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

WiSe 2012/13
Übungsblatt 5
14. November 2012

Algorithmen und Datenstrukturen

Aufgabe 1 Selection Sort I

In der Vorlesung haben Sie bereits das Suchverfahren *Insertion Sort* kennengelernt. Wir betrachten nun ein ähnliches Verfahren, *Selection Sort*.

Dabei wird aus dem noch unsortierten Teil des Arrays das kleinste Element ausgewählt, und dann nötigenfalls an das Ende des bereits sortierten Teils verschoben. Dazu wird das kleinste Element an den Anfang des unsortierten Teils (und damit an das Ende des sortierten Teils) getauscht. Anschließend wird mit dem nächsten Schritt fortgefahren. Auf diese Weise erhält man eine aufsteigend sortierte Permutation des Original-Arrays.

In Pseudocode – wie in der Vorlesung präsentiert – schaut der Algorithmus dann so aus:

Input: Array $A[0..(n-1)]$ von $n \geq 0$ natürlichen Zahlen
Output: Array A aufsteigend sortiert
SelectionSort(A)**:**

```
for j = 0 to n − 1 {
    i = IndexOfMin(A, j);
    if (j ≠ i) {
        Swap(A, i, j);
    }
}
```

Input: Array $A[0..(n-1)]$ von $n > 0$ natürlichen Zahlen; Startindex j
Output: Index $i \geq j$ des kleinsten Elements im Rest $A[k...(n-1)]$
IndexOfMin(A, j)**:**

```
i = j;
Ai = A[j];
for k = j + 1 to n − 1 {
    if (A[k] < Ai) {
        i = k;
        Ai = A[k];
    }
}
return i;
```

Input: Array $A[0..(n-1)]$ von $n > 0$ natürlichen Zahlen; Indices i, j
Output: Array A mit Zellen i und j vertauscht
Swap(A, i, j)**:**

```
k = A[j];
A[j] = A[i];
A[i] = k;
```

Die Korrektheit von Swap wurde praktischerweise bereits in der Vorlesung gezeigt. Formulieren Sie nun für die anderen beiden Funktionen die entsprechenden Vor- und Nachbedingungen und zeigen Sie die partielle Korrektheit von beiden durch entsprechende Prüfung aller im Code gegebenen Anweisungen. Könnte man den Code optimieren?

Aufgabe 2 (P) Selection Sort II

Implementieren Sie nun den *Selection Sort* anhand des in der vorherigen Aufgabe gegebenen Pseudo-Codes und nutzen Sie dabei die gleiche Aufteilung in einzelne Funktionen.

Eine Methode, die Korrektheit zur Laufzeit zu prüfen sind die `assert`-Statements, die in der Form

`assert(`⟨Logische Bedingung⟩`);`

im Quellcode auftauchen. Ist die angegebene Bedingung *wahr*, wird einfach mit den nachfolgenden Befehlen fortgefahren, ist sie jedoch *falsch*, wird ein Fehler angezeigt, und die Programmausführung abgebrochen. Eine – etwas unsaubere, aber wirksame – Methode zur Weitergabe einer genaueren Fehlermeldung ist die Notation

`assert(`⟨Logische Bedingung⟩ `&& "Fehlermeldung und Lösungshinweis");`

wodurch dem Nutzer ein Hinweis zur Problemlösung gegeben werden kann. Die Existenz derartiger, aussagekräftiger Rückmeldungen ist ein Indikator für guten Programmierstil!

Das `assert`-Statement ist – streng genommen – kein Teil der Programmiersprache C/C++, sondern der Standard-Bibliothek. Um also Zugriff auf diese Funktion zu erhalten, importieren Sie sie mittels

```
#include <assert.h>
```

zu Beginn Ihres Programmcodes. Außerdem steht die Funktion normalerweise nur im *Debug*-Modus zur Verfügung, der aber standardmäßig aktiv ist[1].

Fügen Sie also auch die zuvor erarbeiteten Bedingungen in Form von `assert`-Statements ein. Denken Sie dabei auch an sinnvolle Fehlermeldungen und testen Sie den Mechanismus durch bewusst eingebaute Fehler!

Aufgabe 3 Selection Sort III

a) Die Korrektheit des *Selection Sort* wurde oben gezeigt. Zusätzlich zur Verifikation findet im Optimalfall auch noch eine Validierung des Codes statt, wobei das Verhalten von Programmcode in ausgewählten Testfällen mit bekanntem Ergebnis überprüft wird. Entwerfen Sie solche Testfälle um die drei Funktionen zu validieren, und denken Sie dabei insbesondere an Sonderfälle!

b) Gehen wir davon nun aus, dass – im Sinne eines Black-Box-Tests – nur die eigentliche Sortierfunktion „sichtbar“ ist. Kann man durch einen Test validieren, dass auch sicher mittels *Selection Sort* sortiert wird, und nicht etwa mittels *Insertion Sort*? Wie könnte so ein Test aussehen?

Aufgabe 4 (P) Multiplikation durch Addition

Betrachten wir nun eine einfache Funktion, die das Produkt $a \cdot b$ berechnet, indem a-mal die Zahl b addiert wird. Gehen wir von folgender Implementierung aus:

Input: Ganze Zahlen a und b
Output: Ergebnis c des Produktes $a \cdot b$
Multiply(a,b)**:**

```
c = 0;
for i = 1 to a {
        c = c + b;
}
return c;
```

[1] Sollten Sie trotz offensichtlich falscher Bedingungen keine Fehlermeldung erhalten, fügen Sie bitte **vor** dem `#include <assert.h>` in einer separaten Zeile `#undef NDEBUG` ein. Bei Schwierigkeiten melden Sie sich bitte im Forum oder in einer der Tutorübungen.

a) Welche Testfälle würden Sie bei einem White-Box-Test – also bei Kenntnis des Codes – empfehlen, um das Programm zu validieren? Denken Sie auch an negative Werte und Sonderfälle!

b) Implementieren Sie diese Funktion in C/C++ und wenden Sie Ihre Testfälle (etwa mittels `assert` oder eigenen Vergleichen mittels `if`) an! Ist das Programm so in Ordnung, oder sind Verbesserungen notwendig? Führen Sie diese Korrekturen durch!