Module 4 University Questions

Q. Explain the concept of Bloom Filter algorithm with an example. [5M/10M] May-2024/ May 2022

What is Bloom Filter?

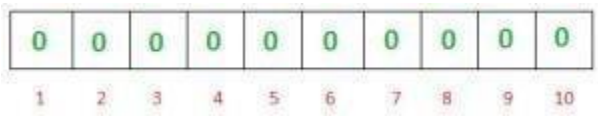
A Bloom filter is a space-efficient probabilistic data structure that is used to test whether an element is a member of a set. For example, checking availability of username is set membership problem, where the set is the list of all registered username. The price we pay for efficiency is that it is probabilistic in nature that means, there might be some False Positive results. False positive means, it might tell that given username is already taken but actually it's not.

Interesting Properties of Bloom Filters

- Unlike a standard hash table, a Bloom filter of a fixed size can represent a set with an arbitrarily large number of elements.
- Adding an element never fails. However, the false positive rate increases steadily as elements are added until all bits in the filter are set to 1, at which point all queries yield a positive result.
- Bloom filters never generate false negative result, i.e., telling you that a username doesn't exist when it actually exists.
- Deleting elements from filter is not possible because, if we delete a single element by clearing bits at indices generated by k hash functions, it might cause deletion of few other elements. Example – if we delete "geeks" (in given example below) by clearing bit at 1, 4 and 7, we might end up deleting "nerd" also Because bit at index 4 becomes 0 and bloom filter claims that "nerd" is not present.

Working of Bloom Filter

A empty bloom filter is a bit array of m bits, all set to zero, like this –



We need k number of hash functions to calculate the hashes for a given input. When we want to add an item in the filter, the bits at k indices h1(x), h2(x), … hk(x) are set, where indices are calculated using hash functions.

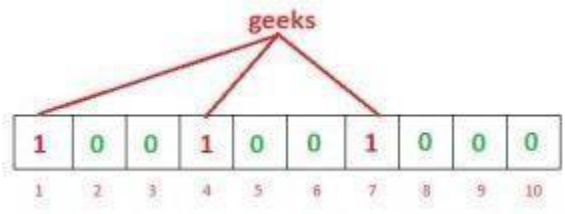
Example – Suppose we want to enter "geeks" in the filter, we are using 3 hash functions and a bit array of length 10, all set to 0 initially. First we'll calculate the hashes as following :

h1("geeks") % 10 = 1

h2("geeks") % 10 = 4

h3("geeks") % 10 = 7
Note: These outputs are random for explanation only.

Now we will set the bits at indices 1, 4 and 7 to 1



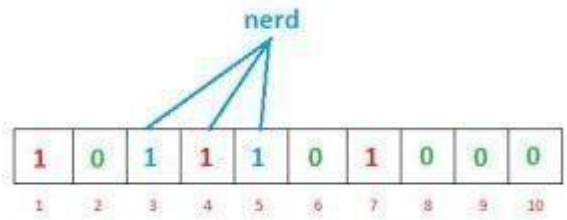
Again we want to enter “nerd”, similarly we’ll calculate hashes

h1(“nerd”) % 10 = 3

h2(“nerd”) % 10 = 5

h3(“nerd”) % 10 = 4

Set the bits at indices 3, 5 and 4 to 1



Now if we want to check “geeks” is present in filter or not. We’ll do the same process but this time in reverse order. We calculate respective hashes using h1, h2 and h3 and check if all these indices are set to 1 in the bit array. If all the bits are set then we can say that “geeks” is probably present. If any of the bit at these indices are 0 then “geeks” is definitely not present.

