# Mining Massive Data

Luciano Gualà

www.mat.uniroma2.it/~guala/MMD_2024.html

**What:** 3 topics, 2 lectures per topic

**Algorithms for Big Data:**
- Monday - 14,00-18,00
- lecturer: prof. Luciano Gualà

**The PageRank algorithm:**
- Tuesday – 14,00-18,00
- lecturer: prof. Andrea Clementi

**Bitcoin and the Lightning Network:**
- Wednesday – 14,00-18,00
- lecturer: prof. Francesco Pasquale

**How** (to get credits)

- attend lectures
- final oral exam and/or class presentation (of uncovered material)

# Algorithms for Big Data

- research field on algorithmic aspects addressing scenarios in which the input is very large
- sometime classic ways of designing and analyzing algorithms are somehow insufficient
- efficiency issues are stressed even more

Example: Given n items, find the most similar ones.
- classic $O(n^2)$-time algorithm.
- goal: almost linear time solution (breaking the $n^2$-time barrier)

Example: Store n items to check membership.
- classic $O(n)$-space data structures
- goal: use very few bits per item
  (sometime less bits than the bits needed to represent the set of items itself)

Example: Store n items in sublinear space.
- an item of size s should be represented with $O(\log s)$ space
- goal: still be able to retrieve some properties of the items from their representations

Example: Streaming algorithms.
- input is given as a stream of items
- can read an element at a time
- entire stream does not even fit in the memory
- goal: be able to compute statistics/functions of the entire input

Example: Given n items, find the most similar ones.

- classic $O(n^2)$-time
- goal: almost linear

**3. Locality Sensitive Hashing**

Example: Store n items to check membership.

- classic $O(n)$-space data structures
- goal: use very few (sometime less bits than the bits needed to represent the set of items itself)

**1. Bloom Filters**

Example: Store n items in sublinear space.

- an item of size s s ... ce
- goal: still be able t ... s from their representations

**3. MinHash signatures**

Example: Streaming algorithms.

- input is given as a stream of items
- can read an elemen
- entire stream does
- goal: be able to compute statistics/functions of the entire input

**2. DGIM algorithm**

# Algorithms for Big Data
# Episode I

reference:
Algorithms for Massive Data (Lecture Notes)
Nicola Prezza
https://arxiv.org/abs/2301.00754

# Bloom Filters

A Bloom Filter is a probabilistic data structure that maintains a set S of elements subject to the following operations:

- insert(x): add element x to S.
- membership(x): return YES if $x \in S$, returns NO if $x \notin S$.

probabilistic:

- if $x \in S$ returns YES with probability 1 (no false negative)
- if $x \notin S$ return NO with probability $1-\delta$

$0<\delta<1$: user-defined error parameter

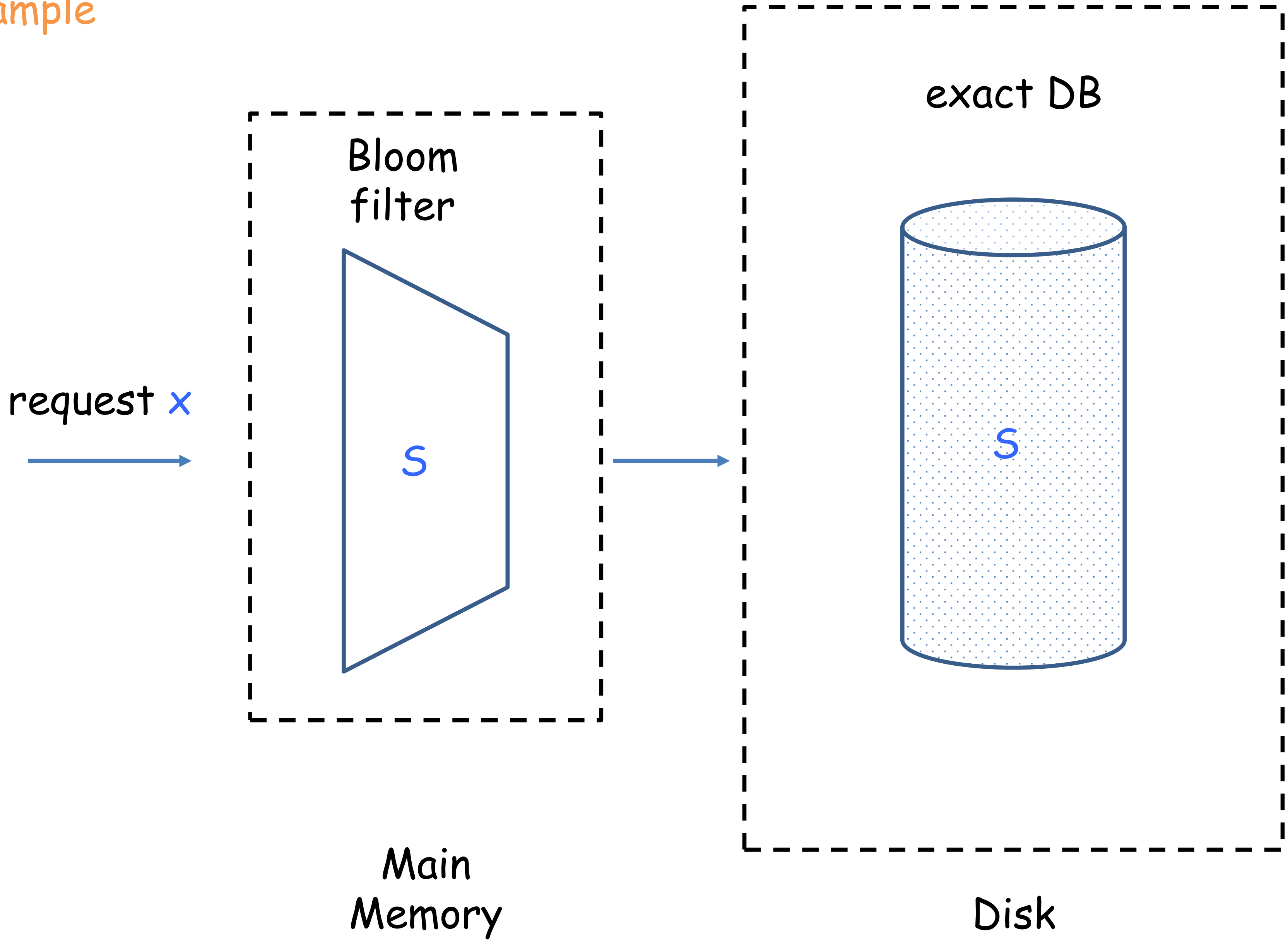
Bloom filter:

- uses $\Theta(n \log (1/\delta))$ bits to store at most n elements from a universe U (really compact)
- n is the capacity of the filter
- not able to retrieve the (actual) element but just say whether it is in S

typical use:

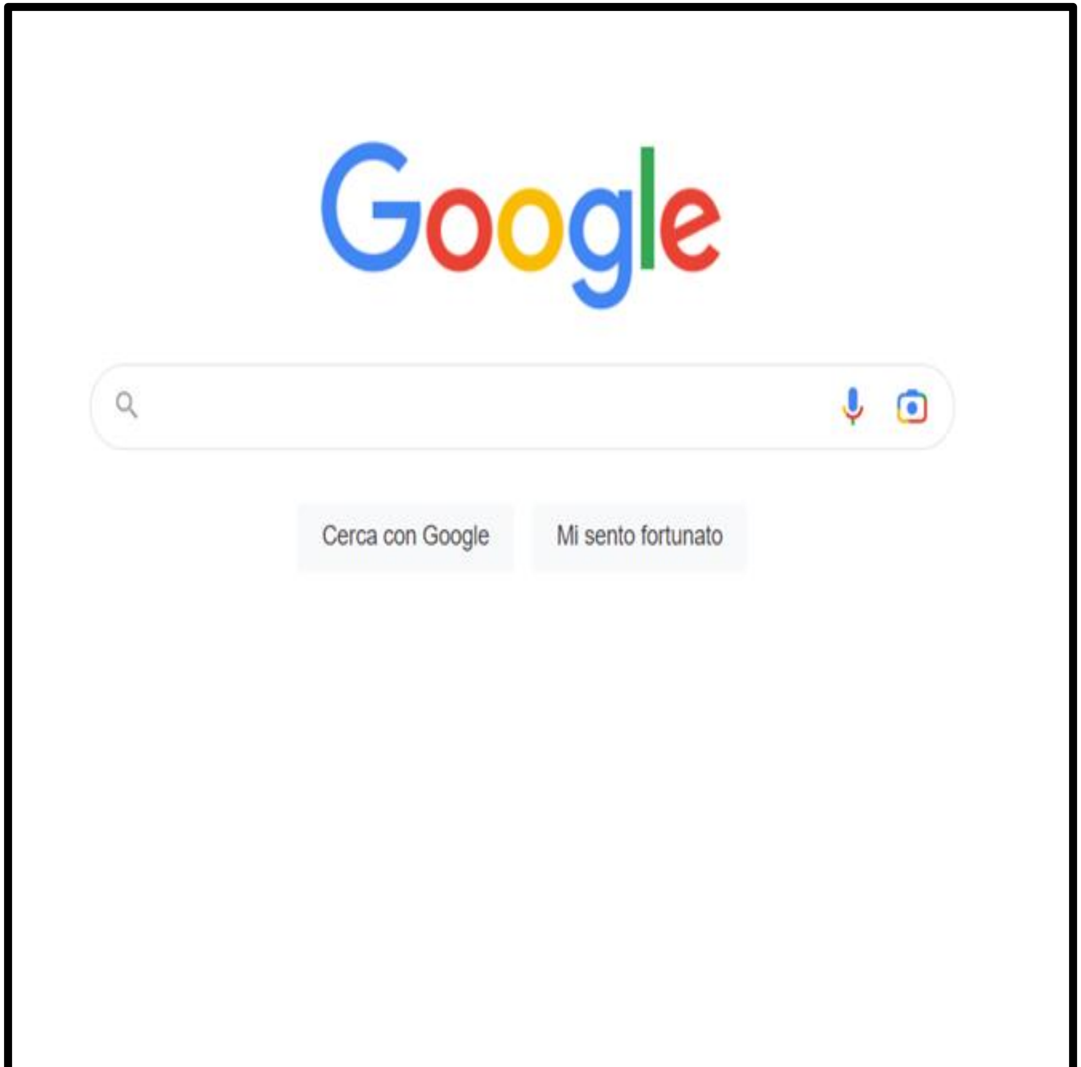
- as an interface to a larger and slower (but exact) DS to quickly filter negative requests
- in a stream used to filter stream elements that do not meet some criterion

