Nonrepetitive sequences

Grzegorz Ryn

Ogólnopolska Konferencja Studentów Matematyki $\theta\beta\mathit{I}\iota\mathit{c}\mathbb{Z}\varepsilon$ 2025

May 10, 2025

Introduction

Words

A *word* is a (possibly infinite) sequence of *characters* from some (finite) alphabet.

Introduction

Words

A *word* is a (possibly infinite) sequence of *characters* from some (finite) alphabet.

A typical problem in *combinatorics on words* asks 'Does exists an infinite sequance over some fixed alphabet, avoiding some *pattern*?'. This area of research was started by Axel Thue in 1906. He asked about the existance of *square-free* words.

Square-free words

Square-free words

A *square* is a consecutive repetition of character sequance. We will refer to a word as *square-free* if it doesn't contain any square.

Square-free words

Square-free words

A *square* is a consecutive repetition of character sequance. We will refer to a word as *square-free* if it doesn't contain any square.

- cb**abcabc**dab

Square-free words

Square-free words

A *square* is a consecutive repetition of character sequance. We will refer to a word as *square-free* if it doesn't contain any square.

- cb**abcabc**dab
- acd**bb**a

Square-free words

Square-free words

A *square* is a consecutive repetition of character sequance. We will refer to a word as *square-free* if it doesn't contain any square.

- cb**abcabc**dab
- acd**bb**a

Binary alphabet

It's easy to check, that we can't construct such sequence with 2-element alphabet.

Square-free words

Square-free words

A *square* is a consecutive repetition of character sequance. We will refer to a word as *square-free* if it doesn't contain any square.

- cb**abcabc**dab
- acd**bb**a

Binary alphabet

It's easy to check, that we can't construct such sequence with 2-element alphabet. There are only two square-free words with three letters:

- aba
- bab

Square-free words

Square-free words

A *square* is a consecutive repetition of character sequance. We will refer to a word as *square-free* if it doesn't contain any square.

- cb**abcabc**dab
- acd**bb**a

Binary alphabet

It's easy to check, that we can't construct such sequence with 2-element alphabet. There are only two square-free words with three letters:

- aba
- bab

Clearly none nof them can be extended.

Thue's result

In 1906, Thue showed, that a three letter alphabet is enough.

Thue's result

In 1906, Thue showed, that a three letter alphabet is enough.

Infinite square-free word construction

- Start with a single letter 'a' or 'b'.

Thue's result

In 1906, Thue showed, that a three letter alphabet is enough.

Infinite square-free word construction

- Start with a single letter 'a' or 'b'.
- Given a square-free word w obtained with this procedure, the following substitions preserve this property:
  - $a \to abac$
  - $b \to bcac$
  - $c \to bcac$ | if the preceeding letter was 'a'
  - $c \to acbc$ | if the preceeding letter was 'b'

Thue's result

In 1906, Thue showed, that a three letter alphabet is enough.

Infinite square-free word construction

- Start with a single letter 'a' or 'b'.
- Given a square-free word w obtained with this procedure, the following substitions preserve this property:
  - $a \to abac$
  - $b \to bcac$
  - $c \to bcac$ | if the preceeding letter was 'a'
  - $c \to acbc$ | if the preceeding letter was 'b'

The original proof of Thue used much more complex substitutions, however the main idea remained unchanged.

List coloring

Let's consider a slight generalization of the previous problem - list coloring. Previously, we worked in a setting with an alphabet which is identical for every position in our sequence. Now, let's assume that for every position $i \in \mathbb{N}$ we have a list L_i of admissable characters.

List coloring

Let's consider a slight generalization of the previous problem - list coloring. Previously, we worked in a setting with an alphabet which is identical for every position in our sequence. Now, let's assume that for every position $i \in \mathbb{N}$ we have a list L_i of admissable characters.

In this setting, we want to find the smallest common size of lists guaranteeing a *proper* (square-free) character assignment. At first glance, it appears to be simpler problem, however in analogous problems such as *graph coloring* it's the opposite. Our situation is more complex and it is an open problem wheather list with three elements each are sufficient.

Algorithmic list coloring

We will show that lists of size 4 are sufficient. Moreover we will introduce a simple algorithm which constructs a desired square-free word of length n, given the lists.

Algorithmic list coloring

We will show that lists of size 4 are sufficient. Moreover we will introduce a simple algorithm which constructs a desired square-free word of length n, given the lists.