False Positive in Bloom Filters

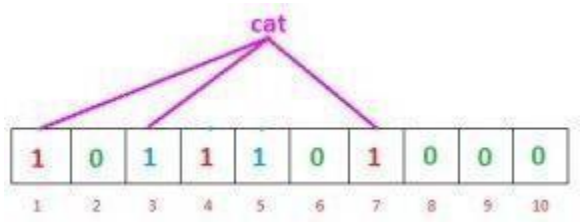
The question is why we said “probably present”, why this uncertainty. Let’s understand this with an example. Suppose we want to check whether “cat” is present or not. We’ll calculate hashes using h1, h2 and h3

h1(“cat”) % 10 = 1

h2(“cat”) % 10 = 3

h3(“cat”) % 10 = 7

If we check the bit array, bits at these indices are set to 1 but we know that “cat” was never added to the filter. Bit at index 1 and 7 was set when we added “geeks” and bit 3 was set we added “nerd”.



So, because bits at calculated indices are already set by some other item, bloom filter erroneously claim that “cat” is present and generating a false positive result. Depending on the application, it could be huge downside or relatively okay.

We can control the probability of getting a false positive by controlling the size of the Bloom filter. More space means fewer false positives. If we want decrease probability of false positive result, we have to use more number of hash functions and larger bit array. This would add latency in addition of item and checking membership.

Probability of False positivity: Let m be the size of bit array, k be the number of hash functions and n be the number of expected elements to be inserted in the filter, then the probability of false positive p can be calculated as:

$$P = \left(1 - \left[1 - \frac{1}{m}\right]^{kn}\right)^k$$

Size of Bit Array: If expected number of elements n is known and desired false positive probability is p then the size of bit array m can be calculated as :

$$m = -\frac{n \ln P}{(ln2)^2}$$

Optimum number of hash functions: The number of hash functions k must be a positive integer. If m is size of bit array and n is number of elements to be inserted, then k can be calculated as :

$$k = \frac{m}{n} ln2$$

OR (GIVE YOUR OWN EXAMPLE)

A set consists of some elements say 8, 10, ...
Check whether 48, 7 lies in this set or Not
Set size = 10
Given hash function

$(3x+3) \bmod 6$

$(3x+7) \bmod 8$

$(2x+9) \bmod 2$

$(2x+3) \bmod 5$

Solution:-

Step 1 : Initialise Array

0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0

Step 2: Apply hash function and calculate value. (for phase 1 Insertion)

	x = 8	x = 10
$(3x+3) \bmod 6$	3	3
$(3x+7) \bmod 8$	7	5
$(2x+9) \bmod 2$	1	1
$(2x+3) \bmod 5$	4	3

Step 3: change bit 0 to 1 from above table. (i.e. 3, 7, 1, 4, 5 location will become 0 to 1.

0	1	2	3	4	5	6	7	8	9
0	1	0	1	1	1	0	1	0	0

Step 4: Apply hash function and calculate value (for phase 2 query)

	48	7
$(3x+3) \bmod 6$	3	0
$(3x+7) \bmod 8$	7	4
$(2x+9) \bmod 2$	1	1
$(2x+3) \bmod 5$	4	2

Since all bits 3,7,1,4,are already set to 1 hence 48 may be present in the set.

Since only 4,1 bits are set to 1 and 0,2 bits are set to 0. Hence 7 is surely not present in the set.

Q. Explain DGIM algorithm for counting ones in a stream with the example. [10M] May- 2024/Dec 2024

Why DGIM algorithm?

Suppose we have a window of length N on a binary stream. We want at all times to be able to answer queries of the form "how many 1's are there in the last k bits?" for any k≤ N. For this purpose we use the DGIM algorithm.

The basic version of the algorithm uses O(log2 N) bits to represent a window of N bits, and allows us to estimate the number of 1's in the window with an error of no more than 50%.

To begin, each bit of the stream has a timestamp, the position in which it arrives. The first bit has timestamp 1, the second has timestamp 2, and so on.

Since we only need to distinguish positions within the window of length N, we shall represent timestamps modulo N, so they can be represented by log2 N bits. If we also store the total number of bits ever seen in the stream (i.e., the most recent timestamp) modulo N, then we can determine from a timestamp modulo N where in the current window the bit with that timestamp is.

We divide the window into buckets, 5 consisting of:

1. The timestamp of its right (most recent) end.
2. The number of 1's in the bucket. This number must be a power of 2, and we refer to the number of 1's as the size of the bucket.

