Karlsruher Institut für Technologie

Institut für Theoretische Informatik
T.T.-Prof. Dr. Thomas Bläsius
https://scale.iti.kit.edu

Pseudocode-Richtlinien

Wie diese Richtlinien zu verstehen sind

Wie wir bereits gelernt haben, ist Pseudocode ein Werkzeug zur Formulierung von Algorithmen. Im Gegensatz zu Erklärungen im Fließtext ist Pseudocode präziser, erlaubt allerdings im Vergleich zu tatsächlichen Programmiersprachen viele Freiheiten. Es gibt keinen einheitlichen Standard für Pseudocode, was schnell zu Missverständnissen führen kann.
Die wichtigsten Merkmale guten Pseudocodes sind Leserlichkeit und eine konsistente Notation. Wir geben dir im Rahmen von Algorithmen I eine Richtlinie an die Hand, an die du dich halten kannst, um sicherzustellen, dass diejenigen, die deinen Pseudocode lesen, alles auch so interpretieren, wie du es gemeint hast. Im Verlaufe des Semesters werden diese Richtlinien um die neuen Konzepte erweitert, die wir in der Vorlesung kennen lernen werden.
Die Algorithmen auf Übungsblättern und Musterlösungen in diesem Modul werden sich ebenfalls möglichst an diese Richtlinien halten.
Solltest Du Fragen zum Pseudocode haben, dann stelle sie gerne im Discord-Channel oder im entsprechenden ILIAS-Forum.

1 Variablen und ihre Typen

Jede Variable hat genau einen Typ (eine Zahl, eine Zeichenkette, ...). Um Verwirrung und Uneindeutigkeiten zu vermeiden, schließen wir Type-Morphing aus: eine Variable hat ihre gesamte Lebenszeit über den gleichen eindeutigen Typ. Variablen sollten, wann immer möglich, sinntragende Namen haben. Zur besseren Verständlichkeit bietet es sich ggf. an, den Typ im Namen zu nennen (Beispiel: `inputString`, `resultList`, ...). Das trägt nicht nur zur Lesbarkeit des Pseudocodes bei, sondern hilft dabei, die Funktion einer Variable in einem Algorithmus zu kommunizieren. Variablenzuweisung findet mit dem Operator $:=$ statt (Beispiel: `startIndex` $:=$ `0`). Oft ist es auch hilfreich, explizit den Typ einer Variable anzugeben, welchen man zwischen `:` und `=` einfügt (`var:` $\mathbb{N} = 5$). Die explizite Verwendung von Typen empfehlen wir besonders, wenn beim Schreiben von Pseudocode noch Unsicherheiten vorhanden sind.

2 Operatoren

- Vergleiche: $<, \leq, =, \geq, >, \neq$
- Logik: $\wedge, \vee, \neg$
- Arithmetik: $\cdot, /, +, -,$ `div` , `mod` , $\lceil\ \rceil, \lfloor\ \rfloor$

3 Kontrollfluss

Auch im Pseudocode gibt es Verzweigungen und Schleifen, wie sie dir schon bekannt sein sollten. Das wichtigste bei handschriftlichem Pseudocode ist, die Anweisungen, die zur Kontrollstruktur gehören, eindeutig von den anderen Anweisungen abzugrenzen.

Genügen Einrückungen?

Nein. Hier gibt es einen großen Unterschied zwischen dem Pseudocode auf Folien und handschriftlichen Abgaben. In automatisch generiertem Pseudocode sind Einrückungen eindeutig – in handschriftlichen Abgaben aller Erfahrung nach nicht. Deswegen sind Einrückungen allein **nicht** ausreichend, um Kontrollblöcke voneinander abzugrenzen. Verwende deswegen immer visuelle Marker (beispielsweise vertikale Linien), um Anweisungsblöcke voneinander abzugrenzen.

3.1 Verzweigungen

if condition **then**
| ...
else if otherCondition **then**
| ...
else
| ...
end

3.2 Schleifen

for $i \in \{1, \ldots, n\}$ **do**
| ...
end

while condition **do**
| ...
end

Bei Schleifenanweisungen wird implizit angenommen, dass die Menge, über die iteriert wird, geordnet ist.

4 Kommentare

Zum besseren Verständnis empfehlen wir Pseudocode zu kommentieren. Der Beginn von Kommentaren kann beispielsweise mit den Symbolen `//` am Ende einer Zeile signalisiert werden.

5 Argumente und Rückgabewerte

Eine Methode, also die Pseudocode-Formulierung eines Algorithmus, benötigt einen Namen und eine Menge an Argumenten in Form von Variablen. Eine Methode kann einen Rückgabewert haben, dessen Typ am Ende bezeichnet werden kann:

NAME(`argument1: type1, argument2: type2, ...`): `returnType`

Die eigentliche Rückgabe erfolgt dann über das Schlüsselwort **return** . Zum Beispiel:

```
DIV(divident : ℕ, divisor : ℕ) : ℕ
    quotient : ℕ = 0                    // wird am Ende zurückgegeben
    temp : ℕ = 1                        // Zählvariable
    while temp · divisor ≤ divident do
        quotient ≔ temp
        temp ≔ temp + 1
    end
    return quotient
```

Der Aufruf der Methode DIV erfolgt dann beispielsweise über DIV(5, 4).

