E/15/385

Weerasinghe S.P.A.P.E

Data Structures and Algorithms

Lab03-Hash Tables

What are mentioned in the lab sheet

Objective: the objective of this laboratory class is to investigate the design issues of a hash table.

The Hash function: an ideal hash function should distribute the keys evenly for all the buckets.
With an ideal hash function your search time would be *M/N* where M is the number of keys and N
is the number of buckets. Furthermore you can make the search fast by changing N

The interface to your hash table should include the following;

1. A method to create a *HashTable* were one can specify the number of buckets needed. This
2. can be done with the constructor itself.
3. A method called void *insert(String key)* which would insert the given key to the *HashTable*.
4. A method called *int search(String key)* which would return the number of times the given

What I did:
All above methods are implemented successfully in my codes.

What are mentioned in the lab sheet
We are asked to

- **should submit** a report on what is the best hash function for this particular purpose.

main task is to develop a suitable hash function that would, as much as possible distribute the
keys evenly.

What I did:
Designed different hash function based on

- lectures done by sir
- Notes provided by sir
- Notes by yale provided on feels

- Use internet
- MIT Lecture notes provided on feels

Basically there are two types of hashing functios

1. Division Method:
2. Multiplication Method:

Other than this there is
3.Universal Hashing-
In this technique all elements are stored in the hash table itself. That is, each table entry contains either an element or NIL. When searching for element (or empty slot), we systematically examine slots until we found an element (or empty slot). There are no lists and no elements stored outside the table. That implies that table can completely "fill up"; the load factor α can never exceed 1.Advantage of this technique is that it avoids pointers (pointers need space too). Instead of chasing pointers, we compute the sequence of slots to be examined. To perform insertion, we successively examine or probe, the hash table until we find an empty slot. The sequence of slots probed "depends upon the key being inserted." To determine which slots to probe, the hash function includes the probe number as a second input

Developed separate HashTableImp for different hash functions,namely

Summing Hash Function - This hash function adds up the integer values of the chars in the string (then need to take the result mod the size of the table). This function is simple to compute, but it often doesn't work very well in practice. Then the modulus M is taken so that it distributes words into buckets.

$$\boldsymbol{Hash}=(\Sigma \boldsymbol{string.charAt(i)})\ \boldsymbol{mod\ M}$$

1. HashTable-Sum of ASCII values of characters

Modular Hashing Function - Modular hashing works for long keys such as strings, too: we simply treat them as huge integers. For example, the code below computes a modular hash function for a String s, where R is a small prime integer (Java uses 31). Here relatively large number are generated as each time Hash is multiplied by R. Hence, overflow may occur and to avoid negative values absolute values are taken. Java may not actually use absolute values because hash Code may have negative values.

$$\boldsymbol{Hash}=\mathbf{0}\ (\boldsymbol{Initially})\ \boldsymbol{Hash}=|\boldsymbol{Hash}*\boldsymbol{R}+\boldsymbol{string.charAt(i)}|\ \boldsymbol{mod\ M}$$

2. HashTableImp1-Default Algorithm in Java (for String hash)
3. HashTableImp2-**DJB2 Algorithm**
4. HashTableImp3-**SDBM Algorithm**
5. HashTableImp4-**Use 71 as factor to multiply(odd number)**
6. HashTableImp5-**Use 523 (larger odd number)**

Position value based Hashing – Simply they are treated as large integers and each character position is assigned with particular integer values. The ASCII value of each character is taken and multiplied with the respective position value and summed together. Here for position value base 128 is used. Note that here absolute value is taken to get rid of negative numbers due to overflow. M is the no of buckets and taking mod M guarantees each word will fall in to a certain bucket.

$$\boldsymbol{Hash} = |\Sigma \mathbf{128}_{\boldsymbol{i}} \times \boldsymbol{string.charA}(\boldsymbol{i})| \ \boldsymbol{mod}\ (\boldsymbol{M})$$

HashTableImp7

To decide which is the best hash function
We were asked as follows

Your report should include graph to depict how the buckets are filled for different number of buckets, for different number of hash functions and different text files. You can decide how to show this (for,example, maximum and minimum number of entries in buckets, average and standard deviation etc.can be used).

- Among these parameters

Basically the performance is inversely proportional to the "Collisions" happens. As we see the structure we can conclude that number of the keys entered to a certain bucket is proportional with the collisions.

