# Question

Given a list of **unique** words, return all the pairs of the ***distinct*** indices (i, j) in the given list, so that the concatenation of the two words words[i] + words[j] is a palindrome.

**Example 1:**

**Input:** words = ["abcd","dcba","lls","s","sssll"]

**Output:** [[0,1],[1,0],[3,2],[2,4]]

**Explanation:** The palindromes are ["dcbaabcd","abcddcba","slls","llssssll"]

**Example 2:**

**Input:** words = ["bat","tab","cat"]

**Output:** [[0,1],[1,0]]

**Explanation:** The palindromes are ["battab","tabbat"]

**Example 3:**

**Input:** words = ["a",""]

**Output:** [[0,1],[1,0]]

**Constraints:**

* 1 <= words.length <= 5000
* 0 <= words[i].length <= 300
* words[i] consists of lower-case English letters.

# Solution

Here's a few words of advice before we get started.

This is a very popular interview question. A concern I've seen brought up on the forums is that this question is too big to do in an interview.

Keep in mind though, that you're being compared to other candidates. They too will struggle with this, unless they've seen it before and memorized it. This however will be obvious to an experienced interviewer. It is the candidate who has clearly never seen it before yet makes great progress (probably not writing a complete implementation) who will be considered the most impressive. The secret would be to prioritize your time so that you are focusing on the core of the problem and not implementations of straightforward helper methods.

For this question, great progress would probably be deriving the intuition discussed in approach 2 and then writing code for the core algorithm of Approach 2 or Approach 3.

Remember that you don't necessarily have to "implement" every helper method. For example, some implementations rely on checking if a part of a string is a palindrome. This detail is easy-level by Leetcode standards, and in particular if you're using a whiteboard, it's a waste of time and space to write it unless you have finished the core algorithm. Simply state how you'd do it and leave it as a method signature unless asked to do otherwise. Also (for Approach 3), keep the TrieNode class simple. Don't waste half the whiteboard writing getters and setters for it.

#### **Approach 1: Brute force**

**Intuition**

The brute force solution is a good place to start. For this question, it is straightforward. Iterate over every possible pair of strings and check whether or not they form a palindrome.

You probably won't be writing this code, there simply won't be time. But make sure you know what it would be, and that you could describe the algorithm line-by-line if needed.

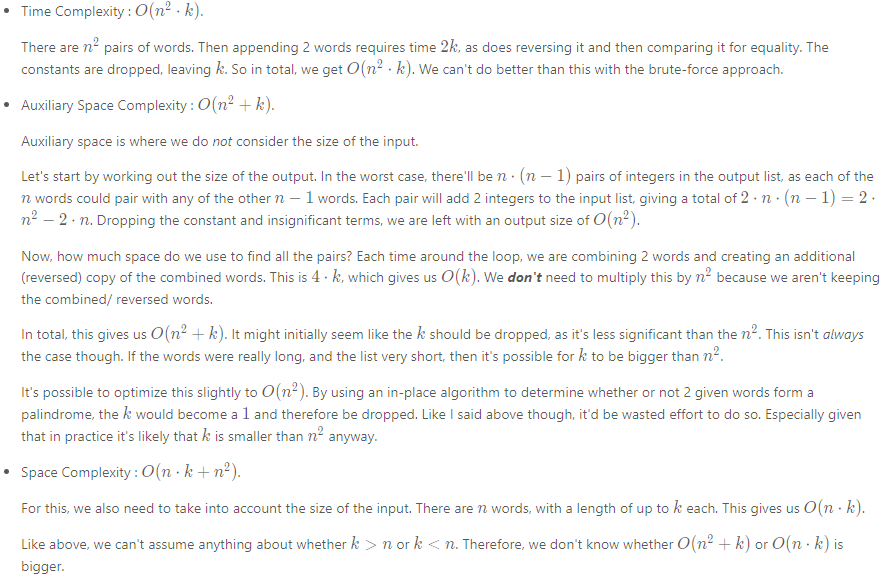
**Algorithm**

We can do this using 2 nested loops, each loop going over each index in the array. For each pair we need to check whether or not it forms a palindrome. There are many ways of doing this step, here I recommend the simplest way: creating the combined word and the reversed combined word and checking if they're equal. Doing the check in a more efficient way at this stage is not worth it — we want to focus our efforts on optimizing the main inefficiencies in this algorithm, which are discussed further in the complexity analysis section.

**An important edge case to be careful of** is where i = j. The problem states that i and j must be distinct (in other words, not the same). Identifying this edge case now is important, because we'll also need to be careful of it when we are optimizing our algorithm.

**Complexity Analysis**

Let *n* be the number of words, and k*k* be the length of the longest word.



#### **Approach 2: Hashing**

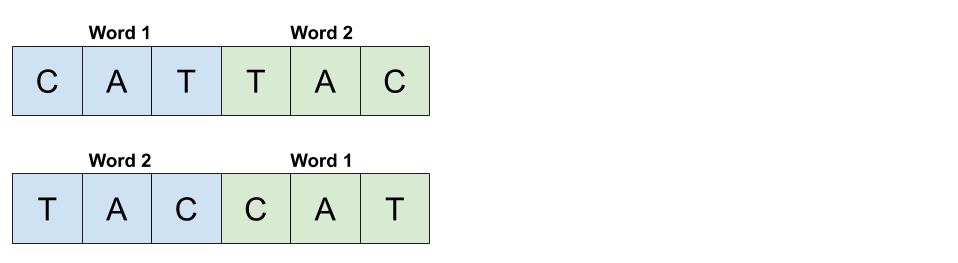
**Intuition**

Testing every pair is too expensive. Is there a way we can avoid checking pairs that will definitely not form a palindrome? Inorder to answer this question, we'll need to explore the properties of pairs that do form a palindrome.

This type of exploration and reasoning can be a bit challenging if you're not used to it, so we'll tackle it with some examples and then we'll try and prove our discoveries more formally. After that, we'll take a look at how it could be implemented efficiently in code.

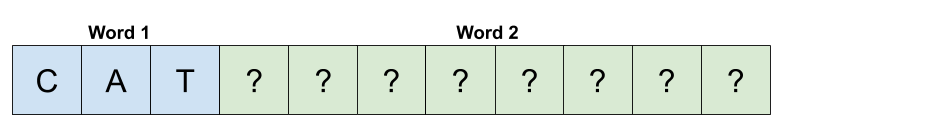
What are the ways we could form a palindrome with 2 words?

