

Learnability of Algorithms

Hardness and Positive Results

Kimion Fountoulakis
Associate Professor @CS U. Waterloo

Paper 1: *On the Statistical Query Complexity of Learning Semiautomata: a Random Walk Approach.* COLT 2026. G. Giapitzakis, K. Fountoulakis, E. Nichani, J. D. Lee .

Paper 2: *Learning to Add, Multiply, and Execute Algorithmic Instructions Exactly with Neural Networks.* NeurIPS 2025. A. Back de Luca, G. Giapitzakis, K. Fountoulakis.

Motivation and big picture

What can learn learning algorithms learn efficiently?

Fundamental limits and capabilities.

Where do fundamental barriers arise? Structural hardness vs. constructive learning.

Relevance to practitioners.

Arithmetic as a benchmark for reasoning

From 2014

To 2025

LEARNING TO EXECUTE

Wojciech Zaremba*
New York University
woj.zaremba@gmail.com

Ilya Sutskever
Google
ilyasu@google.com

ABSTRACT

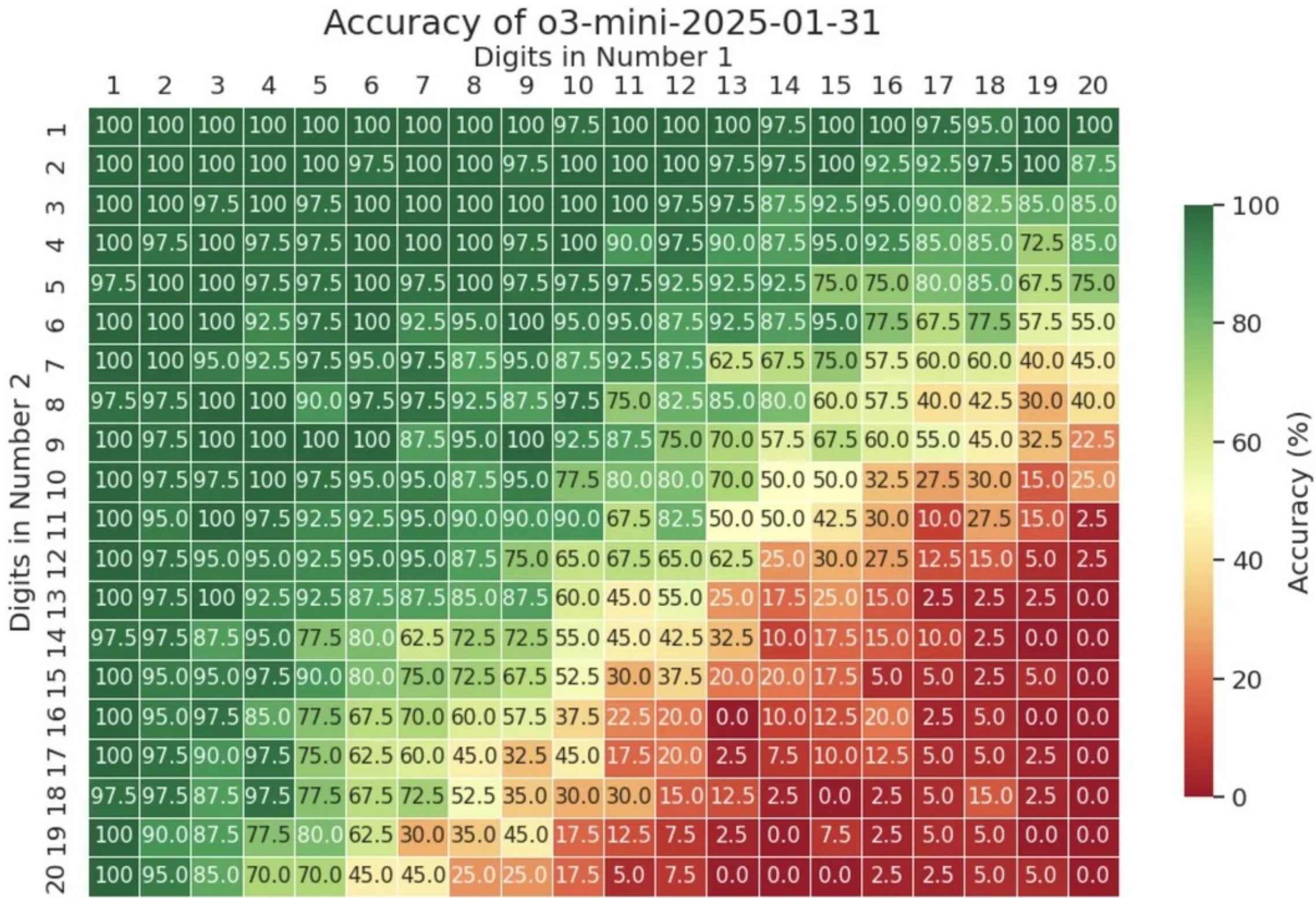
Recurrent Neural Networks (RNNs) with Long Short-Term Memory units (LSTM) are widely used because they are expressive and are easy to train. Our interest lies in empirically evaluating the expressiveness and the learnability of LSTMs in the sequence-to-sequence regime by training them to evaluate short computer programs, a domain that has traditionally been seen as too complex for neural networks. We consider a simple class of programs that can be evaluated with a single left-to-right pass using constant memory. Our main result is that LSTMs can learn to map the character-level representations of such programs to their correct outputs. Notably, it was necessary to use curriculum learning, and while conventional curriculum learning proved ineffective, we developed a new variant of curriculum learning that improved our networks’ performance in all experimental conditions. The improved curriculum had a dramatic impact on an addition problem, making it possible to train an LSTM to add two 9-digit numbers with 99% accuracy.

□

1 INTRODUCTION

Execution of computer programs requires dealing with a number of nontrivial concepts. To execute a program, a system has to understand numerical operations, if-statements, variable assignments, the compositionality of operations, and many more.

Multiplication



Experiment conducted by [Yuntian](#), early 2025.

Arithmetic as a benchmark for reasoning

From 2014

To 2025

LEARNING TO EXECUTE

Wojciech Zaremba*
New York University
woj.zaremba@gmail.com

Ilya Sutskever
Google
ilyasu@google.com

ABSTRACT

1 INTRODUCTION

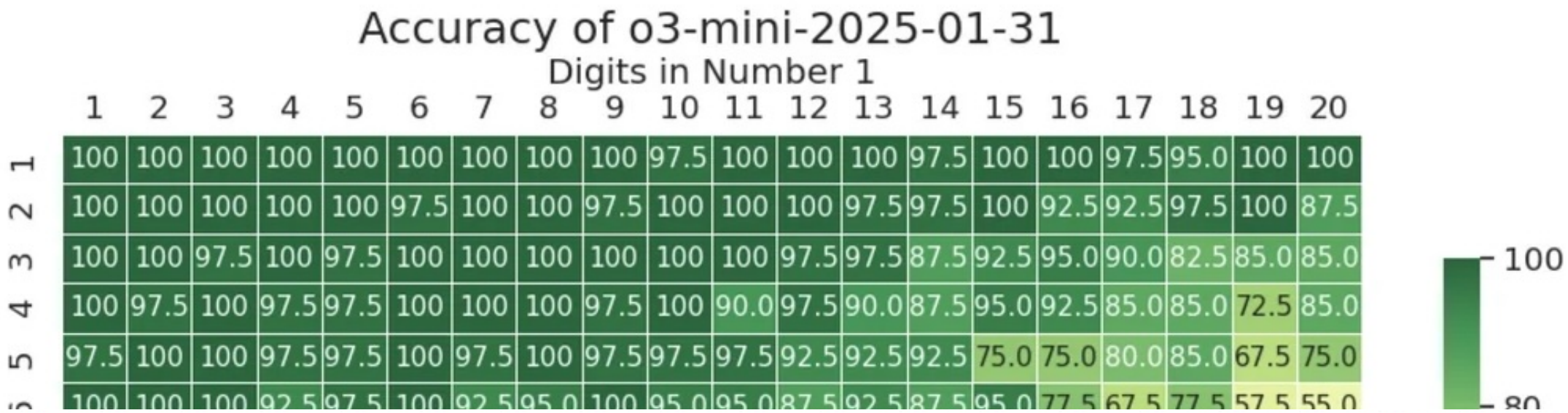
Execution of computer programs requires dealing with a number of nontrivial concepts. To execute a program, a system has to understand numerical operations, if-statements, variable assignments, the compositionality of operations, and many more.

□

1 INTRODUCTION

Execution of computer programs requires dealing with a number of nontrivial concepts. To execute a program, a system has to understand numerical operations, if-statements, variable assignments, the compositionality of operations, and many more.

Multiplication



Experiment conducted by [Yuntian](#), early 2025.

Two papers at a glance

Paper 1: *On the Statistical Query Complexity of Learning Semiautomata: a Random Walk Approach.* [arXiv:2510.04115](https://arxiv.org/abs/2510.04115) (October, 2025). G. Giapitzakis, K. Fountoulakis, E. Nichani, J. D. Lee .

Algorithms: shuffling machines.

End-to-end learning with statistical queries is hard.

Paper 2: *Learning to Add, Multiply, and Execute Algorithmic Instructions Exactly with Neural Networks.* NeurIPS 2025. A. Back de Luca, G. Giapitzakis, K. Fountoulakis.

Two papers at a glance

Paper 1: *On the Statistical Query Complexity of Learning Semiautomata: a Random Walk Approach.* [arXiv:2510.04115](https://arxiv.org/abs/2510.04115) (October, 2025). G. Giapitzakis, K. Fountoulakis, E. Nichani, J. D. Lee .

Algorithms: shuffling machines.

End-to-end learning with statistical queries is hard.

Paper 2: *Learning to Add, Multiply, and Execute Algorithmic Instructions Exactly with Neural Networks.* NeurIPS 2025. A. Back de Luca, G. Giapitzakis, K. Fountoulakis.

Step-by-step learning with gradient descent is efficient.

and exact!

Models: Neural networks

Learning Instructions Step-by-Step with Gradient Descent is Exact and Efficient

 A. Back de Luca, G. Giapitzakis, K. Fountoulakis. *Learning to Add, Multiply, and Execute Algorithmic Instructions Exactly with Neural Networks*. NeurIPS 2025.

Architecture and training

Two-layer feedforward network: $f(x) = W_2 \text{ReLU}(W_1 x)$

We will study this in the *infinite-width limit* (# of hidden neurons going to *infinity*), trained by gradient-descent.

Architecture and training

Two-layer feedforward network: $f(x) = W_2 \text{ReLU}(W_1 x)$

We will study this in the *infinite-width limit* (# of hidden neurons going to *infinity*), trained by gradient-descent.

Popular setting for understanding the training dynamics

[\[1806.07572\] Neural Tangent Kernel: Convergence and ...](#)

by A Jacot · 2018 · Cited by 4123 — This **kernel** is central to describe the generalization of ANNs. While the NTK is random at initialization and varies during training, in the infinite-width limit it converges to a deterministic kernel.

In our paper, we are going to use this setting for proving *exact learning*.

Example: addition (ripple-carry adder)

$$\begin{array}{r} 10 \\ + 11 \\ \hline \end{array}$$

Instruction

bitwise sum
step

Example: addition (ripple-carry adder)

$$\begin{array}{r} 10 \\ +11 \\ \hline \end{array}$$

Instruction

bitwise sum
step

Example: addition (ripple-carry adder)

$$\begin{array}{r} 10 \\ +11 \\ \hline 0 \end{array}$$

Output bit

Instruction

bitwise sum
step

Example: addition (ripple-carry adder)

$$\begin{array}{r} 10 \\ +11 \\ \hline 10 \end{array}$$

Carry Output
bit

Instruction

bitwise sum
step

Example: addition (ripple-carry adder)

$$\begin{array}{r} 10 \\ + 11 \\ \hline 10 \end{array}$$

Carry Output
bit

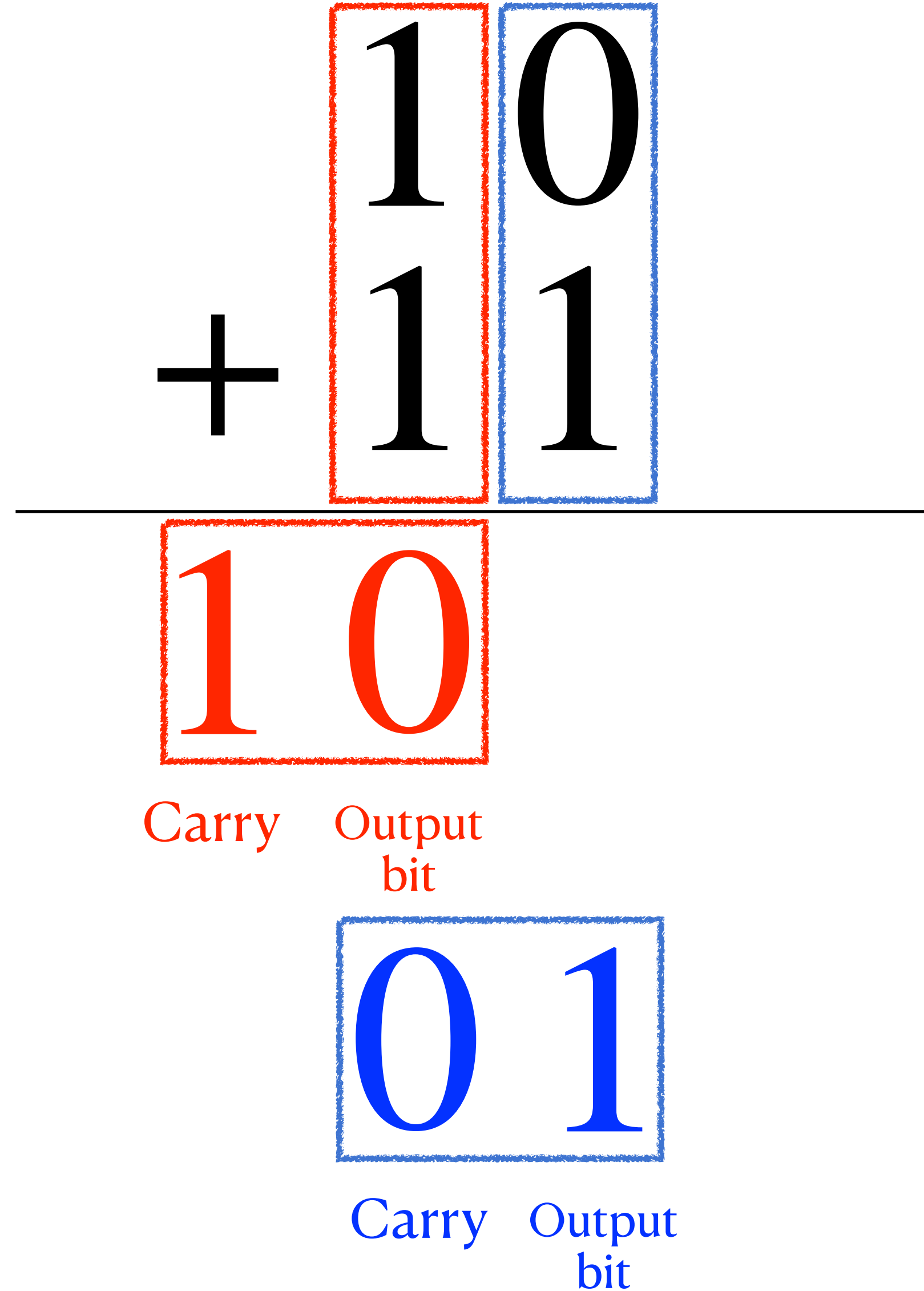
Instruction

bitwise sum
step

Example: addition (ripple-carry adder)

Instruction

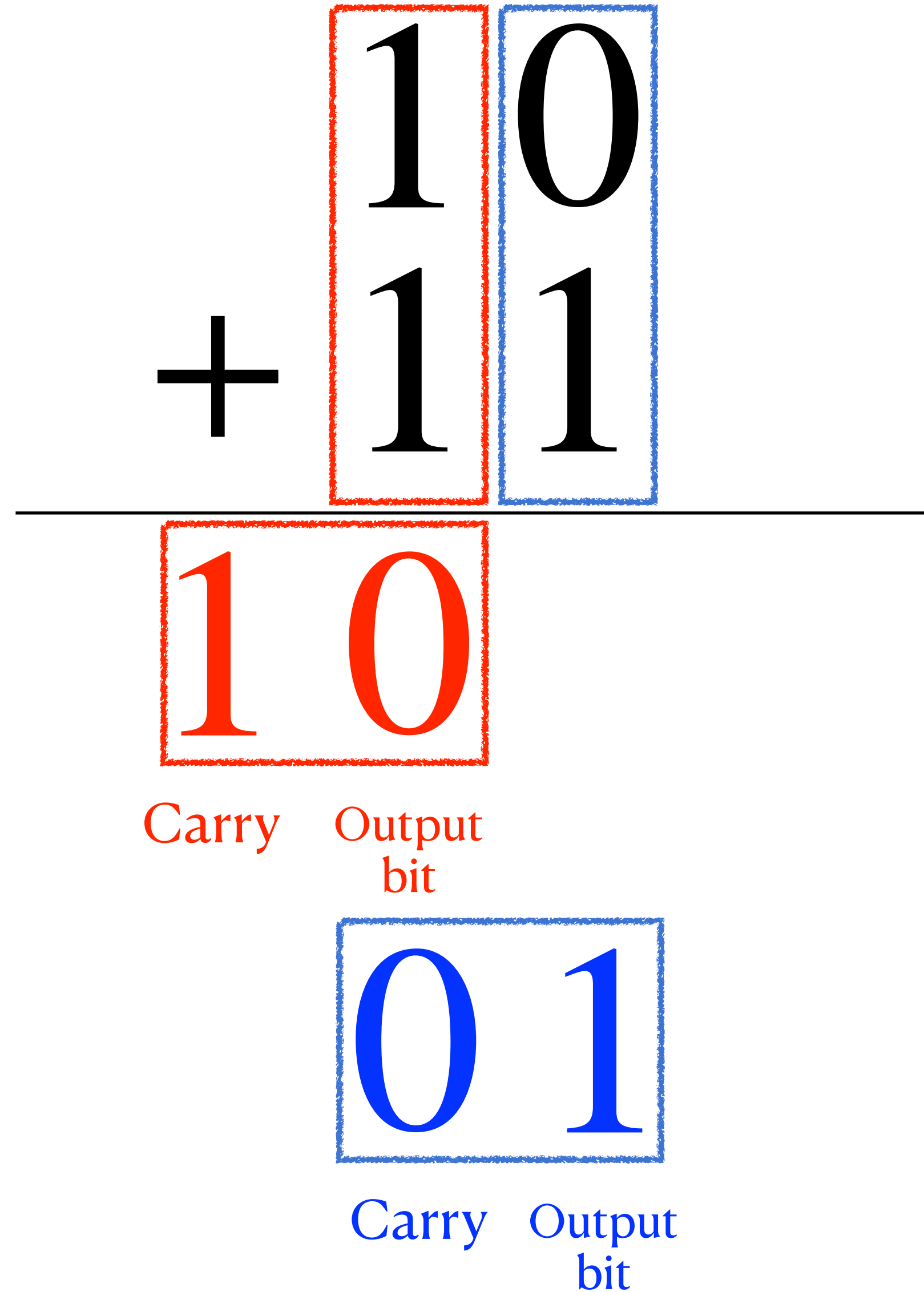
bitwise sum
step



Example: addition (ripple-carry adder)

Instruction

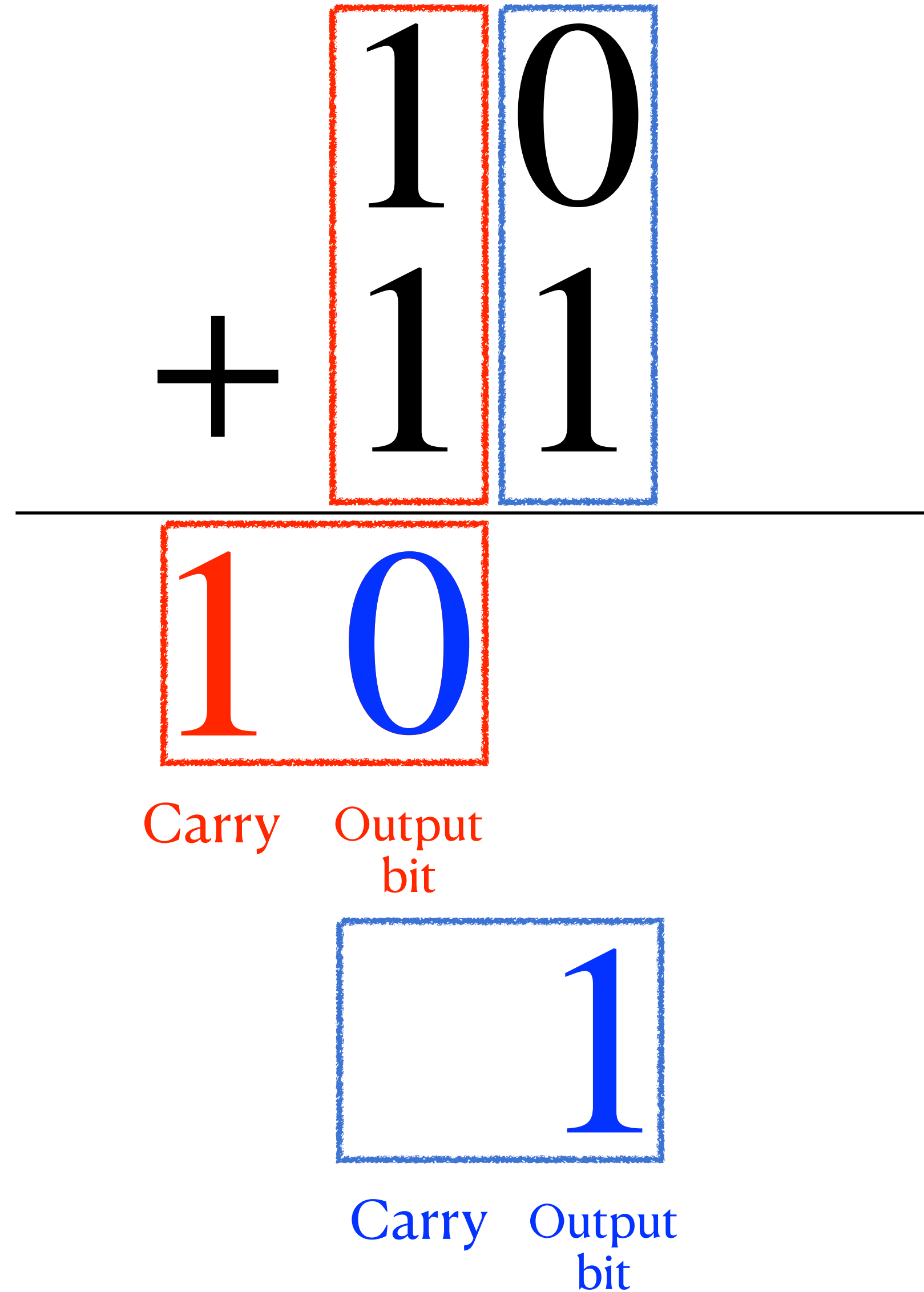
carry propagation
step



Example: addition (ripple-carry adder)

Instruction

carry propagation
step



Example: addition (ripple-carry adder)

$$\begin{array}{r} \\ + \\ \hline 1 \end{array}$$

Carry Output bit Output bit

Instruction

Final output
step

Data representation: example for addition

1 0

1 1

Data representation: example for addition

carry of first number

c_2

1st bit of first number

p_2

1

1st bit of second number

q_2

1

carry of second number

c_1

2nd bit of first number

p_1

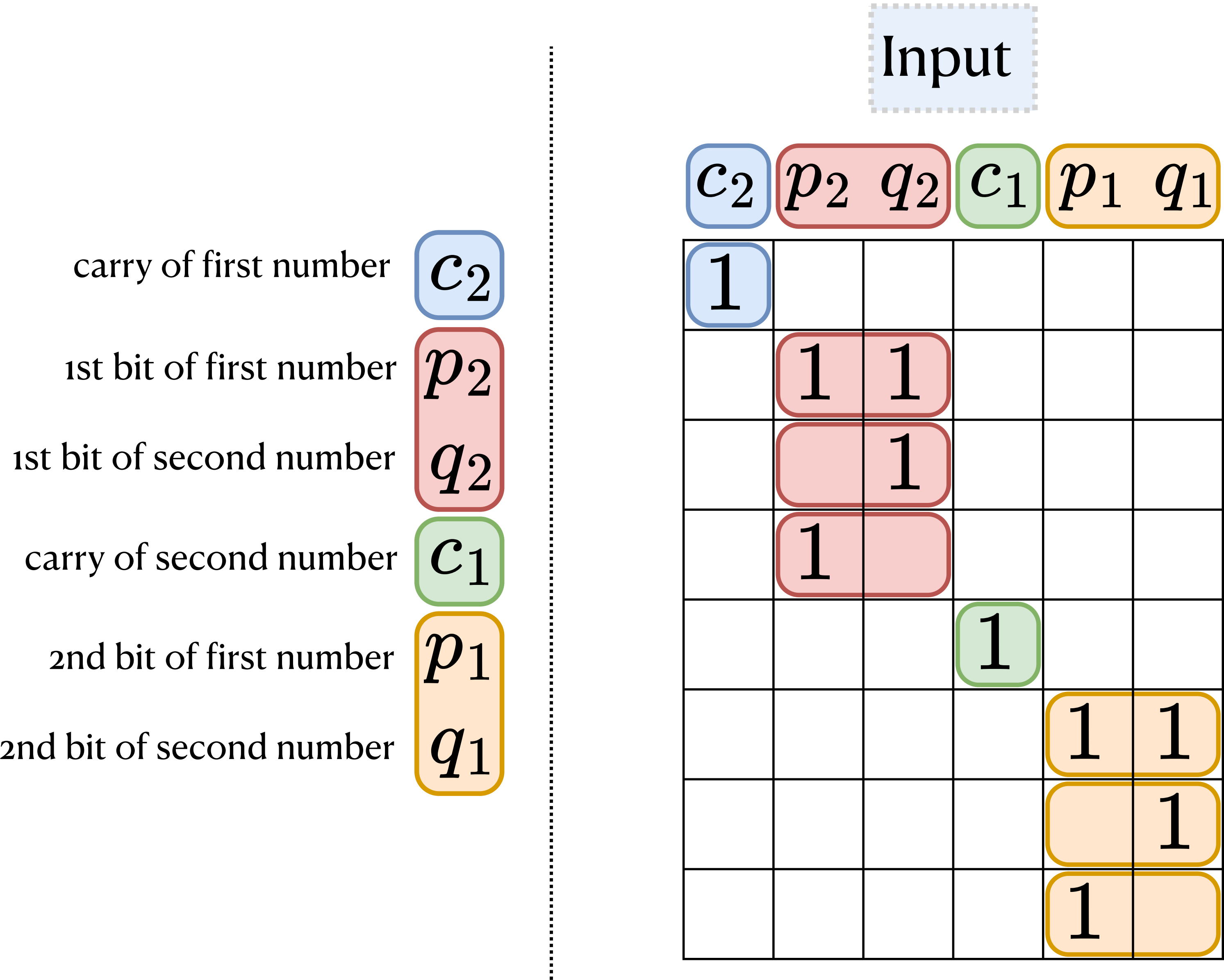
0

2nd bit of second number

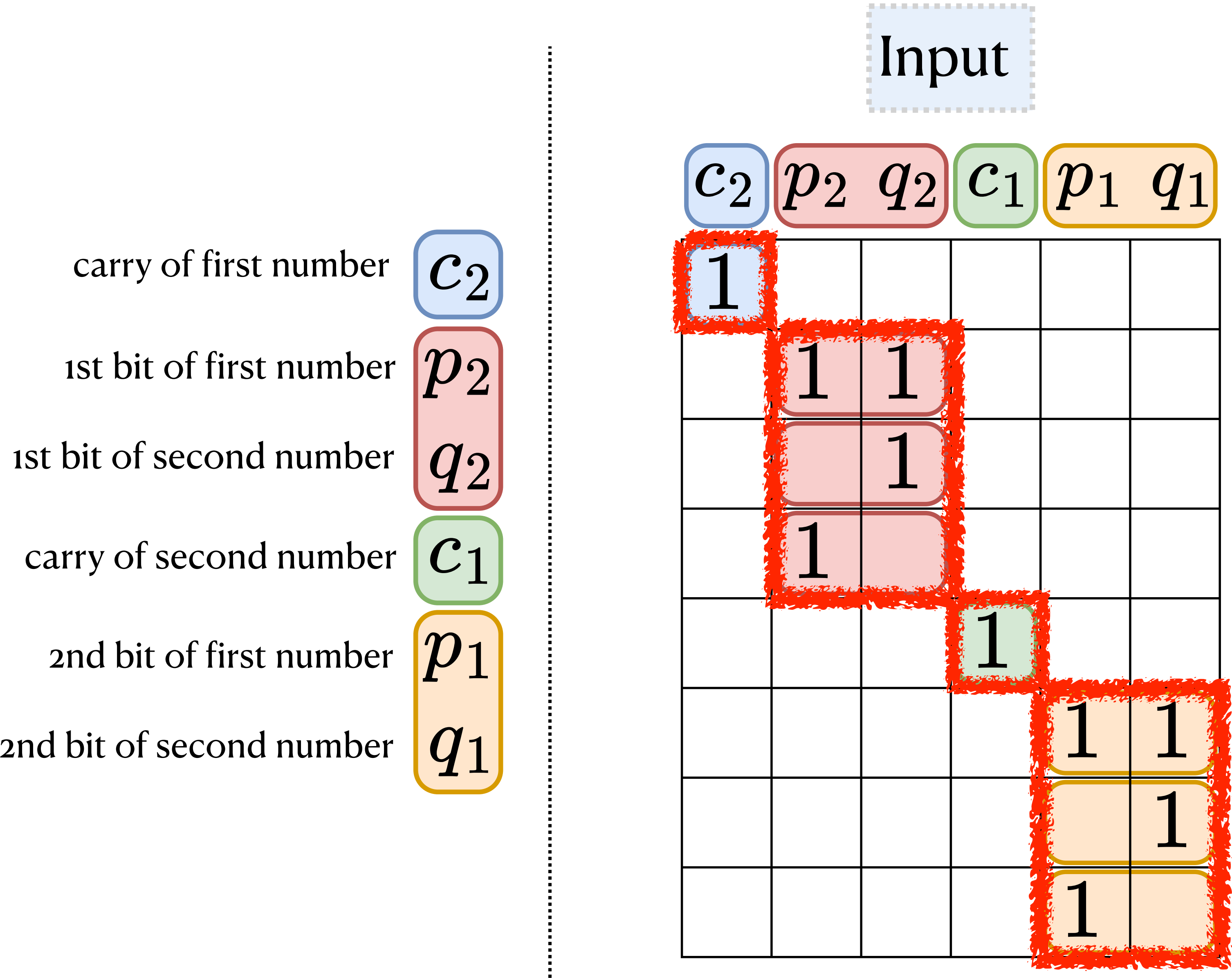
q_1

1

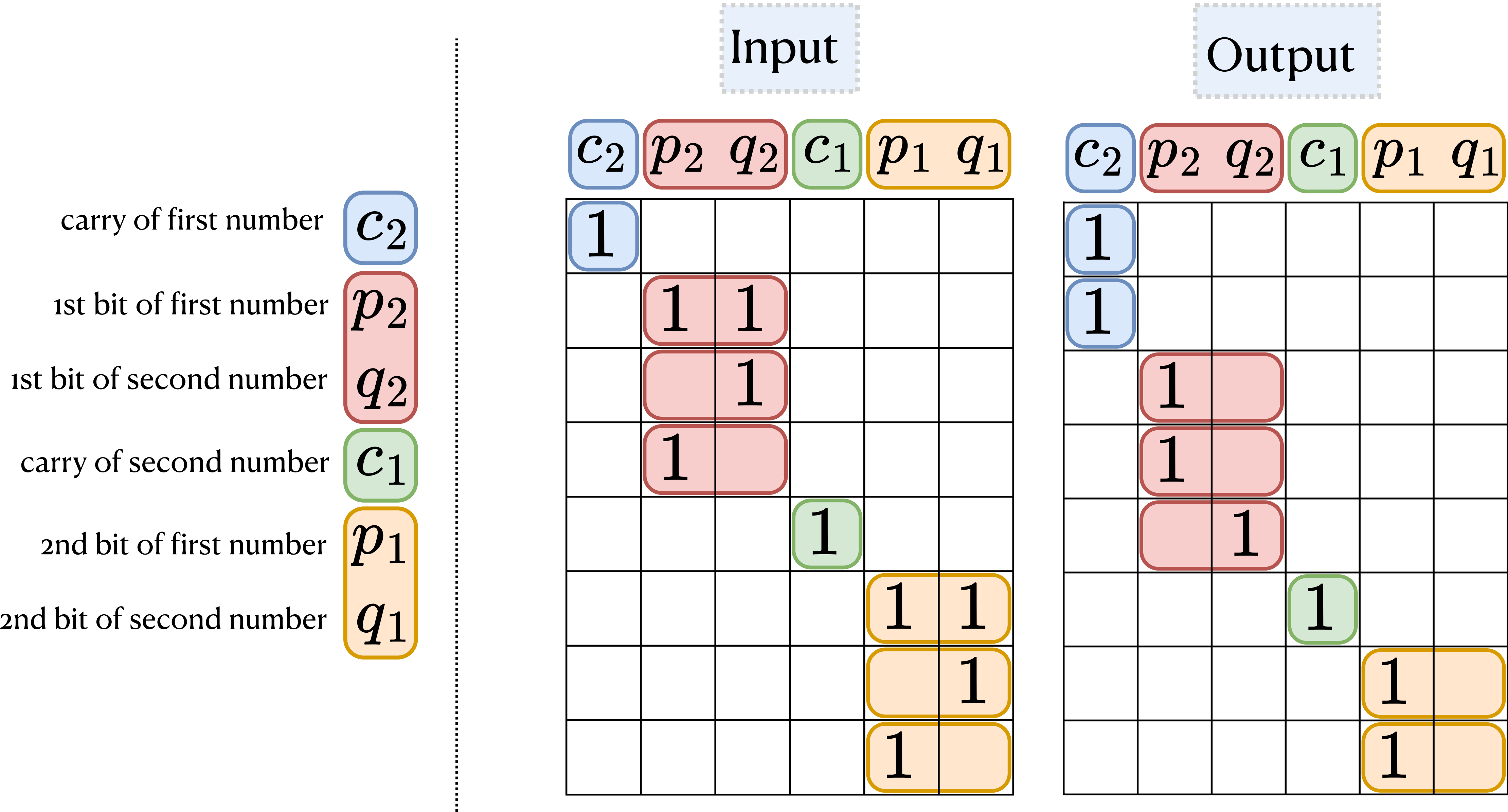
Step-by-step instruction training data for addition



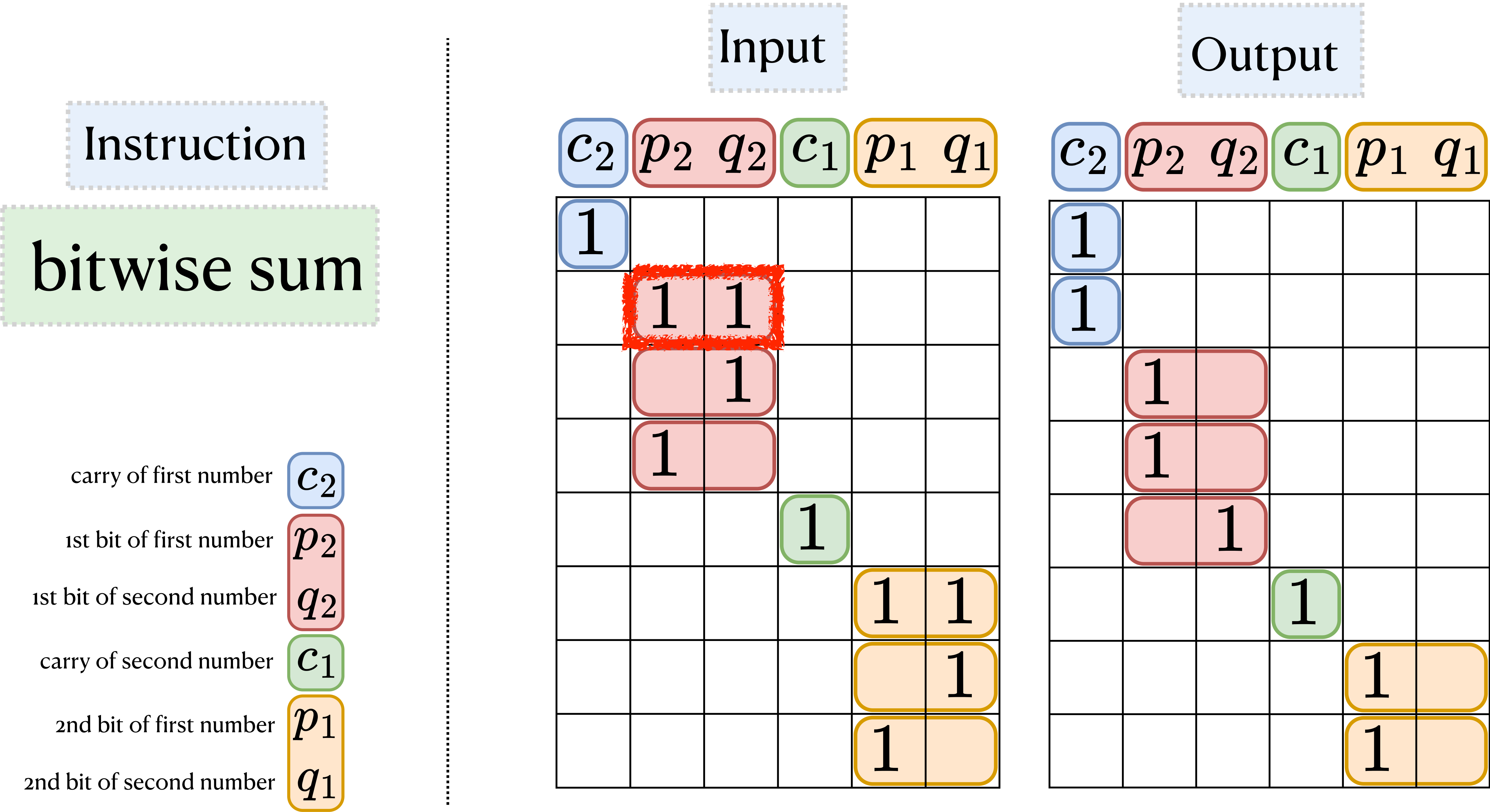
Step-by-step instruction training data for addition



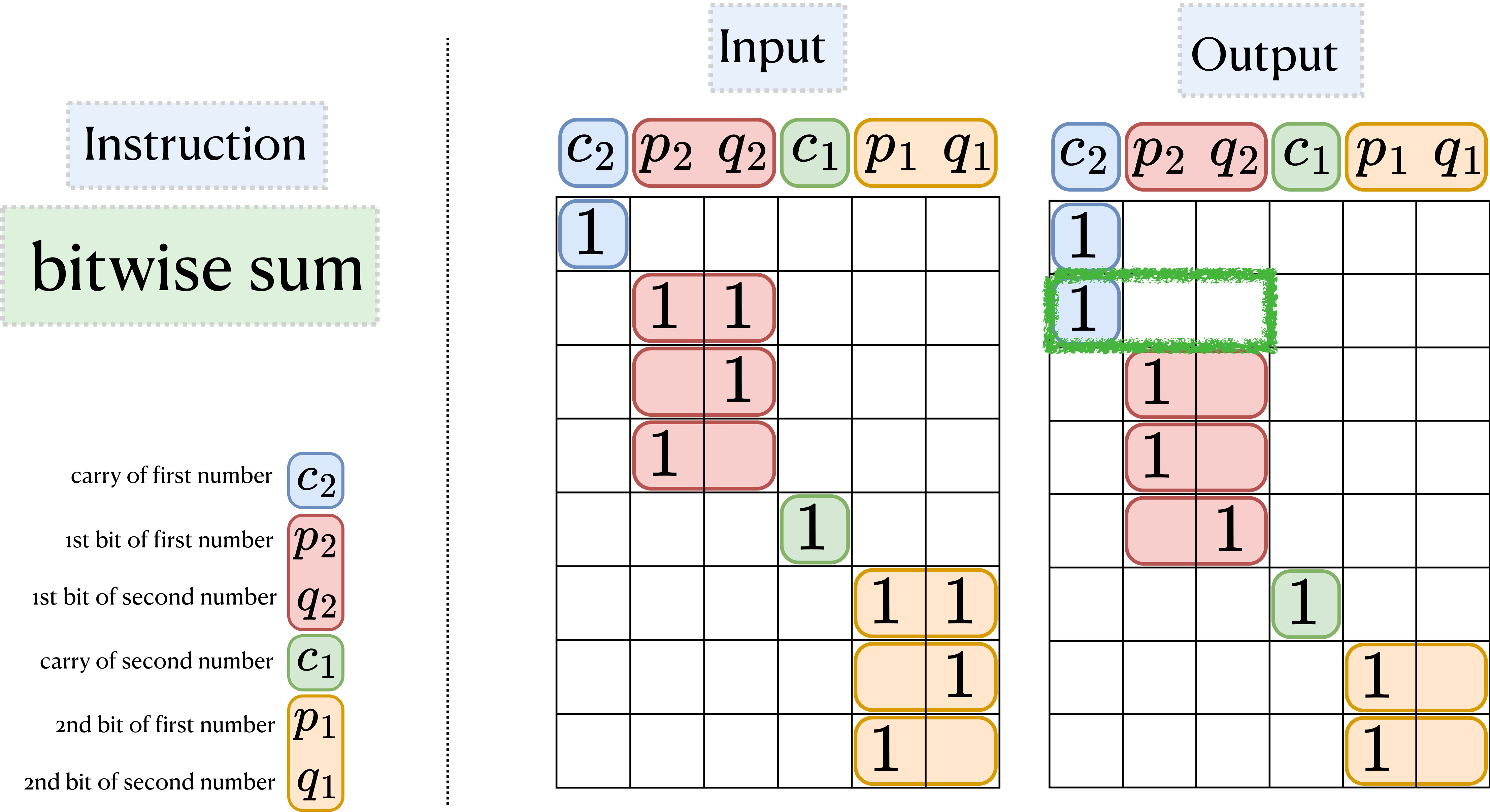
Step-by-step instruction training data for addition



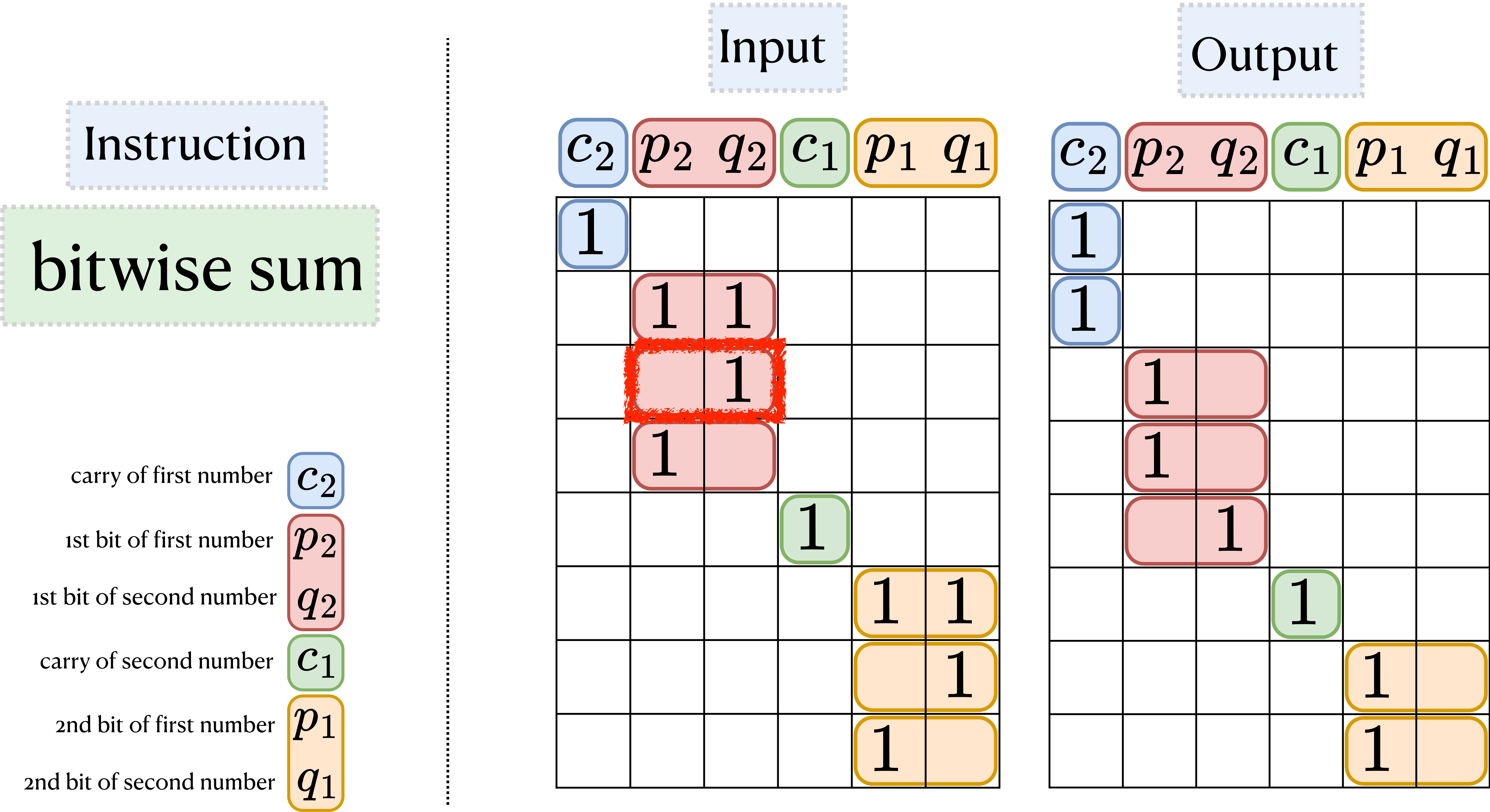
Step-by-step instruction training data for addition