6 Arrays und Listen

In diesem Abschnitt wird der Lesbarkeit halber $\mathbb{N}$ stellvertretend für jeden beliebigen Typen verwendet.

6.1 Arrays

Arrays werden in Algorithmen I bei 0 beginnend indiziert. Wir unterscheiden nicht zwischen dynamischen Arrays und solchen fester Größe.

- Zuweisung mit initialer Größe n: A: $[\mathbb{N};\ n]$
- Zuweisung ohne initiale Größe: A: $[\mathbb{N}]$
- Zweidimensionales Array mit $n \times m$ Einträgen: A: $[[\mathbb{N};\ n];\ m]$
- Zugriff auf das i-te Element: A$[i]$
- Zugriff in zweidimensionalen Arrays: A$[i][j]$

6.2 Listen

- Zuweisung: L: List$\langle\mathbb{N}\rangle$
- Zugriff auf Elemente: erfolgt über die Nutzung von L.first, L.next, L.prev etc.

6.3 Iteration über Listen und Arrays

Um über alle Elemente zu iterieren, die in einem Array bzw. einer Liste enthalten sind, verwenden wir abkürzend die folgende Schreibweise. Dabei wird implizit angenommen, dass über Arrayelemente in aufsteigender Index-Reihenfolge und über Listenelemente vom first- zum last-Element iteriert wird.

$A\colon [\mathbb{N};\ n]$
...
for $a \in A$ **do**
| ...
end

$L\colon \text{List}\langle\mathbb{N}\rangle$
...
for $l \in L$ **do**
| ...
end

6.4 Laufzeiten

Operation	Liste	einfach verkettete Liste	Array	Erklärung ($*$)
first	1	1	1	
last	1	1	1	
insert	1	1*	n	nur insertAfter
remove	1	1*	n	nur removeAfter
pushBack	1	1	1*	amortisiert
pushFront	1	1	n	
popBack	1	n	1*	amortisiert
popFront	1	1	n	
concat	1	1	n	
splice	1	1	n	
findNext	n	n	n	

7 HashMaps

Zum Anlegen einer HashMap verwenden wir folgenden Syntax

```
1: map := HashMap {key type → value type}
```

Beispielsweise

```
1: map := HashMap {ℕ → ℕ}
```

Der Zugriff erfolgt über Funktionen `get(key)`, `set(key, value)` und `delete(key)`, mit welchen in erwartet konstanter Zeit nach Elementen gesucht werden kann, sowie Elemente eingefügt (beziehungsweise überschrieben) und gelöscht werden können.
Ist zu einem Schlüssel k kein Element in der HashMap enthalten, so gibt `get` für Eingabe k den Fehlerwert $\perp$ zurück.

8 Graphen

Wenn wir Graphen im Pseudocode verwenden, benutzen wir dabei keine konkreten Repräsentationen. Stattdessen notieren wir Graphen als Tupel (V, E) mit Knotenmenge V und Kantenmenge E.
Um über alle Knoten bzw. Kanten zu iterieren, verwenden wir die gleiche abkürzende Schreibweise wie für Arrays und Listen. Du darfst annehmen, dass Knoten bzw. Kanten nummeriert sind und diese Nummerierung nutzen:

```
G = (V, E)
...
for v ∈ V do
|   ...
end
```

```
G = (V, E)
...
for i ∈ {0, ..., |V| − 1}
do
|   ...
|   node: Node = v_i
|   ...
end
```

```
G = ({v_0, ..., v_{n−1}}, E)
...
for v_i ∈ V do
|   ...
|   var: ℕ = i
|   ...
|
end
```

Je nachdem, ob ein Graph gerichtet oder ungerichtet ist, kannst du Kanten auch als Tupel bzw. zweielementige Mengen auffassen:

```
G = (V, E)
...
for e ∈ E do
|   ...
end
```

```
G_directed = (V, E)
...
for (u, v) ∈ E do
|   ...
end
```

```
G_undirected = (V, E)
...
for {u, v} ∈ E do
|   ...
end
```

Die Menge der Nachbarn eines Knotens v bezeichnen wir mit $N(v)$:

```
G = (V, E)
...
for v ∈ V do
|   ...
|   grad_v : ℕ = |N(v)|
|   for u ∈ N(v) do
|   |   ...
|   end
|   ...
end
```

9 Union-Find

Die Union-Find Datenstruktur wird mit einer endlichen Menge von Elementen initialsiert. Es stehen zwei Operationen zur Verfügung:

- `find(`x`)` liefert den eindeutigen Vertreter der Menge in der x enthalten ist
- `union(`x`, `y`)` vereint die beiden Mengen in denen x und y enthalten sind (x und y müssen dabei nicht die Vertreter der jeweiligen Mengen sein)

$S = \{1, 2, 3, 4\}$
...
$U = \text{UnionFind}(S)$
U.`union(2, 3)`
$x = U$.`find(3)`
$y = U$.`find(2)` // $x = y$