The simplest way to make a palindrome is to take 2 words that are the reverse of each other and put them together. In this case, we get 2 different palindromes, as we can put either word first.

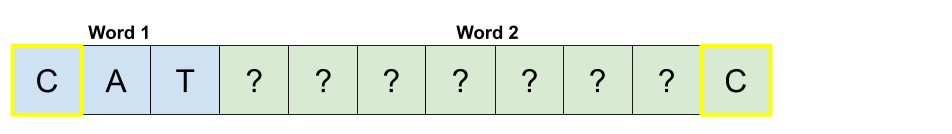


We know that there are always 2 unique palindromes that can be formed by 2 words that are the reverse of each other, because the words must be different. The problem statement is clear that there are no duplicates in the word list.

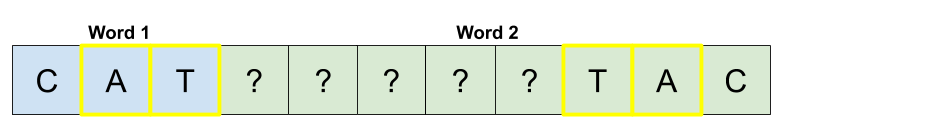
Let's now think about all the words that could pair with a word 1 of "CAT" to make a palindrome. We'll assume that all the possibilities for word 2 we're looking at are 8 letters long. While this assumption might seem too specific, remember that we're just using it as a starting point to identify possible cases. We'll do a more general proof later.



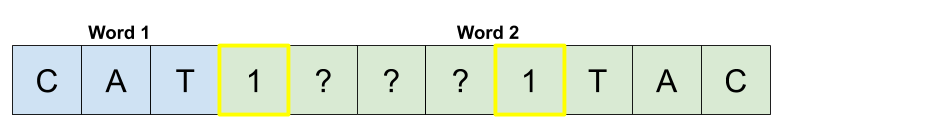
To start with, we know that the last letter of word 2 has to be "C". Otherwise, it would be impossible to form a palindrome.



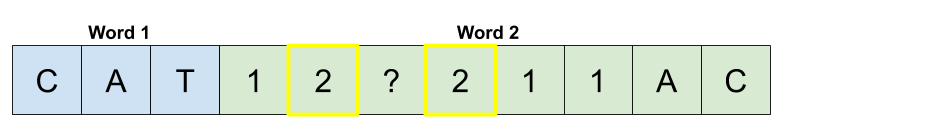
By that same logic, we also know the 2nd to last and 3rd to last characters must be "A" and "T" respectively.



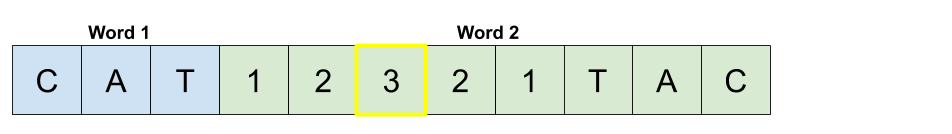
Here's where things start to get a bit interesting. We know that the 2 letters highlighted in the next diagram must be the same for the combined word to be a palindrome. We'll use numbers to show where letters must be the same.



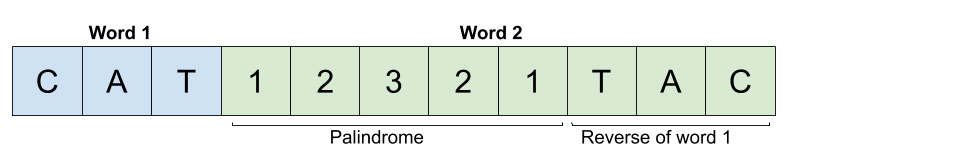
The same argument applies for the next pair of highlighted letters.



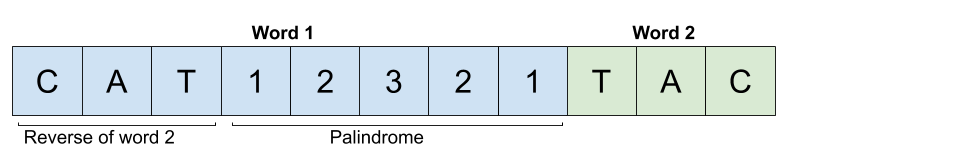
And that last letter in the center can be anything.



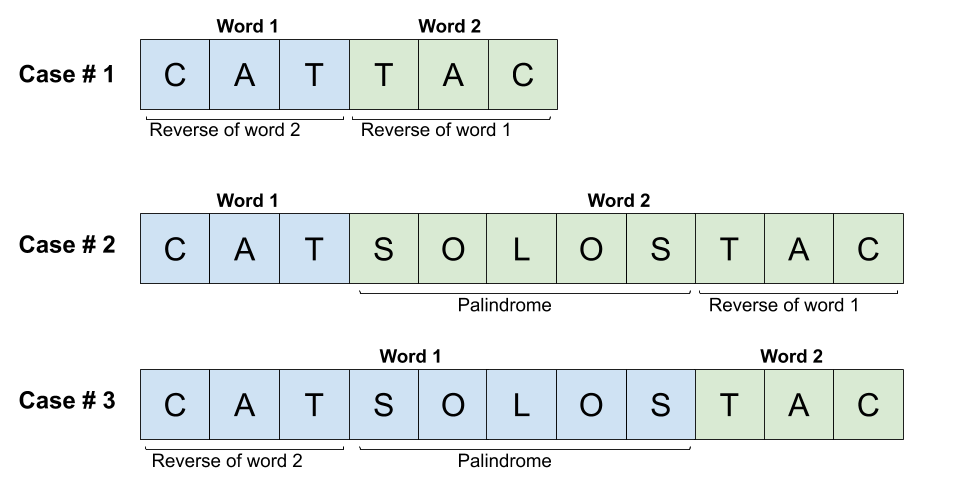
Let's now take a step back and see what we have. Our experimenting has shown us that if word 2 is the concatenation of a 5-letter palindrome and then the reverse of word 1, that the combined pair of word 1 and word 2 is a palindrome.