To represent a bucket, we need log2 N bits to represent the timestamp (modulo N) of its right end. To represent the number of 1's we only need log2 log2 N bits. The reason is that we know this number i is a power of 2, say 2j , so we can represent i by coding j in binary. Since j is at most log2 N, it requires log2 log2 N bits. Thus, O(logN) bits suffice to represent a bucket. There are six rules that must be followed when representing a stream by buckets.

- The right end of a bucket is always a position with a 1.
- Every position with a 1 is in some bucket.
- No position is in more than one bucket.
- There are one or two buckets of any given size, up to some maximum size.
- All sizes must be a power of 2.
- Buckets cannot decrease in size as we move to the left (back in time).

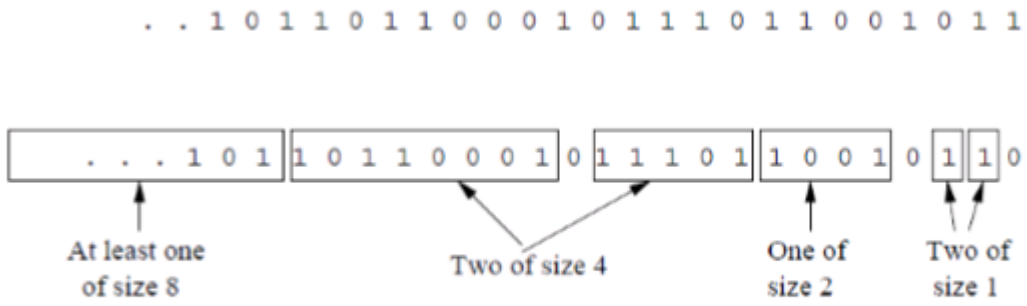


Figure: A bit-stream divided into buckets following the DGIM rules

Q. List and explain different issues in data stream query processing. [5M] May 2023

1. **Unbounded Memory Requirements:**

   Since data streams are potentially unbounded in size, the amount of storage required to compute an exact answer to a data stream query may also grow without bound. While external memory algorithms for handling data sets larger than main memory have been studied, such algorithms are not well suited to data stream applications since they do not support continuous queries and are typically too slow for real-time response. New data is constantly arriving even as the old data is being processed; the amount of computation time per data element must be low, or else the latency of the computation will be too high and the algorithm will not be able to keep pace with the data stream.

2. **Approximate Query Answering:**

   When we are limited to a bounded amount of memory it is not always possible to produce exact answers for data stream queries; however, high-quality approximate answers are often acceptable in lieu of exact answers. Sliding Window: One technique for producing an approximate answer to a data stream query is to evaluate the query not over the entire past history of the data streams, but rather only over sliding windows of recent data from the streams. For example, only data from the last week could be considered in producing query answers, with data older than one week being discarded.

3. **Blocking Operators:**

   A blocking query operator is a query operator that is unable to produce the first tuple of its output until it has seen its entire input. If one thinks about evaluating continuous stream queries using a traditional tree of query operators, where data streams enter at the leaves and final query answers are produced at the root, then the incorporation of blocking operators into the query tree poses problems. Since continuous data

streams may be infinite, a blocking operator that has a data stream as one of its inputs will never see its entire input, and therefore it will never be able to produce any output. Doing away with blocking operators altogether would be problematic, but dealing with them effectively is one of the more challenging aspects of data stream computation.

4. **Queries Referencing Past Data:**

   In the data stream model of computation, once a data element has been streamed by, it cannot be revisited. This limitation means that ad hoc queries that are issued after some data has already been discarded may be impossible to answer accurately. One simple solution to this problem is to stipulate that ad hoc queries are only allowed to reference future data: they are evaluated as though the data streams began at the point when the query was issued, and any past stream elements are ignored (for the purposes of that query). While this solution may not appear very satisfying, it may turn out to be perfectly acceptable for many applications.

