Coded and Uncoded Trace Reconstruction

João Ribeiro
Imperial College London

Deletion channel

0 1 0 0 1 1

Deletion channel

0 1 0 0 1 1

0 1 0 0 1 1

Deletion channel

0 1 0 0 1 1

↓

0 1 0 0 1 1

↓

0 1 1

Deletion channel

0 1 0 0 1 1

0 1 0 0 1 1

0 1 1

Other deletion patterns that lead to same output

0 1 0 0 1 1

0 1 0 0 1 1

0 1 0 0 1 1

Deletion channel

0 1 0 0 1 1

0 1 0 0 1 1

0 1 1

Other deletion patterns that lead to same output

0 1 0 0 1 1

0 1 0 0 1 1

0 1 0 0 1 1

i.i.d. deletions: each bit deleted independently with fixed probability d

Deletion channel

0 1 0 0 1 1

↓

0 1 0 0 1 1

↓

0 1 1

Other deletion patterns that lead to same output

0 1 0 0 1 1

0 1 0 0 1 1

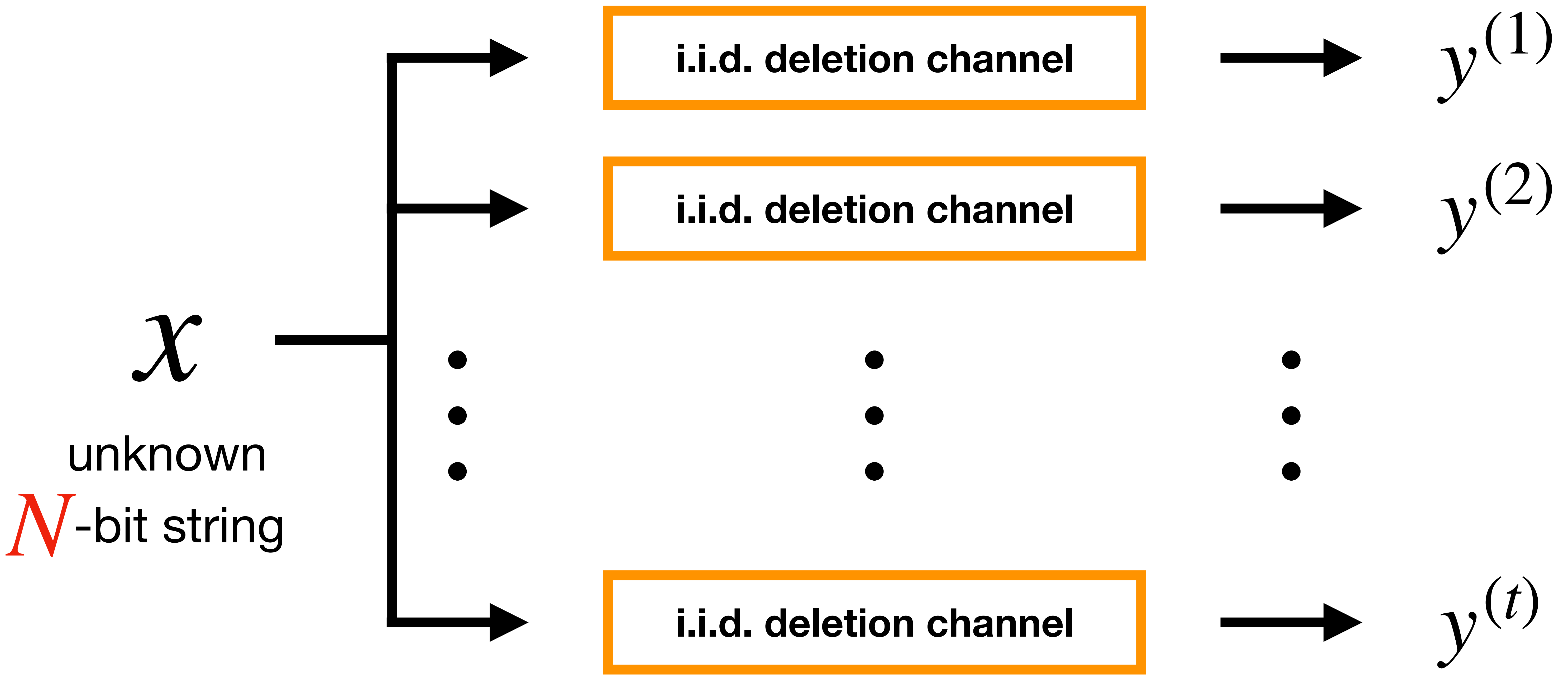
0 1 0 0 1 1

i.i.d. deletions: each bit deleted independently with fixed probability d

Many open problems! E.g., capacity is still unknown…

Trace reconstruction

[Levenshtein01, BatuKannanKhannaMcGregor04]



Goal

Reconstruct x from **traces** $y^{(1)}, \ldots, y^{(t)}$ such that:

- We succeed with high probability $1 - O(1/N)$
- We use as few traces as possible

Original motivation for trace reconstruction

[BatuKannanKhannaMcGregor04]

Ancestor DNA

mutations
(deletions, insertions, substitutions)

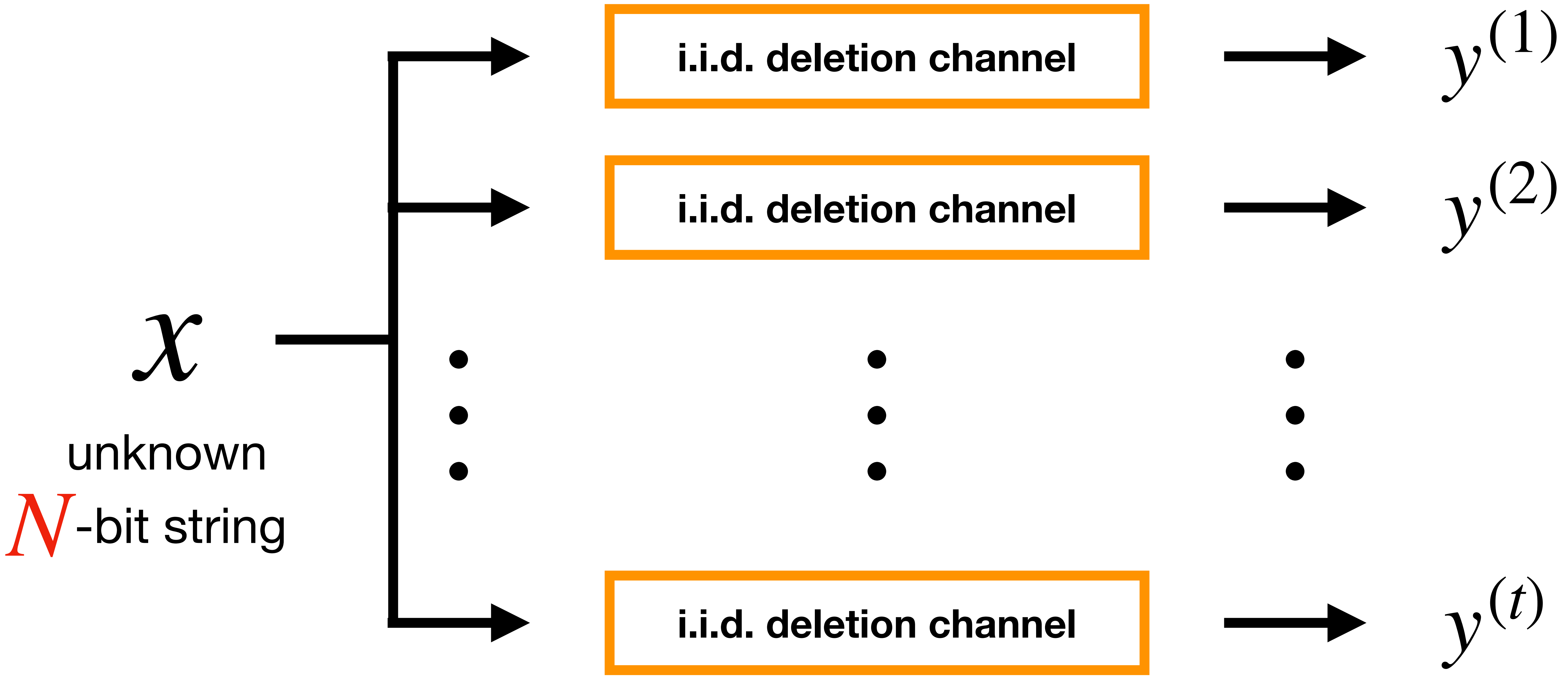
Descendent 1 DNA

Descendent 2 DNA

Descendent 3 DNA

Multiple sequence alignment: Deduce common ancestor DNA from descendants DNA.

Main settings for uncoded trace reconstruction



Worst-case trace reconstruction

*Reconstruction algorithm $\mathscr{R}$ must succeed with high probability simultaneously for **all input strings**.*

$$\forall x : \Pr_{y^{(1)},\ldots,y^{(t)} \leftarrow \mathsf{Del}(x)} [\mathscr{R}(y^{(1)}, \ldots, y^{(t)}) = x] \approx 1$$

Average-case trace reconstruction

***Average** error probability of reconstruction algorithm $\mathscr{R}$ over all input strings is small.*

$$\Pr_{x \leftarrow \{0,1\}^N, y^{(1)},\ldots,y^{(t)} \leftarrow \mathsf{Del}(x)} [\mathscr{R}(y^{(1)}, \ldots, y^{(t)}) = x] \approx 1$$

Worst-case trace reconstruction

Recall: Reconstruction algorithm must succeed for **all strings** with high probability.

Reconstruction algorithms	#traces	deletion probability
Batu, Kannan, Khanna, McGregor 2004	$N \log N$	$d = 1/N^{1/2+\epsilon}$
Holenstein, Mitzenmacher, Panigrahy, Wieder 2008	$\exp(N^{1/2})$	any constant < 1
De, O'Donnell, Servedio 2017 Nazarov, Peres 2017	$\exp(N^{1/3})$	any constant < 1

Mean-based algorithms & worst-case trace reconstruction

$x \longrightarrow$ i.i.d. deletions $\longrightarrow Y$ (padded with 0's)

$\in \{-1,1\}^N$ $\qquad\qquad$ $\in \{-1,0,1\}^N$

Mean-based algorithms & worst-case trace reconstruction

$$x \xrightarrow{\quad} \boxed{\text{i.i.d. deletions}} \xrightarrow{\quad} Y$$

$\in \{-1,1\}^N$ $\qquad\qquad$ $\in \{-1,0,1\}^N$ $\qquad$ (padded with 0's)

Mean trace: $\mu(x) = (\mathrm{E}[Y_1], \mathrm{E}[Y_2], \ldots, \mathrm{E}[Y_N])$ (real-linear function of x)

Mean-based algorithms & worst-case trace reconstruction

x ⟶ i.i.d. deletions ⟶ Y **(padded with 0's)**

$\in \{-1,1\}^N$ $\qquad$ $\in \{-1,0,1\}^N$

Mean trace: $\mu(x) = (\mathrm{E}[Y_1], \mathrm{E}[Y_2], \ldots, \mathrm{E}[Y_N])$ **(real-linear function of x)**

Mean-based algorithm:

- Compute estimate $\hat{\mu}$ of mean trace $\mu(x)$ from T traces;
- Find x such that $\mu(x)$ is closest to $\hat{\mu}$.

[DeO'DonnellServedio17, NazarovPeres17]:
There exist mean-based algorithms using $T = \exp(N^{1/3})$ traces. Moreover, **this is optimal for mean-based algorithms.**

Mean-based algorithms & complex analysis

Mean trace: $\mu(x) = (\mathrm{E}[Y_1], \mathrm{E}[Y_2], \ldots, \mathrm{E}[Y_N])$ **(real-linear function of x)**

#traces required for accurate estimate $\hat{\mu}$ is dictated by

$$\min_{x \neq x'} ||\mu(x) - \mu(x')||_1 = 2 \times \min_{x \in \{-1,0,1\}^N} ||\mu(x)||_1$$

Mean-based algorithms & complex analysis

Mean trace: $\mu(x) = (\mathrm{E}[Y_1], \mathrm{E}[Y_2], \ldots, \mathrm{E}[Y_N])$ **(real-linear function of x)**

#traces required for accurate estimate $\hat{\mu}$ is dictated by

$$\min_{x \neq x'} ||\mu(x) - \mu(x')||_1 = 2 \times \min_{x \in \{-1,0,1\}^N} ||\mu(x)||_1$$

Deletion channel polynomial $P_{d,x}(w) = \sum_{j=1}^{N} \mu(x)_j \cdot w^j, \quad w \in \mathbb{C}$

Mean-based algorithms & complex analysis

Mean trace: $\mu(x) = (\mathrm{E}[Y_1], \mathrm{E}[Y_2], \ldots, \mathrm{E}[Y_N])$ (real-linear function of x)

#traces required for accurate estimate $\hat{\mu}$ is dictated by

$$\min_{x \neq x'} ||\mu(x) - \mu(x')||_1 = 2 \times \min_{x \in \{-1,0,1\}^N} ||\mu(x)||_1$$

$$\boxed{\textbf{Deletion channel polynomial} \quad P_{d,x}(w) = \sum_{j=1}^{N} \mu(x)_j \cdot w^j, \quad w \in \mathbb{C}}$$

Bounding $\min ||\mu(x)||_1$ $\approx$ **Maximizing special polynomial over arc of complex circle**

(can use powerful complex analytic tools!)

Mean-based algorithms & Littlewood polynomials

Deletion channel polynomial $$P_{d,x}(w) = \sum_{j=1}^{N} \mu(x)_j \cdot w^j, \quad w \in \mathbb{C}$$

$$P_{d,x}(w) = (1-d)\sum_{j=1}^{N} x_j \cdot z^j \qquad z = d + (1-d)w$$

Easy to write
in terms of x

Mean-based algorithms & Littlewood polynomials

Deletion channel polynomial $$P_{d,x}(w) = \sum_{j=1}^{N} \mu(x)_j \cdot w^j, \quad w \in \mathbb{C}$$

Littlewood polynomial $A(z), \quad x_j \in \{-1,0,1\}$

$$P_{d,x}(w) = (1-d)\boxed{\sum_{j=1}^{N} x_j \cdot z^j}$$

$$z = d + (1-d)w$$

Easy to write in terms of x

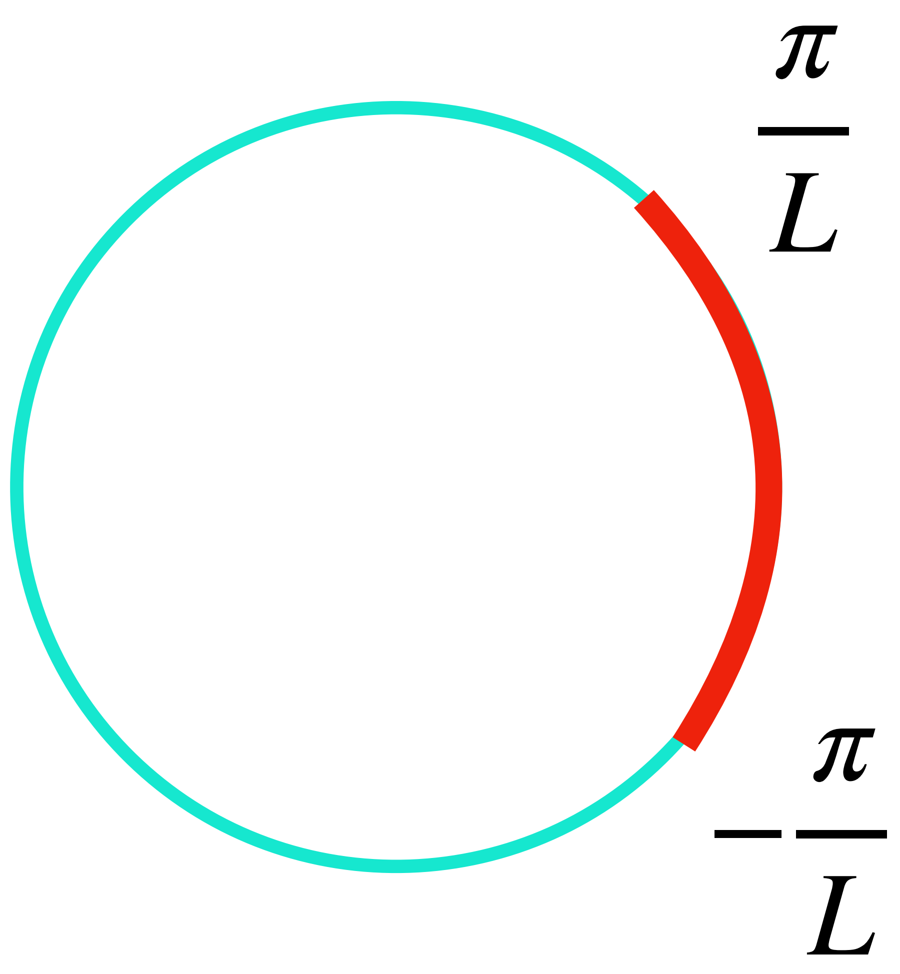
Mean-based algorithms & Littlewood polynomials

Deletion channel polynomial $P_{d,x}(w) = \sum_{j=1}^{N} \mu(x)_j \cdot w^j, \quad w \in \mathbb{C}$

Littlewood polynomial $A(z), \quad x_j \in \{-1,0,1\}$

$$P_{d,x}(w) = (1-d)\boxed{\sum_{j=1}^{N} x_j \cdot z^j} \qquad z = d + (1-d)w$$

Easy to write in terms of x



[BorweinErdélyi97]

$A(z)$ Littlewood polynomial

Mean-based algorithms & Littlewood polynomials

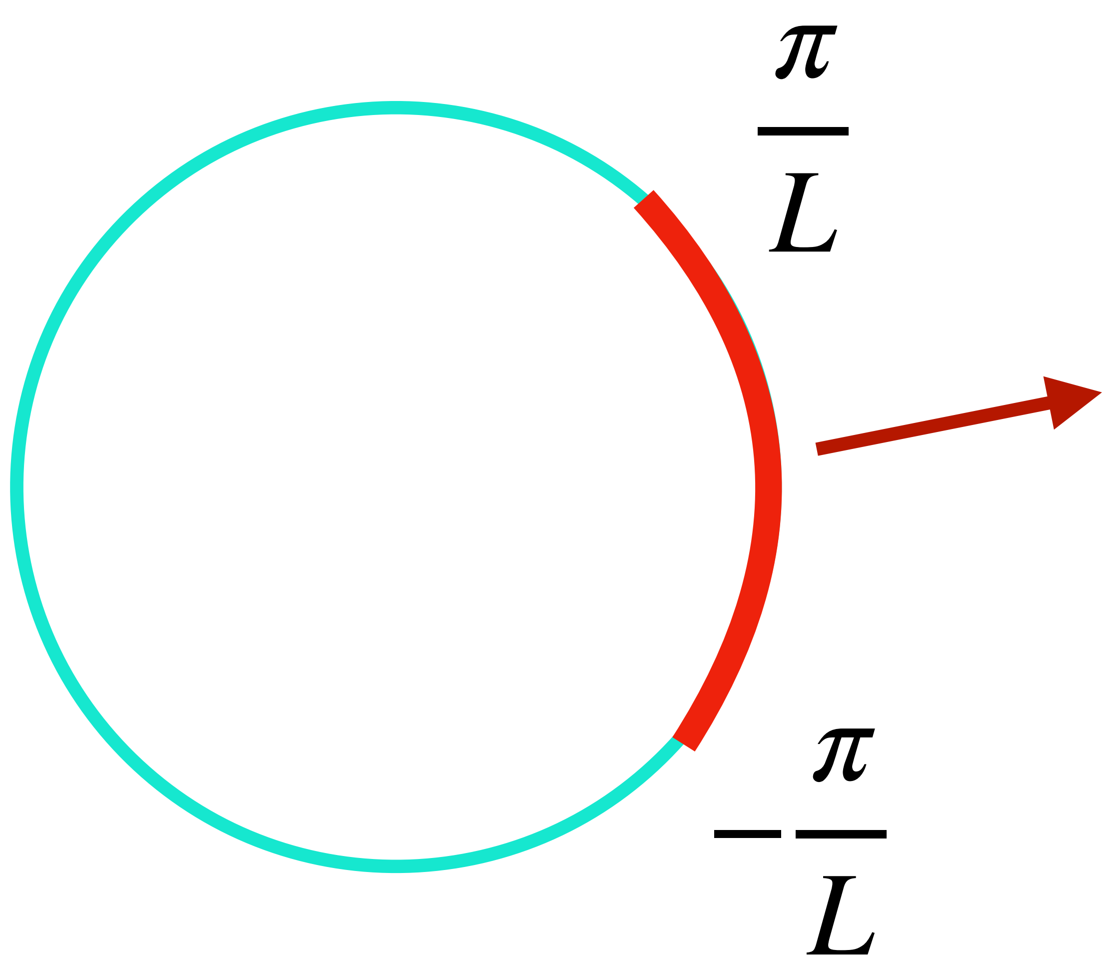
Deletion channel polynomial $$P_{d,x}(w) = \sum_{j=1}^{N} \mu(x)_j \cdot w^j, \quad w \in \mathbb{C}$$

Littlewood polynomial $A(z), \quad x_j \in \{-1,0,1\}$

$$P_{d,x}(w) = (1-d)\boxed{\sum_{j=1}^{N} x_j \cdot z^j}$$

$$z = d + (1-d)w$$

Easy to write in terms of x



[BorweinErdélyi97]