Another case can also be seen here. If instead word 1 was the concatenation of the reverse of word 2 and then a 5-letter palindrome, the combined pair of word 1 and word 2 would also be a palindrome.



We have now identified 3 cases.



Don't forget that the empty string is also a valid word. How could we form a palindrome with it? This is an important edge case we'll now think about.

Appending an empty string with another word will simply give the non-empty string word. If this word was a palindrome by itself, we will have a valid palindrome pair. If it wasn't, we won't. So any words that by themselves are a palindrome will form a palindrome pair with the empty string.

Depending on the implementation you use, you might not need to treat this as a special case, as it is really just a sub case of **case 2** and **case 3**. It's just that the bit that is reversed is of length-0. Make sure to test your implementation on this case though!

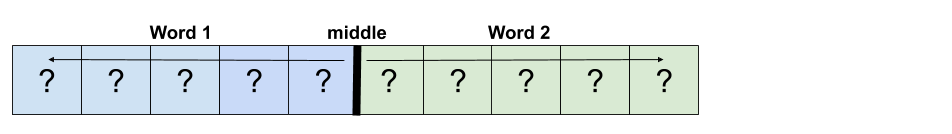
How can we prove that we have identified all the cases?

By experimenting, we've discovered a few cases. But for these kinds of questions, it's very important to convince ourselves that we haven't overlooked any cases. One way we can do this is by considering the relative length of each pair of words. There are 2 cases for the relative lengths within each pair.

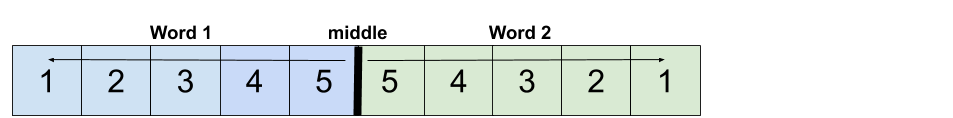
1. The words are both of the same length.
2. The words are of different lengths.

We then need to show how each of these 2 cases fully map onto the palindrome pair cases we've already discovered. We'll do this by considering where the middle of the combined word (word we get by appending the second word to the first word) is.

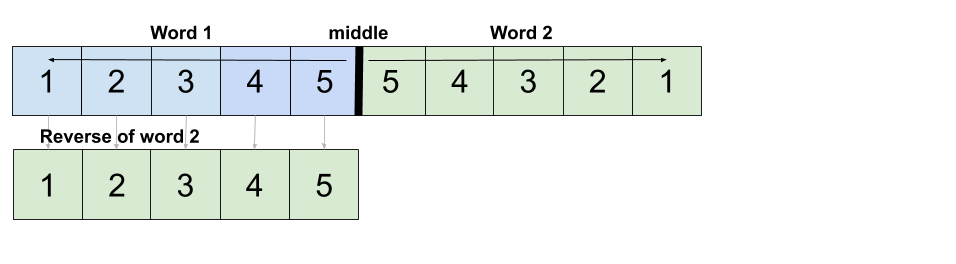
For the first possibility, the center of the combined word is between the two words.



For the pair to form a palindrome, the letters before the center must be the reverse of the letters after the center. The following diagram uses numbers to show where 2 letters must be the same.



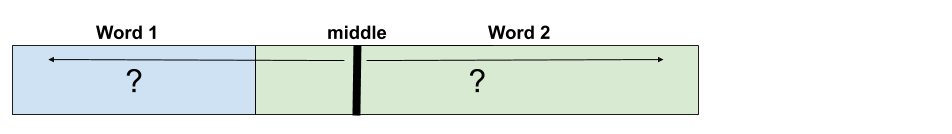
We can also see that this means word 1 must be the reverse of word 2.



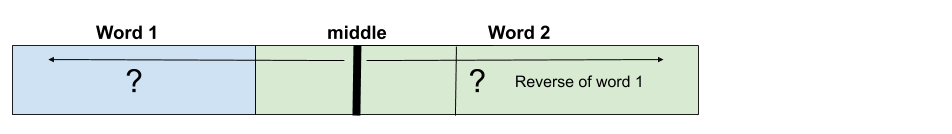
Therefore, when 2 words of the same length form a palindrome, it must be because word 1 is the reverse of word 2 (which also means word 2 is the reverse of word 1). This is equivalent to palindrome pair **case 1**.

For the second relative word-length case, we know that one of the words must be shorter than the other. We'll assume for now that word 1 is shorter. The exact same argument will make will also apply for when word 2 is shorter.

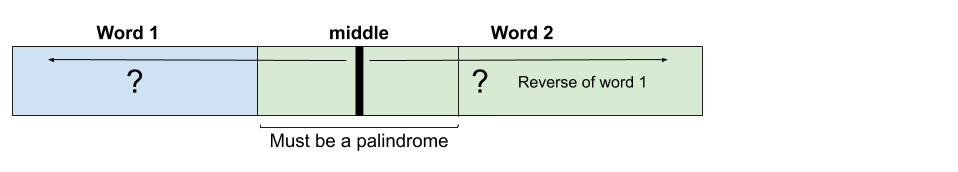
Like before, there must be a middle of the combined word. We know that because word 1 is shorter, word 2 will overlap this center point.



We know that a palindrome must mirror around that center point. Therefore, we know that the end of word 2 must be the reverse of word 1.



We are now left with the region between word 1 and the reverse of word 1. We know that this middle region is divided equally in 2 by the middle line because we took the same number of characters off each end of the combined word. Therefore, for the overall combined word to be a palindrome, the piece in the middle must be a palindrome.



Which is equivalent to palindrome-pair **case 2**.

Using this same line of reasoning, you can easily show that when word 2 is shorter, it is equivalent to palindrome pair **case 3**.

Therefore, we have proven that the only possible ways of forming a palindrome pair out of 2 words are covered by the 3 palindrome-pair cases we discovered during our exploration.

How can we put all this into code?

