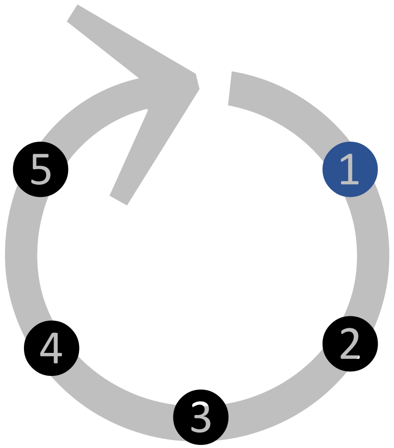
Денис Гроцев
degros

«Невозможный» параллельный алгоритм

1 Работа склада
2 Магические камни
3 Числа Фибоначчи
4 Среднее арифметическое
5 Неотрицательная сумма



1 Работа склада

$\Sigma 0$ ↑230 $\Sigma 230$

- Входной баланс
- Текущий баланс
- Приход
- Расход

1 Работа склада

Σ0 ↑230 Σ230 ↓100 Σ130 ↓100 Σ30

```
balance := 0;
for change in changes {
    balance += change;
}
```

- Входной баланс
- Текущий баланс
- Приход
- Расход

1 Работа склада

Σ0 ↑230 Σ230 ↓100 Σ130 ↓100 Σ30

```
balance := 0;
for change in changes {
    balance += change;
}
```

```
select   sum(change)
    over (order by time)
        as balance
from changes
```

- Входной баланс
- Текущий баланс
- Приход
- Расход

1 Работа склада

Σ0 ↑230 Σ230 ↓100 Σ130 ↓100 Σ30

💡 Haskell компактен

```
balance := 0;
for change in changes {
    balance += change;
}
```

```
select  sum(change)
    over (order by time)
        as balance
from changes
```

```
foldl' (+) 0
```

1 Работа склада

Σ0 ↑230 Σ230 ↓100 Σ130 ↓100 Σ30

💡 Haskell компактен

```
balance := 0;
for change in changes {
    balance += change;
}
```

```
select    sum(change)
    over (order by time)
        as balance
from changes
```

```
foldl' (+) 0
```

```
[-1, 2]
```

1 Работа склада

Σ0 ↑230 Σ230 ↓100 Σ130 ↓100 Σ30

💡 Haskell компактен

```
balance := 0;
for change in changes {
    balance += change;
}
```

```
select    sum(change)
    over (order by time)
        as balance
from changes
```

```
foldl' (+) 0
```

```
[-1, 2]  ~  [0, -1, 2]
```

1 Работа склада

Σ0 ↑230 Σ230 ↓100 Σ130 ↓100 Σ30

💡 Haskell компактен

```
balance := 0;
for change in changes {
    balance += change;
}
```

```
select   sum(change)
    over (order by time)
        as balance
from changes
```

```
foldl' (+) 0
```

[-1, 2] ~ [0, -1, 2] ~ 0 + -1 + 2

1 Работа склада

Σ0 ↑230 Σ230 ↓100 Σ130 ↓100 Σ30

💡 Haskell компактен

```
balance := 0;
for change in changes {
    balance += change;
}
```

```
select    sum(change)
    over (order by time)
        as balance
from changes
```

```
foldl' (+) 0
```

[-1, 2] ~ [0, -1, 2] ~ (0 + -1) + 2

1 Работа склада

Σ0 ↑230 Σ230 ↓100 Σ130 ↓100 Σ30

- Параллельные вычисления
- Горизонтальная масштабируемость
- map reduce

```
balance := 0;
for change in changes {
    balance += change;
}
```

```
select    sum(change)
    over (order by time)
        as balance
from changes
```

• • • • • • • • •

a b c

```
foldl' (+) 0
```

1 Работа склада

Σ0 ↑230 Σ230 ↓100 Σ130 ↓100 Σ30

```
balance := 0;
for change in changes {
    balance += change;
}
```

```
select    sum(change)
    over (order by time)
        as balance
from changes
```

```
foldl' (+) 0
```

• • • • • • • • •

a b c

(• • •)(• • •)(• • •)

- Параллельные вычисления
- Горизонтальная масштабируемость
- map reduce
- про скобки

