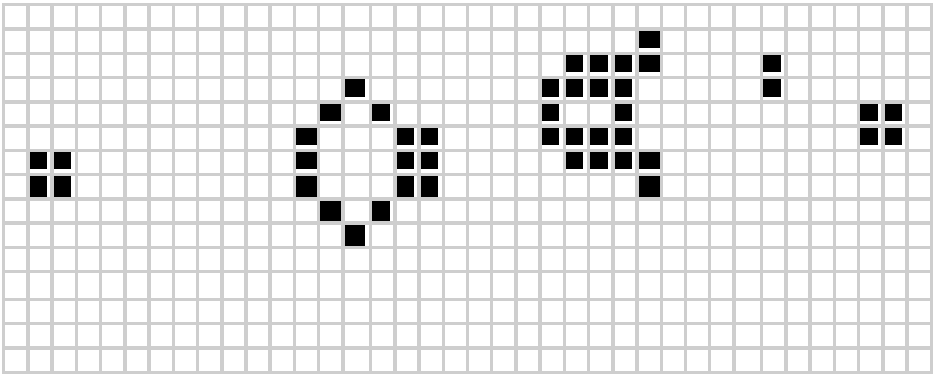
Effizienter Leben

Torsten Gerfertz
Christian Pernegger
Thomas Seidl

The Game of Life



- Regeln:
  - eine lebende Zelle mit weniger als 2 Nachbarn stirbt
  - eine lebende Zelle mit mehr als 3 Nachbarn stirbt
  - eine lebende Zelle mit 2 oder 3 Nachbarn lebt weiter
  - eine tote Zelle mit 3 Nachbarn erwacht zum Leben

Bildnachweis:
„Gosper gun“, *Eric Weisstein's treasure trove.*
http://www.ericweisstein.com/encyclopedias/life/GosperGun.html

Was bisher geschah ...

- *lebende* Zellen werden in Liste gespeichert
- neue Liste für jede Generation
  - indem für jede lebende Zelle alle umliegenden geprüft werden
  - d. h. eine Zelle wird mehrmals auf `alive()` abgefragt
  - `alive()`: Suche auf einfach verketteter Liste, O(n)
- ⇒ 13 Billionen Zyklen / 90 Minuten (auf der g0) für 3000 Generationen

Schritt 1: Algorithmus (1)

- Grundidee beibehalten
- Suche nach Zellen in Listen beschleunigen
  - Lokalität erzeugen und ausnutzen durch sortieren
  - Kriterium: xy := x + y
  - Zelle darf nur einmal vorkommen ⇒ sekundäres Kriterium: x
    - ⇒ Position einer Zelle innerhalb der Liste eindeutig
    - ⇒ Vorhandensein beim Einfügen leicht überprüfbar

Schritt 1: Algorithmus (2)

- für jede Zelle ergeben sich folgende xy-offsets für die umliegenden 24 Zellen:

-4	-3	-2	-1	0
-3	-2	-1	0	1
-2	-1	X	1	2
-1	0	1	2	3
0	1	2	3	4

  - ... direkter Nachbar (8)
  - ... indirekter Nachbar (16)

  (Ursprung links oben!)

- ⇒ nur Zellen mit |offset| ≤ 4 betrachten
  - Liste doppelt verketten und von X ausgehen

Schritt 1: Algorithmus (3)