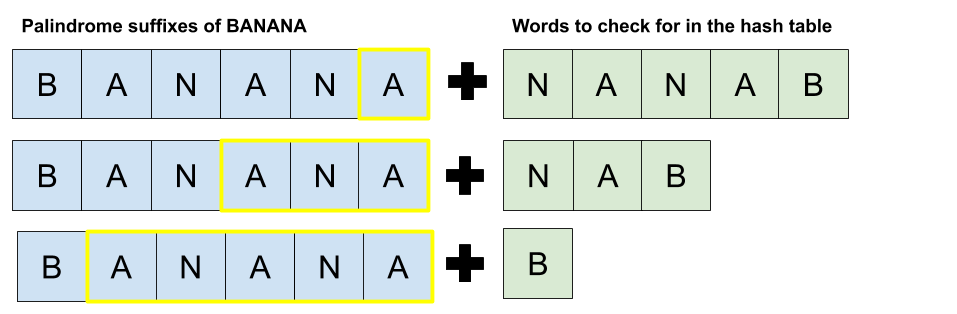
The simplest way to put all of this into code is to iterate over the list of words and do the following for each **word**.

If these initial explanations are confusing, don't panic. There's further examples just below the list.

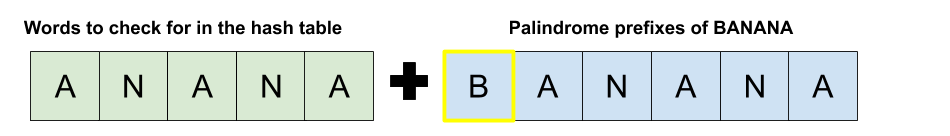
1. Check if the reverse of **word** is present. If it is, then we have a **case 1** pair by appending the reverse onto the end of **word**.
2. For each **suffix** of **word**, check if the **suffix** is a palindrome. **if it is a palindrome**, then reverse the remaining **prefix** and check if it's in the list. If it is, then this is an example of **case 2**.
3. For each **prefix** of **word**, check if the **prefix** is a palindrome. **if it is a palindrome**, then reverse the remaining **suffix** and check if it's in the list. If it is, then this is an example of **case 3**.

For example, imagine we have the word "banana". Start by checking whether or not "ananab" is in the list.

Now identify all palindrome suffixes of "banana". For each one, we take the remaining prefix, reverse it, and check if we have that word in the list.



Do the same for all palindrome prefixes of "banana". There is only one of these.



If we do this for each word, we will get all palindrome pairs exactly once. The most challenging idea here is that we are treating our current word as word 2 for case 2. The reason we do this is because treating it as word 1 would mean we had to guess possible prefixes for word 2, which would be very, very inefficient.

To ensure the implementation is efficient, we can put all the words into a hash table with the word as the key and the original index as the value (as the output must be the original indexes of the words).

**Algorithm**

We'll call a suffix a "valid suffix" of a word if the remainder (prefix) of the word forms a palindrome. The function allValidSuffixes finds all such suffixes. For example, the "valid suffixes" of the word "exempt" are "xempt" (remove "e") and "mpt" (remove 'exe').

We'll call a prefix a "valid prefix" of a word if the remainder (suffix) of the word forms a palindrome. The function allValidPrefixes finds all such prefixes in a similar way to how the allValidSuffixes function does. It is possible to combine more of the code for these functions here, but after going back and forth on the issue, I decided against it for this explanation because while it decreases the length of the code and some repetition, the cognitive load to understand it is higher. In your own code, it would be fine to combine it.

Examples of case 1 can be found by reversing the current word and looking it up. One edge case to be careful of is that if a word is a palindrome by itself, then we don't want to add a pair that includes that same word twice. This case only comes up in case 1, because case 1 is the only case that deals with pairs where the words are of equal length.

Examples of case 2 can be found by calling allValidSuffixes and then reversing each of the suffixes found and looking them up.

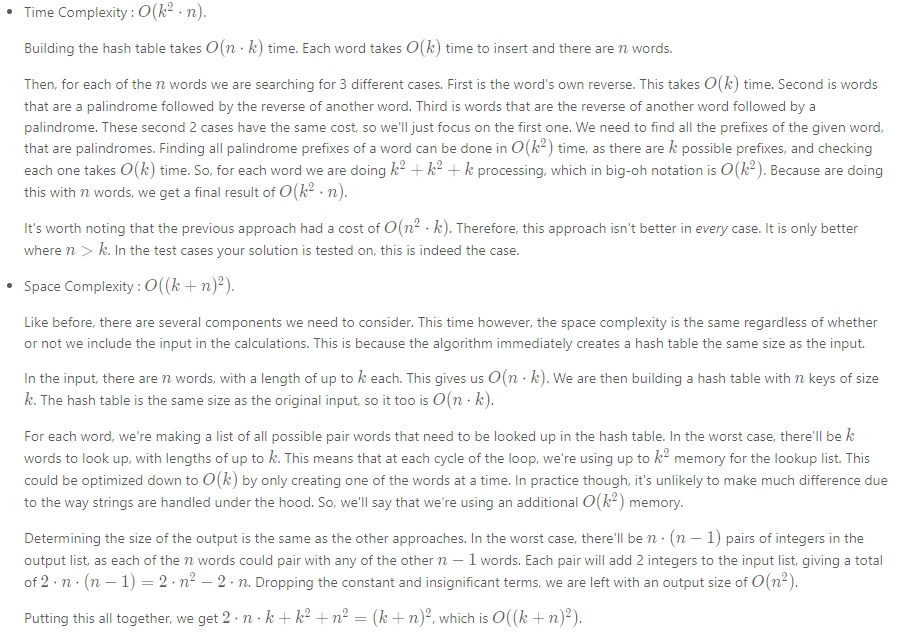
Examples of case 3 can be found by calling allValidPrefixes and then reversing each of the prefixes found and looking them up.

