

Aufgabe 7.1 (P) Quicksort

Ziel dieser Aufgabe ist es, den *Quicksort*-Algorithmus über Arrays ganzer Zahlen zu implementieren. Der Algorithmus hat folgenden Ablauf:

Wähle aus dem zu sortierenden Array ein beliebiges Element aus (im Folgenden *Pivot*-Element genannt):

5	1	3	9	1	5	3
---	---	---	---	---	---	---

Ordne die Elemente des Arrays so um, dass links vom *Pivot*-Element nur Elemente sind, die kleiner als das *Pivot*-Element (Links-Part) und rechts nur solche, die größer oder gleich dem *Pivot*-Element sind (Rechts-Part):

1	1	3	3	5	9	5
---	---	---	---	---	---	---

Wende den Quicksort-Algorithmus auf den Links-Part und auf den Rechts-Part an:

1	1		
3			
3	5	5	9

Sobald die zu sortierenden Teilbereiche die Länge ≤ 1 haben, brich das Verfahren ab:

1	1	3	3	5	5	9
---	---	---	---	---	---	---

Zur Implementierung des gesamten Algorithmus benötigen wir folgende Hilfsfunktionen:

1. *Tausch von Arrayelementen*: Schreiben Sie eine Java-Methode `void swap(int[] numbers, int i, int j)`, die in einem Array ganzer Zahlen `numbers` die Einträge mit den Indizes `i` und `j` vertauscht.
2. *Partitionierung eines Array-Ausschnitts*: Die Java-Funktion `int partition(int[] numbers, int left, int right)` soll zuerst den Wert `numbers[right]` als *Pivotwert* setzen. Danach sollen die Elemente des Arrays `numbers` zwischen den Positionen `left` und `right` *inklusive* der Pivotzelle derart vertauscht werden, dass links des linken Elementes mit dem *Pivotwert* nur kleinere Zahlen und rechts davon nur größere oder gleich große Zahlen stehen. Zurückgegeben wird der Index der bzw. einer Zelle, die danach den Pivotwert enthält.
3. Implementieren Sie mit Hilfe der bisher erarbeiteten Funktionen den Quicksort-Algorithmus rekursiv.
4. Erstellen Sie eine `main`-Routine, in der der Quicksort-Algorithmus für ein Array, dessen Elemente zufällig mit Zahlen belegt sind, getestet wird.

Hinweis: Für $n > 0$ lässt sich eine Zufallszahl zwischen 0 und $n - 1$ mit dem Befehl `int number = (int) (n*Math.random());` erzeugen.

Aufgabe 7.2 (P) Steck-Assembly

In den Hausaufgaben soll ein Interpreter für eine Assembler-Sprache implementiert werden. In dieser Aufgabe sollen Sie ein Beispielprogramm in dieser Assembler-Sprache mit Ihrem Tutor besprechen und verstehen. Es handelt sich um das Programm zur Berechnung des größten gemeinsamen Teilers:

```

1 public class Example extends MiniJava {
2     public static int ggt(int a, int b) {
3         if (b > a) {
4             int temp = b;
5             b = a;
6             a = temp;
7         }
8         while (b != 0) {
9             int temp = b;
10            b = a % b;
11            a = temp;
12        }
13        return a;
14    }
15
16    public static void main(String[] args) {
17        int a = read();
18        int b = read();
19        write(ggt(a, b));
20    }
21 }

```

Das entsprechende Programm in unserer Assembly-Sprache sieht aus wie folgt:

```

1 IN
2 IN
3 LDI ggt
4 CALL 2
5 OUT
6 HALT
7
8 // 2 Argumente (a, b)
9 ggt:
10 ALLOC 1
11 // Tausch von größerer Zahl nach vorne
12 LDS 0
13 LDS -1
14 JLT loop
15 LDS -1
16 STS 1
17 LDS 0
18 STS -1
19 LDS 1
20 STS 0
21 // Hauptschleife
22 loop:
23 LDS 0
24 STS 1

```

```

25 LDS -1
26 LDS 0
27 MOD
28 STS 0
29 LDS 1
30 STS -1
31 LDS -0
32 LDI 0
33 JNE loop
34 LDS -1
35 // Wir geben zwei Argumente und eine lokale Variable frei
36 RETURN 3

```

Lesen Sie die Beschreibung der Assembler-Sprache in den Hausaufgaben, um das Programm zu verstehen!