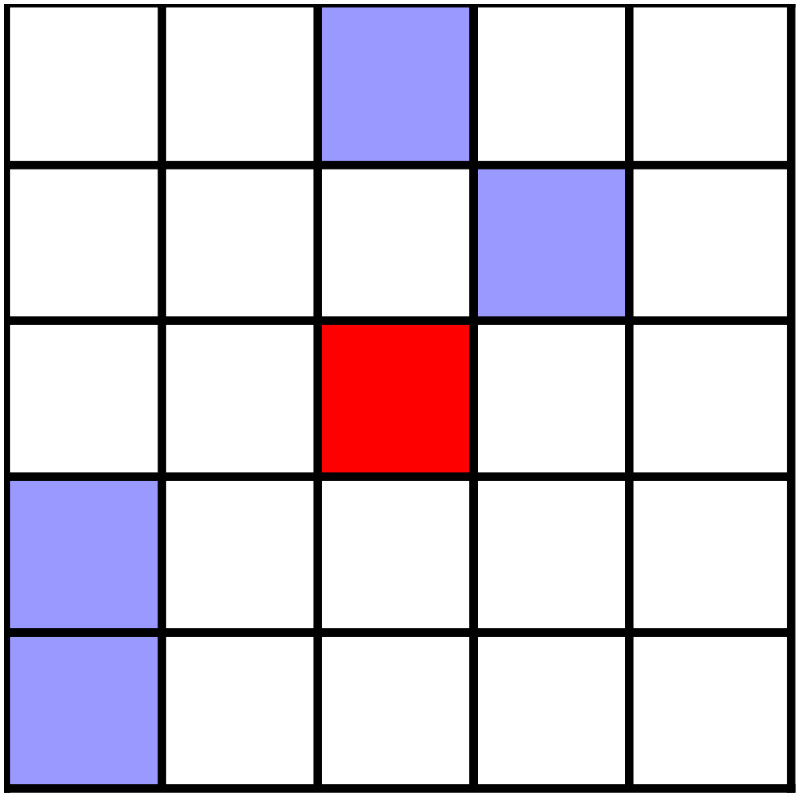
- für alle 24 Nachbarn lebender Zellen:
  - # der Nachbarn ermitteln und speichern
- für die 8 direkten Nachbarn
  - wird auferstehen → zwischenspeichern
- für die Zelle selbst
  - wird sterben → als tot markieren
    - neues struct-Feld „`alive`“
- ganz zum Schluß:
  - alle zwischengespeicherten Zellen → einfügen
  - alle als tot markierten Zellen → löschen

Schritt 1: Algorithmus (4)

- ⇒ Datenstruktur:

```
typedef struct celllist {
  long x;
  long y;
  long xy; /* x + y */
  char alive;
  struct celllist *next;
  struct celllist *prev;
};
```

Schritt 1: Algorithmus (Beispiel)



- ... derzeitige Zelle
- ... lebende Zelle

Schritt 1: Algorithmus (Beispiel)

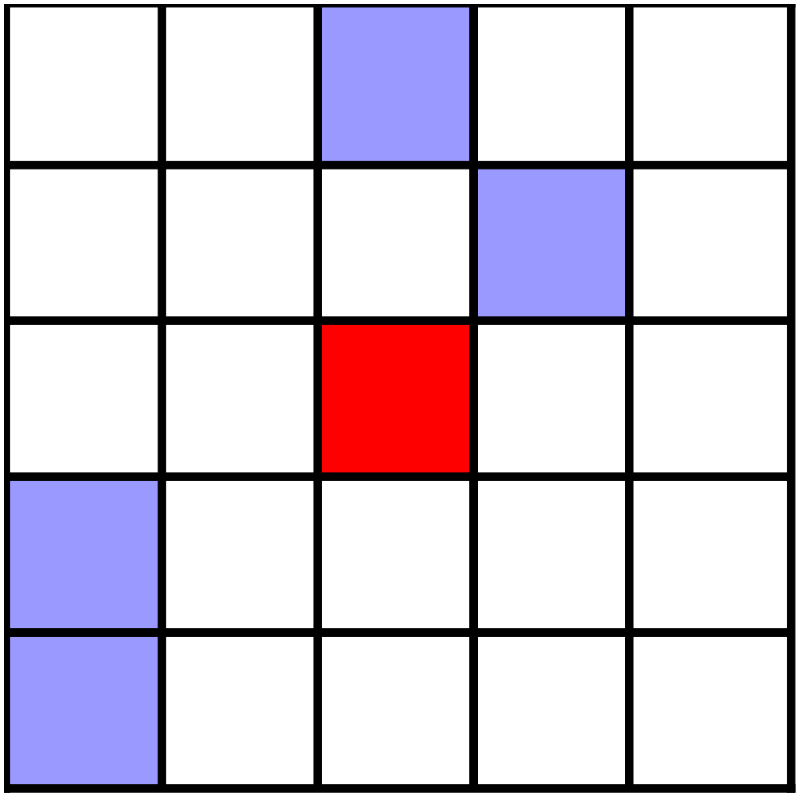
-4	-3	-2	-1	0
-3	-2	-1	0	1
-2	-1	X	1	2
-1	0	1	2	3
0	1	2	3	4

- ... derzeitige Zelle
- ... lebende Zelle

zur Erinnerung:

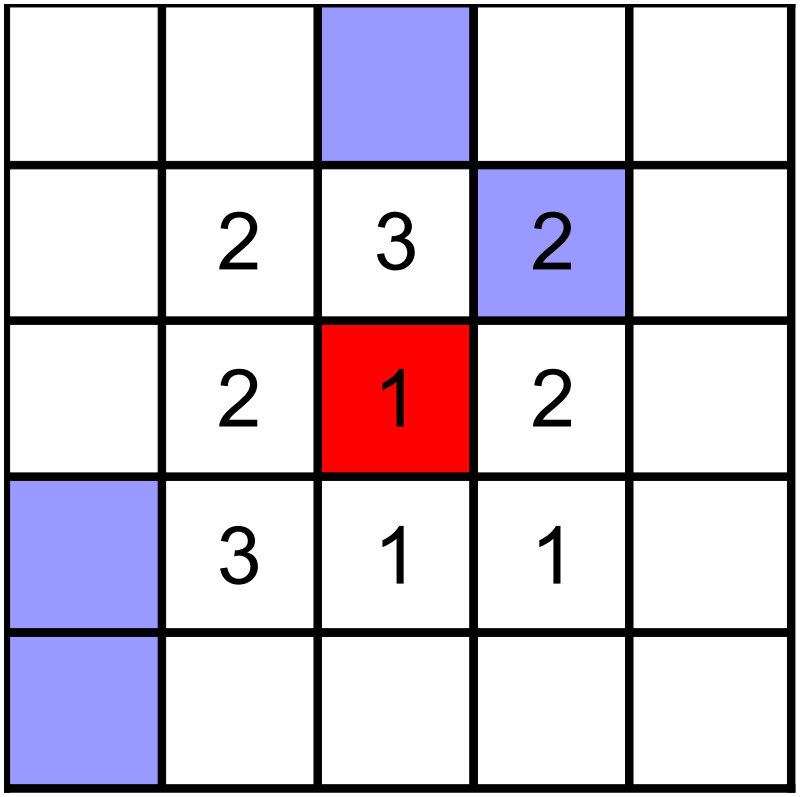
xy-offsets

Schritt 1: Algorithmus (Beispiel)



- ... derzeitige Zelle
- ... lebende Zelle

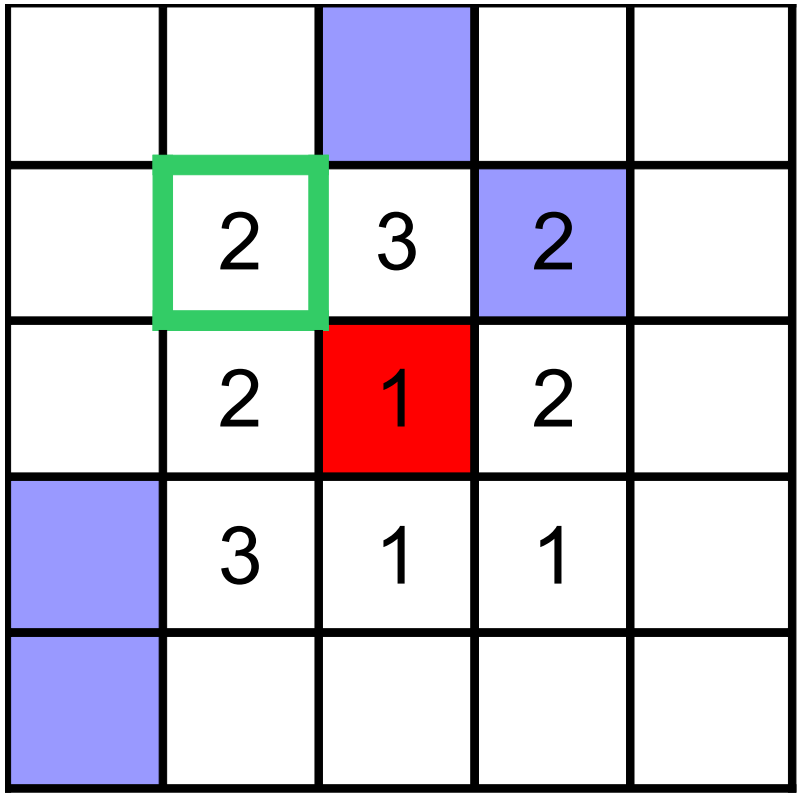
Schritt 1: Algorithmus (Beispiel)



