Karlsruher Institut für Technologie

Programmieren WS 2012/2013

Forschungsgruppe: Verifikation trifft Algorithmik
http://verialg.iti.uka.de/prog_ws12.php
Dr. Carsten Sinz, Florian Merz, Markus Iser

Übungsblatt 3
(v1.0)

Ausgabe:	19.11.2012
Abgabe:	03.12.2012, 13Uhr
Besprechung:	10.12.2011 - 14.12.2011

Allgemeine Hinweise

In diesem Übungsblatt werden Sie einige Aufgaben bearbeiten müssen, bei denen Sie nicht programmieren müssen. Geben Sie Ihre Antwort auf diese Aufgaben jeweils in separaten Textdateien ab.

In Lösungen zu dieser Übungsaufgabe dürfen lediglich Klassen und Methoden aus dem Paket `lava.lang` benutzt werden, also solche die ohne `import`-Anweisung benutzt werden können.

Die Benutzung von `System.out.println()` oder anderen Methoden um Text auszugeben oder Daten in Dateien zu schreiben ist in finalen Abgaben untersagt.

A Wiederholen und Vertiefen

A.1 Operatoren und Präzedenz

A.1.1 Anweisungen und ihre Seiteneffekte

Geben Sie für jede Zeile in Tabelle 1 jeweils den Wert der Variablen a und b nach Ausführungen der Anweisung(en) in der jeweiligen Zeile an. Nehmen Sie die Tabelle dabei nicht als fortlaufenden Programm an, sondern betrachten Sie *jede Zeile für sich*. Gehen Sie davon aus, dass vor Ausführung der jeweiligen Anweisung `a` den Wert 5 und `b` den Wert 23 hat. Erklären Sie bei den mit * gekennzeichneten Zeilen, wie Sie auf das Ergebnis kommen.

Geben Sie Ihre Lösung in einer CSV-Datei[1] ab. Benutzen Sie als Vorlage hierfür zwingend die von uns auf `http://baldur.iti.kit.edu/programmieren/` bereitgestellte Datei `statements.csv`.

A.2 While-Programme

Nach einer wiedermal viel zu langen, durchzockten Nacht hat Pierre Poivre im Eifer des Gefechts versehentlich seine Dose Red Bull über seine Tastatur geleert. Das Getränk hat dabei, trotz sofortiger, gründlichster Reinigung der Tastatur, die Tasten `d` und `f` dauerhaft unbrauchbar gemacht. Da zu dieser späten Stunde der Elektronikfachmarkt seines Vertrauens nicht mehr geöffnet hat, muss er nun seine Übungsaufgabe Programmieren ohne diese Tasten lösen. Entsetzt stellt er dabei fest, dass er ohne f-Taste das Schlüsselwort `if` nicht tippen kann. Nach einem kurzen Schrecken, erinnert er sich

[1] `http://de.wikipedia.org/wiki/CSV_Dateiformat`

Tabelle 1: Anweisungen und ihre Seiteneffekte

Anweisung	a = 5	b = 23	
`b = a;`	5	5	
`a = a + 1;`	6	23	
`++a;`			
`b = a + 1;`			
`b = a + a;`			
`b = ++a;`			
`b = a++;`			
`a = a++;`			
`b = a + ++a;`			*
`b = a + a++;`			*
`b = ++a + ++a;`			*
`b = a++ + ++a;`			*
`b = a = 1;`			*
`b += a += 1;`			*

jedoch daran einmal gehört zu haben, dass man jedes beliebige Programm nur mit der `while`-Schleife als einziger Kontrollstruktur darstellen kann. Schreiben Sie für ihn die Klasse aus Abbildung 1 so um, dass sie kein `d` und `f` mehr enthält. Der Code steht auch auf `http://baldur.iti.kit.edu/programmieren` zum Download zur Verfügung. Ersetzen Sie dabei alle `if-then`, `if-then-else`, `for` und `do-while` Konstrukte durch semantisch Äquivalente `while` Konstrukte.