Average

When considering average it is always the same for every hash function for a particular number of buckets and a particular
set of words. It will not be better parameter when comparing.

Standard Deviation

Standard deviation gives a good idea about how the bucket sizes are distributed.So this is a reasonable parameter. Lower standard deviations implies good performance of a hash function.

Maximum and Minimum

Using these we can calculate the range of distribution.The range between the min and max is an indication of the distribution. Lower the range, better the performance of the hash function. But there can be a cases where min or max and range might not give a real idea about hash function.

Hashing techniques tested
For each implementation hashfnction is changed

1.HashTableImp-**Sum of ASCII values of characters**

For *i=1:length_of_word:*
hash = hash + simple_character_at_i_in_the_word

2. HashTableImp1-**Java default method**
For *i=1:length_of_word:*
*hash = (hash*31) + character_at_i_in_word*

3***.*** HashTableImp2-**DJB2 Algorithm**
For *i=1:length_of_word:*
*hash = (hash*33) + character_at_i_in_word*

4. **.** HashTableImp3-**SDBM Algorithm**
For *i=1:length_of_word:*
*hash = (hash*65599) + character_at_i_in_word*

5. **.** HashTableImp4-
For *i=1:length_of_word:*
*hash = (hash*71) + character_at_i_in_word*

6. **.** HashTableImp5-
For *i=1:length_of_word:*
*hash = (hash*523) + character_at_i_in_word*

7.HashTableImp6-

```
hash = Math.abs(hash + (long)c * (long)Math.pow(128,i));
    i++;

hash = hash % hashTable.length;          //Get the appropriate bucket
hashVal = (int)hash;
```

How to run the Program

1.javac HashTableImp(1,2,3,4,5,6).java
2. javac TestHashTable(1,2,3,4,5,6).java
3. java TestHashTable(1,2,3,4,5,6) bucket 30 sample-text1.txt

Number of items in each buckets and Average and standard deviation of buckets will be displayed.

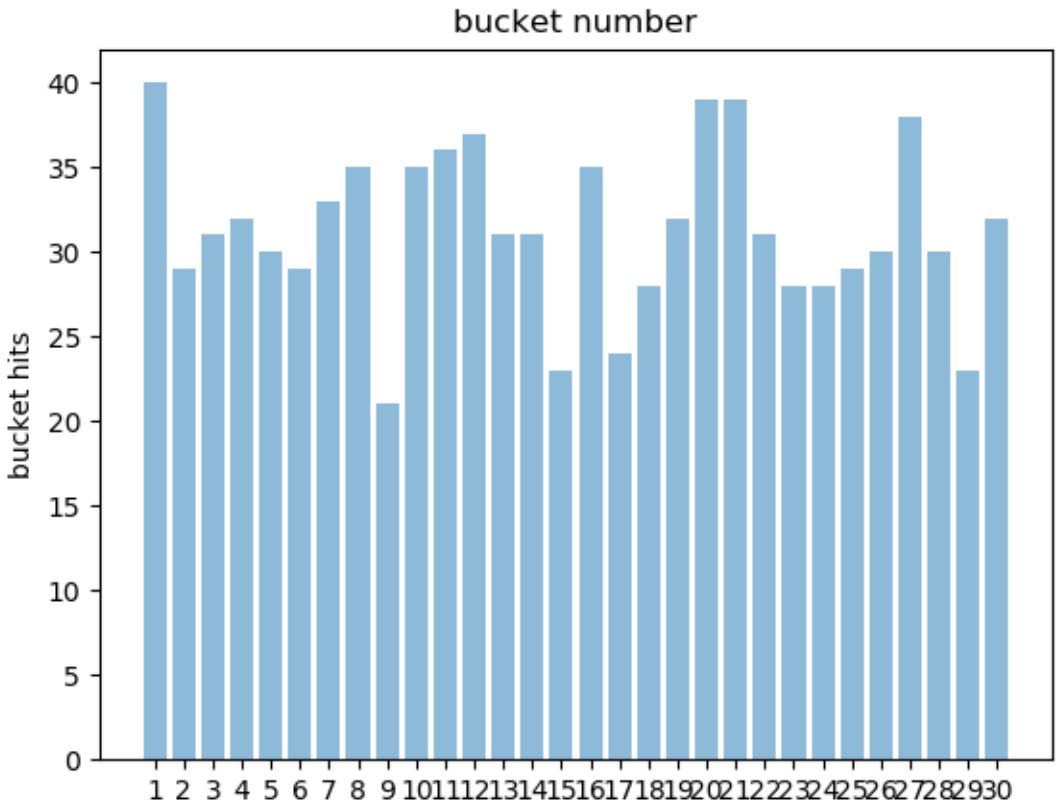
Following is a sample input bucket size is “30” and file name is “sample-text1.txt”

```
Java TestHashTable bucket 30[bucket size] sample-text1.txt[file name]
```

