Five Lectures on CA

4. Sorting

Thomas Worsch

Department of Informatics
Karlsruhe Institute of Technology
http://liinwww.ira.uka.de/~thw/vl-hiroshima/

at Hiroshima University, January 2012

Outline

- Sorting in one-dimensional CA
  - Odd-even transposition sort
  - Knuth's 0-1 principle

- Sorting in two-dimensional CA
  - Sorting order
  - Shearsort
  - Algorithm by Schnorr and Shamir

- Sorting in one-dimensional CA
  - Odd-even transposition sort
  - Knuth's 0-1 principle
- Sorting in two-dimensional CA
  - Sorting order
  - Shearsort
  - Algorithm by Schnorr and Shamir

- Sorting in one-dimensional CA
  - Odd-even transposition sort
  - Knuth's 0-1 principle
- Sorting in two-dimensional CA
  - Sorting order
  - Shearsort
  - Algorithm by Schnorr and Shamir

Problem

Notation

for $x \in A$, $w \in A^*$ denote by

$N_x(w)$ the number of occurences of symbol x in w

Problem

Notation

for $x \in A$, $w \in A^*$ denote by

$N_x(w)$ the number of occurences of symbol x in w

Problem: sorting of bits

given: $A = \{0, 1\}$ $R = \mathbb{Z}$ $N = H_1$

wanted: CA with $Q \supseteq A \cup \{\#\}$ and f, such that each pattern $w \in A^+$ is transformed into $0^{N_0(w)}1^{N_1(w)}$

Algorithm

- $Q = A \cup \{\#\}$
- f given by

$\ell(-1)$	$\ell(0)$	$\ell(1)$	$f(\ell)$
1	0	$\times$	1
$\times$	1	0	0
otherwise:			
$\times$	s	$\times$	s

- Note: f is well defined

Example

#	1	1	0	1	1	1	0	1	0	#
#	1	0	1	1	1	0	1	0	1	#
#	0	1	1	1	0	1	0	1	1	#
#	0	1	1	0	1	0	1	1	1	#
#	0	1	0	1	0	1	1	1	1	#
#	0	0	1	0	1	1	1	1	1	#
#	0	0	0	1	1	1	1	1	1	#

Example 2

#	1	1	1	1	1	1	0	0	0	#
#	1	1	1	1	1	0	1	0	0	#
#	1	1	1	1	0	1	0	1	0	#
#	1	1	1	0	1	0	1	0	1	#
#	1	1	0	1	0	1	0	1	1	#
#	1	0	1	0	1	0	1	1	1	#
#	0	1	0	1	0	1	1	1	1	#
#	0	0	1	0	1	1	1	1	1	#
#	0	0	0	1	1	1	1	1	1	#

Problem: Sorting of "trits"

given: $A = \{0, 1, 2\}$ $R = \mathbb{Z}$ $N = H_1$

wanted: CA with $Q \supseteq A \cup \{\#\}$ and f, such that each pattern $w \in A^+$ is transformed into $0^{N_0(w)}1^{N_1(w)}2^{N_2(w)}$

Let's try ...

Problem: Sorting of "trits"

Problem:

$\ell(-1)$	$\ell(0)$	$\ell(1)$	$f(\ell)$
2	1	0	???

Problem: Sorting of "trits"

Problem:

$\ell(-1)$	$\ell(0)$	$\ell(1)$	$f(\ell)$
2	1	0	???

Solution even cells: look to the left and to the right alternately
odd cells: look to the right and to the left alternately

Algorithm (Odd-even transposition sort)

- arbitrary input alphabet A
- $Q = A \times \{\mathtt{L}, \mathtt{R}, \_\} \cup \{\#\}$ (identify $s \in A$ with $(s, \_)$)
- f given by the following table:

$\ell(-1)$	$\ell(0)$	$\ell(1)$	$f(\ell)$
real sorting:			
$(s, \mathtt{R})$	$(t, \mathtt{L})$	$\times$	$(\max(s,t), \mathtt{R})$
$\times$	$(s, \mathtt{R})$	$(t, \mathtt{L})$	$(\min(s,t), \mathtt{L})$
$\#$	$(t, \mathtt{L})$	$\times$	$(t, \mathtt{R})$
$\times$	$(s, \mathtt{R})$	$\#$	$(s, \mathtt{L})$
initialisation:			
$\#$	$(s, \_)$	$\times$	$(s, \mathtt{L})$
$(s, \mathtt{L})$	$(t, \_)$	$\times$	$(t, \mathtt{L})$
otherwise:			
$\times$	z	$\times$	z

Example

#	1	0	2	1	0	#

#	1	0	2	1	0	#
	L					

#	1	0	2	1	0	#
	R	L				

#	0	1	2	1	0	#
	L	R	L			

#	0	1	2	1	0	#
	R	L	R	L		

#	0	1	1	2	0	#
	L	R	L	R	L	

#	0	1	1	2	0	#
	L	R	L	R	L	

#	0	1	1	0	2	#
	R	L	R	L	R	

#	0	1	0	1	2	#
	L	R	L	R	L	

#	0	0	1	1	2	#
	R	L	R	L	R	

#	0	0	1	1	2	#
	L	R	L	R	L	

#	0	0	1	1	2	#
	R	L	R	L	R	

Lemma

Odd-even transposition sort is correct.

How does one prove that?

- Sorting in one-dimensional CA
  - Odd-even transposition sort
  - Knuth's 0-1 principle
- Sorting in two-dimensional CA
  - Sorting order
  - Shearsort
  - Algorithm by Schnorr and Shamir

Lemma (0-1 principle)

- If a sorting algorithm only uses "compare-and-swap-if-greater" operations

Lemma (0-1 principle)

- If a sorting algorithm only uses "compare-and-swap-if-greater" operations
- and if it is independent of the concrete values to be sorted which elements are compared and when,

Lemma (0-1 principle)

- If a sorting algorithm only uses "compare-and-swap-if-greater" operations
- and if it is independent of the concrete values to be sorted which elements are compared and when,

- then the algorithm works correctly for all input sequences

Lemma (0-1 principle)

- If a sorting algorithm only uses "compare-and-swap-if-greater" operations
- and if it is independent of the concrete values to be sorted which elements are compared and when,

- then the algorithm works correctly for all input sequences
- if and only if it works correctly for input sequences which are bits.

▸ correctness of shearsort

Proof (1)

Assume that $x_1, \ldots, x_n$ is an input sequence which is not sorted correctly.
Will show: There is also an input sequence $x_1^*, \ldots, x_n^*$ of bits which is not sorted correctly.

Proof (1)

Assume that $x_1, \ldots, x_n$ is an input sequence which is not sorted correctly.
Will show: There is also an input sequence $x_1^*, \ldots, x_n^*$ of bits which is not sorted correctly.

Assume:

correct order would be	$x_{\pi(1)}$	$\leq$	$x_{\pi(2)}$	$\leq$	$\cdots$	$\leq$	$x_{\pi(n)}$
the algorithm produces	$x_{\sigma(1)}$	,	$x_{\sigma(2)}$	,	$\cdots$	,	$x_{\sigma(n)}$

Proof (1)

Assume that $x_1, \ldots, x_n$ is an input sequence which is not sorted correctly.
Will show: There is also an input sequence $x_1^*, \ldots, x_n^*$ of bits which is not sorted correctly.

Assume:

correct order would be	$x_{\pi(1)}$	$\leq$	$x_{\pi(2)}$	$\leq$	$\cdots$	$\leq$	$x_{\pi(n)}$
the algorithm produces	$x_{\sigma(1)}$	,	$x_{\sigma(2)}$	,	$\cdots$	,	$x_{\sigma(n)}$

k: first wrong position

$$x_{\sigma(i)} = x_{\pi(i)} \quad \text{for } 1 \leq i < k$$
$$x_{\sigma(k)} > x_{\pi(k)}$$

r: where $x_{\pi(k)}$ ended up instead of k

$$x_{\sigma(r)} = x_{\pi(k)} \quad \text{where } r > k$$

Proof (2)

Define a bit sequence $x_1^*, \ldots, x_n^*$ as follows

$$x_i^* = \left[x_i > x_{\pi(k)}\right] \qquad \text{i. e.} \qquad x_i^* = \begin{cases} 1 & \text{if } x_i > x_{\pi(k)} \\ 0 & \text{if } x_i \leq x_{\pi(k)} \end{cases}$$

for example: $x^*_{\sigma(k)} = \left[x_{\sigma(k)} > x_{\pi(k)}\right] = 1$

$$x^*_{\sigma(r)} = \left[x_{\sigma(r)} > x_{\pi(k)}\right] = 0$$

Proof (2)

Define a bit sequence $x_1^*, \ldots, x_n^*$ as follows

$$x_i^* = \left[x_i > x_{\pi(k)}\right] \qquad \text{i. e.} \qquad x_i^* = \begin{cases} 1 & \text{if } x_i > x_{\pi(k)} \\ 0 & \text{if } x_i \leq x_{\pi(k)} \end{cases}$$

for example: $x^*_{\sigma(k)} = \left[x_{\sigma(k)} > x_{\pi(k)}\right] = 1$

$$x^*_{\sigma(r)} = \left[x_{\sigma(r)} > x_{\pi(k)}\right] = 0$$

Nun gilt:
$$\begin{aligned} x_i > x_j &\implies \left(x_j > x_{\pi(k)} \implies x_i > x_{\pi(k)}\right) \\ &\implies \left(x_j^* = 1 \implies x_i^* = 1\right) \\ &\implies x_i^* \geq x_j^* \end{aligned}$$

und analog $\quad x_i \leq x_j \implies x_i^* \leq x_j^*$.

Proof (3)

Hence for input sequence für die Eingabe $x_1^*, \ldots, x_n^*$ the algorithm produces the same result which one gets

Proof (3)

Hence for input sequence für die Eingabe $x_1^*, \ldots, x_n^*$ the algorithm produces the same result which one gets

if one makes the same swaps for the $x_1^*, \ldots, x_n^*$ bits,
which the algorithm makes for the $x_1, \ldots, x_n$.

The algorithm permutes the x_i^* according to σ.

