

Lösungshinweise/-vorschläge zum Übungsblatt 5: Software-Entwicklung 1 (WS 2017/18)

Die Hinweise und Vorschläge in diesem Dokument sollen der Lösungsfindung dienen und erheben demnach weder Anspruch auf Vollständigkeit noch Korrektheit. Sollten Sie Fehler finden, würden wir uns freuen, wenn Sie uns diese mitteilen. (Kontaktinformationen finden Sie auf unserer Webpräsenz.)

Aufgabe 1 Spezifikationen umsetzen (9 Punkte)

Implementieren Sie die folgenden Prozeduren gemäß ihrer Spezifikation. Verwenden Sie die JUnit Tests aus der Datei `IntersectionTests.java` um Ihre Implementierung zu testen.

Freiwillige Zusatzaufgabe: Implementieren Sie die Prozeduren `subset` und `intersection` so, dass sie mit möglichst wenig Berechnungsschritten auskommen.

```
import java.util.Arrays;

public class Intersection {
    /*
     * requires a != null
     * modifies \nothing
     * ensures \result == (es existiert ein int i in [0,a.length-1], so dass a[i] == x)
     */
    public static boolean contains(int[] a, int x) {
        for (int i = 0; i < a.length; i++) {
            if (a[i] == x) {
                return true;
            }
        }
        return false;
    }

    /*
     * requires a != null
     * modifies \nothing
     * ensures \result == (für alle int i in [0,a.length-2]: a[i] < a[i+1])
     */
    public static boolean increasing(int[] a) {
        for (int i = 0; i < a.length-1; i++) {
            if (a[i] >= a[i + 1]) {
                return false;
            }
        }
        return true;
    }

    /*
     * requires a != null
     *      && b != null
     *      && increasing(a)
     *      && increasing(b)
     * modifies \nothing
     */
}
```

```

ensures \result == (fuer alle int x: !contains(a, x) || contains(b,x))
*/
public static boolean subset(int[] a, int[] b) {
    int j=0;
    for (int i=0; i<a.length; i++) {
        while (j < b.length && a[i] != b[j]) {
            j = j + 1;
        }
        if (j >= b.length) {
            return false;
        }
    }
    return true;
}

/*
requires a != null
        && b != null
        && increasing(a)
        && increasing(b)
modifies \nothing
ensures \result != null
        && (fuer alle int x: (contains(a, x) && contains(b, x)) == contains(\result
, x))
        && increasing(\result)
*/
public static int[] intersection(int[] a, int[] b) {
    int minLength = Math.min(a.length, b.length);
    int[] res = new int[minLength];
    int i = 0;
    int j = 0;
    int k = 0;
    while (i < a.length && j < b.length) {
        if (a[i] < b[j]) {
            i = i + 1;
        } else if (a[i] > b[j]) {
            j = j + 1;
        } else { // a[i] == b[j]
            res[k] = a[i];
            i = i + 1;
            j = j + 1;
            k = k + 1;
        }
    }
    return Arrays.copyOf(res, k);
}
}

```

Aufgabe 2 Spezifizieren und Testen (12 Punkte)

Geben Sie für die folgenden Prozeduren jeweils eine sinnvolle Spezifikation mit Vorbedingungen, Nachbedingungen und gegebenenfalls einer Variablenliste für veränderten Zustand an. Achten Sie darauf, die Vorbedingung so zu wählen, dass keine Exceptions auftreten und die Prozedur terminiert, wenn die Vorbedingung eingehalten wird.

Schreiben Sie außerdem für jede der Prozeduren zwei oder mehr JUnit Testfälle.

Geben Sie eine Datei Procs.java ab, in der die Prozeduren mit den Spezifikationen als Kommentaren stehen und eine Datei ProcsTest.java in der sich die JUnit Tests befinden.

