Mandatory Assignment 2: Distinct elements using hashing

November 5, 2019

1 Exercise 1

We started this exercise with initializing the array A of 32-bit integers with values from Appendix A of the exercise description. Then we calculated the result integer bit by bit using a for loop, taking advantage of the bit-wise operations ("&") on the respective element of array A (A[i]) and the integer(z) that has been passed for the method as a parameter (both of them are 32-bit integers). This resulted in another 32-bit integer, of which we counted the number of 1s using Integer.bitCount() method before applying modulo 2 on it. If this resulted in a 1, we used the compound bit-wise operation ("|=") on the result integer and the new bit and moved it to the right position (for that we used the variable pos which gave us by how many positions we should move the bit from the rightmost position towards the left - this lets us define the bits of the result from left to right).

```
1  public int calculateHashValue(int z) {
2      int x = 0;  // result
3      int pos = 31; // defines how much we need to shif the bit towards the left
4
5      for (int i = 0; i < 32; i++) {
6          if ((Integer.bitCount(z & A[i]) % 2) == 1) {
7              result |= ((Integer.bitCount(x & A[i]) % 2) << pos);
8          }
9          pos--;
10     }
11     return x;
12 }
```

Listing 1: Calculation of the hash value

2 Exercise 2

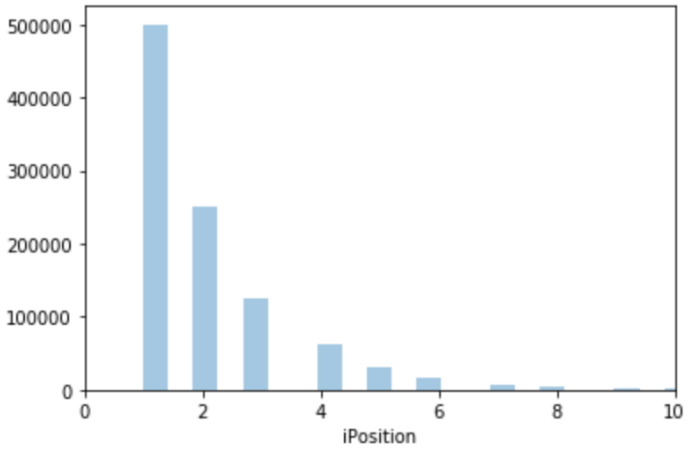
The value returned by the function getRho(x) represents the position of the first 1 in the binary representation of the number x, where x is the hash value of an integer z (x = h(z)). The calculation was done with the aid of the logarithmic function to base 2, since the binary numbers base on the

base-2 numeral system. The function returns the exponent i of 2^i. Do we have for example the number 178, the logarithmic function gives the result 7.47... Converting the float to an integer results in the position of the first 1 in the binary representation by giving the exponent of the next lower 2^i presentation to the number. Since we read from left to right, we have subtracted the result of our calculation from k, our bit-size.

```
1 public int getRho(int x) {
2     int i = 31;
3     if (x >= 0) i = (int) (Math.log(x) / Math.log(2));
4     return k-i;
5 }
```

Listing 2: Calculation of Rho

Plotting the distribution of getRho(x) for one million hash values from the range of 1 to 1,000,000 gives us the following result:



The plot above shows the distribution of positions of the first 1 in the hash values read from the left. The values of the x axis go from 0 to 10, which range comprises most of the values. The distribution satisfies the relation Pr(ρ(y) = i) = 2^{-i}. (As it can be easily seen, half of the hash values have the first 1 in position 1 (when ρ(y) = 1 then Pr(1) = 2^{-1} = 0.5) and quarter of the hash values have the first 1 in position 2 (when ρ(y) = 2 then Pr(2) = 2^{-2} = 0.25) and so forth).

3 Exercise 3

Our HyperLogLogCounter was implemented according to the pseudocode from the exercise with the parameters m=1024 and k=32. Therefore, we initialized an array M of size m with the default values of 0. Afterwards we read the first number of our input and stored it in a local variable named threshold. We made use of this value when we compared the output of our program to it, and printing above or below respectively. For every following number the hash value (see implementation in Exercise 1), the position of the first 1 in the hash-value and the f-value (according

to the given formula) are calculated. The computed values are then stored in the array M within the context M[f-value] = max(M[f-value], getRho(hash value)).
To derive to the number of distinct elements, the other given calculations were implemented in the following way:

```
1  public int getV() {return noOfZeros;}
2
3  public double getE(double Z) {
4         return (m*m*Z* 0.7213/(1 + 1.079/m));}
5
6  private double getZ() {
7         double denominator = 0.0;
8         double Z = 0.0;
9         for (int i = 0; i < m; i++) {
10            denominator += Math.pow(2, (-M[i]));}
11        if (denominator != 0) Z = 1 /denominator;
12        return Z;}
13
14 public double hyperLogLogCounter(){
15        double Z = getZ();
16        double V = (double) getV();;
17        double E = getE(Z);
18        if (E < 2.5*m && V > 0) {
19            E = m * Math.log(m / V);
20        }
21        return E;}
```

Listing 3: Calculations for the distinct element count

The number of zeros (method getV()) is calculated by a class variable, called noOfZeros, which is initialized to m and changed with every replacement of 0 to non-zero element in array M. Furthermore, we used doubles to store the computed values, because the harmonic mean(Z) and also the calculation of E, using the natural logarithm, require double precision to end up with the right result.
Our solution passes all the test on CodeJudge (#1099504). Testing our algorithm on the input sequence 10^6,...,2*10^6 -1 with 1 million distinct elements gives a result of 973089.2722159306 distinct elements.

