Automi cellulari 1D

JUnit + Multithreading

Cos'è un automa cellulare 1D

```
00000000001010101010000000 // Le cellule
```

Immaginiamo una striscia di cellule, ogni cellula puo' essere viva/accesa (1) o morta/spenta (0)

Ad ogni passo, ogni cella guarda:

- se' stessa
- il vicino a sinistra
- il vicino a destra

In base a questi 3 valori, decide se sara' accesa o spenta al passo successivo

```
Generazione 0:  ...0001000...
                    ↓↓↓
Generazione 1:  ...???????...
```

Il risultato dipende dalla **regola** che usiamo

Automi cellulari su Wikipedia sezione "L'esempio più semplice"

Puoi vedere tutti i pattern qui
https://mathworld.wolfram.com/ElementaryCellularAutomaton.html

Le 256 regole di Wolfram

Con 3 celle binarie abbiamo **8 combinazioni** di "vicini" possibili

Per ogni combinazione decidiamo l'output: acceso o spento

```
Pattern:   111  110  101  100  011  010  001  000
Output:     ?    ?    ?    ?    ?    ?    ?    ?
```

8 scelte binarie → 2^8 = **256 regole possibili** (da 0 a 255)

Gli output, letti come numero binario, danno il **nome** alla regola

Wolfram's Elementary Cellular Automata

Esempio: regola 110

```
Pattern:   111  110  101  100  011  010  001  000
Output:     0    1    1    0    1    1    1    0
```

Output in binario: `01101110` = **110** in decimale

Ecco perche' si chiama "regola 110".

La regola 110 e' Turing-completa!

Come funziona calcolaStato

Ai 3 valori corrisponde un "pattern"/combinazione

```
pattern = sinistra + centro + destra # esempio: 010
```

Il bit corrispondente nella regola da' il risultato:

Esempio: regola 110, pattern `010` (= 2)

```
110 (regola) in binario = 0 1 1 0 1 1 1 0
bit:                      7 6 5 4 3 2 1 0
                                    ^
                        bit 2 = 1  →  l'uscita sarà una cella accesa!
```

Test con JUnit

Prima di passare ai multithread, usiamo JUnit per vedere se stiamo simulando correttamente le cellule con una data regola

Avete anche un metodo `stampaStato`, utile per il **debug**, ma non usatelo durante le simulazioni multithread (stampare a video è moooolto lento)

Conversioni utili

```java
// Da binario (stringa) a decimale
String binario = "1010";
int numero = Integer.parseInt(binario, 2);
System.out.println(numero); // Output: 10

// Da decimale a binario (stringa)
int decimale = 10;
String bin = Integer.toBinaryString(decimale);
System.out.println(bin); // Output: 1010
```

Fase 1: Cose da fare

1. Completa `calcolaStato()` in `Automa.java`
2. Completa `nextStato()` in `Automa.java`
3. Fai passare tutti i test in `AutomaTest.java`
4. **Scrivi un tuo test** (c'e' un TODO in `AutomaTest.java`)
5. Prova le regole **30, 90, 110, 184** con `stampaStato()` e confrontale con il risultato del sito

Fase 2: Esercizio sui thread

Vogliamo simulare **tutte le 256 regole** (da 0 a 255) per 1000 generazioni

Ma abbiamo solo **4 "vetrini"** (thread) per fare le simulazioni

Ogni risultato va scritto in `data/regole/<regola>.txt`

Producer-Consumer

1 thread **Produttore**: crea 256 oggetti `Automa`, li mette nella coda

4 thread **Vetrino**: prendono un automa dalla coda, li "evolvono", scrivono su file lo stato finale dopo 1000 generazioni

Segnalare la fine del produttore

Possiamo usare un `boolean` per segnalare che il produttore ha finito, per esempio aggiungiamo un attributo `haFinito` al produttore

- Il produttore setta l'attributo a `true` quando ha finito
- I vetrini controllano: se la lista è vuota e `haFinito` e' `true`, possono uscire dal ciclo e terminare

Fase 2: Cose da fare

1. Completa `Vetrino.java`: deve "consumare" un automa, lo fa evolvere, scrive il risultato sul file
2. Completa `Main.java`: crea produttore, crea vetrini, avvia e aspetta che i thread finiscano
3. Esegui e verifica che vengano creati **256 file** in `data/regole/`