Aufgabe 7.3 (P) Toolbox

Lösen Sie die folgenden Teilaufgaben mittels rekursiven Methoden. Es dürfen keine Schleifen vorkommen.

Eine ganze Zahl $n \in \mathbb{Z}$ ist gerade, wenn $n = \pm 2k$ und ungerade, wenn $n = \pm 2k + 1$ für ein beliebiges $k \in \mathbb{N}_0$ gilt.

Schreiben Sie eine Klasse `Toolbox` und implementieren darin die Methode

- `public static int evenSum(int n)`, welche die Summe aller ganzen geraden Zahlen von 0 bis einschließlich n berechnet, falls $n \geq 0$, oder von n bis 0 berechnet, falls $n < 0$ ist. Als arithmetische Operatoren sind nur die Additions- bzw. Subtraktionsoperatoren `+` und `-` erlaubt (die binären sowie unären Varianten jeweils). D.h. insbesondere der Multiplikationsoperator `*` sowie `*=` sind nicht erlaubt. Gehen Sie davon aus, dass $n \in [-10^4, +10^4]$.

Beispiel: `evenSum(-8)` liefert -20 als Ergebnis.

- `public static int multiplication(int x, int y)`, welche zwei Integer x und y multipliziert und das Ergebnis zurückliefert. Als arithmetische Operatoren sind nur die Additions- bzw. Subtraktionsoperatoren `+` und `-` erlaubt (die binären sowie unären Varianten jeweils). D.h. insbesondere der Multiplikationsoperator `*` sowie `*=` sind nicht erlaubt. Gehen Sie davon aus, dass $x, y \in [-10^4, +10^4]$.
- `public static void reverse(int[] m)`, welche ein `int`-Array m als ersten Parameter erwartet und die Einträge in dem selbigen Array wie folgt vertauscht: Sei m ein Array der Form $[a_1, a_2, \dots, a_n]$, dann soll nach dem Aufruf `reverse(m)` das Array m die Form $[a_n, \dots, a_2, a_1]$ haben. D.h. die (Hilfs-)Methode verändert das Array m und erstellt keine neuen Arrays (auch keine temporären).
- `public static int numberOfOddIntegers(int[] m)`, welche ein `int`-Array m als ersten Parameter erwartet und die Anzahl an ungeraden Integeren aus dem Array m zurückgibt. Das Array m darf nicht verändert werden.

Beispiel: Sei `int[] m = new int[]{4, 7, 42, 5, 1, -5, 0, -4, -3}`, dann liefert `numberOfOddIntegers(m)` als Ergebnis 5.

- `public static int[] filterOdd(int[] m)`, welche ein `int`-Array `m` als ersten Parameter erwartet und ein `int`-Array zurückliefert. Das Ergebnis-Array enthält genau alle ungeraden Integer aus dem Array `m`. Des Weiteren spiegelt das Ergebnis-Array die Ordnung der Elemente aus dem Array `m` wider. D. h. hat das Array `m` an der Stelle i und j mit $i < j$ ungerade Integer x und y , dann finden sich die Integer x und y in dem Ergebnis-Array an Stellen l und k wieder, sodass $l < k$ gilt. Das Array `m` darf nicht verändert werden.

Beispiel: Sei `int[] m = new int[]{4, 7, 42, 5, 1, -5, 0, -4, -3}`, dann liefert `filterOdd(m)` als Ergebnis ein Array, welches äquivalent zu folgendem ist: `new int[]{7, 5, 1, -5, -3}`.

Hinweis: Sie dürfen zusätzliche (Hilfs-)Methoden definieren, aber keine Member-Variablen. Alle Variablen, die nicht innerhalb einer Methode deklariert sind (also weder Parameter noch lokale Variablen sind), sind Member-Variablen.

Aufgabe 7.4 (P) Fakultät

Die Fakultätsfunktion ist für positive Zahlen wie folgt definiert:

$$n! = \prod_{i=1}^n i \quad (n \geq 1)$$

$$0! = 1$$

Schreiben Sie ein Programm `Fak.java`, das die Fakultätsfunktion auf folgende Weisen berechnet.