## Example




access DB on the disk only if the filter says that x is in S

# Example: how Google Chrome detects malicious URLs




- insert the known malicious URLs into a Bloom filter
- only the URLS that pass the filter are checked on Google's remote servers

Let $h_1,\ldots,h_k$ be $k$ hash functions, $h_i : U \longrightarrow \{0,1,\ldots,m-1\}$
- $m$ and $k$ are parameters that will be chosen as function of $n$ and $\delta$

assumption: $h_1,\ldots,h_k$ are independent and completely uniform

completely uniform: $h_i$ maps any $x \in U$ to any *given* bucket with prob. $1/m$

- simplifies the analysis
- almost met in practice if you use a *good* enough hash function (e.g. SHA-256)

Bloom Filter: a bit-vector $B[0,1,\ldots,m-1]$ of size $m$, initially all set to 0.

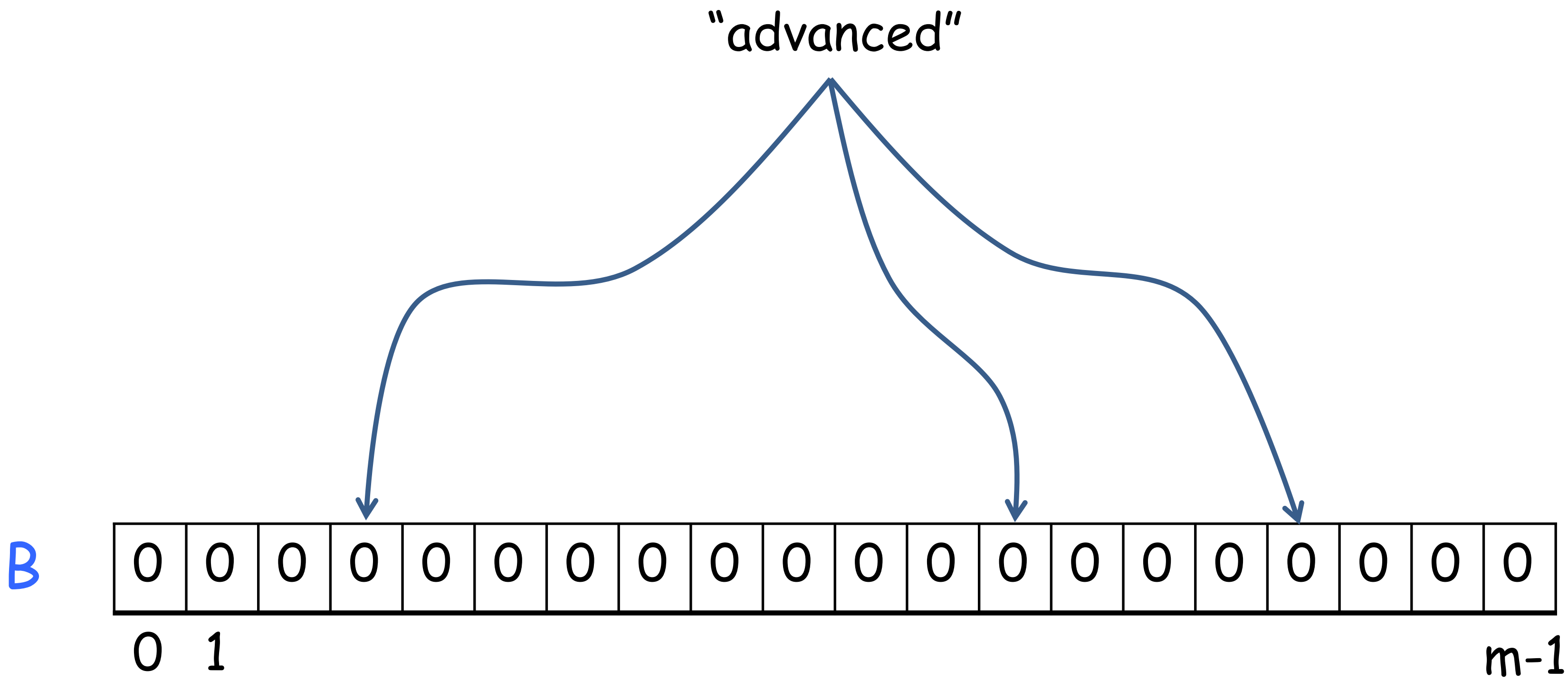
insert($x$): set to 1 all $B[h_i(x)]$, $i=1,\ldots,k$

membership($x$): Return YES iff all $B[h_i(x)]=1$, $i=1,\ldots,k$.

$S$={ }

$k$=3

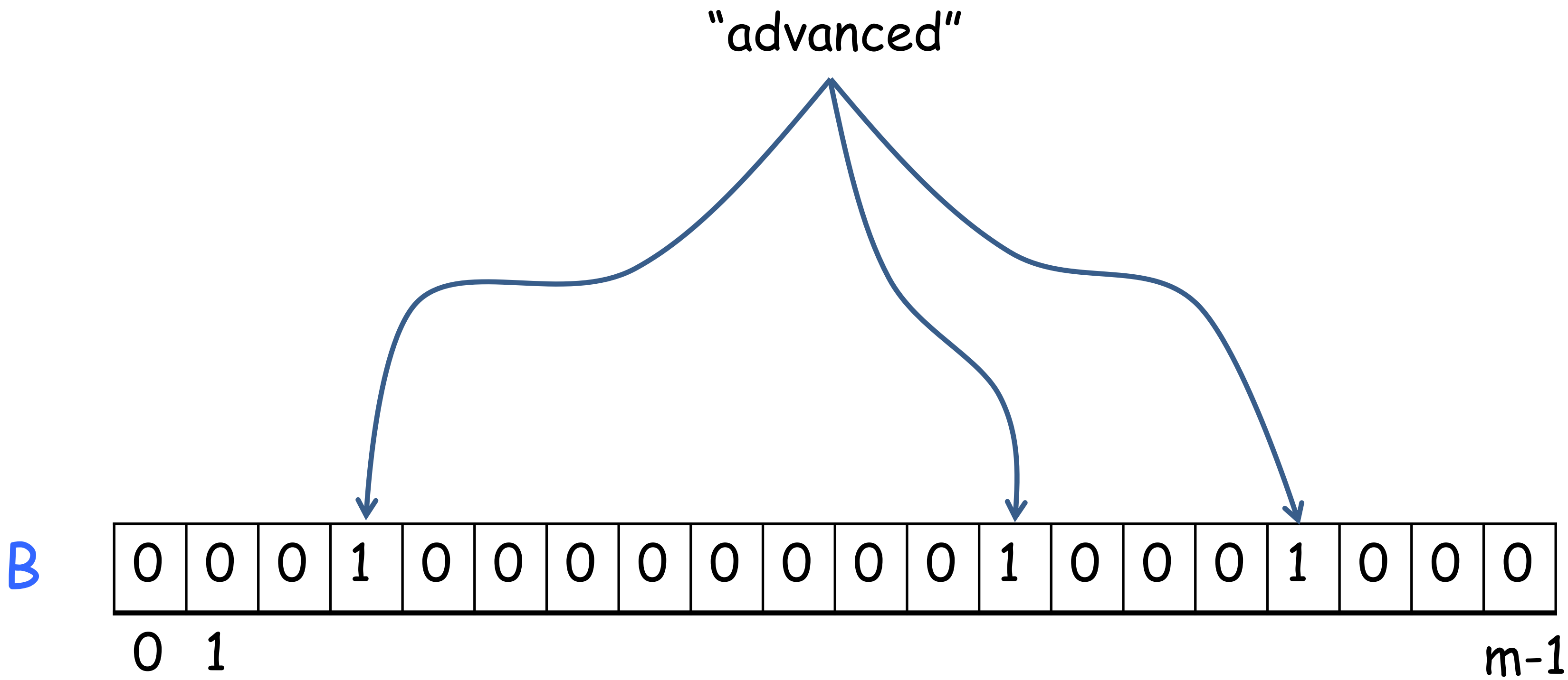
insert("advanced")



