Karlsruher Institut für Technologie

Prof. Dr.-Ing. Anne Koziolek
Architecture-driven Requirements Engineering (ARE)
https://sdqweb.ipd.kit.edu/wiki/Programmieren
programmieren-vorlesung@ipd.kit.edu

Programmieren – Wintersemester 2020/21

Übungsblatt 3 Version 1.0	20 Punkte	Ausgabe: 09.12.2020, ca. 8:00 Uhr Praktomat: 16.12.2020, 13:00 Uhr Abgabefrist: 24.12.2020, 06:00 Uhr

Abgabehinweise

Bitte beachten Sie, dass das erfolgreiche Bestehen der öffentlichen Tests für eine erfolgreiche Abgabe dieses Blattes notwendig ist. Der Praktomat wird Ihre Abgabe zurückweisen, falls eine der nachfolgenden Regeln verletzt ist. Eine zurückgewiesene Abgabe wird automatisch mit null Punkten bewertet. Planen Sie entsprechend Zeit für Ihren ersten Abgabeversuch ein.

- Achten Sie auf fehlerfrei kompilierenden Programmcode.
- Verwenden Sie keine Elemente der Java-Bibliotheken, ausgenommen Elemente der Pakete `java.lang`.
- Achten Sie darauf, nicht zu lange Zeilen, Methoden und Dateien zu erstellen. Sie müssen bei Ihren Lösungen eine maximale Zeilenbreite von 120 Zeichen einhalten.
- Halten Sie alle Whitespace-Regeln ein.
- Halten Sie alle Regeln zu Variablen-, Methoden- und Paketbenennung ein.
- Wählen Sie geeignete Sichtbarkeiten für Ihre Klassen, Methoden und Attribute.
- Nutzen Sie nicht das `default`-Package.
- `System.exit()` und `Runtime.exit()` dürfen nicht verwendet werden.
- Halten Sie die Regeln zur Javadoc-Dokumentation ein.
- Halten Sie auch alle anderen Checkstyle-Regeln an.

Bearbeitungshinweise

Diese Bearbeitungshinweise sind relevant für die Bewertung Ihrer Abgabe, jedoch wird der Praktomat Ihre Abgabe **nicht** zurückweisen, falls eine der nachfolgenden Regeln verletzt ist.

- Beachten Sie, dass Ihre Abgaben sowohl in Bezug auf objektorientierte Modellierung als auch Funktionalität bewertet werden. Halten Sie die Hinweise zur Modellierung im Ilias-Wiki ein.
- Programmcode muss in englischer Sprache verfasst sein.
- Kommentieren Sie Ihren Code angemessen: So viel wie nötig, so wenig wie möglich.
- Die Kommentare sollen einheitlich in englischer oder deutscher Sprache verfasst werden.
- Wählen Sie aussagekräftige Namen für alle Ihre Bezeichner.

Plagiat

Es werden nur selbstständig angefertigt Lösungen akzeptiert. Das Einreichen fremder Lösungen, seien es auch nur teilweise Lösungen von Dritten, aus Büchern, dem Internet oder anderen Quellen, ist ein Täuschungsversuch und führt zur Bewertung „nicht bestanden“. Ausdrücklich ausgenommen hiervon sind Quelltextschnipsel von den Vorlesungsfolien und aus den Lösungsvorschlägen des Übungsbetriebes in diesem Semester. Alle benutzten Hilfsmittel müssen vollständig und genau angegeben werden und alles, was aus Arbeiten anderer unverändert oder mit Abänderungen entnommen wurde, muss deutlich kenntlich gemacht werden. Ebenso stellt die Weitergabe einer Lösung oder von Teilen davon eine Störung des ordnungsgemäßen Ablaufs der Erfolgskontrolle dar. Dieser Ordnungsverstoß kann ebenfalls zum Ausschluss der Erfolgskontrolle führen. Für weitere Ausführungen sei auf die Einverständniserklärung (Disclaimer) verwiesen.

Checkstyle

Der Praktomat überprüft Ihre Quelltexte während der Abgabe automatisiert auf die Einhaltung der Checkstyle-Regeln. Es gibt speziell markierte Regeln, bei denen der Praktomat die Abgabe zurückweist, da diese Regel verpflichtend einzuhalten ist. Andere Regelverletzungen können zu Punktabzug führen. Sie können und sollten Ihre Quelltexte bereits während der Entwicklung auf die Regeleinhaltung überprüfen. Das Programmieren-Wiki im Ilias beschreibt, wie Checkstyle verwendet wird.

