

Succinct Sets

(Golomb-Rice, Elias-Fano, Roaring)

Niv Dayan - CSC2525, Research Topics in Data Management

So far, we looked at approximate sets

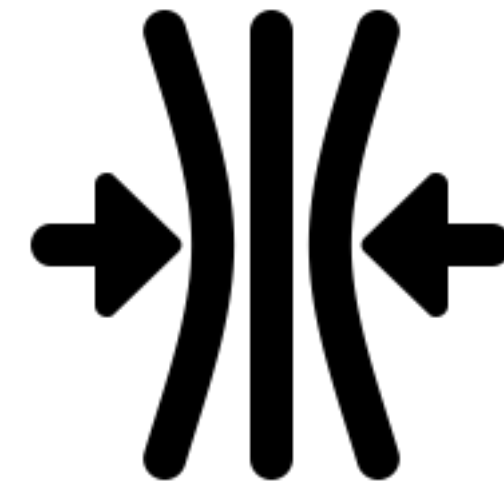


Filters

Filters



**False positive
rate ϵ**

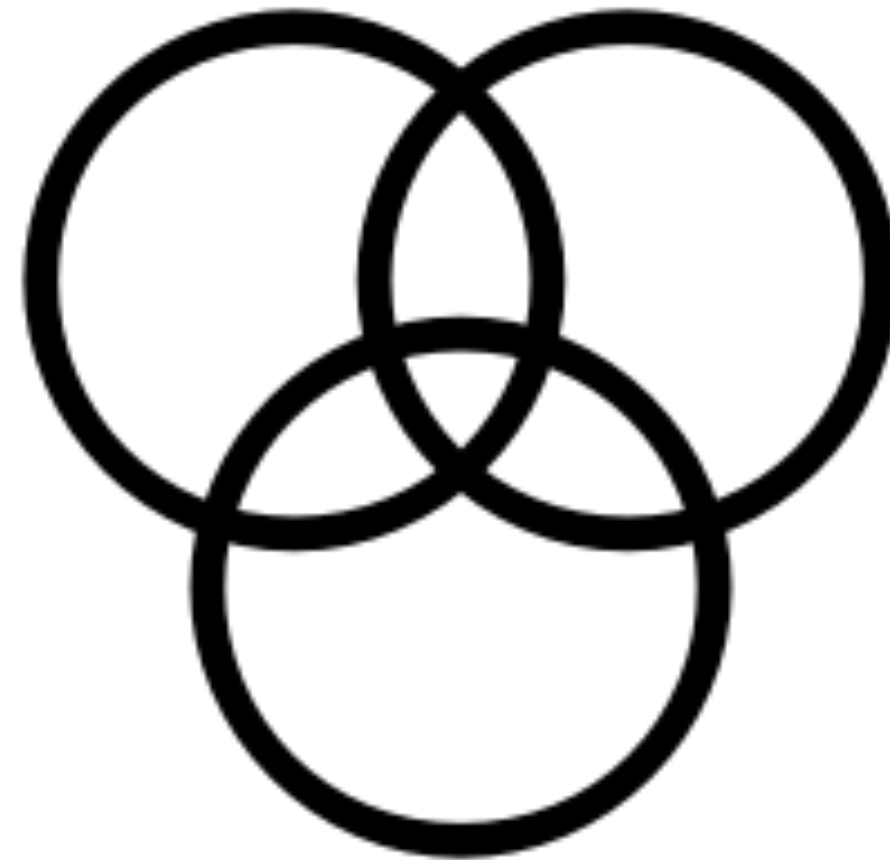


Compact

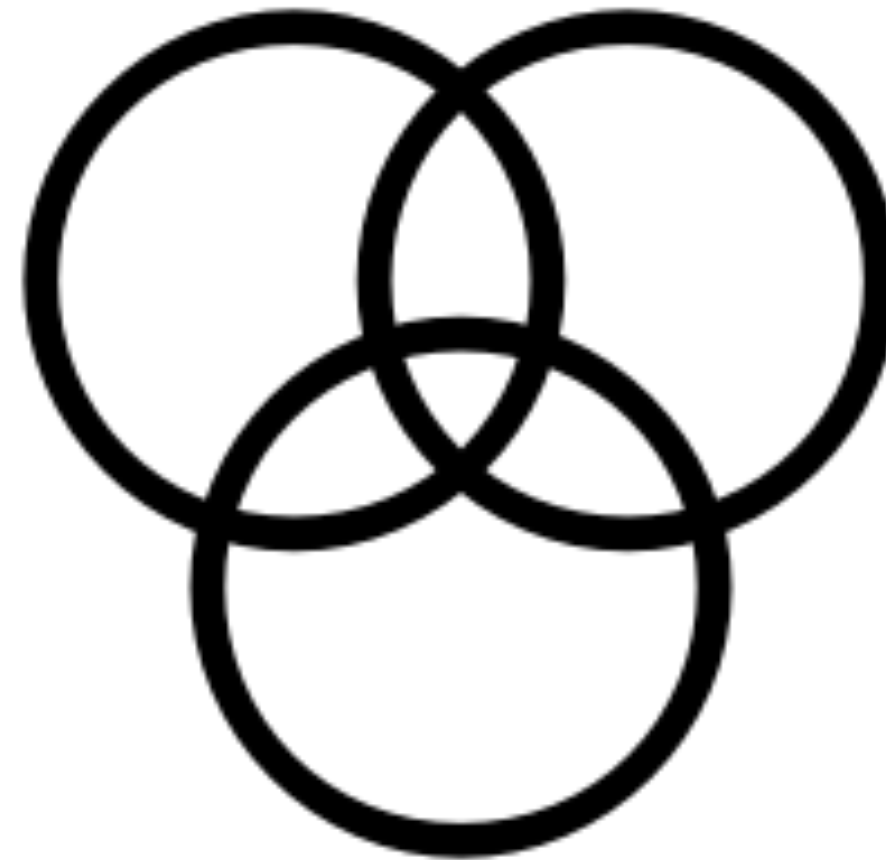


**Eliminate
storage access**

How about exact sets?

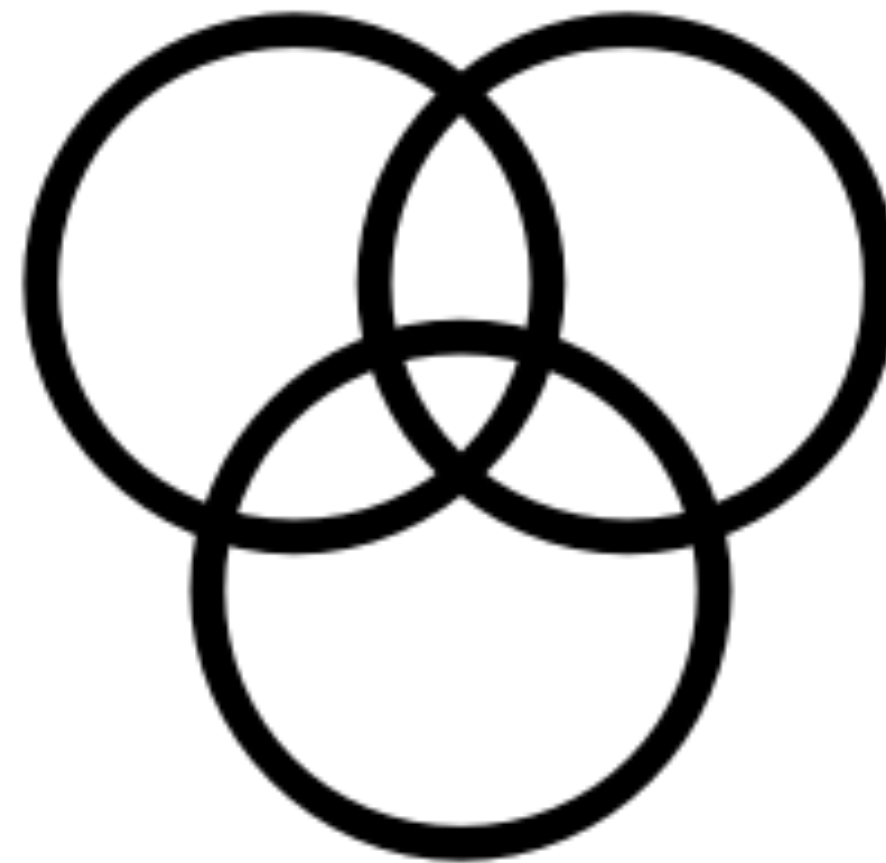


How about exact sets?



N Items of Universe U

How about exact sets?



N Items of Universe U

No false positives or negatives

Examples of exact sets?

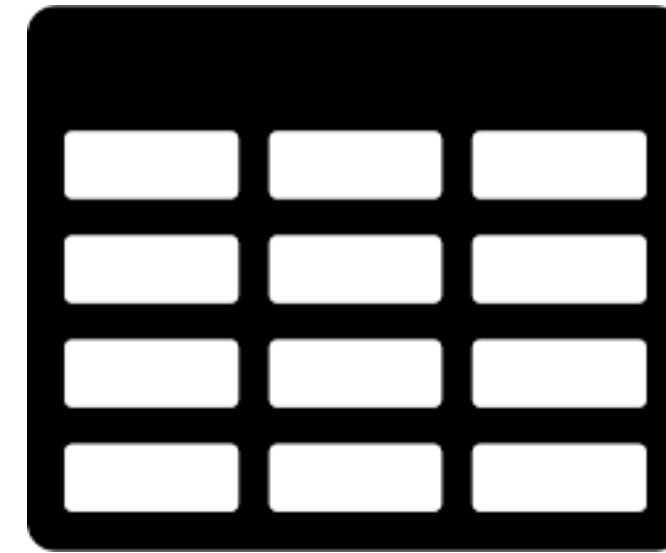
Universe:

Items:



Universe: **UofT Students**

Items: **in this course**



Universe:

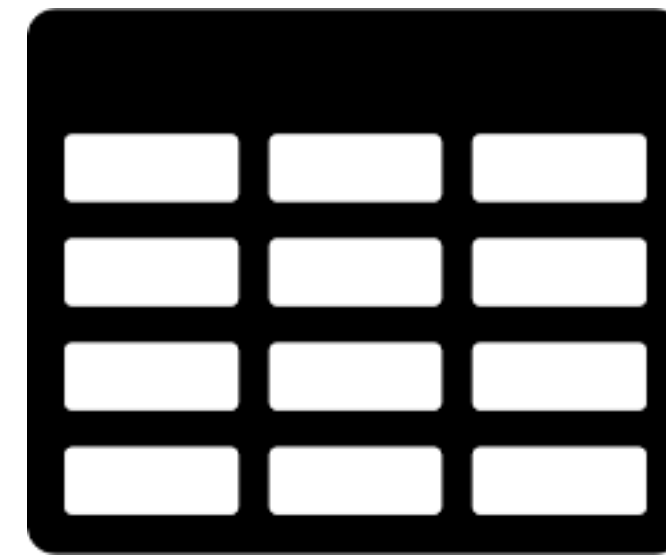
UofT Students

Table rows

Items:

in this course

Deleted



Universe

UofT Students

Table rows

Docs

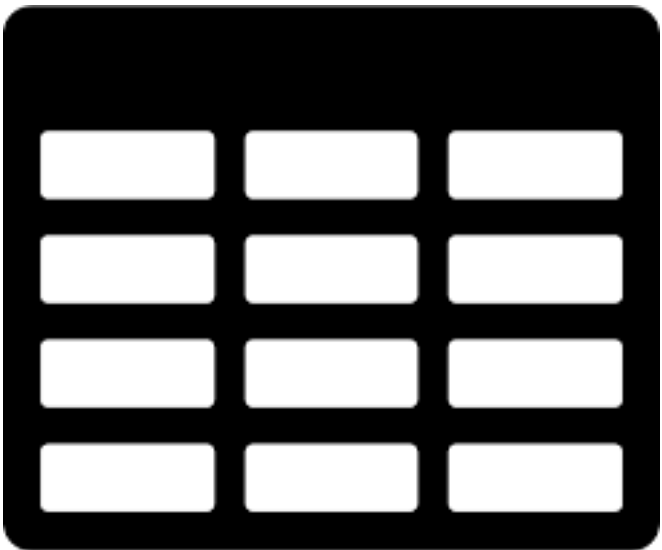
Items

in this course

Deleted

**containing
keyword**

Represent using as little space as possible



Universe

UofT Students

Table rows

Docs

Items

in this course

Deleted

containing
keyword

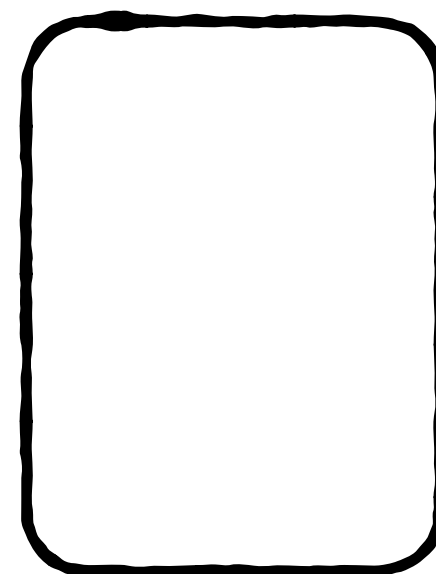
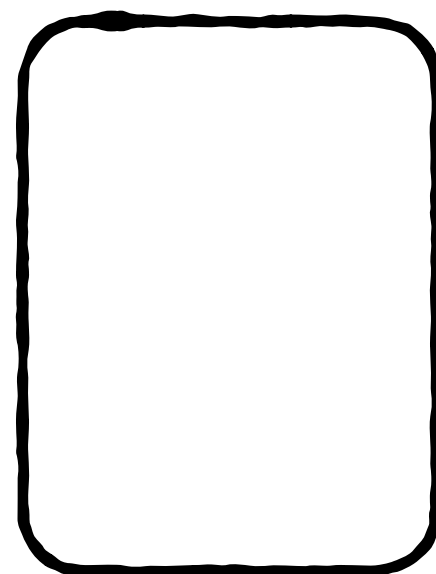
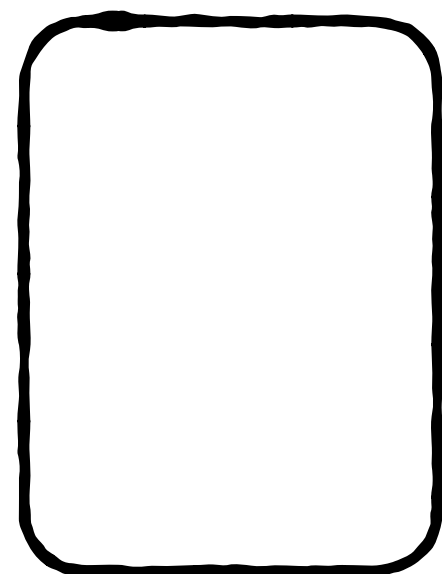
Docs:

1

2

3

...



keywords

Docs:

1

dog

2

dog
cat

3

bat
cat

keywords

Return all documents containing “cat”

Docs:

1

dog

2

dog
cat

3

bat
cat

Inverted Index

Key Docs

dog **1, 2**

cat **2, 3**

bat **3**

1

dog

2

dog

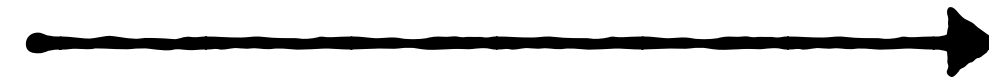
cat

3

bat

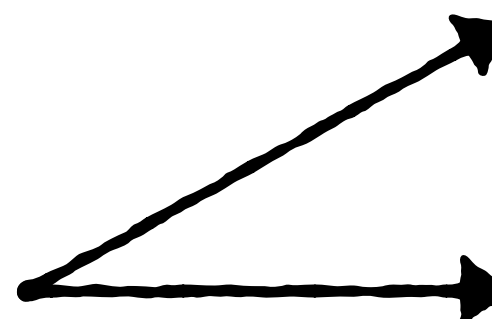
cat

Search: **cat**



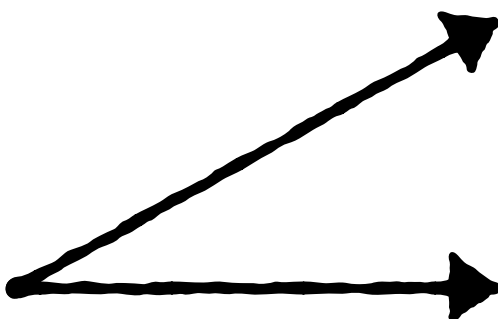
Key	Docs
dog	1, 2
cat	2, 3
bat	3

Search: **cat & dog**

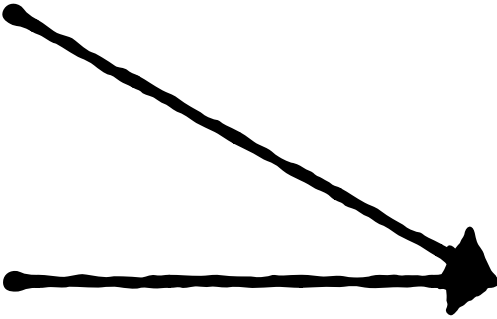


Key	Docs
dog	1, 2
cat	2, 3
bat	3

Search:
cat & dog



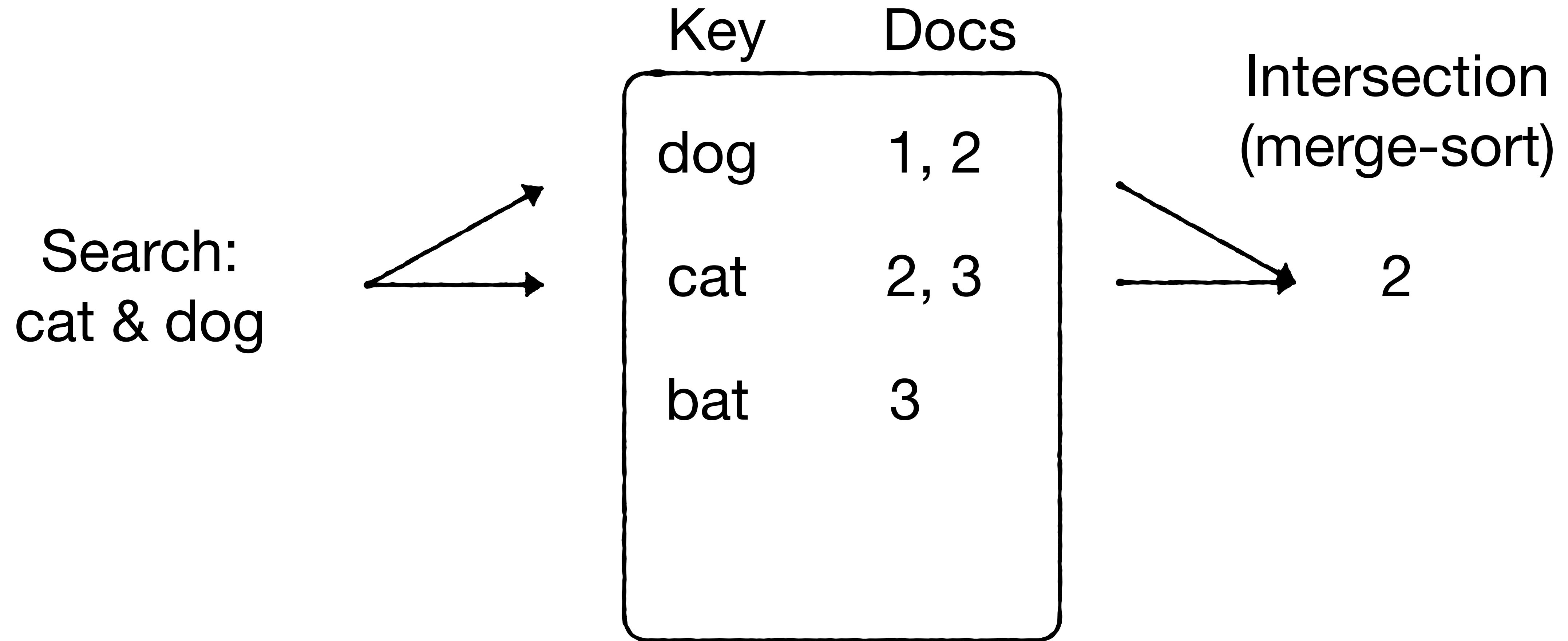
Key	Docs
dog	1, 2
cat	2, 3
bat	3



Intersection
(merge-sort)

2

Part of how search engines work :)



The lists can be long!

Key	Docs
dog	1, 2, 4, 7, 10, 11 ...
cat	2, 3, 4, 6, 10, 12 ...
bat	3, 6, 9, 14, 16, 26 ...

The lists can be long!

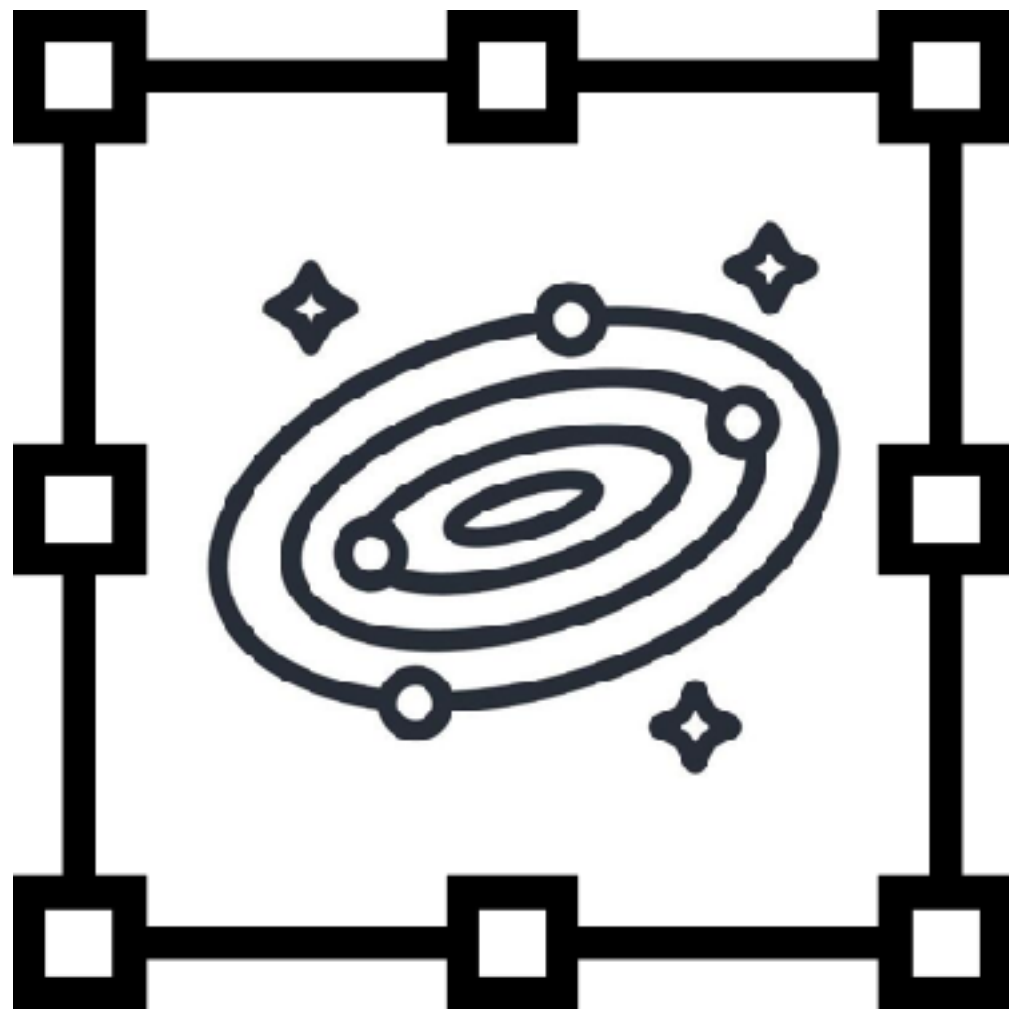
Key	Docs
dog	1, 2, 4, 7, 10, 11 ...
cat	2, 3, 4, 6, 10, 12 ...
bat	3, 6, 9, 14, 16, 26 ...

How to store doc lists as succinctly as possible?

