COMP26120 Lab 3 Report

Sourabh Roy

December 7, 2023

1 Experiment 1

Hypothesis The average time complexity of inserting a single value into my implementation of a hashset is $\mathcal{O}(1)$.
Theoretically, the average insertion time for a single element into a hashset is $\mathcal{O}(1)$. For every input, the hash function of the hashset generates a hash value, which is the index in the hashset where the element will be inserted.
In this implementation, the hash function consists of a polynomial function evaluated using Horner's Rule. Evaluating a polynomial using Horner's Rule takes $\mathcal{O}(t)$ time for a string input of length t. In practical scenario, since the constant factor involved in polynomial hashing is very small, we expect the time of each insertion to be roughly constant.

Experimental Design To test our hypothesis for the average case, we consider a **Random** batch of input dictionaries where the elements of each dictionary is in random order. The **independent variable** is the **size** of the dictionary. Our **dependent variable**, is the **time taken** for inserting *all* the elements of the given dictionary.
5 dictionaries of size 10K, 20K, 30K, 40K and 50K were generated. All files were generated using `random_strings.py`. The size of each dictionary entry was fixed at 40. This was done to match the average entry length of the `henry` dataset.
The query file was kept empty. This was done to avoid calculating the time for finding the queries in the hashset because we are only interested in the insertion time.
The initial size of hashset was fixed at 509, a prime number. This was done to facilitate even distribution of input elements in the hash set. The load factor, which is the threshold for resizing, was kept at 0.6. This is because, a lower load factor, although prompts resizing more frequently, drastically reduces the number of collisions thereby making the hashset faster.
The program `speller_hashset.java` was run on each dictionary 5 times with the empty query file, and the average of the 5 values was recorded.
The time taken was recorded using UNIX `time` command summing the `user` and `sys` values output by the command. **Note**: this method produces the total

time of inserting *all* elements in the dictionary. Since we expect single insertion to take place in $\mathcal{O}(1)$ time, the function for inserting n elements, where n is the size of dictionary in 10K, must be of the form: $f(n) = m * n + c$ which is $\mathcal{O}(n)$.
Note: Java produces a constant overhead run time. We obtain this by calculating $f(0)$. This value was subtracted from each time data and the final result was recorded.

Results The results are shown in Figure 1. The results were plotted using `Matplotlib (python)`. The best fit line for parameters m and c of $f(n)$ were calculated using `Matplotlib`'s `polyfit` functionality.
It is evident from the graphs that the insertion time follows a linear relationship with the size of the dictionary. This claim, is further supported by our approximated function for time of insertion of n elements (in 10K) $f(n) = m * n + c$ which is $\mathcal{O}(n)$. The values for m and c in $f(n)$ have been evaluated to be approximately equal to 0.226420 and 0.244620.Therefore, for a single insertion, the amortized complexity is $\mathcal{O}(1)$.
This proves our hypothesis that the average insertion time of a single element of hashset is indeed $\mathcal{O}(1)$.

