CS11921 - **Fall** 2024

Algorithm Design & Analysis

Cardinality Estimation

Ibrahim Albluwi

How Many Did I See?

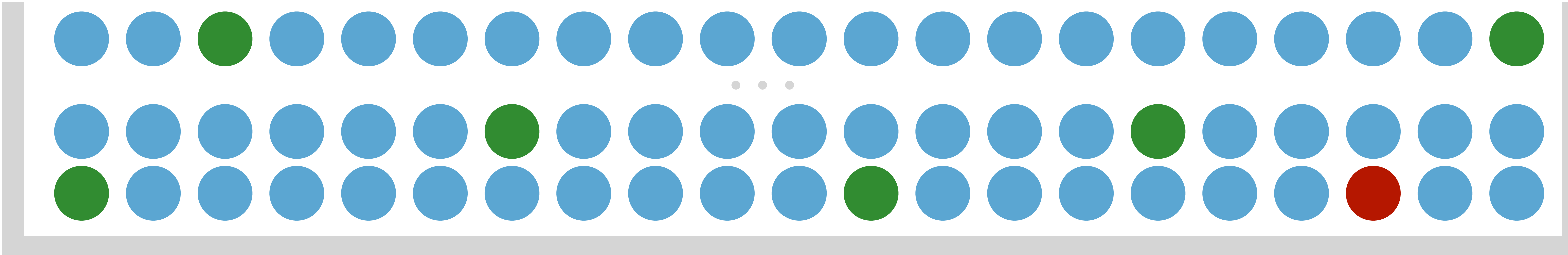
Problem. Count the number of *unique* elements in a data stream.
Requirement. Use $O(\log n)$ space, where n is the number of elements in the stream.
Assumption. An approximate count is acceptable.

Scenario. How many unique viewers for each post?

Naive Solution. Use a *set* for each page storing visitor IDs. The size of the set is the number of unique visitors (the set can be implemented as a hash table or binary search tree).

Uses $\Theta(n)$ memory for each page.

A Detour: How many balls did we see?



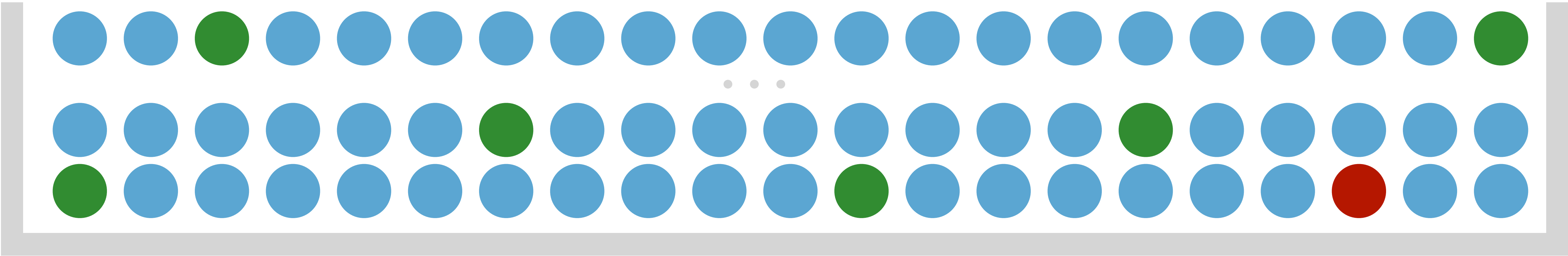
	Probability of being drawn randomly
1000 blue balls	$1000 \,/\, (1000 + 100 + 10) \approx 90\,\%$
100 green balls	$100 \,/\, (1000 + 100 + 10) \approx 9\,\%$
10 red balls	$1 \,/\, (1000 + 100 + 10) \approx 1\,\%$

Question. How many times do we expect to draw from the box before we see each color?

Blue. $\frac{1}{0.9} = 1.11$ **Green.** $\frac{1}{0.09} = 11.11$ **Red.** $\frac{1}{0.01} = 100$

> Let V be an event that occurs in a trial with probability p. The expected number of trials to first occurrence of V in a sequence of independent trials is $1/p$.

A Detour: How many balls did we see?



	Probability of being drawn randomly
1000 blue balls	$1000 \,/\, (1000 + 100 + 10) \approx 90\,\%$
100 green balls	$100 \,/\, (1000 + 100 + 10) \approx 9\,\%$
10 red balls	$1 \,/\, (1000 + 100 + 10) \approx 1\,\%$

Question. How many times do we expect to draw from the box before we see each color?

Blue. $\frac{1}{0.9} = 1.11$

Green. $\frac{1}{0.09} = 11.11$

Red. $\frac{1}{0.01} = 100$

If we see a blue ball, how many balls did we likely draw from the box?

Answer. 1 – 2.

If we see a green ball, how many balls did we likely draw from the box?

Answer. Around 11.

If we see a red ball, how many balls did we likely draw from the box?

Answer. 100.

Solution # 1: Probabilistic Counting

Problem. Count the number of *unique* elements in a data stream.
Requirement. Use $O(\log n)$ space, where n is the number of elements in the stream.
Assumption. An approximate count is acceptable.

Algorithm.

```
m = 0
FOREACH x in the stream:
    h = HASH(x)
    c = COUNT-TRAILING-ONES(h)
    m = MAX(m, c)

RETURN 2^m
```

Filip Flajolet (1983)

Solution # 1: Probabilistic Counting

Problem. Count the number of *unique* elements in a data stream.
Requirement. Use $O(\log n)$ space, where n is the number of elements in the stream.
Assumption. An approximate count is acceptable.

Algorithm.

```
m = 0
FOREACH x in the stream:
   h = HASH(x)
   c = COUNT-TRAILING-ONES(h)
   m = MAX(m, c)

RETURN 2^m
```

convert x to a bit string

E.g. `010100...101100101111`

Filip Flajolet (1983)

Solution # 1: Probabilistic Counting

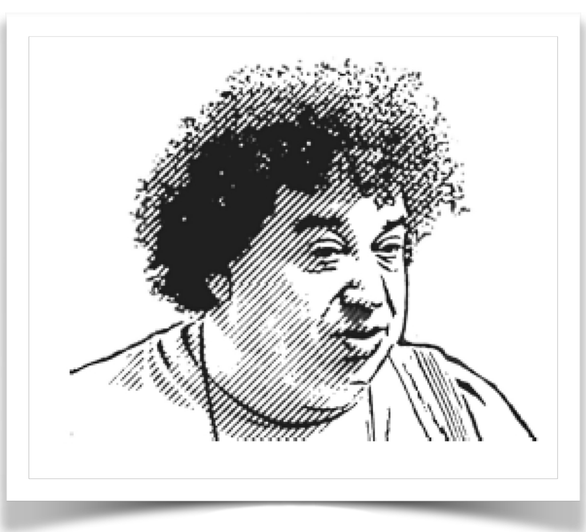
Problem. Count the number of *unique* elements in a data stream.
Requirement. Use $O(\log n)$ space, where n is the number of elements in the stream.
Assumption. An approximate count is acceptable.

