Hashing Project

arboefranck (fpk278), and schmidt-christensen (hqr618)

June 3, 2020

Introduction

The goal of this project is to implement hash-functions and use these in a hashing table utilizing chaining and count-sketch. We will analyse the running time of these hash-functions, with different operations on hash-tables. We will discuss the significance of different variables of the hash-tables such as; size of hash-table, and number of elements to be hashed. We will use C# to implement our hash-functions, hashing-tables, analysis, and experimentation.

Task 1 Implementing hash-functions

We have implemented the hash-functions multiply-shift hashing and multiply-mod-prime hashing. Multiply-mod-prime hashing is implemented in C# using. We understand the task as our a1 and a2 values are static.

```
public static ulong MultiShift(ulong x, int l)
  {
    return (a1 * x) >> (64 - l);
  }
```

```
public static ulong MultiModPrime(ulong x, int l)
  {
    BigInteger p = ((BigInteger)1 << 89) - (BigInteger)1;
    BigInteger y = (a2 * x) + b;
    y = (y & p) + (y >> 89);
    if (y >= p)
    {
      y -= p;
    }
    return (ulong)(y - ((y >> l) << l));
  }
```

We expect the calculation of Multishift to be faster than multimodprime, due to using smaller type values. Multishift's largest calculation is a multiply between a ulong a1 and ulong x. Multimodprime's largest calculation is multiply biginteger a2 and ulong x.

Task 2 Hashtable using Chaining

Our hashtable, in the class FixedSizeGenericHashTable, supports the operations; get(x), set(x), and increment(x,d). Our hashtable is inspired by the following solution on: https://stackoverflow.com/questions/625947/what-is-an-example-of-a-hashtable-implementation-in-c

Task 3 Square Sum

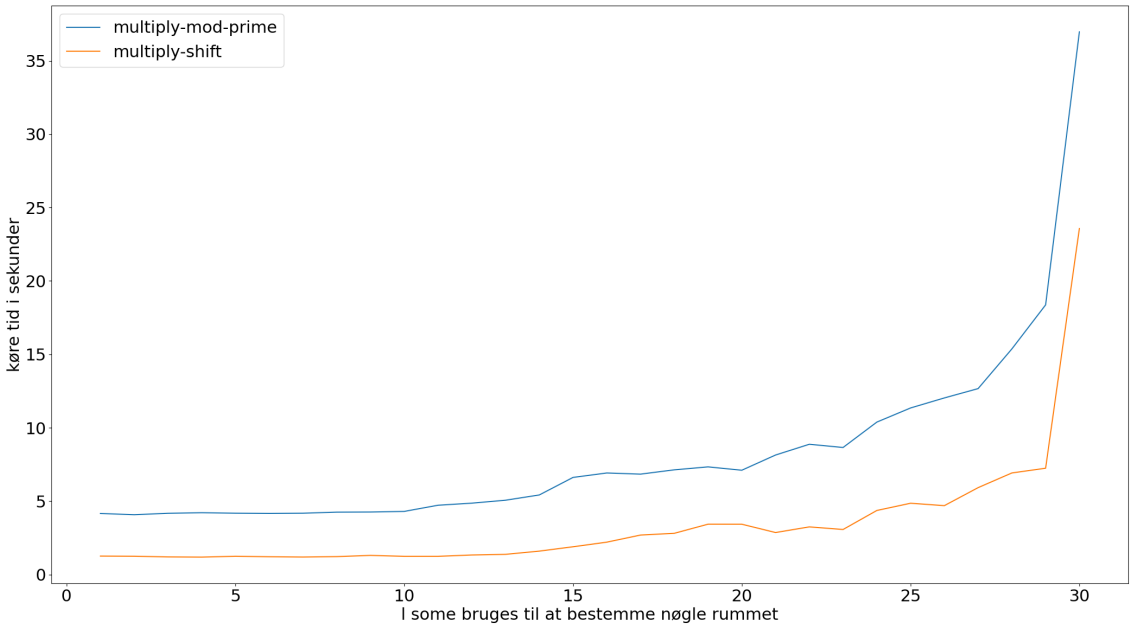

Figure 1: Here we can see the time of the two hash functions plotted next to each other. Discussed here