Beachten Sie, dass der Praktomat überprüfen wird, ob Ihre Abgabe die Buchstaben `d` und `f` enthält und dies auch durch Punktabzug bestraft wird. Aus technischen Gründen wird dieser Test jedoch immer als `passed` angezeigt, auch wenn die Buchstaben `d` und `f` in der Abgabe vorkommen. Öffnen Sie daher immer die Detailbeschreibung des Tests um sicher zu gehen, dass Ihre Abgabe den Anforderungen entspricht.

A.3 Levenshtein-Distanz

Die Levenshtein-Distanz ist eine Metrik für Zeichenketten. Man benutzt sie als Maß, das angibt, wie sehr sich zwei Zeichenketten unterscheiden. Sie sollen im folgenden eine Variante des Levenshtein Algorithmus implementieren. Lesen Sie sich die Beschreibung des Algorithmus vorher genau durch.

Hintergrundwissen

Die Levenshtein-Distanz ist benannt nach ihrem Erfinder dem russischen Mathematiker Wladimir Iossifowitsch Lewenstein, der für seine Arbeit an fehlerkorrigierenden Codes (einschließlich der Levenshtein Distanz) im Jahr 2006 die Richard W. Hamming Medaille erhielt. Die Levenshtein-Distanz wird oft auch als Editierdistanz bezeichnet, weil sie die minimale Anzahl von elementaren Zeichenoperationen (Einfügen, Löschen, Ersetzen) angibt, die nötig sind um ein Wort in ein anderes zu überführen. Einsatzgebiete sind beispielsweise automatische Korrekturvorschläge in Textverarbeitungssystemen oder aber auch die automatische Duplikat-Erkennung. Die Levenshtein-Distanz kann auch als Spezialfall des sogenannten Dynamic Time Warping betrachtet werden. Das ist ein Algorithmus der in der Anthropomatik bei Sprach- und Gestenerkennung von Bedeutung ist.

```
class StrangeClass {
    // 'on't 'are 'oing this at home!
    private boolean evilBoolean = !true;
    private int evilInt = 0;

    public StrangeClass(boolean someBoolean, int someInt) {
        this.evilBoolean = someBoolean;
        this.evilInt = someInt;
    }

    public int lessEvil() {
        for (int i = 0; i < evilInt; i++) {
            evilInt -= i;
        }

        return evilInt;
    }

    public int strangeEvilness() {
        do {
            evilInt *= 3;
        } while (evilInt > 0 && evilInt < 999);

        return evilInt;
    }

    public int moreEvil() {
        if (evilBoolean) {
            evilInt++;
        }

        return evilInt;
    }

    public int changeEvilness() {
        if (evilBoolean) {
            evilInt++;
            evilBoolean = !evilBoolean;
        } else {
            evilInt--;
        }

        return evilInt;
    }

}
```

Abbildung 1: Listing für Aufgabe A.2

Wir beginnen mit ein paar wenigen Definitionen, von denen die meisten intuitiv verständlich sind. Diese Definitionen benutzen wir, um die Funktionsweise des Levenshtein-Algorithmus genau zu beschreiben. Im Folgenden soll die Schreibweise $w[i]$ das i-te Zeichen im Wort w bezeichnen. Dabei legen wir fest, dass $w[0] = \epsilon$, wobei mit ϵ im Allgemeinen das leere Wort ("nichts") gemeint ist. Des weiteren definieren wir, dass die Länge eines Wortes w mit $|w|$ bezeichnet werden soll. Ein Präfix der Länge n von w bezeichnen wir mit $\pi(w, n)$. Auch hier sei $\pi(w, 0) = \epsilon$.

Ein Beispiel: Nehmen wir für w die Zeichenkette "Informatik". Dann ist $w[0] = \epsilon, w[1] = \mathrm{I}, w[2] = \mathrm{n}, w[3] = \mathrm{f}, \ldots$ und so weiter. Des weiteren ist die Länge der Zeichenkette $|w| = 10$. Dann noch zwei Beispiele für Präfixe: $\pi(w, 3) = \mathrm{Inf}$ und $\pi(w, 5) = \mathrm{Infor}$.