Proof (3)

Hence for input sequence für die Eingabe $x_1^*, \ldots, x_n^*$ the algorithm produces the same result which one gets

if one makes the same swaps for the $x_1^*, \ldots, x_n^*$ bits,
which the algorithm makes for the $x_1, \ldots, x_n$.

The algorithm permutes the x_i^* according to σ.

When the algorithm stops, the result is

	$x^*_{\sigma(1)}$	$\cdots$	$x^*_{\sigma(k)}$	$\cdots$	$x^*_{\sigma(r)}$	$\cdots$	$x^*_{\sigma(n)}$
i.e.		$\cdots$	$[x_{\sigma(k)} > x_{\pi(k)}]$	$\cdots$	$[x_{\sigma(r)} > x_{\pi(k)}]$	$\cdots$	
i.e.		$\cdots$	1	$\cdots$	0	$\cdots$	

and that ist a *non sorted* sequence.

Correctness of odd-even transposition sort

- 0-1 principle is applicable
- consider a bit sequence with e ones.

Correctness of odd-even transposition sort

- 0-1 principle is applicable
- consider a bit sequence with e ones.
- look at the algorithm as sending R-signals from left to right:

Correctness of odd-even transposition sort

- 0-1 principle is applicable
- consider a bit sequence with e ones.
- look at the algorithm as sending R-signals from left to right:
  1. The first signal moves the rightmost 1 to its final position (arriving at the latest at time $2 + (n - 1) = n + 1$)

Correctness of odd-even transposition sort

- 0-1 principle is applicable
- consider a bit sequence with e ones.
- look at the algorithm as sending R-signals from left to right:
  1. The first signal moves the rightmost 1 to its final position (arriving at the latest at time $2 + (n - 1) = n + 1$)
  2. The second signal moves the second 1 (counted from the right) to its final position (arriving at the latest at time $4 + (n - 2) = n + 2$)

Correctness of odd-even transposition sort

- 0-1 principle is applicable
- consider a bit sequence with e ones.
- look at the algorithm as sending R-signals from left to right:
  1. The first signal moves the rightmost 1 to its final position (arriving at the latest at time $2 + (n - 1) = n + 1$)
  2. The second signal moves the second 1 (counted from the right) to its final position (arriving at the latest at time $4 + (n - 2) = n + 2$)

  ... and so on

Correctness of odd-even transposition sort

- 0-1 principle is applicable
- consider a bit sequence with e ones.
- look at the algorithm as sending R-signals from left to right:
  1. The first signal moves the rightmost 1 to its final position (arriving at the latest at time $2 + (n - 1) = n + 1$)
  2. The second signal moves the second 1 (counted from the right) to its final position (arriving at the latest at time $4 + (n - 2) = n + 2$)

  ... and so on

  e. The e-th signal moves the leftmost 1 to its final position (arriving at the latest at time $2e + (n - e) = n + e$)

Correctness of odd-even transposition sort

- 0-1 principle is applicable
- consider a bit sequence with e ones.
- look at the algorithm as sending R-signals from left to right:
  1. The first signal moves the rightmost 1 to its final position (arriving at the latest at time $2 + (n - 1) = n + 1$)
  2. The second signal moves the second 1 (counted from the right) to its final position (arriving at the latest at time $4 + (n - 2) = n + 2$)

  ... and so on

  e. The e-th signal moves the leftmost 1 to its final position (arriving at the latest at time $2e + (n - e) = n + e$)
- When all 1 have arrived at their final position the sequence is sorted.

- Sorting in one-dimensional CA
  - Odd-even transposition sort
  - Knuth's 0-1 principle

- Sorting in two-dimensional CA
  - Sorting order
  - Shearsort
  - Algorithm by Schnorr and Shamir

- Sorting in one-dimensional CA
  - Odd-even transposition sort
  - Knuth's 0-1 principle
- Sorting in two-dimensional CA
  - Sorting order
  - Shearsort
  - Algorithm by Schnorr and Shamir

What is two-dimensional sorting?

What is two-dimensional sorting?

this:

1	2	3	4	5
6	7	8	9	10
11	12	13	14	15
16	17	18	19	20
21	22	23	24	25

?

or this:

1	2	6	7	15
3	5	8	14	16
4	9	13	17	22
10	12	18	21	23
11	19	20	24	25

?

or this:

1	2	3	4	5
16	17	18	19	6
15	24	25	20	7
14	23	22	21	8
13	12	11	10	9

?

or this:

1	2	3	4	5
10	9	8	7	6
11	12	13	14	15
20	19	18	17	16
21	22	23	24	25

?

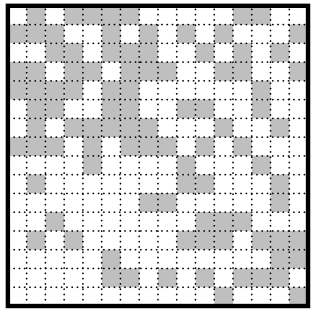
Problem definition

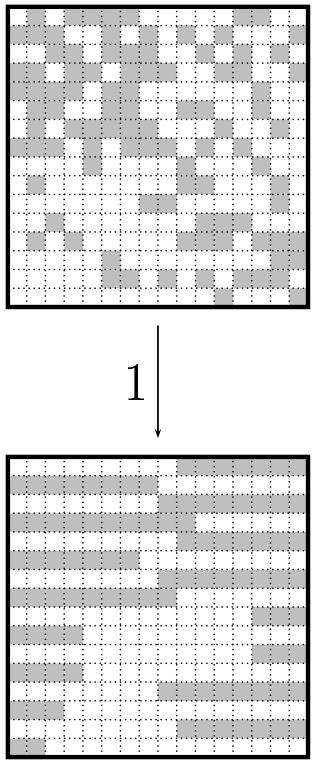
- A finite and totally ordered
- **Wanted:** two-dimensional CA transforming each
  pattern $e : \mathbf{N}_m \times \mathbf{N}_m \to A$ into
  sorted pattern $s : \mathbf{N}_m \times \mathbf{N}_m \to A$ where
  1. for all $a \in A$ the number of a elements is preserved: $N_a(e) = N_a(s)$.
  2. if $i \in \mathbf{N}_m$ is odd, then for $j \in \mathbf{N}_{m-1}$: $s(i,j) \leq s(i,j+1)$.
  3. if $i \in \mathbf{N}_m$ is even, then for $j \in \mathbf{N}_{m-1}$: $s(i,j) \geq s(i,j+1)$.
  4. if $i \in \mathbf{N}_{m-1}$ is odd, then $s(i,n) \leq s(i+1,n)$.
  5. if $i \in \mathbf{N}_{m-1}$ is even, then $s(i,1) \leq s(i+1,1)$.
- "sorting in serpentine form" (not sure about the correct English word)

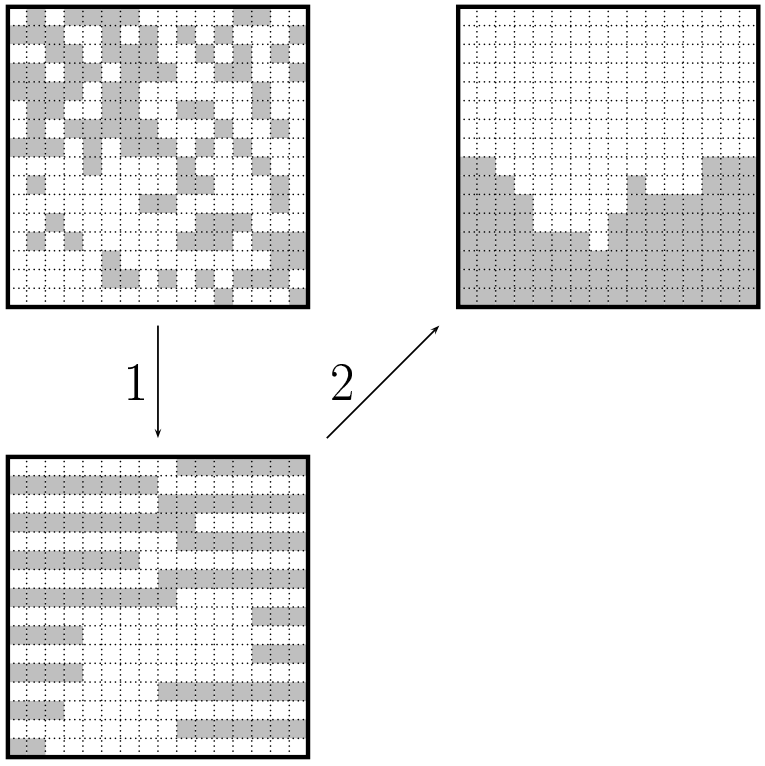
Any ideas?

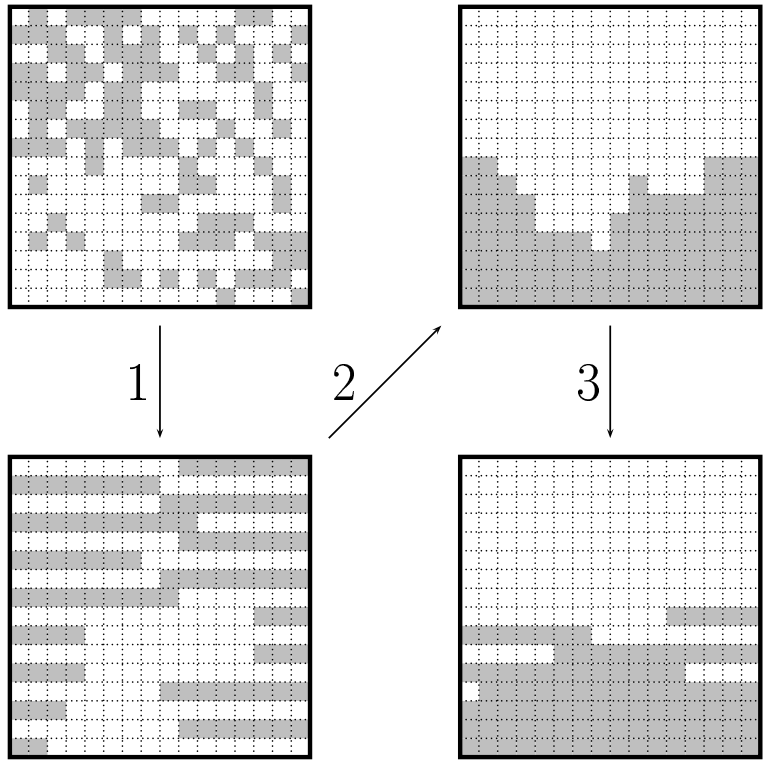
How could a CA achieve that sorting order?