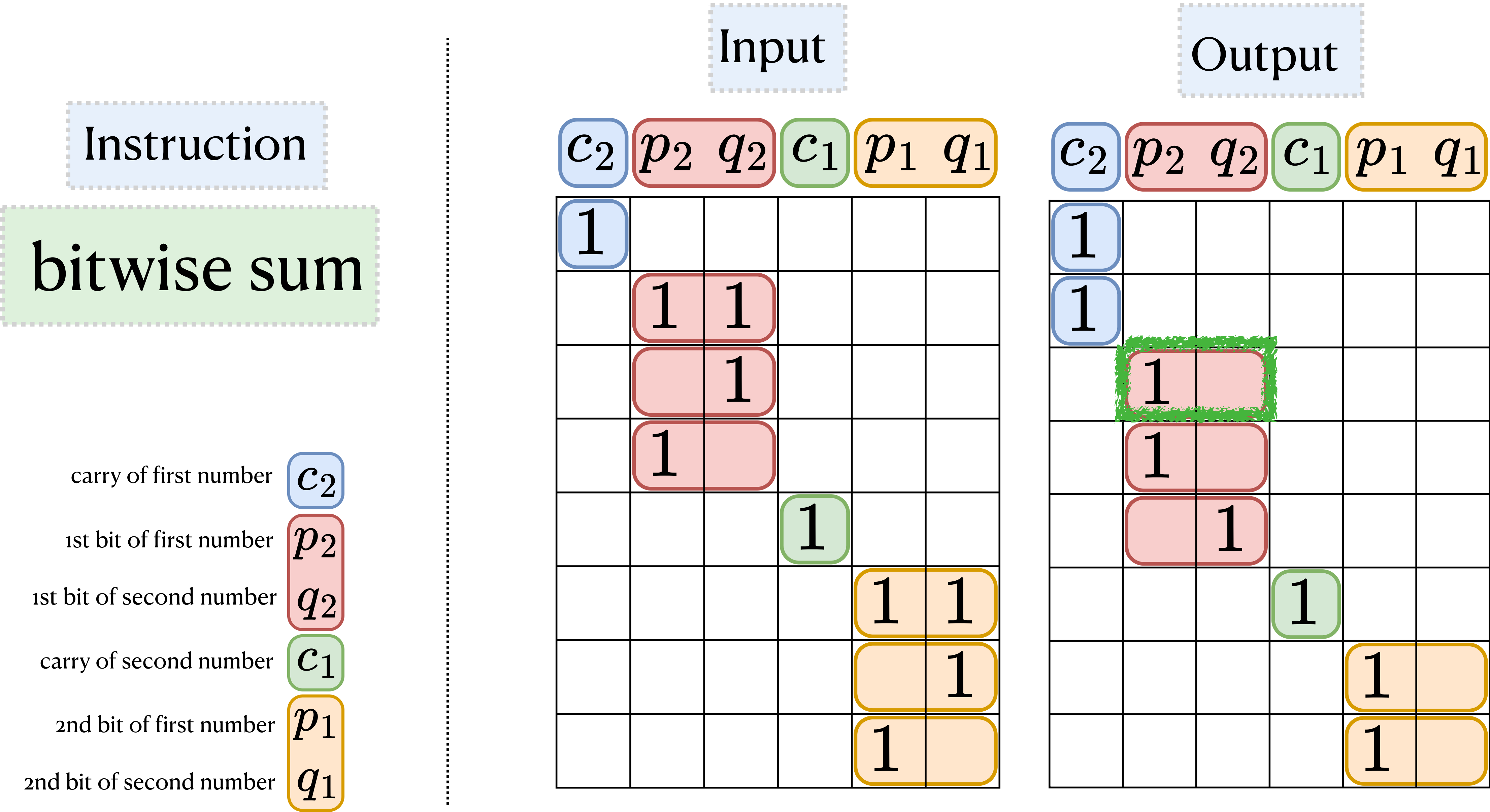
Step-by-step instruction training data for addition



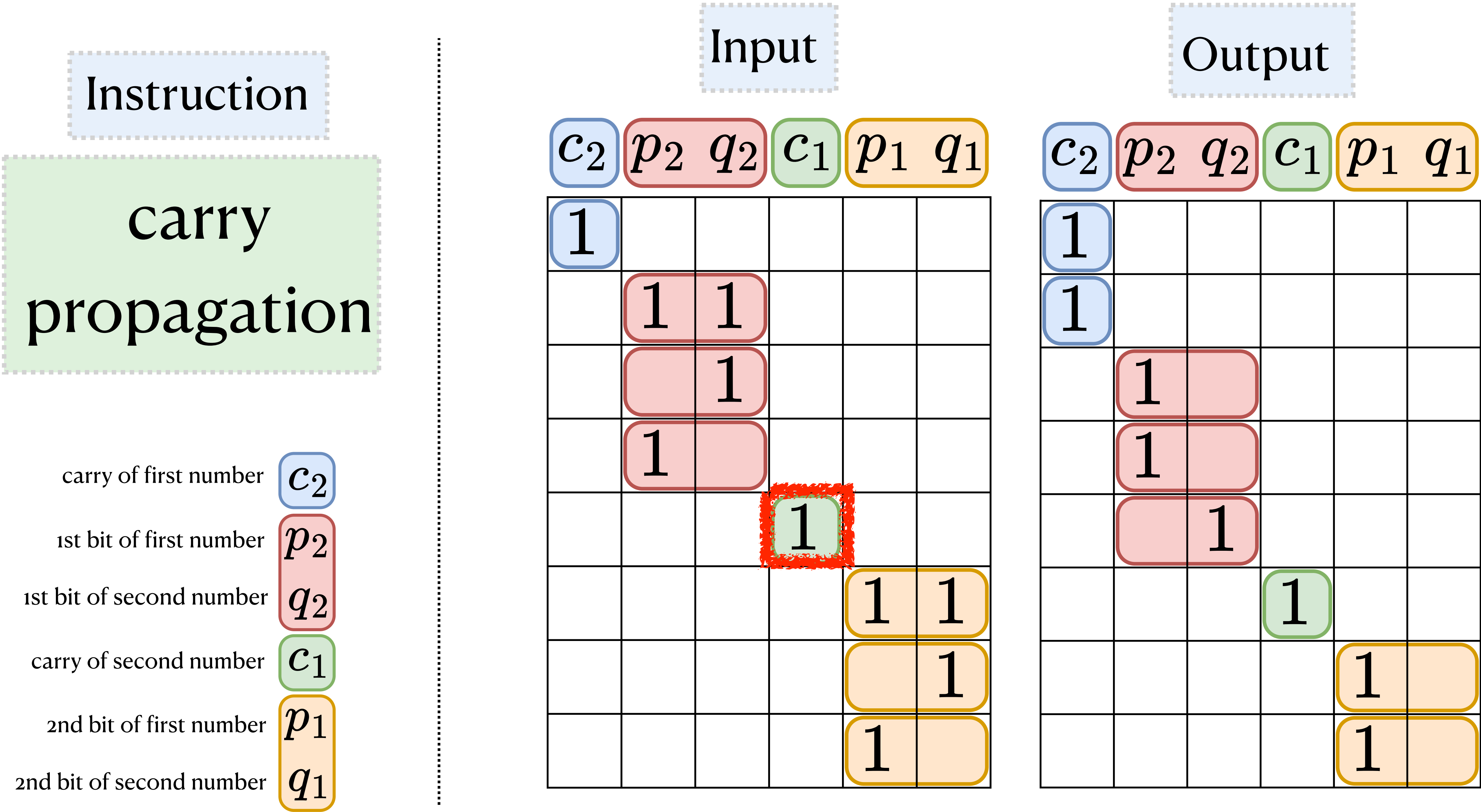
Step-by-step instruction training data for addition



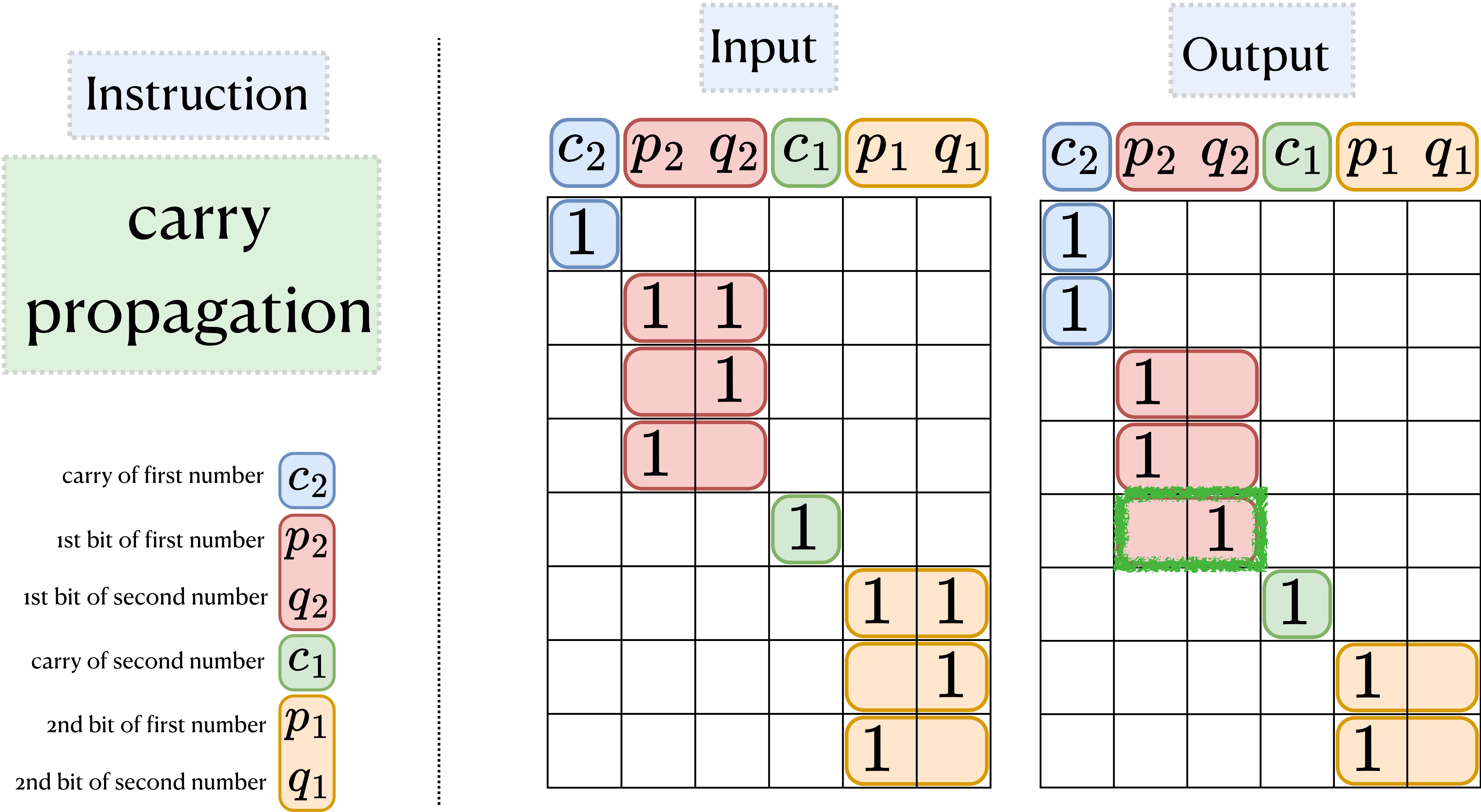
Step-by-step instruction training data for addition



Step-by-step instruction training data for addition



Step-by-step instruction training data for addition



Inference

Algorithm

```
1 function binary_addition(p, q)
2   #  $p, q \in \{0, 1\}^\ell$ 
3    $c = [0]_{i=1}^\ell$ 
4   for  $i=1$  to  $\ell$ :
5      $c, p, q = \text{bitwise\_sum}(p, q)$ 
6      $c, p, q = \text{carry\_propagation}(p, q)$ 
7   end for
8   return  $[c_\ell \mid p]$ 
```

Inference

Algorithm

```
1 function binary_addition(p, q)
2   #  $p, q \in \{0, 1\}^\ell$ 
3    $c = [0]_{i=1}^\ell$ 
4   for  $i=1$  to  $\ell$ :
5      $c, p, q = \text{bitwise\_sum}(p, q)$ 
6      $c, p, q = \text{carry\_propagation}(p, q)$ 
7   end for
8   return  $[c_\ell \mid p]$ 
```

f

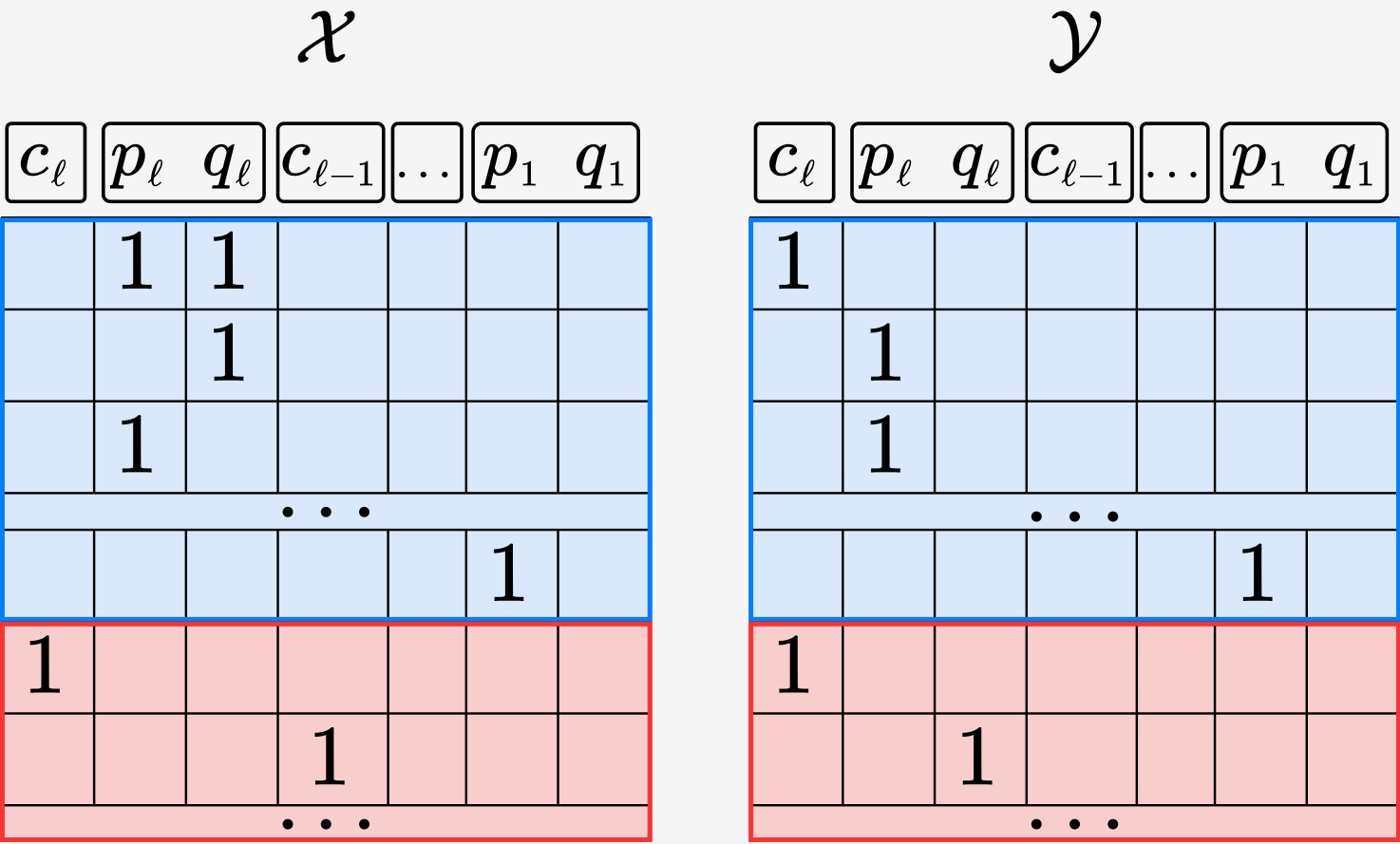
Inference

Algorithm

```
1 function binary_addition(p,q)
2   # p,q ∈ {0,1}^ℓ
3   c = [0]_{i=1}^ℓ
4   for i=1 to ℓ:
5     c,p,q = bitwise_sum(p,q)
6     c,p,q = carry_propagation(p,q)
7   end for
8   return [c_ℓ | p]
```

f

Representation



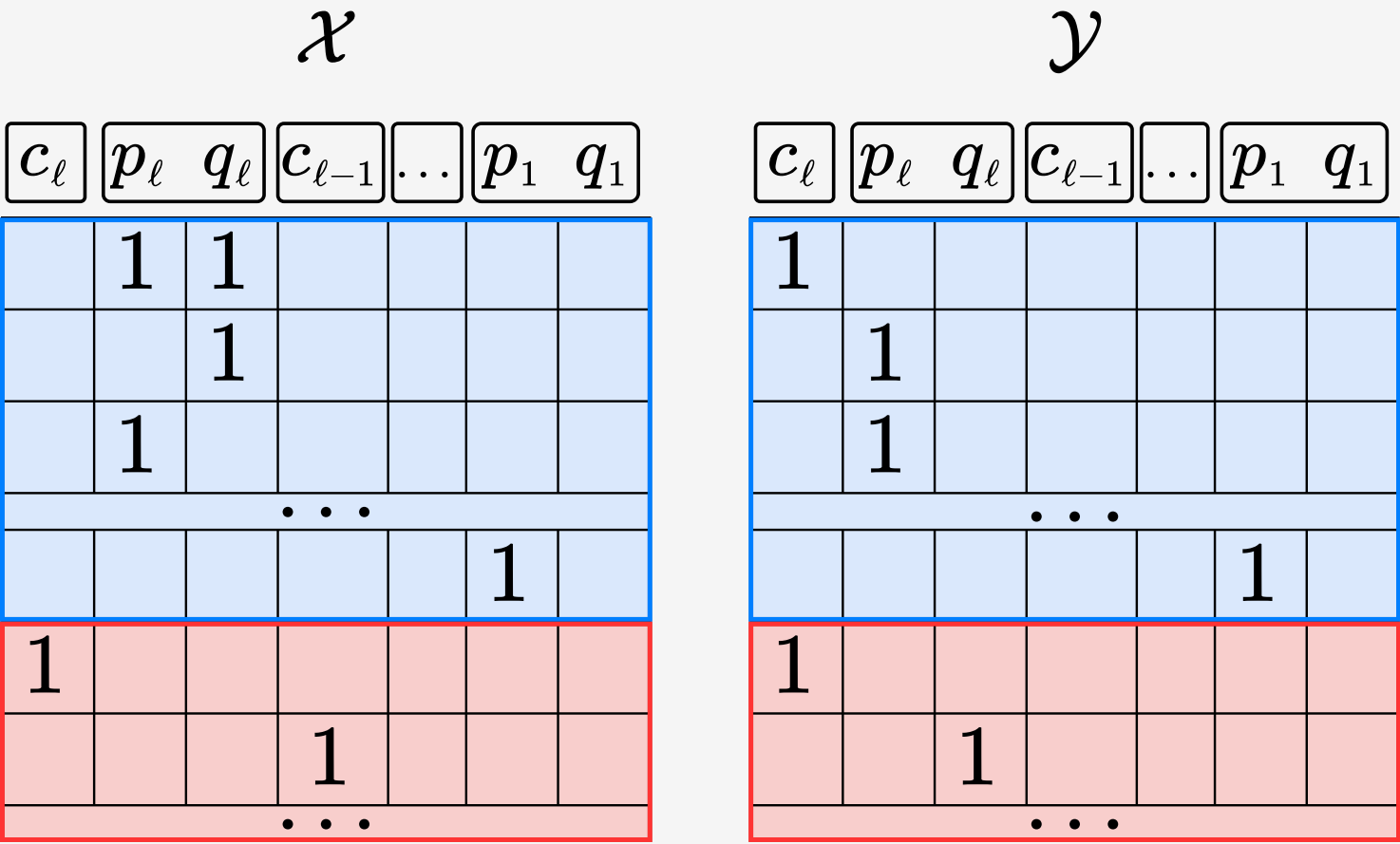
Inference

Algorithm

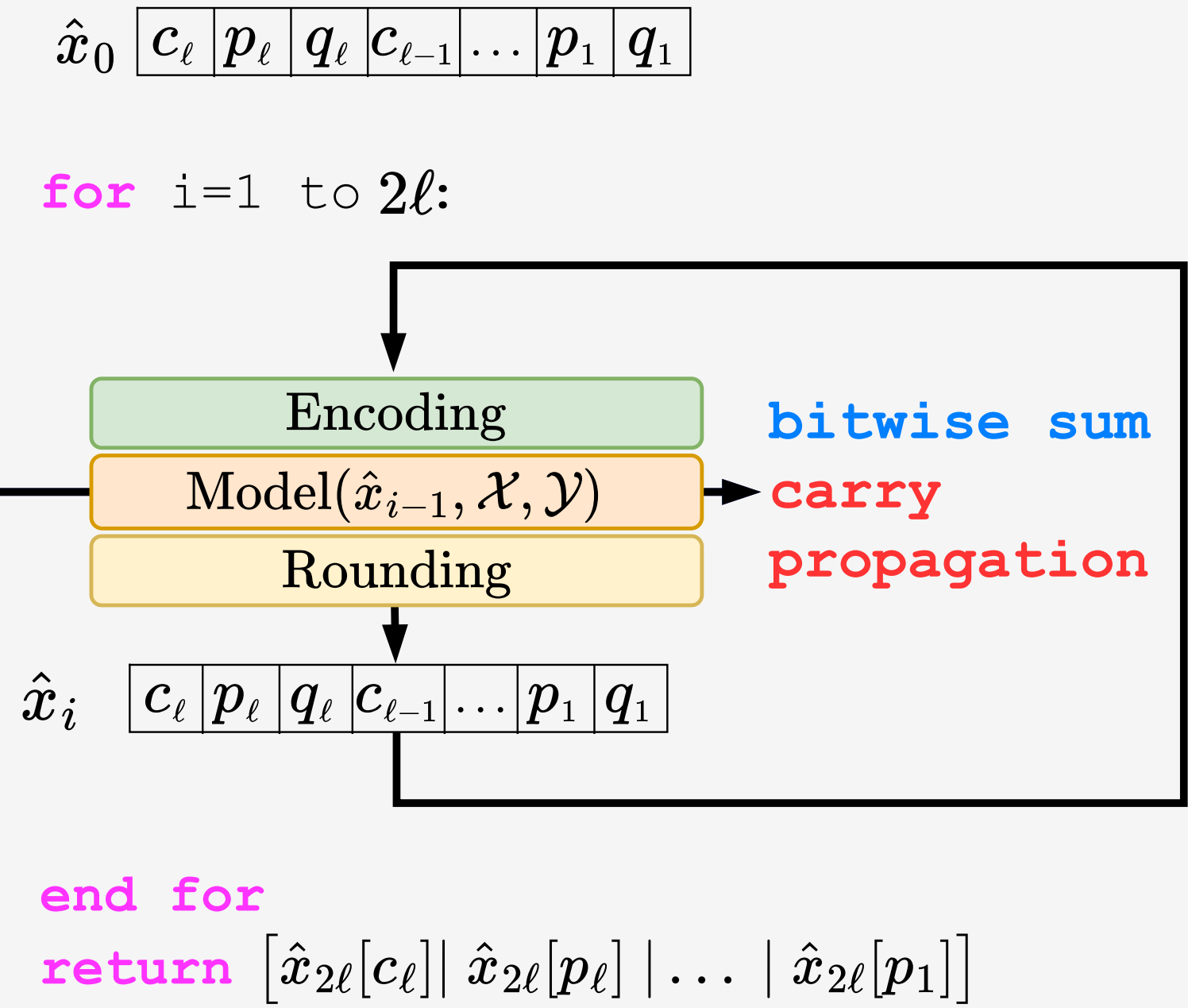
```
1 function binary_addition(p,q)
2   #  $p, q \in \{0,1\}^\ell$ 
3    $c = [0]_{i=1}^\ell$ 
4   for  $i=1$  to  $\ell$ :
5      $c, p, q = \text{bitwise\_sum}(p, q)$ 
6      $c, p, q = \text{carry\_propagation}(p, q)$ 
7   end for
8   return  $[c_\ell \mid p]$ 
```

f

Representation



Execution



Neural networks as random variables

Then the trained model is a random variable which satisfies:

$$f(x) \sim \mathcal{N}(\mu(x), \Sigma(x))$$

$\mu(x)$: the mean behaviour of the neural network

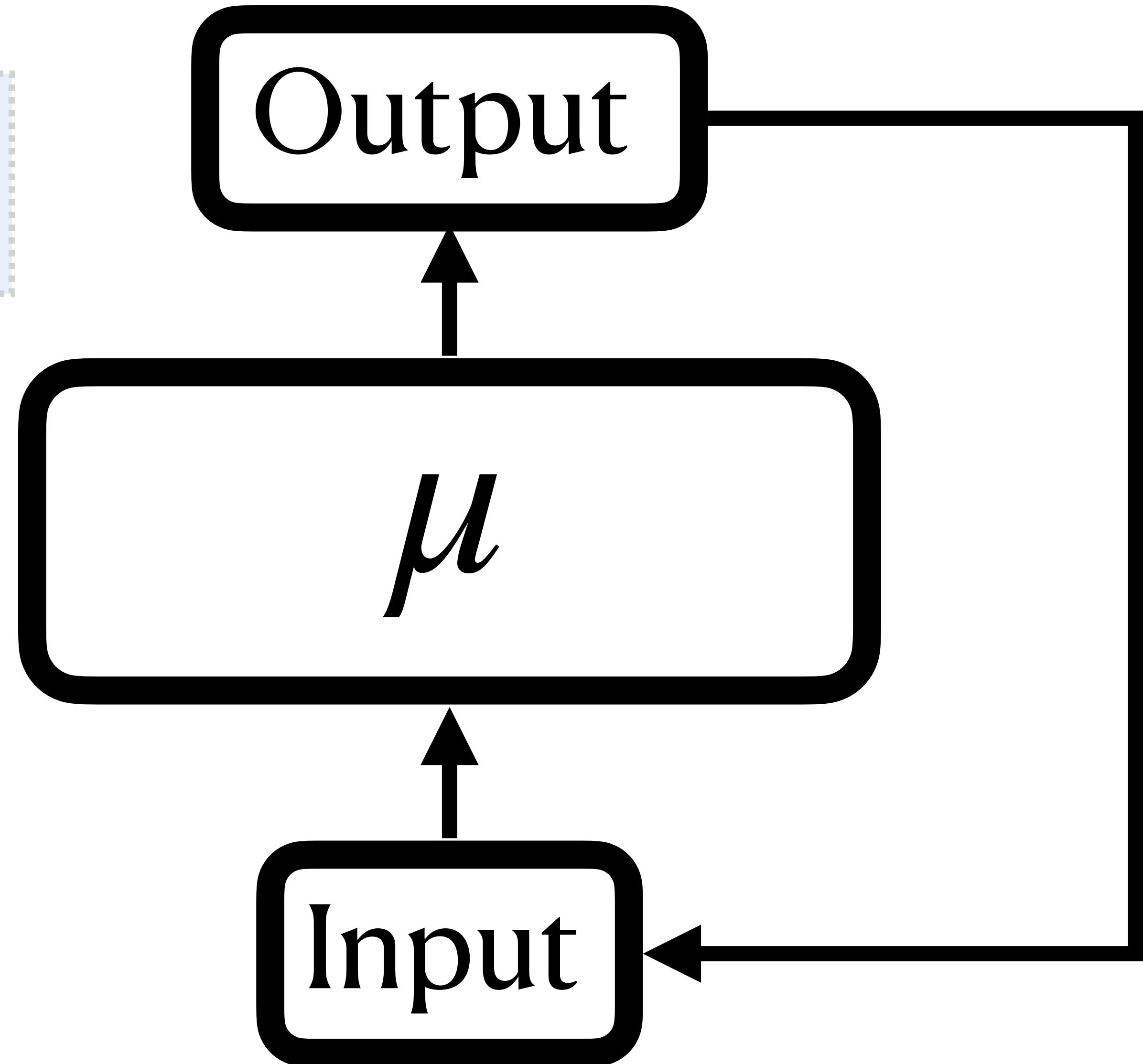
$\Sigma(x)$: the variance of the neural network

Results: the mean $\mu(\hat{x})$

We prove that the mean $\mu(\hat{x})$ in a loop can execute without error:

- Permutation of input binary data
- Addition of binary numbers
- Multiplication of binary numbers
- An one-instruction set computer (OISC)

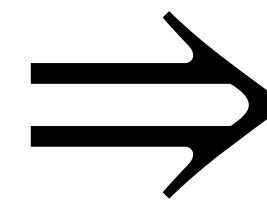
Training data size: logarithmic in all possible inputs.



High level idea of the proof

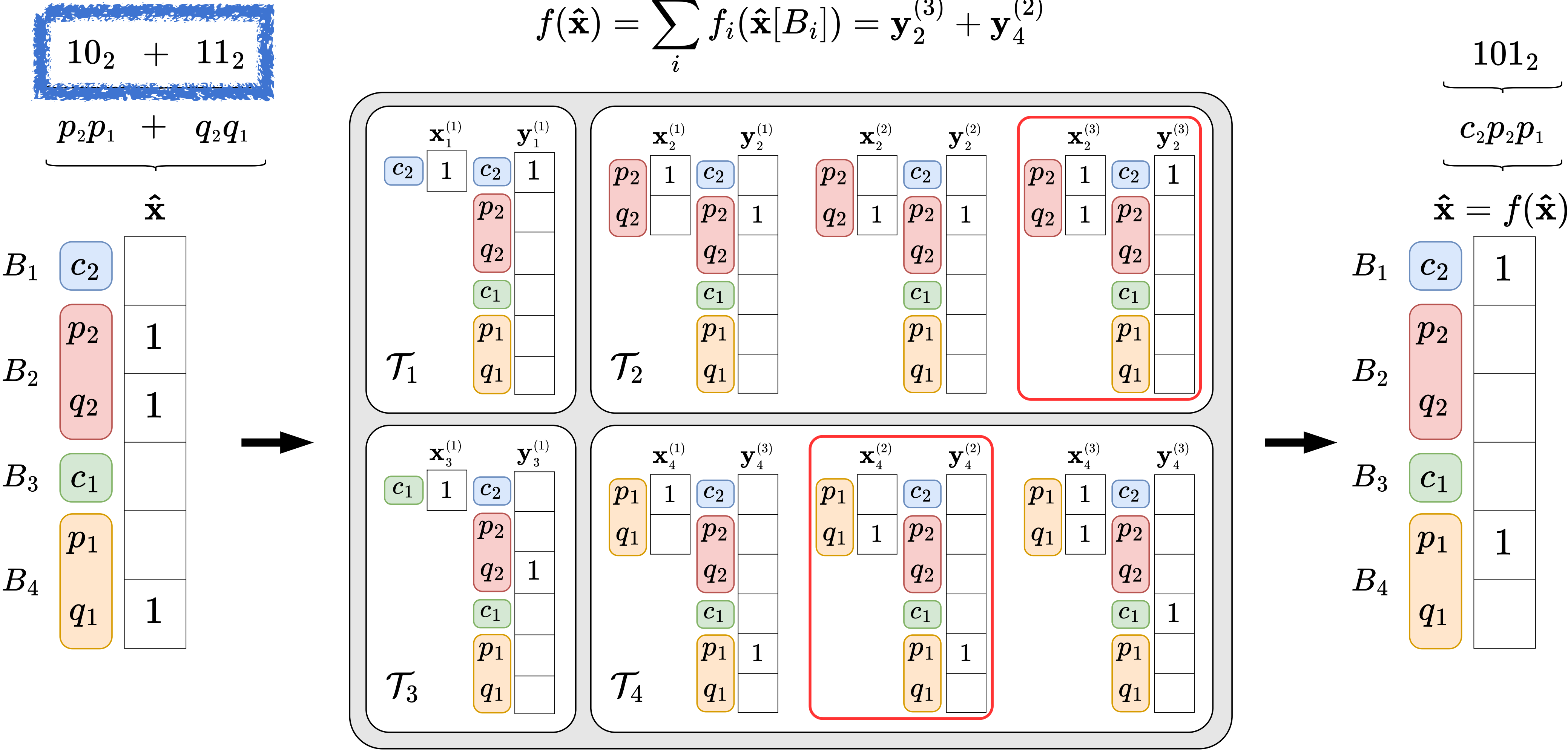
How can we achieve **zero** error?

Train on all possible instructions for
small numbers, **and overfit!**



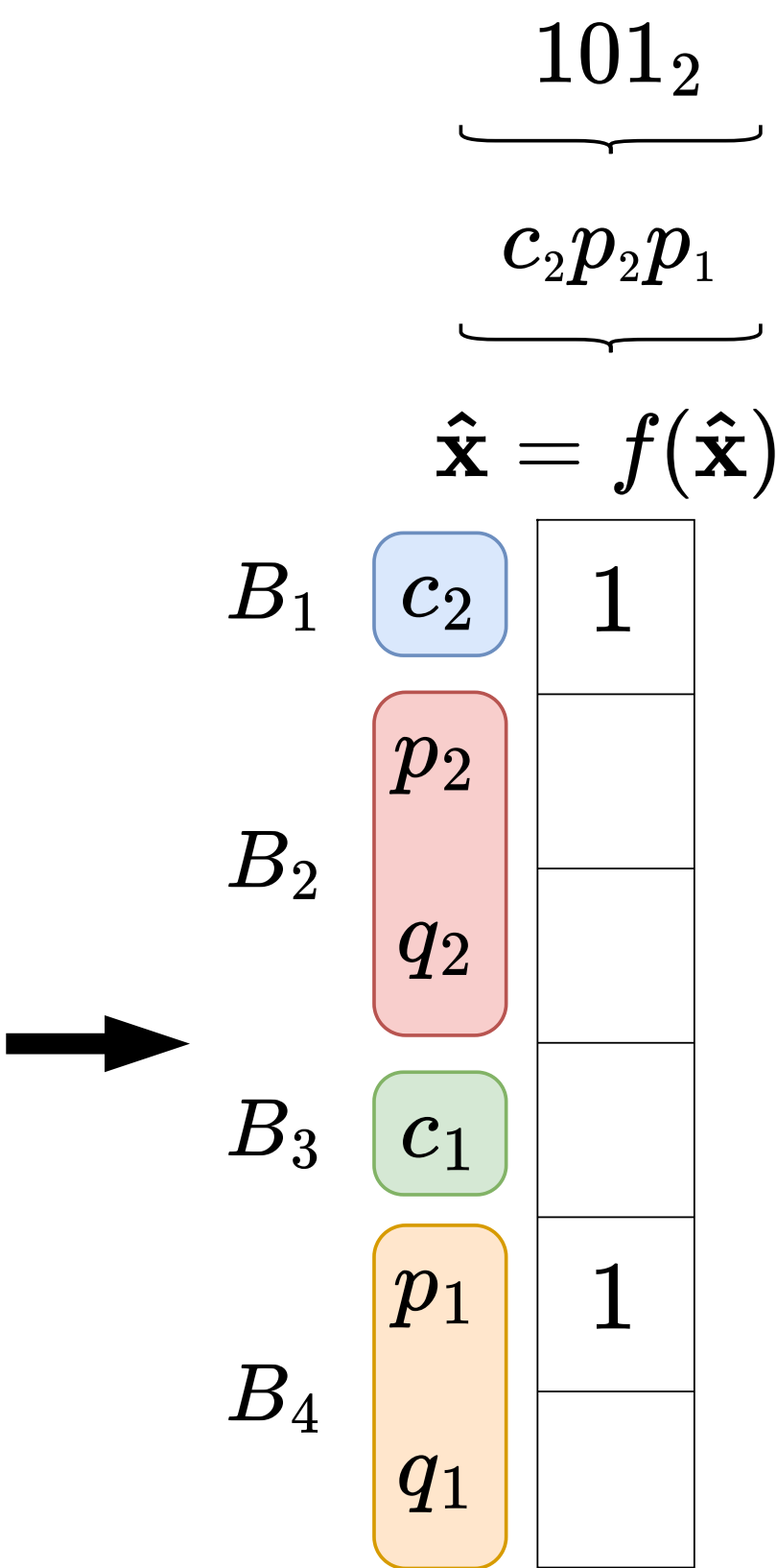
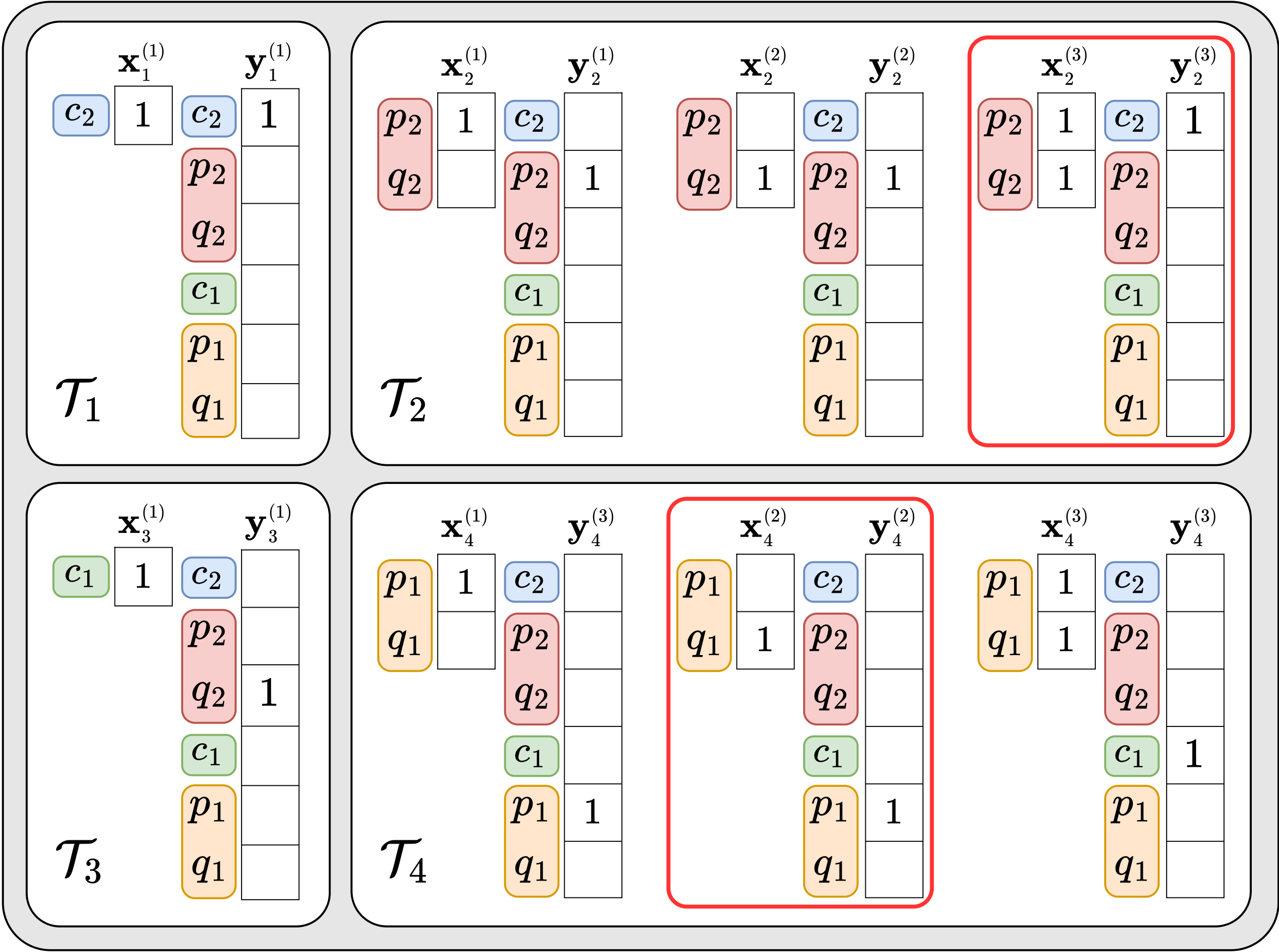
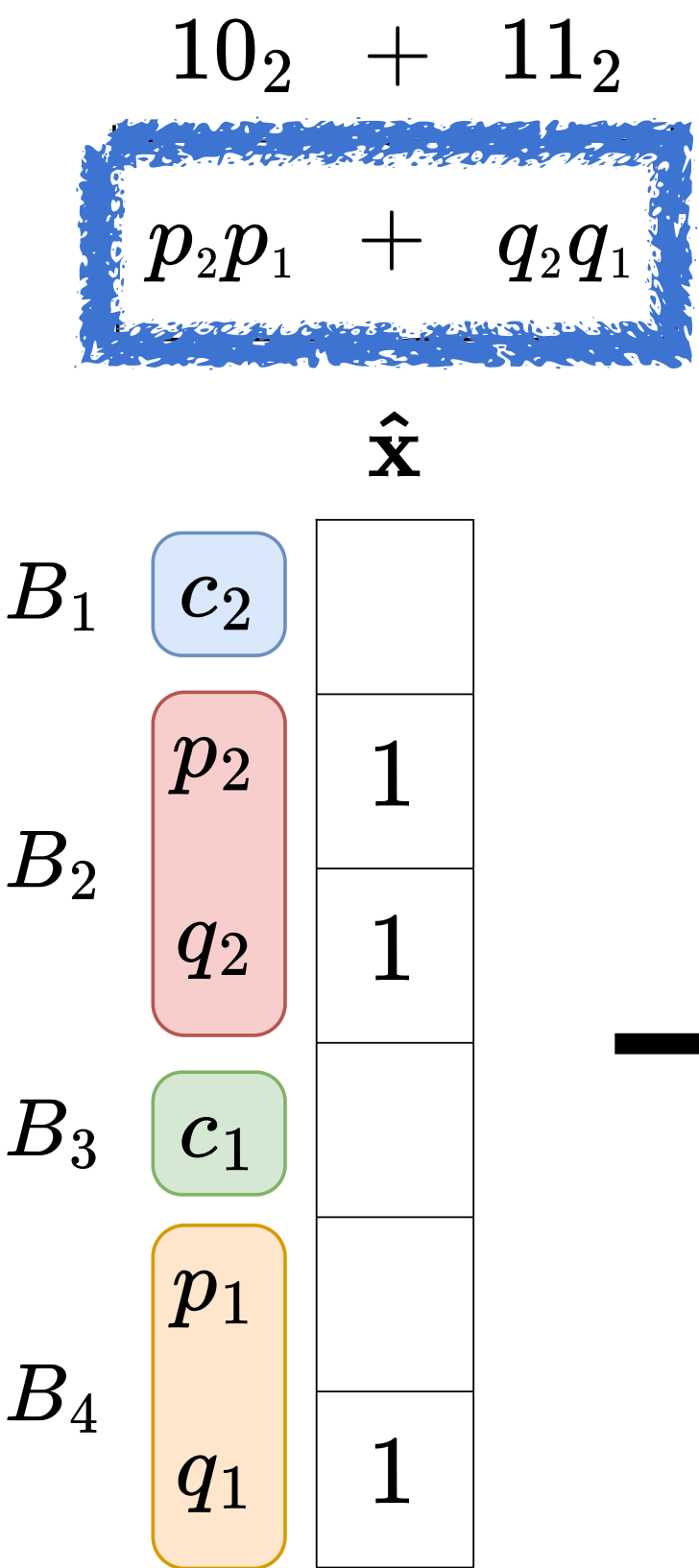
Correct execution for all
possible inputs

Example of execution: addition

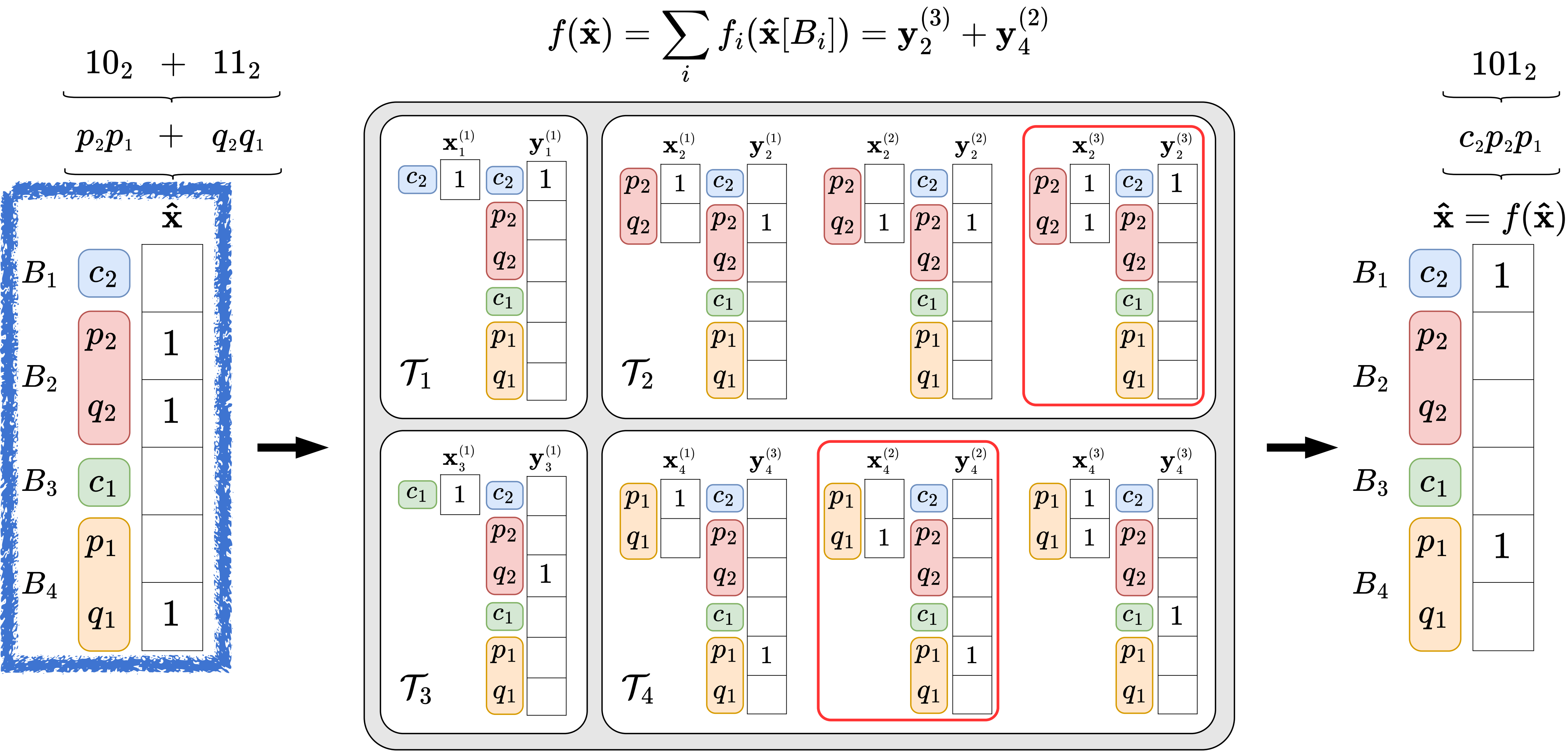


Example of execution: addition

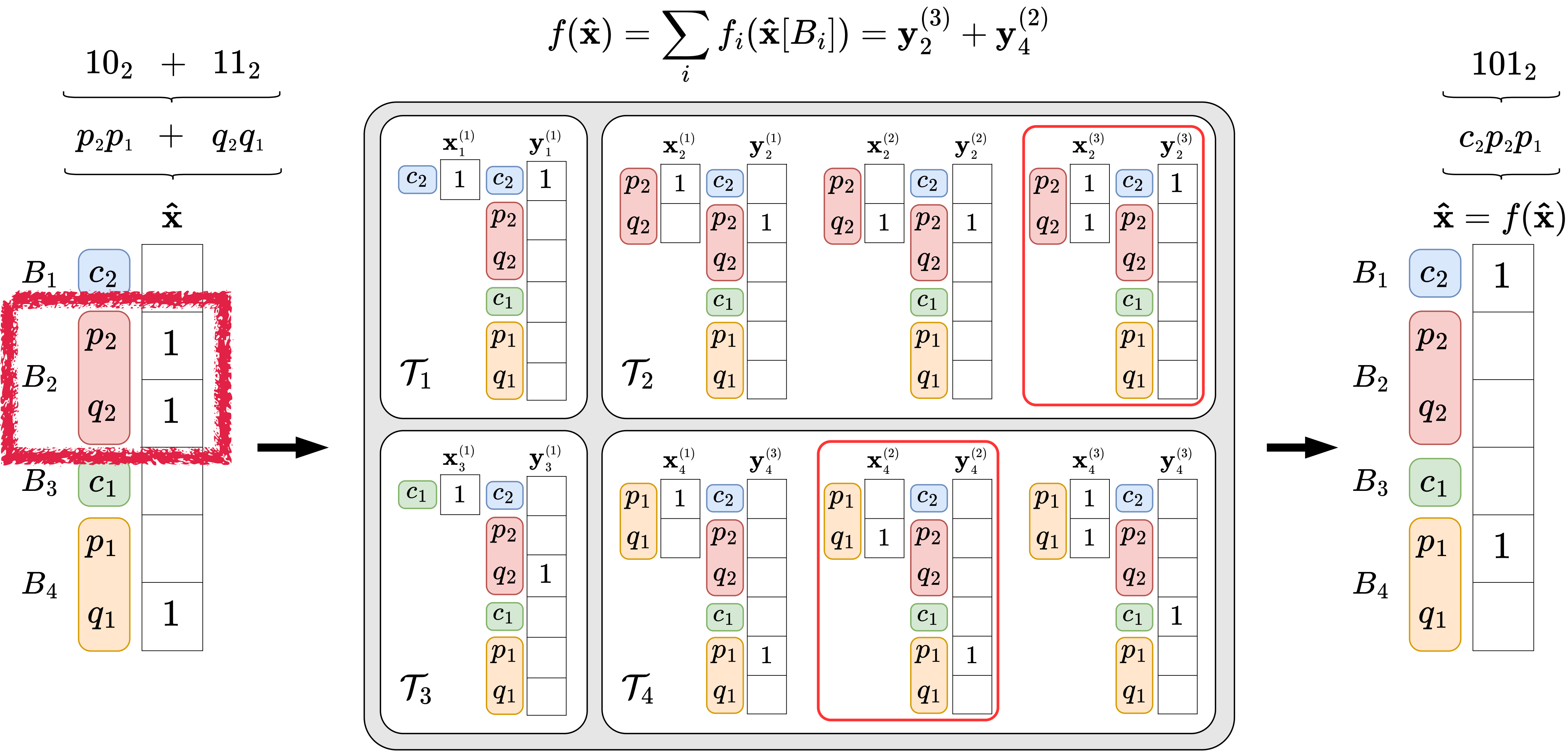
$$f(\hat{\mathbf{x}}) = \sum_i f_i(\hat{\mathbf{x}}[B_i]) = \mathbf{y}_2^{(3)} + \mathbf{y}_4^{(2)}$$



