Casey White

Time Complexity Regular tree(Exhaustive search):

Build: For any digit in the input sequence there are either 3 or 4 letters associated with it, so to analyze time cost for the worst case there are 4 letters in each node of the trie. When the tree is built there is n levels where n is the length of the input sequence. Every level in the trie will have 4 nodes, each with 4 children. So mathematically we can write this growth rate function as O(4^n) where n is the depth of tree/ length of input sequence. I don’t think the GRF is correct. Like at all.

```
//recursive helper to build tree
    private void buildTreeRecursive(TrieNode cur, ArrayList<Integer> input, int depth){
        if(depth >= input.size())      //2
            return;
        char[] curIntChars =
MyDictionary.getNumToChar().get(input.get(depth++)).toCharArray();//5 I think.Idk about to char[]
        for (char ch : curIntChars) {            //4 worst case
            TrieNode nextNode = new TrieNode(); //1
            cur.children.put(ch, nextNode);     //1
           buildTreeRecursive(nextNode, input, depth);//1
        }
    }
```

Growth rate function = 4^n+19n

Retrieve: The time complexity of retrieving all possible words in the trie after it is built, depends on depth n. in my recursive method it follows all possible paths to leaf nodes(possible words). So worst case it will follow the 4 original nodes, each one of those splits into 4 children and so on. Because of the recursive implementation it only visits any node once making the get all valid words method have growth rate function of O(4^n) where n is the depth of tree/ length of input sequence. If at every leaf node it checks the dictionary if it is a valid word. If we search in the hash table, every check is O(1). If we check the dictionary trie, every check is O(n)

```
//get valid words recursive helper
private ArrayList<String> getAllValidWordsHash(TrieNode cur, ArrayList<String> words, String word){
    if(cur.isLeaf() && MyDictionary.getHashDictionary().contains(word))      //3+2. If check trie + n
        words.add(word);
    if(cur.children.isEmpty())//2
        return words;

    Iterator<Character> itr = cur.children.keySet().iterator();//4
    while (itr.hasNext()){//4
        Character childLetter = itr.next();//2
        getAllValidWordsTrie(cur.children.get(childLetter), words, word + childLetter);//2
    }
```

GRF trie = n*4^n + 26n

GRF hash = 4^n + 27n

GRF Total hash = 2*(4^n) + 48n (I don’t know how accurate this is)

GRF Total trie = n*(4^n) + 4^n + 47n   (I don’t know how accurate this is)

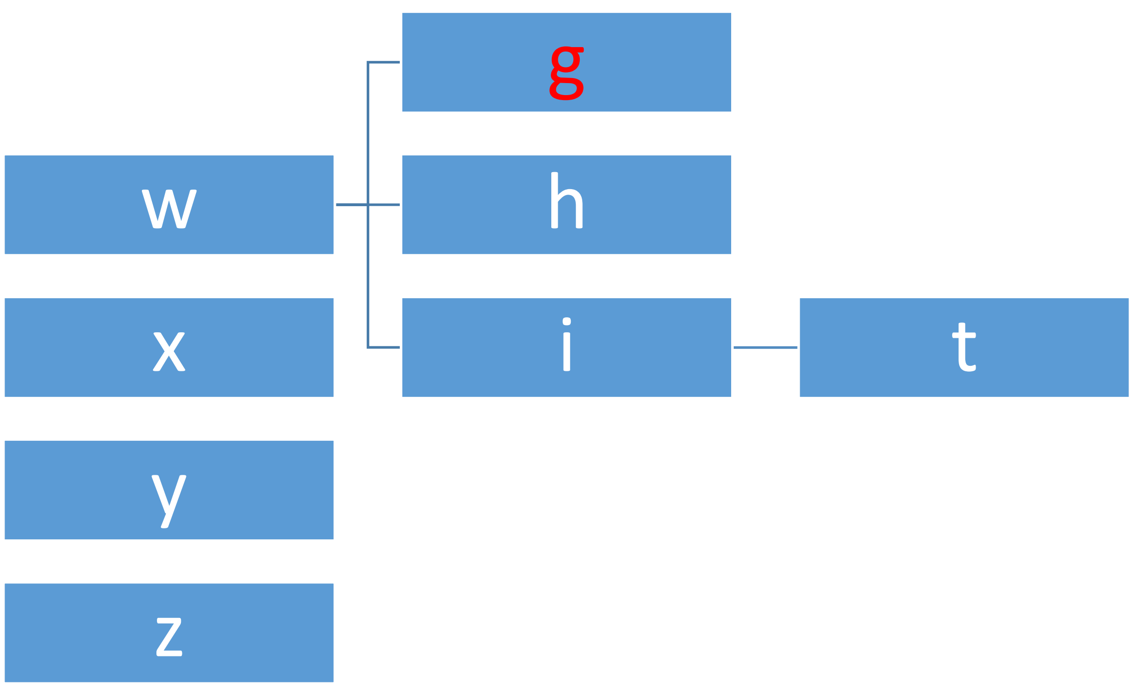
Branch and Bounds Time Complexity:

As you can see from above time complexity analysis, I don't know how to properly do it. So rather than try to bs my way through it, I have explained in the answer to question 4 below, how the branch and bounds will have performance gains over exhaustive search

2) Exhaustive Search: and exhaustive search brute-force algorithm, is an algorithm that exhaustively searches all possible items. What I mean by that is it always checks all the items. No items are skipped no matter how wrong they fit the problems condition. An example will help clear this up. In my program, the user enters n digits. The tree created contains 4^n possible words(worst case). Then upon retrieval, the program will check every one of these possible words to see if it's a word.

3.) Branch and bound: A Branch and bound algorithm will perform the same operation as an exhaustive search algorithm. However the big difference is this algorithm will only check the conditions on items that might work, instead of all items. This implementation mathematically reduces the number of items to search, making it faster. The more unlikely items, the greater performance gain over exhaustive.

