Hashing functions

Hash Table

basics

- "Map"
  - a data structure in which a key/value pair are stored
    - Store and access a value through its key
      - Let M be a map
      - M[k] = v
        - store into M the value at position "k" in the map
      - print M[k]
        - print the value stored at position k
  - Arrays are maps which a very restricted set of allowed keys
  - More generally, maps allow anything to be a key

Maps in action

Go

```go
package main

func main() {
    A:= make(map[string]string)
    A["David"]="Letterman"
    A["Lassie"]="Come Home"
    println(A["Lassie"])
}
```

Java

```java
public class H {
    public static void main(String[] args) {
        Map<String, String> A = new Map<>();
        A.put("David", "Leterman");
        A.put("Lassie", "Come Home");
        System.out.println(A.get("Lassie"));
    }
}
```

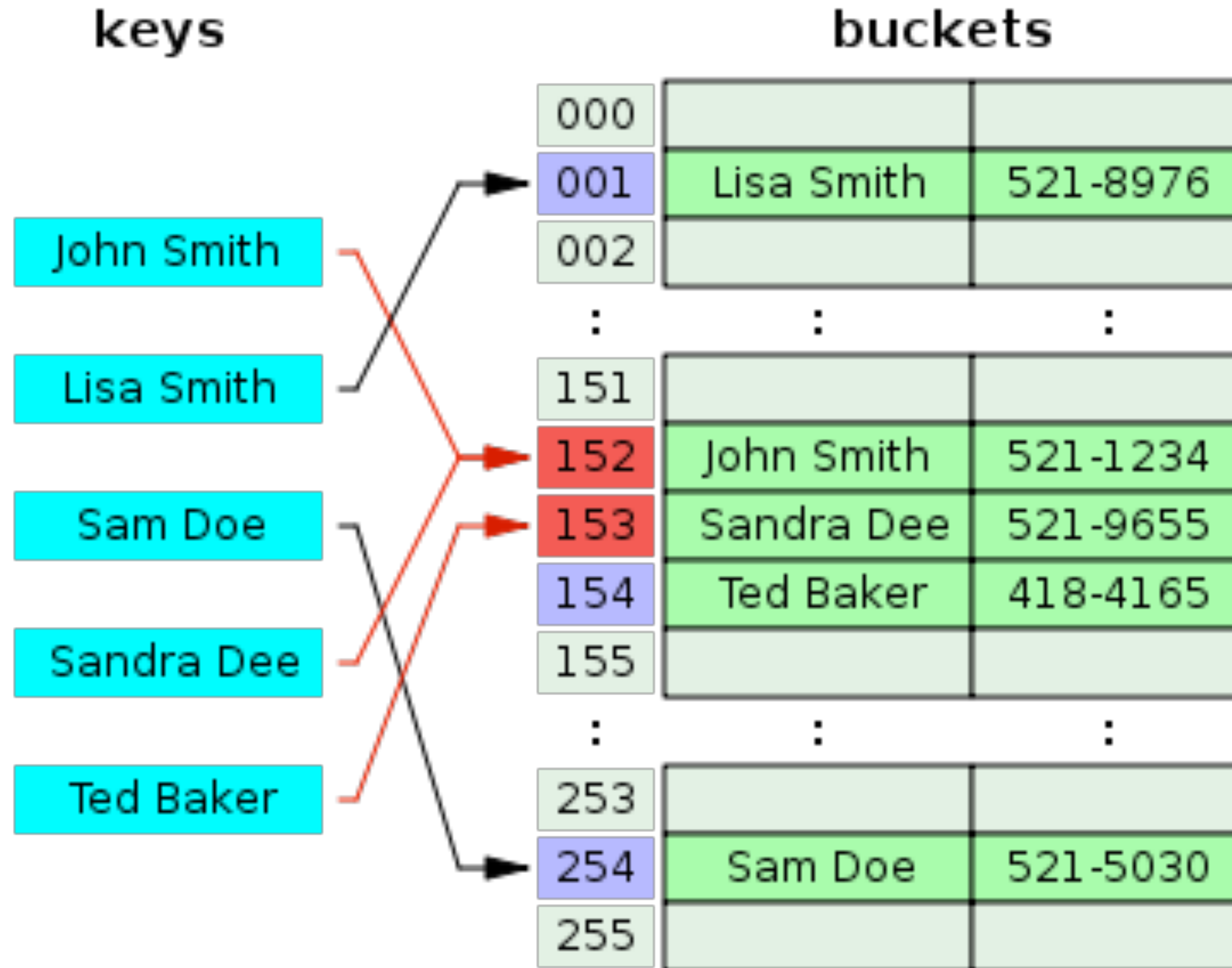
Will not compile, Map is an interface

Python

```python
A={}
A['Lassie']='Come Home'
A['David']='Letterman'

print(A['Lassie'])
```

Hashtables are Maps

they use a hash function to transform keys into integers
those integers are then used to place {Key,Vale} into an array



Time Complexity for Maps

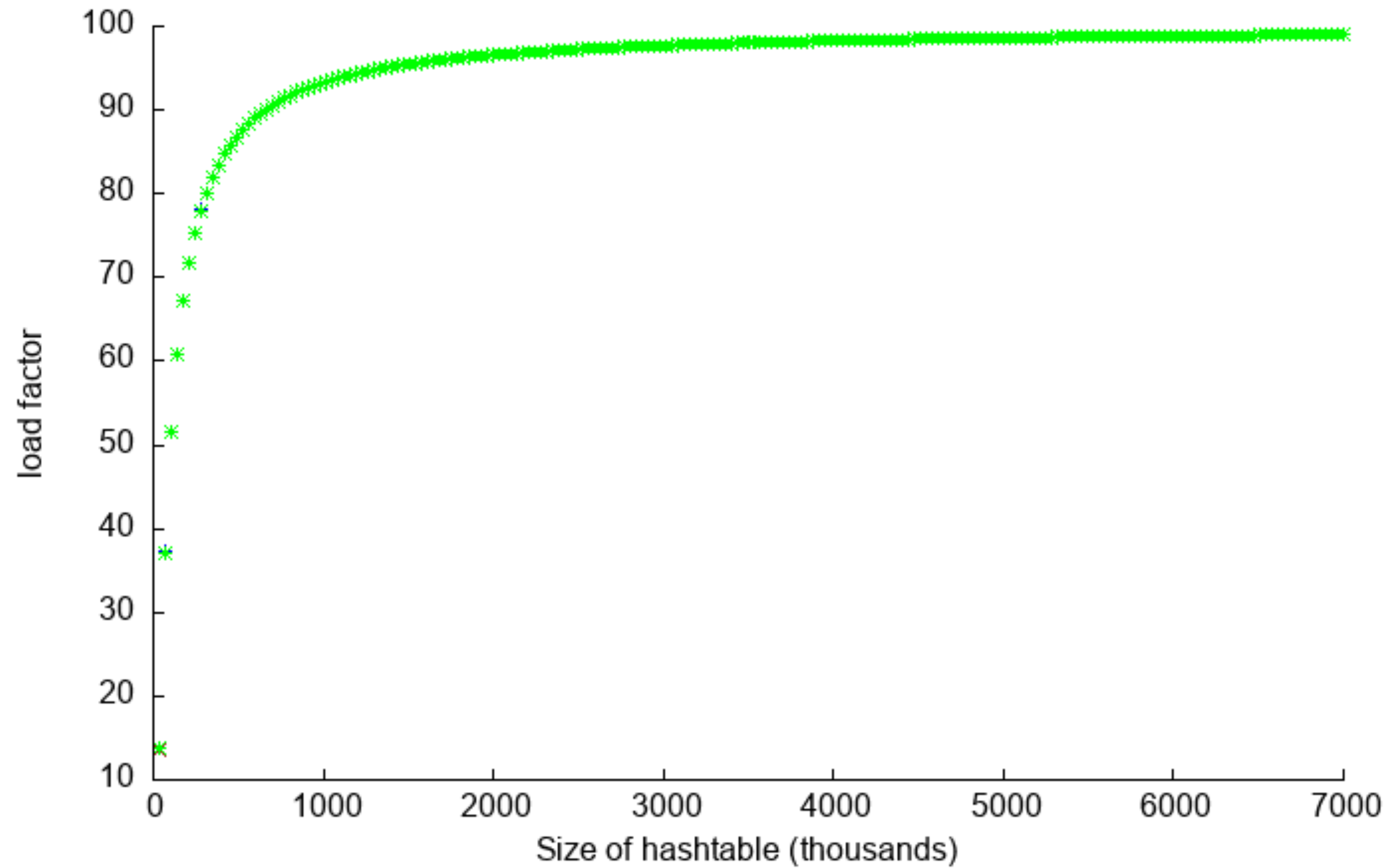
	Insert	Retrieve	Contains
built on unsorted list	O(1)	O(n)	O(n)
built on sorted list	O(n)	O(lg n)	O(lg n)
built on balanced tree	O(lg n)	O(lg n)	O(lg n)
Hashtable (usually)	O(1)	O(1)	O(1)