Lets analyse the hash function results
Bucket size=30

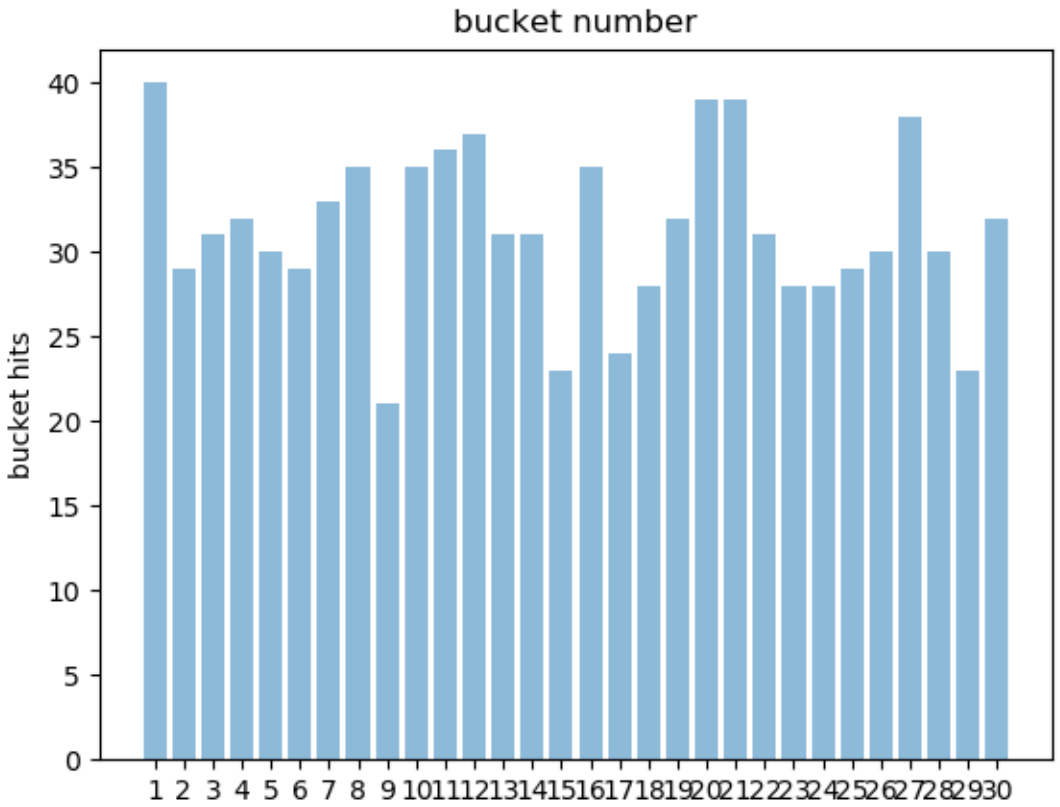
HashTableImp

- Total:939
- Average:31.3
- Standard Deviation:4.723352474011441
- Max=40
- Min=21



