**Word Squares**

Given an array of **unique** strings words, return *all the*[**word squares**](https://en.wikipedia.org/wiki/Word_square)*you can build from*words. The same word from words can be used **multiple times**. You can return the answer in **any order**.

A sequence of strings forms a valid **word square** if the kth row and column read the same string, where 0 <= k < max(numRows, numColumns).

* For example, the word sequence ["ball","area","lead","lady"] forms a word square because each word reads the same both horizontally and vertically.

**Example 1:**

**Input:** words = ["area","lead","wall","lady","ball"]

**Output:** [["ball","area","lead","lady"],["wall","area","lead","lady"]]

**Explanation:**

The output consists of two word squares. The order of output does not matter (just the order of words in each word square matters).

**Example 2:**

**Input:** words = ["abat","baba","atan","atal"]

**Output:** [["baba","abat","baba","atal"],["baba","abat","baba","atan"]]

**Explanation:**

The output consists of two word squares. The order of output does not matter (just the order of words in each word square matters).

**Constraints:**

* 1 <= words.length <= 1000
* 1 <= words[i].length <= 4
* All words[i] have the same length.
* words[i] consists of only lowercase English letters.
* All words[i] are **unique**.

/\*\*

\* @param {string[]} words

\* @return {string[][]}

\*/

var wordSquares = function(words) {

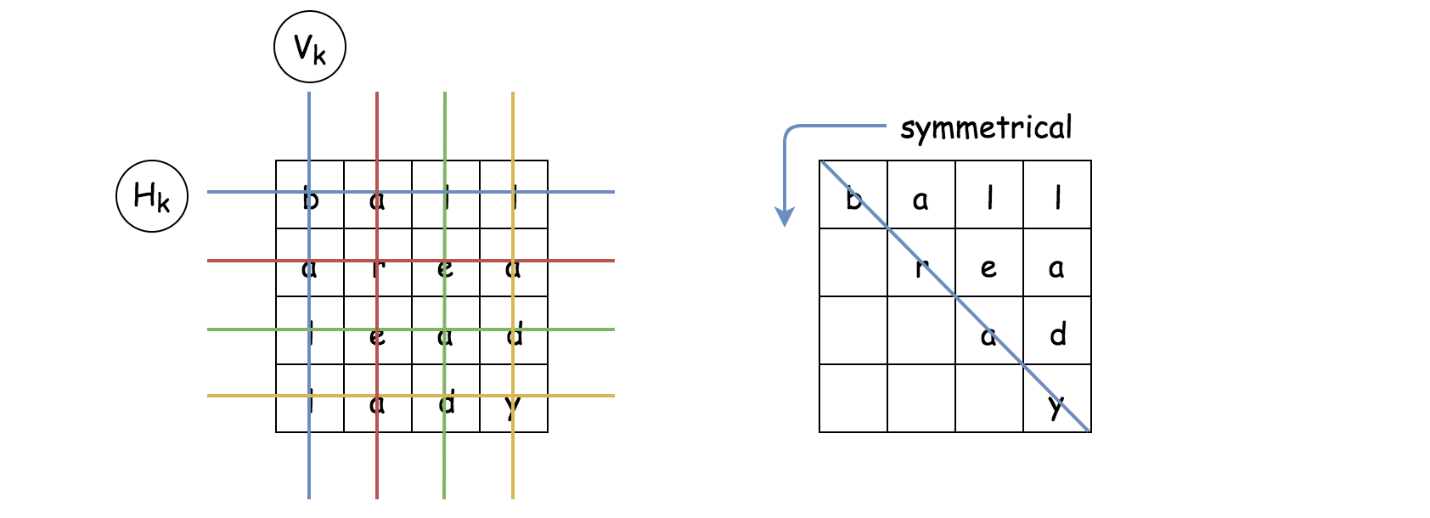
};

Solution

Before diving into the solutions, it could be helpful to take a step back and clarify the requirements of the problem first.

Given a list of non-duplicate words, we are asked to construct all possible combinations of word squares. And here is the definition of ***word square***.

A sequence of words forms a valid *word square*, if and only if each string (H\_k*Hk*​) that is formed horizontally from the kth row equals to the string (V\_k*Vk*​) that is formed vertically from the kth column, *i.e.* H\_k == V\_k \qquad \forall{k} \quad 0 ≤ k < \max(\text{numRows}, \text{numColumns}).*Hk*​==*Vk*​∀*k*0≤*k*<max(numRows,numColumns).