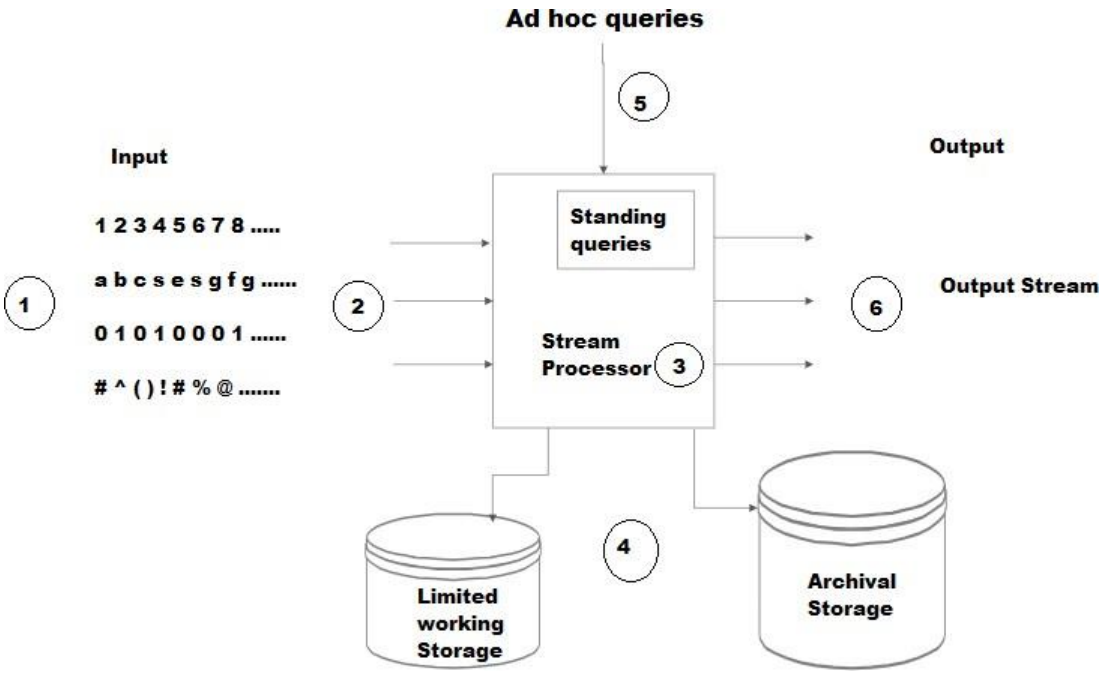
5. **Batch Processing sampling and synopses:** We avoid looking at every data element and restrict query evaluation to some sort of sampling or batch processing technique. In batch processing, rather than producing a continually up to date answer, the data elements are buffered as they arrive and the answer to the query is computed periodically. This is an approximate answer at point in the recent past rather than exact answer at the present moment.

   For some classes of data stream queries where no exact data structure with the desired properties exist, one can often design an approximate data structure that maintains a small synopsis or sketch of the data. (computation per data element will be minimum.)

6. **Sliding Windows:** One technique for approximate query answering is to evaluate the query not over the entire past history of the data streams, but rather only over sliding windows of the recent data from the streams. For example,

   Only last week could be considered in producing query answers, with data older than 1 week being discarded. Recent data, in the real world applications is more important and relevant than the old data.

Q. With neat sketch explain the architecture of the data stream management system. [10M] May 2023/ May 2024



DSMS consists of various layer which are dedicated to perform particular operation which are as follows:

1. Data source Layer

The first layer of DSMS is data source layer as it name suggest it is comprises of all the data sources which includes sensors, social media feeds, financial market, stock markets etc. In the layer capturing and parsing of data stream happens. Basically it is the collection layer which collects the data.

2. Data Ingestion Layer

You can consider this layer as bridge between data source layer and processing layer. The main purpose of this layer is to handle the flow of data i.e., data flow control, data buffering and data routing.

3. Processing Layer

This layer consider as heart of DSMS architecture it is functional layer of DSMS applications. It process the data streams in real time. To perform processing it is uses processing engines like Apache flink or Apache storm etc., The main function of this layer is to filter, transform, aggregate and enriching the data stream. This can be done by derive insights and detect patterns.