Hinweis: Im Abschnitt "Testen von Seiteneffekten" finden Sie ein Beispiel, das zeigt, wie Tests auf Implementierungen in anderen Dateien zugreifen können.

```
/*
requires ar != null && n >= 0 && n <= ar.length
modifies \nothing
ensures \result != null
      && \result.length == n
      && für alle int i in [0,n-1]: \result[i] == ar[i]
*/
public static int[] take(int[] ar, int n) { ... }

/*
requires input != null
modifies input
ensures für alle int i in [0,input.length-1]:
      input[i] == \old(input[input.length - 1 - i])
*/
public static void reverse(int[] input) { ... }

/*
requires snippet != null && snippet.length > 0 && len >= 0
modifies \nothing
ensures \result != null
      && \result.length == len
      && für alle int i in [0, len-1]:
          \result[i] = snippet[i % snippet.length]
*/
public static int[] repeat(int[] snippet, int len) { ... }
```

Testfälle:

```
import static org.junit.Assert.*;
import org.junit.Test;

public class ProcsTest {

    @Test
    public void take3() {
        int[] input = {1,2,3,4};
        int[] res = Procs.take(input, 3);
        int[] expected = {1,2,3};
        assertEquals(expected, res);
    }

    @Test
    public void takeNothingFromNothing() {
        // Randfall: Leeres Array
        int[] input = {};
    }
}
```

```

    int[] res = Procs.take(input, 0);
    int[] expected = {};
    assertEquals(expected, res);
}

@Test
public void takeAll() {
    // Randfall: Alle Elemente nehmen
    int[] input = {1,2,3,4};
    int[] res = Procs.take(input, 4);
    int[] expected = {1,2,3,4};
    assertEquals(expected, res);
}

// Für die reverse Funktion testen wir einmal mit einer geraden und
// einmal mit einer ungeraden Array-Länge. Da wir die Länge durch 2 Teilen
// sind diese Fälle leicht unterschiedlich.

@Test
public void reverseEven() {
    int[] input = {1,2,3,4};
    Procs.reverse(input);
    int[] expected = {4,3,2,1};
    assertEquals(expected, input);
}

@Test
public void reverseOdd() {
    int[] input = {1,2,3,4,5};
    Procs.reverse(input);
    int[] expected = {5,4,3,2,1};
    assertEquals(expected, input);
}

@Test
public void reverseEmpty() {
    // Randfall: Leeres Array
    int[] input = {};
    Procs.reverse(input);
    int[] expected = {};
    assertEquals(expected, input);
}

@Test
public void repeat1() {
    // Randfall: Kleinstes mögliches Eingabe-Array
    int[] snip = {1};
    int[] res = Procs.repeat(snip, 4);
    int[] expected = {1,1,1,1};
    assertEquals(expected, res);
}

@Test
public void repeat2() {
    // Fall, dass snippet genau 2 mal in das Ergebnis passt
    int[] snip = {1, 2};
    int[] res = Procs.repeat(snip, 4);
    int[] expected = {1,2,1,2};
    assertEquals(expected, res);
}

@Test
public void repeat3() {
    // Fall, dass snippet nur 4/3 mal in das Ergebnis passt
    int[] snip = {1, 2, 3};

```

```

    int[] res = Procs.repeat(snip, 4);
    int[] expected = {1,2,3,1};
    assertArrayEquals(expected, res);
}

@Test
public void repeatSmaller() {
    // Fall, dass snippet größer als das Ergebnis ist
    int[] snip = {1, 2, 3, 4};
    int[] res = Procs.repeat(snip, 3);
    int[] expected = {1,2,3};
    assertArrayEquals(expected, res);
}
}

```

Aufgabe 3 Rekursion und Terminierung (16 Punkte)

Geben Sie für die folgenden Prozeduren jeweils eine sinnvolle Vorbedingung an, so dass die Prozedur immer ohne Fehler terminiert, wenn die Vorbedingung erfüllt ist.

Beweisen Sie dann die Terminierung der Prozedur für alle erlaubten Werte. Verwenden Sie dazu das Verfahren aus der Vorlesung.

Hinweise: Sie können den Java Visualizer¹ verwenden, um das Verhalten der Prozeduren zu beobachten. Sollten Sie Probleme haben, eine passende Abstiegsfunktion zu finden, wenden Sie sich an Ihren Tutor.

a)

```

1   static void partition(int[] a, int pivot, int left, int right) {
2       if (left > right) {
3           return;
4       }
5       if (a[left] < pivot) {
6           partition(a, pivot, left+1, right);
7       } else if (a[right] >= pivot) {
8           partition(a, pivot, left, right-1);
9       } else {
10          int t = a[left];
11          a[left] = a[right];
12          a[right] = t;
13          partition(a, pivot, left+1, right-1);
14      }
15  }

```

1. Als Vorbedingung wählen wir

```

requires a != null
        && (left > right || (0 <= left < a.length && 0 <= right < a.length))

```

2. Der gültige Parameterbereich wird nicht verlassen:

- Beim rekursiven Aufruf in Zeile 6 gilt wegen der if-Bedingung in Zeile 2 entweder:
left < right: Dann gilt left+1 <= right < a.length
left == right: Dann gilt left+1 > right
Die Parameter bleiben also im gültigen Bereich.
- Beim rekursiven Aufruf in Zeile 8 gilt entweder right-1 >= 0 oder left > right-1, also

¹Online unter http://cscircles.cemc.uwaterloo.ca/java_visualize/ oder lokal mit Docker
(Befehl `docker run --rm --name java_visualize --privileged -p 80:80 mweber/java_visualize:v1`)

bleiben die Parameter im gültigen Bereich.