As we can see from the definition, for a word square with equal-sized row and column, the resulting letter matrix should be **symmetrical** across its diagonal.

In other words, if we know the upper-right part of the word square, we could infer its lower-left part, and vice versa. This symmetric property of the word square could also be interpreted as the **constraint** of the problem (as in the *constraint programming*), which could help us to narrow down the valid combinations.

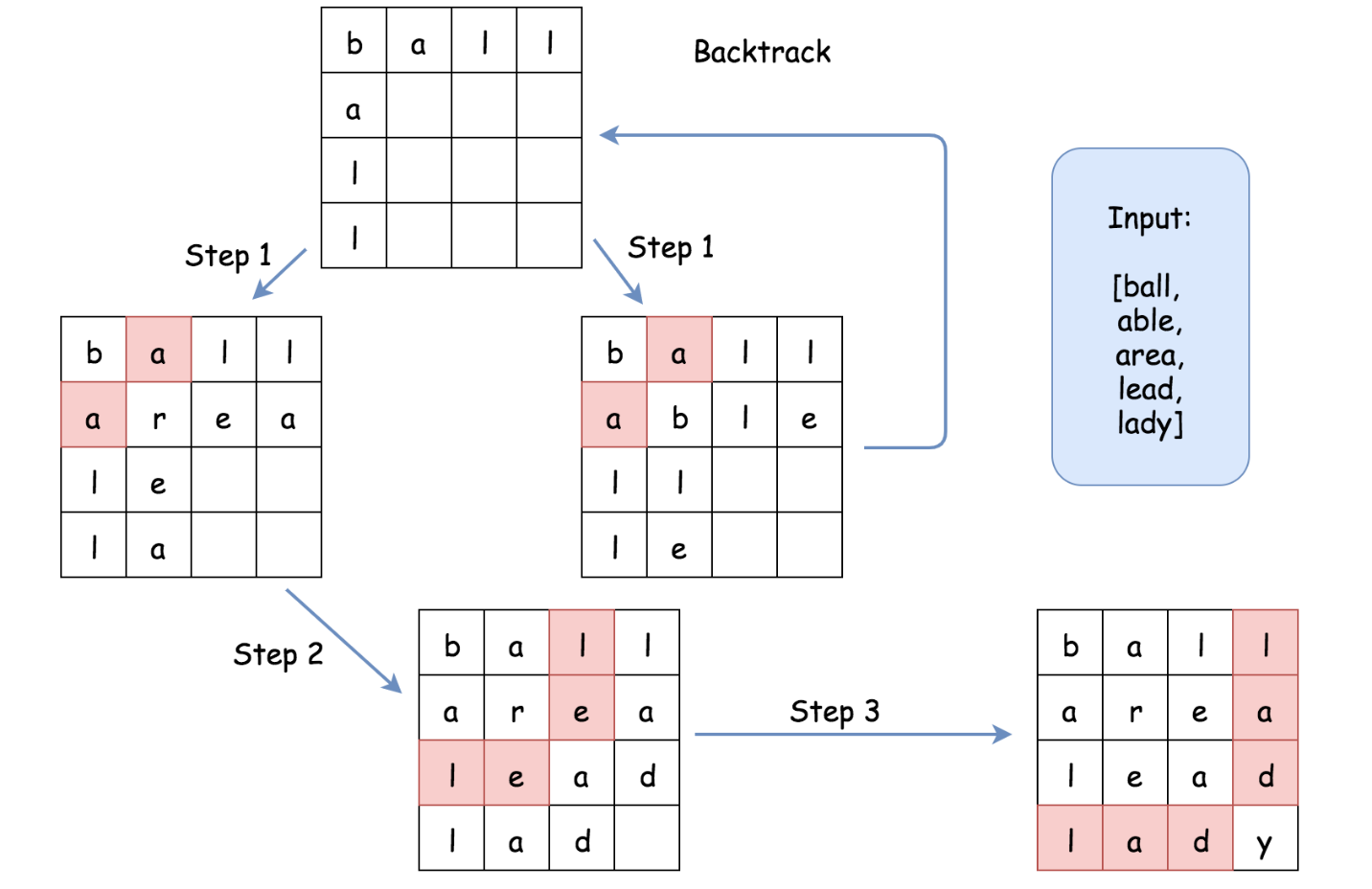
Algorithm: Backtracking

Given a list of words, we are asked to find a combination of words upon with we could construct a *word square*. The backbone of the algorithm to solve the above problem could be surprisingly simple.