4-Universal Hash-function

Our 4-Universal Hash-function utilizes a four independent values $a_0 ... a_4$. From this we obtain four way independence, as each value of a is a different polynomial. We use the algorithms 1 and 2 in 2moment.pdf. The random a values are calculated using https://cswiki.cs.byu.edu/cs-312/randombigintegers

```
public static BigInteger FourUniversal(ulong x)
  {
    BigInteger p = ((BigInteger)1 << 89) - (BigInteger)1;
    BigInteger y = a_values[3];

    for (int i = 2; i >= 0; i--)
    {
      y = (y * x) + a_values[i];
      y = (y & p) + (y >> 89); //89 is b in the algorithm
    }

    return y >= p ? (y -= p) : y;
  }
```

Time of 4-universal hashfunction: 1000 ms
Time of MultiShift hashfunction: 35 ms
Time of MultiModPrime hashfunction: 541 ms

hash-functions for count sketch

We use the algorithm 2 from the notes to calculate the following.

```
public static Tuple<int, int> Hash4Count(BigInteger hashoutput, int m)
    {
      int h = (int)(hashoutput & (BigInteger)(m - 1));
      //89 is the power in p = 2**89 - 1
      // b is either 0 or 1, its the first bit
      int b = (int)(hashoutput >> (89 - 1));
      int s = 1 - (2 * b);
      return Tuple.Create(s, h);
    }
```

Implementing Count Sketch

We run the hash-functions for count sketch with 4-Universal Hash-function on a stream where we do the calculation:

$$SketchArr[h] = SketchArr[h] + (s * d)$$

on each element in the stream. Our Sketch function takes parameters: g, a hashfunction (such as 4-universal hashfunction); sAndH, our hashfunctions s and h; stream, a stream consisting of tuples x and d; m, the size of our countsketch array (this is an int 64 and is a power of 2).

```
public static int[] Sketch(Func<ulong, BigInteger> g, Func<BigInteger, int, Tuple<int, int>>
    sAndH, IEnumerable<Tuple<ulong, int>> stream, int m)
  {
    int[] SketchArr = new int[m];
    foreach (Tuple<ulong, int> item in stream)
    {
      BigInteger x = g(item.Item1);
      int d = item.Item2;
      Tuple<int, int> res = sAndH(x, m);
      int s = res.Item1;
      int h = res.Item2;
      SketchArr[h] = SketchArr[h] + (s * d);
    }
    return SketchArr;
  }
```

Task 7

For Task 7 and 8 with the 400 experiments we n = 3000000 and l = 13, for the stream generator, and the same l for the hash-table used as a control group.

