Technische Universität München
Institut für Informatik I-16
Dr. Tobias Lasser
Jakob Vogel

WiSe 2012/13
Lösungsblatt 6
26. November 2012

Algorithmen und Datenstrukturen

Aufgabe 1 Verifikation der Multiplikation durch Addition (Beispiellösung)

a) Nachdem die Funktion ja korrigiert wurde, unterstützen wir beliebige ganze Zahlen als Eingabe, und können daraus die (triviale) Vorbedingung schließen:

$$VOR \quad = \quad a \in \mathbb{Z} \ \wedge \ b \in \mathbb{Z}$$

Die Nachbedingung ist ebenfalls eher einfach zu formulieren, da wir ja nur fordern, dass im Ergebnis c der korrekte Wert steht, also:

$$NACH \quad = \quad (c = a \cdot b)$$

b) Zum Korrektheitsbeweis der ganzen Funktion muss dann auch gezeigt werden, dass im **else**-Zweig der Fallunterscheidung (in der dann $a > 0$ gilt) die Schleife korrekt arbeitet. Dazu benötigen wir eine Schleifeninvariante, die während der ganzen Ausführung gültig bleibt.

Die Idee ist, dass der bereits in c aufaddierte Teil und der im weiteren Verlauf der Schleife noch zu addierende Teil zusammen das korrekte Ergebnis $a \cdot b$ ergeben:

$$P \quad = \quad (a \cdot b = c + w \cdot b)$$

Dabei beschreibt w die Anzahl der *noch* durchzuführenden Schleifeniterationen bei gegebenem Wert i, also $w = a - (i-1) = a - i + 1$. Letztere Formel ergibt sich aus der Initialisierung $i = 1$ und zeigt, warum Beginn der Zählung bei 0, wie sonst üblich, sehr sinnvoll ist. Zusammengefasst lautet die Schleifeninvariante also:

$$P \quad = \quad (a \cdot b = c + (a - i + 1) \cdot b)$$

c) Beweisen wir also nun die Korrektheit der gesamten Funktion. Dazu müssen wir – Sie erinnern sich – nachvollziehen, ob bei gegebenen Vorbedingungen die Nachbedingung gilt.

Formal nach den Regeln aus der Vorlesung vorgehend, müssen wir zunächst die Korrektheit der äußeren Fallunterscheidung prüfen. Dafür kürzen wir den Code des ersten Falls mit α, den des zweiten Falls mit β und den des **else**-Falls mit γ ab:

$\{VOR\}$
if $(a = 0 \ \vee \ b = 0)$ {
    α
}
else if $(a < 0)$ {
    β
}
else {
    γ
}
$\{NACH\}$

Um die Korrektheit zu beweisen, sind also drei einzelne logische Beziehungen zu zeigen, bei denen jeweils aus der Vorbedingung und der Fall-Bedingung über die entsprechenden Befehle des entsprechenden Falls die Nachbedingung hervorgehen muss:

- $\{VOR \ \wedge \ (a = 0 \ \vee \ b = 0)\}$ α $\{NACH\}$

  Wenn einer der beiden Faktoren eines Produktes 0 ist, ist auch das Ergebnis automatisch 0. α setzt das Ergebnis $c = 0$, wenn genau dieser Fall gegeben ist, und insbesondere gilt hier dann offensichtlich $c = a \cdot b$, also die Nachbedingung $\{NACH\}$. Dementsprechend ist dieser Fall korrekt.

- $\{VOR \ \wedge \ \neg(a = 0 \ \vee \ b = 0) \ \wedge \ a < 0\}$ β $\{NACH\}$

  Zunächst kann die Vorbedingung von β vereinfacht werden:

  $$VOR \ \wedge \ a < 0 \ \wedge \ b \neq 0$$

  Bei einer negativen Zahl $a < 0$ gilt ja allgemein

  $$a = -|a| = -1 \cdot |a| = -1 \cdot (-a) \ ,$$

  und das Produkt $a \cdot b$ lässt sich dann schreiben als

  $$c = a \cdot b = (-1 \cdot (-a)) \cdot b$$

  bzw. wegen des Assoziativgesetzes auch als

  $$c = -1 \cdot ((-a) \cdot b) \ .$$

  Unter der Annahme, dass **Multiply** für positives a das korrekte Ergebnis ergibt (wir zeigen das dann später!), gilt dann auch in diesem Fall $c = a \cdot b$, und damit die Nachbedingung $\{NACH\}$ – die Korrektheit des dritten Falles vorausgesetzt.