Algorithm

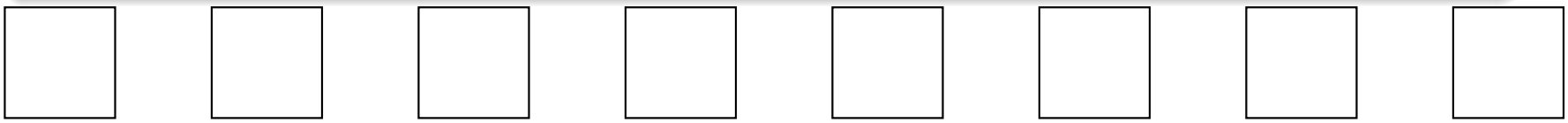
1: Start with an empty word w
2: **while** $|w| < n$ **do**
3: Pick character $c \in L_{|w|}$ uniformly at random
4: Append c to w, i.e. $w \leftarrow wc$
5: **if** $w = w'zz$ **then** $w \leftarrow w'z$
6: **end if**
7: **end while**
8: **return** w

Algorithmic list coloring

We will show that lists of size 4 are sufficient. Moreover we will introduce a simple algorithm which constructs a desired square-free word of length n, given the lists.

Algorithm

1: Start with an empty word w
2: **while** $|w| < n$ **do**
3: Pick character $c \in L_{|w|}$ uniformly at random
4: Append c to w, i.e. $w \leftarrow wc$
5: **if** $w = w'zz$ **then** $w \leftarrow w'z$
6: **end if**
7: **end while**
8: **return** w

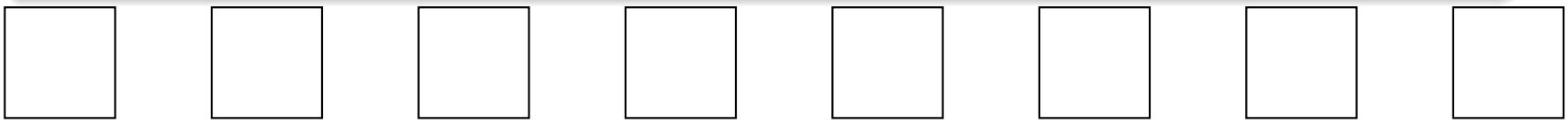


Algorithmic list coloring

We will show that lists of size 4 are sufficient. Moreover we will introduce a simple algorithm which constructs a desired square-free word of length n, given the lists.

Algorithm

1: Start with an empty word w
2: **while** $|w| < n$ **do**
3: Pick character $c \in L_{|w|}$ uniformly at random
4: Append c to w, i.e. $w \leftarrow wc$
5: **if** $w = w'zz$ **then** $w \leftarrow w'z$
6: **end if**
7: **end while**
8: **return** w

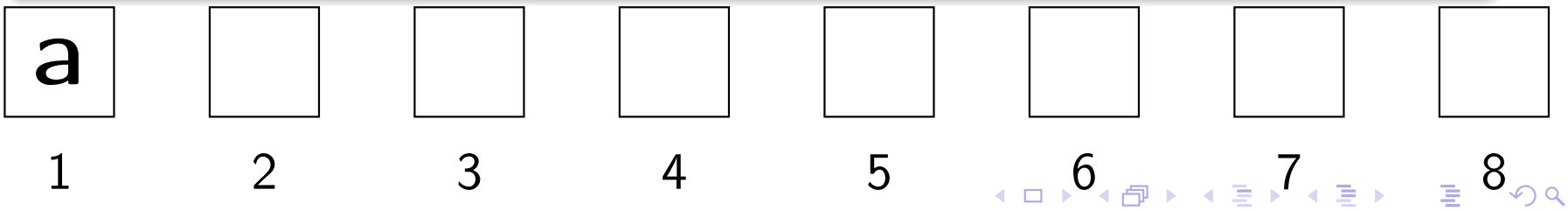


Algorithmic list coloring

We will show that lists of size 4 are sufficient. Moreover we will introduce a simple algorithm which constructs a desired square-free word of length n, given the lists.

Algorithm

1: Start with an empty word w
2: **while** $|w| < n$ **do**
3: Pick character $c \in L_{|w|}$ uniformly at random
4: Append c to w, i.e. $w \leftarrow wc$
5: **if** $w = w'zz$ **then** $w \leftarrow w'z$
6: **end if**
7: **end while**
8: **return** w

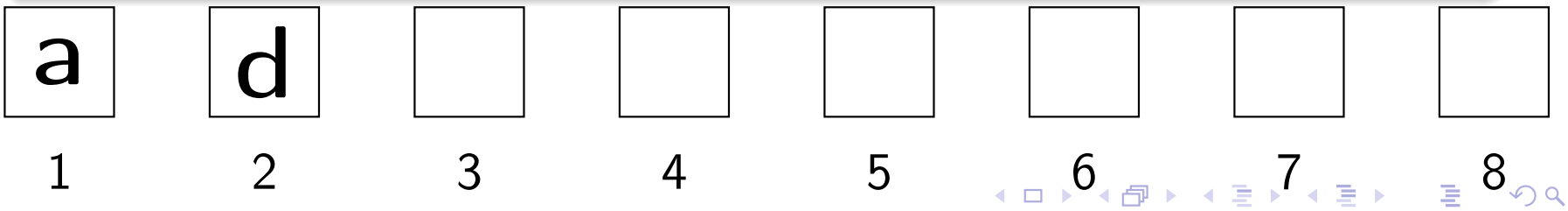


Algorithmic list coloring

We will show that lists of size 4 are sufficient. Moreover we will introduce a simple algorithm which constructs a desired square-free word of length n, given the lists.