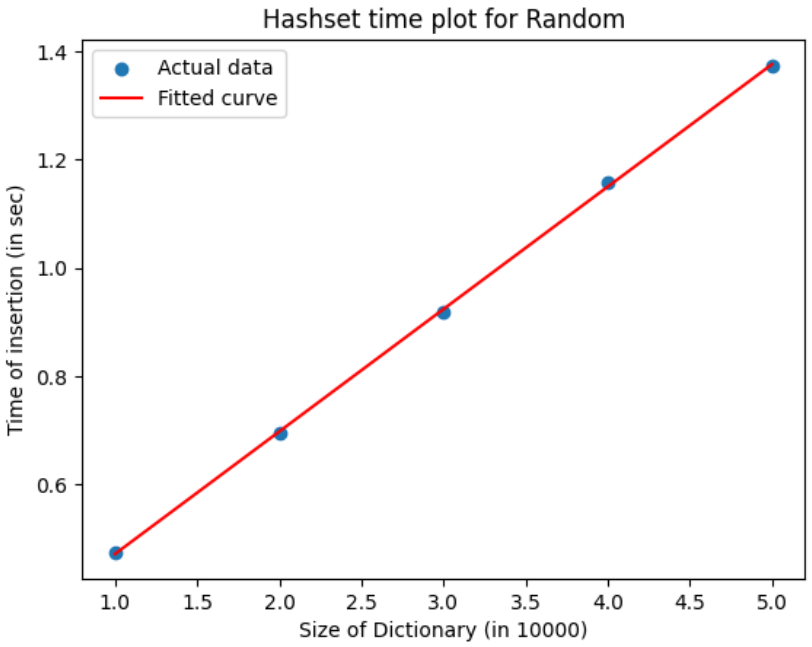


Figure 1: Plot of hashset in average case

2 Experiment 2

Hypothesis The average time complexity of a single insert into my implementation of binary tree is $\mathcal{O}(log(n))$, where n is the number of inputs.
Theoretically, the average time complexity of a single insert in a binary search tree is $\mathcal{O}(h)$, where h is the height of tree. Time complexity of finding h is $\mathcal{O}(log(N))$, where N is the number of nodes of the tree. In the average case, our tree is reasonably balanced. In this case, the number of nodes is almost equal to the number of inputs. So time complexity of $h \approx \mathcal{O}(log(n))$ where n is the number of inputs. Thus we expect the time complexity of single insert to be $\mathcal{O}(log(n))$.

Experimental Design To investigate the performance of our binary search tree in average case, we consider a **Random** batch of input dictionaries whose elements are in random order.
5 dictionaries of size 10K, 20K, 30K, 40K and 50K were generated for this experiment. All files were generated using `random_strings.py`. Length of each entry in each dictionary was fixed at 40. This was done to match the average entry size of `henry` dataset.
The **independent variable** is the **size** of dictionary and the **dependent variable** is the **time** taken to insert all elements of the dictionary. The query file has been kept empty for all experiments. This was done to avoid calculating the time for searching the elements inside the binary search tree because we are only interested in insertion times.
The program `speller_bstree.java` was run on each dictionary 5 times with the empty query file, and the average of the 5 values was recorded.
The time of insertion was measured using the UNIX `time` command by summing the `sys` and `usr` values output by the command. **Note**: this method produces the total time of inserting the entire dictionary. Since we expect each insertion to take place in either $\mathcal{O}(log(n))$, the function $f(n)$ for the time of insertion of n (in 10K) elements must be of the form $f(n) = a * n * log(n) + b$ which is $\mathcal{O}(n * log(n))$.
Note: Java produces a constant overhead run time. We obtain this by calculating f(0). This value was subtracted from each time data and the final result was recorded.

Results The results have been illustrated in Figure 2. The results were plotted using `Python Matplotlib` library. The approximate values for the parameters of $f(n)$ were calculated using `curve_fit` functionality of `SciPy` library.

For the Random batch, the total insertion time function is approximately $f(n) = v_1 * n * log(n) + v_2$ which is $\mathcal{O}(n * log(n))$. Values for v_1 and v_2 have been approximated as 0.065187 and 0.470512 respectively. This aligns with Figure 2 affirming that $f(n)$ is a good fit to the data. Therefore, the amortized time complexity for inserting a single element is $\mathcal{O}(log(n))$ thereby, confirming our