It would be possible to simplify further (not done here) by recognizing that **case 1** is really just a special case of **case 2** and **case 3**. This is because the empty string is a palindrome prefix/ suffix of any word.

|  |
| --- |
| class Solution {  private List<String> allValidPrefixes(String word) {  List<String> validPrefixes = new ArrayList<>();  for (int i = 0; i < word.length(); i++) {  if (isPalindromeBetween(word, i, word.length() - 1)) {  validPrefixes.add(word.substring(0, i));  }  }  return validPrefixes;  }  private List<String> allValidSuffixes(String word) {  List<String> validSuffixes = new ArrayList<>();  for (int i = 0; i < word.length(); i++) {  if (isPalindromeBetween(word, 0, i)) {  validSuffixes.add(word.substring(i + 1, word.length()));  }  }  return validSuffixes;  }  // Is the prefix ending at i a palindrome?  private boolean isPalindromeBetween(String word, int front, int back) {  while (front < back) {  if (word.charAt(front) != word.charAt(back)) return false;  front++;  back--;  }  return true;  }  public List<List<Integer>> palindromePairs(String[] words) {  // Build a word -> original index mapping for efficient lookup.  Map<String, Integer> wordSet = new HashMap<>();  for (int i = 0; i < words.length; i++) {  wordSet.put(words[i], i);  }  // Make a list to put all the palindrome pairs we find in.  List<List<Integer>> solution = new ArrayList<>();  for (String word : wordSet.keySet()) {  int currentWordIndex = wordSet.get(word);  String reversedWord = new StringBuilder(word).reverse().toString();  // Build solutions of case #1. This word will be word 1.  if (wordSet.containsKey(reversedWord)  && wordSet.get(reversedWord) != currentWordIndex) {  solution.add(Arrays.asList(currentWordIndex, wordSet.get(reversedWord)));  }  // Build solutions of case #2. This word will be word 2.  for (String suffix : allValidSuffixes(word)) {  String reversedSuffix = new StringBuilder(suffix).reverse().toString();  if (wordSet.containsKey(reversedSuffix)) {  solution.add(Arrays.asList(wordSet.get(reversedSuffix), currentWordIndex));  }  }  // Build solutions of case #3. This word will be word 1.  for (String prefix : allValidPrefixes(word)) {  String reversedPrefix = new StringBuilder(prefix).reverse().toString();  if (wordSet.containsKey(reversedPrefix)) {  solution.add(Arrays.asList(currentWordIndex, wordSet.get(reversedPrefix)));  }  }  }  return solution;  }  } |

**Complexity Analysis**

Let *n* be the number of words, and k*k* be the length of the longest word.



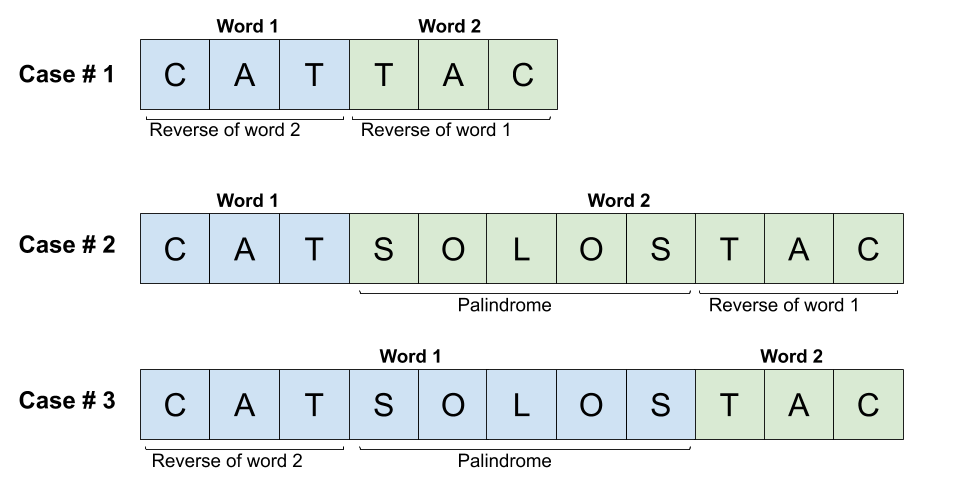
#### **Approach 3: Using a Trie**

**Intuition**

This section assumes you've previously been introduced to the Trie data structure. If you are not familiar with the Trie, work through [*Leetcode's module on them first*](https://leetcode.com/explore/learn/card/trie/) You'll also need to have read the previous section's intuition, as this section further builds on those ideas.

From the previous section, you probably noticed that the prefixes and suffixes of each word were important. If you're familiar with the Trie data structure, you may be wondering if there's a way we could use one to solve this problem. It turns out there is, so let's investigate!

We'll start by reminding ourselves of the palindrome pair cases we discovered in the previous section's intuition.



Now, we want to build some kind of Trie with the words. Then, we want to go down the list of words and identify all words from the Trie that our current word from the list would form a palindrome pair with. In words, we are looking for:

1. Words in the Trie that are the reverse of our current word.
2. Words in the Trie that start with the reverse of our current word and then finish in a palindrome.
3. Words in the Trie that are the reverse of the first part of our current word, and then what's left of our current word forms a palindrome.

Because we are interested in the reverse of words, it makes sense to put all the words into the Trie in reverse. You could also put the words forward into the Trie, and then reversed each word in the list. Both approaches are equally valid, and have their own pros and cons in terms of clarity.

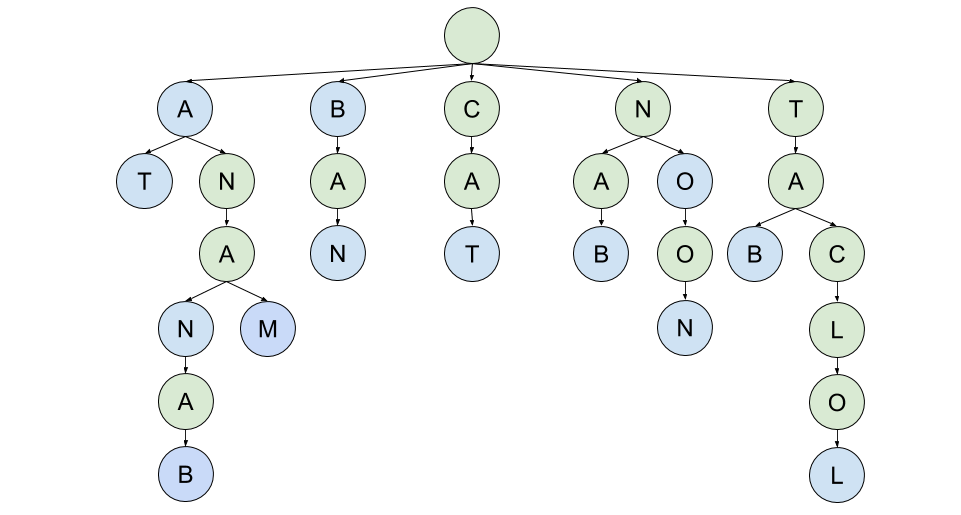
Anyway, let's jump to an example now. Our word list is as follows:

words = [ "A", "B", "BAN", "BANANA", "BAT", "LOLCAT", "MANA", "NAB", "NANA", "NOON", "ON", "TA", "TAC"]

We'll start by inserting the reverse of each word into a Trie, as shown in the following animation.