$A(z)$ Littlewood polynomial

$$\max |A(e^{i\theta})| \geq e^{-cL}$$

Mean-based algorithms & Littlewood polynomials

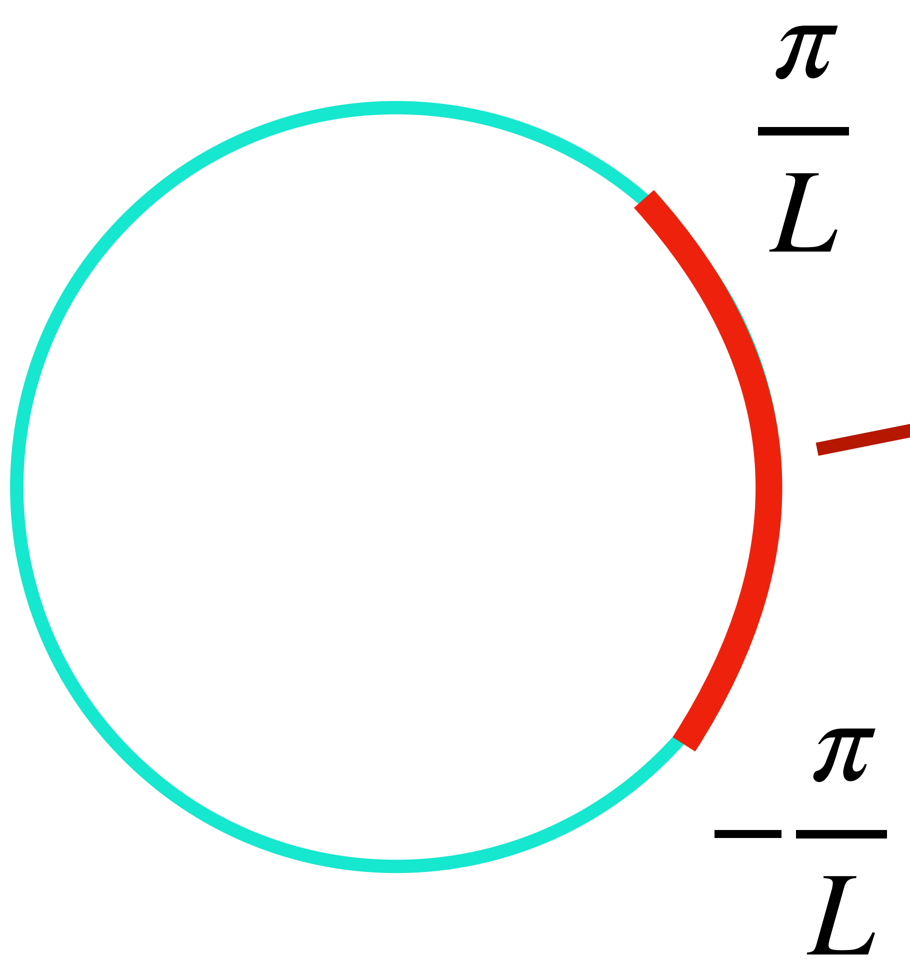
Deletion channel polynomial $P_{d,x}(w) = \sum_{j=1}^{N} \mu(x)_j \cdot w^j, \quad w \in \mathbb{C}$

Littlewood polynomial $A(z), \quad x_j \in \{-1,0,1\}$

$$P_{d,x}(w) = (1-d)\boxed{\sum_{j=1}^{N} x_j \cdot z^j} \qquad z = d + (1-d)w$$

Easy to write in terms of x

w **such that** $z = e^{i\theta}, \quad \theta \in [-\pi/L, \pi/L]$



[BorweinErdélyi97]

$A(z)$ Littlewood polynomial

$\max |A(e^{i\theta})| \geq e^{-cL}$

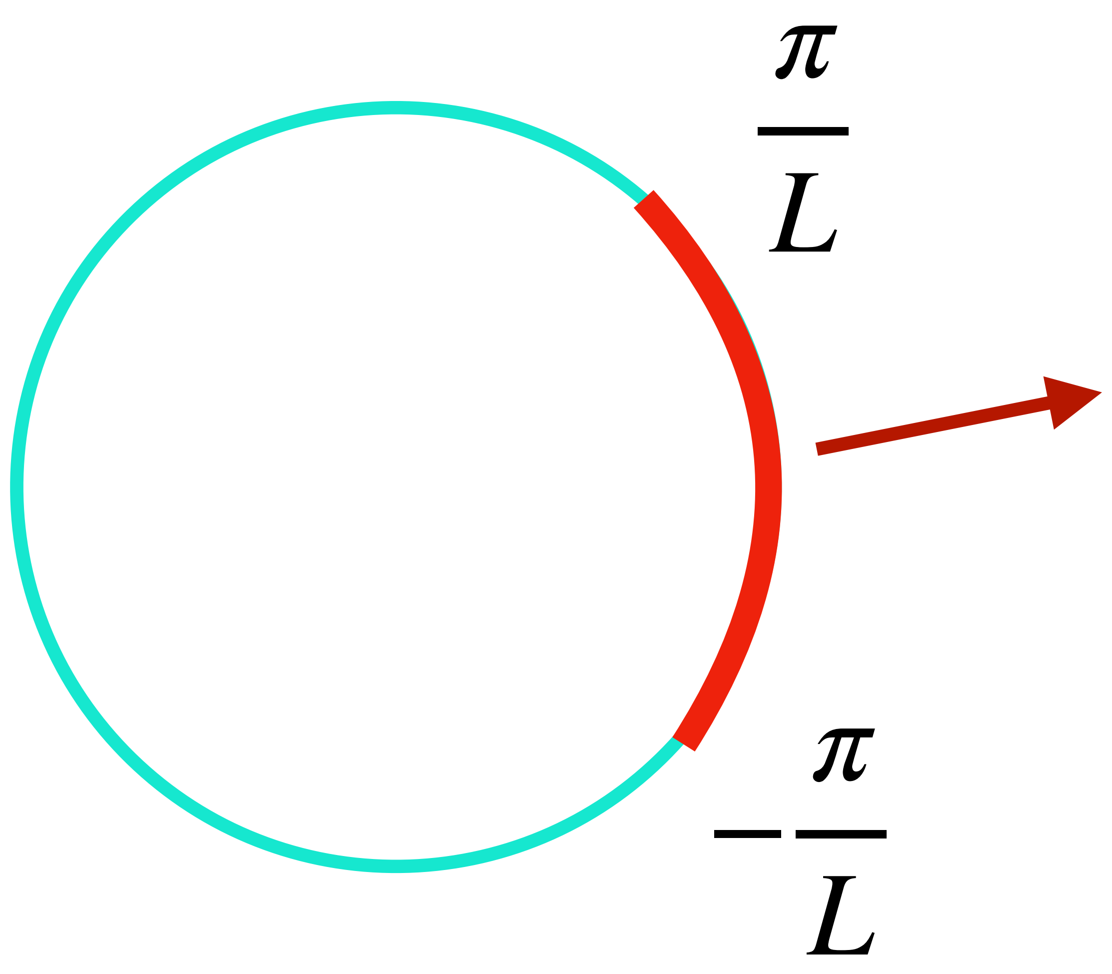
Mean-based algorithms & Littlewood polynomials

Deletion channel polynomial $P_{d,x}(w) = \sum_{j=1}^{N} \mu(x)_j \cdot w^j, \quad w \in \mathbb{C}$

Littlewood polynomial $A(z), \quad x_j \in \{-1,0,1\}$

$$P_{d,x}(w) = (1-d) \boxed{\sum_{j=1}^{N} x_j \cdot z^j} \qquad z = d + (1-d)w$$

Easy to write in terms of x



[BorweinErdélyi97]

$A(z)$ Littlewood polynomial

$\max |A(e^{i\theta})| \geq e^{-cL}$

w **such that** $z = e^{i\theta}, \quad \theta \in [-\pi/L, \pi/L]$

$$\sum_j |\mu_j(x)| \cdot |w|^j \geq (1-d)\,|A(e^{i\theta})|$$

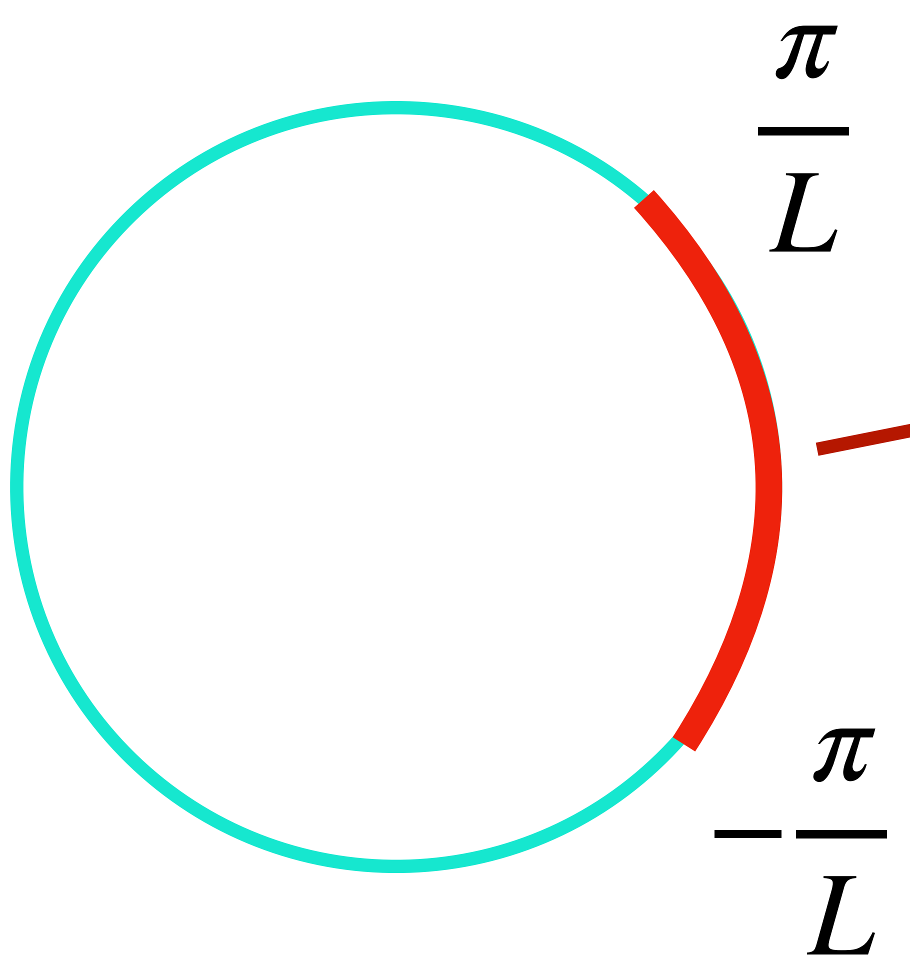
(△-ineq)

Mean-based algorithms & Littlewood polynomials

Deletion channel polynomial $$P_{d,x}(w) = \sum_{j=1}^{N} \mu(x)_j \cdot w^j, \quad w \in \mathbb{C}$$

Littlewood polynomial $A(z), \quad x_j \in \{-1,0,1\}$

$$\boxed{P_{d,x}(w) = (1-d)\boxed{\sum_{j=1}^{N} x_j \cdot z^j} \qquad z = d + (1-d)w \qquad \text{Easy to write in terms of x}}$$



[BorweinErdélyi97]

$A(z)$ Littlewood polynomial

$$\max |A(e^{i\theta})| \geq e^{-cL}$$

w **such that** $z = e^{i\theta}, \quad \theta \in [-\pi/L, \pi/L]$

$$\sum_j |\mu_j(x)| \cdot |w|^j \geq (1-d)\,|A(e^{i\theta})| \qquad \text{(△-ineq)}$$

$$\implies ||\mu(x)||_1 \geq e^{-\frac{c'N}{L^2}} \cdot |A(e^{i\theta})| \qquad \text{(simple trig)}$$

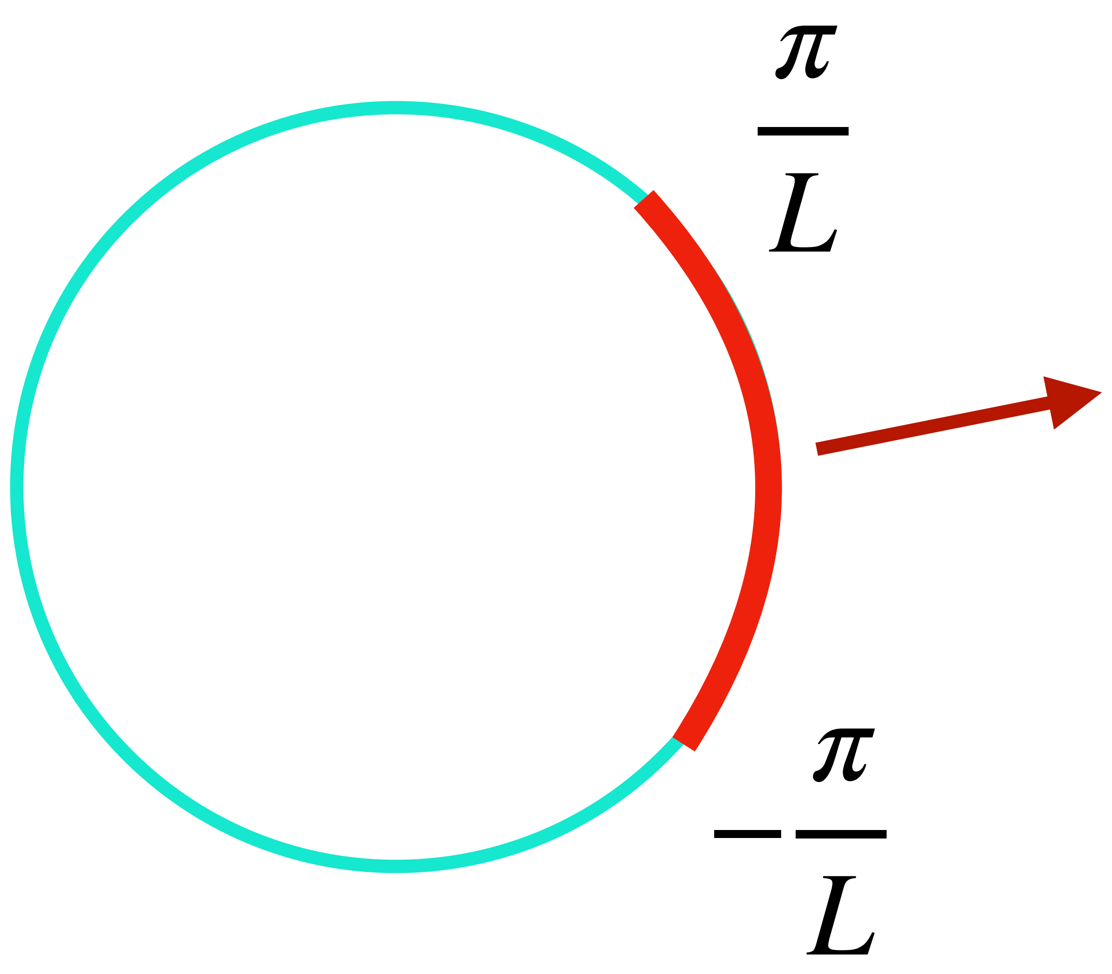
Mean-based algorithms & Littlewood polynomials

Deletion channel polynomial $P_{d,x}(w) = \sum_{j=1}^{N} \mu(x)_j \cdot w^j, \quad w \in \mathbb{C}$

Littlewood polynomial $A(z), \quad x_j \in \{-1,0,1\}$

$$P_{d,x}(w) = (1-d)\boxed{\sum_{j=1}^{N} x_j \cdot z^j} \qquad z = d + (1-d)w$$

Easy to write in terms of x



[BorweinErdélyi97]

$A(z)$ Littlewood polynomial

$$\max |A(e^{i\theta})| \geq e^{-cL}$$

w **such that** $z = e^{i\theta}, \quad \theta \in [-\pi/L, \pi/L]$

$$\sum_j |\mu_j(x)| \cdot |w|^j \geq (1-d)|A(e^{i\theta})| \qquad \textbf{(}\triangle\textbf{-ineq)}$$

$$\implies ||\mu(x)||_1 \geq e^{-\frac{c'N}{L^2}} \cdot |A(e^{i\theta})| \qquad \textbf{(simple trig)}$$

$$\implies ||\mu(x)||_1 \geq e^{-\frac{cN}{L^2}} \cdot \max_{\theta \in [-\pi/L, \pi/L]} |A(e^{i\theta})|$$

Mean-based algorithms & Littlewood polynomials

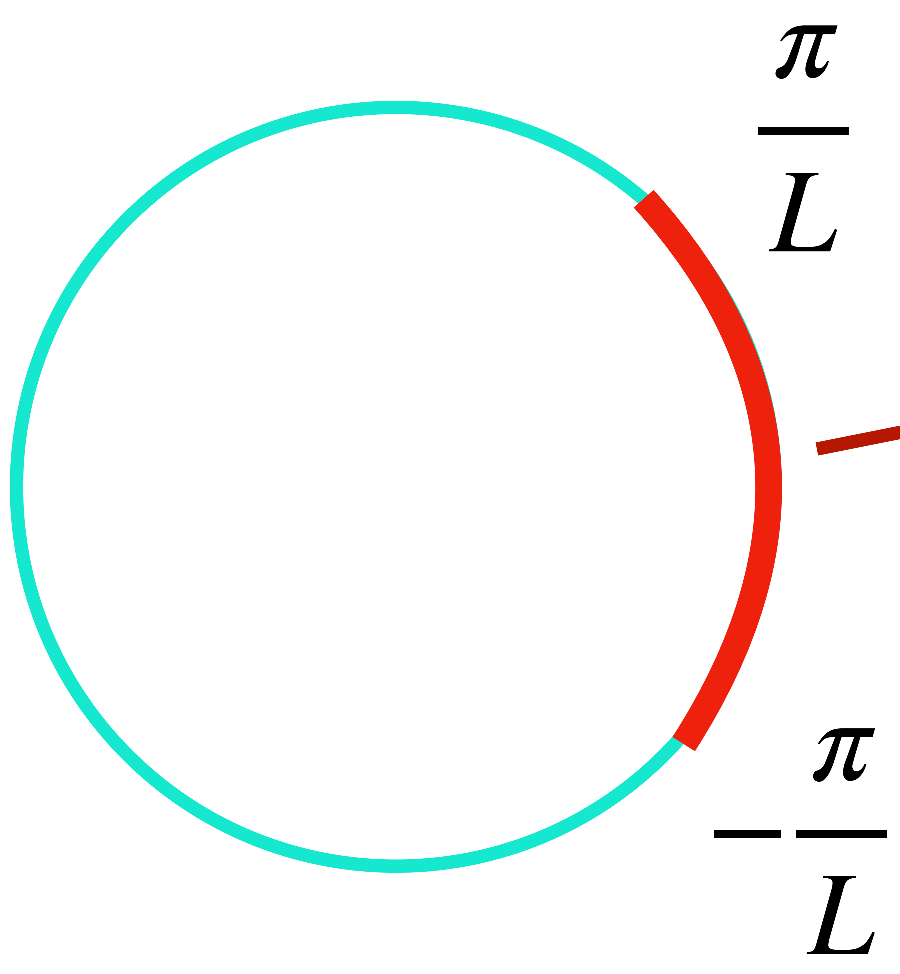
Deletion channel polynomial $$P_{d,x}(w) = \sum_{j=1}^{N} \mu(x)_j \cdot w^j, \quad w \in \mathbb{C}$$

Littlewood polynomial $A(z), \quad x_j \in \{-1,0,1\}$

$$P_{d,x}(w) = (1-d)\boxed{\sum_{j=1}^{N} x_j \cdot z^j}$$

$$z = d + (1-d)w$$

Easy to write in terms of x



[BorweinErdélyi97]

$A(z)$ Littlewood polynomial

$$\max |A(e^{i\theta})| \geq e^{-cL}$$

w **such that** $z = e^{i\theta}, \quad \theta \in [-\pi/L, \pi/L]$

$$\sum_j |\mu_j(x)| \cdot |w|^j \geq (1-d)\,|A(e^{i\theta})|$$ **(△-ineq)**

$$\implies ||\mu(x)||_1 \geq e^{-\frac{c'N}{L^2}} \cdot |A(e^{i\theta})|$$ **(simple trig)**

$$\implies ||\mu(x)||_1 \geq e^{-\frac{cN}{L^2}} \cdot \max_{\theta \in [-\pi/L, \pi/L]} |A(e^{i\theta})|$$

$$\geq e^{-cN^{1/3}}$$ **([BE97] + Max wrt L)**

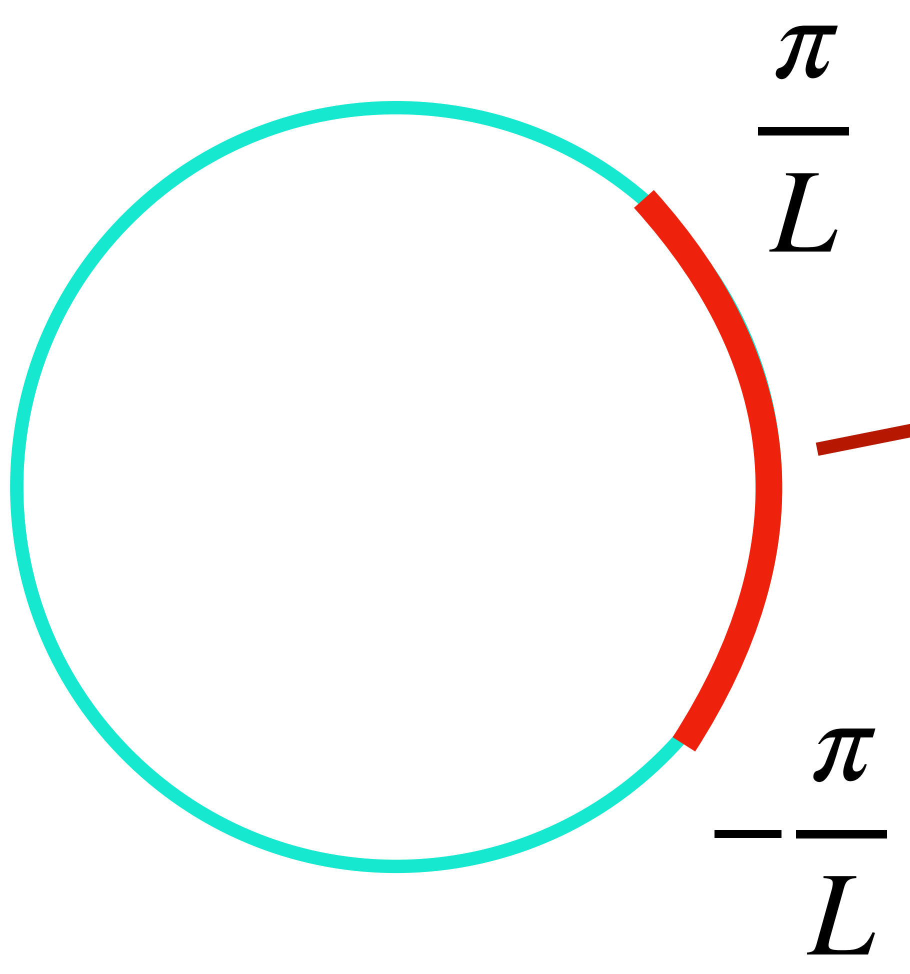
Mean-based algorithms & Littlewood polynomials