The idea is that we construct the word square ***row by row*** from top to down. At each row, we simply do *trial and error*, *i.e.* we try with one word, if it does not meet the constraint then we try another one.

As one might notice, the above idea of the algorithm is actually known as [backtracking](https://leetcode.com/explore/learn/card/recursion-ii/472/backtracking/2654/), which is often associated with *recursion* and *DFS* (Depth-First Search) as well.

Let us illustrate the idea with an example. Given a list of words [ball, able, area, lead, lady], we should pile up 4 words together in order to build a word square.



* Let us start with the word ball as the first word in the word square, *i.e.* the word that we would put in the first row.
* We then move on to the second row. Given the *symmetric* property of the word square, we now know the letters that we should fill on the first *column* of the second row. In other words, we know that the word in the second row should start with the *prefix* a.
* Among the list of words, there are two words with prefix a (*i.e.* able, area). Both of them could be the candidates to fill the second row of the square. We then should try both of them in the next step.
* In the next step (1), let us fill the second row with the word able. Then we could move on to the third row. Again, due to the symmetric property, we know that the word in the third row should start with the prefix ll. Unfortunately, we do not find any word start with ll. As a result, we could no longer move forwards. We then abandon this path, and ***backtrack*** to the previous state (with the first row filled).
* As an alternative next step (1), we could try with the word area in the second row. Once we fill the second row, we would know that in the next row, the word to be filled should start with the prefix le. And this time, we find the candidate (*i.e.* lead).
* As a result, in the next step (2), we fill the third row with the word lead. So on and so forth.
* *At the end, if one repeats the above steps with each word as the starting word, one would exhaust all the possibilities to construct a valid word square*.

One could follow the [code template of backtracking](https://leetcode.com/explore/learn/card/recursion-ii/472/backtracking/2793/) from our Explore card to implement the above algorithm. Here is one example.

class Solution:

def wordSquares(self, words: List[str]) -> List[List[str]]:

self.words = words

self.N = len(words[0])

results = []

word\_squares = []

for word in words:

# try with every word as the starting word

word\_squares = [word]

self.backtracking(1, word\_squares, results)

return results

def backtracking(self, step, word\_squares, results):

if step == self.N:

results.append(word\_squares[:])

return

prefix = ''.join([word[step] for word in word\_squares])

# find out all words that start with the given prefix

for candidate in self.getWordsWithPrefix(prefix):

# iterate row by row

word\_squares.append(candidate)

self.backtracking(step+1, word\_squares, results)

word\_squares.pop()

def getWordsWithPrefix(self, prefix):

for word in self.words:

if word.startswith(prefix):

yield word

At the first glance of the code, the problem does not seem to be as daunting as it is labeled. Actually if one could come up the skeleton of code in the interview, it would be fair to say that one has *scored* the interview.

The above implementation is correct and would pass most of the test cases in the online judge. However, it would run into Time Limit Exceeded exception for certain test cases with large input. But, there is no need for dismay, since we've already figured out the *hard* part of the algorithm. We just need to iron out the last bit of *optimization* which actually could be a followup question during the interview.

Approach 1: Backtracking with HashTable

**Intuition**

As one might notice in the above backtracking algorithm, the bottleneck lies in the function getWordsWithPrefix() which is to find all words with the given prefix. At each invocation of the function, we were iterating through the entire input list of words, which is of linear time complexity \mathcal{O}(N)O(*N*).

One of the ideas to optimize the getWordsWithPrefix() function would be to process the words beforehand and to build a data structure that could speed up the lookup procedure later.

As one might recall, one of the data structures that provide a fast lookup operation is called ***hashtable*** or dictionary. We could simply build a hashtable with all possible prefixes as *keys* and the words that are associated with the prefix as the *values* in the table. Later, given the prefix, we should be able to list all the words with the given prefix in constant time \mathcal{O}(1)O(1).