1 / 15

For ease of reference, here's the final Trie we got after inserting all the words.



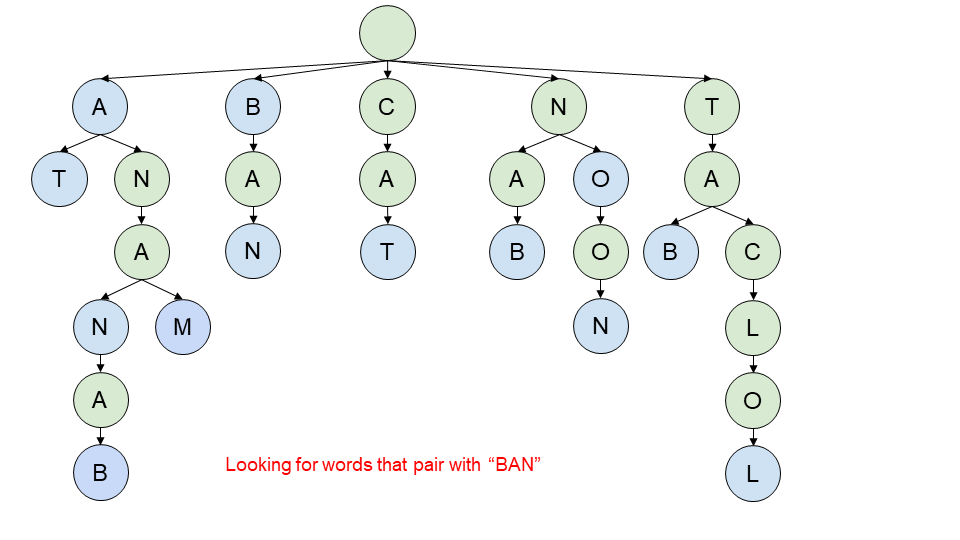
Great! We have a Trie. So, how do we use it? We'll look at each of the 3 cases, one-by-one.

Case 1 with the Trie

Case 1 is where a palindrome pair is formed by 2 words that are the reverse of each other. We'll use the word "BAN" as our example. The reverse of "BAN" is "NAB". Therefore, we need to use our Trie to see if the word "NAB" exists.

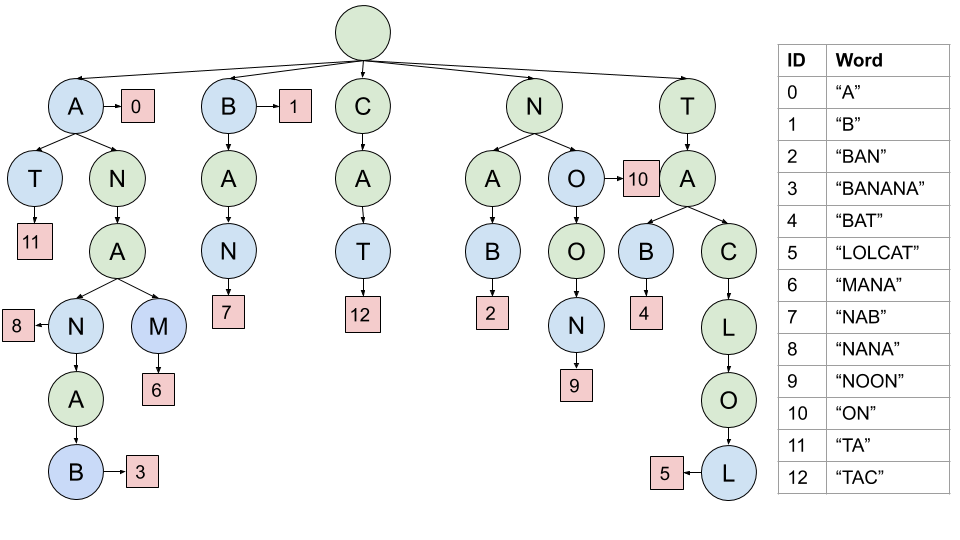
How will "NAB" appear in the Trie? Well, remember how all words were inserted into the Trie ***backwards***? This means that "NAB" will appear as "BAN" in the Trie. Therefore, we are simply searching for the word itself, in this case "BAN". If we can find the word in the Trie, **and** be on a blue (end of word) node when we're done, we know the reverse exists.

Here's an animation showing how we determine that the reverse of "BAN" is in the Trie.



1 / 8

Remember that for the output, we need to give the indexes of each pair. Currently, finding this information would be annoying. To fix it, we'll add an index field onto each end of word node. If we do that, this is our resulting Trie.



So, we knew that "BAN" had an index of 2. When we get to the end of the word it matches with, we see that it is word 7. The first word of the pair was "BAN", and "NAB" was the second. Therefore, we can add the pair [2, 7] to our output list.

Case 2 with the Trie

Case 2 is the one where the first word is shorter than the second word. The second word starts with a palindrome, and ends with the reverse of the first word. So, how will this look in our Trie?

Well, let's just have a look. The example we'll work with this time is "TAC". Like before, we know that the last 3 letters of the second word must start with "CAT". Now, remembering that these would have been inserted in reverse, we will start by looking for "TAC". Once we have found those letters, we would expect to not yet be at the end of a word, but for there to be a word that only has a palindrome left.

Here's an animation showing this search.