Assumptions

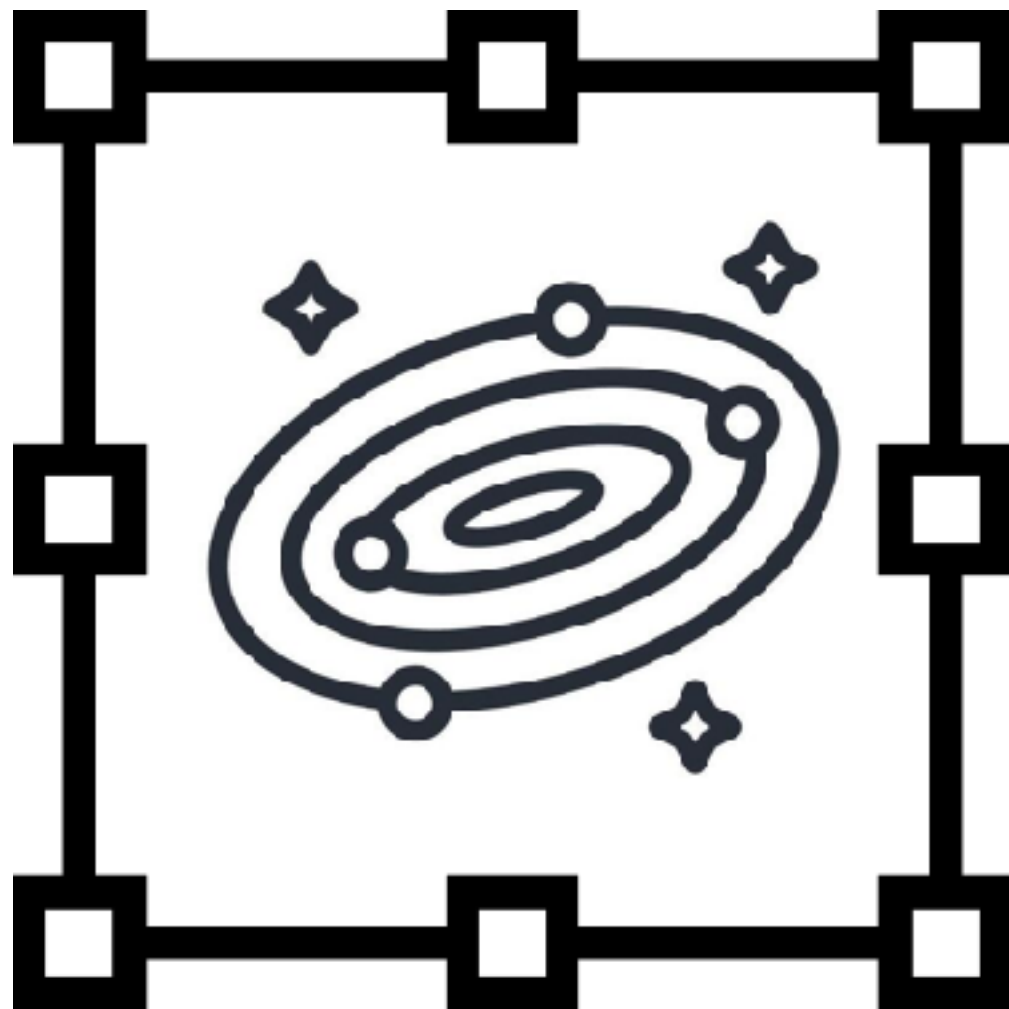
Assumptions

Bounded Universe

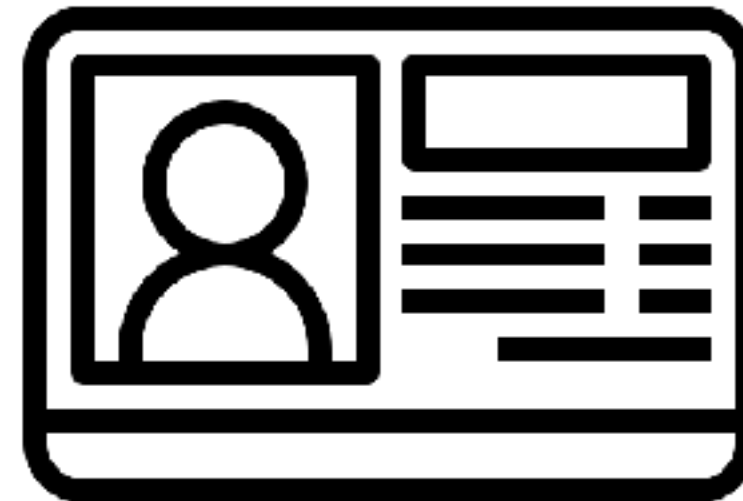


Assumptions

Bounded
Universe

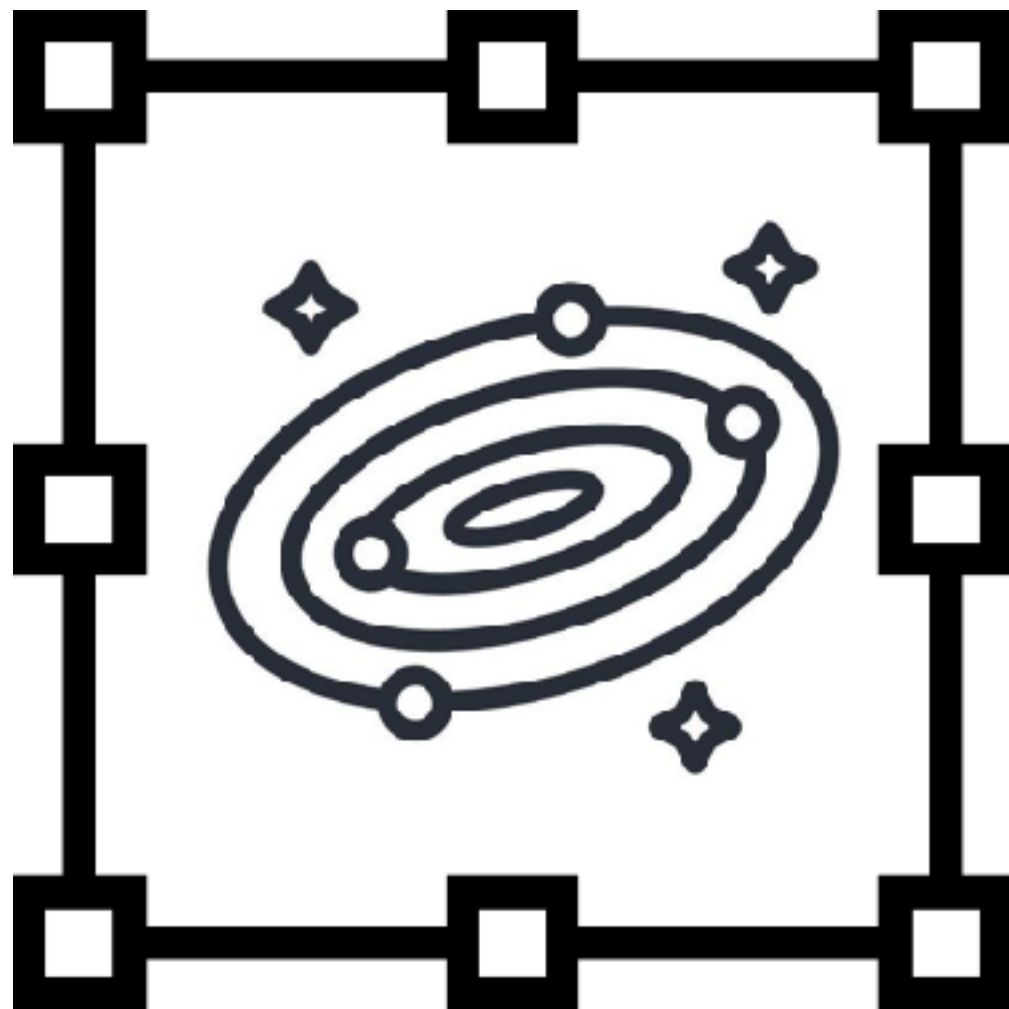


**unique integer ID
per item**

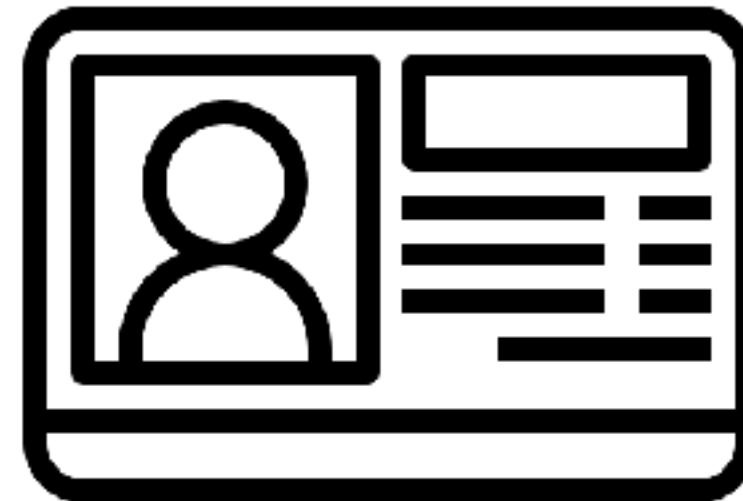


Assumptions

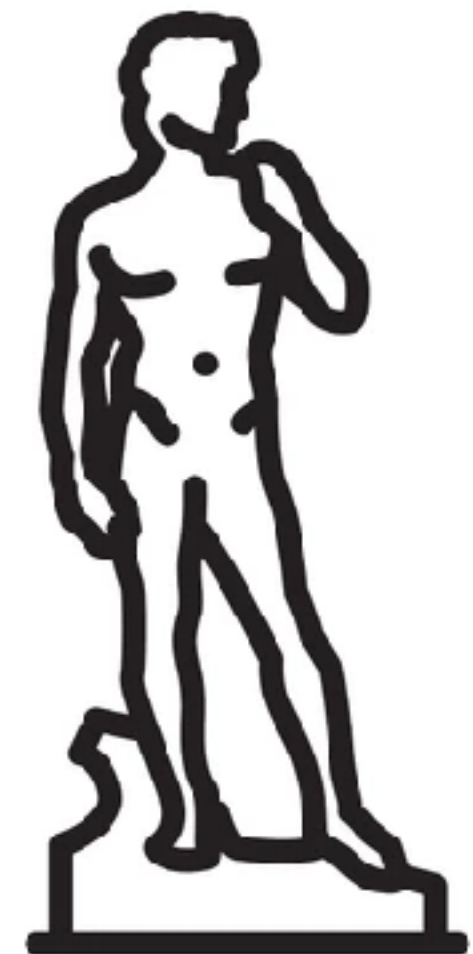
Bounded
Universe



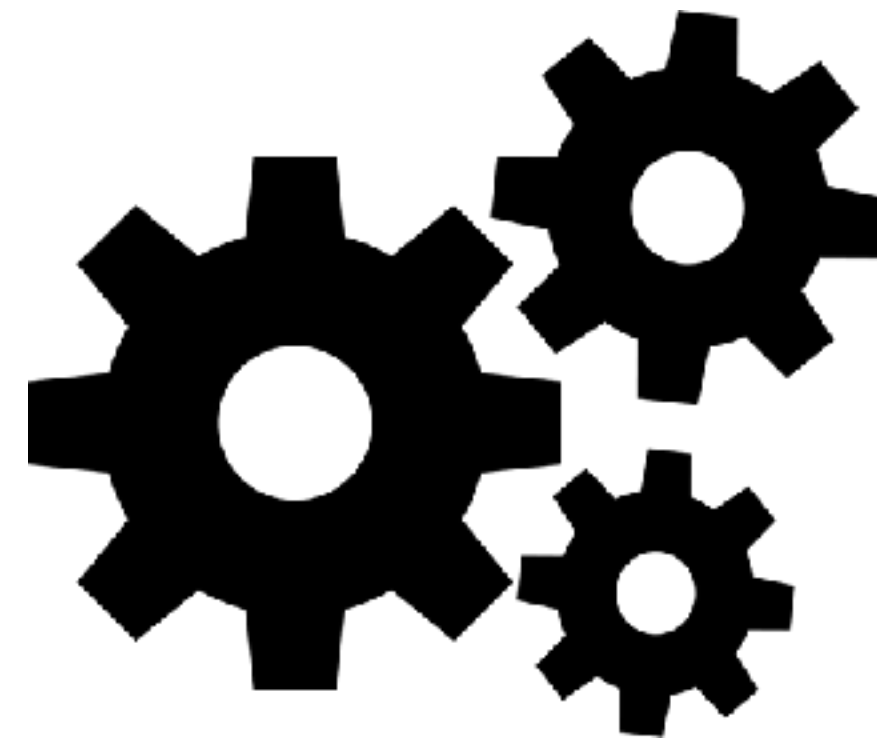
unique integer ID
per item



No duplicates



Set Implementations?

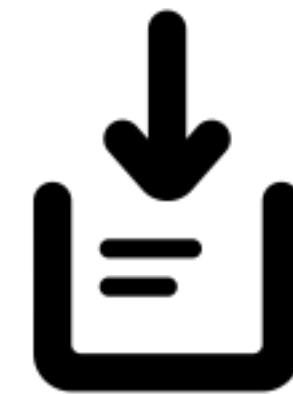


Set Implementations?

Query



Insert



Unordered array

Query $O(N)$

Insert $O(1)$

[...]

Unordered
array

**Ordered
array**

Query

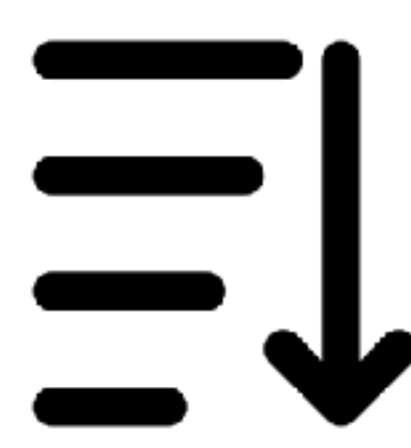
$O(N)$

$O(\log_2 N)$

Insert

$O(1)$

$O(N)$



Unordered
array

Ordered
array

**Binary
Tree**

Query

$O(N)$

$O(\log_2 N)$

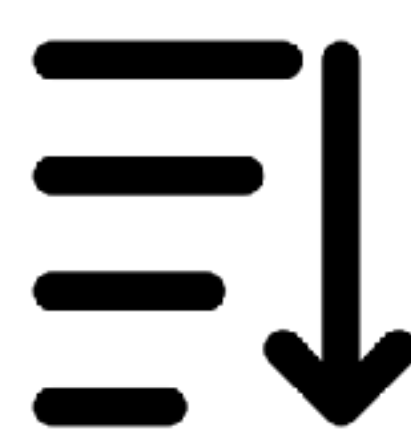
$O(\log_2 N)$

Insert

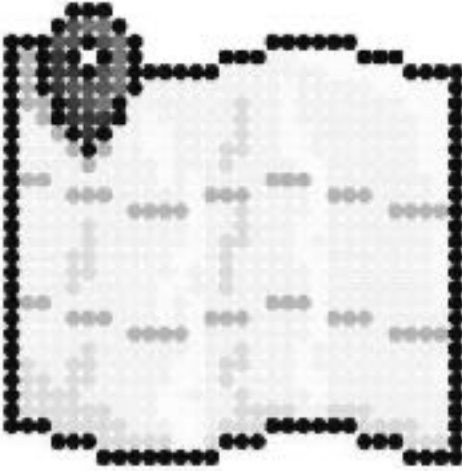
$O(1)$

$O(N)$

$O(\log_2 N)$



	Unordered array	Ordered array	Binary Tree	Hash Table
Query	$O(N)$	$O(\log_2 N)$	$O(\log_2 N)$	$O(1)$
Insert	$O(1)$	$O(N)$	$O(\log_2 N)$	$O(1)$
				

	Unordered array	Ordered array	Binary Tree	Hash Table	Bitmap
Query	$O(N)$	$O(\log_2 N)$	$O(\log_2 N)$	$O(1)$	$O(1)$
Insert	$O(1)$	$O(N)$	$O(\log_2 N)$	$O(1)$	$O(1)$
					

	Unordered array	Ordered array	Binary Tree	Hash Table	Bitmap
Query	$O(N)$	$O(\log_2 N)$	$O(\log_2 N)$	$O(1)$	$O(1)$
Insert	$O(1)$	$O(N)$	$O(\log_2 N)$	$O(1)$	$O(1)$
Space?					

	Unordered array	Ordered array	Binary Tree	Hash Table	Bitmap
Query	$O(N)$	$O(\log_2 N)$	$O(\log_2 N)$	$O(1)$	$O(1)$
Insert	$O(1)$	$O(N)$	$O(\log_2 N)$	$O(1)$	$O(1)$

Space

$> N \cdot \log_2(U)$ bits

	Unordered array	Ordered array	Binary Tree	Hash Table	Bitmap
Query	$O(N)$	$O(\log_2 N)$	$O(\log_2 N)$	$O(1)$	$O(1)$
Insert	$O(1)$	$O(N)$	$O(\log_2 N)$	$O(1)$	$O(1)$
Space	 $> N \cdot \log_2(U) \text{ bits}$				 $U \text{ bits}$

	Full Keys Structures				Bitmap
Query	$O(N)$	$O(\log_2 N)$	$O(\log_2 N)$	$O(1)$	$O(1)$
Insert	$O(1)$	$O(N)$	$O(\log_2 N)$	$O(1)$	$O(1)$
Space	$> N \cdot \log_2(U)$ bits				U bits



Full Keys

**100110
010101
101110**

Bitmap

Space

$N \cdot \log_2(U)$

U

Which to use if we only care about space?



Full Keys

**100110
010101
101110**

Bitmap

Space

$N \cdot \log_2(U)$

U

Which to use if we only care about space?



Full Keys

**100110
010101
101110**

Bitmap

$$N \cdot \log_2(U) = U$$



Full Keys

100110
010101
101110

Bitmap

$$N = \frac{U}{\log_2(U)}$$



Full Keys

100110
010101
101110

Bitmap

$$N < \frac{U}{\log_2(U)}$$

An upward-pointing arrow is positioned above the equation, pointing towards the 'Full Keys' icon.



Full Keys

100110
010101
101110

Bitmap

$$N > \frac{U}{\log_2(U)}$$
A black arrow pointing upwards from the equation to the 'Bitmap' label above.

We're done. Home time



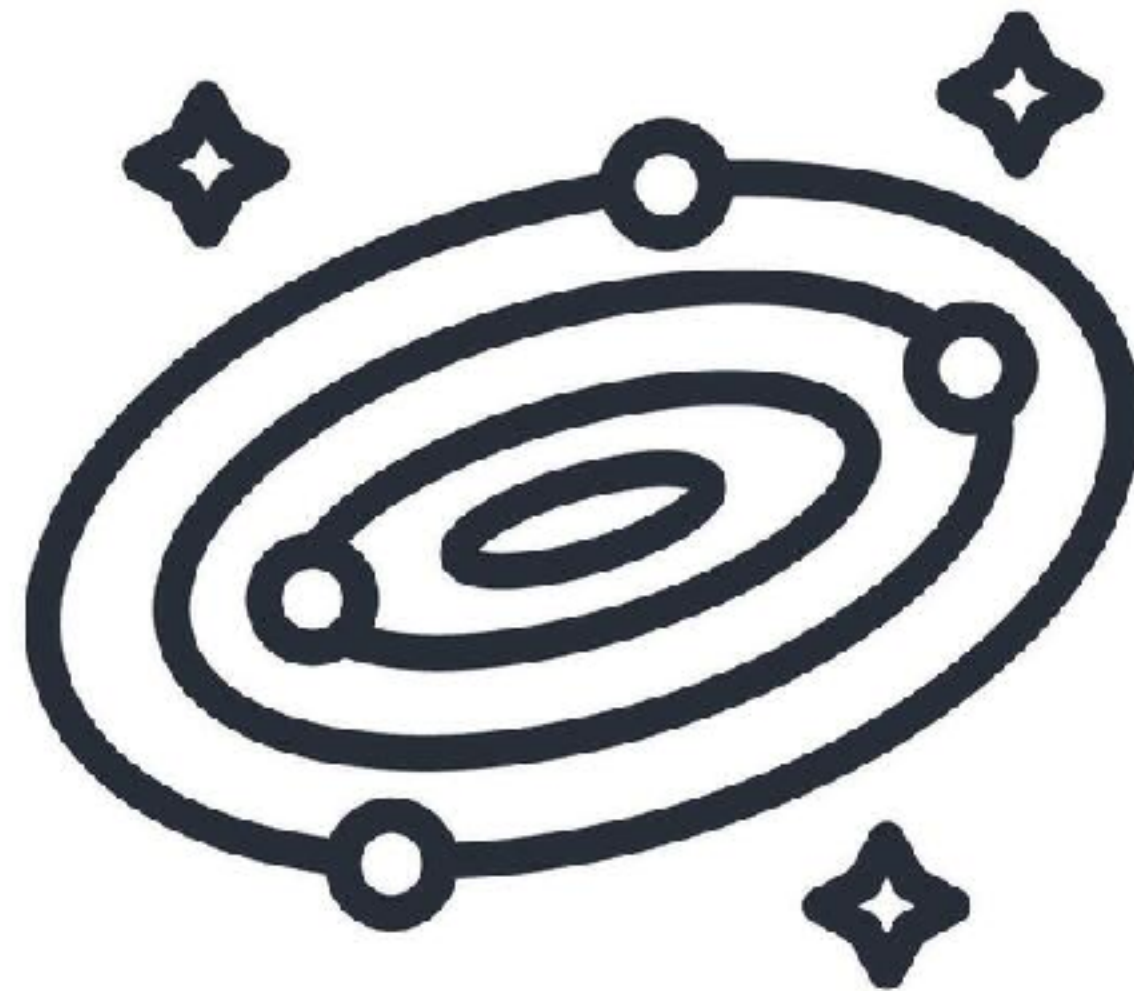
$\min(N \cdot \log_2(U), U)$

Though not quite :)



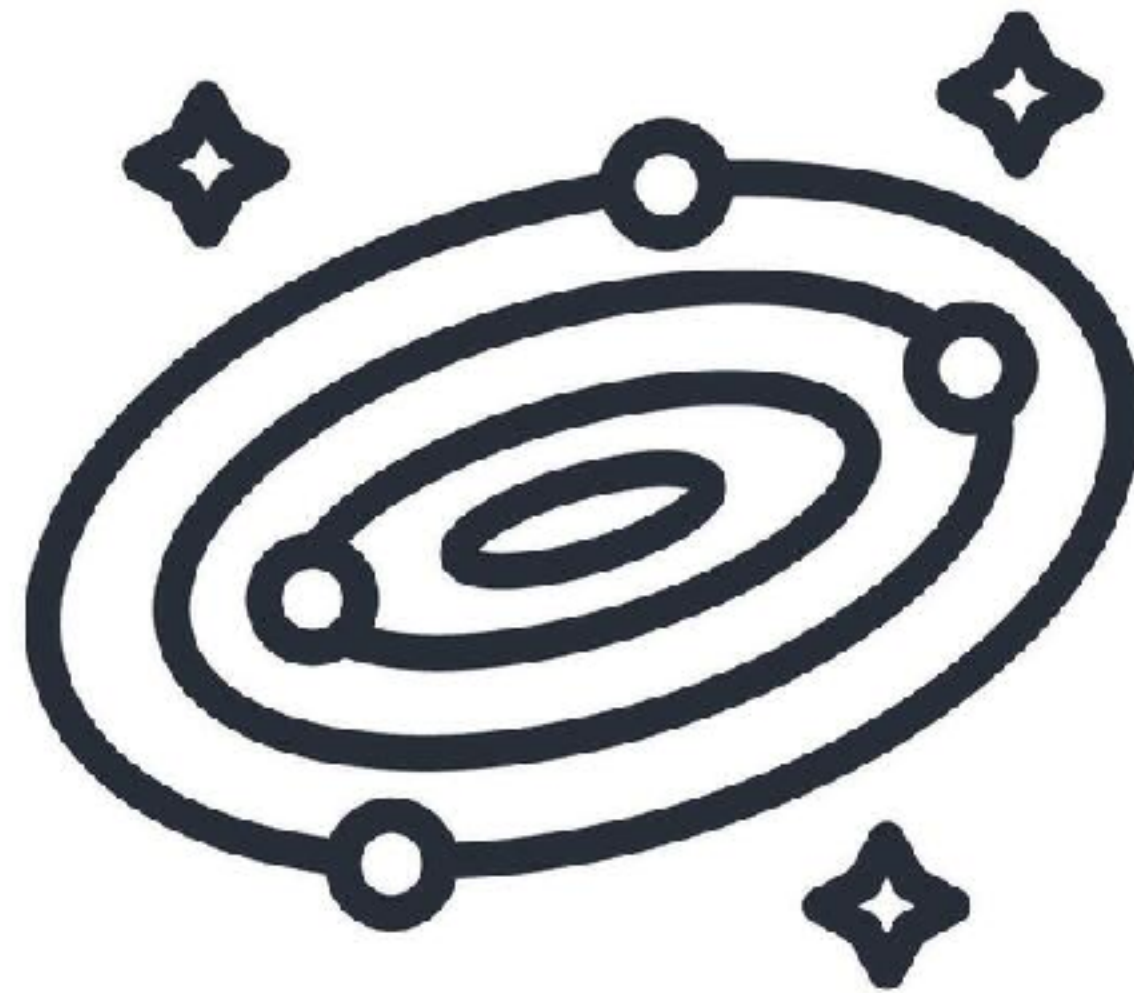
$\min(N \cdot \log_2(U), U)$

Revisit Space Lower Bound for Exact Set

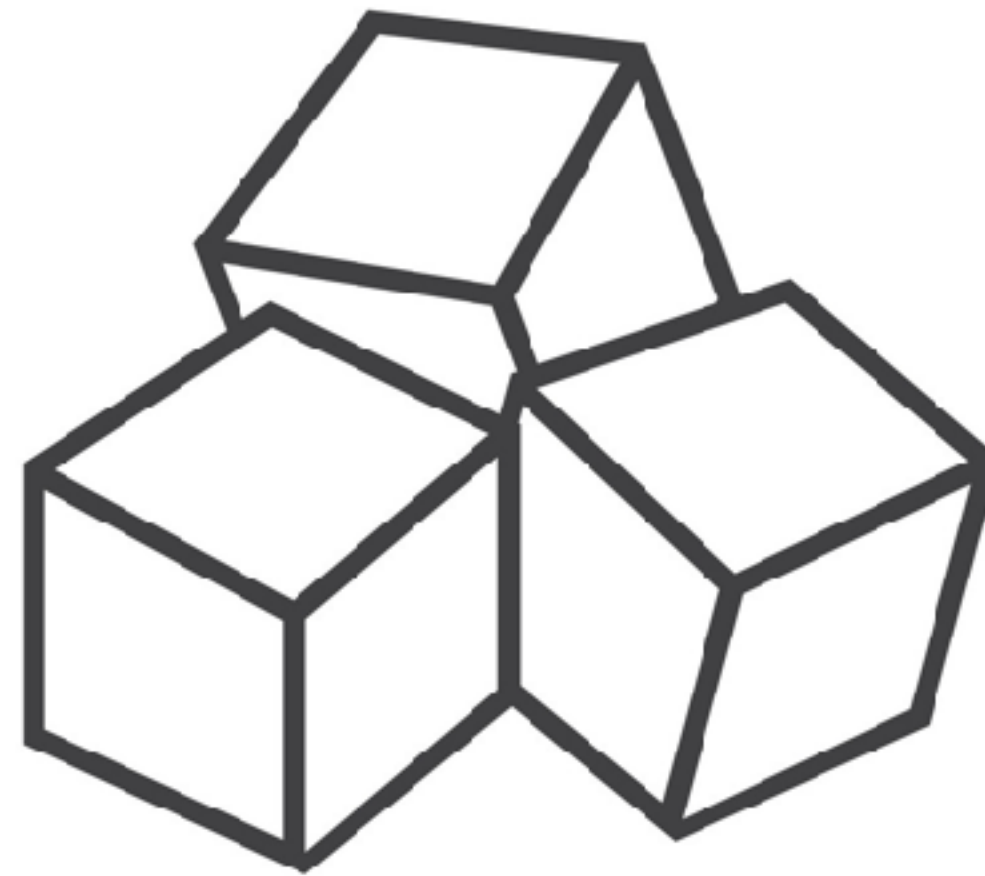


Revisit Space Lower Bound for Exact Set

Out of Universe U , store N entries



(U choose N) combinations



$\log_2(\text{U choose N})$ bits

to encode a unique combination

$$\log_2 \left(\frac{U!}{N! \cdot (U - N)!} \right) \text{ bits}$$

$$\log_2 \left(\frac{U!}{N! \cdot (U - N)!} \right) \text{ bits}$$



How to simplify?