Terminal-Klasse

Laden Sie für diese Aufgabe die Terminal-Klasse herunter und platzieren Sie diese unbedingt im Paket `edu.kit.informatik`. Die Methode `Terminal.readLine()` liest eine Benutzereingabe von der Konsole und ersetzt `System.in`. Die Methode `Terminal.printLine()` schreibt eine Ausgabe auf die Konsole und ersetzt `System.out`. Verwenden Sie für jegliche Konsoleneingabe oder Konsolenausgabe die Terminal-Klasse. Verwenden Sie in keinem Fall `System.in` oder `System.out`. Laden Sie die Terminal-Klasse niemals zusammen mit Ihrer Abgabe hoch.

Abgabemodalitäten

Die Praktomat-Abgabe wird am **Mittwoch, den 16. Dezember 2020 um 13:00 Uhr**, freigeschaltet. Achten Sie unbedingt darauf, Ihre Dateien im Praktomat bei der richtigen Aufgabe vor Ablauf der Abgabefrist hochzuladen. Beginnen Sie frühzeitig mit dem Einreichen, um Ihre Lösung dahingehend zu testen, und verwenden Sie das Forum, um eventuelle Unklarheiten zu klären.

- Geben Sie Ihre Klassen zu Aufgabe A in Einzelarbeit `*.java`-Dateien ab.
- Geben Sie Ihre Klassen zu Aufgabe B in Gruppenarbeit als `*.java`-Dateien ab.
- Geben Sie Ihre Klassen zu Aufgabe C in Einzelarbeit als `*.java`-Dateien ab.

Gruppenarbeit

Gruppenarbeit ist nur für besonders gekennzeichnete Aufgaben erlaubt und wird nur bis einschließlich des dritten Übungsblattes durchgeführt. Alle Aufgaben, die explizit als Gruppenarbeit gekennzeichnet sind, müssen als Gruppe erledigt werden. Eine Einzelarbeit wird nicht benotet. Aufgaben, die nicht ausdrücklich als Gruppenarbeit gekennzeichnet sind, müssen allein bearbeitet werden. Alle Gruppenmitglieder müssen die gleiche Gruppenlösung im Praktomat einreichen. Wenn ein Mitglied keine Lösung einreicht, erhält es keine Punkte. Die innerhalb einer Gruppe eingereichten Lösungen müssen identisch sein. Ist dies nicht der Fall, wird die Gruppenaufgabe der gesamten Gruppe mit null Punkten bewertet. Einzige Ausnahme ist der Fall der Nichtabgabe.
Die Gruppen bestehen in der Regel aus 3 Mitgliedern. Gruppen können nur aus Mitgliedern eines Tutoriums bestehen. Gruppen mit Mitgliedern aus verschiedenen Tutorien sind nicht möglich. Im ersten Tutorium muss eine Gruppe dem Tutor die Matrikelnummer aller Mitglieder melden. Dies ist spätestens vor Beginn der Abgabefrist im Praktomat erforderlich. Geschieht dies nicht, kann die Einreichung nicht benotet werden.
Bitte lesen Sie auch alle anderen Informationen, die im entsprechenden Ilias-Kurs bereitgestellt werden, sorgfältig durch und überprüfen Sie sie regelmäßig und selbständig auf Änderungen.

`https://ilias.studium.kit.edu/goto.php?target=crs_1255116`

Aufgabe A: Tic-Tac-Toe (7 Punkte)

Tic-Tac-Toe ist ein einfaches Spiel, in dem 2 Spieler ihre Zeichen auf die freien Zellen eines quadratischen Spielfeldes der Größe 3×3 Zellen abwechselnd platzieren. In diesem Spiel hat ein Spieler das Zeichen `X` und der andere Spieler das Zeichen `O`. Der Spieler, der als erster 3 Zeichen in eine Zeile, Spalte oder Diagonale platzieren kann, gewinnt.

In dieser Aufgabe sind alle Zellen des Spielfeldes eindeutig durchnummeriert, so wie in Abbildung 0.1 dargestellt ist.