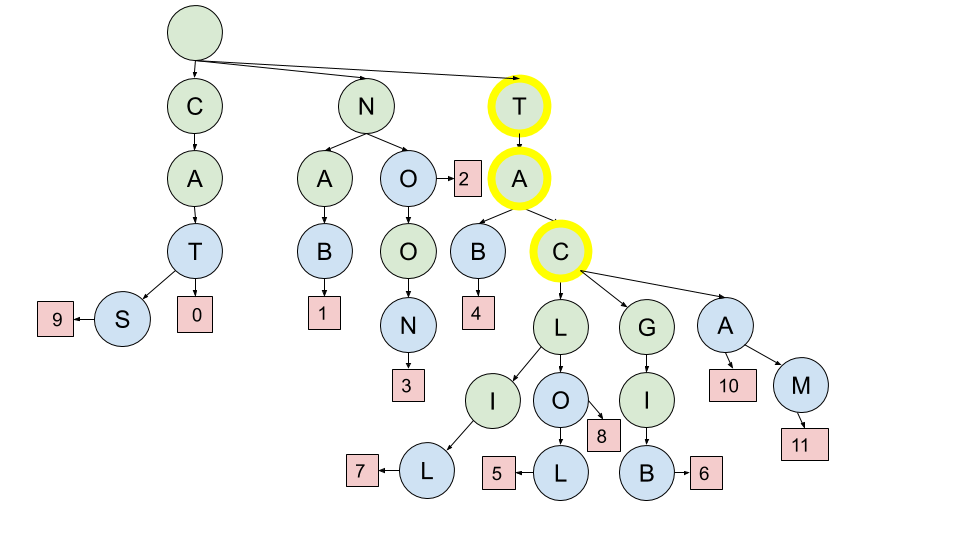
1 / 8

In this particular example, there was only one word left, and it did indeed form a palindrome.

A quick understanding check

Before we continue on with case 3, it's time to make sure you're following okay. Here's a different Trie, (closely) related to the one above. Like before, the word we're looking at is the word "TAC". We want to find all second words that it forms a palindrome pair with. Answer the following questions before scrolling down to the explanation.

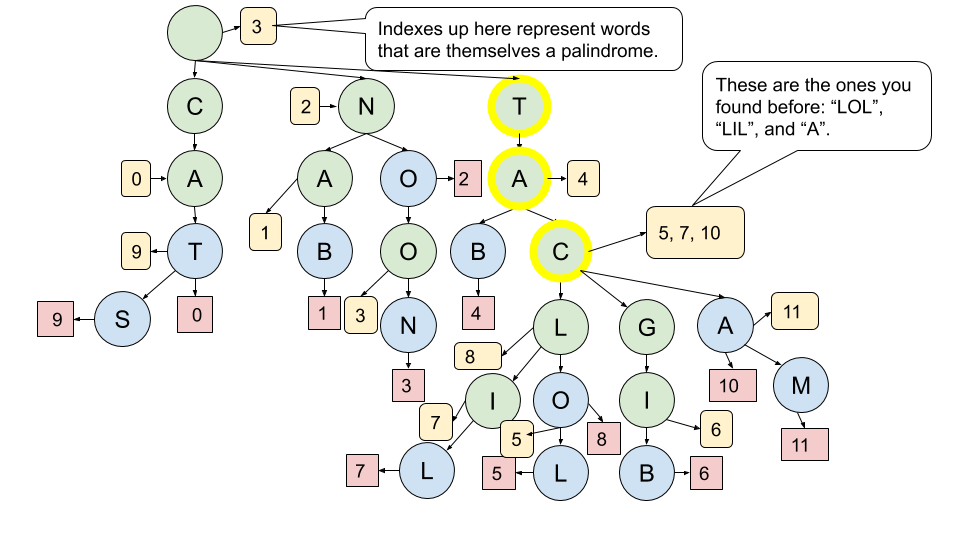
1. What are ALL the words that end in "CAT"?
2. What are all the words that form a palindrome pair with "TAC"?
3. How do we know the word "CAT" itself is not in the Trie?



1. We know that any blue circles "below" the letters we have found so far represent the end of a word. Remember that they should be read backwards! The answer is, therefore: "LILCAT". "OLCAT", "LOLCAT", "BIGCAT", "MACAT" and "ACAT".
2. To do this, you need to look for palindromes hanging below the highlighted "C". ALL of the words below the C are: "LIL", "OL", "LOL", "BIG", "MA", and "A". Of these, the palindromes are "LIL", "LOL", and "A", which correspond to the words "LOLCAT", "LILCAT", and "ACAT". Therefore, we know each of these 3 words will form a palindrome pair with "TAC".
3. If "CAT" were in the Trie, we'd expect to see the "C" at the end of the highlighted letters be blue and have an index field. It doesn't. Therefore, we know the word "CAT" is not in the Trie.

It might have been a little annoying having to carefully read each branch below the word "TAC", that ended in a blue circle. Luckily, there's an easy way we can improve the Trie structure to simplify this process.

When we insert a word, we can start by determining all of its palindrome prefixes. Now, on each node we'll attach a list of all words that have a palindrome remaining on them. For the example you worked through, this is the words you identified in part 2. The new Trie for that example would be as follows. The indexes are shown in yellow.



Have a think about our original example. What would these new lists be for it?

Case 3 with the Trie

Case 3 is the one where the first word is longer than the second word. In terms of our Trie, it would come up where we get to a blue node and still have some letters left from our current word. If those letters that are left form a palindrome, then we have a case 3 palindrome pair. Again, let's look at an example. This time, we are searching for the word "BANANA". Both times we reach a blue node, there is a palindrome remaining. Therefore, we find 2 pairs in this example.

1 / 9

This case is conceptually simpler than case 2. The key thing to remember is that we only do this palindrome check if we are on a blue node. If there is a palindrome remaining on our word, then a single pair is formed with the word that ended at that blue node (remember, blue nodes can only represent the end of a single word. There were no duplicates in the input list). Also, don't look at the "palindrome remaining" lists that we added for case 2, as this would lead to invalid pairs.

**Algorithm**

We start by building the Trie. For each word, reverse it and identify its palindrome prefixes (suffixes of the reversed word). Insert the word into the Trie, and mark the final letter as an ending node, and include the word's index. Also, while inserting, note any points where the remainder of the word is a palindrome suffix by including the index in an additional list (used for case 2).

