Übungen Programmieren in Clojure Serie 8

1. Logging

Definieren Sie ein Atom namens `*log*`, das einen Vektor von Strings enthalten kann. Entwickeln Sie folgende Funktionen:

(a) `new-log`, die `*log*` leer initialisiert

(b) `log`, die einen Eintrag anhängt

(c) `print-log`, die alle Einträge auf die Standardausgabe ausgibt.

2. Shuffle

Was passiert in folgender Funktion?

```
(defn run [nvecs nitems nthreads niters]
  (let [vec-refs (vec (map (comp ref vec)
                           (partition nitems (range (* nvecs nitems)))))
        swap #(let [v1 (rand-int nvecs)
                    v2 (rand-int nvecs)
                    i1 (rand-int nitems)
                    i2 (rand-int nitems)]
                (dosync
                 (let [temp (nth @(vec-refs v1) i1)]
                   (alter (vec-refs v1) assoc i1 (nth @(vec-refs v2) i2))
                   (alter (vec-refs v2) assoc i2 temp))))
        report #(do
                  (prn (map deref vec-refs))
                  (println "Distinct:"
                           (count (distinct (apply concat (map deref vec-refs
                               ))))))]
    (report)
    (dorun (apply pcalls (repeat nthreads #(dotimes [_ niters] (swap)))))
    (report)))
```

3. Nochmals Shuffle

Worin unterscheidet sich die folgende Funktion von der vorherigen und welchen Einfluss hat der Unterschied auf die Laufzeit?

```
(defn run [nvecs nitems nthreads niters]
  (let [vec-refs (vec (map vec
                       (partition nitems (map ref (range (* nvecs nitems)))))
                          )
        swap #(let [v1 (rand-int nvecs)
                    v2 (rand-int nvecs)
                    i1 (rand-int nitems)
                    i2 (rand-int nitems)]
                (dosync
                 (let [temp @(nth (vec-refs v1) i1)]
                   (ref-set (nth (vec-refs v1) i1)  @(nth (vec-refs v2) i2))
                   (ref-set (nth (vec-refs v2) i2) temp))))
        report #(do
                  (println "Distinct:"
```

```
                        (count (distinct (map deref (apply concat vec-
                            refs))))))]
    (dorun (apply pcalls (repeat nthreads #(dotimes [_ niters] (swap)))))
    (report)))
```

4. Verklemmung

Definieren Sie zwei Refs `a` und `b`.

Versuchen Sie eine Verklemmung zu erzeugen, indem Sie in einem Thread die Ref `a` zunächst gesichert lesen (`ensure`) und dann die Ref `b` schreiben – und in einem zweiten Thread dasselbe mit vertauschten Refs tun.

Was passiert? Erläutern Sie das Verhalten von Clojures STM.

5. Game of Life

John Conway hat 1970 sein *Game of Life* erfunden. Auf einem Spielfeld kann jede Zelle zwei Zustände annehmen: *lebend* oder *tot*.

Das Feld wird mit einer Anfangsgeneration bevölkert, dann entwickelt sich das Spiel Generation für Generation nach folgenden Regeln:

- Eine tote Zelle mit genau drei lebenden Nachbarn wird in der Folgegeneration neu geboren.
- Lebende Zellen mit weniger als zwei lebenden Nachbarn sterben in der Folgegeneration an Einsamkeit.
- Eine lebende Zelle mit zwei oder drei lebenden Nachbarn bleibt in der Folgegeneration lebend.
- Lebende Zellen mit mehr als drei lebenden Nachbarn sterben in der Folgegeneration an Überbevölkerung.

Verwenden Sie die Datei `life.clj` als Startpunkt, um das Spiel des Lebens zu implementieren.

`life.clj` verwendet für die Visualisierung *quil*, siehe https://github.com/quil/quil. Damit Sie quil nutzen können, müssen Sie in die Datei `project.clj` die Abhängigkeit
`:dependencies [quil "2.6.0"]` eintragen.

Ihre Aufgabe besteht darin, das Spiel zu implementieren und mit quil zu visualisieren.

6. Zahl der Lösungen des Damenproblems

Schreiben Sie eine Funktion, die die Zahl der Lösungen des Damenproblems für `n` Damen ermittelt.