DS-221 Introduction to Scalable Systems
Assignment - 2B

Submitted by - Ocima Kamboj
Serial Number - 06-02-01-10-51-18-1-15899

Searching

Contents

1 Machine Specifications

The following table gives the specifications of the machine which was used to analyze the code. This was found out using the *lscpu* command.

Architecture:	x86_64
CPU op-mode(s):	32-bit, 64-bit
Byte Order:	Little Endian
CPU(s):	8
On-line CPU(s) list:	0-7
Thread(s) per core:	2
Core(s) per socket:	4
Socket(s):	1
NUMA node(s):	1
Vendor ID:	AuthenticAMD
CPU family:	21
Model:	2
Model name:	AMD Opteron(tm) Processor 3380
Stepping:	0
CPU MHz:	1400.000
CPU max MHz:	2600.0000
CPU min MHz:	1400.0000
BogoMIPS:	5200.29
Virtualization:	AMD-V
L1d cache:	16K
L1i cache:	64K
L2 cache:	2048K
L3 cache:	8192K
NUMA node0 CPU(s):	0-7

2 Hash Table Analysis

2.1 Description

Let the input size be n. Let the capacity of our hashtable be C. Thus, we form an array of size C. Each position in the array points to a list. This list contains the $< key, value >$ pairs that get mapped to the same bucket, or the same array index. To map the key to the array index, the hash function that was used is $key(mod)C$.
Thus, when an insertion has to take place, we find the array location at which the $< key, value >$ pair has to be inserted, we go to that location, and add this pair to the end of the list.
When a lookup is called, we find the array index that can contain the particular key with the help of hash function, we go to that array location, and linearly scan through the list to find the required $< key, value >$ pair.

2.2 Time Complexity Analysis - Insertion

2.2.1 Analytical Time Complexity

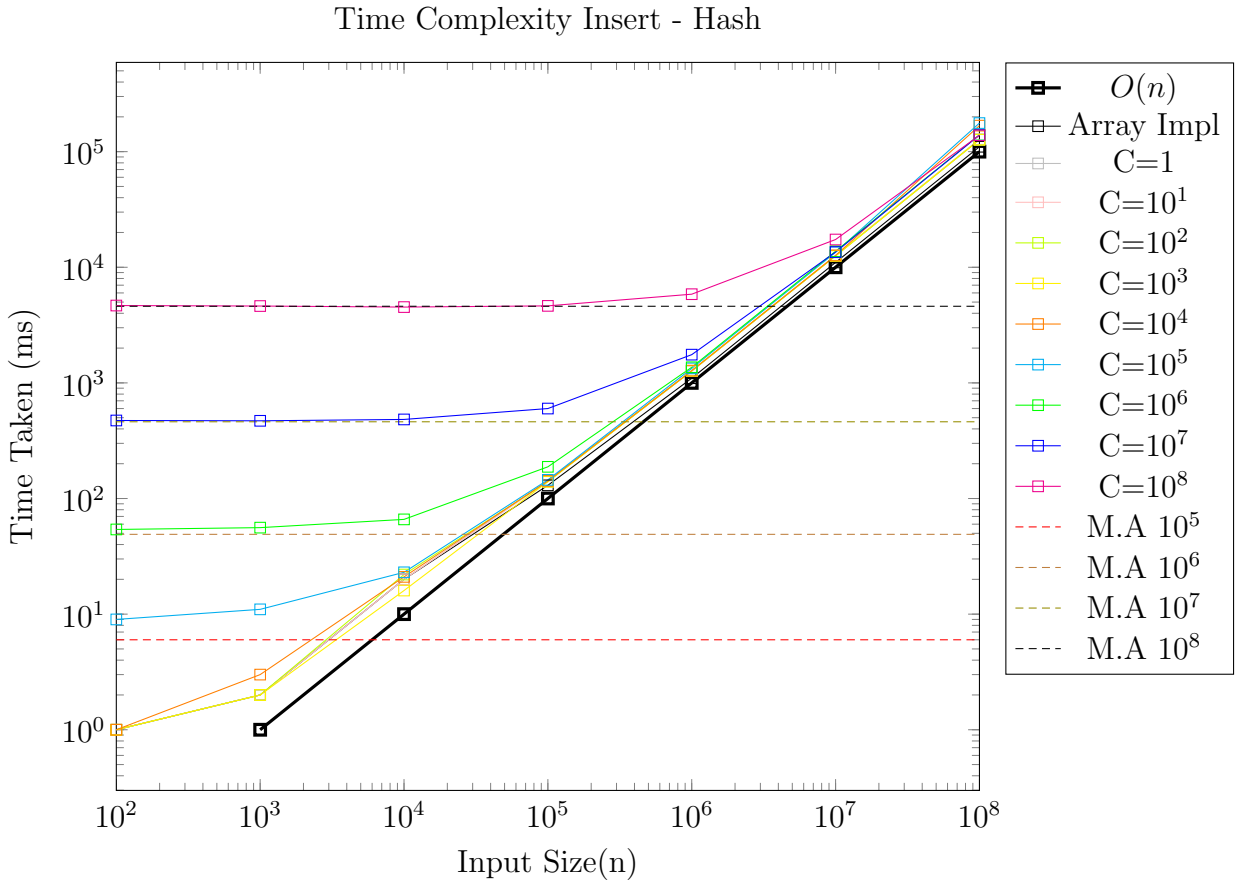
Whenever an insertion is called for a $< key, value >$ pair, we first find the respective array location through the hash function. This takes $O(1)$ time. We then go that array location, and insert the pair at the end of the list, which again takes $O(1)$ time. Thus, the overall time take for a single insertion is $O(1)$. Therefore, to insert n pairs, the time taken would be $O(n)$.
According to this analysis, the insertion time should stay constant for a particular input size no matter the hashtable capacity.