- ... derzeitige Zelle
- ... lebende Zelle

Anzahl der Nachbarn

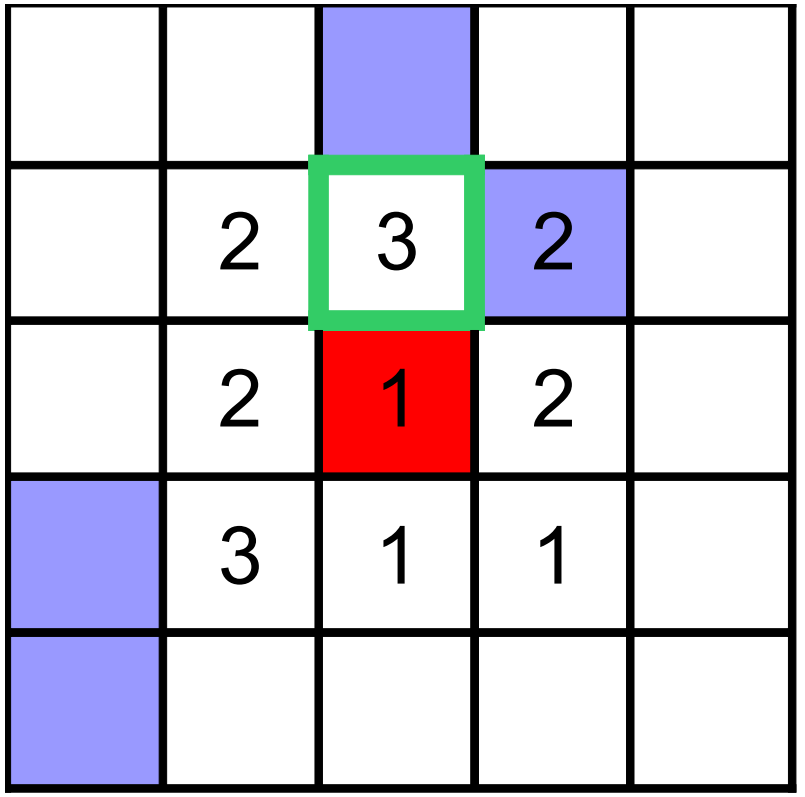
Schritt 1: Algorithmus (Beispiel)



- ... derzeitige Zelle
- ... lebende Zelle
- ... betrachtete Zelle

⇒ nichts tun

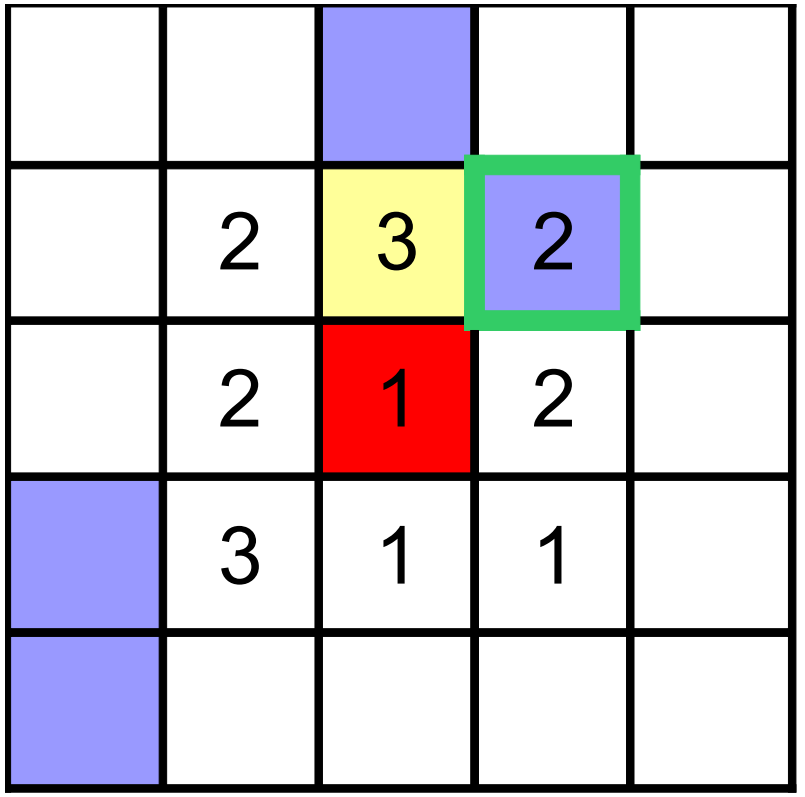
Schritt 1: Algorithmus (Beispiel)



- ... derzeitige Zelle
- ... lebende Zelle
- ... betrachtete Zelle

⇒ wird auferstehen → in temporäre Liste aufnehmen

Schritt 1: Algorithmus (Beispiel)

