ASSIGNMENT-2

Aadhithya Ganesh - 24220492

Dharnesh Vasudev IR - 24201142

Raghul Prasath Ravikumar - 24218327

Abstract

In modern search interfaces, instant word suggestions improve user experience by reducing searching time and precise word queries. This project implements an autocomplete system using a Trie (prefix tree) data structure to support fast prefix lookups and ranking based on usage frequency. We discuss system design, data-structure and algorithm choices, complexity analysis, implementation details, and experimental results.

Problem Selection

We selected the autocomplete use case because it shows us a real-world scenario where efficient, scalable data structures and algorithms are important in delivering good user experiences. Autocompletion is found in e-commerce search bars, code editors, and even in mobile keyboards and presents clear performance and correctness challenges:

- ❖ **Performance Constraints:** Suggestions must appear almost instantaneously when users type.
- ❖ **Data-Structure Requirements:** Fast prefix lookup needs a tree-based data structures over others. Also, the need for fuzzy matching and ranking adds algorithmic complexity.
- ❖ **Scalability:** The solution must handle increasing dictionary sizes and large query loads without degrading user experience.

By focusing on autocompletion with a Trie, we could explore and demonstrate key algorithmic concepts like prefix trees, depth-first traversal, subsequence matching, and frequency-based query retrieval.

System Design

1. **Trie Traversal:** Walk the trie following each character of the input. If a character isn't found, fallback to fuzzy matching (subsequence check).
2. **Candidate Gathering:** From the node at the end of traversal, perform a depth-first search (DFS) to collect all descendant words.
3. **Scoring & Ranking:** Each candidate receives a score everytime it is searched by the user. We sort candidates by score (and break ties lexicographically). Return the top *k* (e.g., k = 10) suggestions.

Data Structures

- ✓ **Trie (Prefix Tree):** A Trie allows prefix lookup in O(length of the string) time, independent of dictionary size. Trie is a tree-like data structure which is used to search and store large collections of strings efficiently. Structuring nodes in a way they can be retrieved by traversing down the path of the tree .It is also known as a prefix tree as it stores the common prefix only one time as shared prefixes reduce memory usage in practice. Alternatives like hash tables cannot enumerate prefixes efficiently.

- ✓ **HashMap (Dictionary):** A HashMap provides O(1) average time for updating and retrieving word selection frequencies, enabling real-time ranking by popularity. The HashMap's constant-time performance make it ideal for frequency tracking. Alternatives like balanced trees require O(log N) per update or lookup, which can be costly under high query volumes.

Algorithms

1. **Depth-First Search (DFS):** It is used when we want to explore all the possible nodes in the tree. Enumerates descendant words from a prefix node in O(k·l) time, where k is the number of matched words and l is the average word length.
2. **Subsequence Fuzzy Matching:** Checks if the user's input is a subsequence of each word in the dictionary, handling the typo errors of the user.
3. **Quick sort:** An efficient, in-place, divide-and-conquer sorting algorithm. It selects a pivot element, partitions the list into values less than and greater than the pivot, then recursively sorts the partitions. Average time complexity is O(n log n), and it requires O(log n) additional stack space due to recursion.

Implementation:

1. **Prefix Lookup:** Attempts to find the Trie node for the exact searchString.
2. **Fuzzy Fallback:** If no node, collects all words with the same first character and applies a subsequence check to suggest near-matches.
3. **Candidate Gathering:** Uses DFS from the found node to gather all prefix-matching words.
4. **Frequency Map Construction:** Builds a map of each candidate's previous search counts from searchedWords.
5. **Sorting & Limiting:** Sorts candidates by frequency (using quicksort_dict_by_value) and selects the top 10.
6. **User Selection:** In interactive mode, prompts the user to choose a suggestion; in test mode, picks randomly.
7. **Frequency Update:** Increments the count for the selected word in searchedWords, enabling adaptive ranking over time.

Complexity Analysis:

Time Complexity Analysis:

Steps	**Operation**	**Complexity**	**description**
Prefix traversal	Trie.StartsWith(searchString)	O(p)	p = input prefix length
Candidate Gathering	DFS(node, SearchString)	O(k·l)	k = Number of matched words; l=average matched-word length
Fuzzy Matching	is_subsequence(sortedSearchString, sortedWord)	O(s+t)	s = user input length; t = candidate word length
Sorting	quicksort_dict_by_value(currentWordsCount)	O(k log k)	Sorting k candidates

Time Complexity for Overall Query = $O(p + k \cdot \ell + \Sigma(s+t) + k \log k)$

Space Complexity

Trie Storage:

- Worst-case: $O(N \cdot m)$ nodes, where N = number of words, m = average word length.

Word Search Function:

- Result list: $O(r)$, where r = number of candidates collected.
- CurrentWordsCount dict: $O(r)$ entries tracking frequency counts.
- Recursion stack (DFS): $O(d)$, where $d \leq$ maximum depth of Trie (~m).
- Quick Sort: $O(\log k)$ on average, since each recursive partitioning depth is proportional to the height of the recursion tree.