Why "usually" for hashtables

- Rehashing
- Load factor
  - percentage of unused spaces
  - (number of items in table) / (size of table)
- "Good" hashing function

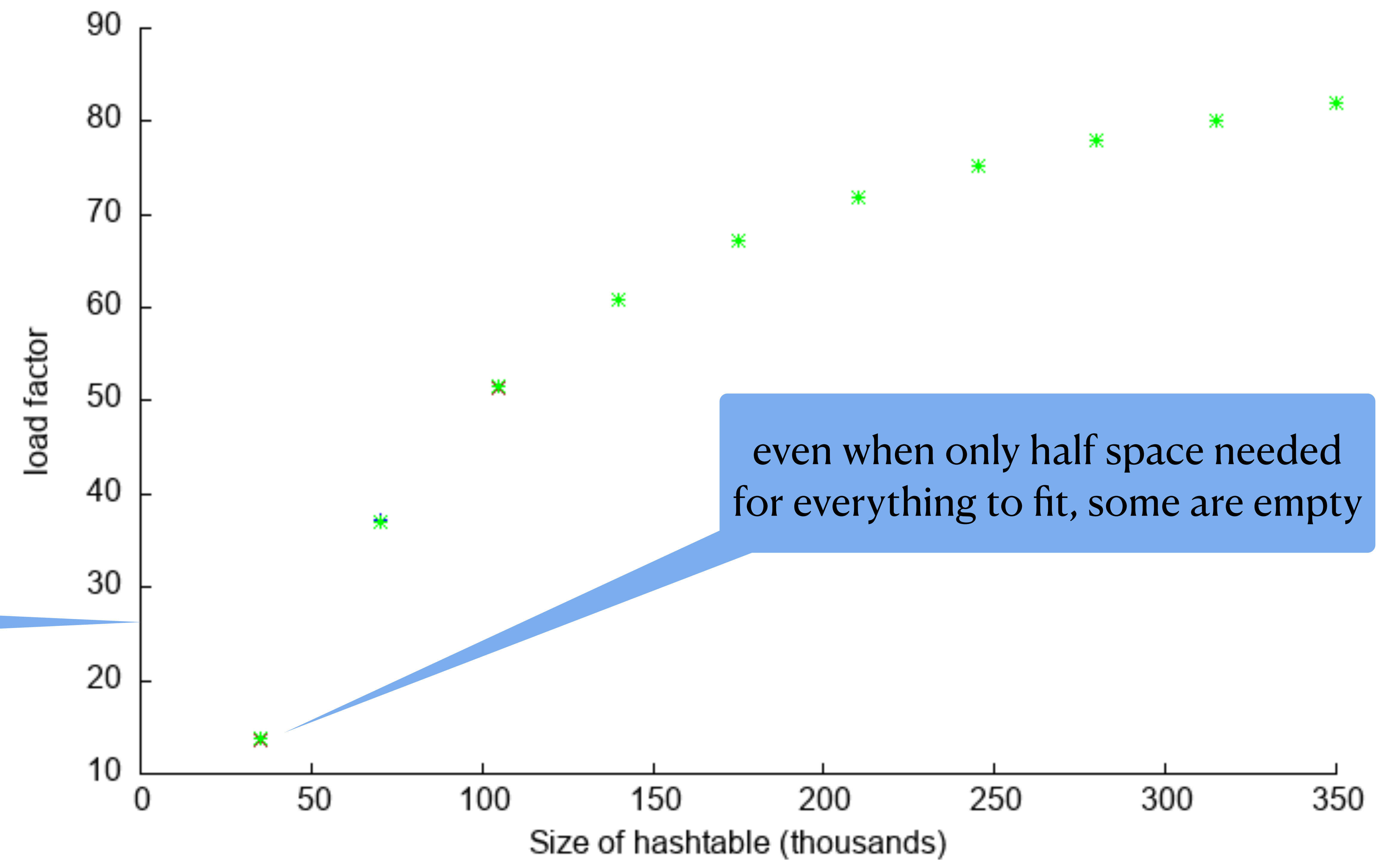
Load Factor

Percentage of unused spaces



3 different hashing functions in both plots

- Java Library Hasher
- Horners method (51)
- Horners Method (283)



even when only half space needed for everything to fit, some are empty

Same as previous, but only to 350,000

Hashing function

- Goal
  - Make keys a fixed size (int in Java is 4 bytes)
  - Uniformly distribute keys
    - minimize collisions
  - Keys in `0..(N−1)`

- Good
  - fast to compute
  - minimize duplications (ie collisions)

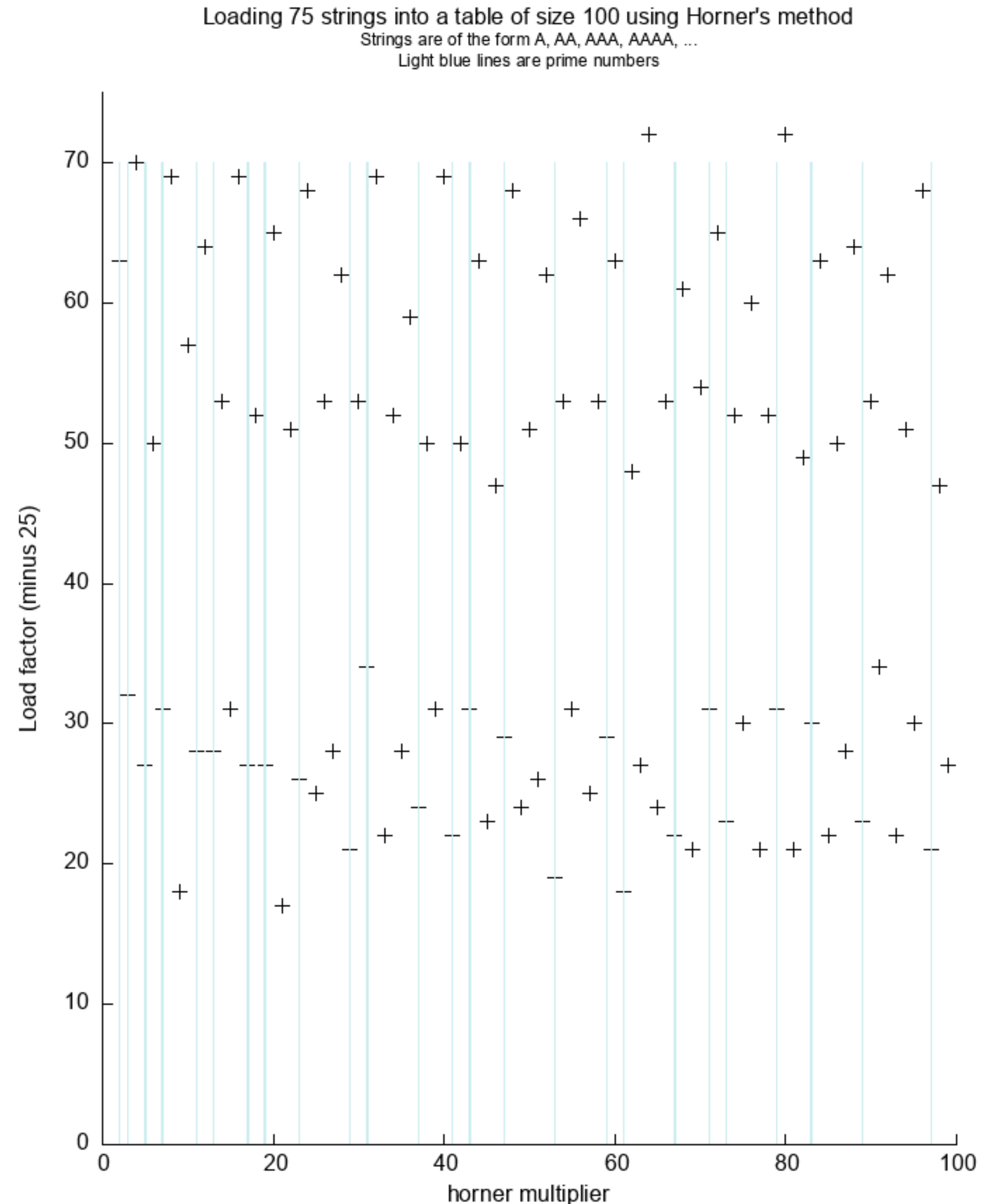
Horner's Method

```java
public int Horner(String ss) {
    int mul = 51;
    int ll = 0;
    for (int i = 0; i < ss.length(); i++) {
        ll *= mul;
        ll += ss.charAt(i);
    }
    return ll;
}
```

