252-0027-00: Einführung in die Programmierung

Übungsblatt 8

Abgabe: 19. November 2019, 10:00

Checken Sie mit Eclipse wie bisher die neue Übungs-Vorlage aus.

Aufgabe 1: Enthalten mit Abstand (Bonus!)

Achtung: Diese Aufgabe gibt Bonuspunkte (siehe "Leistungskontrolle" im `www.vvz.ethz.ch`). Die Aufgabe muss eigenhändig und alleine gelöst werden. Die Abgabe erfolgt wie gewohnt per Push in Ihres Git-Repository auf dem ETH-Server. Verbindlich ist der letzte Push vor dem Abgabetermin. Bitte lesen Sie zusätzlich die allgemeinen Regeln.

Gegeben seien zwei Strings `s` und `t` und ein Integer `k` mit $k \geq 0$. Schreiben Sie ein Programm, das zurückgibt ob die Zeichen aus `t` in beliebiger Reihenfolge als Subsequenz in `s` vorkommen, in welcher die in der Subsequenz aufeinanderfolgende Zeichen *maximal* einen Abstand von `k` in `s` haben. Der Abstand zwischen zwei Zeichen `a` und `b` in einer Sequenz ist die Anzahl Zeichen zwischen `a` und `b`. Zum Beispiel in `"a12345b"` haben das `a` und das `b` einen Abstand von 5.

Beispiele:

- `s = "abbbc"`, `t = "cab"`, `k = 1`. Das Programm sollte `true` zurückgeben, da `"abc"` in `s` als Subsequenz mit Abstand maximal 1 vorkommt (Die Subsequenz in `s` ist das erste, das dritte, und das letzte Zeichen).
- `s = "abbbbc"`, `t = "cab"`, `k = 1`. Das Programm sollte `false` zurückgeben, da entweder der Abstand zwischen `"a"` und `"b"` oder der Abstand zwischen `"b"` und`"c"` immer grösser ist als 1.
- `s = "abc"`, `t = "cbab"`, `k = 1`. Das Programm sollte `false` zurückgeben, da es keine Subsequenz in **s** gibt, welche zwei `"b"` enthält.

Vervollständigen Sie die Methode `enthalten()` in der Klasse `EnthaltenMitAbstand`. Die Methode hat drei Argumente: die beiden Strings `s` und `t` und der Integer `k`. Sie dürfen davon ausgehen, dass die Strings nie `null` sind und dass der Integer grösser oder gleich 0 ist. In der Datei "EnthaltenMitAbstandTest.java" finden Sie ein paar einfache Tests. Wir empfehlen Ihre Lösung ausgiebig zu testen. **Tipp:** Lösen Sie die Aufgabe rekursive.

Aufgabe 2: Teilfolgen

Schreiben Sie ein Programm `Teilfolgen`, welches mithilfe von Rekursion alle Teilfolgen der Länge `N` eines Strings `S` auf der Konsole ausgibt. Lesen Sie `S` und `N` (eine nicht-negative ganze Zahl) von der Konsole ein.

Eine Teilfolge ist definiert als eine Folge von Zeichen (`String`), die durch das Weglassen von Zeichen aus einer ursprünglichen Folge von Zeichen entsteht. Beispielsweise sind alle Teilfolgen der Länge `N=2` des `Strings` `S="apple"`:

`"ap", "al", "ae", "pp", "pl", "pe", "le"`

Ob Sie im Beispiel die Teilfolge `"ap"` einmal oder zweimal ausgeben spielt für diese Aufgabe keine Rolle.

Aufgabe 3: Warteschlangen-Simulation

Doppelt verkettete Listen eignen sich gut als Warteschlangen (engl. "Queue"). "Software-Warteschlangen" werden in vielen Anwendungen verwendet, um Objekte oder Daten zwischenzuspeichern, bevor sie verarbeitet werden. In dieser Aufgabe sollen Sie sie hingegen verwenden, um die realen Warteschlangen vor den Kassen in Ihrer nächsten Migros-Filiale zu simulieren.