2.2.2 Empirical Time Complexity

The insertion times were observed for input sizes 10^2, 10^3, 10^4, 10^5, 10^6, 10^7, and 10^8 for hashtable capacities 1, 10, 10^2, 10^3, 10^4, 10^5, 10^6, 10^7, and 10^8 each. As insertion is $O(n)$, if we plot a graph of time taken vs. the input size, the expectation is we should see a linear plot no matter the capacity. Thus, if we plot this graph for different capacities, all the lines corresponding to different capacities should overlap with each other and with $O(n)$ line.
When capacity is 1, we should get the same performance as when we implement the dictionary using an unsorted array.
The observations are given in the table below, and then plotted in a graph.

M.A. = Memory Allocation Time								
Capacity	M.A	n=10^2	n=10^3	n=10^4	n=10^5	n=10^6	n=10^7	n=10^8
Array Impl		1	2	20	130	1098	11018	110250
1	1	1	2	20	142	1263	12649	128122
10	1	1	2	20	140	1268	12645	126859
10^2	1	1	2	22	140	1270	12522	126623
10^3	1	1	2	16	140	1272	12584	127010
10^4	1	1	3	21	142	1277	12648	167886
10^5	6	9	11	23	145	1334	13483	176680
10^6	49	54	56	66	188	1365	13765	139798
10^7	461	473	470	483	600	1755	13542	139100
10^8	4600	4667	4621	4527	4636	5851	17382	138438

Table 1: Running Time in milliseconds for Insertion



Observation from the Graph - We see that as the capacity is increasing, the time required for insertion for a particular input size n is also increasing. For smaller capacities, the observed trend lies very close to the $O(n)$ line and thus satisfies the analytical time complexity. For bigger capacities, we see that we are taking alot of time to insert even for small input sizes, which is unexpected.

Explanation of the Observed Trend - When the first insertion takes place, we first form

an array of capacity C. Thus, the compiler has to allocate contiguous memory for this purpose. Bigger the capacity, larger the memory allocation that needs to be done. This takes up time. Thus, when hashtables with bigger capacities are being constructed, we spend some amount of time at the beginning for this memory allocation, which results in unexpectedly high times for even small input sizes.
To confirm this hypothesis, the time was noted for just the memory allocation that takes place at the beginning of the insertion. These have been plotted in the graph. This corresponds to the time of just the initialisation of the hashtable, and not any file read time or insertion time.
We note that this initialization indeed takes alot of time, resulting in a high time taken even for small input sizes.

2.3 Time Complexity Analysis - Lookup

2.3.1 Analytical Time Complexity

Whenever a lookup is called for a $< key, value >$ pair, we first find the respective array location at which the given key might be present through the hash function. This takes $O(1)$ time. We then go that array location, and scan through the list present at that location to find the respective key. If the keys are uniformly distributed in the array locations, which means each array location has an average of n/C $< key, value >$ pairs in its list, then the time taken for lookup would be $O(n/C)$. Therefore, the overall time complexity for one lookup is $O(n/C)$.

2.3.2 Empirical Time Complexity

The lookup times were observed for input sizes 10^2, 10^3, 10^4, 10^5, 10^6, 10^7, and 10^8 for hashtable capacities 1, 10, 10^2, 10^3, 10^4, 10^5, 10^6, 10^7, and 10^8 each. The number of lookups done in each case were 10 to keep the results comparable, as the average time taken for one lookup will be just $1/10^{th}$ of that.
We first need to know the average number of keys mapped to a given array location for each case.