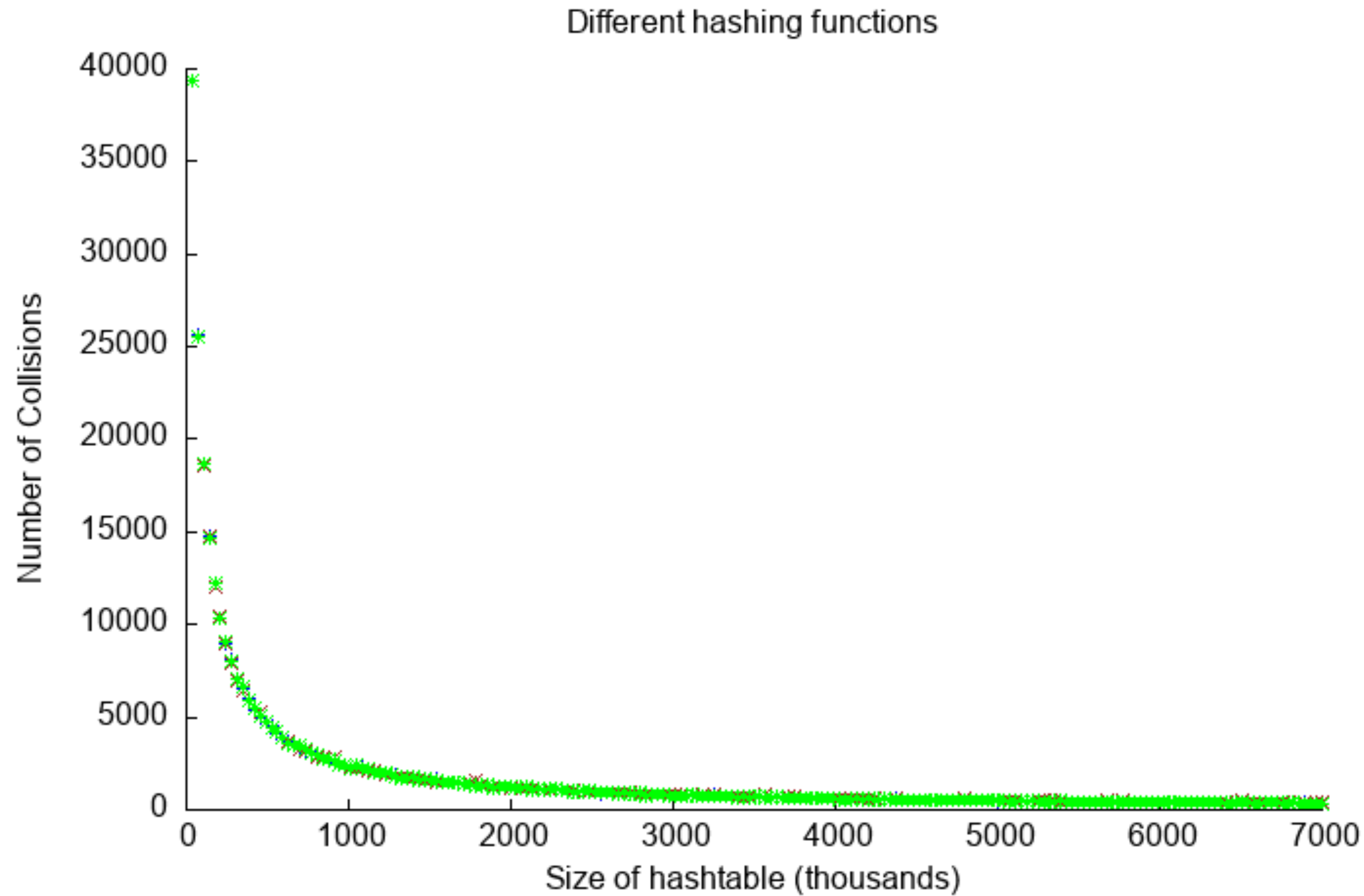
```java
public int Horner2(String ss) {
    BigInteger mul = BigInteger.valueOf(51);
    BigInteger ll = BigInteger.valueOf(0);
    for (int i = 0; i < ss.length(); i++) {
        ll = ll.multiply(mul);
        ll.add(BigInteger.valueOf(ss.charAt(i)));
    }
    return ll.intValue();
}
```

Horner Analysis

- Why not BigInteger version?
  - Speed!
    - H=914ms for 100,000,000 7 char strings
    - HBig=5437ms for 100,000,000 7 char strings
      - gets worse as strings get longer!
    - What is lost in H vs HBig?
- Why a largish prime?