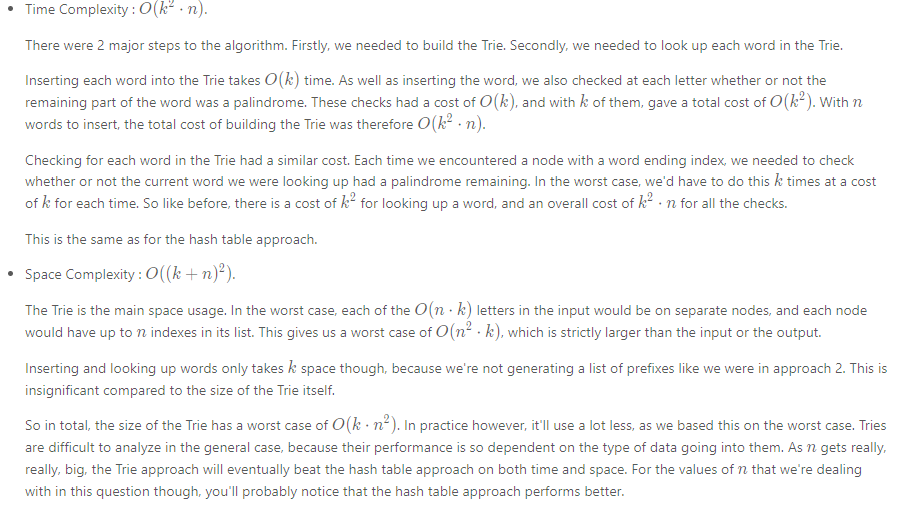
Then, we go back through the list of words, looking each up in the Trie. Any of the following situations give us palindrome pairs.

1. We have no letters left on the word and are at a word end node (case 1).
2. We have no letters left on the word and there are indexes in the list attached to the node (case 2).
3. We have a palindrome left on the word and are on a word end node (case 3).

|  |
| --- |
| class TrieNode {  public int wordEnding = -1; // We'll use -1 to mean there's no word ending here.  public Map<Character, TrieNode> next = new HashMap<>();  public List<Integer> palindromePrefixRemaining = new ArrayList<>();  }  class Solution {  // Is the given string a palindrome after index i?  // Tip: Leave this as a method stub in an interview unless you have time  // or the interviewer tells you to write it. The Trie itself should be  // the main focus of your time.  public boolean hasPalindromeRemaining(String s, int i) {  int p1 = i;  int p2 = s.length() - 1;  while (p1 < p2) {  if (s.charAt(p1) != s.charAt(p2)) return false;  p1++; p2--;  }  return true;  }  public List<List<Integer>> palindromePairs(String[] words) {  TrieNode trie = new TrieNode();  // Build the Trie  for (int wordId = 0; wordId < words.length; wordId++) {  String word = words[wordId];  String reversedWord = new StringBuilder(word).reverse().toString();  TrieNode currentTrieLevel = trie;  for (int j = 0; j < word.length(); j++) {  if (hasPalindromeRemaining(reversedWord, j)) {  currentTrieLevel.palindromePrefixRemaining.add(wordId);  }  Character c = reversedWord.charAt(j);  if (!currentTrieLevel.next.containsKey(c)) {  currentTrieLevel.next.put(c, new TrieNode());  }  currentTrieLevel = currentTrieLevel.next.get(c);  }  currentTrieLevel.wordEnding = wordId;  }  // Find pairs  List<List<Integer>> pairs = new ArrayList<>();  for (int wordId = 0; wordId < words.length; wordId++) {  String word = words[wordId];  TrieNode currentTrieLevel = trie;  for (int j = 0; j < word.length(); j++) {  // Check for pairs of case 3.  if (currentTrieLevel.wordEnding != -1  && hasPalindromeRemaining(word, j)) {  pairs.add(Arrays.asList(wordId, currentTrieLevel.wordEnding));  }  // Move down to the next trie level.  Character c = word.charAt(j);  currentTrieLevel = currentTrieLevel.next.get(c);  if (currentTrieLevel == null) break;  }  if (currentTrieLevel == null) continue;  // Check for pairs of case 1. Note the check to prevent non distinct pairs.  if (currentTrieLevel.wordEnding != -1 && currentTrieLevel.wordEnding != wordId) {  pairs.add(Arrays.asList(wordId, currentTrieLevel.wordEnding));  }  // Check for pairs of case 2.  for (int other : currentTrieLevel.palindromePrefixRemaining) {  pairs.add(Arrays.asList(wordId, other));  }  }  return pairs;  }  } |

**Complexity Analysis**

Let *n* be the number of words, and k*k* be the length of the longest word.



#### **Additional Discussion: Online Algorithms**

This section is beyond what is needed for an interview, and is included only for interest.

When developing algorithms for the real world, an often desirable property is that the algorithm works **online**. This does ***not mean on the internet***, instead it means that the algorithm can still work if the input data is provided bit-by-bit. In this case, it'd be that we want to feed the algorithm the words one at a time, and each time, we want to update the list of all pairs without doing too much extra work.

So, let's think through how this would work for approach 2. We'd simply be maintaining a hash table of words to indexes. Each time a new word arrives, we'd need to add it to the hash table and also check which existing words it'd form a palindrome pair with. It's a little bit different to before, because we need to find all pairs with previous words that

For case 1, this is straightforward. We simply check if its reverse is already in the hash table. If it is, then we have 2 new pairs (the new word can be either first or second).

But it breaks for case 2 and case 3. It's straightforward to find pairs where our new word is the longer word of the pair (i.e. second in case 2 and first in case 3), however not where the new word is shorter. The problem is that the additional letters of the longer word could be anything, and therefore we have no way of knowing what to look up in the index. Approach 2 worked as an offline algorithm because pairs were always identified by starting with their longer word, and then looking up their shorter word. Going the other way is intractable.

Approach 3, however, works differently. If we build up a Trie as we go, we can always identify words from the Trie that will form the second half of the pair. It doesn't matter whether it is the current word, or the word from the Trie, that is longer. This solves half the problem—each time we get a new word, we can efficiently find all "second" words for it.

We aren't done yet though—the algorithm wouldn't find pairs where our current word was second. We still need to find a way of identifying all "first" words for the current word. It turns out that if we hadn't reversed words when putting them into the Trie, but instead had reversed the word we are looking up, that we'd be looking up "first" words in the Trie.

Therefore, we can make an online algorithm by maintaining 2 Tries—one with the words forward, and one with the words in reverse. The reverse Trie tells us where the new word will be the first word of a pair, and the forward Trie tells us where the new word will be the second of a pair.