Algorithm

1: Start with an empty word w
2: **while** $|w| < n$ **do**
3: Pick character $c \in L_{|w|}$ uniformly at random
4: Append c to w, i.e. $w \leftarrow wc$
5: **if** $w = w'zz$ **then** $w \leftarrow w'z$
6: **end if**
7: **end while**
8: **return** w

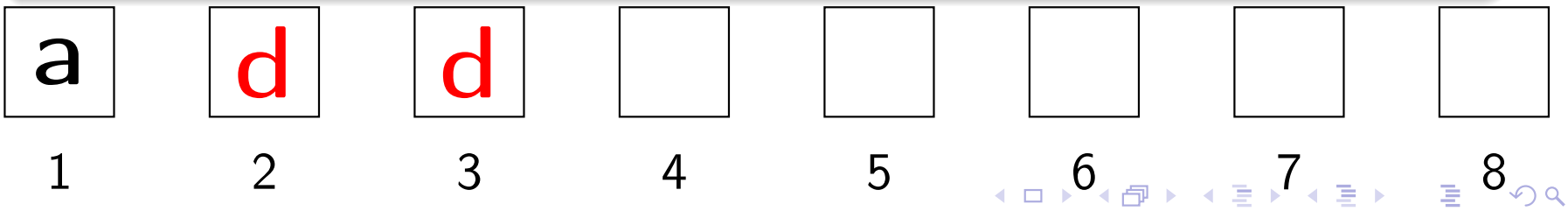


Algorithmic list coloring

We will show that lists of size 4 are sufficient. Moreover we will introduce a simple algorithm which constructs a desired square-free word of length n, given the lists.

Algorithm

1: Start with an empty word w
2: **while** $|w| < n$ **do**
3: Pick character $c \in L_{|w|}$ uniformly at random
4: Append c to w, i.e. $w \leftarrow wc$
5: **if** $w = w'zz$ **then** $w \leftarrow w'z$
6: **end if**
7: **end while**
8: **return** w

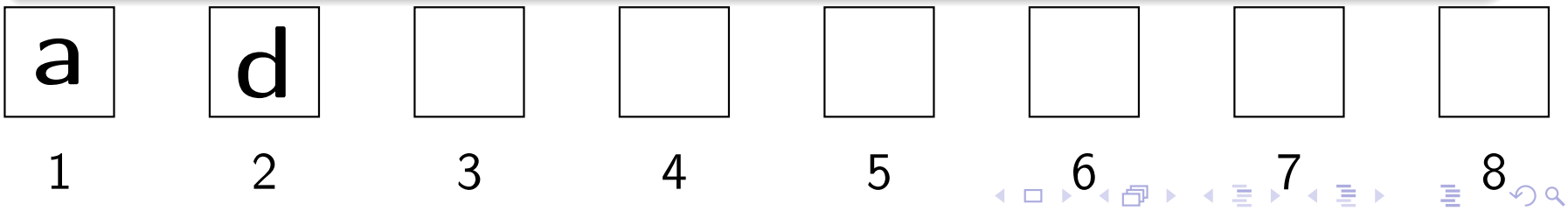


Algorithmic list coloring

We will show that lists of size 4 are sufficient. Moreover we will introduce a simple algorithm which constructs a desired square-free word of length n, given the lists.

Algorithm

1: Start with an empty word w
2: **while** $|w| < n$ **do**
3: Pick character $c \in L_{|w|}$ uniformly at random
4: Append c to w, i.e. $w \leftarrow wc$
5: **if** $w = w'zz$ **then** $w \leftarrow w'z$
6: **end if**
7: **end while**
8: **return** w

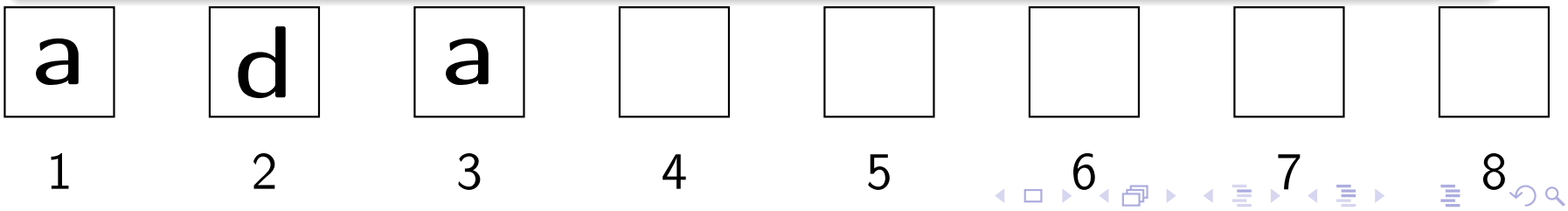


Algorithmic list coloring

We will show that lists of size 4 are sufficient. Moreover we will introduce a simple algorithm which constructs a desired square-free word of length n, given the lists.