Collisions



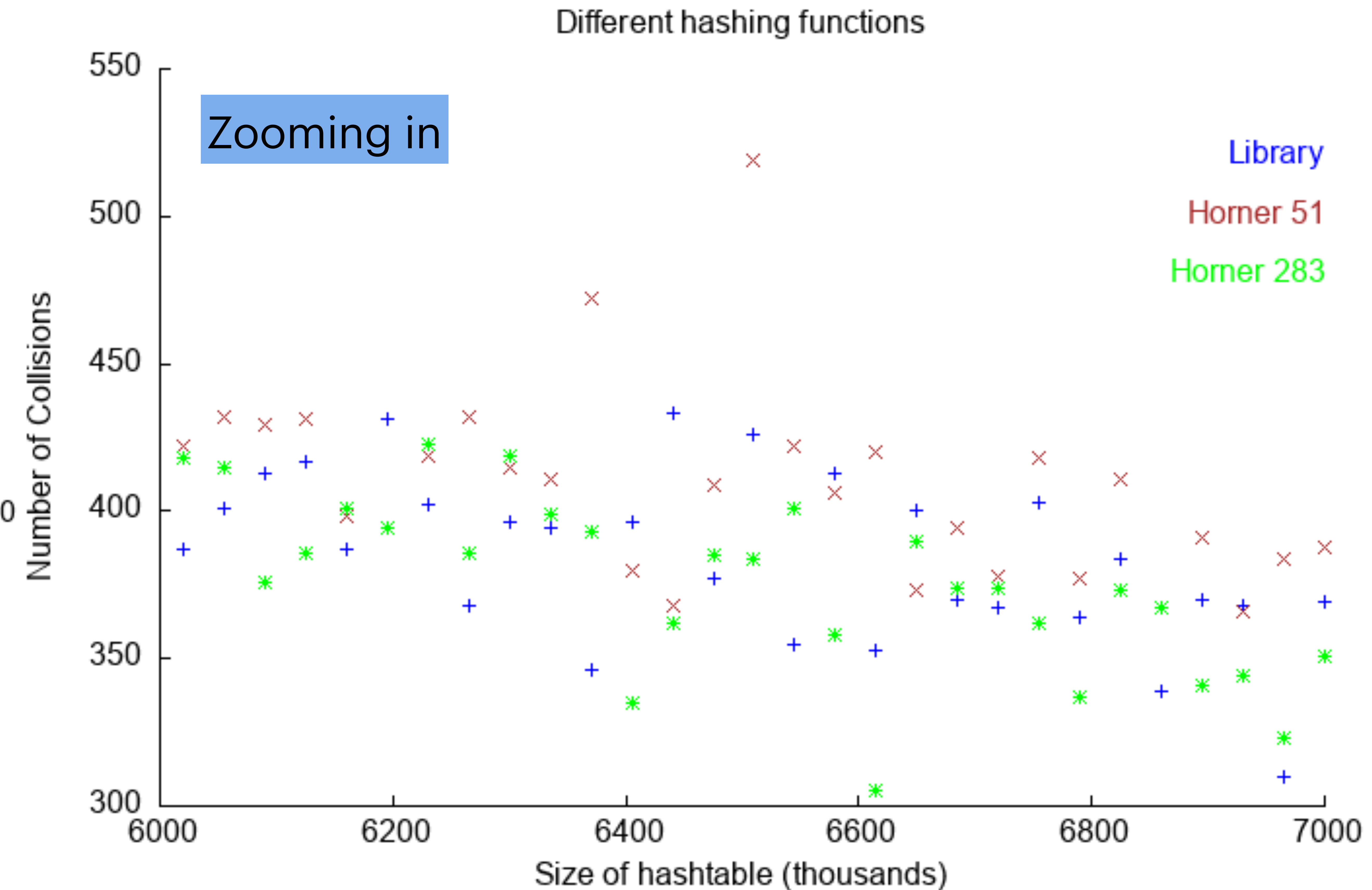
How do you count collisions?

if A,B,C all go to the same spot how many collisions are there?

3 or 2 or 3?

if A,B,C,D how many?

4 or 3 or 6?



More Hashing functions

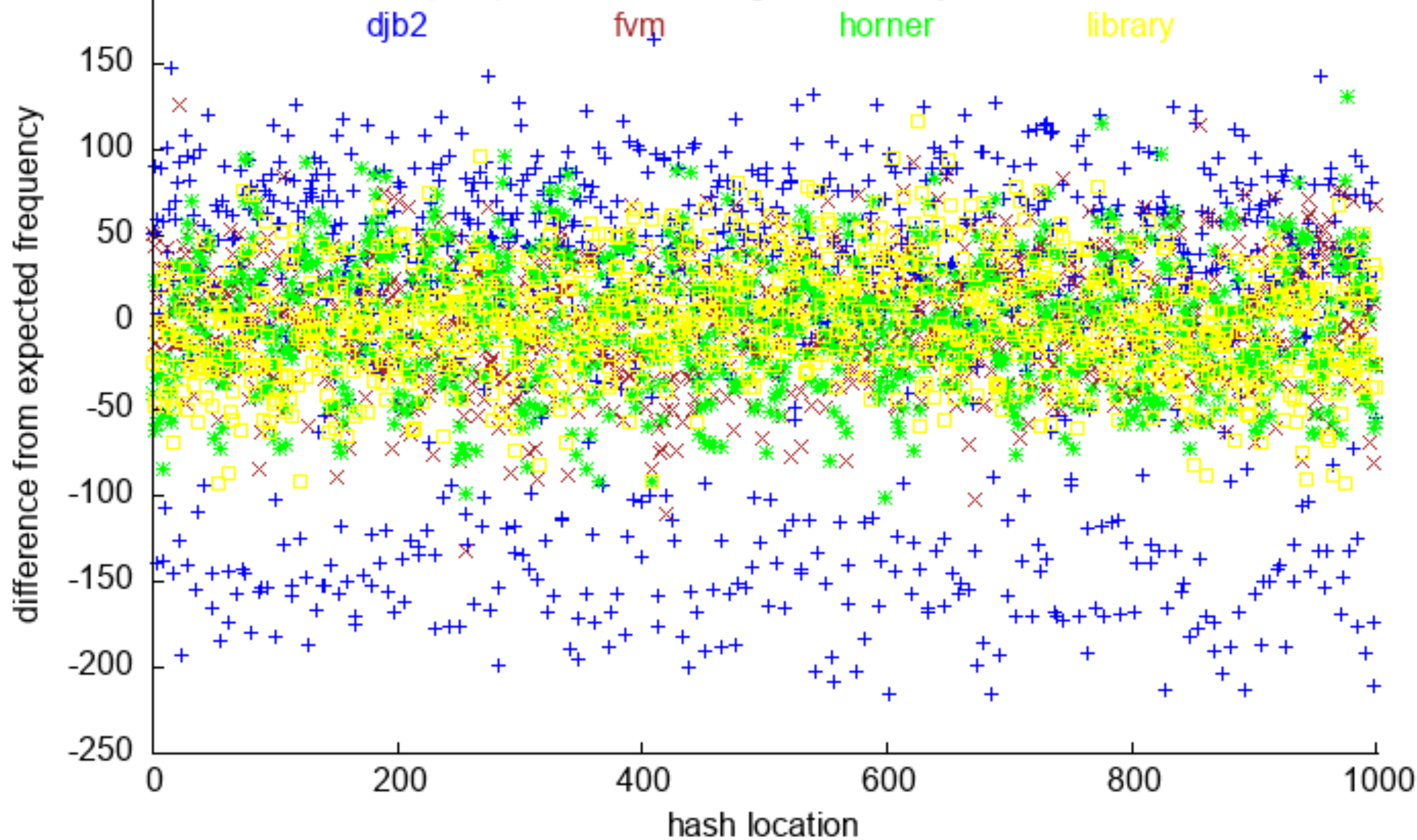
```
//Glenn Fowler, Landon Curt Noll and Kiem-Phong Vo
// 1991
private static final long uint64Offset = 0xcbf29ce484222325L;
private static final long uint64Prime = 0x00000100000001b3L;

public long fvnHash(String data) {
    long hash = uint64Offset;
    for (int i = 0; i < data.length(); i++) {
        hash ^= data.charAt(i);
        hash *= uint64Prime;
    }
    return hash;
}
```

"^=" is XOR followed by =
same as hash = hash ^ data.charAt(i)
(which is faster?)

