1. Run-length encoding (RLE)

Description

Run-length encoding is a lossless data compression technique which stores streams of data as one value and count and reduces size of the original data. This technique is simple and useful for data aggregation as it can represent many repetitive values that are not necessary in smaller form.

Specification

From buffer, store each sequence of repetitive data into a shared value. Repeat until the original data is summarized. Print the result out at extra features section.

Implementation

Two functions were created for run-length encoding. First function is to determine the size of output RLE data. Second one is to perform the RLE aggregation by summarizing common values consecutively. The second function depends on first function because size is required in advanced in order to allocate memory for dynamic RLE data in main function. Both functions share core logic by continuously comparing two neighbor values and counting/storing only if they are different. Said logic is as demonstrated below:

```
SizeRLE = getSizeRLE(B);
RLE = (float *)malloc(SizeRLE * sizeof(float));
RLE = getRLE(B, RLE);

printRLE(RLE, SizeRLE);
```

```
int getSizeRLE(float *B)
{
  int i, SizeRLE = 0, prev = 0;

  for (i = 0; i < BUFFER_SIZE; i++)
  {
    if (B[i] != prev)
    {
      prev = B[i];
      SizeRLE++;
    }
  }

  return SizeRLE;
}
```

```
float *getRLE(float *B, float *RLE)
{
  int i, rle_i = 0, prev = 0;

  // Continuosly store new value that
  for (i = 0; i < BUFFER_SIZE; i++)
  {
    if (B[i] != prev)
    {
      prev = B[i];
      RLE[rle_i] = prev;
      rle_i++;
    }
  }

  return RLE;
}
```

2. Symbolic Aggregate ApproXimation (SAX)

Description

SAX by Jessica Lin is an algorithm that represents time-series data in alphabet format. It reduces storage/computational cost and can be simply and flexibly used for many purposes such as pattern matching and outlier detection.

Specification

Normalize and reduce the buffer data into piecewise aggregate approximation (PAA) data of an appropriate segments based on standard deviation. When PAA data is obtained, assign each value to a letter by using breakpoints of.6 alphabets. Print the result out at extra features section.

Implementation

Two most important functions were functions to get the PAA and SAX. First, the number of segments is selected according to suitable standard deviation threshold. Then, use this number of segments to allocate memory for SAX PAA values and SAX values. Calling function to get PAA involves data normalizing and segmenting, and the function to get SAX will assign the right letter of each PAA value according to alphabet breakpoints. Below is the core logic representing this process:

```
SaxW = getSaxW(StdDev);

SAX_PAA = (float *)malloc(SaxW * sizeof(float));
SAX = (char *)malloc(SaxW * sizeof(char));

SAX_PAA = getSaxPAA(B, SAX_PAA, SaxW, StdDev);
SAX = getSAX(SAX_PAA, SAX, SaxW);

printSaxPAA(SAX_PAA, SaxW);
printSAX(SAX, SaxW);
```

****Print outputs of both features are available in screenshots.**