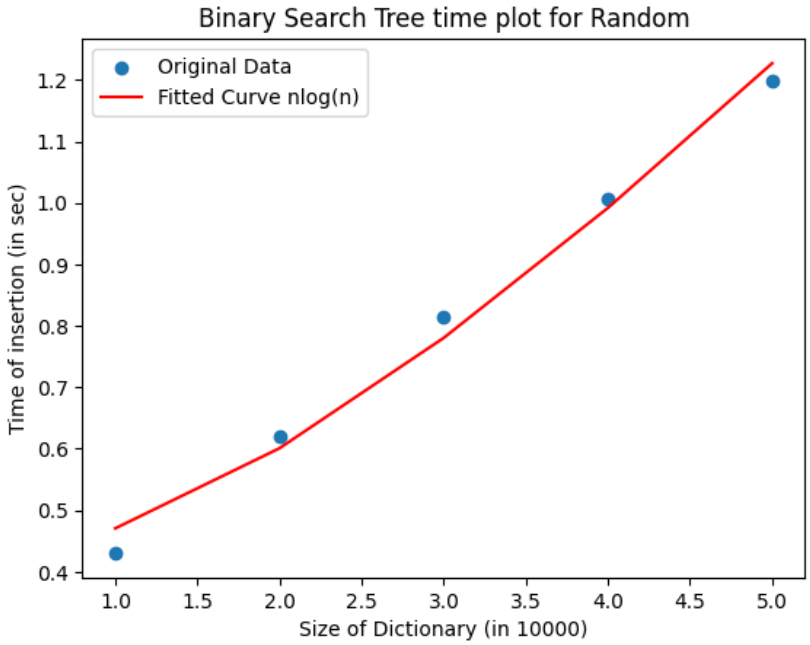


Figure 2: Plot of binary search tree in average case

hypothesis.
Note: Due to limited quantity of data points, the plot is almost linear. This might not be the case when the experiment is performed with more data.

3 Conclusion

We expect binary search tree to become slower than a hashset for sufficiently large n, where n is the size of dictionary in 10K. This is because, $\mathcal{O}(nlog(n))$ grows faster than $\mathcal{O}(n)$.
Graphically solving for the intersection point, we get $n \in \{0.9924, 8.3550\}$. Here n is the size of dictionary in 10K. This implies that for a random dictionary of size $n > 8.3550$ (i.e. 83,550) or $n < 0.9924$ (i.e. 9,924) hashset will be faster than a binary search tree for inserting elements. For $0.9924 < n < 8.3550$, hashset will be slower than a binary search for insertion. This agrees with our observations with regards to `henry`, a dataset of size $>$ 250K. Figure 3 illustrates this phenomena. **Note**: this result might not be accurate because we are extrapolating functions developed by limited number of data points. Moreover, the extent of randomness of dictionary is an important factor. Varying this factor in multiple larger dictionaries can lead to better results (see Appendix E).
The intersection points were solved using `fsolve` functionality of `SciPy`. Initial

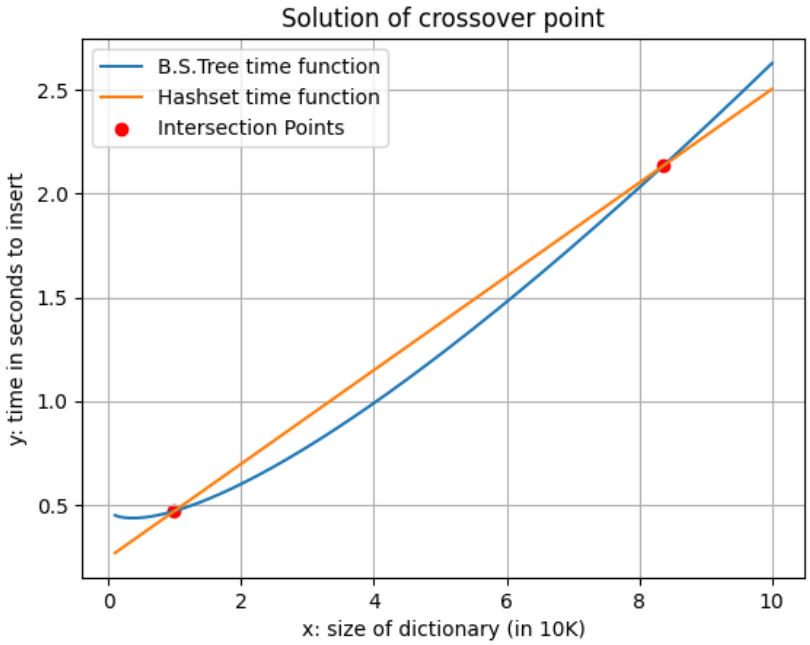


Figure 3: The crossover points for a randomly ordered dictionary

guesses were made according to the plots.

