Linnaeus University

1DV516 – Algorithms and data structures
Assignment 1

Student: Fredric Eriksson Sepúlveda
Student ID: fe222pa@student.lnu.se

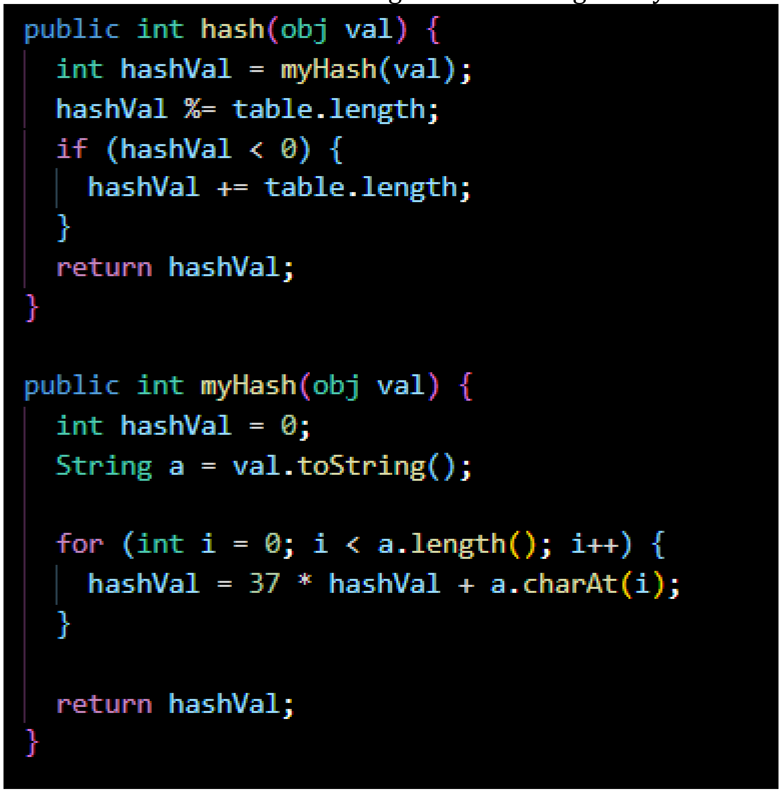
Table of contents

Contents

Problem 2

Aim: Experiment and analyze how well your hash function works

The hash function that I am using is considered good by the book.

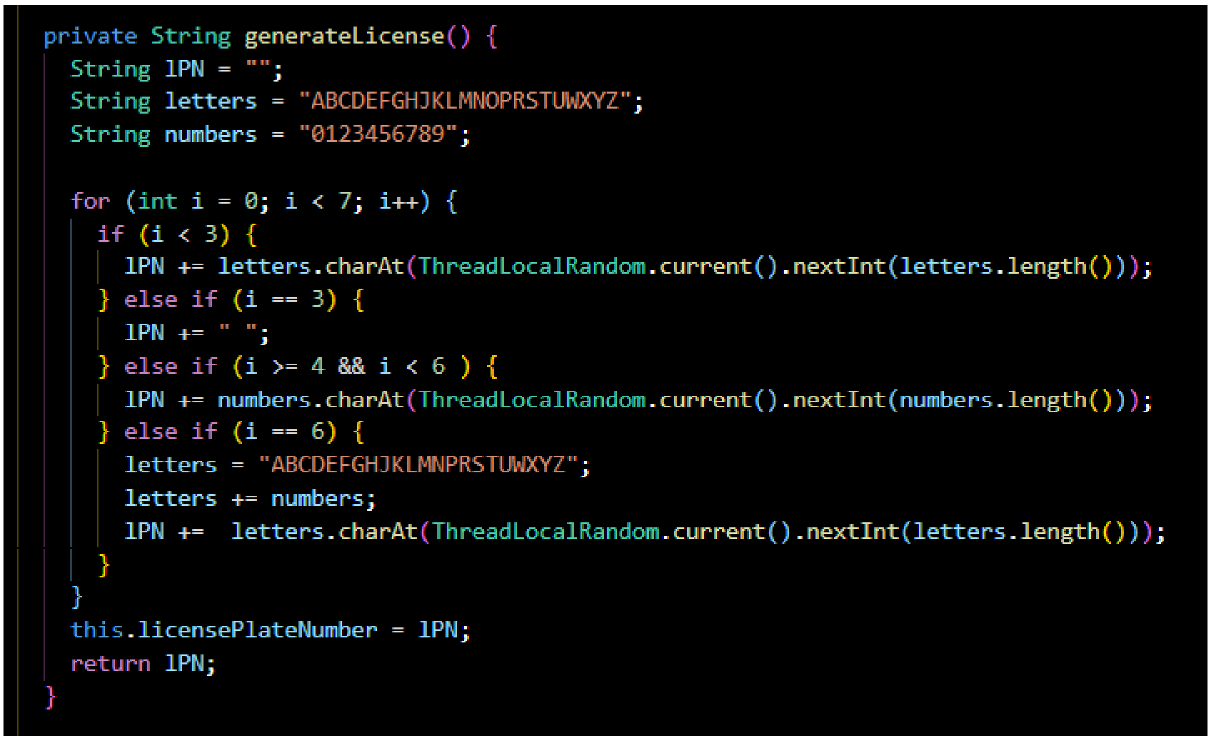


```
public int hash(obj val) {
    int hashVal = myHash(val);
    hashVal %= table.length;
    if (hashVal < 0) {
        hashVal += table.length;
    }
    return hashVal;
}

public int myHash(obj val) {
    int hashVal = 0;
    String a = val.toString();

    for (int i = 0; i < a.length(); i++) {
        hashVal = 37 * hashVal + a.charAt(i);
    }

    return hashVal;
}
```