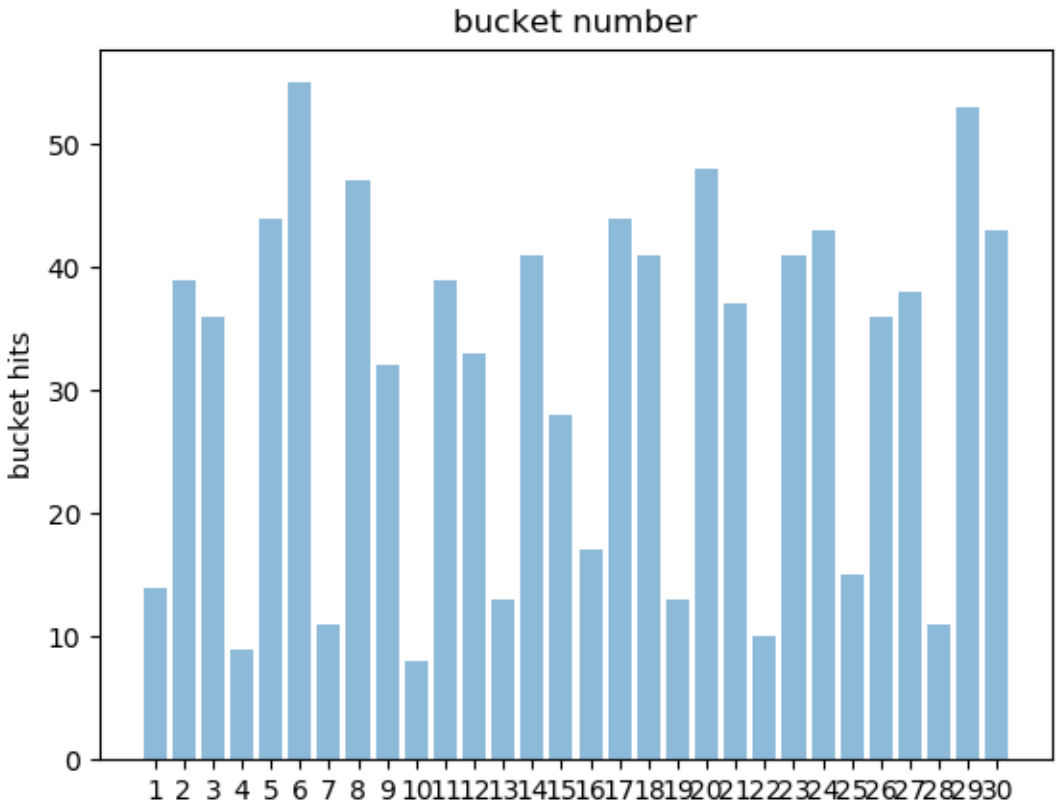
hashTableImp1

- Total:939
- Average:31.3
- Standard Deviation:4.723352474011441
- Max=40
- Min=21



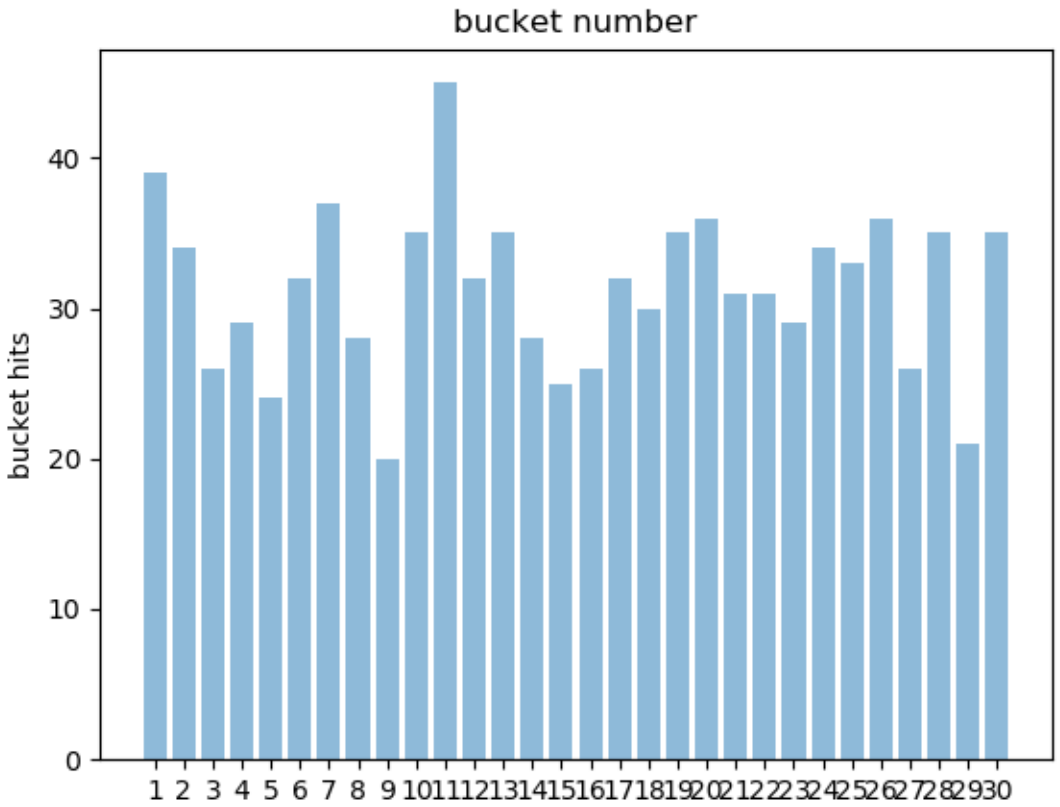
HashTableImp2

- Total:939
- Average:31.3
- Standard Deviation:14.639332587032444
- Max=55
- Min=7