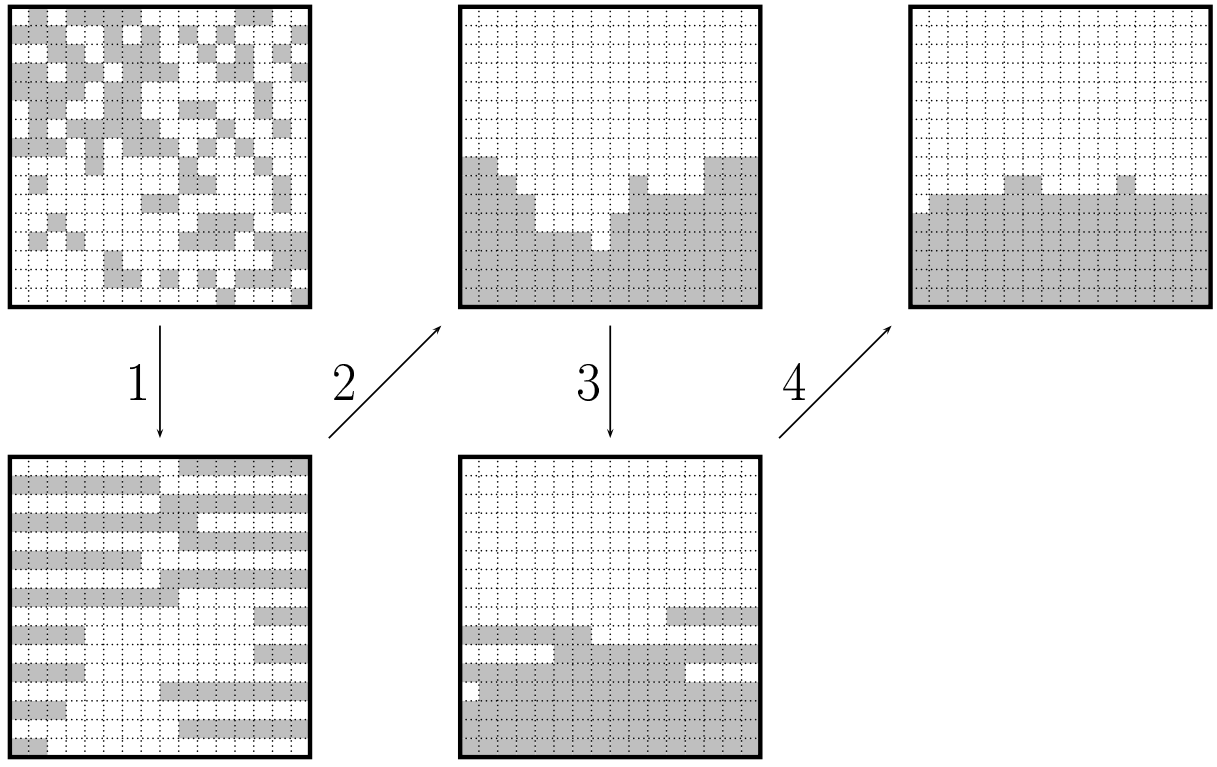
- Sorting in one-dimensional CA
  - Odd-even transposition sort
  - Knuth's 0-1 principle
- Sorting in two-dimensional CA
  - Sorting order
  - Shearsort
  - Algorithm by Schnorr and Shamir