Get the object make it into a string and get a value from calculating each character's value

Then that value is mod by the table length and if it becomes less than 0 then just add the total.

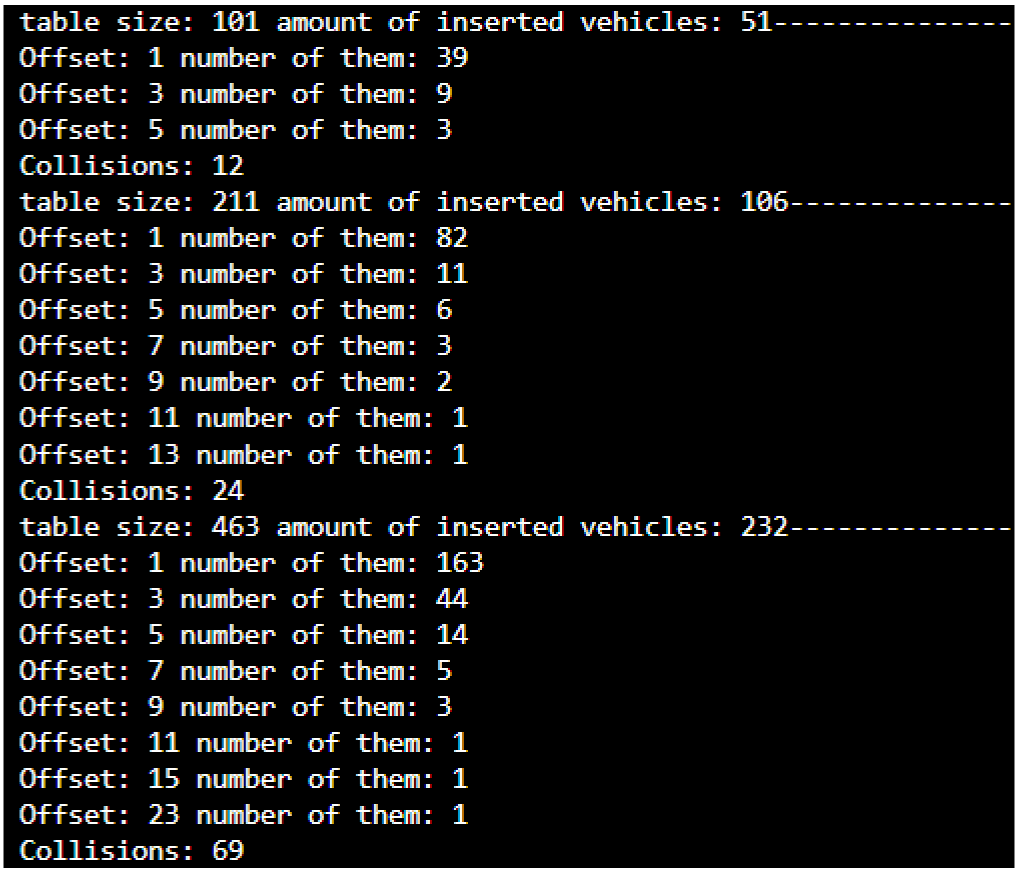
My manner of obtaining license plates is:



