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

WiSe 2012/13
Lösungsblatt 5
19. November 2012

Algorithmen und Datenstrukturen

Aufgabe 1 Selection Sort I (Beispiellösung)

Zunächst formulieren wir die Vor- und Nachbedingungen zu den drei Teilfunktionen, anschließend überprüfen wir, ob die Funktionen jeweils die gestellten Anforderungen erfüllen.

a) Die Vorbedingung der Sortierfunktion **SelectionSort** sind gültige Eingabedaten, die Nachbedingung ist, dass das Array eine aufsteigend sortierte Permutation der Originaldaten enthält. Formeller geschrieben:

$$\begin{aligned} VOR_{sort} &= n \geq 0 \\ NACH_{sort} &= A \text{ enthält Permutation der Originaldaten } \wedge \\ &\quad A[0] \leq A[1] \leq ... \leq A[n-1] \end{aligned}$$

Betrachten wir als nächstes die Hilfsfunktion **IndexOfMin**, die den Index i des minimalen Werts aus $A[j..(n-1)]$ zurückgibt. Die Vorbedingung ist hier unter anderem, dass j ein gültiger Index ist, die Nachbedingung dann, dass i ebenfalls gültig und mindestens so groß wie j ist, und kein Wert in $A[j..(n-1)]$ kleiner ist als $A[i]$. Formeller[1]:

$$\begin{aligned} VOR_{min} &= n > 0 \ \wedge \ 0 \leq j < n \\ NACH_{min} &= j \leq i < n \ \wedge \ A[i] \leq A[k] \ \ \forall k \in \{j, ..., n-1\} \end{aligned}$$

Zuletzt haben wir die Funktion **Swap**, die zwei Zellen des Arrays austauscht. Eine derartige Funktion haben Sie bereits in der Vorlesung gesehen, wir erweitern hier noch um die Vorbedingung nach gültigen Array-Indices. Insgesamt also:

$$\begin{aligned} VOR_{swap} &= n > 0 \ \wedge \ 0 \leq i, j < n \ \wedge \ A[i] = a \ \wedge \ A[j] = b \\ NACH_{swap} &= A[i] = b \ \wedge \ A[j] = a \ \wedge \ A[k] \text{ unverändert } \ \forall k \in \{0, ..., n-1\} \setminus \{i, j\} \end{aligned}$$

Beide Bedingungen gelten jeweils für beliebige $a, b \in \mathbb{Z}$ bzw. $\forall a, b \in \mathbb{Z}$.

b) Nun überprüfen wir, ob die Sortierfunktion **SelectionSort** bei gegebener Vorbedingung die Nachbedingung erfüllt. Dazu benötigen wir zunächst eine Schleifen-Invariante für die **for**-Schleife. Diese besteht aus mehreren Teilen:

- Laut $NACH_{sort}$ darf A am Schluss nur eine Permutation seiner Originaldaten enthalten, das muss somit auch für die Schleifeninvariante gelten.
- Wir wählen ja aus dem unsortierten Rest immer das kleinste Element aus und fügen es an das Ende des bereits sortierten Anfangs an. Diese Sortierungseigenschaft müssen wir für die bereits bearbeiteten Indices 0 bis $j-1$ sicherstellen; also muss gelten, dass $A[0] \leq A[1] \leq ... \leq A[j-1]$.

[1] Der sogenannte „All-Quantor“ $\forall k$ bedeutet, dass die vor dem Symbol notierte Bedingung für **alle** Werte k gelten muss, wie sie nach dem Symbol definiert sind.

- *Das alleine reicht aber noch nicht!* Auch wenn der hintere Teil unsortiert sein darf, darf dort – wir wählen ja immer das kleinste Element aus dem Rest aus – keine Zahl stehen, die kleiner als die größte Zahl im bereits sortierten Teil ist. Die größte Zahl im sortierten Teil steht (wegen der Sortierung!) in $A[j-1]$, also muss gelten, dass $A[j-1] \leq A[j..(n-1)]$.