Algorithm.

```
m = 0
FOREACH x in the stream:
    h = HASH(x)
    c = COUNT-TRAILING-ONES(h)
    m = MAX(m, c)

RETURN 2^m
```

number of trailing 1s = 4

E.g. `010100...10110010`**`1111`**



Filip Flajolet (1983)

Solution # 1: Probabilistic Counting

Problem. Count the number of *unique* elements in a data stream.
Requirement. Use $O(\log n)$ space, where n is the number of elements in the stream.
Assumption. An approximate count is acceptable.

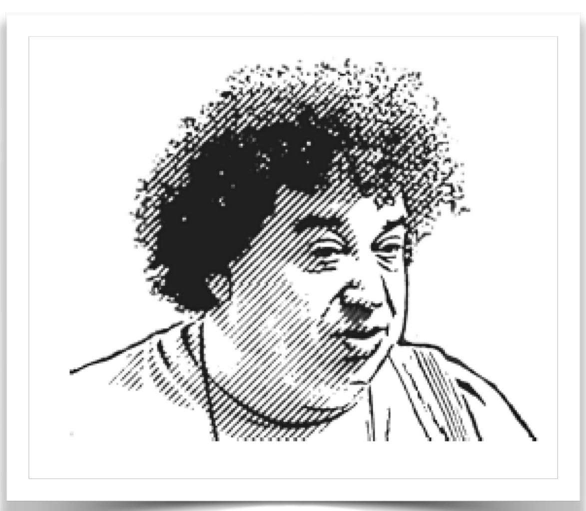
Algorithm.

```
m = 0
FOREACH x in the stream:
    h = HASH(x)
    c = COUNT-TRAILING-ONES(h)
    m = MAX(m, c)

RETURN 2^m
```

number of trailing 1s = 4

E.g. `010100...10110010`**`1111`**

update the maximum number of trailing 1s seen so far



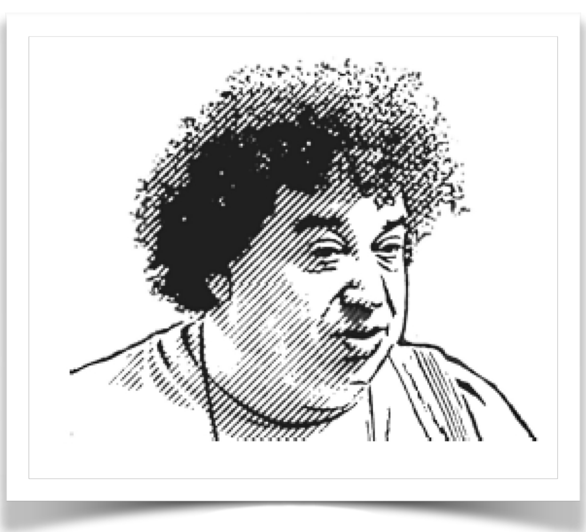
Filip Flajolet (1983)

Solution # 1: Probabilistic Counting

Problem. Count the number of *unique* elements in a data stream.
Requirement. Use $O(\log n)$ space, where n is the number of elements in the stream.
Assumption. An approximate count is acceptable.

Algorithm.

```
m = 0
FOREACH x in the stream:
   h = HASH(x)
   c = COUNT-TRAILING-ONES(h)
   m = MAX(m, c)

RETURN 2^m
```



Filip Flajolet (1983)

Example.

Datastream:

11110010011001111111111101100001
01010101011101111001100010011000
00000100101100001001110111110011
10110011010101011001110110110111
00111110101011011000110001100011
00111010000101001000010100010100

$m = 0$

Solution # 1: Probabilistic Counting

Problem. Count the number of *unique* elements in a data stream.
Requirement. Use $O(\log n)$ space, where n is the number of elements in the stream.
Assumption. An approximate count is acceptable.

Algorithm.

```
m = 0
FOREACH x in the stream:
    h = HASH(x)
    c = COUNT-TRAILING-ONES(h)
    m = MAX(m, c)

RETURN 2^m
```

Filip Flajolet (1983)

Example.

Datastream:

```
11110010011001111111111101100001
01010101011101111001100010011000
00000100101100001001110111110011
10110011010101011001110110110111
00111110101011011000110001100011
00111010000101001000010100010100
```

$m = 1$

Solution # 1: Probabilistic Counting

Problem. Count the number of *unique* elements in a data stream.
Requirement. Use $O(\log n)$ space, where n is the number of elements in the stream.
Assumption. An approximate count is acceptable.

Algorithm.

```
m = 0
FOREACH x in the stream:
   h = HASH(x)
   c = COUNT-TRAILING-ONES(h)
   m = MAX(m, c)

RETURN 2^m
```

Filip Flajolet (1983)

Example.

Datastream:

```
11110010011001111111111101100001
01010101011101111001100010011000
00000100101100001001110111110011
10110011010101011001110110110111
00111110101011011000110001100011
00111010000101001000010100010100
```

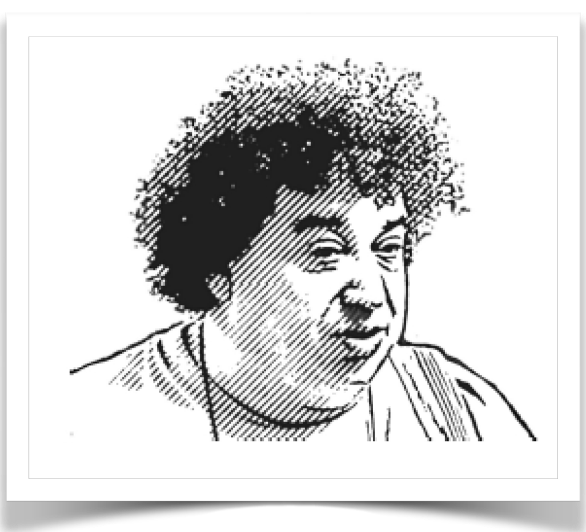
$m = 1$

Solution # 1: Probabilistic Counting

Problem. Count the number of *unique* elements in a data stream.
Requirement. Use $O(\log n)$ space, where n is the number of elements in the stream.
Assumption. An approximate count is acceptable.

Algorithm.

```
m = 0
FOREACH x in the stream:
    h = HASH(x)
    c = COUNT-TRAILING-ONES(h)
    m = MAX(m, c)

RETURN 2^m
```



Filip Flajolet (1983)

Example.

Datastream:
1111001001100111111111111011000**1**
01010101011101111001100010011000
000001001011000010011101111100**11**
10110011010101011001110110110111
00111110101011011000110001100011
00111010000101001000010100010100

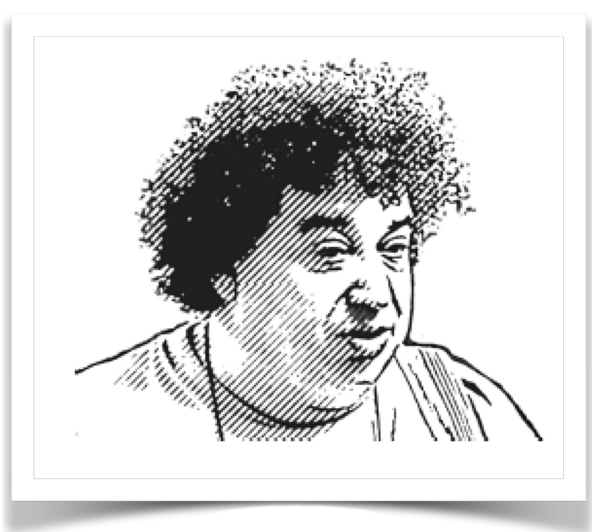
$m = 2$

Solution # 1: Probabilistic Counting

Problem. Count the number of *unique* elements in a data stream.
Requirement. Use $O(\log n)$ space, where n is the number of elements in the stream.
Assumption. An approximate count is acceptable.