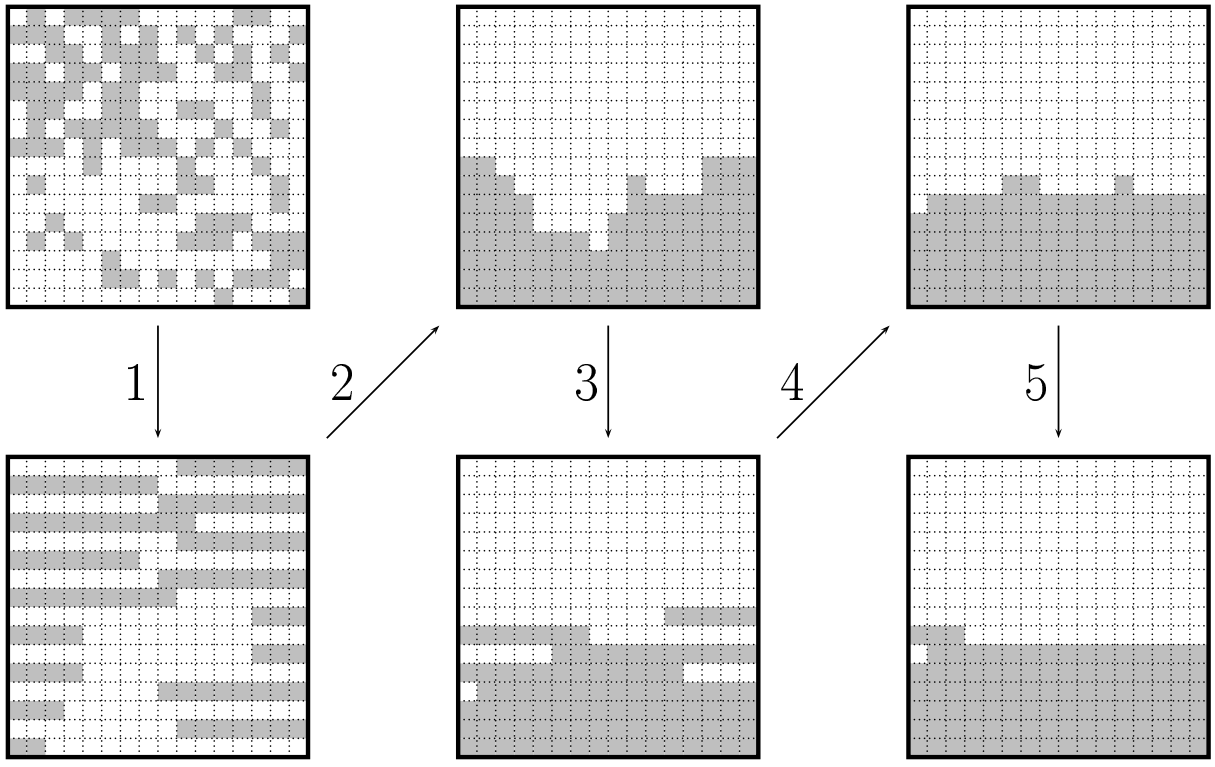


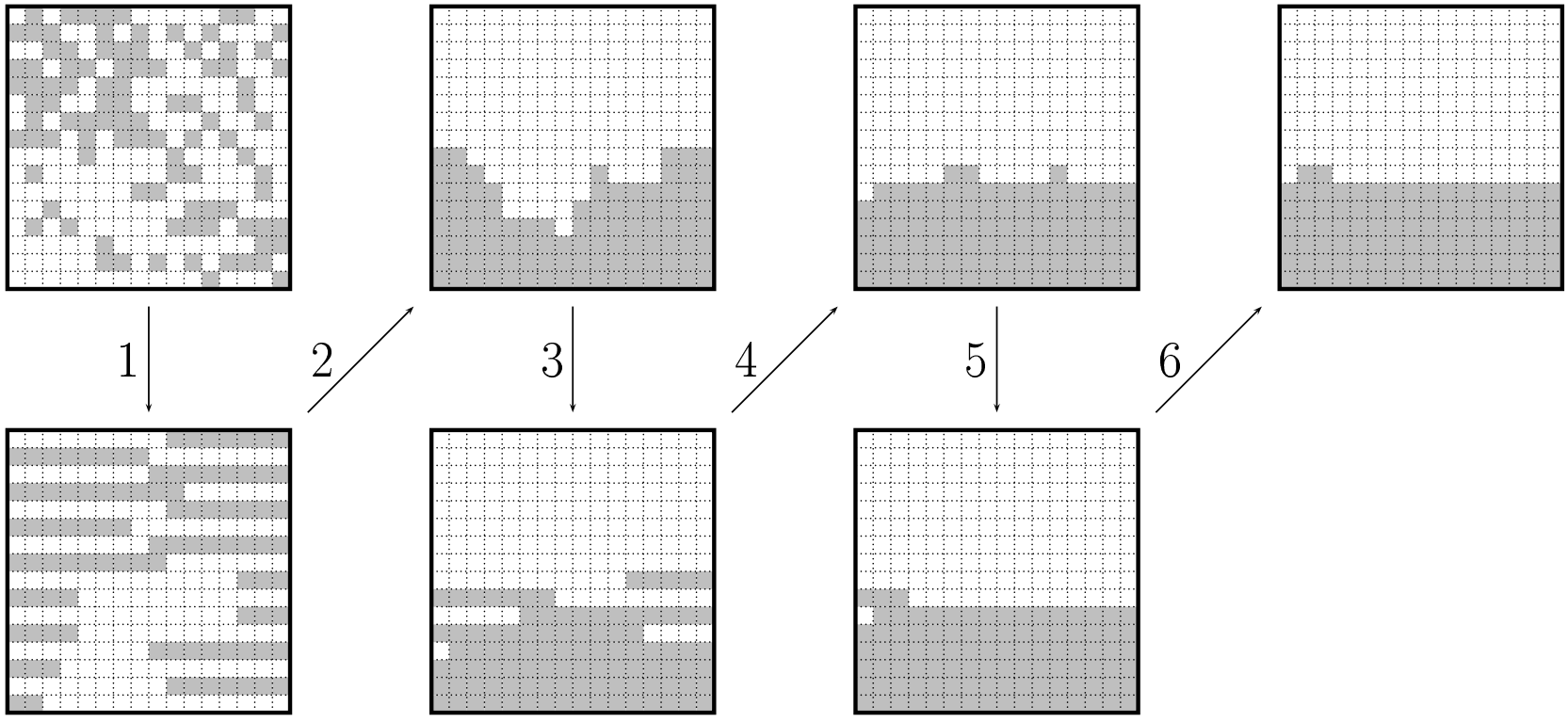

1

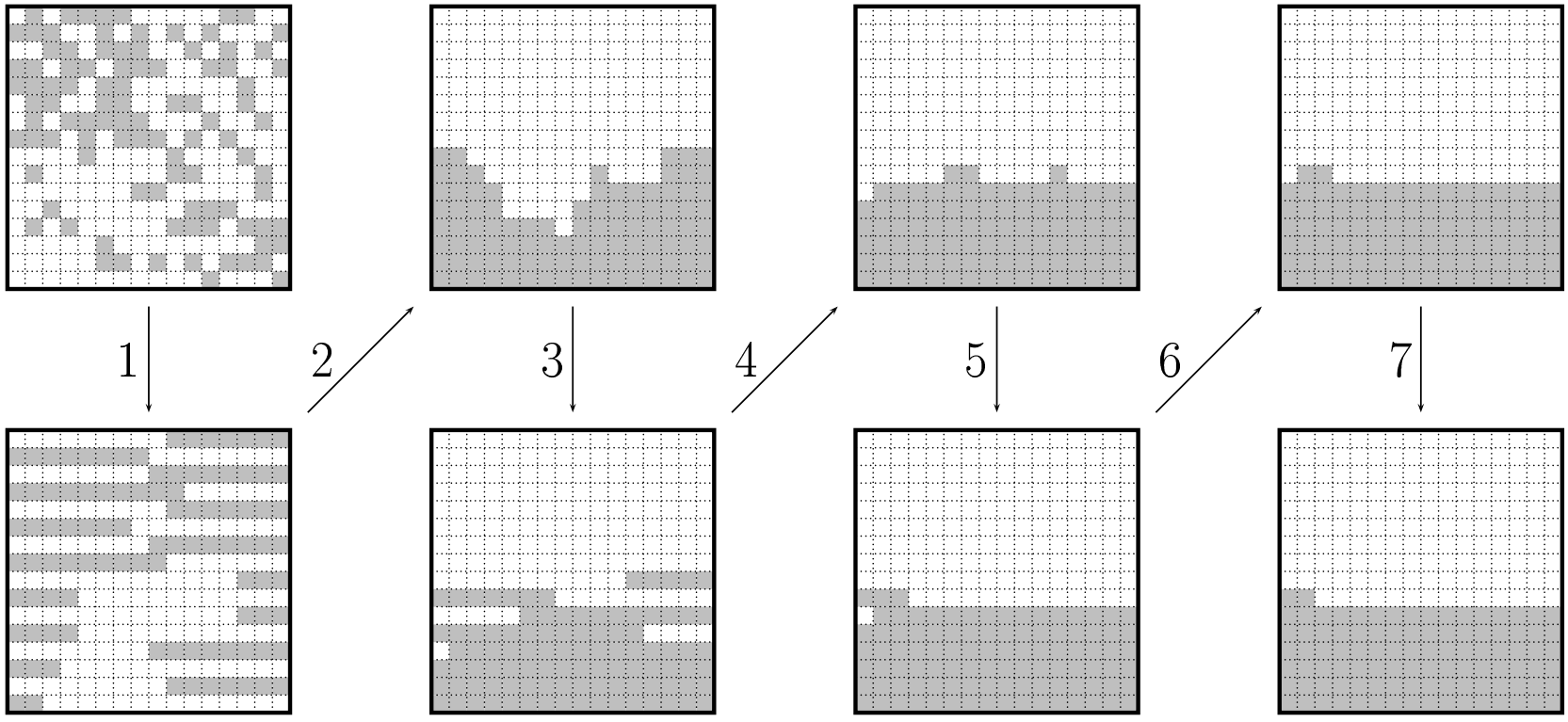

1
2


1
2
3


1
2
3
4


1
2
3
4
5


1
2
3
4
5
6



▸ proof of correctness

Algorithm (Shearsort)

$1 + 2\lceil \log \sqrt{n} \rceil$ phases:

initialization: send a modulo-2-counter down each column to
to distinguish even and odd rows and
establish checker board of `L` and `R` for the OETS

odd phases: each *row* is sorted:

- odd rows increasing to the right
- even rows decreasing to the right

even phases: each *column* is sorted

- all columns increasing downward

phase switching: use FSSP with generals at both ends of each row
to synchronize every $\sqrt{n}$ steps (needed by OETS)

Lemma

The Shearsort algorithm is correct and
needs $O(\sqrt{n} \log n)$ steps
(where n is the total number of elements to be sorted).

Lemma

The Shearsort algorithm is correct and
needs $O(\sqrt{n} \log n)$ steps
(where n is the total number of elements to be sorted).

in a moment:

It is *not* self-evident, that the algorithm is correct!

Correctness of Shearsort (1)

The 0-1 principle can be used. ▸ 0-1-principle

Consider a configuration after an even number of phases:

Do you remember?

Correctness of Shearsort (1)

The 0-1 principle can be used. ▸ 0-1-principle

Consider a configuration after an even number of phases:

Do you remember?

- at the top some pure 0-rows
- then some mixed rows
- at the bottom some pure 1-rows

claim:

Correctness of Shearsort (1)

The 0-1 principle can be used. ▸ 0-1-principle

Consider a configuration after an even number of phases:

Do you remember?

- at the top some pure 0-rows
- then some mixed rows
- at the bottom some pure 1-rows

claim: Two successive phases
reduce the number of mixed rows by at least one half

Correctness of Shearsort (2)

- consider two successive mixed rows
- after sorting the rows we have one of the following situations (or a mirror image of it):

$0\cdots01\cdots11\cdots1$	$0\cdots01\cdots1$	$0\cdots00\cdots01\cdots1$
$1\cdots11\cdots10\cdots0$	$1\cdots10\cdots0$	$1\cdots10\cdots00\cdots0$

- then sort the tiny "columns"

$0\cdots01\cdots10\cdots0$	$0\cdots00\cdots0$	$0\cdots00\cdots00\cdots0$
$1\cdots11\cdots11\cdots1$	$1\cdots11\cdots1$	$1\cdots10\cdots01\cdots1$

Correctness of Shearsort (2)

- consider two successive mixed rows
- after sorting the rows we have one of the following situations (or a mirror image of it):

$0\cdots01\cdots11\cdots1$	$0\cdots01\cdots1$	$0\cdots00\cdots01\cdots1$
$1\cdots11\cdots10\cdots0$	$1\cdots10\cdots0$	$1\cdots10\cdots00\cdots0$

- then sort the tiny "columns"

$0\cdots01\cdots10\cdots0$	$0\cdots00\cdots0$	$0\cdots00\cdots00\cdots0$
$1\cdots11\cdots11\cdots1$	$1\cdots11\cdots1$	$1\cdots10\cdots01\cdots1$

  - in each case at least one pure row and at most one mixed
- If we would sort the full columns by
  - first sorting pairs of rows
  - then shift pure 0-rows up and pure 1-rows down
  - then sort full columns

  the result would be the same.
- Hence, the number x of mixed rows decreases by at least $\frac{x-1}{2}$.

Correctness of Shearsort (3)

- initially at most $\sqrt{n}$ mixed rows
- hence after $\lceil \log \sqrt{n} \rceil$ row and column phases
  - at most one mixed row is left
  - which gets sorted during the last row phase
- each phase can be done in $\Theta(\sqrt{n})$ steps (OETS)
- total time: $\Theta(\sqrt{n} \log \sqrt{n}) = \Theta(\sqrt{n} \log n)$

Isn't the correctness of shearsort obvious?

Isn't the correctness of shearsort obvious?

No! **No!** **No!** **No!** **No!** **No!**

Isn't the correctness of shearsort obvious?

No! **No!** **No!** **No!** **No!** **No!**

- Let's change shearsort:
  always sort all rows in increasing order
- What happens?

Isn't the correctness of shearsort obvious?

No! **No!** **No!** **No!** **No!** **No!**

- Let's change shearsort:
  always sort all rows in increasing order
- What happens?
- This algorithm does not produce a result where the concatenation of all rows is a sorted row!
- counter example:

1	8
2	9

Can we sort faster than shearsort?

- shearsort needs $\Theta(\sqrt{n}\log n)$ steps
- the obvious lower bound is $2\sqrt{n} + O(1)$

Can we sort faster than shearsort?

- shearsort needs $\Theta(\sqrt{n}\log n)$ steps
- the obvious lower bound is $2\sqrt{n} + O(1)$
  - it might happen that the largest element has to travel from the upper left to the lower right corner
- there is a non-obvious lower bound of $3\sqrt{n}$
- Are there faster algorithms?

Can we sort faster than shearsort?

- shearsort needs $\Theta(\sqrt{n}\log n)$ steps
- the obvious lower bound is $2\sqrt{n} + O(1)$
  - it might happen that the largest element has to travel from the upper left to the lower right corner
- there is a non-obvious lower bound of $3\sqrt{n}$
- Are there faster algorithms? Yes!