Deletion channel polynomial $P_{d,x}(w) = \sum_{j=1}^{N} \mu(x)_j \cdot w^j, \quad w \in \mathbb{C}$

Littlewood polynomial $A(z), \quad x_j \in \{-1,0,1\}$

$$P_{d,x}(w) = (1-d)\boxed{\sum_{j=1}^{N} x_j \cdot z^j} \qquad z = d + (1-d)w$$

Easy to write in terms of x



[BorweinErdélyi97]

$A(z)$ Littlewood polynomial

$\max |A(e^{i\theta})| \geq e^{-cL}$

w **such that** $z = e^{i\theta}, \quad \theta \in [-\pi/L, \pi/L]$

$$\sum_j |\mu_j(x)| \cdot |w|^j \geq (1-d)\,|A(e^{i\theta})| \qquad \textbf{(△-ineq)}$$

$$\implies ||\mu(x)||_1 \geq e^{-\frac{c' N}{L^2}} \cdot |A(e^{i\theta})| \qquad \textbf{(simple trig)}$$

$$\implies ||\mu(x)||_1 \geq e^{-\frac{cN}{L^2}} \cdot \max_{\theta \in [-\pi/L, \pi/L]} |A(e^{i\theta})|$$

$$\geq e^{-cN^{1/3}} \qquad \textbf{([BE97] + Max wrt L)}$$

$e^{O(N^{1/3})}$ **traces suffice to distinguish mean traces** $\Longleftarrow$

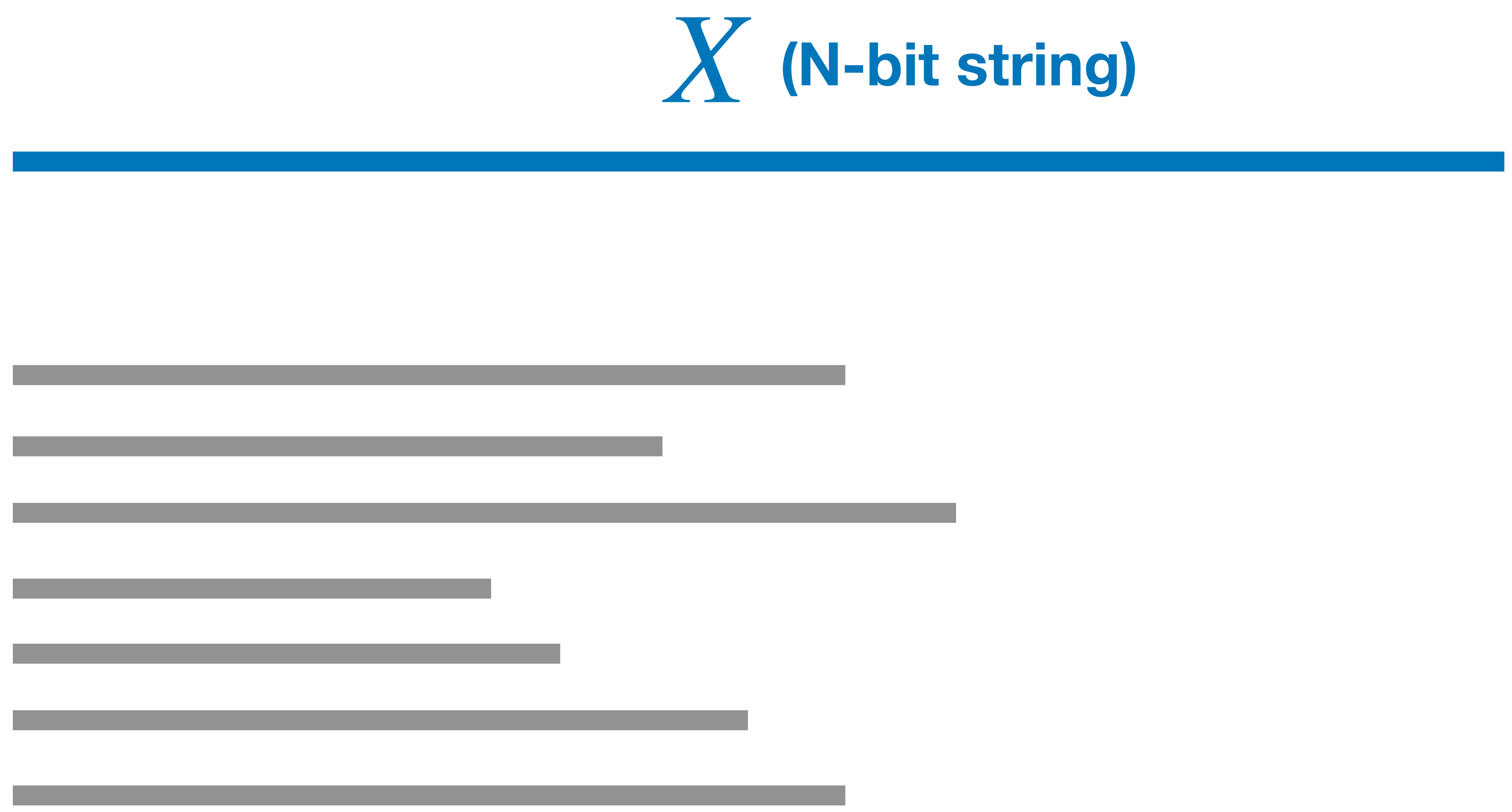
Average-case trace reconstruction

Recall: Average error probability of reconstruction algorithm must be $O(1/N)$.

Reconstruction algorithms	#traces	deletion probability
Batu, Kannan, Khanna, McGregor 2004	$\log N$	$d = 1/\log N$
Holenstein, Mitzenmacher, Panigrahy, Wieder 2008	**poly**(N)	$d < c$ for small absolute constant c
Peres, Zhai 2017	$\exp(\log^{1/2} N)$	$d < 1/2$
Holden, Pemantle, Peres 2018	$\exp(\log^{1/3} N)$	$d < 1$

A recipe for average-case trace reconstruction

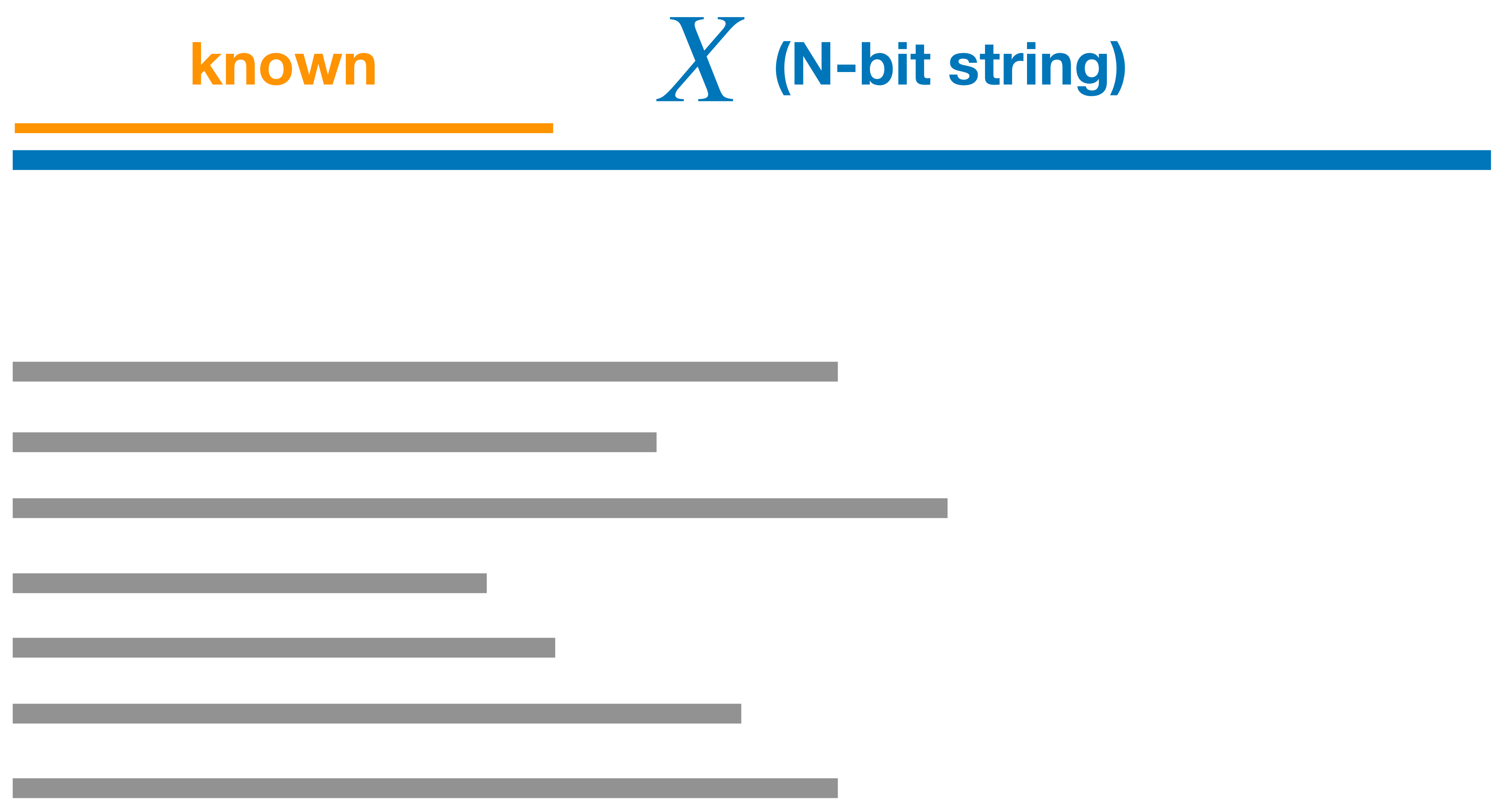
[HolensteinMitzenmacherPanigrahyWieder08]



A recipe for average-case trace reconstruction

[HolensteinMitzenmacherPanigrahyWieder08]

1) Bootstrapping: Learn first bits of X "for free"



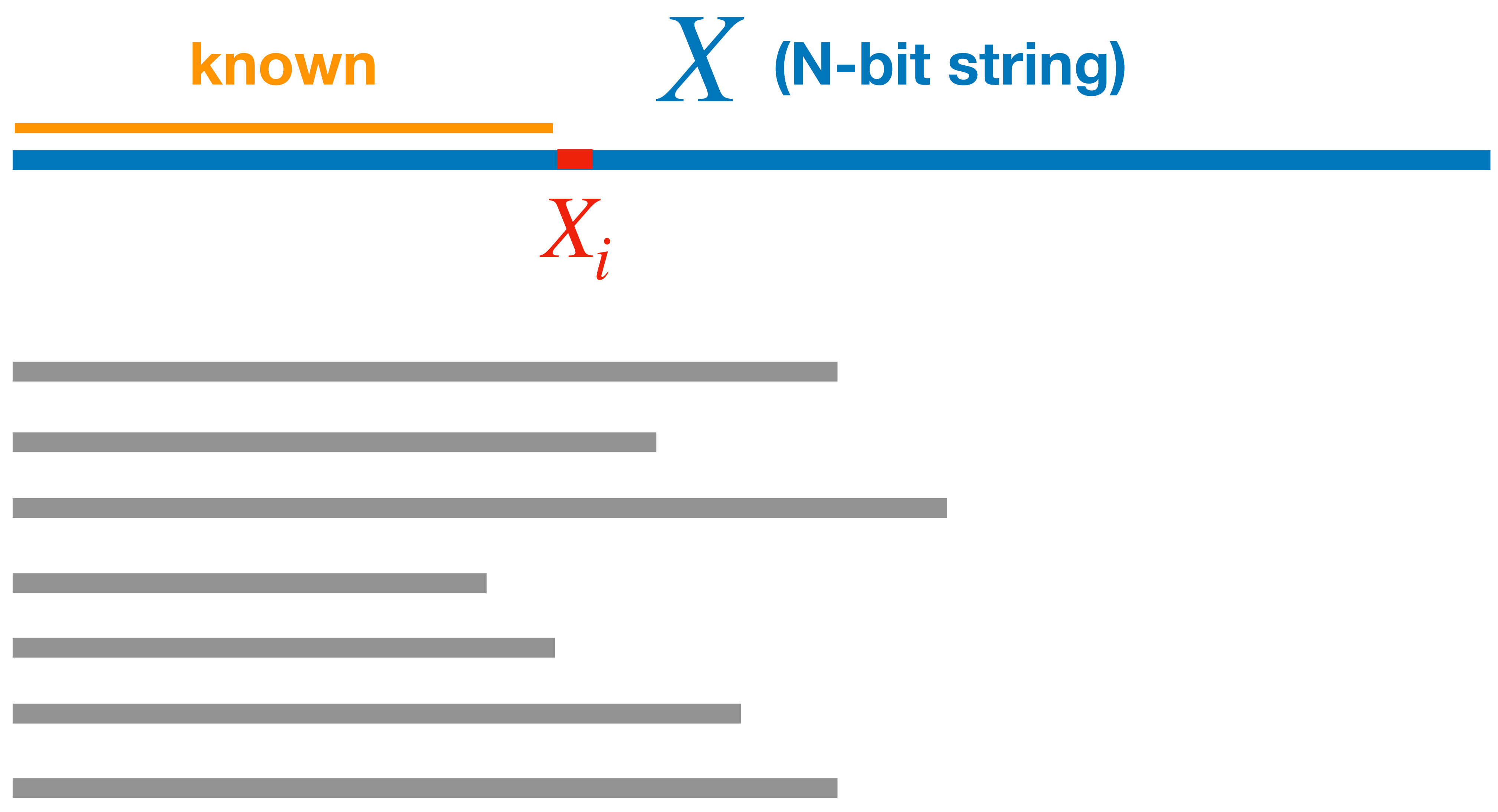
A recipe for average-case trace reconstruction

[HolensteinMitzenmacherPanigrahyWieder08]

1) Bootstrapping: Learn first bits of X "for free"

Suppose we know $X_1, X_2, \ldots, X_{i-1}$

Goal: Find X_i



A recipe for average-case trace reconstruction

[HolensteinMitzenmacherPanigrahyWieder08]

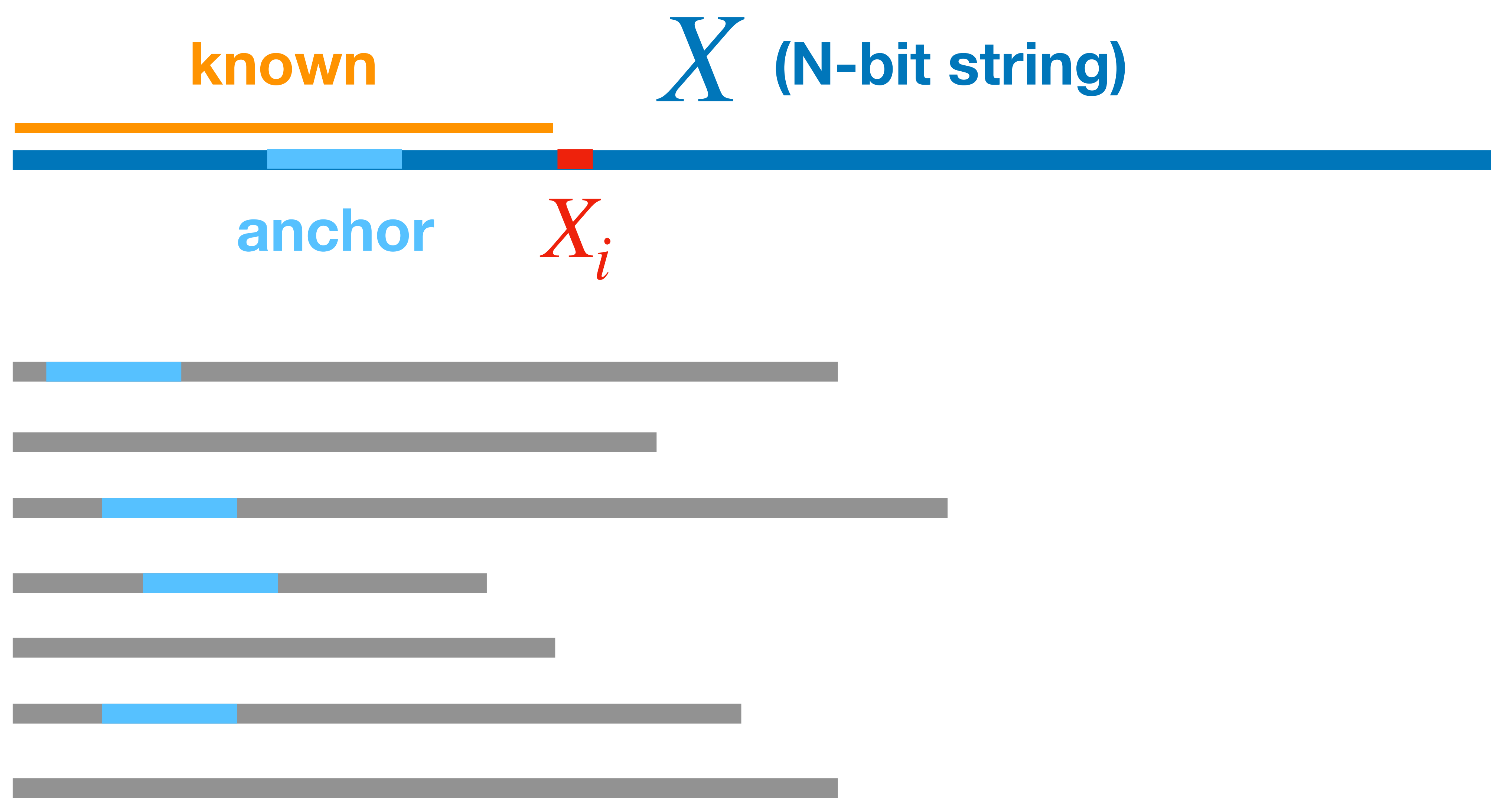
1) Bootstrapping: Learn first bits of X "for free"

Suppose we know $X_1, X_2, \ldots, X_{i-1}$

Goal: Find X_i

2) Trace alignment: Align by **anchor** close to X_i

"If X is random, whp anchor in trace comes from anchor in X."



A recipe for average-case trace reconstruction

[HolensteinMitzenmacherPanigrahyWieder08]

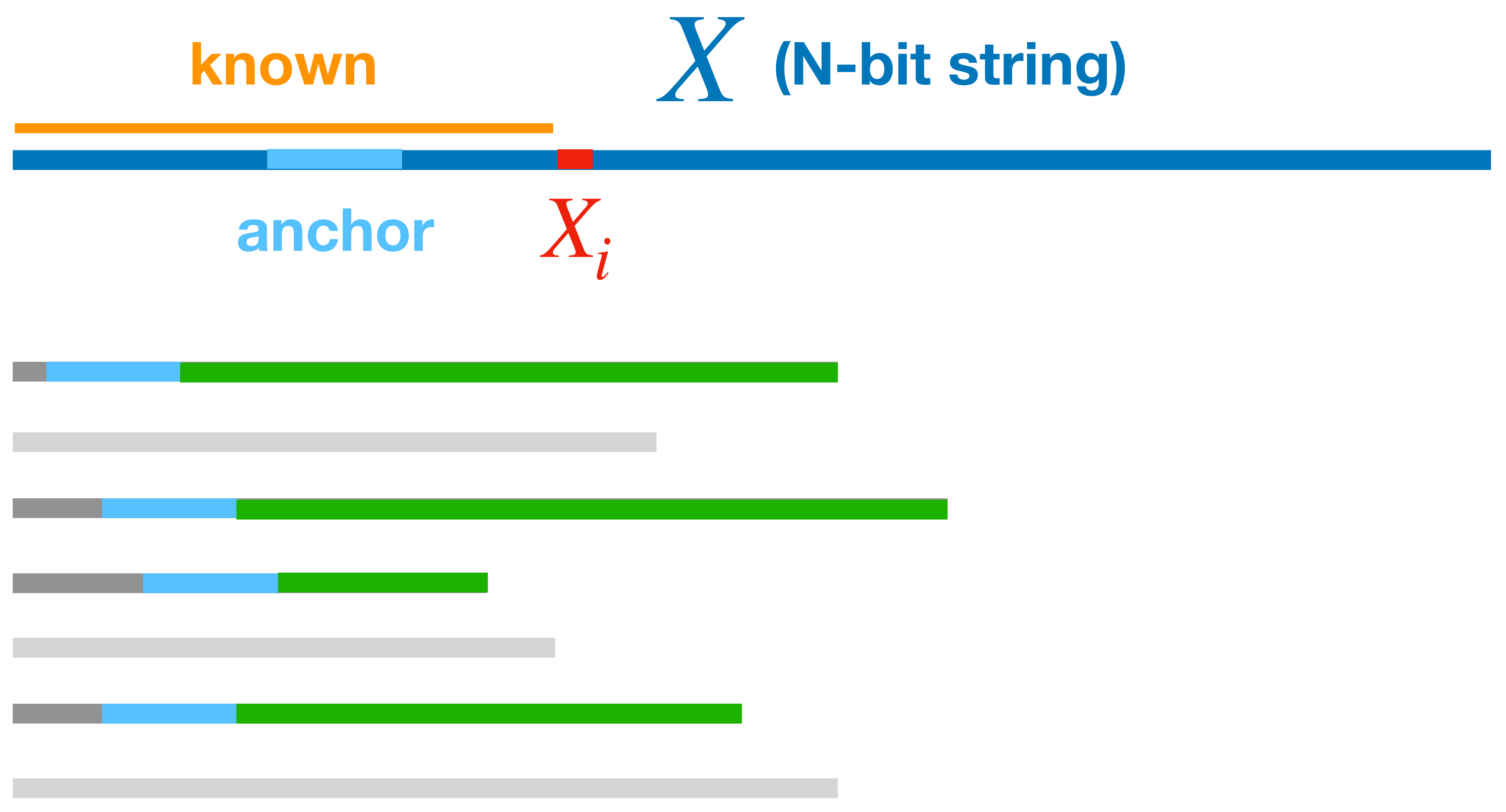
1) Bootstrapping: Learn first bits of X "for free"

Suppose we know $X_1, X_2, \ldots, X_{i-1}$

Goal: Find X_i

2) Trace alignment: Align by **anchor** close to X_i

"If X is random, whp anchor in trace comes from anchor in X."



