<https://leetcode.com/problems/palindrome-pairs/>

You are given a 0-indexed array of **unique** strings words.

A **palindrome pair** is a pair of integers (i, j) such that:

0 <= i, j < words.length, i != j, and words[i] + words[j] (the concatenation of the two strings) is a palindrome.

Return an array of all the **palindrome pairs** of words.

You must write an algorithm with O(sum of words[i].length) runtime complexity.

**Example 1:**

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

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

Explanation: The palindromes are ["abcddcba","dcbaabcd","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]]

Explanation: The palindromes are ["a","a"]

**Constraints:**

1 <= words.length <= 5000

0 <= words[i].length <= 300

words[i] consists of lowercase English letters.

**Attempt 1: 2023-06-28**

**Solution 1: Hash Table (60 min)**

class Solution {

    public List<List<Integer>> palindromePairs(String[] words) {

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

        // {key, val} -> {word, index}

        Map<String, Integer> map = new HashMap<String, Integer>();

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

            map.put(words[i], i);

        }

        // Special cases: "" can be combine with any palindrome string

        if(map.containsKey("")) {

            int emptyStrIndex = map.get("");

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

                if(i != emptyStrIndex && isPalindrome(words[i])) {

                    result.add(Arrays.asList(emptyStrIndex, i));

                    result.add(Arrays.asList(i, emptyStrIndex));

                }

            }

        }

        // Find all string and reverse string pairs

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

            String reverse\_str = reverse(words[i]);

            if(map.containsKey(reverse\_str)) {

                int found = map.get(reverse\_str);

                // If "found == i" means current word itself is a palindrome,

                // but without compensate word, we cannot use it as a pair

                if(found != i) {

                    // We don't have to add reversed index pair{found, i} again

                    // because when traversal for loop of index till 'found',

                    // we will automatically add pair{found, i}

                    // e.g

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

                    // Output: [0,1],[1,0]

                    // when i = 0, we get found = 1, we add pair {0, 1} first

                    // when i = 1, we get found = 0, we add pair {1, 0} then

                    // no need to add {1, 0} when only traverse till index = 0

                    result.add(Arrays.asList(i, found));

                }

            }

        }

        // Find the pair s1, s2 cut cases, 'cut' start from 1

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

            String word = words[i];

            for(int cut = 1; cut < word.length(); cut++) {

                // Case1 : s1[0:cut) is palindrome and s1[cut:] = reverse(s2) => (s2, s1)

                // e.g

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

                // Output: [3,2] -> s2("s") + s1("lls") = "slls"

                //        [2,4] -> s2("lls") + s1("sssll") = "llssssll"

                if(isPalindrome(word.substring(0, cut))) {

                    String reverse\_cut = reverse(word.substring(cut));

                    if(map.containsKey(reverse\_cut)) {

                        int found = map.get(reverse\_cut);

                        if(found != i) {

                            // s2 index is 'found' must before s1 index 'i'

                            result.add(Arrays.asList(found, i));

                        }

                    }

                }

                // Case2 : s1[cut:] is palindrome and s1[0:cut) = reverse(s2) => (s1, s2)

                // Input: words = ["esll","se","sssll"]

                // Output: [0,1] -> s1("esll") + s2("se") = "esllse"

                if(isPalindrome(word.substring(cut))) {

                    String reverse\_cut = reverse(word.substring(0, cut));

                    if(map.containsKey(reverse\_cut)) {

                        int found = map.get(reverse\_cut);

                        if(found != i) {

                            // s2 index is 'found' must after s1 index 'i'

                            result.add(Arrays.asList(i, found));

                        }

                    }

                }

            }

        }

        return result;

    }

    private String reverse(String s) {

        StringBuilder sb = new StringBuilder(s);

        return sb.reverse().toString();

    }

    private boolean isPalindrome(String s) {

        int i = 0;

        int j = s.length() - 1;

        while(i < j) {

            if(s.charAt(i) != s.charAt(j)) {

                return false;

            }

            i++;

            j--;

        }

        return true;

    }

}

Apparently there is an O(n^2 \* k) naive solution for this problem, with n the total number of words in the words array and k the average length of each word: for each word, we simply go through the words array and check whether the concatenated string is a palindrome or not.

Time Complexity:

Assume average length of word is k, n = number of words in the list, then the time complexity is: O(n \* k^2) because of this for loop

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

        String cur = words[i];

        for(int cut = 1; cut < cur.length(); cut++){

            if(isPalindrome(cur.substring(0, cut))){

                ...

            }

            if(isPalindrome(cur.substring(cut))){

                ...

            }

        }

    }

**Refer to**

<https://leetcode.com/problems/palindrome-pairs/solutions/79210/the-easy-to-unserstand-java-solution/>

There are several cases to be considered that isPalindrome(s1 + s2):

Case1: If s1 is a blank string, then for any string that is palindrome s2, s1+s2 and s2+s1 are palindrome.

Case 2: If s2 is the reversing string of s1, then s1+s2 and s2+s1 are palindrome.

Case 3: If s1[0:cut] is palindrome and there exists s2 is the reversing string of s1[cut+1:] , then s2+s1 is palindrome.

Case 4: Similiar to case3. If s1[cut+1: ] is palindrome and there exists s2 is the reversing string of s1[0:cut] , then s1+s2 is palindrome.