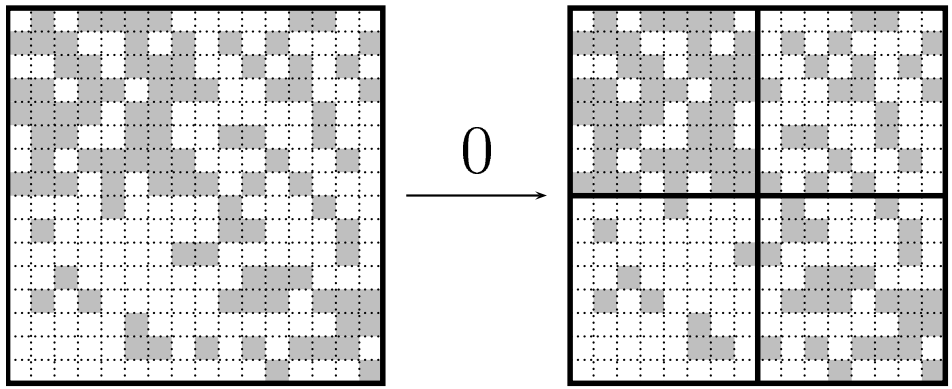
- Sorting in one-dimensional CA
  - Odd-even transposition sort
  - Knuth's 0-1 principle
- Sorting in two-dimensional CA
  - Sorting order
  - Shearsort
  - Algorithm by Schnorr and Shamir

Algorithm (by Schnorr and Shamir)

- number of elements to be sorted: $n = k^8$
- squares of size $m \times m = \sqrt{n} \times \sqrt{n} = k^4 \times k^4$
- the algorithm consists of 9 phases

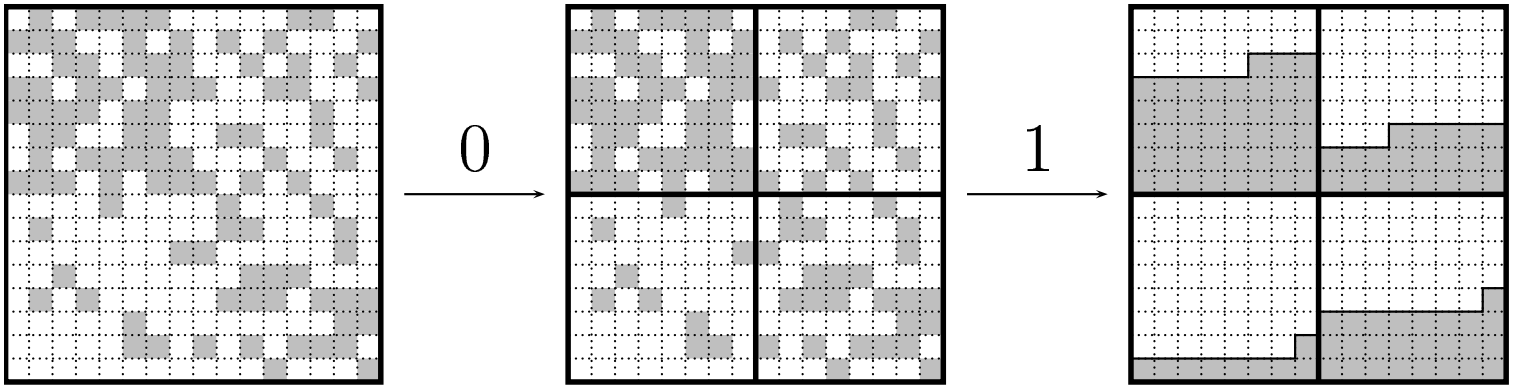
Phase 0

Cut the pattern in $k \times k$ blocks of size $k^3 \times k^3$. ($k = \sqrt[4]{m} = \sqrt[8]{n}$)



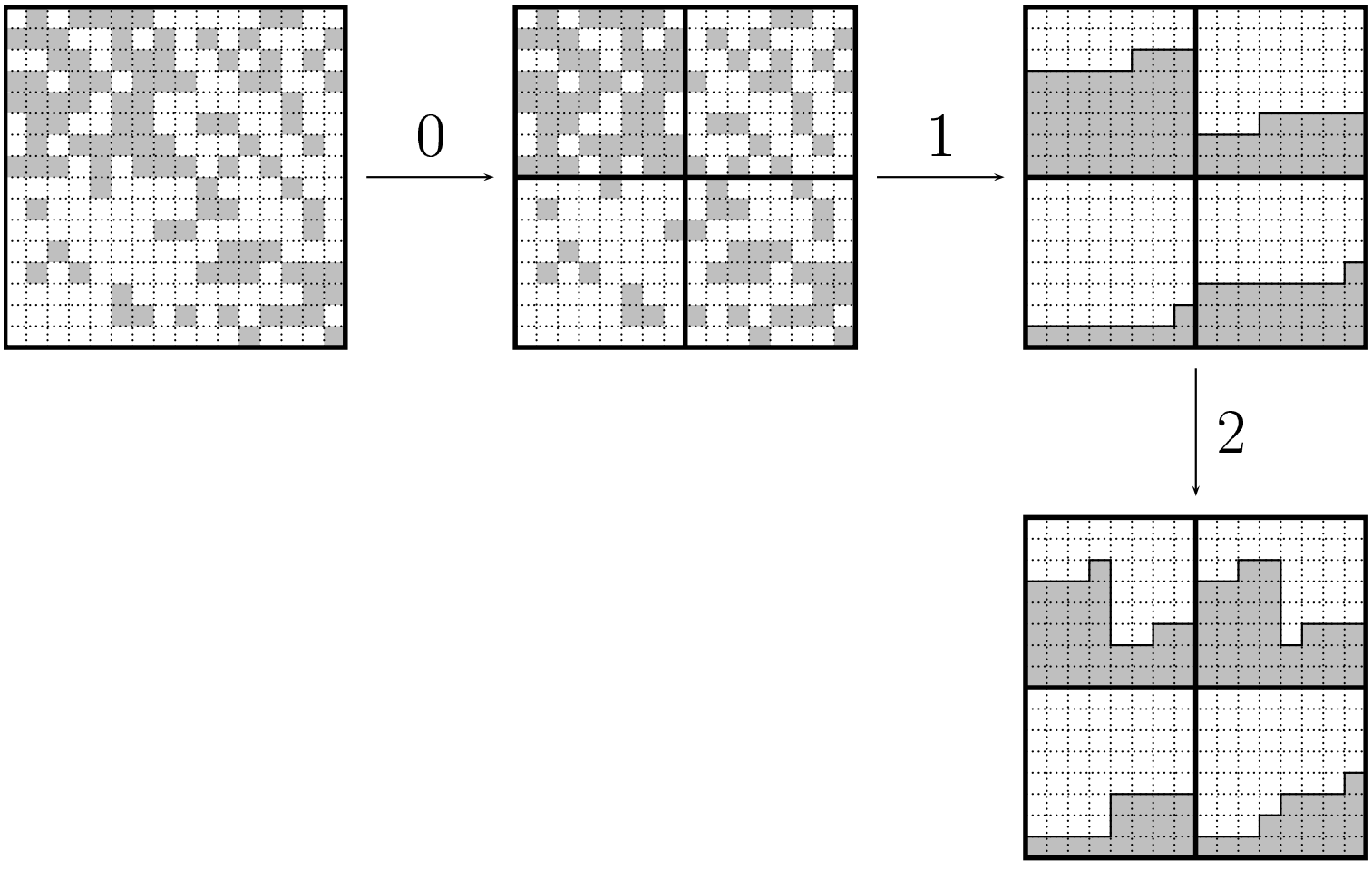
Phase 1

Sort each block in serpentine form.



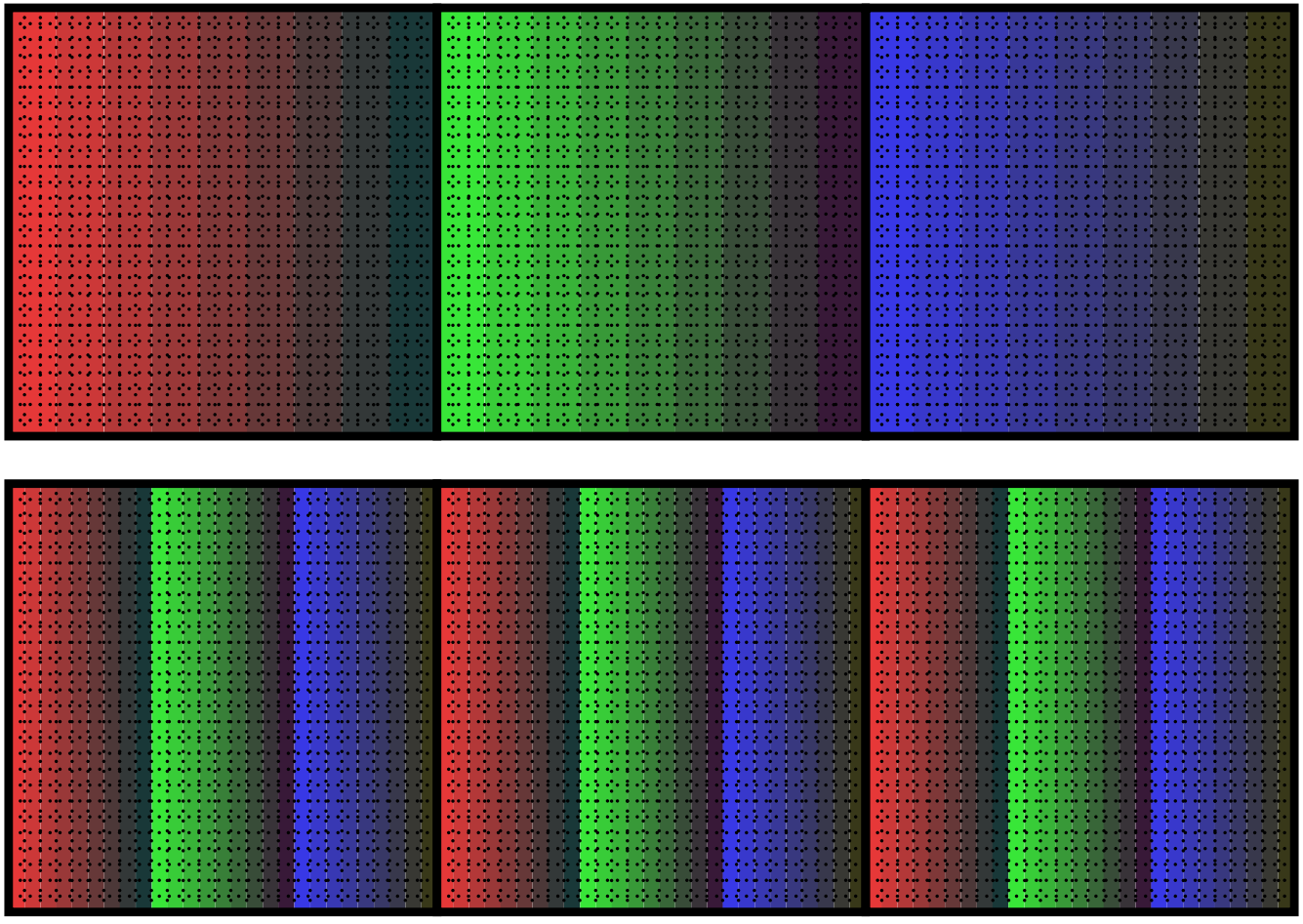
Phase 2

Distribute the full columns of each column of blocks evenly to all columns of blocks.