4 Data Statement

The raw data for Experiment 1 can be found in Appendix A. The process of generating dictionaries, computing the time and plotting graphs for this experiment was manually performed, details of which are in Appendix B. These can also be found in the `comp26120_2023` gitlab repository in the `lab3 directory`. The raw data for Experiment 2 (and Appendix E) can be found in Appendix C. The process of generating dictionaries, computing the time and plotting graphs for this experiment was manually performed, details of which are in Appendix D. These can also be found in the `comp26120_2023` gitlab repository in the `lab3 directory`.

A Raw Data for Experiment 1

Random Batch	
Size	Time (s)
10000	0.487
10000	0.47
10000	0.472
10000	0.47
10000	0.472
20000	0.694
20000	0.675
20000	0.707
20000	0.705
20000	0.69
30000	0.91
30000	0.893
30000	0.925
30000	0.924
30000	0.939
40000	1.176
40000	1.143
40000	1.143
40000	1.172
40000	1.16
50000	1.379
50000	1.367
50000	1.395
50000	1.368
50000	1.361

B Scripts for Experiment 1

B.1 Generating Dictionaries

All dictionaries were generated using `random_strings.py` by modifying the length of strings to 40 and size as 10K, 20K, 30K, 40K and 50K and using the following command:

```
$ python random_strings.py > [file name]
```

The empty query file `x` was generated as:

```
$ echo "x" > x
```

Then `x` was emptied.

B.2 Script for Computing Run Times

Following line of code was used to measure time:

```
$ time java comp26120.speller_hashset -d
../data/[custom data folder name]/[dictionary name] -m 0 -vv x
```

where [custom data folder name] is the name of the folder that contained all relevant dictionaries.

C Raw Data for Experiment 2

Random Batch	
Size	Time (s)
10000	0.425
10000	0.423
10000	0.44
10000	0.426
10000	0.439
20000	0.613
20000	0.632
20000	0.613
20000	0.627
20000	0.611
30000	0.816
30000	0.817
30000	0.834
30000	0.801
30000	0.804
40000	1.005
40000	0.991
40000	1.003
40000	1.019
40000	1.018
50000	1.204
50000	1.193
50000	1.207
50000	1.189
50000	1.206

D Scripts for Experiment 2

D.1 Generating Dictionaries

All dictionaries were generated using `random_strings.py` by modifying the length of strings to 40 and size as 10K, 20K, 30K, 40K and 50K and using

the following command:

```
$ python random_strings.py > [file name]
```

The empty query file `x` was generated as:

```
$ echo "x" > x
```

Then `x` was emptied.

D.2 Script for Computing Run Times

Following line of code was used to measure time:

```
$ time java comp26120.speller_bstree -d
../data/[custom data folder name]/[dictionary name] -m 0 -vv x
```

where [custom data folder name] is the name of the folder that contained all relevant dictionaries.

E Experiment 3 (Extension)

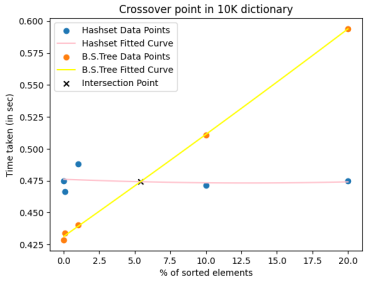
Hypothesis The **minimum** percentage of sorted elements in a dictionary which turns a binary tree slower than a hashset, depends on the size n (here, in 10K) of the dictionary. More generally, it follows a strictly decreasing function of the form $f(n) = a * (1/n) + b$.
Theoretically, binary search tree becomes unbalanced when the dictionary is sorted or reverse sorted. This makes it slower than a hashset in such situations. Moreover, the insertion time in a larger dictionaries containing sorted elements is slower than that of a smaller dictionary containing same quantity of sorted elements. Therefore, there must be an inverse relationship between the size of dictionary and the minimum percentage of sorted elements which slows down the tree.

