BinarySearchAutocomplete

1. addAll
   - Code
     - The addAll method first adds all new terms to the elements list using elements.addAll(terms);. Then, it sorts the entire list with sort(CharSequence::compare)
   - Analysis
     - Adding a constant number of terms to the existing list of N terms results in a list of size N+k, where k is a constant.
     - Sorting this list takes Θ(NlogN) time since k is negligible compared to N.
     - The overall runtime is dominated by the sorting operation → **Θ(NlogN)**
2. allMatches
   - Code
     - The method starts by performing a binary search: int i = Collections.binarySearch(elements, prefix, CharSequence::compare);. It then adjusts the starting index and enters a loop
   - Analysis
     - The binary search takes O(logN) time.
     - The while loop iterates over the elements starting from start and adds matching terms to result.
     - In the worst case, where all N terms match the prefix (e.g., if the prefix is empty or very common), the loop runs Θ(N) times.
     - Each iteration involves result.add() and a prefix check with Autocomplete.isPrefixOf(), both of which are O(1) operations assuming constant-length strings.
     - The total runtime is **Θ(N)** in the worst case.

SequentialSearchAutocomplete

1. addAll
   - Code
     - The addAll method simply adds all terms to the list: elements.addAll(terms);
   - Analysis
     - Since we assume that the ArrayList can accommodate new terms without resizing, each addition is O(1).
     - Adding a constant number of terms takes **Θ(1)** time.
2. allMatches
   - Code
     - The method iterates over all elements
   - Analysis
     - The loop runs once for each term in the list.

    - Each iteration involves a prefix check and possibly adding to the result
    - Therefore, the total runtime is **Θ(N)**

TernarySearchTreeAutocomplete`

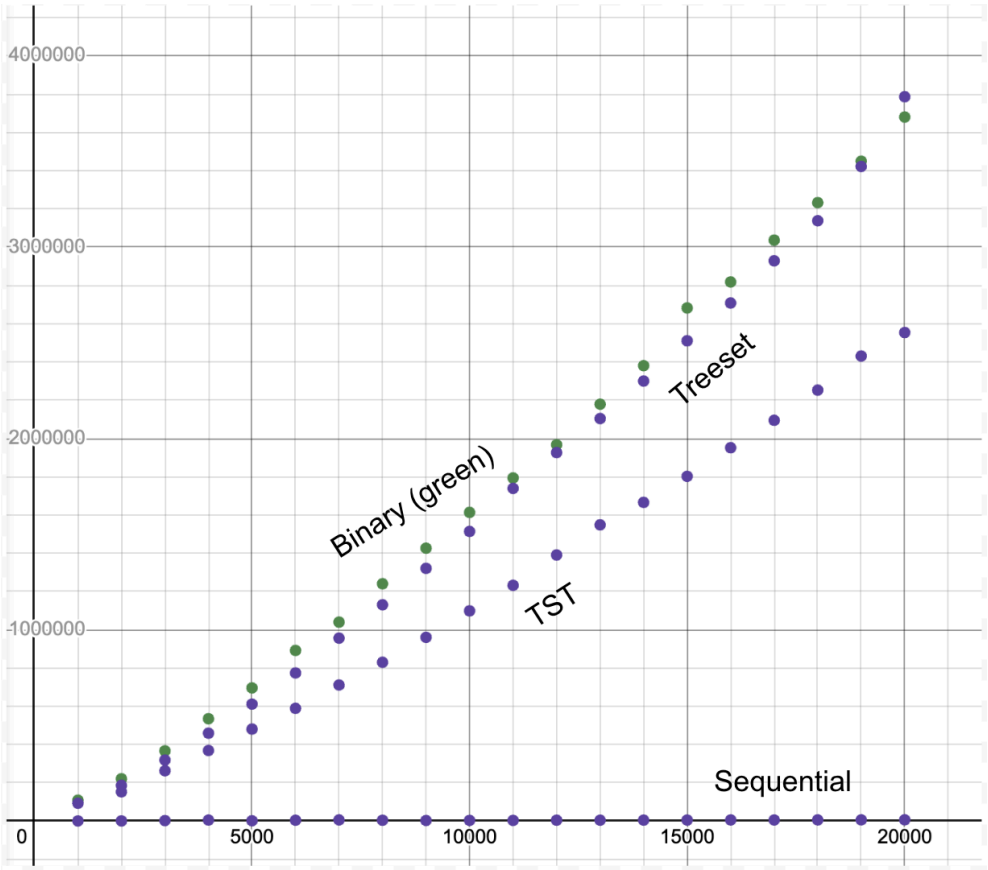
1. addAll
    - Code
        - The addAll method adds each term using the put method:
    - Analysis
        - The put method recursively inserts characters into the TST. Since all strings are of constant length, the depth of recursion is constant.
        - Each insertion takes Θ(1) time.
        - Adding a constant number of terms results in **Θ(1)** total time.

