Th7: "Online" Algorithms (Data Streams): Ideas and code

In computer science, streaming algorithms are algorithms for processing data streams in which the input is presented as a sequence of items and can be examined in only a few passes, typically just one. These algorithms are designed to operate with limited memory, generally logarithmic in the size of the stream and/or in the maximum value in the stream, and may also have limited processing time per item.

Data stream algorithms process a continuous stream of data with only a limited possibility to store past records. Online machine learning covers methods that update their models after observing a new event and can immediately serve predictions based on the updated model.

Some basic concepts of data streams algorithms are the following:

1. Streaming Model:
   - In the streaming model, data arrives one item at a time, and the algorithm processes each item immediately without storing the entire dataset. The goal is to use limited memory to perform computations or derive summaries that approximate certain properties of the data.

2. Sketches and Summaries:
   - One approach to dealing with data streams is to use sketches or summaries. These are compact representations of the data that can be updated as new elements arrive. Examples include Count-Min Sketch for approximate frequency counting and HyperLogLog for distinct element counting.

3. Sampling:
   - Rather than processing every item in the stream, algorithms may use random sampling to select a subset of items for analysis. This can be useful for approximating aggregates like averages, variances, or quantiles.

4. Approximate Queries:
   - Many data stream algorithms provide approximate answers to queries due to the limited memory and the inability to store all data. These algorithms trade accuracy for efficiency and scalability.

5. Sliding Windows:
   - Data stream algorithms often operate on sliding windows, where only the most recent data is considered. This helps in dealing with evolving data over time while still maintaining a limited memory footprint.

6. Frequency Estimation:
   - Algorithms for estimating the frequency of items in a data stream are common. These algorithms aim to provide a good estimate of the frequency of each item without storing all occurrences.

7. Outlier Detection:
   - Identifying outliers or anomalies in the stream is another important application. Algorithms may use statistical techniques or heuristics to identify unusual patterns or deviations from the norm.

8. Join and Aggregation:
   - Some data stream algorithms deal with joining streams or aggregating values based on certain keys. These algorithms often need to provide approximate results due to the limited memory available.

These techniques play a crucial role in handling large-scale, high-velocity data streams in real-time applications.

In the following section I will show the pseudo-code necessary to realize the “Counting elements in a stream” algorithm and “Calculating moving average” Algorithm discussed in class.

-Counting elements in a stream pseudo-code:

The idea is to determine the frequency of occurrences of different elements in a continuous flow of data, often without storing the entire dataset due to memory constraints. This problem is common in various applications such as network monitoring, sensor data analysis, and online analytics. The goal is to provide approximate counts or summaries of the data in real-time.
Many counting algorithms provide error bounds or guarantees on the accuracy of their estimates. Users can control the level of accuracy by adjusting parameters or choosing different algorithms based on their specific needs.

```
Initialize an empty hash table (dictionary) to store element counts.

for each element in the data stream:
  if element is not in the hash table:
    add element to the hash table with count 1
  else:
    increment the count of the element in the hash table

end for

for each unique element in the hash table:
  print(element, count)
```

-Calculating moving average pseudo-code:

Calculating a moving average is a technique used to smooth out fluctuations in time-series data by creating an average value that "moves" through the data as new observations become available. The moving average is particularly useful for identifying trends or patterns in the data while reducing the impact of short-term fluctuations or noise.
The primary purpose of a moving average is to provide a smoothed representation of the data by reducing the impact of short-term fluctuations or noise. This is achieved by replacing each data point with an average of neighboring points within a specified window.

```
Initialize variables:
  sum = 0      // Running sum of elements in the window
  window_size = k  // Size of the moving window
  queue = empty queue or list to store the last k elements

for each element in the data stream:
  // Add the new element to the queue
  queue.enqueue(element)

  // Update the running sum with the new element and remove the oldest element if the window is full
  sum = sum + element
  if queue.size() > window_size:
    oldest_element = queue.dequeue()
    sum = sum - oldest_element

  // Calculate and print the moving average
  moving_average = sum / min(queue.size(), window_size)
  print(moving_average)

end for
```

Bibliography

-https://en.wikipedia.org/wiki/Streaming_algorithm
https://link.springer.com/referenceworkentry/10.1007/978-3-319-77525-8_326