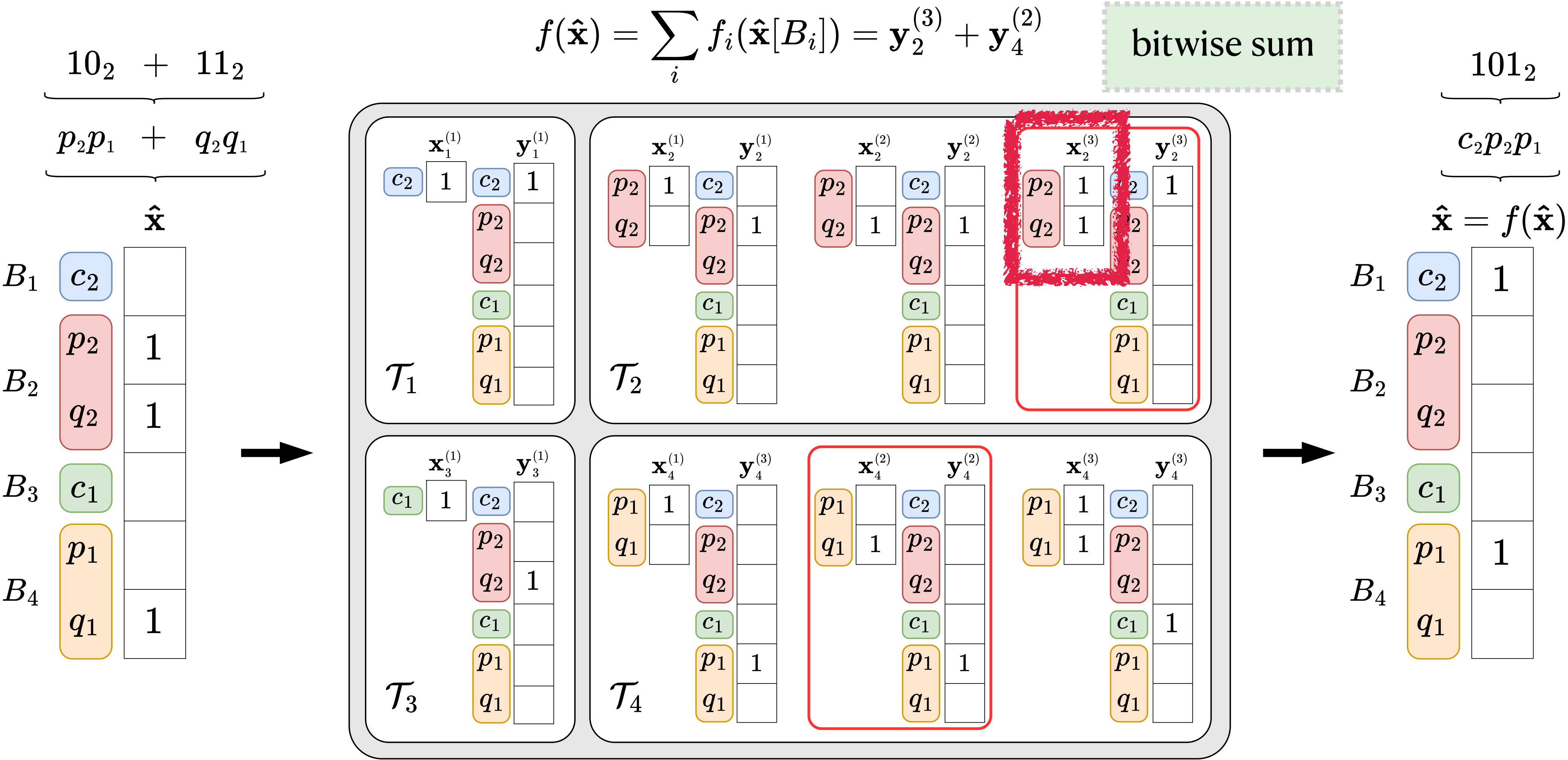
Example of execution: addition



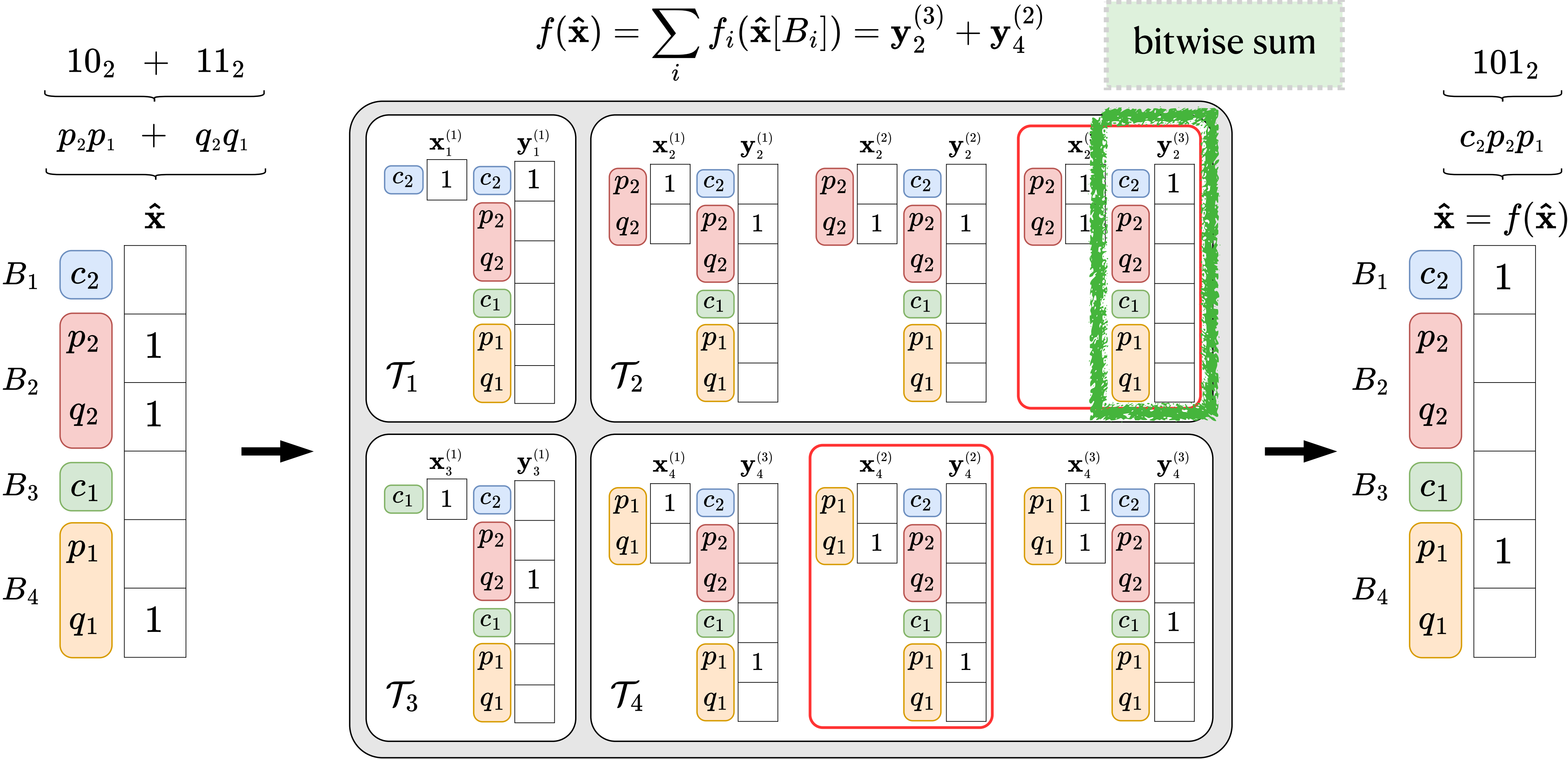
Example of execution: addition



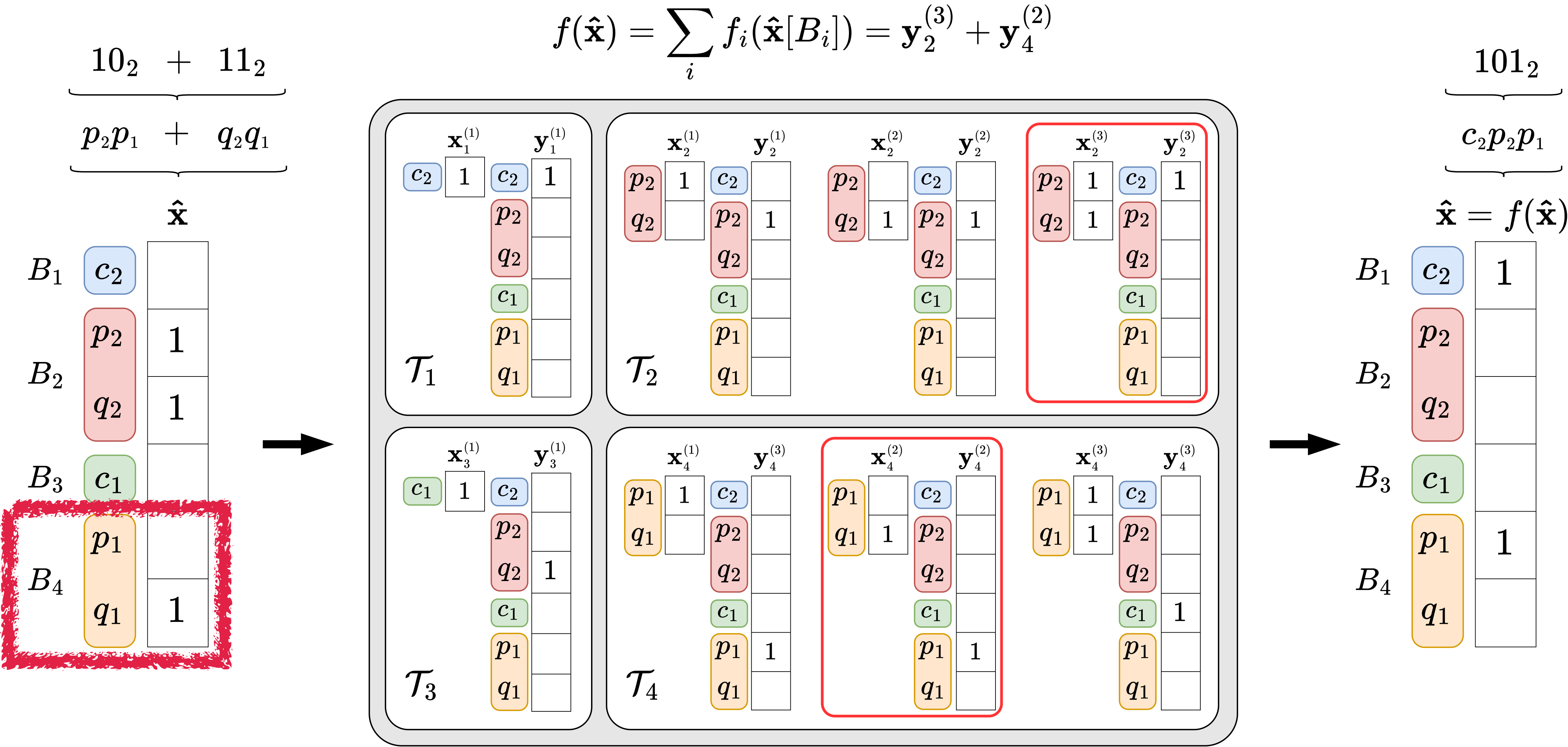
Example of execution: addition



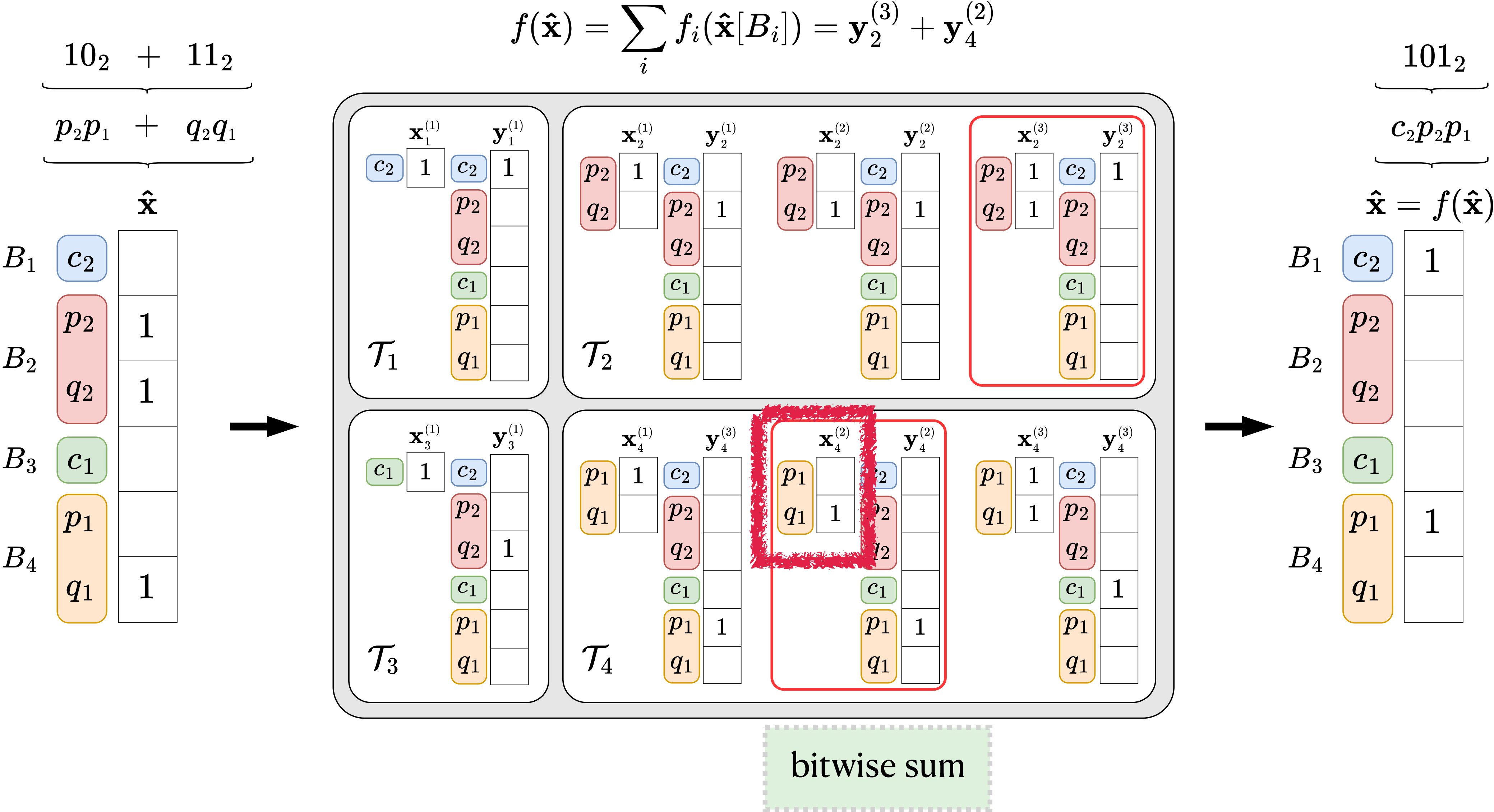
Example of execution: addition



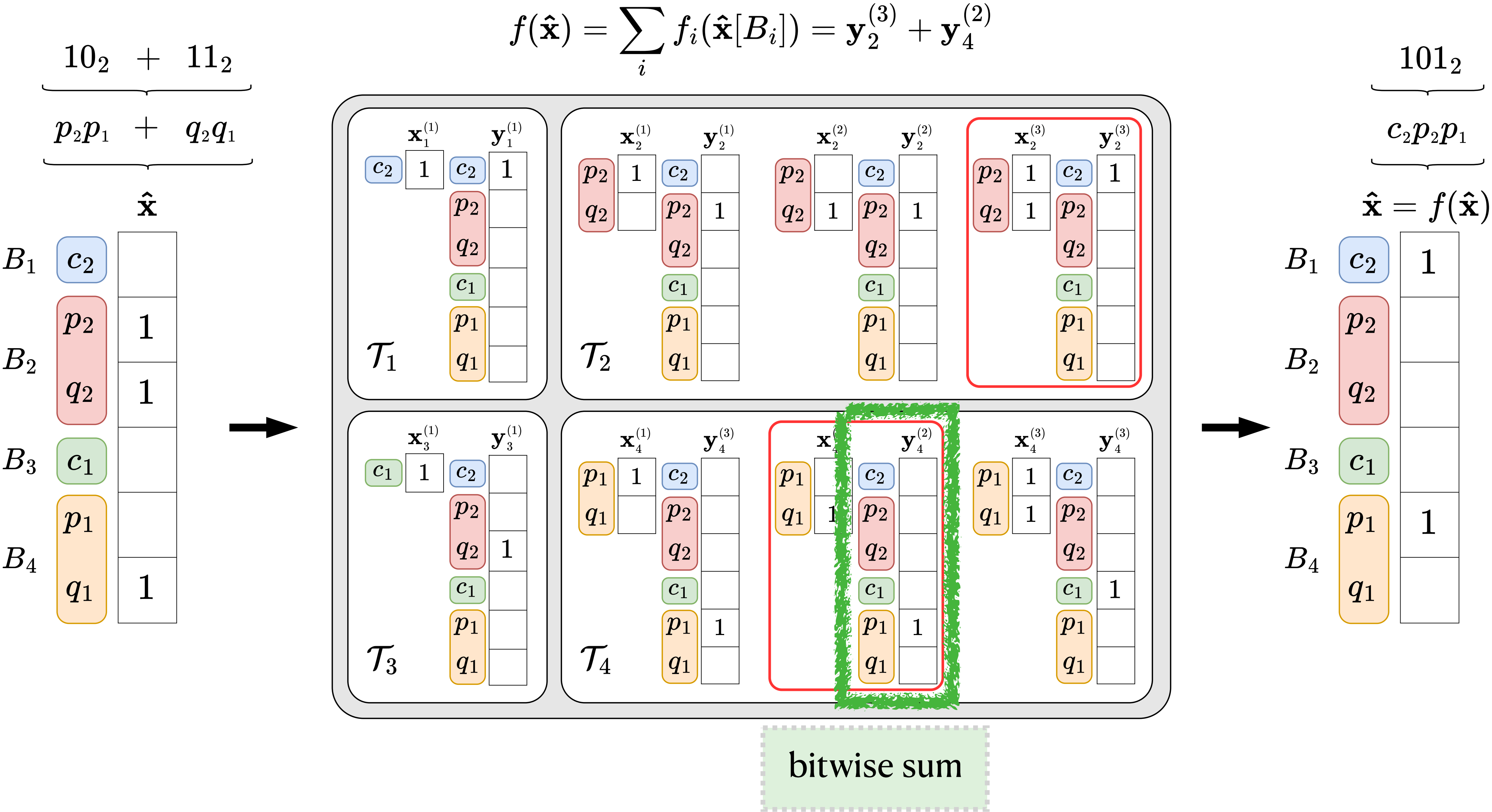
Example of execution: addition



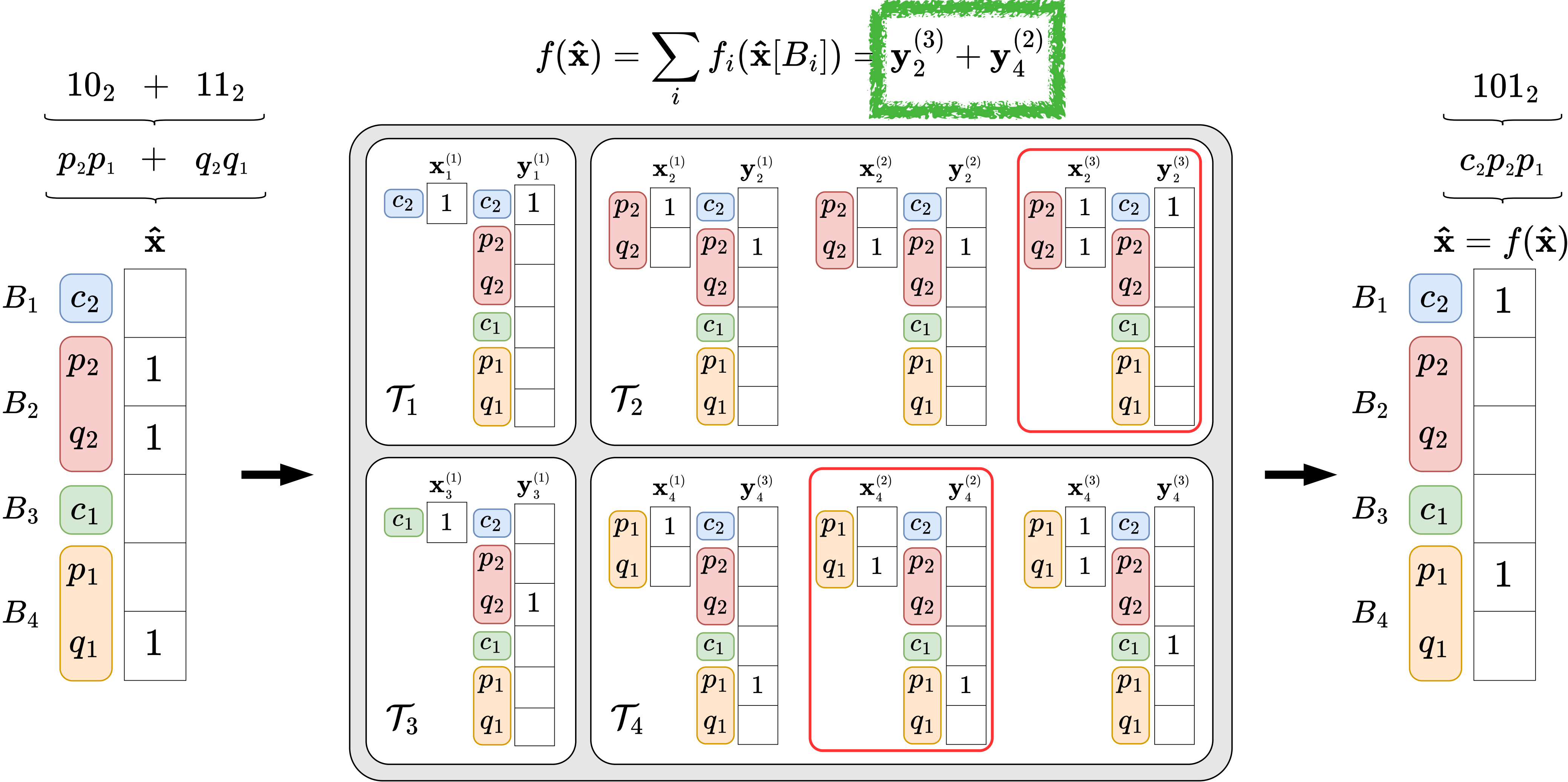
Example of execution: addition



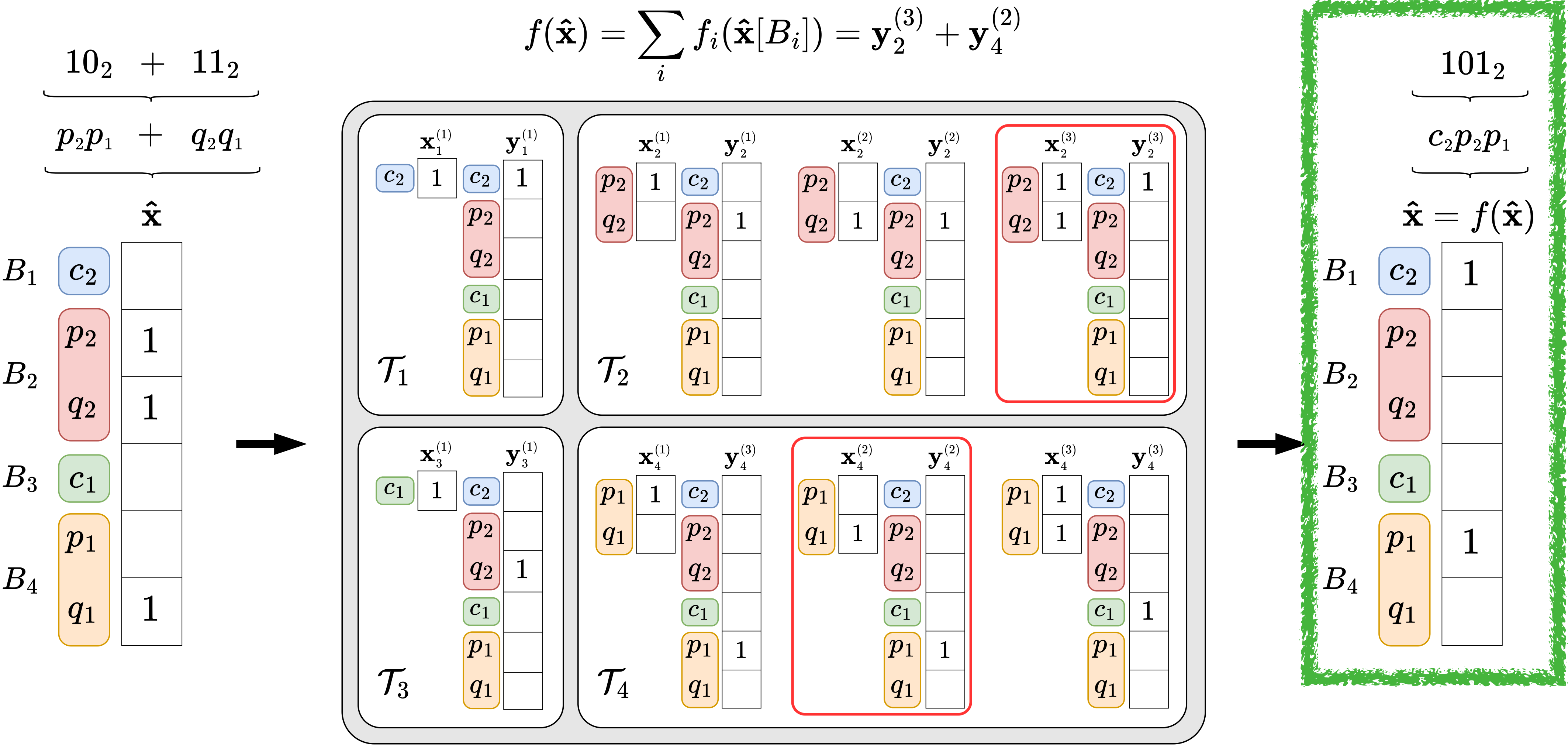
Example of execution: addition



Example of execution: addition



Example of execution: addition



What about other tasks? Sufficient conditions

Assumption: If there are constant number of conflicts per output bit among the output training data Y .

What about other tasks? Sufficient conditions

Assumption: If there are constant number of conflicts per output bit among the output training data Y .

Assumption: The blocks are $\mathcal{O}(1)$ size.

What about other tasks? Sufficient conditions

Assumption: If there are constant number of conflicts per output bit among the output training data Y .

Assumption: The blocks are $\mathcal{O}(1)$ size.

Then the looped mean executes $x^{(t)} := f(x^{(t-1)})$, for $t = 1, \dots, T$, without error, using only logarithmic number of examples for training.

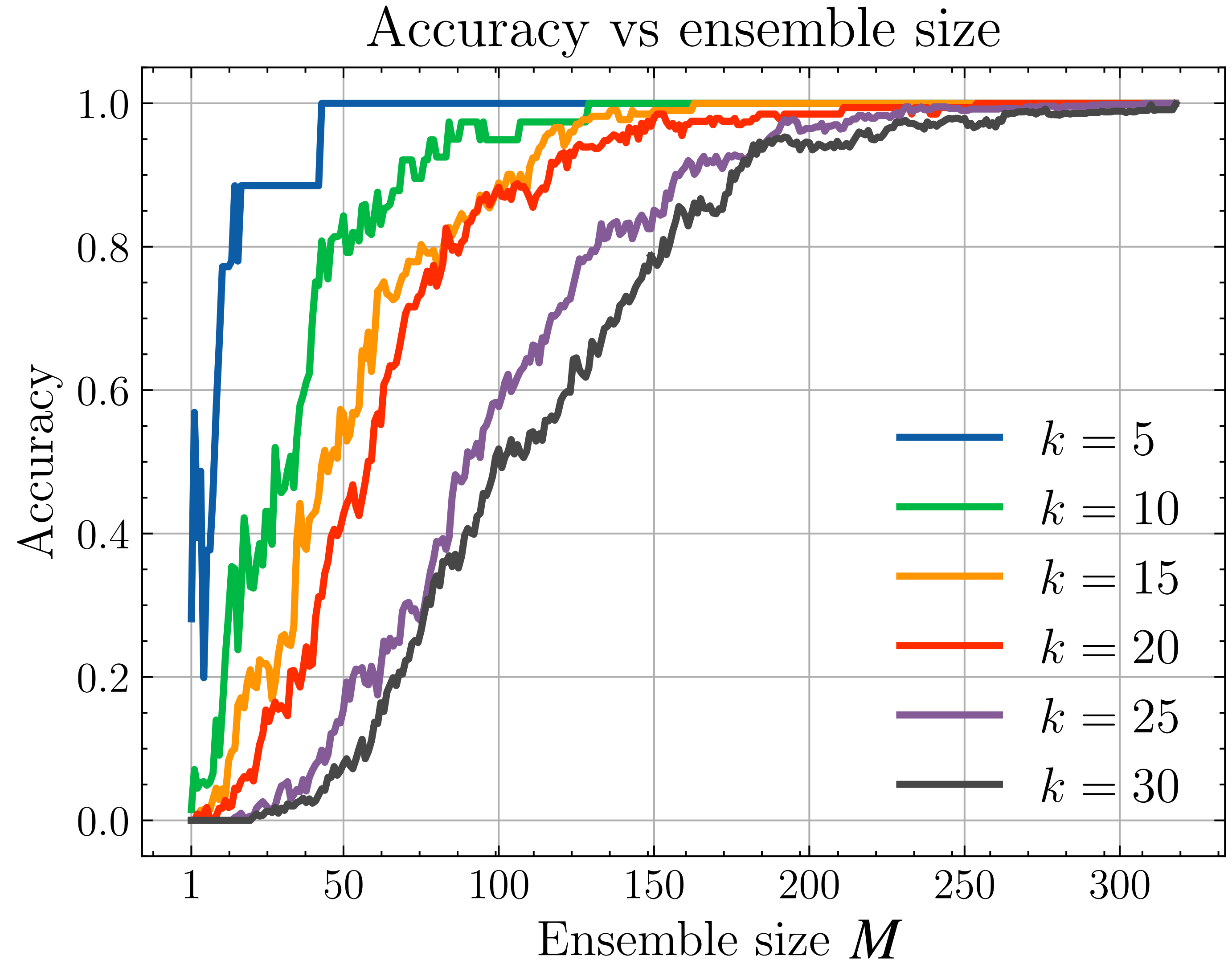
Training is expensive (variance)

Task: copy bits from input to output

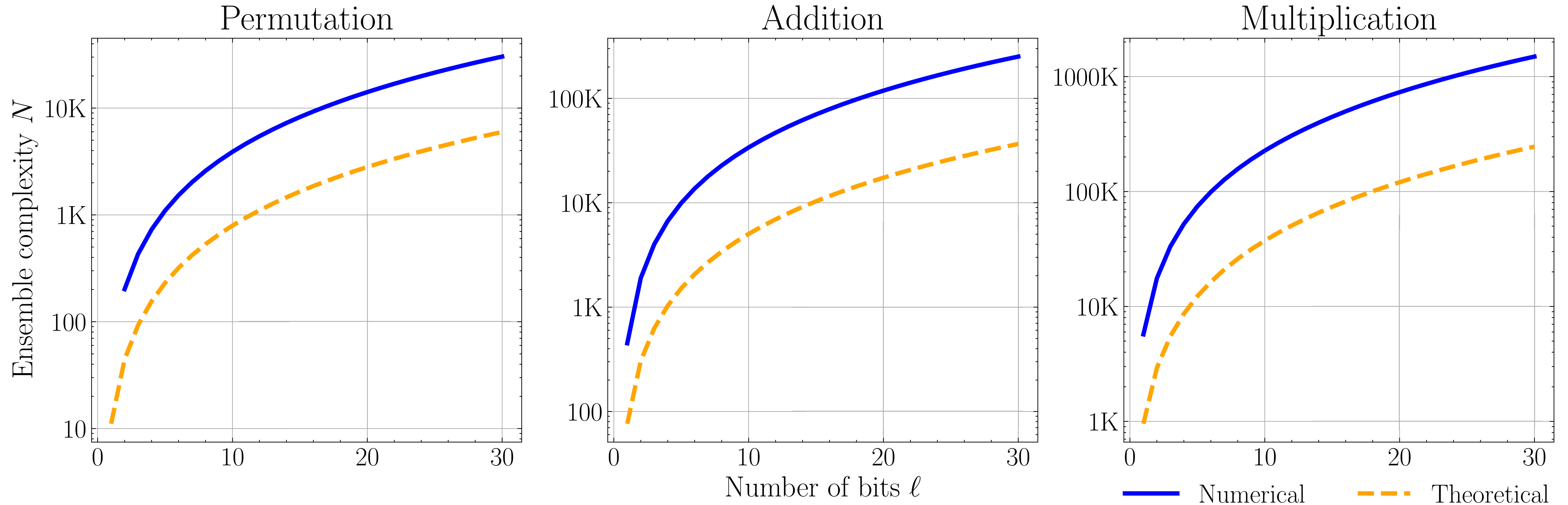
k: input length

ensemble size M : number of independently trained models to be averaged

We need to average 300 independently trained models (3 million parameters each) to learn to copy $k=30$ bits without error.

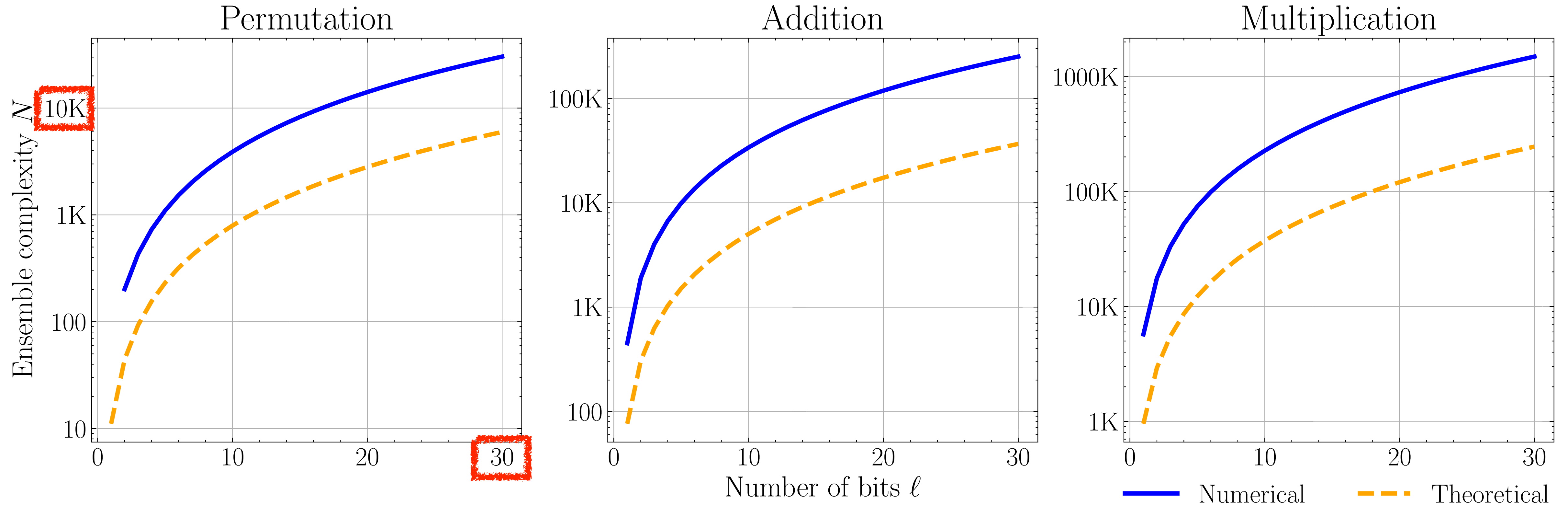


Exact learning is expensive



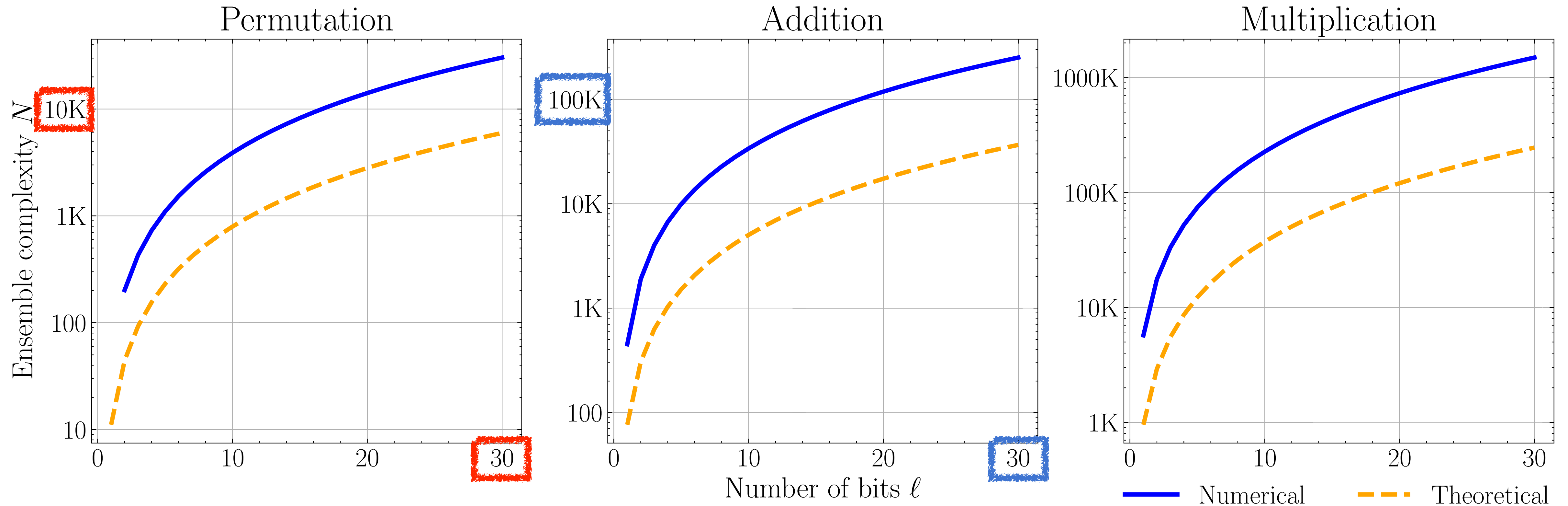
The ensemble size scales quadratically in the input bits ℓ

Exact learning is expensive



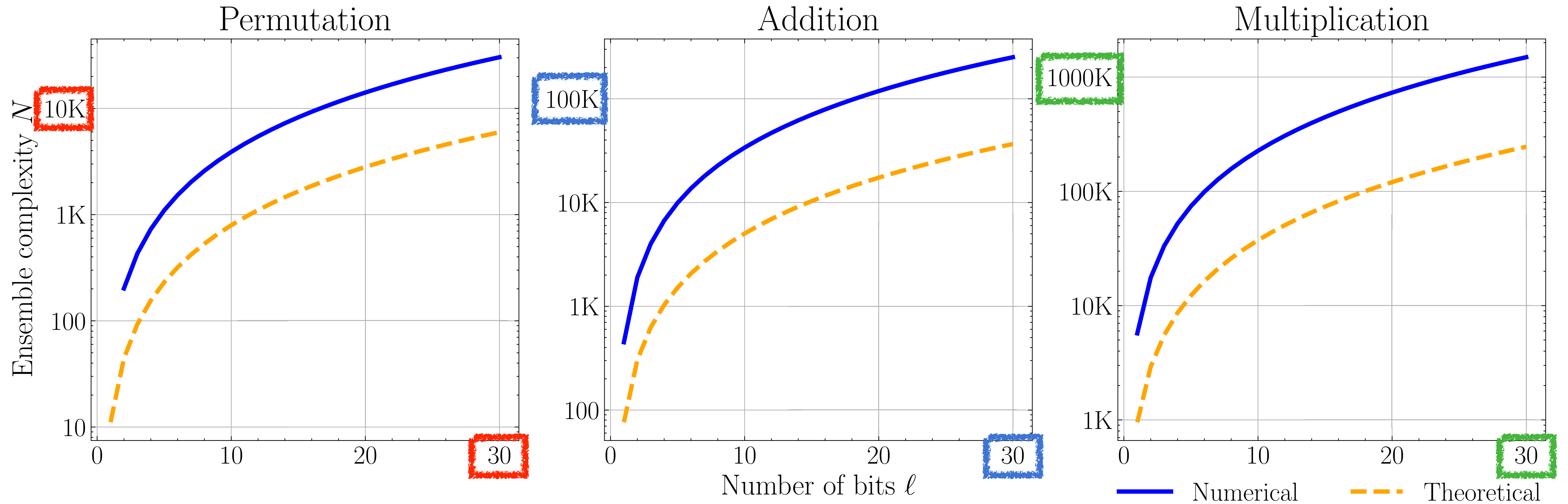
The ensemble size scales quadratically in the input bits ℓ

Exact learning is expensive



The ensemble size scales quadratically in the input bits ℓ

Exact learning is expensive



The ensemble size scales quadratically in the input bits ℓ

Learning End-to-End is Hard

 G. Giapitzakis, K. Fountoulakis, E. Nichani, J. D. Lee . ***On the Statistical Query Complexity of Learning Semiautomata: a Random Walk Approach.*** COLT 2026.