Algorithm.

```
m = 0
FOREACH x in the stream:
    h = HASH(x)
    c = COUNT-TRAILING-ONES(h)
    m = MAX(m, c)

RETURN 2^m
```



Filip Flajolet (1983)

Example.

Datastream:

```
11110010011001111111111101100001
01010101011101111001100010011000
00000100101100001001110111110011
10110011010101011001110110110111
00111110101011011000110001100011
00111010000101001000010100010100
```

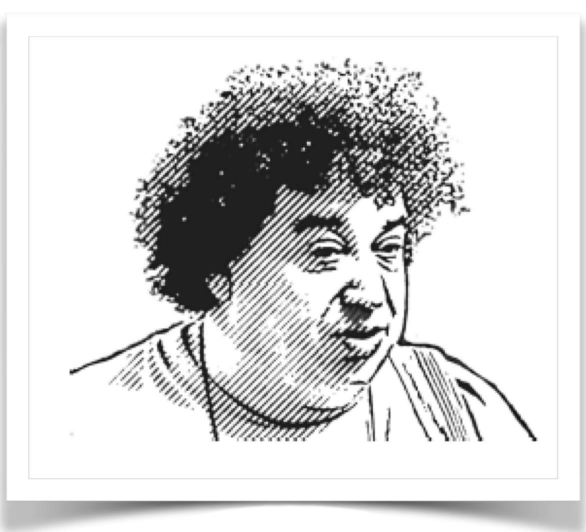
$m = 3$

Solution # 1: Probabilistic Counting

> **Problem**. Count the number of *unique* elements in a data stream.
> **Requirement.** Use $O(\log n)$ space, where n is the number of elements in the stream.
> **Assumption.** An approximate count is acceptable.

Algorithm.

```
m = 0
FOREACH x in the stream:
   h = HASH(x)
   c = COUNT-TRAILING-ONES(h)
   m = MAX(m, c)

RETURN 2^m
```



Filip Flajolet (1983)

Example.

Datastream:

1111001001100111111111110110000**1**
01010101011101111001100010011000
000001001011000010011101111100**11**
10110011010101011001110110110**111**
001111101010110110001100011000**11**
00111010000101001000010100010100

$m = 3$

Solution # 1: Probabilistic Counting

Problem. Count the number of *unique* elements in a data stream.
Requirement. Use $O(\log n)$ space, where n is the number of elements in the stream.
Assumption. An approximate count is acceptable.

Algorithm.

```
m = 0
FOREACH x in the stream:
   h = HASH(x)
   c = COUNT-TRAILING-ONES(h)
   m = MAX(m, c)

RETURN 2^m
```

Filip Flajolet (1983)

Example.

Datastream:

1111001001100111111111110110000**1**
01010101011101111001100010011000
000001001011000010011101111100**11**
10110011010101011001110110110**111**
001111101010110110001100011000**11**
00111010000101001000010100010100

$m = 3$

Solution # 1: Probabilistic Counting

Problem. Count the number of *unique* elements in a data stream.
Requirement. Use $O(\log n)$ space, where n is the number of elements in the stream.
Assumption. An approximate count is acceptable.

Algorithm.

```
m = 0
FOREACH x in the stream:
    h = HASH(x)
    c = COUNT-TRAILING-ONES(h)
    m = MAX(m, c)

RETURN 2^m
```

Filip Flajolet (1983)

Example.

Datastream:

1111001001100111111111110110000**1**
01010101011101111001100010011000
000001001011000010011101111100**11**
10110011010101011001110110110**111**
001111101010110110001100011000**11**
00111010000101001000010100010100

$m = 3$

Estimated number of unique elements = $2^3 = 8$

What is going on?

Solution # 1: Probabilistic Counting

All possible bit-strings of size 4:

```
0000
0001
0010
0011
0100
0101
0110
0111
1000
1001
1010
1011
1100
1101
1110
1111
```

Solution # 1: Probabilistic Counting

All possible bit-strings of size 4:

```
0000
0001
0010
0011
0100
0101
0110
0111
1000
1001
1010
1011
1100
1101
1110
1111
```

P(seeing 1 trailing 1)

$= 1/2$

Seeing 1 trailing 1
indicates
$\geq 1/(1/2) = 2^1 = 2$
were likely seen

Solution # 1: Probabilistic Counting

All possible bit-strings of size 4:

0000	0000
000**1**	0001
0010	0010
001**1**	00**11**
0100	0100
010**1**	0101
0110	0110
011**1**	01**11**
1000	1000
100**1**	1001
1010	1010
101**1**	10**11**
1100	1100
110**1**	1101
1110	1110
111**1**	11**11**

P(seeing 1 trailing 1) $= 1/2$

Seeing 1 trailing 1 indicates $\geq 1/(1/2) = 2^1 = 2$ were likely seen

P(seeing 2 trailing 1s) $= 1/4$

Seeing 2 trailing 1s indicates $\geq 1/(1/4) = 2^2 = 4$ were likely seen

Solution # 1: Probabilistic Counting

All possible bit-strings of size 4:

0000	0000	0000
000**1**	0001	0001
0010	0010	0010
001**1**	00**11**	0011
0100	0100	0100
010**1**	0101	0101
0110	0110	0110
011**1**	01**11**	0**111**
1000	1000	1000
100**1**	1001	1001
1010	1010	1010
101**1**	10**11**	1011
1100	1100	1100
110**1**	1101	1101
1110	1110	1110
111**1**	11**11**	1**111**

P(seeing 1 trailing 1) $= 1/2$

P(seeing 2 trailing 1s) $= 1/4$

P(seeing 3 trailing 1s) $= 1/8$

Seeing 1 trailing 1 indicates $\geq 1/(1/2) = 2^1 = 2$ were likely seen

Seeing 2 trailing 1s indicates $\geq 1/(1/4) = 2^2 = 4$ were likely seen

Seeing 3 trailing 1s indicates $\geq 2^3 = 8$ were likely seen

Solution # 1: Probabilistic Counting

All possible bit-strings of size 4:

0000	0000	0000	0000
000**1**	0001	0001	0001
0010	0010	0010	0010
001**1**	00**11**	0011	0011
0100	0100	0100	0100
010**1**	0101	0101	0101
0110	0110	0110	0110
011**1**	01**11**	0**111**	0111
1000	1000	1000	1000
100**1**	1001	1001	1001
1010	1010	1010	1010
101**1**	10**11**	1011	1011
1100	1100	1100	1100
110**1**	1101	1101	1101
1110	1110	1110	1110
111**1**	11**11**	1**111**	**1111**
P(seeing 1 trailing 1) $= 1/2$	P(seeing 2 trailing 1s) $= 1/4$	P(seeing 3 trailing 1s) $= 1/8$	P(seeing 4 trailing 1s) $= 1/16$
Seeing 1 trailing 1 indicates $\geq 1/(1/2) = 2^1 = 2$ were likely seen	Seeing 2 trailing 1s indicates $\geq 1/(1/4) = 2^2 = 4$ were likely seen	Seeing 3 trailing 1s indicates $\geq 2^3 = 8$ were likely seen	Seeing 4 trailing 1s indicates $\geq 2^4 = 16$ were likely seen