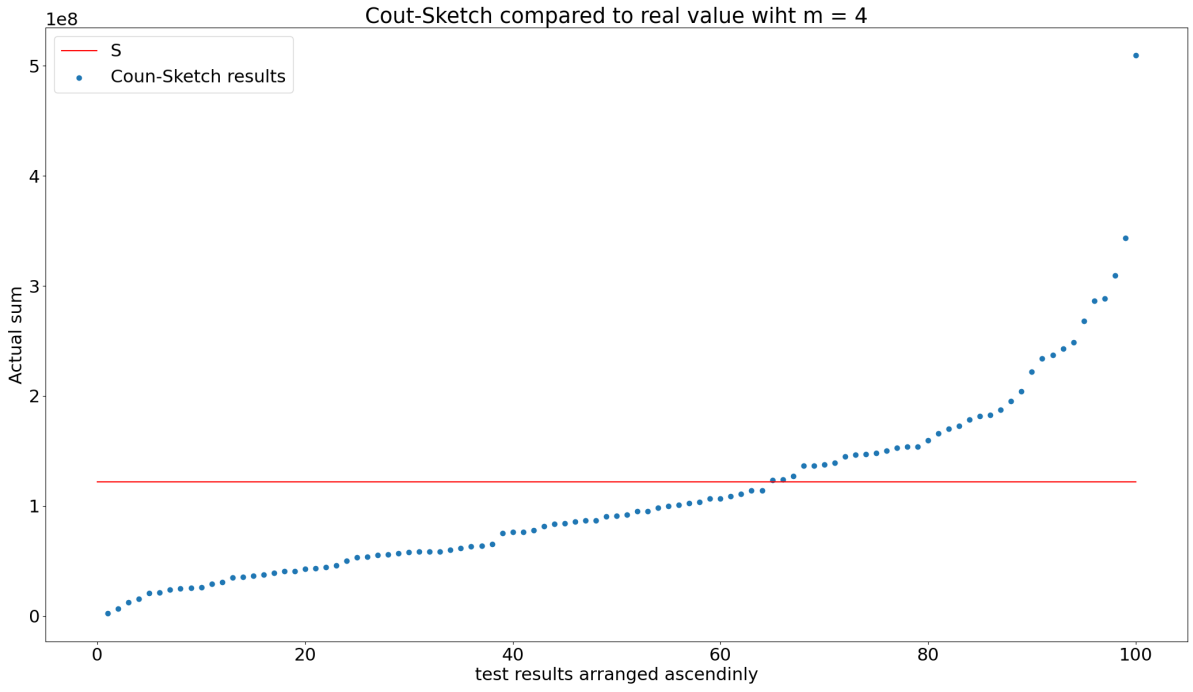

Figure 2: Plotting all 100 points and the S horizontal line, With m=4

The median is calculated from taking the median of the 9 subarray's medians. MSE is calculated from the 100 values of X and the S.

Var[X]:	$7.0454096 \cdot 10^{15}$
MSE:	$7.450787 \cdot 10^{15}$
Difference:	$0.4053774 \cdot 10^{15}$

Median:	90583150
S:	122072000
Difference:	31488850

Sketch Time:	3240 ms
Hash Time:	1342 ms
Difference:	Factor: 2.4

Discussed here

We calculate the median values of the 9 subarrays.