$S$={"advanced"}

$k$=3

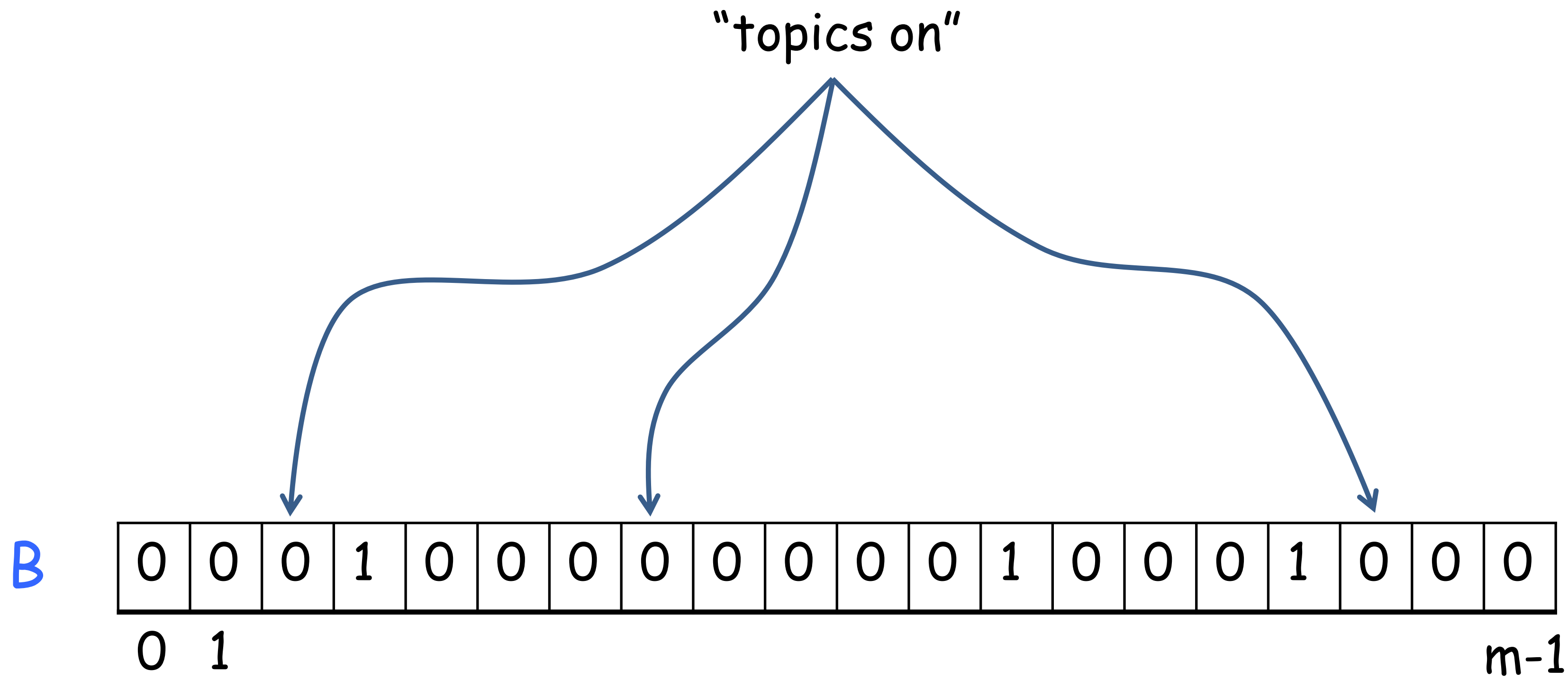
insert("advanced")



S={"advanced"}

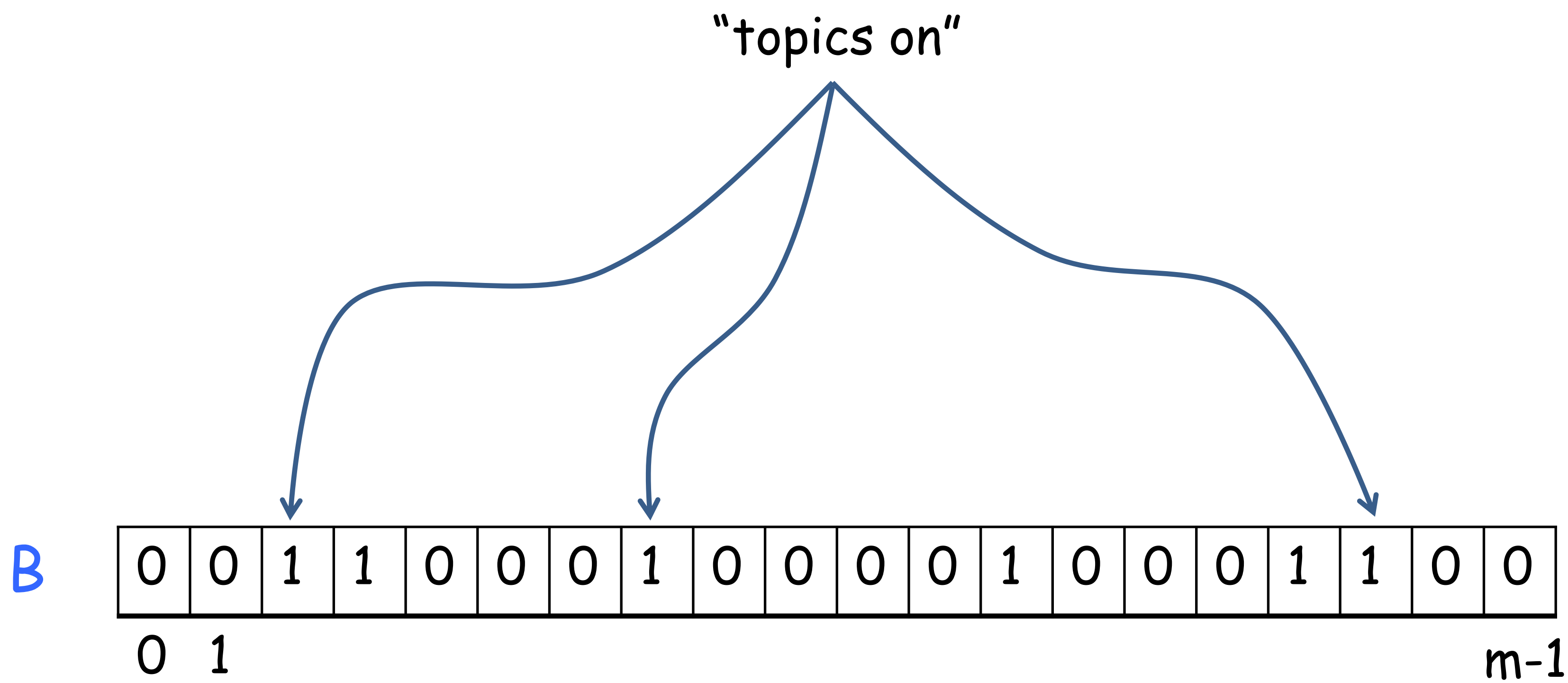
k=3

insert("topics on")



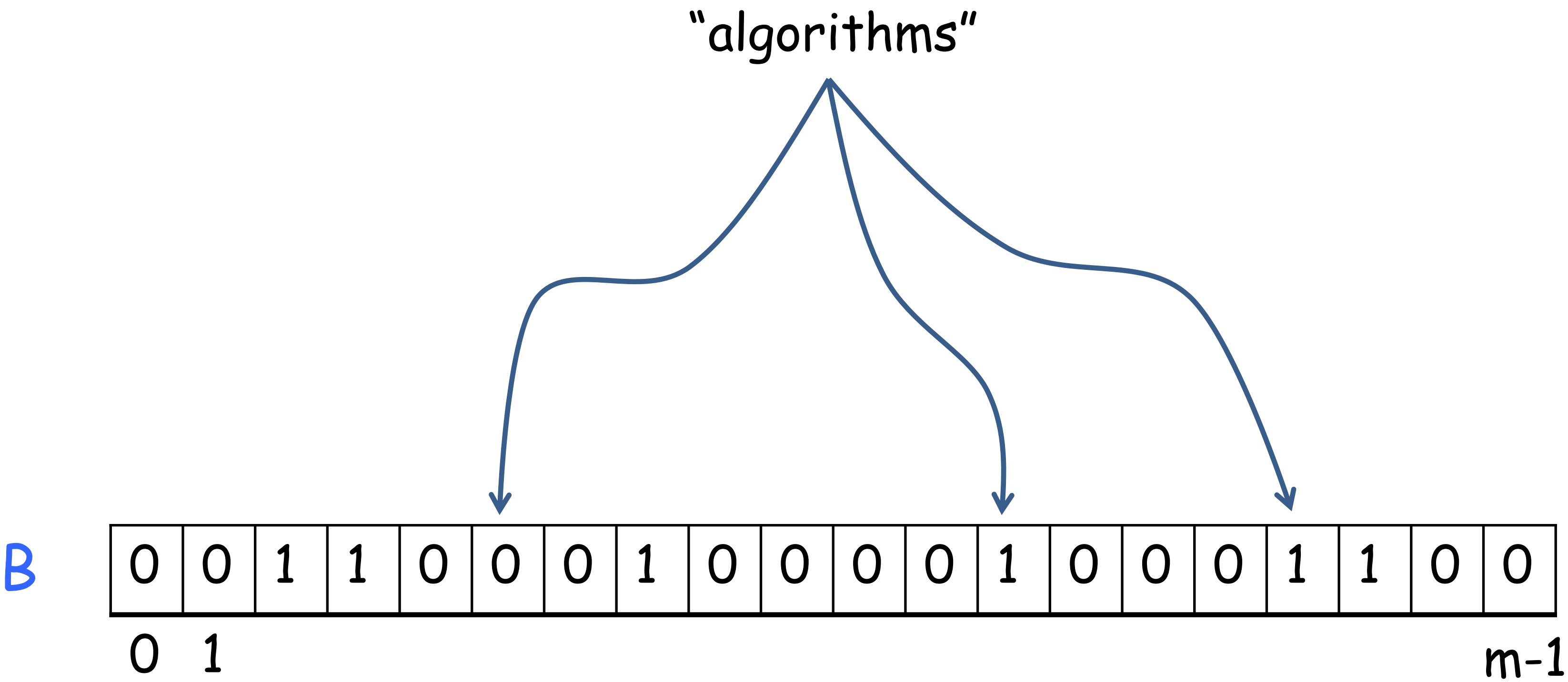
S={"advanced", "topics on"}  k=3

insert("topics on")