Insgesamt können wir also die Invariante aufschreiben als

$$\begin{aligned} P \quad = \quad & A \text{ enthält Permutation der Originaldaten} \ \wedge \\ & A[0] \leq A[1] \leq ... \leq A[j-1] \leq A[j..(n-1)] \end{aligned}$$

Die **for**-Schleife kann ja in eine äquivalente **while**-Schleife umgewandelt werden, und wir erhalten:

```
j = 0;
while (j < n) {
        ...
        j = j + 1;
}
```

Damit nun kann der Korrektheitsbeweis erbracht werden. Vergegenwärtigen Sie sich nochmals die Gesamtstruktur des Sortieralgorithmus in der Variante mit **while**-Schleife. Die darin als Platzhalter bereits enthaltenen Bedingungen $\{C_i\}$ werden wir nachfolgend durchgehen und genau bestimmen.

```
{C0}
    j = 0;
{C1}
    while (j < n) {
{C2}
        i = IndexOfMin(A, j);
{C3}
        if (j ≠ i) {
            Swap(A, i, j);
        }
{C4}
        j = j + 1;
{C5}
    }
{C6}
```

Die erste Bedingung $\{C_0\}$ ist genau die Vorbedingung von **SelectionSort**, also:

$$C_0 \quad = \quad VOR_{sort} \quad = \quad n \geq 0$$

Die folgende Bedingung $\{C_1\}$ unterscheidet sich durch die Initialisierung von j. Da wir unmittelbar nachfolgend in die Schleife einlaufen und bereits über alle notwendigen Informationen verfügen, prüfen wir nun auch die Schleifeninvariante. Diese *gilt* offensichtlich, da das Array A noch nicht geändert wurde und somit natürlich (eine Permutation seiner) Originaldaten enthält. Da $j = 0$, ist auch der bereits sortierte Teil leer, und damit sind sowohl alle Elemente dort sortiert, als auch ist kein Element im unsortierten Teil kleiner als das größte sortierte Element. Also $C_1 \Rightarrow P$, und es gilt:

$$\begin{aligned} C_1 \quad &= \quad C_0 \ \wedge \ j = 0 \ \wedge \ P \\ &= \quad n \geq 0 \ \wedge \ j = 0 \ \wedge \ P \end{aligned}$$

Im Falle eines leeren Arrays gilt $n = 0$, damit ist die Schleifenbedingung $j < n$ *nicht* erfüllt und es gilt folglich $j \geq n$. In diesem Fall überspringen wir die Schleife direkt, und es gilt *in diesem Fall* für $\{C_6\}$:

$$C_6 \quad = \quad n = 0 \ \wedge \ j = 0 \ \wedge \ P$$

Daraus folgt nun durch Einsetzen von j und n in P direkt die Nachbedingung, da die Permutationseigenschaft (mangels Änderung und Inhalt) und die Sortierungseigenschaft (mangels Inhalt) trivial erfüllt sind. Also gilt im Falle des leeren Arrays

$$C_6 \stackrel{n=0}{=} C_1 \wedge \neg(j < n) \quad \Rightarrow \quad NACH_{sort}$$

Sei nachfolgend das Array daher nicht leer, also $n > 0$, und folglich die Schleifenbedingung $j < n$ erfüllt. In diesem Fall gilt für $\{C_2\}$:

$$C_2 \quad = \quad n > 0 \ \wedge \ 0 \leq j < n \ \wedge \ P$$

Der mittlere Teil des Ausdrucks ist also die Schleifenbedingung, der letze Teil die Invariante. Wir betrachten $j = 0$ nicht mehr gesondert, da wir nicht jeden einzelnen Schleifendurchlauf separat behandeln, sondern uns – wie ein Computer – mit dem logischen Ablauf beschäftigen. Um weiter die Korrektheit der Schleife zu zeigen, müssen wir nun sicherstellen, dass die Schleifeninvariante am Ende immer noch gilt, es ist also zu zeigen

$$C_6 = P \ \wedge \ \ldots$$

und wir „hangeln“ uns dazu durch den weiteren Code. Wir begnügen uns also vorläufig mit der Feststellung, dass C_2 zu den Vorbedingungen VOR_{min} der **IndexOfMin** kompatibel ist.

Damit gelten automatisch bei $\{C_3\}$ die Nachbedingungen $NACH_{min}$ der **IndexOfMin**. Wir zeigen deren Korrektheit nachher separat und nehmen sie hier einfach an. Also:

$$\begin{aligned} C_3 \quad = \quad & n > 0 \ \wedge \ 0 \leq j < n \ \wedge \ P \ \wedge \\ & j \leq i < n \ \wedge \ A[i] \leq A[k] \ \ \forall k \in \{j, \ldots, n-1\} \end{aligned}$$

Da wir am Array nichts geändert haben bleibt insbesondere die Invariante weiterhin gültig. Wir wissen nun aber, dass an Stelle i ein minimaler Wert $A[i]$ steht, also kein anderer Wert $A[k]$ im unsortierten Rest $j \leq k < n$ kleiner als $A[i]$ ist.