Solution # 1: Probabilistic Counting

All possible bit-strings of size 4:

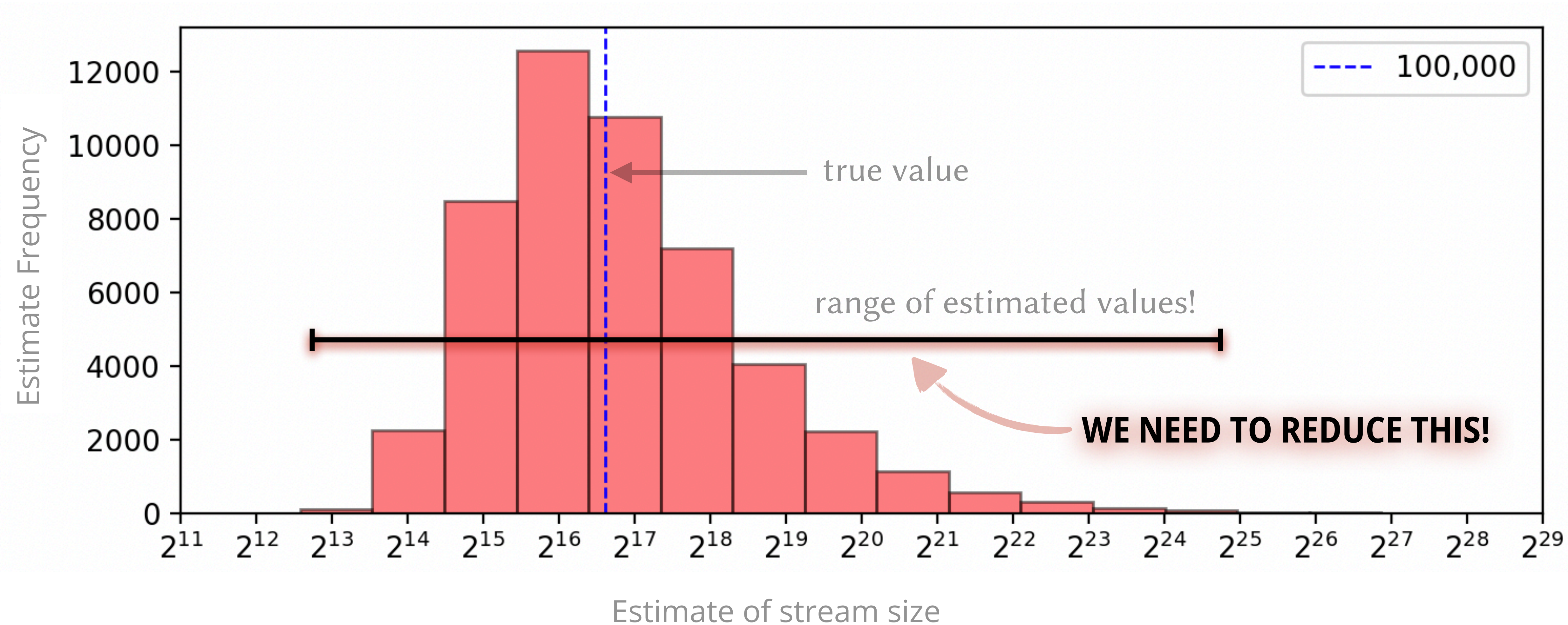
0000	0000	0000	0000
000**1**	0001	0001	0001
0010	0010	0010	0010
001**1**	00**11**	0011	0011
0100	0100	0100	0100
010**1**	0101	0101	0101
0110	0110	0110	0110
011**1**	01**11**	0**111**	0111
1000	1000	1000	1000
100**1**	1001	1001	1001
1010	1010	1010	1010
101**1**	10**11**	1011	1011
1100	1100	1100	1100
110**1**	1101	1101	1101
1110	1110	1110	1110
111**1**	11**11**	1**111**	**1111**
P(seeing 1 trailing 1) $= 1/2$	P(seeing 2 trailing 1s) $= 1/4$	P(seeing 3 trailing 1s) $= 1/8$	P(seeing 4 trailing 1s) $= 1/16$

Probabilistically speaking: Seeing many trailing 1s ≡ seeing many unique elements!

Solution # 1: Analysis

How good is this estimate?

Performed 50,000 experiments. Generated a stream of size 100,000 in each experiment.



The estimate is good on average, but there are too many poor estimates!

If we estimate 1000 times the value of $5 + 5$ to be 7, and then estimate 1000 times the value of $5 + 5$ to be 13, we get an average estimate of 10, which is excellent! However, every single estimate is bad!

Solution # 2: Stochastic Averaging

Idea. Reduce variance by averaging the result of multiple independent experiments.

Method. Use multiple counters, each with a different hash function.

Alternative Method.

- Subdivide the stream into $M = 2^k$ buckets.
- Find the maximum number of trailing 1s in each bucket.
- Find the *mean* of the maximums.

Algorithm.

```
m[M] = {0}
FOREACH x in the stream:
    h = HASH(x)
    c = COUNT-TRAILING-ONES(h)
    d = GET-FIRST-BITS(h, k)
    m[d] = MAX(m[d], c)

RETURN M * (2^mean(m)) / 0.77351
```

Solution # 2: Stochastic Averaging

Idea. Reduce variance by averaging the result of multiple independent experiments.

Method. Use multiple counters, each with a different hash function.

Alternative Method.

- Subdivide the stream into $M = 2^k$ buckets.
- Find the maximum number of trailing 1s in each bucket.
- Find the *mean* of the maximums.

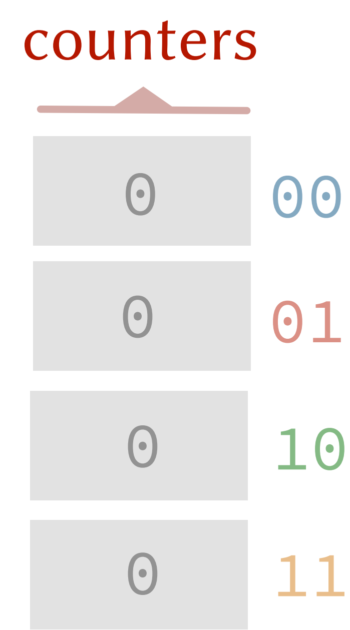
Algorithm.

```
m[M] = {0}
FOREACH x in the stream:
    h = HASH(x)
    c = COUNT-TRAILING-ONES(h)
    d = GET-FIRST-BITS(h, k)
    m[d] = MAX(m[d], c)

RETURN M * (2^mean(m)) / 0.77351
```

array of $M = 2^k$ counters initialized to 0

use first k-bits to know which counter to update

Solution # 2: Stochastic Averaging Example



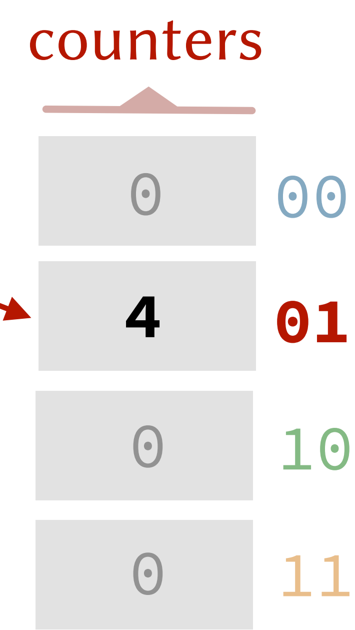
Algorithm.

```
m[M] = {0}
FOREACH x in the stream:
    h = HASH(x)
    c = COUNT-TRAILING-ONES(h)
    d = GET-FIRST-BITS(h, k)
    m[d] = MAX(m[d], c)

RETURN M * (2^mean(m)) / 0.77351
```