1. In der Methode `public static int facRec(int n)` soll die Fakultätsfunktion *rekursiv* (aber nicht end-rekursiv) berechnet werden. Der Parameter `n` gibt dabei den Integer an, für den die Fakultät berechnet werden soll.
2. In der Methode `public static int facTailRec(int n)` soll die Fakultätsfunktion *endrekursiv* berechnet werden. Dafür benötigen Sie eine zusätzliche Hilfsmethode `private static int facTailRecHelper(int n, int k)`. Der Parameter `n` gibt dabei den Integer an, für den die Fakultät berechnet werden soll. Der Parameter `k` gibt das bisher berechnete Zwischenergebnis an.
3. In der Methode `facIt(int n, int k)` soll die Fakultätsfunktion *iterativ* durch Umwandlung der Endrekursion aus der Methode `facTailRec(int n, int k)` berechnet werden. D. h. die Berechnung erfolgt in einer `while(true){...}` Schleife.

Aufgabe 7.5 (P) Das Ende der Rekursion

- Gegeben sind die folgenden Funktionen $f : \mathbb{N} \rightarrow \mathbb{N}$ und $g : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$.

$$f(x) = g(x, 0)$$

$$g(x, y) = \begin{cases} x^y & x < 10 \\ (x \% 10)^y + g(x/10, y + 1) & x \geq 10 \end{cases}$$

Hierbei sei `%` der Modulo-Operator wie in Java und `/` der Operator für ganzzahlige Division ebenfalls wie in Java. D. h. $12345/10 = 1234$ und $12345 \% 10 = 5$.

Erstellen Sie zunächst eine rekursive, aber nicht endrekursive Implementierung der Funktion f in der Methode `public static int frec(int x)`.

Implementieren Sie die Funktion f dann *endrekursiv* in der Methode `public static int ftailrec(int x)`. Benötigen Sie dazu eine Hilfsfunktion?

Verwenden Sie *keine* Schleifen und keine Membervariablen in Ihrer Implementierung.

- Entschleifen Sie die Partitionsfunktion aus der Quicksortaufgabe.

Die Hausaufgabenabgabe erfolgt über Moodle. Bitte geben Sie Ihren Code als UTF8-kodierte (ohne BOM) Textdatei(en) mit der Dateiendung `.java` ab. Geben Sie **keine** Projektdateien Ihrer Entwicklungsumgebung ab. Geben Sie **keinen** kompilierten Code ab (`.class`-Dateien). Geben Sie Ihren Code **nicht** als Archiv (z.B. als `.zip`-Datei) ab. Nutzen Sie **keine** Ordner in Moodle. Nutzen Sie **keine** Packages. Achten Sie darauf, dass Ihr Code kompiliert. Bitte vermerken Sie aus Datenschutzgründen nicht Ihren Namen oder Ihre Matrikelnummer im Code. Hausaufgaben, die sich nicht im vorgegebenen Format befinden, werden nur mit Punktabzug oder gar nicht bewertet.

Aufgabe 7.6 (H) SlowSort und EvenSlowerSort [4 Punkte]

In der Vorlesung haben Sie *Divide and Conquer* Algorithmen kennengelernt. In dieser Aufgabe werden Sie einen *Multiply and Surrender* Algorithmus implementieren, den besonders effizienten *Slowsort*-Algorithmus.

Slowsort ist ein rekursiver Algorithmus, der ein Array in zwei Teile aufteilt und am Ende die jeweils größten Elemente vergleicht. Er läuft ab wie folgt:

1. Sortiere die erste Hälfte des Arrays rekursiv mittels SlowSort.
2. Sortiere die zweite Hälfte des Arrays rekursiv mittels SlowSort.
3. Finde das Maximum des gesamten Arrays, indem man die jeweils letzten Elemente der Hälften vergleicht, und platziere es an das Ende des gesamten Arrays.
4. Sortiere das Array ohne das letzte Element (dem Maximum aus 3.)

Um die Laufzeit weiter zu verbessern, führen wir eine Variante des *SlowSort*-Algorithmus ein, den *EvenSlowerSort*. Bei diesem Algorithmus unterteilen wir das Array in *drei* Teile (im Gegensatz zu *zwei* in *SlowSort*).