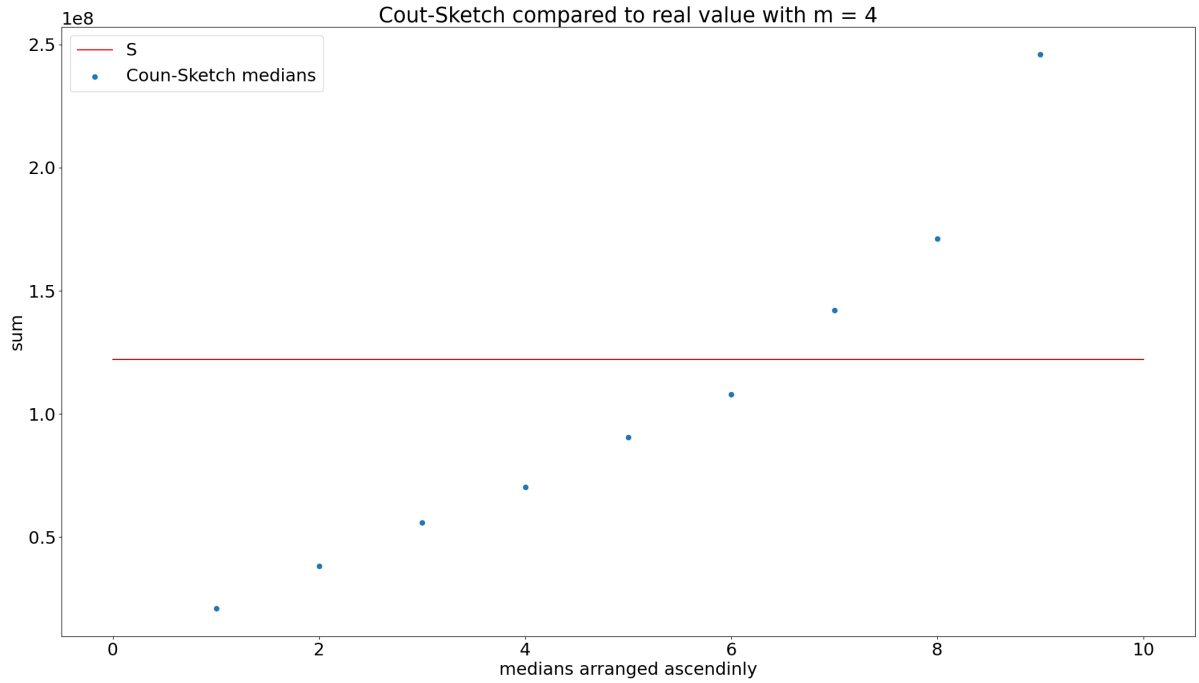

Figure 3: With m=4

We calculate the time for multishift hashfunction countsketch in this task so that we can compare the results to task 8. Our time results are based on comparison, between the time it takes to calculate the keys, insert into either our linked list, and calculate the squared sum. In