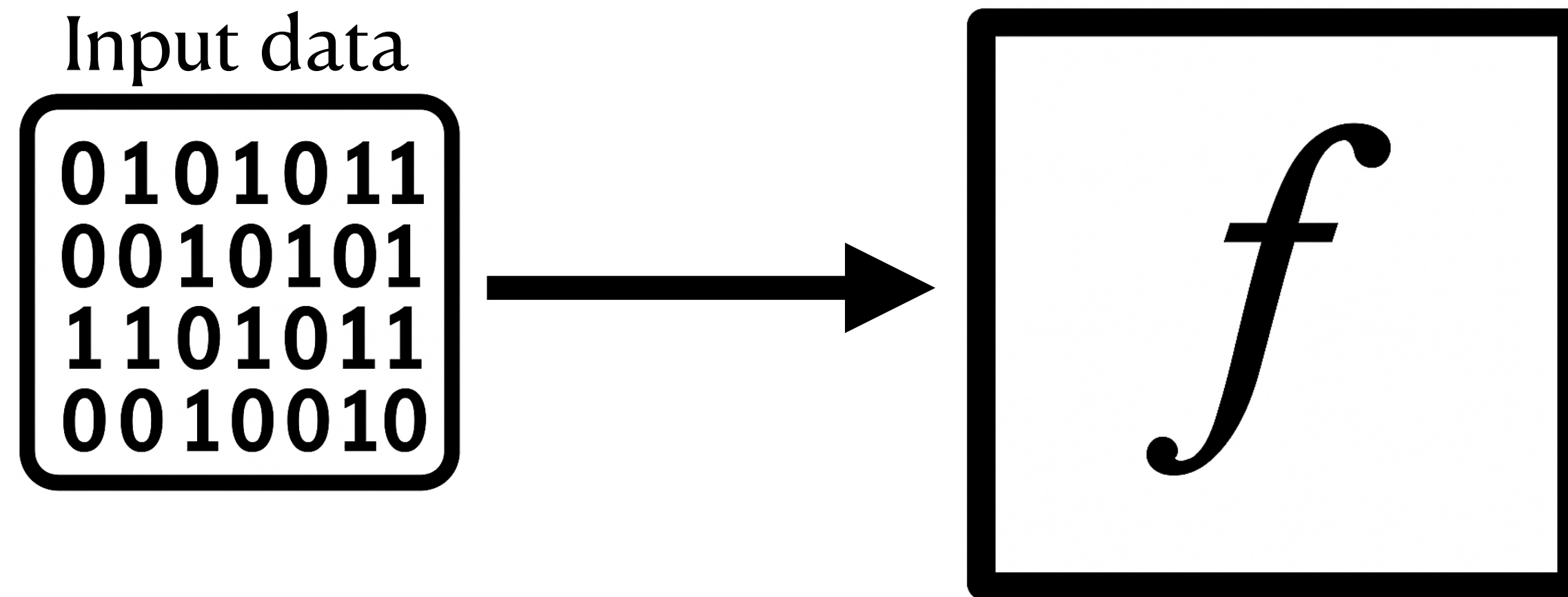
Computational model: semiautomata

Computational model: semiautomata

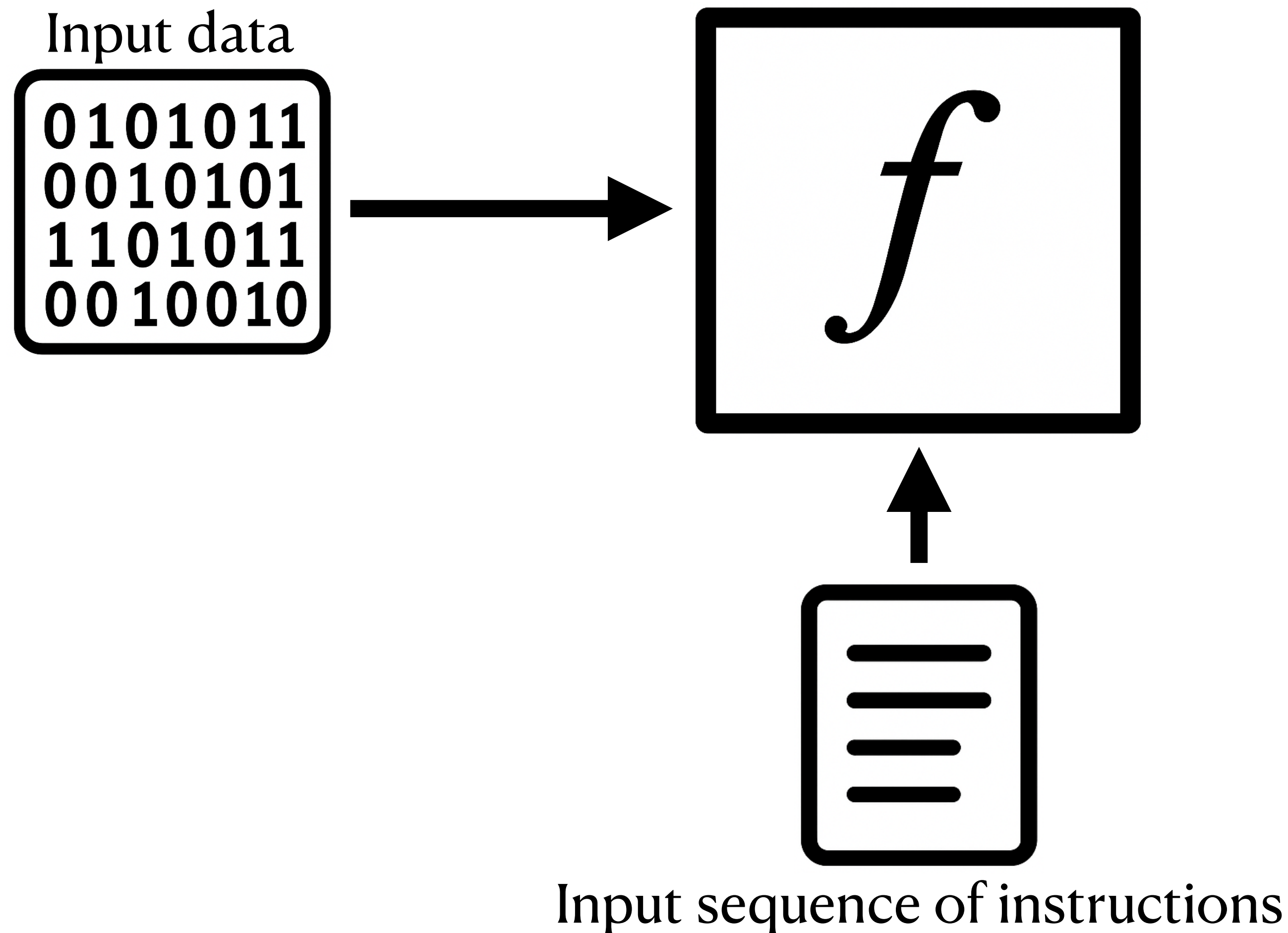
Input data

0	1	0	1	0	1	1
0	0	1	0	1	0	1
1	1	0	1	0	1	1
0	0	1	0	0	1	0

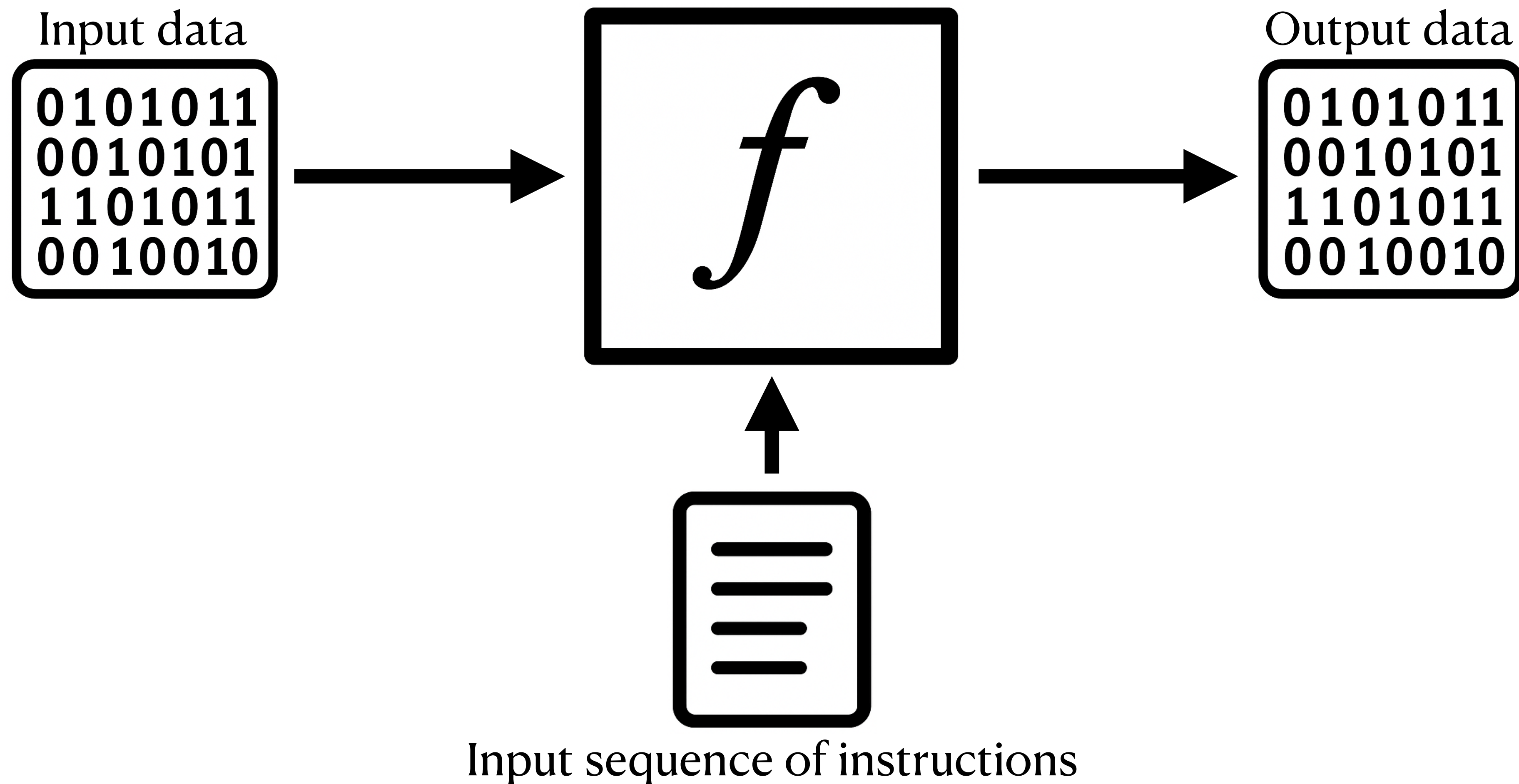
Computational model: semiautomata



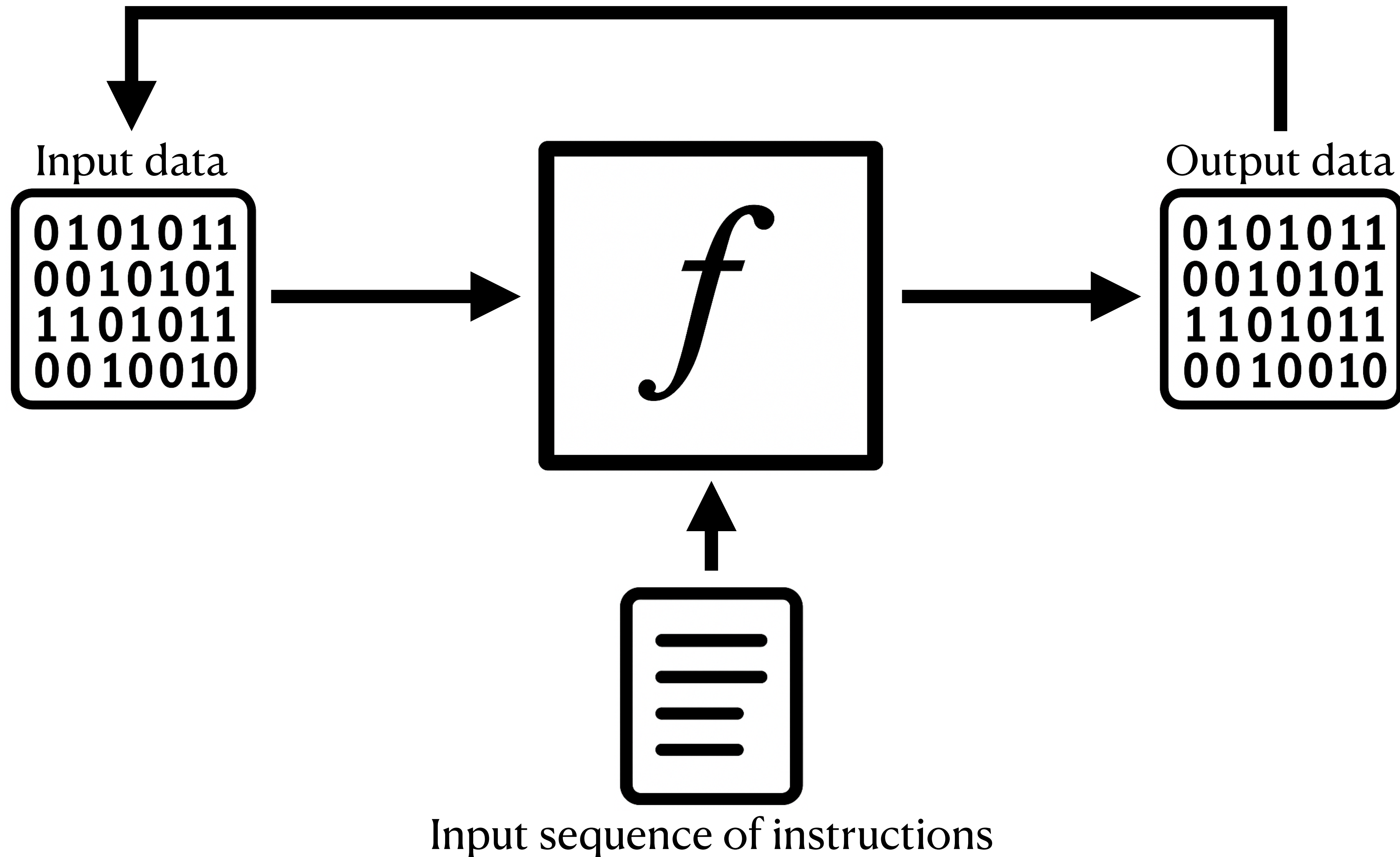
Computational model: semiautomata



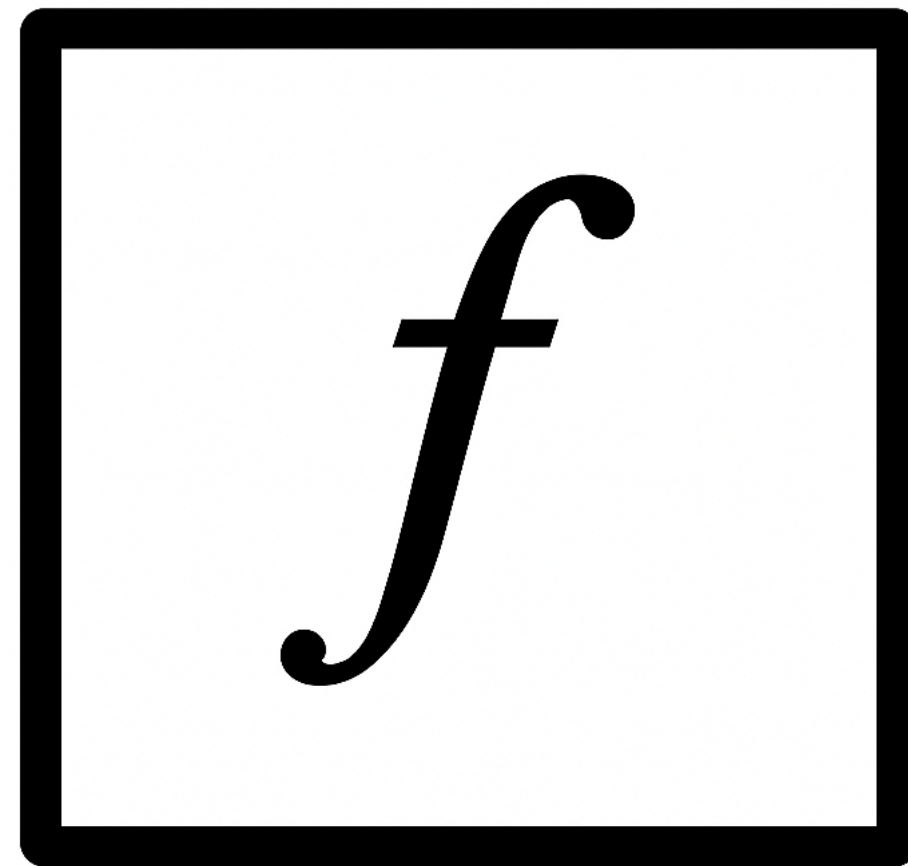
Computational model: semiautomata



Computational model: semiautomata



Computational model: semiautomata



State transition function

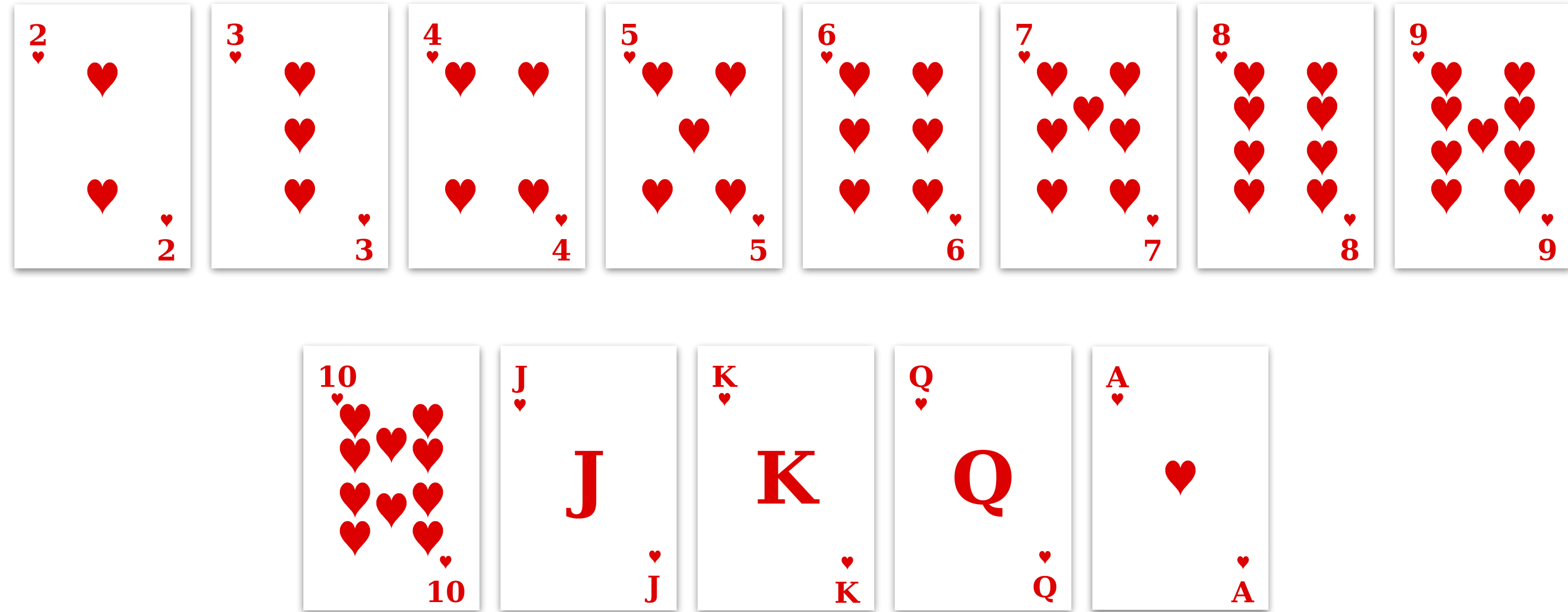
Discrete rules

Conditional branching

Composition of the above

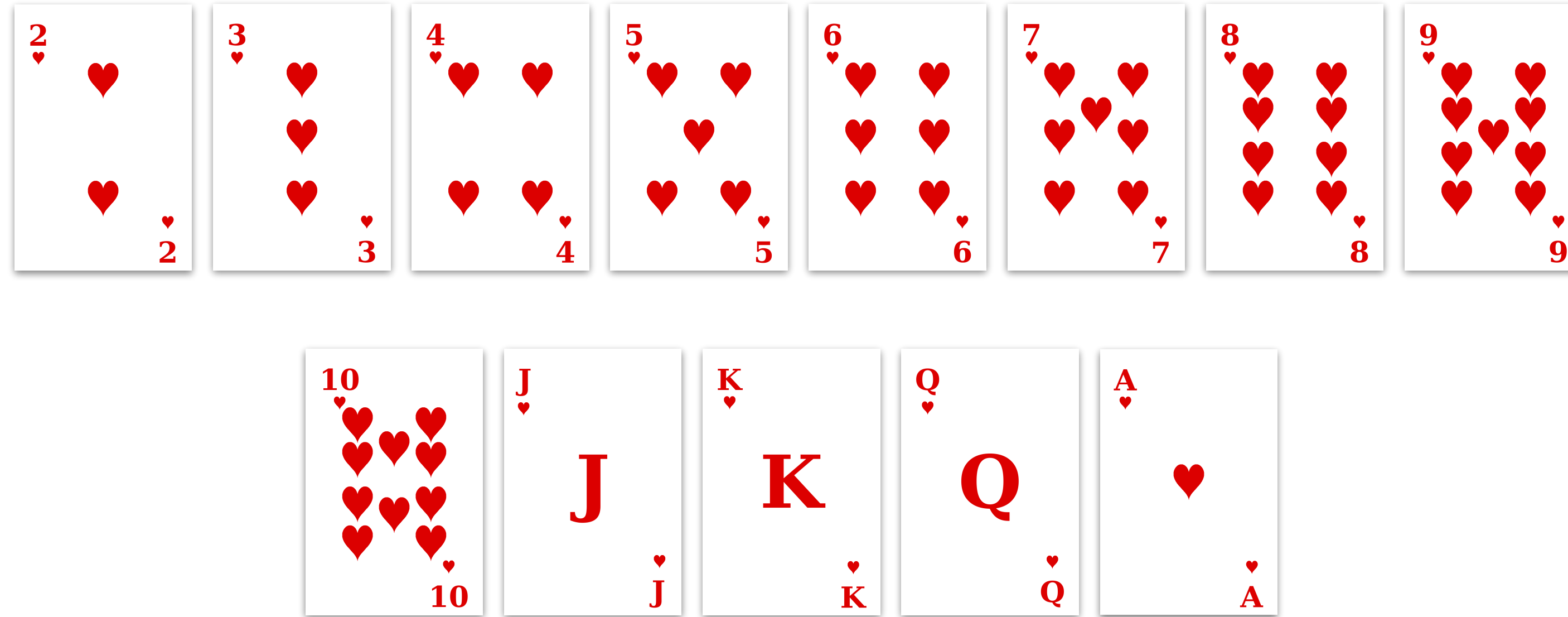
Example: shuffling

Input data:



Example: shuffling

Input data:



Input sequence of instructions:

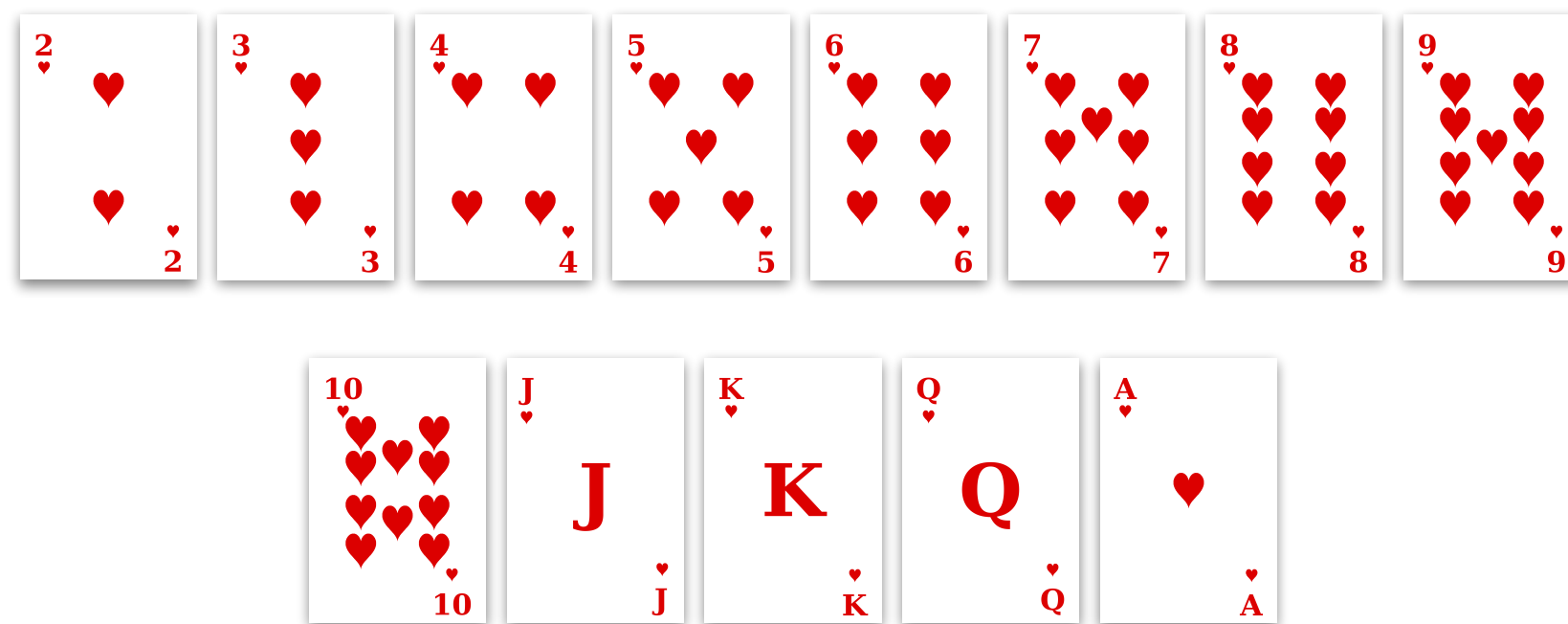
First, swap positions 5 and 12
Then, swap positions 3 and 10

⋮

Total #swaps: $\binom{N}{2}$

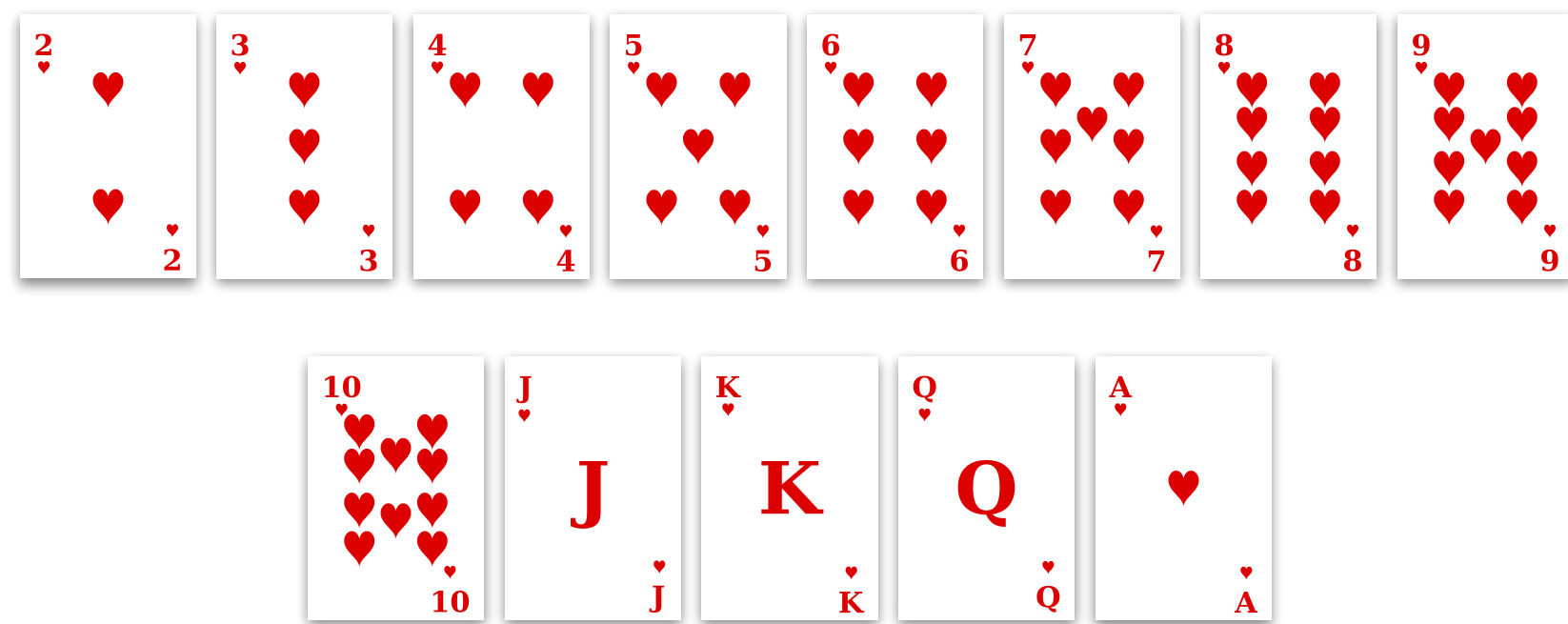
Example: shuffling

Input data



Example: shuffling

Input data

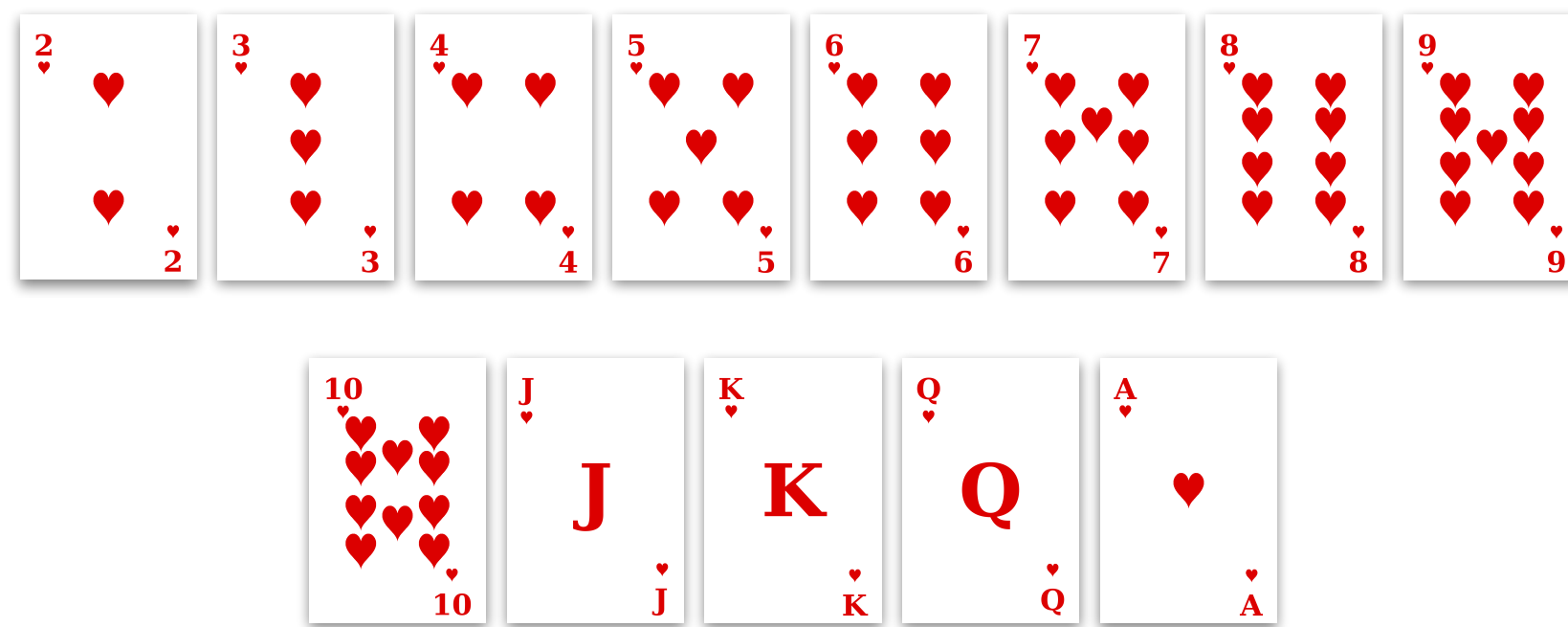


... swap (3,10) swap (5,12)

Input sequence of instructions:

Example: shuffling

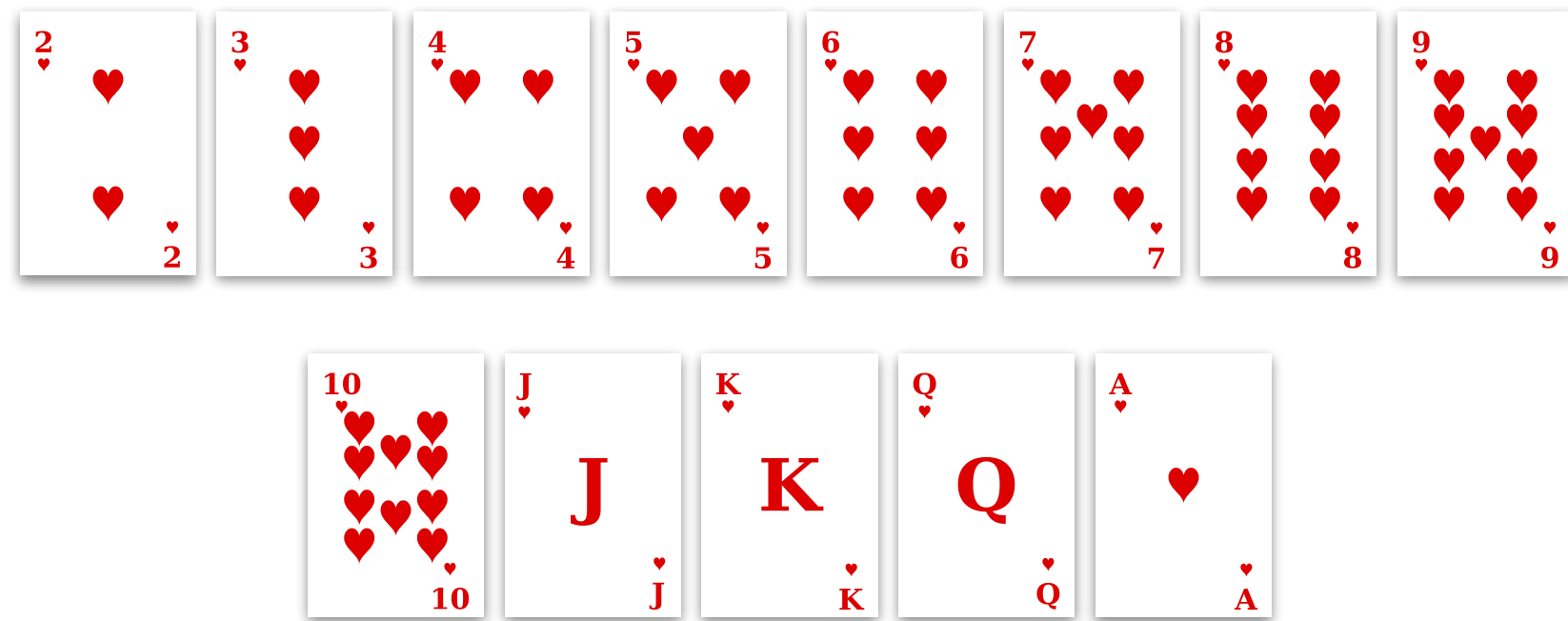
Input data



... swap (3,10) swap (5,12)

1-st instruction

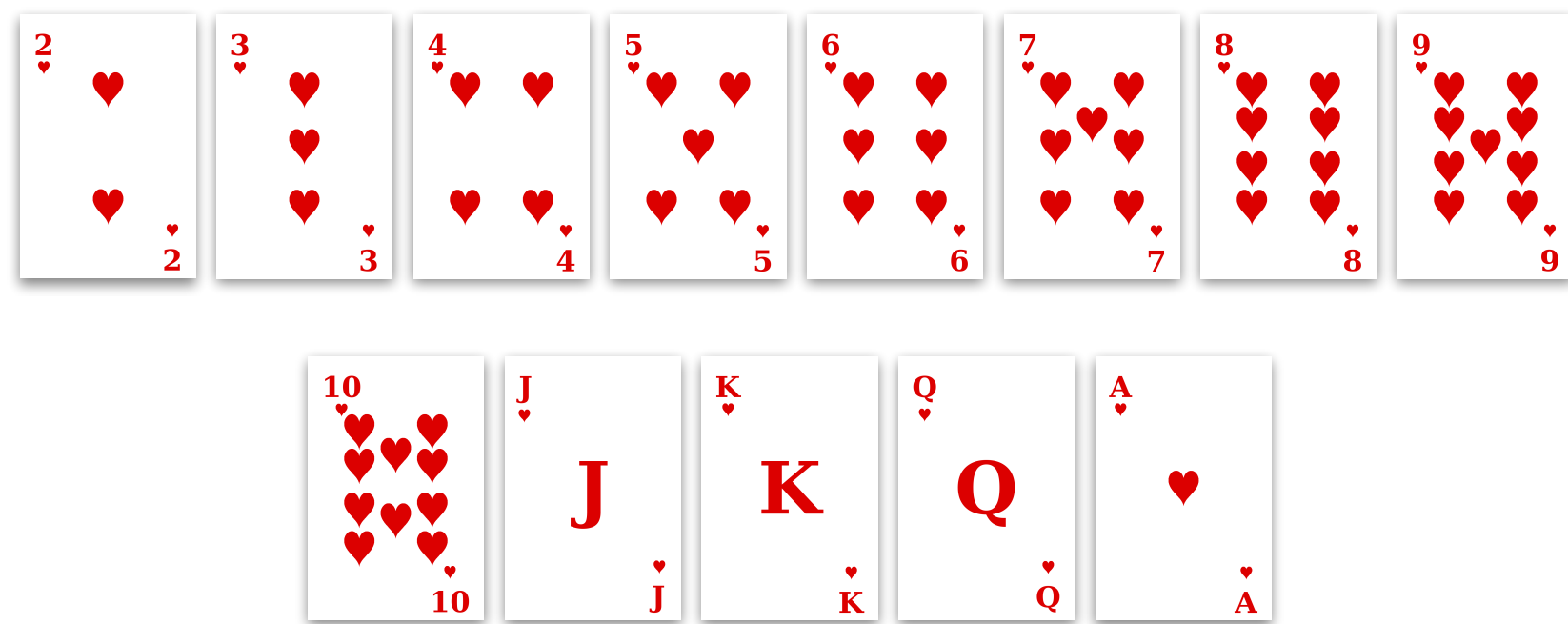
Example: shuffling



... swap (3,10) swap (5,12)

2-nd instruction

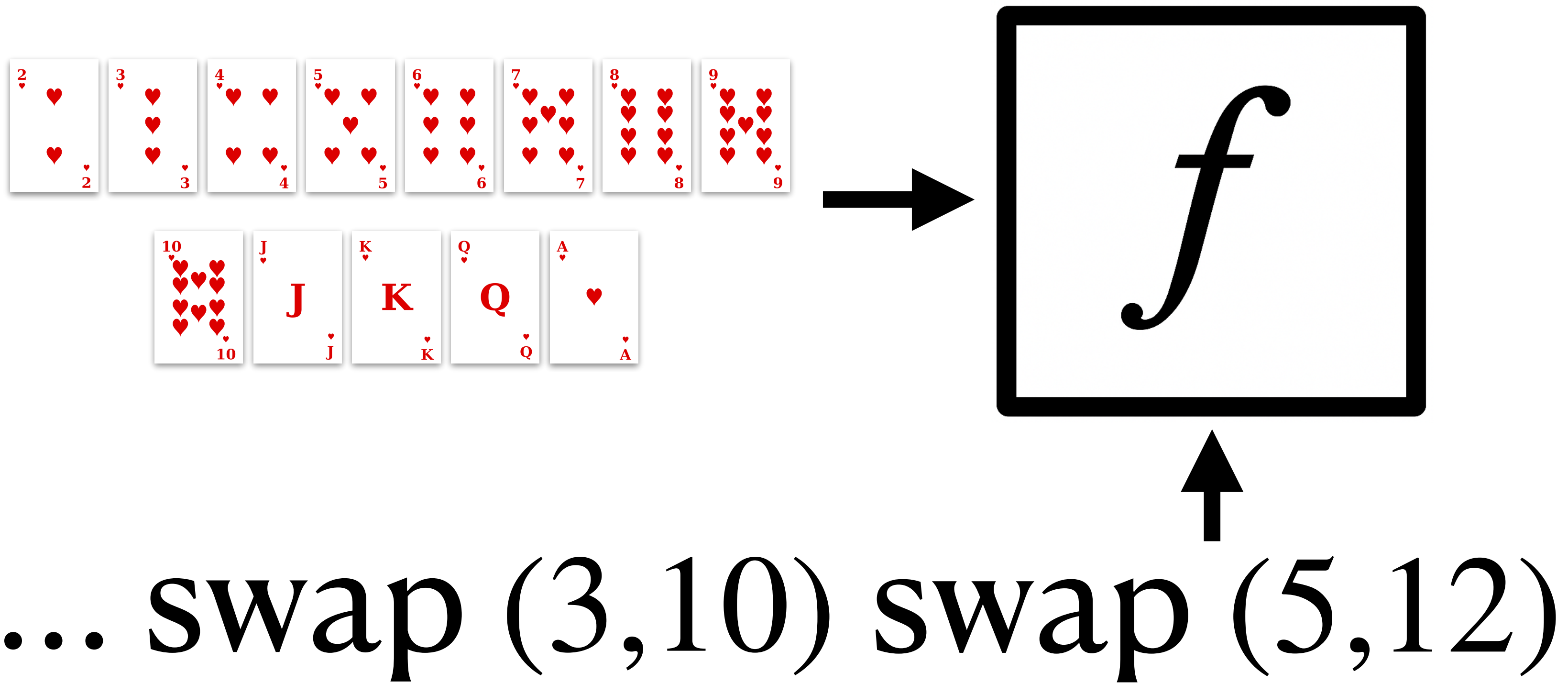
Example: shuffling



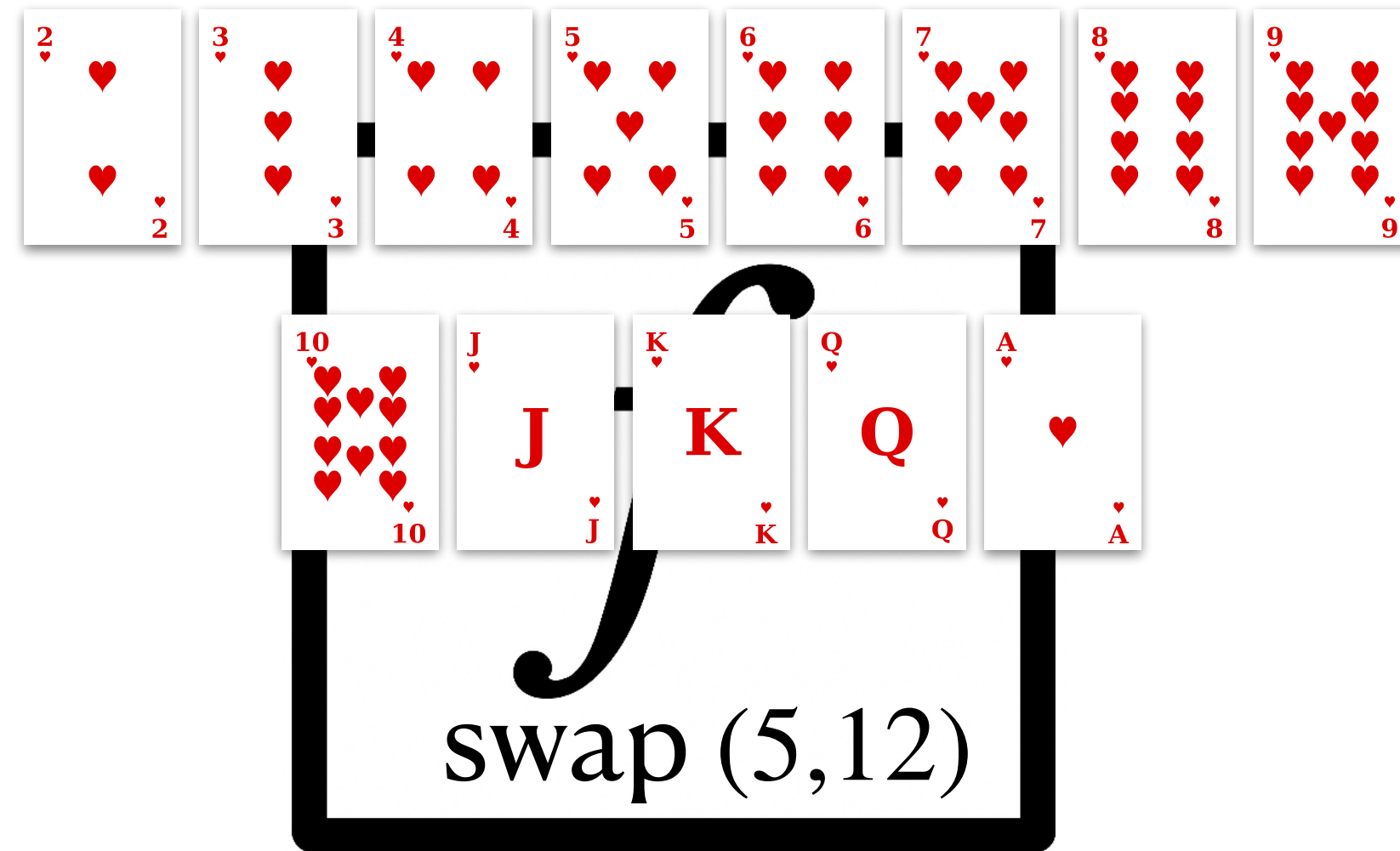
... swap (3,10) swap (5,12)

etc

Example: shuffling

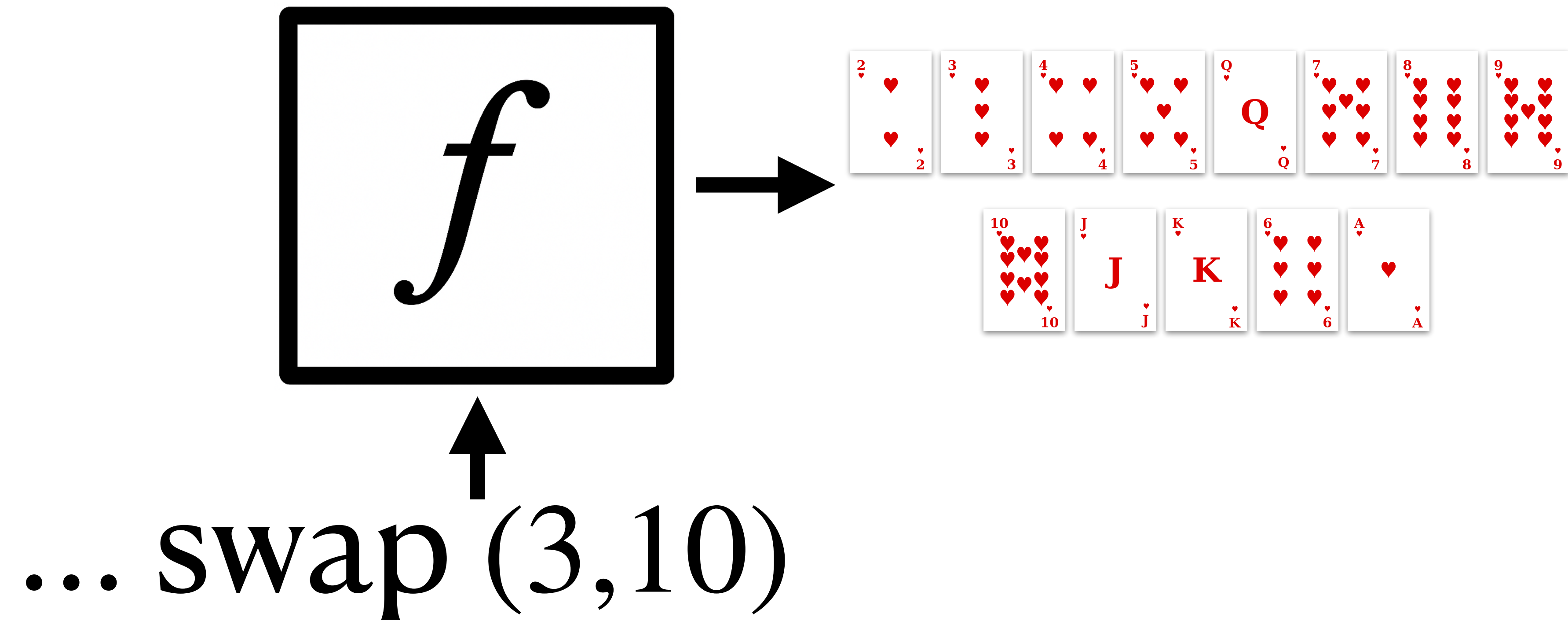


Example: shuffling

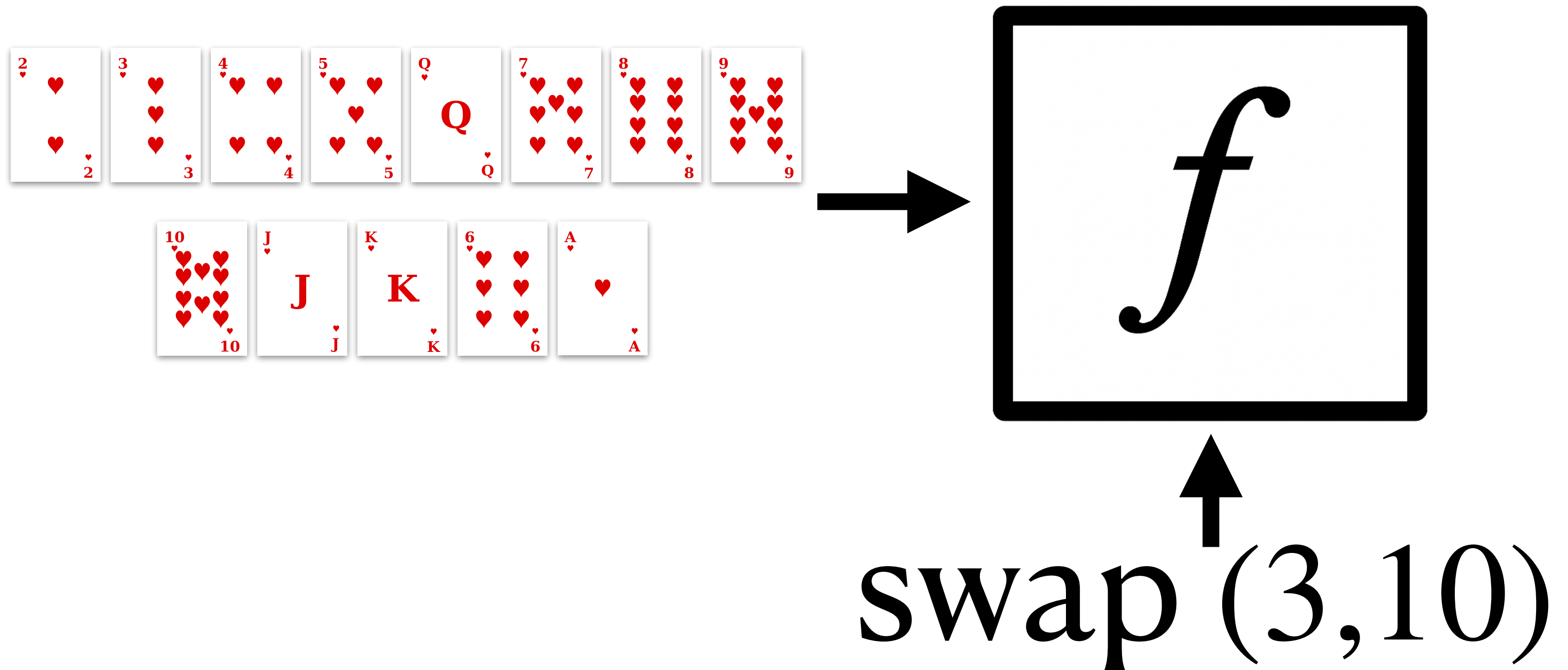


... swap (3,10)