Experimental Design To test the hypothesis we investigated the performance of both hashset and binary search tree on random input dictionaries with varying percentage of sorted words. **Note**: only sorted case has been considered. The case for reverse sort is symmetrical and hence similar. We have 2 **independent** variables: the **percentage of sorted words** in the dictionary and the **size** of dictionary (in 10K).
We fix the dictionary size and vary the percentage of sorted elements. The crossover percentage for a particular size, when a binary tree is slower than hashset, is calculated graphically. Each such data is further used to generate our function $f(n)$. This process is repeated for all dictionary sizes.
Our **dependent** variable was the **time** taken to insert the dictionary.
5 dictionaries of sizes 10K, 20K, 30K, 40K and 50K were generated. For each

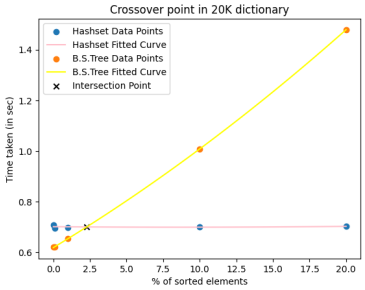
dictionary, 5 copies were created but each copy had a different percentage of sorted elements viz. 0.01, 0.1, 1.0, 10.0, 20.0. All dictionaries were created using `AlmostSort.java`. **Note**: `AlmostSort.java` creates a dictionary of a given size and runs bubble sort on the elements for a given number of times. It can be observed that running the bubble sort k times for $k > 0$, leads to the last k elements being sorted completely. This also turns out to be longest sequence of sorted elements in the dictionary. We consider this sequence of k elements in calculating the percentage of sorted elements.
The query file was kept empty to ignore the time of searching from our calculations. The `speller_hashset.java` and `speller_bstree.java` program was run on each type of dictionary 5 times. The average of these 5 values was recorded.
Time was measured using the UNIX `time` command summing the `user` and `sys` values output by the command. **Note**: Java produces a constant overhead run time. We obtain this by calculating f(0). This value was subtracted from each time data and the final result was recorded.

Results Results of the experiments have been displayed in Figure 4. The results were plotted using `Python Matplotlib` library. The intersection points were solved using `fsolve` functionality of `SciPy`. Initial guesses were made according to the plots. The approximate values for the parameters of $f(n)$ were calculated using `curve_fit` functionality of `SciPy`. Our function is approximately $f(n) = v_1 * (1/n) + v_2$, where $v_1 = 6.051321$ and $v_2 = -0.694567$. Clearly this is a good fit to the data as evidenced in Figure 5.
Note: This proves that there is a monotonically decreasing relationship between the size of a dictionary and the percentage of sorted elements in it which trigger a slow-down effect in binary search tree. **However**, this does NOT yet confirm whether the actual relationship is **uniquely** of the form $f(n)$ as shown earlier. More quantity of data, more gradual variation in the percentages of sorted words must be considered for improvements.

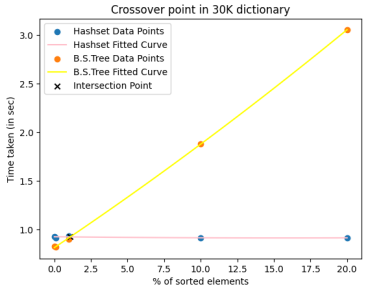
Discussion As can be seen from Figure 5, if any dictionary of size $\geq$ 10K has percentage of sorted words $p \geq 5.5$, then the binary search tree on that dictionary will be slower than a hashset.



