

Time for inserting words in each dictionary against dictionary

size

From ChatGPT

- **dictionary.ContainsKey(key)**: Checking whether the dictionary contains a specific key also typically has constant time complexity on average, $O(1)$, assuming a good hash function and a low collision rate.
- Reading each line from the file (**sr.ReadLine()**) and inserting it into the dictionary (**word_DS.Insert(line)**) is executed for each line in the file.
- If the average number of lines in each file is denoted by m , then this operation will execute m times per file.
- Therefore, the time complexity of this loop is $O(m)$, where m is the number of lines in the file.

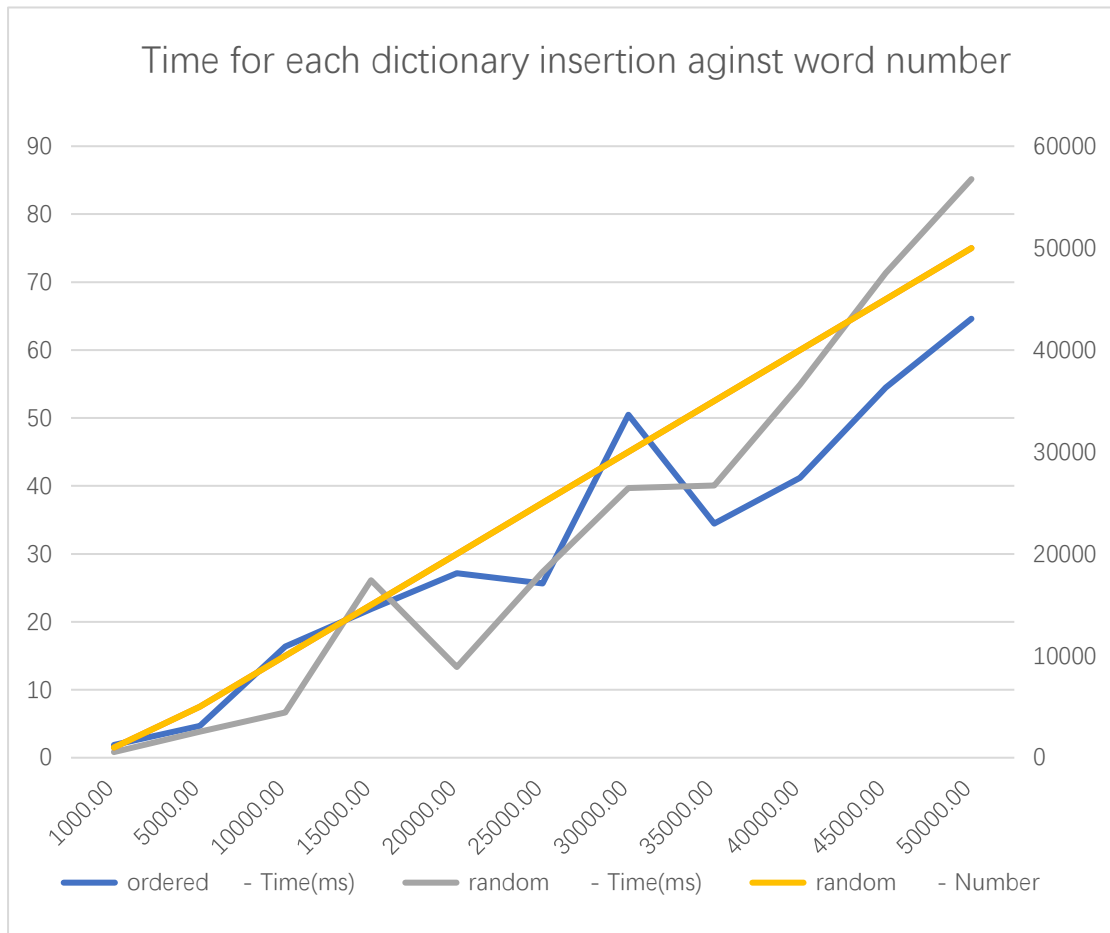
Therefore inside the Stopwatch start() and stop(), the time complexity is $O(n)$ for n is refer to how many lines in the text file.