Solution # 2: Stochastic Averaging Example

bucket number (**01**) trailing 1's (**1111**)

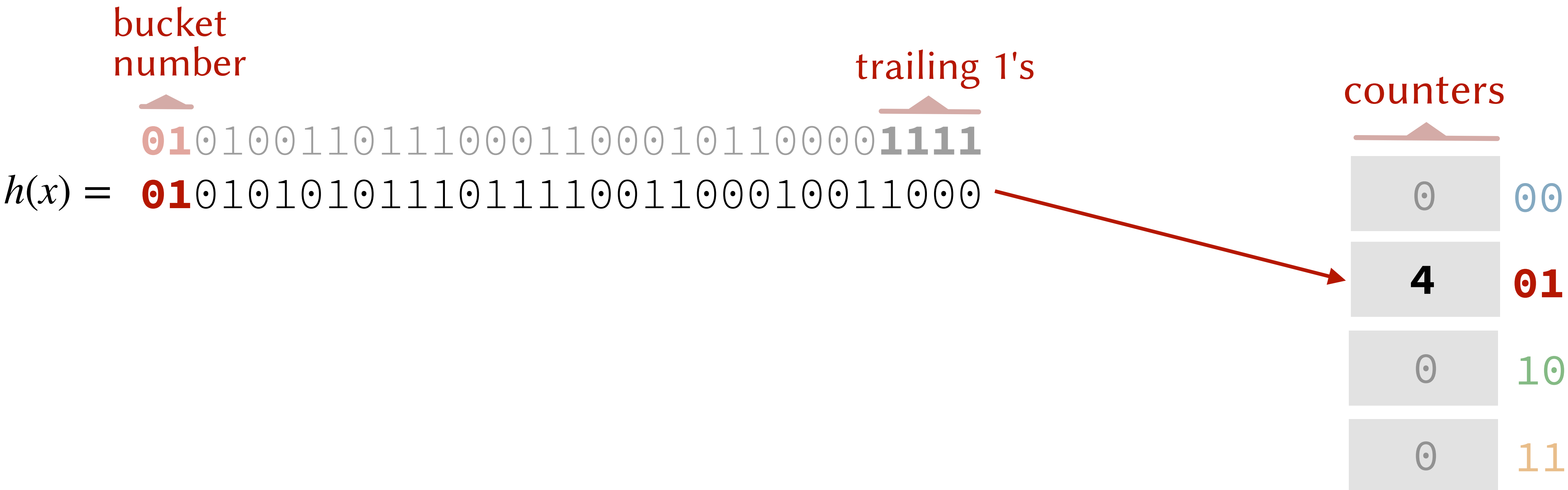
$h(x) =$ **01**01001101110001100010110000**1111**



Algorithm.

```
m[M] = {0}
FOREACH x in the stream:
    h = HASH(x)
    c = COUNT-TRAILING-ONES(h)
    d = GET-FIRST-BITS(h, k)
    m[d] = MAX(m[d], c)

RETURN M * (2^mean(m)) / 0.77351
```

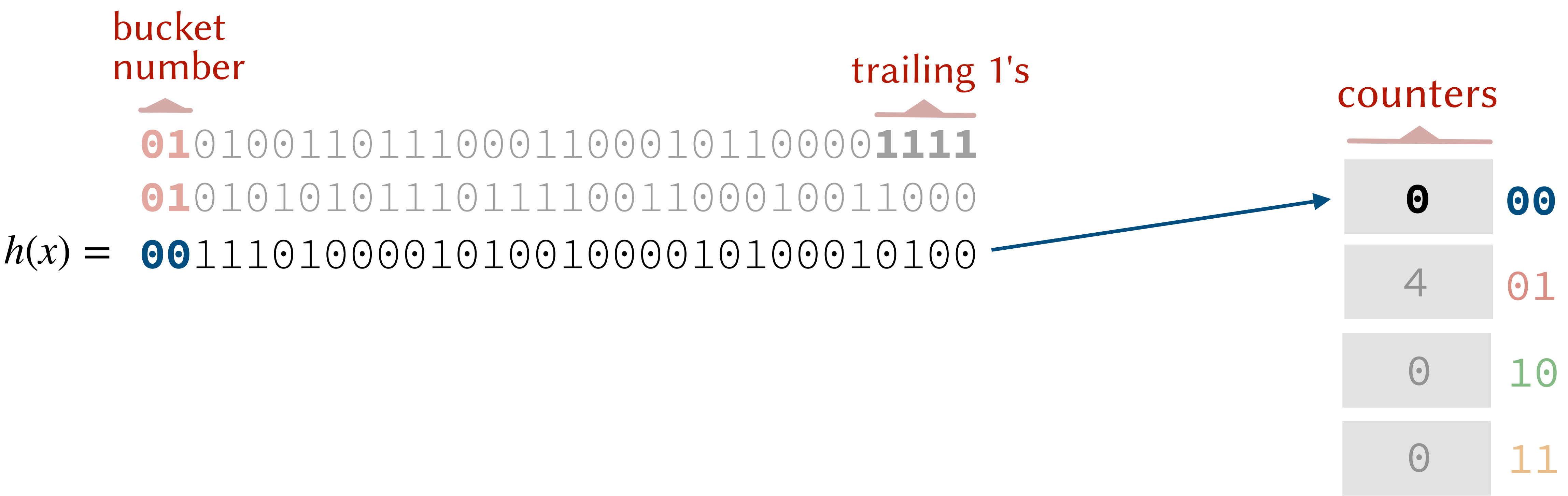
Solution # 2: Stochastic Averaging Example



Algorithm.

```
m[M] = {0}
FOREACH x in the stream:
    h = HASH(x)
    c = COUNT-TRAILING-ONES(h)
    d = GET-FIRST-BITS(h, k)
    m[d] = MAX(m[d], c)

RETURN M * (2^mean(m)) / 0.77351
```