4. Storage Layer

Once data is processed we need to store the processed data in any storage unit. Storage layer consist of various storage like NoSQL database, distributed database etc., It helps to ensure data durability and availability of data in case of system failure.

5. Querying Layer

As mentioned above it support 2 types of query ad hoc query and standard query. This layer provides the tools which can be used for querying and analyzing the stored data stream. It also have SQL like query languages or programming API. This queries can be question like how many entries are done? which type of data is inserted? etc.,

6. Visualization and Reporting Layer

This layer provides tools for performing visualization like charts, pie chart, histogram etc., On the basis of this visual representation it also helps to generate the report for analysis.

7. Integration Layer

This layer is responsible for integrating DSMS application with traditional system, business intelligence tools, data warehouses, ML application, NLP applications. It helps to improve already present running applications.

The layers are responsible for working of DSMS applications. It provides scalable and fault tolerance application which can handle huge volume of streaming data. These layers can be changed according to the business requirements.

Q. Suppose the stream S is ={2,1,6,1,5,9,2,3,5} Let hash functions h(x)=ax+b mod -[10M]May 2023

Suppose the stream is S = {2, 1, 6, 1, 5, 9, 2, 3, 5}. Let hash functions h(x) = ax + b mod 16 for some a and b, treat result as a 4-bit binary integer. Show how the Flajolet- Martin algorithm will estimate the number of distinct elements, h(x) = 4x + 1 mod 16.

Q. Suppose the stream is $S = \{2,1,6,1,5,9,2,3,5\}$. Let hash functions $h(x) = ax + b \bmod 16$ for some a and b, treat result as a 4 bit binary integer. Show how the FM algorithm will estimate the number of distinct elements $h(x) = 4x + 1 \bmod 16$. [May 2023]

Step 1:- Evaluation of hash fn.

$h(2) = (4*2+1) \bmod 16 = 9$

$h(1) = (4*1+1) \bmod 16 = 5$

$h(6) = (4*6+1) \bmod 16 = 9$

$h(5) = (4*5+1) \bmod 16 = 5$

$h(9) = (4*9+1) \bmod 16 = 5$

$h(3) = (4*3+1) \bmod 16 = 13$

Step 2:- Binary representation.