$S$={"advanced", "topics on"} $k$=3

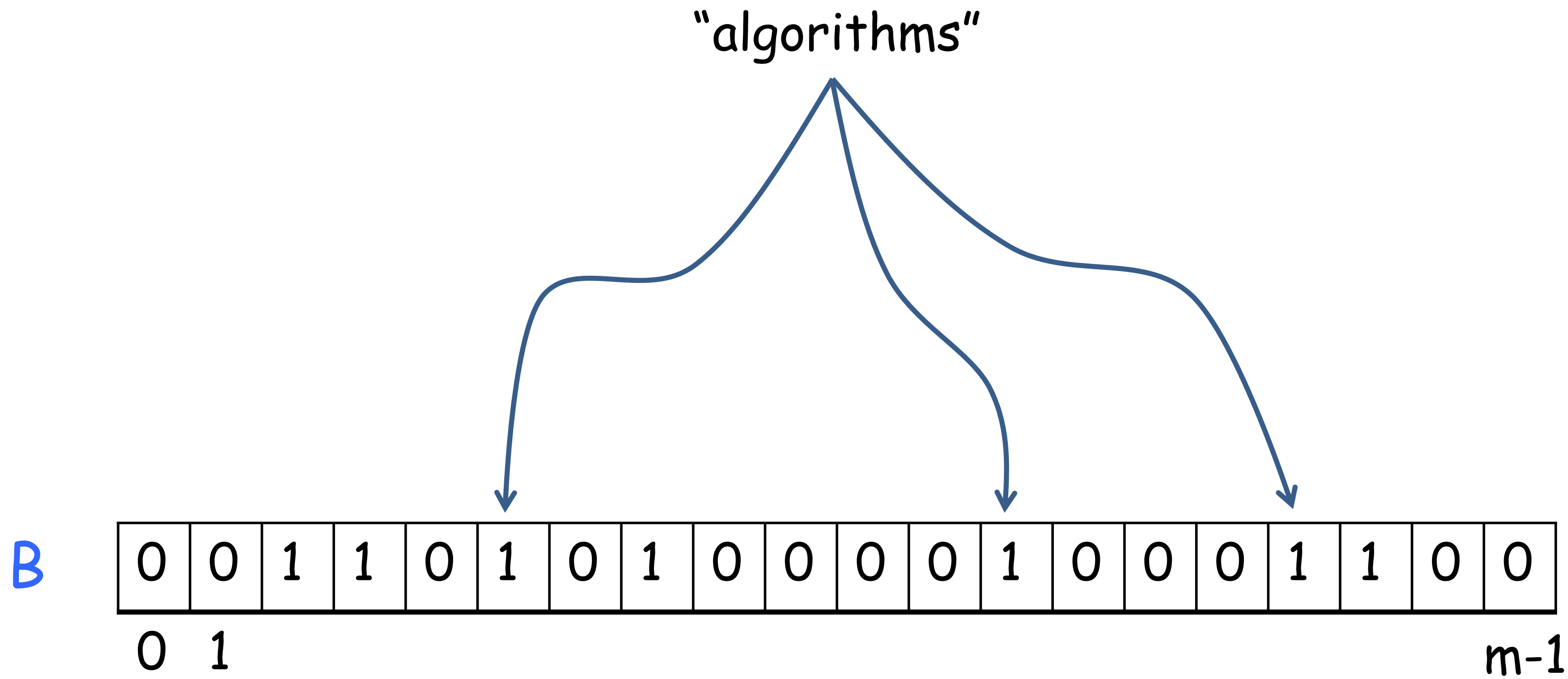
insert("algorithms")



S={“advanced”, “topics on”, “algorithms”}

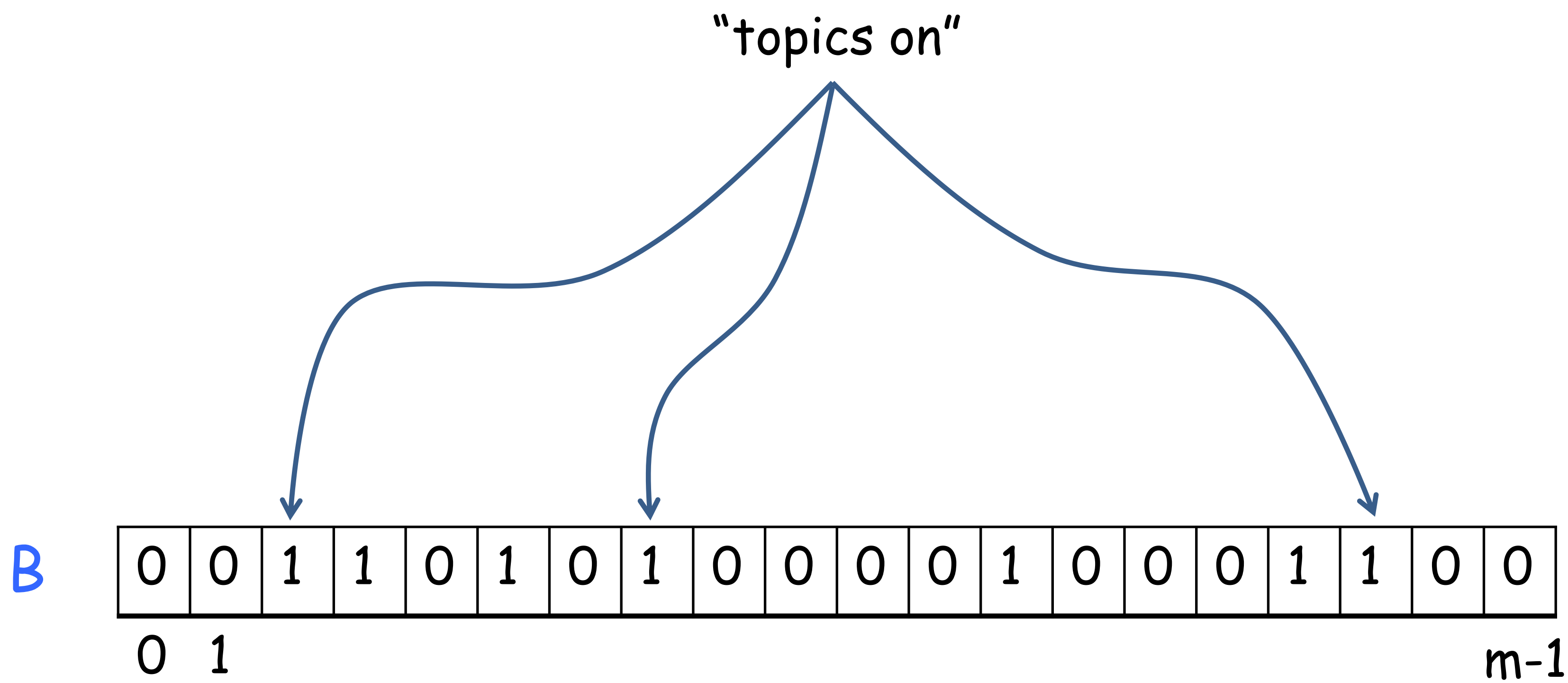
k=3

insert(“algorithms”)