Insert function defined in DictionaryDS class, where each operation get their time complexity $O(1)$ added up as $O(4) = O(1)$

Therefore, exact overall time complexity is $O(n)$

Chart for insertion

Yellow line stands for $T(n) = k \cdot n$, in this chart k is about 0.0015
blue and grey lines stand for order text file and random text file.



it is clearly showed the growth of either ordered text and random text roughly match the linear growth of $O(n)$.

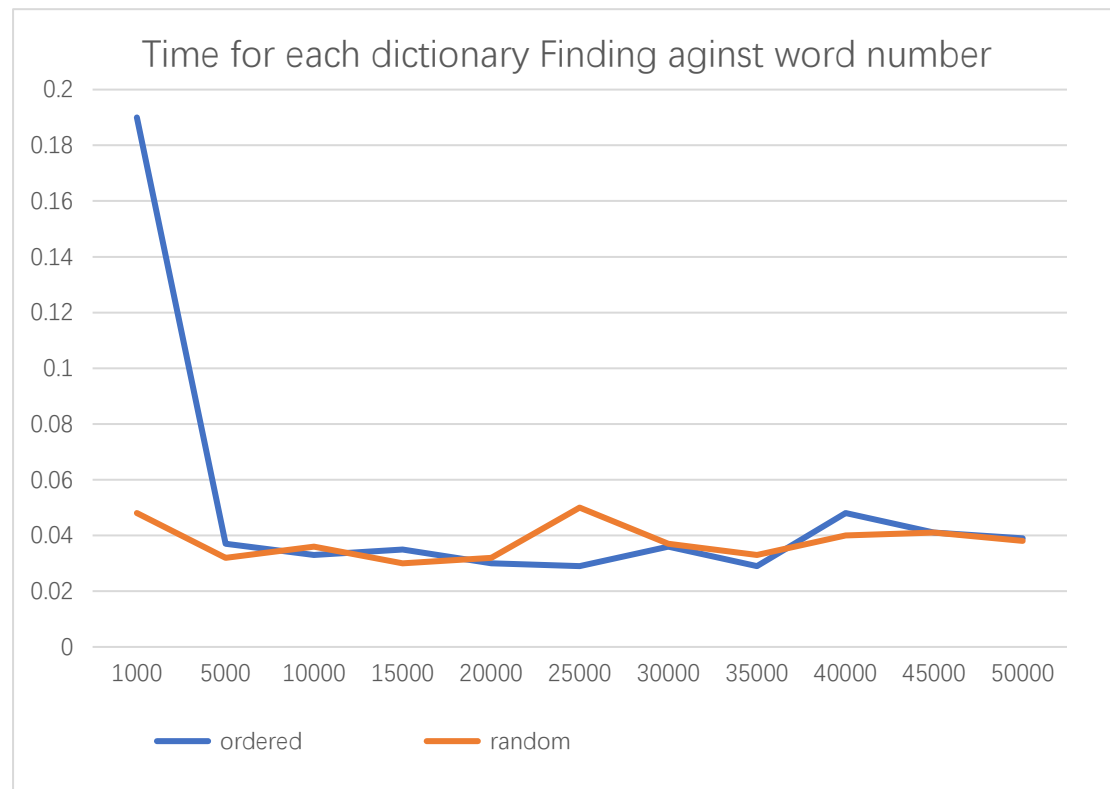
At some number of words the time record for ordered one is higher more than random one, some number on the contrary.

Time for finding certain word in each dictionary (in worst

occasion → not found) against dictionary size

`dictionary.ContainsKey(key)`: has constant time complexity on average, $O(1)$, assuming a good hash function and a low collision rate.

Chart for finding



Figures share the roughly flat increase during words increase except for number of 1000 words. That means with exception for that strange number, the time for finding in any size of dictionary remain a same about 0.04 milliseconds.

Clearly the time complexity is $O(1)$.

Why figure for ordered 1000 text is outstandingly higher than any larger dictionary?

