CMPT 224
April 3, 2019

Kai Sackville-Hii (301310336)
ksackvil@sfu.ca

Assignment 5

Hash Function Analysis

For our implementation design we have two hash tables, one for the institution and one for the students. The students hash table (StudentHashTable) has a larger max input (4,000,000 students) than the institution hash table (1000 institutions), more inputs means a better tested function so we will use the students hash table for this experiment. The students hash table has two indexing keys: "student name" (last name, first name) and "student number" (9-digit number). The institution member "students transferring to" stores the key "students name" ordered in ascending alphabetical sort by last name, and the member "students transferring from" stores the key "student number" in descending order. Each member can then access the students directly by using their subsequent index key. For this experiment we chose to use the "student number" index over a "student name" for multiple reasons. First, students are more likely to have the same name than the same student number, it can also be assumed that students in the same institution with the same name will have different student numbers making it less common to have matching student numbers in our experiment. Secondly, we assume that the student numbers all will have the same length with 9-digits, whereas we cannot predict the length of a student's name, so using the student numbers will give us a nicely formatted inputs for our experiment. We will test two hash functions, one simple and one optimized, with a scaled down model of our hash table where our max inputs goes from 4,000,000 to 500. Also, we will assume our hash table will have a load factor of 0.5, meaning that our max capacity will be $2*500 = 1000$.

Function 1:

Let's design a simple hash function for the given inputs. This hash function will be given a student numbers, it will separate this string representation of a number into its 9-character digits. Each character will be converted into its integer representation (ASCII value) and then the sum of all 9 characters will be taken. Then we will take the modulo of this sum to the max capacity of our hash table to figure out the output index. Figure 1 shows our hash function code. This function is incredibly simple, however as we will show it is not the optimal function for our problem. Testing this function with a capacity of 1000 and 500 input student numbers yields the results shown in table 1 below. These results show that there were only 47 unique indices produced by our hash function, and 43 of these had at least one collision so we only get 1 – to – 1 mapping in 4 indexes. The largest number of collisions produced (largest frequency) was 29, and the range of indices was 449 – 498 (49 index difference), both poor results. Since we are just taking the sum of each char, we can predict the theoretical range for our function. The input "000000000" would produce the smallest index, and the input "999999999" would produce the largest. '0' converted to an int is 48, taking the sum (48*9)%1000 gives the smallest possible index at 432. '9' converted to an int is 57, taking the sum (57*9)%1000 gives the largest possible index at

Kai Sackville-Hii (301310336)
ksackvil@sfu.ca

513. Therefore, it can be predicted that any input into our hash function will yield an index in the range of 432 – 513.

An example synonym from the tests are the student numbers “246938201” and “572831711”, both of which have the index 467. The large number of collisions produced by our hash function can be explained by the small output range of 432 – 513, with 500 inputs and only 81 possible unique values there is always going to be collisions. Figure 2 visualizes this limited range. It shows how all these indices get squished into the relatively small range causing many collisions. The Big-O efficiency of hash function 1 is O(1), making it simple to compute.

```
int hashFunc1(string inputKey)
{
    int p = 0;
    int sum = 0;

    sum = (int)inputKey[0] + (int)inputKey[1] + (int)inputKey[2] +(int)inputKey[3]+
(int)inputKey[4] +(int)inputKey[5] + (int)inputKey[6] + (int)inputKey[7] +
(int)inputKey[8];

    p = sum % MAX_INPUT_SIZE;

    return p;
}
```

Figure 1: Hash Function 1 Code.

Attribute	Value
Number of Unique Indexes	47
Index Range	449 - 498
Largest Frequency	29
Many-to-1 Maps	43

Table 1: Hash Function 1 Results.