Die Simulation läuft in diskreten Schritten ab. Das heisst, die Zeit wird in gleichgrosse Intervalle unterteilt und in jedem Intervall passieren gewisse Dinge. Zum Beispiel kann in einem Intervall ein Kassierer einen Gegenstand scannen oder eine Person die Warteschlange wechseln. Die Simulation hat folgende Parameter:

Anzahl Kassen und Kassier-Effizienz: Eine Liste $[p_0, p_1, ..., p_n]$ der Länge n mit $0 \leq p_i < 1$. Es gibt n (besetzte) Kassen und der i-te Kassierer scannt in jedem Zeitschritt mit Wahrscheinlichkeit p_i einen Gegenstand aus dem Warenkorb der vordersten Person in seiner Warteschlange. Wenn er den letzten Gegenstand einer Person gescannt hat, verlässt diese Person die Kasse.

Erschein-Wahrscheinlichkeit: Eine Zahl e, wobei $0 \leq e < 1$. In jedem Zeitschritt erscheint mit Wahrscheinlichkeit e eine neue Person und steht an einer zufälligen Schlange an.

Warenkorb-Grösse: Eine natürliche Zahl w. Jede Person hat zu Beginn zwischen 0 und w Gegenstände in seinem Warenkorb. Die genaue Anzahl jeder Person ist zufällig.

Faulheit der Leute: Eine natürliche Zahl f. Da gewisse Schlangen länger werden können als andere, möchten die Leute vielleicht mal wechseln, abhängig von ihrer Faulheit. Eine Person wechselt (in jedem Zeitschritt) zur momentan kürzesten Schlange, falls diese um mindestens f kürzer ist als die aktuelle Position[1] dieser Person in seiner Schlange.

Implementieren Sie diese Simulation in einer Klasse `MigrosSimulation`, welche diese Parameter im Konstruktor entgegen nimmt. Die Klasse soll eine `run()`-Methode haben, welche die Anzahl Zeitschritte entgegen nimmt und die Simulation durchführt. Danach können die Resultate der Simulation mit Accessor-Methoden ("getter") ausgelesen werden. Die Implementation der Simulation soll von externem Zugriff geschützt sein ("encapsulation"). Folgende Resultate sollen zur Verfügung gestellt werden:

[1] Positionen sind 0-indexiert, d.h., die vorderste Person hat Position 0, die nächste Position 1, usw.

`getFinished()`: Eine Liste mit allen Personen, welche es bis zum Ende der Simulation durch den Kassenbereich "geschafft" haben. Jede Person enthält die ursprüngliche Anzahl Gegenstände in ihrem Warenkorb und die Zeit (in Anzahl Zeitschritten), die sie in den Warteschlangen verbracht hat.

`getAvgQueueLengths()`: Ein Array, welches die *durchschnittliche* Länge jeder Warteschlange während der gesamten Simulation enthält.

`getMaxQueueLengths()`: Ein Array mit den *maximalen* Längen der Warteschlangen.

Ändern Sie dazu Ihre `Person`-Klasse so ab, dass sie statt Körpergrösse, usw. die für die Simulation benötigten Felder enthält. Schreiben Sie schliesslich eine Klasse `MigrosSimulationApp` mit einer `main()`-Methode, welche einige (oder alle) Parameter für die Simulation von der Konsole einliest, die Simulation erstellt und durchführt und schliesslich einige interessante Daten ausgibt. Verwenden Sie zum Beispiel die Methoden aus der gegebenen `Utils`-Klasse, um die durchschnittlichen und maximalen Längen der Warteschlangen schön auszugeben.

Zusatz-Aufgabe: Wenn Sie wollen, können Sie die Simulation erweitern, indem Sie z.B. die Zeit um die Schlange zu wechseln simulieren, während der Simulation Kassen öffnen und schliessen oder weitere "Persönlichkeitsmerkmale" einfliessen lassen.

Aufgabe 4: Dominator

Im vorletzten Übungsblatt haben Sie Black-Box Testing kennengelernt. Wie in der letzten Aufgabe sollen Sie wieder Tests für eine Methode schreiben, deren Implementierung Sie nicht kennen. In Ihrem "U08"-Projekt befindet sich eine "blackbox.jar"-Datei, welche eine kompilierte Klasse `BlackBox` enthält. Den Code dieser Klasse können Sie nicht sehen, aber sie enthält eine Methode `int[] dominators(int[] elevations)`, welche Sie aufrufen können.