A recipe for average-case trace reconstruction

[HolensteinMitzenmacherPanigrahyWieder08]

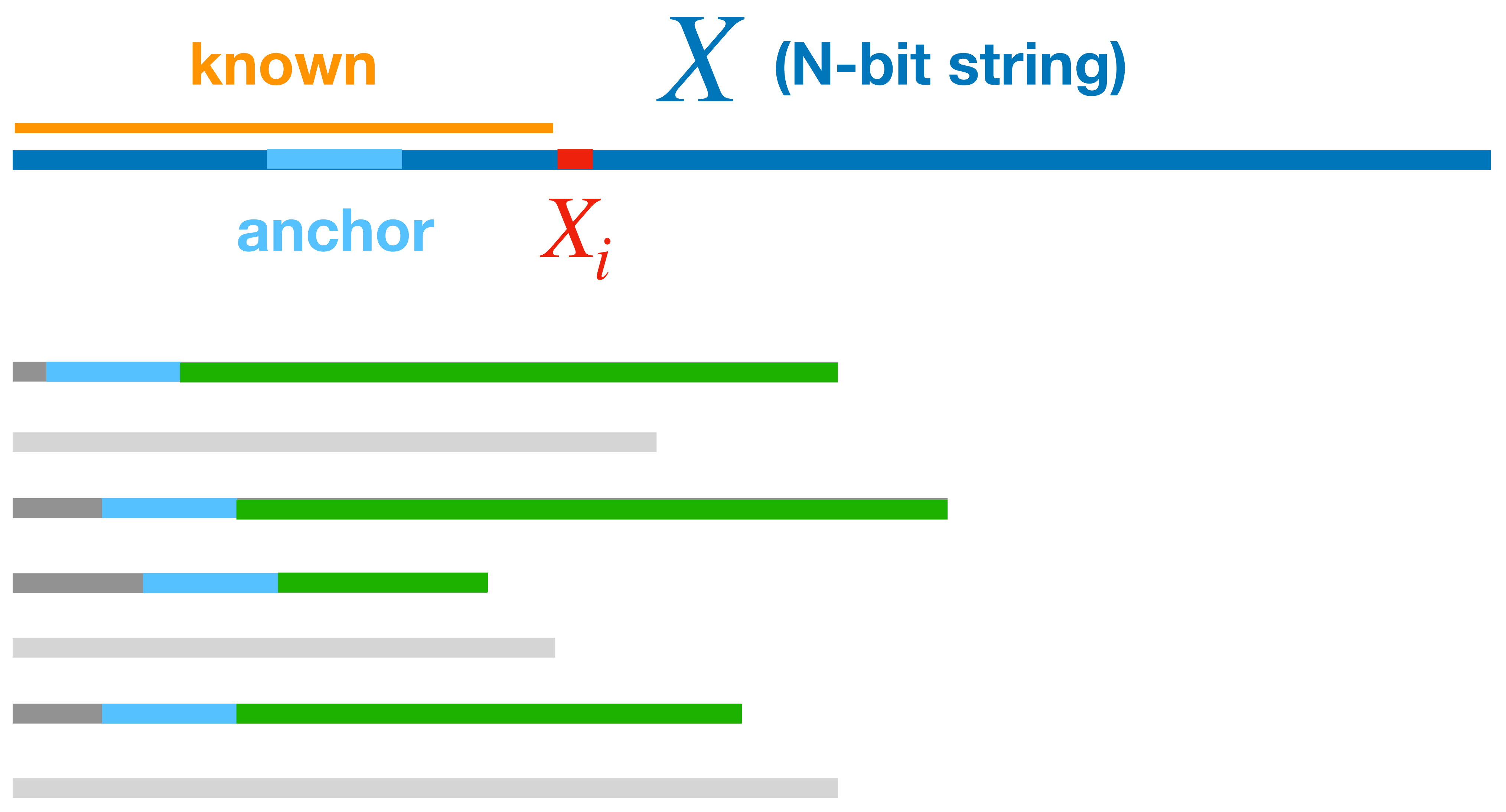
1) Bootstrapping: Learn first bits of X "for free"

Suppose we know $X_1, X_2, \ldots, X_{i-1}$

Goal: Find X_i

2) Trace alignment: Align by **anchor** close to X_i

"If X is random, whp anchor in trace comes from anchor in X."



3) Reconstruction: Estimate special bit

$Y =$ distribution of **"trace after anchor"**

"There is special position $Y_{j^\star}$ *that is decently influenced by* X_i *"*

$$|\Pr[Y_{j^\star} = 1 \mid X_i = 1] - \Pr[Y_{j^\star} = 1 \mid X_i = 0]| \geq \frac{1}{N^C}$$

A recipe for average-case trace reconstruction

[HolensteinMitzenmacherPanigrahyWieder08]

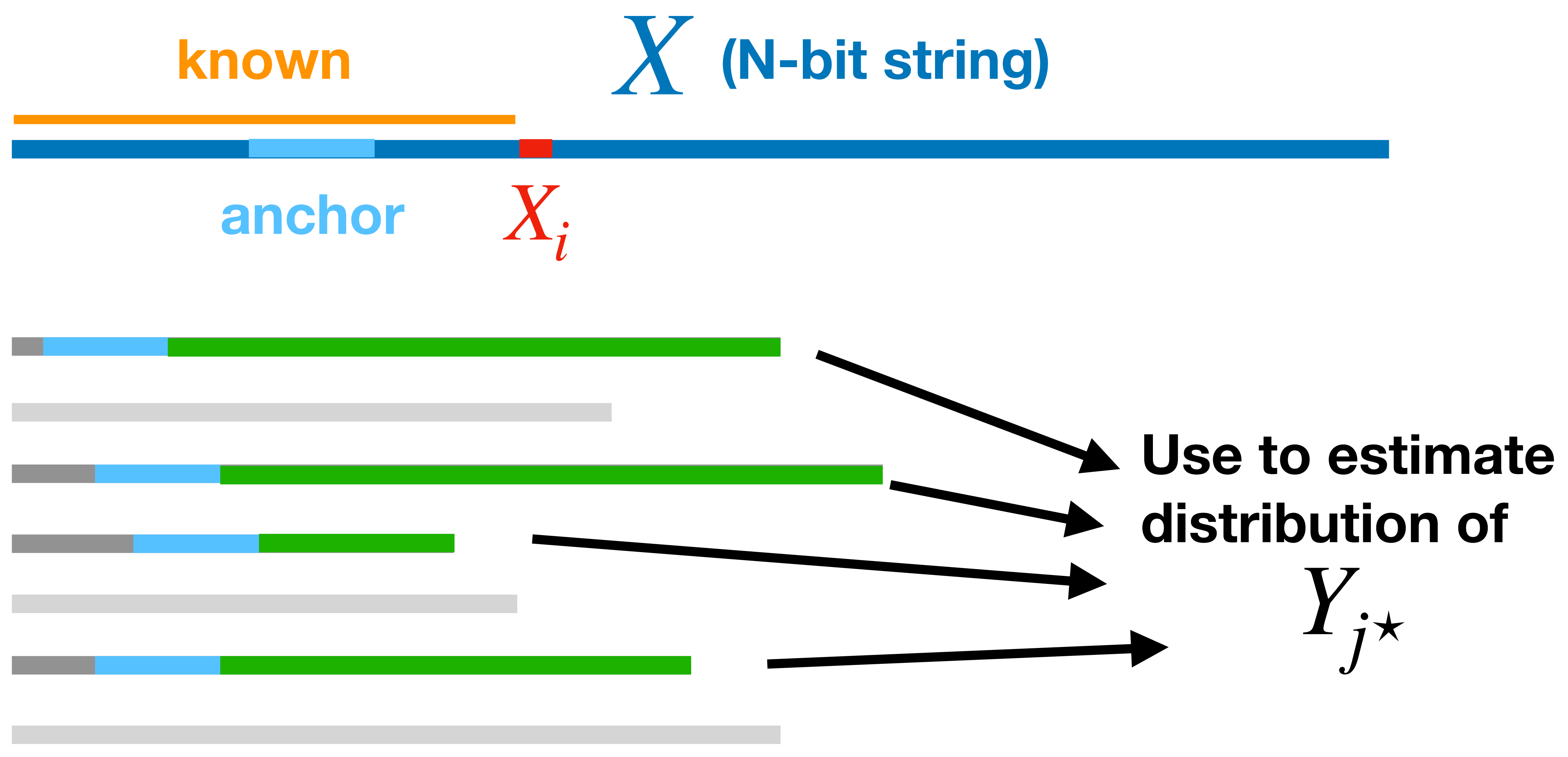
1) Bootstrapping: Learn first bits of X "for free"

Suppose we know $X_1, X_2, \ldots, X_{i-1}$

Goal: Find X_i

2) Trace alignment: Align by **anchor** close to X_i

"If X is random, whp anchor in trace comes from anchor in X."



3) Reconstruction: Estimate special bit

$Y =$ distribution of **"trace after anchor"**

"There is special position $Y_{j^\star}$ that is decently influenced by X_i"

$$|\Pr[Y_{j^\star} = 1 \mid X_i = \underline{1}] - \Pr[Y_{j^\star} = 1 \mid X_i = \underline{0}]| \geq \frac{1}{N^C}$$

A recipe for average-case trace reconstruction

[HolensteinMitzenmacherPanigrahyWieder08]

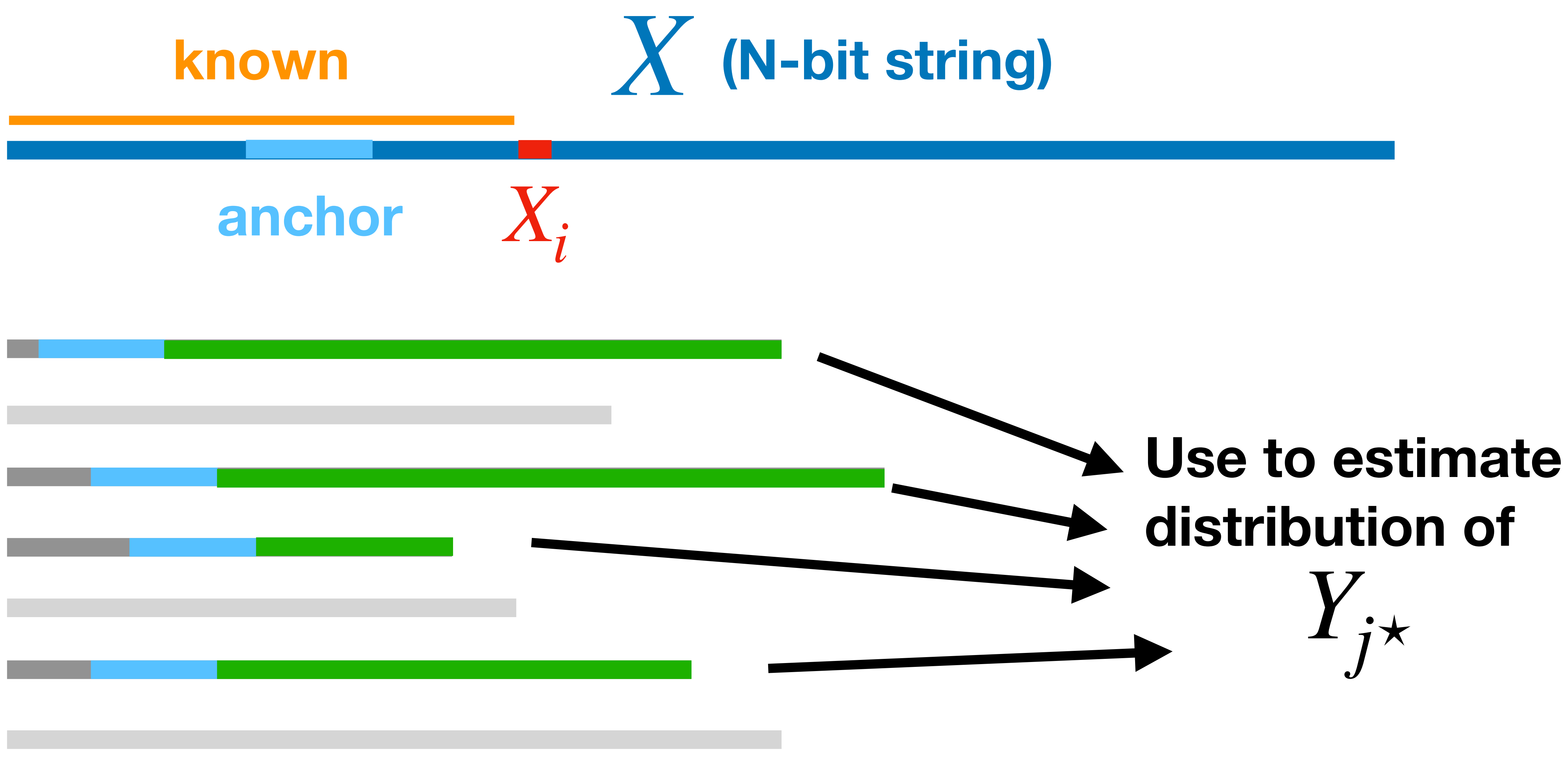
1) Bootstrapping: Learn first bits of X "for free"

Suppose we know $X_1, X_2, \ldots, X_{i-1}$

Goal: Find X_i

2) Trace alignment: Align by **anchor** close to X_i

"If X is random, whp anchor in trace comes from anchor in X."



3) Reconstruction: Estimate special bit

$Y =$ distribution of **"trace after anchor"**

"There is special position $Y_{j^\star}$ that is decently influenced by X_i"

$$|\Pr[Y_{j^\star} = 1 \mid X_i = \underline{1}] - \Pr[Y_{j^\star} = 1 \mid X_i = \underline{0}]| \geq \frac{1}{N^C}$$

$\boxed{\text{Recover } X_i \text{ whp using } \textbf{poly}(N) \text{ traces!}}$

Lower bounds for trace reconstruction

General recipe for lower bounds:

- **Worst-case:** Show it is hard to distinguish between two specific strings with few traces;
- **Average-case:** Worst-case LB + we expect bad string of length $\frac{\log N}{2}$ to show up $\approx \sqrt{N}$ times in random N-bit string.

Lower bounds #traces	Worst-case	Average-case
McGregor, Price, Vorotnikova 2014	N	$\log^2 N$
Holden, Lyons 2018	$N^{1.25}$	$\log^{2.25} N$
Chase 2019	$N^{1.5}$	$\log^{2.5} N$

Lower bounds for trace reconstruction

[McGregorPriceVorotnikova14]

N traces **necessary and sufficient** to distinguish

$0^{N-1}10^{N}$ **vs.** $0^{N}10^{N-1}$

$$0000\boxed{10}0000$$
$$0000\boxed{01}0000$$

[HoldenLyons18, Chase19]

$N^{1.5}$ traces **necessary and sufficient** to distinguish

$(01)^{N-1}10(01)^{N}$ **vs.** $(01)^{N}10(01)^{N-1}$

$$0101\boxed{1001}0101$$
$$0101\boxed{0110}0101$$

Summing up…

	Worst-case trace reconstruction	Average-case trace reconstruction
Upper bounds #traces	$\exp(O(N^{1/3}))$ [DeO'DonnellServedio17, NazarovPeres17]	$\exp(O(\log^{1/3} N))$ [HoldenPemantlePeres18]
Lower bounds #traces	$\approx N^{3/2}$ [HoldenLyons18, Chase19]	$\approx \log^{5/2} N$ [HoldenLyons18, Chase19]

Summing up…

	Worst-case trace reconstruction	Average-case trace reconstruction
Upper bounds #traces	$\exp(O(N^{1/3}))$ [DeO'DonnellServedio17, NazarovPeres17]	$\exp(O(\log^{1/3} N))$ [HoldenPemantlePeres18]
Lower bounds #traces	$\approx N^{3/2}$ [HoldenLyons18, Chase19]	$\approx \log^{5/2} N$ [HoldenLyons18, Chase19]

almost exponential gap

Summing up…

	Worst-case trace reconstruction	Average-case trace reconstruction
Upper bounds #traces	$\exp(O(N^{1/3}))$ [DeO'DonnellServedio17, NazarovPeres17]	$\exp(O(\log^{1/3} N))$ [HoldenPemantlePeres18]
Lower bounds #traces	$\approx N^{3/2}$ [HoldenLyons18, Chase19]	$\approx \log^{5/2} N$ [HoldenLyons18, Chase19]

almost exponential gap

***All results over $\{0,1\}^N$, as it is the hardest setting**

Trace reconstruction and portable DNA-based storage

[YazdiGabrysMilenkovic17, Organick+18]

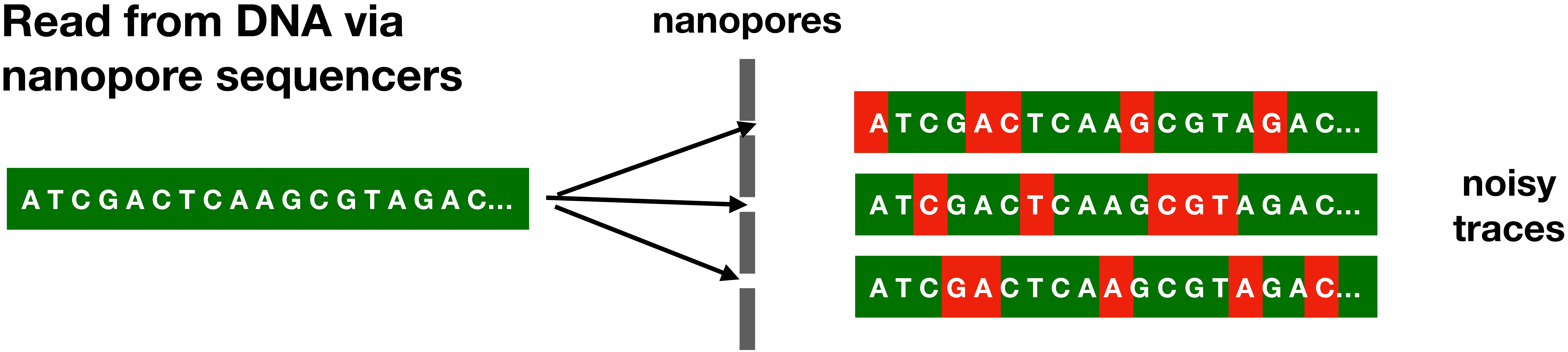
Write process: Data is encoded into {A,G,T,C} alphabet, then synthesized into DNA strand.

Trace reconstruction and portable DNA-based storage

[YazdiGabrysMilenkovic17, Organick+18]

Write process: Data is encoded into {A,G,T,C} alphabet, then synthesized into DNA strand.

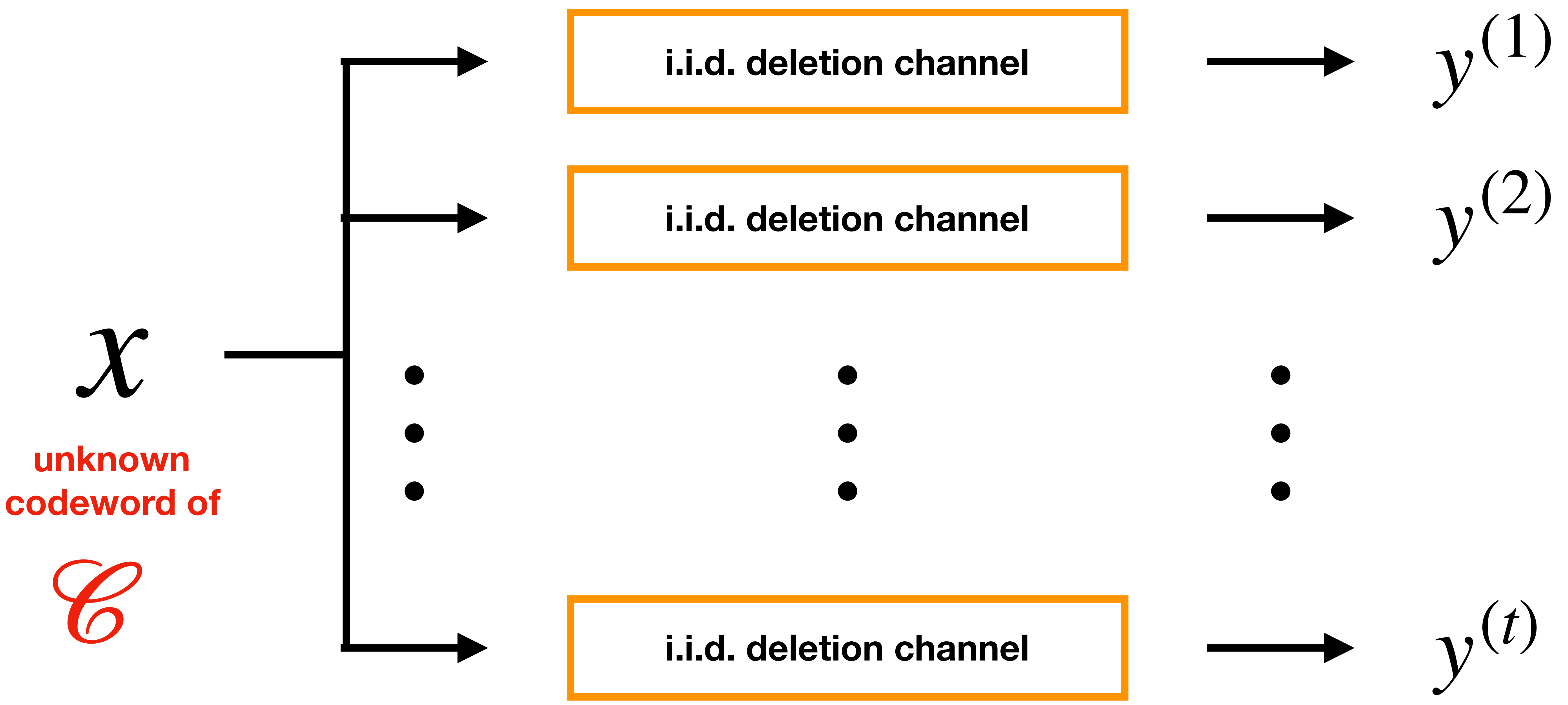
Read from DNA via nanopore sequencers



Trace reconstruction for the coding schemes in [YGM17, OAC+18] is based on heuristics.