Kai Sackville-Hii (301310336)
ksackvil@sfu.ca

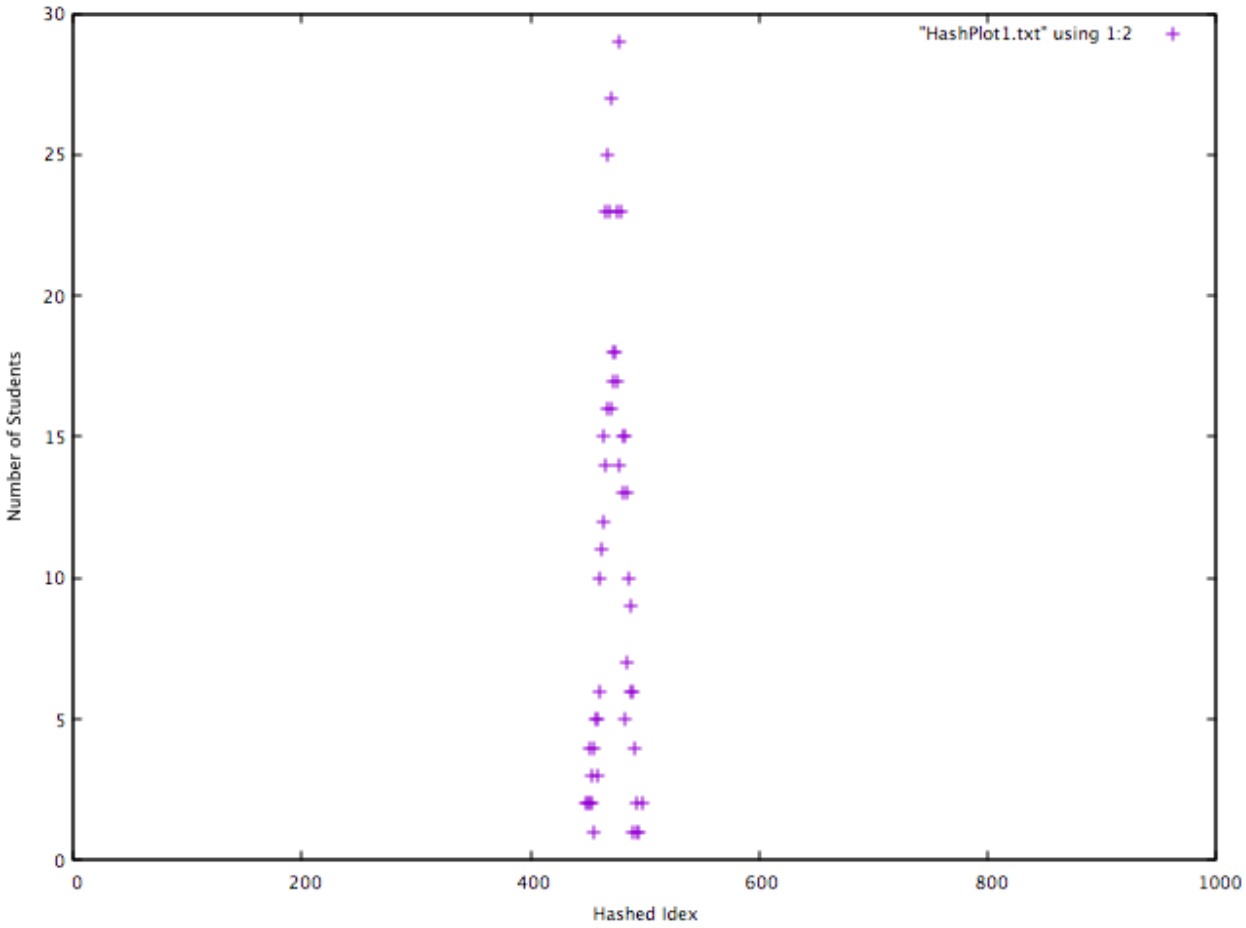

Figure 2: Hash Function 1 Output

Kai Sackville-Hii (301310336)
ksackvil@sfu.ca

Function 2:

To improve upon the first hash function, we must "spread" out the range of indices it produces. To do this we need to increase the sum produced before modulo arithmetic is done. Let's modify the first function so that each time we take the sum we multiple the previous sum by 9. The function then looks like the code from Figure 3. Note that we need to take the absolute value of this some because it is possible for it to be negative, and we do not have negative indices. This function was tested in the same way as before, these results are shown in table 2. Notice that the range has drastically increased and now covers entire hash table with 397 unique indexes, and that the largest frequency has decreased from 29 to 3. Indeed, we have successfully "spread" out the indexes, this can be confirmed visually by figure 4 in which we see the dots covering the entire range. This new function produces collisions on a much smaller scale than the first (3 verses 29), but more often than the first (89 verse 43). Since function 2 spreads out the index's, it is expected that the scale of the collisions will be reduced, and the frequency increased; this point can be visualized by comparing figure 2 to figure 4. An example synonym from the tests are the student numbers "501140764" and "106499968", both of which have the index 404. The Big-O efficiency of hash function 2 is O(n), making it okay to compute for small n's but not as efficient as hash function 1. However since we assume the input size is always going to be 9 (9-char student number) n will never be big enough to render hash function 2 inefficient.

```
int hashFunc2(string inputKey)
{
    int p = 0;
    int sum = 0;

    for (int i = 0; i < inputKey.size(); i++)
    {
        sum = (9 * sum) + (int)inputKey[i];
    }

    sum = abs(sum);
    p = sum % MAX_INPUT_SIZE;

    return p;
}
```

Figure 3: Hash Function 2 Code

Kai Sackville-Hii (301310336)
ksackvil@sfu.ca

Attribute	Value
Number of Unique Indexes	397
Index Range	0 - 999
Largest Frequency	3
Many-to-1 Maps	89

Table 2: Hash Function 2 Results.