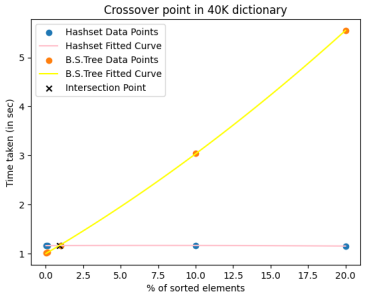
(a) Crossover % in 10K at 5.408



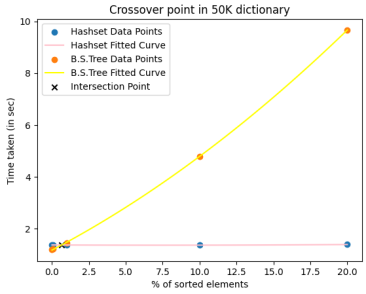
(b) Crossover % in 20K at 2.285



(c) Crossover % in 30K at 1.090



(d) Crossover % in 40K at 0.899



(e) Crossover % in 50K at 0.662

Figure 4: Crossover % in various dictionaries

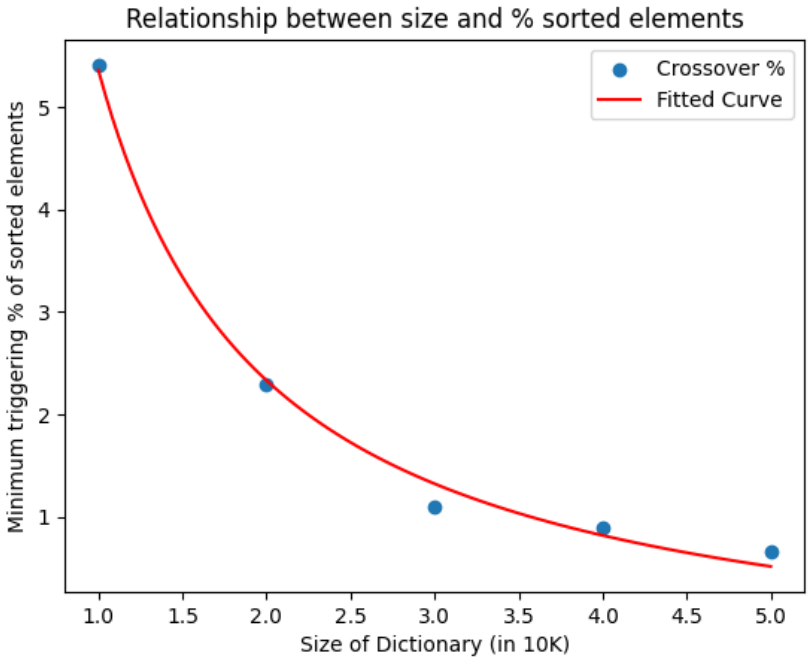


Figure 5: $f(n) = 6.051321 * (1/n) - 0.694567$

F Raw Data for Appendix E

10K Hashset		10K BS Tree	
%Sorted	Time (s)	%Sorted	Time (s)
0.01	0.46	0.01	0.425
0.01	0.486	0.01	0.428
0.01	0.471	0.01	0.425
0.01	0.486	0.01	0.424
0.01	0.471	0.01	0.441
0.1	0.472	0.1	0.44
0.1	0.46	0.1	0.441
0.1	0.458	0.1	0.439
0.1	0.473	0.1	0.425
0.1	0.47	0.1	0.425
1	0.486	1	0.441
1	0.486	1	0.44
1	0.505	1	0.439
1	0.489	1	0.442
1	0.475	1	0.44
10	0.471	10	0.504
10	0.486	10	0.492
10	0.471	10	0.52
10	0.472	10	0.519
10	0.456	10	0.52
20	0.471	20	0.586
20	0.488	20	0.598
20	0.471	20	0.595
20	0.471	20	0.596
20	0.472	20	0.595

