Tries

(Arup Guha, University of Central Florida)

A trie is a type of tree which efficiently stores a dictionary of words. An idea usually shown to first semester students that a trie uses is the idea of a frequency array. In a frequency array of characters freq[0] stores the number of a's seen, freq[1] stores the number of b's seen, and so forth. The salient part of this type of storage is that the character is implicit in the index. We never actually store 'a', 'b', or any of the letters. Rather, we exploit the ordering of the letters, and assign a meaning based on the index within which something is stored. A trie exploits this same exact idea of implicitly storing a letter via the index into an array.

A trie is a tree, where each node represents a letter in the prefix of some word stored in a dictionary of words. The most basic trie node has the following instance variables:

```
trienode[] next = new trienode[26];
boolean flag;
```

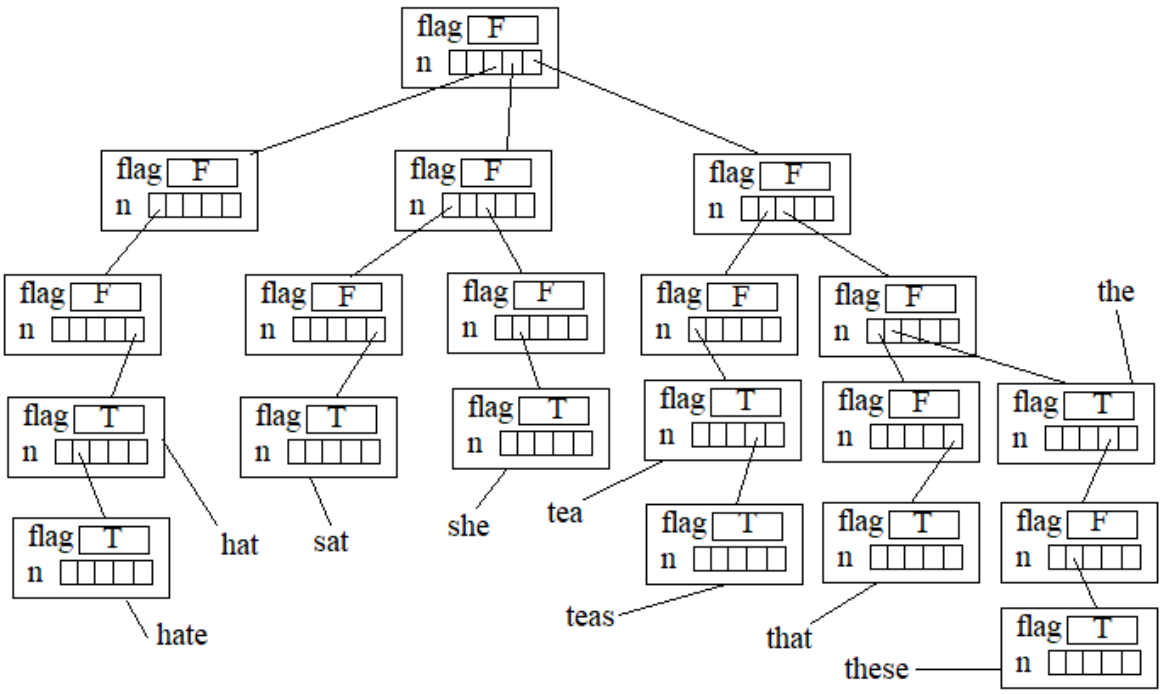
The *flag* will store true if this prefix is actually a word, and false if it is not. The reference *next*[*i*] will be null if there is no word in the dictionary that starts with the given prefix followed by the letter *i* represents. Otherwise, the reference would point to a node representing the given prefix followed by the letter *i* represents.

To visualize a trie, let's use a reduced alphabet with the letters A, E, H, S, and T assigning these to the numbers 0, 1, 2, 3 and 4, respectively. Let our trie store the following words:

HAT, HATE, SAT, SHE, TEA, TEAS, THAT, THE and THESE

Note that the root node of a trie represents the empty prefix.

A picture of the trie which stores these words is included on the next page. In order to make the picture fit nicely, 'T' is used to represent true and 'F' is used to represent false, and a lowercase *n* is used to represent the array *next*. The nodes which represent each prefix that correspond to a word are also indicated. Notice that the key is that which links are active is how all the different words are stored. Thus, all we need to do to store a word is just store "true" in the node that represents that prefix.