- Beim rekursiven Aufruf in Zeile 13 werden `left` und `right` wie in den vorherigen Fällen verändert.

3. Als Abstiegsfunktion wählen wir $h(a, p, l, r) = \max(0, 2 + r - l)$

4. Für die rekursiven Aufrufe gilt dann:

- Aufruf in Zeile 6:

```
h(a, p, l+1, r)
= max(0, 2 + r - (l+1))
= max(0, 1 + r - l)
// l <= r gilt wegen Bedingung in Zeile 2
= 1 + r - l
< 2 + r - l
= h(a, p, l, r)
```

- Aufruf in Zeile 8:

```
h(a, p, l, r-1)
= max(0, 2 + r-1 - l)
= max(0, 1 + r - l)
// l <= r gilt wegen Bedingung in Zeile 2
= 1 + r - l
< 2 + r - l
= h(a, p, l, r)
```

- Aufruf in Zeile 13:

```
h(a, p, l+1, r)
= max(0, 2 + r-1 - (l+1))
= max(0, r - l)
// l <= r gilt wegen Bedingung in Zeile 2
= r - l
< 2 + r - l
= h(a, p, l, r)
```

b)

```
1 static int sum(int[][] a, int i, int j, int n, int m) {
2     if (j >= m || i > n) {
3         return 0;
4     } else if (i == n) {
5         return sum(a, 0, j+1, n, m);
6     } else {
7         return a[i][j] + sum(a, i+1, j, n, m);
8     }
9 }
```

1. Als Vorbedingung wählen wir:

```
requires a != null
    && a.length >= n
    && m >= 0
    && (für alle i in [0,n]: a[i] != null && a[i].length >= m)
    && i >= 0
    && j >= 0
```

2. Die Vorbedingung wird für rekursive Aufrufe erfüllt:

- In Zeile 5 wird i und j verändert: Für $i' = 0$ und $j' = j+1$ gilt offensichtlich die Vorbedingung $i' = 0 \geq 0$ und $j' = j+1 \geq 0$.
- Zeile 7 gilt auch wieder $i' = i+1 \geq 0$ und $j' = j \geq 0$

3. Als Abstiegsfunktion wählen wir:

$h(a, i, j, n, m) = (|m - j|, |n - i|)$ und die noethersche Ordnung $(\mathbb{N} \times \mathbb{N}, \leq_{lex})$.

4. Für die rekursiven Aufrufe gilt dann:

- Aufruf in Zeile 5: zu zeigen ist $h(a, 0, j + 1, n, m) < h(a, i, j, n, m)$

$$h(a, 0, j + 1, n, m) = (|m - (j + 1)|, |n - 0|)$$

Wegen der if-Bedingungen gilt $j < m$ und $i \leq n$, daher können die Betragszeichen weggelassen werden:

$$\dots = (m - j - 1, n) < (m - j, n - i) = (|m - j|, |n - i|) = h(a, i, j, n, m)$$

- Aufruf in Zeile 7: zu zeigen ist $h(a, i + 1, j, n, m) < h(a, i, j, n, m)$

$$h(a, i + 1, j, n, m) = (|m - j|, |n - (i + 1)|)$$

Wegen den beiden if-Bedingungen gilt hier $i < n$, daher können die Betragszeichen weggelassen werden:

$$\dots = (|m - j|, n - i - 1) < (|m - j|, n - i) = (|m - j|, |n - i|) = h(a, i, j, n, m)$$

Aufgabe 4 Prozedurabstraktion (11 Punkte)

Laden Sie sich das unten abgedruckte Programm `Histograms.java` herunter.