Coded trace reconstruction

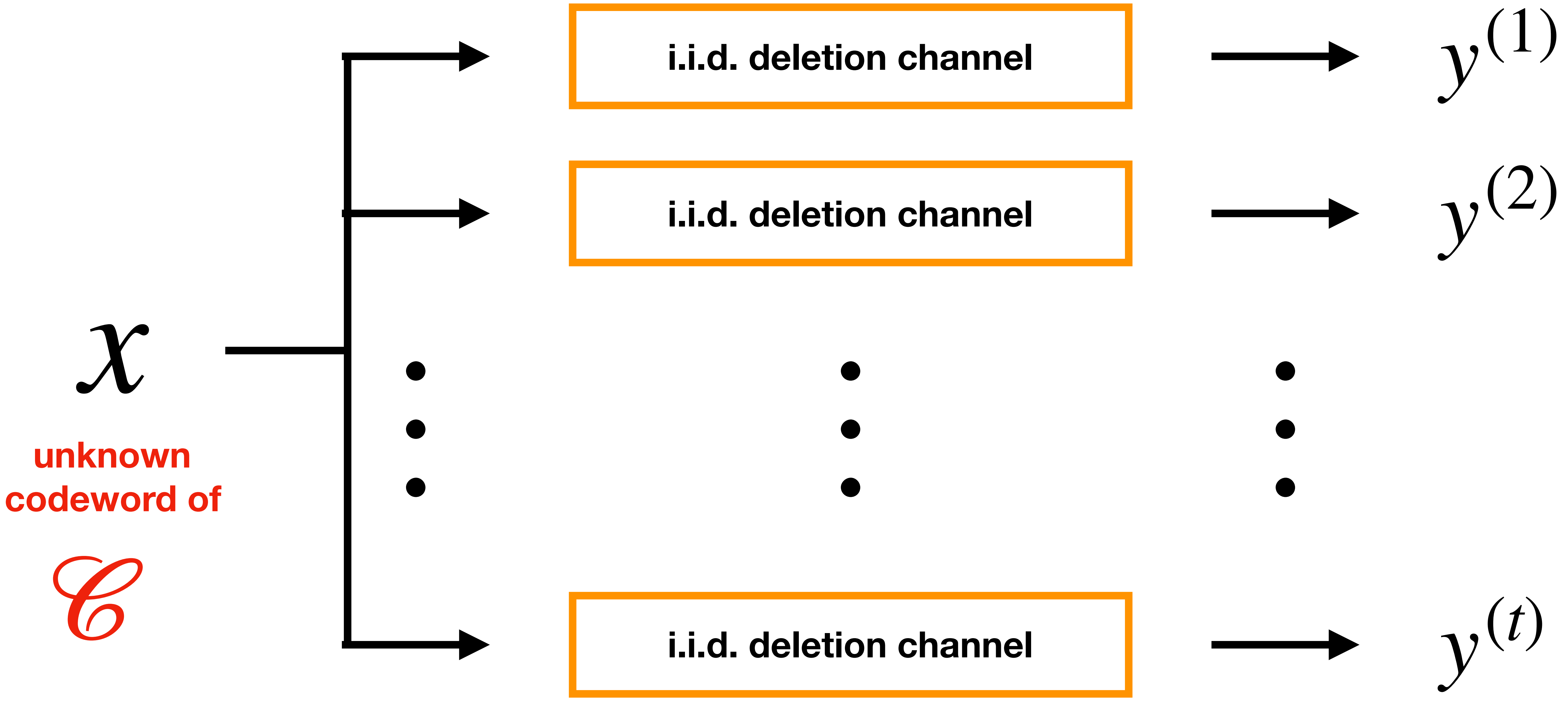
[CheraghchiGabrysMilenkovicRibeiro19]



$\mathscr{C} \subseteq \{0,1\}^N$ **with** $|\mathscr{C}| = 2^{RN}$

Coded trace reconstruction

[CheraghchiGabrysMilenkovicRibeiro19]

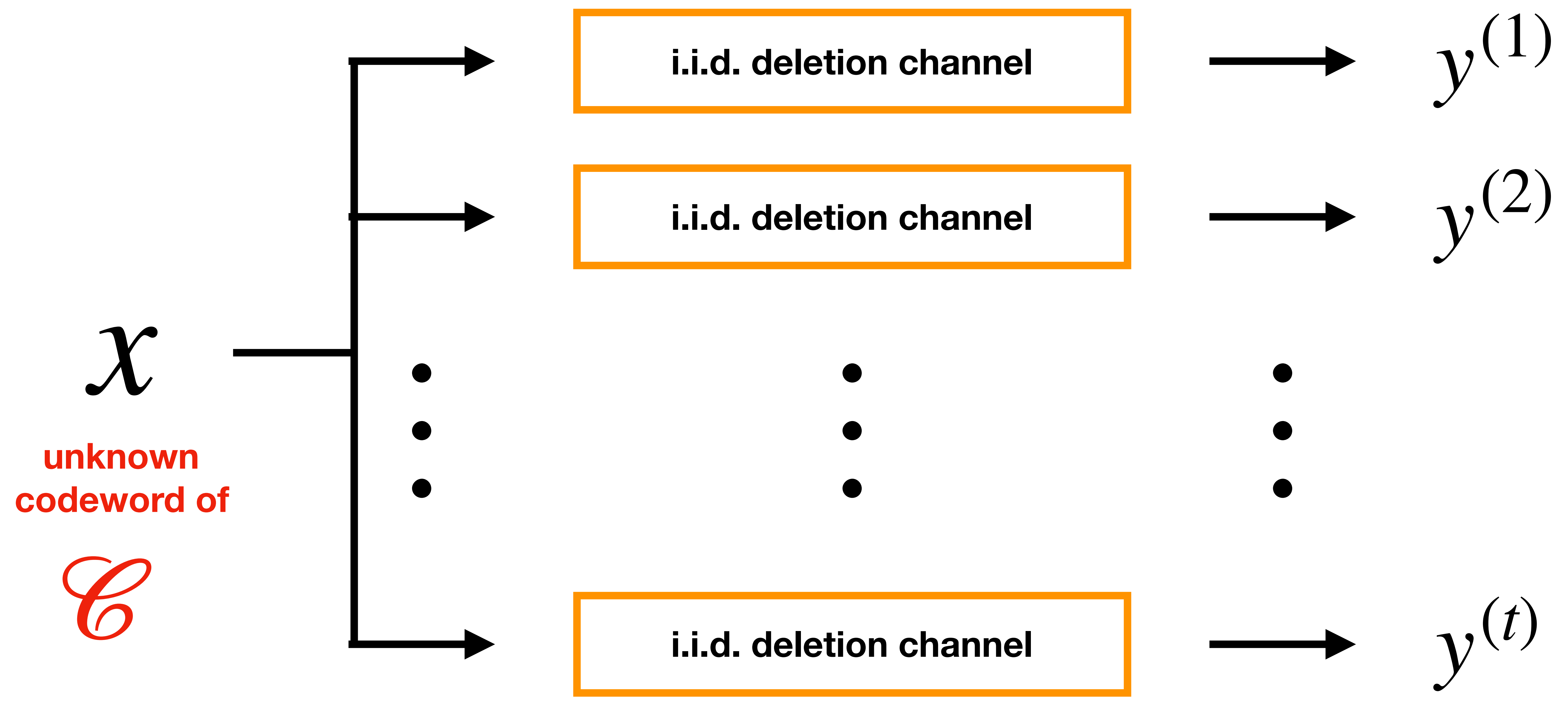


$\mathscr{C} \subseteq \{0,1\}^N$ **with** $|\mathscr{C}| = 2^{RN}$

- high rate $R = 1 - \epsilon, \quad \epsilon = o_N(1)$

Coded trace reconstruction

[CheraghchiGabrysMilenkovicRibeiro19]

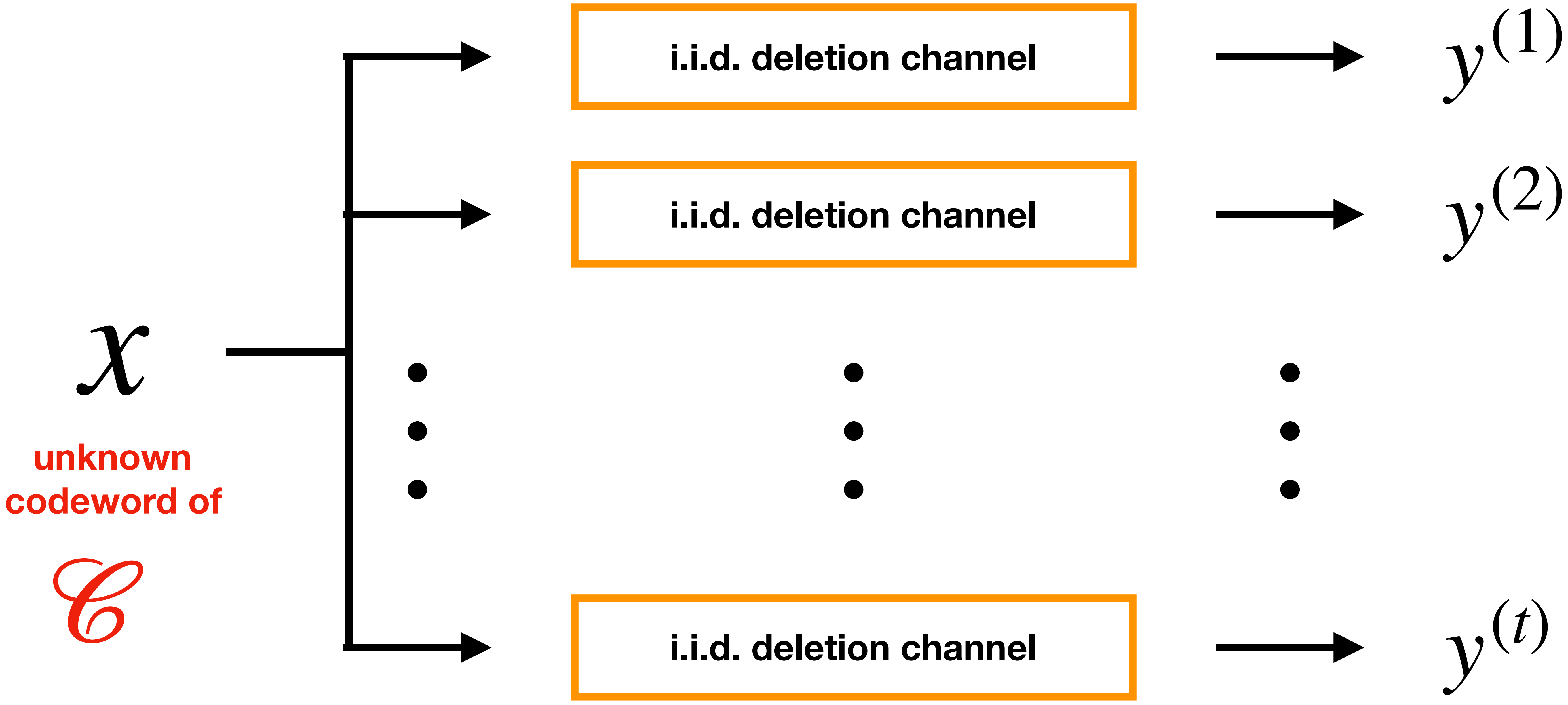


$\mathscr{C} \subseteq \{0,1\}^N$ **with** $|\mathscr{C}| = 2^{RN}$

- high rate $R = 1 - \epsilon, \quad \epsilon = o_N(1)$
- reconstructed from few traces ($t = t(1/\epsilon)$ slow-growing) with probability $1 - O(1/N)$

Coded trace reconstruction

[CheraghchiGabrysMilenkovicRibeiro19]



$\mathscr{C} \subseteq \{0,1\}^N$ **with** $|\mathscr{C}| = 2^{RN}$

- high rate $R = 1 - \epsilon, \quad \epsilon = o_N(1)$
- reconstructed from few traces ($t = t(1/\epsilon)$ slow-growing) with probability $1 - O(1/N)$
- **efficiently** encodable and reconstructable (time **poly**(N))

Some related work

Haeupler, Mitzenmacher 2014	First-order capacity for small deletion probability **(constant number of traces only)**
Abroshan, Venkataramanan, Dolecek, Guillén i Fàbregas 2019	Coding for multiple deletion channels with **constant number of deletions only**. Uses VT codes. Builds up on work about file synchronization.
Average-case trace reconstruction	Implies existence of large codes reconstructable from few traces. **No efficient encoding + we want to use even fewer traces.**

PART I:

Markers + worst-case trace reconstruction

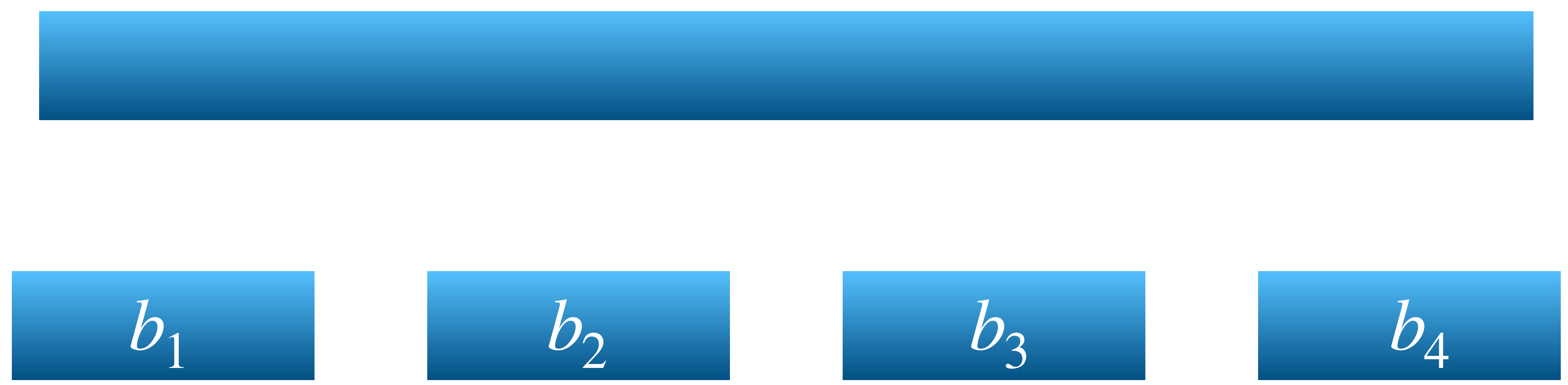
Basic construction

x N**-bit string**

Basic construction

x N**-bit string**

split into blocks with $m = O(\log^2 N)$ **bits**

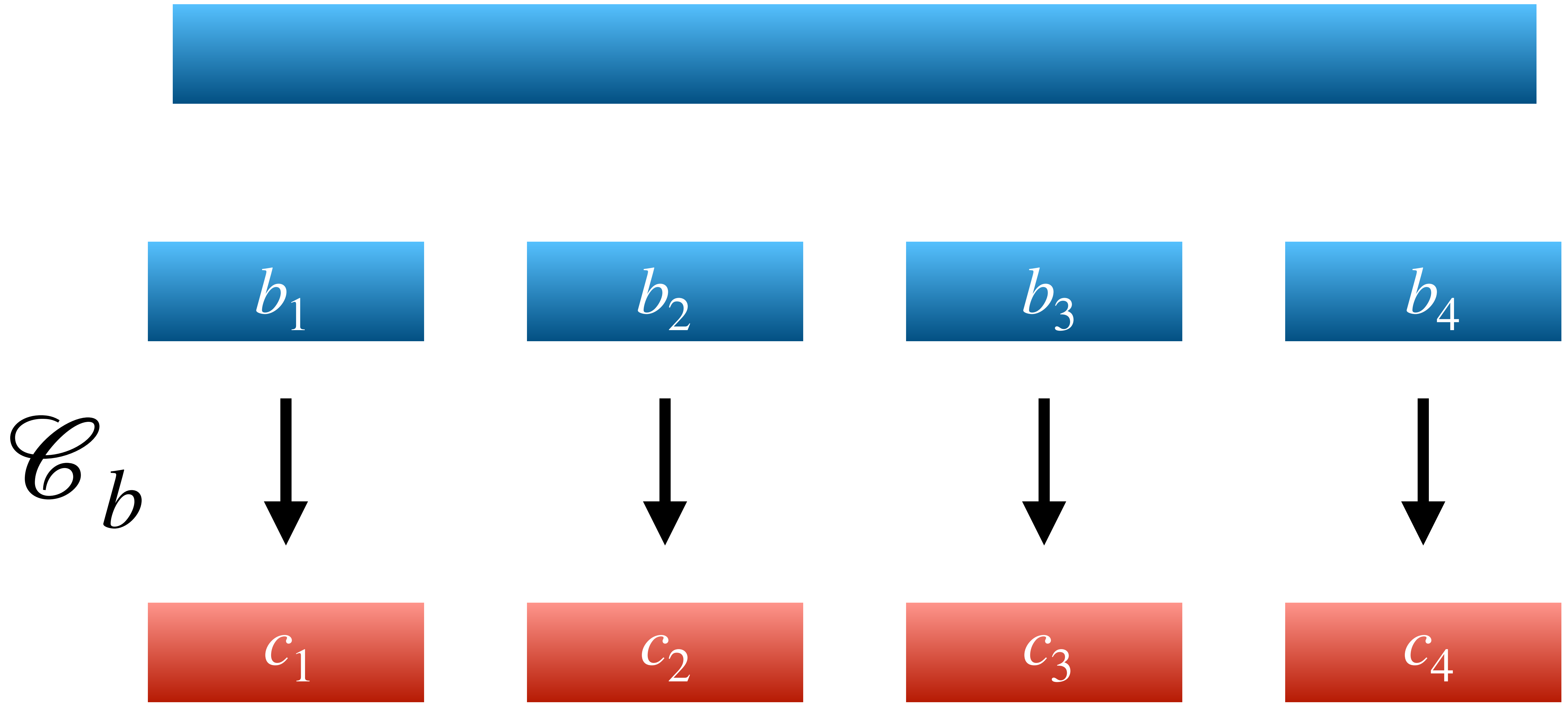


Basic construction

x $\quad$ N-bit string

split into blocks with $m = O(\log^2 N)$ bits

encode each block using appropriate sub-code $\mathscr{C}_b$



Basic construction

x $\quad$ N**-bit string**

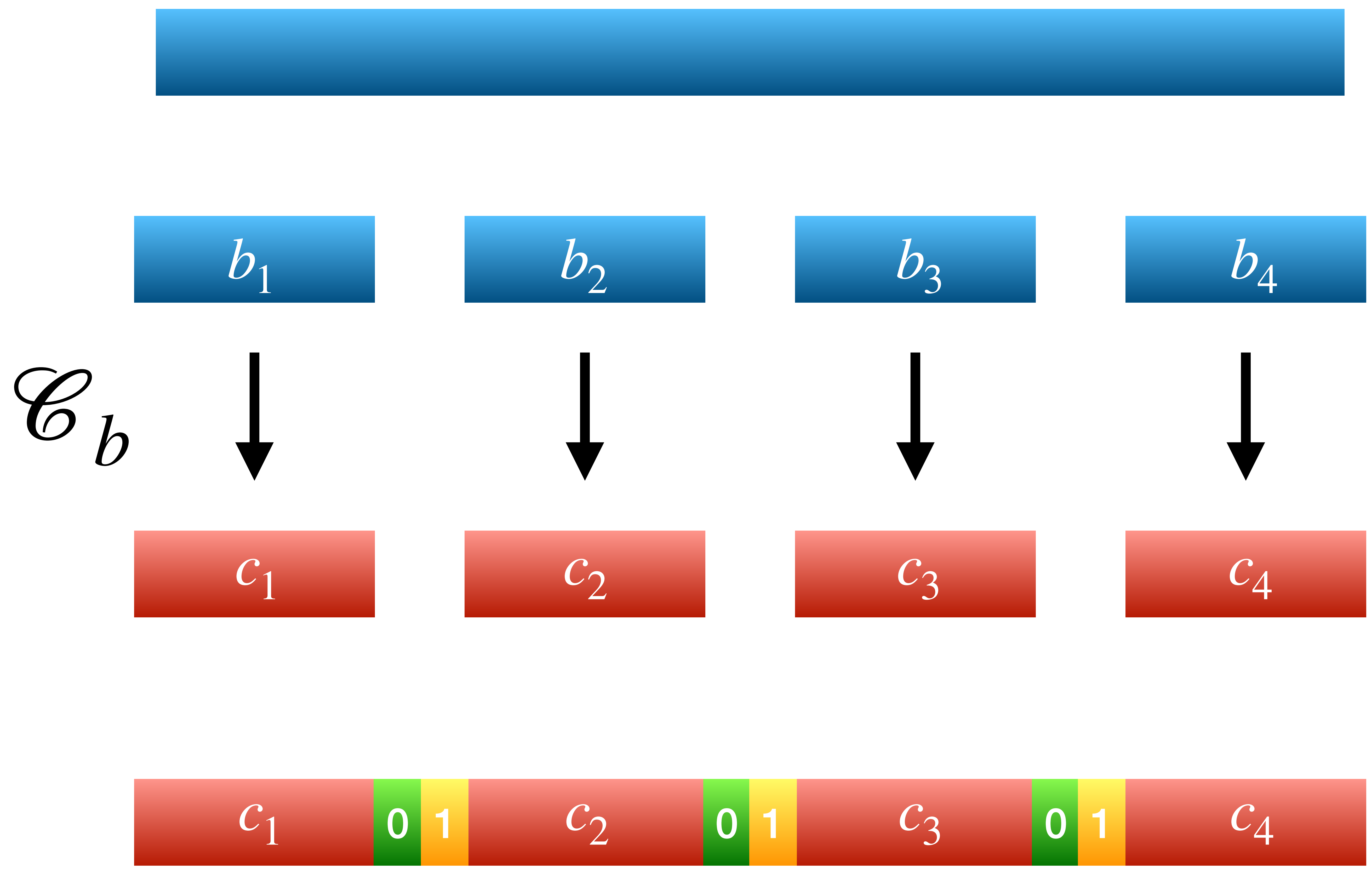
split into blocks with
$m = O(\log^2 N)$ **bits**

encode each block using appropriate sub-code $\mathscr{C}_b$

add markers between blocks

0 $= 0000\ldots00$
O(log N) bits

1 $= 1111\ldots11$
O(log N) bits



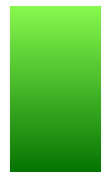
Reconstruction

$= 0000\ldots00$
O(log N) bits

$= 1111\ldots11$
O(log N) bits

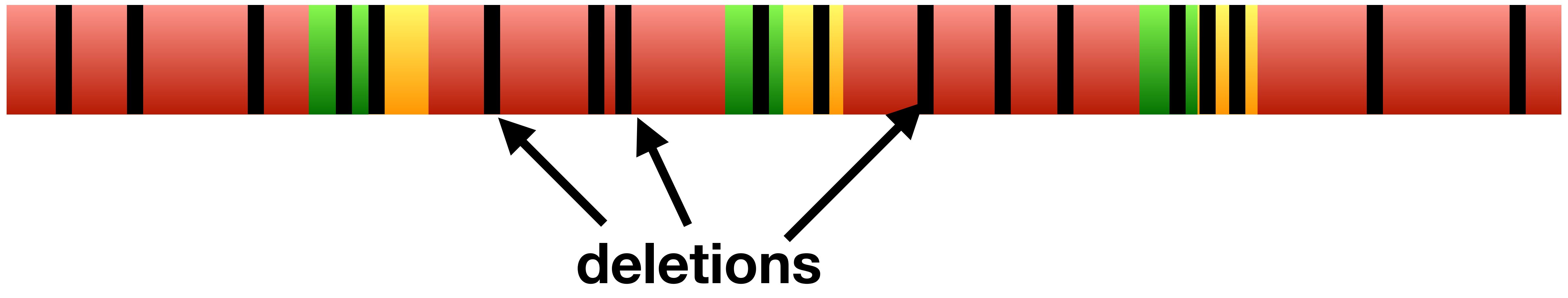
$\in \mathscr{C}_b$

Reconstruction

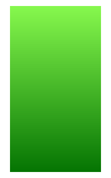
 $= 0000\ldots00$
O(log N) bits

 $= 1111\ldots11$
O(log N) bits

 $\in \mathscr{C}_b$



Reconstruction

 $= 0000\ldots00$

O(log N) bits

 $= 1111\ldots11$

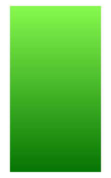
O(log N) bits

 $\in \mathscr{C}_b$

Trace of codeword

Key observation: part of trace coming from a marker still looks like a marker.