0	1	2
3	4	5
6	7	8

Abbildung 0.1: Eindeutige Durchnummerierung der Zellen des Spielfelds

Im Gegensatz zur klassischen Variante, indem das Spiel endet, sobald ein Spieler gewinnt, betrachten wir in dieser Aufgabe eine modifizierte Variante, in der das Spiel fortgesetzt wird, bis alle Felder besetzt sind. Sie können aber annehmen, dass es trotz einer Spielfortsetzung nie zu einer Spielsituation kommt, in der 2 Spieler gewinnen. Abbildung 0.2 illustriert 3 Beispiele, bei denen *Spieler 1* mit dem Zeichen `X` gewinnt.

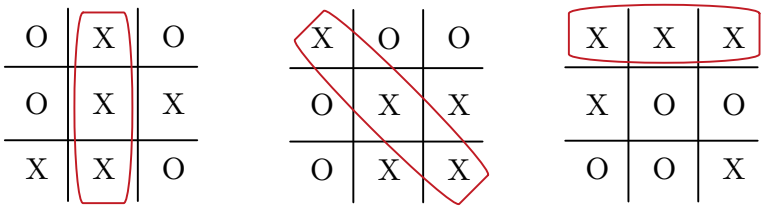

Abbildung 0.2: 3 Beispiele für Gewinnsituationen

A.1 Aufgabenstellung

Ihr Programm nimmt genau 9 `int`-Werte als Kommandozeilenargumente entgegen. Jede Zahl gibt eine eindeutige Zelle auf dem Spielfeld an, siehe hierzu Abbildung 0.1. Der erste Spieler beginnt immer das Spiel. Die Spieler platzieren Ihre Zeichen abwechselnd auf einer freien Zelle des Spielfelds. Das erste Kommandozeilenargument beschreibt die Zellennummer, auf die der *Spieler 1* sein Zeichen setzt. Das zweite Kommandozeilenargument beschreibt die Zellennummer, auf die der *Spieler 2* sein Zeichen platziert, und so weiter.

Implementieren Sie ein Programm, welches überprüft, welcher Spieler das Spiel gewinnt.

A.2 Ausgabe

Es wird genau eine Zeile auf das Terminal ausgegeben. In dieser Zeile wird ausgegeben, welcher Spieler in welchem Zug das Spiel gewonnen hat. Die Züge sind beginnend mit 1 lückenlos und fortlaufend durchnummeriert. Gewinnt der *Spieler 1* im Zug `Zugnummer`, wird `P1 wins Zugnummer` ausgegeben. Falls *Spieler 2* im Zug `Zugnummer` gewinnt, wird `P2 wins Zugnummer` ausgegeben und falls das Spiel unentschieden ausgeht, wird `draw` ausgegeben.

A.3 Beispielablauf

Der Aufruf `java TicTacToe 0 4 2 3 1 7 5 8 6` führt zum in der Abbildung 0.3 dargestellten Spielablauf, wobei *Spieler 1* das Zeichen `X` und *Spieler 2* das Zeichen `O` hat. Hierbei ist der Klassenname `TicTacToe` ist nicht vorgeschrieben. Wie Sie sehen, wird das Spiel fortgesetzt, obwohl der erste Spieler bereits im fünften Zug das Spiel gewonnen hat. Für dieses Beispiel gibt Ihr Programm `P1 wins 5` aus.

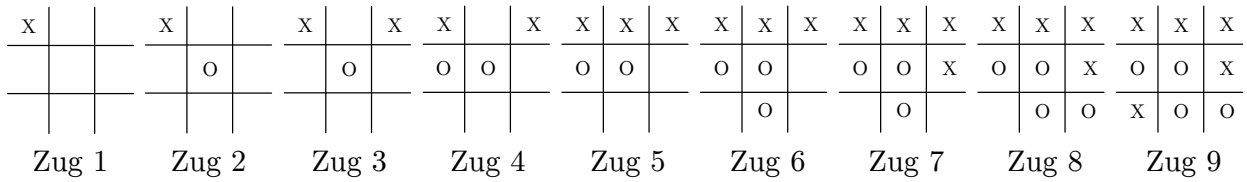

Abbildung 0.3: Beispiel eines Spielverlaufs

Aufgabe B: Modellierung einer Bank (6 Punkte)

In dieser Aufgabe werden wir vereinfacht eine Bank objektorientiert modellieren. Überlegen Sie sich bei dieser Aufgabe, ob es sich bei den jeweiligen Attributen um Klassen- oder Objektvariablen handelt. Versehen Sie, soweit sinnvoll, alle Attribute und Methoden mit passenden Sichtbarkeitsmodifizierern. Nutzen Sie für alle Ihre Klassen das Paket `edu.kit.informatik`.