James Stirling
1692 - 1770

$$\log_2 \left(\frac{U!}{N! \cdot (U - N)!} \right) \text{ bits}$$



**Stirling's
approximation:**

$$X! \approx \sqrt{2 \cdot \pi \cdot X} \cdot \left(\frac{X}{e}\right)^X$$

$$\log_2 \left(\frac{U!}{N! \cdot (U - N)!} \right) \text{ bits}$$



Stirling's
approximation:

$$X! \approx \sqrt{2 \cdot \pi \cdot X} \cdot \left(\frac{X}{e}\right)^X$$

$$\log_2 \left(\frac{U!}{N! \cdot (U - N)!} \right) \text{ bits}$$

**Plug in &
simplify**

$$\log_2 \left(\frac{U!}{N! \cdot (U - N)!} \right) \approx \mathbf{N \cdot \log_2(U / N) + N \cdot \log_2(e)}$$



Omitted last time



$$\log_2 \left(\frac{U!}{N! \cdot (U - N)!} \right) \approx N \cdot \log_2(U / N) + \mathbf{N \cdot \log_2(e)}$$



Omitted last time
Important this time



$$\log_2 \left(\frac{U!}{N! \cdot (U - N)!} \right) \approx N \cdot \log_2(U / N) + \mathbf{N \cdot \log_2(e)}$$



Bits / Entry Lower Bound

$$\log_2(U / N) + \log_2(e)$$

Bits / Entry Lower Bound

$$\log_2(U / N) + \mathbf{1.44}$$



Full keys

$\log_2(U)$

100110
010101
101110

Bitmap

U/N



Lower Bound

$\log_2(U / N) + 1.44$



Full keys

$$\log_2(U)$$

100110
010101
101110

Bitmap

$$U/N$$



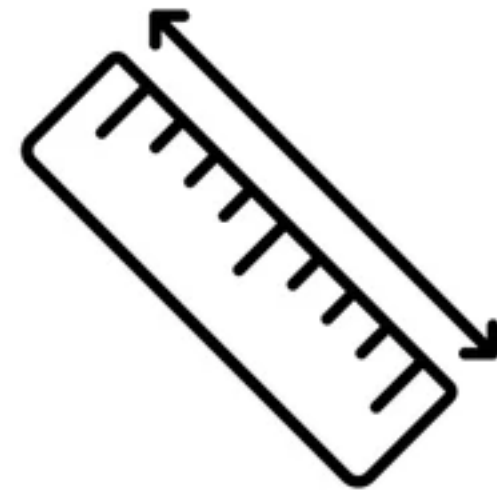
Lower Bound

$$\log_2(U / N) + 1.44$$

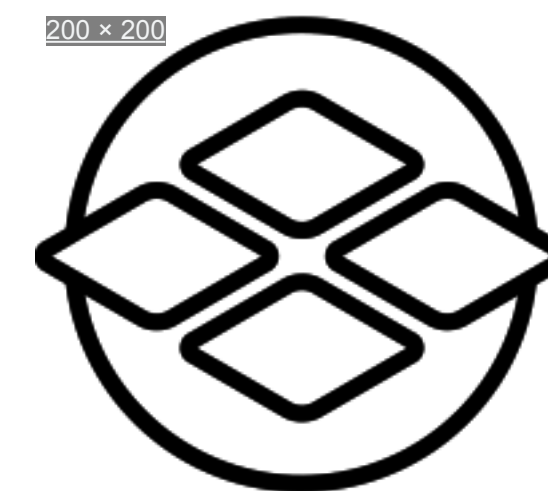
Can we have a data structure that approaches the lower bound for any N and U?

Agenda

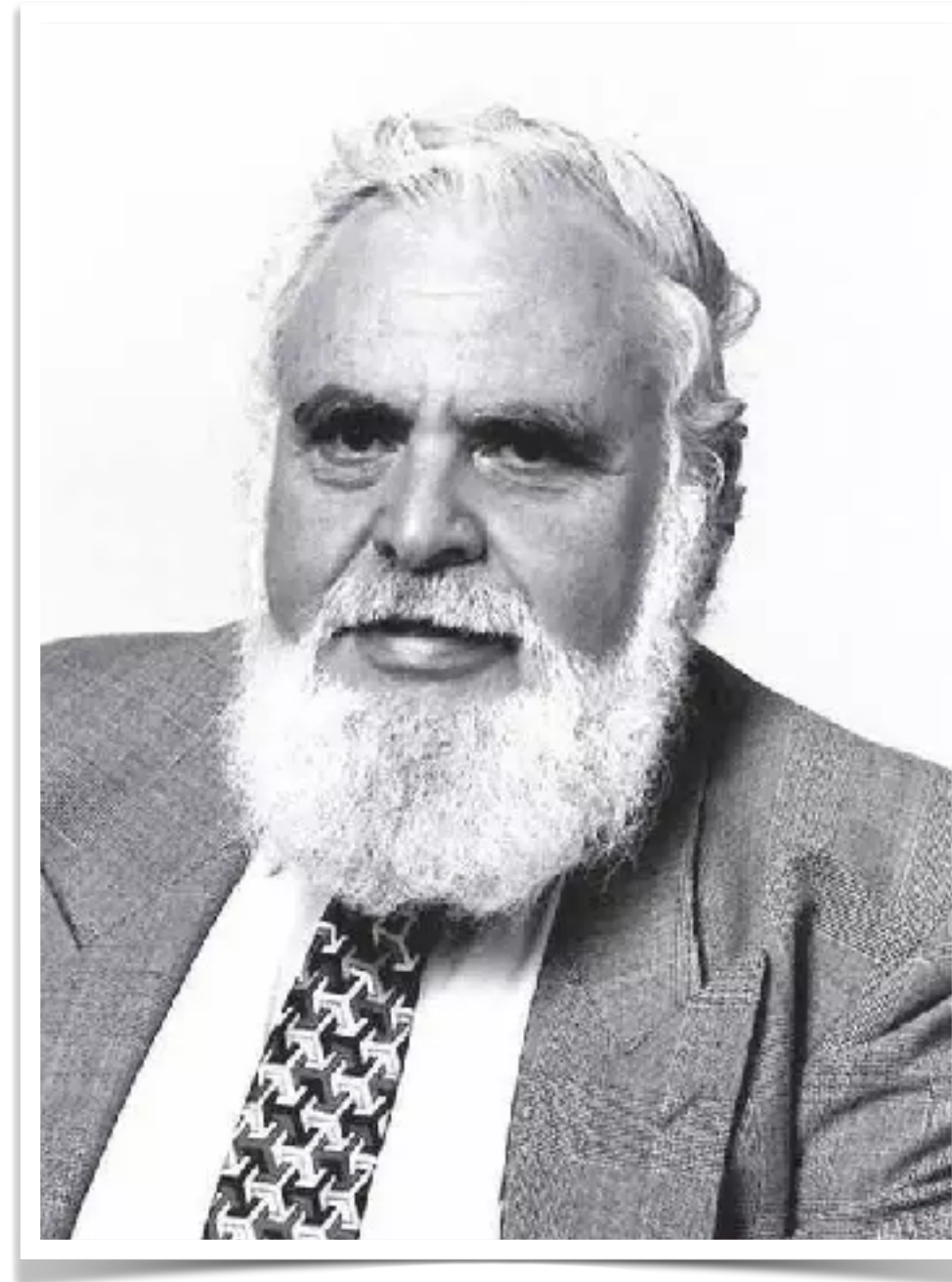
**Run-Length
Golomb Codes**



**Elias-Fano
Encoding**



Golomb Coding (1960s)

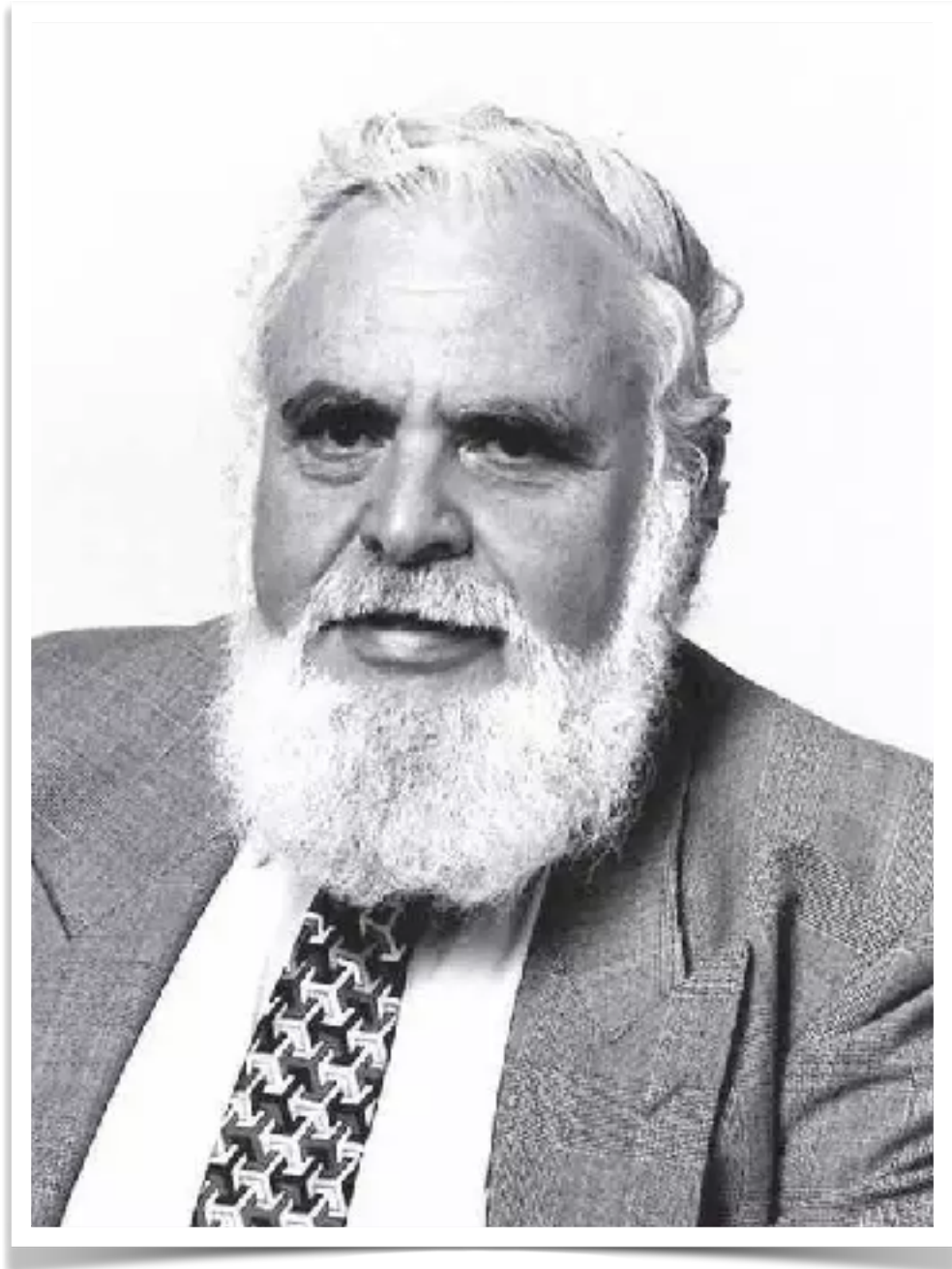


Solomon W. Golomb

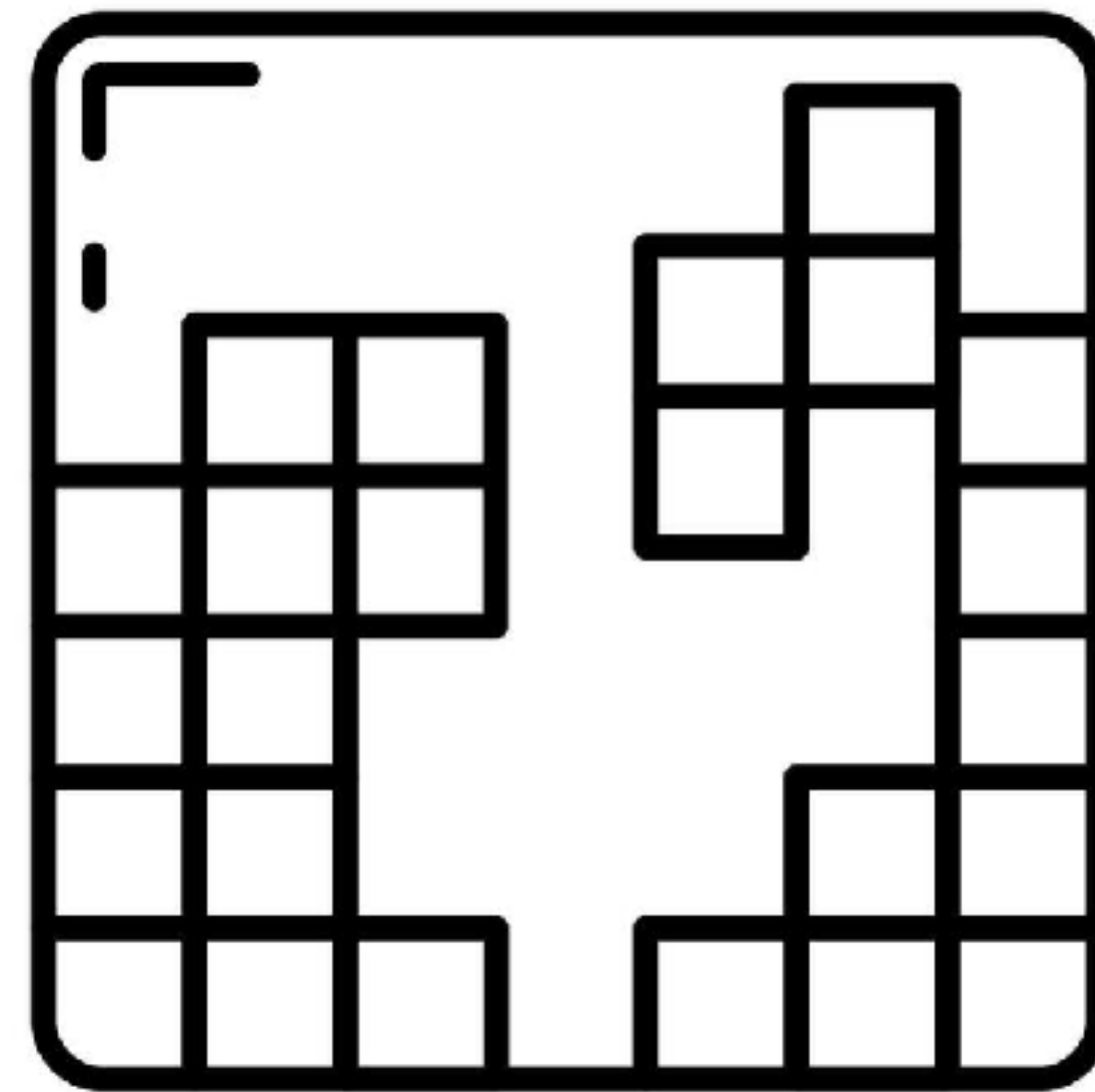
1932 - 2016

USC

Golomb Coding (1960s)



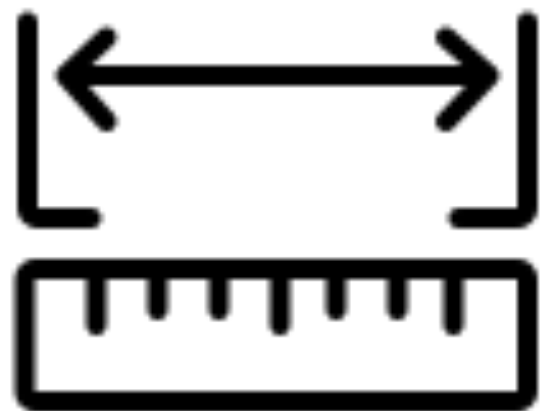
Solomon W. Golomb
1932 - 2016
USC



Was into game design
Inspired Tetris

Golomb Coding

**Encodes var-
length integers**



**Fewer bits for entries
closer to zero**

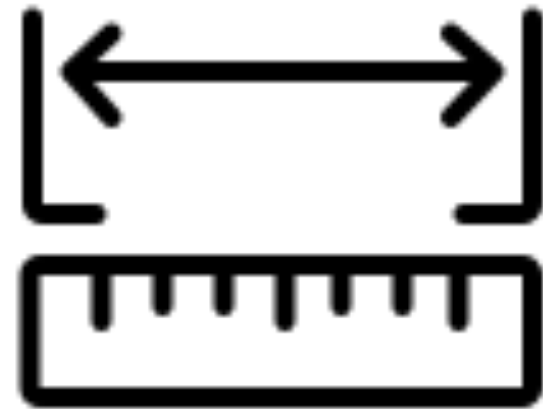


**Good when smaller
values are more
common**



Golomb Coding

Encodes var-length integers



Fewer bits for entries closer to zero



Good when smaller values are more common



Generalizes Unary Codes

Unary Coding Recap

Dec

Unary

0

0

1

10

2

110

3

1110

4

11110

5

111110

6

1111110

Dec	Unary	Freq.
0	0	1/2
1	10	1/4
2	110	1/8
3	1110	1/16
4	11110	1/32
5	111110	1/64
6	1111110	1/128

Dec	Unary	Freq.
0	0	1/2
1	10	1/4
2	110	1/8
3	1110	1/16
4	11110	1/32
5	111110	1/64
6	1111110	1/128

Avg. code length?

Dec	Unary		Freq.
0	0	·	1/2
1	10	·	1/4
2	110	·	1/8
3	1110	·	1/16
4	11110	·	1/32
5	111110	·	1/64
6	1111110	·	1/128
Avg. code length?			2 bits

Dec	Unary		Freq.
-----	-------	--	-------

0	0	·	1/2
---	---	---	-----

1	10	·	1/4
---	----	---	-----

2	110	·	1/8
---	-----	---	-----

3	1110	·	1/16
---	------	---	------

4	11110	·	1/32
---	-------	---	------

5	111110	·	1/64
---	--------	---	------

6	1111110	·	1/128
---	---------	---	-------

Avg. code length?	2 bits	Optimal
-------------------	--------	---------

What if the frequencies decrease stepwise?

Dec

Unary

Freq.

0

0

1

10

2

110

3

1110

4

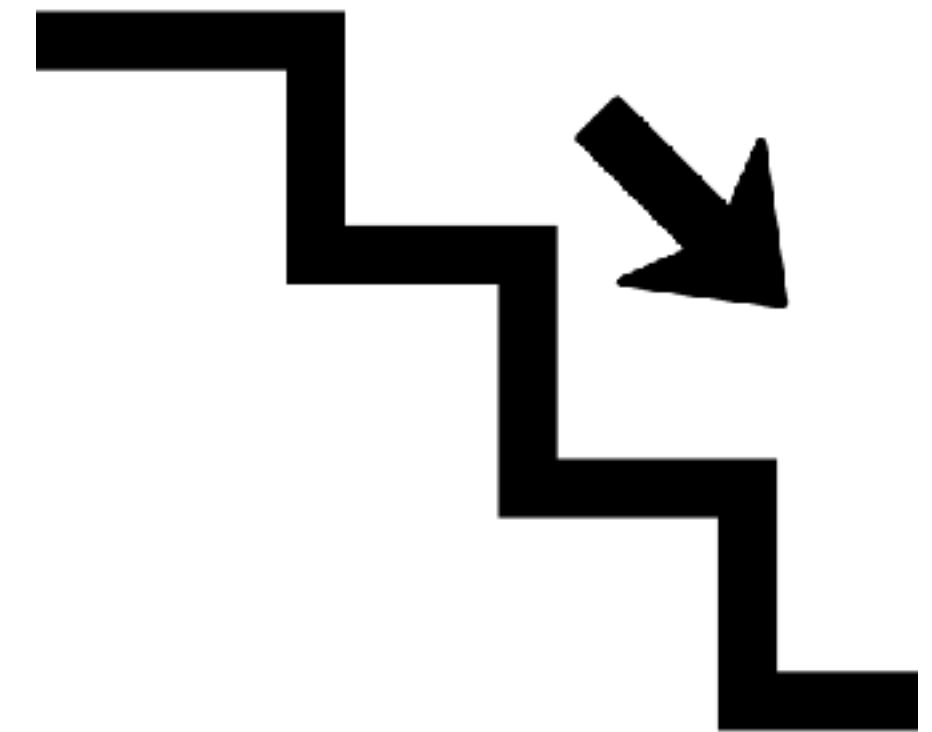
11110

5

111110

6

1111110



What if the frequencies decrease stepwise?

Dec	Unary	Freq.
0	0	1/4
1	10	1/4
2	110	
3	1110	
4	11110	
5	111110	
6	1111110	

What if the frequencies decrease stepwise?

Dec	Unary	Freq.
0	0	1/4
1	10	1/4
2	110	1/8
3	1110	1/8
4	11110	
5	111110	
6	1111110	

What if the frequencies decrease stepwise?

Dec	Unary	Freq.
0	0	1/4
1	10	1/4
2	110	1/8
3	1110	1/8
4	11110	1/16
5	111110	1/16
6	1111110	

What if the frequencies decrease stepwise?

Dec	Unary	Freq.
0	0	1/4
1	10	1/4
2	110	1/8
3	1110	1/8
4	11110	1/16
5	111110	1/16
6	1111110	1/32

What if the frequencies decrease stepwise?

Dec	Unary	Freq.
0	0	1/4
1	10	1/4
2	110	1/8
3	1110	1/8
4	11110	1/16
5	111110	1/16
6	1111110	1/32
Avg. code length?		3.5 bits

Let step size be M (e.g., M = 2)

Dec	Unary	Freq.
0	0	1/4
1	10	1/4
2	110	1/8
3	1110	1/8
4	11110	1/16
5	111110	1/16
6	1111110	1/32

Let step size be M (e.g., **M = 4**)

Dec	Unary	Freq.
0	0	1/8
1	10	1/8
2	110	1/8
3	1110	1/8
4	11110	1/16
5	111110	1/16
6	1111110	1/16

Let step size be M (e.g., **M = 4**)

Dec	Unary	Freq.
0	0	1/8
1	10	1/8
2	110	1/8
3	1110	1/8
4	11110	1/16
5	111110	1/16
6	1111110	1/16
Avg. code length?		6.5 bits

Can you improve on unary for large steps? ($M > 1$)

Dec	Unary	Freq.
0	0	1/4
1	10	1/4
2	110	1/8
3	1110	1/8
4	11110	1/16
5	111110	1/16
6	1111110	1/32

(1) Adjust unary to match step size

Dec	Unary	Freq.
0	0	1/4
1	0	1/4
2	10	1/8
3	10	1/8
4	110	1/16
5	110	1/16
6	1110	1/32

(2) Add binary code to identify items in same step

Dec	Unary	Binary	Freq.
0	0		1/4
1	0		1/4
2	10		1/8
3	10		1/8
4	110		1/16
5	110		1/16
6	1110		1/32

(2) Add binary code to identify items in same step

Dec	Unary	Binary	Freq.
0	0	0	1/4
1	0	1	1/4
2	10	0	1/8
3	10	1	1/8
4	110	0	1/16
5	110	1	1/16
6	1110	0	1/32

Dec	Unary	Binary	Freq.
0	0	0	1/4
1	0	1	1/4
2	10	0	1/8
3	10	1	1/8
4	110	0	1/16
5	110	1	1/16
6	1110	0	1/32

$\log_2(M)$ bits

Dec	Unary	Binary	Freq.
0	0	0	1/4
1	0	1	1/4
2	10	0	1/8
3	10	1	1/8
4	110	0	1/16
5	110	1	1/16
6	1110	0	1/32

Avg. code length?

Dec	Unary	Binary	Freq.
0	0	0	1/4
1	0	1	1/4
2	10	0	1/8
3	10	1	1/8
4	110	0	1/16
5	110	1	1/16
6	1110	0	1/32

Avg. code length?

3 bits

Dec	Unary	Binary	Freq.
0	0	0	1/4
1	0	1	1/4
2	10	0	1/8
3	10	1	1/8
4	110	0	1/16
5	110	1	1/16
6	1110	0	1/32

Avg. code length? **3 bits < 3.5 bits with unary**

How about step size $M = 4$

Dec	Unary	Binary	Freq.
0	0	0	1/8
1	0	1	1/8
2	10	0	1/8
3	10	1	1/8
4	110	0	1/16
5	110	1	1/16
6	1110	0	1/16

How about step size $M = 4$

Dec	Unary	Binary	Freq.
0	0	00	1/8
1	0	01	1/8
2	0	10	1/8
3	0	11	1/8
4	10	00	1/16
5	10	01	1/16
6	10	10	1/16

How about step size $M = 4$

Dec	Unary	Binary	Freq.
0	0	00	1/8
1	0	01	1/8
2	0	10	1/8
3	0	11	1/8
4	10	00	1/16
5	10	01	1/16
6	10	10	1/16

Avg. code length?

4 bits

< 6.5 with unary

Golomb coding integer X given step size M

Unary part

Binary part

Golomb coding integer X given step size M

Unary

$\lfloor X/M \rfloor$

Binary

$X \bmod M$

Golomb coding integer X given step size M

e.g., $X=100$, $M = 16$

Unary

$\lfloor X/M \rfloor$

Binary

$X \bmod M$

Golomb coding integer X given step size M

e.g., $X=100$, $M = 16$

Unary

$\lfloor X/M \rfloor$

$\lfloor 100/16 \rfloor = 6$

1111110

Binary

$X \bmod M$

Golomb coding integer X given step size M

e.g., $X=100$, $M = 16$

Unary

$\lfloor X/M \rfloor$

$\lfloor 100/16 \rfloor = 6$

1111110

Binary

$X \bmod M$

$100 \bmod 16 = 4$

0100

Golomb coding integer X given step size M

e.g., $X=100$, $M = 16$

Unary

$\lfloor X/M \rfloor$

Binary

$X \bmod M$

code: 11111100100

What if step is not a power of 2? e.g., $M = 3$

What if step is not a power of 2? e.g., $M = 3$

Unary

Binary

0

00

0

01

0

10

10

00

10

01

10

10

110

00

What if step is not a power of 2? e.g., M = 3

Unary

Binary

Freq.

0

00

1/6

0

01

1/6

0

10

1/6

10

00

1/12

10

01

1/12

10

10

1/12

110

00

1/24

Avg. code length:

4 bits

What if step is not a power of 2? e.g., $M = 3$

Unary	Binary
0	00
0	01
0	10
10	00
10	01
10	10
110	00

Are we being wasteful?