1 Работа склада

Σ0 ↑230 Σ230 ↓100 Σ130 ↓100 Σ30

```
balance := 0;
for change in changes {
    balance += change;
}
```

```
select   sum(change)
    over (order by time)
        as balance
from changes
```

- Параллельные вычисления
- Горизонтальная масштабируемость
- map reduce
- про скобки
- ассоциативность + init = моноид

```
foldl' (+) 0
```

```
• • • • • • • • •
  a      b      c
(• • •)(• • •)(• • •)
(• •)(• • • • •)(• •)
```

=

1 Работа склада

Σ0 ↑230 Σ230 ↓100 Σ130 ↓100 Σ30 ↓100

```
balance := 0;
for change in changes {
    balance += change;
}
```

```
select    sum(change)
    over (order by time)
        as balance
from changes
```

```
foldl' (+) 0
```

1 Работа склада

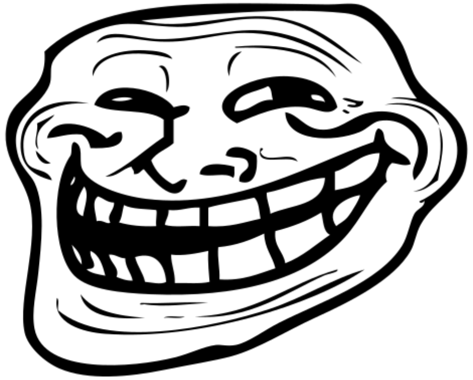
Σ0 ↑230 Σ230 ↓100 Σ130 ↓100 Σ30 ↓100 Σ0

```
balance := 0;
for change in changes {
    balance += change;
}
```

```
select    sum(change)
   over (order by time)
      as balance
from changes
```

```
foldl' (+) 0
```

- Счёт с неотрицательной суммой из реального мира.
- Не может быть отрицательных:
  - коробок на складе
  - овец в стаде
  - рублей в кошельке



1 Работа склада

Σ0 ↑230 Σ230 ↓100 Σ130 ↓100 Σ30 ↓100 Σ0

💡 Шаг итерации зависит от предыдущего.

```
balance := 0;
for change in changes {
    balance += change;
    if balance < 0 then
        balance := 0;
}
```

```
select  sum(change)
    over (order by time)
        as balance
from changes
```

```
foldl' (⊞) 0
x ⊞ y = max 0 (x + y)
```

1 Работа склада

Σ0 ↑230 Σ230 ↓100 Σ130 ↓100 Σ30 ↓100 Σ0

```
balance := 0;
for change in changes {
    balance += change;
    if balance < 0 then
        balance := 0;
}
```

```
select sum(change)
    over (order by time)
        as balance
from changes
```

```
foldl' (⊞) 0
x ⊞ y = max 0 (x + y)
```

- Шаг итерации зависит от предыдущего.
- SQL агрегат.

1 Работа склада

Σ0 ↑230 Σ230 ↓100 Σ130 ↓100 Σ30 ↓100 Σ0

```
balance := 0;
for change in changes {
    balance += change;
    if balance < 0 then
        balance := 0;
}
```

```
select sum(change)
    over (order by time)
        as balance
from changes
```

```
foldl' ⊞ 0
x ⊞ y = max 0 (x + y)
```

$(0 \boxplus -1) \boxplus 2 = 0 \boxplus 2 = 2$

$0 \boxplus (-1 \boxplus 2) = 0 \boxplus 1 = 1$

$\neq$

- 💡 Шаг итерации зависит от предыдущего.
- 💡 SQL агрегат.
- 💡 Неотрицательная сумма ⊞ неассоциативна.

1 Работа склада

Σ0 ↑230 Σ230 ↓100 Σ130 ↓100 Σ30 ↓100 Σ0

```
balance := 0;
for change in changes {
    balance += change;
    if balance < 0 then
        balance := 0;
}
```

```
select sum(change)
  over (order by time)
    as balance
from changes
```

Q&A

```
foldl' ⊞ 0
x ⊞ y = max 0 (x + y)
```

(0 ⊞ -1) ⊞ 2 = 0 ⊞ 2 = 2

0 ⊞ (-1 ⊞ 2) = 0 ⊞ 1 = 1

≠

- Шаг итерации зависит от предыдущего.
- SQL агрегат.
- Неотрицательная сумма ⊞ неассоциативна.
- Параллельно = ассоциативно
- Противоречие □