EvenSlowerSort ist ein rekursiver Algorithmus, der ein Array in drei Teile aufteilt und am Ende die jeweils größten Elemente vergleicht. Er läuft ab wie folgt:

1. Sortiere den ersten Teil des Arrays rekursiv mittels EvenSlowerSort.
2. Sortiere den zweite Teil des Arrays rekursiv mittels EvenSlowerSort.
3. Sortiere den dritten Teil des Arrays rekursiv mittels EvenSlowerSort.
4. Finde das Maximum des gesamten Arrays, indem man die jeweils letzten Elemente der Teile vergleicht, und platziere es an das Ende des gesamten Arrays.
5. Sortiere das Array ohne das letzte Element (dem Maximum aus 4)

Implementieren Sie die folgenden beiden Methoden:

1. `private static void slowSort(int[] array)`. Die Methode bekommt als einzigen Parameter ein Array gefüllt mit unbekannten Werten und sortiert dieses Array mit dem oben beschriebenen *SlowSort*-Algorithmus.
2. `private static void evenSlowerSort(int[] array)`. Die Methode bekommt als einzigen Parameter ein Array gefüllt mit unbekannten Werten und sortiert dieses Array mit dem oben beschriebenen *EvenSlowerSort*-Algorithmus.

Hinweis: Sie dürfen, wenn benötigt, Hilfsmethoden schreiben. Es ist überdies erlaubt, dass `slowSort()` bzw. `evenSlowerSort()` lediglich eine Hilfsmethode aufruft, welche dann das rekursive Verfahren implementiert.

Aufgabe 7.7 (H) Springuine [6 Punkte]

In dieser Aufgabe geht es noch einmal darum, Pinguinen zu helfen.¹ Die Tierpflegerin

¹Beim Sporttreiben haben die Pinguine leider viele Kalorien verloren und sind nun auf rasche Hilfe angewiesen, um nicht zu verhungern. <http://www.itv.com/news/westcountry/2016-04-25/penalty-shoot-out-at-the-penguin-enclosure/>.

möchte die knuffigen Tiere füttern und durchquert dazu das Pinguingehege, bis sie alle Pinguine angetroffen hat (indem sie auf ein angrenzendes Feld gekommen ist). Da der Tierpfleger leider große Ähnlichkeit mit einem gefräßigen Walross aufweist, nehmen die verängstigten Tierchen Reißaus. Je nach ihrem Fluchtverhalten lassen die Pinguine sich in genau fünf Kategorien einteilen, wobei sie grundsätzlich auf *vertikal oder horizontal* benachbarte Felder wechseln (mit Ausnahme der Springuine):

1. Fauluine (Integer-Konstante PENGUIN_000): Bewegen sich nie.
2. Zufulline (Integer-Konstante PENGUIN_001): Laufen auf ein zufällig gewähltes (vertikal oder horizontal) benachbartes Feld.
3. Wechsuline (Integer-Konstante PENGUIN_010): Laufen ganz zu Beginn schrittweise nach *rechts*, d. h. mit einem Offset `[1][0]`, bis sie erstmals an eine Wand (WALL) stoßen. Dann drehen sie aus ihrer eigenen Sicht nach links und wenden danach die Rechte-Hand-Regel (RHR, siehe Aufgabe 6.4) an. Erreichen sie dabei den Eingang, drehen sie sich um 180 Grad und laufen weiter gemäß RHR. Für die Berechnung des nächsten Feldes nach RHR zählen angrenzende Pinguine prinzipiell als freie Felder (also nicht als Wand), jedoch muss ggf. gewartet werden, bis der Artgenosse den Platz verlassen hat.
4. Springuine (Integer-Konstante PENGUIN_011): Springen auf ein beliebiges freies Feld im Pinguingehege.
5. Schlauline (Integer-Konstante PENGUIN_100): Maximieren in jedem Spielschritt, falls möglich, die Distanz zum Tierpfleger. Die Distanz ist als Summe der vertikalen und horizontalen Distanz definiert. Beispiel: Unterscheiden sich die x-Werte um 3 und die y-Werte um 2, ist die Distanz $3 + 2 = 5$. Gibt es mehrere Möglichkeiten, darf eine davon frei gewählt werden.

Sind alle Felder besetzt, auf die der Pinguin wechseln möchte, bleibt er stehen. Im Pinguingehege gibt es genau einen Eingang bei `[1][0]`, von dem aus die Tierpflegerin ihre Fütterungstour startet.