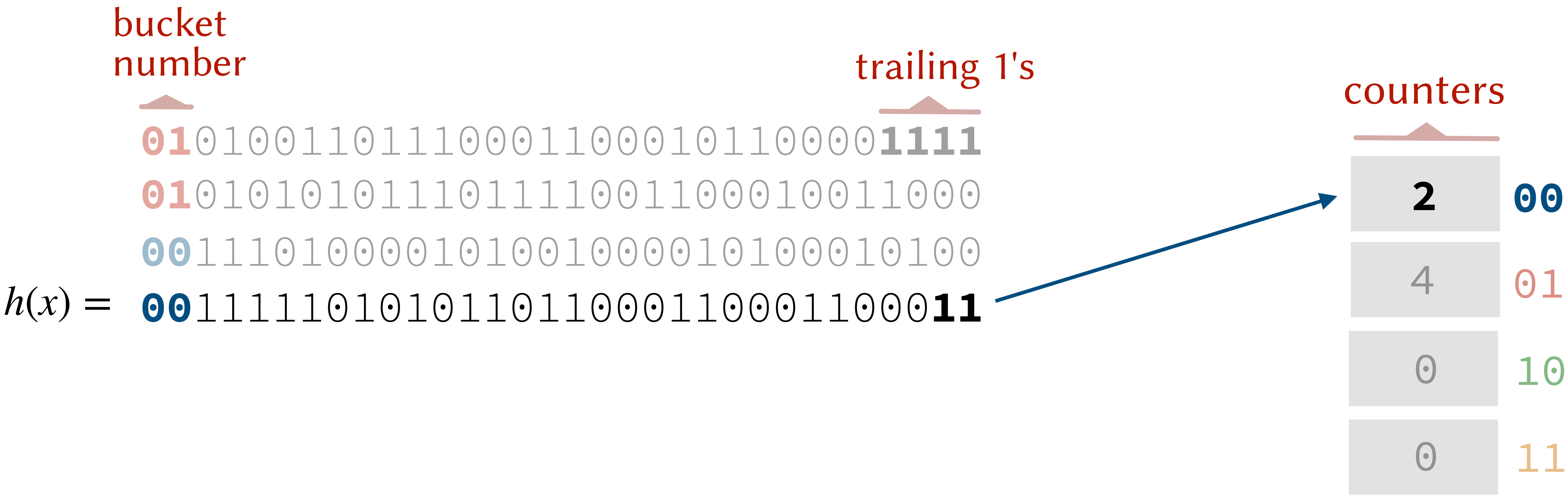
Solution # 2: Stochastic Averaging Example



Algorithm.

```
m[M] = {0}
FOREACH x in the stream:
    h = HASH(x)
    c = COUNT-TRAILING-ONES(h)
    d = GET-FIRST-BITS(h, k)
    m[d] = MAX(m[d], c)

RETURN M * (2^mean(m)) / 0.77351
```

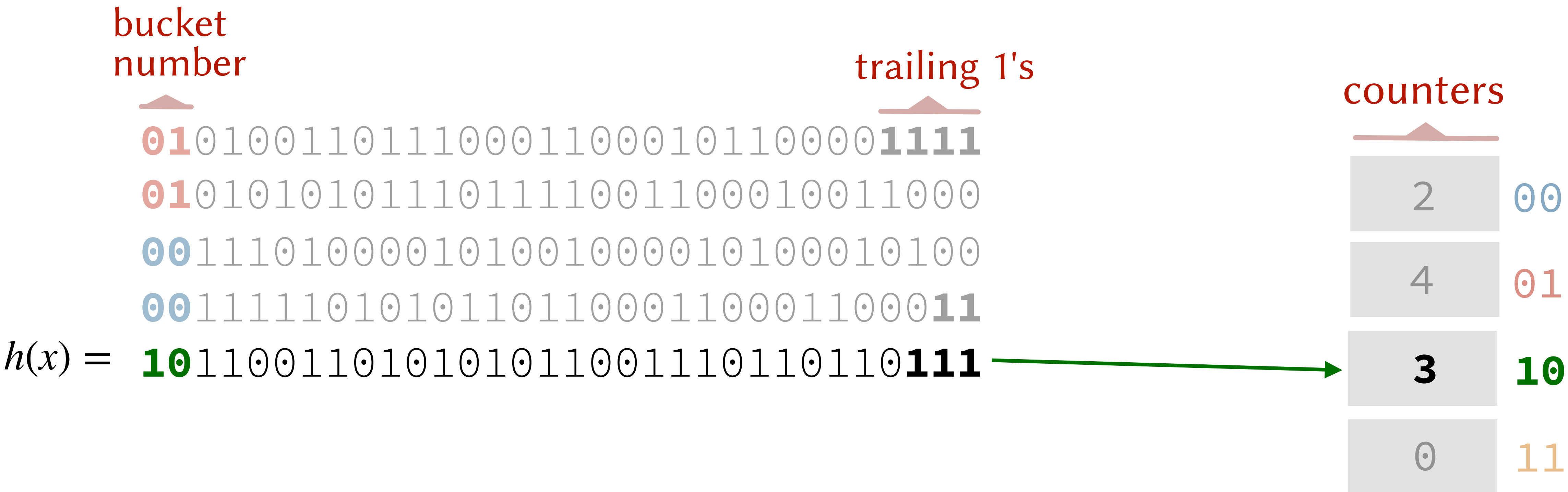
Solution # 2: Stochastic Averaging Example



Algorithm.

```
m[M] = {0}
FOREACH x in the stream:
    h = HASH(x)
    c = COUNT-TRAILING-ONES(h)
    d = GET-FIRST-BITS(h, k)
    m[d] = MAX(m[d], c)

RETURN M * (2^mean(m)) / 0.77351
```

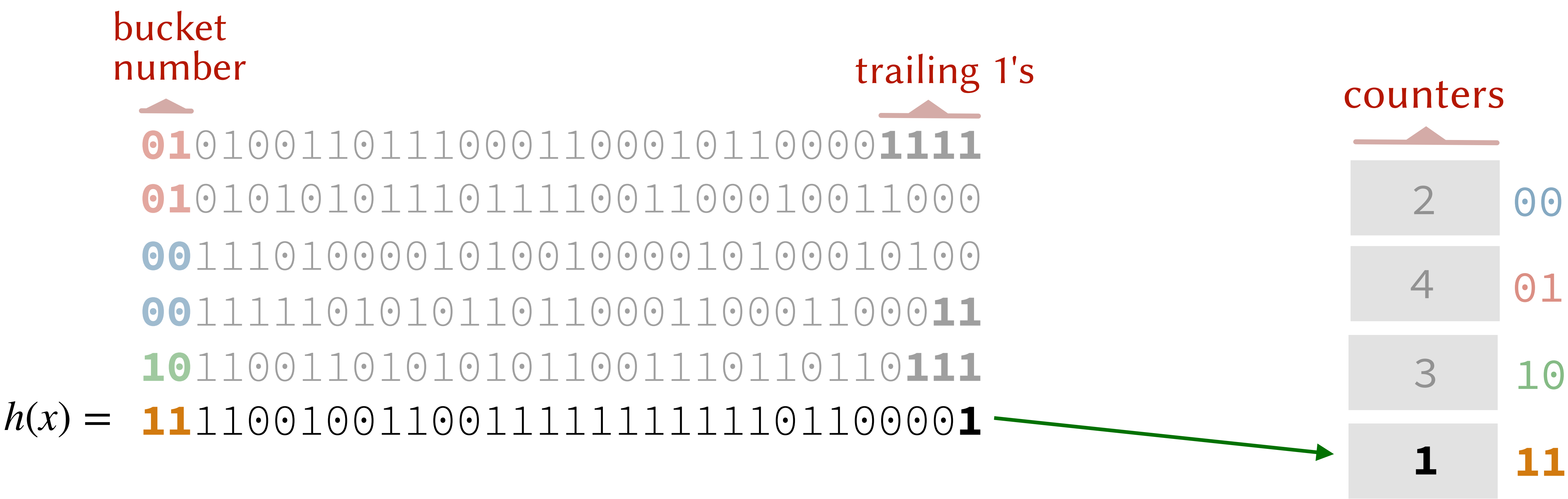
Solution # 2: Stochastic Averaging Example