	C=1	C=10	C=10^2	C=10^3	C=10^4	C=10^5	C=10^6	C=10^7	C=10^8
n=10^2	10^2	10	1.53846	1	1	1	1	1	1
n=10^3	10^3	10^2	10	1.49031	1	1	1	1	1
n=10^4	10^4	10^3	10^2	10	1.47995	1	1	1	1
n=10^5	10^5	10^4	10^3	10^2	10	1.48798	1	1	1
n=10^6	10^6	10^5	10^4	10^3	10^2	10.0002	1.48802	1	1
n=10^7	10^7	10^6	10^5	10^4	10^3	10^2	10.0001	1.48713	1
n=10^8	10^8	10^7	10^6	10^5	9999.99	999.999	99.9972	9.94015	3.41445

Table 2: Average number of keys mapped to one array index

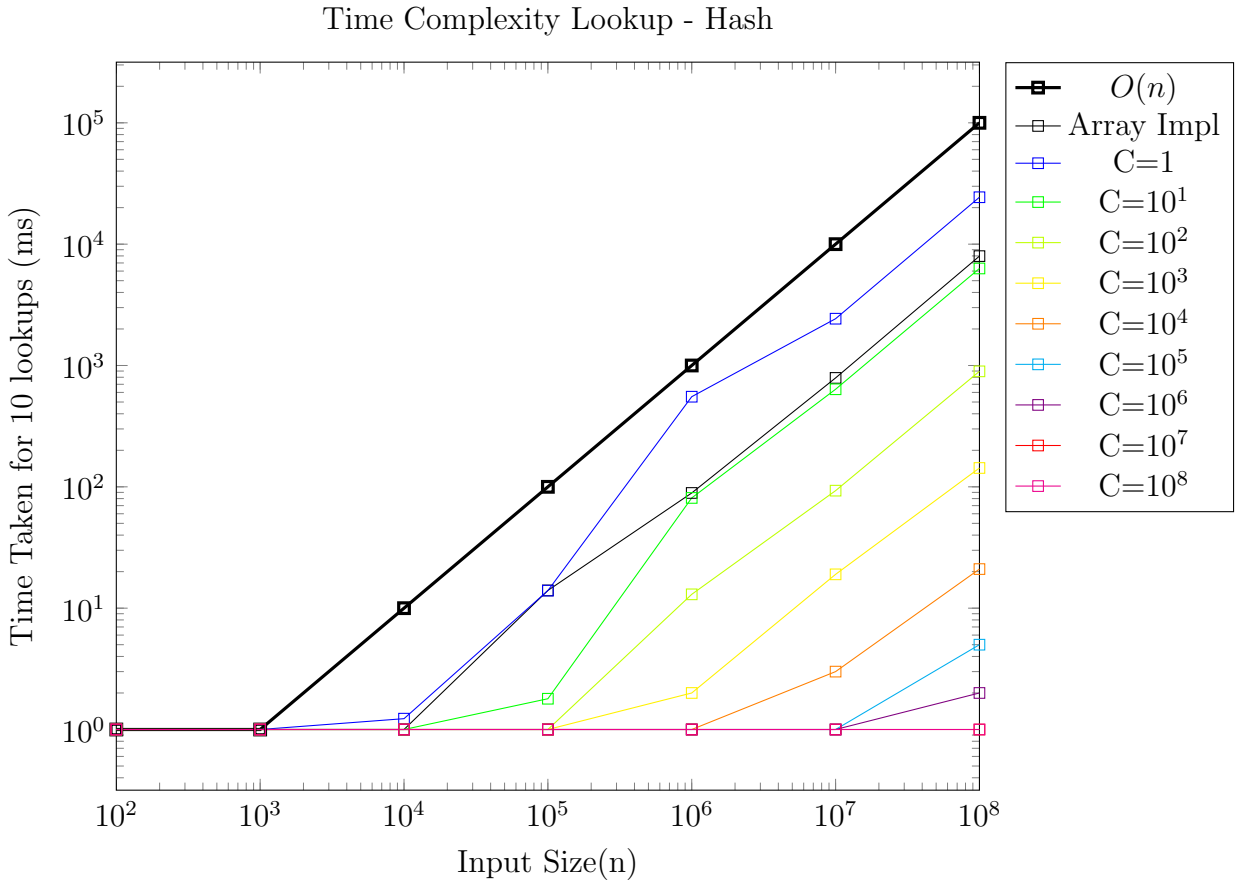
If the keys were uniformly distributed, then, the average number of keys mapped to one array index when $n = C$ would be one. From the table, we observe that this number is

around 1.5, and 3 when n=10^8. When $C > n$, the average number of keys mapping to one location is 1. When $C < n$, the average number of keys mapping to one location is around n/C.
Thus, the time taken for one lookup will be around $O(1)$ for $n \leq C$. For $n > C$, the time taken is $O(n/C)$. So, if we plot a graph of lookup time vs input size, we should see a constant line until $n = C$, and then we should see a linear curve afterwards.
The observations are in the table below.

