie Ihren selbst erstellten Code. Die Hausübung muss bis zum Abgabedatum im Lernportal Informatik 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.*
*Falls Sie die Hausübung in einer Lerngruppe bearbeitet haben, geben Sie dies bitte deutlich bei der Abgabe an. Alle anderen Mitglieder der Lerngruppe müssen als Abgabe einen Verweis auf die gemeinsame Bearbeitung einreichen, damit die Abgabe im Lernportal Informatik auch für sie bewertet werden kann. Beachten Sie dazu die Hinweise bei der Aufgabenabgabe im Lernportal Informatik!*
**Abgabedatum: Freitag, 3. 6. 2011, 16:00 Uhr**
Denken Sie bitte daran, Ihren Code hinreichend gemäß den Vorgaben zu kommentieren (Racket: Vertrag, Beschreibung und Beispiel sowie zwei Testfälle pro Funktion; Vertrag, Beschreibung und Beispiel für jede **local** definierte Funktion; Vertrag (ohne Namen) und kurze Beschreibung für jeden **lambda**-Ausdruck; Java: JavaDoc). Zerlegen Sie Ihren Code sinnvoll und versuchen Sie, wo es möglich ist, bestehende Funktionen wiederzuverwenden. Wählen Sie sinnvolle Namen für Hilfsfunktionen und Parameter.
*Benutzen sie als Sprachlevel "Zwischenstufe mit Lambda".*

# 7 Zum Aufwärmen: Palindrome (K) (4 Punkte)

Implementieren Sie zwei Funktionen `palindrome` und `doubled-palindrome`, die eine Liste konsumieren und eine zu einem Palindrom erweiterte Liste zurückgeben. Bei der zweiten Variante soll dabei jedes Zeichen dupliziert werden. Geben Sie mindestens 2 Testfälle außer den im Template vorgegebenen Tests an. Ein Palindrom ist eine Liste, die von vorn und von hinten gelesen gleich bleibt. Dabei soll das letzte Element der Ursprungsliste bei dem *einfachen* Palindrom *nicht* verdoppelt werden.
Beispiel: (palindrome '(a b c d))= '(a b c d c b a), (doubled−palindrome '(1 2 3))= (1 1 2 2 3 3 2 2 1 1)
Verwenden Sie keine explizite Rekursion, sondern `foldr` und/oder `foldl` für die Bearbeitung! Die Nutzung von *invert* bzw. *rcons* oder vergleichbarer Funktionen ist natürlich **nicht** erlaubt.

# 8 Neubeschaffung des Hausrats (6 Punkte)

Ihr bester Freund ist völlig verzweifelt: Ein Brand hat während seines Urlaubs seine Wohnung und alle Besitztümer, vom Bett bis zu den Büchern, komplett zerstört. Ihr Freund steht nun praktisch ohne Alles—nur noch mit den Kleidern und den Badeutensilien in seinem Koffer—da.
Glücklicherweise stellt seine Hausratsversicherung unbürokratisch eine größere Menge Geld bereit, damit er seinen Hausrat erneut beschaffen kann. Ihr Freund hat dafür schon diverse Kataloge durchgesehen—und ist immer noch verzweifelt. Er präsentiert Ihnen einen lange Liste von Gegenständen, die er unbedingt braucht—etwa ein Bett—oder cool findet—wie ein *iPad 2*.
Ihre Aufgabe ist es, Ihrem Freund zu helfen. Auf seiner Einkaufsliste hat er die Gegenstände mit *Namen, Preis, Dringlichkeit* und *„Coolness"* aufgeführt. Da das Geld der Versicherung—naturgemäß—nicht für alles langt, sollen Sie für ihn berechnen, welche Gegenstände die „optimale" Auswahl darstellen.
Zur Modellierung verwenden wir dabei die folgende Datenstruktur:

```
1 ;; items represents items found in an appartment or house
2 ;; name: String − the name of the item
3 ;; urgency: number − the more urgent, the higher the value
4 ;; coolness: number − the higher, the "cooler" the item
```

```
;; price: number − the price of the item in Euro
(define−struct item (name urgency coolness price))
```

Ein *item* repräsentiert also einen Gegenstand, indem sein Name, die Wichtigkeit der Anschaffung sowie sein „Coolness-Faktor“ und abschließend sein Preis angegeben wird.
Ihre Aufgabe ist es nun, aus der Liste der potentiellen Objekte die beste Teilmenge von Objekten zu finden, so dass die folgenden Regeln erfüllt sind:

- Der Gesamtpreis der ausgewählten Objekte übersteigt nicht die Versicherungssumme. Diese wird der Prozedur als Parameter übergeben.
- Ihr Freund übergibt Ihnen seine „Prioritäten“ als eine Funktion der Form item −> **number**. Dies könnte einfach die Wichtigkeit, die „Coolness“ oder eine gewichtete Mischung beider Faktoren sein.

Implementieren Sie zuerst eine Prozedur evaluate−item−sum: (listof item)(item −> **number**)−> **number**, die eine Liste von möglichen Gegenständen sowie eine Auswertungsprozedur—etwa item−coolness—konsumiert und die Gesamtbewertung der gewählten Elemente berechnet.
Implementieren Sie die Prozedur choose−items: (listof item)**number** (item −> **number**)−> (listof item), die eine Liste der potenziellen Gegenstände, die verfügbaren Mittel und die Auswertungsprozedur konsumiert. Als Ergebnis ist eine Liste von Elementen zu produzieren, deren Gesamtpreis unter der Versicherungssumme liegt und deren Gesamtwert gemäß der Auswertungsprozedur maximal ist.
**Hinweis:** Auch wenn die Versicherungssumme im Fall Ihres Freundes feststeht, müssen Sie sie als Parameter übergeben, um auch anderen Menschen helfen zu können.
**Hinweis:** Zeichnen Sie sich einen Baum auf, bei dem Sie an jedem Knoten für ein Element bestimmen, ob das Element genommen wird ($\rightarrow$ rechter Sohn) oder nicht ($\rightarrow$ linker Sohn).