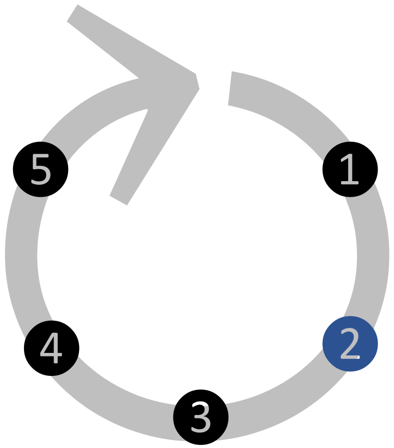
1 Работа склада

2 Магические камни

3 Числа Фибоначчи

4 Среднее арифметическое

5 Неотрицательная сумма



2 Магические камни

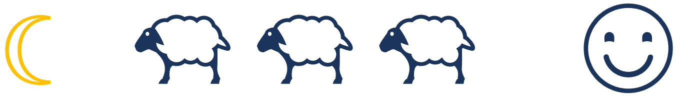
☺

10 000 лет назад

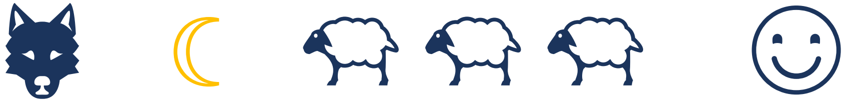
2 Магические камни

🐑🐑🐑 ☺

2 Магические камни



2 Магические камни



High load ⩾ 10 000 лет.

2 Магические камни

High load ⩾ 10 000 лет.

2 Магические камни

💡 High load ⩾ 10 000 лет.

2 Магические камни

High load ⩾ 10 000 лет.

2 Магические камни

💡 High load ⩾ 10 000 лет.

2 Магические камни

💡 High load ⩾ 10 000 лет.

2 Магические камни

High load ⩾ 10 000 лет.

2 Магические камни

High load ⩾ 10 000 лет.

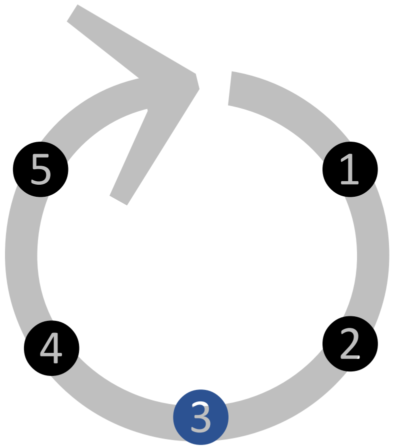
2 Магические камни

- High load ⩾ 10 000 лет.
- Закон Кларка: любая достаточно развитая технология неотличима от магии.

2 Магические камни

- High load ⩾ 10 000 лет.
- Закон Кларка: любая достаточно развитая технология неотличима от магии.
- Магия математики.

1 Работа склада
2 Магические камни
3 Числа Фибоначчи
4 Среднее арифметическое
5 Неотрицательная сумма



3 Числа Фибоначчи

0, 1

3 Числа Фибоначчи

0, 1, 1, 2, 3, 5, … $F_{n+1} = F_n + F_{n-1}$

3 Числа Фибоначчи

0, 1, 1, 2, 3, 5, … $F_{n+1} = F_n + F_{n-1}$ O(ln n)

3 Числа Фибоначчи

0, 1, 1, 2, 3, 5, … $F_{n+1} = F_n + F_{n-1}$ O(ln n)

$(F_{n-1} \quad F_n)$

Состояние исходного алгоритма – пара чисел.

3 Числа Фибоначчи

0, 1, 1, 2, 3, 5, … $F_{n+1} = F_n + F_{n-1}$ O(ln n)

$(F_{n-1} \quad F_n)$ $A = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}$ копия текущего
предыдущий и текущий

$$(0 \quad 1) \cdot \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} = (1 \quad 1)$$

- Состояние исходного алгоритма – пара чисел.
- Переход между состояниями линейный (матрица).