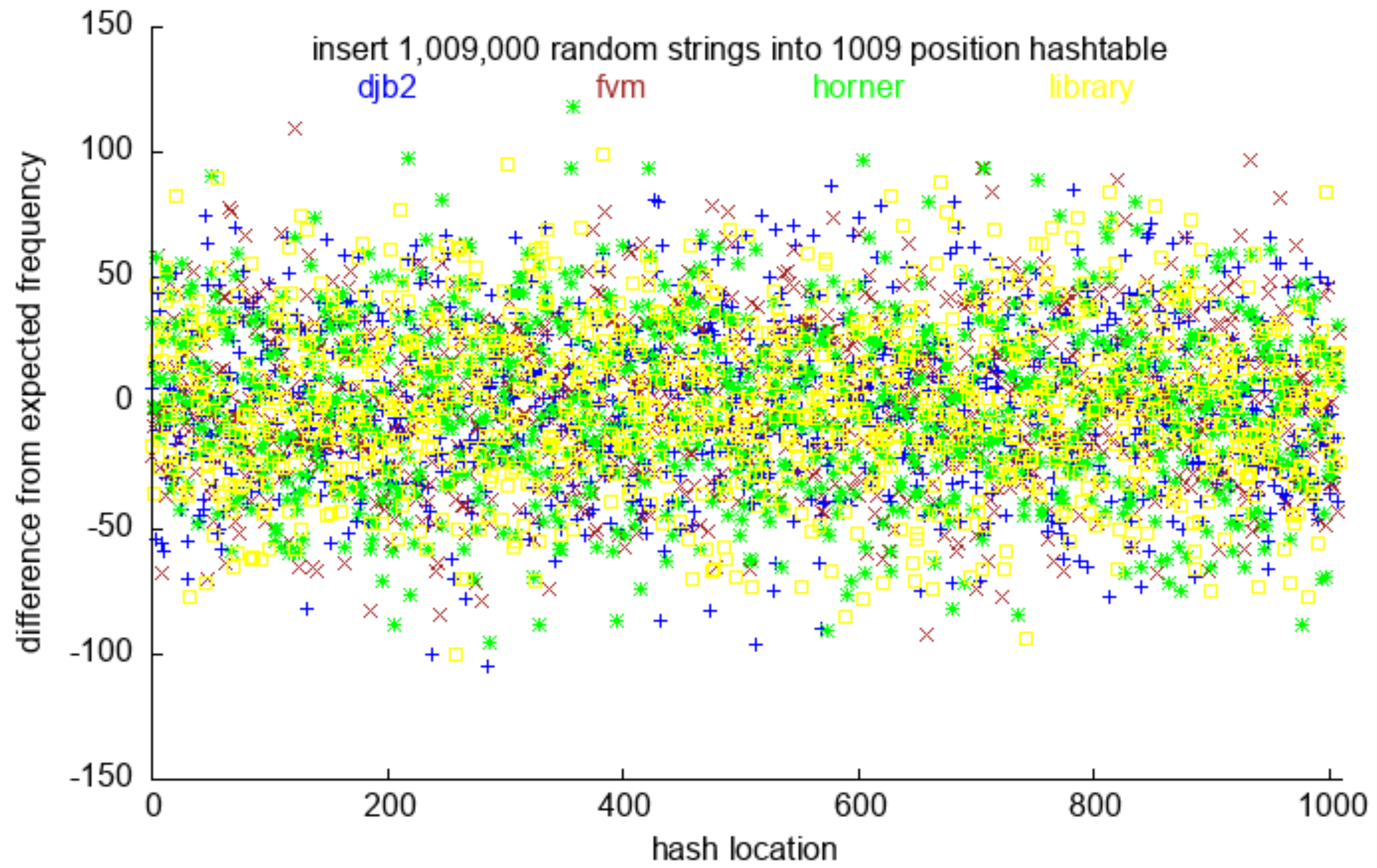
```
// by Ken Thompson
public long sdbmHash(String data) {
    long hash = 0;
    for (int i = 0; i < data.length(); i++) {
        hash = data.charAt(i) + (hash << 6) + (hash << 16) - hash;
    }
    return hash;
}
```

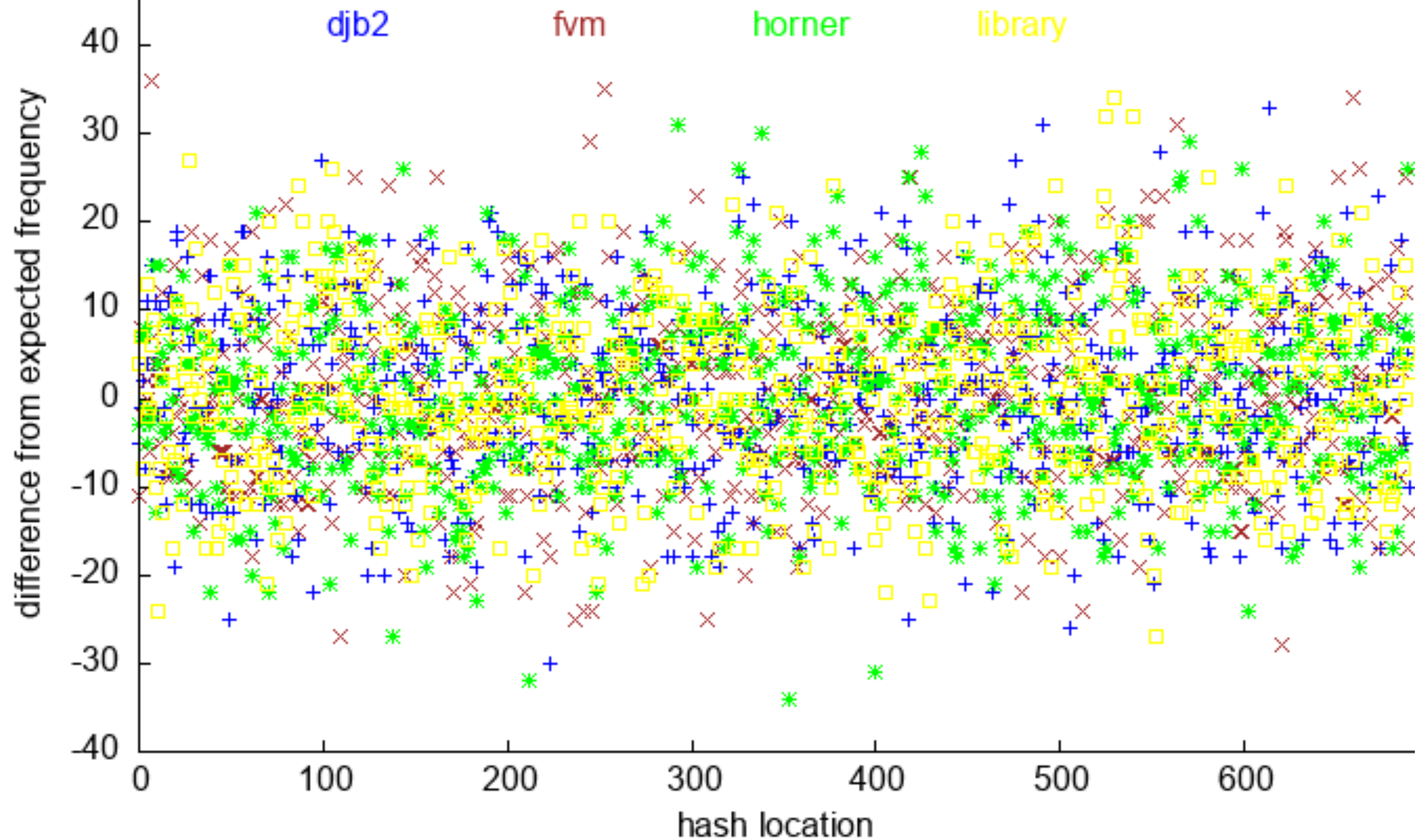
```
// Daniel J. Bernstein
public long djb2(String data) {
    long hash = 5381;

    for (int i = 0; i < data.length(); i++) {
        hash += data.charAt(i) + hash + hash<<5;
    }

    return hash;
}
```

"<<" is bit shift to left.
"<< 5" is equivalent to "* 32"
but takes 1/2 as much time (in Java)