Capacity	n=10^2	n=10^3	n=10^4	n=10^5	n=10^6	n=10^7	n=10^8
Array Impl	1	1	1	14	89	790	7971
1	1	1	1	14	553	2430	24362
10	1	1	1	2	81	637	6306
10^2	1	1	1	1	13	93	895
10^3	1	1	1	1	2	19	143
10^4	1	1	1	1	1	3	21
10^5	1	1	1	1	1	1	5
10^6	1	1	1	1	1	1	2
10^7	1	1	1	1	1	1	1
10^8	1	1	1	1	1	1	1

Table 3: Running Time in milliseconds for 10 lookups

The plot is shown below -



Observation from the graph - We see that the lookup time is increasing as the capacities are decreasing, which is expected. The expectation was that the lookup time would remain constant for $n \leq C$. The observations show that the lookup time is remaining constant for $n \leq 10^3 C$. For $n > 10^3 C$, the plot lines are parallel to $O(n)$, thus showing the linear time complexity.The Array Implementation seems to be performing better than the hash table of capacity one.

Explanation of the observed trend - The lookup times are better than expected as they are remaining constant for $n \leq 10^3 C$. Apart from that, the trend is as expected.

The array implementation is performing better than the hashtable of capacity one, whereas, one might expect that the performance should be same. This seems to be happening due to the extra computation we are doing in calculating hash function for each key.

3 Hash Table Implementation ($C = n$)

Based on the results for insertion and lookup in the previous section, we implement our hashtable with a capacity equal to the input size to get both fast insertion and lookups.

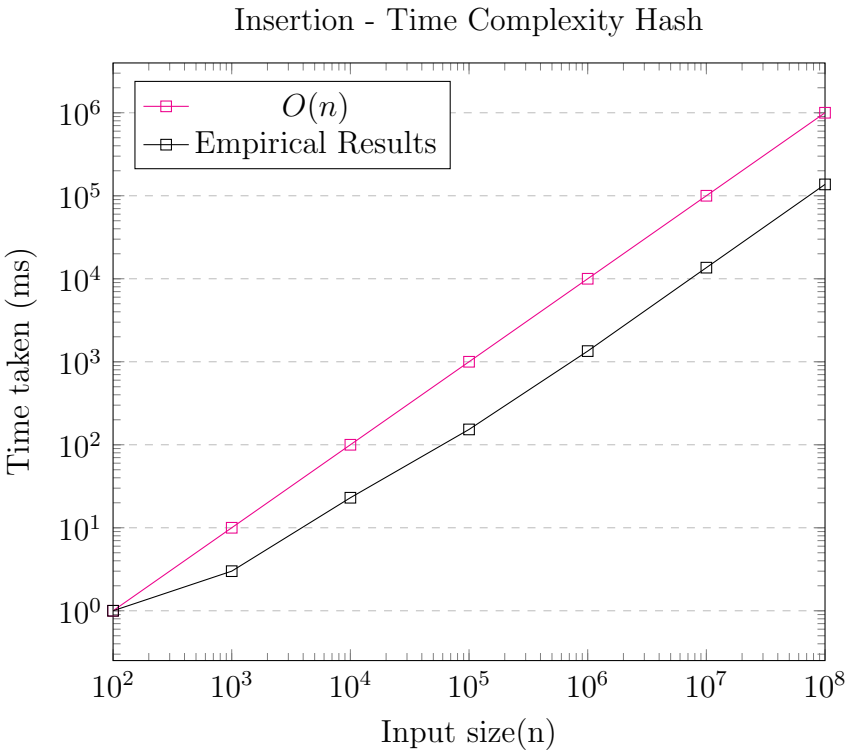
3.1 Insertion

3.1.1 Analytical Time Complexity

Based on the discussion in Section 2.2.1, the time for n insertions would be $O(n)$.

3.1.2 Empirical Time Complexity

The observations are in the following plot -



From this graph, we can conclude that the time complexity for n insertions is $O(n)$.