Algorithm

1: Start with an empty word w
2: **while** $|w| < n$ **do**
3: Pick character $c \in L_{|w|}$ uniformly at random
4: Append c to w, i.e. $w \leftarrow wc$
5: **if** $w = w'zz$ **then** $w \leftarrow w'z$
6: **end if**
7: **end while**
8: **return** w

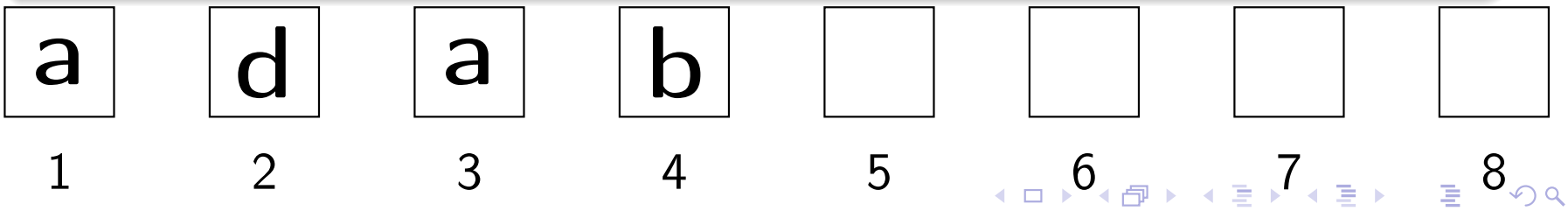


Algorithmic list coloring

We will show that lists of size 4 are sufficient. Moreover we will introduce a simple algorithm which constructs a desired square-free word of length n, given the lists.

Algorithm

1: Start with an empty word w
2: **while** $|w| < n$ **do**
3: Pick character $c \in L_{|w|}$ uniformly at random
4: Append c to w, i.e. $w \leftarrow wc$
5: **if** $w = w'zz$ **then** $w \leftarrow w'z$
6: **end if**
7: **end while**
8: **return** w

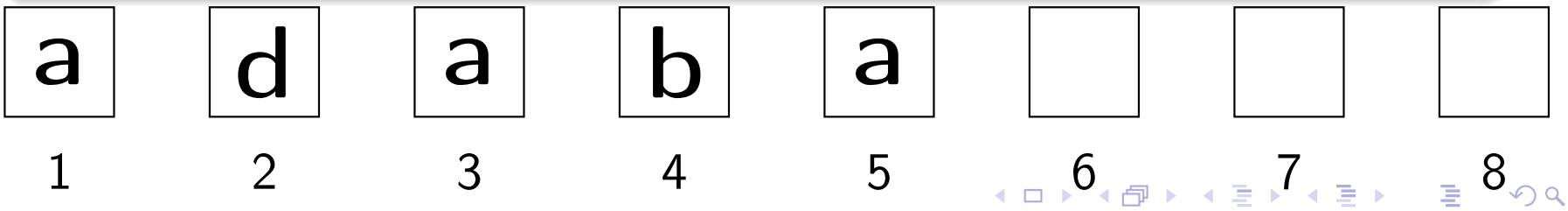


Algorithmic list coloring

We will show that lists of size 4 are sufficient. Moreover we will introduce a simple algorithm which constructs a desired square-free word of length n, given the lists.

Algorithm

1: Start with an empty word w
2: **while** $|w| < n$ **do**
3: Pick character $c \in L_{|w|}$ uniformly at random
4: Append c to w, i.e. $w \leftarrow wc$
5: **if** $w = w'zz$ **then** $w \leftarrow w'z$
6: **end if**
7: **end while**
8: **return** w

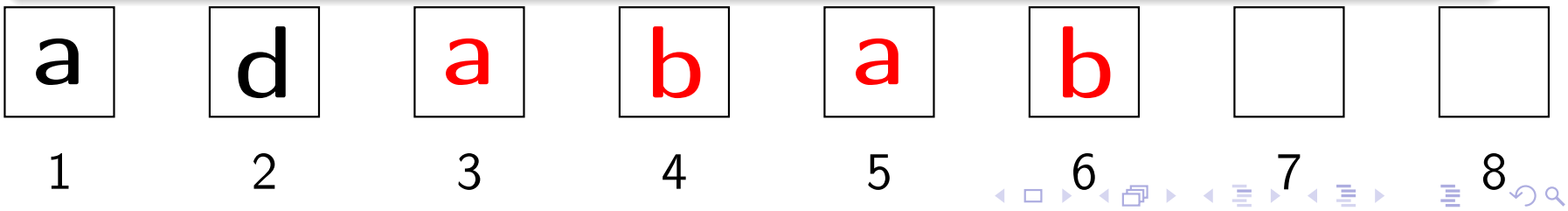


Algorithmic list coloring

We will show that lists of size 4 are sufficient. Moreover we will introduce a simple algorithm which constructs a desired square-free word of length n, given the lists.