3 Числа Фибоначчи

0, 1, 1, 2, 3, 5, … $F_{n+1} = F_n + F_{n-1}$ O(ln n)

$(F_{n-1} \quad F_n)$ $A = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}$ копия текущего предыдущий и текущий

$$(0 \quad 1) \cdot \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} = (1 \quad 1)$$

$$(1 \quad 1) \cdot \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} = (1 \quad 2)$$

$$(1 \quad 2) \cdot \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} = (2 \quad 3)$$

$$(2 \quad 3) \cdot \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} = (3 \quad 5)$$

…

$$(F_{n-1} \quad F_n) \cdot \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} = (F_n \quad F_{n+1})$$

- Состояние исходного алгоритма – пара чисел.
- Переход между состояниями линейный (матрица).

3 Числа Фибоначчи

$0, 1, 1, 2, 3, 5, \ldots\ F_{n+1} = F_n + F_{n-1}$ O(ln n)

$(F_{n-1} \quad F_n)$ $\quad A = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}$

копия текущего
предыдущий и текущий

$$(0 \quad 1) \cdot \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} = (1 \quad 1)$$

$$(1 \quad 1) \cdot \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} = (1 \quad 2)$$

$$(1 \quad 2) \cdot \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} = (2 \quad 3)$$

$$(2 \quad 3) \cdot \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} = (3 \quad 5)$$

…

$$(F_{n-1} \quad F_n) \cdot \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} = (F_n \quad F_{n+1}) = (0 \quad 1) \cdot A^n$$

- Состояние исходного алгоритма – пара чисел.
- Переход между состояниями линейный (матрица).
- Линейная композиция матриц тоже матрица (замкнутость).

3 Числа Фибоначчи

0, 1, 1, 2, 3, 5, … $F_{n+1} = F_n + F_{n-1}$ O(ln n)

$\begin{pmatrix} F_{n-1} & F_n \end{pmatrix}$ $A = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}$

копия текущего
предыдущий и текущий

$$\begin{pmatrix} 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 1 \end{pmatrix}$$

$$\begin{pmatrix} 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 2 \end{pmatrix}$$

$$\begin{pmatrix} 1 & 2 \end{pmatrix} \cdot \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} = \begin{pmatrix} 2 & 3 \end{pmatrix}$$

$$\begin{pmatrix} 2 & 3 \end{pmatrix} \cdot \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} = \begin{pmatrix} 3 & 5 \end{pmatrix}$$

(•)(• •)(• • • •)

…

$$\begin{pmatrix} F_{n-1} & F_n \end{pmatrix} \cdot \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} = \begin{pmatrix} F_n & F_{n+1} \end{pmatrix} = \begin{pmatrix} 0 & 1 \end{pmatrix} \cdot A^{n}$$

$A^2, A^4, A^8, \dots A^{2\wedge k \leqslant n}$ $\quad \prod_{k=1}^{2\wedge k \leqslant n} \mathrm{bit}(k, n) \cdot A^{2\wedge k}$

- Состояние исходного алгоритма – пара чисел.
- Переход между состояниями линейный (матрица).
- Линейная композиция матриц тоже матрица (замкнутость).
- Ассоциативная композиция себя 2^n раз считается за O(ln(n)).

3 Числа Фибоначчи

0, 1, 1, 2, 3, 5, … $F_{n+1} = F_n + F_{n-1}$ O(ln n)

$(F_{n-1} \quad F_n)$ $A = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}$ копия текущего
предыдущий и текущий

$$(0 \quad 1) \cdot \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} = (1 \quad 1)$$

$$(1 \quad 1) \cdot \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} = (1 \quad 2)$$

$$(1 \quad 2) \cdot \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} = (2 \quad 3)$$

$$(2 \quad 3) \cdot \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} = (3 \quad 5)$$

…

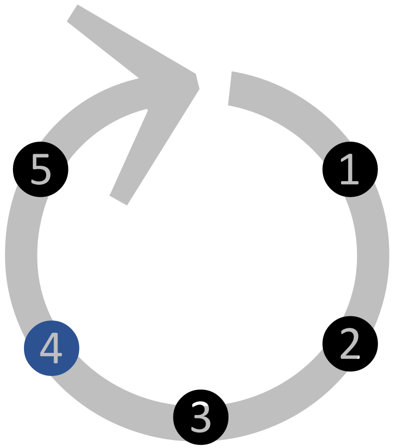
$$(\bullet)(\bullet \quad \bullet)(\bullet \quad \bullet \quad \bullet \quad \bullet)$$

$$(F_{n-1} \quad F_n) \cdot \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} = (F_n \quad F_{n+1}) = (0 \quad 1) \cdot A^n$$