ChatGPT replies:

If `dictionary.ContainsKey(key)` is taking more time when the dictionary count is low compared to when it has a higher count, there are a few potential reasons for this behavior:

1. **Hash Function Collision:** Hash table-based dictionaries rely on a hash function to distribute keys across the underlying data structure. If the hash function produces many collisions for a small number of keys, the lookup time can increase even with a low count.
2. **Resizing Overhead:** Some dictionary implementations resize themselves when they reach a certain load factor to maintain efficient performance. If this resizing operation occurs frequently with a low count, it can contribute to increased lookup time.
3. **Data Distribution:** The distribution of keys within the dictionary can affect lookup

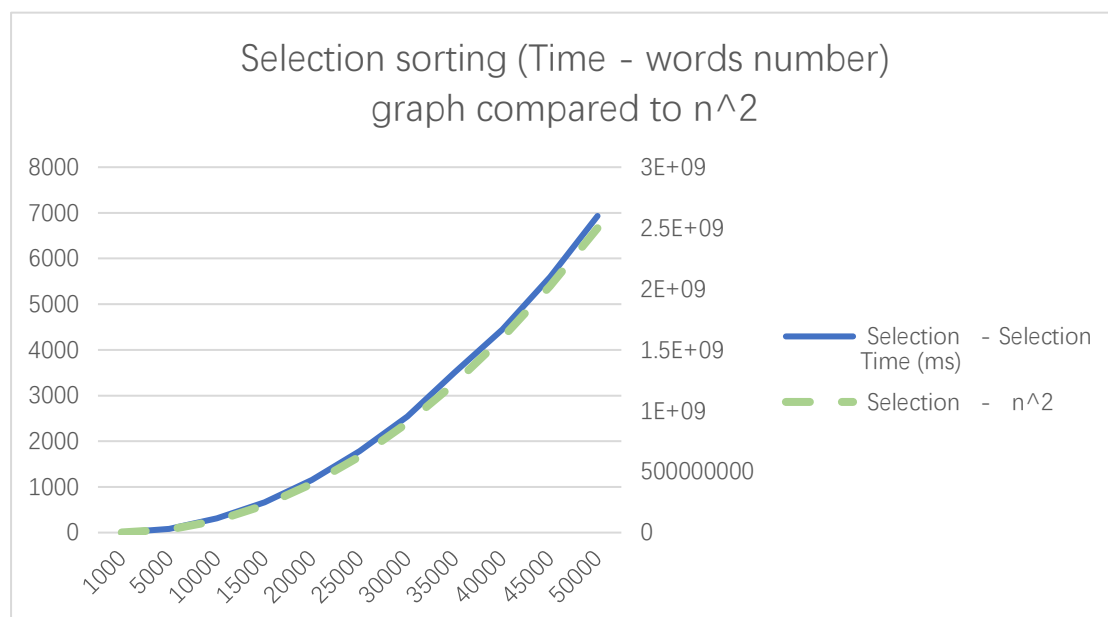
time. If the keys are not evenly distributed, it can lead to longer lookup times, even with a low count.

4. **Caching or Optimizations:** Sometimes, unexpected factors like caching behavior or compiler optimizations can influence the performance characteristics, making it seem counterintuitive.

Time for sorting arrays by two methods (selection and merge)

The time complexity of Selection Sort is $O(n^2)$, where n is the number of elements in the array. This is because there are two nested loops iterating over the array, each potentially iterating over the entire array.