Umsetzung:

Zum Codegerüst gehören die Klasse `PenguinPen` zur Darstellung des Geheges (und die Bilddateien `tux1.png` bis `tux5.png`) sowie die Klasse `HilfPingu`, in der Sie die Methode

```
public static void move(int direction)
```

implementieren sollen. Diese wird aufgerufen, wenn die Tierpflegerin sich bewegen will, also eine der Bewegungs-Tasten gedrückt wurde. Die Konstanten für den Aufruf befinden sich in der Klasse `PenguinPen`: `MOVE_LEFT`, `MOVE_RIGHT`, `MOVE_UP`, `MOVE_DOWN`. Ist das gewünschte Nachbarfeld frei, soll der nächste *Spielschritt* ausgeführt und der Gehegezustand in der globalen Variablen `penguinPen` der Klasse `HilfPingu` aktualisiert und die Aktualisierung durch Aufruf der `PenguinPen.draw`-Methode ausgegeben werden. Zu einem Spielschritt gehören (in dieser Reihenfolge, wobei Sie auch die letzten beiden Schritte vertauschen dürfen):

1. Füttern der Pinguine auf angrenzenden Feldern. Als angrenzend gelten dabei alle (≤ 8) Felder, die horizontal bzw. vertikal bzw. diagonal genau einen Schritt entfernt sind. Die Pinguine sollen vom Pinguingehege verschwinden².

²Diese Pinguine haben Vertrauen in die Tierpflegerin gewonnen und laufen ihr fortan auf dem weiteren Weg durch das Pinguingehege hinterher, was aber nicht angezeigt werden soll bzw. kann.






2. Aktualisieren der Positionen der Pinguine gemäß obigen Fluchtregeln in beliebiger Reihenfolge. Kann dabei ein Pinguin unter Beachtung aller Regeln nicht ziehen, bleibt er, wo er ist.
3. Aktualisieren der Position des Tierpflegers (Konstante `ZOOKEEPER`), falls möglich.

Sie dürfen in der Klasse `HilfPingu` gerne weitere Methoden und Attribute hinzufügen, aber den vorhandenen Code unterhalb der Markierung nicht ändern, sondern nur *ergänzen*.

Zum Debuggen, Testen und Erleichtern der Korrektur geben Sie bitte auch jede einzelne Veränderung auf der Konsole aus. Beispiel:

```
...
(3,3) wird frei -- Pinguin ist satt.
(2,2) ==> (2,3) -- Spielschritt.
(3,6) ==> (4,6) -- Pinguin flüchtet.
(5,7) ==> (5,8) -- Pinguin flüchtet.
...
```

Hinweis: In der Tabelle finden Sie die Zuordnung der Pinguinarten zum ausgegebenen Symbol (alternativ: Farbe des angezeigten Punktes).

Art	Konstante	Symbol	alternativ
Fauluine	<code>PENGUIN_OOO</code>		gelb
Zufulline	<code>PENGUIN_OOI</code>		blau
Wechsuline	<code>PENGUIN_OIO</code>		schwarz
Springuine	<code>PENGUIN_OII</code>		magenta
Schlauline	<code>PENGUIN_IOO</code>		grün

Hinweis: Für $n > 0$ lässt sich eine Zufallszahl zwischen 0 und $n - 1$ mit dem Befehl `int number = (int) (n*Math.random());` erzeugen.

Hinweis: Nutzen Sie die Methode `generatePenguinPen(int width, int height)` zur Erzeugung eines zufälligen Startzustands der übergebenen Breite und Höhe und die Methode `generateStandardPenguinPen(int width, int height)` zur Erzeugung des immer gleichen Startzustands.

Tip: Der Zufallswert zur Erzeugung des Startzustands wird auf der Konsole ausgegeben und kann für die Methode `generatePenguinPen(int width, int height, int seed)` zur nochmaligen Erzeugung desselben Startzustands eingesetzt werden.

Aufgabe 7.8 (H) Steckmaschine

[10 Punkte]

Wie Sie bereits wissen, können die Prozessoren von Computern komplexe Programme – wie z.B. solche, die in Java geschrieben sind – nicht direkt verarbeiten. Stattdessen müssen diese in einfache Befehle übersetzt werden; man nennt ein Programm in dieser Form ein *Assemblerprogramm* bzw. spricht von der Assembler-Sprache³. In dieser Aufgabe soll ein Interpreter für eine einfache Assemblersprache („Steck-Assembly“) in Java implementiert werden.

³Dies ist allerdings ein Stück weit irreführend, weil jede Maschine ihre eigene Form von Assembler-Sprache verwendet.