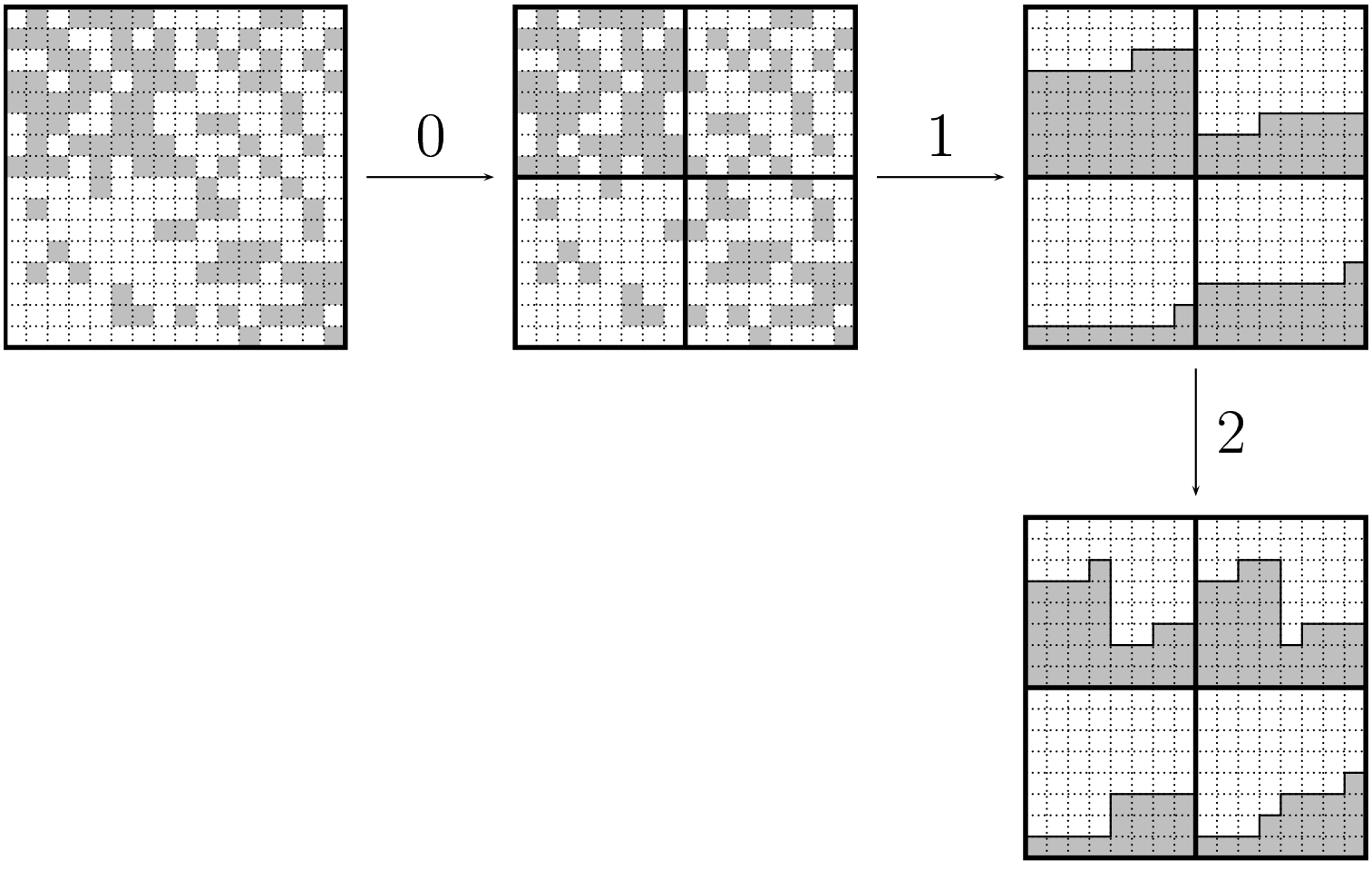
Rearragement of full columns (for phase 2)

shown just for one full row of $k = 3$ blocks



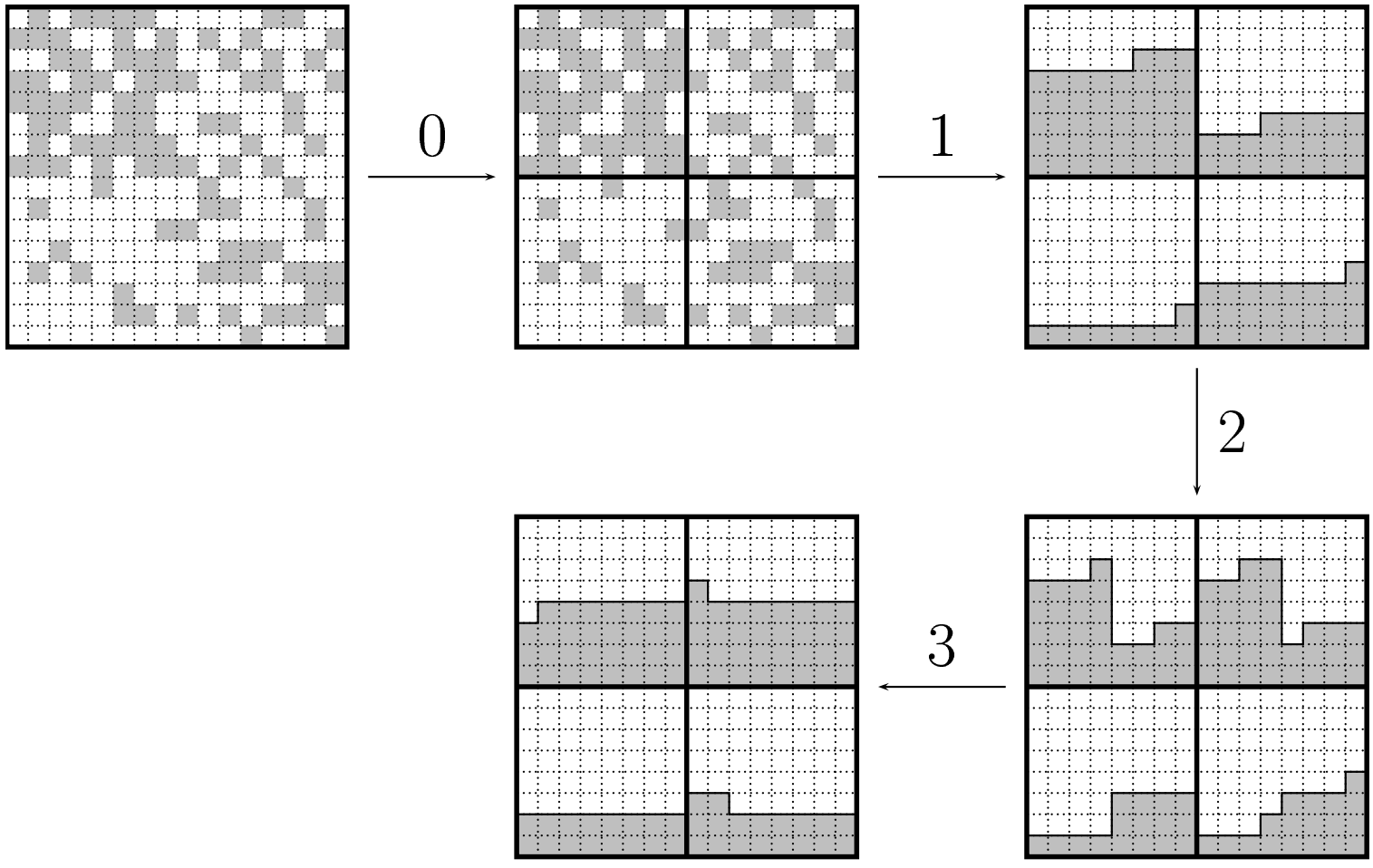
Phase 2

Distribute the full columns of each column of blocks evenly to all columns of blocks.