Das Programm nimmt eine Anzahl von Segmenten als Programm-Parameter und liest dann eine Tabelle mit Bezeichnern und `int`-Werten über die Standard-Eingabe ein. Das Programm berechnet dann die Verteilung der Werte und gibt ein Histogramm auf der Konsole aus. Als Beispiel können Sie die Datei `klausur.txt` herunterladen und das Programm wie folgt aufrufen:

```
java Histograms 20 < klausur.txt
```

Das Programm besteht aus einer einzelnen `main`-Prozedur. Daher ist es schwer JUnit Tests für das Programm zu schreiben.

- a) Teilen Sie das Programm in mehrere kleinere Prozeduren auf. Beachten Sie dabei die Hinweise zur Abstraktion durch Prozeduren aus der Vorlesung. Geben Sie für diese Teilaufgabe die verbesserte Datei `Histograms.java` ab.

```
public class Histograms {

    public static void main(String[] args) {
        if (args.length == 0) {
            System.out.println("Geben Sie die Anzahl der Segmente im "
                + "Histogramm als Programmparameter an.");
        }
        int sections = Integer.parseInt(args[0]);
        String[] input = ReadStrings.readStrings();
        int[] values = extractValues(input);

        int min = min(values);
        int max = max(values);

        int[] countPerSection = countPerSection(values, min, max, sections);

        int[] normalizedCounts = normalize(countPerSection, 50);
    }
}
```

```

    printBars(normalizedCounts);
}

/*
requires barLengths != null
modifies System.out
ensures Schreibt für jede Zahl n in barLengths eine Zeile mit n '|' -Zeichen
*/
public static void printBars(int[] barLengths) {
    for (int i = 0; i < barLengths.length; i++) {
        int n = barLengths[i];
        for (int j = 0; j < n; j++) {
            System.out.print("|");
        }
        System.out.println();
    }
}

/*
requires values != null
    && es gibt einen Eintrag mit Wert > 0 in values
    && newMax >= 0
modifies \nothing
ensures \result != null
    && \result.length == values.length
    && max(\result) == newMax
    && für alle int i in [0, values.length-1]:
        \result[i] == values[i]*newMax/max(values)
*/
public static int[] normalize(int[] values, int newMax) {
    int[] res = new int[values.length];
    int max = max(values);
    for (int i = 0; i < values.length; i++) {
        res[i] = values[i] * newMax / max;
    }
    return res;
}

/*
requires values != null
    && für alle int x in values: min <= x <= max
    && sections > 0
modifies \nothing
ensures \result != null
    && \result.length == sections
    && für alle int i in [0, sections-1]:
        \result[i] == (Anzahl der Einträge in values mit Wert x, so dass
            x >= min + i*(1+max-min)/sections
            und x < min + (i+1)*(1+max-min)/sections)
*/
public static int[] countPerSection(int[] values, int min, int max, int
sections) {
    // Die Anzahl der Werte für jedes Segment:
    int[] counts = new int[sections];
    // Abstand zwischen min und max:
    int range = (max - min);
    for (int i = 0; i < values.length; i++) {
        int section = (values[i] - min) * sections / (range+1);
        counts[section] = counts[section] + 1;
    }
    return counts;
}

/*

```



```

    requires values != null && values.length > 0
    modifies \nothing
    ensures \result ist die kleinste Zahl in values
    */
    public static int min(int[] values) {
        int res = values[0];
        for (int i = 1; i < values.length; i++) {
            if (values[i] < res) {
                res = values[i];
            }
        }
        return res;
    }

    /*
    requires values != null && values.length > 0
    modifies \nothing
    ensures \result ist die größte Zahl in values
    */
    public static int max(int[] values) {
        int res = values[0];
        for (int i = 1; i < values.length; i++) {
            if (values[i] > res) {
                res = values[i];
            }
        }
        return res;
    }

    /*
    requires input != null
        && für alle i in [0, input.length / 2]:
            input[i] ist ein String, der einen int-Wert repräsentiert
    modifies \nothing
    ensures \result != null
        && \result.length == input.length / 2
        && für alle i in [0, input.length / 2]:
            \result[i] == (input[i] umgewandelt zu int)
    */
    public static int[] extractValues(String[] input) {
        int[] result = new int[input.length / 2];
        for (int i = 0; i < result.length; i++) {
            result[i] = Integer.parseInt(input[i * 2 + 1]);
        }
        return result;
    }
}