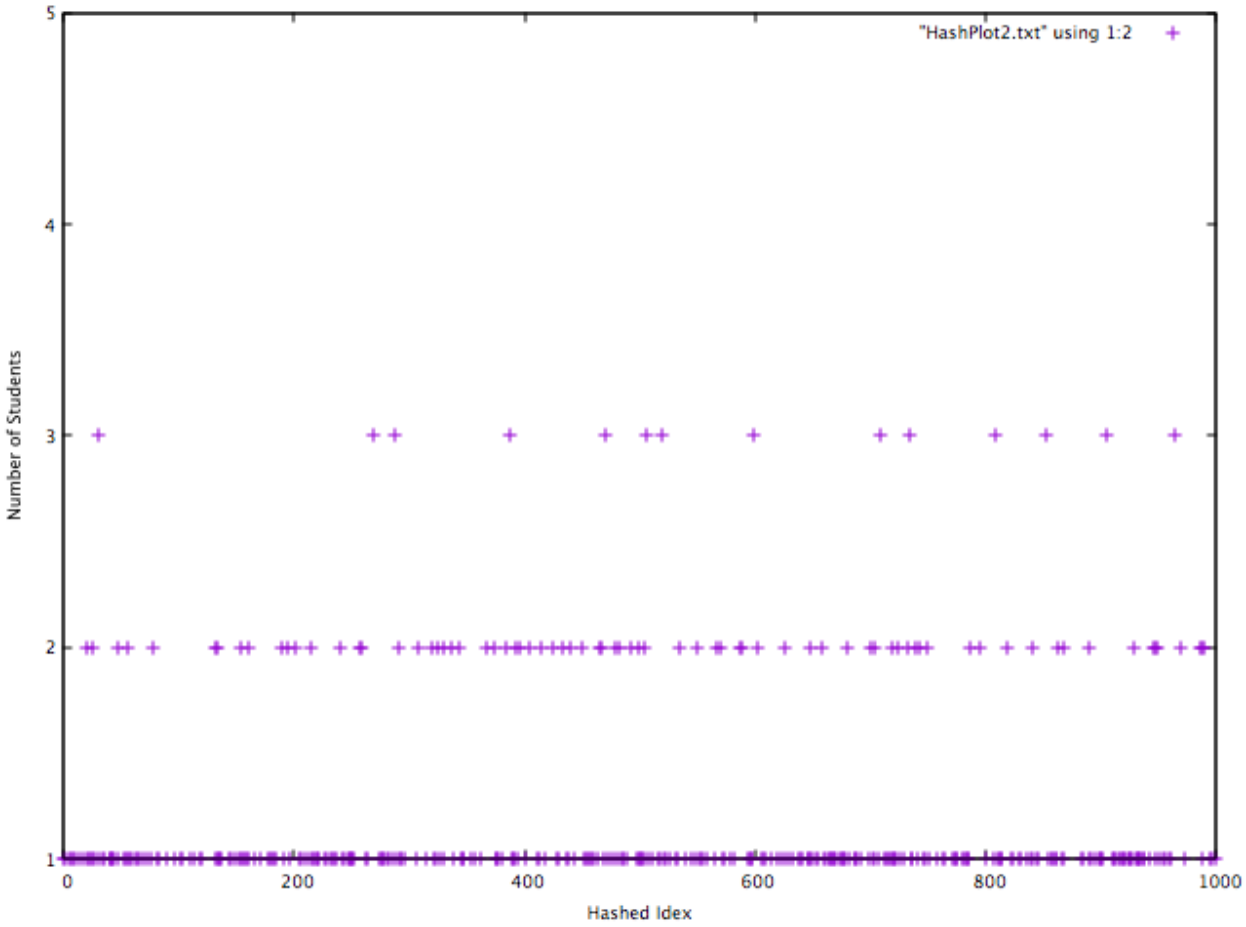

Figure 4: Hash Function 1 Output.