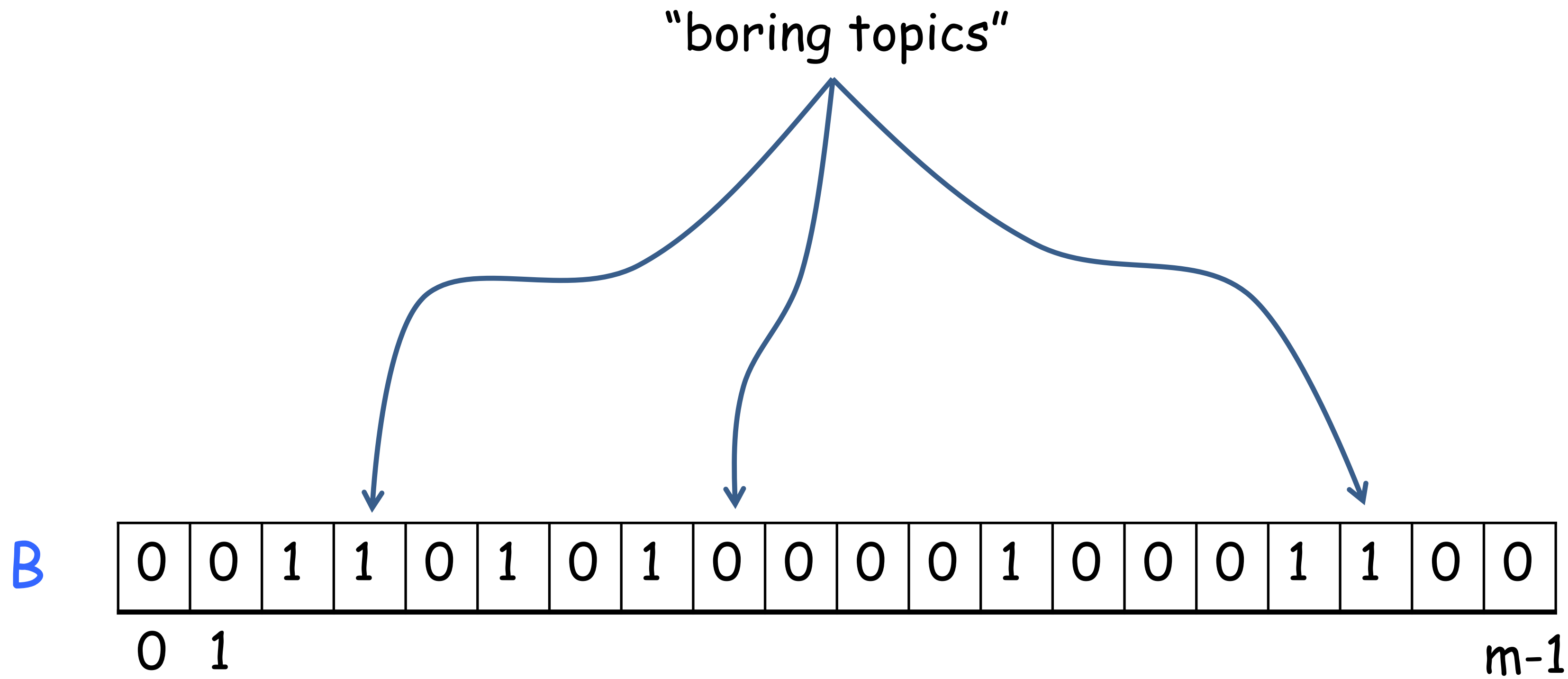
S={"advanced", "topics on", "algorithms"} k=3

membership("topics on") YES correct answer



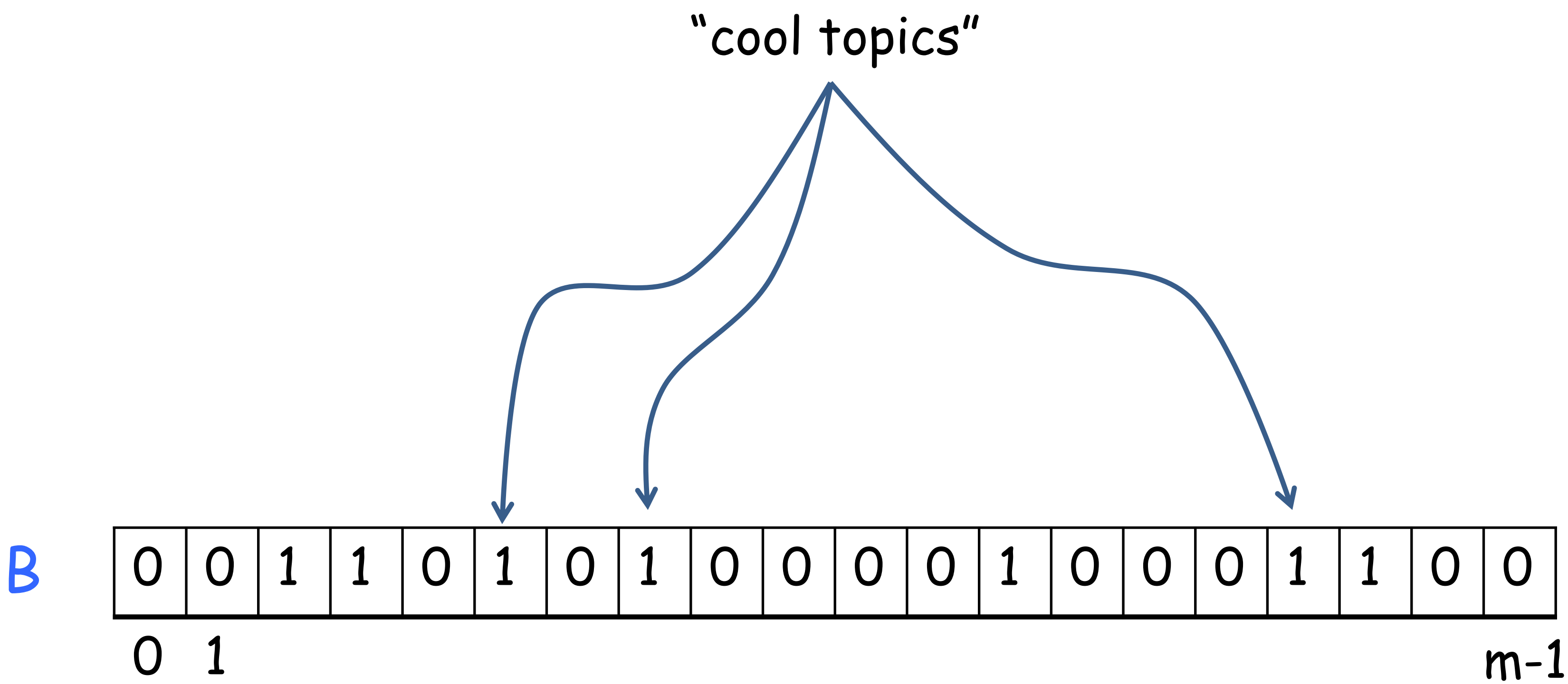
S={“advanced”, “topics on”, “algorithms”} k=3

membership(“boring topics”) NO correct answer



S={“advanced”, “topics on”, “algorithms”} k=3

membership(“cool topics”) YES false positive!



# The analysis

prob that a given bit is still 0 after the first insertion: $(1-1/m)^k$

after all the n insertions: $(1-1/m)^{nk} = \Big[\underbrace{(1-1/m)^m}_{\approx e^{-1}}\Big]^{nk/m} \approx \underbrace{e^{-nk/m}}_{p}$

prob that a given bit is 1 after all insertions: $1-p$

prob of a false positive: $(1-p)^k = \left[1 - e^{-nk/m}\right]^k$

minimized for $k=(m/n)\ln 2$

prob of a false positive: $(1/2)^{(m/n)\ln 2} = \delta$

$\Rightarrow$ $m = n \log_2 e \log_2(1/\delta) \approx 1.44\, n \log_2(1/\delta)$

$k = (m/n)\ln 2 = \log_2(1/\delta)$

**Theorem**

Let $0 < \delta < 1$ be a user-defined parameter, and let n be a maximum capacity. Using $k = \log_2(1/\delta)$ hash functions and $m = n \log_2 e \log_2(1/\delta)$ bits of space, the Bloom Filter guarantees false positive probability at most $\delta$, provided that no more than n elements are inserted into the set.

## Example

we want to store $n=10^7$ malicious URLs with false positive probability $\delta=0.1$

The average URL length is around 77 bytes

⇒ just storing all URLs would require 734 MiB

choosing $k=3$ and $m=38.100.000$

the Bloom filter space: 5.73 MiB (about 5 bits per URL)

⇒ 128 times less space than the plain URLs!

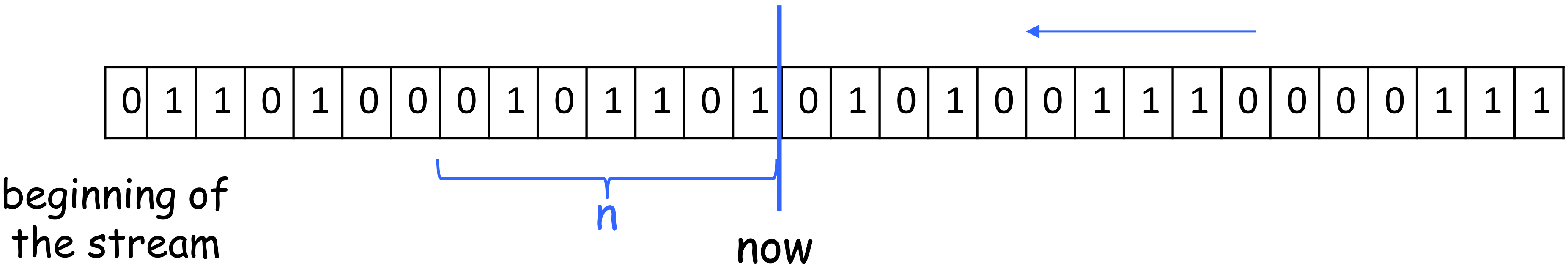
speeds up negative queries by one order of magnitude

(assuming the filter resides locally in RAM and the URLs are on a separate server or on a local disk)

# Counting 1s in a window

Datar-Gionis-Indyk-Motwani's (DGIM) algorithm

# The problem




goal: process a stream of bits in order to answer queries of the type:

- how many 1s in the last n bits?