Um die Distanz zweier Wörter w_1 und w_2 zu berechnen, erzeugt der Levenshtein-Algorithmus eine Matrix $M^{|w_2|+1 \times |w_1|+1}$, die **an jeder Stelle** $m(i, j)$ die Editierdistanz der Präfixe $\pi(w_2, i)$ und $\pi(w_1, j)$ enthält. Dabei kann die Editierdistanz für zwei Präfixe an jeder Stelle aus den Editierdistanzen der jeweils kürzeren Präfixe berechnet werden (Beachten Sie, dass im Falle von Matrizen der Zeilenindex immer vor dem Spaltenindex genannt wird).

Hintergrundwissen

Der Levenshtein-Algorithmus ist ein Beispiel für sogenannte dynamische Programmierung. Dynamische Programmierung kann man überall dort einsetzen, wo sich die Lösung für ein Optimierungsproblem aus den optimalen Lösungen für dessen Teilprobleme zusammensetzen lässt. Man beginnt dann mit der direkten Lösung eines elementaren Teilproblems und benutzt diese Ergebnisse für die Lösung des nächst größeren Problems. Beim Levenshtein-Algorithmus ergibt sich die Editierdistanz zweier Wörter aus den Editierdistanzen ihrer Präfixe. Begonnen wird mit der Editierdistanz der leeren Präfixe $\pi(w_2, 0)$ und $\pi(w_1, 0)$ also ϵ.

Um die minimale Anzahl an Elementaroperationen zu berechnen die nötig ist um w_1 in w_2 zu überführen, definieren wir zunächst für jede der drei Elementaroperationen Einfügen, Löschen und Ersetzen eine Kostenfunktionen. Für $(i, j) \in \{0, 1, \ldots, |w_2|\} \times \{0, 1, \ldots, |w_1|\}$ seien insert, delete und replace wie folgt definiert:

	ϵ	n	ä	h	m	l	i	c	h
ϵ	0	1	2	3	4	5	6	7	8
d	1	1	2	3	4	5	6	7	8
ä	2	2	1	2	3	4	5	6	7
m	3	3	2	2	2	3	4	5	6
l	4	4	3	3	3	2	3	4	5
i	5	5	4	4	4	3	2	3	4
c	6	6	5	5	5	4	3	2	3
h	7	7	6	5	6	5	4	3	2

delete

insert

Abbildung 2: Beispiel für $w_1 =$ nähmlich und $w_2 =$ dämlich

$$\text{delete}(i,j) = \begin{cases} \infty, & \text{falls } j < 1 \\ m(i,j-1)+1, & \text{sonst} \end{cases}$$

$$\text{insert}(i,j) = \begin{cases} \infty, & \text{falls } i < 1 \\ m(i-1,j)+1, & \text{sonst} \end{cases}$$

$$\text{replace}(i,j) = \begin{cases} \infty, & \text{falls } i < 1 \text{ oder } j < 1 \\ m(i-1,j-1), & \text{falls } i \geq 1 \text{ und } j \geq 1 \text{ und } w_1[j] = w_2[i] \\ m(i-1,j-1)+1, & \text{sonst} \end{cases}$$

Nach einem Initialisierungsschritt in dem $m(0,0) = 0$ gesetzt wird, werden alle weiteren Einträge zeilenweise aus den vorherigen Einträgen berechnet, indem für $m(i,j)$ und $i,j \geq 1$ jeweils das Minimum der Kostenfunktionen gewählt wird:

$$m(i,j) = \min\left\{ \text{delete}(i,j), \text{insert}(i,j), \text{replace}(i,j) \right\}$$

Beachten Sie, dass oben zugunsten einer schlanken Definition ∞ im Wertebereich der Funktionen vorkommt. Benutzen Sie in Ihrem Code stattdessen `Integer.MAX_VALUE`.

In Abbildung 2 sehen Sie eine voll ausgefüllte Levenshtein-Matrix für die beiden Zeichenketten "nähmlich" und "dämlich". Die Levenshtein-Distanz ist in diesem Fall 2. Bevor Sie dazu übergehen den Algorithmus zu implementieren, stellen Sie sicher, dass sie das Beispiel verstehen, indem Sie die Initialisierungsschritte und die Berechnung der einzelnen Werte nachvollziehen.

Schreiben Sie jetzt einen modifizierten Levenshtein-Algorithmus, indem Sie den Code aus Abbildung 3 ergänzen. Die Methode `getDistance()` soll dabei die Levenshtein Distanz der im Konstruktor übergebenen Worte `word1` und `word2` zurückgeben. Die Methoden `delete`, `insert` und `replace` sollen dabei den oben beschrieben Funtionen entsprechen.