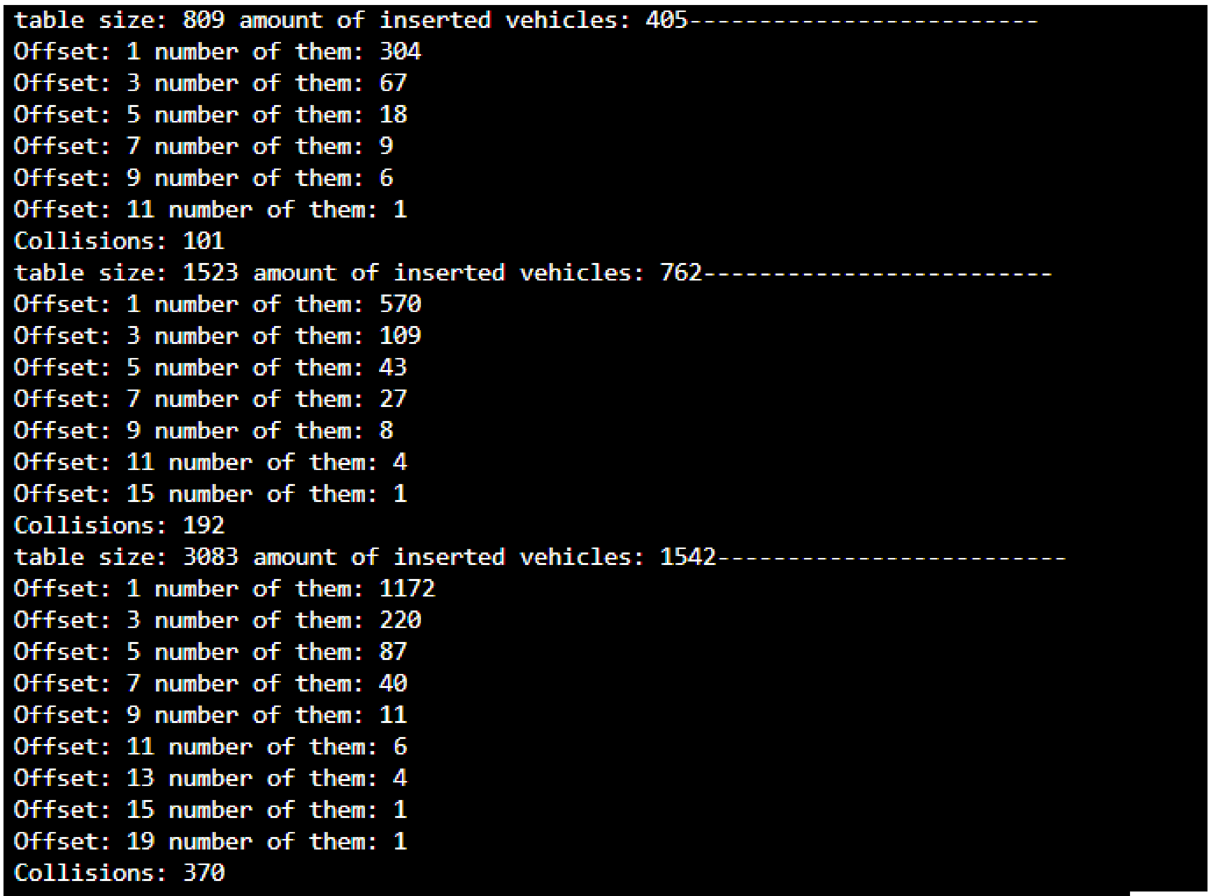
```java
private String generateLicense() {
  String lPN = "";
  String letters = "ABCDEFGHJKLMNOPRSTUWXYZ";
  String numbers = "0123456789";

  for (int i = 0; i < 7; i++) {
    if (i < 3) {
      lPN += letters.charAt(ThreadLocalRandom.current().nextInt(letters.length()));
    } else if (i == 3) {
      lPN += " ";
    } else if (i >= 4 && i < 6 ) {
      lPN += numbers.charAt(ThreadLocalRandom.current().nextInt(numbers.length()));
    } else if (i == 6) {
      letters = "ABCDEFGHJKLMNPRSTUWXYZ";
      letters += numbers;
      lPN +=  letters.charAt(ThreadLocalRandom.current().nextInt(letters.length()));
    }
  }
  this.licensePlateNumber = lPN;
  return lPN;
}
```

I follow the Swedish rules such as 3 letters space 2 numbers and letters + numbers
And taking out forbidden letters such as I, Q, V, and O In the last one.