$A^2, A^4, A^8, \ldots A^{2\wedge k \leqslant n}$ $\prod_{k=1}^{2\wedge k \leqslant n} \mathrm{bit}(k, n) \cdot A^{2\wedge k}$

- Состояние исходного алгоритма – пара чисел.
- Переход между состояниями линейный (матрица).
- Линейная композиция матриц тоже матрица (замкнутость).
- Ассоциативная композиция себя 2^n раз считается за O(ln(n)).
- В матрице A^n конечный результат закодирован:
4 числа распаковываются в 1.
- Дополнительная информация, которую забываем, нужна чтобы сделать вычисление ассоциативным.

1 Работа склада
2 Магические камни
3 Числа Фибоначчи
4 Среднее арифметическое
5 Неотрицательная сумма



4 Среднее арифметическое

$a \oplus b = (a + b) / 2$

4 Среднее арифметическое

$a \oplus b = (a + b) / 2$

$$(4 \oplus 8) \oplus 12 = 6 \oplus 12 = 9$$
$$4 \oplus (8 \oplus 12) = 4 \oplus 10 = 7$$
$$\neq$$

💡 Среднее ⊕ при попарном подсчёте неассоциативно.

4 Среднее арифметическое

$a \oplus b = (a + b) / 2$

$$
\begin{aligned}
(4 \oplus 8) \oplus 12 &= 6 \oplus 12 = 9 \\
4 \oplus (8 \oplus 12) &= 4 \oplus 10 = 7
\end{aligned}
\quad \neq
$$

```
data Avg n = Avg { count :: Integer, total :: n }
```

💡 Среднее ⊕ при попарном подсчёте неассоциативно.
💡 Число оборачивается в ассоциативный моноид с дополнительной информацией о количестве штук.

4 Среднее арифметическое

a ⊕ b = (a + b) / 2

(4 ⊕ 8) ⊕ 12 = 6 ⊕ 12 = 9
4 ⊕ (8 ⊕ 12) = 4 ⊕ 10 = 7 ≠

```
data Avg n = Avg { count :: Integer, total :: n }

instance (Num n, Ord n) ⇒ Semigroup (Avg n) where
    a <> b = Avg (count a + count b) (total a + total b)

instance (Num n, Ord n) ⇒ Monoid (Avg n) where
    mempty = Avg 0 0
```

💡 Среднее ⊕ при попарном подсчёте неассоциативно.
💡 Число оборачивается в ассоциативный моноид с дополнительной информацией о количестве штук.

4 Среднее арифметическое

a ⊕ b = (a + b) / 2

(4 ⊕ 8) ⊕ 12 = 6 ⊕ 12 = 9 ≠
4 ⊕ (8 ⊕ 12) = 4 ⊕ 10 = 7

```
data Avg n = Avg { count :: Integer, total :: n }

instance (Num n, Ord n) ⇒ Semigroup (Avg n) where
   a <> b = Avg (count a + count b) (total a + total b)

instance (Num n, Ord n) ⇒ Monoid (Avg n) where
   mempty = Avg 0 0

unwrap a = total a / fromIntegral (count a)

avg = unwrap . mapReduce (Avg 1)
```

💡 Среднее ⊕ при попарном подсчёте неассоциативно.
💡 Число оборачивается в ассоциативный моноид с дополнительной информацией о количестве штук.
💡 Результат среднего распаковывается из моноида.

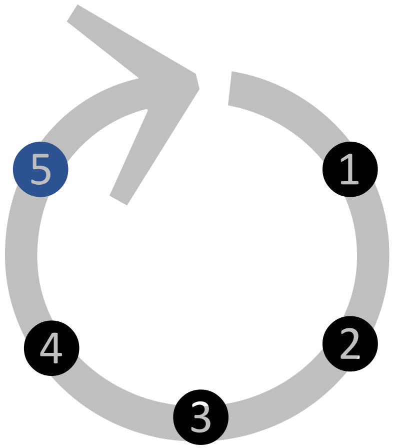
1 Работа склада

2 Магические камни

3 Числа Фибоначчи

4 Среднее арифметическое

5 Неотрицательная сумма



5.1 Неотрицательная сумма
Группа Гротендика

$(\mathbb{N}, +, 0) \rightsquigarrow (\mathbb{Z}, +, -, 0)$

grow - fall ~ (fall, grow)

-2 = (5, 3) = (15, 13) = (105, 103) = …

💡 Группа Гротендика представляет любое целое число в виде пары неотрицательных.