Chart for selection sorting

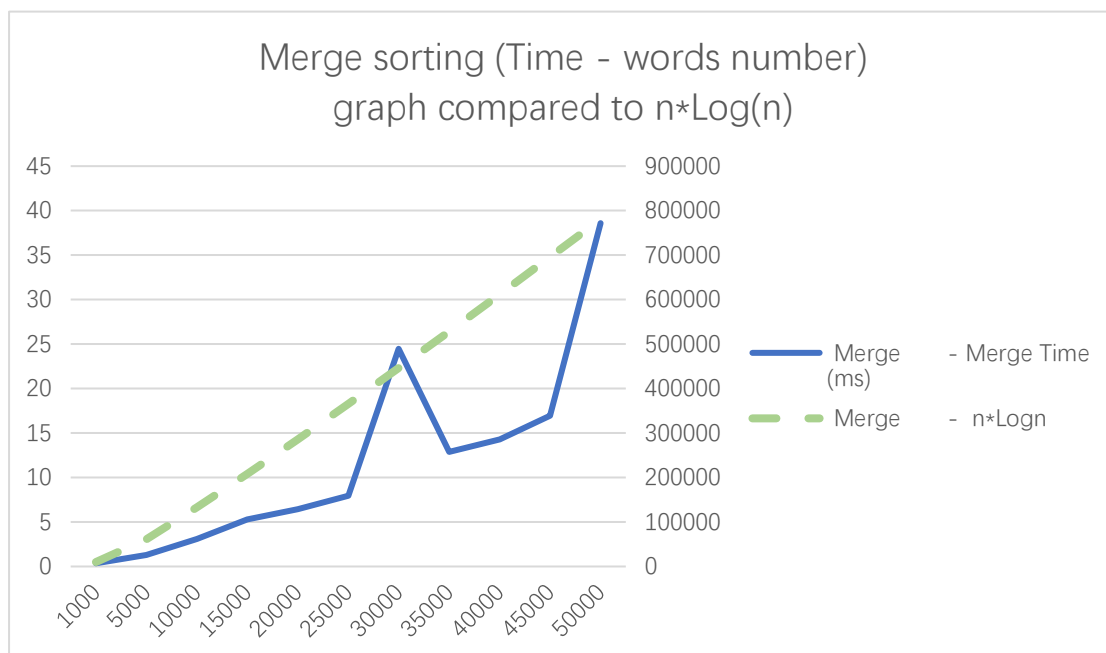


In this chart, blue line represent time for selection sort to an array of increasing words number. as compared it with formular of $T(n) = k * n^2$ ($k = 500000$) , they are highly overlapped. Therefore, Time complexity for this function is $O(n^2)$.

Chart for merge sorting

From ChatGPT

1. **Divide Step:** In the MergeSortOp method, the array is divided into halves recursively until each subarray has only one element. This process takes $O(\log n)$ time because the array is halved in each recursive call.
2. **Conquer Step:** In the Merge method, merging two sorted subarrays of length $n/2$ takes $O(n)$ time.
3. Since there are $\log n$ levels of recursion and each level has a total of n work, the total time for the conquer step is $O(n \log n)$.



In this chart, blue line represent time for merge sort to an array of increasing words number. as compared it with formular of $T(n) = k \cdot n \cdot \log(n)$ ($k = 20000$), the increase pattern are Therefore, Time complexity for this function is $O(n^2)$.

Why figure does not highly overlap with time complexity expected

From ChatGPT

1. **Constant Factors and Lower Order Terms:** The theoretical time complexity considers the dominant factor in the algorithm's performance as the input size grows. However, it doesn't account for constant factors and lower-order terms that can affect the runtime for smaller input sizes. In practical scenarios, these factors can impact the observed runtime.
2. **Hardware and Compiler Optimizations:** Modern processors and compilers may optimize certain operations, leading to faster execution times than what might be predicted by theoretical analysis alone.

Compiler optimizations, CPU caching, and parallel processing capabilities can all contribute to faster execution.

3. **Data Distribution and Characteristics:** The efficiency of merge sort can be influenced by the specific distribution and characteristics of the data being sorted. For example, if the input data is nearly sorted or has a specific pattern, the algorithm may complete faster than expected due to reduced merging operations.
4. **Implementation Details:** The efficiency of the merge sort implementation, including memory access patterns, data structure overhead, and other algorithmic optimizations, can affect the actual runtime performance.
5. **Input Size:** For small input sizes, the observed runtime may not strictly adhere to the theoretical time complexity. As the input size grows, the algorithm's performance tends to converge towards its asymptotic time complexity.