```

- b) Schreiben Sie mindestens 2 JUnit-Testfälle für jede Prozedur aus Teil a), welche keinen Effekt auf System.out hat. Schreiben Sie insgesamt mindestens 10 Testfälle. Geben Sie für die Tests eine zweite Datei namens HistogramsTest.java ab.

```

import static org.junit.Assert.*;
import org.junit.Test;

public class HistogramsTest {

    @Test
    public void testNormalizeOneEntry() {
        int[] input = {1};
        int[] res = Histograms.normalize(input, 5);
    }
}

```

```

        int[] expected = {5};
        assertEquals(expected, res);
    }

    @Test
    public void testNormalize() {
        int[] input = {10, 20, 100, 50};
        int[] res = Histograms.normalize(input, 10);
        int[] expected = {1, 2, 10, 5};
        assertEquals(expected, res);
    }

    @Test
    public void testCountsPerSectionOneSection() {
        int[] input = {0,1,2,3,4};
        int[] res = Histograms.countPerSection(input, 0, 4, 1);
        int[] expected = {5};
        assertEquals(expected, res);
    }

    @Test
    public void testCountsPerSection3Sections() {
        int[] input = {1,2,3,4,5,6,7,8,9};
        int[] res = Histograms.countPerSection(input, 1, 9, 3);
        int[] expected = {3,3,3};
        assertEquals(expected, res);
    }

    @Test
    public void testCountsPerSection3Sections2() {
        int[] input = {1,4,5,6,7,8};
        int[] res = Histograms.countPerSection(input, 1, 9, 3);
        int[] expected = {1,3,2};
        assertEquals(expected, res);
    }

    @Test
    public void testMin1() {
        int[] input = {7};
        int res = Histograms.min(input);
        assertEquals(7, res);
    }

    @Test
    public void testMin2() {
        int[] input = {7,-4,9};
        int res = Histograms.min(input);
        assertEquals(-4, res);
    }

    @Test
    public void testMax1() {
        int[] input = {7};
        int res = Histograms.max(input);
        assertEquals(7, res);
    }

    @Test
    public void testMax2() {
        int[] input = {7,-4,9};
        int res = Histograms.max(input);
        assertEquals(9, res);
    }
}

```

```

@Test
public void testExtractValuesEmpty() {
    String[] input = {"a"};
    int[] res = Histograms.extractValues(input);
    int[] expected = {};
    assertEquals(expected, res);
}

@Test
public void testExtractValuesEven() {
    String[] input = {"a", "1", "b", "2"};
    int[] res = Histograms.extractValues(input);
    int[] expected = {1, 2};
    assertEquals(expected, res);
}

@Test
public void testExtractValuesOdd() {
    String[] input = {"a", "1", "b", "2", "c", "3", "d"};
    int[] res = Histograms.extractValues(input);
    int[] expected = {1, 2, 3};
    assertEquals(expected, res);
}
}

```

Freiwillige Zusatzaufgabe (0 Punkte)

Die folgende Aufgabe können Sie selbstständig bearbeiten und abgeben. Dazu finden Sie im Exclaim eine zusätzliche Übung “SE1WS17Z”, wo Sie Ihre Lösung als Programm `Superpowers.java` hochladen können. Die Abgaben werden automatisch getestet, aber nicht bewertet.

Bei der Lösung der Aufgaben kommt es auch auf die Effizienz des Algorithmus an. Mit einer langsamen Lösung werden Sie wahrscheinlich für große Eingaben einen Timeout erhalten.

Diese Aufgabe stammt von Christian Müller und wurde für das “TUM ICPC Training Camp 2016” entworfen und steht unter der Lizenz “Creative Commons Attribution-ShareAlike”.



Superpowers

Science has done it! We unlocked the secret of superpowers! In fact, it is rather easy. Using the patented DNA-Resequencer, we can rewrite the DNA of one lucky person so that he will develop either super strength or the ability to telepathically talk to tomatoes (depending on prior genetic makeup).

To find out which one will happen, the scientists have to count the number of times a specific sequence (which differs for each person) occurs in the subjects DNA. Can you help them with that?

Input

The first line of the input contains an integer t . t test cases follow.

Each test case consists of two lines. The first line contains w , the DNA of a potential subject. The second line contains p , the specific sequence that influences the superpower.

Output

For each test case, print a line containing “Case # i : x ” where i is its number, starting at 1 and x is the amount of times p occurs in w . Each line of the output should end with a line break.

Constraints

- $1 \leq t \leq 20$
- $1 \leq |p| \leq |w| \leq 50000$
- Both p and w consist only of the letters “A”, “C”, “G” and “T”