- $\{VOR \ \wedge \ \neg(a = 0 \ \vee \ b = 0) \ \wedge \ \neg(a < 0)\}$ γ $\{NACH\}$

  Auch hier können wir zunächst die Vorbedingung von γ vereinfachen:

  $$C_0 \ := \ (VOR \ \wedge \ a > 0 \ \wedge \ b \neq 0)$$

  Die in γ abstrahierten Befehle bestehen aus einer ganzen Kette von Anweisungen:

  $\{C_0\}$
  $c = 0;$
  $i = 1;$
  $\{C_1\}$
  while $(i \leq a)$ {
  $\{C_2\}$
  $c = c + b;$
  $i = i + 1;$
  $\{C_3\}$
  }
  $\{NACH\}$

  Um die Korrektheit dieser Schleife zu zeigen, müssen formell wieder drei logische Beziehungen zutreffen, wobei wir nun die Schleifeninitialisierung mit γ_{init} und den Schleifenkörper mit $\gamma_{\text{körper}}$ bezeichnen:

  – $C_1 \Rightarrow P$

    Für diese Implikation, die die Korrektheit des Schleifeneintritts sicherstellt, müssen wir erst den Ausdruck

    $$\{C_0\} \ \gamma_{\text{init}} \ \{C_1\}$$

klären um $\{C_1\}$ herzuleiten. Es ergibt sich:

$$C_1 \quad := \quad (C_0 \ \wedge \ c = 0 \ \wedge \ i = 1)$$

Betrachten wir nun die Schleifeninvariante, und prüfen wir deren Gültigkeit. Es war ja

$$P \quad = \quad (a \cdot b = c + (a - i + 1) \cdot b) \ ,$$

und wir können die Werte von c und i einsetzen:

$$c + (a - i + 1) \cdot b = 0 + (a - 1 + 1) \cdot b = a \cdot b$$

Dieser Ausdruck ist *wahr*, also gilt P, und der Schleifeneintritt ist verifiziert.

– $\{P \ \wedge \ (i \leq a)\} \ \gamma_{\text{körper}} \ \{P\}$

Nachfolgend muss geprüft werden, ob die Schleifeninvariante nach der Durchführung des Schleifenkörpers gültig bleibt[1]. Es gilt ja:

$$C_2 \quad := \quad (P \ \wedge \ i \leq a)$$

Im Schleifenkörper werden zwei Variablen, c und i, neu gesetzt, und wir bezeichnen deren *neue* Werte vorübergehend mit c' und i'. Direkt aus dem Programmcode $\gamma_{\text{körper}}$ ergibt sich der Zusammenhang :

$$\begin{aligned} c' &= c + b \\ i' &= i + 1 \end{aligned}$$

Nachdem *vor* dem Schleifenkörper die Invariante galt, können wir schreiben:

$$a \cdot b = c + (a - i + 1) \cdot b$$

Nun soll die Invariante auch *nach* dem Schleifenkörper gelten. Es ergibt sich unter Nutzung des alten Zustandes:

$$\begin{aligned} c' + (a - i' + 1) \cdot b &= (c + b) + (a - (i + 1) + 1) \cdot b \\ &= c + b + (a - i) \cdot b \\ &= c + (a - i + 1) \cdot b \\ &= a \cdot b \end{aligned}$$

Dies ist exakt die Schleifeninvariante, und somit ist auch der Schleifenkörper als korrekt verifiziert.

– $P \ \wedge \ \neg(i \leq a) \Rightarrow NACH$

Abschließend muss sichergestellt sein, dass bei gegebener Schleifeninvariante aber verletzter Schleifenbedingung (nach dem Abbruch der Wiederholungen) die Nachbedingung folgt.
Durch die schrittweise Erhöhung von i im Schleifenkörper muss nicht nur $i > a$, sondern sogar $i = a + 1$ gelten. Setzen wir dies wieder in die Formel aus der Schleifeninvariante ein:

$$\begin{aligned} a \cdot b &= c + (a - i + 1) \cdot b \\ &= c + (a - (a + 1) + 1) \cdot b \\ &= c + (a - a - 1 + 1) \cdot b \\ &= c + 0 \cdot b \\ &= c \end{aligned}$$

[1] Praktischerweise war die Schleifeninvariante beim *Selection Sort* auf dem vorherigen Blatt nach jeder Anweisung des Schleifenkörpers gültig. Dies muss nicht notwendigerweise so sein, und die Schleife ist korrekt, wenn die Invariante *am Ende* der Schleife (wieder) gilt.

Dies ist exakt die Nachbedingung, die Schleifenterminierung ist somit ebenfalls als korrekt verifiziert.

Multiply ergibt also auch im Fall positiver Werte für a das korrekte Ergebnis, womit indirekt auch der vorherige Fall negativen Werts mit bewiesen wurde.