4.) To improve performance In my program with the Branch and bound implementation, when the tree is being built the program will check if the next child to be added is a valid prefix to a word, and it is not it will not be added. For example if the input sequence is 948 the program will first check if w,x,y,z are prefixes to valid English words, and of course they are. So now we look the children, the prefix "wg" is not valid so g will not be added to the tree. Next it will check "wh" which is a prefix to many words like "which","where","who" just to name a few. Because it is valid it is added to the tree. Please note that I am aware branch and bounds is traditionally implemented on the retrieval of the words rather than the building of the trie, but in this program larger words above about 13 letters in length will throw Java out of memory Exceptions because it will have to create 4^n full paths. This was my method to prevent this.



The “wg” prefix is not added. The “wh” and “wi” are added because there are a lot of words that they are a prefix to. Then the next children are added each being checked if isPrefix. I did not show all of them just demonstrating the concept. For larger input sequences the effect is much larger. If the user entered “78737342425” where the length (n) is 11. it first creates “p”, then check “pt” is prefix which it isn’t. because that node is never created. 4^(11-2) operations are not performed to make the tree. (because already on level 2). That’s 262,144 operations saved.

Output:

T9 words menu

1.) Test using a hashTable dictionary

2.) Test a trie dictionary

3.) Test Branch and Bounds

4.) Run Complete Comparative test

5.) Test dictionary(enter word)

6.) Exit

Please enter a menu option:4

******Running 5 tests using all 3 implementations of the solver******

Testing sequence: [2, 2, 2]

Testing using HashTable Dictionary

Method Ran	Time Elapsed
Build time	4,965 nano-seconds
Get all Words	8,688 nano-seconds
Total	0.014 milli-seconds

Printing all valid words:

aac abb abc aca acc

bbc

cab

Testing using Trie Dictionary

Method Ran	Time Elapsed
Build time	1,862 nano-seconds
Get all Words	12,102 nano-seconds
Total	0.014 milli-seconds

Printing all valid words:

aac abb abc aca acc

bbc

cab

Testing using Branch and bounds

Method Ran	Time Elapsed
Branch And Bounds Build Time:	12,722 nano-seconds
Get all Valid Words	8,688 nano-seconds
Total	0.021 milli-seconds

Printing all valid words:

aac abb abc aca acc

bbc

cab

Testing sequence: [3, 3, 3, 3]

Testing using HashTable Dictionary

Method Ran	Time Elapsed
Build time	4,344 nano-seconds
Get all Words	14,584 nano-seconds
Total	0.019 milli-seconds

Printing all valid words:

feed

Testing using Trie Dictionary

Method Ran	Time Elapsed
Build time	4,034 nano-seconds
Get all Words	15,205 nano-seconds
Total	0.019 milli-seconds

Printing all valid words:

feed

Testing using Branch and bounds

Method Ran	Time Elapsed
Branch And Bounds Build Time:	16,135 nano-seconds
Get all Valid Words	4,655 nano-seconds
Total	0.021 milli-seconds

Printing all valid words:

feed

Testing sequence: [8, 3, 7, 8, 4, 6, 4]

Testing using HashTable Dictionary

Method Ran	Time Elapsed
Build time	110,158 nano-seconds
Get all Words	368,951 nano-seconds

Total 0.479 milli-seconds

Printing all valid words:

testing

Testing using Trie Dictionary

Method Ran Time Elapsed

Build time 108,606 nano-seconds

Get all Words 267,481 nano-seconds

Total 0.376 milli-seconds

Printing all valid words:

testing

Testing using Branch and bounds

Method Ran Time Elapsed

Branch And Bounds Build Time: 16,446 nano-seconds

Get all Valid Words 5,276 nano-seconds

Total 0.022 milli-seconds

Printing all valid words:

testing

Testing sequence: [9, 9, 2, 2, 6, 3, 6, 3, 2, 3, 5]

Testing using HashTable Dictionary

Method Ran Time Elapsed

Build time 13,564,299 nano-seconds

Get all Words 42,786,203 nano-seconds

Total 56.351 milli-seconds

Printing all valid words:

zwaanendael

Testing using Trie Dictionary

Method Ran	Time Elapsed
Build time	12,463,341 nano-seconds
Get all Words	52,225,326 nano-seconds
Total	64.689 milli-seconds

Printing all valid words:

zwaanendael

Testing using Branch and bounds

Method Ran	Time Elapsed
Branch And Bounds Build Time:	19,859 nano-seconds
Get all Valid Words	7,447 nano-seconds
Total	0.027 milli-seconds

Printing all valid words:

zwaanendael

Testing sequence: [7, 8, 7, 3, 7, 3, 4, 2, 4, 2, 5]

Testing using HashTable Dictionary

Method Ran	Time Elapsed
Build time	15,470,494 nano-seconds
Get all Words	54,693,171 nano-seconds

Total 70.164 milli-seconds

Printing all valid words:

superficial

Testing using Trie Dictionary

Method Ran	Time Elapsed
Build time	17,124,721 nano-seconds
Get all Words	54,394,038 nano-seconds
Total	71.519 milli-seconds

Printing all valid words:

superficial

Testing using Branch and bounds

Method Ran	Time Elapsed
Branch And Bounds Build Time:	35,995 nano-seconds
Get all Valid Words	9,930 nano-seconds
Total	0.046 milli-seconds

Printing all valid words:

superficial

Average time using hash dictionary: 25.405

Average time using trie dictionary: 27.323

Average time using branch and bounds: 0.027

Branch and bounds is 926.162 times faster than hashtable dictionary

Branch and bounds is 996.090 times faster than trie dictionary