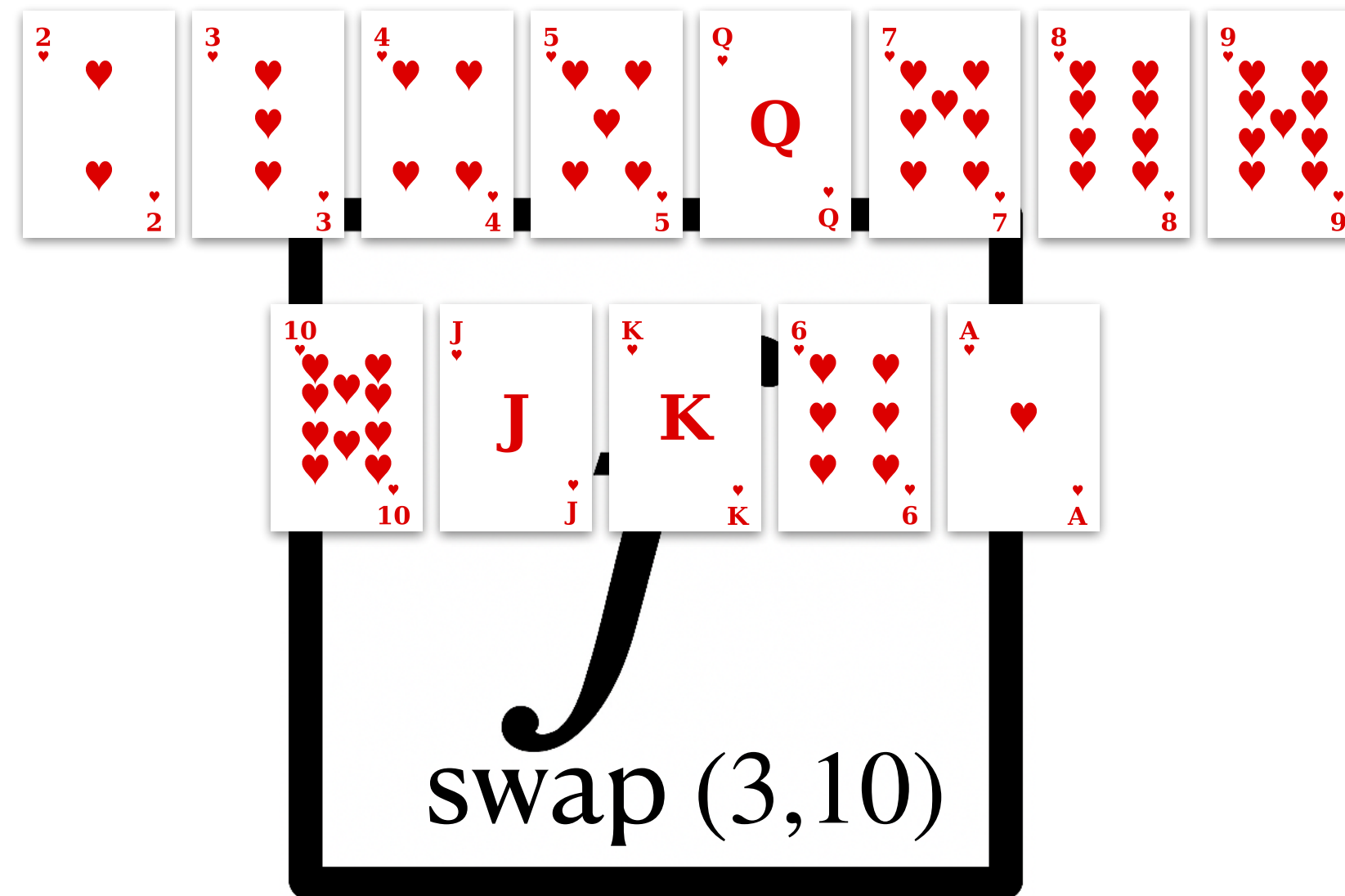
Example: shuffling



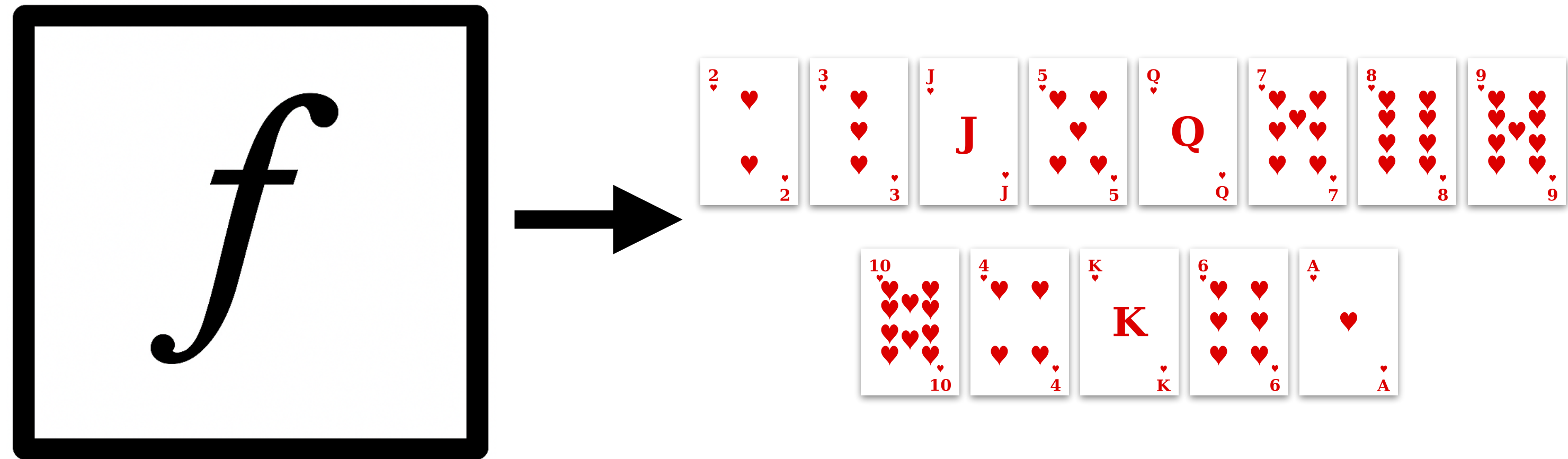
Example: shuffling



Example: shuffling



Example: shuffling



Other examples

Simple machines: permuting/copying data.

Arithmetic: addition, multiplication etc.

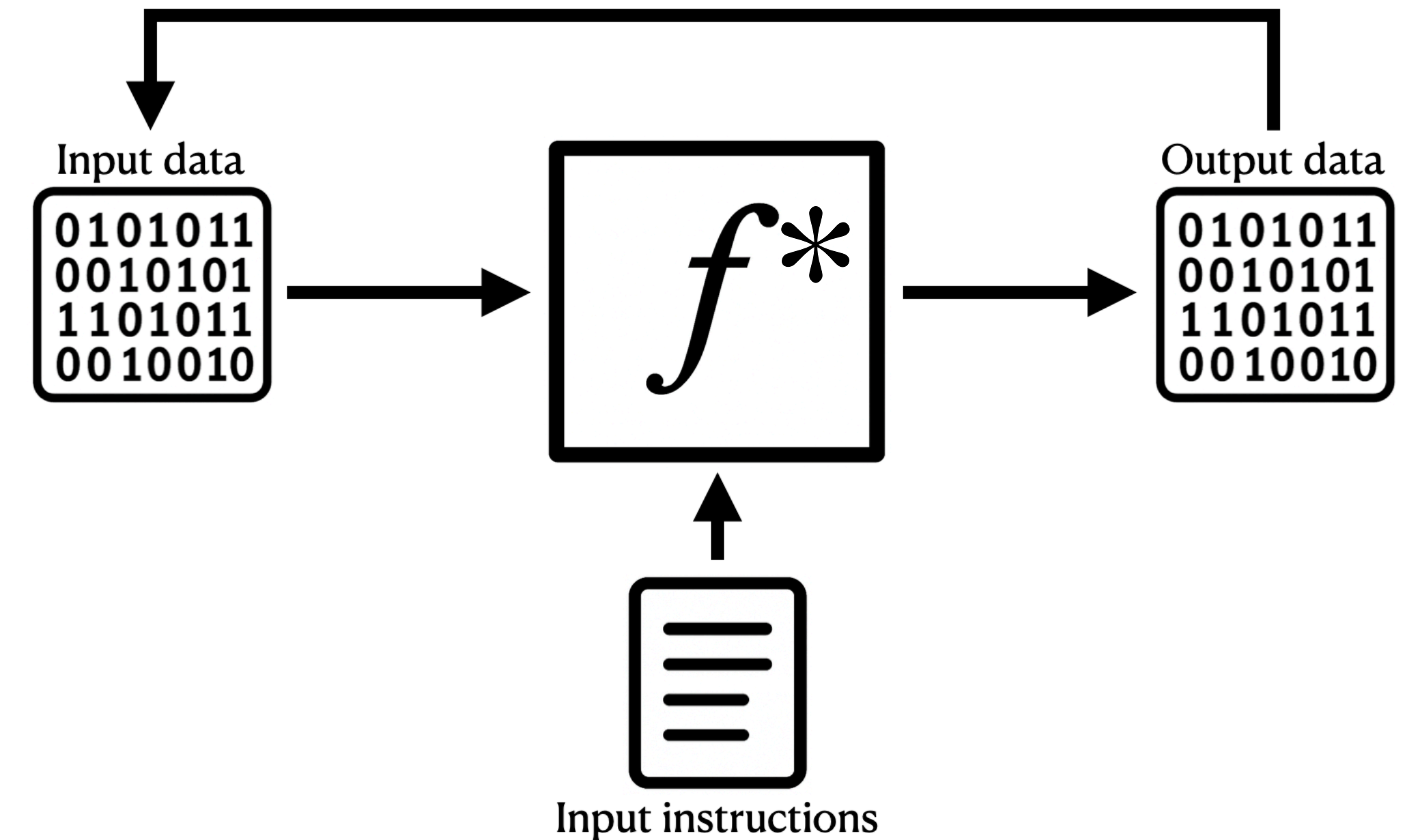
Turing machines: one-instruction set computer (OISC).

Learning problem setting

- Fix a set of *discrete instructions*
- Fix a set of N discrete data

Learning problem setting

- Fix a set of *discrete instructions*
- Fix a set of N discrete data
- Pick a target f^* (unknown to the learner)



- **Goal:** given input-output pairs of f^* , or statistical queries about it, *can the learner identify f^** , and at what computational cost?

End-to-end training data

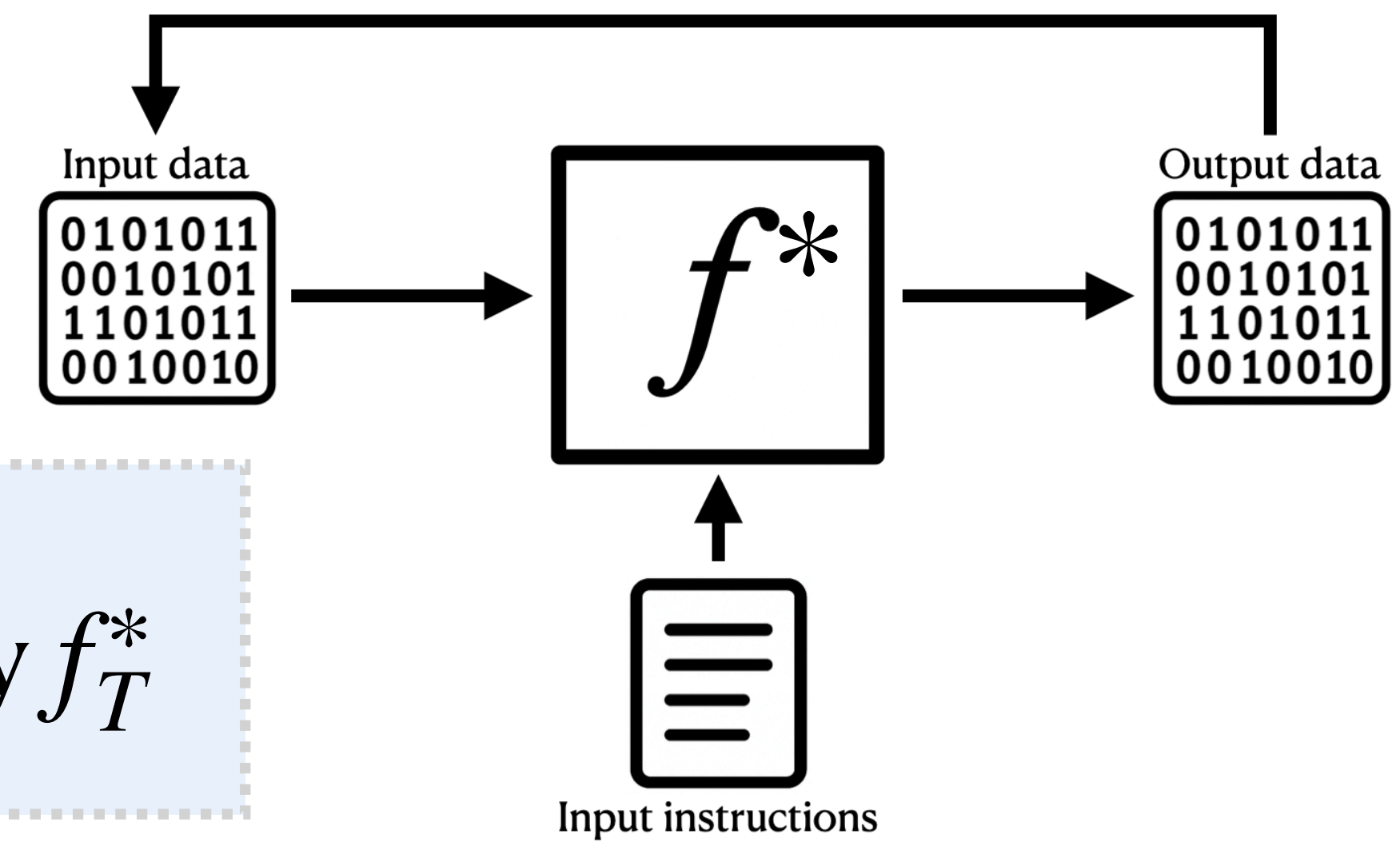
input starting data

```
0101011
0010101
1101011
0010010
```

input instructions



output *the last iteration* T of f^* , denoted by f_T^*



Learning via statistical queries (SQ)

Humans learn by asking questions to an oracle. So, how can a machine learn?

Learning via statistical queries (SQ)

Humans learn by asking questions to an oracle. So, how can a machine learn?

Using statistical queries!

Learning via statistical queries (SQ)

Humans learn by asking questions to an oracle. So, how can a machine learn?

Using statistical queries!

$$x := (\text{instructions}, \text{data}) \sim D$$

Learning via statistical queries (SQ)

Humans learn by asking questions to an oracle. So, how can a machine learn?

Using statistical queries!

$x := (\text{instructions, data}) \sim D$

The learner submits statistic h
and tolerance τ to the oracle

Learning via statistical queries (SQ)

Humans learn by asking questions to an oracle. So, how can a machine learn?

Using statistical queries!

$x := (\text{instructions, data}) \sim D$

The learner submits statistic h and tolerance τ to the oracle

The oracle replies v such that $\left| v - \mathbb{E}_{x \sim D}[h(x, f_T^*(x))] \right| \leq \tau$

Learning via statistical queries (SQ)

Humans learn by asking questions to an oracle. So, how can a machine learn?

Using statistical queries!

Example: Gradient Descent is an SQ learning algorithm

$$\min_{\theta} \mathbb{E}_{x \sim D} \left[\ell_{\theta}(x, f^*(x)) \right]$$

Learning via statistical queries (SQ)

Humans learn by asking questions to an oracle. So, how can a machine learn?

Using statistical queries!

Example: Gradient Descent is an SQ learning algorithm

Set $h(x, f^*(x)) = \nabla_{\theta} \ell_{\theta}(x, f^*(x))$ (gradient statistic)

Learning via statistical queries (SQ)

Humans learn by asking questions to an oracle. So, how can a machine learn?

Using statistical queries!

Example: Gradient Descent is an SQ learning algorithm

Set $h(x, f^*(x)) = \nabla_{\theta} \ell_{\theta}(x, f^*(x))$ (gradient statistic)

The oracle returns: approximation of $\mathbb{E}_{x \sim D} [\nabla_{\theta} \ell_{\theta}(x, f^*(x))]$ up to tolerance τ

Hardness result

Theorem

Consider the following parameters:

- N states
- The instruction set has size $\Theta(N^3 \log N)$
- $\Theta(N^2 \log N)$ iterations

Then, there are algorithms f^* , which require super-polynomial (in N) number of statistical queries or tolerance to be learned.

High-level idea of the proof

Class of
semiautomata

Sub-class of
shuffling machines

High-level idea of the proof

Class of
semiautomata

Sub-class of
shuffling machines

Goal:
detect the target
shuffling machine f^*

High-level idea of the proof

Class of
semiautomata

Sub-class of
shuffling machines

Goal:
detect the target
shuffling machine f^*

Eliminate machines using information from statistical queries until we learn f^*

High-level idea of the proof

Class of
semiautomata

Sub-class of
shuffling machines

Region
eliminated by
1-st query

Goal:
detect the target
shuffling machine f^*

Eliminate machines using information from statistical queries until we learn f^*

High-level idea of the proof

Class of
semiautomata

Sub-class of
shuffling machines

2-nd query

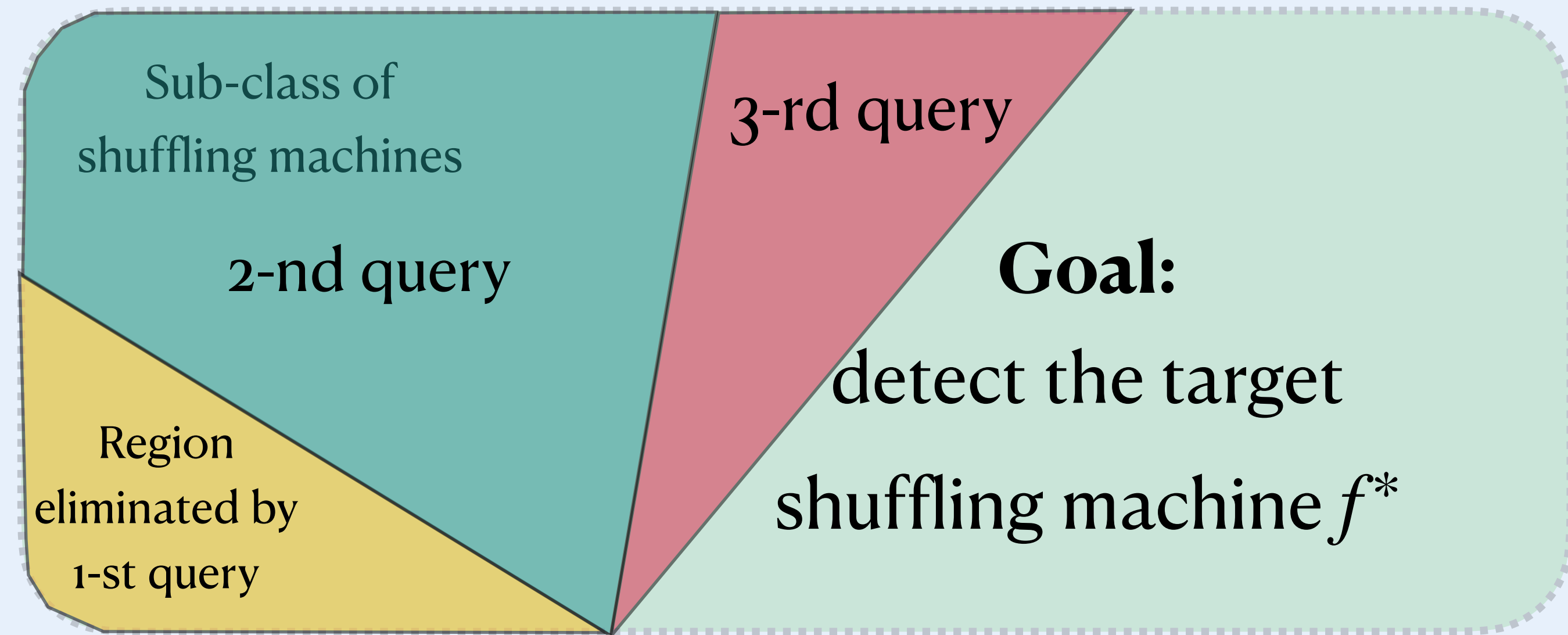
Region
eliminated by
1-st query

Goal:
detect the target
shuffling machine f^*

Eliminate machines using information from statistical queries until we learn f^*

High-level idea of the proof

Class of
semiautomata



Eliminate machines using information from statistical queries until we learn f^*

High-level idea of the proof

Class of
semiautomata

Sub-class of
shuffling machines

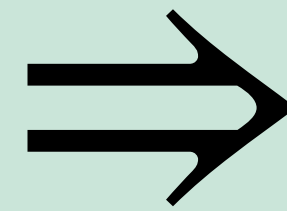
size: $N!$ + The machines are uncorrelated $\Rightarrow$ Eliminate At most polynomially many per query

High-level idea of the proof

Class of
semiautomata

Sub-class of
shuffling machines

• • •



**Superpolynomially
many queries**

Proof sketch

Step 1) Shuffling with swaps

Proof sketch

Step 1) Shuffling with swaps

Step 2) Create exponentially many shuffling machines

Proof sketch

Step 1) Shuffling with swaps

Step 2) Create exponentially many shuffling machines

Step 3) Show that they are almost uncorrelated, which implies hardness

Proof sketch

Step 1) Shuffling with swaps

Step 2) Create exponentially many shuffling machines

Step 3) Show that they are almost uncorrelated, which implies hardness

Step 3.1) Reduce correlation to a random walk

Proof sketch

Step 1) Shuffling with swaps

Step 2) Create exponentially many shuffling machines

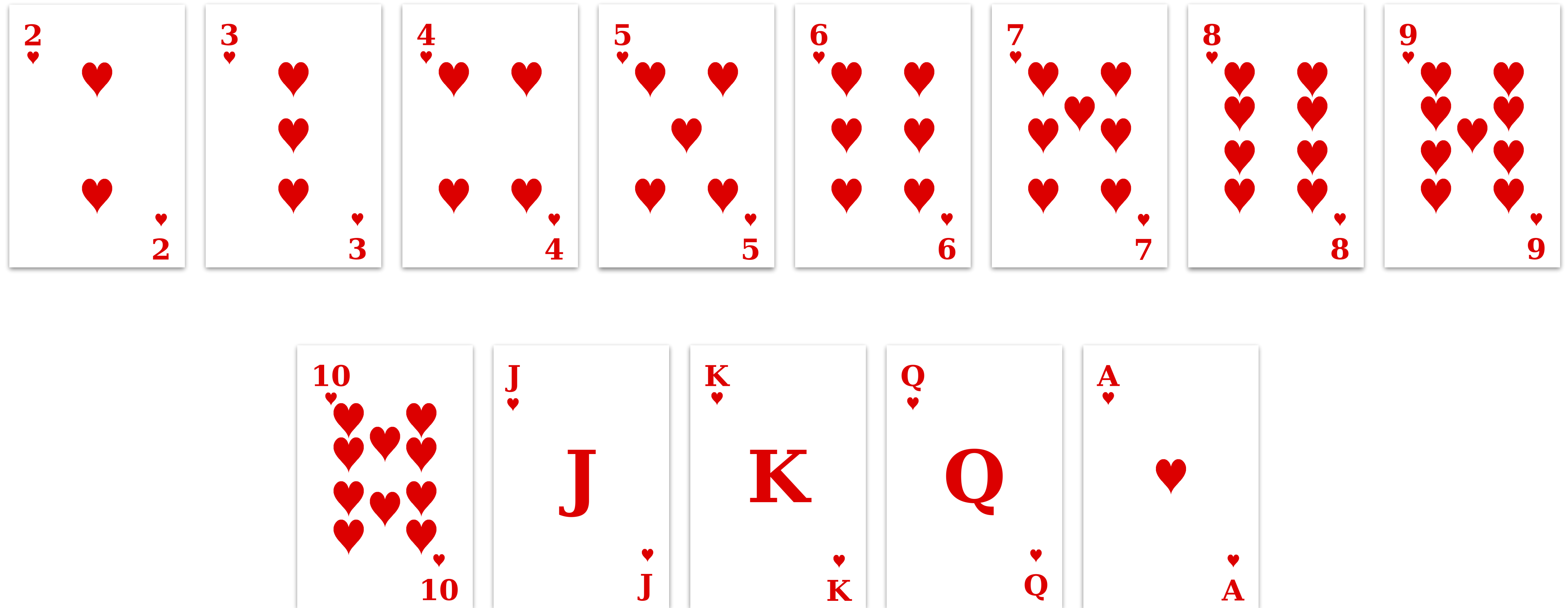
Step 3) Show that they are almost uncorrelated, which implies hardness

Step 3.1) Reduce correlation to a random walk

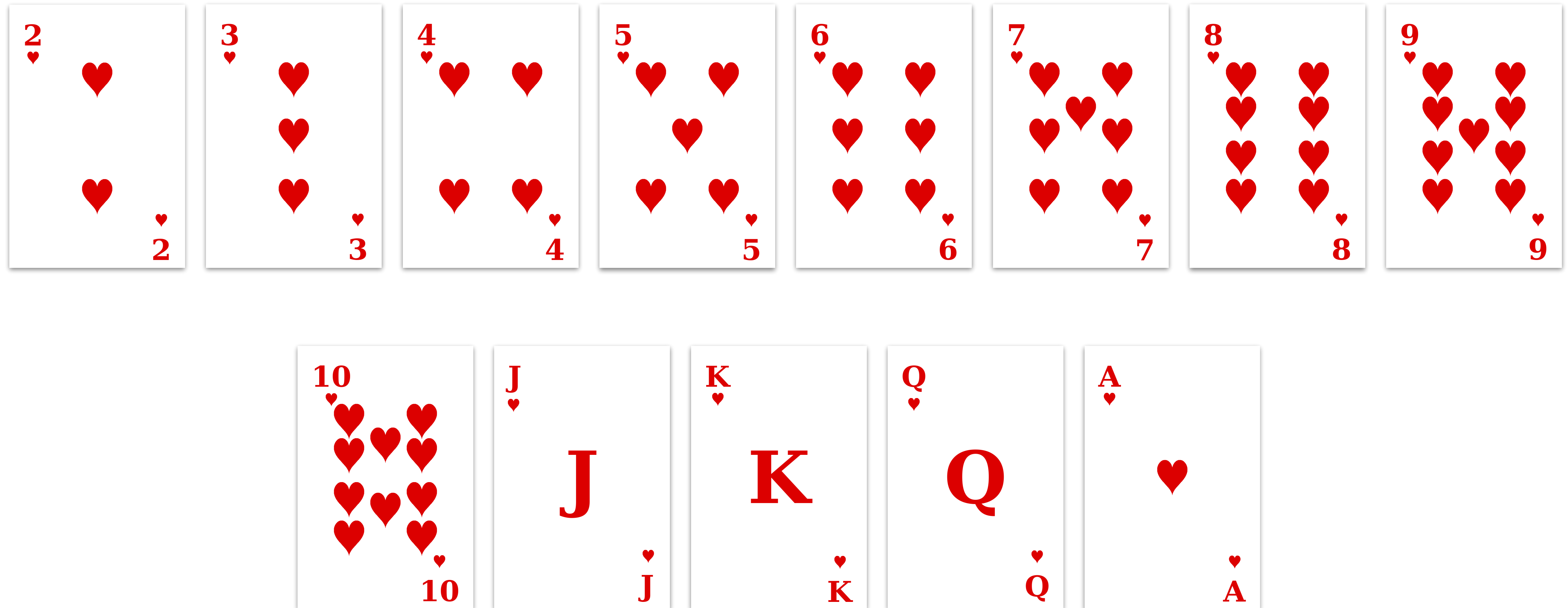
Step 3.2) Use tools from representation theory

Step 1: Shuffling with Swaps

Shuffling with swaps

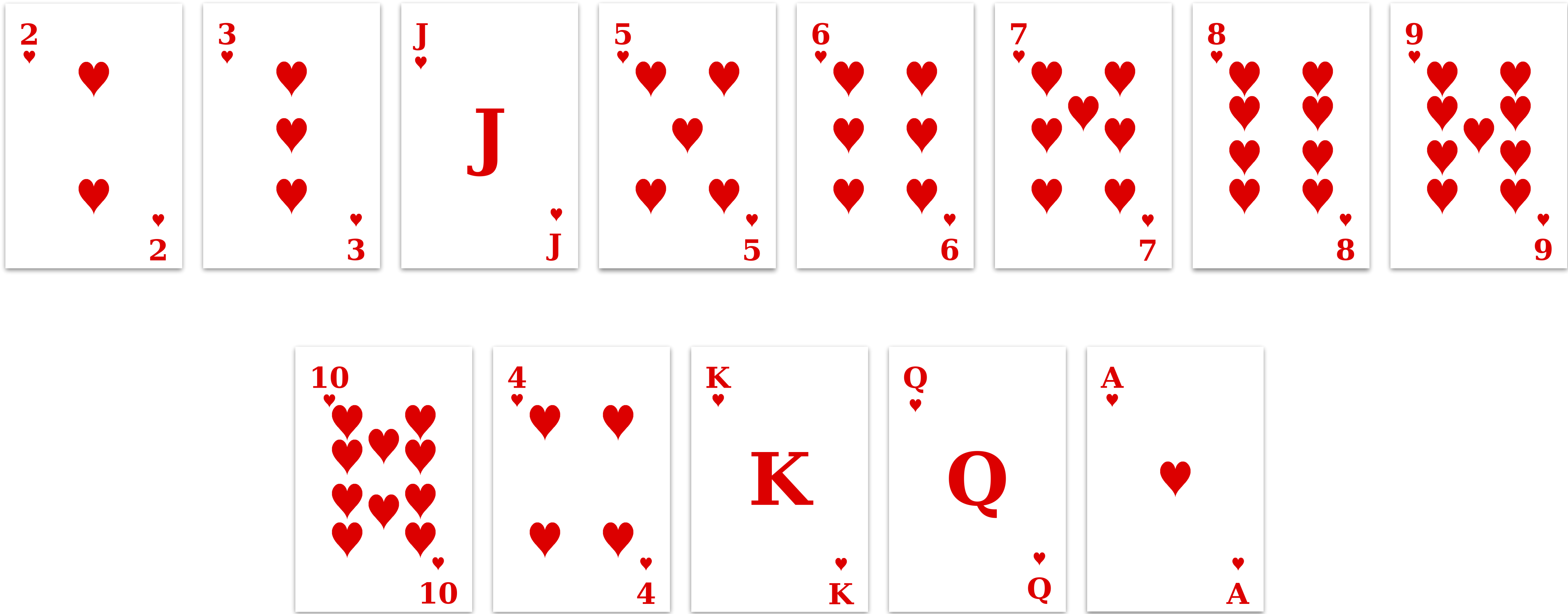


Shuffling with swaps

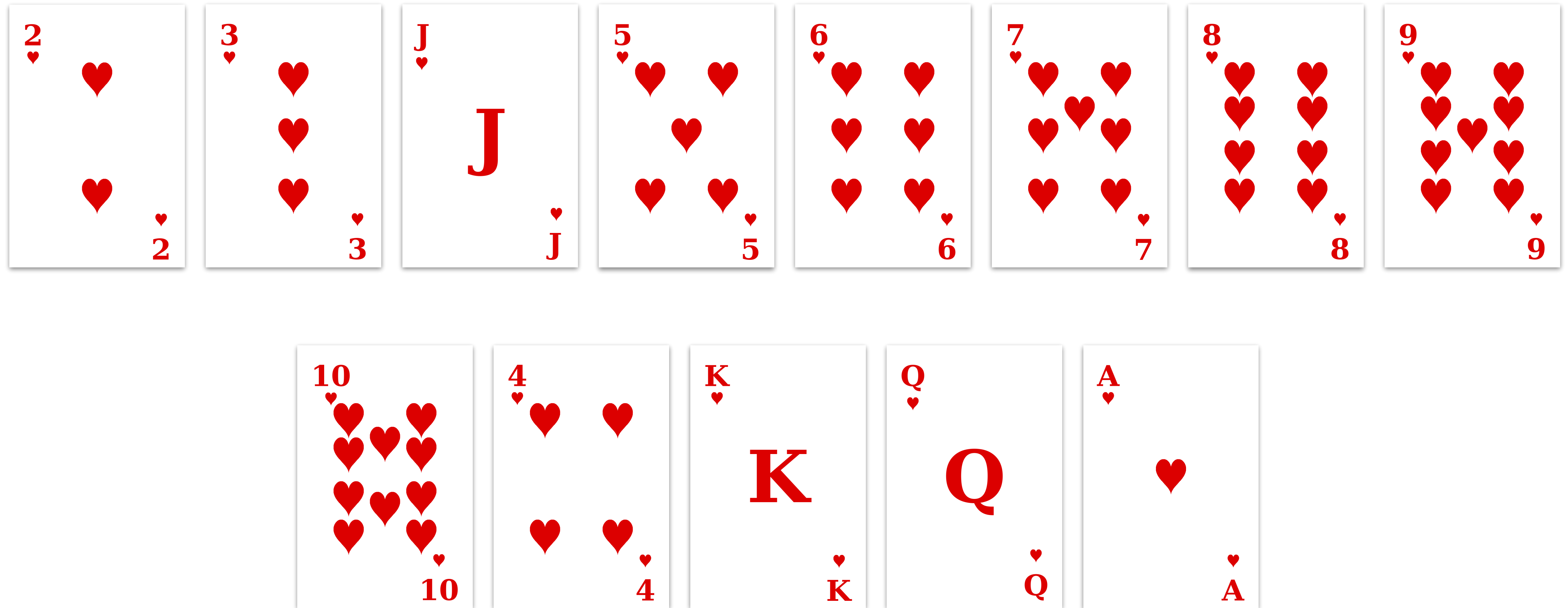


Instruction: swap positions 3 and 10

Shuffling with swaps

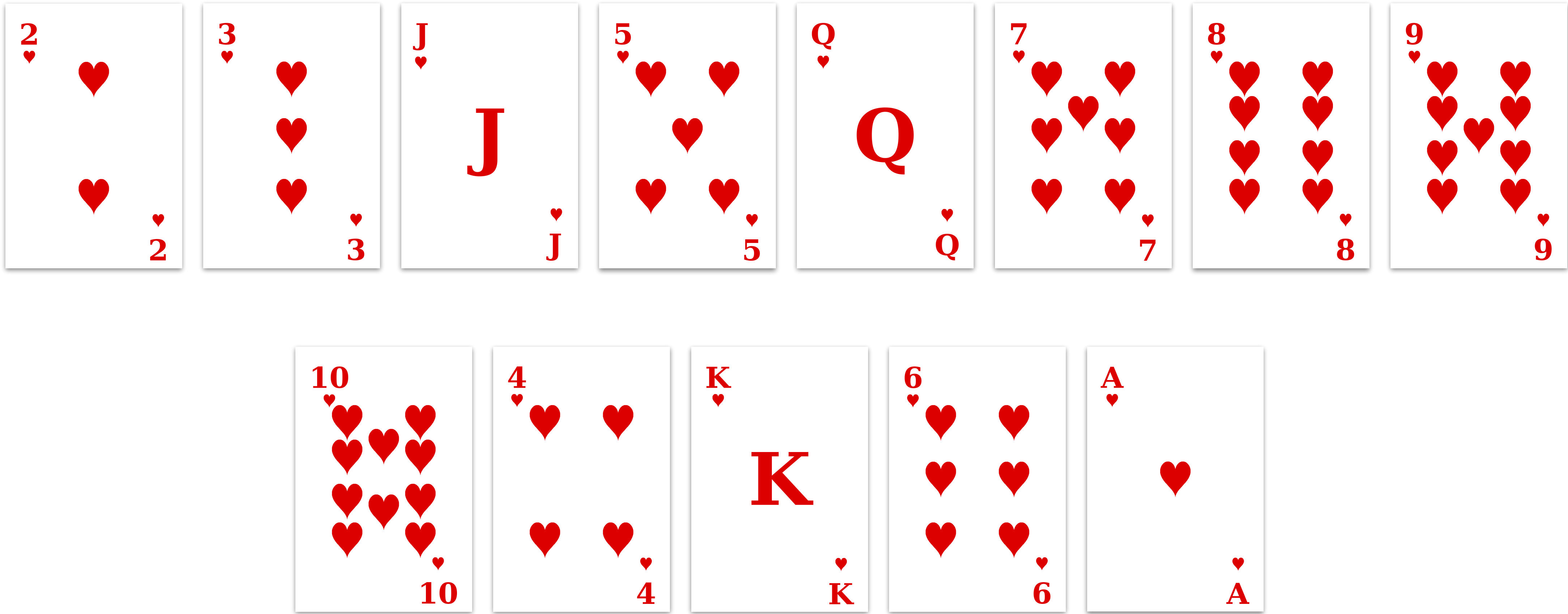


Shuffling with swaps



Instruction: swap positions 5 and 12

Shuffling with swaps



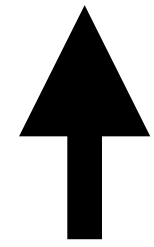
Total #swaps: $\binom{N}{2}$

Step 2: Exponentially Many Shuffling Machines

Shuffling with filters

Before

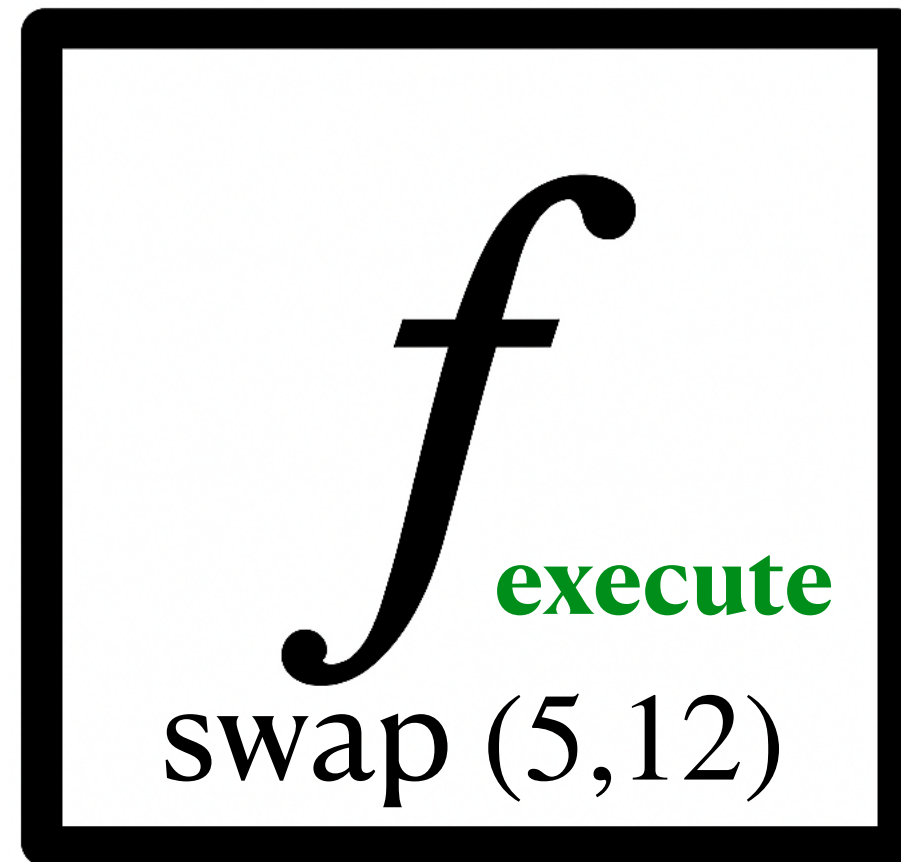
f



... swap (5,12)

Shuffling with filters

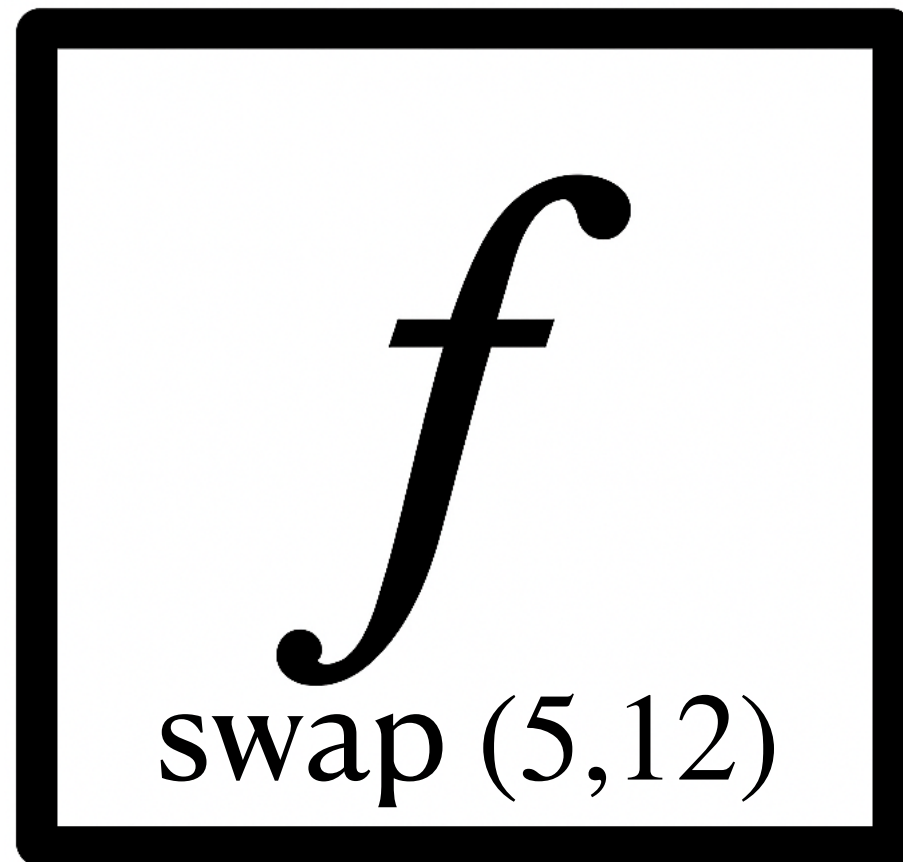
Before



... swap (3,10)

Shuffling with filters

Before

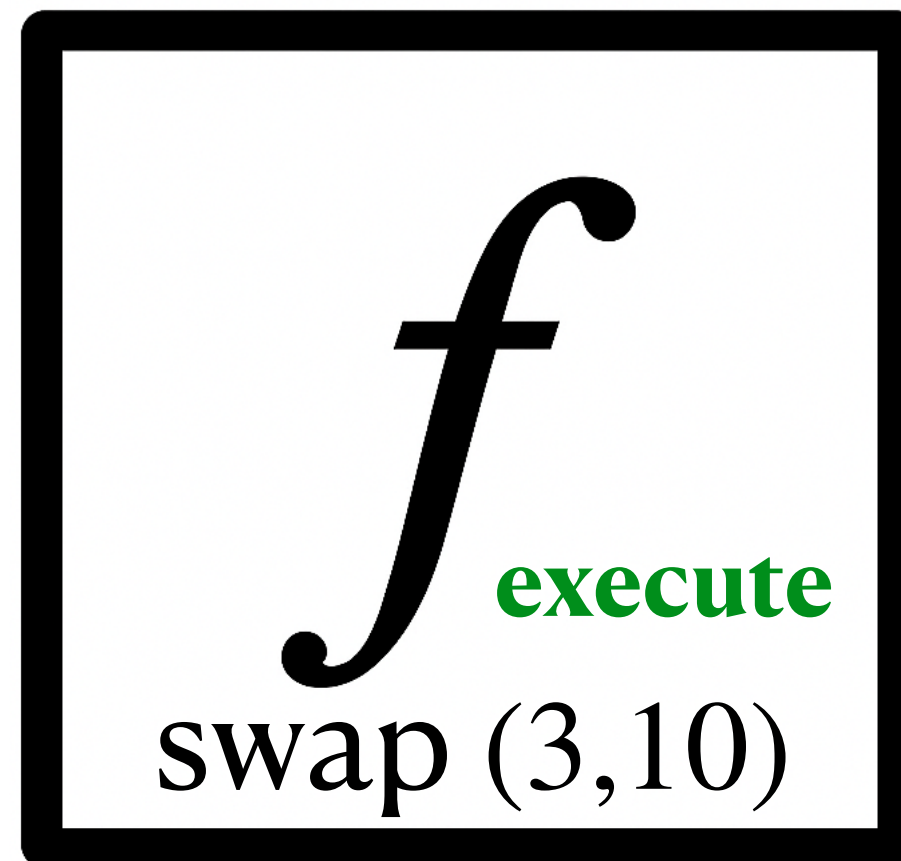


... swap (3,10)

A thick black arrow points upwards from the text "swap (3,10)" to the bottom center of the square box above.