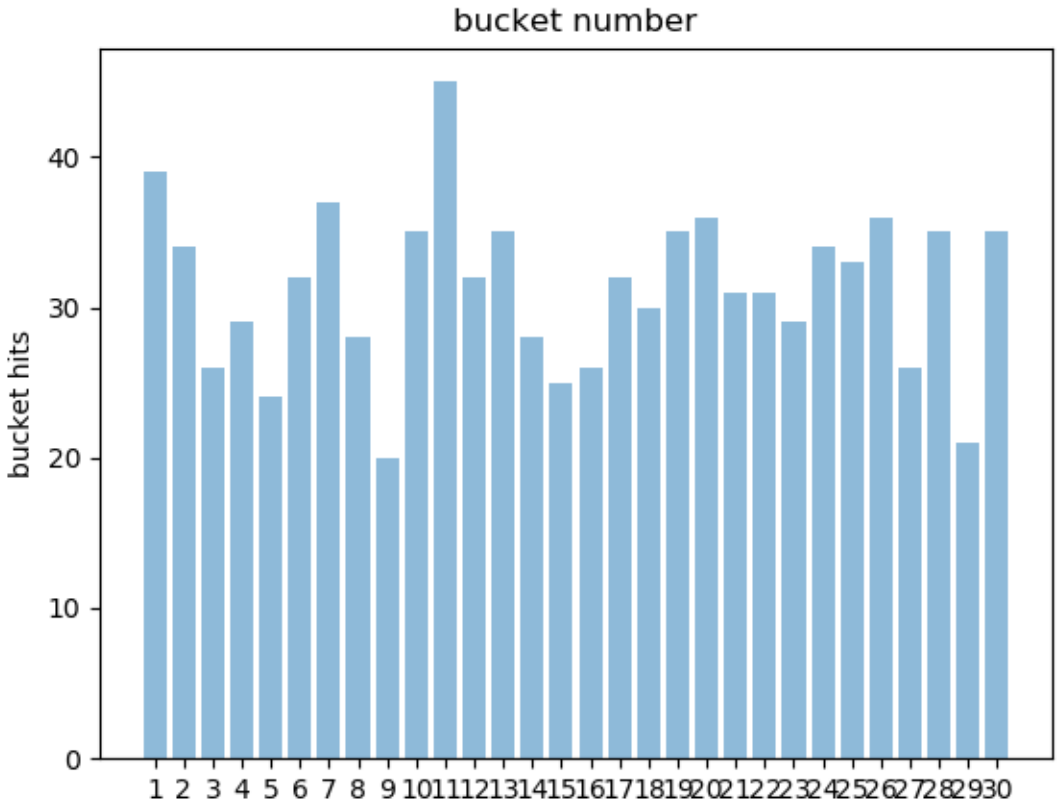
hashTabelImp3

- Total:939
- Average:31.3
- Standard Deviation:5.320719743958518
- Max=48
- Min=21