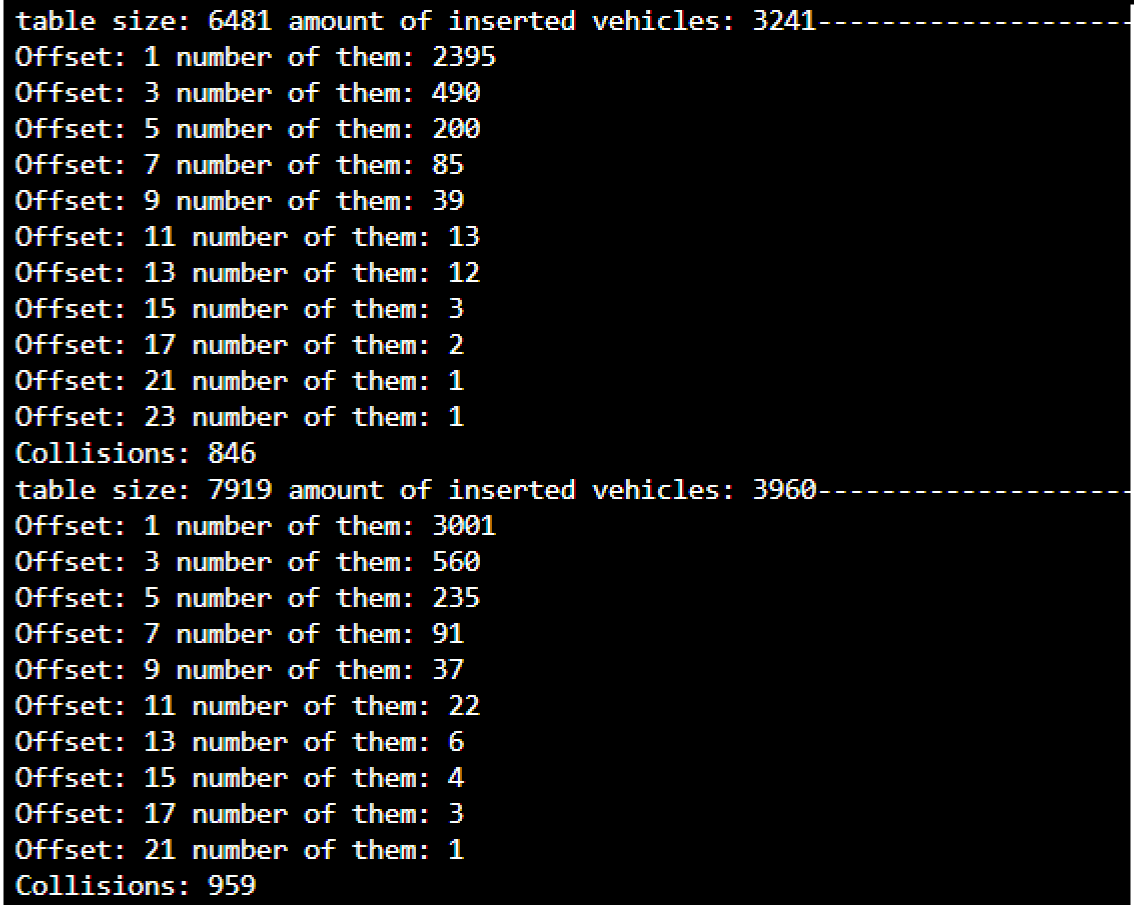
Results



```
table size: 101 amount of inserted vehicles: 51----------------
Offset: 1 number of them: 39
Offset: 3 number of them: 9
Offset: 5 number of them: 3
Collisions: 12
table size: 211 amount of inserted vehicles: 106--------------
Offset: 1 number of them: 82
Offset: 3 number of them: 11
Offset: 5 number of them: 6
Offset: 7 number of them: 3
Offset: 9 number of them: 2
Offset: 11 number of them: 1
Offset: 13 number of them: 1
Collisions: 24
table size: 463 amount of inserted vehicles: 232--------------
Offset: 1 number of them: 163
Offset: 3 number of them: 44
Offset: 5 number of them: 14
Offset: 7 number of them: 5
Offset: 9 number of them: 3
Offset: 11 number of them: 1
Offset: 15 number of them: 1
Offset: 23 number of them: 1
Collisions: 69
```



```
table size: 809 amount of inserted vehicles: 405------------------------
Offset: 1 number of them: 304
Offset: 3 number of them: 67
Offset: 5 number of them: 18
Offset: 7 number of them: 9
Offset: 9 number of them: 6
Offset: 11 number of them: 1
Collisions: 101
table size: 1523 amount of inserted vehicles: 762------------------------
Offset: 1 number of them: 570
Offset: 3 number of them: 109
Offset: 5 number of them: 43
Offset: 7 number of them: 27
Offset: 9 number of them: 8
Offset: 11 number of them: 4
Offset: 15 number of them: 1
Collisions: 192
table size: 3083 amount of inserted vehicles: 1542------------------------
Offset: 1 number of them: 1172
Offset: 3 number of them: 220
Offset: 5 number of them: 87
Offset: 7 number of them: 40
Offset: 9 number of them: 11
Offset: 11 number of them: 6
Offset: 13 number of them: 4
Offset: 15 number of them: 1
Offset: 19 number of them: 1
Collisions: 370
```