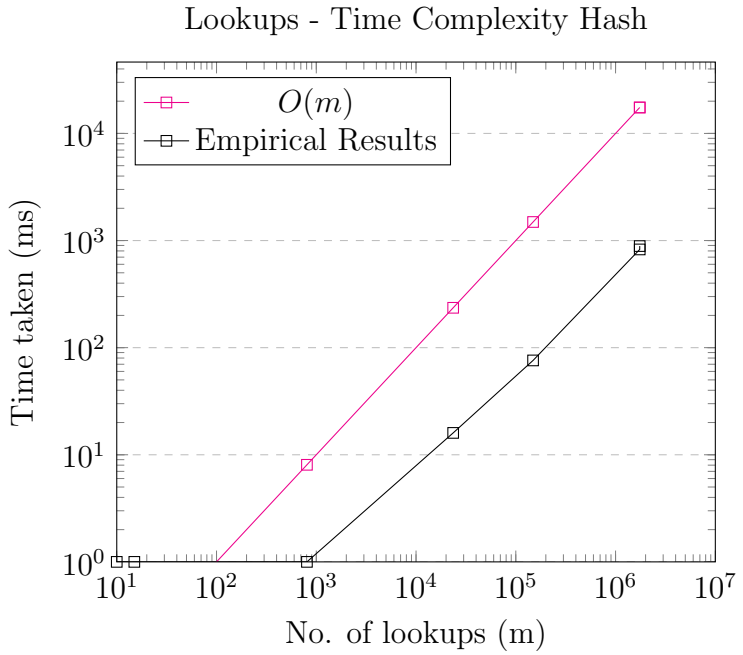
3.2 Lookup

3.2.1 Analytical Time Complexity

As we have chosen the capacity to be equal to the input size, based on the discussion in sections 2.3.1 and 2.3.2, the time for one lookup will be $O(1)$. Therefore, if we are performing m lookups, the overall time complexity would be $O(m)$.

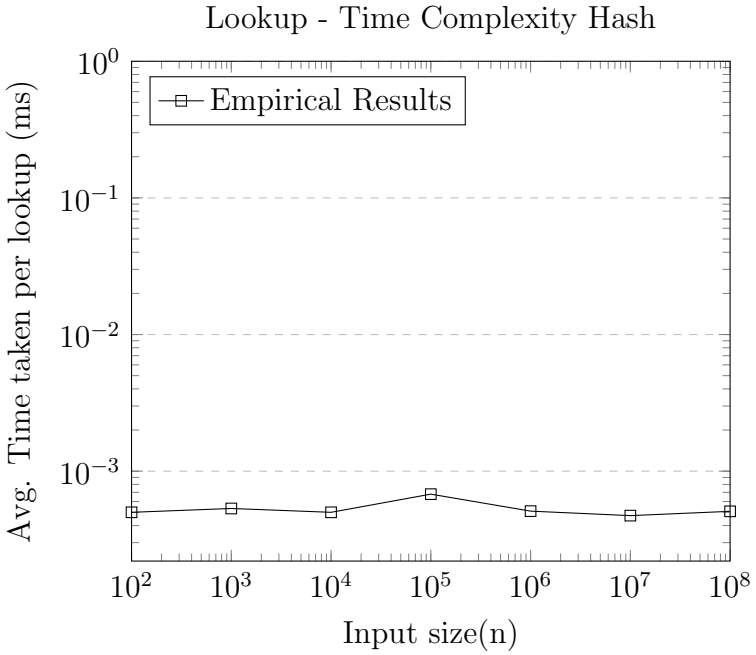
3.2.2 Empirical Time Complexity

The observations are in the following plot -



This satisfies our analysis of the lookup being $O(m)$.

We can also do the analysis for the average time taken per lookup. For this, we divide the observed time by the number of lookups performed. In this case, we should get $O(1)$ time.



This shows that the time taken per lookup is indeed $O(1)$.

4 Unsorted Array Implementation

4.1 Description

We form an array equal to the input size n. Whenever an insert is called, we go to the first unoccupied array location, and insert the $< key, value >$ pair there. We maintain a counter to keep a track of this index. For the lookup, we linearly scan through this array until we find the required key.