4 Exercise 4

To avoid confusion, the code for this exercise is stored in new classes. Our Input generator can be found in `InputGenerator4.java`. We instantiate it in the class `HyperLogLogCounter4.java` with n = 1000 and 1000 different random seeds. In summary, we have 1000 runs, for 3 different values of m: 1024, 2048, and 4096. Running this class resulted in the following histograms:

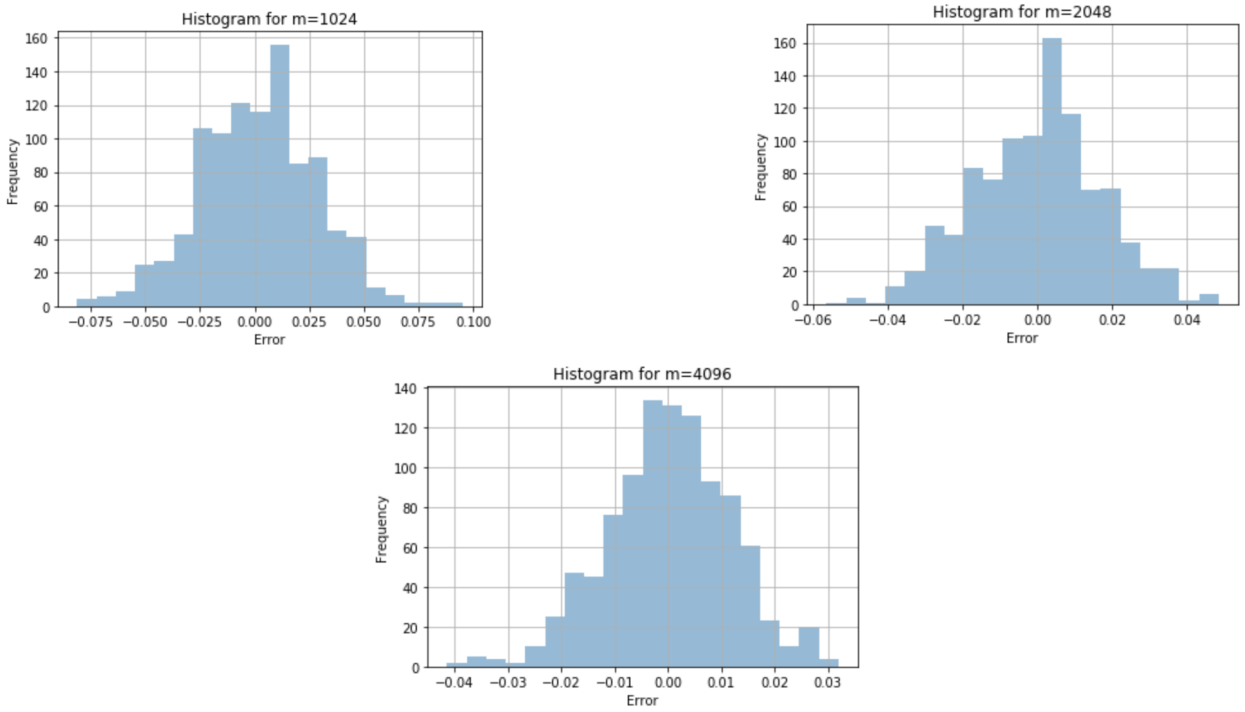


Figure 1: Three different histograms of error distribution

The y axis represents number of runs for each error value. We noticed that bins are normally distributed.
To calculate the fraction of runs that report a value in n(1 ± σ) and n(1 ± 2*σ), we wrote the following code in `HyperLogLogCounter4_2`:

```
1
2           double sigma =  0.022980970388562793; // change to 0.0325 for 1024,
                0.022980970388562793 for 2048, 0.01625 for...
3           double count = counter2.hyperLogLogCounter(); // change to counter, counter2 and
                counter3
4
5           if(count >(1000*(1-sigma)) && count < (1000*(1+sigma))){
6               withinSigma++;
7           }
8           if(count >(1000*(1 - 2*sigma)) && count < (1000*(1 + 2*sigma))){
9               withinTwoSigma++;
10          }
11          if(!(count >(1000*(1 - 2*sigma)) && count < (1000*(1 + 2*sigma)))){
12              notWithin++;
13          }
14      }
15      System.out.println("within one sigma: " + withinSigma + " and within two sigmas: " +
            withinTwoSigma + "and outside: " + notWithin);
16  }
```

Listing 4: Calculation of the fractions of runs

The results are following:

m	values in n(1 ± σ)	values in n(1 ± 2σ)	between σ and 2σ	outside	σ value
1024	799	982	183	18	0.0325
2048	818	992	174	8	0.022981
4096	954	1000	46	0	0.01625

For all m, most of the results are within one sigma distance and more than 90 percent is within two sigmas. The larger the m, the better the results - for m = 4096, 954 out of 1000 values were reported to be within one sigma and none outside the distance of two sigmas.