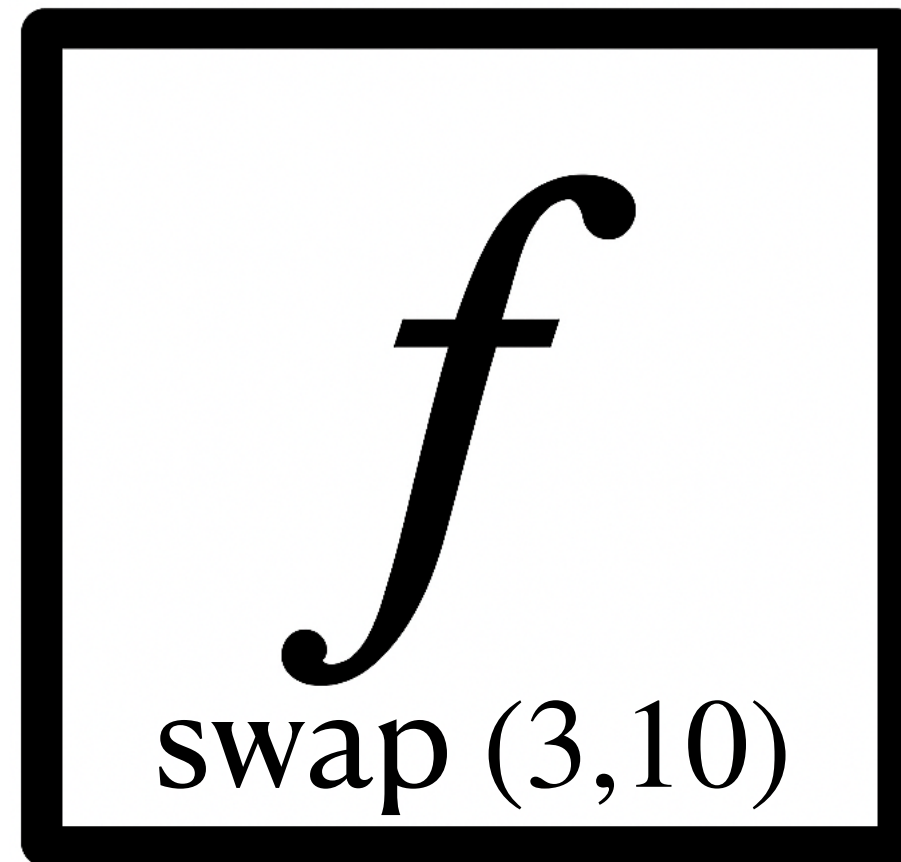
Shuffling with filters

Before

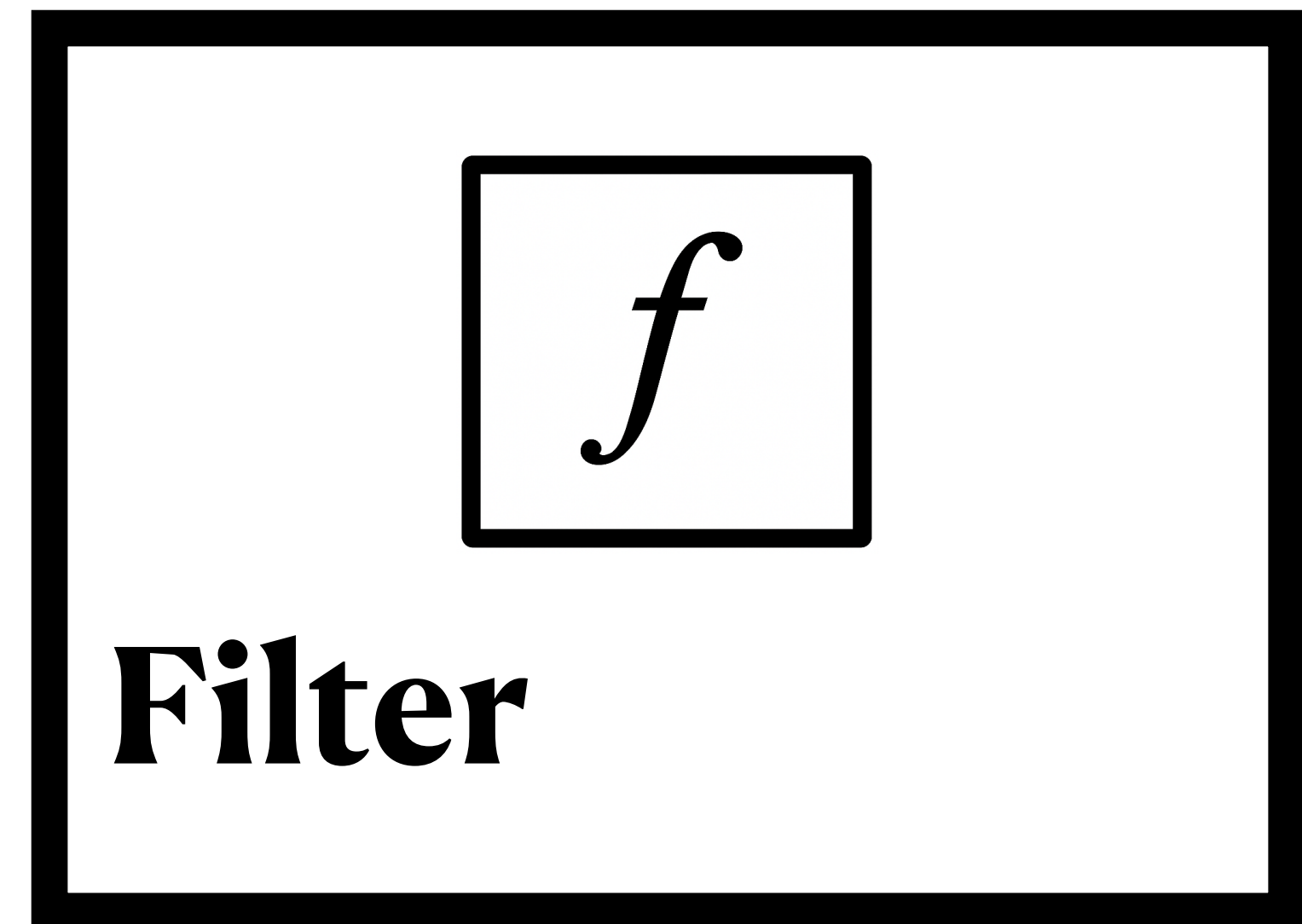


Shuffling with filters

Before



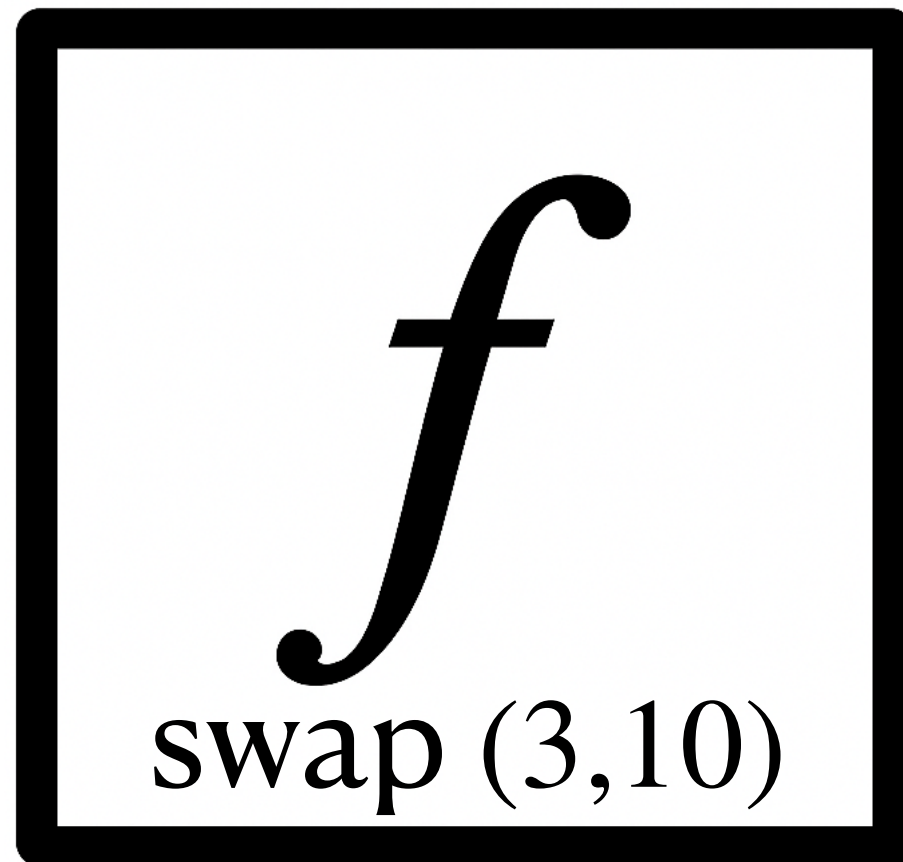
After



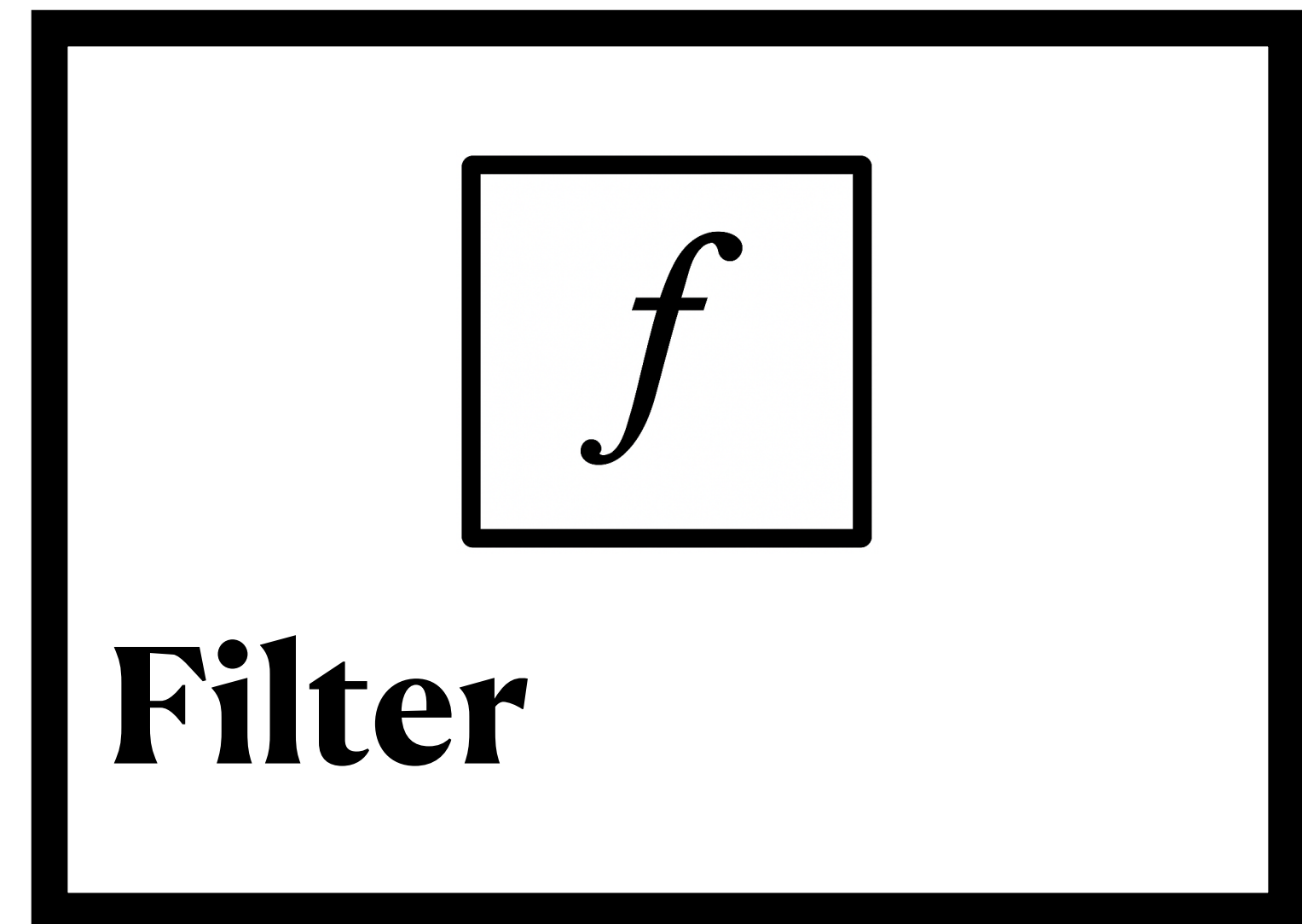
The filter decides if its going to execute an instruction or not

Shuffling with filters

Before

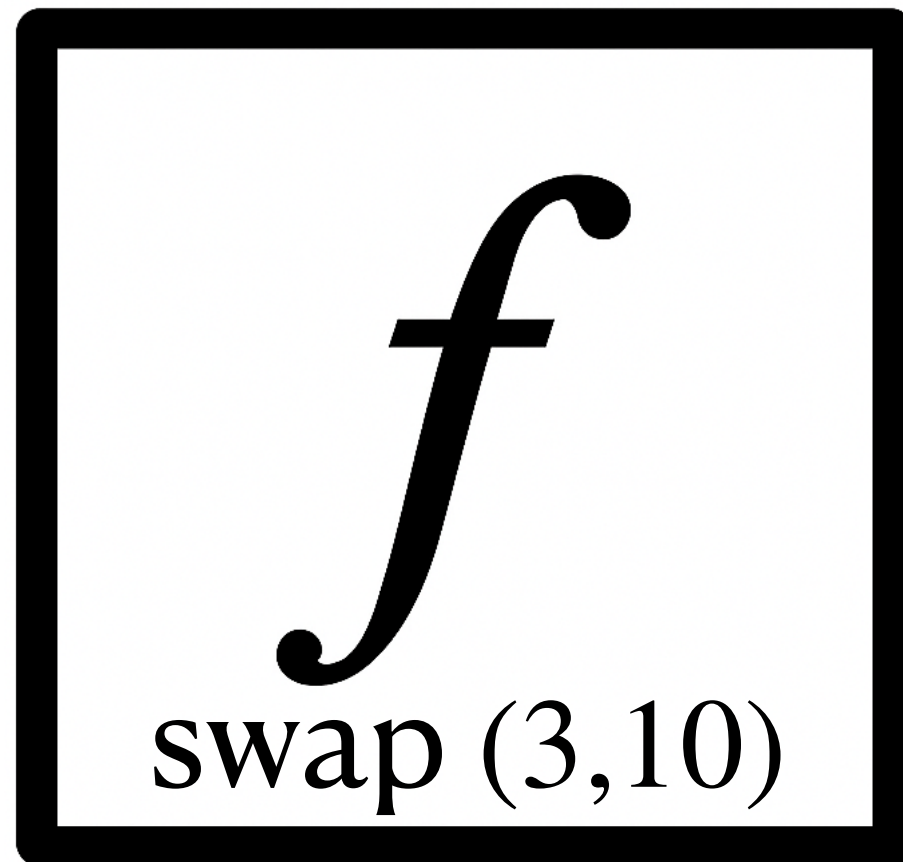


After



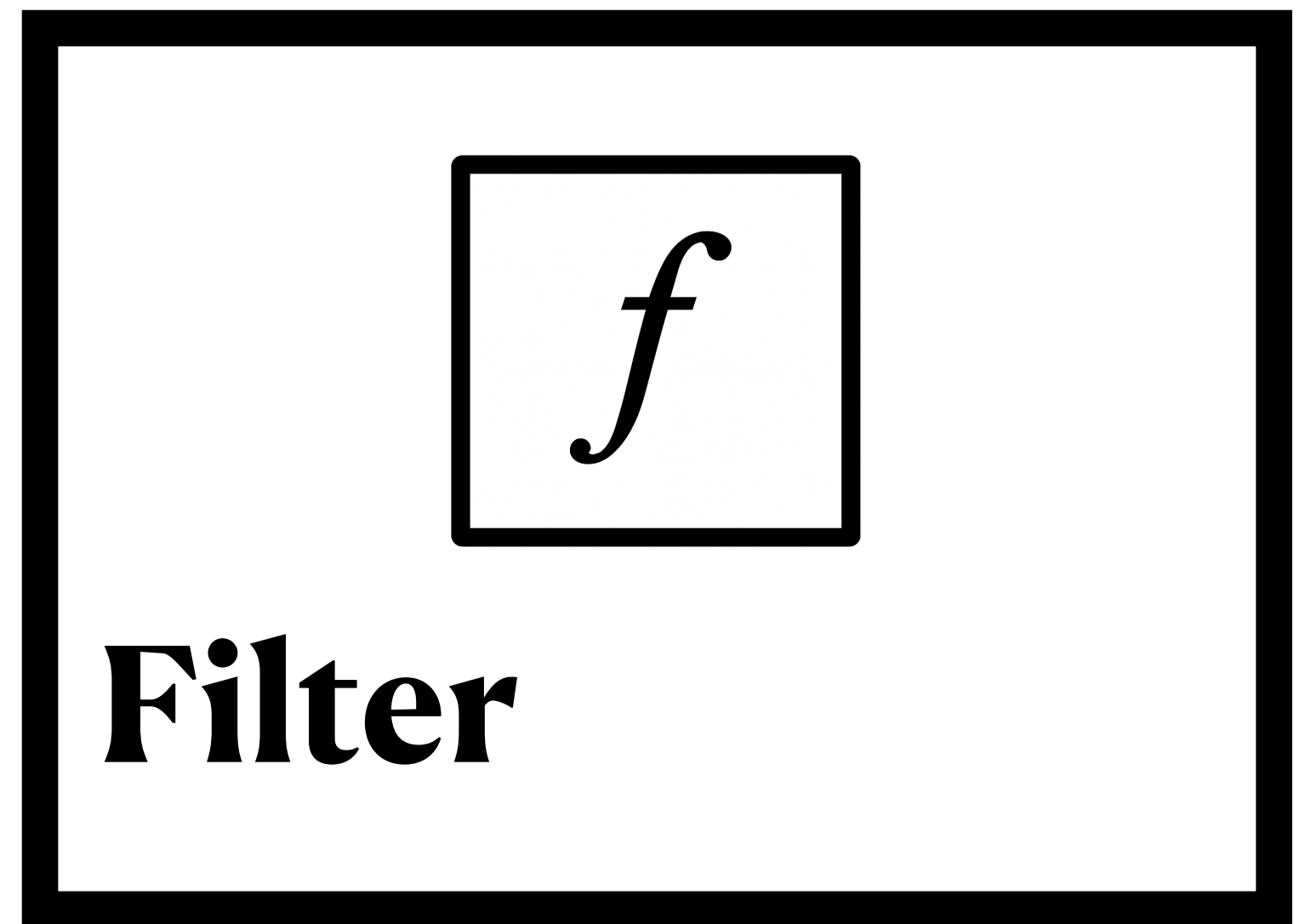
Shuffling with filters

Before



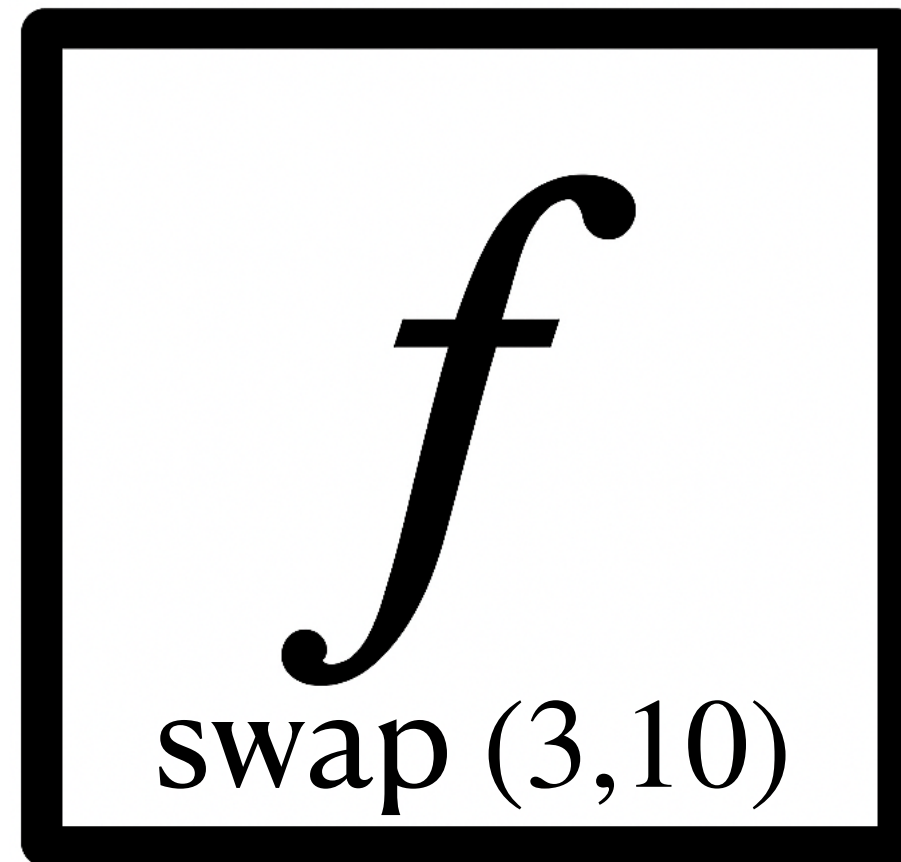
After

... swap (5,12)

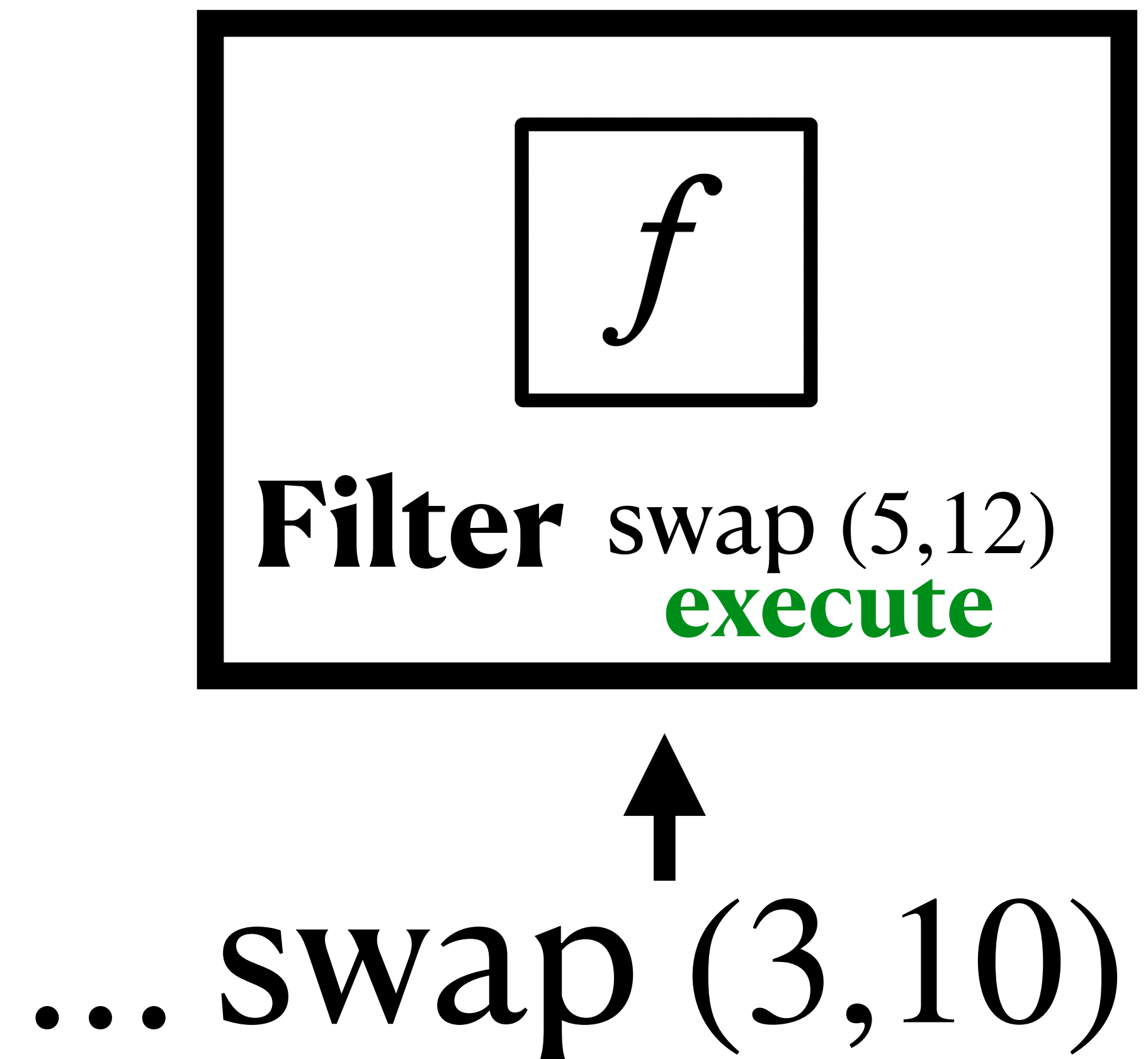


Shuffling with filters

Before

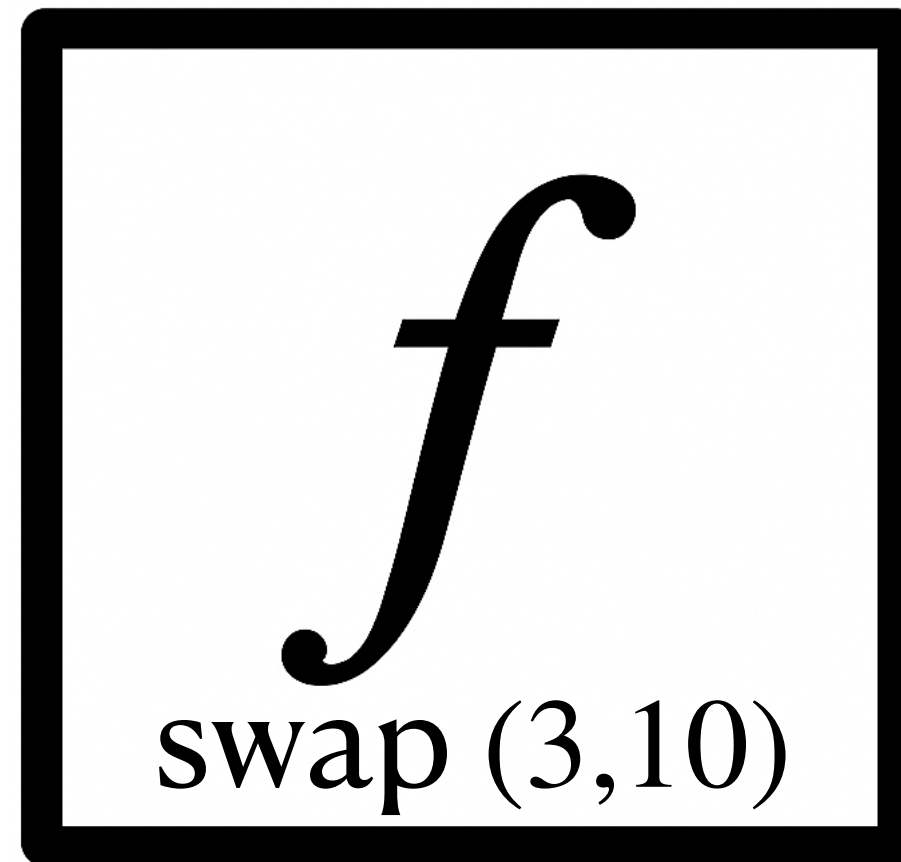


After



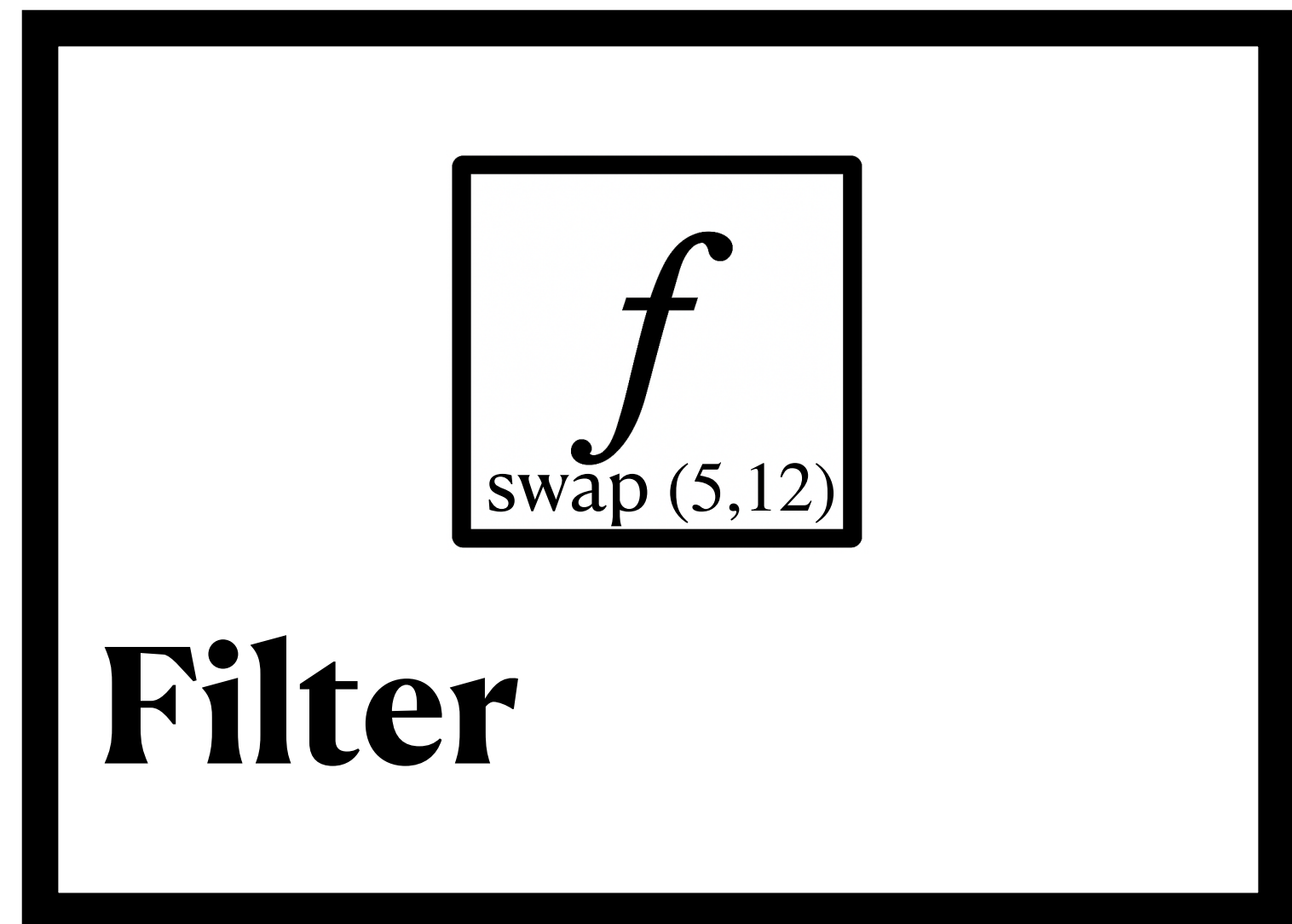
Shuffling with filters

Before



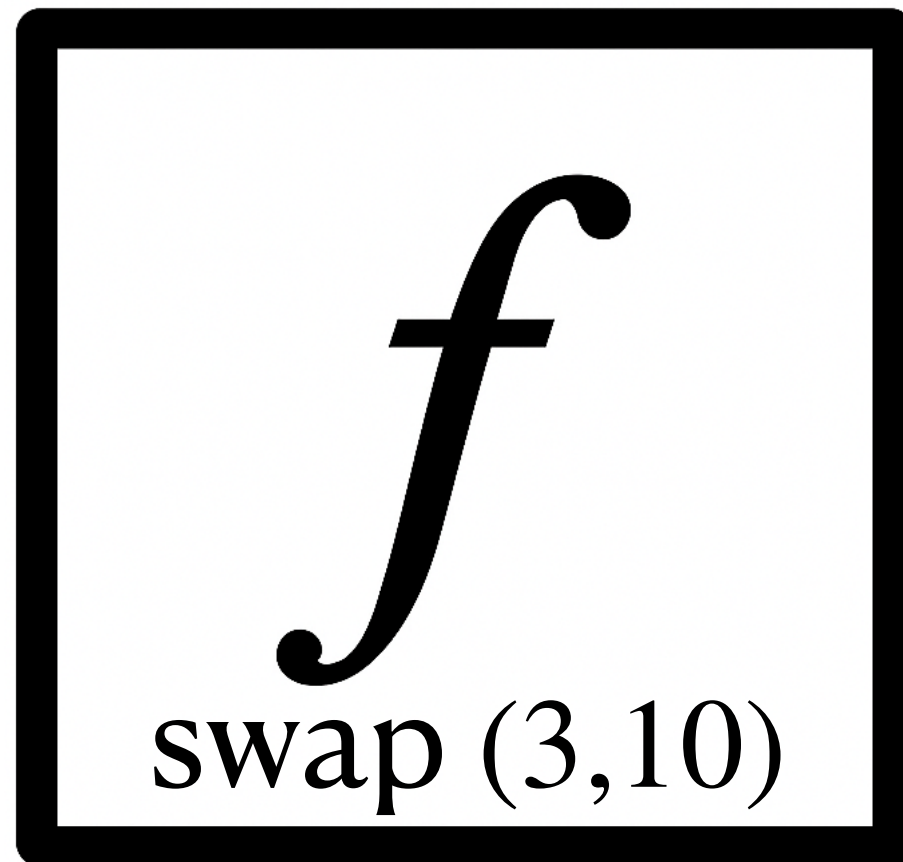
After

... swap (3,10) ↑

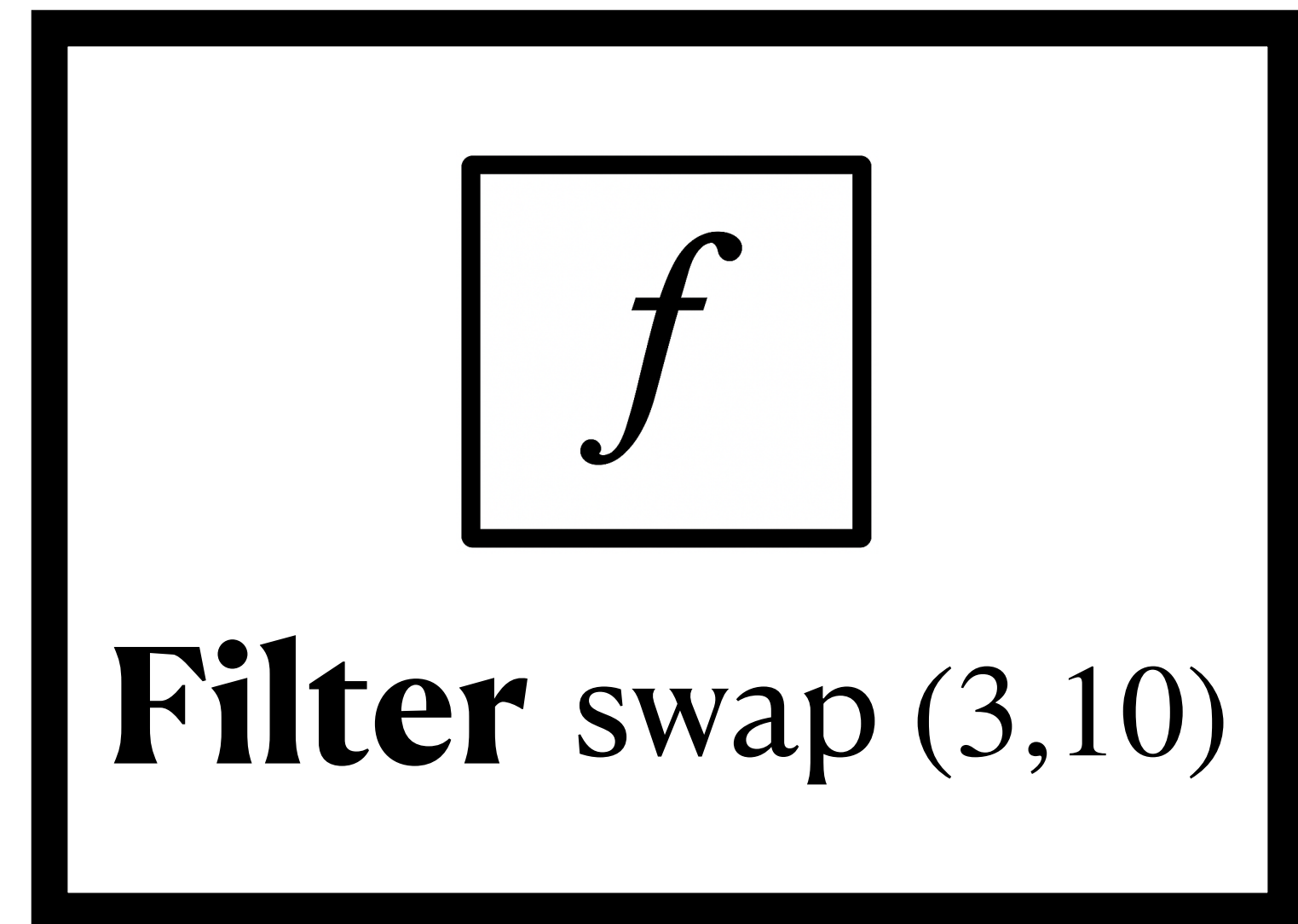


Shuffling with filters

Before

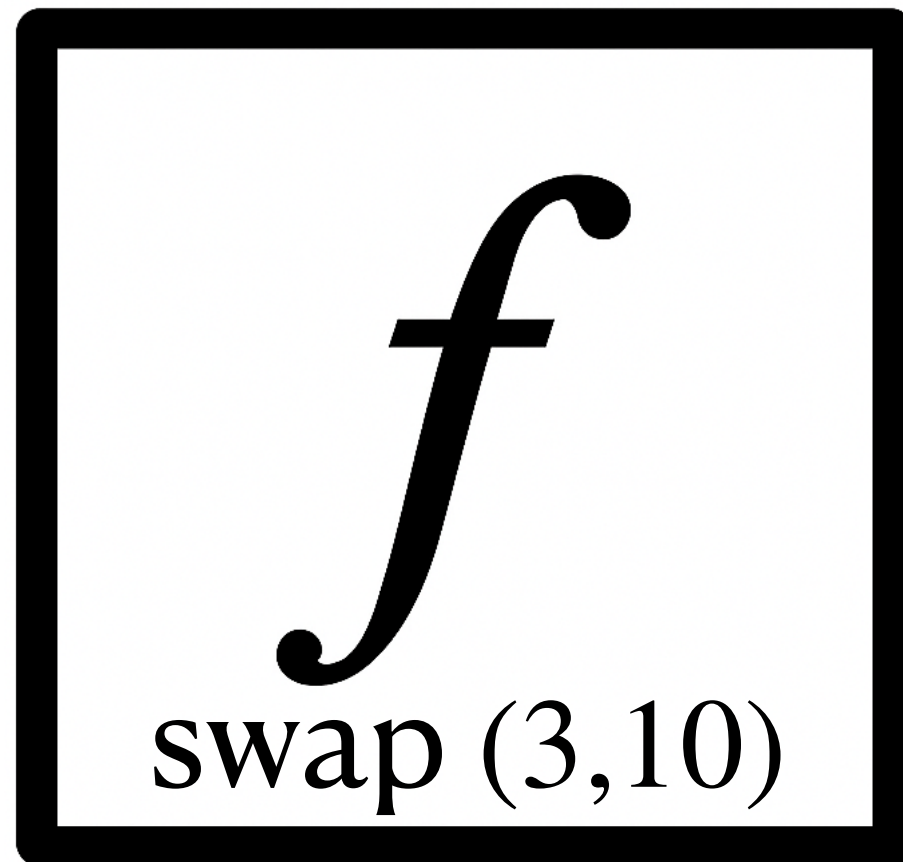


After

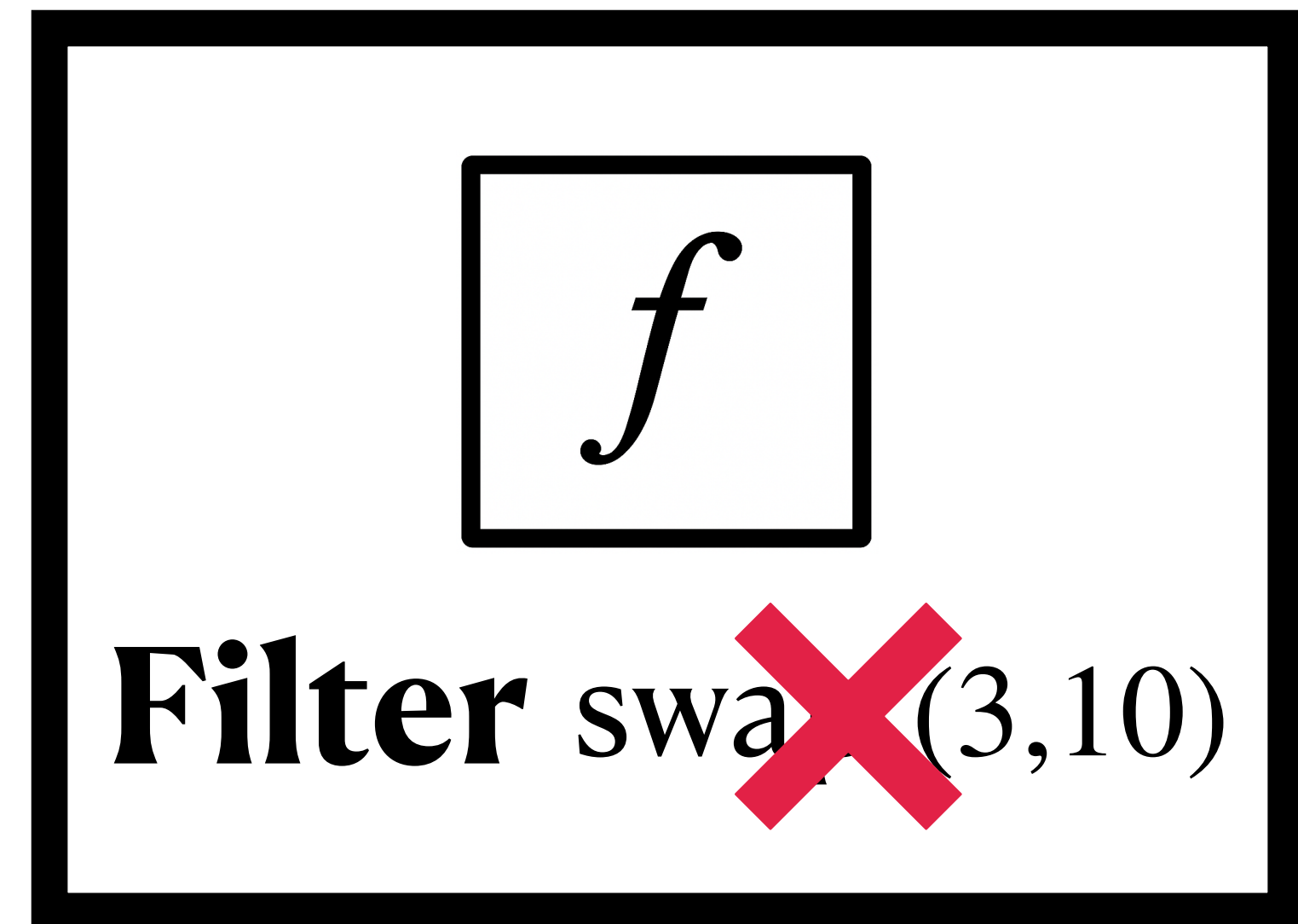


Shuffling with filters

Before



After



Filter function

swap (1,2) swap (1,3) ... swap (1, N)

swap (2,3) ... swap (2, N)

⋮

swap ($N - 1, N$)

Set of #swaps, size: $\binom{N}{2}$

Filter function



swap (1,2) swap (1,3) ... swap (1, N)



swap (2,3) ... swap (2, N)



⋮

swap ($N - 1, N$)



Filter function



swap (1,2) swap (1,3) ... swap (1, N)



swap (2,3) ... swap (2, N)

Flip all coins

⋮



swap ($N - 1, N$)

Filter function



~~swap~~(1,2) swap (1,3) ... ~~swap~~(1,N)



~~swap~~(2,3) ... swap (2,N)



⋮

Flip all coins

Tails: filter the swap

Heads: don't filter



~~swap~~(N-1,N)

Filter function



~~swap~~(1,2) swap (1,3) ... ~~swap~~(1,N)



~~swap~~(2,3) ... swap (2,N)