**Algorithm**

* We build upon the backtracking algorithm that we listed above, and tweak two parts.
* In the first part, we add a new function buildPrefixHashTable(words) to build a hashtable out of the input words.
* Then in the second part, in the function getWordsWithPrefix() we simply query the hashtable to retrieve all the words that possess the given prefix.

Here are some sample implementations. The idea is inspired by the post of [gabbu](https://leetcode.com/problems/word-squares/discuss/91360/3-Python-Solutions-with-very-detailed-explanations) in the discussion forum.

class Solution {

int N = 0;

String[] words = null;

HashMap<String, List<String>> prefixHashTable = null;

public List<List<String>> wordSquares(String[] words) {

this.words = words;

this.N = words[0].length();

List<List<String>> results = new ArrayList<List<String>>();

this.buildPrefixHashTable(words);

for (String word : words) {

LinkedList<String> wordSquares = new LinkedList<String>();

wordSquares.addLast(word);

this.backtracking(1, wordSquares, results);

}

return results;

}

protected void backtracking(int step, LinkedList<String> wordSquares,

List<List<String>> results) {

if (step == N) {

results.add((List<String>) wordSquares.clone());

return;

}

StringBuilder prefix = new StringBuilder();

for (String word : wordSquares) {

prefix.append(word.charAt(step));

}

for (String candidate : this.getWordsWithPrefix(prefix.toString())) {

wordSquares.addLast(candidate);

this.backtracking(step + 1, wordSquares, results);

wordSquares.removeLast();

}

}

protected void buildPrefixHashTable(String[] words) {

this.prefixHashTable = new HashMap<String, List<String>>();

for (String word : words) {

for (int i = 1; i < this.N; ++i) {

String prefix = word.substring(0, i);

List<String> wordList = this.prefixHashTable.get(prefix);

if (wordList == null) {

wordList = new ArrayList<String>();

wordList.add(word);

this.prefixHashTable.put(prefix, wordList);

} else {

wordList.add(word);

}

}

}

}

protected List<String> getWordsWithPrefix(String prefix) {

List<String> wordList = this.prefixHashTable.get(prefix);

return (wordList != null ? wordList : new ArrayList<String>());

}

}

**Complexity Analysis**

* Time complexity: \mathcal{O}(N\cdot26^{L})O(*N*⋅26*L*), where N*N* is the number of input words and L*L* is the length of a single word.
  + It is tricky to calculate the exact number of operations in the backtracking algorithm. We know that the trace of the backtrack would form a n-ary tree. Therefore, the upper bound of the operations would be the total number of nodes in a full-blossom n-ary tree.
  + In our case, at any node of the trace, at maximum it could have 26 branches (*i.e.* 26 alphabet letters). Therefore, the maximum number of nodes in a 26-ary tree would be approximately 26^L26*L*.
  + In the loop around the backtracking function, we enumerate the possibility of having each word as the starting word in the word square. As a result, in total the overall time complexity of the algorithm should be \mathcal{O}(N\cdot26^{L})O(*N*⋅26*L*).
  + As large as the time complexity might appear, in reality, the actual trace of the backtracking is much smaller than its upper bound, thanks to the *constraint* checking (symmetric of word square) which greatly prunes the trace of the backtracking.
* Space Complexity: \mathcal{O}(N\cdot{L} + N\cdot\frac{L}{2}) = \mathcal{O}(N\cdot{L})O(*N*⋅*L*+*N*⋅2*L*​)=O(*N*⋅*L*) where N*N* is the number of words and L*L* is the length of a single word.
  + The first half of the space complexity (*i.e.* N\cdot{L}*N*⋅*L*) is the values in the hashtable, where we store L*L* times all words in the hashtable.
  + The second half of the space complexity (*i.e.* N\cdot\frac{L}{2}*N*⋅2*L*​) is the space took by the keys of the hashtable, which include all prefixes of all words.
  + In total, we could say that the overall space of the algorithm is proportional to the total words times the length of a single word.