20K Hashset		20K BS Tree	
%Sorted	Time (s)	%Sorted	Time (s)
0.01	0.706	0.01	0.628
0.01	0.705	0.01	0.628
0.01	0.721	0.01	0.611
0.01	0.705	0.01	0.626
0.01	0.706	0.01	0.611
0.1	0.691	0.1	0.614
0.1	0.689	0.1	0.626
0.1	0.704	0.1	0.626
0.1	0.691	0.1	0.626
0.1	0.703	0.1	0.61
1	0.705	1	0.643
1	0.705	1	0.643
1	0.69	1	0.666
1	0.707	1	0.66
1	0.689	1	0.658
10	0.708	10	1.008
10	0.691	10	1.019
10	0.689	10	1.009
10	0.704	10	0.987
10	0.708	10	1.018
20	0.707	20	1.474
20	0.705	20	1.523
20	0.708	20	1.465
20	0.704	20	1.473
20	0.689	20	1.459

30K Hashset		30K BS Tree	
%Sorted	Time (s)	%Sorted	Time (s)
0.01	0.926	0.01	0.814
0.01	0.929	0.01	0.816
0.01	0.942	0.01	0.834
0.01	0.926	0.01	0.832
0.01	0.925	0.01	0.837
0.1	0.909	0.1	0.817
0.1	0.943	0.1	0.817
0.1	0.914	0.1	0.83
0.1	0.925	0.1	0.831
0.1	0.894	0.1	0.834
1	0.939	1	0.926
1	0.925	1	0.909
1	0.942	1	0.894
1	0.94	1	0.897
1	0.93	1	0.895
10	0.924	10	1.879
10	0.909	10	1.916
10	0.91	10	1.891
10	0.926	10	1.855
10	0.907	10	1.862
20	0.927	20	3.055
20	0.894	20	3.031
20	0.912	20	3.024
20	0.927	20	3.1
20	0.923	20	3.054

40K Hashset		40K BS Tree	
%Sorted	Time (s)	%Sorted	Time (s)
0.01	1.129	0.01	1.02
0.01	1.161	0.01	1.005
0.01	1.175	0.01	0.969
0.01	1.176	0.01	1.018
0.01	1.146	0.01	1.018
0.1	1.16	0.1	1.033
0.1	1.176	0.1	1.035
0.1	1.173	0.1	1.05
0.1	1.159	0.1	0.997
0.1	1.145	0.1	1.003
1	1.162	1	1.12
1	1.159	1	1.204
1	1.144	1	1.145
1	1.175	1	1.161
1	1.16	1	1.174
10	1.127	10	3.052
10	1.177	10	3.037
10	1.16	10	2.959
10	1.176	10	3.152
10	1.175	10	2.991
20	1.129	20	5.452
20	1.144	20	5.434
20	1.159	20	5.443
20	1.142	20	5.803
20	1.177	20	5.628

50K Hashset		50K BS Tree	
%Sorted	Time (s)	%Sorted	Time (s)
0.01	1.379	0.01	1.174
0.01	1.362	0.01	1.22
0.01	1.379	0.01	1.182
0.01	1.331	0.01	1.193
0.01	1.381	0.01	1.195
0.1	1.364	0.1	1.222
0.1	1.379	0.1	1.204
0.1	1.363	0.1	1.221
0.1	1.378	0.1	1.24
0.1	1.39	0.1	1.192
1	1.392	1	1.413
1	1.387	1	1.445
1	1.402	1	1.444
1	1.387	1	1.445
1	1.351	1	1.474
10	1.379	10	4.76
10	1.331	10	4.647
10	1.378	10	5.109
10	1.345	10	4.646
10	1.396	10	4.819
20	1.38	20	9.425
20	1.394	20	10.06
20	1.41	20	9.439
20	1.393	20	9.505
20	1.378	20	9.878