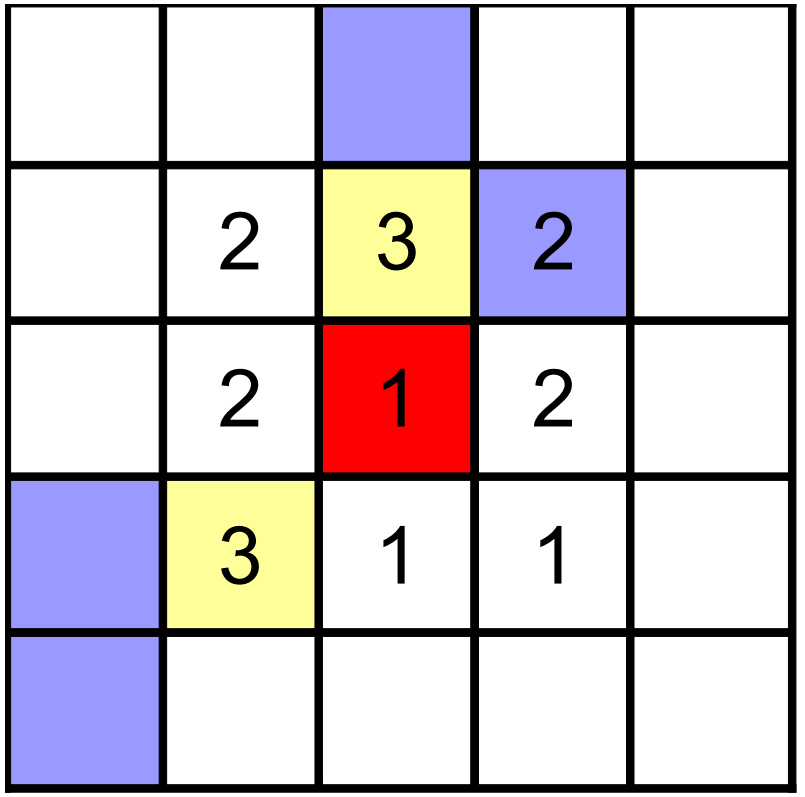


- ... derzeitige Zelle
- ... lebende Zelle
- ... betrachtete Zelle
- ... auferst. Zelle

⇒ schon am Leben → kommt selber dran → wird jetzt ignoriert

...

Schritt 1: Algorithmus (Beispiel)

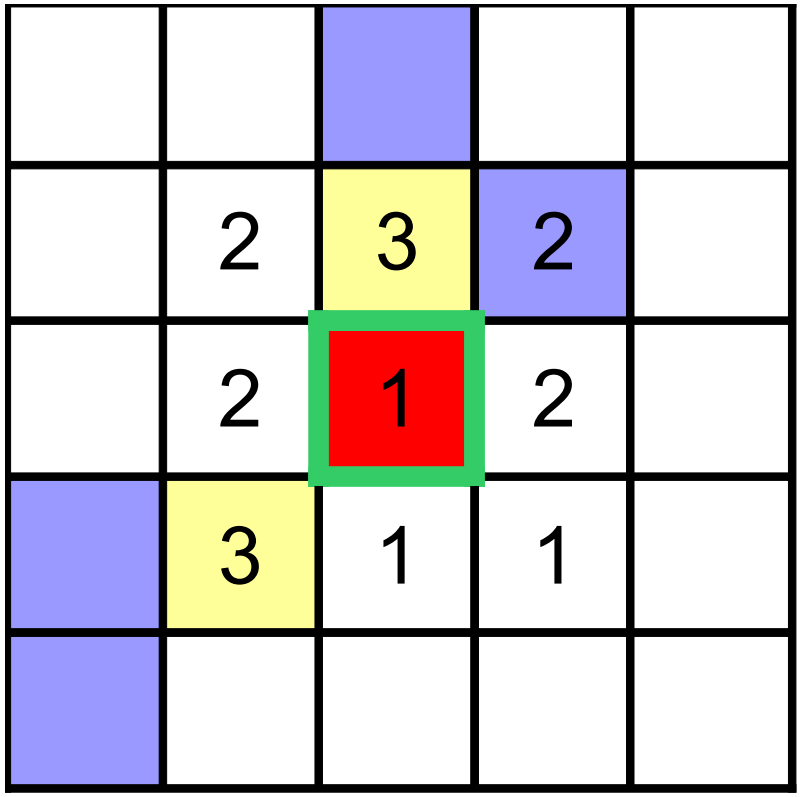


- ... derzeitige Zelle
- ... lebende Zelle
- ... betrachtete Zelle
- ... auferst. Zelle

⇒ 2 Zellen werden hinzugefügt

Zwischenergebnis

Schritt 1: Algorithmus (Beispiel)



- ... derzeitige Zelle
- ... lebende Zelle
- ... betrachtete Zelle
- ... auferst. Zelle

⇒ 1 lebender Nachbar
→ Zelle stirbt
→ `alive = 0`

überlebt

derzeitige Zelle?