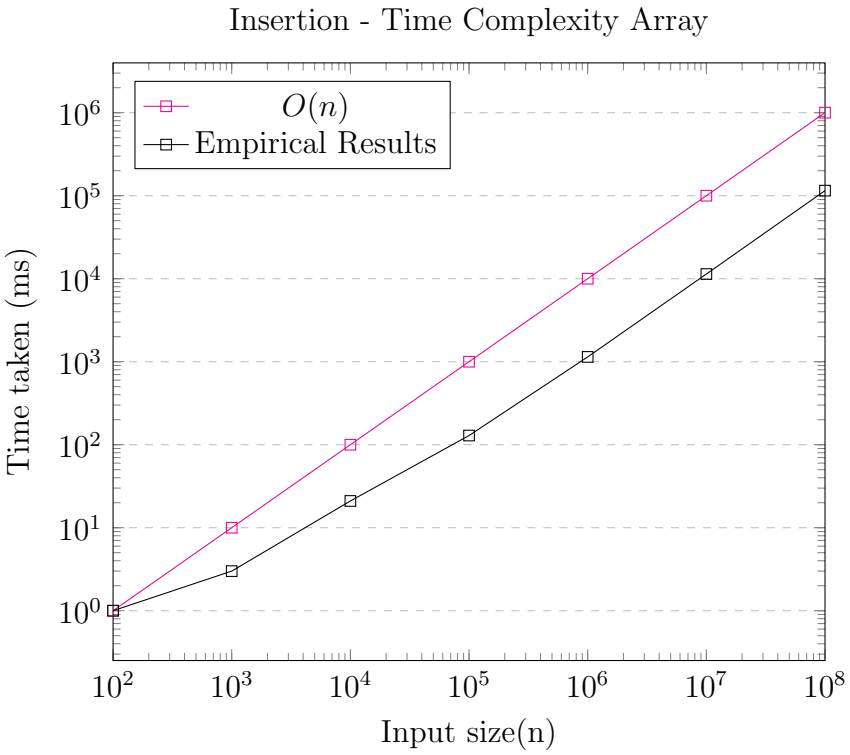
4.2 Insertion

4.2.1 Analytical Time Complexity

Whenever an insert is called, we just go the first unused location and insert the $< key, value >$ pair there. This should take $O(1)$ time. For performing n inserts, the time taken would be $O(n)$.

4.2.2 Empirical Time Complexity

The observations are plotted below -



From this graph, we can conclude that the time complexity for n insertions is $O(n)$.

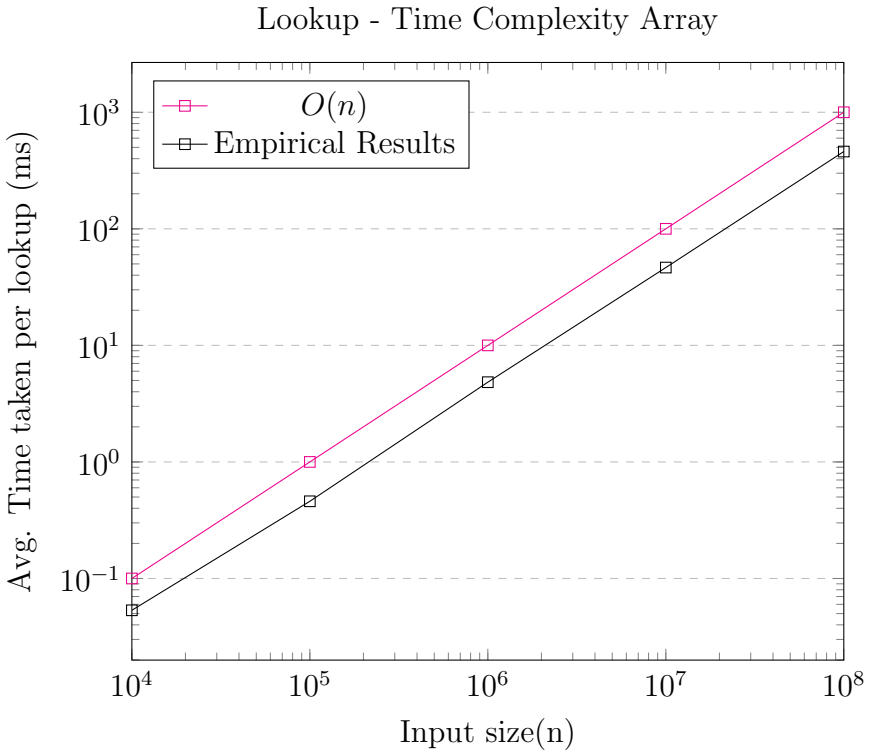
4.3 Lookup

4.3.1 Analytical Time Complexity

For one lookup, we have to linearly scan through the array to find the required key. This takes $O(n)$ amount of time. To perform m lookups, we will require $mO(n)$ or $O(mn)$ amount of time.

4.3.2 Empirical Time Complexity

For each size n, we observe the lookup time for some m no of lookups. We divide the observed time by this m to get the average time per lookup. This should be $O(n)$ as per the expectation.



From this graph, we can conclude that the time complexity for a single lookup is $O(n)$.