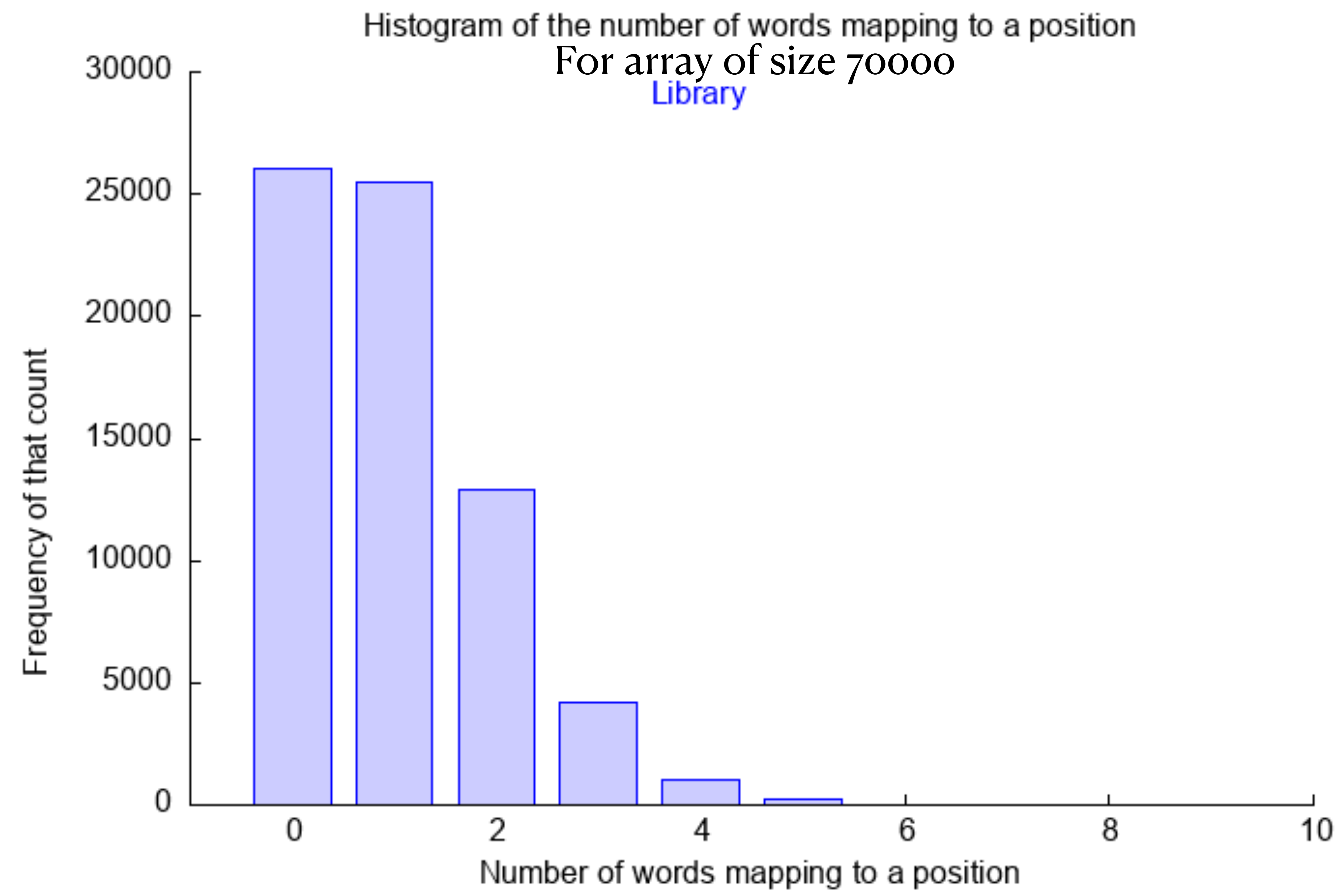
Collisions by hashing functions
insert 1,000,000 random strings into 1000 position hashtable
djb2
fvm
horner
library
200
150
100
50
0
-50
-100
-150
-200
-250
difference from expected frequency
0
200
400
600
800
1000
hash location

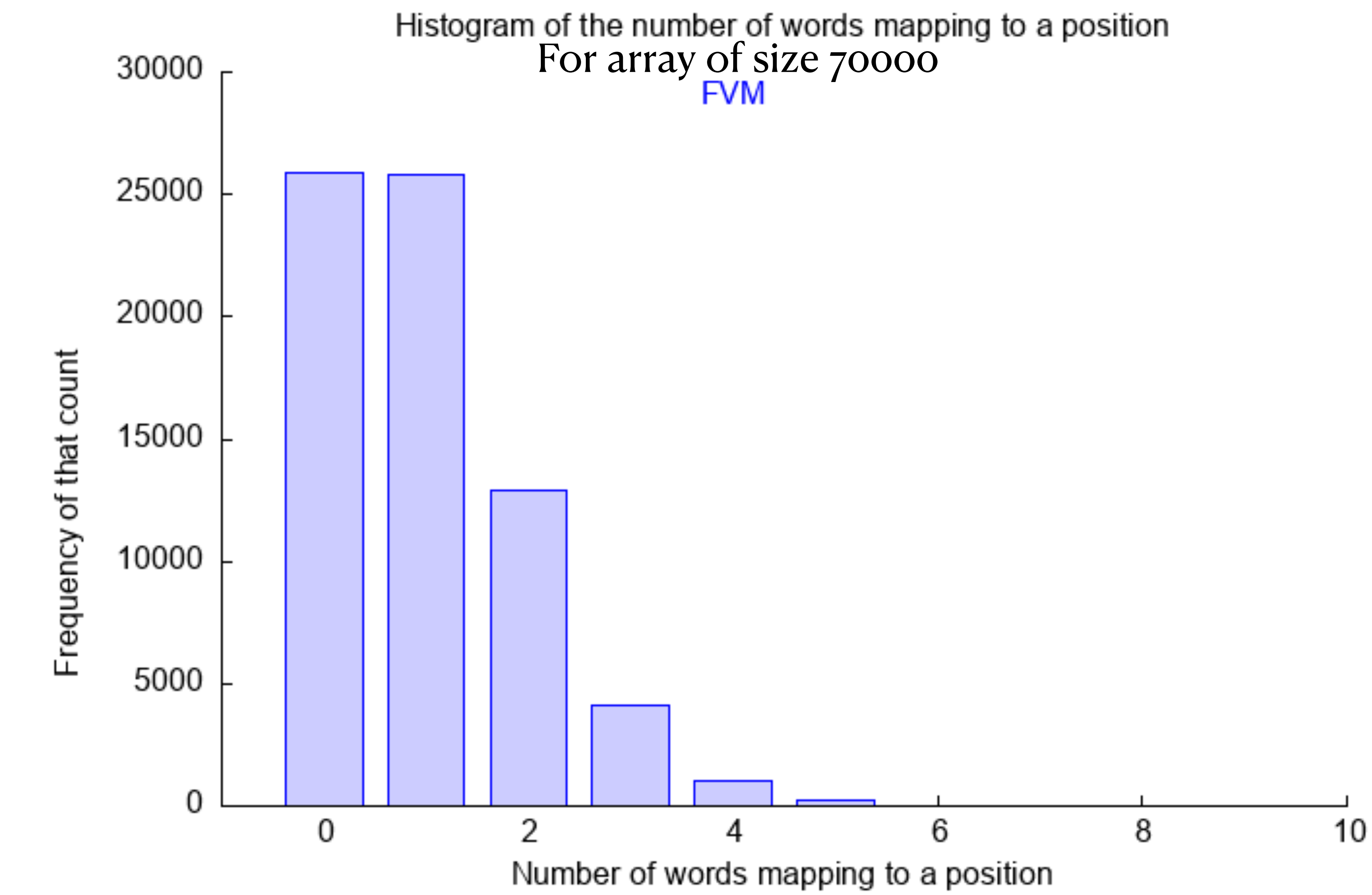

Collisions by hashing functions
insert 1,009,000 random strings into 1009 position hashtable
djb2
fvm
horner
library
difference from expected frequency
hash location
150
100
50
0
-50
-100
-150
0
200
400
600
800
1000