Erweitern Sie alle Klassen um geeignete Abfragemethoden (Getter) und oder Änderungsmethoden (Setter). Sind Zugriffsmethode für ein Attribut mit dem Namen `Attributname` sinnvoll vorhanden, nennen Sie die entsprechenden Methodennamen `getAttributname` oder `setAttributname`. Zum Beispiel lauten die Abfragemethode und Änderungsmethoden für ein Attribut `String name` `getName` und `setName`. Begründen Sie für jede Klasse, warum für ein Attribut eine oder keine Zugriffsmethode erstellt wurde. Überlegen Sie sich hierfür welche Attribute frei änderbar sein sollten.

B.1 Klasse `Account`

Implementieren Sie eine Klasse `Account`, die ein Konto darstellt. Die Klasse `Account` besitzt folgende Attribute: `int accountNumber` für die Darstellung der Kontonummer, `int bankCode` für die Darstellung der Bankleitzahl und `int balance` für die Darstellung des Kontostandes.

Implementieren Sie einen geeigneten Konstruktor, der die Bankleitzahl und Kontonummer genau in dieser Reihenfolge entgegennimmt und diese initialisiert. Der Kontostand ist beim Initialisieren eines Kontos stets 0.

Implementieren Sie zudem folgende Methoden für Kontos:

- `public boolean withdraw(int amount)`: Diese Methode repräsentiert das Auszahlen. Dabei wird der Kontostand um den übergebenen Betrag (`amount`) verringert. Würde der Kontostand durch die Auszahlung negativ werden, so findet die Auszahlung nicht statt, d.h. der Kontostand wird nicht geändert. In diesem Fall wird `false` zurückgegeben, andernfalls wird `true` zurückgegeben.
- `public void deposit(int amount)`: Diese Methode repräsentiert das Einzahlen. Dabei wird der Kontostand um den übergebenen Betrag (`amount`) erhöht.
- `public boolean transfer(Account account, int amount)`: Diese Methode repräsentiert das Überweisen. Dabei entspricht `account` dem Zielkonto und `amount` dem Betrag, der überwiesen werden soll. Der Kontostand des Quellkontos, von dem der Betrag überwiesen wird, wird um den übergebenen Betrag verringert und der Kontostand des Zielkontos, worauf der Betrag überwiesen wird, wird um diesen Betrag erhöht. Würde der Kontostand des Quellkontos durch die Auszahlung negativ werden, so findet die Auszahlung nicht statt, d.h. die Überweisung findet nicht statt. In diesem Fall wird `false` zurückgegeben, andernfalls wird `true` zurückgegeben.

B.2 Klasse `Bank`

Implementieren Sie eine Klasse `Bank` zur Darstellung der Banken. Die Klasse `Bank` hat eine Bankleitzahl `bankCode`) vom Typ `int` und ein Array von Konten (`accounts`) mit einer initialen Länge von 8. Implementieren Sie einen geeigneten Konstruktor, welcher die Bankleitzahl entgegennimmt und die Bank initialisiert.