Die Rahmenklasse aus Abbildung 3 steht auch auf `http://baldur.iti.kit.edu/programmieren` zum Download zur Verfügung.

Modifizieren Sie den Algorithmus so, dass Einfügen und Auslassen des Buchstaben "h" nach Vokalen ("a", "e", "i", "o", "u") und nach Umlauten ("ä", "ö", "ü") nichts kostet. Machen Sie mithilfe von Kommentaren alle diejenigen Stellen kenntlich, durch die sich Ihr Algorithmus vom klassischen (unmodifizierten) Levenshtein-Algorithmus unterscheidet. Geben Sie nur den modifizierten Algorithmus im Praktomaten ab. Beachten Sie auch, dass Sie, um die Praktomat-Tests zu bestehen, zunächst Teilaufgabe B.2 lösen müssen.

A.4 Geometrie

In dieser Aufgabe sollen sie einige geometrische Funktionen und Algorithmen in `Java`-Code umsetzen[2].

Hierfür müssen Sie zunächst die Klasse `Point` implementieren. Einen Rahmen hierfür haben wir in Abbildung 4 dargestellt. Die Methode `Point.distance(Point point)`

[2]Da die Funktionalität aller hier zu programmierenden Klassen automatisch getestet wird, sollten Sie besonders darauf achten Klassen und Methoden nicht umzubenennen, sowie Anzahl und Typ der Argumente nicht zu verändern. Darüberhinaus sollten Sie darauf achten, dass Klassen und Methoden für die auf `http://baldur.iti.kit.edu/programmieren/` verfügbare SVG-Ausgabeklasse sichtbar sind.

```
class Levenshtein {
    Levenshtein(String word1, String word2) {
    }

    int getDistance() {
    }

    int delete(int i, int j) {
    }

    int insert(int i, int j) {
    }

    int replace(int i, int j) {
    }
}
```

Abbildung 3: Listing für Aufgabe A.3

Abbildung 4: Listing für Aufgabe A.4: Die Klasse Point

```
class Point {
    Point(double x, double y) {
    }

    double getX() {
    }

    double getY() {
    }

    double distance() {
    }

    double manhattanDistance() {
    }

    Point mirror(Point point) {
    }

    Point mirror(Line line) {
    }
}
```

Abbildung 5: Listing für Aufgabe A.4: Die Klasse Line

```
class Line {
    Line(Point start, Point end) {
    }

    Point getStartPoint() {
    }

    Point getEndPoint() {
    }

    Point interpolate(double t) {
    }

    Line mirror(Point point) {
    }

    Line mirror(Line line) {
    }
}
```

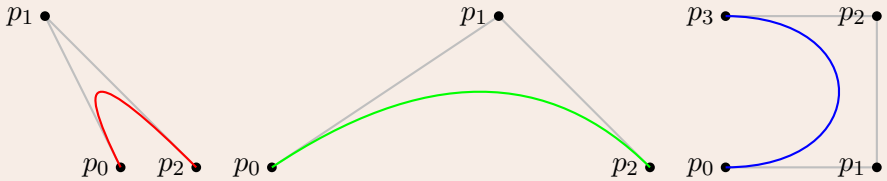
soll dabei den Abstand dieses Punkts zum Punkt `point` zurückgeben. Die Methode `Point.manhattanDistance(Point point)` den Abstand entlang der Gitterlinien des Koordinatensystems. Die Methode `Point.mirrorAt(Point point)` soll eine Punktspiegelung umsetzen. Sie gibt ein neues `Point` Objekt zurück, dessen Position der Spiegelung des `this`-Objekts an dem Argument `point` entspricht. Analog hierzu setzt die Methode `Point.mirrorAt(Line line)` die Spiegelung an einer Geraden um.

Die Klasse `Point` können Sie dann verwenden um eine Klasse `Line` zu implementieren. Orientieren Sie sich hierbei am Programmcode aus Abbildung 5. Die `Line.mirrorAt`-Methoden sollen analog zu denen der Klasse `Point` arbeiten. Die Methode `interpolate` müssen Sie erst in Teilaufgabe A.4.1 implementieren.