5.2 Неотрицательная сумма
Идея

[история]
[↓100 ↑50]

5.2 Неотрицательная сумма
Идея

```
  x       [ история         ]  y
Σ0        [↓100 Σ0      ↑50] Σ50
```

5.2 Неотрицательная сумма

Идея

x	[история			]	y
Σ0	[↓100	Σ0	↑50	]	Σ50
Σ230	[↓100	Σ130	↑50	]	Σ180

5.2 Неотрицательная сумма
Идея

x	[история		]	y		
Σ0	[↓100	Σ0	↑50]	Σ50	~	[↑50]
Σ230	[↓100	Σ130	↑50]	Σ180	~	[↓50]

5.2 Неотрицательная сумма
Идея

```
  x       [ история          ]   y
Σ0        [↓100 Σ0      ↑50] Σ50
Σ230      [↓100 Σ130 ↑50] Σ180
…
```

```
^ y
|   __/
|     (fall, grow)
+-----> x
```

y(x) = grow + max(0, x - fall)

- История изменений сжимается в функцию текущего остатка с двумя параметрами (fall, grow).
- fall показывает, сколько входного баланса сначала израсходуется.
- grow показывает, на сколько затем выходной баланс увеличится.

5.2 Неотрицательная сумма
Идея

x	[история]	y
Σ0	[↓100 Σ0 ↑50]	Σ50
Σ230	[↓100 Σ130 ↑50]	Σ180
…		

```
^ y
|  __/
|    (fall, grow)
+-----> x
```

[(↓100 ↑0) (↓0 ↑50)]

y(x) = grow + max(0, x - fall)

- 💡 История изменений сжимается в функцию текущего остатка с двумя параметрами (fall, grow).
- 💡 fall показывает, сколько входного баланса сначала израсходуется.
- 💡 grow показывает, на сколько затем выходной баланс увеличится.
- 💡 Рост предыдущего сперва компенсируется падением следующего.

5.2 Неотрицательная сумма

Идея

```
  x      [ история           ]  y
Σ0       [↓100 Σ0      ↑50] Σ50
Σ230     [↓100 Σ130 ↑50] Σ180
...
```

```
^ y
|   __/
|     (fall, grow)
+-----> x
```

[(↓100 ↑0)   (↓0 ↑50)]

$$y(x) = grow + \max(0, x - fall)$$

[↓100 (↑0   ↓0) ↑50]

- 💡 История изменений сжимается в функцию текущего остатка с двумя параметрами (fall, grow).
- 💡 fall показывает, сколько входного баланса сначала израсходуется.
- 💡 grow показывает, на сколько затем выходной баланс увеличится.
- 💡 Рост предыдущего сперва компенсируется падением следующего.

5.2 Неотрицательная сумма
Идея

```
  x      [ история         ]  y
Σ0       [↓100 Σ0      ↑50] Σ50
Σ230     [↓100 Σ130 ↑50] Σ180
...
```

```
^ y
|   __/
|     (fall, grow)
+-----> x
```

[(↓100 ↑0)   (↓0 ↑50)]

$$y(x) = grow + \max(0, x - fall)$$

[↓100 (↑0   ↓0) ↑50]

[(↓100   ↑50)]

(↓a↑) (↓b↑) = ↓a(↑ ↓)b↑

- История изменений сжимается в функцию текущего остатка с двумя параметрами (fall, grow).
- fall показывает, сколько входного баланса сначала израсходуется.
- grow показывает, на сколько затем выходной баланс увеличится.
- Рост предыдущего сперва компенсируется падением следующего.