- `public int createAccount()`: Diese Methode repräsentiert das Anlegen eines neuen Kontos innerhalb der Bank. Jedes Konto, das innerhalb dieser Bank erstellt wird, bekommt die Bankleitzahl der Bank. Die Kontonummern werden fortlaufend vergeben, beginnend bei 0. Innerhalb einer Bank wird keine Kontonummer jemals erneut vergeben. Das neue Konto wird im ersten freien Feld des Arrays der Konten innerhalb der Bank eingefügt. Die Methode gibt schließlich die Kontonummer des neuen Kontos zurück. Werden mehr Konten erzeugt als die Größe des Arrays, wird intern die Länge des Arrays um seine bisherige Länge verdoppelt. Das neue Konto wird, wie oben beschrieben, dem Array hinzugefügt.
- `public boolean removeAccount(int accountNumber)`: Diese Methode repräsentiert das Löschen eines Kontos innerhalb dieser Bank. Existiert die zu löschende Kontonummer, wird `true` zurückgegeben. Zudem wird das zugehörige Konto vom Array von Konten entfernt. Sind weniger als $\frac{1}{4}$ des Arrays besetzt, wird die Länge des Arrays halbiert. Dabei wird die Länge des Arrays nie kleiner als die initiale Länge des Arrays, d.h. 8. Beim Löschen eines Array-Elementes darf keine Lücke entstehen. Dies bedeutet, dass alle Elemente, die nach dem zu löschenden Account kommen, nach *vorne* aufrücken. Existiert kein Konto mit der vorgegebenen Kontonummer, wird `false` zurückgegeben.
- `public boolean containsAccount(int accountNumber)`: Diese Methode gibt an, ob es ein Konto mit der Kontonummer `accountNumber` im Array existiert. Existiert ein solches Konto wird `true` zurückgegeben, andernfalls `false`.
- `public boolean internalBankTransfer(int fromAccountNumber, int toAccountNumber, int amount)`: (Ohne den Zeilenumbruch zwischen dem zweiten Methodenparameter und seinem Datentyp) Diese Methode repräsentiert das Überweisen. Dabei entspricht `fromAccountNumber` der Kontonummer des Quellkontos und `toAccountNumber` der Kontonummer des Zielkontos und `amount` dem Betrag, der überwiesen werden soll. Der Kontostand des Quellkontos, von dem der Betrag überwiesen wird, wird um den übergebenen Betrag verringert und der Kontostand des Zielkontos, worauf der Betrag überwiesen wird, wird um diesen Betrag erhöht. Weiter können Sie davon ausgehen, dass nur Überweisungen innerhalb der Konten einer Bank stattfinden. Existieren die Kontonummer des Quellkontos oder Zielkontos nicht oder wird der Kontostand des Quellkontos durch diese Überweisung negativ, findet keine Überweisung statt. In diesem Fall wird `false` zurückgegeben. Findet die Überweisung erfolgreich statt, wird `true` zurückgegeben.
- `public int length()`: Diese Methode gibt die aktuelle Länge des Arrays (und nicht die Anzahl der besetzten Array-Felder) zurück.
- `public int size()`: Diese Methode gibt die Anzahl der besetzten Arrayfelder (`accounts`) zurück.

- `public Account getAccount(int index)`: Diese Methode gibt gibt das gespeicherte Element mit dem Index `index` im Array zurück. In allen anderen Fällen, beispielsweise wenn das entsprechende Array-Feld leer ist oder wenn der Index im Array gar nicht vorkommt, wird `null` zurückgegeben.

B.3 Klasse `Test`

Schreiben Sie schließlich zu den folgenden Klassen eine separate Klasse `Test` mit einer `main`-Methode. In dieser Methode soll jede Klasse mindestens zweimal mit unterschiedlichen Attributen instanziiert und jede Methode mindestens zweimal mit unterschiedlichen Parametern aufgerufen werden.

Aufgabe C: Routen Planen (7 Punkte)

Betrachten Sie zum Einstieg den in Abbildung 0.4 dargestellten Graphen. Dieser Graph beschreibt Städte (dargestellt als Knoten) und Routen zwischen Städten (dargestellt als Kanten). Wir gehen davon aus, dass alle Städte beginnend mit 0 lückenlos und fortlaufend durchnummeriert sind. Auf Basis dieser Daten lassen sich unter anderem Fragestellungen beantworten, wie z.B. ob es einen Pfad zwischen zwei Städten mit einer bestimmten Anzahl an besuchten Städten gibt?

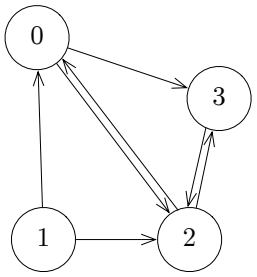

Abbildung 0.4: Beispiel eines Graphen. Knoten repräsentieren Städte. Kanten repräsentieren die Routen zwischen Städten.

In dieser Aufgabe programmieren Sie einen vereinfachten Routenplaner, der seinen Benutzern diese Fragestellung beantwortet.

C.1 Grundlagen: Graph

Ein *Graph* besteht aus *Knoten*, die durch *Kanten* miteinander verbunden sein können. Bei einem *gerichteten Graph* besitzen die Kanten außerdem eine Richtung, jede Kante zeigt von einem Startknoten auf einen Zielknoten. Wenn zwischen zwei Knoten höchstens eine Kante je Richtung erlaubt ist, spricht man von einem *Graph ohne Mehrfachkanten*.