⋮

Flip all coins

Tails: filter the swap

Heads: don't filter

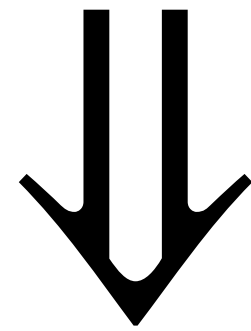
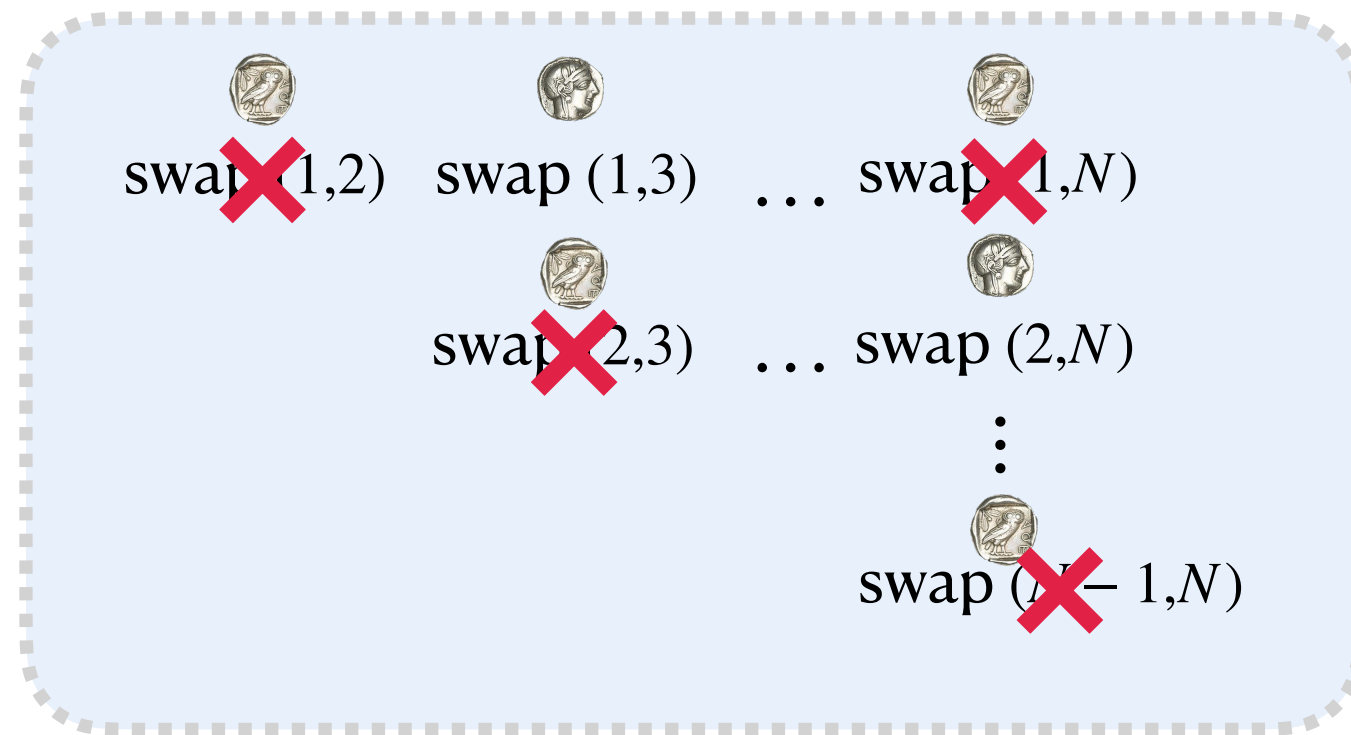
*The result is a single
filter function!*



~~swap~~(N-1,N)

Exponentially many shuffling machines

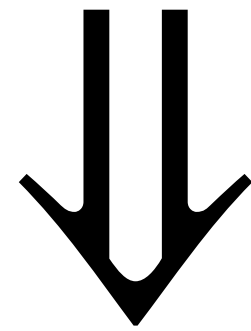
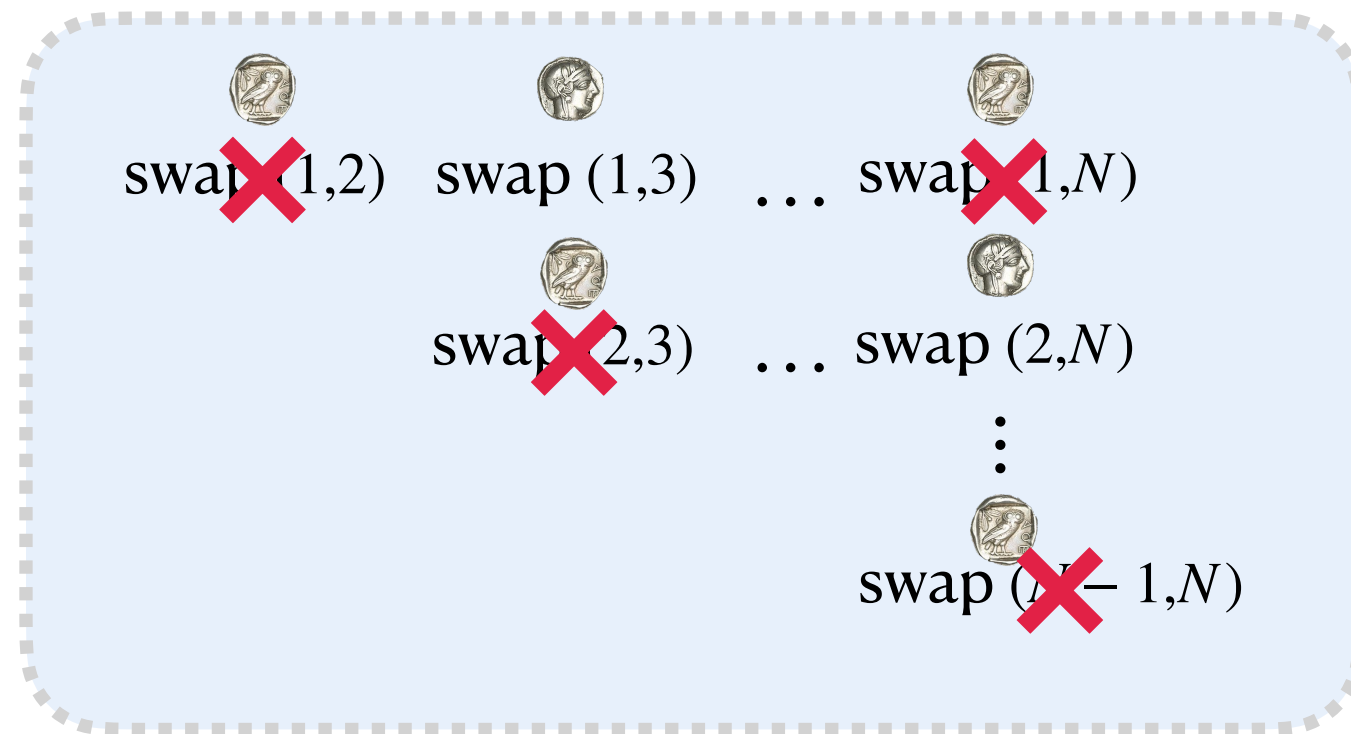
Filter function 1



New machine

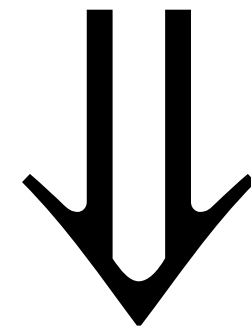
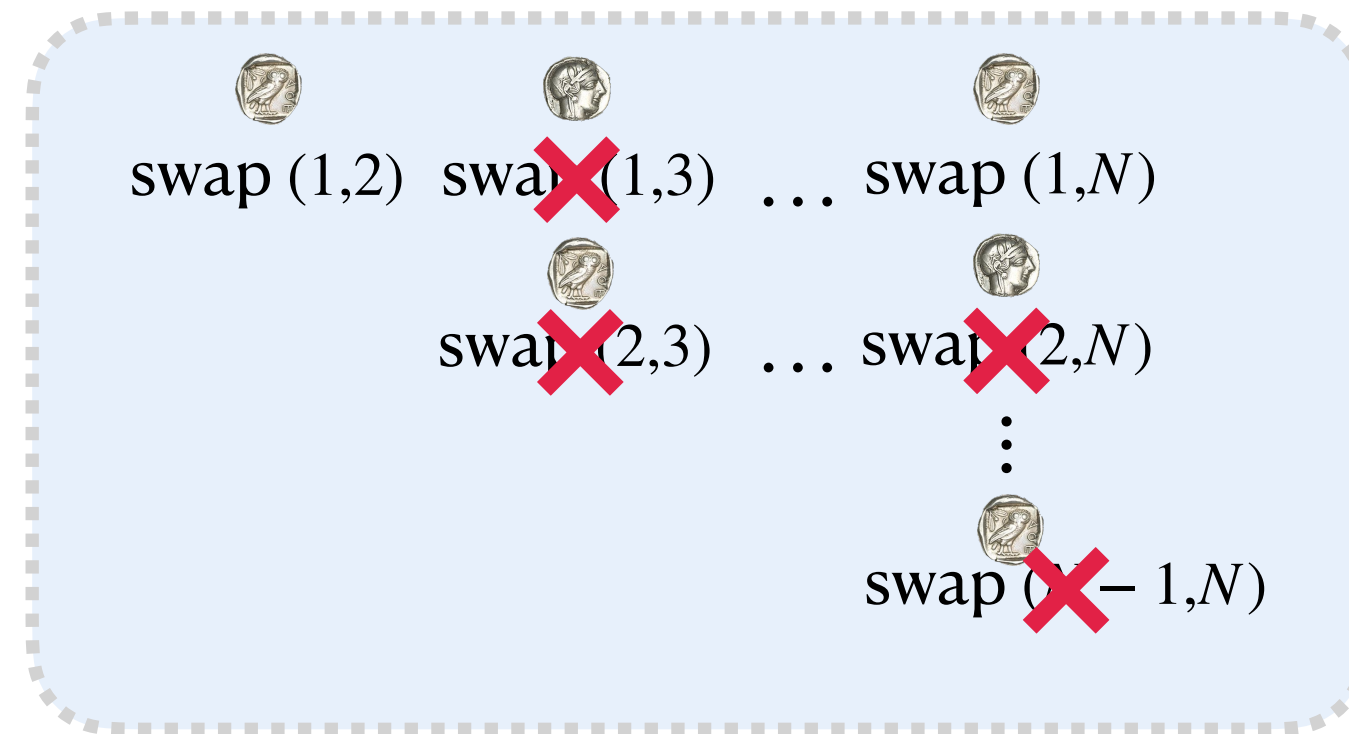
Exponentially many shuffling machines

Filter function 1



New machine

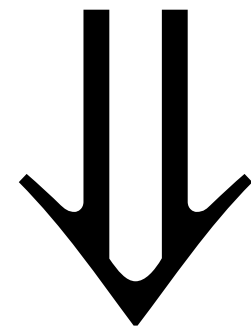
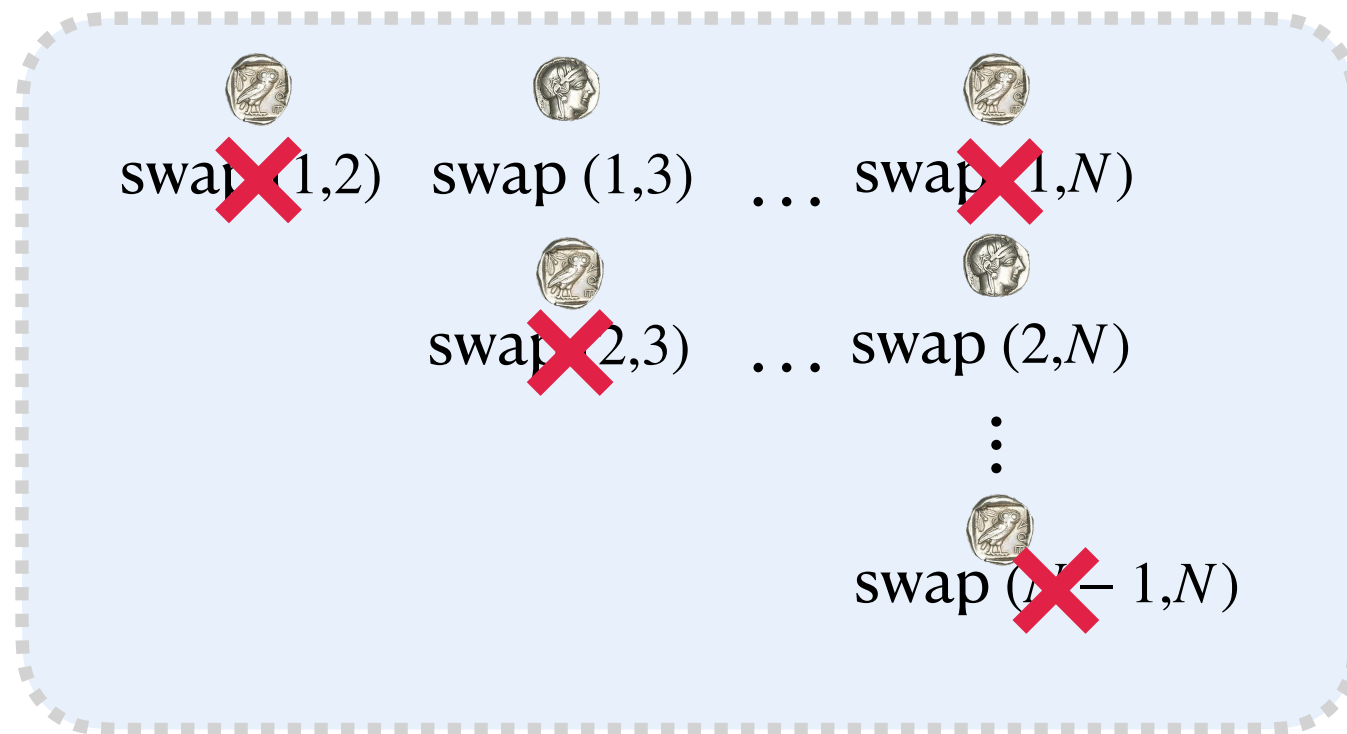
Filter function 2



New machine

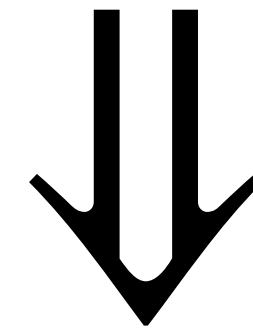
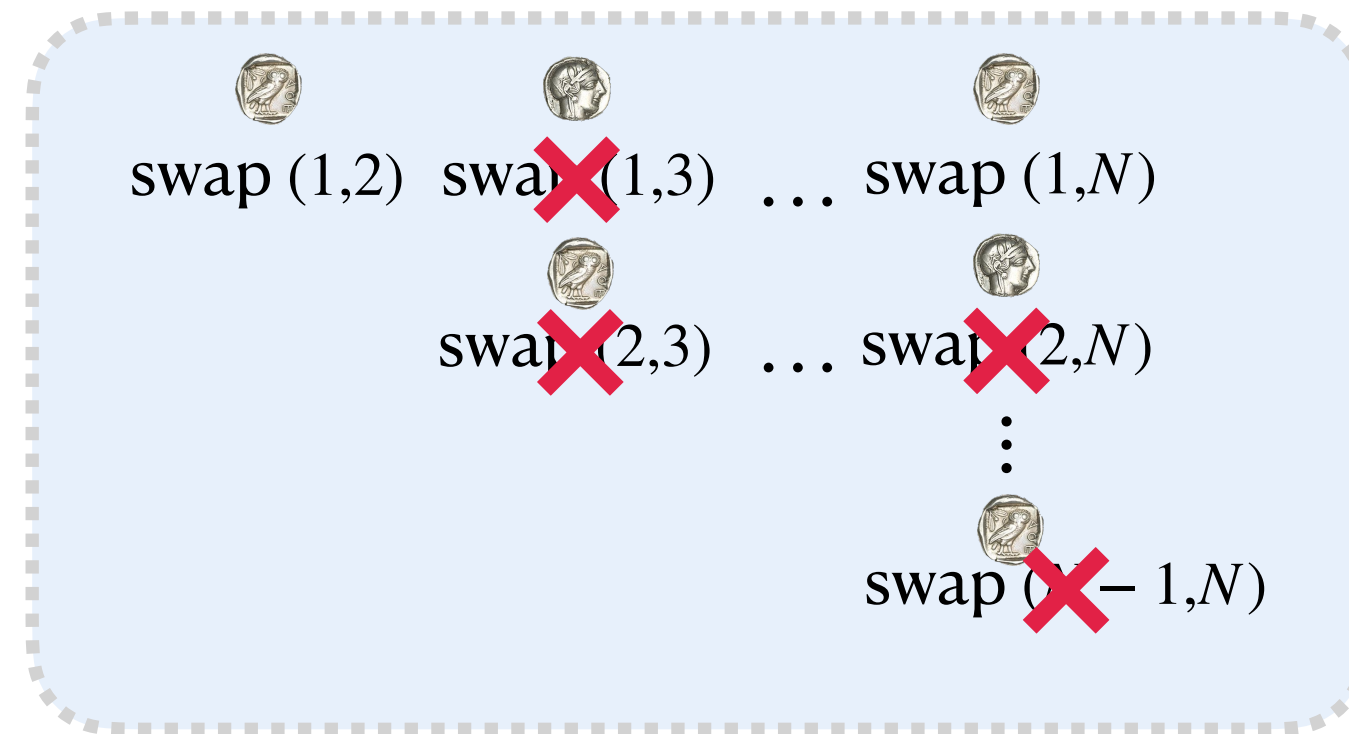
Exponentially many shuffling machines

Filter function 1



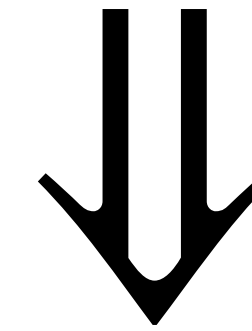
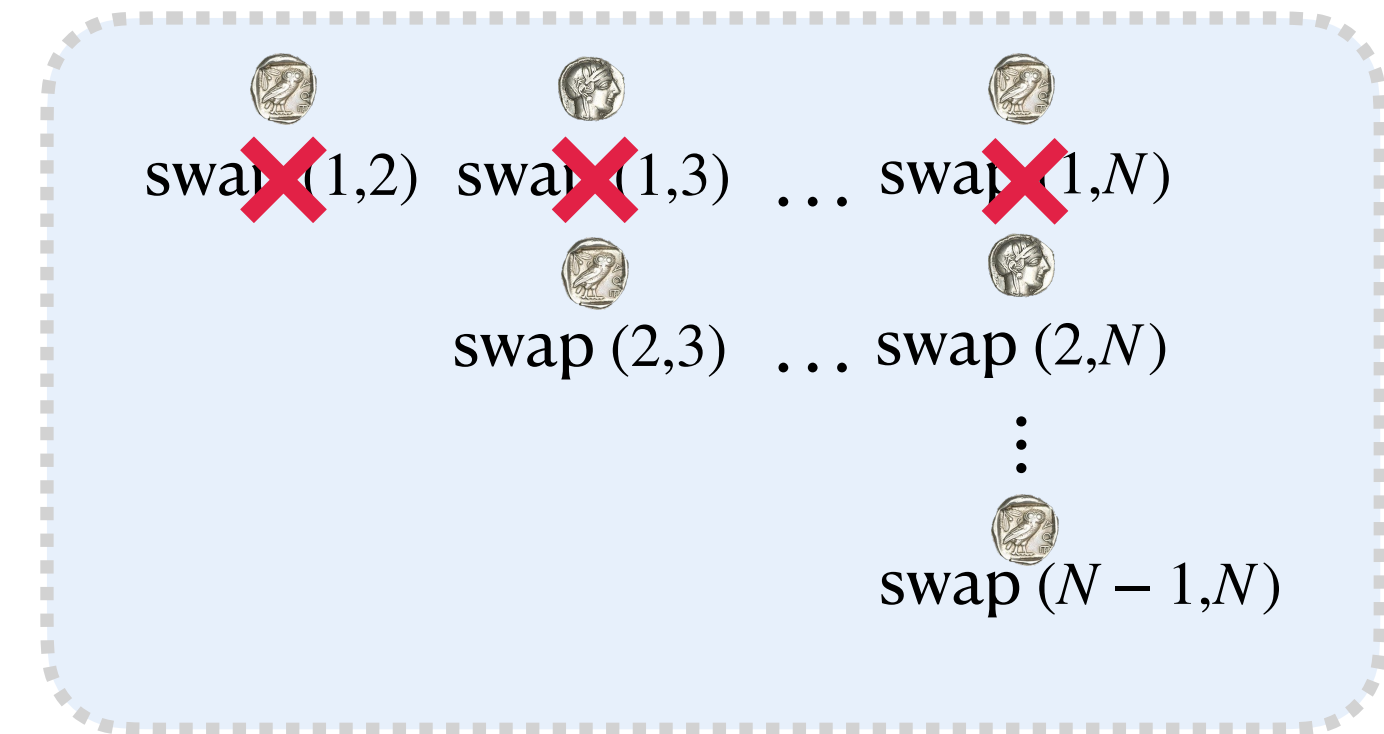
New machine

Filter function 2



New machine

Filter function 3



New machine



Repeat this experiment $N!$ times.

Step 3: Correlation of Machines and Hardness

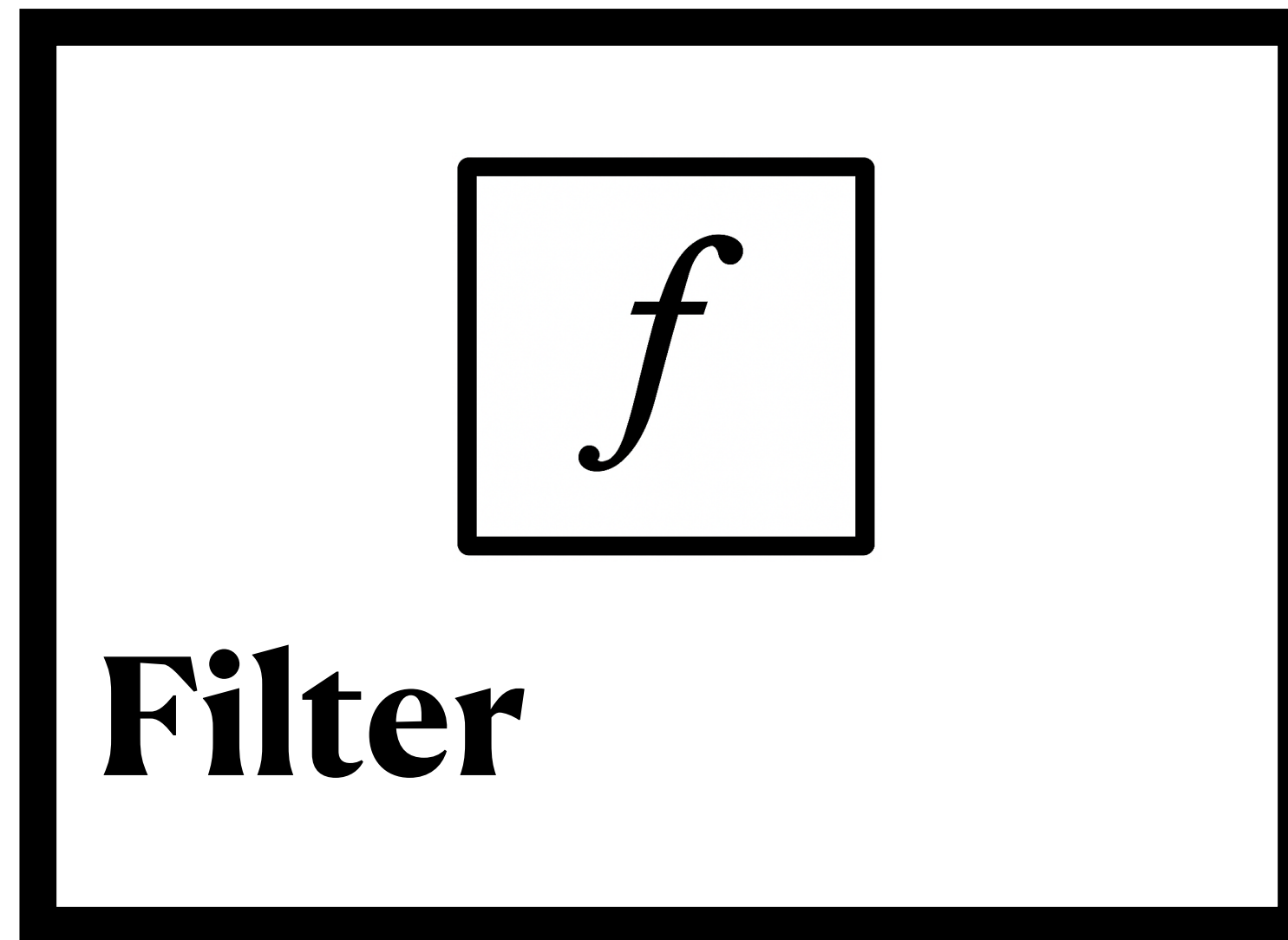
Copies of swaps

How can we make the machines uncorrelated?

$$\mathbb{P}(\text{agreement}) = \Theta\left(\frac{1}{N}\right)$$

Make $k = \Theta(N \log N)$ copies of each swap instruction.

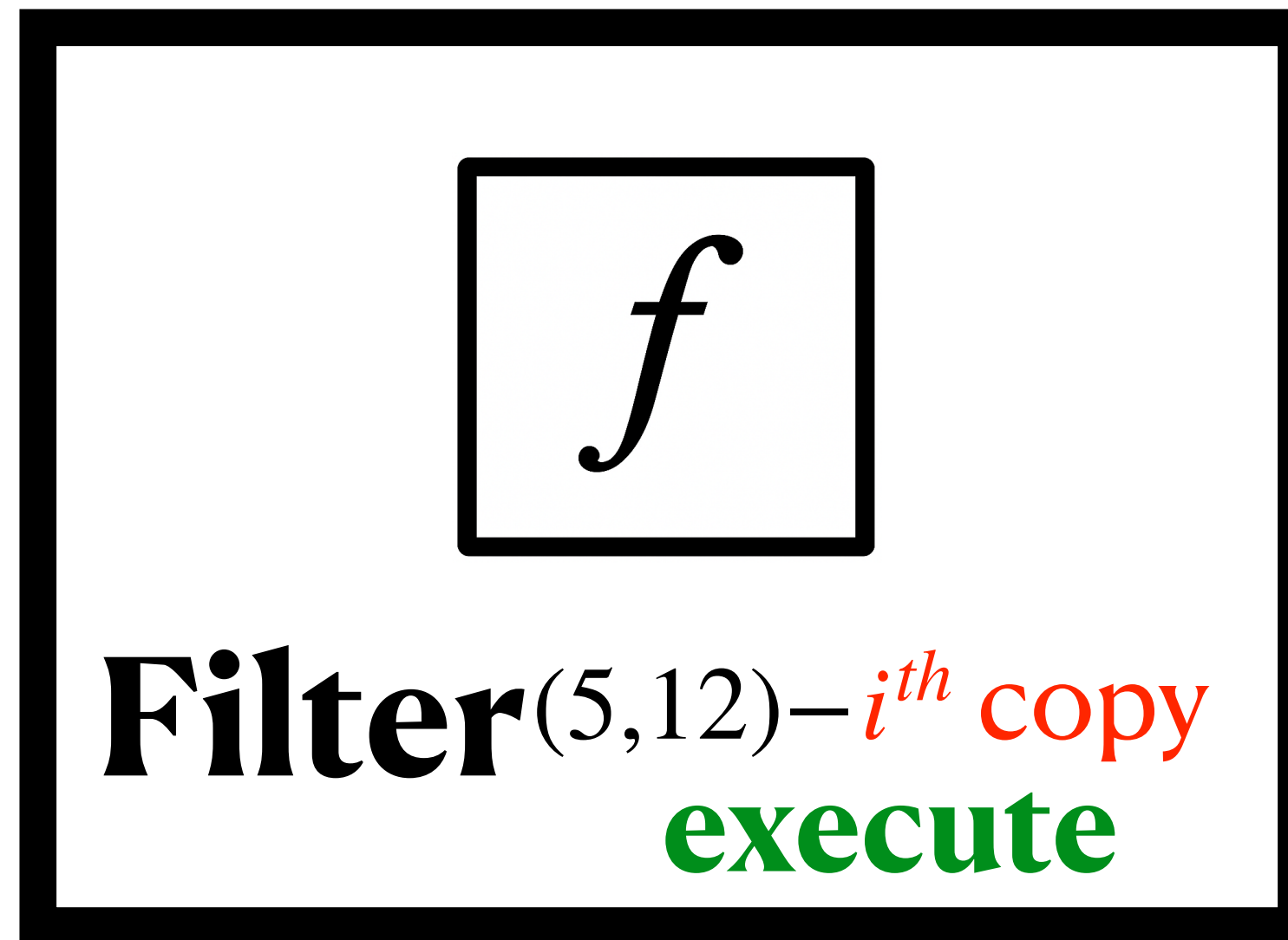
Copies of swaps



... (5,12) — i^{th} copy

An upward-pointing arrow is positioned between the text and the Filter block, pointing from the text to the bottom edge of the Filter block.

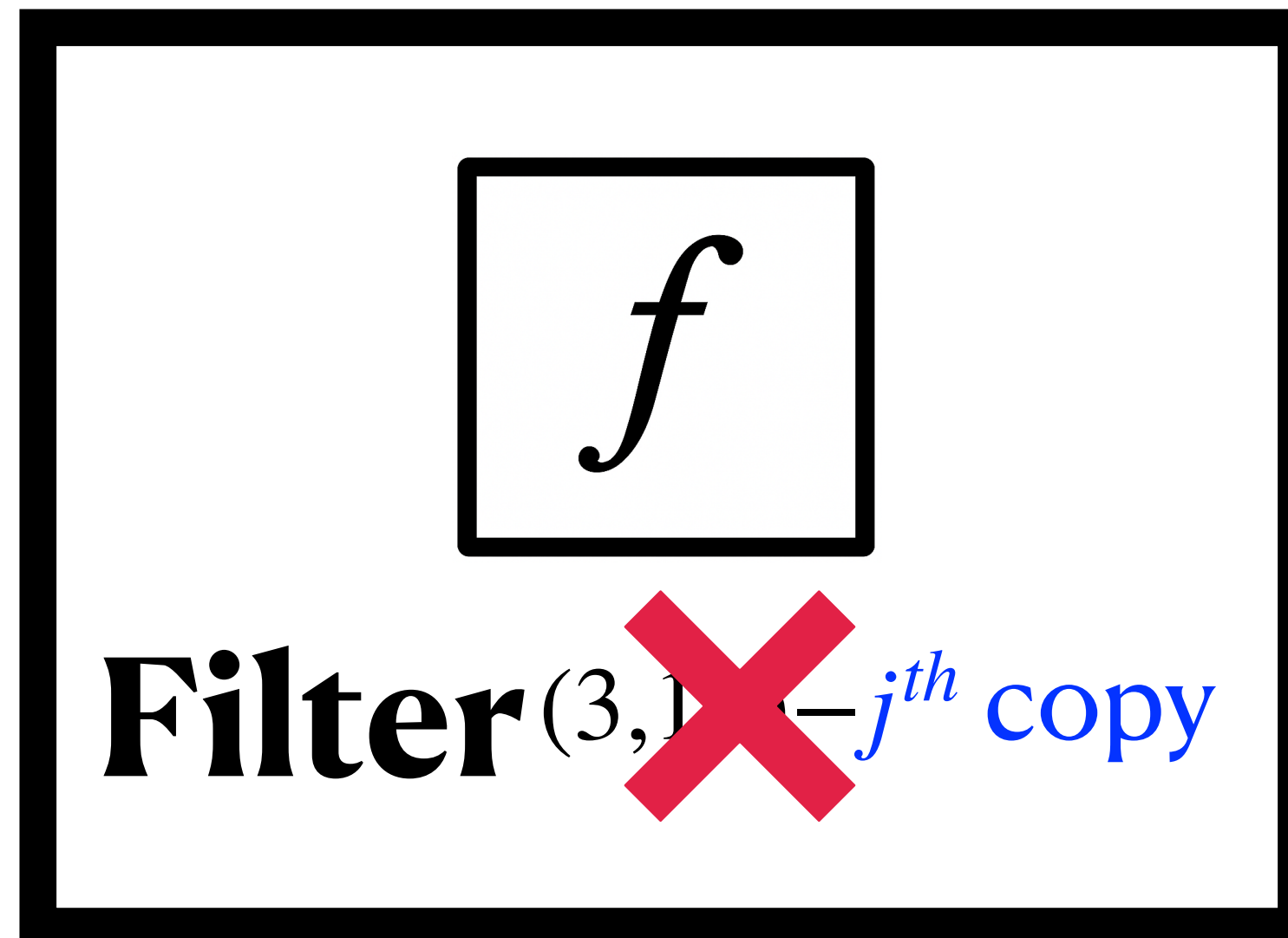
Copies of swaps



... $(3,10)-j^{th}$ copy

An upward-pointing arrow is positioned between the text above and the text below.

Copies of swaps



Copies of swaps

swap (1,2) swap (1,3) ... swap (1, N)

swap (2,3) ... swap (2, N)

⋮

swap ($N - 1, N$)

1-st copy

swap (1,2) swap (1,3) ... swap (1, N)

swap (2,3) ... swap (2, N)

⋮

swap ($N - 1, N$)

2-nd copy

• • •

swap (1,2) swap (1,3) ... swap (1, N)

swap (2,3) ... swap (2, N)

⋮

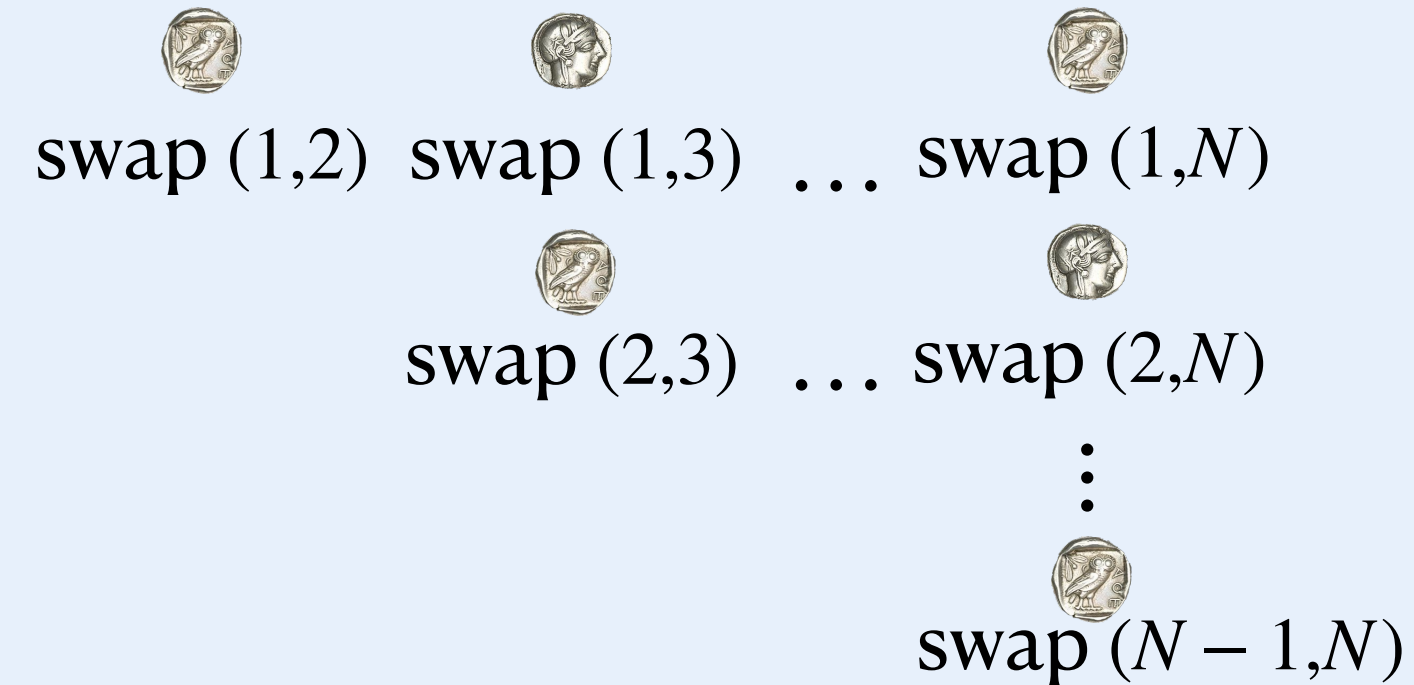
swap ($N - 1, N$)

k-th copy

Make $k = \Theta(N \log N)$ copies of each swap.

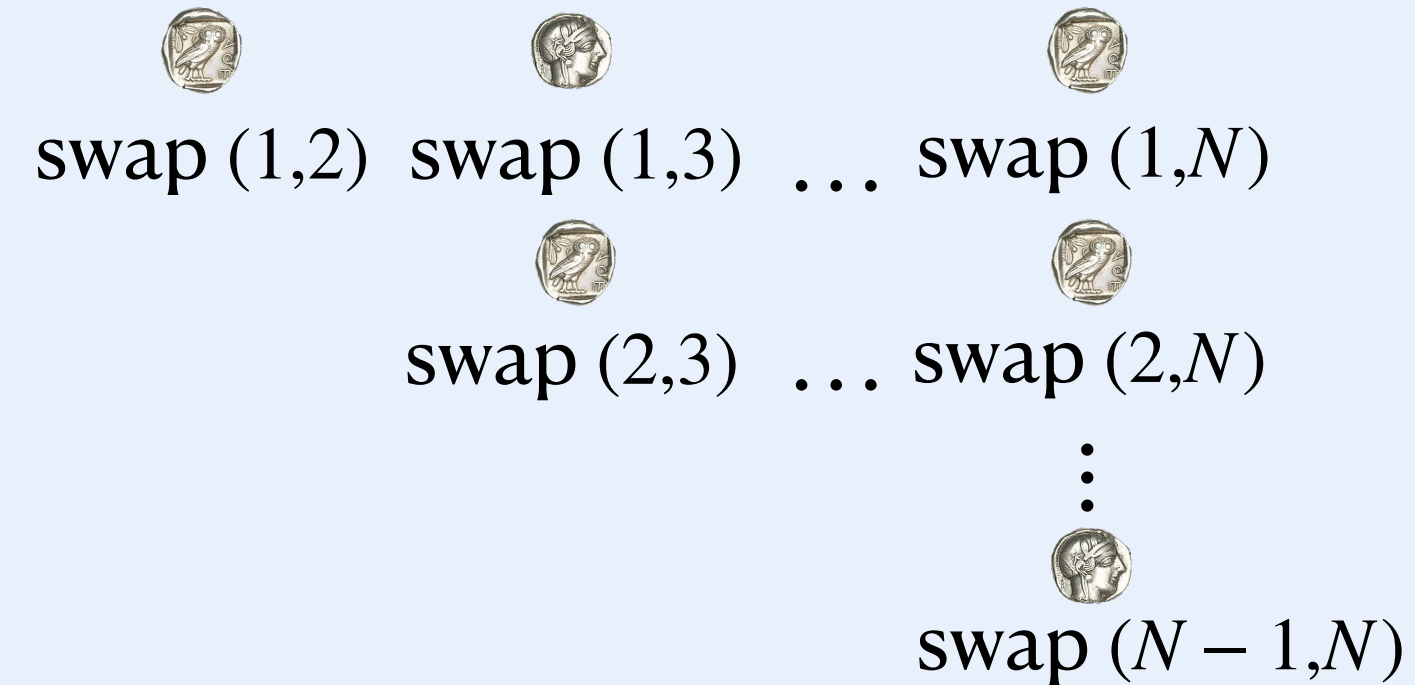
Total #swaps: $k \binom{N}{2}$

Copies of swaps



$\text{swap}(1,2)$ $\text{swap}(1,3)$ $\dots$ $\text{swap}(1,N)$
 $\text{swap}(2,3)$ $\dots$ $\text{swap}(2,N)$
 $\vdots$
 $\text{swap}(N-1,N)$

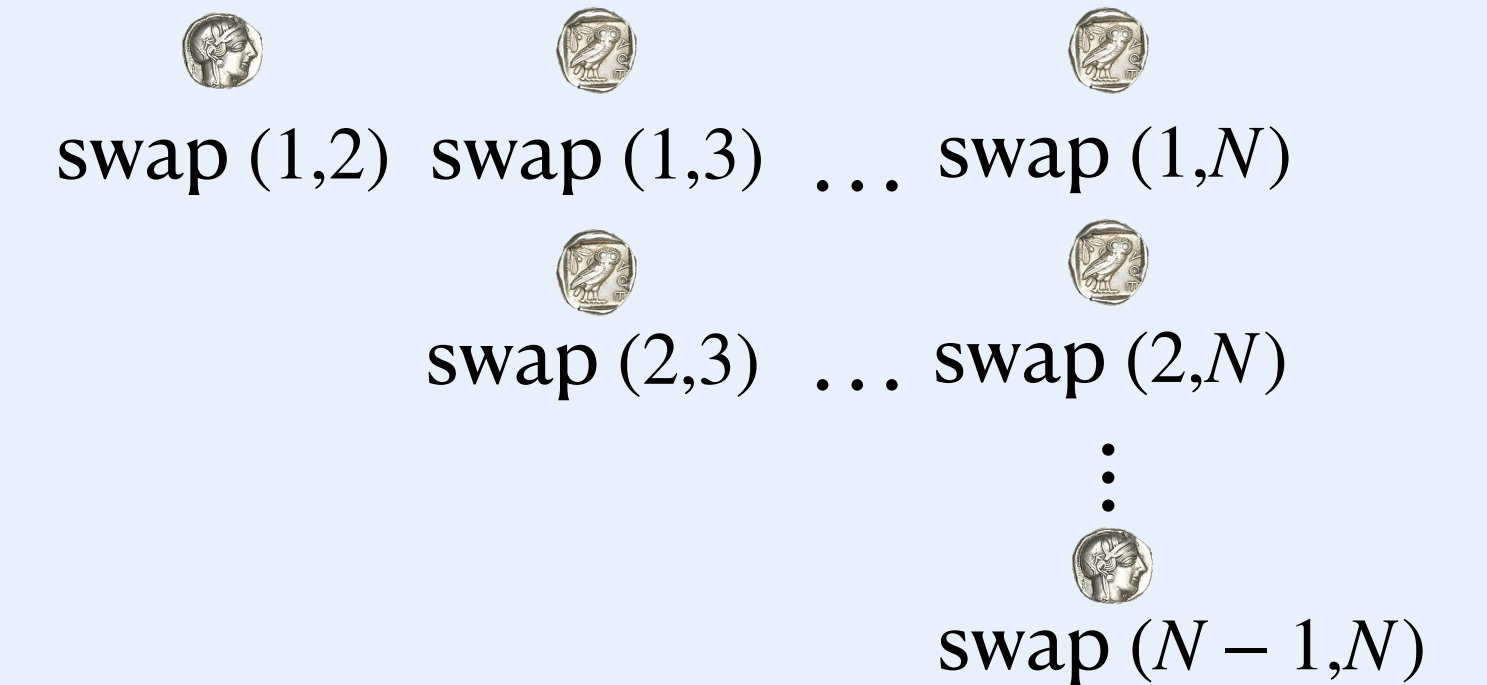
1-st copy



$\text{swap}(1,2)$ $\text{swap}(1,3)$ $\dots$ $\text{swap}(1,N)$
 $\text{swap}(2,3)$ $\dots$ $\text{swap}(2,N)$
 $\vdots$
 $\text{swap}(N-1,N)$

2-nd copy

$\dots$



$\text{swap}(1,2)$ $\text{swap}(1,3)$ $\dots$ $\text{swap}(1,N)$
 $\text{swap}(2,3)$ $\dots$ $\text{swap}(2,N)$
 $\vdots$
 $\text{swap}(N-1,N)$

k-th copy

Make $k = \Theta(N \log N)$ copies of each swap.

Total #swaps: $k \binom{N}{2}$

Random walk reduction

How do we compute $P(\text{agreement})$?

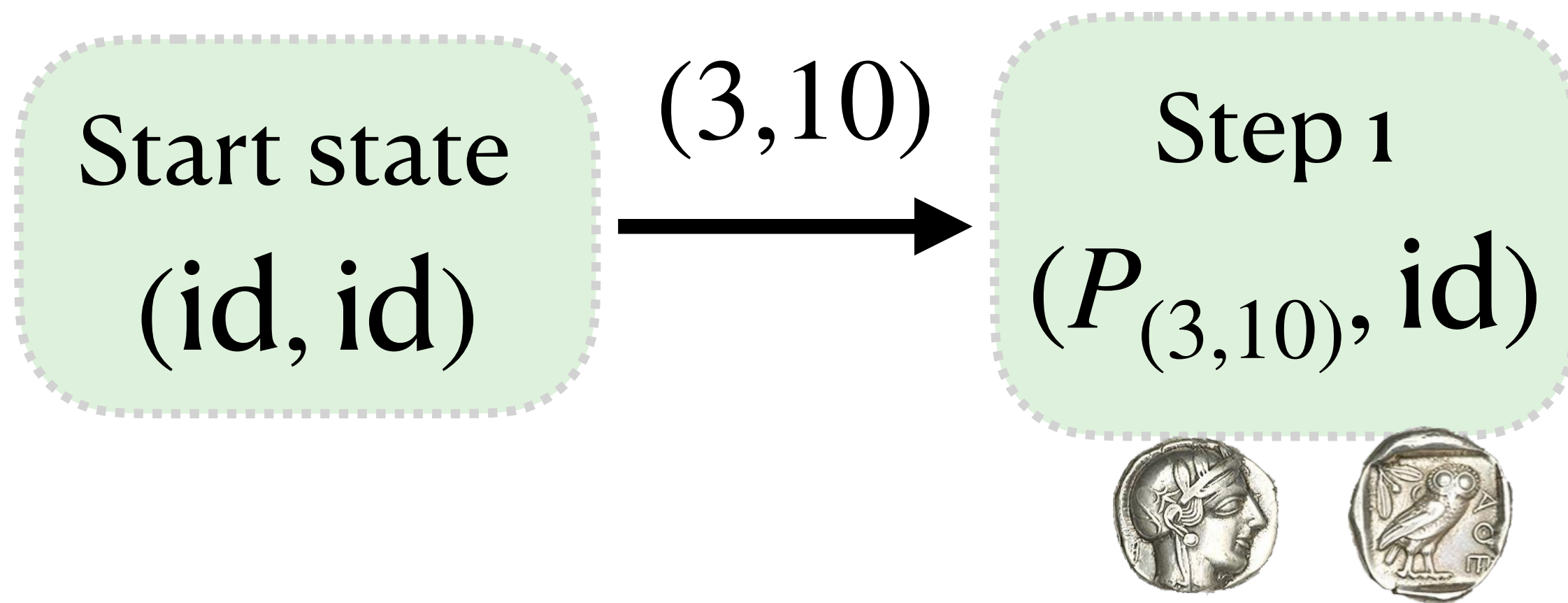
Random walk reduction

How do we compute $P(\text{agreement})$?