Unary

Binary

0

00

0

01

0

10

10

00

10

01

10

10

110

00

**2 bits to represent 3
symbols per step**

Binary

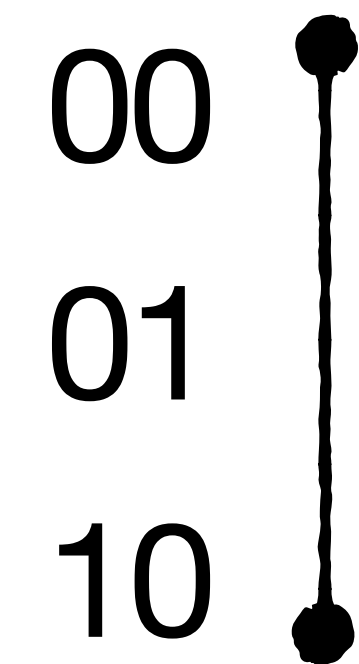
00
01
10



2 bits to represent 3
symbols per step

$\lceil \log_2(M) \rceil$

Binary



00

01

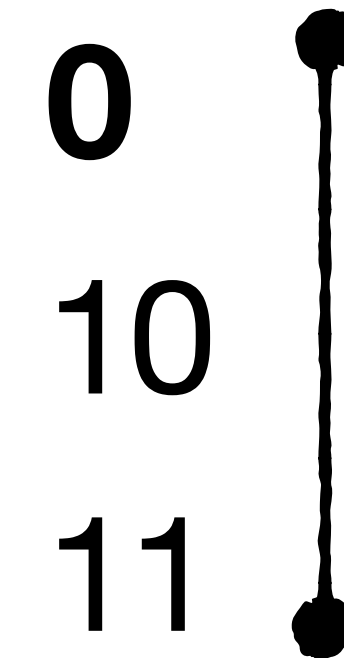
10

2 bits to represent 3
symbols per step

$\lceil \log_2(M) \rceil$

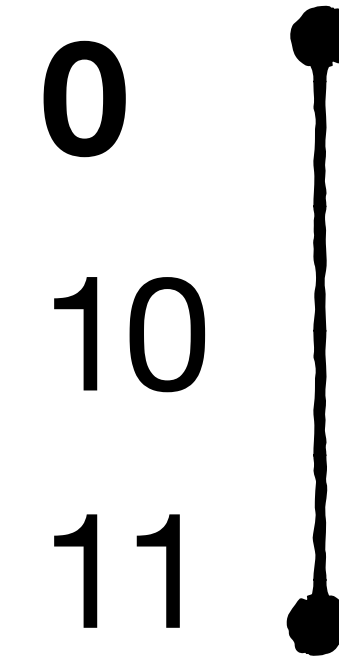
**Can we optimize
ceiling away?**

Binary



$\lceil \log_2(M) \rceil$

Binary



0

10

11

$5/3 = 1.66$ bits / code

$\lceil \log_2(M) \rceil$

Binary

0

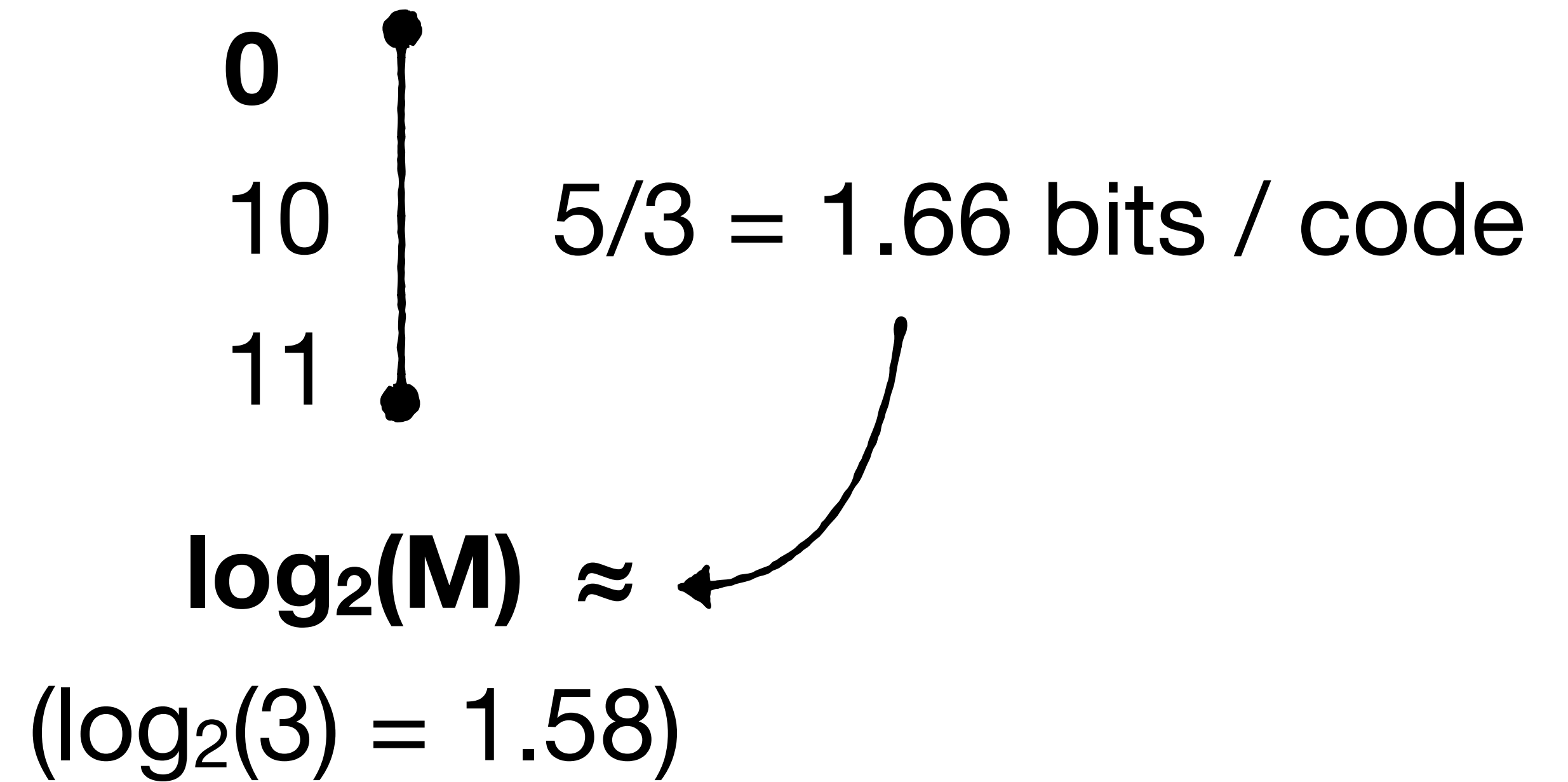
10

11

$5/3 = 1.66 \text{ bits / code}$

$\log_2(M) \approx$

$(\log_2(3) = 1.58)$



Truncated Binary Encoding (e.g., $M = 3$)

Dec

0

1

2

0

10

11

Dec	Truncated Binary Encoding (e.g., $M = 3$)
0	0
1	10
2	11

**Crucial that no code is prefix of
another code. Why?**

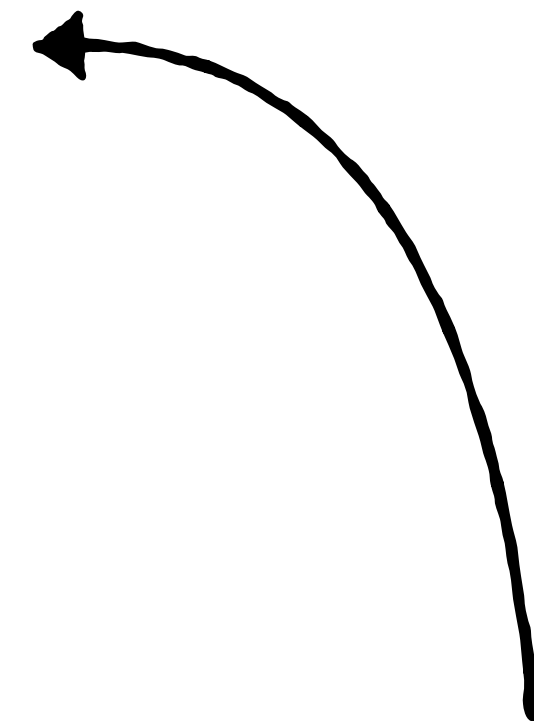
Dec	Truncated Binary Encoding (e.g., $M = 3$)
0	0
1	10
2	11

Crucial that no code is prefix of
another code. Why?

To decode unambiguously

Truncated Binary Encoding (e.g., $M = 3$)

Dec	
0	0
1	01
2	11



This would make things harder :)

Dec	Truncated Binary Encoding (e.g., $M = 3$)
0	0
1	10
2	11

Lets see examples for different step sizes

Truncated Binary Encoding (e.g., $M = 4$)

Dec

0

00

1

01

2

10

3

11

Truncated Binary Encoding (e.g., $M = 5$)

Dec

0

00

1

01

2

10

3

110

4

111

Truncated Binary Encoding (e.g., $M = 6$)

Dec

0

00

1

01

2

100

3

101

4

110

5

111

Truncated Binary Encoding (e.g., $M = 7$)

Dec

0

00

1

010

2

011

3

100

4

101

5

110

6

111

Truncated Binary Encoding (e.g., $M = 8$)

Dec

0

000

1

001

2

010

3

011

4

100

5

101

6

110

8

111

Back to example with $M = 3$

Unary	Binary	Freq.
0	0	$1/6$
0	10	$1/6$
0	11	$1/6$
10	0	$1/12$
10	10	$1/12$
10	11	$1/12$
110	0	$1/24$

Back to example with $M = 3$

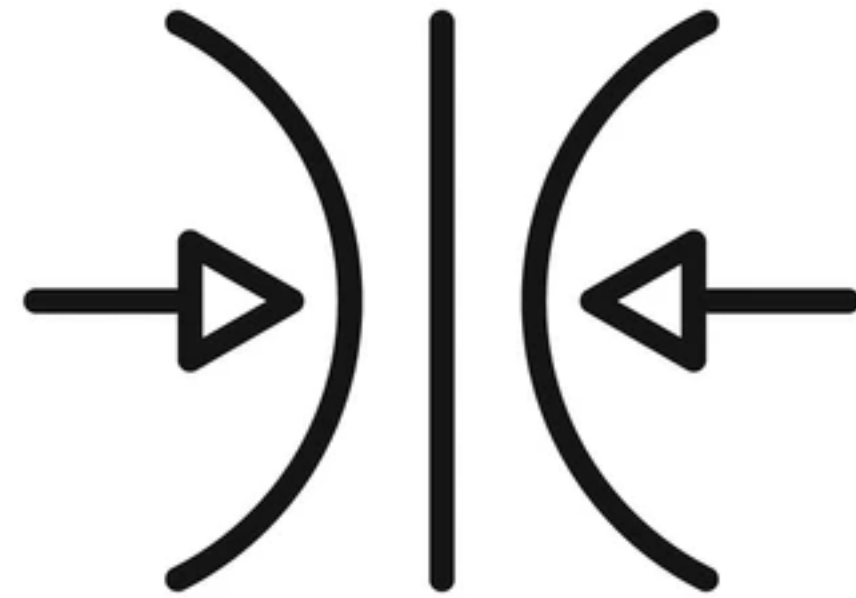
Unary	Binary	Freq.
0	0	1/6
0	10	1/6
0	11	1/6
10	0	1/12
10	10	1/12
10	11	1/12
110	0	1/24

Avg. code length?

3.66 bits

< 4 bit

How to use Golomb Coding for Exact Set Compression?



How to use Golomb Coding for Exact Set Compression?

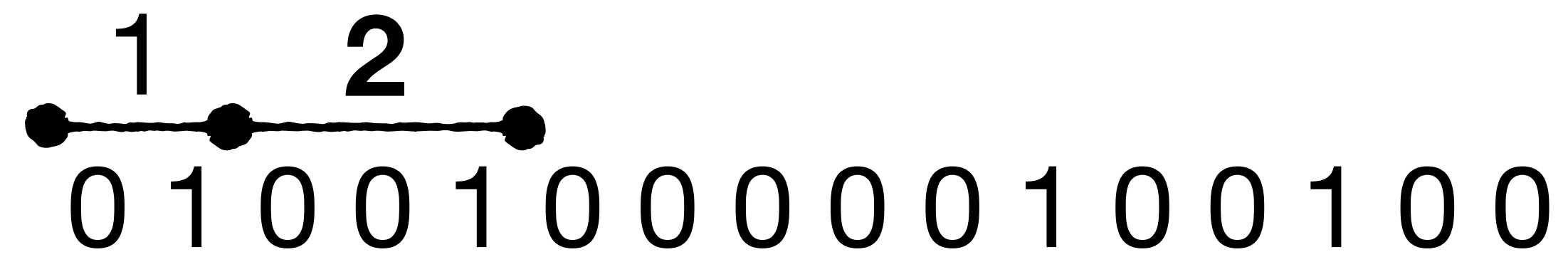
0 1 0 0 1 0 0 0 0 0 1 0 0 1 0 0

View set as bitmap

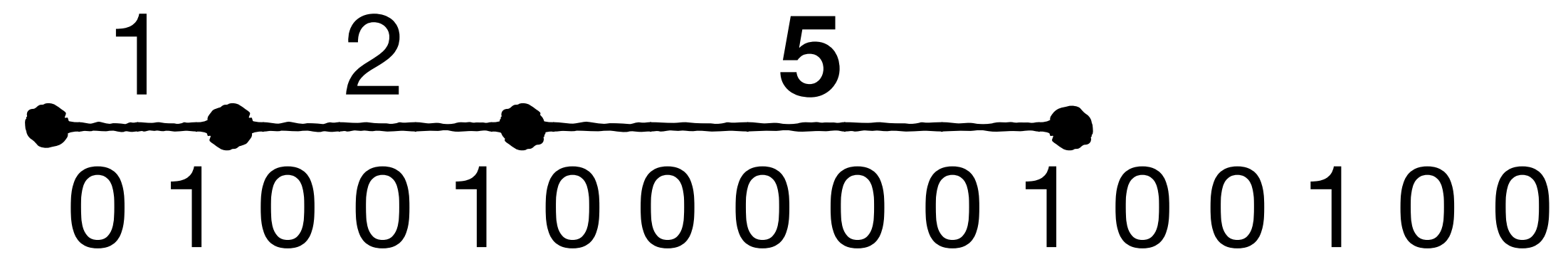
Encode distances between 1s

1
●————●
0 1 0 0 1 0 0 0 0 0 1 0 0 1 0 0

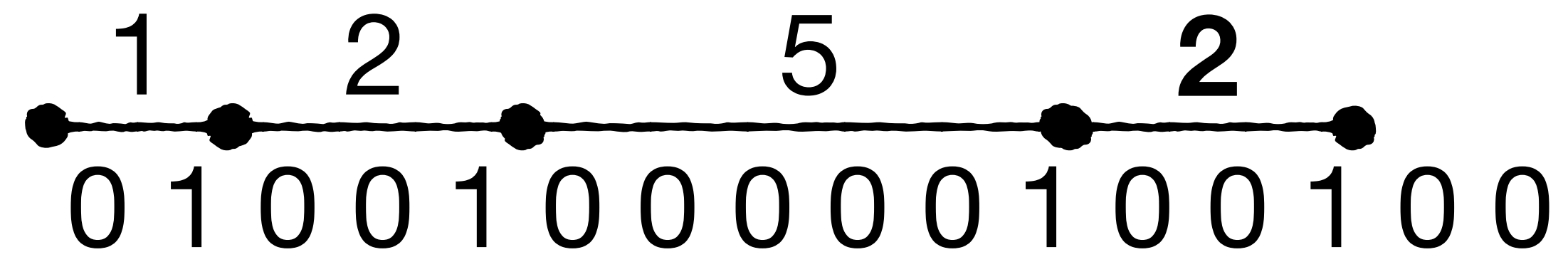
Encode distances between 1s



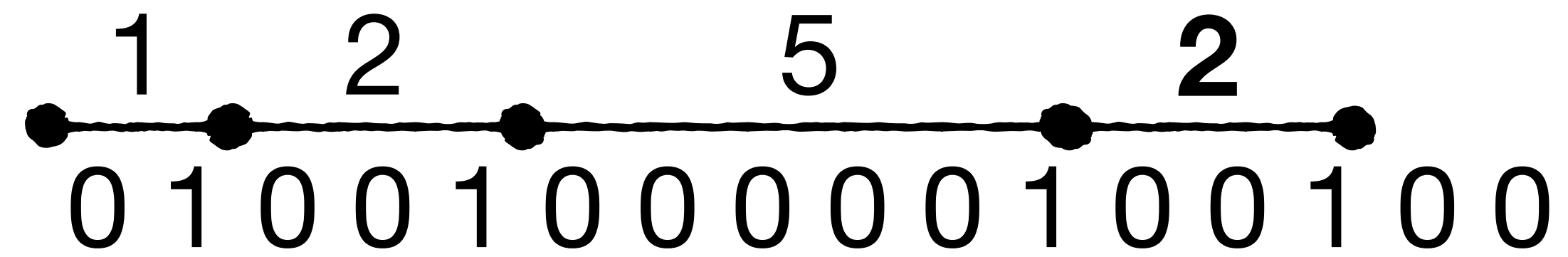
Encode distances between 1s



Encode distances between 1s

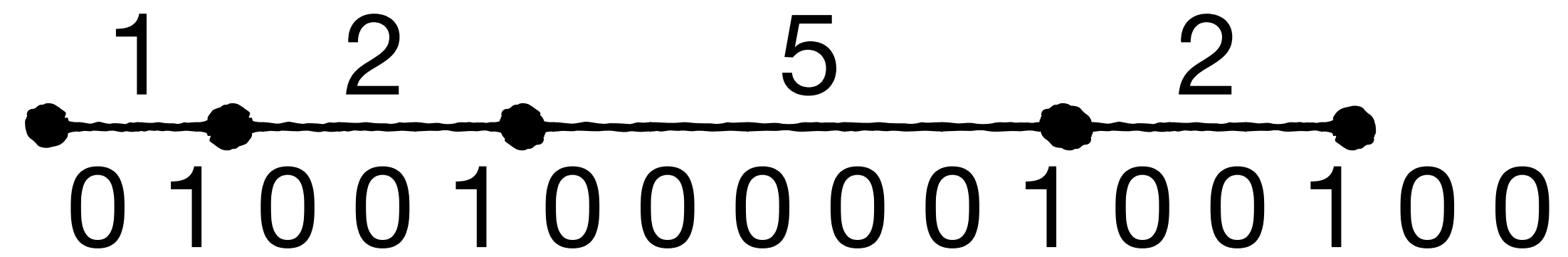


Encode distances between 1s



Run-Length Encoding

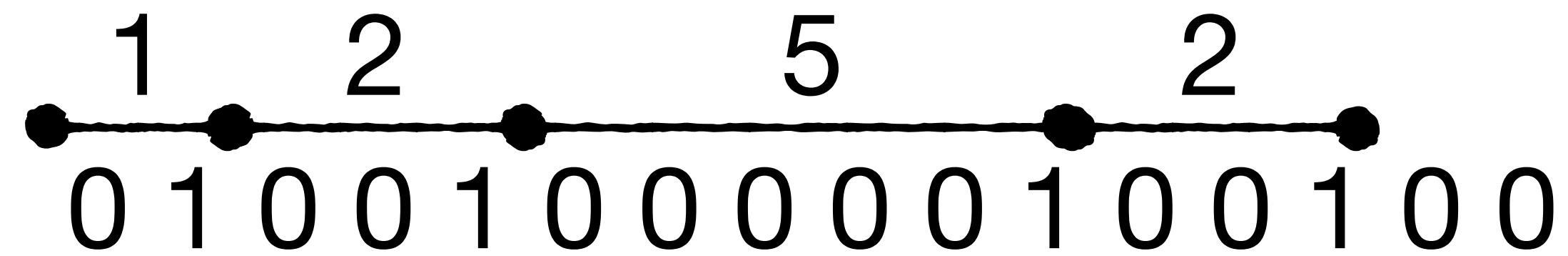
Use Golomb encoding with $M = U/N$



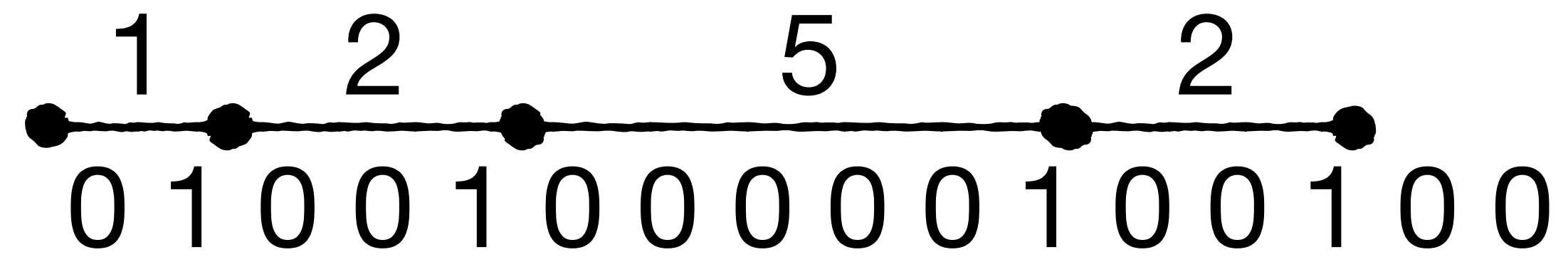
Avg. distance between 1s



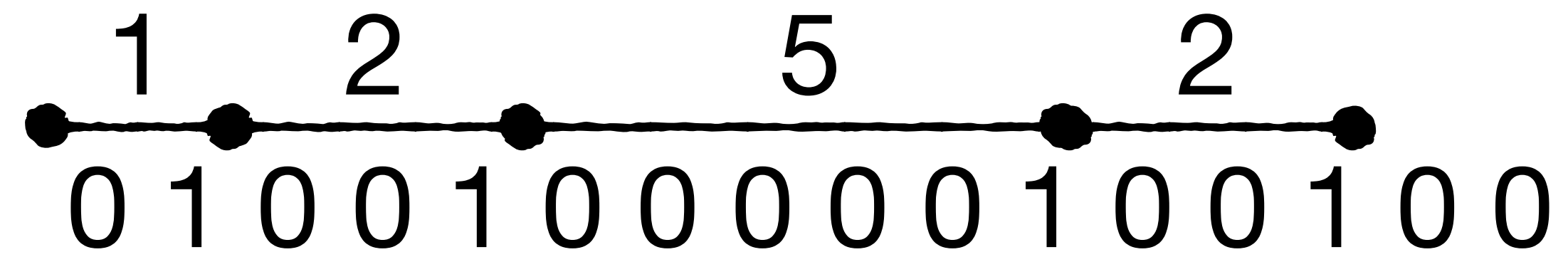
Use Golomb encoding with $M = \mathbf{U/N}$



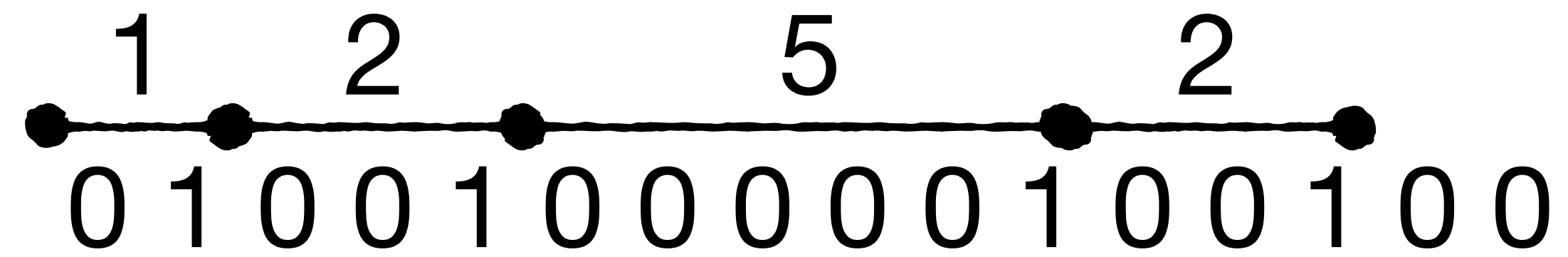
Use Golomb encoding with **$M = U/N = 16 / 4 = 4$**



Use Golomb encoding with **$M = U/N = 4$**

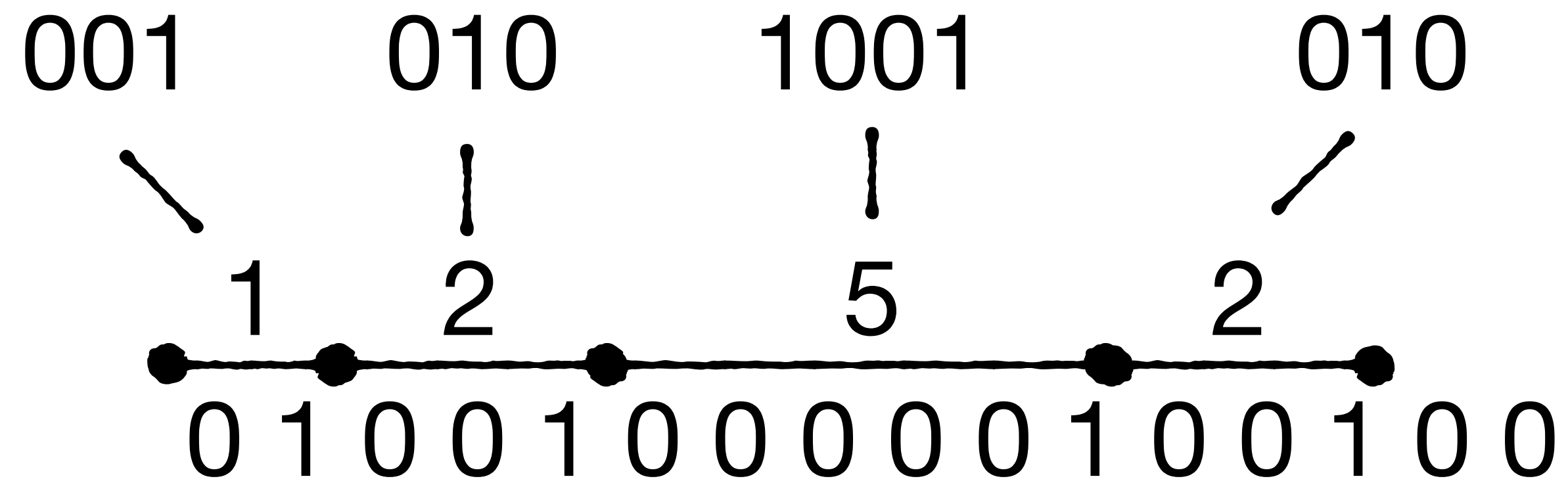


Use Golomb encoding with **M** = **U/N** = 4



Dec	Golomb
0	000
1	001
2	010
3	011
4	1000
5	1001

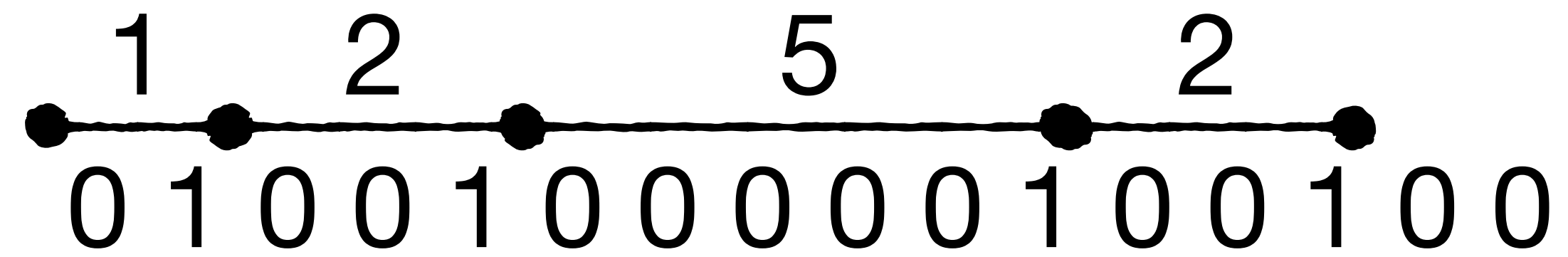
Use Golomb encoding with **M = U/N = 4**



Dec	Golomb
0	000
1	001
2	010
3	011
4	1000
5	1001

Use Golomb encoding with **$M = U/N = 4$**

0010101001010



Use Golomb encoding with **$M = U/N = 4$**

13 bits

0010101001010

16 bits

0100100000100100

Avg. code length with respect to N and U?