Aufbauend auf diesen beiden Klassen sollen Sie nun auch De Casteljaus Algorithmus in `Java` realisieren. Dabei handelt es sich um einen einfachen, geometrisch anschaulichen Algorithmus zur Berechnung von Punkten auf einer Bezier-Kurven, den wir in Abschnitt A.4.2 für quadratische Bezierkurven kurz beschreiben.

Hintergrundwissen

Bezierkurven wurden in den 60er Jahren unabhängig voneinander von Pierre Bézier und Paul de Casteljau entwickelt um die Form von Autokarosserien mathematisch zu beschreiben. Sie werden heutzutage in vielen Bereichen der Informatik eingesetzt, beispielsweise um glatte, geschwungene Oberflächen zu beschreiben, für Animationen, sowie in der Typographie für Schriftumrisse. Sie zeichnen sich durch ihre einfache und intuitive Handhabung aus. Diese beruht auf einer geringen Anzahl Kontrollpunkten, die den Kurvenverlauf definieren und deren Einfluss auf Kurvenverlauf meist intuitiv erfasst werden kann.



Bevor Sie jedoch mit der Implementierung des DeCasteljau Algorithmus beginnen können, müssen Sie zunächst einen Algorithmus zur linearen Interpolation zweier Punkte berechnen.

A.4.1 Lineare Interpolation

Implementieren Sie in der Klasse `Line` die Methode `interpolate()`, die einen Paramter `t` vom Typ `double` nimmt und einen Punkt zurückgibt, der auf der Linie, die durch die beiden Punkte definiert wird, liegt. Der Parameter `t` bestimmt dabei wie weit der resultierende Punkt vom Startpunkt oder und vom Endpunkt entfernt auf der Linie liegt. Ein t von 0 heißt, dass das Ergebnis direkt auf dem Startpunkt liegt, ein t von 1 bedeutet, dass das Ergebnis direkt beim Endpunkt liegt. Ein t mit $0 < t < 1$ liegt zwischen Start- und Endpunkt.

Für den Rückgabewert r der Methode soll gelten:

$$r = (x_r, y_r)$$
$$x_r = (1-t)x_{p_0} + tx_{p_1}$$
$$y_r = (1-t)y_{p_0} + ty_{p_1}$$

wobei $p_0 = (x_{p_0}, y_{p_0})$ der Startpunkt der Linie ist und $p_1 = (x_{p_1}, y_{p_1})$ der Endpunkt der Linie ist.

Nicht bewertete Zusatzfrage

Was passiert bei $t < 0$?
Was passiert bei $t > 1$?

A.4.2 Quadratische Bezierkurven

Fügen Sie nun Ihrer Implementierung die Klasse `QuadraticBezier` hinzu. Implementieren Sie hierfür die `mirrorAt`-Methoden analog zu `Point` und `Line`.

Implementieren Sie die Methode `interpolate`, mit deren Hilfe Punkte berechnet werden können, die auf der quadratischen Bezierkurven liegen, die durch die Punkt p_0 (`start`), p_1 (`control`) und p_2 (`end`) definiert ist, liegen.

Implementieren Sie `interpolate` indem Sie zunächst die Punkte q_0 und q_1 berechnen. Hierbei soll q_0 das Ergebnis der linearen Interpolation von p_0 und p_1 sein und q_1 das Ergebnis der linearen Interpo-

Abbildung 6: Listing für Aufgabe A.4.2: Die Klasse QuadraticBezier

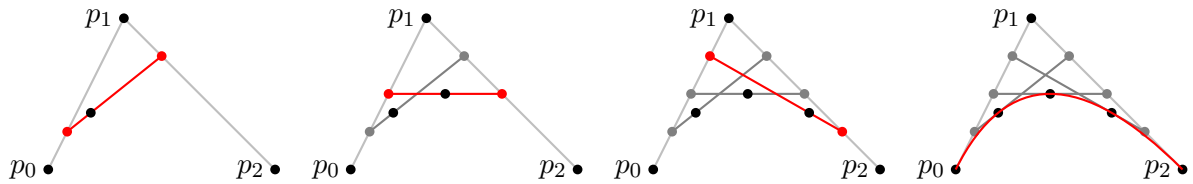
```
class QuadraticBezier {
    QuadraticBezier(Point start, Point control, Point end) {
    }

    Point interpolate(double t) {
    }

    QuadraticBezier mirror(Point point) {
    }

    QuadraticBezier mirror(Line line) {
    }
}
```