Algorithm.

```
m[M] = {0}
FOREACH x in the stream:
    h = HASH(x)
    c = COUNT-TRAILING-ONES(h)
    d = GET-FIRST-BITS(h, k)
    m[d] = MAX(m[d], c)

RETURN M * (2^mean(m)) / 0.77351
```

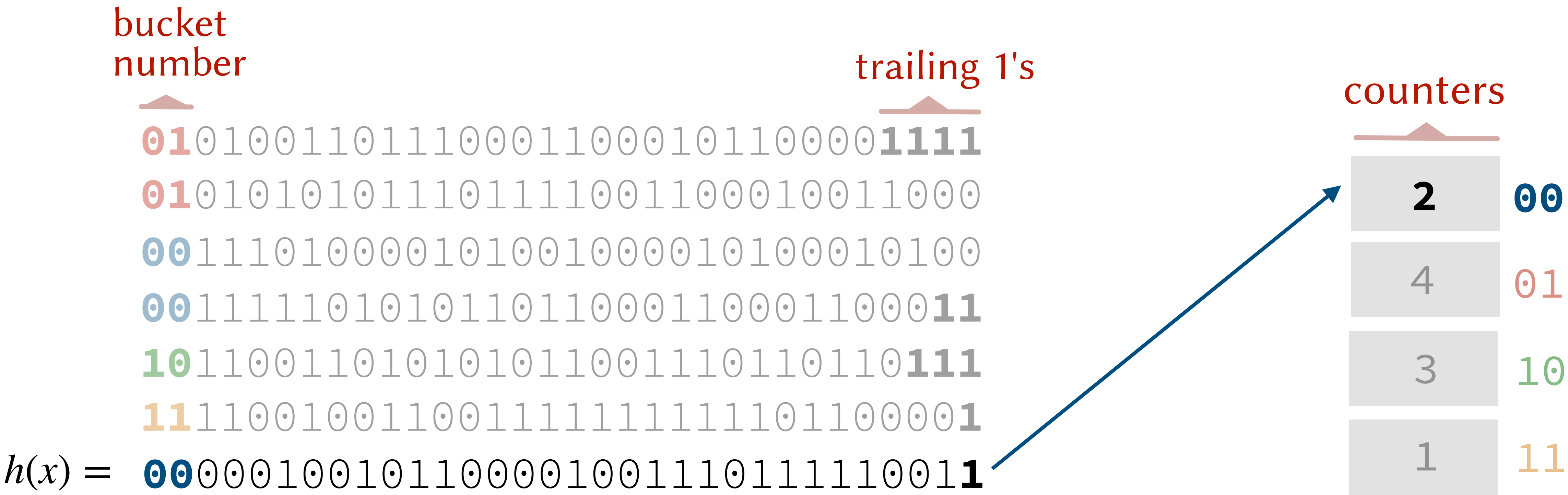
Solution # 2: Stochastic Averaging Example



Algorithm.

```
m[M] = {0}
FOREACH x in the stream:
    h = HASH(x)
    c = COUNT-TRAILING-ONES(h)
    d = GET-FIRST-BITS(h, k)
    m[d] = MAX(m[d], c)

RETURN M * (2^mean(m)) / 0.77351
```

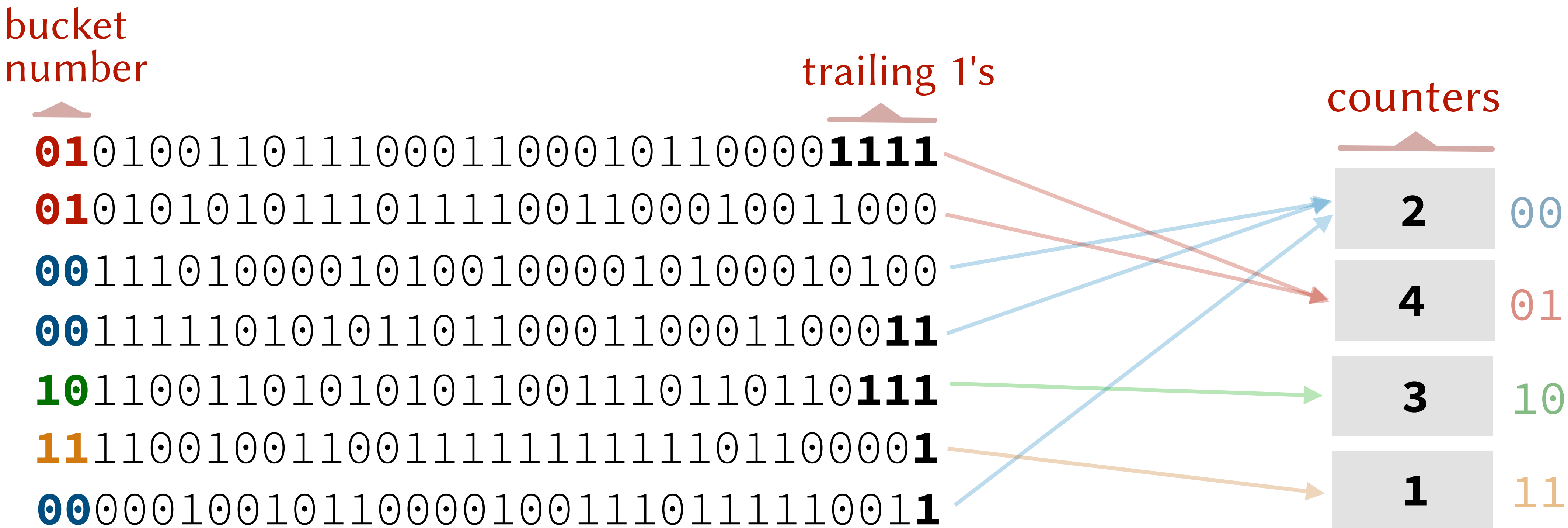
Solution # 2: Stochastic Averaging Example



Algorithm.

```
m[M] = {0}
FOREACH x in the stream:
    h = HASH(x)
    c = COUNT-TRAILING-ONES(h)
    d = GET-FIRST-BITS(h, k)
    m[d] = MAX(m[d], c)

RETURN M * (2^mean(m)) / 0.77351
```

Solution # 2: Stochastic Averaging Example



Algorithm.