2. allMatches
    - Code
        - The method searches for the node matching the prefix and then collects
    - Analysis
        - Finding the node corresponding to the prefix takes Θ(1) time
        - The collect method may traverse the entire TST in the worst case
        - If all N terms share the prefix the traversal visits all nodes.
        - Therefore, the runtime is **Θ(N)** in the worst case.

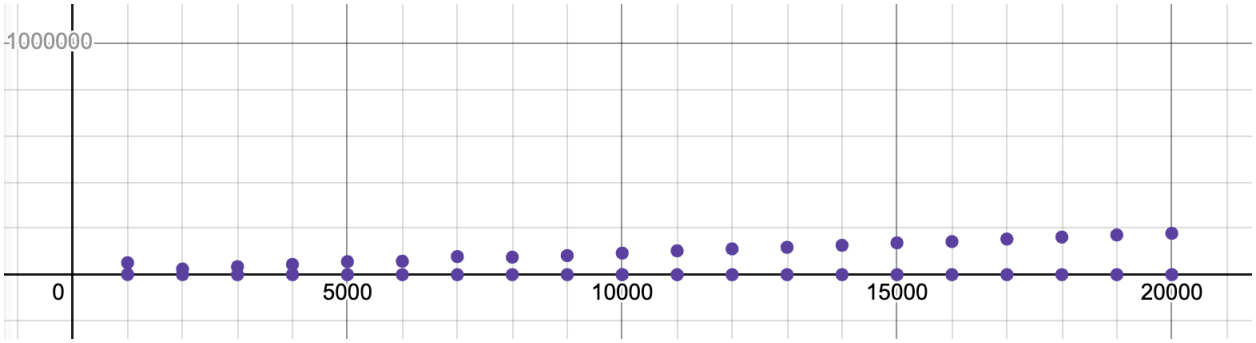
TreeSetAutocomplete

1. addAll
    - Code
        - The addAll method adds all terms to the TreeSet
    - Analysis
        - Each insertion into a TreeSet takes O(logN) time
        - Adding a constant number of terms results in **Θ(logN)** total time
2. allMatches
    - Code
        - The method finds the starting point and iterates over the tailSet
    - Analysis
        - Finding the ceiling takes O(logN) time.
        - The tailSet view allows iteration over terms greater than or equal to start.
        - In the worst case, where all terms match the prefix, the loop runs N times.
        - Each iteration involves a prefix check and adding to the result (Both O(1))
        - Therefore, the total runtime is **Θ(N)** in the worst case.

Implementation	`addAll` Runtime	`allMatches` Runtime
BinarySearchAutocomplete	$\Theta(N \log N)$	$\Theta(N)$
SequentialSearchAutocomplete	$\Theta(1)$	$\Theta(N)$
TernarySearchTreeAutocomplete	$\Theta(1)$	$\Theta(N)$
TreeSetAutocomplete	$\Theta(\log N)$	$\Theta(N)$



- Sequential (purple at the bottom):
    - For adding elements to a simple list, the insertion is straightforward, with minimal overhead beyond resizing as needed. This results in lower runtimes for addAll if the list grows smoothly without frequent resizing.
- TST:
    - Adding multiple elements involves multiple node insertions and balancing, leading to moderate runtimes as shown. TSTs are efficient for prefix-based lookups but can incur extra overhead for structural adjustments during bulk insertions.
- Binary (green):
    - Adding elements to a binary search tree, the addAll function incurs the cost of maintaining order and potentially rebalancing the tree, hence the visible increase in runtime with larger inputs.
- TreeSet:
    - TreeSets have a higher insertion overhead than simple lists due to the balancing required to maintain tree properties. This results in the steepest runtime increase, as shown on the graph.



For the allMatches method, this graph suggests that the runtime remains consistently low across different input sizes.

Sequential as an Outlier:

- The line representing Sequential clearly demonstrates worse performance than other implementations. The steady increase in runtime suggests that Sequential is performing a linear search through the dataset, with runtime directly proportional to the number of elements. This method's inefficiency stems from examining each element individually to find matches, resulting in O(n) complexity for each search. In contrast, more optimized data structures can provide significantly faster lookups, especially for large datasets, by narrowing down the search space more effectively.

Constant Runtime:

- The flat line implies that allMatches has minimal or consistent overhead, due to efficient retrieval mechanisms that don't depend heavily on the dataset size. This could be the case if the method leverages an optimized index or lookup structure that quickly retrieves matching elements.

Low Variability:

- The lack of growth in runtime suggests that the allMatches method efficiently handles increasing input sizes, likely using a data structure well-suited for prefix or pattern matching (e.g., tries or hash maps)