Unary?

Binary?

Avg. code length with respect to N and U?

Unary?

Binary?

$$\approx \log_2(U/N)$$

By definition of how we set M

Avg. code length with respect to N and U?

Unary?

≈ 1.5

Binary?

$\approx \log_2(U/N)$

Assuming set bits are evenly spaced

Avg. code length with respect to N and U?

Unary?

≈ 1.5

Binary?

$\approx \log_2(U/N)$

Assuming set bits are evenly spaced

Most common unary codes are 0 and 10

Avg. code length with respect to N and U?

Unary?

≈ 1.5

Binary?

$\approx \log_2(U/N)$

Assuming set bits are evenly spaced

Most common unary codes are 0 and 10

Avg. of 1 and 2 bits is 1.5

Avg. code length with respect to N and U?

Unary?

≈ 1.5



Skew may bring this up to 2

Binary?

$\approx \log_2(U/N)$

	Golomb	Lower Bound
Bits /entry	$\approx 1.5 + \log_2(U/N)$	$1.44 + \log_2(U/N)$

Golomb

Lower Bound

Bits /entry

$$\approx 1.5 + \log_2(U/N)$$

$$1.44 + \log_2(U/N)$$

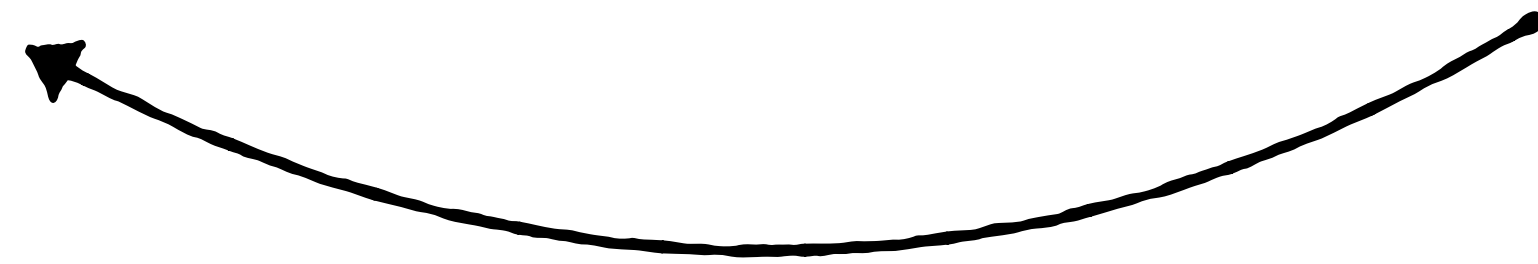
Almost hitting lower bound for any N and U :)

Full keys

Golomb

$\log_2(U)$

$\approx 1.5 + \log_2(U/N)$



**Approaches this
for small N**

Full keys

Golomb

Bitmap

$\log_2(U)$

$\approx 1.5 + \log_2(U/N)$

U/N



**Approaches this
for large N**

Enables trade-offs in-between :)

Full keys

Golomb

Bitmap

$\log_2(U)$

$\approx 1.5 + \log_2(U/N)$

U/N

Problem?



Golomb

$$\approx 1.5 + \log_2(U/N)$$

Problem? Checking if element at given offset takes $O(N)$ time



Golomb

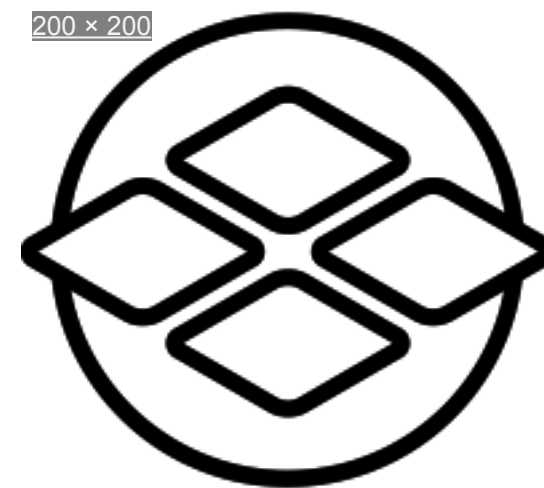
$$\approx 1.5 + \log_2(U/N)$$

Agenda

**Golomb
Coding**



**Elias - Fano
Encoding**



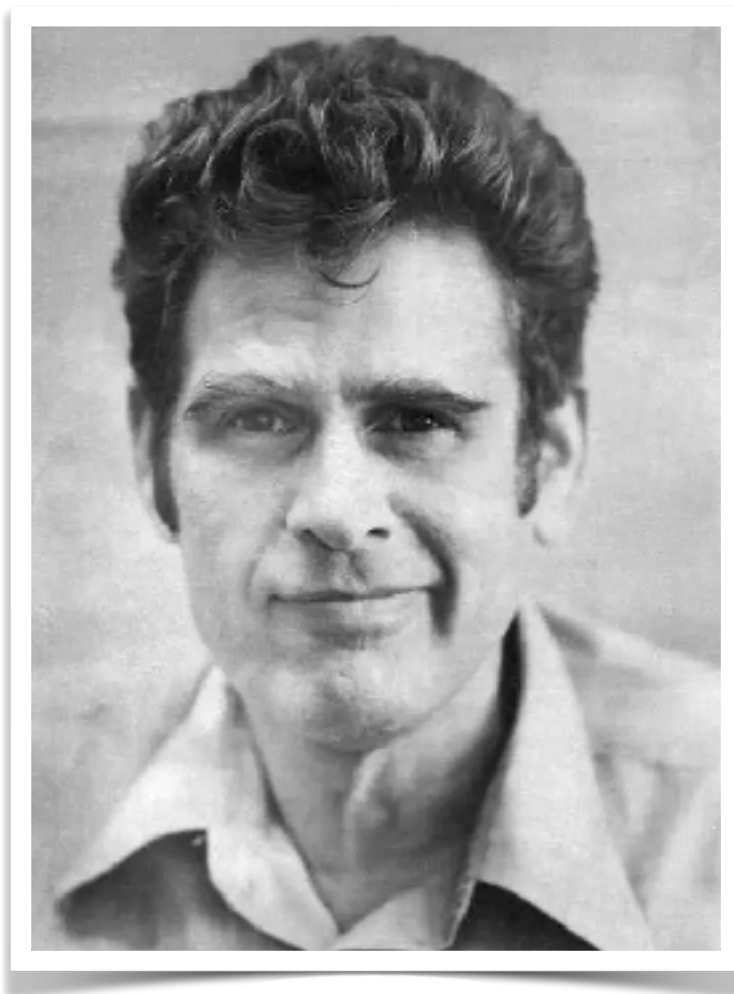
**Roaring
Bitmaps**



Peter Elias

Efficient Storage and Retrieval by
Content and Address of Static Files

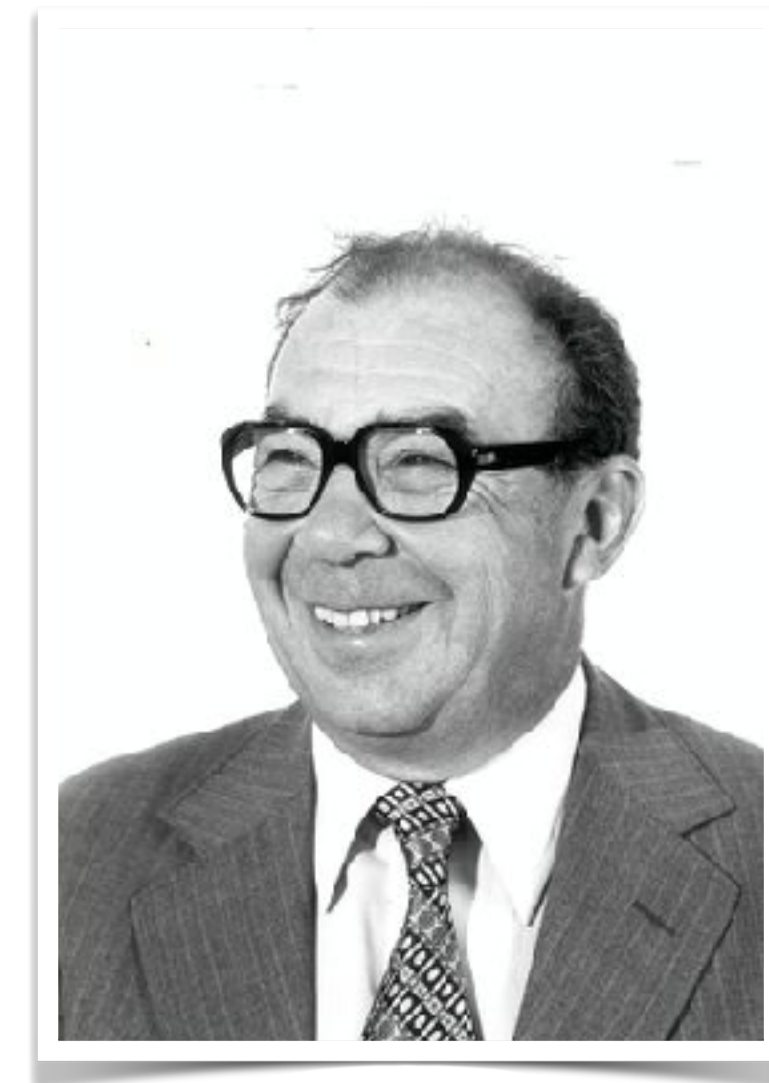
1974



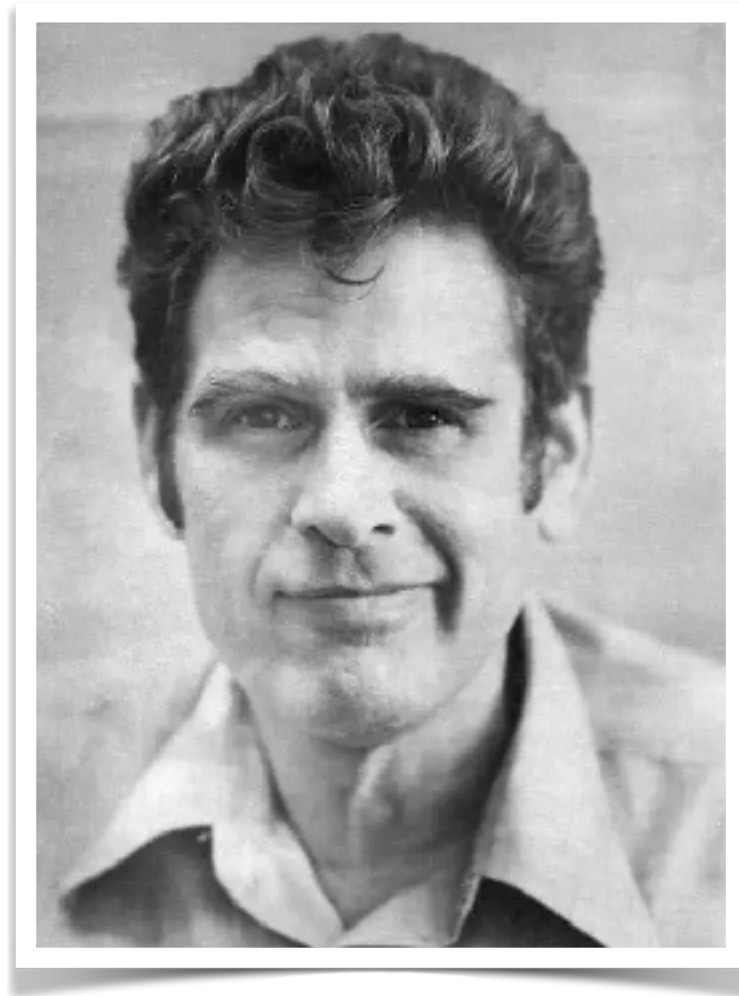
Robert Mario Fano

On the Number of Bits Required to
Implement an Associative Memory.

1971

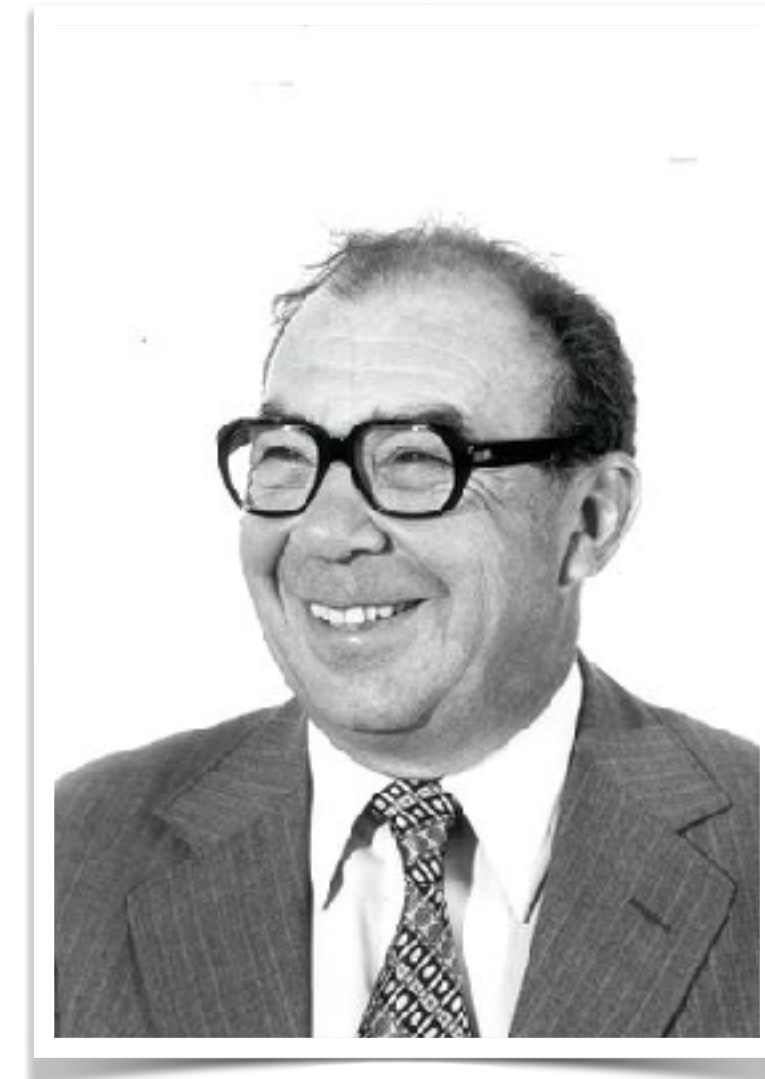


Peter Elias



1923 - 2001
MIT

Robert Mario Fano



1917 - 2016
MIT

Pioneers of information theory

Encoding Intuition

Encoding Intuition

Decimal

Binary

1

0001

2

0010

12

1100

15

1111

Decimal

Binary

1

00 01

2

00 10

12

11 00

15

11 11



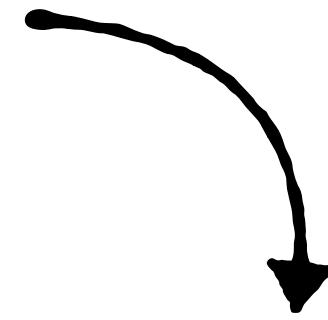
High-order bits repeat

Decimal	Binary
0	000 0
1	000 1
2	001 0
3	001 1
12	110 0
13	110 1
14	111 0
15	111 1



With denser keys, more repetition

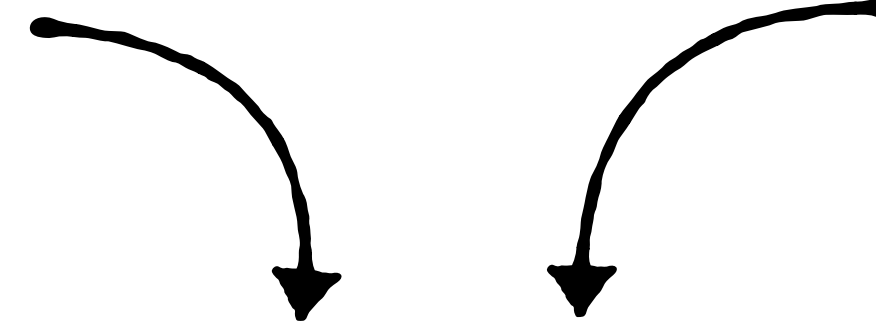
**Encode prefix
implicitly
(Compress)**



000	0
000	1
001	0
001	1
110	0
110	1
111	0
111	1

**Encode prefix
implicitly
(Compress)**

**Encode suffix
Explicitly**



000	0
000	1
001	0
001	1
110	0
110	1
111	0
111	1

Encoding

Partition universe based on first $\lceil \log_2(N) \rceil$ bits of keys

Encoding

Partition universe based on first $\lceil \log_2(N) \rceil$ bits of keys

creates P partitions, where $2^N > P \geq N$ and P is power of 2

Encoding

Partition universe based on first $\lceil \log_2(N) \rceil$ bits of keys
creates P partitions, where $2N > P \geq N$ and P is power of 2



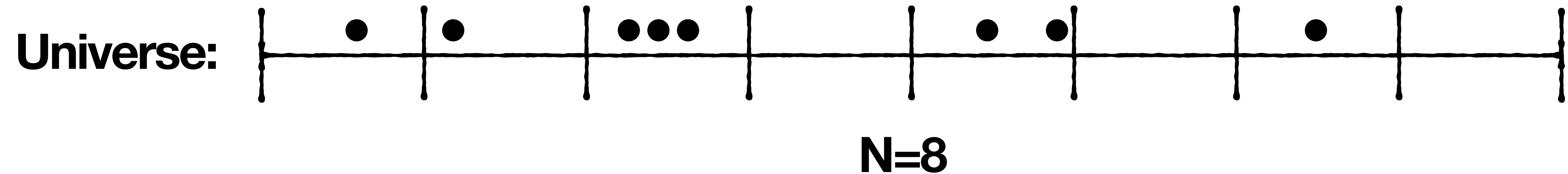
Encoding

Partition universe based on first $\lceil \log_2(N) \rceil$ bits of keys
creates P partitions, where $2N > P \geq N$ and P is power of 2

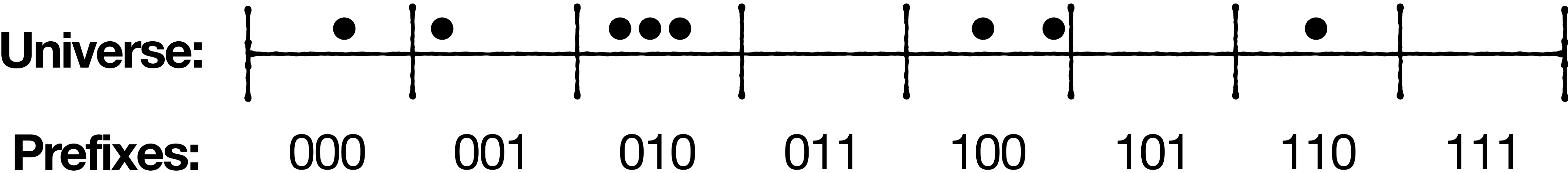


Encoding

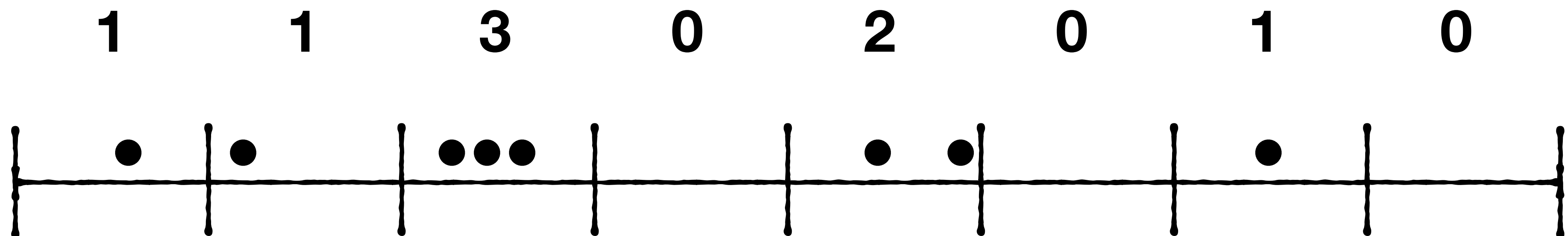
Partition universe based on first $\lceil \log_2(N) \rceil$ bits of keys
creates P partitions, where $2N > P \geq N$ and P is power of 2