Abbildung 7: Quadratische Bezierkurven (v.l.n.r.: $t = 0.25$, $t = 0.5$, $t = 0.75$ und die Bezierkurve durch diese Punkte)



lation von p_1 und p_2. Benutzen Sie jeweils das Argument t als Interpolationsparameter. Interpolieren Sie nun wieder entlang der Linie durch q_0 und q_1 mit dem Faktor t und geben Sie diesen Wert zurück. Abbildung 7 illustriert diesen Algorithmus graphisch für $t = 0.25$, $t = 0.5$ und $t = 0.75$.

? Nicht bewertete Zusatzfrage

Was passiert jetzt bei $t < 0$?
Was passiert bei $t > 1$?

A.4.3 Kubische Bezierkurven

Implementieren Sie schließlich auch noch die Klasse `CubicBezier` aus Abbildung 8, die kubische Bezierkurven berechnen kann. Gehen Sie dabei analog zu Teilaufgabe A.4.2 vor.

A.4.4 Abgabe

Beachten Sie, dass Sie wie auch bei Teilaufgabe A.3, um die Praktomat-Tests zu bestehen, zunächst Teilaufgabe B.2 lösen müssen.

B Datenkapselung, Pakete, JavaDoc

Sie haben in den Tutorien gelernt wie man Kommentare im JavaDoc Stil schreibt. Desweiteren haben Sie in der Vorlesung vom `package` Konzept gehört und etwas über Datenkapselung gelernt. Sie sollen in diesem Teil des Blattes Ihr neues Wissen auf Teilaufgaben des letzten Teils anwenden. In diesem Blatt bekommen Sie explizit Punkte hierfür. In den folgenden Übungsblättern wird es Punktabzüge

Abbildung 8: Listing für Aufgabe A.4.3: Die Klasse CubicBezier

```
class CubicBezier {
    CubicBezier(Point start, Point control1, Point control2, Point end) {
    }

    Point interpolate(double t) {
    }

    CubicBezier mirror(Point point) {
    }

    CubicBezier mirror(Line line) {
    }
}
```

für fehlende JavaDac Kommentare und fehlende oder fehlerhafte Datenkapselung geben.

B.1 Datenkapselung

Versehen sie jede Klasse, jedes Attribut und jede Methode aus Aufgabe A.3 und aus Aufgabe A.4 mit dem jeweils sinnvollsten Zugriffsmodifikator (`private`, `protected` oder `public` oder keiner). Erklären Sie jeweils in einem Kommentar, warum Sie sich für diesen Modifikator entschieden haben, beziehungsweise warum Sie sich gegen einen Modifikator entschieden haben.

B.2 Pakete

Erstellen Sie ein Paket `levenshtein` und fügen Sie Ihre Implementierung des Levenshtein Algorithmus aus Aufgabe A.3 diesem Paket hinzu.

Erstellen Sie ein Paket `geometry` und fügen Sie Ihre Implementierung aus Aufgabe A.4 diesem Paket hinzu.

Beachten Sie, dass Sie Pakete immer als Ganzes abgeben müssen, d.h. Sie können die Dateien des Pakets nicht einzeln hochladen, sondern müssen eine zip-Datei erstellen die sowohl den Paket-Ordner, als auch sämtliche Dateien das Pakets enthält und diese dann in den Praktomaten hochladen. Der Name des Ordners in der zip-Datei muss dabei dem Namen des Pakets entsprechen. Einzeln hochgeladen Dateien werden vom Praktomaten immer im Default-Paket abgelegt und daher möglichweise von automatisierten Tests nicht gefunden.

B.3 JavaDoc

Annotieren Sie alle Klassen aus Aufgabe A.3 und Aufgabe A.4 vollständig mit JavaDoc Kommentaren. Dies schließt auch Methoden, Klassen und Attribute ein, die sie von uns ohne JavaDoc Kommentare bekommen haben.