Formeller ausgedrückt ist ein Graph G ein geordnetes Paar (V, E), wobei V die Menge seiner Knoten (*Vertices*) und E die Menge seiner Kanten (*Edges*) bezeichnet. Eine Kante $e \in E$ ist ein geordnetes Paar (v_s, v_t) mit $v_s \in V$ und $v_t \in V$. Dabei bezeichnet v_s den Startknoten und v_t den Zielknoten der Kante e.

C.1.1 Routen-Graph

Ein *Routen-Graph* ist ein gerichteter Graph $G = (V, E)$ ohne Mehrfachkanten. Jeder Knoten $v \in V$ repräsentiert eine Stadt. Eine Stadt ist eindeutig durch eine Zahl gekennzeichnet. Jede Kante $e = (v_s, v_t) \in E$ repräsentiert eine Route. Dabei ist v_s die Startstadt und v_t die Zielstadt.

C.1.2 Serialisierung eines Routen-Graphen

Unter einer *Serialisierung* verstehen wir im Rahmen dieser Aufgabe die textuelle Repräsentation eines Routen-Graphen: ein Format, um einen solchen Graphen in eine Datei zu speichern bzw. aus einer Datei auszulesen.

Eine Serialisierung besteht aus der Beschreibung der Kanten des Graphen. Dieser enthält eine oder mehrere Zeilen in beliebiger Reihenfolge. Jede Zeile beschreibt eine Route (Kante) und ist folgendermaßen aufgebaut:

`<Startstadt> <Zielstadt>`

Jede Route verläuft von einer Startstadt zu einer Zielstadt, wobei gilt Startstadt $\neq$ Zielstadt. `<Startstadt>` und `<Zielstadt>` sind jeweils durch eine eindeutige Zahl repräsentiert. Eingabedatei für das in Abbildung 0.4 dargestellte Beispiel:

```
1 1 0
2 1 2
3 2 0
4 0 2
5 2 3
6 3 2
7 0 3
```

C.2 Grundlagen: Adjazenzmatrix

C.2.1 Matrix

Eine *Matrix* ist eine tabellarische (d.h. 2-dimensionale) Anordnung von Elementen, in unserem Fall von ganzen Zahlen. Der Eintrag $m_{i,j}$ bezeichnet die Zahl, die in der i-ten Zeile und j-ten Spalte einer Matrix M steht. Nachfolgend drücken wir dies in Kurzschreibweise aus: $M = (m_{i,j})$.

C.2.2 Adjazenzmatrix

Eine *Adjazenzmatrix* ist eine spezielle Matrix zur Repräsentation eines Graphen.

Schritt 1: Knoten durchnummerieren Wir gehen davon aus, dass alle Knoten $v \in V$ fortlaufend beginnend mit 0 durchnummeriert sind. Diese Nummerierung erlaubt es uns, $(2, 3) \in E$ zu schreiben, um auszudrücken, dass eine Kante von der Stadt mit der Nummer 2 zur Stadt mit der Nummer 3 verläuft. Beachten Sie, dass diese Nummerierung durch die Eingabe fest vorgegeben ist.

Schritt 2: Adjazenzmatrix bilden Ein Graph $G = (V, E)$ lässt sich nun wie folgt als Adjazenzmatrix $A = (a_{i,j})$ darstellen, wobei die Anzahl der Zeilen bzw. Spalten in A der Knotenanzahl des Graphen entspricht:

$$a_{i,j} = \begin{cases} 1 & falls\ (i,j) \in E \\ 0 & sonst \end{cases}$$

Steht an der Stelle $a_{i,j}$ eine 1 bedeutet das somit, dass Knoten i mit Knoten j durch die Kante (i, j) verbunden ist. Ist $a_{i,j}$ hingegen 0, bedeutet das, dass die Knoten i und j nicht durch eine Kante verbunden sind.

Beispiel Der Graph aus Abbildung 0.4 lässt sich durch die Adjazenzmatrix A darstellen:

$$A = \begin{pmatrix} 0 & 0 & 1 & 1 \\ 1 & 0 & \mathbf{1} & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

In Verbindung mit der oben eingeführten Beispiel-Nummerierung drückt der Eintrag $a_{1,2} = 1$ (dargestellt in Fettdruck) aus, dass ausgehend von Knoten mit der Knotennummer 1 eine Kante auf Knoten mit der Knotennummer 3 zeigt.