Anschließend kommt nun die Fall-Unterscheidung. Am Ende diese Unterscheidung muss gelten, dass das zuvor gefundene minimale Element am Ende des sortierten bzw. am Anfang des unsortierten Listenteils (also an Stelle j statt an i) steht. Bei $j = i$ ist dies trivial der Fall, und die **Swap** Methode kann übersprungen werden. Anderenfalls ist die Vorbedingung VOR_{swap} von **Swap** erfüllt, und nachfolgend sind dann gemäß Nachbedingung $NACH_{swap}$ die Elemente i und j vertauscht; insbesondere also liegt der minimale Wert nun an der Stelle j. Auch enthält dadurch A nach wie vor eine Permutation seiner Originalwerte. In beiden Fällen gilt für $\{C_4\}$ folglich:

$$\begin{aligned} C_4 \quad = \quad & n > 0 \ \wedge \ 0 \leq j < n \ \wedge \ P \ \wedge \\ & A[j] \leq A[(j+1)..(n-1)] \end{aligned}$$

Da der Wert von j nach wie vor unverändert ist, und im rechten unsortierten Teil Werte vertauscht werden dürfen – was vor allem die Permutations-Forderung erfüllt – gilt also insbesondere immer noch die Schleifeninvariante P. Der wichtige Punkt aber ist, dass an Stelle j nun ein für den unsortierten Rest minimaler Wert steht.

Als letzte Aktion in der Schleife wird nun noch die „Trennstelle“ j erhöht. Dadurch bleibt ebenfalls durch den Tausch im vorhergehenden Schritt die Schleifeninvariante gültig, und

der Schleifenkörper ist korrekt. Durch die Addition von 1 bleibt entweder $j < n$, wodurch die Schleife dann wiederholt wird, oder es gilt $j = n$, was zum Schleifenabbruch führt. Es gilt bei $\{C_5\}$ also:

$$C_5 \quad = \quad n > 0 \ \wedge \ (0 \leq j < n \ \vee \ j \geq n) \ \wedge \ P$$

Wenn die Schleife verlassen wurde, muss die Negation der Schleifenbedingung gelten, also streng genommen $j \geq n$. Die Schleifenbedingung P bleibt aber bestehen, die komplette Schleife ist also korrekt, und es gilt somit im Fall eines nicht-leeren Arrays:

$$\begin{aligned} C_6 \ &\stackrel{n>0}{=} \ n > 0 \ \wedge \ j \geq n \ \wedge \ P \\ &= \ n > 0 \ \wedge \ j \geq n \ \wedge \ A \text{ enthält Permutation der Originaldaten} \ \wedge \\ &\quad A[0] \leq A[1] \leq ... \leq A[j-1] \leq A[j..(n-1)] \end{aligned}$$

Da bei $j \geq n$ aber der unsortierte Rest verschwindet, vereinfacht sich die Formel zu

$$\begin{aligned} C_6 \ &\stackrel{n>0}{=} \ n > 0 \ \wedge \ A \text{ enthält Permutation der Originaldaten} \ \wedge \\ &\quad A[0] \leq A[1] \leq ... \leq A[n-1] \end{aligned}$$

Dies enthält nun genau die Nachbedingung der Sortierung, und es gilt:

$$C_6 \stackrel{n>0}{=} C_5 \wedge \neg(j < n) \quad \Rightarrow \quad NACH_{sort}$$

Wir haben nun gezeigt, dass sowohl im leeren wie im nicht-leeren Fall der Sortieralgorithmus bei gegebenen Vorbedingungen die garantierten Nachbedingungen erfüllt, der Code ist also verifiziert korrekt.

Wie in der Frage angedeutet, kann er aber noch etwas optimiert werden. Im letzten Schleifendurchgang ist $j = n-1$, der unsortierte Rest ist also genau von Länge 1. Es ist hier trivial, das minimale Element zu finden, und ein Tausch muss dann auch garantiert nicht mehr stattfinden. Insofern genügt es, die Schleife nur bis (inklusive) $j = n-2$ laufen zu lassen.

c) Überprüfen wir nun auch noch die Korrektheit von **IndexOfMin**. Zunächst benötigen wir wieder die Schleifeninvariante, wobei hier der Index i stets so gesetzt sein muss, dass kein Element $A[j..(k-1)]$ kleiner sein darf als $A[i]$. Der Index k bezeichnet dabei die gerade untersuchte Stelle im Array. Wir schreiben also etwa:

$$P \quad = \quad A[i] \leq A[j..(k-1)]$$

Auch in diesem Fall muss die **for**-Schleife ersetzt werden. Außerdem entfernen wir die Variable Ai, die für den korrekten Ablauf nicht notwendig ist, sondern als Hilfestellung für die nachfolgende Implementierung gedacht war. Wir betrachten folgenden Pseudo-Code:

```
{C0}
    i = j;
    k = j + 1;
{C1}
    while (k < n) {
{C2}
        if (A[k] < A[i]) {
            i = k;
        }
{C3}
        k = k + 1;
```

$\{C_4\}$
}
$\{C_5\}$

Die Bedingung $\{C_0\}$ entspricht also wieder der Vorbedingung VOR_{min}, insbesondere ist der Index j gültig.