Inserting a node in a trie

If we do this iteratively, we can simply start at the root, and then follow the links corresponding to the appropriate letters in the word. If a link doesn't exist, we must create that node (and each of the subsequent ones). When we get to the last node, we must simply change its *flag* value to true, to indicate that the word to be inserted ends there.

Recursively, we can pass in the node the insertion is being done on, the word, and an integer *k*, indicating how many letters have been read in so far. If we use this system, our base case is when *k* is the length of the word. If this is the case, we just set flag to true and return. Otherwise, we recursively insert into the next node. If the next node doesn't exist, we create it first. Here is C code for trie insertion:

```
typedef struct TrieNode {
    struct TrieNode *children[26];
    int flag; // 1 if the string is in the trie, 0 otherwise
} TrieNode;

void insert(TrieNode *tree, char word[], int k) {

    if (k == strlen(word)) {
        tree->flag = 1;
        return;
    }
```

```
    int nextIndex = word[k] - 'a';
    if (tree->children[nextIndex] == NULL)
        tree->children[nextIndex] = init();
    insert(tree->children[nextIndex], word, k+1);
}
```

Searching for a node in a trie

Searching is fairly similar to inserting. Just follow the path using the letters in the word, either iteratively or recursively. The key difference is that if a *next* link is null, instead of creating a node, that is proof that the word being searched for is NOT stored in the trie, so you just stop and return false.

Insert/Search Run Times

It should come as no surprise that the run-times for both inserting and searching for a word in a trie are O(*len*), where *len* is the length of the word in question. Thus, tries are ideal for real dictionaries of words, where no individual word is too long, but there are many words to search from. The maximum storage of a trie is on the order of the total number of letters of all of the words in the dictionary, but may be less than this due to all shared prefix letters.

Storing Extra Information in a Trie Node

When using tries in programming contests, it's typical that in order to solve the problem in question, the basic trie node must be edited, to store extra information. One simple piece of information would be an extra integer representing the number of words stored in that particular subtree. When we add a field to store some piece of information in each node, we must then update that piece of information in every node. For this specific example, when we insert a word, we must add 1 to the *numwords* field of each node ancestor node of the last node of the word. The only thing that would change of the insert shown on the previous page is adding this line to it first:

```
    tree->numwords += 1;
```

Now, imagine having to answer queries of the form, "How many words start with the prefix 'trans'?" To answer this question, we just search to the node corresponding to the prefix "trans", and then just return what's stored in the *numwords* field of that node.

Solving a Problem: What word has the most prefixes in it that are ALSO words?

Consider the word "intention". Its prefixes, "i", "in", "intent" and "intention", are all words. In general, consider solving the problem: given a trie storing a dictionary of words, find which word

has the most prefixes which are also words. The answer to this question is the SAME as finding the path in a trie from the root to a leaf which go through the most number of nodes storing 1 or true for their *flag* field. Here is some C code which solves this problem:

```
int maxNumPrefixWords(TrieNode *root) {

    if (root == NULL) return 0;
    int maxChild = 0;
    int i;
    for (i=0; i<26; i++)
        maxChild = max(maxChild,
                       maxNumPrefixWords(root->children[i]));

    return maxChild + root->flag;
}
```

Basically, we want the best path of all of our subtrees. Then, we want to take this value and add our root node value to this number, since our root node adds to any of the subtree paths.

Alien Rhyme Problem (Google Code Jam 2019 Round 1A)

In this problem, you are given up to a thousand words, with a max length of 50 and would like to pair up as many of them as possible such that the suffix in each pair is unique and no other word in the group has the same suffix.

Although it might not seem like it, we can use the same exact ideas as those previously mentioned to solve the problem.

First, notice that suffixes of words are stored in different places in a trie, so this storage system would be unnatural to help pair up words. But…consider the idea of reversing each word. Then, suffixes become prefixes (in reversed order). Furthermore, if we just keep a *numWords* field in each trie node, and if this field stores the number 2, this means that exactly 2 words have this suffix and we should pair them up! So, to solve the problem – read in all of the words in reverse order, storing the number of words in each subtrie. This data structure is now nicely set up to solve the problem at hand.
So, we'll do the following:

1. Recursively solve the problem for each sub-node. We can definitely add up all of these answers, since no two sub-nodes will store the same suffix.

But, this will miss a few corner cases. If in doing this, there are at least two words that weren't paired up, we are allowed to pair up these two words by then making the root node the accented node (since it's not accented for any of the other words). If there are more than 2 words that

weren't picked it's okay, since we're allowed to discard some words. But, we can just keep 2 of them.