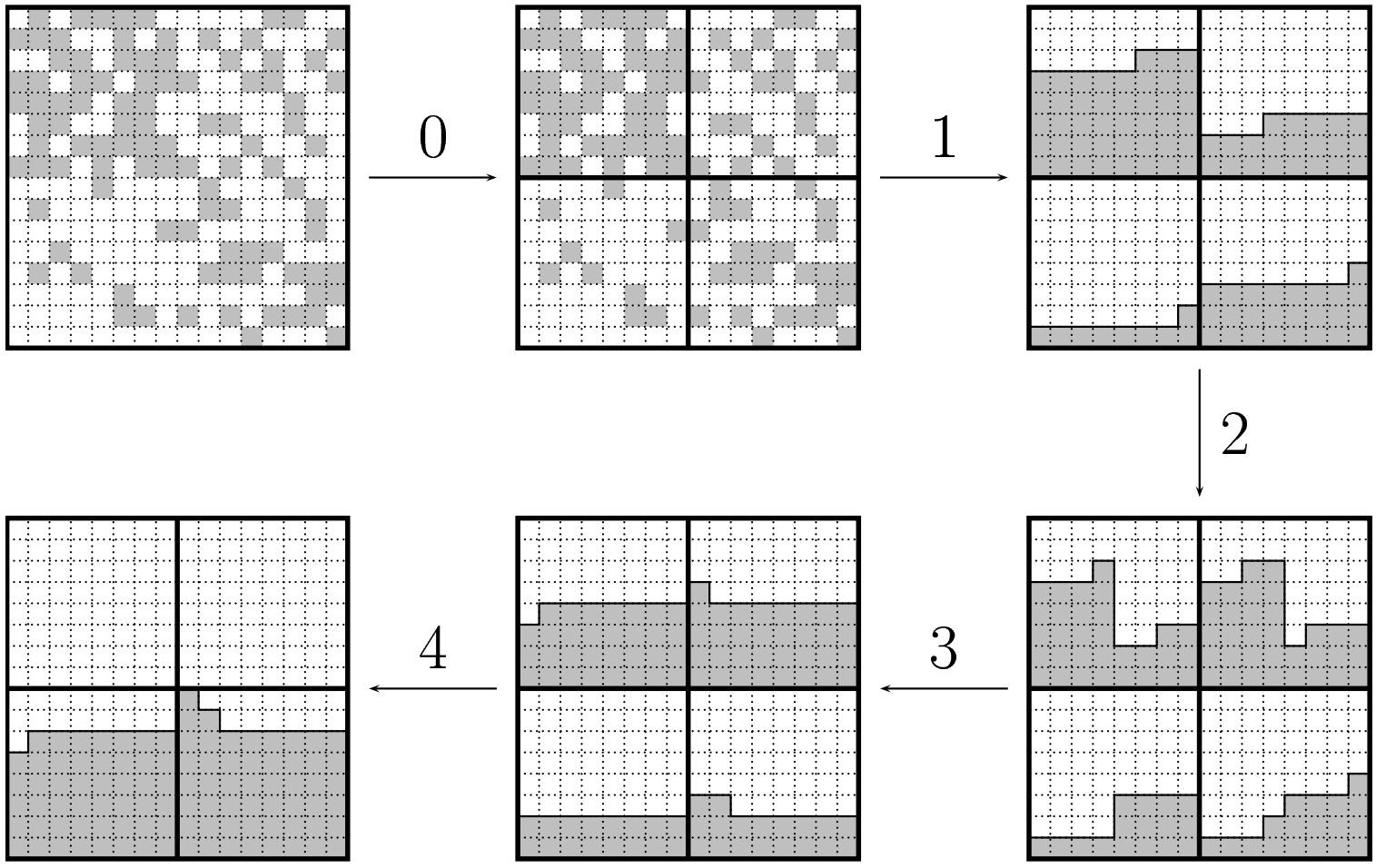
Phase 3

Sort each block in serpentine form.



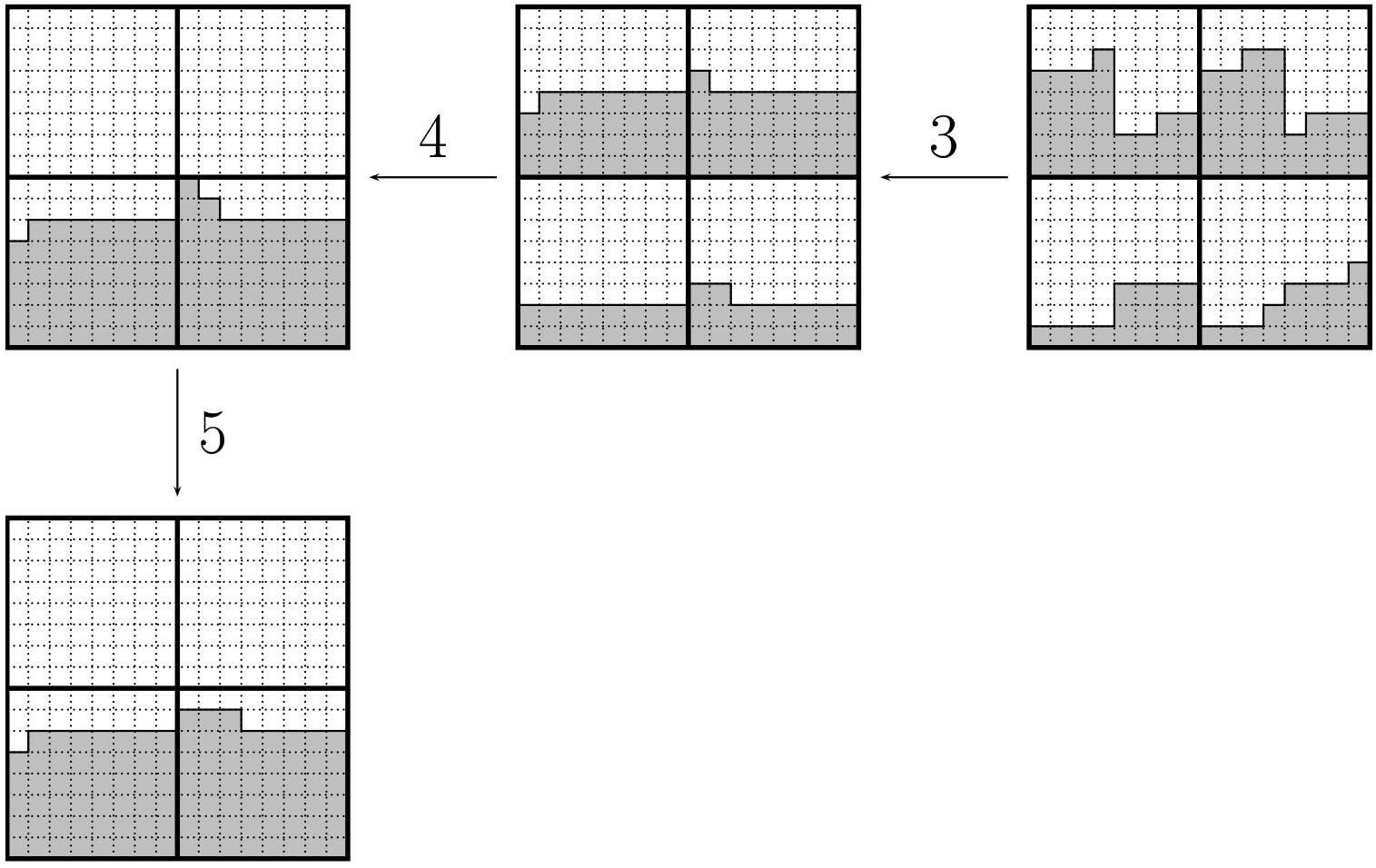
Phase 4

Sort each full column (small values to the top).



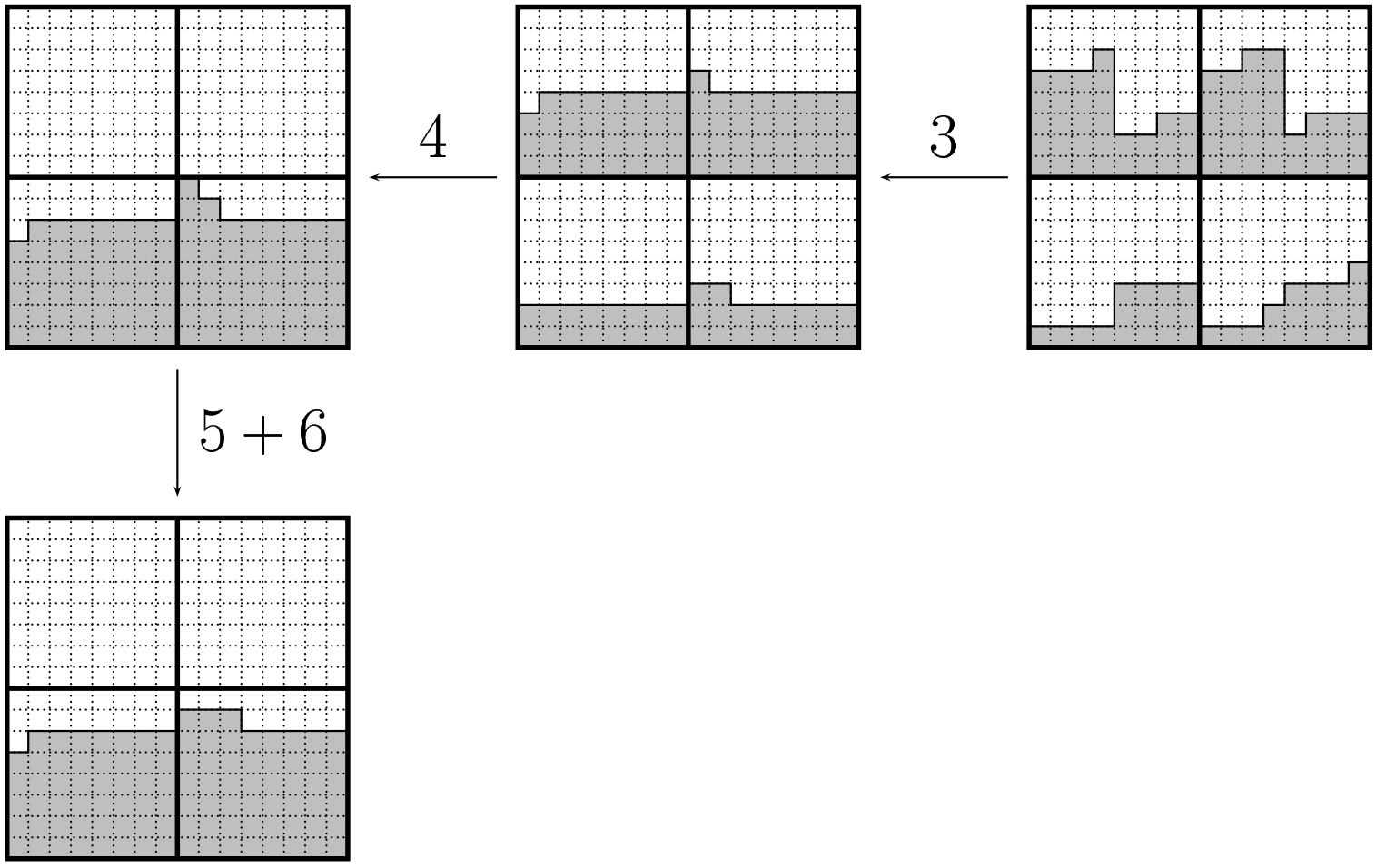
Phase 5

In each column of blocks sort pairs of blocks $1+2$, $3+4$, ... together in serpentine form.