Es folgen die beiden Initialisierungen von i und k. Vor allem ist der Index i ebenfalls gültig und ein Wert $A[i]$ existiert. Weiterhin sind bei der Bedingung $\{C_1\}$ nun alle Voraussetzungen für die Schleifen-Invariante P erfüllt, da $i = j = k-1$ und $A[i] \leq A[i]$ somit immer wahr ist.

Sollte das Array nach dem Start-Index j keine weiteren Elemente mehr haben, also $n = j+1$, so gilt jetzt $k = n$ und die Schleifenbedinung $k < n$ ist *nicht* erfüllt. In diesem Fall wird die Schleife übersprungen, und $A[i]$ ist offensichtlich auch das minimale Element im Rest – es gibt ja keine anderen Elemente! In diesem Fall also ist die Funktion korrekt, und es gilt:

$$C_5 \overset{j=n-1}{=} C_1 \wedge \neg(k < n) \quad \Rightarrow \quad NACH_{min}$$

Sei das Array nun also größer, und wir laufen in die Schleife ein. Insofern gilt bei $\{C_2\}$ hauptsächlich, dass $k < n$ gilt, und demzufolge k ein gültiger Index ist. Daneben gilt natürlich die Schleifeninvariante:

$$C_2 \quad = \quad n > 0 \ \wedge \ 0 < k < n \ \wedge \ \underbrace{A[i] \leq A[j..(k-1)]}_{= P}$$

Wir werden später den Index k erhöhen, und benötigen dann auch in diesem Fall die Gültigkeit der Invarianten P. Nach der Fallunterscheidung bei $\{C_3\}$ muss also gelten:

$$\begin{aligned} C_3 \quad &= \quad \ldots \ \wedge \ A[i] \leq A[j..k] \\ &= \quad \ldots \ \wedge \ P \ \wedge \ A[i] \leq A[k] \end{aligned}$$

Beachten Sie hierbei, dass statt $k-1$ wie oben nun der End-Index k auf der rechten Seite steht. Dies bedeutet also, dass nun auch $A[k]$ nicht kleiner als $A[i]$ sein darf. Sollte sowieso $A[k] \geq A[i]$ gelten, so gilt die Bedingung der Fallunterscheidung nicht, und die Forderung für C_3 ist offensichtlich erfüllt. Sollte dagegen die Bedingung der Fallunterscheidung zutreffen, so setzen wir $i = k$ neu auf das jetzt kleinste bekannte Element, und die Forderung ist auch hier erfüllt.

Abschließend muss vor $\{C_4\}$ der Index k erhöht werden, wodurch aus der Bedingung C_3 wieder die „normale“ Invariante wird und somit die Korrektheit des Schleifenkörpers gezeigt ist. Durch die Erhöhung von k aber wird vielleicht die Schleifenbedingung verletzt. Somit gilt:

$$C_4 \quad = \quad n > 0 \ \wedge \ (0 < k < n \ \vee \ k \geq n) \ \wedge \ P$$

Sollte nun die Schleife verlassen werden, wenn also $k \geq n$ gilt, so folgt bei $\{C_5\}$ direkt die Nachbedingung $NACH_{min}$ von **IndexOfMin**, da dann alle Array-Elemente untersucht sind und es nach wie vor kein Element gibt, das kleiner ist als $A[i]$. Der Index i zeigt also auf ein minimales Element in $A[j..(n-1)]$:

$$C_5 \overset{j<n-1}{=} C_4 \wedge \neg(k < n) \quad \Rightarrow \quad NACH_{min}$$

Somit ist die Funktion als korrekt verifiziert.

d) Bitte führen Sie äquivalentes fr̈ die Methode **Swap** selbst durch! Der Vertausch-Mechanismus wurde ja bereits in der Vorlesung gezeigt.

Aufgabe 2 Selection Sort II (Beispiellösung)

Die Lösung steht als Quellcode zur Verfügung, den Sie über die Website beziehen können.