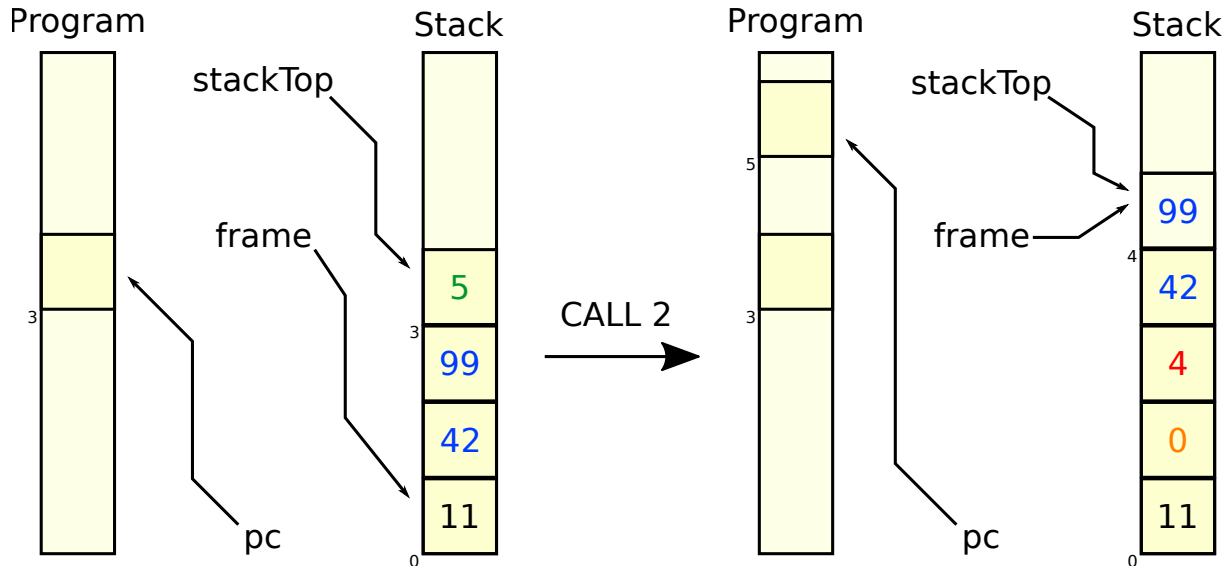
Die Maschine, die hier simuliert werden soll, verfügt als Speicher lediglich über einen **int**-Stack der Größe 128. Alle Instruktionen liegen in einem separaten Programmspeicher; sie konsumieren ihre Argumente vom Stack und legen ihre Ergebnisse wiederum auf den Stack. Zusätzlich verfügen einige Instruktionen über ein einzelnes Argument, welches direkt im Programm als Teil der Instruktion geführt wird, ein sogenanntes *Immediate*. Es stehen folgende Instruktionen zur Verfügung:

Instruktion	Immediate	Beschreibung
NOP	keins	Diese Instruktion hat keinerlei Effekt.
ADD	keins	Addiert o_1 und o_2
SUB	keins	Berechnet $o_1 - o_2$
MUL	keins	Multipliziert o_1 mit o_2
MOD	keins	Berechnet den Divisionsrest von o_1/o_2
LDI	v (16 Bit)	Lädt einen 16-Bit-Wert v auf den Stack; die oberen 16 Bit sollen den Wert 0 haben.
LDS	i (16 Bit)	Kopiert die durch i indizierte Frame-Zelle nach oben auf den Stack
STS	i (16 Bit)	Nimmt einen Wert vom Stack und speichert ihn in der durch i indizierten Frame-Zelle ab
JUMP	i (16 Bit)	Springt zur Instruktion mit dem Index i
JE	i (16 Bit)	Springt zur Instruktion mit dem Index i , wenn $o_1 = o_2$ gilt; o_1 und o_2 werden vom Stack entfernt.
JNE	i (16 Bit)	Springt zur Instruktion mit dem Index i , wenn $o_1 \neq o_2$ gilt; o_1 und o_2 werden vom Stack entfernt.
JLT	i (16 Bit)	Springt zur Instruktion mit dem Index i , wenn $o_1 < o_2$ gilt; o_1 und o_2 werden vom Stack entfernt.
CALL	n (16 Bit)	Ruft die Funktion an Index o_1 mit n Argumenten auf (siehe unten)
RETURN	n (16 Bit)	Kommt von einer Funktion zurück (siehe unten)
IN	keins	Liest einen Wert mittels <code>read()</code> ein und legt ihn auf den Stack
OUT	keins	Gibt o_1 mittels <code>write()</code> aus
HALT	keins	Beendet das Programm
ALLOC	n (16 Bit)	Reserviert Platz für n lokale Variablen; nur am Funktionsanfang gültig