Algorithm

1: Start with an empty word w
2: **while** $|w| < n$ **do**
3: Pick character $c \in L_{|w|}$ uniformly at random
4: Append c to w, i.e. $w \leftarrow wc$
5: **if** $w = w'zz$ **then** $w \leftarrow w'z$
6: **end if**
7: **end while**
8: **return** w

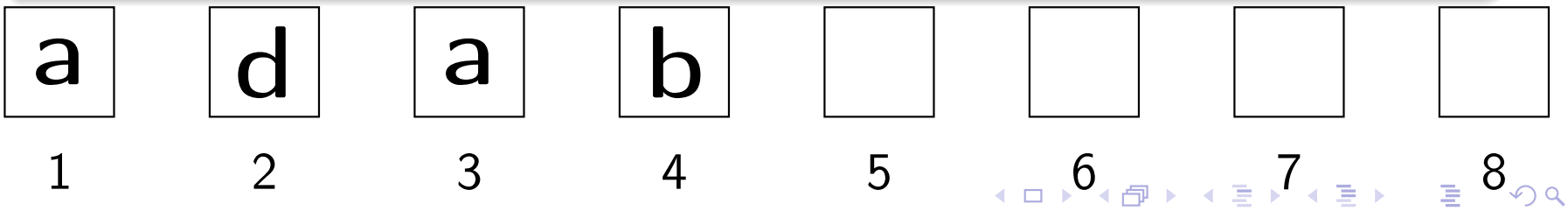


Algorithmic list coloring

We will show that lists of size 4 are sufficient. Moreover we will introduce a simple algorithm which constructs a desired square-free word of length n, given the lists.

Algorithm

1: Start with an empty word w
2: **while** $|w| < n$ **do**
3: Pick character $c \in L_{|w|}$ uniformly at random
4: Append c to w, i.e. $w \leftarrow wc$
5: **if** $w = w'zz$ **then** $w \leftarrow w'z$
6: **end if**
7: **end while**
8: **return** w

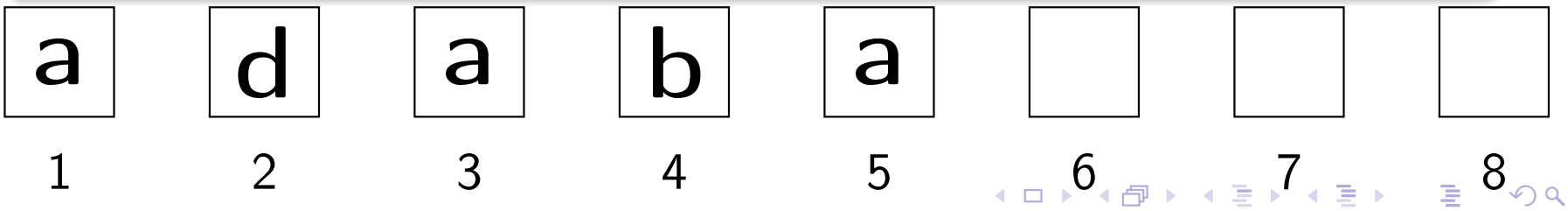


Algorithmic list coloring

We will show that lists of size 4 are sufficient. Moreover we will introduce a simple algorithm which constructs a desired square-free word of length n, given the lists.

Algorithm

1: Start with an empty word w
2: **while** $|w| < n$ **do**
3: Pick character $c \in L_{|w|}$ uniformly at random
4: Append c to w, i.e. $w \leftarrow wc$
5: **if** $w = w'zz$ **then** $w \leftarrow w'z$
6: **end if**
7: **end while**
8: **return** w

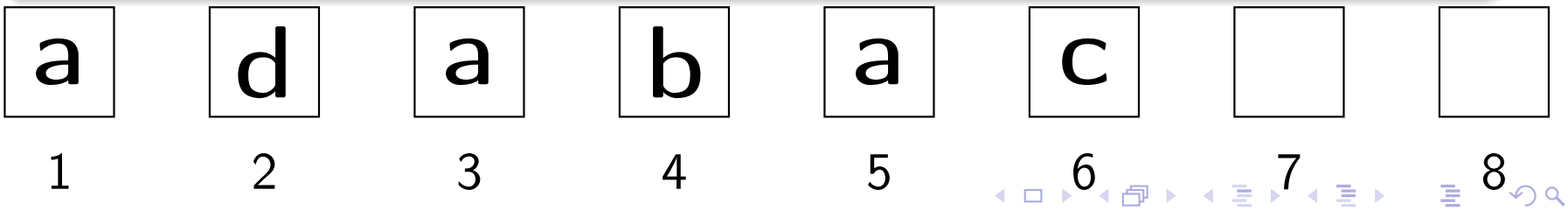


Algorithmic list coloring

We will show that lists of size 4 are sufficient. Moreover we will introduce a simple algorithm which constructs a desired square-free word of length n, given the lists.