Die Methode `dominators` hat den Parameter `elevations`. Das Array `elevations` enthält die Höhenwerte (in Metern über Meer) für eine Serie von Punkten. Für jeden Punkt P (ein Index in `elevations`) berechnet die Methode, durch welchen anderen Punkt er *dominiert* wird. Der Rückgabewert ist ein Array, hier mit `result` bezeichnet, welches für jeden Punkt P die Position (Index im `elevations`-Array) eines Dominators von P enthält. Falls `elevations` `null` ist, ist auch `result` `null`.

Ein *Dominator* D eines Punktes P hat von allen Punkten, welche höher als P liegen, die kleinste (horizontale) Distanz zu P. Falls zwei solche Punkte existieren, ist der höhere der beiden der einzige Dominator. Falls beide dieser Punkte gleich hoch sind, gibt es zwei mögliche Dominatoren. Falls kein Dominator für P existiert, enthält `result[`P`]` die Zahl `-1`.

Ein Beispiel finden Sie in der Datei "BlackBoxTest.java" in Ihrem Projekt. Der gegebene Test `testSimple()` übergibt als Argument die Höhenserie `{3, 1, 2, 5, 3, 6, 5, 6}`. Die Methode `dominators` sollte nun entweder `result={3, 0, 3, 5, 5, -1, 5, -1}` oder `result={3, 0, 3, 5, 5, -1, 7, -1}` zurückgeben. Andernfalls enthält die Implementierung einen Fehler.

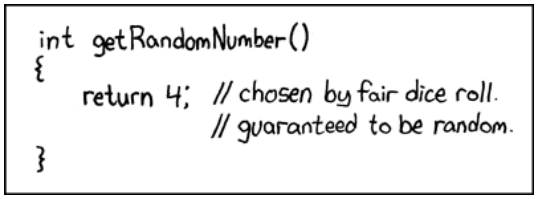
Der zweite gegebene Test `testReturnsNull()` prüft, dass die `dominators`-Methode `null` zurück gibt, wenn `null` als Argument übergeben wird. *Dieser Test schlägt für die gegebene `BlackBox`-Implementierung fehl.* Damit zeigt dieser Test, dass sich diese Implementierung nicht vollständig an die obige Spezifikation hält. Da Sie nur diese eine Implementierung zur Verfügung haben, wird dieser Test auch zum Zeitpunkt der Abgabe fehlschlagen.

Schreiben Sie nun solange weitere Tests (in "BlackBoxTest.java"), bis sie zuversichtlich sind, dass Ihre Tests die Spezifikation genügend abdecken. Um die Stärke Ihrer Tests zu beurteilen, werden wir verschiedene, teilweise fehlerhafte `BlackBox`-Implementierungen mithilfe Ihrer Tests prüfen. Je mehr Fehler Ihre Tests aufdecken, desto besser. Tests sollten fehlschlagen, falls die Implementierung fehlerhaft ist, und erfolgreich durchlaufen, falls keine Fehler vorhanden sind.

Warnung: Da Unit-Tests generell höchstens einige Sekunden dauern, werten wir Tests, die länger als eine Minute dauern, als fehlerhaft. Sie können davon ausgehen, dass keine der `BlackBox`-Implementierungen Endlosschleifen enthalten.

Aufgabe 5: Self-avoiding Random Walks

In dieser Aufgabe simulieren Sie das Schicksal eines Wolfs, der sich genau in der Mitte einer Stadt befindet und aus dieser ausbrechen will. Die Stadt besteht aus $2 \times N$ Strassen, die in einem regelmässigen Gitter angeordnet sind. N Strassen verlaufen in der West-Ost Richtung, N in der Nord-Süd Richtung. (N ist ungerade und > 1.) Die Stadt kann (für diese Aufgabe) also vollständig durch die $N \times N$ Kreuzungen der Strassen beschrieben werden.



xkcd: Random Number by Randall Munroe (CC BY-NC 2.5)