$\log_2(P)$ bits partition ID corresponds to prefix of entries in the partition

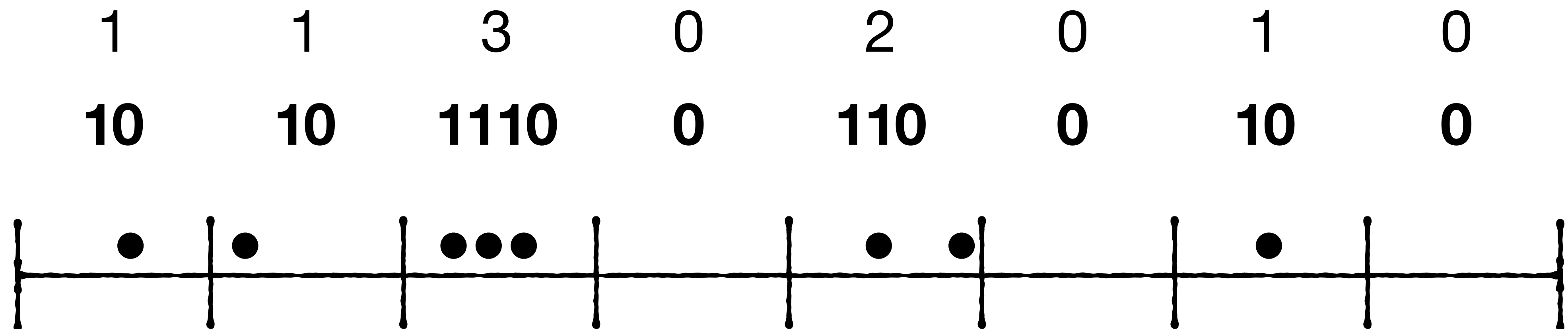


Consider # entries in each partition

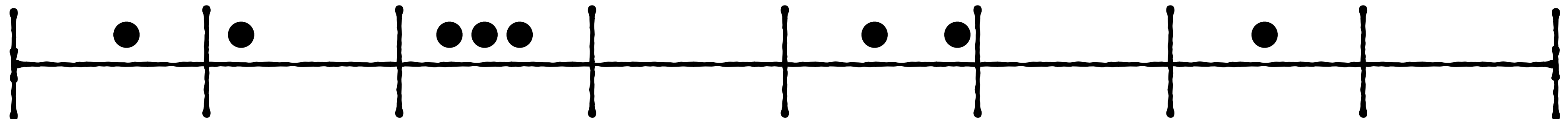


Consider # entries in each partition

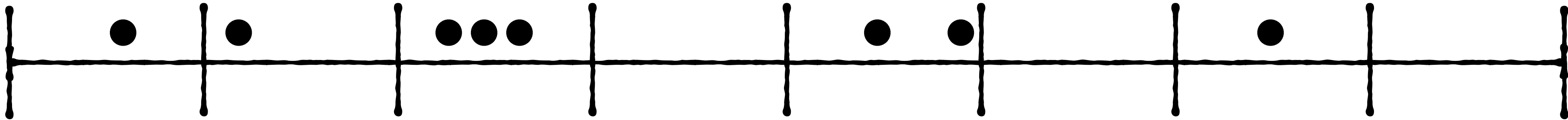
Encode in Unary



Store contiguously: 10 10 1110 0 110 0 10 0

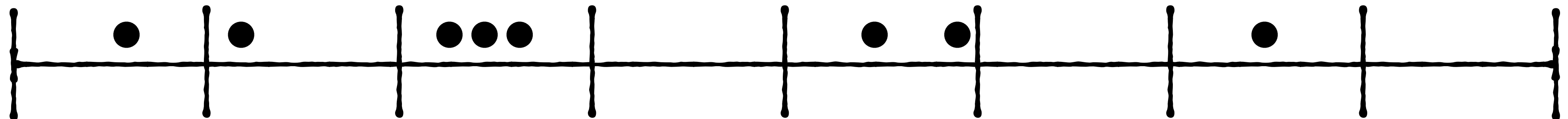


Store contiguously: **P + N bits**
10 10 1110 0 110 0 10 0



Store contiguously: $P + N$ bits
10 10 11 0 0 1 1 0 0 1 0 0

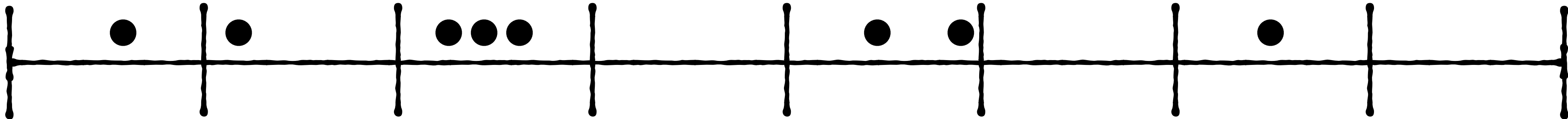
0s = # partition



Store contiguously: $P + N$ bits
10 10 1110 0 110 0 10 0

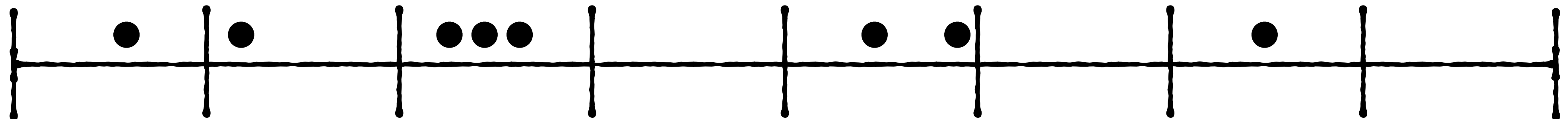
0s = # partitions

1s = # entries



unary prefix counts

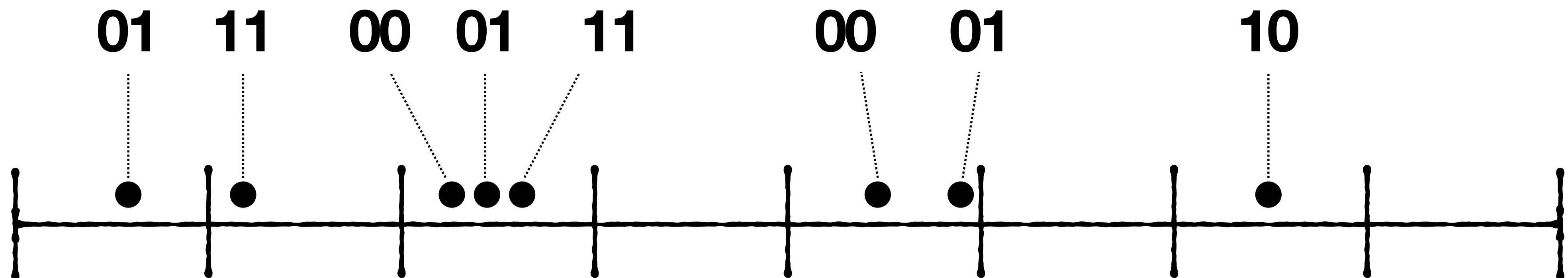
10 10 1110 0 110 0 10 0



unary prefix counts

10 10 1110 0 110 0 10 0

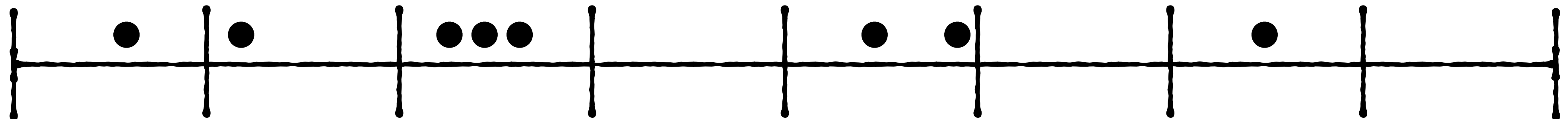
suffixes:



unary prefix counts

10 10 1110 0 110 0 10 0

Store contiguously: 01 11 00 01 11 00 01 10

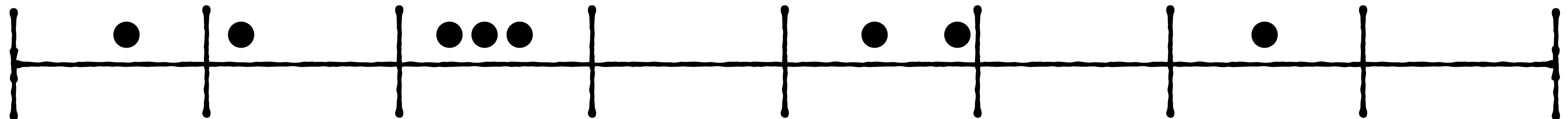


unary prefix counts

10 10 1110 0 110 0 10 0

binary suffixes

01 11 00 01 11 00 01 10

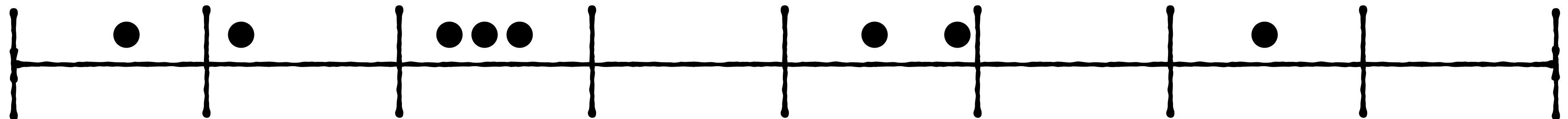


unary prefix counts

10 10 1110 0 110 0 10 0

binary suffixes: $\log_2(\mathbf{U}) - \log_2(\mathbf{P})$

01 11 00 01 11 00 01 10

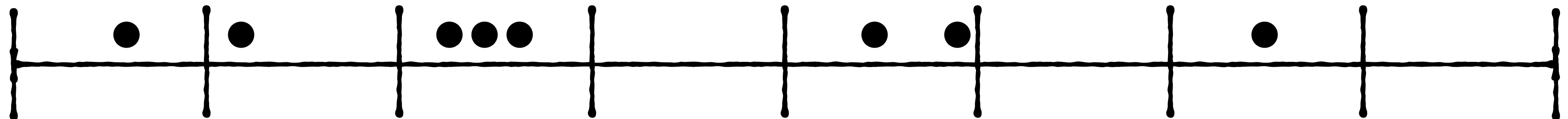


unary prefix counts

10 10 1110 0 110 0 10 0

binary suffixes: $\log_2(\mathbf{U/P})$

01 11 00 01 11 00 01 10

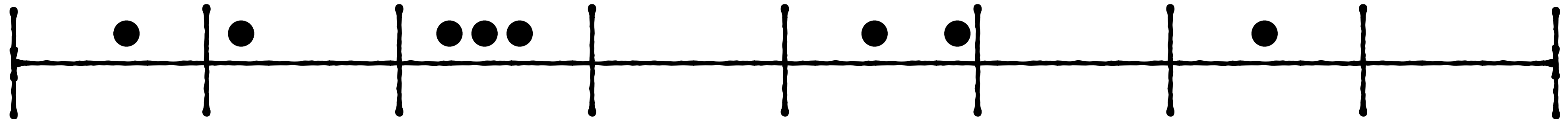


unary prefix counts

binary suffixes

Concatenate:

10 10 1110 0 110 0 10 0 : 01 11 00 01 11 00 01 10



unary prefix counts

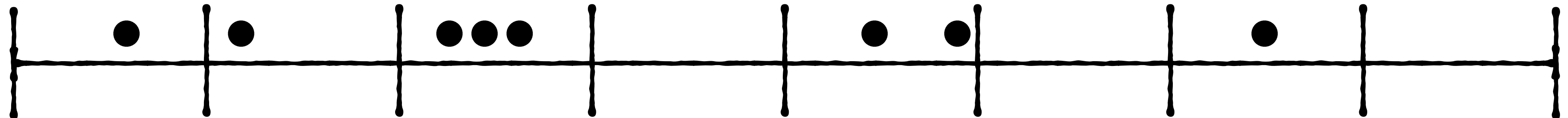
binary suffixes

10 10 1110 0 110 0 10 0 : 01 11 00 01 11 00 01 10

Size (bits):

$P + N$

$N \cdot \log_2(U/P)$



unary prefix counts

binary suffixes

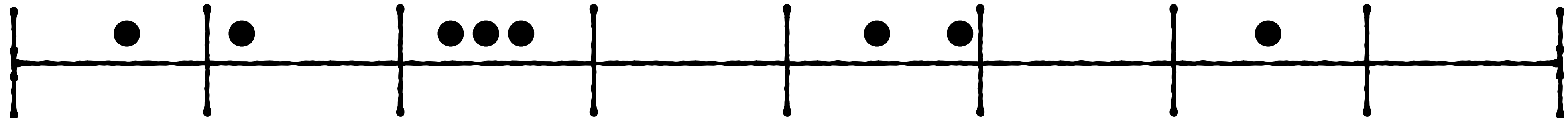
10 10 1110 0 110 0 10 0 : 01 11 00 01 11 00 01 10

Size (bits):

$P + N$

$N \cdot \log_2(U/P)$

This is precise! Not an average :)



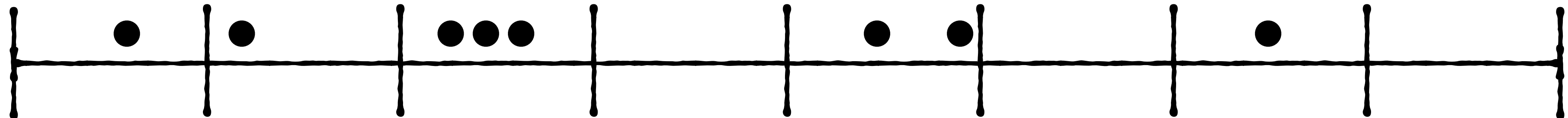
Can fit in a fixed amount of space allocated in advance

$\left| 10\ 10\ 1110\ 0\ 110\ 0\ 10\ 0 \vdots 01\ 11\ 00\ 01\ 11\ 00\ 01\ 10 \right|$

Size (bits):

$P + N$

$N \cdot \log_2(U/P)$



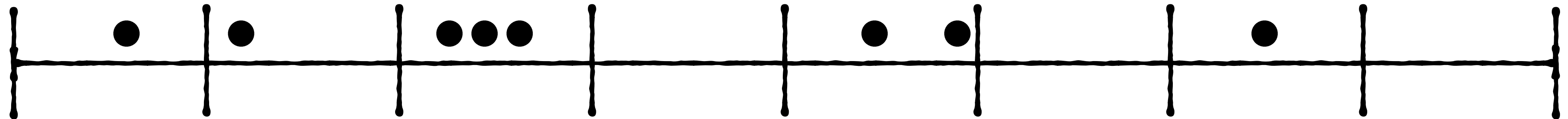
10 10 1110 0 110 0 10 0 : 01 11 00 01 11 00 01 10

Size (bits):

P + N

$$N \cdot \log_2(U/P)$$

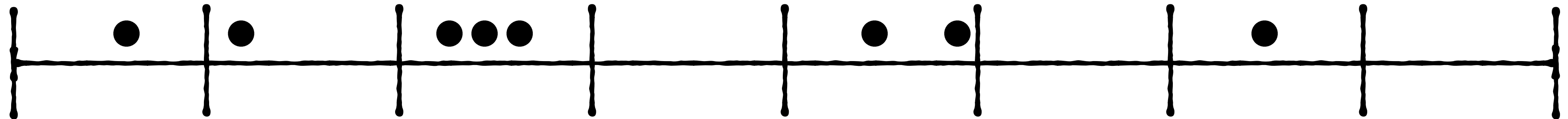
Since $2N > P \geq N$, plug N for P



10 10 1110 0 110 0 10 0 : 01 11 00 01 11 00 01 10

Size (bits):

2 N

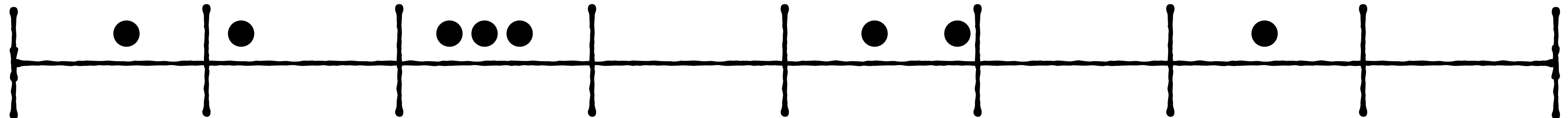
$$N \cdot \log_2(U/N)$$


10 10 1110 0 110 0 10 0 : 01 11 00 01 11 00 01 10

Size (bits / entry):

2

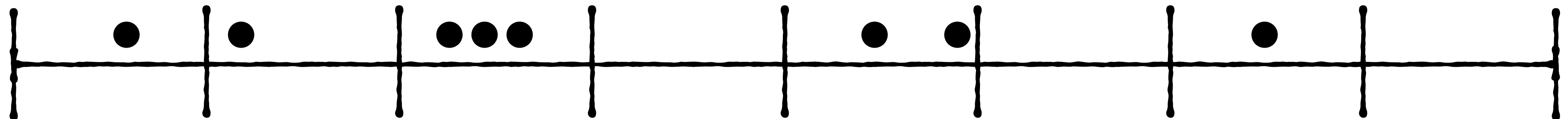
$\log_2(U/N)$



10 10 1110 0 110 0 10 0 : 01 11 00 01 11 00 01 10

Size (bits / entry):

$$2 + \log_2(U/N)$$

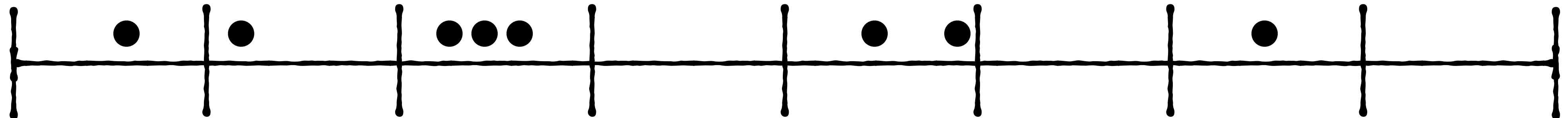


10 10 1110 0 110 0 10 0 : 01 11 00 01 11 00 01 10

Size (bits / entry):

$$2 + \log_2(U/N)$$

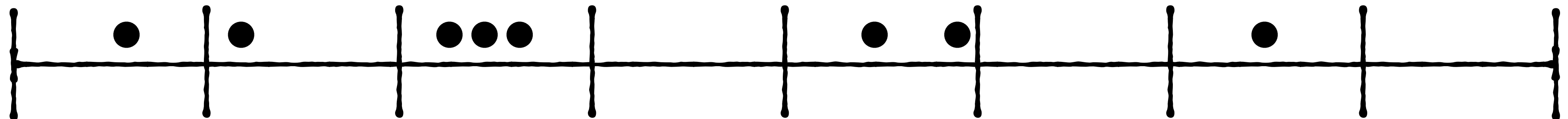
Tight upper bound



Elias-Fano

Size (bits / entry):

$$2 + \log_2(U/N)$$



Elias-Fano

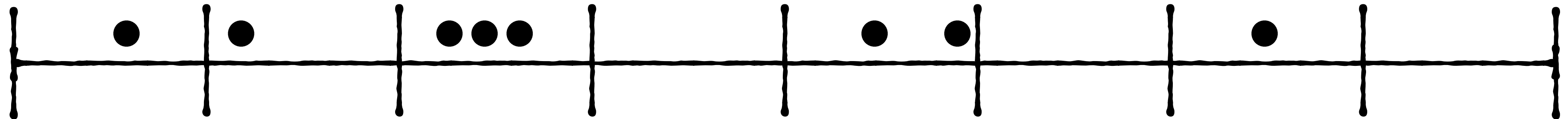
Golomb

Size (bits / entry):

$$2 + \log_2(U/N)$$

$>$

$$1.5 + \log_2(U/N)$$



Elias-Fano

Golomb

Size (bits / entry):

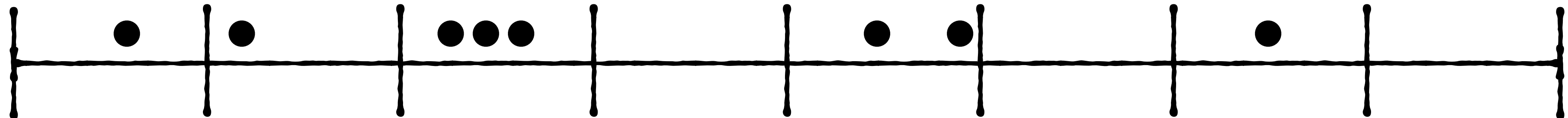
$$2 + \log_2(U/N)$$

$>$

$$1.5 + \log_2(U/N)$$



faster to access



Elias-Fano

Golomb

Size (bits / entry):

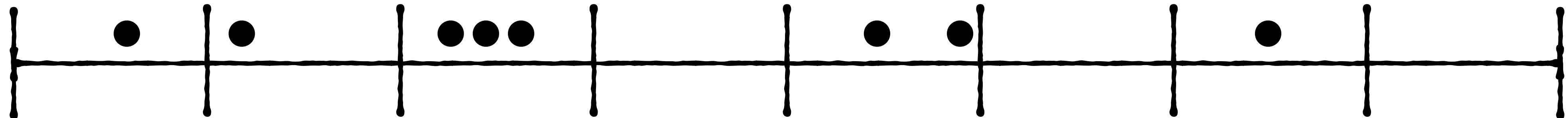
$$2 + \log_2(U/N)$$

$>$

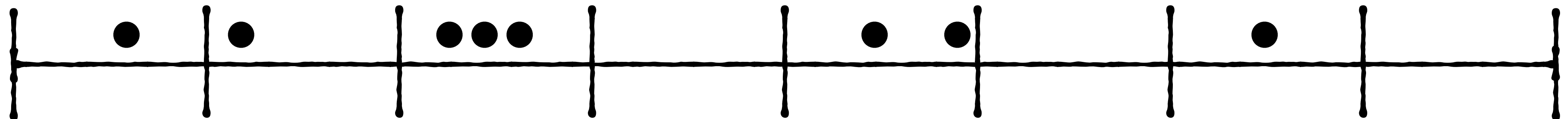
$$1.5 + \log_2(U/N)$$



faster to access

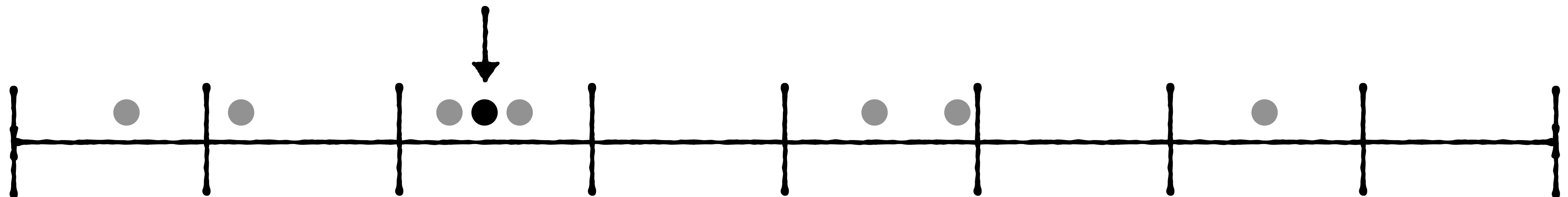


What element is at offset i of the sorted order?



What element is at offset **3** of the sorted order?

10 10 1110 0 110 0 10 0 : 01 11 00 01 11 00 01 10



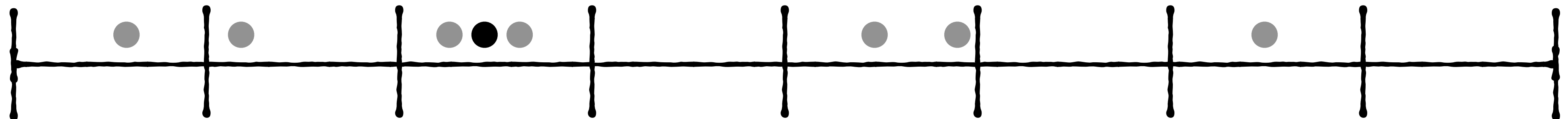
What element is at offset 3 of the sorted order?

10 10 11 10 0 1 10 0 10 0 : 01 11 00 01 11 00 01 10



0 Counter: 0

1 Counter: -1



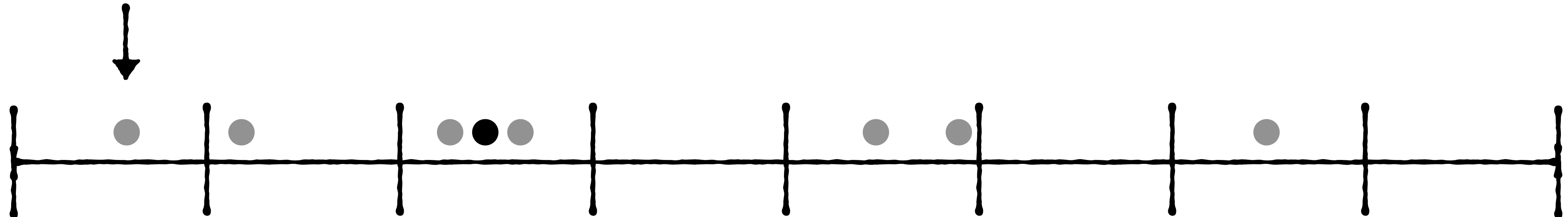
What element is at offset 3 of the sorted order?

10 10 11 10 0 1 1 0 0 1 0 0 : 0 1 1 1 0 0 0 1 1 0 1 0



0 Counter: 0

1 Counter: 0



What element is at offset 3 of the sorted order?

10 10 11 10 0 1 10 0 10 0 : 01 11 00 01 11 00 01 10



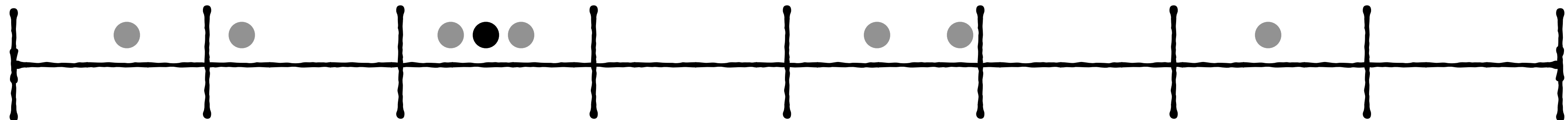
1

0



0 Counter:

1 Counter:



What element is at offset 3 of the sorted order?

10 10 11 10 0 1 10 0 10 0 : 01 11 00 01 11 00 01 10

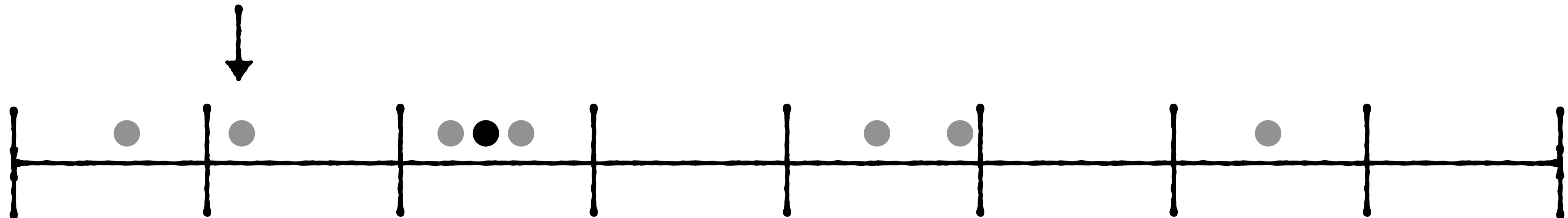


1

1

0 Counter:

1 Counter:



What element is at offset 3 of the sorted order?

10 10 11 10 0 1 1 0 0 1 0 0 : 0 1 1 1 0 0 0 1 1 1 0 0 0 1 1 0

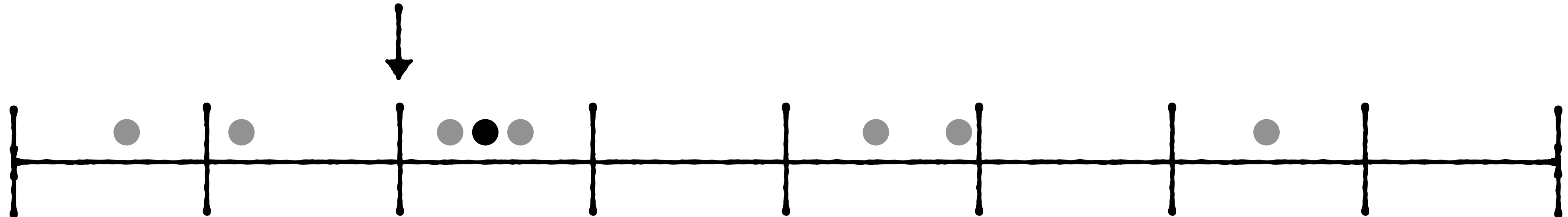


2

1

0 Counter:

1 Counter:



What element is at offset 3 of the sorted order?

10 10 11 10 0 1 10 0 10 0 : 01 11 00 01 11 00 01 10

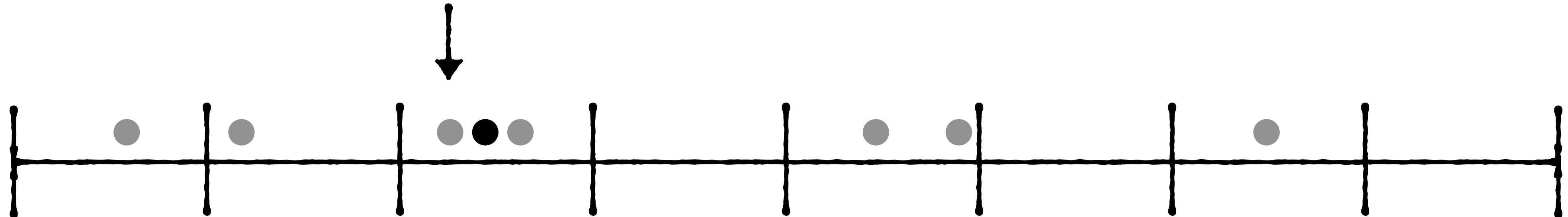


2

2

0 Counter:

1 Counter:



What element is at offset 3 of the sorted order?