5.3 Неотрицательная сумма
Реализация

```
data Gro n = Gro { fall :: n , grow :: n }
```

5.3 Неотрицательная сумма
Реализация

```
data Gro n = Gro { fall :: n , grow :: n }

wrap d = if d < 0 then Gro -d 0
                  else Gro  0 d
```

5.3 Неотрицательная сумма
Реализация

```
data Gro n = Gro { fall :: n , grow :: n }

wrap d = if d < 0 then Gro -d 0
                  else Gro  0 d

instance (Num n, Ord n) ⇒ Semigroup (Gro n) where
  a <> b = Gro (fall a + fall c)
               (grow b + grow c)
    where
      c = wrap (grow a - fall b)
```

5.3 Неотрицательная сумма
Реализация

```
data Gro n = Gro { fall :: n , grow :: n }

wrap d = if d < 0 then Gro -d 0
                  else Gro  0 d

instance (Num n, Ord n) ⇒ Semigroup (Gro n) where
  a <> b = Gro (fall a + fall c)
               (grow b + grow c)
   where
    c = wrap (grow a - fall b)

instance (Num n, Ord n) ⇒ Monoid (Gro n) where
  mempty = Gro 0 0
```

5.3 Неотрицательная сумма
Реализация

```
data Gro n = Gro { fall :: n , grow :: n }

wrap d = if d < 0 then Gro -d 0
                  else Gro  0 d

instance (Num n, Ord n) ⇒ Semigroup (Gro n) where
    a <> b = Gro (fall a + fall c)
                 (grow b + grow c)
      where
        c = wrap (grow a - fall b)

instance (Num n, Ord n) ⇒ Monoid (Gro n) where
    mempty = Gro 0 0

grosum :: (Num n, Ord n) ⇒ [n] → n
grosum = grow . mapReduce wrap
```

💡 Параллельная реализация неотрицательной суммы существует!

5.4 Неотрицательная сумма Верификация

```
prop_eq     xs = grosum xs == foldl' (⊞) 0 xs
prop_monoid = monoid (mempty :: Gro Int)
```

💡 Quickcheck тесты проверяют соответствие спецификации и свойства моноида.

5.4 Неотрицательная сумма
Верификация

```
prop_eq     xs = grosum xs == foldl' (⊞) 0 xs
prop_monoid = monoid (mempty :: Gro Int)
```

```
max(0, a + max(0, b)) = max(0, a) + max(0, b + min(0, a))
min (0, a + min(0, b)) = min (0, a) + min(0, b + max(0, a))
```

- 💡 Quickcheck тесты проверяют соответствие спецификации и свойства моноида.
- 💡 Формальное доказательство …

5.4 Неотрицательная сумма
Верификация

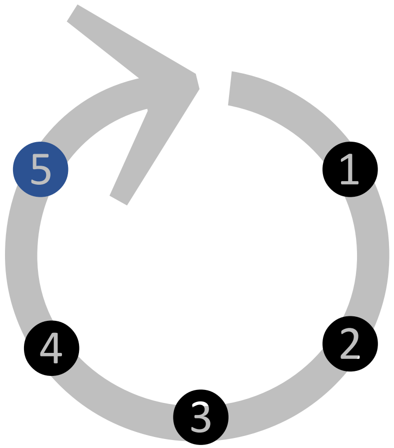
```
prop_eq     xs = grosum xs == foldl' (⊞) 0 xs
prop_monoid = monoid (mempty :: Gro Int)

max(0, a + max(0, b)) = max(0, a) + max(0, b + min(0, a))
min (0, a + min(0, b)) = min (0, a) + min(0, b + max(0, a))

                    ^b
max(0, a) = 0       |   max(0, a) = a
min(0, a) = a       |   min(0, a) = 0
max(0, b) = b       |   max(0, b) = b
------------------- + ------------------- >a
max(0, a) = 0       |   max(0, a) = a
min(0, a) = a       |   min(0, a) = 0
max(0, b) = 0       |   max(0, b) = 0
```

💡 Quickcheck тесты проверяют соответствие спецификации и свойства моноида.
💡 Формальное доказательство проверяется в 4 квадрантах плоскости (a,b).

1 Работа склада
2 Магические камни
3 Числа Фибоначчи
4 Среднее арифметическое
5 Неотрицательная сумма



Считаем параллельно – ищем моноид

- Начинаем с полного состояния последовательного алгоритма.

Фибоначчи	(F_{n-1}, F_n)	(0, 1)
Среднее	(count, total)	(0, 0)
grosum	(fall, grow)	(0, 0)

Считаем параллельно – ищем моноид

- Начинаем с полного состояния последовательного алгоритма.
- Строим функцию перехода между этими состояниями.