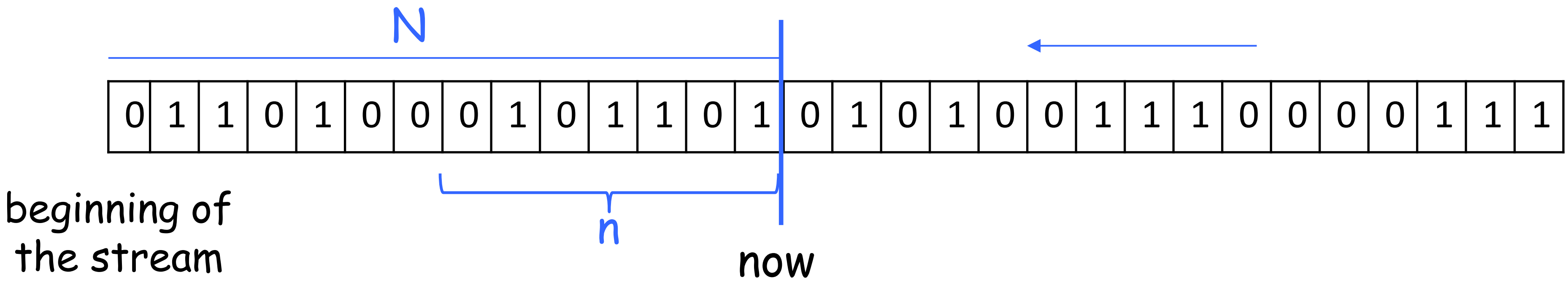
motivation: (approximately) count the events that meet a certain criterion.

Example:
Bank transactions are marked with a flag=1 when exceed a given threshold. Queries can be used to detect if the credit card's owner has changed behavior (hence detect potential frauds)

Example:
Posts/tweets are marked with a flag=1 when they are about a given topic. Queries can be used to detect if the interest on the topic changes.

main challenge: the stream is too large to be entirely stored.

# The problem




goal: design a data structure maintaining a sequence of N bits subject to:

- query(n): return the number of 1s in the last n bits;
- update(b): add the next bit $b \in \{0,1\}$ to the sequence

notice: if you want exact answers you need $\Omega(N)$ bits.

## DGIM data structure:

- quality: $1+\varepsilon$ approximated answers (for any $\varepsilon>0$)
- size: $O(\varepsilon^{-1} \log^2 N)$ bits
- update time: $O(\log N)$
- query time: $O(\varepsilon^{-1} \log n)$

## DGIM data structure:

Let $B = \lceil 1/\varepsilon \rceil$.

Group the bits of the sequence in groups $G_1,\ldots,G_t$ satisfying:

1. each $G_i$ begins and ends with a 1-bit;
2. between adjacent groups $G_i$ $G_{i+1}$ there are only 0-bits;
3. each $G_i$ contains $2^k$ 1-bits, for some $k \geq 0$;
4. for any $1 \leq i < t$, if $G_i$ contains $2^k$ 1-bits, then $G_{i+1}$ contains either $2^k$ or $2^{k-1}$ 1-bits;
5. for each k except the largest one, the number $Z_k$ of groups containing $2^k$ 1-bits satisfies $B \leq Z_k \leq B+1$. For the largest k, we only require $Z_k \leq B+1$.

$$0\ \overset{G_1\,(8)}{\boxed{1\ 1\ 0\ 1\ 1\ 1\ 1\ 1\ 1}}\ 0\ 0\ \overset{G_2\,(4)}{\boxed{1\ 1\ 0\ 1\ 1}}\ 0\ \overset{G_3\,(4)}{\boxed{1\ 0\ 1\ 1\ 1}}\ 0\ \overset{G_4\,(2)}{\boxed{1\ 1}}\ 0\ \overset{G_5\,(2)}{\boxed{1\ 1}}\ \overset{G_6\,(1)}{\boxed{1}}$$

$B=1$

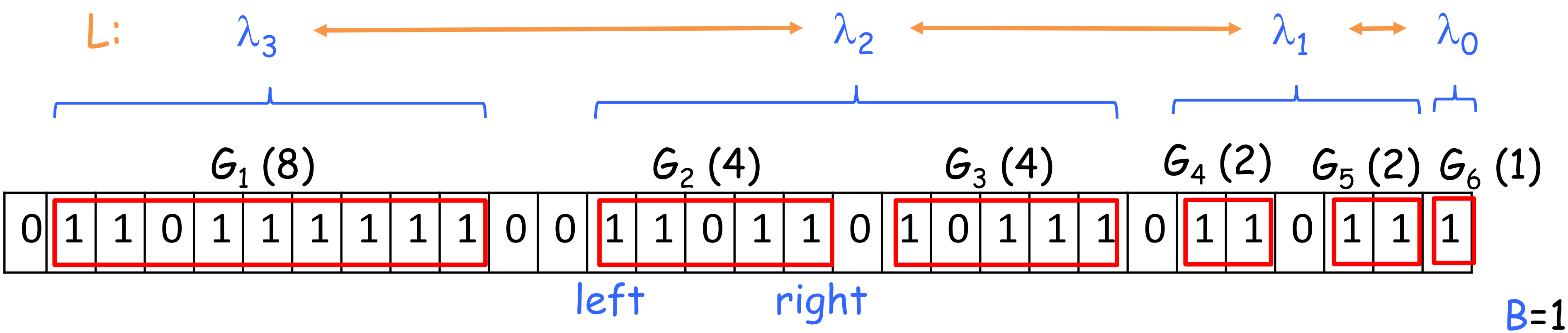
DGIM data structure:

group $G_j$ is a pair of integers (left,right)

all adjacent groups having $2^i$ 1-bits are maintained by a doubly-linked list $\lambda_i$

$\lambda_i$ stores: head, tail, and size

L: a global doubly-linked list storing all lists $\lambda_i$




- storing $G_j$ requires $O(\log N)$ bits
- $|L| = O(\log N)$
- $|\lambda_i| \leq B+1 = O(\varepsilon^{-1})$

overall size of the DS: $O(\varepsilon^{-1} \log^2 N)$ bits

## update operation

update($b$): add the next bit $b \in \{0,1\}$ to the sequence

If $b$=0 do nothing. Otherwise ($b$=1):

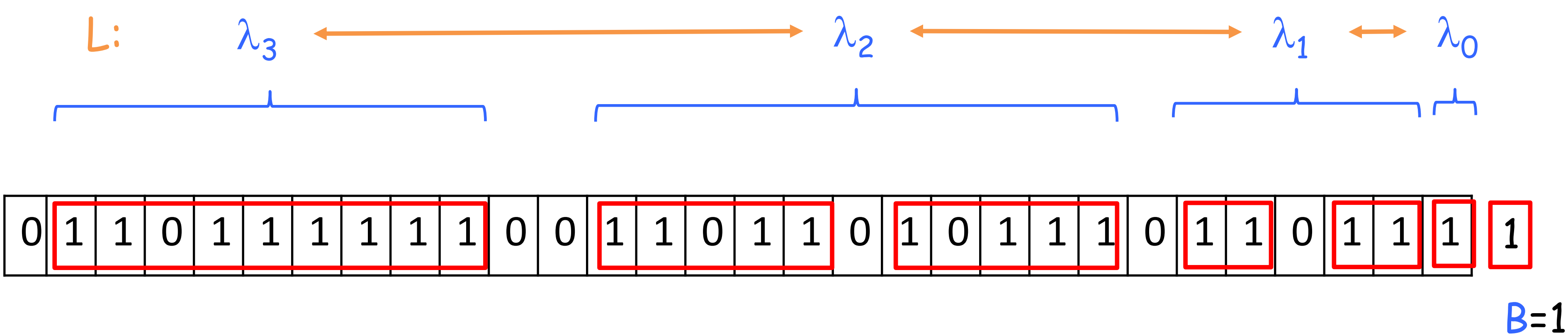
1. Create a new group with the new 1-bit and add it to $\lambda_0$;
2. if $\lambda_0$ contains B+2 groups, merge the two leftmost groups thus forming a new group of 2 1-bits and add it to $\lambda_1$ as a new rightmost group (notice that $\lambda_0$ now has B groups);
3. repeat step 2 for $\lambda_i$ , i=1,2,...;

# update operation

update($b$): add the next bit $b \in \{0,1\}$ to the sequence

If $b$=0 do nothing. Otherwise ($b$=1):

1. Create a new group with the new 1-bit and add it to $\lambda_0$;
2. if $\lambda_0$ contains B+2 groups, merge the two leftmost groups thus forming a new group of 2 1-bits and add it to $\lambda_1$ as a new rightmost group (notice that $\lambda_0$ now has B groups);
3. repeat step 2 for $\lambda_i$ , i=1,2,...;

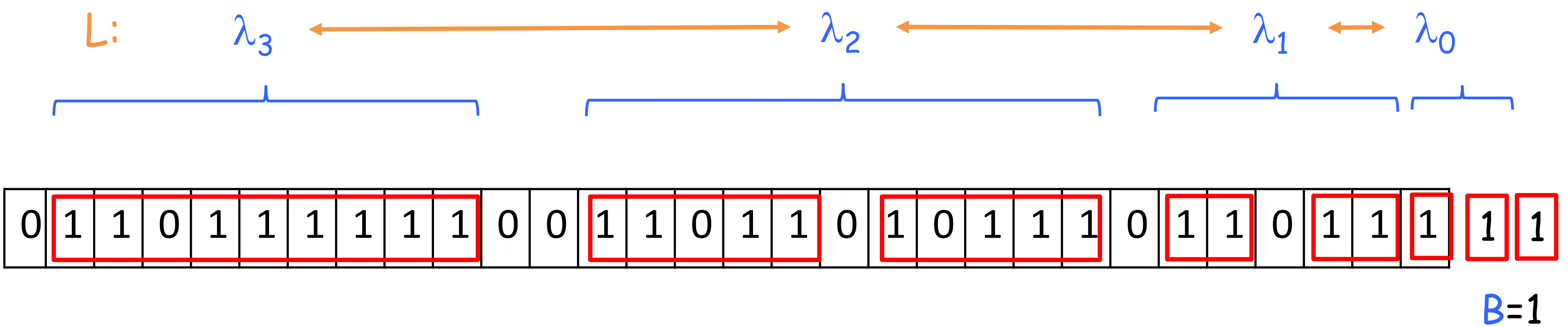


## update operation

update($b$): add the next bit $b \in \{0,1\}$ to the sequence

If $b$=0 do nothing. Otherwise ($b$=1):