Reconstruction

 $= 0000\ldots00$

O(log N) bits

 $= 1111\ldots11$

O(log N) bits

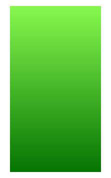
 $\in \mathscr{C}_b$

Trace of codeword

Key observation: part of trace coming from a marker still looks like a marker.

If $\mathscr{C}_b$ is chosen appropriately, long runs of 0's in trace **only** come from markers

Reconstruction

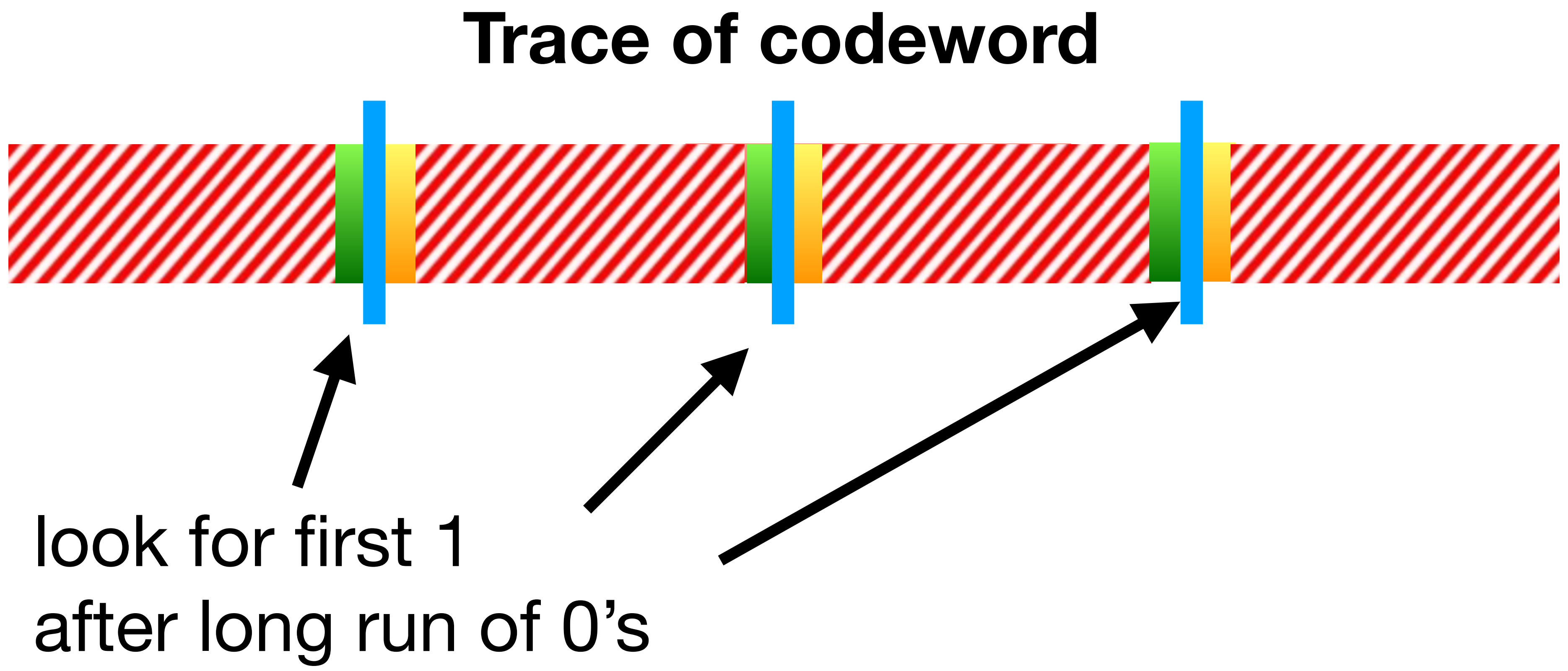
 $= 0000\ldots00$
O(log N) bits

 $= 1111\ldots11$
O(log N) bits

 $\in \mathscr{C}_b$

Key observation: part of trace coming from a marker still looks like a marker.

If $\mathscr{C}_b$ is chosen appropriately, long runs of 0's in trace **only** come from markers

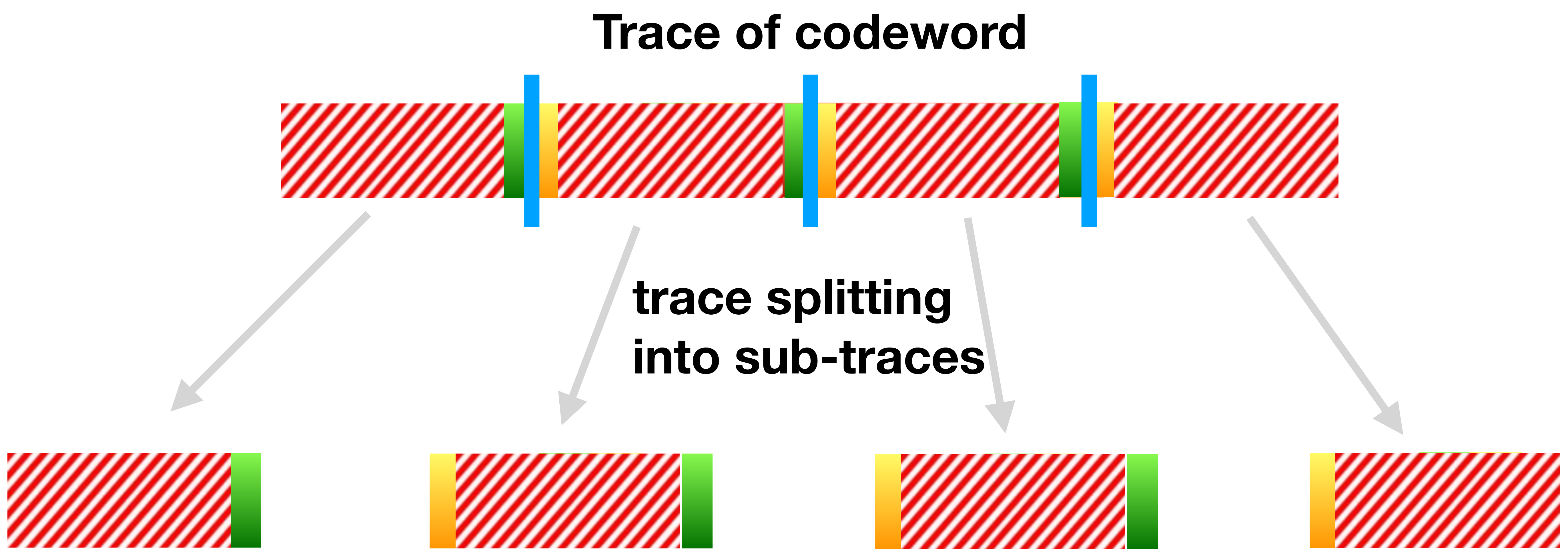


Reconstruction

$= 0000\ldots00$ **O(log N) bits**

$= 1111\ldots11$ **O(log N) bits**

$\in \mathscr{C}_b$



Key observation: part of trace coming from a marker still looks like a marker.

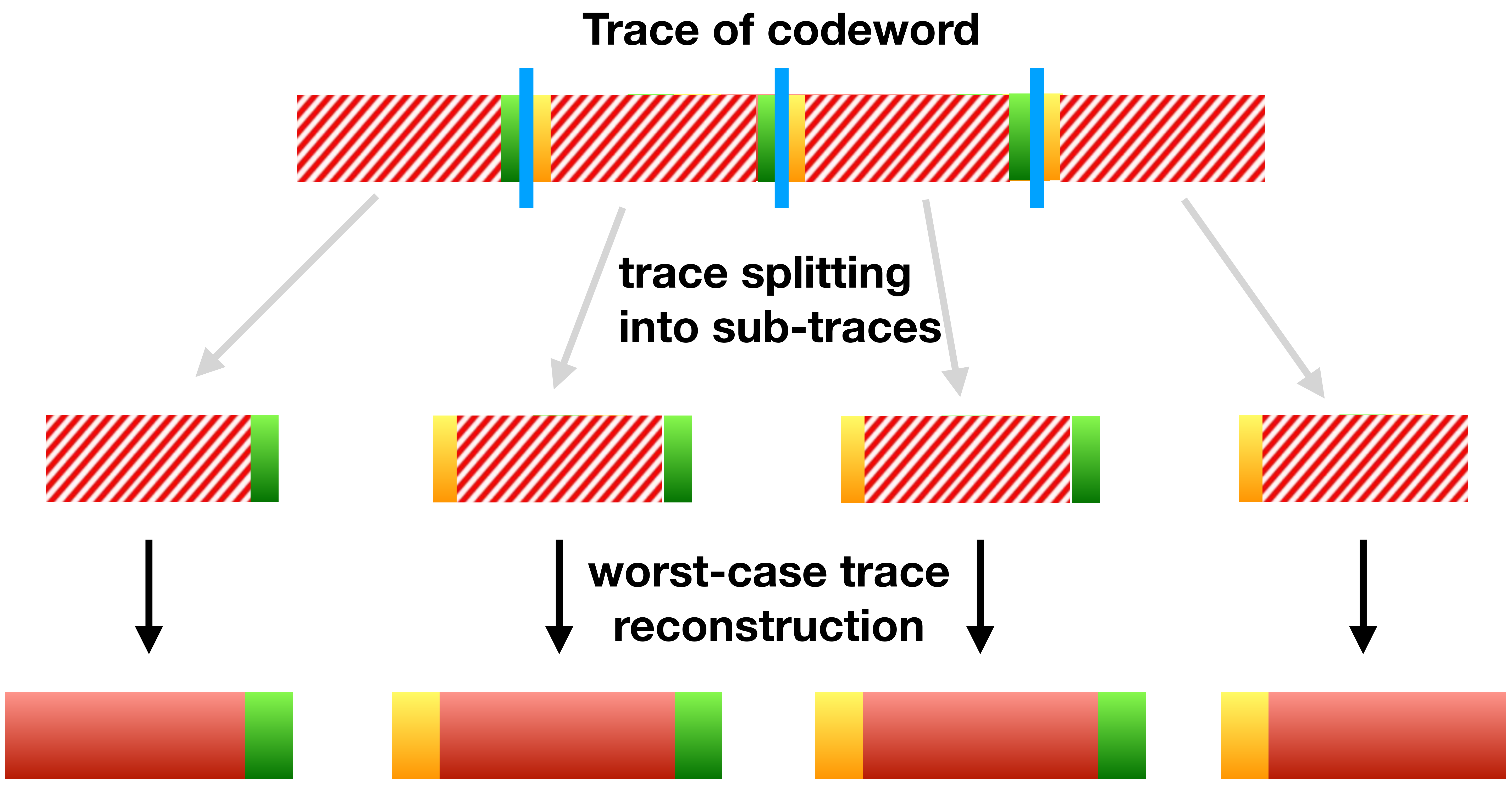
If $\mathscr{C}_b$ is chosen appropriately, long runs of 0's in trace **only** come from markers

Reconstruction

$= 0000\ldots00$
O(log N) bits

$= 1111\ldots11$
O(log N) bits

$\in \mathscr{C}_b$



Key observation: part of trace coming from a marker still looks like a marker.

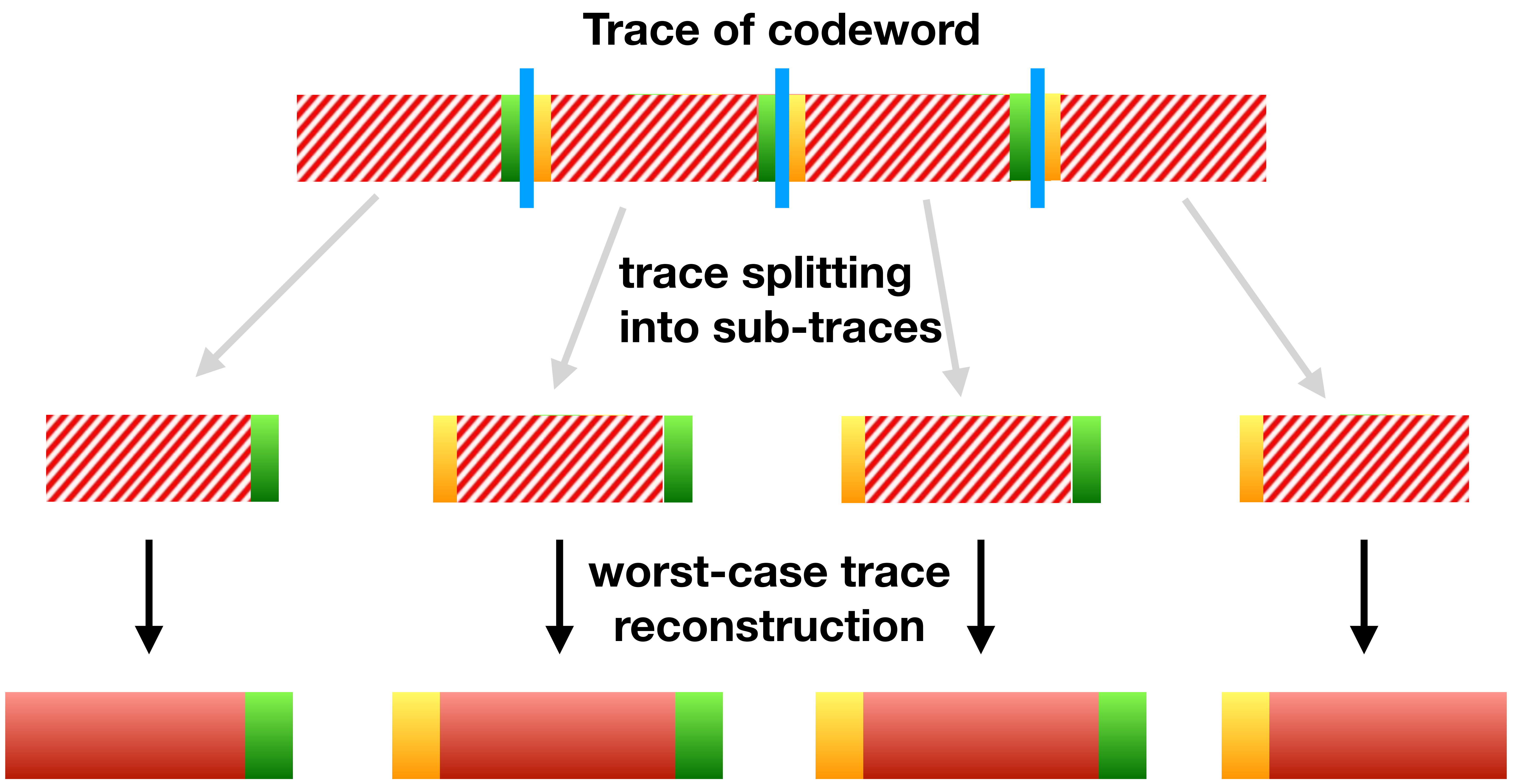
If $\mathscr{C}_b$ is chosen appropriately, long runs of 0's in trace **only** come from markers

Reconstruction

$= 0000\ldots00$ **O(log N) bits**

$= 1111\ldots11$ **O(log N) bits**

$\in \mathscr{C}_b$



Key observation: part of trace coming from a marker still looks like a marker.

If $\mathscr{C}_b$ is chosen appropriately, long runs of 0's in trace **only** come from markers

Efficiently encodable/reconstructable code with rate $1 - O(1/\log N)$ using $\exp(\log^{1/3+\gamma} N)$ traces for any constant deletion prob.

A tool for designing the sub-code: Almost k-wise independent strings

ϵ-almost k-wise independent distribution: $X \in \{0,1\}^m$

$$\forall i_1, \ldots, i_k, x_1, \ldots, x_k : |\Pr[X_{i_1} = x_1, \ldots, X_{i_k} = x_k] - 2^{-k}| \leq \epsilon$$

A tool for designing the sub-code: Almost k-wise independent strings

ϵ-almost k-wise independent distribution: $X \in \{0,1\}^m$

$$\forall i_1, \ldots, i_k, x_1, \ldots, x_k : |\Pr[X_{i_1} = x_1, \ldots, X_{i_k} = x_k] - 2^{-k}| \leq \epsilon$$

[AlonGoldreichHåstadPeralta92]:
"For decent parameters, can efficiently generate such strings from few uniformly random bits"

A tool for designing the sub-code: Almost k-wise independent strings

ϵ**-almost** k**-wise independent distribution:** $X \in \{0,1\}^m$

$$\forall i_1, \ldots, i_k, x_1, \ldots, x_k : |\Pr[X_{i_1} = x_1, \ldots, X_{i_k} = x_k] - 2^{-k}| \le \epsilon$$

[AlonGoldreichHåstadPeralta92]:
"For decent parameters, can efficiently generate such strings from few uniformly random bits"

For every m, $k = O(\log m)$ and $\epsilon = \frac{1}{\textbf{poly}(m)}$ there is a **poly**(m)-computable function

$$g : \{0,1\}^t \to \{0,1\}^m$$

with $t = O(\log m)$ such that $g(U_t)$ is ϵ-almost k-wise independent.