5 Sorted Array/BSearch Implementation

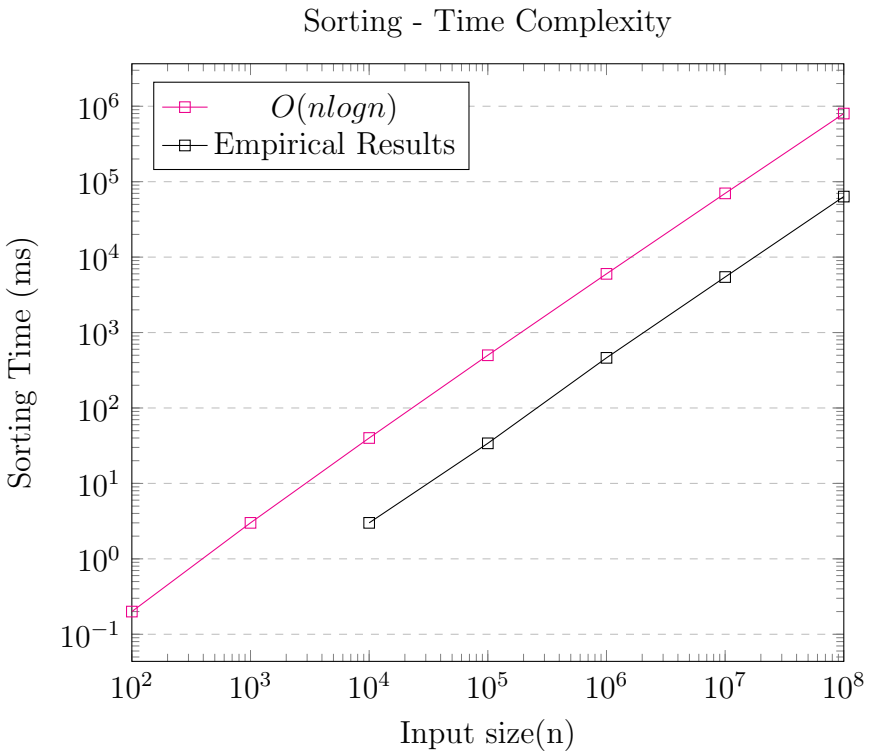
5.1 Description

The initialization and the inserts are done in the same way as the unsorted array. The difference is that when all the inserts have been done, we sort this array. Thus, when a lookup is called, we perform binary search on this sorted array for the required key.

5.2 Insertion

5.2.1 Analytical Time Complexity

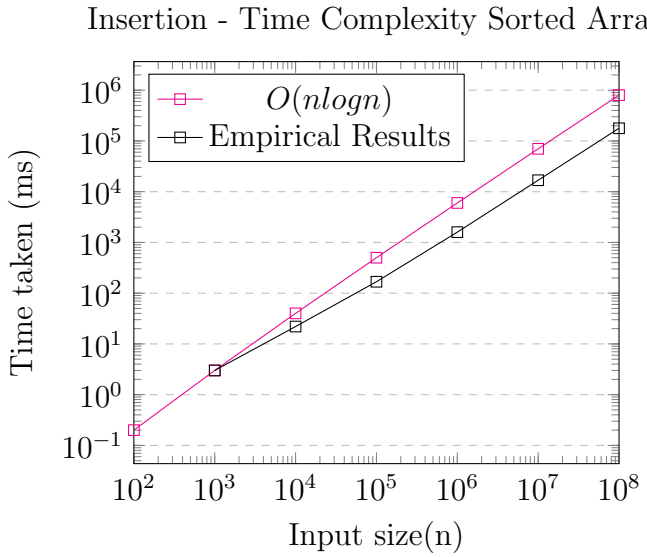
As in the unsorted array implementation, the time complexity for n inserts would be $O(n)$ before any sorting takes place. To find the overall time complexity, we need to take into account the time complexity of the sorting function. The function that was used was the C++ STL sort function, which is $O(nlogn)$. To verify this, the sorting time was observed for different array sizes.



From this graph, we can conclude that the time complexity for sorting is $O(nlogn)$. Therefore, the overall time complexity for insertion would be $O(nlogn)$.

5.2.2 Empirical Time Complexity

The observations are plotted below -



From this graph, we can conclude that the time complexity for n insertions is $O(nlogn)$.

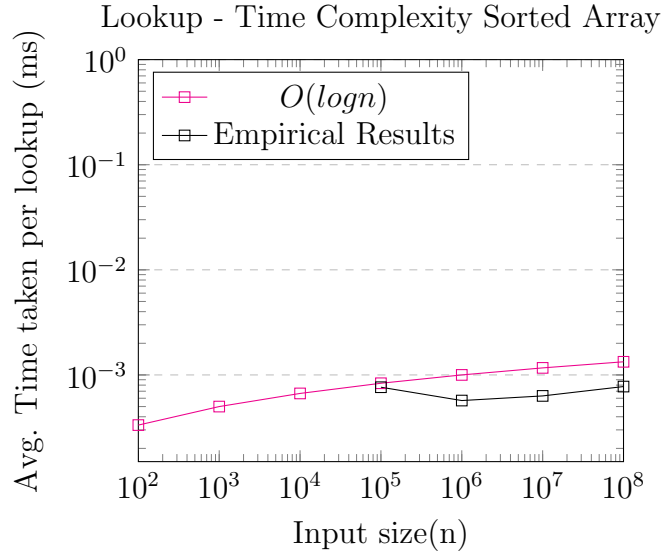
5.3 Lookup

5.3.1 Analytical Time Complexity

For one lookup, we perform binary search on the array to find the key. Thus, the time taken for one lookup would be $O(logn)$. To perform m lookups, the time taken would be $mO(logn)$ or $O(mlogn)$.