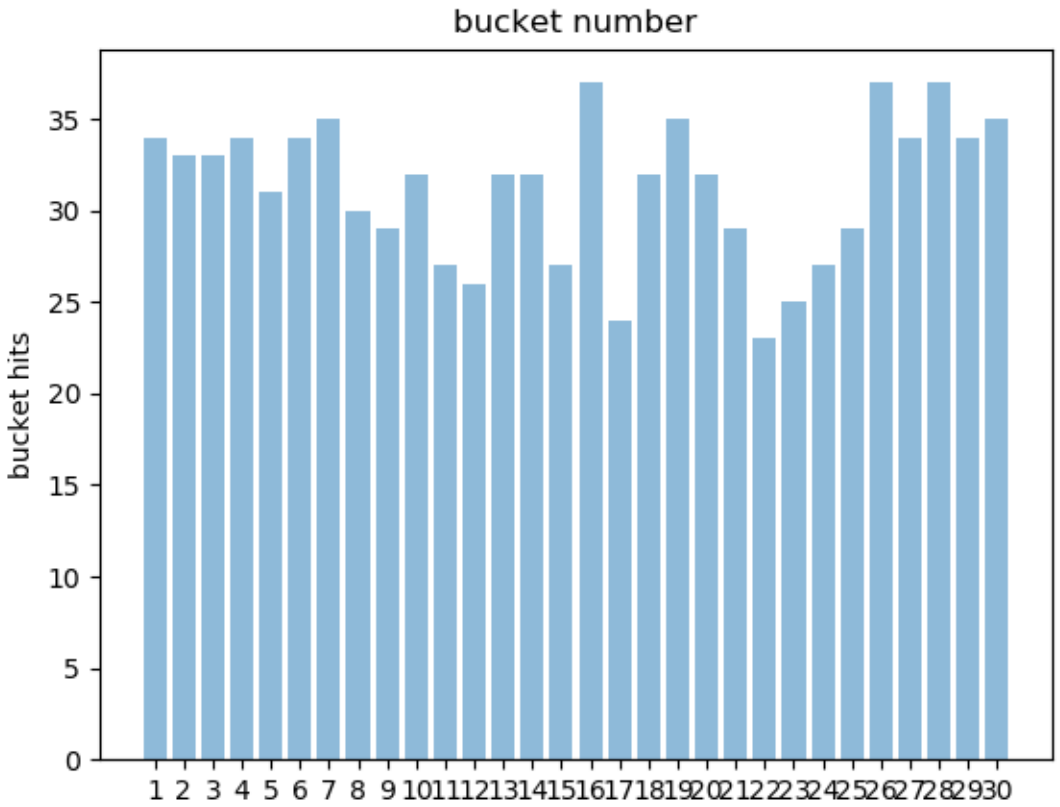
hashTableImp4

- Total:939
- Average:31.3
- Standard Deviation:5.320719743958518
- Max=48
- Min=21



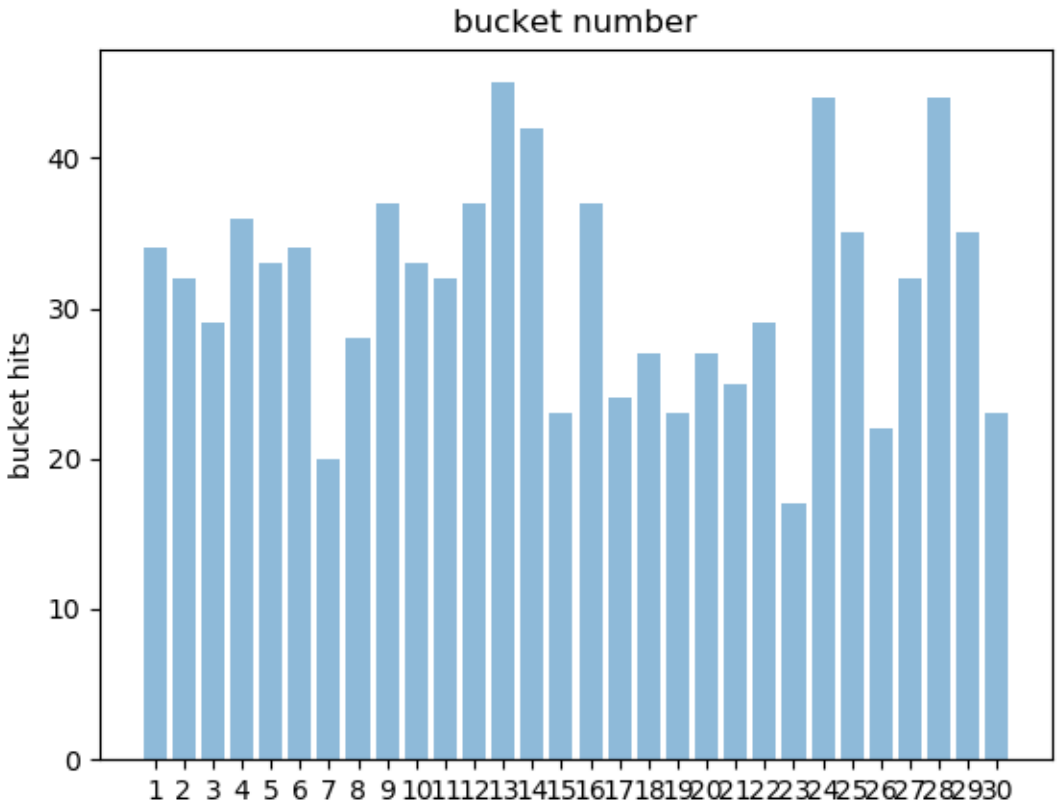
HashTableImp5

- Total:939
- Average:31.3
- Standard Deviation:3.7828638085120114
- Max=37
- Min=22