Nachdem nun alle drei Fälle des **if-elseif-else**-Statements als korrekt verifiziert wurden, ist die gesamte **Multiply** korrekt. □

Aufgabe 2 Rechnen mit Landau-Symbolen (Beispiellösung)

Die Definitionen der beiden für uns relevanten Landau-Symbole waren:

$$\begin{aligned}\Theta(g) &:= \{f : \mathbb{R} \to \mathbb{R} \ : \text{ es existieren } c_1, c_2 > 0 \text{ und } n_0 \in \mathbb{N}, \text{ so dass für alle } n \geq n_0 \text{ gilt:} \\ &\qquad 0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)\} \\ O(g) &:= \{f : \mathbb{R} \to \mathbb{R} \ : \text{ es existieren } c > 0 \text{ und } n_0 \in \mathbb{N}, \text{ so dass für alle } n \geq n_0 \text{ gilt:} \\ &\qquad 0 \leq f(n) \leq c \cdot g(n)\}\end{aligned}$$

Bedenken Sie, dass trotz der Mengeneigenschaft normalerweise die abkürzenden Schreibweisen verwendet werden. Damit sind nun die folgenden Behauptungen zu zeigen:

a) $7n^4 = O(n^5)$

Wir betrachten die Ungleichung aus der Definition von $O(n)$:

$$0 \leq 7n^4 \leq c \cdot n^5$$

Für $n \geq 1$ kann ohne Umkehr der Ungleichung durch n^4 dividiert werden:

$$0 \leq 7 \leq c \cdot n$$

Wählen wir nun $c = 7$ und $n_0 = 1$, so gilt die Ungleichung für alle $n \geq n_0$. Somit ist die Behauptung gezeigt. □

b) $n^2/2 - 2n = \Theta(n^2)$

Auch hier betrachten wir die Ungleichung aus der Definition von $\Theta(n^2)$, teilen aber auf in eine linke und eine rechte Seite.

Zunächst also die Abschätzung der unteren Schranke:

$$c_1 \cdot n^2 \leq \frac{n^2}{2} - 2n$$

Auch hier können wir für $n \geq 1$ durch n^2 ohne Umkehr der Ungleichung dividieren:

$$c_1 \leq \frac{1}{2} - \frac{2}{n}$$

Diese Ungleichung gilt etwa für $c_1 = \frac{1}{4}$ und alle $n \geq n_0 = 8$.

Ähnlich kann man die obere Schranke abschätzen:

$$\frac{n^2}{2} - 2n \leq c_2 \cdot n^2$$

Wieder teilen wir – wie oben – durch n^2 und erhalten:

$$\frac{1}{2} - \frac{2}{n} \leq c_2$$

Diese Ungleichung gilt dann etwa für $c_2 = \frac{1}{2}$ bei gleich gewähltem $n_0 = 8$ wie oben.

Insgesamt haben wir nun die beiden Schranken, $c_1 = \frac{1}{4}$ und $c_2 = \frac{1}{2}$, und eine Grenze, $n_0 = 8$, identifiziert, so dass die komplette Ungleichung

$$0 \leq c_1 \cdot n^2 \leq \frac{n^2}{2} - 2n \leq c_2 \cdot n^2$$

für alle $n \geq n_0$ gilt. Also gilt die Behauptung. □

c) $2^{n+1} = O(2^n)$

Wie oben betrachten wir die Ungleichung aus der Definition von $O(2^n)$:

$$0 \leq 2^{n+1} \leq c \cdot 2^n$$

Wir brauchen also wieder sinnvolle Werte für c und n_0, so dass diese Ungleichung für alle $n \geq n_0$ gilt. Wir zerlegen

$$2^{n+1} = 2 \cdot 2^n$$

und erhalten damit $c = 2$ und $n_0 = 1$. Somit gilt die Behauptung. □

d) $2^{2n} \neq O(2^n)$

Angenommen, dass diese Behauptung *nicht* zutrifft, dann existieren auch Konstanten c und $n_0 > 0$, so dass die Ungleichung

$$0 \leq 2^{2n} \leq c \cdot 2^n$$

für alle $n \geq n_0$ gilt. Wir zerlegen 2^{2n} und erhalten

$$2^{2n} = 2^n \cdot 2^n \leq c \cdot 2^n \Leftrightarrow 2^n \leq c \ .$$

Die Folge 2^n ist aber unbeschränkt, und es existiert keine Konstante c, die für beliebige Werte von n die Forderung $c \geq 2^n$ erfüllt. Dies ist ein Widerspruch zur Annahme, und die die zu zeigende Behauptung gilt somit. □

Aufgabe 3 Komplexität von Selection Sort (Beispiellösung)

a) Die **Swap**-Methode besteht aus exakt 3 Befehlen, hätte im RAM-Modell also den Aufwand bzw. die Komplexität 3. Damit ist die Methode offensichtlich $O(1)$. Der Beweis wird hier nicht exakt ausgeführt, bei Wahl von $c \geq 3$ jedenfalls ist die Ungleichung aus der Definition von O erfüllt.