Here is Java code that solves the key part of the problem:

```
// Solves the problem for depth.
public int solve(int depth) {

	// We can grab both...
	if (numWords == 2 && depth > 0) return 2;

	// Try adding all the words from subtrees.
	int res = 0;
	for (int i=0; i<26; i++) {
		if (next[i] != null)
			res += next[i].solve(depth+1);
	}

	if (numWords-res >= 2 && depth > 0) res += 2;

	return res;
}
```

Bless You Autocorrect (2016 Nordic Collegiate Programming Contest)

In this problem you are given a dictionary of words and the algorithm a phone uses to speed up typing and for given words to type, have to calculate the fewest button presses to type the word out. Except for typing letters, you are allowed two other keys to press - the tab key and backspace. After typing more than 0 letters, typing the tab key will fill in the rest of the letters of the most common word that starts with the prefix you already typed. For example, if you already typed "auto" and "autocorrect" was the most common word in the dictionary with prefix "auto", then pressing the tab would result in "autocorrect" being typed on the phone. Thus, five button presses would suffice instead of 11 to type "autocorrect."

The input contains up to 10^5 words for the dictionary and up to 10^5 strings to type (which may or may not be part of the existing dictionary). The dictionary words are given in order of frequency. Also, the total input file (all characters in all cases) won't exceed 1 megabyte, which means that if there are lots of words in the file, their average length isn't so long.

A trie is a natural choice to search for strings in this problem, but what has to be adapted is how to account for tab savings. If we just iterate through the trie trying to type a string, we'll just get the default cost of doing so.

In each node of the trie, it's important to quickly know what would happen if we pressed the tab button after typing this prefix. A simple way to do this is for each trie node, store the *index* from the original dictionary that node would "predict." We do this by simply inserting the dictionary words in order from most to least frequent, and when inserting a word, if it creates a node that didn't previously exist, when we create the node, we store this *index* in it, to indicate that the word from this *index* created this node, which means it's the most frequent word with this *index*. Here is the insert code:

```
public void insert(char[] word, int index) {

    if (level == word.length) return;

    if (next[word[level]-'a'] == null)
        next[word[level]-'a'] = new tri(level+1, index,
                                        word.length);

    next[word[level]-'a'].insert(word, index);
}
```

See that ONLY if the next node is null do we create it storing *index*. If a node already exists, we never change its *index* value, since that word was inserted before, and thus, was more frequent.

Now that we have this information, when searching for the best path for a word, as we walk the path for that word, when we are at any node, we can see where that node would jump us to (for free). Using the old example, let's say we are at the node corresponding to "auto", and this node stored 17 in its *index* and the 17th most frequent word was "autocorrect". Let's say we wanted to type "autocorrelation", then, with the tab, we get the letters “correct” for free. What we can do is keep track of an alternate score, one for taking the jump from the current node. We add the number of extra letters added to our word in this jump, which in this case is 7. But, now, as we continue walking through autocorrelation, when we get to the ‘c’, we see that its *index* value is also 17 (meaning that if we typed “autoc” and hit tab, it would give us “autocorrect”). This means, that, for our jumping path, this c doesn’t cost us anything, so we subtract 1 from our alternate cost. Namely, our alternate cost assumes that we’ll have to erase all the extra letters initially, but then, as we trace the path and see that a letter matches, we can subtract 1 from our alternate cost, because that means that is a letter we won’t have to erase if we do that jump. At any given time, we can only have one “active” jump that we are evaluating, so, when our jump no longer helps us, for example, when we get to “autocorrel”, then perhaps there might be a new jump to use, or no jump at all. Here is the code that carries out this logic iteratively (*list* is just the string to be scored):

```
public static int solve(char[] list) {

     int curRes = 0, alt = 10000000;
     int prevIndex = -1;

     tri ptr = words;
```

```
        // Go through each letter.
        for (int i=0; i<list.length; i++) {

            // Possibility of just typing this letter.
            curRes++;

            // Have to type the rest.
            if (ptr.next[list[i]-'a'] == null)
                return curRes + list.length - (i+1);

            // New jump.
            if (prevIndex != ptr.next[list[i]-'a'].pIndex)
                alt = curRes + ptr.next[list[i]-'a'].pLen - i;

            // Same as old jump but gaining a letter.
            else
                alt--;

            // See if the jump is better.
            curRes = Math.min(curRes, alt);

            // Update variables for next iteration.
            prevIndex = ptr.next[list[i]-'a'].pIndex;
            ptr = ptr.next[list[i]-'a'];
        }

        return curRes;
    }
```