CS 61B Fall 2025

Tries and Sorting

Discussion 11: November 10, 2025

1 A Wordsearch

Given an N by N wordsearch and N words, devise an algorithm (using pseudocode or describe it in plain English) to solve the wordsearch in $O(N^3)$. For simplicity, assume no word is contained within another, i.e., if the word **"bear"** is given, **"be"** wouldn't also be given.

If you are unfamiliar with wordsearches or want to gain some wordsearch solving intuition, see below for an example wordsearch.

Example Wordsearch:

S F T A T R D V R G K E V I N
A T S K L N X F I H D P X H Z
C N E D X A J Z G U N A I R U
J Y I L C V A N E S S A V P O
A B A R L K F J Q U S Y H I C
V Z U I U A T Q K D A A G R D
F S P E I D S T A A S N M Y N
W C T T S S H Q T W H N G A U
S I W H P E A A E N L I T N V
T Y I A G L D B R K E Y T Y K
A L N N J A J G E T Y A P E A
C X F I K I M U S L I T Y P R
E M U A M N T M A Z X A H U E
Y R A E K E Y W I K O Y O P N
R B C A Q J V Q I A C R O E F

Anirudh	Anniyat	Vanessa	Ryan
Ashley	Elaine	Isabel	
David	Stella	Karen	
Ethan	Kevin	Teresa	
Stacey	Dawn		

Hint: Add the words to a **Trie**, and you may find the `longestPrefixOf` operation helpful. Recall that `longestPrefixOf` accepts a `String key` and returns the longest prefix of `key` that exists in the **Trie**, or `null` if no prefix exists.

Solution:
Algorithm: Begin by adding all the words we are querying for into a **Trie**. Next, we will iterate through each letter in the wordsearch and see if any words **start** with that letter. For a word to start with a given letter, note that it can go in one of eight directions — N, NE, E, SE, S, SW, W, NW.

Looking at each direction, we will check if the string going in that direction has a prefix that exists in our **Trie**, which we can do using `longestPrefixOf`. Note that words are not nested inside of others, so **at most** one word can start from a given letter in a given direction. As such, if `longestPrefixOf` returns a word, we know it is the only word that goes in that direction from that letter.

For instance, if we are at the letter "S" in the middle of the top row of the wordsearch above and are considering the direction west, we would want to see if the string `"SOHUMC"` has a prefix that exists in the given wordsearch. To efficiently perform this query, we call `longestPrefixOf("SOHUMC")`, which, in this case, returns `"SOHUM"`, and we proceed by removing `"SOHUM"` from our **Trie** to signal that we found the word `"SOHUM"`.

We will repeat this process until all the words have been found, i.e. when the **Trie** is empty. Finally, note that this is a very open-ended problem, so this is one of **many** possible solutions.

Runtime: We look at N^2 letters. At each letter, we execute eight calls to `longestPrefixOf` which runs in time linear to the length of the inputted string, which can be of at most length N, since that is the height and width of the wordsearch. Thus, if we perform on the order of N work per letter and we look at N^2 letters, the runtime is $O(N^3)$.

2 Longest Prefix

Fill in the **`longestPrefixOf(String word)`** method below such that it returns the longest prefix of **`word`** that is also a prefix of a key in the trie.

For example, if a **`TrieSet t`** contains keys **`{"cryst", "tries", "cr"}`**, then **`t.longestPrefixOf("crystal")`** returns **`"cryst"`** and **`t.longestPrefixOf("crys")`** returns **`"crys"`**.

The code uses the **`StringBuilder`** class to build strings character-by-character. To add a character to the end of the **`StringBuilder`**, use the **`append(char c)`** method. Once all characters have been appended, the resulting String is returned by the **`toString()`** method.

```
  StringBuilder sb = new StringBuilder();
  sb.append('a');
  sb.append('b');
  System.out.println(sb.toString()); // "ab"
```

```
public class TrieSet {
    private Node root;
    private class Node {
        boolean isKey;
        Map<Character, Node> map;
        private Node() {
            isKey = false;
            map = new HashMap<>();
        }
    }

    public String longestPrefixOf(String word) {
        int n = word.length();
        StringBuilder prefix = new StringBuilder();
        Node curr = _____________________;
        for (_____________________________________) {
            ____________________________________________________
            ____________________________________________________
            ____________________________________________________
            ____________________________________________________
            ____________________________________________________
            ____________________________________________________
        }
        return _____________________________________________
    }
}
```

Solution:

```
public String longestPrefixOf(String word) {
    int n = word.length();
    StringBuilder prefix = new StringBuilder();
    Node curr = root;
    for (int i = 0; i < n; i++) {
        char c = word.charAt(i);
        if (!curr.map.containsKey(c)) {
            break;
        }
        curr = curr.map.get(c);
        prefix.append(c);
    }
    return prefix.toString();
}
```

3 Identifying Sorts

Below you will find intermediate steps in performing various sorting algorithms on the same input list. The steps do not necessarily represent consecutive steps in the algorithm (that is, many steps are missing), but they are in the correct sequence. For each of them, select the algorithm it illustrates from among the following choices: insertion sort, selection sort, mergesort, quicksort (first element of sequence as pivot), and heapsort. When we split an odd length array in half in mergesort, assume the larger half is on the right.

Input list: 1429, 3291, 7683, 1337, 192, 594, 4242, 9001, 4392, 129, 1000

(a) 1429, 3291, 7683, 1337, 192, 594, 4242, 9001, 4392, 129, 1000
1429, 3291, 192, 1337, 7683, 594, 4242, 9001, 129, 1000, 4392
192, 1337, 1429, 3291, 7683, 129, 594, 1000, 4242, 4392, 9001

Solution: Mergesort. One identifying feature of mergesort is that the left and right halves do not interact with each other until the very end. Further, note that the first line has had several steps applied to it, and yet is completely unchanged. This is reflective of how mergesort at first simply partitions the array without sorting anything.

(b) 1337, 192, 594, 129, 1000, 1429, 3291, 7683, 4242, 9001, 4392
192, 594, 129, 1000, 1337, 1429, 3291, 7683, 4242, 9001, 4392
129, 192, 594, 1000, 1337, 1429, 3291, 4242, 4392, 7683, 9001

Solution: Quicksort. First item was chosen as pivot, so the first pivot is 1429, meaning the first iteration should break up the array into something like $| < 1429 | = 1429 | > 1429$

(c) 1337, 1429, 3291, 7683, 192, 594, 4242, 9001, 4392, 129, 1000
192, 1337, 1429, 3291, 7683, 594, 4242, 9001, 4392, 129, 1000
192, 594, 1337, 1429, 3291, 7683, 4242, 9001, 4392, 129, 1000

Solution: Insertion Sort. Insertion sort starts at the front, and for each item, move to the front as far as possible. These are the first few iterations of insertion sort so the right side is left unchanged

(d) 1429, 3291, 7683, 9001, 1000, 594, 4242, 1337, 4392, 129, 192
7683, 4392, 4242, 3291, 1000, 594, 192, 1337, 1429, 129, 9001
129, 4392, 4242, 3291, 1000, 594, 192, 1337, 1429, 7683, 9001

Solution: Heapsort. This one's a bit more tricky. Basically what's happening is that the second line is in the middle of heapifying this list into a maxheap. Then we continually remove the max and place it at the end.

In all these cases, the final step of the algorithm will be this: 129, 192, 594, 1000, 1337, 1429, 3291, 4242, 4392, 7683, 9001