b) Zur Untersuchung der Komplexität von **IndexOfMin** betrachten wir zunächst den Pseudocode und nummerieren dabei die relevanten Zeilen:

```
1       i = j;
2       k = j + 1;
3       while (k < n) {
4           if (A[k] < A[i]) {
5               i = k;
            }
6           k = k + 1;
        }
```

Bei einem einzigen Durchlauf des Codes ergeben sich damit die folgenden Aufrufhäufigkeiten pro Zeile, wobei t_k die Anzahl der Fälle enthält, bei denen in Zeile 5 wegen erfolgreichen Tests in Zeile 4 eingesprungen wird:

Zeile	Kosten	Häufigkeit
1	c_1	1
2	c_2	1
3	c_3	$n-j$
4	c_4	$n-j-1$
5	c_5	t_k
6	c_6	$n-j-1$

Insgesamt ergibt sich also die folgende Laufzeit:

$$\begin{aligned} T(n) &= 1+1+(n-j)+(n-j-1)+t_k+(n-j-1) \\ &= 3\cdot n-3\cdot j+t_k \end{aligned}$$

Im *besten Fall* muss Zeile 5 nie aufgerufen werden, also muss dann $A[k] \geq A[i]$ immer gelten. Dies passiert genau dann, wenn A bereits aufsteigend sortiert ist, und es gilt dann $t_k = 0$. Umgekehrt muss im *schlechtesten Fall* Zeile 5 in jeder Iteration aufgerufen werden. Dann gilt $A[k] < A[i]$ immer, A ist also dann genau umgekehrt (absteigend) sortiert, und es gilt dann $t_k = n-j-1$. In der Realität ist aber der Inhalt von Zeile 5 extrem schnell umsetzbar, während der eigentliche Vergleich in Zeile 4 recht teuer ist, so dass man im Laufzeitverhalten wohl nur geringe Unterschiede wird messen können.

Die genaue Laufzeit ist aber oft nicht von Interesse, und man schätzt mittels O-Notation (also dem entsprechenden Landau-Symbol) ab, um ein Gefühl für die Laufzeit im *durchschnittlichen Fall* (*average case*) zu erhalten. Außerdem wird der Startindex j ignoriert. Damit erhält man, dass die Laufzeit $T(n) = O(n)$ ist. Dies ergibt sich aus der Tatsache, dass im aller-schlechtesten Fall ($j = 0$ und absteigende Sortierung) $t_k = n$ und damit $T(n) \leq 4 \cdot n$ gilt, und somit ein $c \geq 4$ die Ungleichung aus der Definition von O erfüllt.

Tatsächlich könnte man auch informell argumentieren, dass bei steigender Problemgröße n die **for**-Schleife als dominate Anweisung häufiger durchgeführt wird, und damit „offensichtlich“ die Methode eine Komplexität von $O(n)$ haben muss.

c) Betrachten wir abschließend die eigentliche Sortierfunktion **SelectionSort**. Enthalten ist primär eine **for**-Schleife, die $n-1$ mal durchlaufen wird (also $O(n)$ ist) und dabei jedes Mal zunächst die $O(n)$-Funktion **IndexOfMin** und dann – nach einem Vergleich – nötigenfalls die $O(1)$-Funktion **Swap** durchführt. Dazu kommt noch $O(1)$-Aufwand zur Initialisierung der Schleife. Dominant ist also der etwa n-malige Aufruf einer $O(n)$-Funktion, wodurch die Funktion insgesamt dann von Komplexität $O(n^2)$ ist.

Interessanter ist die Bestimmung des *Worst-Case*-Szenarios für diese Funktion. Wie oben ausgeführt, ist die absteigende Sortierung der schlechteste Fall für die **IndexOfMin**. Sortiert man ein derartig vorsortierte Liste mit Selection Sort, so wird beim ersten Swap das kleinste mit grössten Element vertauscht, dann das zweit-kleinste mit dem zweit-grössten, usw. Dadurch ist schon nach dem halben Durchlauf des Arrays die Sortierung perfekt, und es sind insgesamt nur $n/2$ Vertauschungen notwendig.

Ist dagegen das Array so vorbelegt, dass zunächst das grösste Element enthalten ist, und danach die restlichen Elemente in aufsteigender Sortierung folgen, so ist die Neusetzung des minimalen Index in der **IndexOfMin** nur einmal nötig, aber es müssen n Vertauschoperationen durchgeführt werden, da das größte Element vor sich hergeschoben wird.

Daraus lässt sich folgern, dass der tatsächliche Worst-Case massiv davon abhängt, ob eine Vertauschung mittels **Swap** teurer ist als die Rangvergleiche in der **IndexOfMin**.