HashTableImp6

- Total:939
- Average:31.3
- Standard Deviation:7.232569294085609



When we compare hash codes in HashTable1,2,3,4,5,6 we can see that average hits per bucket is same. Although all hash codes were able to distribute keys through buckets in an almost uniform manner, when consider minimum, maximum hits and Standard Deviation of hits/bucket, **hashTableImp5** outperforms all other hash codes. (a type of modular hashing function)

HashTableImp5-
For *i=1:length_of_word:*
*hash = (hash*523) + character_at_i_in_word*

Thus we can say that in a limited memory system where number of collisions in a bucket is not a critical issue, hashTableImp5 will be more suitable to expect a better performance of the hash table.

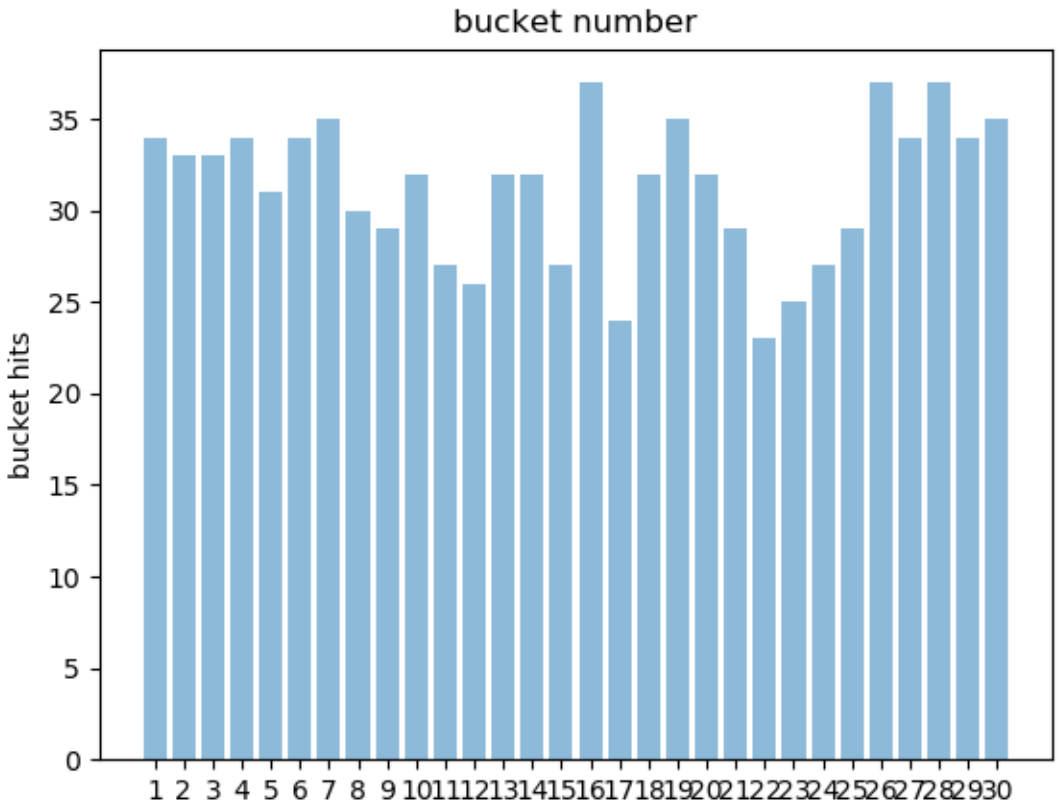
PART 2:

Use of same hash function for different file(sample-text1 and sample-text2)