Es sei hier o_1 das oberste Stack-Element, o_2 das zweit-oberste Stack-Element. Da die Steckmaschine keine Register und nur einen Stack kennt, müssen auch lokale Variablen auf dem Stack abgespeichert werden. Dazu verwaltet die Maschine den sogenannten *Frame-Pointer*. Der Frame-Pointer zeigt auf diejenige Stack-Zelle, ab der lokale Variablen zu finden sind. Direkt vor dem Frame-Pointer befinden sich die Parameter der Funktion. Der Frame-Pointer lässt sich über die Instruktionen **LDS** und **STS** verwenden – so lädt zum Beispiel die Instruktion **LDS -1** das vorletzte Funktionsargument, die Instruktion **STS 1** überschreibt die erste lokale Variable mit dem obersten Stack-Wert. Der Frame-Pointer muss beim Aufruf von Unterprogrammen auf dem Stack gespeichert und beim Rücksprung wiederhergestellt werden, da sich der Frame-Pointer immer auf die aktuell ausführende Funktion bezieht.

Für Funktionsaufrufe bietet unsere Maschine die **CALL n**-Instruktion an, die eine Funktion mit n Argumenten aufruft. Dazu nimmt sie zunächst die Funktionsadresse f (Index

in das Programm-Array) vom Stack. Anschließend werden n Argumente vom Stack konsumiert. Die Instruktion legt nun die Rücksprungadresse (wiederum ein Index in das Programm-Array) und den aktuellen Frame-Pointer auf den Stack; dahinter kommen die Funktionsargumente. Achten Sie bei Ihrer Implementierung darauf, die Argumente in unveränderter Reihenfolge nach dem Frame-Pointer auf den Stack zu legen. Die folgende Abbildung zeigt ein Beispiel eines Aufrufs einer Funktion mit zwei Argumenten:



Im Bild ist die Adresse der aufgerufenen Funktion grün, Funktionsargumente blau, der gespeicherte Frame-Pointer orange und die Rücksprungadresse rot markiert. Die **RETURN**-Instruktion erwartet den Rückgabewert ganz oben auf dem Stack. Sie enthält zusätzlich ein Immediate, welches angibt, wie viele Stack-Zellen der Funktion entfernt werden müssen - dies entspricht immer der Summe der formalen Parameter und lokalen Variablen. Die Instruktion stellt den ursprünglichen Frame-Pointer und Befehlszähler wieder her und legt den Rückgabewert oben auf den Stack.

Jede Instruktion wird durch einen **int**-Wert repräsentiert. Die oberen 16 Bit enthalten dabei den *Opcode* der Instruktion (vgl. die Konstanten im Code-Gerüst weiter unten), die unteren 16 Bit enthalten entweder das Immediate (wenn ein solches vorhanden ist) oder sind gleich 0. Wir stellen ein Assemblerprogramm dar, indem wir pro Zeile eine Instruktion schreiben. Die Instruktionen haben die oben aufgeführten Namen, einige von ihnen haben ein Immediate, welches direkt auf den Namen der Instruktion folgt. Durch *labels* können Stellen im Programm für Sprünge markiert werden. Ein kleines Beispielprogramm, welches Zahlen ab 0 ausgibt, könnte so aussehen:

```

1  ALLOC 1
2  LDI 0
3  STS 1
4  from:
5  LDS 1
6  OUT
7  LDS 1
8  LDI 1
9  ADD
10 STS 1
11 JUMP from