1. Create a new group with the new 1-bit and add it to $\lambda_0$;
2. if $\lambda_0$ contains B+2 groups, merge the two leftmost groups thus forming a new group of 2 1-bits and add it to $\lambda_1$ as a new rightmost group (notice that $\lambda_0$ now has B groups);
3. repeat step 2 for $\lambda_i$ , i=1,2,...;

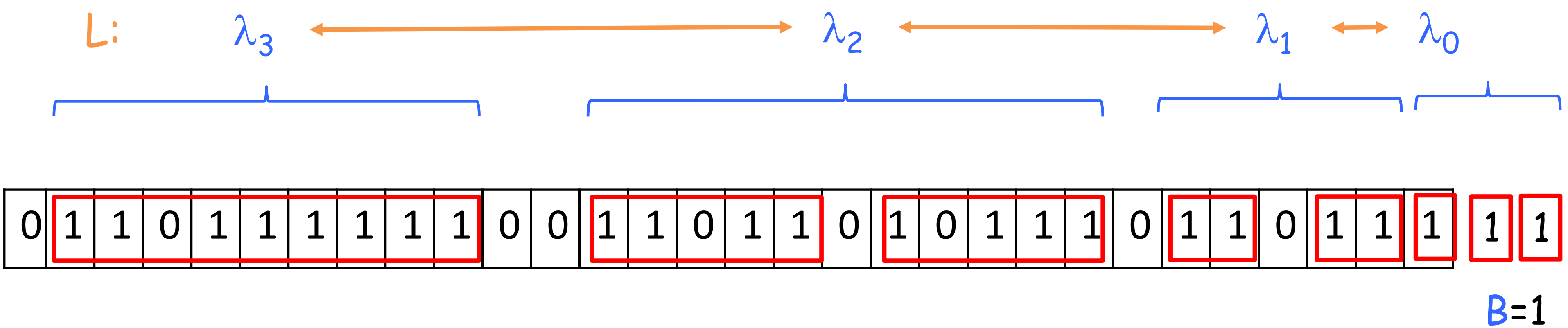


# update operation

update($b$): add the next bit $b \in \{0,1\}$ to the sequence

If $b=0$ do nothing. Otherwise ($b=1$):

1. Create a new group with the new 1-bit and add it to $\lambda_0$;
2. if $\lambda_0$ contains B+2 groups, merge the two leftmost groups thus forming a new group of 2 1-bits and add it to $\lambda_1$ as a new rightmost group (notice that $\lambda_0$ now has B groups);
3. repeat step 2 for $\lambda_i$ , i=1,2,...;

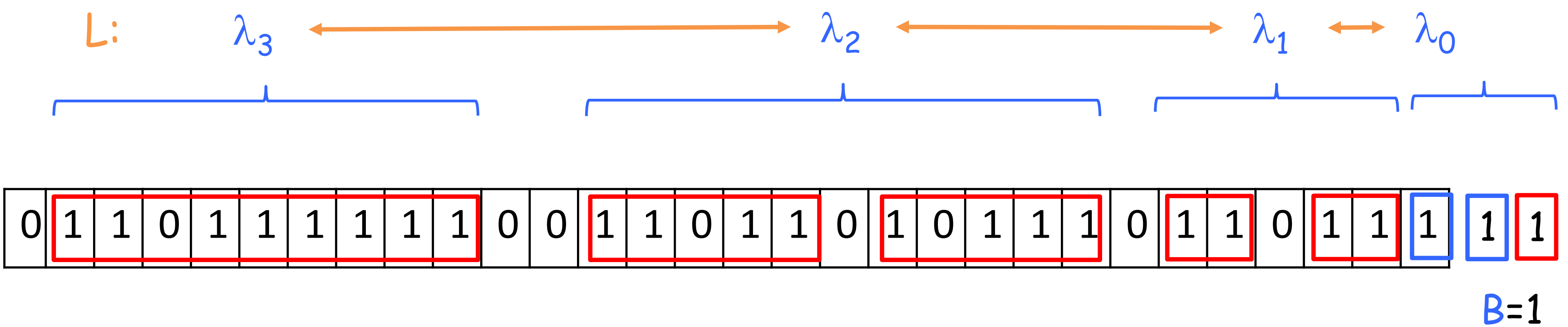


# update operation

update($b$): add the next bit $b \in \{0,1\}$ to the sequence

If $b=0$ do nothing. Otherwise ($b=1$):

1. Create a new group with the new 1-bit and add it to $\lambda_0$;
2. if $\lambda_0$ contains B+2 groups, merge the two leftmost groups thus forming a new group of 2 1-bits and add it to $\lambda_1$ as a new rightmost group (notice that $\lambda_0$ now has B groups);
3. repeat step 2 for $\lambda_i$ , i=1,2,...;

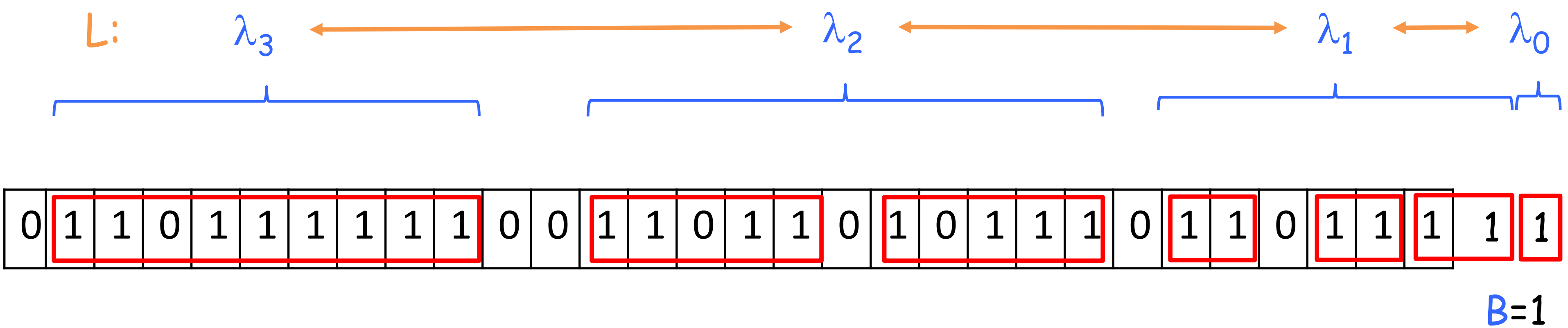


## update operation

update($b$): add the next bit $b \in \{0,1\}$ to the sequence

If $b$=0 do nothing. Otherwise ($b$=1):

1. Create a new group with the new 1-bit and add it to $\lambda_0$;
2. if $\lambda_0$ contains B+2 groups, merge the two leftmost groups thus forming a new group of 2 1-bits and add it to $\lambda_1$ as a new rightmost group (notice that $\lambda_0$ now has B groups);
3. repeat step 2 for $\lambda_i$ , i=1,2,...;

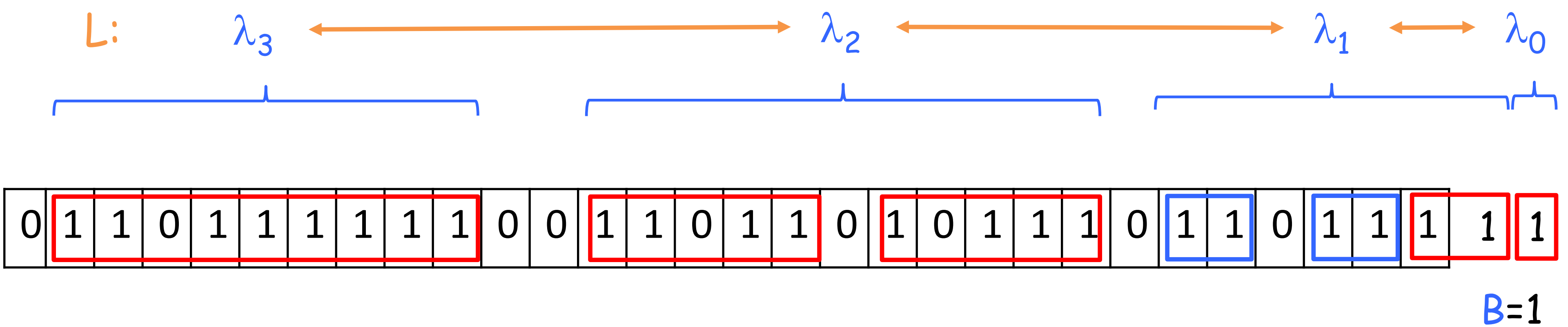


# update operation

update(b): add the next bit $b \in \{0,1\}$ to the sequence

If $b=0$ do nothing. Otherwise ($b=1$):