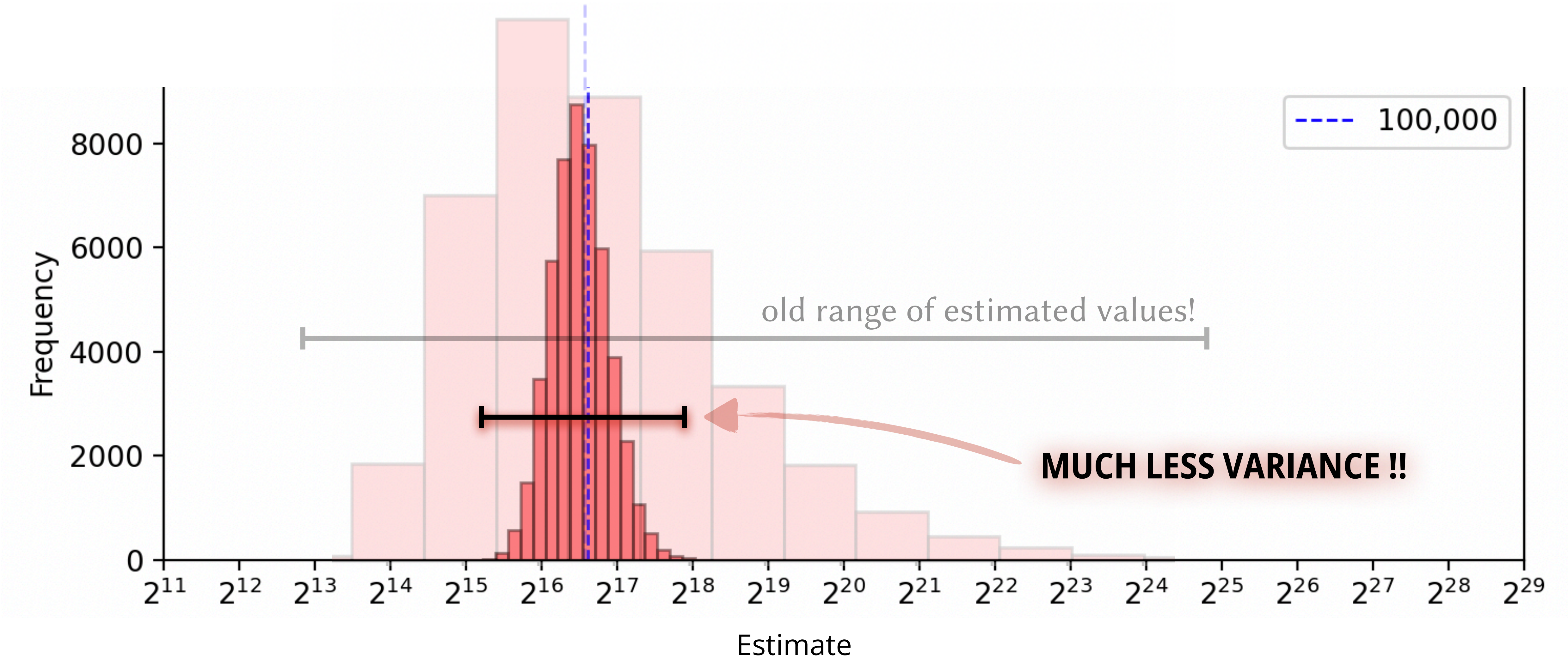
```
m[M] = {0}
FOREACH x in the stream:
    h = HASH(x)
    c = COUNT-TRAILING-ONES(h)
    d = GET-FIRST-BITS(h, k)
    m[d] = MAX(m[d], c)

RETURN M * (2^mean(m)) / 0.77351
```

upscaling and a correction factor are needed

Solution # 2: Analysis

How good is this estimate?

Performed 50,000 experiments. Generated a stream of size 100,000 in each experiment.



Refinements.

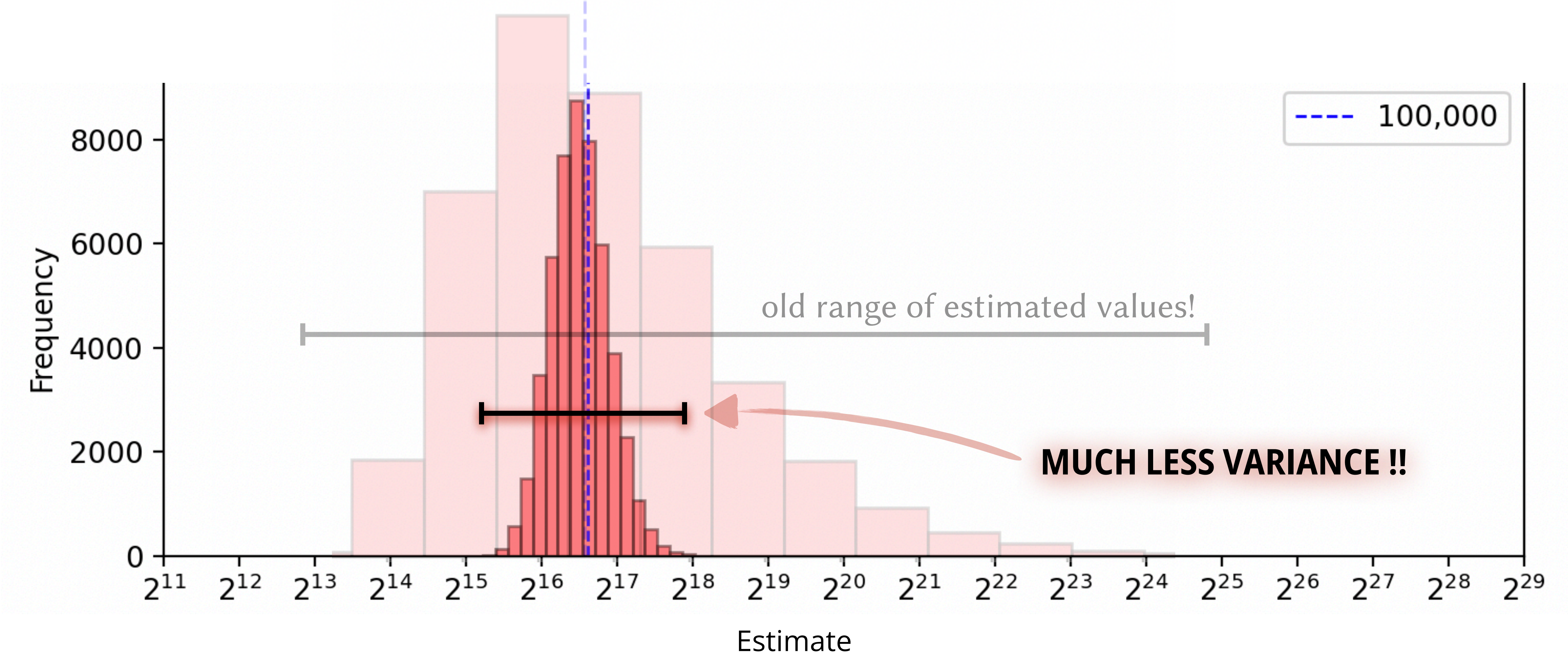
- Discard highest 30% counters.
- Use harmonic instead of arithmetic mean.
- Several others!

Result. An error rate of $1.04/\sqrt{M}$
E.g. error $\approx 3\,\%$ using $M = 10$ counters!

Solution # 2: Analysis

How good is this estimate?

Performed 50,000 experiments. Generated a stream of size 100,000 in each experiment.



Memory Requirements.

- Storing a number n requires $\sim \log_2(n)$ bits.
- We don't need to store n, we need to store the maximum number of trailing 1s in n, which is at most $\sim \log_2(\log_2(n))$ bits!

Result. An error rate of $1.04/\sqrt{M}$
E.g. error $\approx 3\,\%$ using $M = 10$ counters!

Solution # 2: Analysis

Memory Requirements.

- Storing a number n requires $\sim \log_2(n)$ bits.
- We don't need to store n, we need to store the maximum number of trailing 1s in n, which is requires at most $\sim \log_2(\log_2(n))$ bits!

Example.

Assume the stream has $n = 2^{30}$ elements ($\approx 10^9$ elements).

The maximum number of trailing 1s $= \log_2(2^{30}) = 30$.

To store the number 30, we need $\log_2(30)$ bits.