10 10 11 10 0 1 10 0 10 0 : 01 11 00 01 11 00 01 10

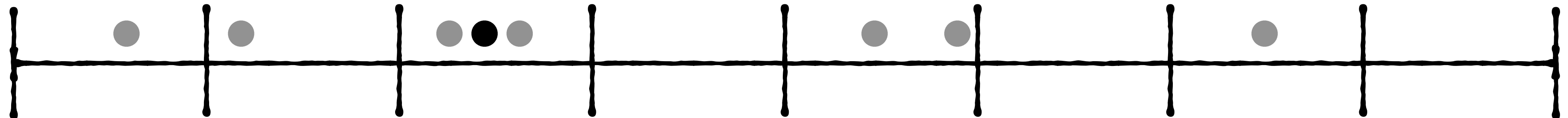


2

3

0 Counter:

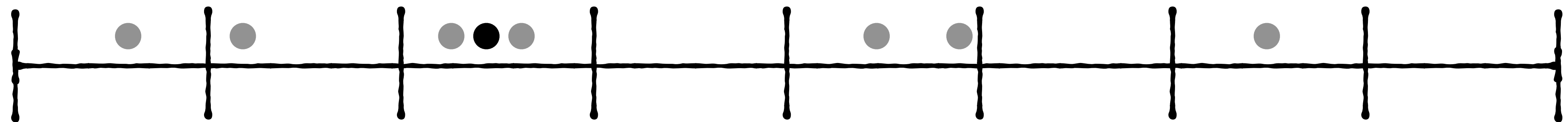
1 Counter:



What element is at offset 3 of the sorted order?

10 10 11 10 0 1 10 0 10 0 : 01 11 00 01 11 00 01 10

↑
2
3



Prefixes:

000

001

010

011

100

101

110

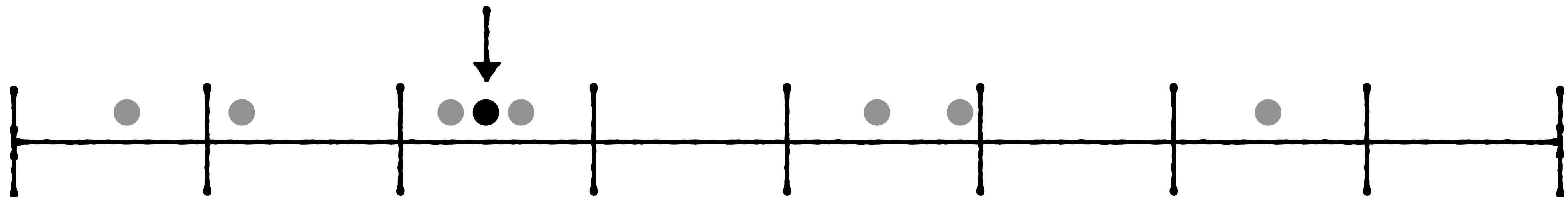
111

What element is at offset 3 of the sorted order?

10 10 11 10 0 1 10 0 10 0 : 01 11 00 01 11 00 01 10



Fetch 3rd suffix

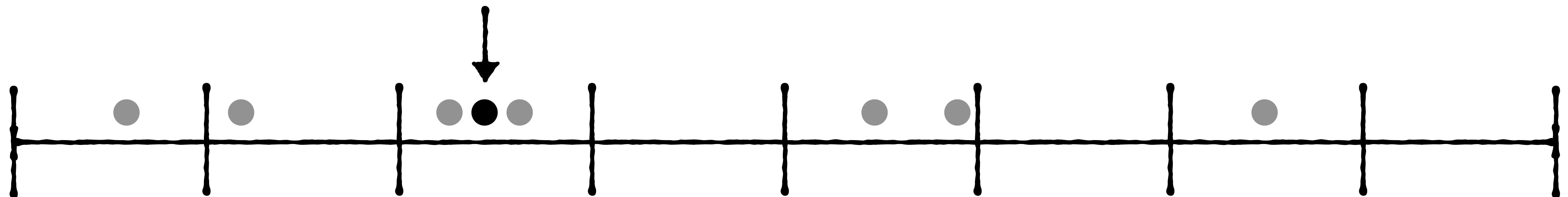


010

What element is at offset 3 of the sorted order?

10 10 11 10 0 1 10 0 10 0 : 01 11 00 11 00 01 10

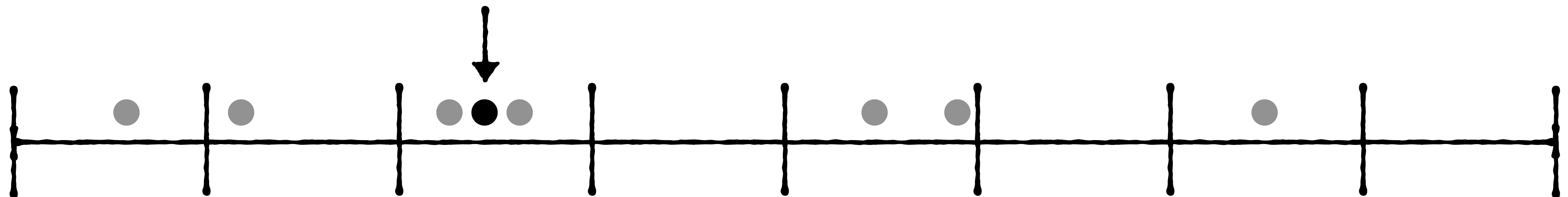
Concatenate: 01001



What element is at offset 3 of the sorted order?

10 10 11 10 0 1 10 0 10 0 : 01 11 00 11 00 01 10

We're done :) 01001



Rank & Select

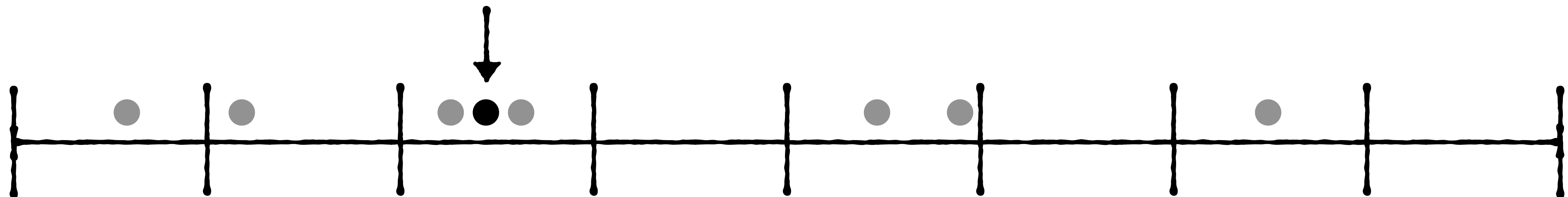
10 10 1110 0 110 0 10 0 : 01 11 00 01 11 00 01 10



**Worst case
lookup**

Traverse 2 · N bits

One random access



Rank & Select

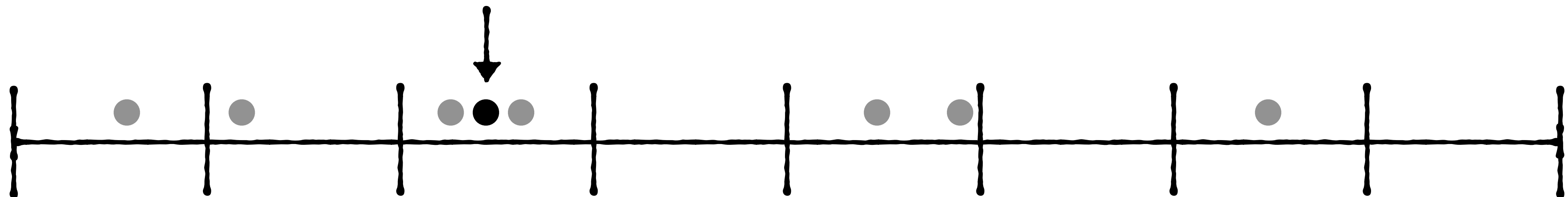
10 10 1110 0 110 0 10 0 : 01 11 00 01 11 00 01 10



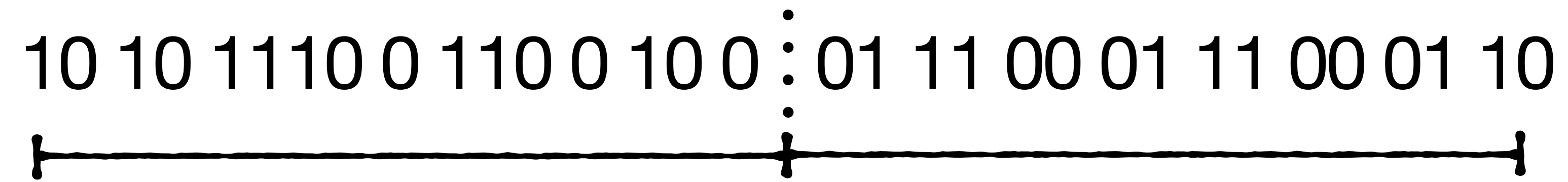
**Worst case
lookup**

Traverse $2 \cdot N$ bits

$< N 1.5 + N \log_2(U/N)$
w. run-length Golomb



10 10 1110 0 110 0 10 0 : 01 11 00 01 11 00 01 10



Traverse $2 \cdot N$ bits



Can we make this faster?

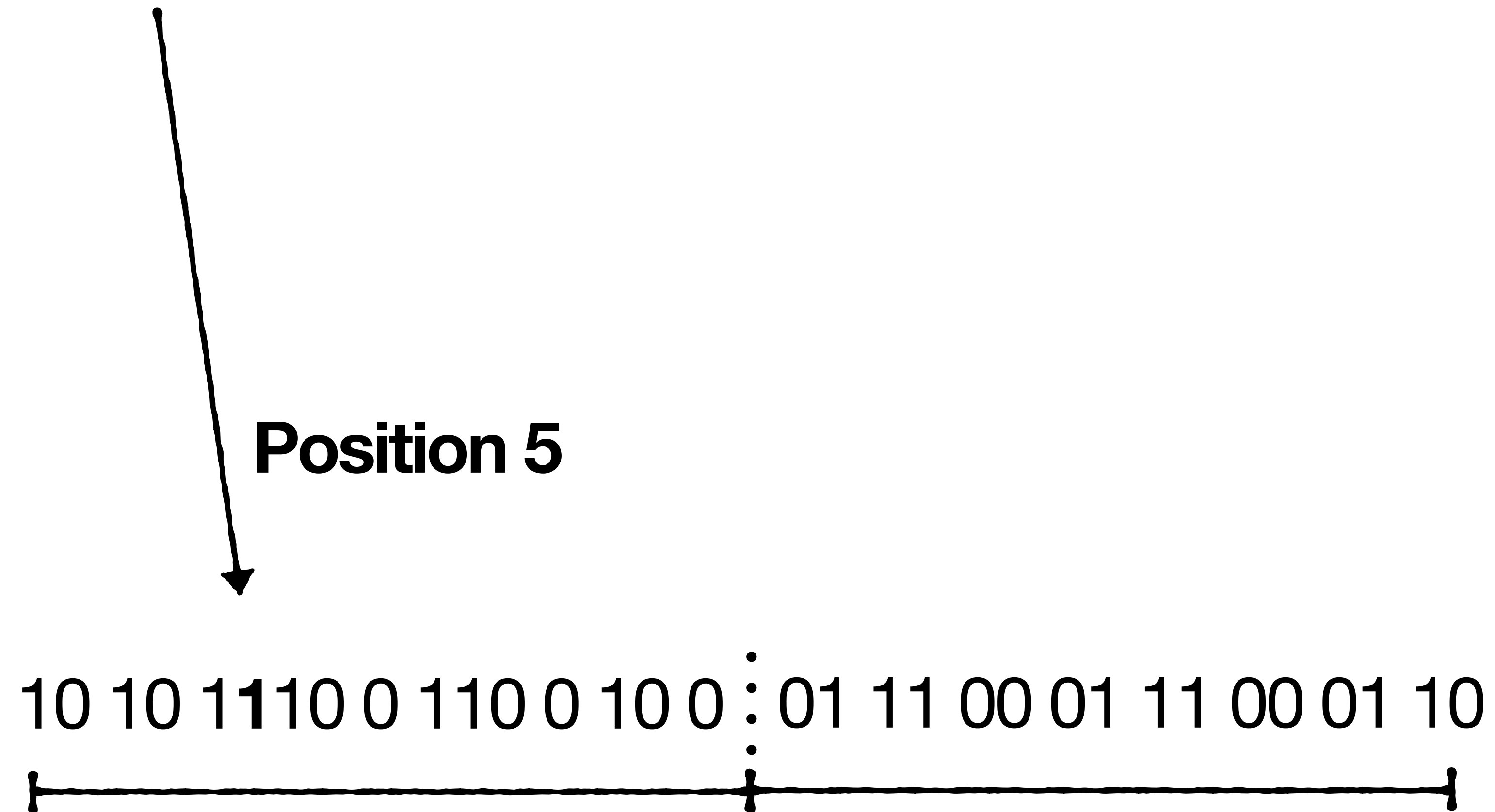
Rank & Select

**Common primitive commands
that can be sped up**

**Frame a lookup in
terms of these**

Select_x(i): return position of ith bit set to x in bitmap

Select₁(3): return position of 3rd bit set to 1 in bitmap



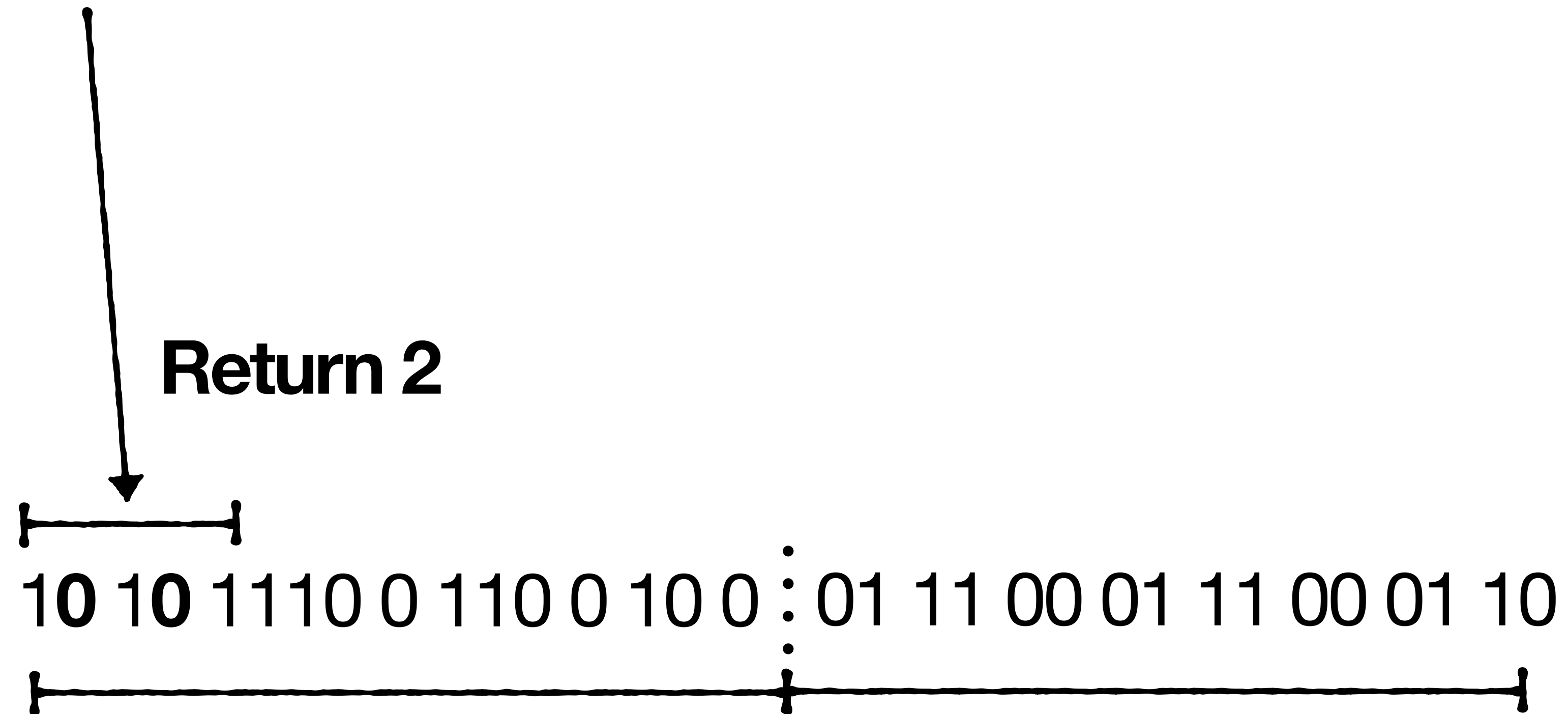
$\text{Select}_x(i)$: return position of i^{th} bit set to x in bitmap

$\text{rank}_x(i)$: return # bits set to x up to position i

10 10 1110 0 110 0 10 0 : 01 11 00 01 11 00 01 10
└──────────────────┴──────────────────────────────────┘

Select_x(i): return position of ith bit set to x in bitmap

rank₀(5): return # bits set to 0 up to position 5



prefix of i^{th} element in Elias Fano = # zeros before i^{th} 1

10 10 1110 0 110 0 10 0 $\begin{smallmatrix} \cdot \\ \vdots \\ \cdot \end{smallmatrix}$ 01 11 00 01 11 00 01 10

prefix of i^{th} element in Elias Fano = # zeros before i^{th} 1
= $\text{Select}_1(i) - i$

10 10 1110 0 110 0 10 0 $\begin{smallmatrix} \cdot \\ \vdots \\ \cdot \end{smallmatrix}$ 01 11 00 01 11 00 01 10

prefix of i^{th} element in Elias Fano = # zeros before i^{th} 1

$$= \text{Select}_1(i) - i$$



bits before

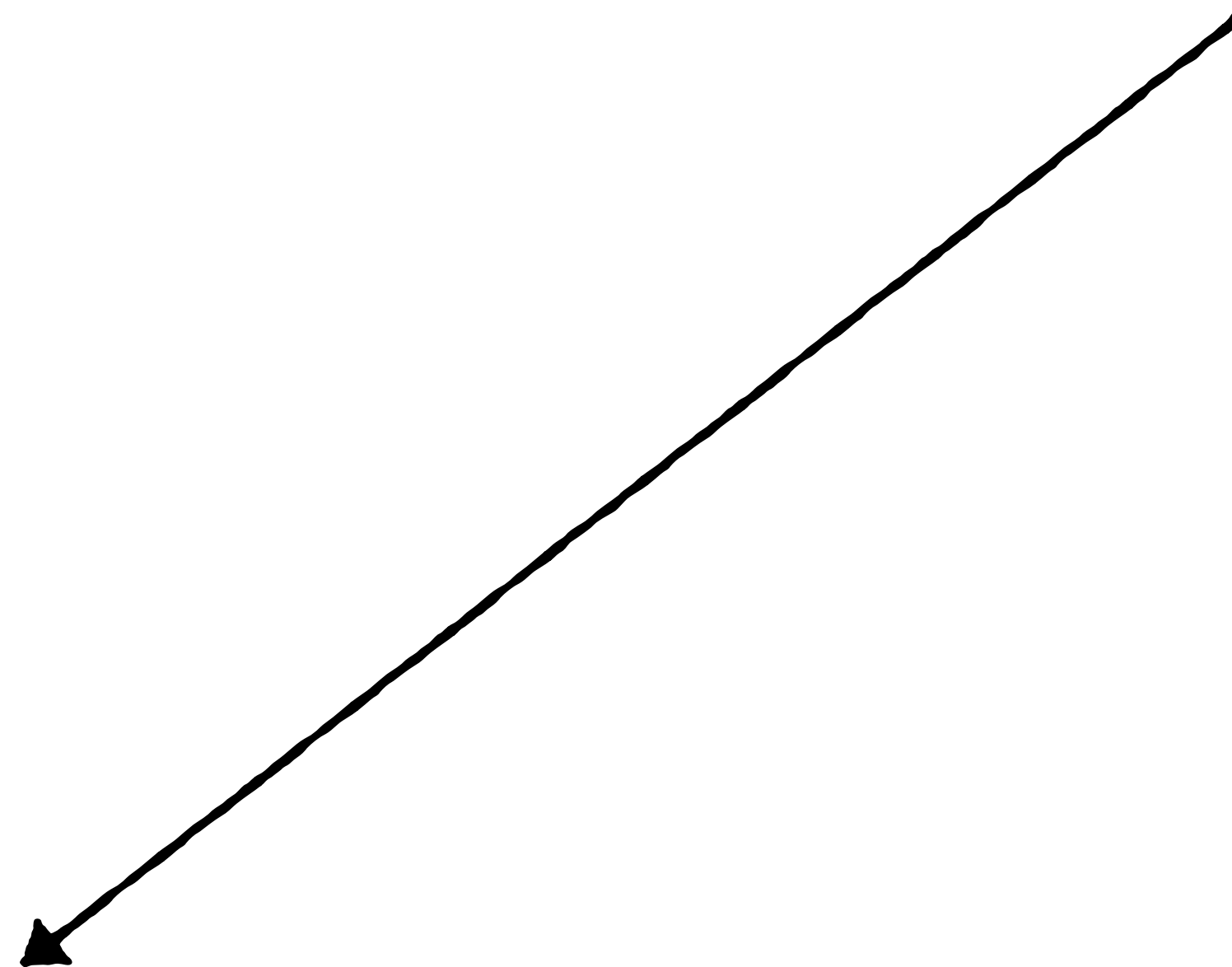
1 before

10 10 1110 0 110 0 10 0 : 01 11 00 01 11 00 01 10

prefix of 3rd element in Elias Fano = # zeros before 3rd 1

$$= \text{Select}_1(3) - 3$$

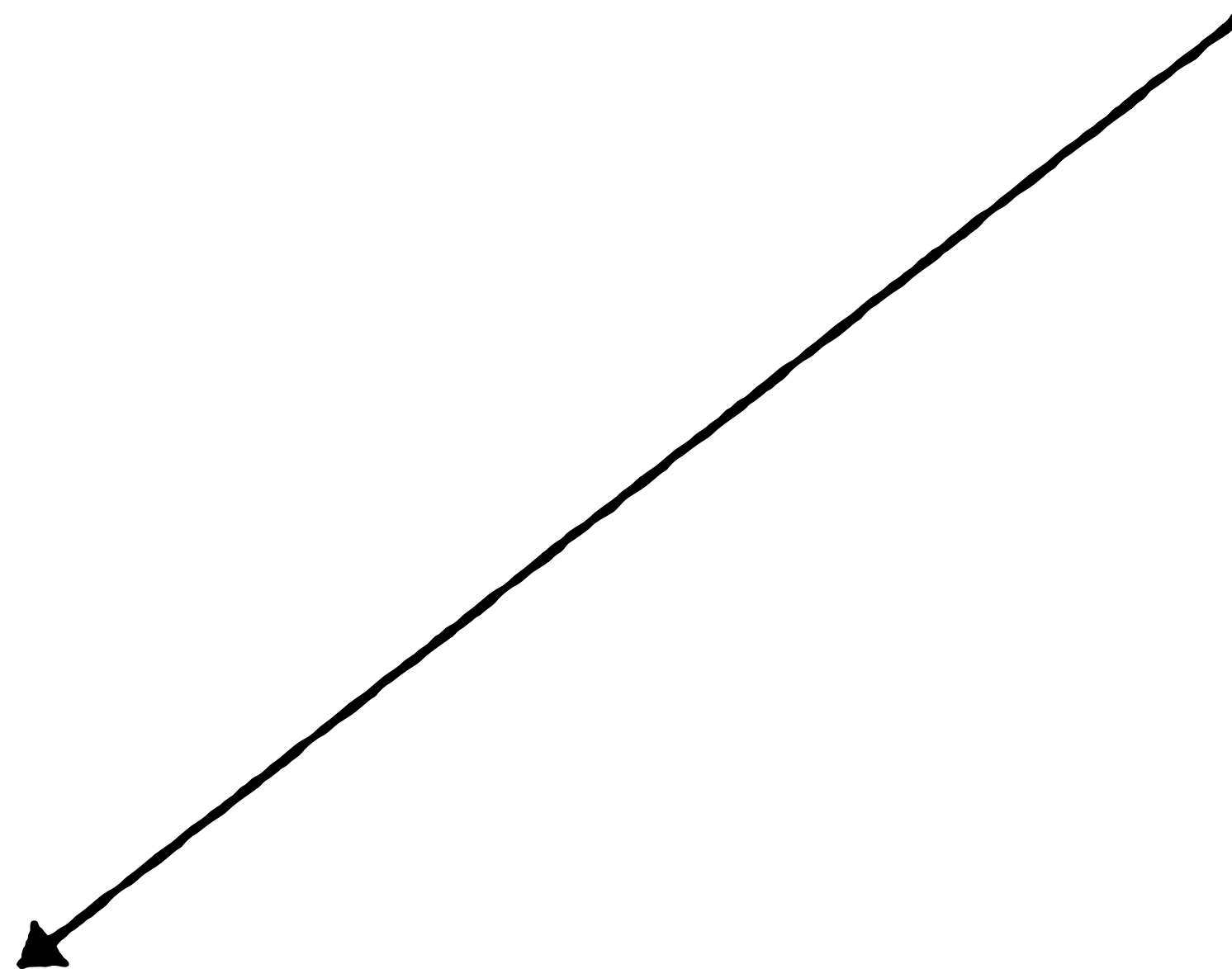
10 10 1110 0 110 0 10 0 : 01 11 00 01 11 00 01 10