1. Create a new group with the new 1-bit and add it to $\lambda_0$;
2. if $\lambda_0$ contains B+2 groups, merge the two leftmost groups thus forming a new group of 2 1-bits and add it to $\lambda_1$ as a new rightmost group (notice that $\lambda_0$ now has B groups);
3. repeat step 2 for $\lambda_i$ , i=1,2,...;

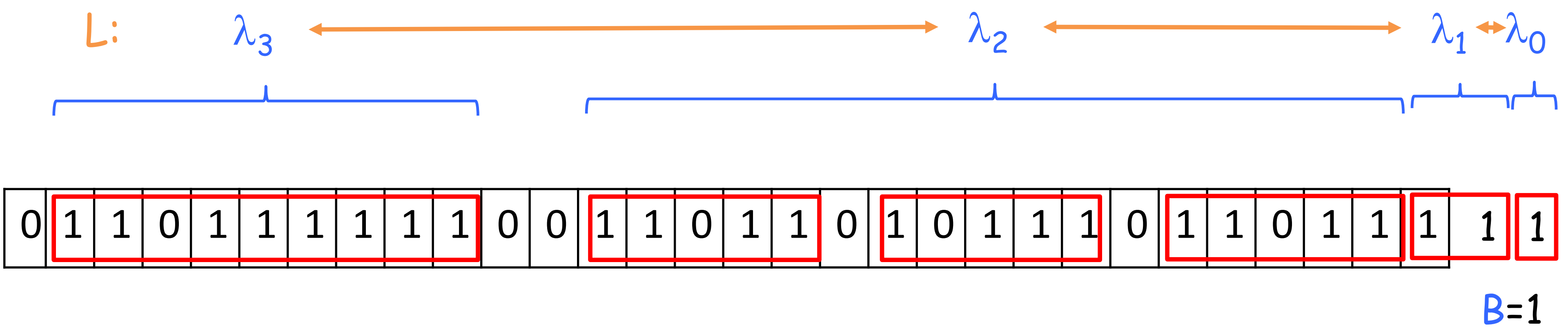


## update operation

update($b$): add the next bit $b \in \{0,1\}$ to the sequence

If $b$=0 do nothing. Otherwise ($b$=1):

1. Create a new group with the new 1-bit and add it to $\lambda_0$;
2. if $\lambda_0$ contains B+2 groups, merge the two leftmost groups thus forming a new group of 2 1-bits and add it to $\lambda_1$ as a new rightmost group (notice that $\lambda_0$ now has B groups);
3. repeat step 2 for $\lambda_i$ , i=1,2,...;

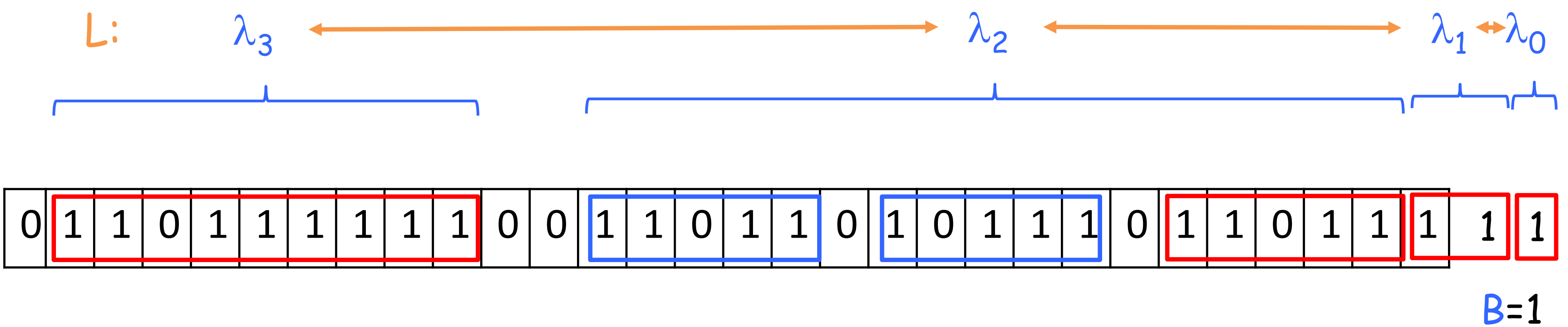


## update operation

update($b$): add the next bit $b \in \{0,1\}$ to the sequence

If $b$=0 do nothing. Otherwise ($b$=1):

1. Create a new group with the new 1-bit and add it to $\lambda_0$;
2. if $\lambda_0$ contains B+2 groups, merge the two leftmost groups thus forming a new group of 2 1-bits and add it to $\lambda_1$ as a new rightmost group (notice that $\lambda_0$ now has B groups);
3. repeat step 2 for $\lambda_i$ , i=1,2,...;

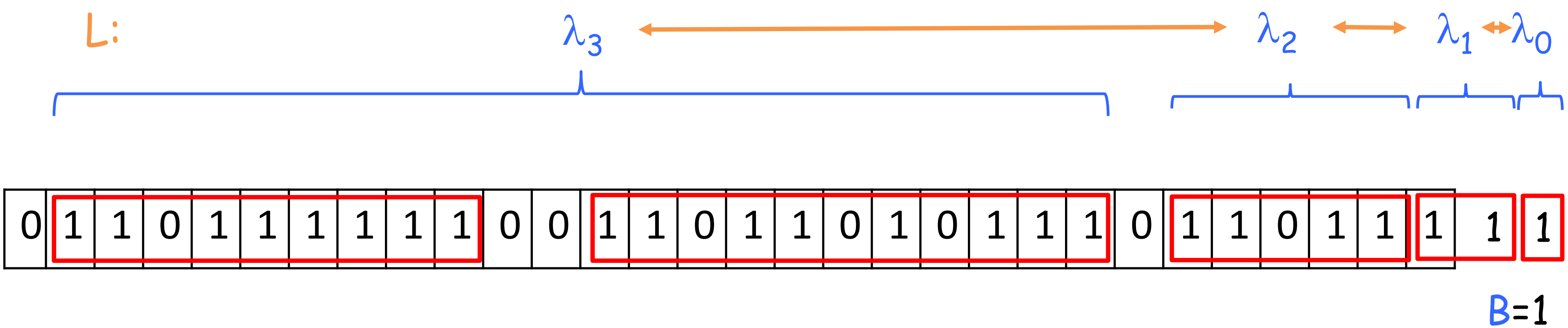


# update operation

update(b): add the next bit $b \in \{0,1\}$ to the sequence

If $b=0$ do nothing. Otherwise ($b=1$):

1. Create a new group with the new 1-bit and add it to $\lambda_0$;
2. if $\lambda_0$ contains B+2 groups, merge the two leftmost groups thus forming a new group of 2 1-bits and add it to $\lambda_1$ as a new rightmost group (notice that $\lambda_0$ now has B groups);
3. repeat step 2 for $\lambda_i$ , i=1,2,...;



## update operation

update($b$): add the next bit $b \in \{0,1\}$ to the sequence

If $b$=0 do nothing. Otherwise ($b$=1):

1. Create a new group with the new 1-bit and add it to $\lambda_0$;
2. if $\lambda_0$ contains B+2 groups, merge the two leftmost groups thus forming a new group of 2 1-bits and add it to $\lambda_1$ as a new rightmost group (notice that $\lambda_0$ now has B groups);
3. repeat step 2 for $\lambda_i$ , i=1,2,...;

update time:

- creating/merging/moving a group takes O(1) time
- number of iterations O(|L|)

⇒

overall update time: O(log N)

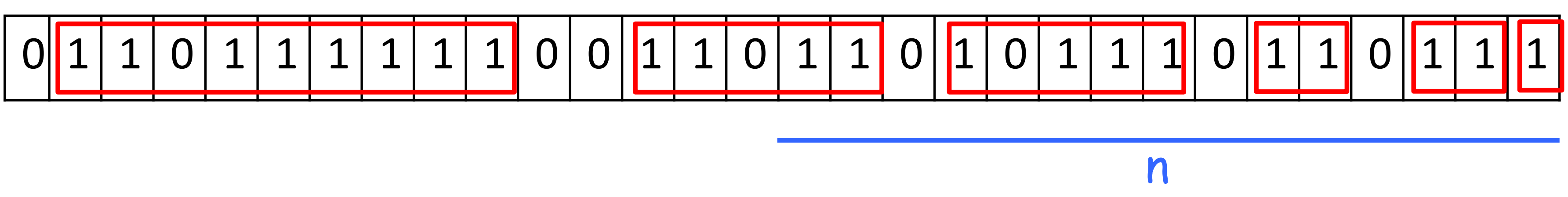
# query operation

query(n): return the number of 1s in the last n bits

- find all groups intersecting the last n bits
- return the number of 1-bits they contain

query time:

- navigating all groups from the streaming's head
- $O(\varepsilon^{-1} \log n)$ time



query operation: approximation

Let $k$ be the integer s.t. the leftmost intersecting group has $2^k$ 1-bits

$y$: right answer
$x$: returned answer

notice: if $k$=0 then $x$=$y$ (so assume $k$>0)

$x \leq y + 2^k - 1$

$y \geq B\,2^{k-1} + B\,2^{k-2} + \ldots + B\,2^1 + B\,2^0 = B(2^k - 1)$

$x/y \leq (y + 2^k - 1)/\,y = 1 + (2^k - 1)/\,y$

$\leq 1 + 1/B \leq 1 + \varepsilon$

$2^k$ 1-bits

| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 1 |

$n$