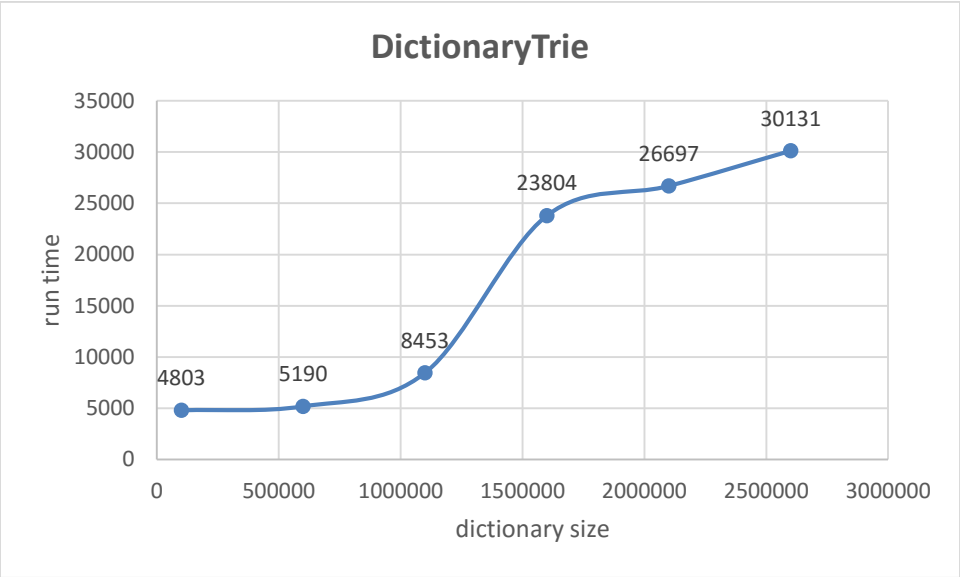
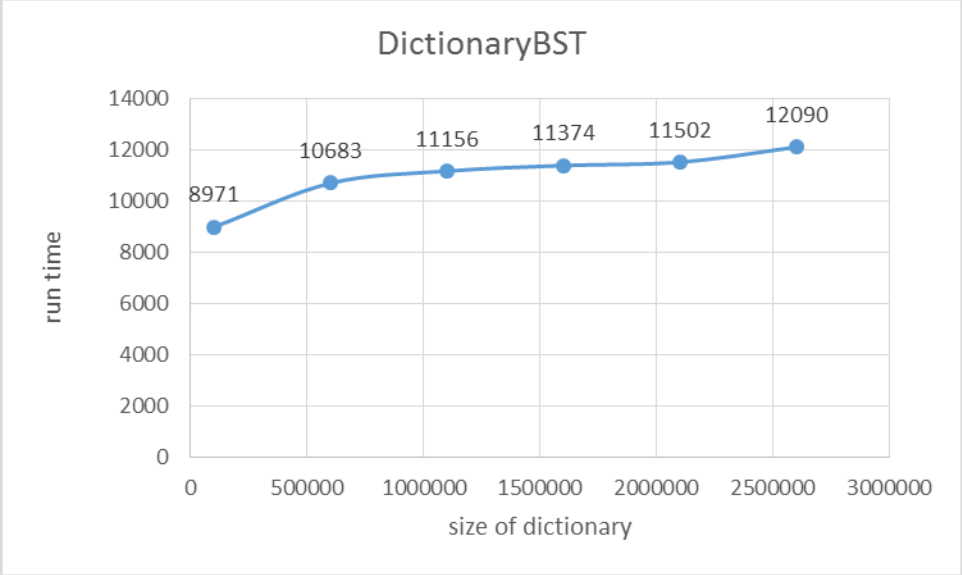
Duwei Wang, A10169673
Qihan Guan, A92413483

Final Report

Graphs:

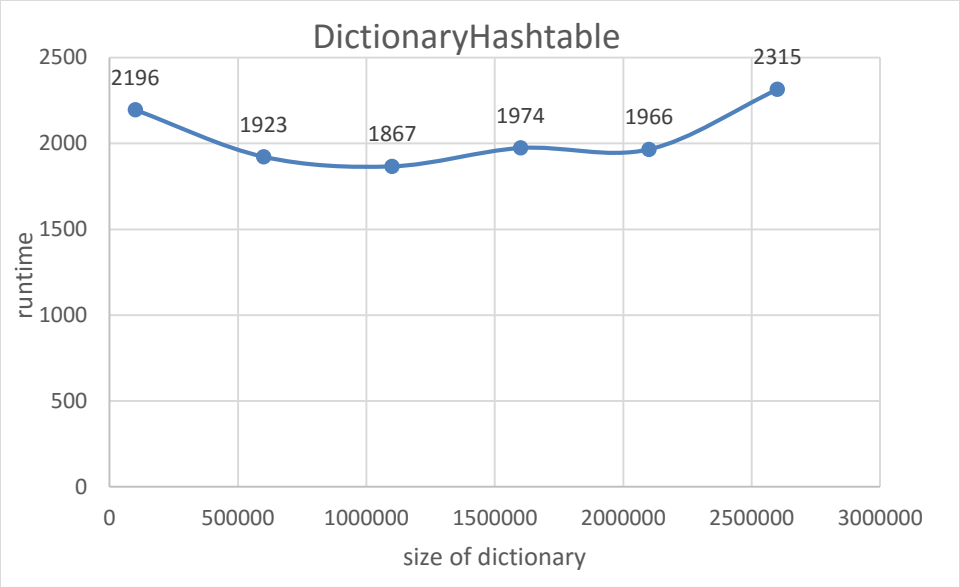


This looks like a tiny bit like a sigmoid function (if you ignore axis). But it looks like runtime is definitely affected by size of dictionary, though not consistently.



This appears to vaguely resemble a logarithmic curve.

Duwei Wang, A10169673
Qihan Guan, A92413483



This curve seems to have no real pattern, but runtimes seem to hover around 2000.

1. For the WORST case find scenarios (string not present in dictionary).
   (Our benchDict only varies n)

DictionaryTrie – multiway trie. Depends on length of word as well as number of unique chars in the dictionary and also the number of words.

Our process: start at the root. Go through the whole word, searching letter by letter. So if we start at first letter, we will go through all of the first letter's children to get to the second letter, etc. Assuming worst case, we have to search for the whole word until we realize it is not in the tree .

Then, at worst, we have to search through all of a letter's children until we find the next letter. So the total would be a theoretical O(D * number of children). The number of children depends on the size of dictionary AND the number of unique characters.

We think it is O(D*number of children). If we use D and C to somehow describe the # of children, it might be something like O(D*C), because the worst case for # of children is 27, and that depends on the complexity of the dictionary.

DictionaryBST – C++ set – The worst case scenario would be O(logn). This is because it is a binary search tree structure and those have a consistent runtime of O(logn) (where n is the container size, aka the same 'n' used for this analysis) , even in the worst case.

DictionaryHashtable – c++ unordered_set - The HashTable has a theoretical worst case of O(n) insertion time. However, that only happens with a crowded hashTable, when bumping has to occur. So we should not be expecting O(n), and instead expect O(1), the constant runtime.

Duwei Wang, A10169673
Qihan Guan, A92413483

2. For our multiway Trie, the graph is complex and reflects the more complex relationship between the 3 variables; thus, it appears to be consistent with our expectations for a 'weird' curve result because there's some further relationship between N and C; we did not reach the worst case where every single node has 27 children.
   For BST, our graph looked logarithmic so that matched our expectations.
   Hashtables have a theoretical worst case runtime of O(n) and should be O(1) under most circumstances. Our results did not appear to be very linear so I assume that we did not hit the worst case of 'many collisions during inserting', which would result in the O(n) for find; this is probably a reasonable assumption.

3. For predictCompletions, we first start by find the prefix. For the worst case of finding a prefix, it is the same as the worst case of find (for part 1) which is O(first letter of word* number of children it has + second letter of word * number of children it has + ….) which would be O(N*27) at worst, or O(D*C).
   We then do a naïve and comprehensive search starting from the prefix (for runtime purposes, BFS and DFS have the same runtime on a multiway trie). For the worst case, the prefix is the root, which would result in a runtime of $O(N*D^2)$. This is because we call findWords(), a recursive function, by N times. In findWords() we loop through the children vector, which costs time O(D). Thus, total time complexity should be $O(N*D^2)$. For every visit of a node, if it is the last letter we will pop it into a priority queue (min heap) that has a max size of num_completions; the time it takes to make a comparison is always O(1) because it is a min heap, but insertion will take O(log num_completions) and popping out a result is also O(1). This approach saves us time from the reference because the reference's insertion time is O(log N) at worst, because the max size of the priority queue is N. Thus our pqueue insert time is a constant; so the overall worst runtime for this step is $O(N*D^2)$.

   Because $O(N*D^2)$ is bigger, the overall worst case runtime is $O(N*D^2)$