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

Среднее (count a + 1)
(total a + x)

grosum y(x) = grow + max(0, x-fall)

Считаем параллельно – ищем моноид

- Начинаем с полного состояния последовательного алгоритма.
- Строим функцию перехода между этими состояниями.
- Композиция функций ассоциативна.

$(f \circ g) \circ h = f \circ (g \circ h)$

Считаем параллельно – ищем моноид

- Начинаем с полного состояния последовательного алгоритма.
- Строим функцию перехода между этими состояниями.
- Композиция функций ассоциативна.
- Пытаемся выразить композицию в **компактной** форме, где информации меньше, чем в полной истории.

Фибоначчи

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

Среднее

(count a + count b)
(total a + total b)

grosum

(fall a + fall c)
(grow b + grow c)
c = wrap (grow a-fall b)

Считаем параллельно – ищем моноид

- Начинаем с полного состояния последовательного алгоритма.
- Строим функцию перехода между этими состояниями.
- Композиция функций ассоциативна.
- Пытаемся выразить композицию в **компактной** форме, где информации меньше, чем в полной истории.
- Profit!

```
select
  grosum(change) over (order by time)
        as balance
from changes
```

```
type GrosumImpl as object
( grow number
, fall number
, static function ODCIAggregateInitialize(sctx in out GrosumImpl) return number
, member function ODCIAggregateIterate(self in out GrosumImpl, value in number) return number
, member function ODCIAggregateTerminate(self in GrosumImpl, returnValue out number, flags in number) return number
, member function ODCIAggregateMerge(self in out GrosumImpl, ctx2 in GrosumImpl) return number
)

create or replace function WFM.grosum(input number) return number parallel_enable aggregate using GrosumImpl;
```

```
type body GrosumImpl is

static function ODCIAggregateInitialize(sctx in out GrosumImpl)
return number is
begin
   sctx := GrosumImpl(0, 0);
   return ODCIConst.Success;
end;

member function ODCIAggregateIterate(self in out GrosumImpl, value in number) return number is
begin
   self.grow := self.grow + value;
   if self.grow < 0 then
      self.fall := self.fall - self.grow;
      self.grow := 0;
   end if;
   return ODCIConst.Success;
end;

member function ODCIAggregateTerminate(self in GrosumImpl, returnValue out number, flags in number) return number is
begin
   returnValue := self.grow;
   return ODCIConst.Success;
end;

member function ODCIAggregateMerge(self in out GrosumImpl, ctx2 in GrosumImpl) return number is
begin
   if self.grow < ctx2.fall then
      self.fall := self.fall + ctx2.fall - self.grow;
      self.grow := ctx2.grow;
   else
      self.grow := self.grow + ctx2.grow - ctx2.fall;
   end if;
   return ODCIConst.Success;
end;

end;
```

```java
@RequiredArgsConstructor
private static final class Grosum {
    public final int fall, grow;

    public static final Grosum EMPTY = new Grosum(0, 0);

    public static Grosum wrap(int n) {
        return n >= 0 ? new Grosum(0, n) : new Grosum(-n, 0);
    }

    public static Grosum merge(Grosum a, Grosum b) {
        Grosum c = wrap(a.grow - b.fall);
        return new Grosum(a.fall + c.fall, b.grow + c.grow);
    }

    public static int grosum(IntStream stream) {
        return stream
                .mapToObj(Grosum::wrap)
                .reduce(Grosum.EMPTY, Grosum::merge)
                .grow;
    }
}
```

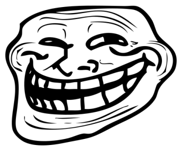
```java
System.out.println(Grosum.grosum(IntStream
        .range(1, 100)
        .parallel()
        .map(i -> (i * 137) % 199 - 99)
));
System.out.println(Grosum.grosum(IntStream.of(-1, 2)));
```

Считаем параллельно – ищем моноид

- Начинаем с полного состояния последовательного алгоритма.
- Строим функцию перехода между этими состояниями.
- Композиция функций ассоциативна.
- Пытаемся выразить композицию в **компактной** форме,
  где информации меньше, чем в полной истории.
- Profit!

```
select
  grosum(change) over (order by time)
          as balance
from changes
```

Q&A



degros