hashtable using chaining we use multishift hashfunction. In our array with countsketch utilizing 4-universal hashfunction.

Task 8

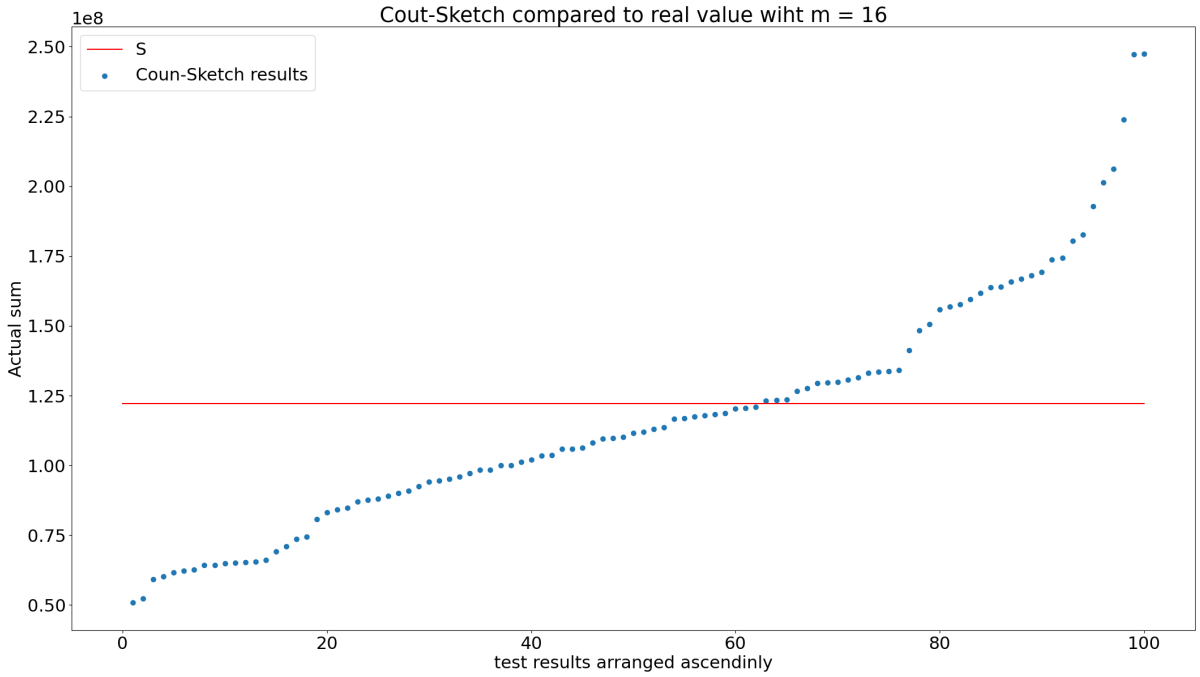

Figure 4: Plotting all 100 points and the S horizontal line, With m=16

Var[X]:	$1.8626967 \cdot 10^{15}$
MSE:	$1.7772083 \cdot 10^{15}$
Difference:	$0.854884 \cdot 10^{15}$