An jeder Kreuzung entscheidet der Wolf, in welche Richtung er fliehen will. Wenn der Wolf an der Kreuzung (i, j) steht, so kann er (dank seiner feinen Nase) feststellen, welche der Nachbar-Kreuzungen $(i+1, j)$, $(i-1, j)$, $(i, j+1)$, und $(i, j-1)$ er schon besucht hat[2]. Eine einmal besuchte Kreuzung wird nicht nochmal besucht, d.h., der Wolf wählt zufällig unter den Richtugen, die ihn zu einer neuen, noch nicht besuchten Kreuzung führen. Wenn der Wolf den Stadtrand (d.h. die 1. oder die N. Strasse in nord-südlicher oder west-östlicher Richtung) erreicht hat, ist die Flucht erfolgreich verlaufen. Wenn der Wolf an eine Kreuzung kommt, von der aus er keine unbesuchte Kreuzung erreichen kann, dann ist die Flucht zu Ende und der Wolf wird (entsprechende Bewilligung vorausgesetzt) von den Jägern erschossen. Abbildung 1 illustriert dies anhand von 5×5 und 7×7 Städten.

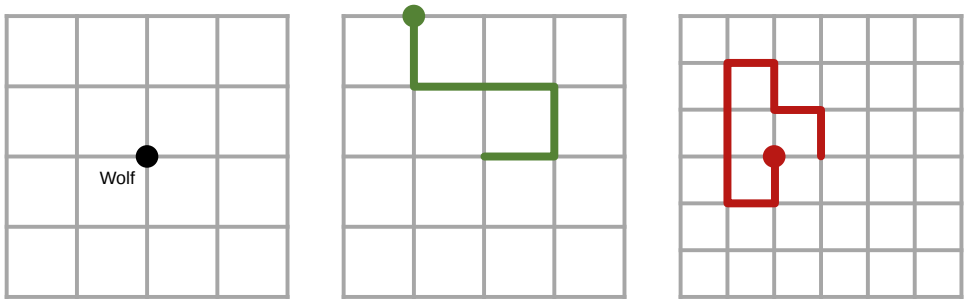


Abbildung 1: Wolf in der Stadt. Ausganglage, erfolgreiche Flucht und fehlgeschlagene Flucht

[2]Er hat mindestens eine dieser Kreuzungen im vorherigen Schritt besucht um zu (i, j) zu kommen.

a) Schreiben Sie ein Programm `RandomWalks`, welches die Flucht des Wolfes simuliert. Sie können die Grösse der Stadt (N) sowie die Anzahl der Simulationen von der Konsole einlesen oder fest in zwei Variablen vorgeben. Nach der Simulation soll Ihr Programm ausgeben, in wieviel Prozent der Fälle der Wolf aus der Stadt fliehen konnte. Ausserdem, wie lang im Durchschnitt der Pfad war, der es einem Wolf erlaubte zu fliehen und wie lang (im Durchschnitt) der Pfad war, wenn der Wolf nicht aus der Stadt fliehen konnte. Die Länge des Pfades wird durch die Anzahl der besuchten Kreuzungen bestimmt, inklusive der Kreuzung, an welcher der Wolf startete.

   Tipps: Verwenden Sie ein 2-dimensionales Array von `booleans`, um sich die bereits besuchten Kreuzungen zu merken. Um per Zufall eine Richtung zu wählen, wird sich die Methode `Random.nextInt(int)` als nützlich erweisen.

   Experimentieren Sie mit Ihrem Programm und vergleichen Sie z.B. die Wahrscheinlichkeit einer Flucht aus einer 9×9 Stadt mit der einer grösseren Stadt. Diese Simulation ist ein einfaches Beispiel für das Problem der "self-avoiding-paths" (der Pfade, die sich nicht kreuzen), welches eine Abstraktion für viele interessante Probleme aus Chemie, Pharmazie, Biologie und anderen Gebieten ist. (Nach Sedgewick & Wayne, Einführung in die Programmierung mit Java.)

b) Nun sollen Sie die gelaufenen Pfade auf ein Fenster zeichnen. Falls ein Pfad erfolgreich war, malen Sie ihn in grün, ansonsten in rot. Dazu müssen Sie sich den Pfad während der Simulation in einer `LinkedIntList` merken und in einem zweiten Schritt (wenn Sie wissen, ob der Pfad erfolgreich war) in der richtigen Farbe zeichnen. Jeden Schritt des Wolfs können Sie als `int` kodiert der Liste hinzufügen, beispielsweise kodiert als 0=Norden, 1=Osten, usw. Es ist Ihnen überlassen, ob Sie die Pfade schrittweise (animiert) zeichnen, oder jeden Pfad auf einmal.