C.2.3 Anzahl der Pfade zwischen zwei Knoten berechnen

Auf Basis einer *Adjazenzmatrix* lässt sich berechnen, wie viele Pfade von Knoten i zu Knoten j verlaufen, die eine bestimmte Pfadlänge besitzen. Pfade der Länge 1 sind in unserem Fall direkte Routen. Diese kann man direkt aus der Adjazenzmatrix ablesen. Pfade der Länge 2 sind in unserem Fall Routen mit genau einer Stadt (bzw. einem Knoten) dazwischen. Diese lassen sich mittels *Matrix-Multiplikation* bestimmen, indem die Adjazenzmatrix mit sich selbst multipliziert wird.

Für zwei quadratische Matrizen $A = (a_{i,j})$ und $B = (b_{i,j})$ erhält man die Ergebnismatrix $C = (c_{i,j})$ folgendermaßen:

$$c_{i,j} = \sum_{k=1}^{m} a_{i,k} \cdot b_{k,j}$$

Dabei ist m die Anzahl der Zeilen bzw. Spalten von A und B. Weil A und B quadratisch sind, besitzt C auch m Zeilen und Spalten.

Sei A die Adjazenzmatrix aus Abschnitt C.2.2, dann enthält die aus der Multiplikation $A \times A$ resultierende Matrix C die Anzahl aller Pfade der Länge 2 zwischen zwei bestimmten Knoten:

$$C = A \times A = \begin{pmatrix} 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \times \begin{pmatrix} 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 2 \\ 0 & 0 & 2 & 1 \\ 1 & 0 & 0 & 1 \end{pmatrix}$$

Aus der Ergebnismatrix C können wir ablesen, dass es genau einen Pfad der Länge 2 gibt, der von der Stadt mit der Nummer 1 zur Stadt mit der Nummer 2 führt (weil $c_{1,2} = 1$). Für die Route von der Stadt mit der Nummer 1 zur Stadt mit der Nummer 3 gibt es zwei Pfade der Länge 2 (weil $c_{1,3} = 2$). Die Ergebnismatrix beantwortet zwar nicht, welche Pfade dies sind, doch mit einem Blick auf den Graphen sind die beiden Pfade leicht zu entdecken: Stadt mit der Nummer 1 → Stadt mit der Nummer 2 → Stadt mit der Nummer 3, sowie Stadt mit der Nummer1 → Stadt mit der Nummer 0 → Stadt mit der Nummer 2.

Statt $A \times A$ schreiben wir im Folgenden kürzer A^2. Statt $A \times A \times A$ schreiben wir A^3, und so weiter.

Berechnet man die Matrix A^l, kann man daraus die Anzahl aller Pfade ablesen, die genau die Länge l haben.

C.3 Kommandozeilenargumente

Ihr Programm nimmt 4 Kommandozeilenargumente entgegen. Sie können davon ausgehen, dass alle Kommandozeilenargumente gültig sind.

Das erste Kommandozeilenargument ist ein Pfad auf eine Textdatei. Diese Datei beschreibt einen Routen-Graphen und ist nach dem in Abschnitt C.1.2 eingeführten Format aufgebaut. Um den Inhalt dieser Datei einzulesen muss die Methode `readFile` der Klasse `Terminal` verwendet werden.

Das zweite Kommandozeilenargument ist die Nummer der Stadt, von der der gesuchte Pfad ausgeht. Sie können davon ausgehen, dass dieses Kommandozeilenargument immer einer gültigen Stadtnummer entspricht. Das dritte Kommandozeilenargument ist die Nummer der Stadt, zu der der gesuchte Pfad führt. Sie können davon ausgehen, dass dieses Kommandozeilenargument immer einer gültigen Stadtnummer entspricht. Das vierte Kommandozeilenargument entspricht der gesuchten Länge des Pfades zwischen der Startstadt (2. Kommandozeilenargument) und der Zielstadt (3. Kommandozeilenargument).

C.4 Ausgabeformat

Die Ausgabe Ihres Programms besteht aus genau einer Zahl, welche die Anzahl der Pfade mit einer bestimmten Länge (4. Kommandozeilenargument) zwischen der Startstadt (2. Kommandozeilenargument) und der Zielstadt (3. Kommandozeilenargument) basierend auf dem Graphen der Eingabedatei (1. Kommandozeilenargument) ausgibt. Existiert kein Pfad mit der gesuchten Länge, wird 0 ausgegeben. Im obigen Beispiel in Abschnitt C.2.3 gibt es für die Route von der Stadt mit der Nummer 1 zur Stadt mit der Nummer 3 zwei Pfade der Länge 2.