Median:	110911936
S:	122072000
Difference:	11160064

Sketch Time:	3216 ms
Hash Time:	1342 ms
Difference:	Factor: 2.4

Discussed here

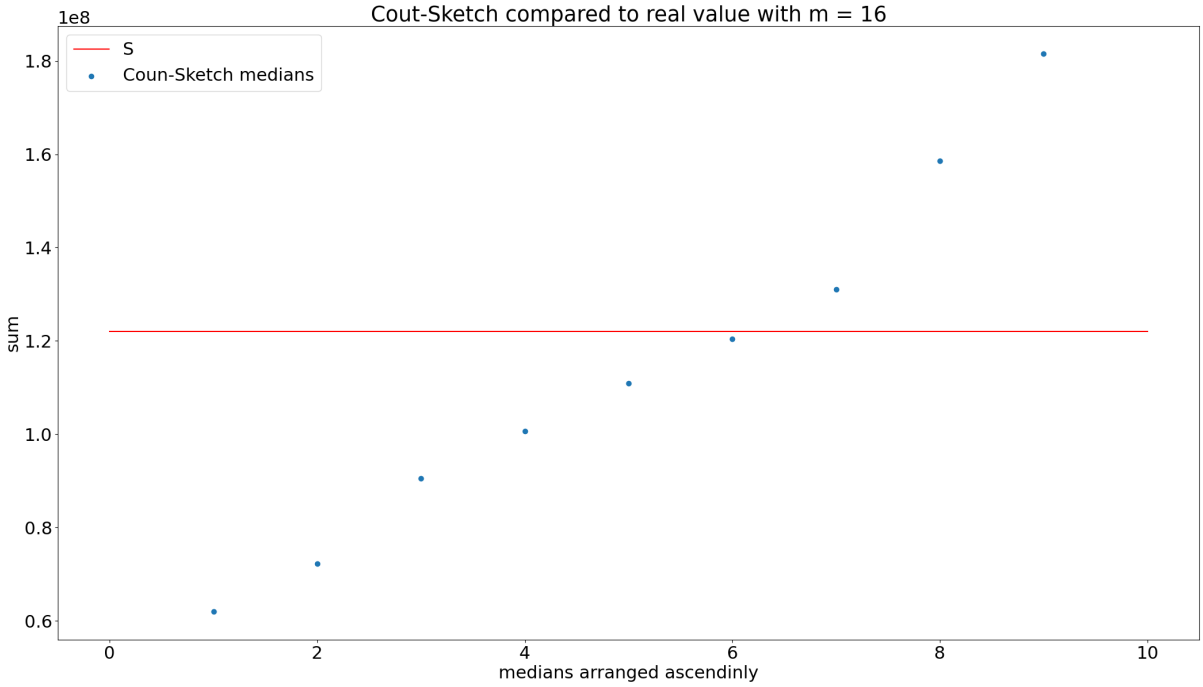

Figure 5: With m=16

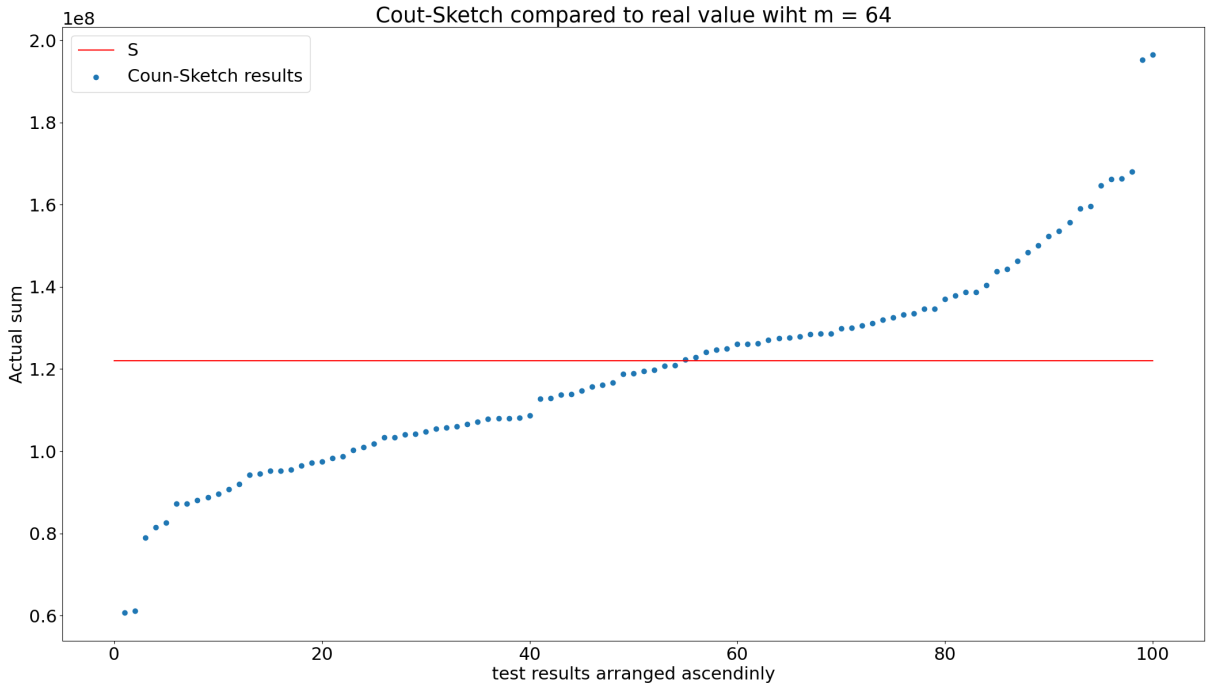

Figure 6: Plotting all 100 points and the S horizontal line, With m=64

Var[X]:	$4.6567418 \cdot 10^{14}$
MSE:	$6.289212 \cdot 10^{14}$
Difference:	$0.40278206 \cdot 10^{14}$