Algorithm

1: Start with an empty word w
2: **while** $|w| < n$ **do**
3: Pick character $c \in L_{|w|}$ uniformly at random
4: Append c to w, i.e. $w \leftarrow wc$
5: **if** $w = w'zz$ **then** $w \leftarrow w'z$
6: **end if**
7: **end while**
8: **return** w

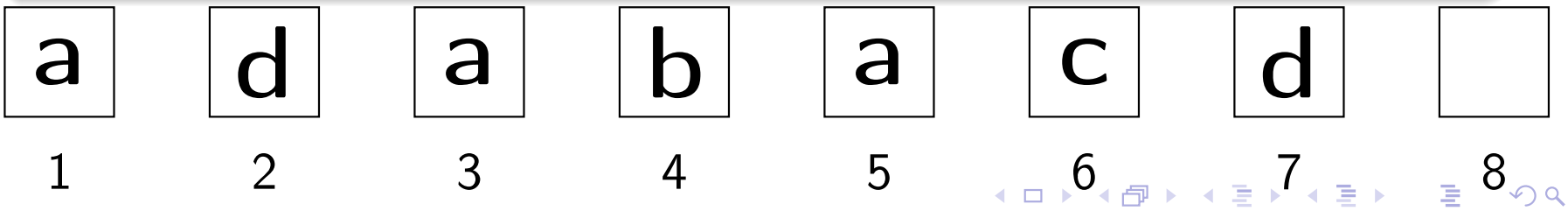


Algorithmic list coloring

We will show that lists of size 4 are sufficient. Moreover we will introduce a simple algorithm which constructs a desired square-free word of length n, given the lists.

Algorithm

1: Start with an empty word w
2: **while** $|w| < n$ **do**
3: Pick character $c \in L_{|w|}$ uniformly at random
4: Append c to w, i.e. $w \leftarrow wc$
5: **if** $w = w'zz$ **then** $w \leftarrow w'z$
6: **end if**
7: **end while**
8: **return** w



Algorithmic list coloring

We will show that lists of size 4 are sufficient. Moreover we will introduce a simple algorithm which constructs a desired square-free word of length n, given the lists.

Algorithm

1: Start with an empty word w
2: **while** $|w| < n$ **do**
3: Pick character $c \in L_{|w|}$ uniformly at random
4: Append c to w, i.e. $w \leftarrow wc$
5: **if** $w = w'zz$ **then** $w \leftarrow w'z$
6: **end if**
7: **end while**
8: **return** w

a	d	a	b	a	c	d	c
1	2	3	4	5	6	7	8

Entropy compression argumen

With some intuition about how the algorithm works, we can attempt to prove it. The idea of the proof is based on the Moser and Tardos' proof of algorithmic Lovász Local Lemma. Their method was later called *Entropy compression argument* by Terrance Tao at his blog.

Entropy compression argumen

With some intuition about how the algorithm works, we can attempt to prove it. The idea of the proof is based on the Moser and Tardos' proof of algorithmic Lovász Local Lemma. Their method was later called *Entropy compression argument* by Terrance Tao at his blog.

proof

- Assume that there exists a 4-element list assignment s.t. it's impossible to construct a square-free word of length n from these lists.

Entropy compression argumen

With some intuition about how the algorithm works, we can attempt to prove it. The idea of the proof is based on the Moser and Tardos' proof of algorithmic Lovász Local Lemma. Their method was later called *Entropy compression argument* by Terrance Tao at his blog.

proof

- Assume that there exists a 4-element list assignment s.t. it's impossible to construct a square-free word of length n from these lists.
- Let $M >> n$ be a fixed integer - number of algorithm's steps. We will enumearate all 4^M possible executions. Clearly, at the end of every step the length of w is smaller than n.

Entropy compression argumen

With some intuition about how the algorithm works, we can attempt to prove it. The idea of the proof is based on the Moser and Tardos' proof of algorithmic Lovász Local Lemma. Their method was later called *Entropy compression argument* by Terrance Tao at his blog.

proof

- Assume that there exists a 4-element list assignment s.t. it's impossible to construct a square-free word of length n from these lists.
- Let $M >> n$ be a fixed integer - number of algorithm's steps. We will enumearate all 4^M possible executions. Clearly, at the end of every step the length of w is smaller than n.
- We will produce a *log* of each execution - an alternative represantation of each of 4^M runs.

Proof - cont.

Log

Let $c_1, c_2, \ldots, c_M$ be the choices of characters throughout the execution. Let's analyze the length of the word w throughout the execution.

Proof - cont.

Log

Let $c_1, c_2, \ldots, c_M$ be the choices of characters throughout the execution. Let's analyze the length of the word w throughout the execution. At each step, it can increase by one or decrease by arbitrary value s.t. w is non-empty. Let $d_1 = 1, d_2, \ldots, d_M$ be the sequence of the differences in length of w after each step of our algorithm. I.e. d_j is the difference between length of w after j-th step and $(j-1)$-th step.

Proof - cont.