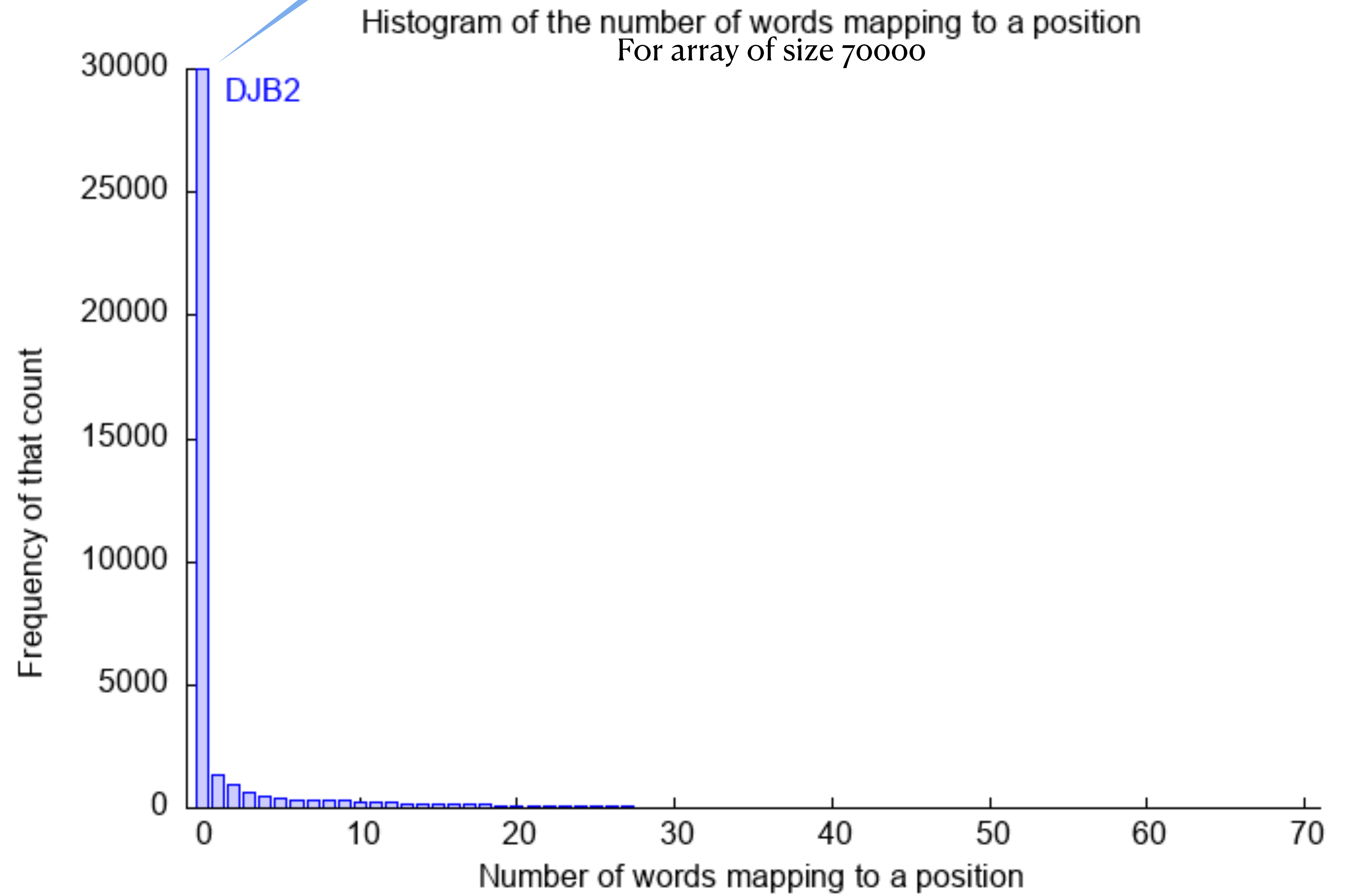


Collisions by hashing functions
insert Scott,Austen,Gibon unique strings into 691 position hashtable
djb2
fvm
horner
library
difference from expected frequency
hash location
50
40
30
20
10
0
-10
-20
-30
-40
0
100
200
300
400
500
600