$h(2) = 9 = 1001$

$h(1) = 5 = 0101$

$h(6) = 9 = 1001$

$h(5) = 5 = 0101$

$h(9) = 5 = 0101$

$h(3) = 13 = 1101$

maximum

Step 3:- count of trailing zeros = 0

$R = 2^r$

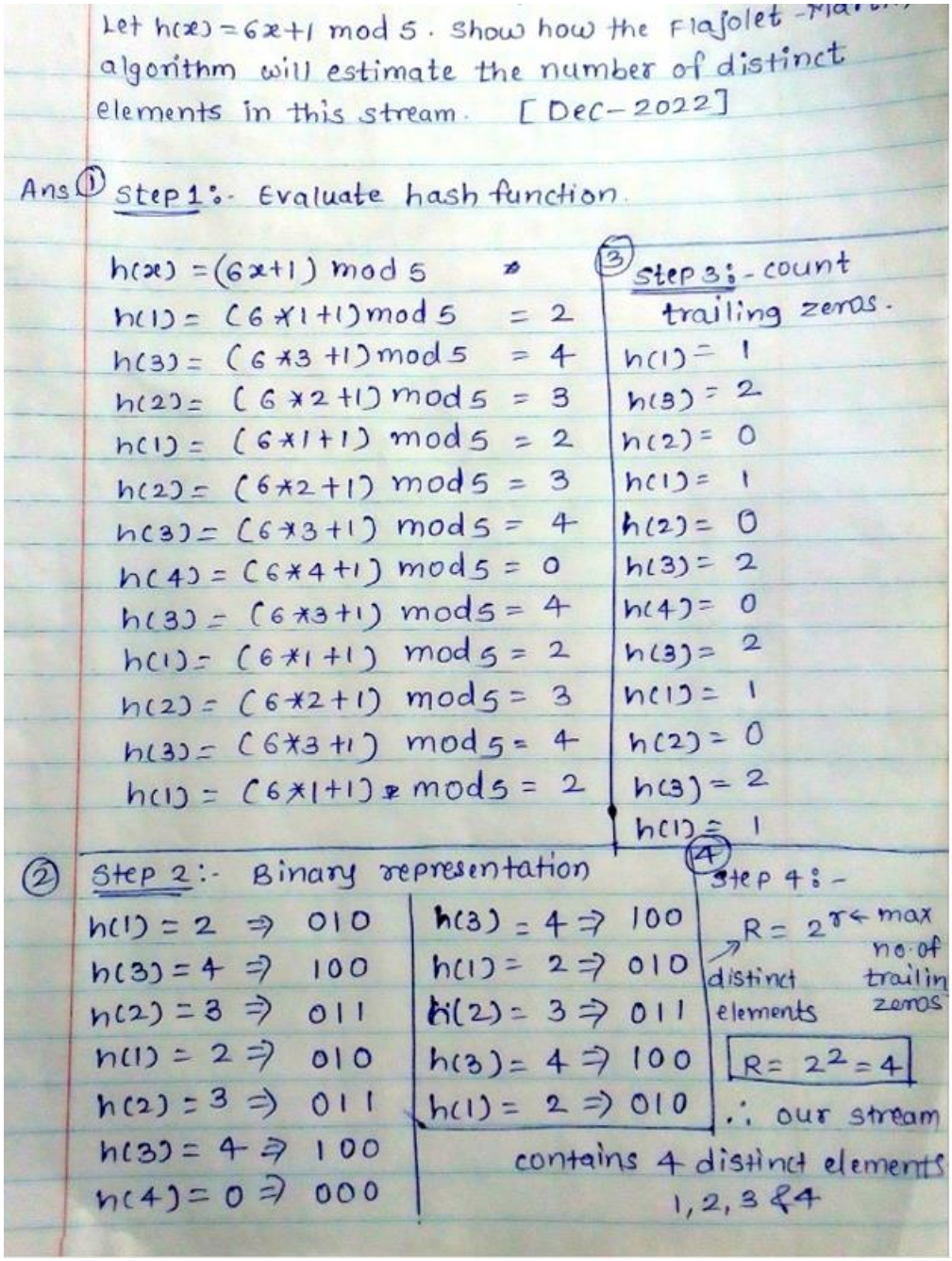
$R = 2^0 = 1$

R → distinct elements

r → maximum number of trailing zeros.

Q. May 2022

Suppose the stream is 1, 3, 2, 1, 2, 3, 4, 3, 1, 2, 3, 1. Let h(x) = 6x + 1 mod 5. Show how the Flajolet- Martin algorithm will estimate the number of distinct elements in this stream.



Q. List all six constraints that must be satisfied for representing a stream by buckets using DGIM algorithm with examples.[5M]Dec 2023

There are six rules that must be followed when representing a stream by buckets.

- The right end of a bucket is always a position with a 1.
- Every position with a 1 is in some bucket.
- No position is in more than one bucket.
- There are one or two buckets of any given size, up to some maximum size.
- All sizes must be a power of 2.

- Buckets cannot decrease in size as we move to the left (back in time).