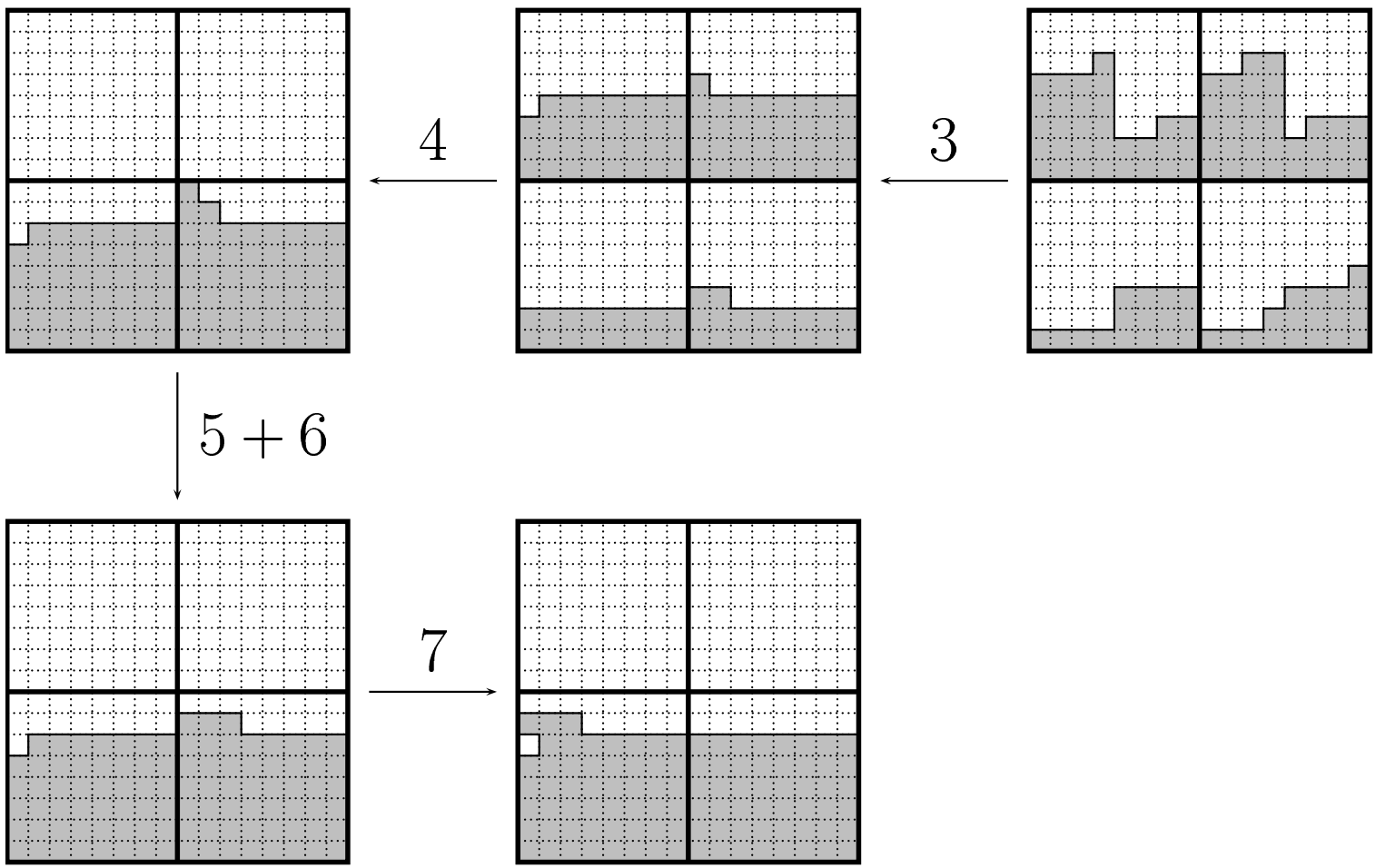
Phase 6

In each column of blocks sort pairs of blocks $2+3$, $4+5$, ... together in serpentine form.



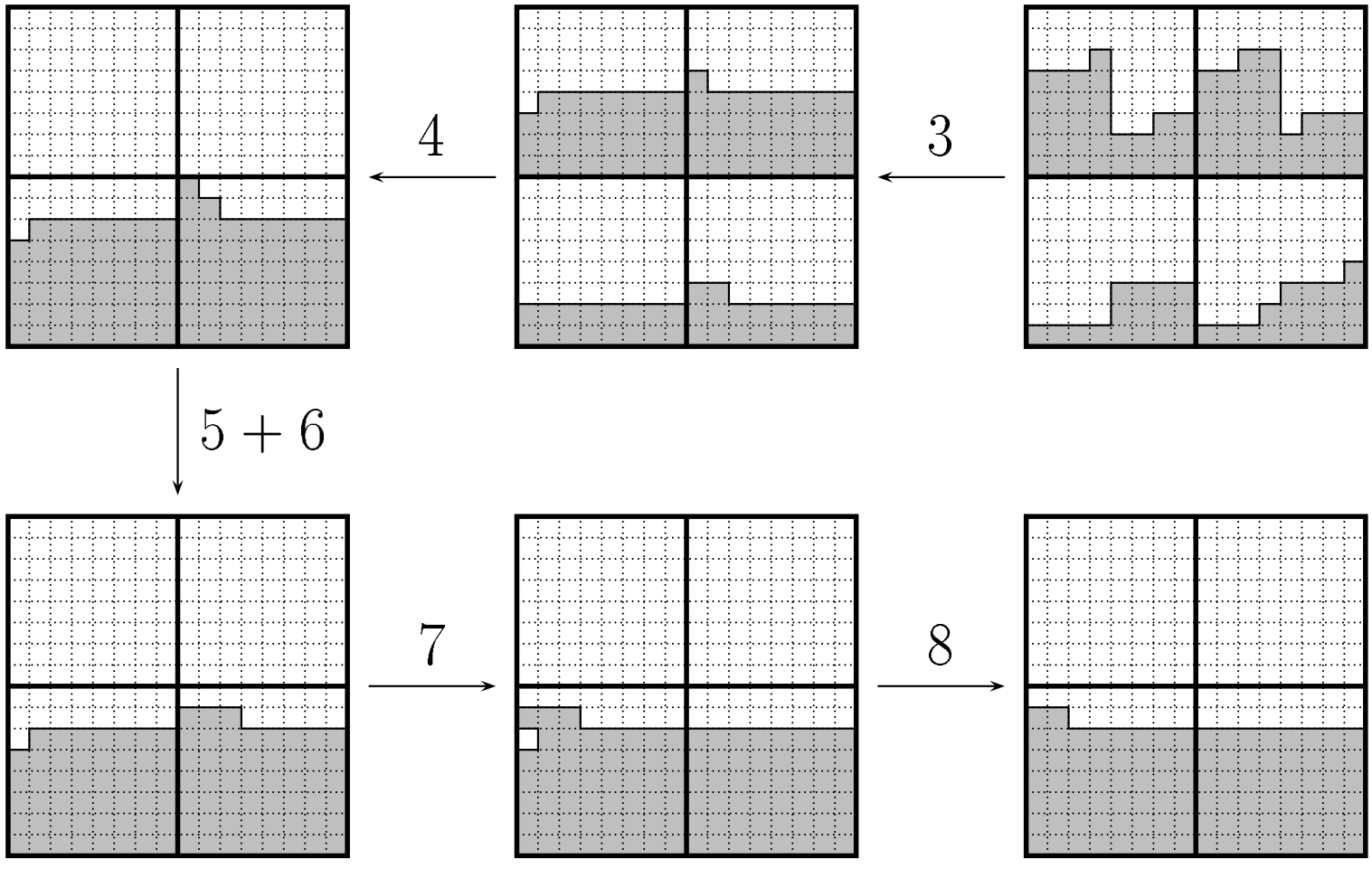
Phase 7

Sort each full row (odd rows increasing, even rows decreasing)



Phase 8

On the large serpentine for the full square do $2k^3$ steps of odd-even transposition sort.



Theorem

The algorithm by Schnorr and Shamir is sorting squares of size $\sqrt{n} \times \sqrt{n}$ in $\Theta(\sqrt{n})$ steps.

By now it should be obvious how useful the 0-1 sorting lemma is.

Proof (1)

Correctness: We can use the 0-1 sorting lemma.
we have after

- phase 1: in each block at most one mixed row:
  two columns of the same block differ by at most 1.
- phase 2: columns of one block are distributed evenly:
  two blocks of the same row of blocks differ by at most k
- phase 3: since this difference $k < k^3$,
  in each row of blocks there are at most 2 mixed rows
- phase 4: in each column of blocks there are at most k mixed rows

Proof (2)

we have after:

phases 5+6: Each column of blocks is in serpentine form.
during phases 3–6 no exchange between different columns of blocks; as after phase 2:
two columns of blocks differ by at most k^2.
Since $k^2 < k^3$, after phase 6 there are at most two full mixed rows.

phase 7: There are at most two full mixed rows:
one with many 0s and at most $k^2 \cdot k = k^3$ 1s,
the other with many 1s and at most $k^2 \cdot k = k^3$ 0s.
Only $\leq k^3 + k^3$ sorting steps in phase 8 are needed fix the rest. Hence after

phase 8: everything is sorted.

Proof (3)

Implementation details

- partitioning the square into blocks; e.g. horizontally:
  - mark cell $m^{3/4}$
    (have seen marking of cell $m^{1/2}$ in lecture two)
  - duplicate this length to the right:
    from the left and the right end of a block send signals to the right with speeds 1 and $1/2$ respectively
- switching between phases: FSSP
- redistribution of full columns during phase 2: tricky,
  but possible in $\Theta(k^4)$ steps

Proof (4)

time needed when using shearsort for sorting the blocks:

$$\Theta(k^4) + k^3 \cdot \log k^3 + \Theta(k^4) + k^3 \cdot \log k^3 + k^4 \\ + k^3 \cdot \log k^3 + k^3 \cdot \log k^3 + k^4 + k^3 \in \Theta(\sqrt{n})$$

since $k^4 = \sqrt{n}$.

Outlook

sorting of data items which do not fit into one cell, for example:

- sorting numbers having the same length:

  [0011;0101;0110;1011;0010;0011;1001;0000]

  is transformed into

  [0000;0010;0011;0011;0101;0110;1001;1011]

- sorting numbers of different length

  [11;101;110;1011;10;11;1001;0]

  is transformed into

  [0;10;11;11;101;110;1001;1011]

Summary

- one-dimensional CA:
  - n items stored in n cells can be sorted in $O(n)$ steps.
- two-dimensional CA:
  - the serpentine is a reasonable sorting order.
  - Sorting $n = \sqrt{n} \times \sqrt{n}$ elements in serpentine order is possible in $O(\sqrt{n})$ steps.