Schritt 1: Messung

- Benchmark (wieder 3000 Generationen):

```
    32.052.473      Real usecs
85.259.574.120      Real cycles
    32.051.796      Proc usecs
85.257.774.030      Proc cycles
       308.950      I/O cycles
33.015.668.344      PAPI_TOT_INS
   143.573.374      PAPI_BR_MSP
```

- ⇒ ca. um Faktor 157 schneller

Schritt 2: Algorithmus (5)

-4	-3	-2	-1	0
-3	-2	-1	0	1
-2	-1	X	1	2
-1	0	1	2	3
0	1	2	3	4

- umliegende Zellen werden mehrfach überprüft
  - 2 Zellen kann man auslassen
  - wählt man offset(X) ± 2, kann man Felder mit |offset| = 4 ganz außer Acht lassen

Schritt 2: Messung

- Benchmark (wieder 3000 Generationen):

```
    24.535.286   Real usecs
65.263.856.300   Real cycles
    24.534.755   Proc usecs
65.262.446.150   Proc cycles
       311.920   I/O cycles
25.234.572.835   PAPI_TOT_INS
   130.138.326   PAPI_BR_MSP
```

- ⇒ ca. um Faktor 206 schneller (als Original)
  unerwartet große Verbesserung (31%)

Schritt 3: Implementierung (1)

- Profiling mit gprof:

```
% time
90.11 addCell
 9.36 oneGeneration
 0.45 cleanCells
 0.09 newCell
 0.04 addPrevious
 0.04 readLife
```

- ⇒ Konzentration auf
  - `addCell()`

Schritt 3: Implementierung (2)

- `addCell()`
  - zuerst: Liste wird von Anfang an bis zur passenden Position durchgegangen
  - Idee: Lokalität auch beim Einfügen ausnutzen
  - $\Rightarrow$ `addCell()` gibt gerade eingefügte Zelle zurück

- $\Rightarrow$ deutlich verkürzte Suchzeit

Schritt 3: Messung

- Benchmark (wieder 3000 Generationen):

```
      4.339.295   Real usecs
11.542.523.050   Real cycles
      4.334.003   Proc usecs
11.528.445.620   Proc cycles
        299.890   I/O cycles
 8.553.186.945   PAPI_TOT_INS
   197.567.489   PAPI_BR_MSP
```

- $\Longrightarrow$ ca. um Faktor 1173 schneller (als Original)
- $\Longrightarrow$ ca. um Faktor 5,5 schneller (als alte Lösung)

Schritt 4: Implementierung (3)

- (Re-)Profiling mit gprof:

```
% time
90.67   (9.36) oneGeneration
 5.80  (90.11) addCell
 2.14   (0.45) cleanCells
 0.73   (0.09) newCell
 0.34   (0.04) addPrevious
 0.24   (0.01) addNext
```

- ⇒ addCell() jetzt etwa 150x schneller
- ⇒ Konzentration auf
  - `oneGeneration()`

Schritt 4: Implementierung (4)

- `oneGeneration()`
  - iteriert immer noch ab Listenanfang
  - `addCell()` liefert aber eine Zelle in der Mitte
  - $\Rightarrow$ in der Mitte der Liste anfangen

- analog für `cleanCells()` und `writeLife()`!
- Ergebnis: nur 1,3% schneller
  - $\Rightarrow$ bloßes Iterieren über alle Zellen (ohne Tests) hat wenig Einfluß

Endergebnis

- Original:

```
13.466.543.701.890    Proc cycles
 7.524.249.259.736    PAPI_TOT_INS
     3.968.070.045    PAPI_BR_MSP
```

- Optimiertes Programm:

```
    11.057.721.840    Proc cycles
     8.653.404.354    PAPI_TOT_INS
       196.890.314    PAPI_BR_MSP
```

- $\Longrightarrow$ ca. um Faktor 1219 schneller (als Original)