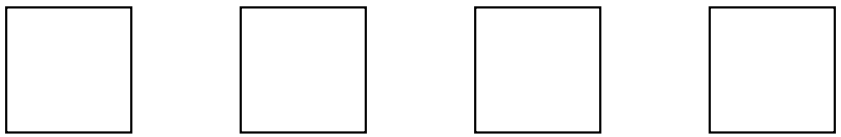
Log

Let $c_1, c_2, \ldots, c_M$ be the choices of characters throughout the execution. Let's analyze the length of the word w throughout the execution. At each step, it can increase by one or decrease by arbitrary value s.t. w is non-empty. Let $d_1 = 1, d_2, \ldots, d_M$ be the sequence of the differences in length of w after each step of our algorithm. I.e. d_j is the difference between length of w after j-th step and $(j-1)$-th step. For a fixed execution, we will refer to a pair (d, w) - sequence of differences and the final word as a *log*.

Proof - cont.

Log

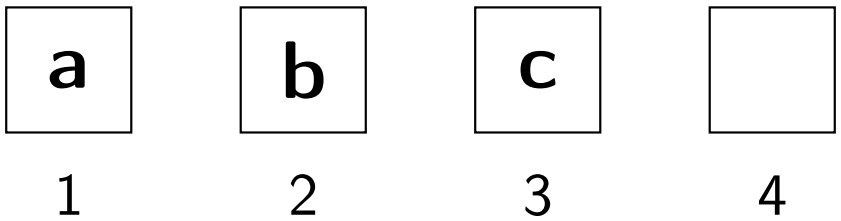
Let $c_1, c_2, \ldots, c_M$ be the choices of characters throughout the execution. Let's analyze the length of the word w throughout the execution. At each step, it can increase by one or decrease by arbitrary value s.t. w is non-empty. Let $d_1 = 1, d_2, \ldots, d_M$ be the sequence of the differences in length of w after each step of our algorithm. I.e. d_j is the difference between length of w after j-th step and $(j-1)$-th step. For a fixed execution, we will refer to a pair (d, w) - sequence of differences and the final word as a *log*.



Proof - cont.

Log

Let $c_1, c_2, \ldots, c_M$ be the choices of characters throughout the execution. Let's analyze the length of the word w throughout the execution. At each step, it can increase by one or decrease by arbitrary value s.t. w is non-empty. Let $d_1 = 1, d_2, \ldots, d_M$ be the sequence of the differences in length of w after each step of our algorithm. I.e. d_j is the difference between length of w after j-th step and $(j-1)$-th step. For a fixed execution, we will refer to a pair (d, w) - sequence of differences and the final word as a *log*.

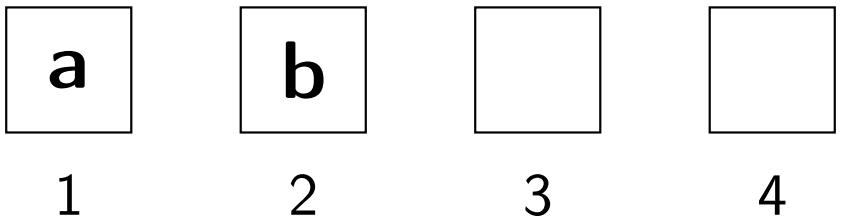


Diff	1	1	1	-1	1
Choice					

Proof - cont.

Log

Let $c_1, c_2, \ldots, c_M$ be the choices of characters throughout the execution. Let's analyze the length of the word w throughout the execution. At each step, it can increase by one or decrease by arbitrary value s.t. w is non-empty. Let $d_1 = 1, d_2, \ldots, d_M$ be the sequence of the differences in length of w after each step of our algorithm. I.e. d_j is the difference between length of w after j-th step and $(j-1)$-th step. For a fixed execution, we will refer to a pair (d, w) - sequence of differences and the final word as a *log*.

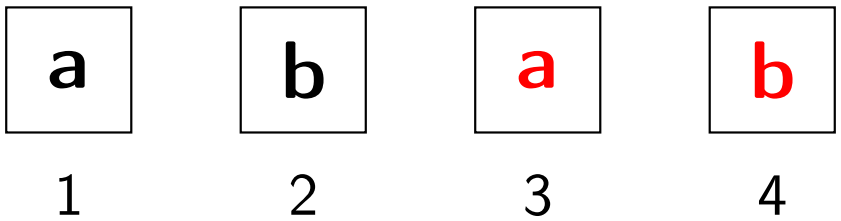


Diff	1	1	1	-1	1
Choice					c

Proof - cont.

Log

Let $c_1, c_2, \ldots, c_M$ be the choices of characters throughout the execution. Let's analyze the length of the word w throughout the execution. At each step, it can increase by one or decrease by arbitrary value s.t. w is non-empty. Let $d_1 = 1, d_2, \ldots, d_M$ be the sequence of the differences in length of w after each step of our algorithm. I.e. d_j is the difference between length of w after j-th step and $(j-1)$-th step. For a fixed execution, we will refer to a pair (d, w) - sequence of differences and the final word as a *log*.