We will use HashTableImp5 with bucket size of 30

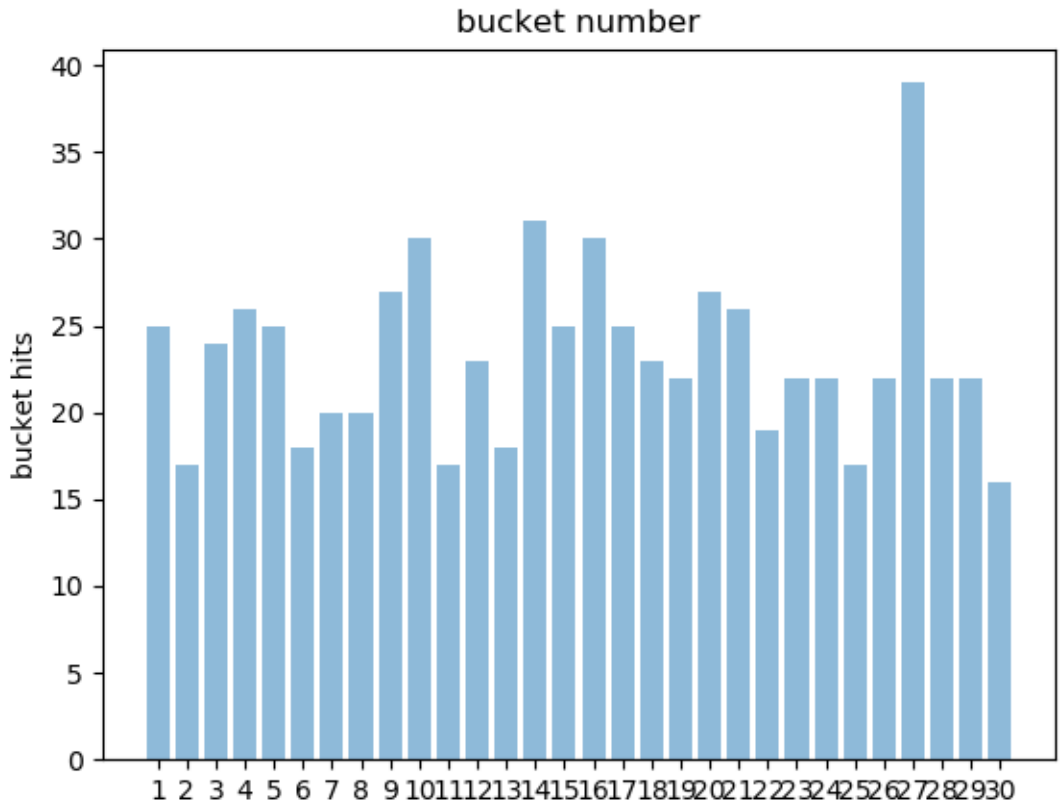
Sample-textfile1

- HashTableImp5
- Total:939
- Average:31.3
- Standard Deviation:3.7828638085120114
- Max=37
- Min=22



Sample-textfile2

- Total:700
- Average:23.333334
- Standard Deviation:4.853405195549564
- Max=37
- Min=24



#Buckets	100	50	30	20	10	5
Average	8.52	18.04	30.73333	46.6	94.2	189.4
Std. Deviation	3.163795	4.38616	4.94593	7.344386	11.3031	16.54811
Maximum	17	27	40	61	115	209
Minimum	0	5	21	33	74	161
Upper Bound	14.84759	26.81232	40.62519	61.28877	116.8062	222.4962
Lower Bound	2.19241	9.26768	20.84147	31.91123	71.59381	156.3038

Table1:Sample-text1 parameters with HashTableImp5 for differnet num of Bucket sizes

#Buckets	100	50	30	20	10	5
Average	5.95	12.9	22.16667	33.75	68.5	138
Std. Deviation	2.531304	3.894868	3.89943	3.793086	3.930649	6.324555
Maximum	12	22	32	40	75	150
Minimum	1	7	14	25	63	132
Upper Bound	11.01261	20.68974	29.96553	41.33617	76.3613	150.6491
Lower Bound	0.887392	5.110263	14.36781	26.16383	60.6387	125.3509

Table1:Sample-text2 parameters with HashTableImp5 for differnet num of Bucket sizes

When the same hash function is used for different files it has given different results since the complexity of vocabulary used in 2 files is almost completely different therefore the domain of generated hash codes are also completely different. However despite the complexity of 2 files when an odd number is used as the multiplier in the hash function better uniformity has been occurred than when an even number is used. Also several odd numbers has given the best uniformity above all other odd multipliers.