Start state
(id, id)

Random walk reduction

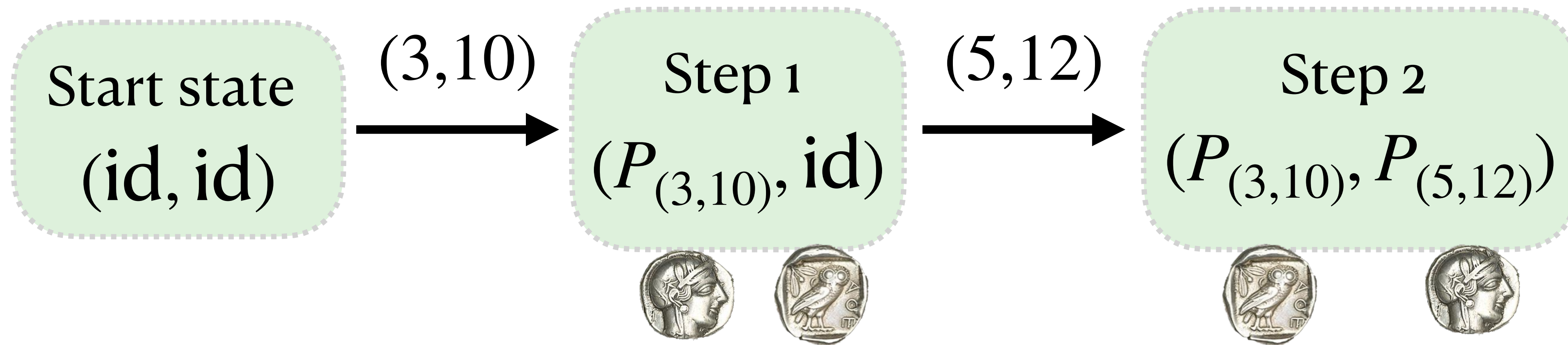
How do we compute $P(\text{agreement})$?



$P_{(3,10)}$: swap of 3 and 10

Random walk reduction

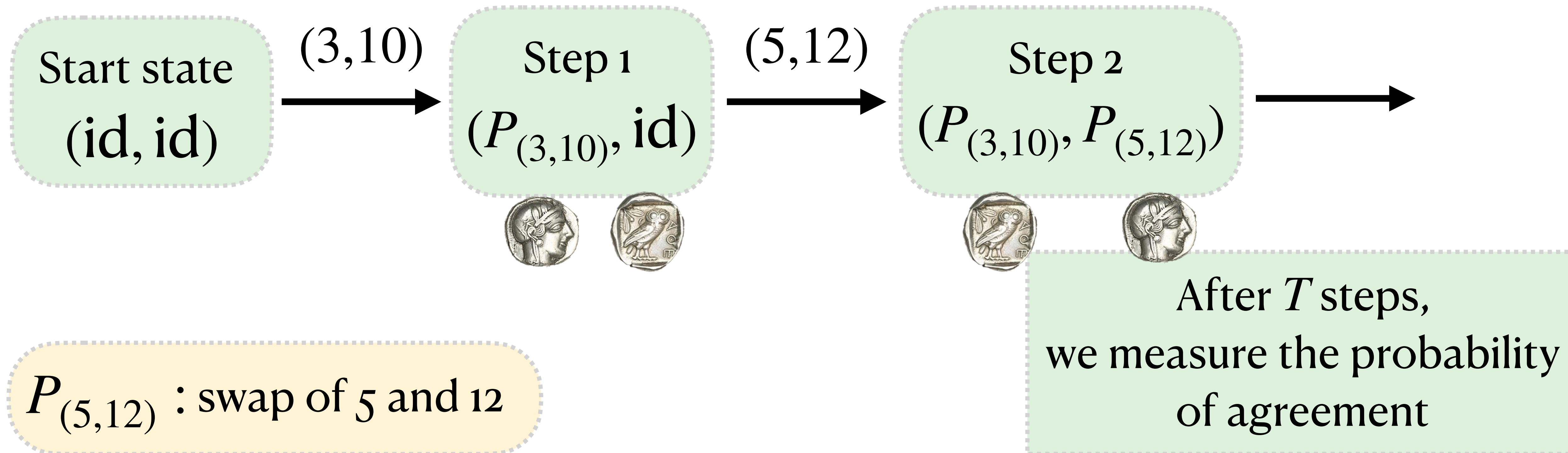
How do we compute $P(\text{agreement})$?



$P_{(5,12)}$: swap of 5 and 12

Random walk reduction

How do we compute $P(\text{agreement})$?



Random walk reduction

How do we compute $P(\text{agreement})$?

$$= \sum_{(P_T, P'_T)} \mathbb{P}(\text{same state} \mid \text{Step T } (P_T, P'_T)) \times \mathbb{P}(\text{Step T } (P_T, P'_T))$$

Random walk reduction

How do we compute $P(\text{agreement})$?

$$= \sum_{(P_T, P'_T)} \mathbb{P}(\text{same state} \mid \text{Step T } (P_T, P'_T)) \times \mathbb{P}(\text{Step T } (P_T, P'_T))$$

$$\mathbb{P}(\text{same state} \mid \text{Step T } (P_T, P'_T)) = \frac{1}{N} + \mathbb{E}(P_T, P'_T)$$

Random walk reduction

How do we compute $P(\text{agreement})$?

$$= \frac{1}{N} + \sum_{(P_T, P'_T)} \mathbb{E}(P_T, P'_T) \times \mathbb{P}(\text{Step T } (P_T, P'_T))$$

Random walk reduction

How do we compute $P(\text{agreement})$?

$$\dots = \frac{1}{N} + \langle \mathbb{E}, \mathbb{P} \rangle$$

From probability to Fourier coefficients

How do we compute $P(\text{agreement})$?

$$\frac{1}{N} + \langle E, P \rangle = \frac{1}{N} + \langle \hat{E}, \hat{P} \rangle$$

Plancherel
formula

Inner product of
Fourier coefficients

A single Fourier coefficient survives

How do we compute $P(\text{agreement})$?

$$\frac{1}{N} + \langle \hat{\mathbb{E}}, \hat{\mathbb{P}} \rangle = \frac{1}{N} + \hat{\mathbb{E}}(\text{std}) \hat{\mathbb{P}}(\text{std})$$

Great Orthogonality Theorem

Out of all Fourier coefficients, only the one that corresponds to the standard representation of S_N remains.

Bound on the Fourier coefficient

How do we compute $P(\text{agreement})$?

$$\frac{1}{N} + \hat{\mathbb{E}}(\text{std}) \hat{\mathbb{P}}(\text{std}) = \frac{1}{N} \pm O\left(\frac{1}{N!}\right)$$

Properties of the standard (std)
representation of the S_N

and

$$T = \Theta(N^2 \log N)$$

Uncorrelated machines imply hardness

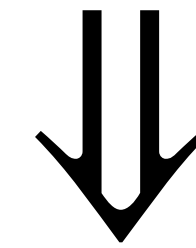
Uncorrelated machines imply hardness

Question: how many shuffling machines can the learner eliminate at most with each query?

Uncorrelated machines imply hardness

We can eliminate at most:

$$\frac{N-1}{\tau^2 - \left(\mathbb{P}(\text{agreement}) - \frac{1}{N} \right) N}$$



$$\mathbb{P}(\text{agreement}) - \frac{1}{N} = \mathcal{O} \left(\frac{1}{N!} \right)$$

$$\Theta \left(\frac{N}{\tau^2} \right)$$

Polynomially many, or we need very low tolerance τ .

Certifying correctness is hard (under overparametrization)

 A. Back de Luca, K. Fountoulakis. . ***Certification from Examples is Hard for Circuits and Transformers under Minimal Overparametrization.*** [arXiv:2605.22964](https://arxiv.org/abs/2605.22964) (May, 2026).

Problem: when do we know we've learned the target function?

Suppose your learner outputs a hypothesis $h : \{0,1\}^n \rightarrow \{0,1\}$ from a hypothesis class

How many labelled examples do I need to certify that it matches the target as n grows?

1. Learn a hypothesis
from data



2. Obtain a labelled
set S



3. Can S certify that
 $h = f^*$ everywhere?

A certificate eliminates all incorrect hypothesis

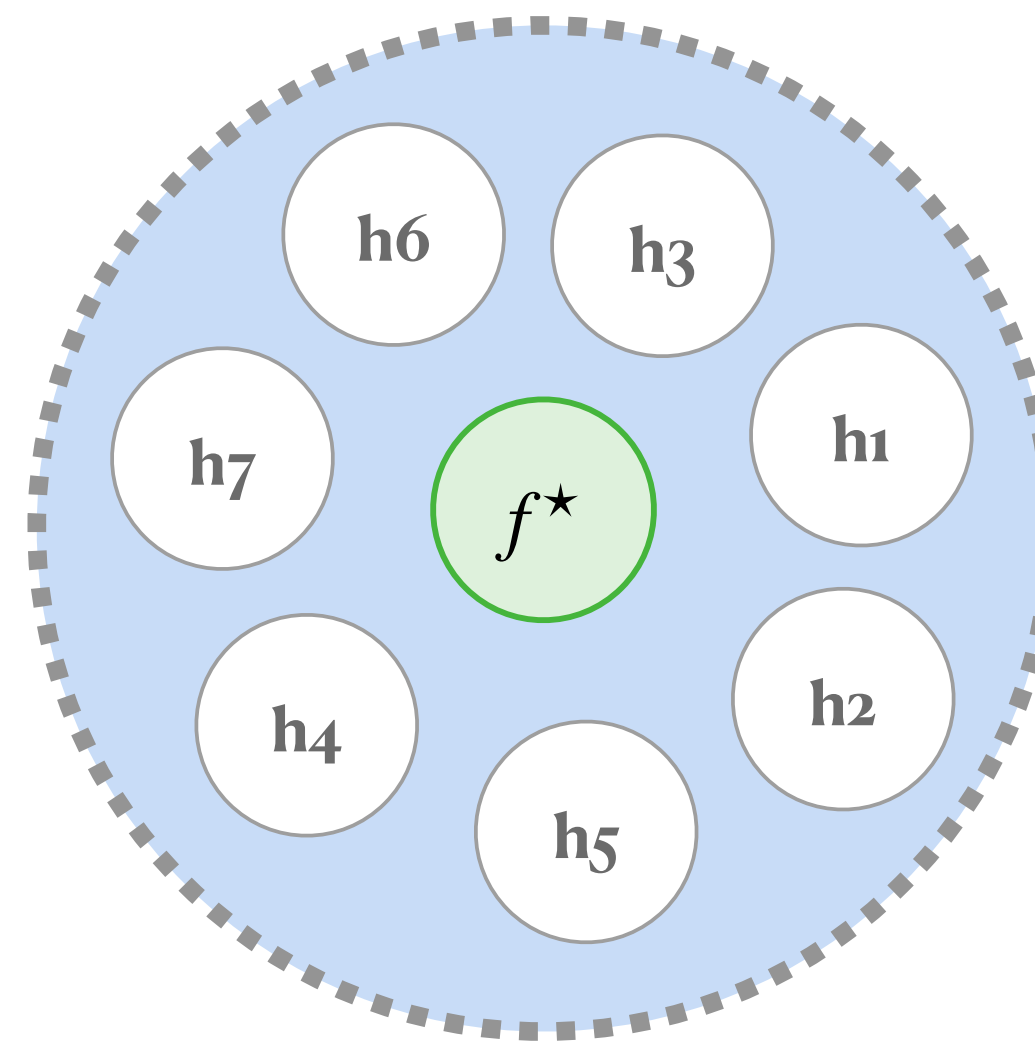
Target: $f^\star$ in hypothesis class $\mathcal{H}$

Certificate candidate: labeled set S

Version space

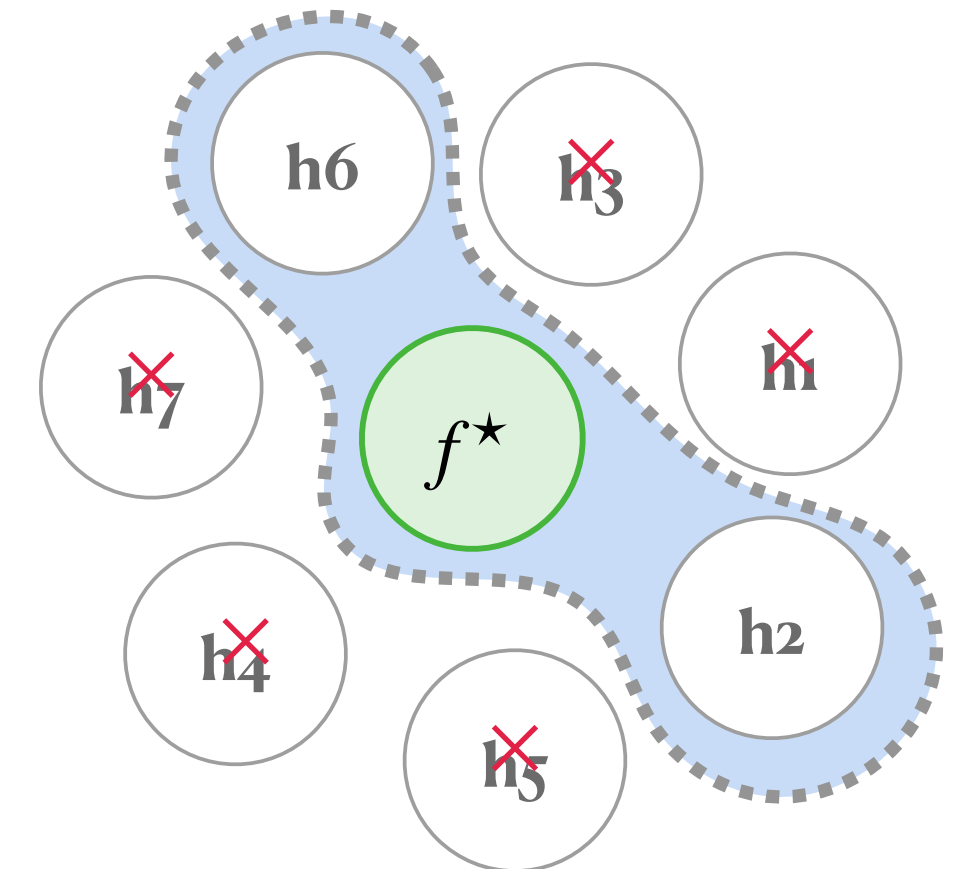
$$VS(S) = \{h \in \mathcal{H} : h \text{ agrees with } f^\star \text{ on } S\}$$

Hypothesis class $\mathcal{H}$



Candidate S 

Version space $VS(S)$

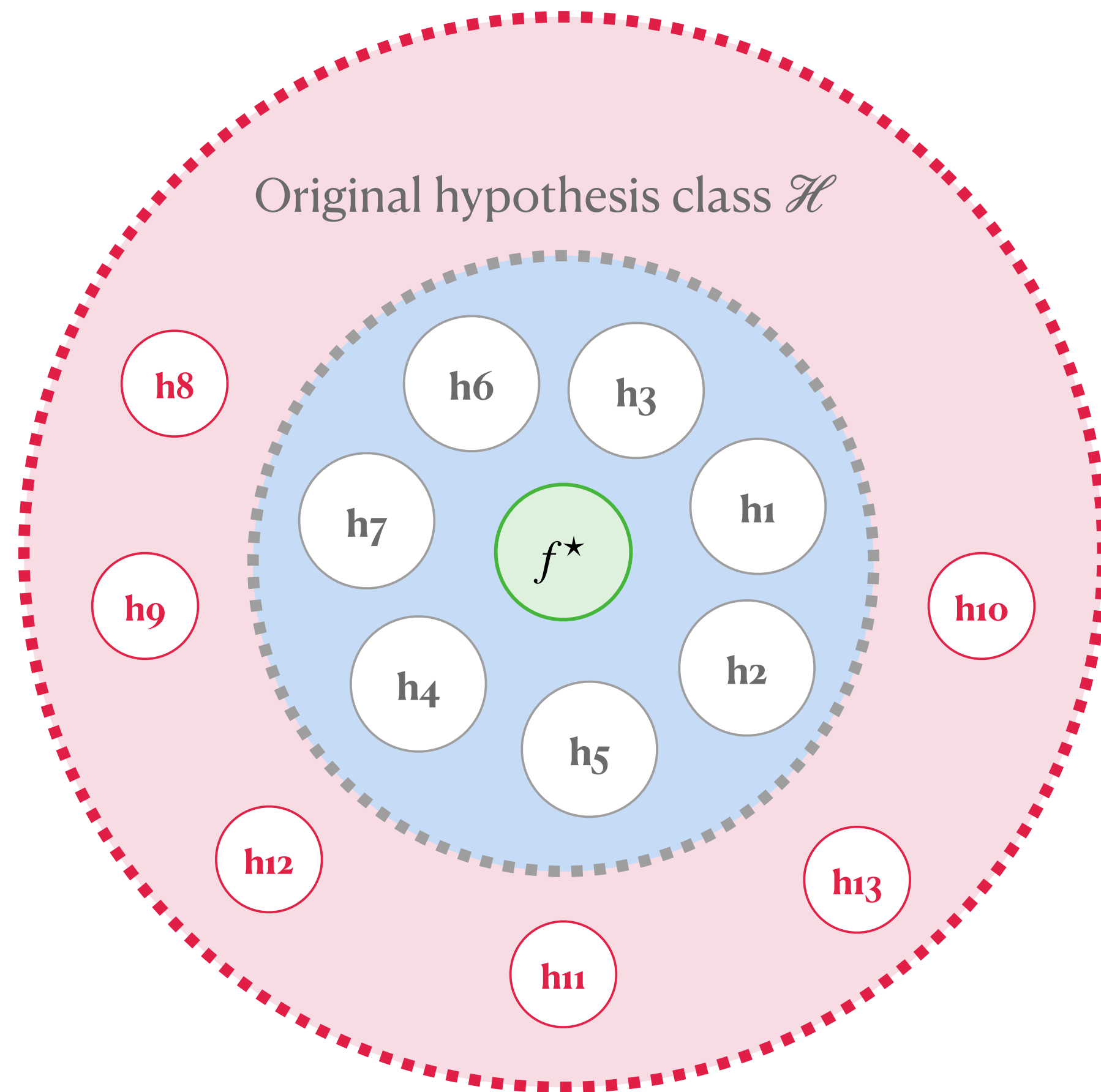


S is a certificate for $f^\star$ iff every surviving hypothesis has the same input-output behavior as $f^\star$

$$VS(S) = \{f^\star\}$$

Certification depends on the hypothesis classes

Enlarged class $\mathcal{H}^+$ (slightly overparameterized)



The target has not changed.
Only its surrounding hypothesis class.

A tiny capacity increase can create exponentially many
new look-alike hypotheses.

How does it affect the certificate size?

It becomes exponential in n for all $f^* \in \mathcal{H}$ (Thm. 4.1/2)

Proof idea: (1) partition the domain

Fix a set of input coords $I \in [n]$

Each binary pattern π on I defines a *trigger block* B_π and together they partition the domain.

Trivial case: $n = 3$ and $I = \{1,2,3\} \rightarrow$ one example for each B_π

B_{000}
 $x = 000$

B_{001}
 $x = 001$

B_{010}
 $x = 010$

B_{011}
 $x = 011$

B_{100}
 $x = 100$

B_{101}
 $x = 101$

B_{110}
 $x = 110$

B_{111}
 $x = 111$

Proof idea: (2) build a deceiver per trigger block

Fix a set of input coords $I \in [n]$

Each binary pattern π on I defines a *trigger block* $B_\pi := \{x \in \{0,1\}^n : x_I = \pi\}$ and together they partition the domain.

Trivial case: $n = 3$ and $I = \{1,2,3\} \rightarrow$ one example for each B_π

B_{000}
 $x = 000$

B_{001}
 $x = 001$

B_{010}
 $x = 010$

B_{011}
 $x = 011$

B_{100}
 $x = 100$

B_{101}
 $x = 101$

B_{110}
 $x = 110$

B_{111}
 $x = 111$

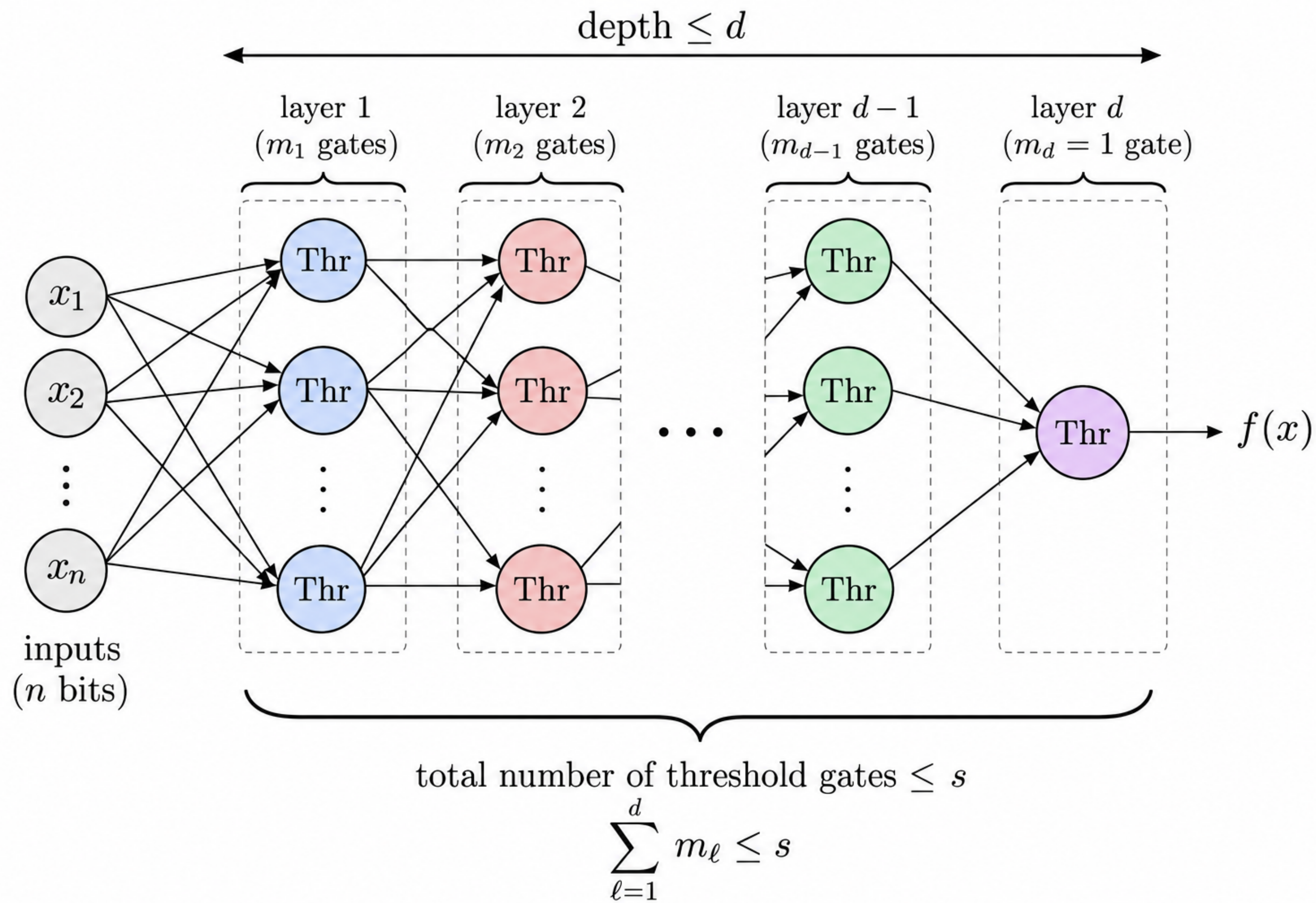
Build a *deceiver* g_π per trigger block B_π

$$g_\pi(x) = \begin{cases} \neg f^\star(x) & \text{if } x \in B_\pi \\ f^\star(x) & \text{otherwise} \end{cases}$$

To eliminate each deceiver, the certificate needs one example from its trigger block B_π

There are exponentially many trigger blocks, so certificate size is $\exp(n) \rightarrow$ Prop. 4.1

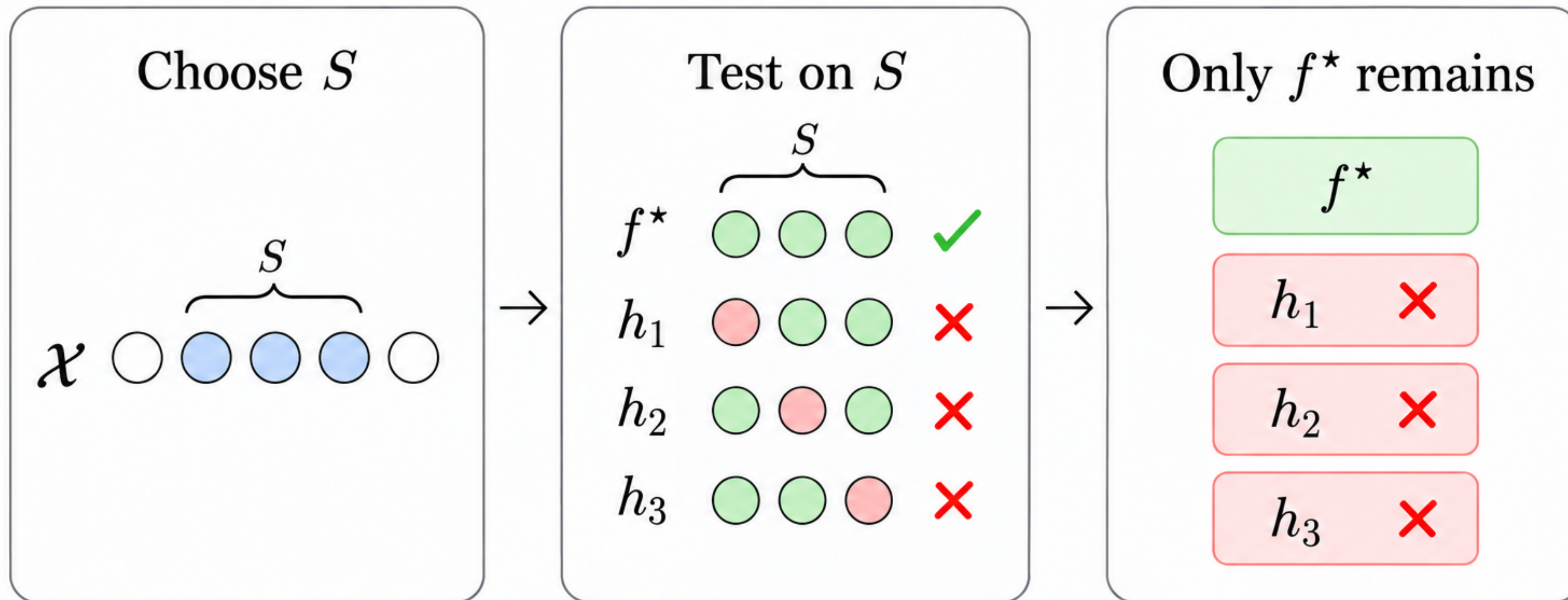
Some definitions



$$\text{THR}_{w,\theta}(z) := \begin{cases} 1, & \sum_{j=1}^m w_j z_j \geq \theta, \\ 0, & \sum_{j=1}^m w_j z_j < \theta. \end{cases}$$

Some definitions

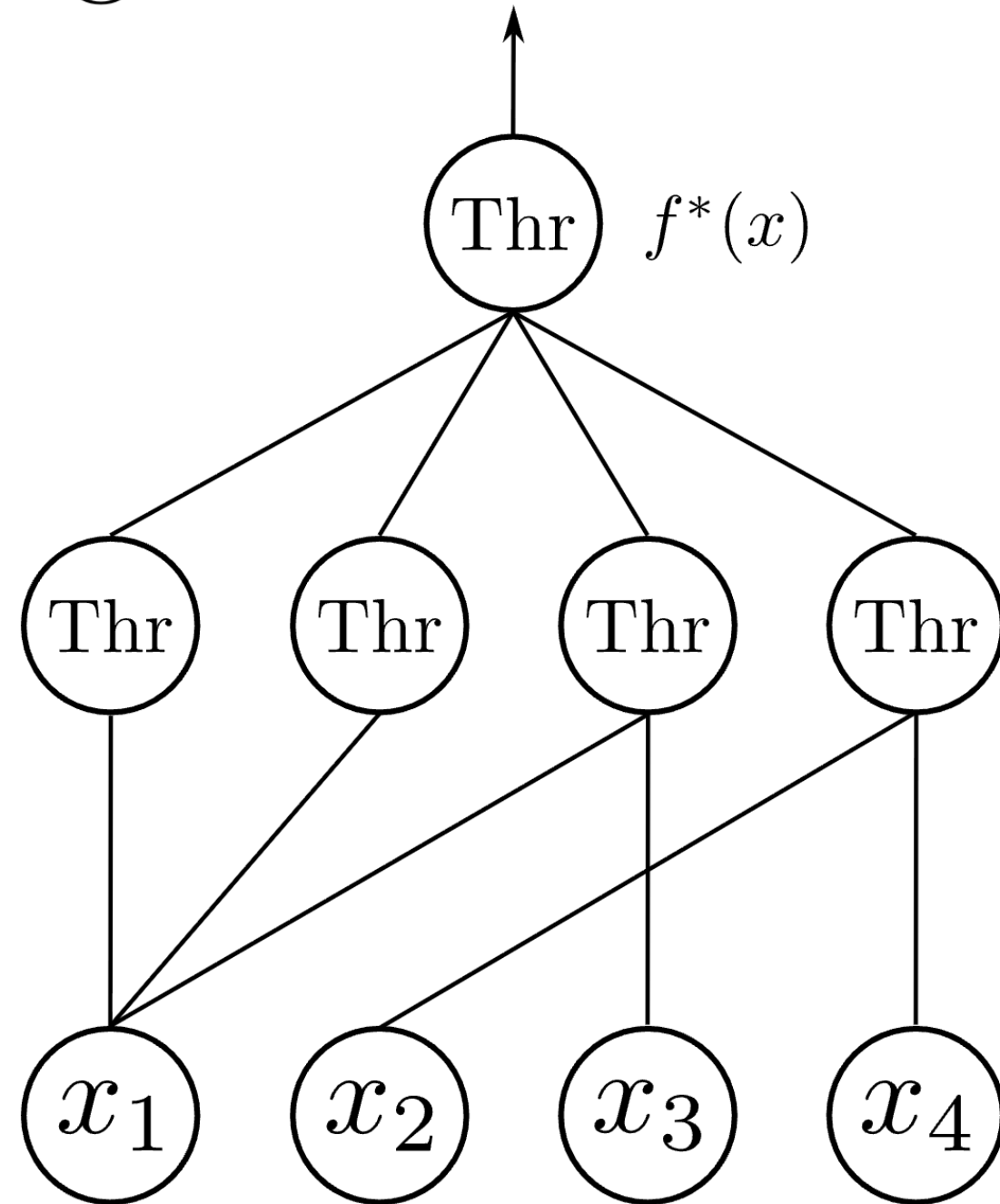
$$\text{cert}(f^*, H) := \min_{S \subseteq \mathcal{X}} \left\{ |S| : \forall h \in H, (\forall x \in S, h(x) = f^*(x)) \Rightarrow h = f^* \right\}$$



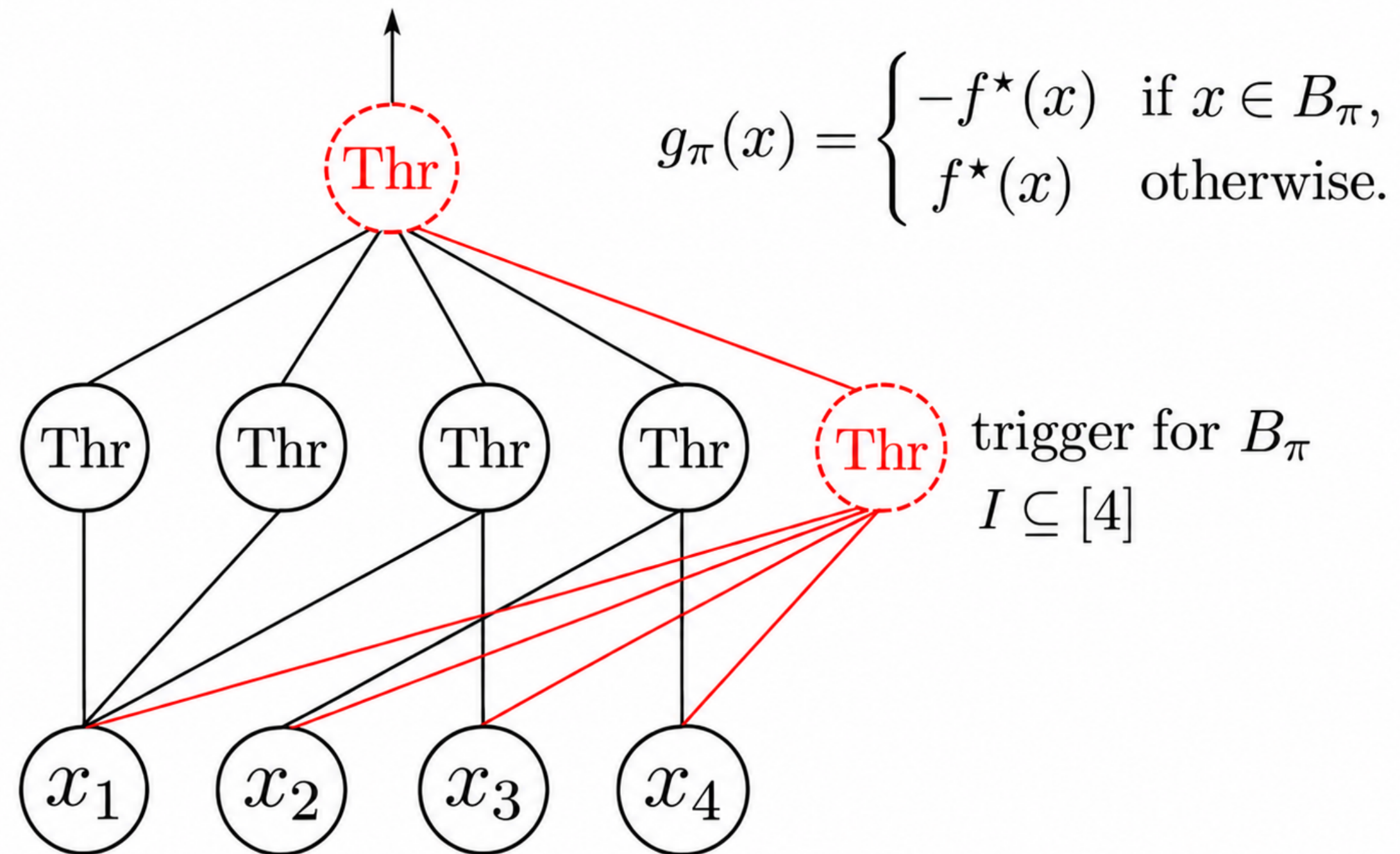
$\text{cert}(f^*, H) = \text{smallest } |S| \text{ that eliminates every } h \neq f^*.$

Threshold circuits: one extra gate is enough (Thm 4.1)

Original



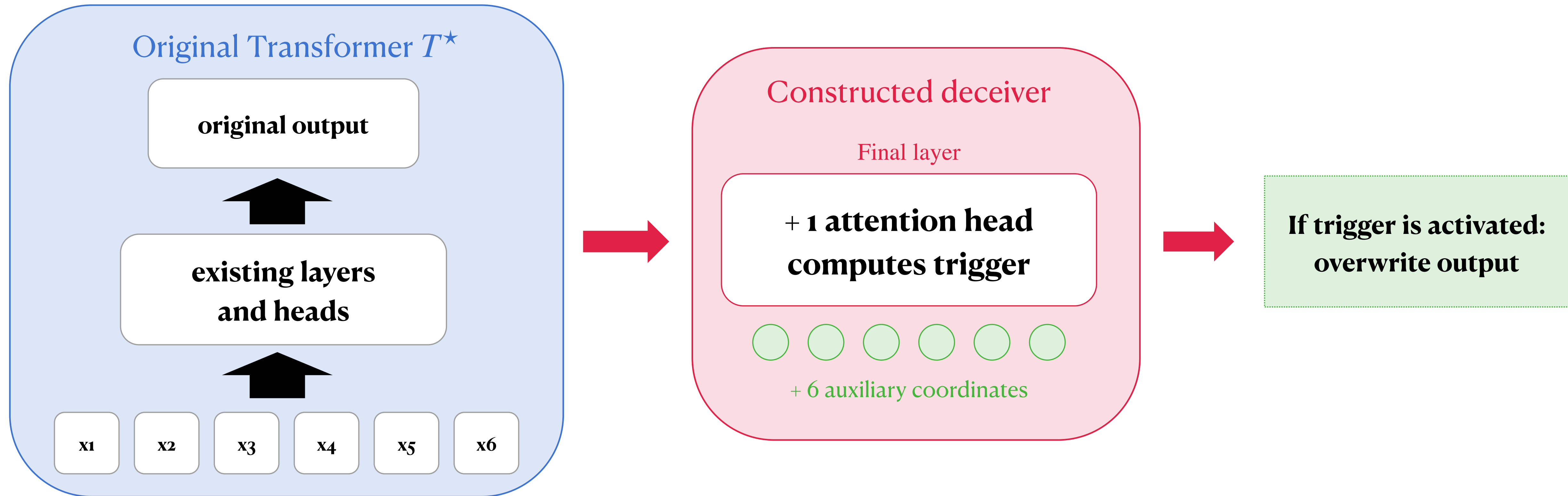
Deceiver



Adding just one more gate (same depth d) $\rightarrow \text{cert}(f^*, TC_{d,s(n)+1}^0) \geq 2^{\Omega(n)}$

Holds for every target in the smaller class, including f^* s.t. $\text{cert}(f^*, TC_{d,s(n)}^0) \leq n^{O(1)}$

The same principle applies for Transformers (Thm 4.2)



Same exponential barrier exists! (Thm 4.2)

What if we allow some mistakes?

Relative-error guarantee

Allow a constant fraction
 $\varepsilon \in [0,1]$ of the full domain

Can hide exponentially many absolute mistakes!

Near-perfect absolute guarantee

Allow at most
 $R = \text{poly}(n)$ mistakes

Still needs $2^{\Omega(n)}$ examples

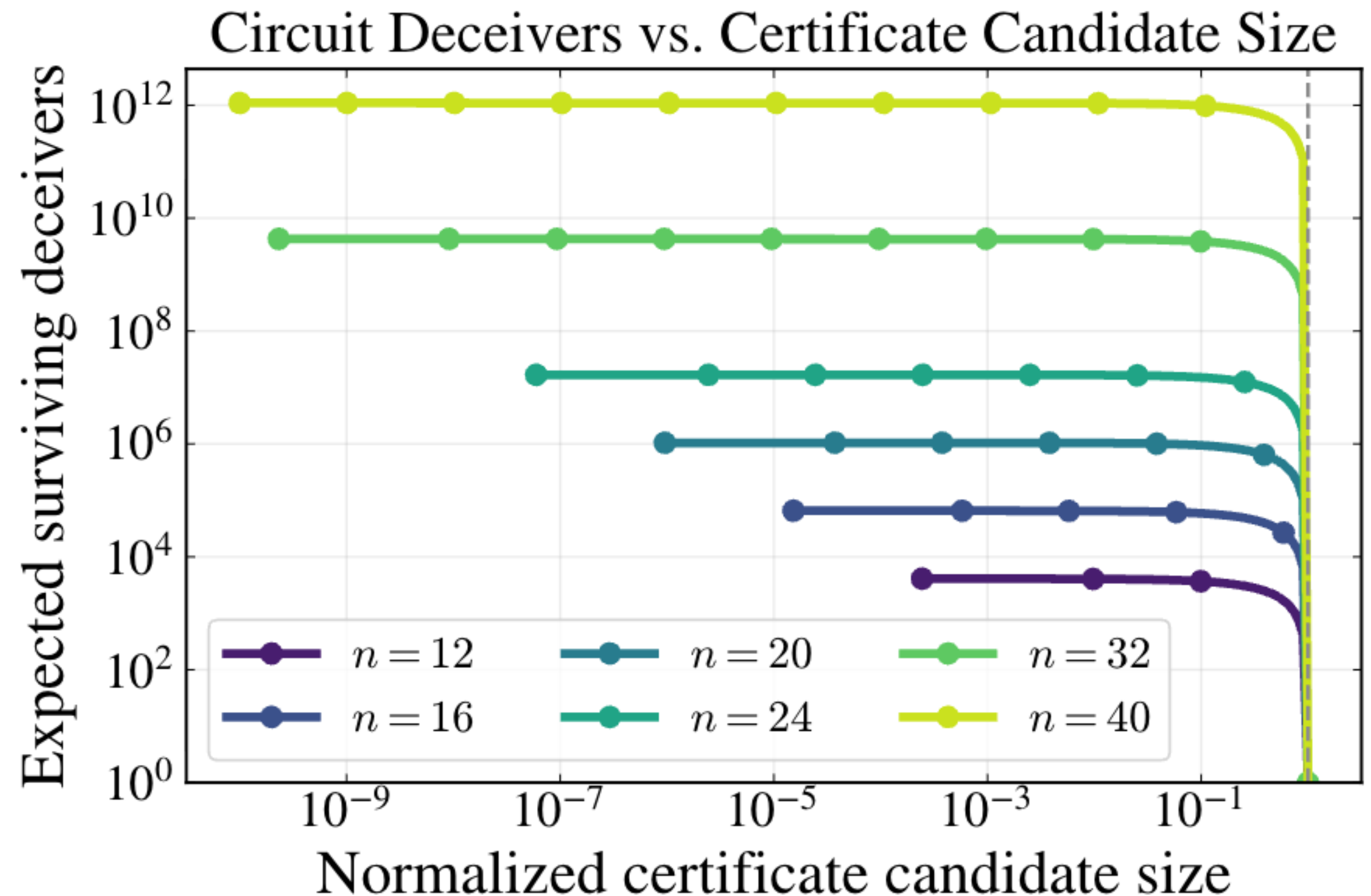
The domain has 2^n inputs. A constant fraction of 2^n is already $\exp(n)$, while $\text{poly}(n)$ mistakes is a much stronger guarantee.

Experiment: Poly examples barely eliminates deceiver circuits

For n -bit addition recognition:

- Target circuit uses 3 thr gates
- One trigger deceiver for each binary pattern (2^n deceivers)

When candidate size is $\text{poly}(n)$, almost all constructed deceivers survive.



Certificate candidate size is normalized by 2^n

Experiment: certifying trained Transformers

Task: recognizing 10-bit addition

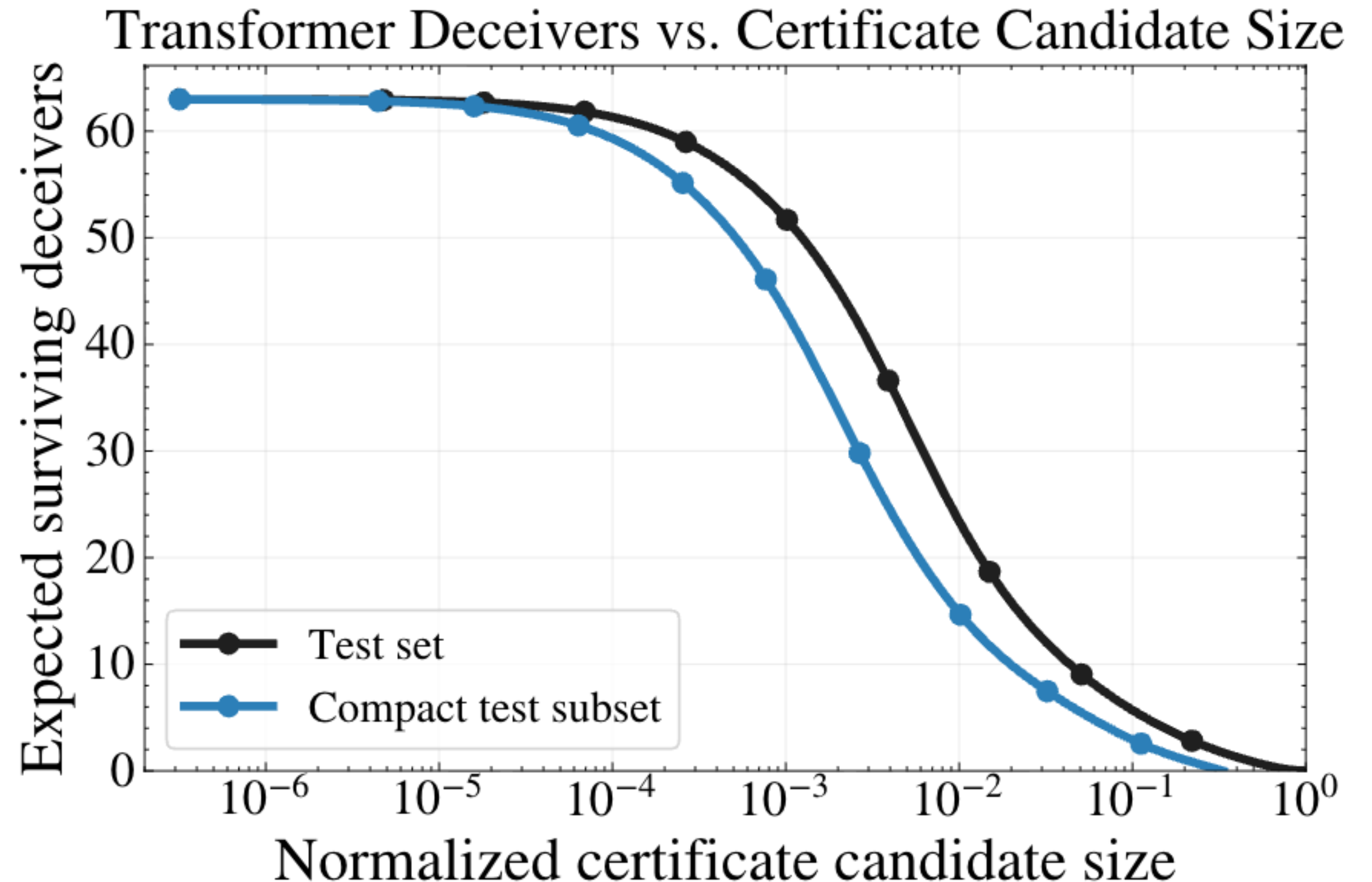
150 models trained w. $\geq 99.9\%$ train/val acc.

87 make no errors on the held-out set

63 make at least one mistake

Certificate candidates: uniformly sampled subsets of the held-out set.

Compact test subset: 1/3 of held-out data, set to cover 77.2% of observed error locations.



Ruling out all incorrect hypotheses still requires large certificates!

Thank you!