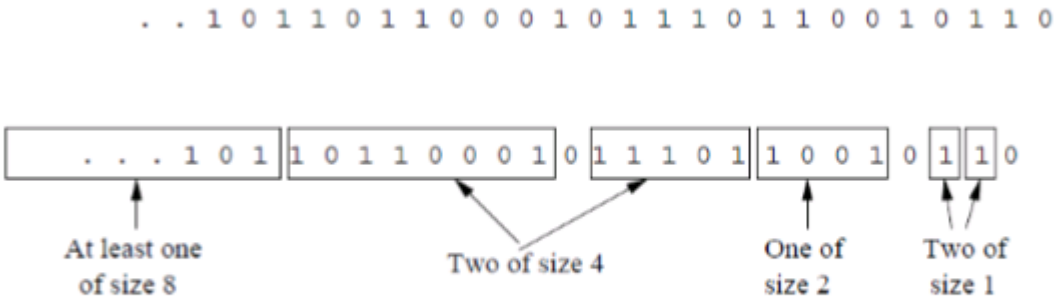
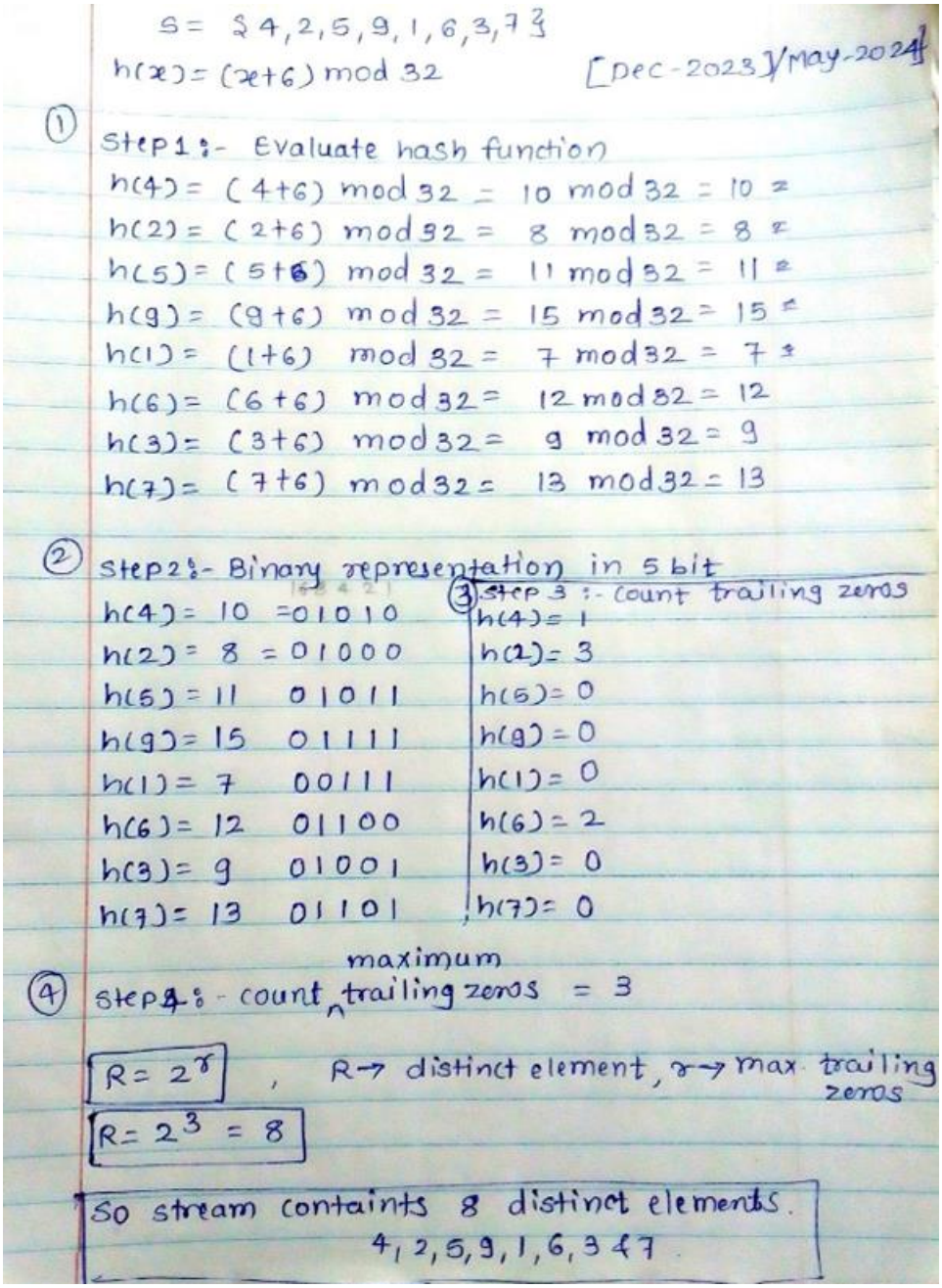