Designing the sub-code

Desired property: No long runs of 0's in trace of $c \in \mathscr{C}_b$ → allows for trace splitting

Designing the sub-code

Desired property: No long runs of 0's in trace of $c \in \mathcal{C}_b$ → allows for trace splitting

$g : \{0,1\}^t \to \{0,1\}^m$ generator from [AGHP92] with $t = O(\log m)$

$\implies$ $g(U_t)$ **satisfies property with high probability**

Designing the sub-code

Desired property: No long runs of 0's in trace of $c \in \mathscr{C}_b$ → allows for trace splitting

$g : \{0,1\}^t \to \{0,1\}^m$ generator from [AGHP92] with $t = O(\log m)$

$\Longrightarrow$ $g(U_t)$ **satisfies property with high probability**

To encode $x \in \{0,1\}^m$:

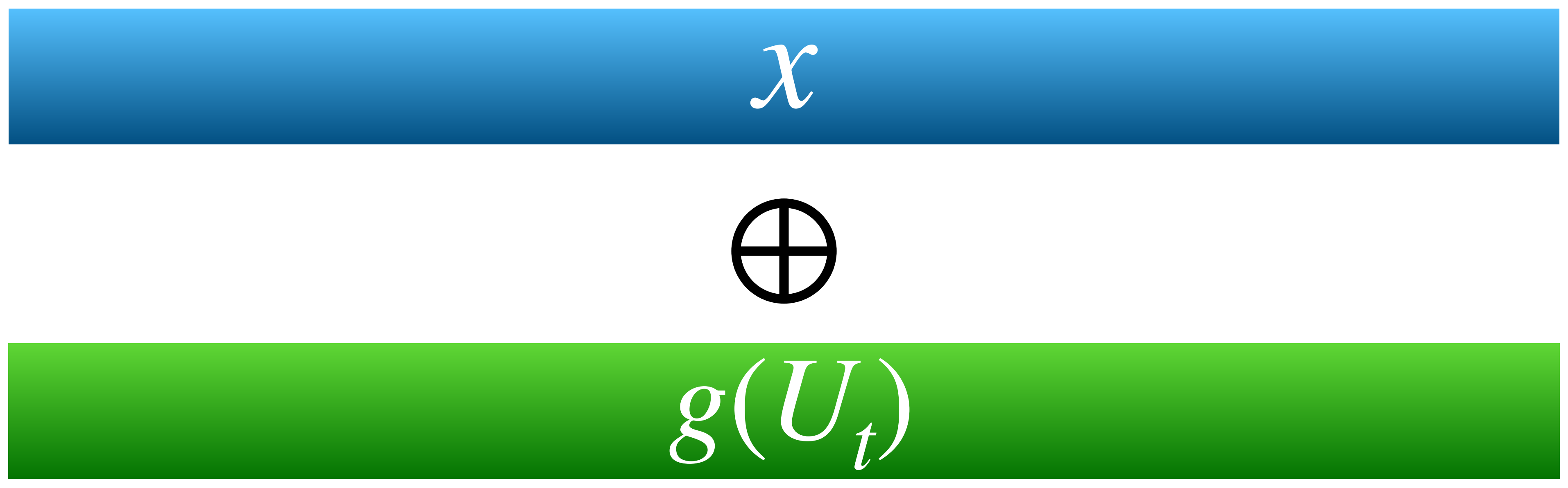
Designing the sub-code

Desired property: No long runs of 0's in trace of $c \in \mathscr{C}_b$ → allows for trace splitting

$g : \{0,1\}^t \to \{0,1\}^m$ generator from [AGHP92] with $t = O(\log m)$

$\Longrightarrow$ $g(U_t)$ **satisfies property with high probability**

To encode $x \in \{0,1\}^m$:



Designing the sub-code

Desired property: No long runs of 0's in trace of $c \in \mathscr{C}_b$ → allows for trace splitting

$g : \{0,1\}^t \rightarrow \{0,1\}^m$ generator from [AGHP92] with $t = O(\log m)$

$\Longrightarrow$ $g(U_t)$ **satisfies property with high probability**

To encode $x \in \{0,1\}^m$:

x satisfies property whp

$\oplus$

$g(U_t)$

Designing the sub-code

Desired property: No long runs of 0's in trace of $c \in \mathscr{C}_b$ → allows for trace splitting

$g : \{0,1\}^t \to \{0,1\}^m$ generator from [AGHP92] with $t = O(\log m)$

$\Longrightarrow$ $g(U_t)$ **satisfies property with high probability**

To encode $x \in \{0,1\}^m$**:**

x — satisfies property whp

$\oplus$

$g(z)$ — So there is **good** fixing $U_t = z$ that enforces property

Designing the sub-code

Desired property: No long runs of 0's in trace of $c \in \mathscr{C}_b$ → allows for trace splitting

$g : \{0,1\}^t \to \{0,1\}^m$ generator from [AGHP92] with $t = O(\log m)$

$\implies$ $g(U_t)$ **satisfies property with high probability**

To encode $x \in \{0,1\}^m$**:**

$$\text{Enc}(x) = \underbrace{z}_{\log m} \,\big|\, x \oplus g(z)$$

(z chosen so that property is satisfied)

PART II:

Markers +

modified average-case trace reconstruction

Subsequence-unique strings

[HolensteinMitzenmacherPanigrahyWieder08]

A string x is w**-subsequence-unique** if no substring of length w can be obtained by deleting bits of another substring of length $1.1w$ (except for trivial containment).

Subsequence-unique strings

[HolensteinMitzenmacherPanigrahyWieder08]

A string x is w**-subsequence-unique** if no substring of length w can be obtained by deleting bits of another substring of length $1.1w$ (except for trivial containment).

Deletion probability small enough constant $\Longrightarrow$ Efficient trace reconstruction algorithm for all length m w-subsequence-unique strings with $w = O(\log m)$ using **poly**(m) traces.

Subsequence-unique strings

[HolensteinMitzenmacherPanigrahyWieder08]

A string x is w**-subsequence-unique** if no substring of length w can be obtained by deleting bits of another substring of length $1.1w$ (except for trivial containment).

Deletion probability small enough constant $\Longrightarrow$ Efficient trace reconstruction algorithm for all length m w-subsequence-unique strings with $w = O(\log m)$ using **poly**(m) traces.

Key observation: an ϵ-almost k-wise independent string is w-subsequence-unique with high probability for decent parameters k and ϵ.

Better codes for small deletion probability

Use masking to encode blocks into "almost" subsequence-unique strings

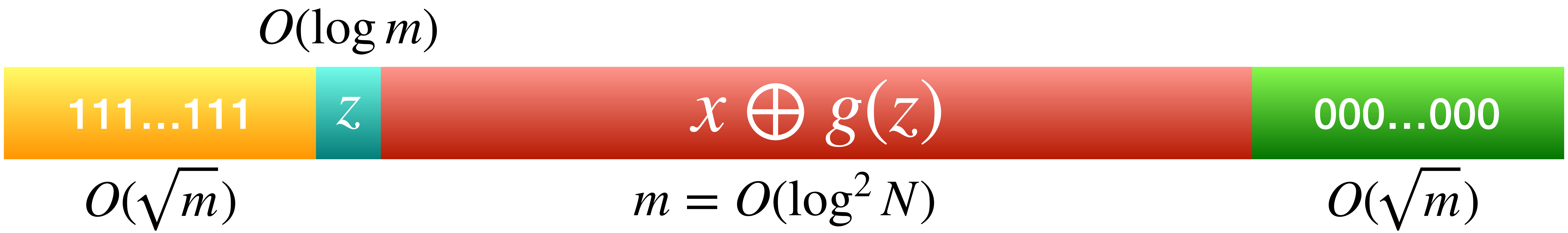
$x \in \{0,1\}^m$ $\xrightarrow{\text{encoded as}}$ z | $x \oplus g(z)$

$\log m$ w-subsequence-unique

$w = O(\log m)$

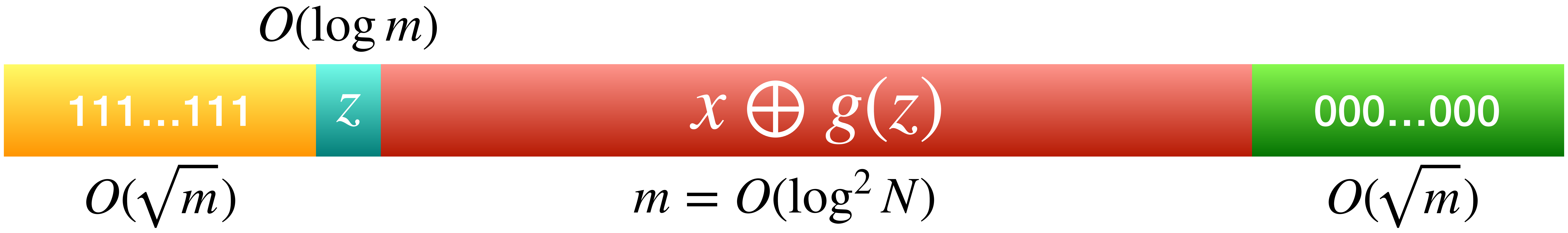
Better codes for small deletion probability

Use masking to encode blocks into “almost” subsequence-unique strings



HMPW trace reconstruction algorithm doesn’t work for strings like this…

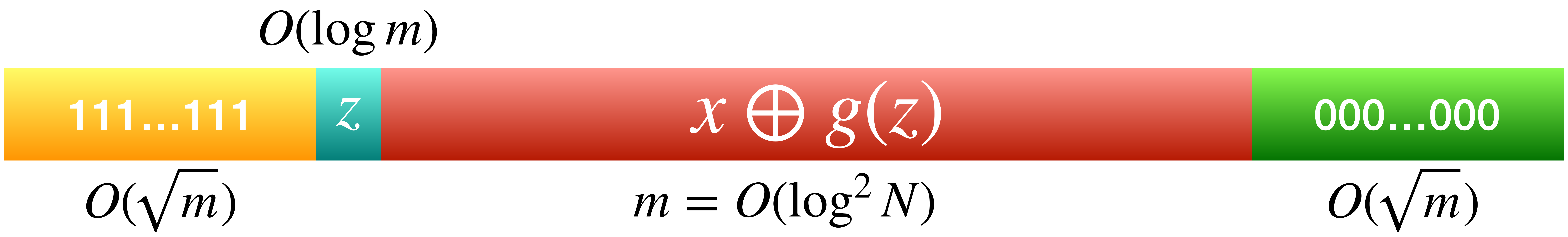
Better codes for small deletion probability



First barrier: HMPW algorithm needs to know first $\log m$ bits of $x \oplus g(z)$ "for free"

Original bootstrapping procedure requires $2^{\sqrt{m}} =$ **poly**(N) traces in this case…

Better codes for small deletion probability



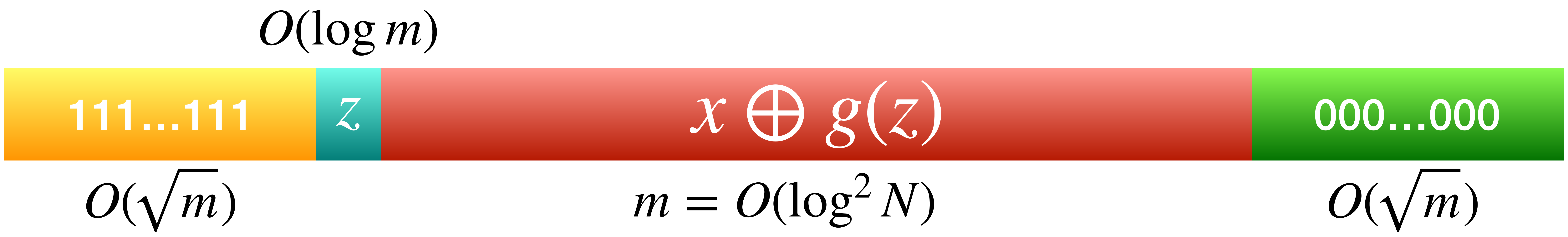
First barrier: HMPW algorithm needs to know first $\log m$ bits of $x \oplus g(z)$ "for free"

Original bootstrapping procedure requires $2^{\sqrt{m}} = \textbf{poly}(N)$ traces in this case…

Exploit properties of masking:

$$x' = \color{red}{00\ldots00}\,||\,x \xrightarrow[\textbf{encoded as}]{} z\,||\,(\color{red}{x'} \oplus g(z))$$

Better codes for small deletion probability



First barrier: HMPW algorithm needs to know first $\log m$ bits of $x \oplus g(z)$ "for free"

Original bootstrapping procedure requires $2^{\sqrt{m}} = \textbf{poly}(N)$ traces in this case…

Exploit properties of masking:

$x' = 00\ldots00 \,||\, x \xrightarrow{} z \,||\, (x' \oplus g(z))$

encoded as

$z' =$ systematic encoding of $0 \,||\, z$ robust against constant fraction of deletions+insertions

Better codes for small deletion probability

$O(\log m)$

111…111	z	$x \oplus g(z)$	000…000
$O(\sqrt{m})$		$m = O(\log^2 N)$	$O(\sqrt{m})$

First barrier: HMPW algorithm needs to know first $\log m$ bits of $x \oplus g(z)$ "for free"

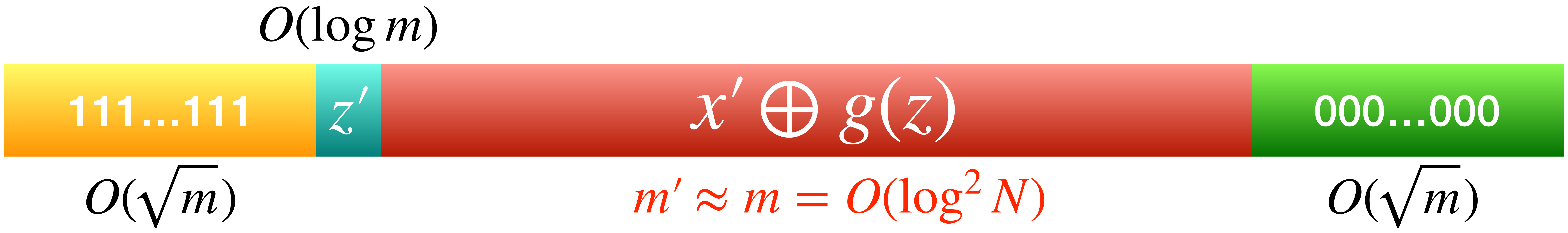
Original bootstrapping procedure requires $2^{\sqrt{m}} = \mathbf{poly}(N)$ traces in this case…

Exploit properties of masking:

$$x' = 00\ldots00 \,||\, x \xrightarrow{\text{encoded as}} z' \,||\, (x' \oplus g(z))$$

$z' =$ systematic encoding of $0 \,||\, z$ robust against constant fraction of deletions+insertions

Better codes for small deletion probability



First barrier: HMPW algorithm needs to know first $\log m$ bits of $x \oplus g(z)$ “for free”

Original bootstrapping procedure requires $2^{\sqrt{m}} = \textbf{poly}(N)$ traces in this case…

Exploit properties of masking:

$x' = 00\ldots00 \,||\, x \xrightarrow[\textbf{encoded as}]{} z' \,||\, (x' \oplus g(z))$

$z' =$ systematic encoding of $0 \,||\, z$ robust against constant fraction of deletions+insertions

Better codes for small deletion probability

$O(\log m)$

111…111	z'	$x' \oplus g(z)$	000…000
$O(\sqrt{m})$		$m' \approx m = O(\log^2 N)$	$O(\sqrt{m})$

First barrier: HMPW algorithm needs to know first $\log m$ bits of $x \oplus g(z)$ "for free"

Original bootstrapping procedure requires $2^{\sqrt{m}} =$ **poly**(N) traces in this case…

Exploit properties of masking:

$x' = 00\ldots00 \,||\, x \xrightarrow{\text{encoded as}} z' \,||\, (x' \oplus g(z))$

$z' =$ systematic encoding of $0 \,||\, z$ robust against constant fraction of deletions+insertions

Bootstrapping only requires $O(m)$ traces now!

Better codes for small deletion probability

A few more things:

- Actually need a stronger property than subsequence-uniqueness because of markers;
- Need to make sure trace splitting works.

Almost k-wise independent string satisfies both of them with high probability.

There is an absolute constant $d^\star \in (0,1)$ such that for all $d \leq d^\star$ there exists an efficiently encodable/reconstructable code with rate $1 - O(1/\log N)$ using **polylog**(N) traces against i.i.d. deletions with probability d.

Coded trace reconstruction over large alphabets

[BrakensiekLiSpang19]

There exist efficiently encodable/reconstructable codes with rate $1-\epsilon$ over an alphabet of size $O_\epsilon(1)$ using $O(\log_{1/d}(1/\epsilon))$ traces.

Moreover, **this is tight.**

Coded trace reconstruction over large alphabets

[BrakensiekLiSpang19]

There exist efficiently encodable/reconstructable codes with rate $1-\epsilon$ over an alphabet of size $O_\epsilon(1)$ using $O(\log_{1/d}(1/\epsilon))$ traces.

Moreover, **this is tight.**

Lower bound:

Capacity of T-use deletion channel $\leq$ Capacity of T-use erasure channel $= 1-d^T$

Coded trace reconstruction over large alphabets

[BrakensiekLiSpang19]

There exist efficiently encodable/reconstructable codes with rate $1-\epsilon$ over an alphabet of size $O_\epsilon(1)$ using $O(\log_{1/d}(1/\epsilon))$ traces.

Moreover, **this is tight.**

Lower bound:

Capacity of T-use deletion channel $\leq$ Capacity of T-use erasure channel $= 1-d^T$

$$\implies 1-\epsilon \leq 1-d^T \implies T \geq \log_{1/d}(1/\epsilon)$$

A tool for the upper bound: Synchronization strings

[HaeuplerShahrasbi17]

A string S is a τ-**synchronization string** if for all $i < j < k$

$$ED(S_{[i,j)}, S_{[j,k)}) > (1 - \tau)(k - i)$$

A tool for the upper bound: Synchronization strings

[HaeuplerShahrasbi17]

A string S is a τ-**synchronization string** if for all $i < j < k$

$$ED(S_{[i,j)}, S_{[j,k)}) > (1 - \tau)(k - i)$$

"close" to maximal edit distance everywhere

A tool for the upper bound: Synchronization strings

[HaeuplerShahrasbi17]

A string S is a τ-**synchronization string** if for all $i < j < k$

$$ED(S_{[i,j)}, S_{[j,k)}) > (1 - \tau)(k - i)$$

“close” to maximal edit distance everywhere

For every τ, can efficiently construct a τ-synch-string of length N over an alphabet of size **poly**$(1/\tau)$.

Why are synchronization strings useful?

$$c = (c_1, c_2, \ldots, c_N) \in \Sigma^N \xrightarrow[]{\textbf{naive indexing}} ((c_1,1), (c_2,2), \ldots, (c_N, N)) \in (\Sigma \times [N])^N$$

Why are synchronization strings useful?

$$c = (c_1, c_2, \ldots, c_N) \in \Sigma^N \xrightarrow{\text{naive indexing}} ((c_1,1), (c_2,2), \ldots, (c_N, N)) \in (\Sigma \times [N])^N$$

$$(c_1,1) \quad (c_2,2) \quad (c_3,3) \quad (c_4,4)$$

$$\downarrow$$

$$(c_1,1) \quad \color{red}{(c_2,2)} \quad (c_3,3) \quad \color{red}{(c_4,4)}$$

$$\Big\downarrow \ \color{red}{\textbf{deletions}}$$

$$(c_1,1) \quad (c_3,3)$$

Why are synchronization strings useful?

$$c = (c_1, c_2, \ldots, c_N) \in \Sigma^N \xrightarrow{\text{naive indexing}} ((c_1,1), (c_2,2), \ldots, (c_N, N)) \in (\Sigma \times [N])^N$$

$$(c_1,1) \quad (c_2,2) \quad (c_3,3) \quad (c_4,4)$$

$$\downarrow$$

$$(c_1,1) \quad (c_2,2) \quad (c_3,3) \quad (c_4,4)$$

$$\downarrow \text{ deletions}$$

$$(c_1,1) \quad (c_3,3)$$

equivalent to

$$c_1 \quad c_2 \quad c_3 \quad c_4$$

$$\downarrow$$

$$c_1 \quad c_2 \quad c_3 \quad c_4$$

$$\downarrow \text{ erasures}$$

$$c_1 \quad ? \quad c_3 \quad ?$$

Why are synchronization strings useful?

[HaeuplerShahrasbi17]: *"Can efficiently transform deletions into (worst-case) erasures by indexing with synchronization string, with* ***constant*** *alphabet size blowup"*

$$S \in \Sigma^N \xrightarrow{dN \text{ deletions}} S' \in \Sigma^{(1-d)N}$$

τ-synchronization string

Why are synchronization strings useful?

[HaeuplerShahrasbi17]: *"Can efficiently transform deletions into (worst-case) erasures by indexing with synchronization string, with **constant** alphabet size blowup"*

$$S \in \Sigma^N \xrightarrow{dN \text{ deletions}} S' \in \Sigma^{(1-d)N}$$

τ-synchronization string

"Error-free" indexing algorithm:

S'_j →

- find correct index i_j such that S_{i_j} is mapped to S'_j by deletion process
- abort and output special symbol $?$ **(misdecoding)**

Why are synchronization strings useful?

[HaeuplerShahrasbi17]: *“Can efficiently transform deletions into (worst-case) erasures by indexing with synchronization string, with* ***constant*** *alphabet size blowup”*

$$S \in \Sigma^N \xrightarrow{dN \text{ deletions}} S' \in \Sigma^{(1-d)N}$$

τ-synchronization string

“Error-free” indexing algorithm:

S'_j → find correct index i_j such that S_{i_j} is mapped to S'_j by deletion process

S'_j → abort and output special symbol $?$ **(misdecoding)**

[HaeuplerShahrasbi17]:

There is efficient error-free indexing algorithm with at most

$$\frac{\tau dN}{1-\tau}$$

misdecodings.

Coded trace reconstruction from synchronization strings

[BrakensiekLiSpang19]: Efficient code over alphabet of size $O_\epsilon(1)$ using $O(\log_{1/d}(1/\epsilon))$ traces

S $(\tau = \epsilon^C)$ -synchronization string over alphabet of size $O_\epsilon(1)$

$\mathscr{C}$ rate $\approx 1 - \epsilon$ code robust against $\approx \epsilon^3$ fraction of erasures

Construction: *Index codewords of* $\mathscr{C}$ *with* S

Coded trace reconstruction from synchronization strings

[BrakensiekLiSpang19]: Efficient code over alphabet of size $O_\epsilon(1)$ using $O(\log_{1/d}(1/\epsilon))$ traces

S $(\tau = \epsilon^C)$ -synchronization string over alphabet of size $O_\epsilon(1)$

$\mathscr{C}$ rate $\approx 1 - \epsilon$ code robust against $\approx \epsilon^3$ fraction of erasures

Construction: *Index codewords of* $\mathscr{C}$ *with* S

$(c_1, \ldots, c_N) \in \mathscr{C} \longrightarrow \boxed{(c_1, S_1)} \ \boxed{(c_2, S_2)} \ \cdots \ \boxed{(c_N, S_N)}$

Coded trace reconstruction from synchronization strings

[BrakensiekLiSpang19]: Efficient code over alphabet of size $O_\epsilon(1)$ using $O(\log_{1/d}(1/\epsilon))$ traces

Reconstruction:

Trace of codeword	(x_{11}, x_{12})	(x_{21}, x_{22})	(x_{31}, x_{32})	(x_{41}, x_{42})	$\cdots$

Coded trace reconstruction from synchronization strings

[BrakensiekLiSpang19]: Efficient code over alphabet of size $O_\epsilon(1)$ using $O(\log_{1/d}(1/\epsilon))$ traces

Reconstruction:

symbols from synch string

Trace of codeword (x_{11}, x_{12}) (x_{21}, x_{22}) (x_{31}, x_{32}) (x_{41}, x_{42}) $\cdots$

Coded trace reconstruction from synchronization strings

[BrakensiekLiSpang19]: Efficient code over alphabet of size $O_\epsilon(1)$ using $O(\log_{1/d}(1/\epsilon))$ traces

Reconstruction:

symbols from synch string

Trace of codeword (x_{11}, x_{12}) (x_{21}, x_{22}) (x_{31}, x_{32}) (x_{41}, x_{42}) $\cdots$

run error-free indexing algorithm on second symbol

i_1 ? i_3 ?