Median:	118853600
S:	122072000
Difference:	3218400

Sketch Time:	3129 ms
Hash Time:	1342 ms
Difference:	Factor: 2.3

Discussed here

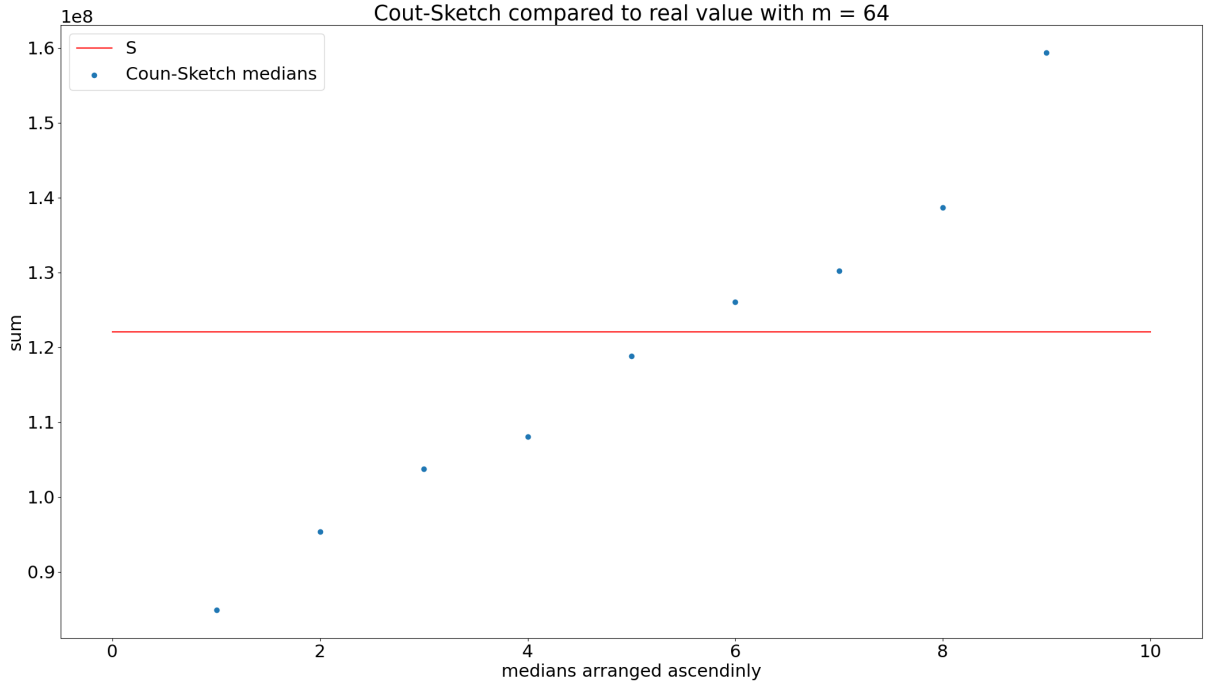

Figure 7: With m=64

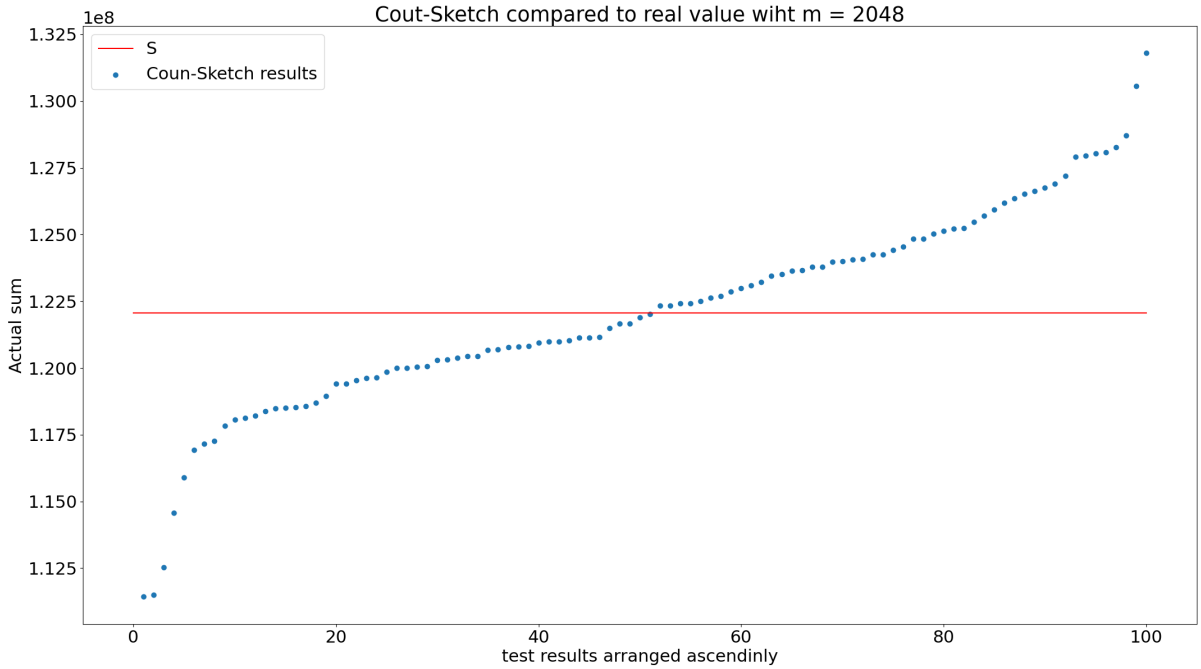

Figure 8: Plotting all 100 points and the S horizontal line, With m=2048

Var[X]:	$1.4552318 \cdot 10^{13}$
MSE:	$1.4054584 \cdot 10^{13}$
Difference:	$0.497734 \cdot 10^{13}$