```

Hier wurde das Label *from* für den Rücksprung am Ende verwendet. Beginnen Sie Ihre Implementierung mit folgendem Gerüst:

```
1 import java.util.Scanner;
2
3 public class Interpreter extends MiniJava {
4     public static final int NOP = 0;
5     public static final int ADD = 1;
6     public static final int SUB = 2;
7     public static final int MUL = 3;
8     public static final int MOD = 4;
9     public static final int LDI = 5;
10    public static final int LDS = 6;
11    public static final int STS = 7;
12    public static final int JUMP = 8;
13    public static final int JE = 9;
14    public static final int JNE = 10;
15    public static final int JLT = 11;
16    public static final int IN = 12;
17    public static final int OUT = 13;
18    public static final int CALL = 14;
19    public static final int RETURN = 15;
20    public static final int HALT = 16;
21    public static final int ALLOC = 17;
22
23    static void error(String message) {
24        throw new RuntimeException(message);
25    }
26
27    public static String readProgramConsole() {
28        @SuppressWarnings("resource")
29        Scanner sin = new Scanner(System.in);
30        StringBuilder builder = new StringBuilder();
31        while (true) {
32            String nextLine = sin.nextLine();
33            if (nextLine.equals("")) {
34                nextLine = sin.nextLine();
35                if (nextLine.equals(""))
36                    break;
37            }
38            if (nextLine.startsWith("//"))
39                continue;
40            builder.append(nextLine);
41            builder.append('\n');
42        }
43        return builder.toString();
44    }
45
46    public static void main(String[] args) {
47    }
```

Die Konstanten im oberen Teil des Gerüsts entsprechen den *Opcodes*, die Methode `String readProgramConsole()` erlaubt es, ein Programm von der Benutzerin einzulesen, welches diese auf der Konsole eingibt (die Eingabe wird durch zwei leere Zeilen beendet). Die Methode filtert dabei automatisch Kommentar-Zeilen. Gehen Sie nun wie folgend beschrieben vor.

1. Implementieren Sie zunächst die Methode `public static int[] parse(String textProgram)`, die ein Assembler-Programm in Textform in ein Array von Instruktionen überführt.
2. Implementieren Sie nun die Java-Methode `public static int pop()` und `public static void push(int value)`, die einen Wert vom Stack nehmen bzw. auf diesen legen. **Hinweis:** Sie müssen ggf. weitere globale Variablen hinzufügen.
3. Implementieren Sie die Methode `public static int execute(int[] program)`, die ein gegebenes Programm ab der ersten Instruktion ausführt. Die Methode gibt dasjenige Element zurück, welches nach der Ausführung ganz oben auf dem Stack zu finden ist.

Hinweis: Achten Sie auf negative Immediates. Ein negatives Immediate erkennen Sie daran, dass das 16. Bit gesetzt ist. In diesem Fall wird die 16-Bit-Zahl auf 32 Bits erweitert, indem man mit 1er-Bits auffüllt. Man nennt diesen Vorgang *sign extension*. Negative Immediates werden insbesondere für die Adressierung von Funktionsargumenten benötigt. Befassen Sie sich erst in diesem Zusammenhang mit negativen Immediates.

Hinweis: Gehen Sie bei dieser Aufgabe in Schritten vor. Erlauben Sie zunächst einfache Programme, die Additionen sowie I/O mit dem Benutzer erlauben. Erweitern Sie Ihre Implementierung anschließend um Sprünge. Im letzten Schritt fügen Sie dann die Unterstützung für lokale Variablen (hier benötigen Sie den Frame-Pointer) sowie Funktionsaufrufe hinzu. Lokale Variablen und Funktionsaufrufe sind anspruchsvoll.

Hinweis: Behandeln Sie Fehler in Ihrem Programm sinnvoll. Nutzen Sie dazu jeweils einen Aufruf der Methode `public static void error(String message)`, die das Programm mit einem Fehler abbricht.

Hinweis: Testen Sie Ihr Programm mit dem GGT-Programm aus den P-Aufgaben sowie folgendem rekursiven Programm zur Berechnung der Fakultät:

```

1  IN
2  LDI fak
3  CALL 1
4  OUT
5  HALT
6
7  // 1 Argument (n)
8  fak:
9  // Lokale variable (x)
10 ALLOC 1

```

```

11 // Wir setzen x = 1
12 LDI 1
13 STS 1
14 // Ist n == 1?
15 LDS 0
16 LDI 1
17 // Abbruchbedingung erfüllt?
18 JE end
19 // Rekursiver Aufruf mit n-1
20 LDI 1
21 LDS 0
22 SUB
23 LDI fak
24 CALL 1
25 // n*fak(n-1)
26 LDS 0
27 MUL
28 STS 1
29 end:
30 LDS 1
31 // Wir geben ein Argument und eine lokale Variable frei
32 RETURN 2

```