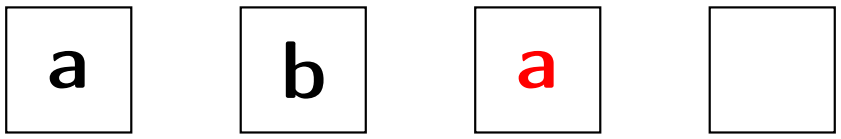


Diff	1	1	1	-1	1
Choice					c

Proof - cont.

Log

Let $c_1, c_2, \ldots, c_M$ be the choices of characters throughout the execution. Let's analyze the length of the word w throughout the execution. At each step, it can increase by one or decrease by arbitrary value s.t. w is non-empty. Let $d_1 = 1, d_2, \ldots, d_M$ be the sequence of the differences in length of w after each step of our algorithm. I.e. d_j is the difference between length of w after j-th step and $(j-1)$-th step. For a fixed execution, we will refer to a pair (d, w) - sequence of differences and the final word as a *log*.

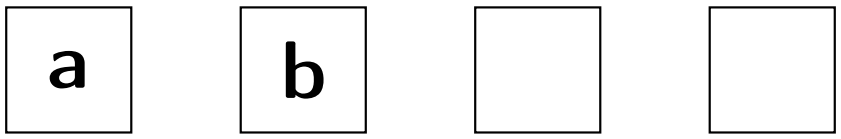


Diff	1	1	1	-1	1
Choice				b	c

Proof - cont.

Log

Let $c_1, c_2, \ldots, c_M$ be the choices of characters throughout the execution. Let's analyze the length of the word w throughout the execution. At each step, it can increase by one or decrease by arbitrary value s.t. w is non-empty. Let $d_1 = 1, d_2, \ldots, d_M$ be the sequence of the differences in length of w after each step of our algorithm. I.e. d_j is the difference between length of w after j-th step and $(j-1)$-th step. For a fixed execution, we will refer to a pair (d, w) - sequence of differences and the final word as a *log*.

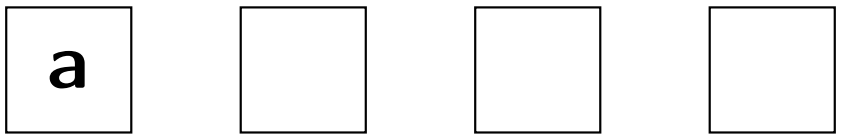


Diff	1	1	1	-1	1
Choice			a	b	c

Proof - cont.

Log

Let $c_1, c_2, \ldots, c_M$ be the choices of characters throughout the execution. Let's analyze the length of the word w throughout the execution. At each step, it can increase by one or decrease by arbitrary value s.t. w is non-empty. Let $d_1 = 1, d_2, \ldots, d_M$ be the sequence of the differences in length of w after each step of our algorithm. I.e. d_j is the difference between length of w after j-th step and $(j-1)$-th step. For a fixed execution, we will refer to a pair (d, w) - sequence of differences and the final word as a *log*.

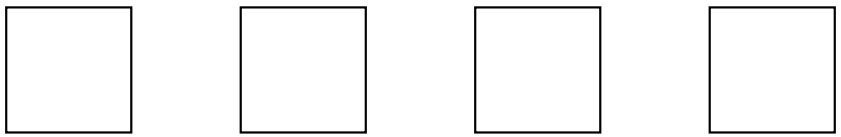


Diff	1	1	1	-1	1
Choice		b	a	b	c

Proof - cont.

Log

Let $c_1, c_2, \ldots, c_M$ be the choices of characters throughout the execution. Let's analyze the length of the word w throughout the execution. At each step, it can increase by one or decrease by arbitrary value s.t. w is non-empty. Let $d_1 = 1, d_2, \ldots, d_M$ be the sequence of the differences in length of w after each step of our algorithm. I.e. d_j is the difference between length of w after j-th step and $(j-1)$-th step. For a fixed execution, we will refer to a pair (d, w) - sequence of differences and the final word as a *log*.



1 2 3 4

Diff	1	1	1	-1	1
Choice	a	b	a	b	c

Proof - cont.

Since each execution can produce a log and every log corresponds to some execution, there is equal number of each of them.

Proof - cont.

Since each execution can produce a log and every log corresponds to some execution, there is equal number of each of them. The number of sequences d can be bounded by $n \cdot C_M$, where C_M is the M-th Catalan number. Hence the number of logs can be bounded by $4^n \cdot n \cdot C_M$. Let's note that $C_M = o(4^M)$. Hence:

$$4^M \leq n \cdot C_M \cdot 4^n$$

Proof - cont.

Since each execution can produce a log and every log corresponds to some execution, there is equal number of each of them. The number of sequences d can be bounded by $n \cdot C_M$, where C_M is the M-th Catalan number. Hence the number of logs can be bounded by $4^n \cdot n \cdot C_M$. Let's note that $C_M = o(4^M)$. Hence:

$$4^M \leq n \cdot C_M \cdot 4^n = o(4^M)$$

Hence there exists a valid assignment regardless of the choice of the lists. □