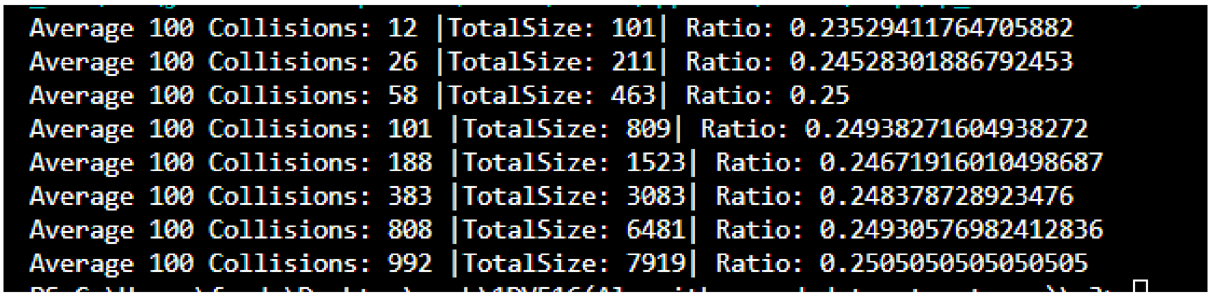


```
table size: 6481 amount of inserted vehicles: 3241-------------------
Offset: 1 number of them: 2395
Offset: 3 number of them: 490
Offset: 5 number of them: 200
Offset: 7 number of them: 85
Offset: 9 number of them: 39
Offset: 11 number of them: 13
Offset: 13 number of them: 12
Offset: 15 number of them: 3
Offset: 17 number of them: 2
Offset: 21 number of them: 1
Offset: 23 number of them: 1
Collisions: 846
table size: 7919 amount of inserted vehicles: 3960-------------------
Offset: 1 number of them: 3001
Offset: 3 number of them: 560
Offset: 5 number of them: 235
Offset: 7 number of them: 91
Offset: 9 number of them: 37
Offset: 11 number of them: 22
Offset: 13 number of them: 6
Offset: 15 number of them: 4
Offset: 17 number of them: 3
Offset: 21 number of them: 1
Collisions: 959
```