5.3.2 Empirical Time Complexity

As in the case of unsorted array, we do our analysis with the average time per lookup, which should be $O(logn)$.



From this graph, we can conclude that the time complexity for a single lookup is $O(logn)$.

6 Comparison of the three Implementations

6.1 Insertion

6.1.1 Analytical Analysis

Based on the discussion in sections 3, 4, and 5, the time taken for n inserts in all the three cases would be $O(n)$.

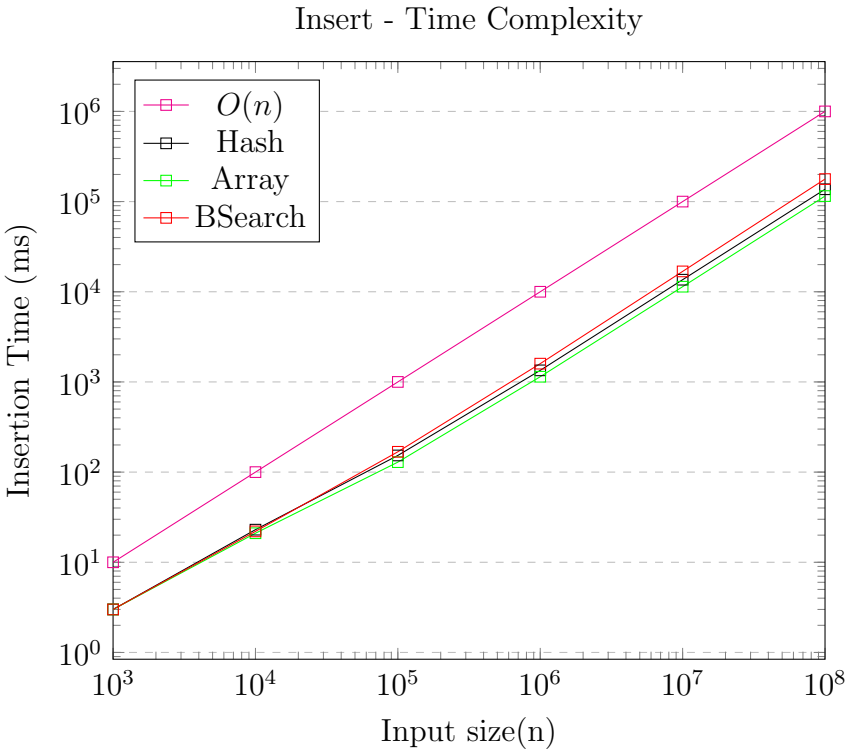
6.1.2 Empirical Analysis

The time taken for performing n inserts for the three different implementations are given in the following table.

Insertion Time in milliseconds			
Input Size(n)	Hash	Array	Bsearch
10^2	0	0	0
10^3	3	3	3
10^4	23	21	22
10^5	153	129	168
10^6	1344	1142	1595
10^7	13604	11382	16817
10^8	136994	115109	177706

Table 4: Time taken for n insert operations

Although the times for all three are $O(n)$ and lie very close to each other, we observe that the insertion times are Array<Hash<BSearch.
BSearch is taking the largest time due to the sorting that is done after all the inserts have taken place.
Among Array and Hash Table Implementation, the extra computation done by the hash function during each insert seems to have increased the time taken as compared to the Array Implementation.
This observations have also been plotted in the graph below for a clearer picture.



6.2 Lookup

6.2.1 Analytical Analysis

Based on the discussion in sections 3, 4, and 5, the time taken for a single lookup would be $O(1)$ for Hash Table Implementation, $O(n)$ for Unsorted Array Implementation, and $O(logn)$ for BSearch Implementation.

6.2.2 Empirical Analysis

The time taken for lookups is summarised in the following table.

Lookup Time in milliseconds				
Input Size(n)	No. of Lookups	Hash	Array	Bsearch
10^2	10	0.005	0.005	0.005
10^3	15	0.008	0.075	0.009
10^4	806	0.403	43	0.412
10^5	23559	16	10827	18
10^6	149004	76	720050	85
10^7	1746869	826	*23 hrs(est.)*	1103
10^8	1746869	888	*230 hrs(est.)*	1354

Table 5: Time taken for lookup operation

The time taken for array implementation for sizes 10^7 and 10^8 were estimated using the $O(n)$ trend. To find the average per lookup time for Array Implementation, smaller number

of lookups were done.

As expected, the Hash Table gives the best lookup time, followed by BSearch which is also giving good results upto $n = 10^8$. Array Implementation gives the worst performance by a huge margin.

The average per lookup time is plotted in the graph below.