Figure: A bit-stream divided into buckets following the DGIM rules

Suppose the stream is S = {4, 2, 5 ,9, 1, 6, 3, 7}. Let hash functions h(x) = x + 6 mod 32 for some a and b, treat result as a 5-bit binary integer. Show how the Flajolet- Martin algorithm will estimate the number of distinct elements in this stream.



S = {4,2,5,9,1,6,3,7}

h(x) = (x+6) mod 32 [Dec-2023]/May-2024

(1) Step 1:- Evaluate hash function

h(4) = (4+6) mod 32 = 10 mod 32 = 10

h(2) = (2+6) mod 32 = 8 mod 32 = 8

h(5) = (5+6) mod 32 = 11 mod 32 = 11

h(9) = (9+6) mod 32 = 15 mod 32 = 15

h(1) = (1+6) mod 32 = 7 mod 32 = 7

h(6) = (6+6) mod 32 = 12 mod 32 = 12

h(3) = (3+6) mod 32 = 9 mod 32 = 9

h(7) = (7+6) mod 32 = 13 mod 32 = 13

(2) Step 2:- Binary representation in 5 bit

h(4) = 10 = 01010

h(2) = 8 = 01000

h(5) = 11 01011

h(9) = 15 01111

h(1) = 7 00111

h(6) = 12 01100

h(3) = 9 01001

h(7) = 13 01101

(3) Step 3 :- Count trailing zeros

h(4) = 1

h(2) = 3

h(5) = 0

h(9) = 0

h(1) = 0

h(6) = 2

h(3) = 0

h(7) = 0

(4) Step 4 :- count maximum trailing zeros = 3

$R = 2^r$, R → distinct element, r → max trailing zeros

$R = 2^3 = 8$

So stream containts 8 distinct elements.
4, 2, 5, 9, 1, 6, 3 & 7.

Suppose the stream is S = {4, 2, 5 ,9, 1, 6, 3, 7}. Let hash functions h(x) = 3x + 7 mod 32 for some a and b, treat result as a 5-bit binary integer. Show how the Flajolet- Martin algorithm will estimate the number of distinct elements in this stream.

[May 2024]

S = {4,2,5,9,1,6,3,7}

h(x) = (3x + 7) mod 32

① Step 1:- Evaluate hash function

h(4) = (3 * 4) + 7 mod 32 = 19 mod 32 = 19
h(2) = (3 * 2) + 7 mod 32 = 13 mod 32 = 13
h(5) = (3 * 5) + 7 mod 32 = 22 mod 32 = 22
h(9) = (3 * 9) + 7 mod 32 = 34 mod 32 = 2
h(1) = (3 * 1) + 7 mod 32 = 10 mod 32 = 10
h(6) = (3 * 6) + 7 mod 32 = 25 mod 32 = 25
h(3) = (3 * 3) + 7 mod 32 = 16 mod 32 = 16
h(7) = (3 * 7) + 7 mod 32 = 28 mod 32 = 28

② Step 2:- Binary representation

h(4) = 19 = 10011	h(4) = 0
h(2) = 13 = 01101	h(2) = 0
h(5) = 22 = 10110	h(5) = 1
h(9) = 2 = 00010	h(9) = 1
h(1) = 10 = 01010	h(1) = 1
h(6) = 25 = 11001	h(6) = 0
h(3) = 16 = 10000	h(3) = 4
h(7) = 28 = 11100	h(7) = 2

③ Step 3:- count trailing zeros.

④ Step 4:-

$R = 2^r$, R → distinct element
r → maximum number of zeros.

$R = 2^4 = 16$ approximately

So stream contain 16 distinct elements.