Analysis

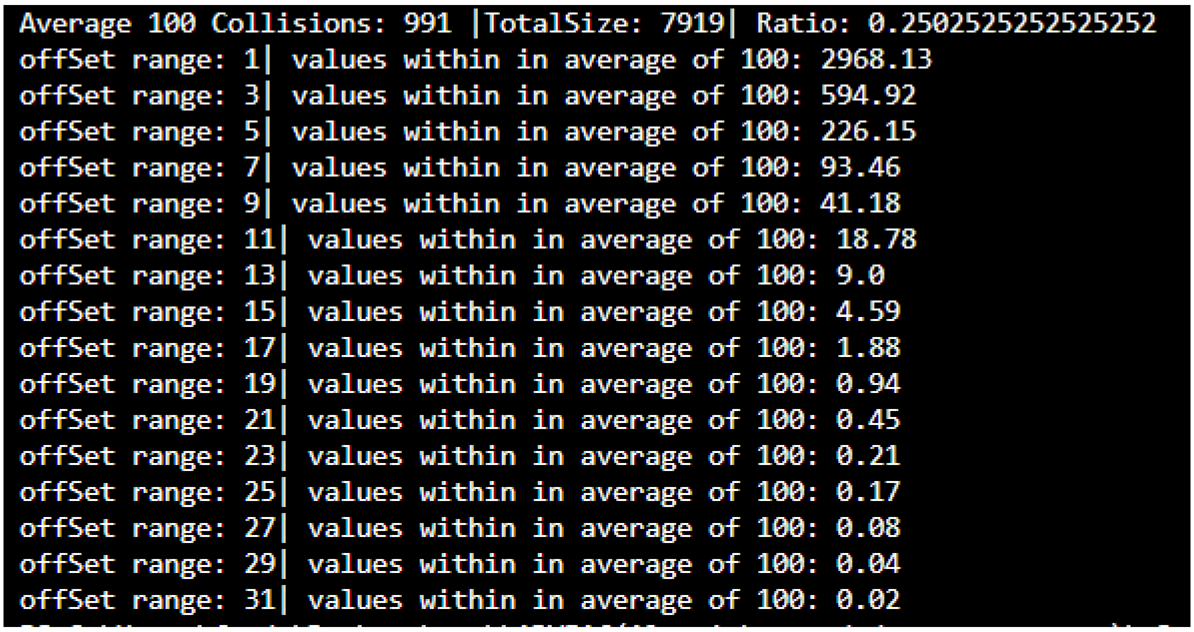
The highest offset is 23 and the most common offset is 3
The highest ratio of collisions/inserted is 0.2974 at table size 463
The lowest is 0.2264 at table size 211.

Higher tables do not deviate from 0.23 to 0.29



```
Average 100 Collisions: 12 |TotalSize: 101| Ratio: 0.23529411764705882
Average 100 Collisions: 26 |TotalSize: 211| Ratio: 0.24528301886792453
Average 100 Collisions: 58 |TotalSize: 463| Ratio: 0.25
Average 100 Collisions: 101 |TotalSize: 809| Ratio: 0.24938271604938272
Average 100 Collisions: 188 |TotalSize: 1523| Ratio: 0.24671916010498687
Average 100 Collisions: 383 |TotalSize: 3083| Ratio: 0.248378728923476
Average 100 Collisions: 808 |TotalSize: 6481| Ratio: 0.24930576982412836
Average 100 Collisions: 992 |TotalSize: 7919| Ratio: 0.2505050505050505
```

This is a more comprehensive test made with 100 iterations
And it seems that the ratio lies between 0.23 and 0.25



```
Average 100 Collisions: 991 |TotalSize: 7919| Ratio: 0.2502525252525252
offSet range: 1| values within in average of 100: 2968.13
offSet range: 3| values within in average of 100: 594.92
offSet range: 5| values within in average of 100: 226.15
offSet range: 7| values within in average of 100: 93.46
offSet range: 9| values within in average of 100: 41.18
offSet range: 11| values within in average of 100: 18.78
offSet range: 13| values within in average of 100: 9.0
offSet range: 15| values within in average of 100: 4.59
offSet range: 17| values within in average of 100: 1.88
offSet range: 19| values within in average of 100: 0.94
offSet range: 21| values within in average of 100: 0.45
offSet range: 23| values within in average of 100: 0.21
offSet range: 25| values within in average of 100: 0.17
offSet range: 27| values within in average of 100: 0.08
offSet range: 29| values within in average of 100: 0.04
offSet range: 31| values within in average of 100: 0.02
```

An example of how the offset skew in a large number of tests, is that all tests show that most skew towards smaller values rather than higher ones.
This means that quadratic probing is proving to be effective at fighting collisions.

Since the scalability does not present any major changes in behavior means that the hash table works well.
Regarding the hash function: Since the object inserted has a 25% chance of landing in a collision, we would rely much on the quadratic probing to resolve the conflicts. Which in part might affect performance.

Problem 5

Setup

The experiments were performed this way:
Size is asked at the start and then a new array of that size filled with random integers is created.

The Quicksort class contains a cutoff of sorts mostly to prevent errors while doing experiments and would not affect much of the outcomes because the sorter would run in a very small table that many sorters would work on almost the same time.

Experiment

For quicksortInsert and quicksortHeap I create a random array of designated size and keep track of it, then I calculate the "max depth" which is in short terms my prediction for best performance time if and only if the partition is done perfectly at the middle of every partition.