Approach 2: Backtracking with Trie

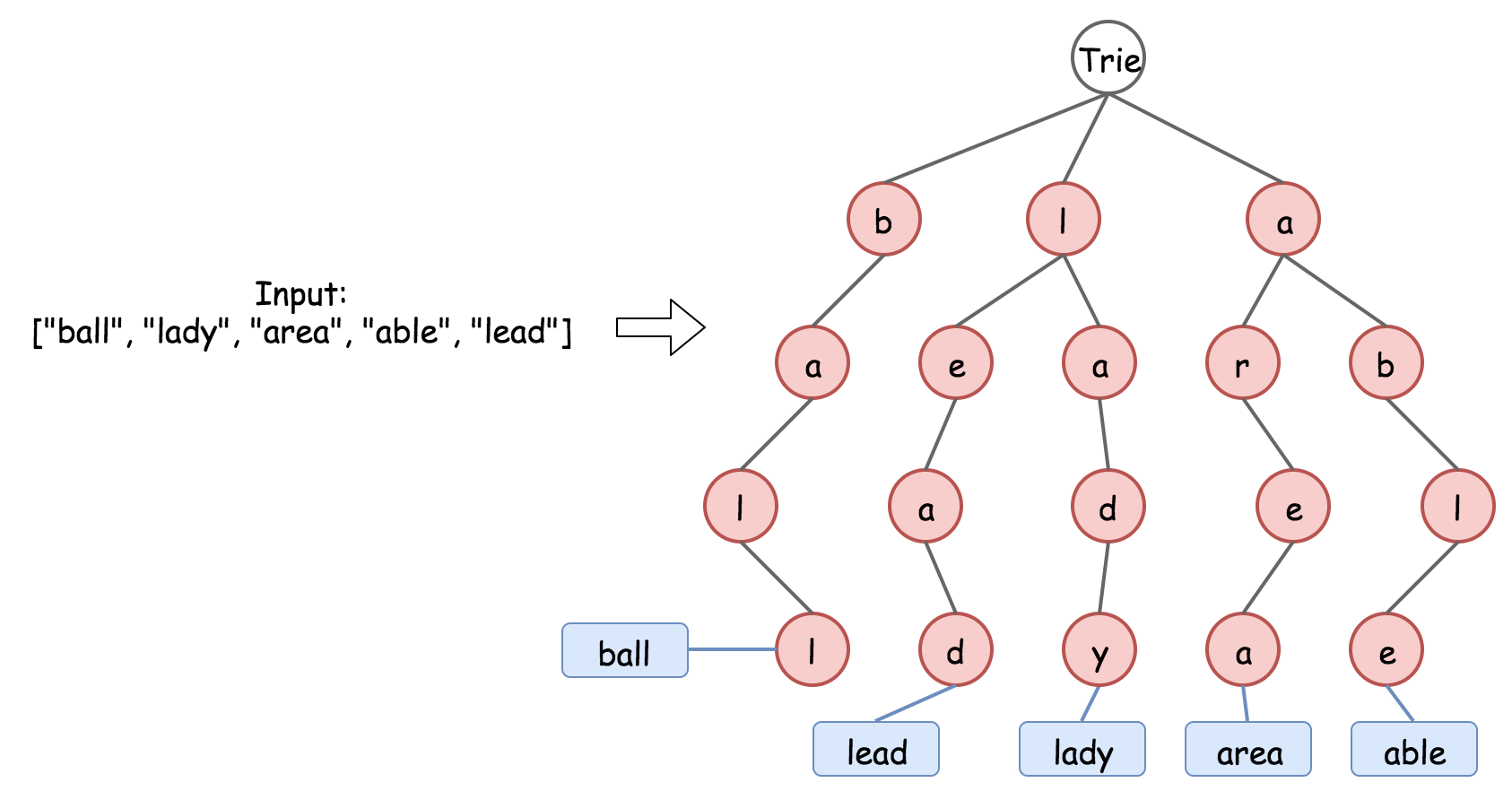
**Intuition**

Speaking about prefix, there is another data structure called ***Trie*** (also known as ***prefix tree***), which could find its use in this problem.

In the above approach, we have reduce the time complexity of retrieving a list of words with a given prefix from the linear \mathcal{O}(N)O(*N*) to the constant time \mathcal{O}(1)O(1). In exchange, we have to spend some extra space to store all the prefixes of each words.

The Trie data structure provides a *compact* and yet still *fast* way to retrieve words with a given prefix.

In the following graph, we show an example on how we can build a Trie from a list of words.