Aufgabe 3 Selection Sort III (Beispiellösung)

a) Um die drei Funktionen zu validieren, benötigen wir Testfälle zum individuellen Testen der drei Kandidaten. Testfälle im Allgemeinen sollten sowohl die Standard-Funktionalität abtesten, wie auch das Verhalten in Sonderfällen prüfen.

Im Unterschied zum `assert`, das zur Laufzeit die Bedingungen prüft, erfolgt das Testen separat. Man erstellt also ein separates Programm, dass die geschriebenen Funktionen aufruft und prüft.

- Für die **SelectionSort** sollten zufällig befüllte Arrays verschiedener Längen generiert werden, und dann nachher mittels komponentenweiser Vergleiche überprüft werden, ob tatsächlich sortiert wurde. Das entspricht also einem direkten Abprüfen der Nachbedingung $NACH_{sort}$.

  Als Sonderfall ist zu prüfen, ob ein leeres Array auch korrekt behandelt wird. In diesem Fall heisst das, dass es nicht zum Absturz kommt und keine Änderungen vorgenommen werden.
- Für die **IndexOfMin** wäre es wohl das einfachste, Arrays beliebigen Inhalts zu verwenden, bei denen der Index des minimalen Elements bekannt ist. Genauso könnte man natürlich wieder zufällige Arrays verwenden, bei denen dann die Nachbedingung $NACH_{min}$ getestet wird.

  Als Sonderfälle wären hier zu bedenken, dass das minimale Element ganz vorne oder ganz hinten stehen kann, und vielleicht auch der Fall $j = n - 1$.
- Bei der **Swap** bietet es sich an, ein zufälliges Array zu verwenden und es zu kopieren. Anschließend könnten zwei zufällige Elemente vertauscht werden. Auch dann könnte wieder die Nachbedingung $NACH_{swap}$ direkt überprüft werden.

In allen Fällen haben wir nicht geprüft, was bei ungültigen Eingabewerten passiert. Wir haben also keine Werte gewählt, bei denen die jeweiligen Vorbedingungen verletzt werden. Dieses Vorgehen ist legitim, da das Verhalten der Funktion ja dann undefiniert, und somit jegliche Variante legal ist. Bei echtem Code sollte man natürlich trotzdem die Vorbedingungen auch testen und mit einer Fehlermeldung die weitere Ausführung verweigern.

b) Wenn nur die Sortierfunktion zugänglich ist, kann natürlich **nicht** überprüft werden, ob tatsächlich mittels einem speziellen Sortierverfahren sortiert wurde. Das Ergebnis – eine aufsteigende Reihe – ist ja stets das selbe!

Aufgabe 4 Multiplikation durch Addition (Beispiellösung)

a) Der Unterschied White-Box- bzw. Black-Box-Test kommt hier insofern zum Tragen, als wir sehen, dass verschiedene Werte von b eigentlich nicht geprüft werden müssen. Die b-Werte werden ja nur aufaddiert, und mögliche Fehler ergeben sich alleine aus der Behandlung von a. Schon beim ersten Blick sollte klar sein, dass genau das hier problematisch ist, da wir negative Werte nicht korrekt behandeln.

Beim Testen müssen wir also verschiedene Szenarien erfassen. Der Vergleich kann dabei jeweils gegen die Standard-Implementierung `a * b` erfolgen. Diesen Luxus haben wir normalerweise natürlich nicht direkt und müssen uns selbst um die Referenzergebnisse kümmern.

- Zunächst sind die Fälle mit positivem a zu testen. Im Falle des White-Box-Tests genügt ein beliebiger (zufälliger?) Wert für b. Um ganz sicher zu gehen, würde man wahrscheinlich trotzdem auch die Fälle negativer Werte für b und $b = 0$ testen.
- Dann wären die negativen Werte für a zu prüfen, wobei für b wieder gleiches wie oben gilt. Dieser Test wird beim gegebenen Code – je nach Implementierung – entweder gar nicht terminieren, oder schief gehen.
- Zuletzt wäre noch der Sonderfall $a = 0$ zu bedenken. Aber auch Trivialfälle wie $a = \pm 1$ könnte man eigens testen.

b) *Die Lösung steht primär als Quellcode zur Verfügung, den Sie über die Website beziehen können. Dieser Text umfasst lediglich einige ergänzende Anmerkungen.*

Wie bereits oben erwähnt, ist der Code falsch und behandelt negative Werte nicht korrekt. Es gibt verschiedene Arten, den Code hierfür zu korrigieren. In der Musterlösung nutzen wir unter anderem das Rechengesetz

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