That is not always the case because I am inserting random variables however it gives a good insight into how to predict and think about the findings.

The value is obtained by counting how many times can you divide the size until you get a value that is not 1.

To prevent out-of-bound errors I made a clause that triggers if the depth assigned is not reached but the size of the list is 3 or less, and sort that. These will be seen in the raw data however they are quite simple to tell apart since any values obtained after "max depth" must have used it, and that is the reason I set my depth too high in the test to weed out the just sort 3 of a list, which would be the last value.

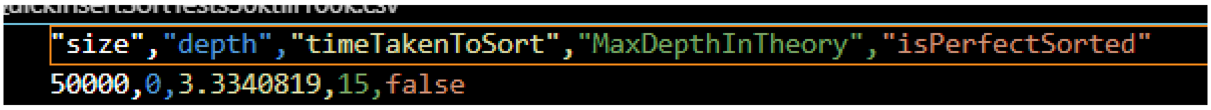
There will be two CSV files made, one for a smaller size but starts at depth 0 and another with higher sizes but depth starts at 10
The aim is to try to obtain values that are in seconds.

The way I keep track if a partition is "perfect" is that I check if ever in the depth, the size of the partition lies between 1 and 100 and it is obtained at least once, which I know is not the perfect way of determining it but it gives a good rough estimate.

QuickInsertSort

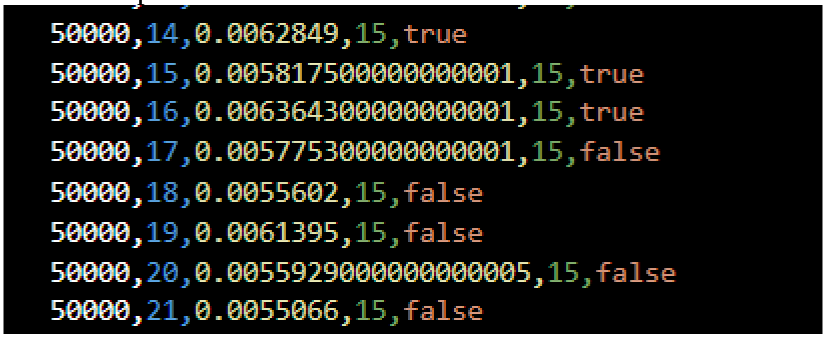
Most important values:

At start



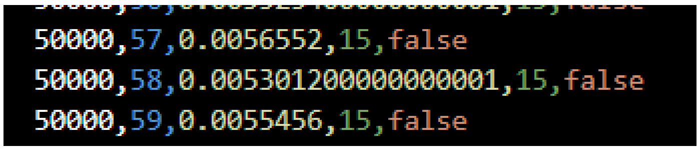
```
"size","depth","timeTakenToSort","MaxDepthInTheory","isPerfectSorted"
50000,0,3.3340819,15,false
```

At max depth



```
50000,14,0.0062849,15,true
50000,15,0.005817500000000001,15,true
50000,16,0.006364300000000001,15,true
50000,17,0.005775300000000001,15,false
50000,18,0.0055602,15,false
50000,19,0.0061395,15,false
50000,20,0.0055929000000000005,15,false
50000,21,0.0055066,15,false
```

Final values



```
50000,57,0.0056552,15,false
50000,58,0.005301200000000001,15,false
50000,59,0.0055456,15,false
```

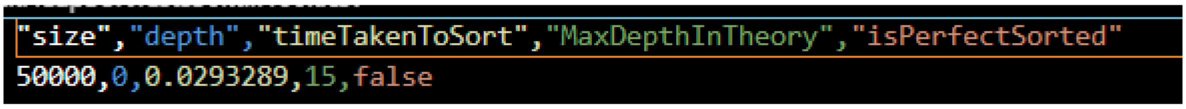
The first just tells us how long it takes to run normal insertSort in the array, the values around 15 tell us the best in theory performance
And the last one just tells us how fast it would be if all tables were sorted by size 3 instead.

Since value 58 is the fastest it mostly means that insert sort prefers that quicksort sorts most of the things and delegates really small tables to itself.