Actual value is 62330





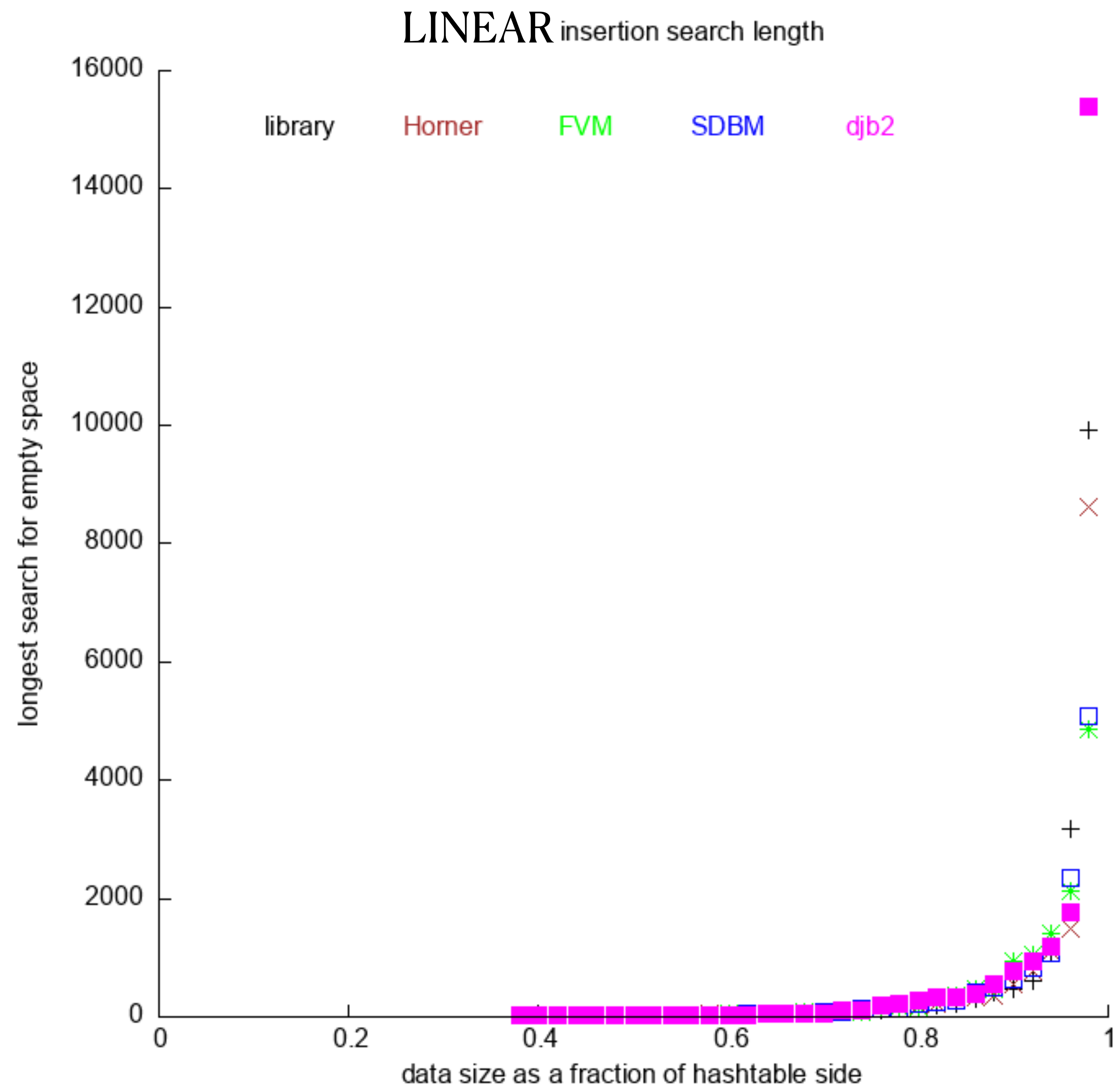


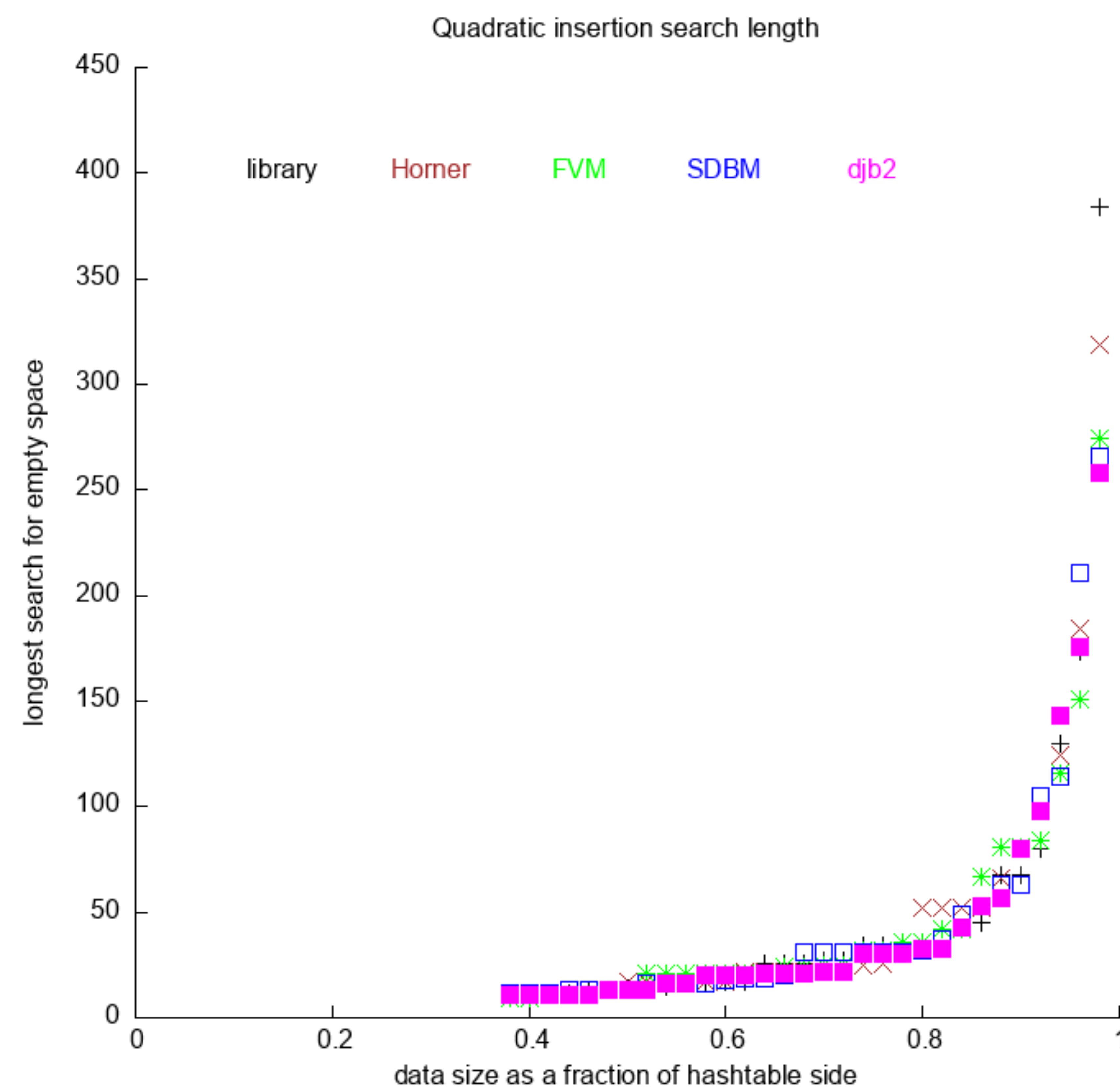
Collision Handling in Hashtables

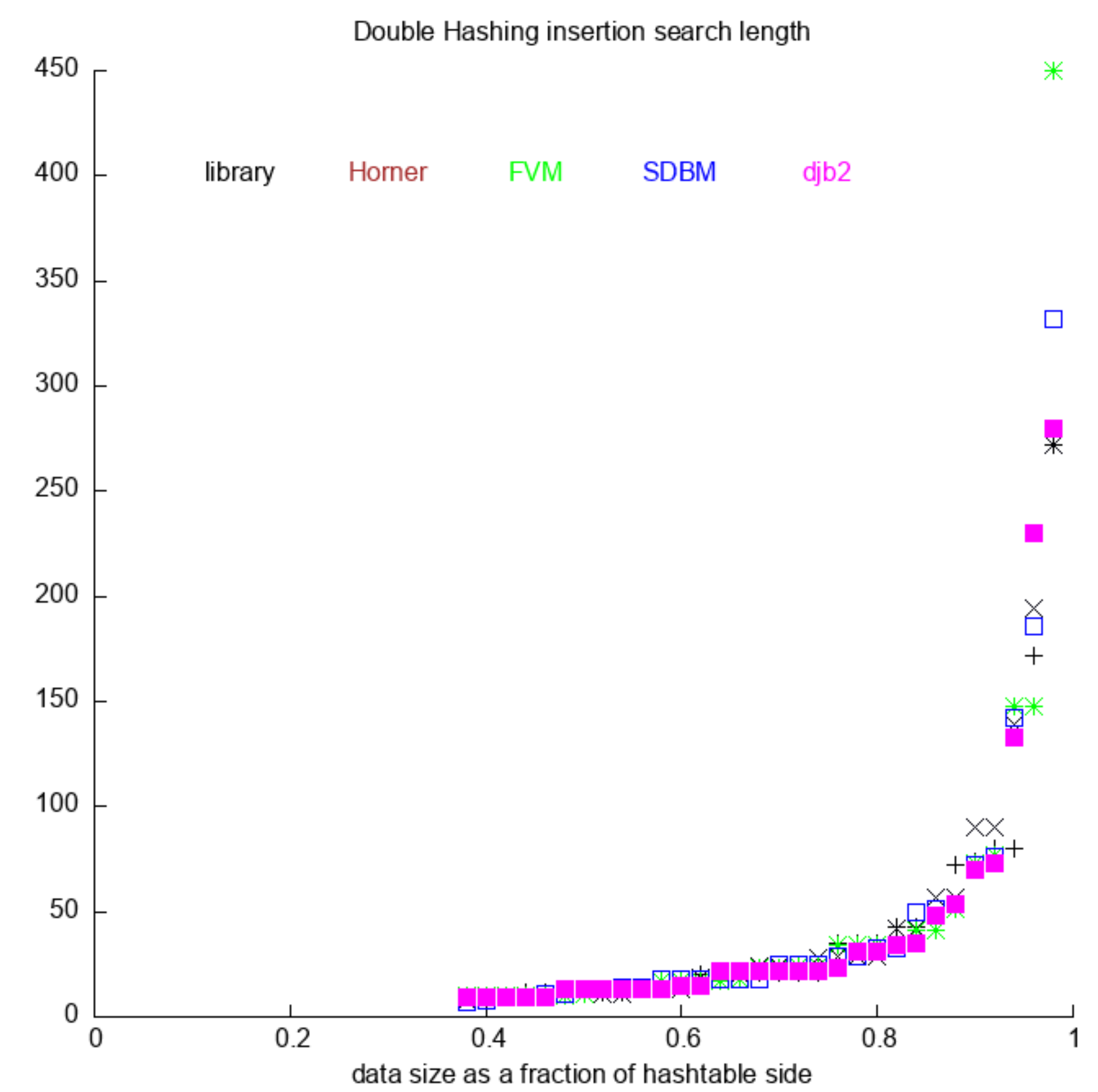
- Separate Chaining
  - number of collisions at a location is length of chain
- Probing
  - linear
  - quadratic
  - double hash

Probe search length

Hash table N=99901







Focusing on Quad and DH

SDBM only