Median:	121783850
S:	122072000
Difference :	288150

Sketch Time:	3125 ms
Hash Time:	1342 ms
Difference:	Factor: 2.3

Discussed here

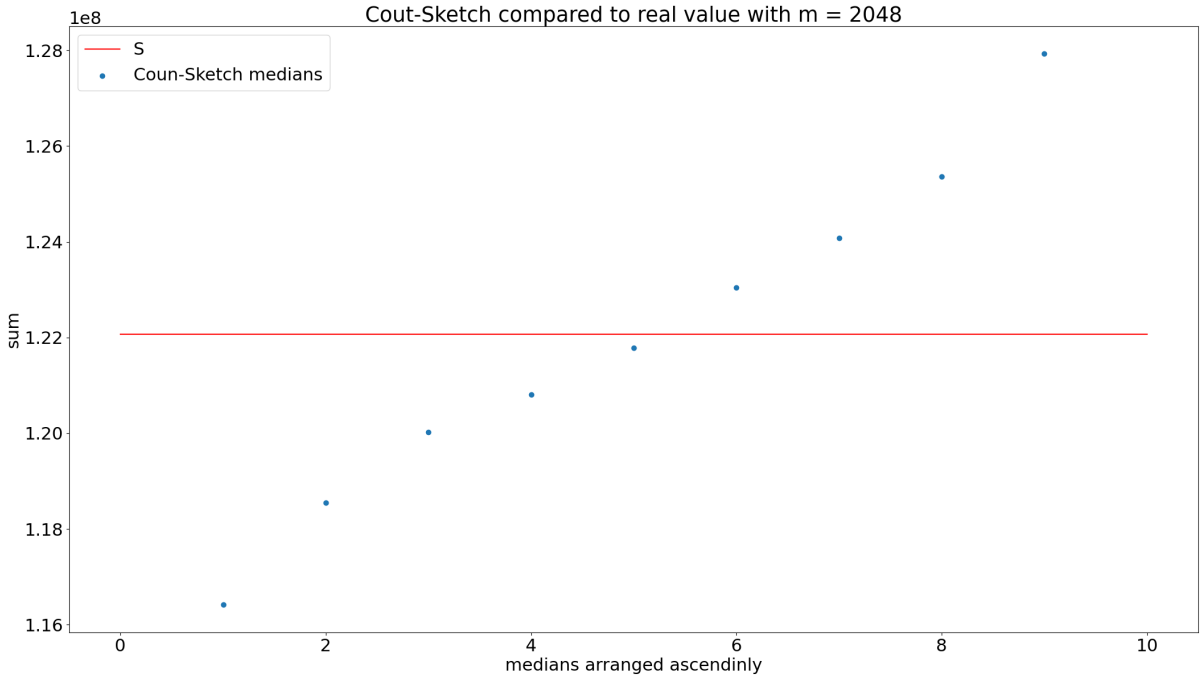

Figure 9: With m=2048

Discussion

1.c (with 4-universal)

Our comparison between MultiShift, MultiModPrime, and 4-universal, shows us that the 4-universal is slower. This follows our expectations, as MultiShift performs 3 simple operations on 64 bit numbers. Where MultiModPrime performs 9 - 10 operations, whereof most are on BigIntegers, so this should be slower. In the end 4-Universal performs 17-18 operations where all are on BigIntegers. When we perform Count-Sketch, we will further have to calculate h(x) and s, using our hashfunctions for Count-Sketch function. We will thus expect an even further increase in the time taken to calculate these values.

3

From our tests it seems that the two functions are different by a factor of ≈ 3, so they are big O of the same function. Which also makes sense since the only difference is the calculation of the hash value, where Multi mod does a few more operations and on larger numbers. So the extra time is not based on input size.

7 & 8

Looking at our experiments 4, 16, 64, 2048, it shows that Count-Sketch is slower by a factor of 2.3. Which is for each of the 100 values, so its average is way slower. Count-Sketch should be faster because it uses a smaller array, but because the 4 way universal hash function, and the hash function for Count-Sketch, takes so much more time than multiply shift, that time advantage is lost. You do however, still save space when using Count-Sketch, not only because the array in the hash table is larger, but specially because each index in the hash table is a linked list. Perhaps if the universe of keys were much larger than the array in the hash table, we would see Count-Sketch become faster. As this would cause the linked lists in the hash table to be longer, but since we have an array of the size of the universe of keys, collisions are not very common. We did do a few test for this, setting l for the hash table size down to 8 or 5, while keeping l for the stream as 13, this moved the factor in time difference closer to 2, but it was still slower.
We can see that the higher our m is the lower our difference between the sums will be, and the MSE is closer to the Var[X].