QuickHeapSort

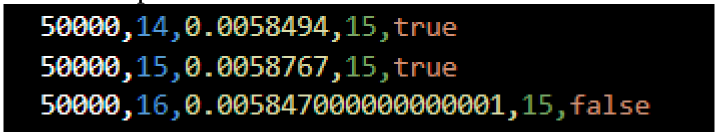
Most important values:

Start:



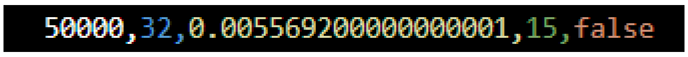
```
"size","depth","timeTakenToSort","MaxDepthInTheory","isPerfectSorted"
50000,0,0.0293289,15,false
```

At max depth



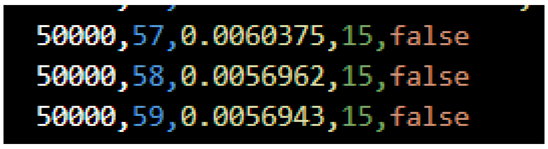
```
50000,14,0.0058494,15,true
50000,15,0.0058767,15,true
50000,16,0.005847000000000001,15,false
```

Fastest



```
50000,32,0.005569200000000001,15,false
```

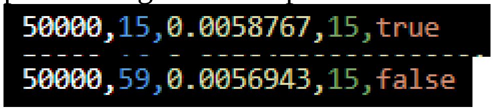
Final



```
50000,57,0.0060375,15,false
50000,58,0.0056962,15,false
50000,59,0.0056943,15,false
```

Depth 0 tells us the time it takes to run HeapSort in the whole array
Values around Depth 15 tell us the best-case partitioning
Values towards the end Tell us how the program performs by just sorting whenever the size is 3 in many cases

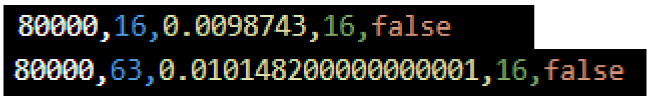
By the looks of it, two cases are going on in my raw data, either Max depth is the fastest or the highest depth is the fastest.
To explain these cases I think this is what is going on:
Since the tables are arranged randomly there might be a situation where there is a perfect partitioning at max depth when that is the case heap sort is running there



```
50000,15,0.0058767,15,true
50000,59,0.0056943,15,false
```

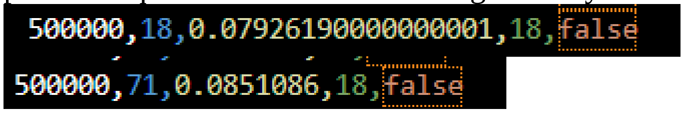
Apparently in this case running heapsort at the "perfect" partitioning makes it go slower.

This is the other case:

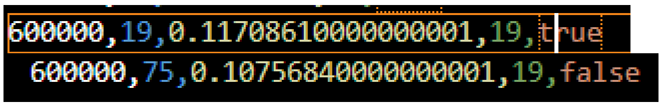


```
80000,16,0.0098743,16,false
80000,63,0.010148200000000001,16,false
```

Which is faster
Because it is running in a large array but not too large plus the higher the depth the more problems quicksort has with handling memory.



```
500000,18,0.07926190000000001,18,false
500000,71,0.0851086,18,false
```



```
600000,19,0.11708610000000001,19,true
600000,75,0.10756840000000001,19,false
```

Examples are also seen in the higher-value table

To recap running heapsort when there is no perfect partition makes it faster, but the other way makes it slower. The added time I suspect is just quicksort partitioning and doing extra work. I would argue what happens, is that the initial cost of running heapsort makes it not worth it in very small tables (around 1 and 100), and in bigger tables then it’s preferred over quicksort.
So if the depth is high and the table is small quicksort sorting is better, if the depth is high and the table is large heapsort is preferred.

Findings

Findings: Quicksort itself as a sorter is the fastest sorter, however it has 2 weaknesses:
running QuickSort in small tables wastes time.
running Quicksort with a less-than-desirable pivot generation makes the program run slow and also utilizes a large amount of memory and running heapSort when there is a perfect partitioning makes it go slower.

Thoughts

So in a way, the smartest way of optimizing this program is to define a depth and a maximum size.

Depth would keep track of the number of recursions done and by the look of the results depth value should be “MaxDepth” of the total size. If Depth is exceeded, run heapsort since it would work faster and utilize less memory.

The maximum value is there to check the size of the partitioned array, if that array is lower or equal to mV then run insertSort since that is the best scenario for insert sort.

So I think both should be used not one over the other.
InsertSort with max depth for the best case
HeapSort with max depth for the worst case