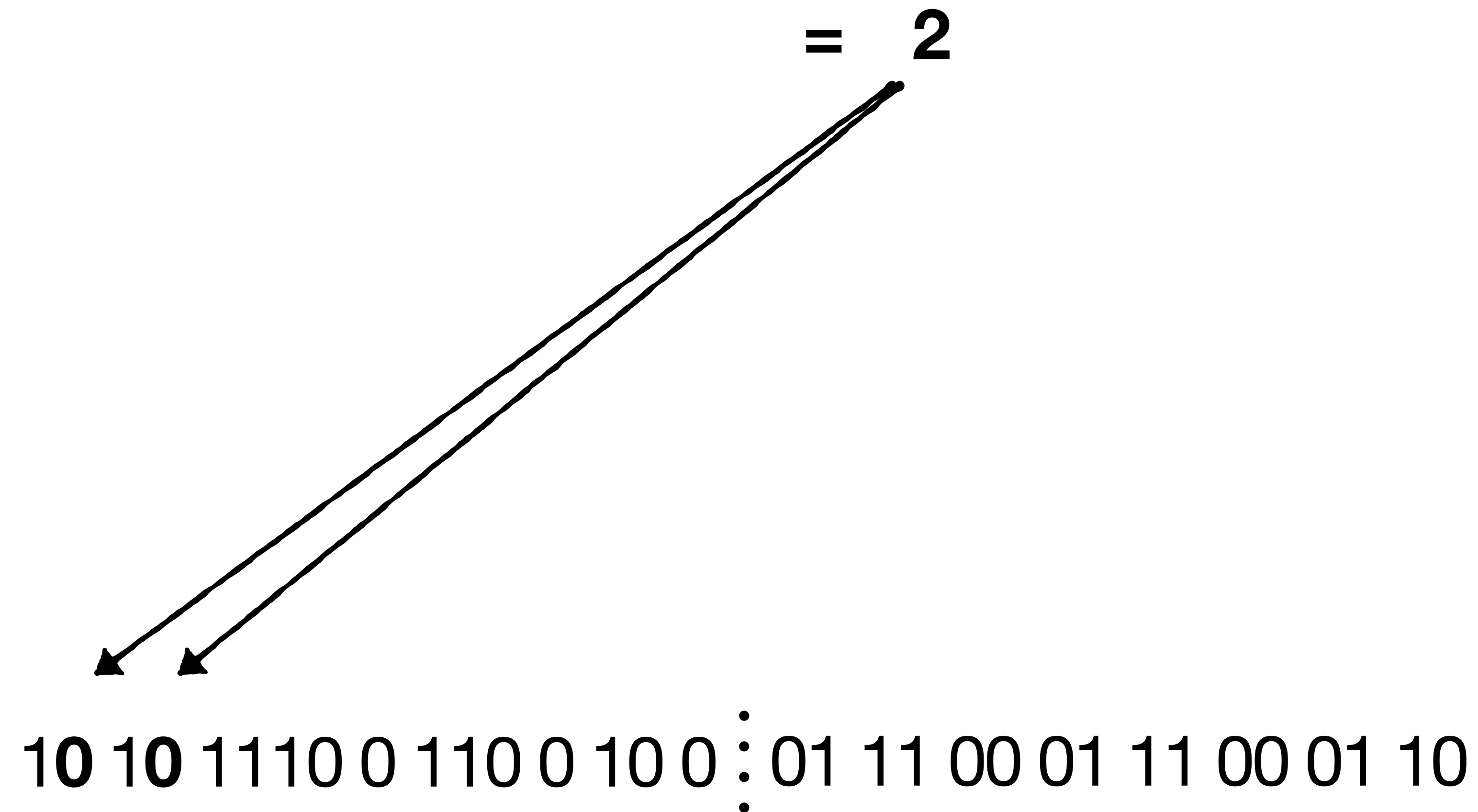
prefix of 3rd element in Elias Fano = # zeros before 3rd 1

$$= 5 - 3$$

10 10 1110 0 110 0 10 0 ∴ 01 11 00 01 11 00 01 10



prefix of 3rd element in Elias Fano = # zeros before 3rd 1



Making Rank Fast

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

Partition

| 1 0 1 0 | 1 1 1 0 | 0 1 1 0 | 1 1 0 0 |

Count # 1s to left of each partition

2

| 1 0 1 0 | 1 1 1 0 | 0 1 1 0 | 1 1 0 0 |

Count # 1s to left of each partition

2 5

| 1 0 1 0 | 1 1 1 0 | 0 1 1 0 | 1 1 0 0 |

Count # 1s to left of each partition

2 5 7

| 1 0 1 0 | 1 1 1 0 | 0 1 1 0 | 1 1 0 0 |

	0		2		5		7													
	1	0	1	0		1	1	1	0		0	1	1	0		1	1	0	0	

Rank₁(10)?

$$\begin{array}{ccccccccc}
 & & 0 & & & 2 & & & 7 \\
 \left| \begin{array}{cccc} 1 & 0 & 1 & 0 \end{array} \right| & \left| \begin{array}{cccc} 1 & 1 & 1 & 0 \end{array} \right| & & \left| \begin{array}{cc} 1 & 0 \end{array} \right| & \left| \begin{array}{ccc} 1 & 1 & 0 \end{array} \right| & \left| \begin{array}{cc} 0 & 0 \end{array} \right|
 \end{array}$$

$$\text{Rank}_1(10) = 5 + 0 + 1 = 6$$



rank in $O(1)$



Counters take space

0 2 7

| 1 0 1 0 | 1 1 1 0 | 1 0 | 1 1 0 0 |

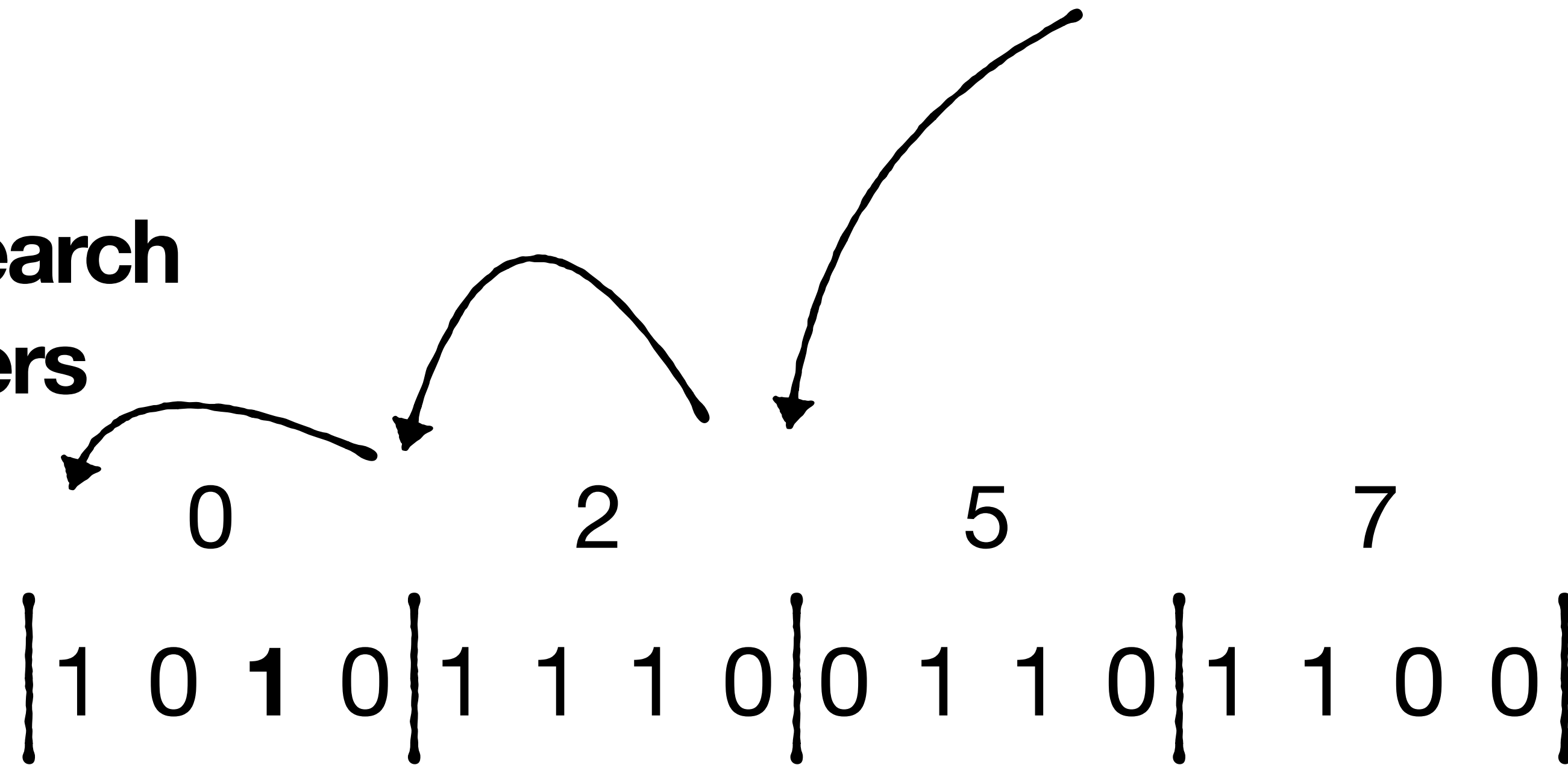
└───────────────────────────────────┘

How about select? e.g. $\text{select}_1(1)$

0	2	5	7
1 0 1 0	1 1 1 0	0 1 1 0	1 1 0 0

How about select? e.g. $\text{select}_1(1)$

**Binary search
counters**



Scan

Select in $O(\log_2(N))$

Can we do better?

0	2	5	7
1 0 1 0	1 1 1 0	0 1 1 0	1 1 0 0

Store samples of select results

select₁(0) select₁(4) select₁(8) select₁(12)

| 1 0 1 0 | 1 1 1 0 | 0 1 1 0 | 1 1 0 0 |

Store samples of select results

select₁(0) select₁(4) select₁(8) select₁(12)

0

↓
| 1 0 1 0 | 1 1 1 0 | 0 1 1 0 | 1 1 0 0 |

Store samples of select results

select₁(0) **select₁(4)** select₁(8) select₁(12)

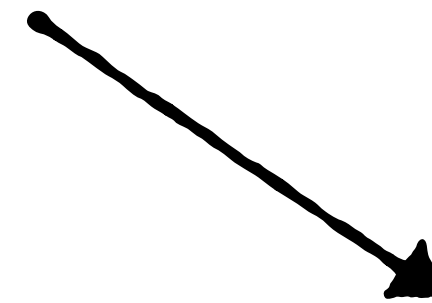
0

6



| 1 0 1 0 | 1 1 1 0 | 0 1 1 0 | 1 1 0 0 |

select₁(0) select₁(4) select₁(8) select₁(12)
0 6 **13**



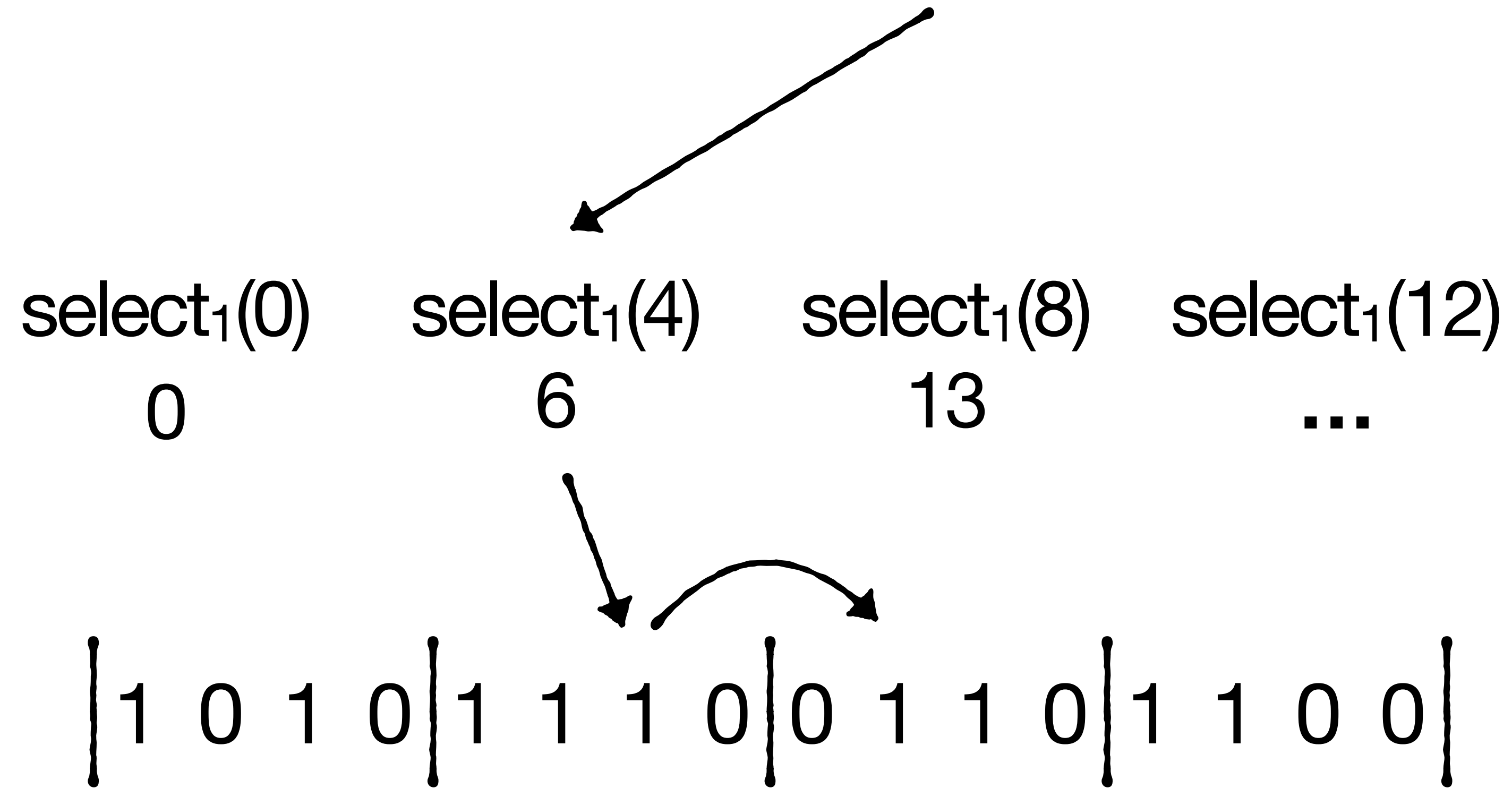
| 1 0 1 0 | 1 1 1 0 | 0 1 1 0 | 1 1 0 0 |

How to handle select(6)?

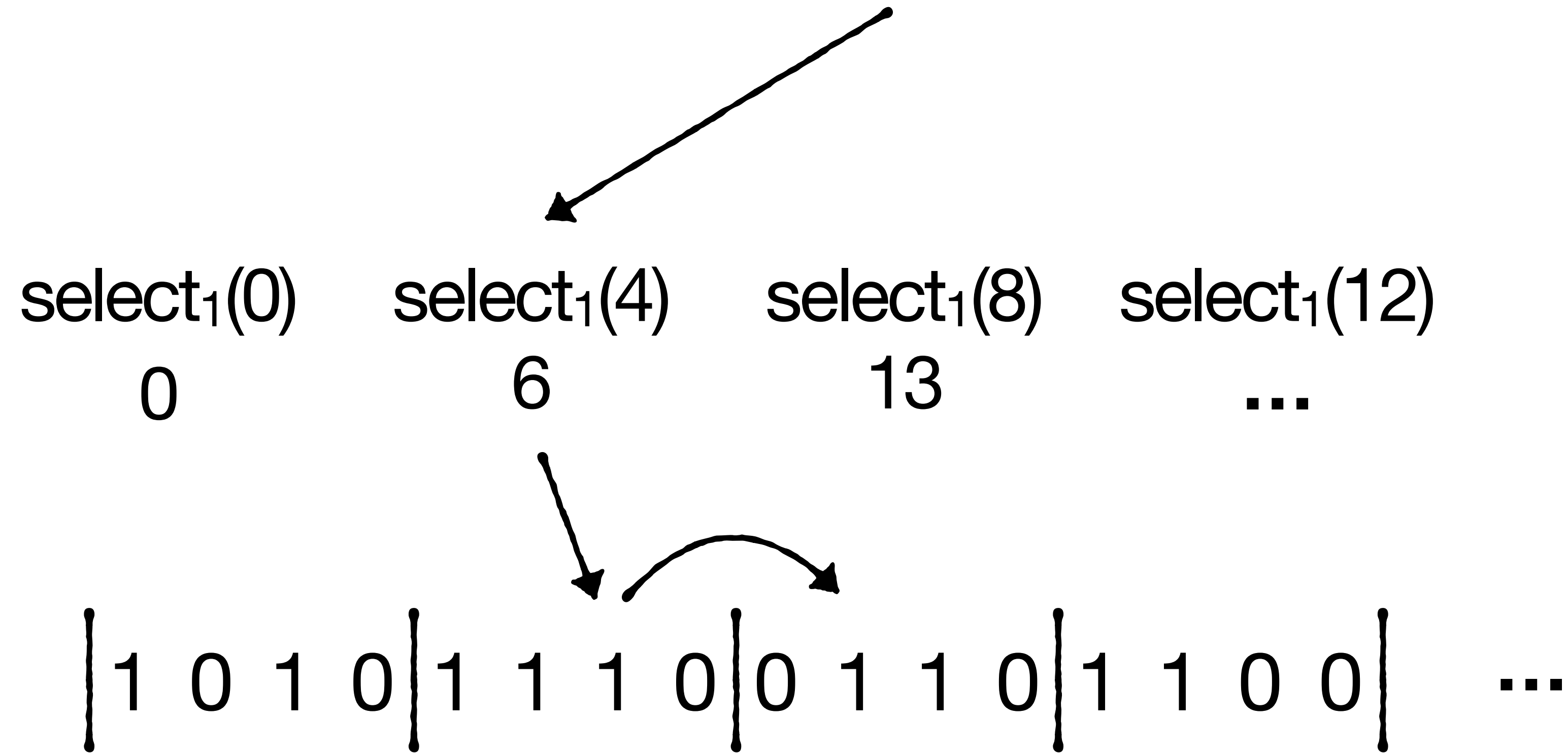
select ₁ (0)	select ₁ (4)	select ₁ (8)	select ₁ (12)	...
0	6	13	...	

1 0 1 0	1 1 1 0	0 1 1 0	1 1 0 0	...
---------	---------	---------	---------	-----

How to handle **select(5)**?

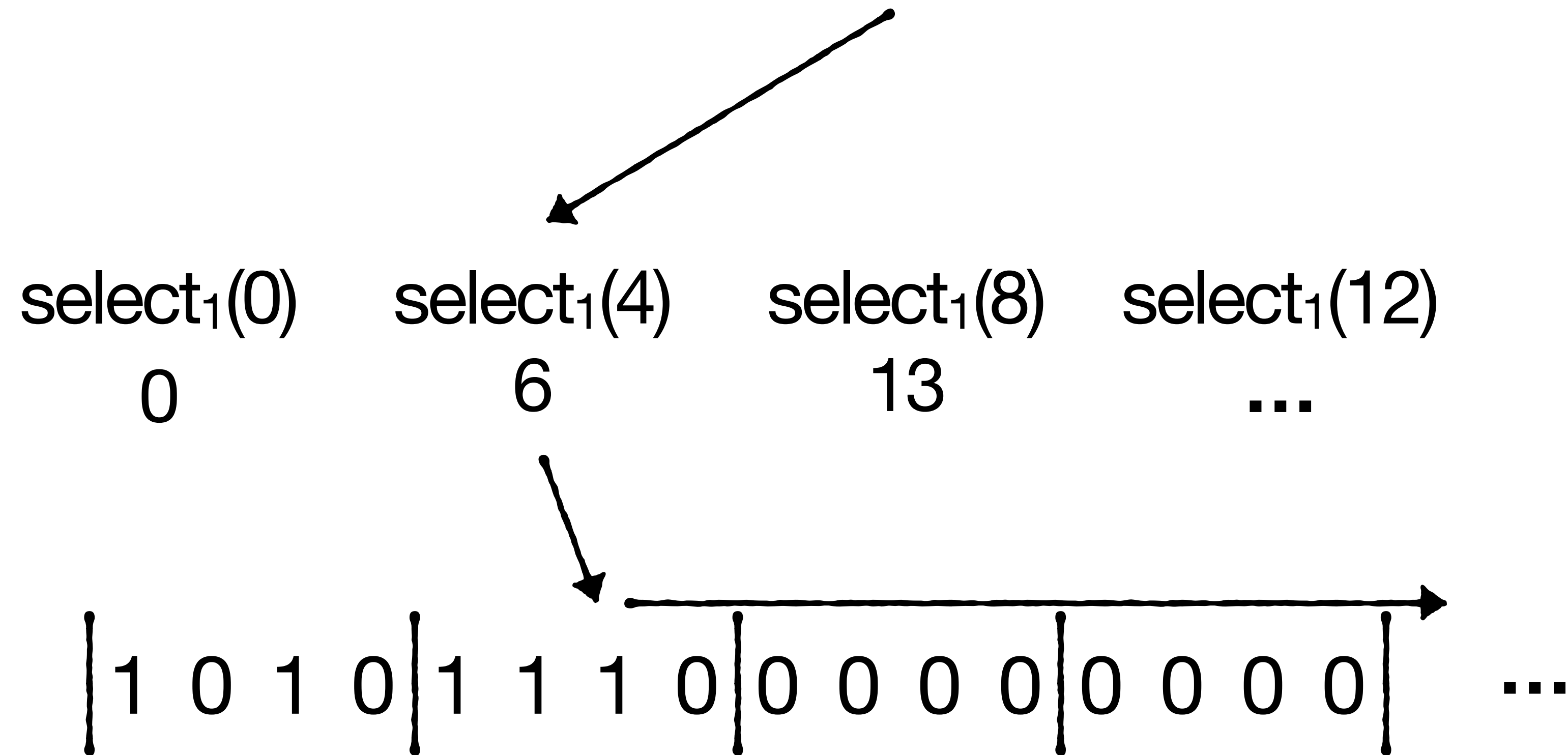


How to handle $\text{select}(5)$?



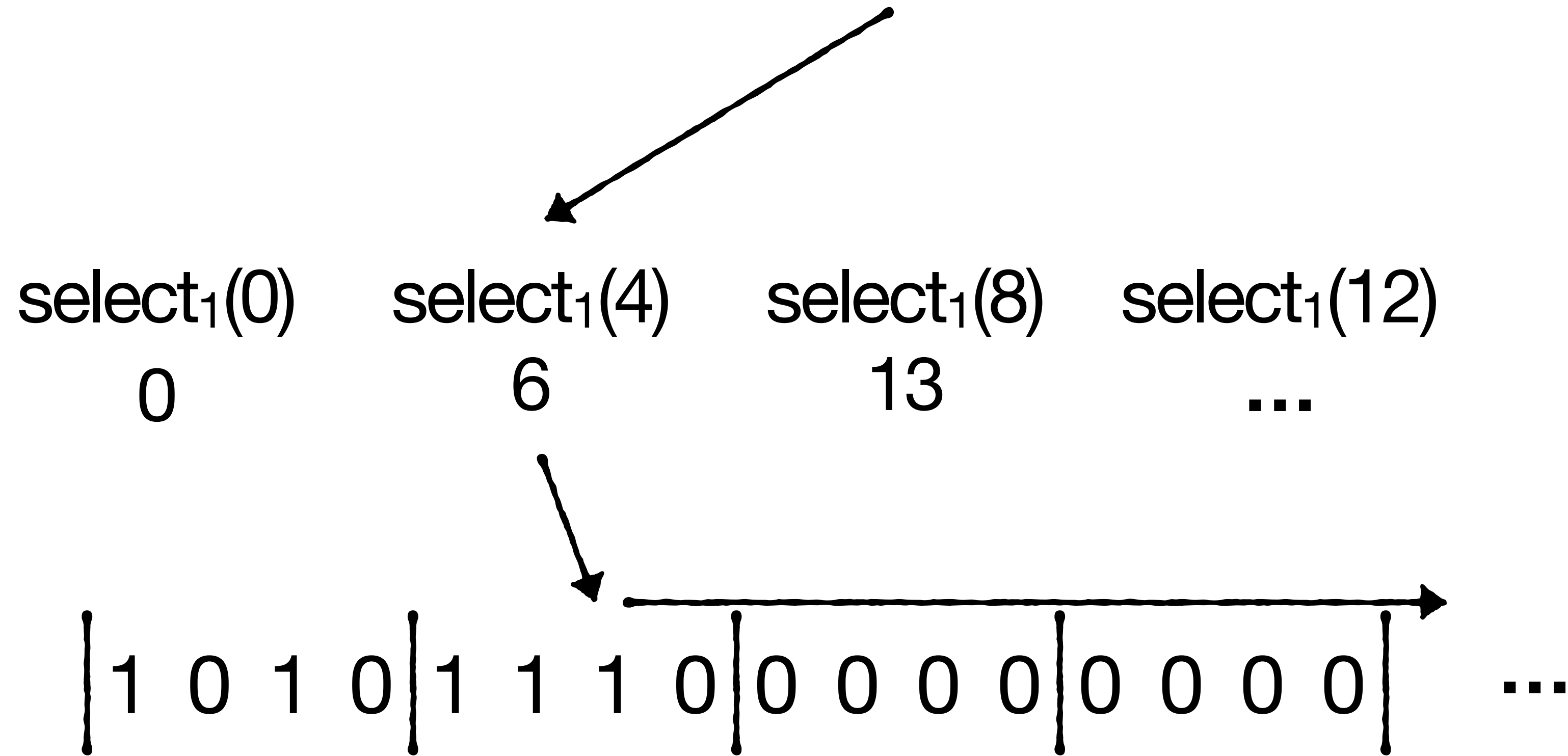
Select in $O(1)$ time assuming next bit is close

How to handle $\text{select}(5)$?



However, next 1 can be far

How to handle $\text{select}_1(5)$?



However, next 1 can be far

There are ways of fixing this, but we can also resort to binary search as before if so

Enables trade-offs in-between :)

Sorted Array

Golomb

Elias Fano

Bitmap

Bits

$\log_2(U)$

$1.5 + \log_2(U/N)$

$2 + \log_2(U/N)$

U/N

Enables trade-offs in-between :)

Sorted Array

Golomb

Elias Fano

Bitmap

Bits

$\log_2(U)$

$1.5 + \log_2(U/N)$

$2 + \log_2(U/N)$

U/N

Access

$O(1)$

$O(N)$

$O(1)$

$O(1)$

Enables trade-offs in-between :)

	Sorted Array	Golomb	Elias Fano	Bitmap
Bits	$\log_2(U)$	$1.5 + \log_2(U/N)$	$2 + \log_2(U/N)$	U/N
Access	$O(1)$	$O(N)$	$O(1)$	$O(1)$
Insert	$O(N)$	$O(N)$	$O(N)$	$O(1)$

And now, a student presentation