Coded trace reconstruction from synchronization strings

[BrakensiekLiSpang19]: Efficient code over alphabet of size $O_\epsilon(1)$ using $O(\log_{1/d}(1/\epsilon))$ traces

Reconstruction:

symbols from synch string

Trace of codeword (x_{11}, x_{12}) (x_{21}, x_{22}) (x_{31}, x_{32}) (x_{41}, x_{42}) $\cdots$

run error-free indexing algorithm on second symbol

$\boxed{i_1}$ $?$ $\boxed{i_3}$ $?$

recover $c_{i_1} = x_{11}$

treat as erasure

recover $c_{i_3} = x_{31}$

treat as erasure

Coded trace reconstruction from synchronization strings

[BrakensiekLiSpang19]: Efficient code over alphabet of size $O_\epsilon(1)$ using $O(\log_{1/d}(1/\epsilon))$ traces

Reconstruction:

symbols from synch string

Trace of codeword (x_{11}, x_{12}) (x_{21}, x_{22}) (x_{31}, x_{32}) (x_{41}, x_{42}) $\cdots$

run error-free indexing algorithm on second symbol

$\boxed{i_1}$ $?$ $\boxed{i_3}$ $?$

recover $c_{i_1} = x_{11}$

treat as erasure

recover $c_{i_3} = x_{31}$

treat as erasure

#erasures $\leq$ #misdecodings + #(symbols deleted in every trace) $\leq \dfrac{\epsilon^3 N}{2} + d^T N \leq \epsilon^3 N$

Coded trace reconstruction from synchronization strings

[BrakensiekLiSpang19]: Efficient code over alphabet of size $O_\epsilon(1)$ using $O(\log_{1/d}(1/\epsilon))$ traces

Reconstruction:

symbols from synch string

Trace of codeword (x_{11}, x_{12}) (x_{21}, x_{22}) (x_{31}, x_{32}) (x_{41}, x_{42}) $\cdots$

run error-free indexing algorithm on second symbol

$\boxed{i_1}$ $?$ $\boxed{i_3}$ $?$

recover $c_{i_1} = x_{11}$

treat as erasure

recover $c_{i_3} = x_{31}$

treat as erasure

$\mathscr{C}$ **corrects this many erasures!**

#erasures $\leq$ #misdecodings + #(symbols deleted in every trace) $\leq \dfrac{\epsilon^3 N}{2} + d^T N \leq \epsilon^3 N$

From large alphabets to binary codes

Construction: *Concatenate good code over large alphabet with marker-based inner code*

x N**-bit string**

From large alphabets to binary codes

Construction: *Concatenate good code over large alphabet with marker-based inner code*

x **N-bit string**

$\mathscr{C}_{\text{out}}$ **large alphabet code correcting many substitutions**

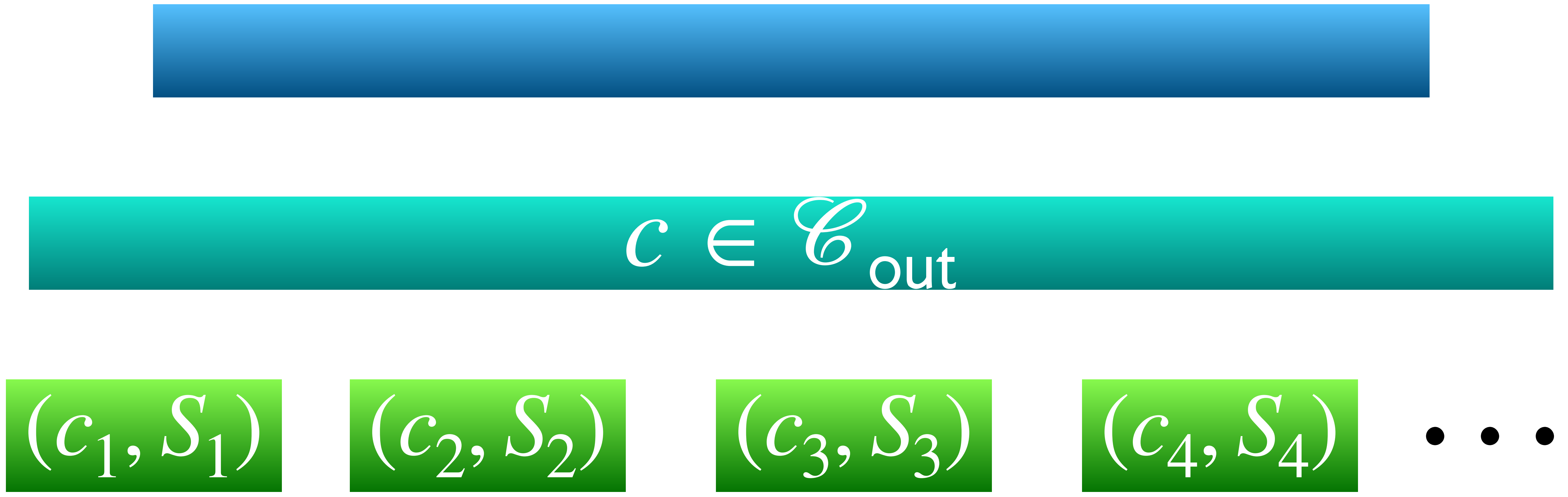
From large alphabets to binary codes

Construction: *Concatenate good code over large alphabet with marker-based inner code*

x N**-bit string**

$\mathscr{C}_{\text{out}}$ **large alphabet code correcting many substitutions**

S $1/3$**-synchronization string over alphabet of size** $O(1)$



From large alphabets to binary codes

Construction: *Concatenate good code over large alphabet with marker-based inner code*

x N-bit string

$\mathscr{C}_{\text{out}}$ large alphabet code correcting many substitutions

$c \in \mathscr{C}_{\text{out}}$

S $1/3$-synchronization string over alphabet of size $O(1)$

(c_1, S_1) (c_2, S_2) (c_3, S_3) (c_4, S_4) $\cdots$

---

concatenate with binary code

(c_i, S_i)

From large alphabets to binary codes

Construction: *Concatenate good code over large alphabet with marker-based inner code*

x $\quad N$**-bit string**

$\mathscr{C}_{\text{out}}$ **large alphabet code correcting many substitutions** $\quad c \in \mathscr{C}_{\text{out}}$

S $\quad 1/3$**-synchronization string over alphabet of size** $O(1)$

(c_1, S_1) (c_2, S_2) (c_3, S_3) (c_4, S_4) $\cdots$

---

concatenate with binary code

(c_i, S_i) $\rightarrow$ $\mathrm{Enc}_R(c_i) = $ $\underbrace{0}_{\log m}\ \underbrace{\overline{c_i}}_{m}\ \underbrace{1}_{\log m}$

From large alphabets to binary codes

Construction: *Concatenate good code over large alphabet with marker-based inner code*

x — **N-bit string**

$\mathscr{C}_{\text{out}}$ — **large alphabet code correcting many substitutions** — $c \in \mathscr{C}_{\text{out}}$

S — **$1/3$-synchronization string over alphabet of size $O(1)$** — (c_1, S_1) (c_2, S_2) (c_3, S_3) (c_4, S_4) $\cdots$

---

concatenate with binary code

(c_i, S_i)

$\text{Enc}_R(c_i) =$ | 0 | $\overline{c_i}$ | 1 |

$\log m$ $\quad m \quad$ $\log m$

$\overline{c_i}$ *"Dense" codeword TBD*

From large alphabets to binary codes

Construction: *Concatenate good code over large alphabet with marker-based inner code*

x — N-**bit string**

$\mathscr{C}_{\text{out}}$ — **large alphabet code correcting many substitutions** — $c \in \mathscr{C}_{\text{out}}$

S — $1/3$-**synchronization string over alphabet of size** $O(1)$ — (c_1, S_1) (c_2, S_2) (c_3, S_3) (c_4, S_4) $\cdots$

concatenate with binary code

(c_i, S_i)

$\mathrm{Enc}_R(c_i) = \underbrace{0}_{\log m}\ \underbrace{\overline{c_i}}_{m}\ \underbrace{1}_{\log m}$

$\overline{c_i}$ *"Dense" codeword TBD*

$\mathrm{Enc}_S(S_i) = 0\ 1\ 0\ 1\ \cdots$

$\log m$

From large alphabets to binary codes

Construction: *Concatenate good code over large alphabet with marker-based inner code*

x — N**-bit string**

$\mathscr{C}_{\text{out}}$ — **large alphabet code correcting many substitutions** — $c \in \mathscr{C}_{\text{out}}$

S — $1/3$**-synchronization string over alphabet of size** $O(1)$ — (c_1, S_1) (c_2, S_2) (c_3, S_3) (c_4, S_4) $\cdots$

---

concatenate with binary code

(c_i, S_i)

$\text{Enc}_R(c_i) =$ 0 $\overline{c_i}$ 1 ($\log m$, m, $\log m$)

$\overline{c_i}$ *"Dense" codeword TBD*

$\text{Enc}_S(S_i) =$ 0 1 0 1 $\cdots$ ($\log m$)

Robust against deletions, with two possible runlengths, same number of runs

Reconstruction

Reconstruction

Reconstruction

1) Trace alignment:

Long runs **only come from markers** $\Longrightarrow$ Find traces of (most) synch symbols

Reconstruction

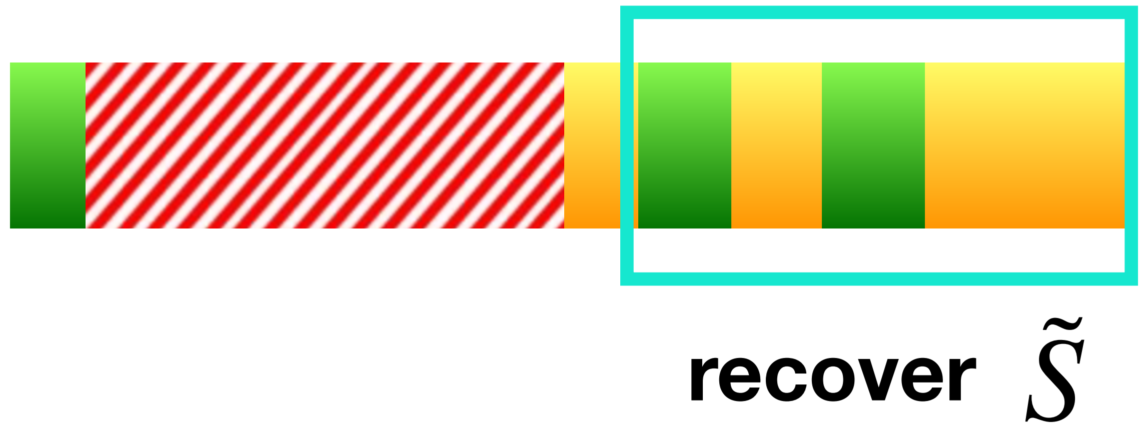
1) Trace alignment:

Long runs **only come from markers** $\Longrightarrow$ Find traces of (most) synch symbols

Reconstruction

1) Trace alignment:

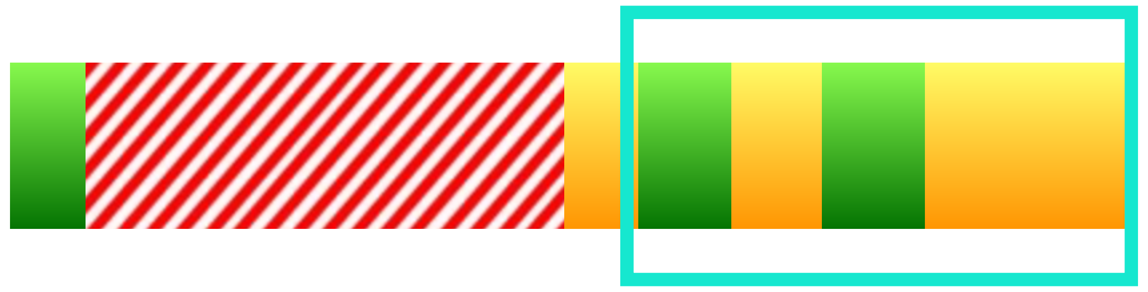
Long runs **only come from markers** $\Longrightarrow$ Find traces of (most) synch symbols



Reconstruction

1) Trace alignment:

Long runs **only come from markers** $\Longrightarrow$ Find traces of (most) synch symbols

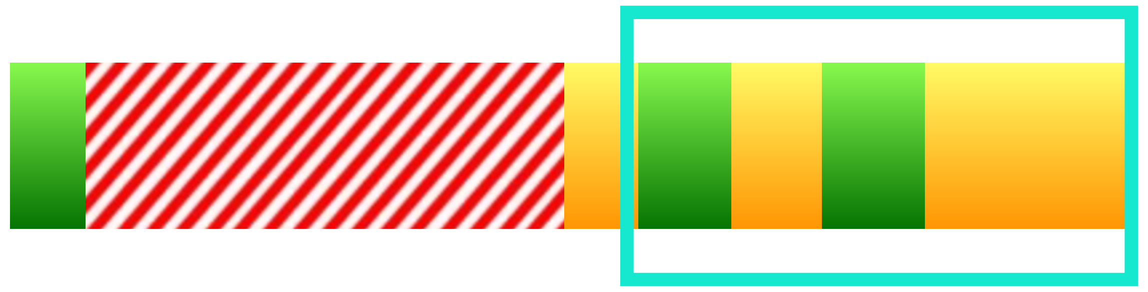


recover $\tilde{S}$ $\longrightarrow$ **run error-free indexing algo**

Reconstruction

1) Trace alignment:

Long runs **only come from markers** $\Longrightarrow$ Find traces of (most) synch symbols

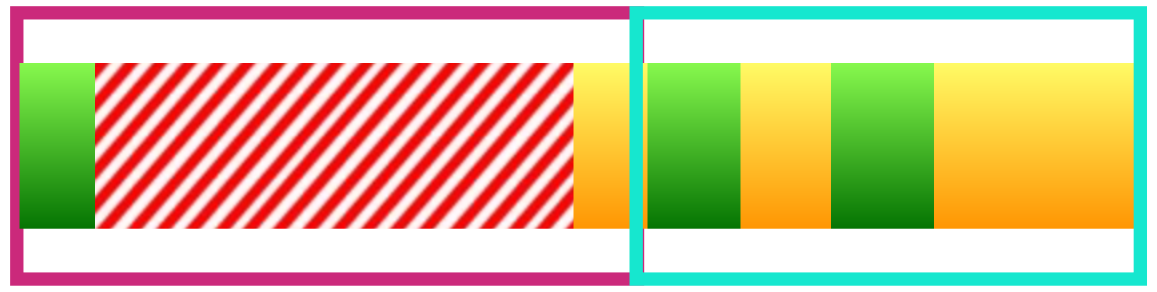


recover $\tilde{S}$ $\longrightarrow$ **run error-free indexing algo** $\longrightarrow$ **Recover index** i

Reconstruction

1) Trace alignment:

Long runs **only come from markers** $\Longrightarrow$ Find traces of (most) synch symbols

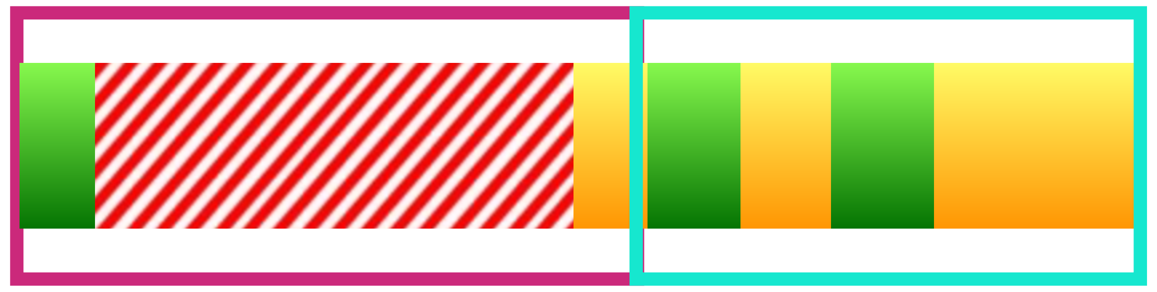


trace of c_i **whp** — **recover** $\tilde{S}$ $\longrightarrow$ **run error-free indexing algo** $\longrightarrow$ **Recover index** i

Reconstruction

1) Trace alignment:

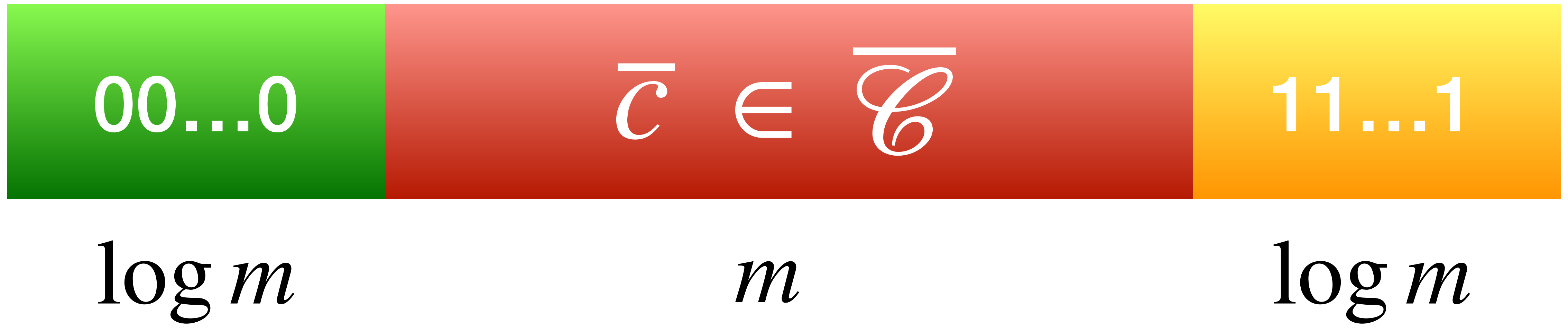
Long runs **only come from markers** $\Longrightarrow$ Find traces of (most) synch symbols



trace of c_i whp | **recover $\tilde{S}$** $\longrightarrow$ **run error-free indexing algo** $\longrightarrow$ **Recover index i**

2) Use special trace rec properties of $\overline{\mathscr{C}}$ + error-correction properties of $\mathscr{C}_{\text{out}}$

Designing the (existential) inner code



Key observation: Since markers are short, **nearly all strings of this type are good for any average-case trace rec algorithm!**

Designing the (existential) inner code

00…0	$\bar{c} \in \overline{\mathscr{C}}$	11…1
$\log m$	m	$\log m$

Key observation: Since markers are short, **nearly all strings of this type are good for any average-case trace rec algorithm!**

$\Longrightarrow$ If m is **small**, can **brute force** high-rate code $\overline{\mathscr{C}}$ to require **few traces + be dense**!

Designing the (existential) inner code

00…0	$\bar{c} \in \overline{\mathscr{C}}$	11…1
$\log m$	m	$\log m$

Key observation: Since markers are short, **nearly all strings of this type are good for any average-case trace rec algorithm!**

$\Longrightarrow$ If m is **small**, can **brute force** high-rate code $\overline{\mathscr{C}}$ to require **few traces + be dense**!

$\Longrightarrow$ Using [HPP18], reconstruct $\overline{\mathscr{C}}$ with $\exp(\log^{1/3} m)$ traces.

Combining everything

Set inner code length to $m = \frac{1}{\epsilon} \log\left(\frac{1}{\epsilon}\right)$ to ensure rate $\geq 1 - \epsilon$

[BrakensiekLiSpang19]

For any constant deletion probability and every ϵ, there exists a code of rate $1 - \epsilon$ that can be reconstructed from $\exp(\log^{1/3}(1/\epsilon))$ traces.

Combining everything

Set inner code length to $m = \frac{1}{\epsilon} \log\left(\frac{1}{\epsilon}\right)$ to ensure rate $\geq 1 - \epsilon$

[BrakensiekLiSpang19]

For any constant deletion probability and every ϵ , there exists a code of rate $1 - \epsilon$ that can be reconstructed from $\exp(\log^{1/3}(1/\epsilon))$ traces.

Caveat: Inner code construction takes time $\exp(m)$.

$\Longrightarrow$ Overall code construction only efficient for $\epsilon \geq \frac{\log \log N}{\log N}$

Summing up…

We can exploit worst-case and average-case trace reconstruction to design efficient high-rate codes requiring significantly fewer traces, or satisfying other nice properties.

	rate = 1 - ε	#traces	efficient encoding & reconstruction	observations
Markers + modified avg-case trace rec [CGMR19]	$\epsilon = \frac{1}{\log N}$	$\log^C N$	✓	Deletion probability smaller than absolute constant
Synch string + Markers + existential trace rec [BLS19]	$\epsilon = \frac{1}{\log N}$	$\exp((\log\log N)^{1/3})$	✓	Code construction is not efficient
Synch string + Markers + existential trace rec [BLS19]	$\epsilon \geq \frac{\log\log N}{\log N}$	$\exp(\log^{1/3}(1/\epsilon))$	✓	Arbitrary constant deletion probability

Summing up…

We can exploit worst-case and average-case trace reconstruction to design efficient high-rate codes requiring significantly fewer traces, or satisfying other nice properties.

	rate = 1 - ε	#traces	efficient encoding & reconstruction	observations
Markers + modified avg-case trace rec [CGMR19]	$\epsilon = \frac{1}{\log N}$	$\log^C N$	✓	Deletion probability smaller than absolute constant
Synch string + Markers + existential trace rec [BLS19]	$\epsilon = \frac{1}{\log N}$	$\exp((\log\log N)^{1/3})$	✓	Code construction is not efficient
Synch string + Markers + existential trace rec [BLS19]	$\epsilon \geq \frac{\log\log N}{\log N}$	$\exp(\log^{1/3}(1/\epsilon))$	✓	Arbitrary constant deletion probability

Lower bound [BLS19]: Arbitrary code of rate $1-\epsilon$ $\Longrightarrow$ requires $\approx \log^{5/2}\left(\frac{1}{\epsilon}\right)$ traces

Future work

- Efficient high-rate codes using even fewer traces;
- Bridge gap between bounds for (coded and uncoded) trace reconstruction;
- High-rate codes that handle deletions **and** random insertions with few traces;

Future work

- Efficient high-rate codes using even fewer traces;
- Bridge gap between bounds for (coded and uncoded) trace reconstruction;
- High-rate codes that handle deletions **and** random insertions with few traces;

Thanks!