As we can see, basically Trie is a n-aray tree, where each node represents a character in a prefix. The Trie data structure is **compact** to store the prefixes, since it deduplicate the redundant prefixes, *e.g.* the prefixes of le and la would share one node. And yet it is still fast to retrieve words from the Trie. It takes \mathcal{O}(L)O(*L*) to retrieve a word, where L*L* is the length of the word, which is much faster than the brute-force enumeration.

**Algorithm**

* We build upon the backtracking algorithm that we listed above, and tweak two parts.
* In the first part, we add a new function buildTrie(words) to build a Trie out of the input words.
* Then in the second part, in the function getWordsWithPrefix(prefix) we simply query the Trie to retrieve all the words that possess the given prefix.

Here are some sample implementations. Note that, we tweak the Trie data structure a bit, in order to further optimize the time and space complexity.

* Instead of labeling the word at the leaf node of the Trie, we label the word at each node so that we don't need to perform a further traversal once we reach the last node in the prefix. This trick could help us with the time complexity.
* Instead of storing the actual words in the Trie, we keep only the index of the word, which could greatly save the space.

class TrieNode {

HashMap<Character, TrieNode> children = new HashMap<Character, TrieNode>();

List<Integer> wordList = new ArrayList<Integer>();

public TrieNode() {}

}

class Solution {

int N = 0;

String[] words = null;

TrieNode trie = null;

public List<List<String>> wordSquares(String[] words) {

this.words = words;

this.N = words[0].length();

List<List<String>> results = new ArrayList<List<String>>();

this.buildTrie(words);

for (String word : words) {

LinkedList<String> wordSquares = new LinkedList<String>();

wordSquares.addLast(word);

this.backtracking(1, wordSquares, results);

}

return results;

}

protected void backtracking(int step, LinkedList<String> wordSquares,

List<List<String>> results) {

if (step == N) {

results.add((List<String>) wordSquares.clone());

return;

}

StringBuilder prefix = new StringBuilder();

for (String word : wordSquares) {

prefix.append(word.charAt(step));

}

for (Integer wordIndex : this.getWordsWithPrefix(prefix.toString())) {

wordSquares.addLast(this.words[wordIndex]);

this.backtracking(step + 1, wordSquares, results);

wordSquares.removeLast();

}

}

protected void buildTrie(String[] words) {

this.trie = new TrieNode();

for (int wordIndex = 0; wordIndex < words.length; ++wordIndex) {

String word = words[wordIndex];

TrieNode node = this.trie;

for (Character letter : word.toCharArray()) {

if (node.children.containsKey(letter)) {

node = node.children.get(letter);

} else {

TrieNode newNode = new TrieNode();

node.children.put(letter, newNode);

node = newNode;

}

node.wordList.add(wordIndex);

}

}

}

protected List<Integer> getWordsWithPrefix(String prefix) {

TrieNode node = this.trie;

for (Character letter : prefix.toCharArray()) {

if (node.children.containsKey(letter)) {

node = node.children.get(letter);

} else {

// return an empty list.

return new ArrayList<Integer>();

}

}

return node.wordList;

}

}

**Complexity Analysis**

* Time complexity: \mathcal{O}(N\cdot26^{L}\cdot{L})O(*N*⋅26*L*⋅*L*), where N*N* is the number of input words and L*L* is the length of a single word.
  + Basically, the time complexity is same with the Approach #1 (\mathcal{O}(N\cdot26^{L})O(*N*⋅26*L*)), except that instead of the constant-time access for the function getWordsWithPrefix(prefix), we now need \mathcal{O}(L)O(*L*).
* Space Complexity: \mathcal{O}(N\cdot{L} + N\cdot\frac{L}{2}) = \mathcal{O}(N\cdot{L})O(*N*⋅*L*+*N*⋅2*L*​)=O(*N*⋅*L*) where N*N* is the number of words and L*L* is the length of a single word.
  + The first half of the space complexity (*i.e.* N\cdot{L}*N*⋅*L*) is the word indice that we store in the Trie, where we store L*L* times the index for each word.
  + The second half of the space complexity (*i.e.* N\cdot\frac{L}{2}*N*⋅2*L*​) is the space took by the prefixes of all words. In the worst case, we have no overlapping among the prefixes.
  + Overall, this approach has the same space complexity as the previous approach. Yet, in running time, it would consume less memory thanks to all the optimization that we have done.