To make the search faster, build a HashMap to store the String-idx pairs.

My code:

public class Solution {

public List<List<Integer>> palindromePairs(String[] words) {

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

    if(words == null || words.length == 0){

        return res;

    }

    //build the map save the key-val pairs: String - idx

    HashMap<String, Integer> map = new HashMap<>();

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

        map.put(words[i], i);

    }

    //special cases: "" can be combine with any palindrome string

    if(map.containsKey("")){

        int blankIdx = map.get("");

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

            if(isPalindrome(words[i])){

                if(i == blankIdx) continue;

                res.add(Arrays.asList(blankIdx, i));

                res.add(Arrays.asList(i, blankIdx));

            }

        }

    }

    //find all string and reverse string pairs

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

        String cur\_r = reverseStr(words[i]);

        if(map.containsKey(cur\_r)){

            int found = map.get(cur\_r);

            if(found == i) continue;

            res.add(Arrays.asList(i, found));

        }

    }

    //find the pair s1, s2 that

    //case1 : s1[0:cut] is palindrome and s1[cut+1:] = reverse(s2) => (s2, s1)

    //case2 : s1[cut+1:] is palindrome and s1[0:cut] = reverse(s2) => (s1, s2)

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

        String cur = words[i];

        for(int cut = 1; cut < cur.length(); cut++){

            if(isPalindrome(cur.substring(0, cut))){

                String cut\_r = reverseStr(cur.substring(cut));

                if(map.containsKey(cut\_r)){

                    int found = map.get(cut\_r);

                    if(found == i) continue;

                    res.add(Arrays.asList(found, i));

                }

            }

            if(isPalindrome(cur.substring(cut))){

                String cut\_r = reverseStr(cur.substring(0, cut));

                if(map.containsKey(cut\_r)){

                    int found = map.get(cut\_r);

                    if(found == i) continue;

                    res.add(Arrays.asList(i, found));

                }

            }

        }

    }

    return res;

}

public String reverseStr(String str){

    StringBuilder sb= new StringBuilder(str);

    return sb.reverse().toString();

}

public boolean isPalindrome(String s){

    int i = 0;

    int j = s.length() - 1;

    while(i <= j){

        if(s.charAt(i) != s.charAt(j)){

            return false;

        }

        i++;

        j--;

    }

    return true;

}

**Solution 2: Trie (600 min)**

class Solution {

    public List<List<Integer>> palindromePairs(String[] words) {

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

        TrieNode root = new TrieNode();

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

            addWord(root, words[i], i);

        }

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

            search(words, i, root, result);

        }

        return result;

    }

    private void addWord(TrieNode root, String word, int index) {

        // Construct a word into Trie reversely

        for(int i = word.length() - 1; i >= 0; i--) {

            int j = word.charAt(i) - 'a';

            if(root.next[j] == null) {

                root.next[j] = new TrieNode();

            }

            // If current substring (or full length string) is a

            // palindrome, add current word's index into node.list

            if(isPalindrome(word, 0, i)) {

                root.list.add(index);

            }

            root = root.next[j];

        }

        // After finishing scan current word reversely record word's

        // index in words array into last node to declear finish

        // construction of current word at this node

        root.index = index;

        // Tricky point: always consider empty string "" as a natural

        // palindrome, hence, any string won't miss its full length

        // palindrome counterpart, e.g "abcd" -> "dcba"

        // Check Example 3

        root.list.add(index);

    }

    private void search(String[] words, int i, TrieNode root, List<List<Integer>> result) {

        for(int j = 0; j < words[i].length(); j++) {

            // root.index >= 0 means its a full constructed word till current node in Trie

            // root.index != i means the match condition word should not be searching word itself

            // isPalindrome(...) to find if remain substring start from j till last char is a palindrome or not, accompany with add word reversely when build Trie, self-consistent for palindrome pair checking

            if(root.index >= 0 && root.index != i && isPalindrome(words[i], j, words[i].length() - 1)) {

                result.add(Arrays.asList(i, root.index));

            }

            // Update to next level and try to find if that match conditions in next loop,

            // end with no more characters

            root = root.next[words[i].charAt(j) - 'a'];

            // Stop early if not find successor node in Trie branch

            if(root == null) {

                return;

            }

        }

        // Finished scan current word, then look at its last node to see if any

        // palindrome pair index stored in node.list as candidate

        for(int j : root.list) {

            // Should not be same index, since same index means one word itself

            // If missing this, test out in Example 3

            if(i == j) {

                continue;

            }

            result.add(Arrays.asList(i, j));

        }

    }

    private boolean isPalindrome(String word, int i, int j) {

        while(i < j) {

            if(word.charAt(i++) != word.charAt(j--)) {

                return false;

            }

        }

        return true;

    }

}

class TrieNode {

    // 'index' default as -1 used to identify if a

    // word finished construction in Trie or not

    int index;

    TrieNode[] next;

    // 'list' used to store word's index in words array

    // if prefix substring of this word which ending at

    // next node of current node is a palindrome after

    // building each word reversely into Trie

    List<Integer> list;

    public TrieNode() {

        index = -1;

        next = new TrieNode[26];

        list = new ArrayList<Integer>();

    }

}

Time Complexity:

O(n \* k^2) java solution with Trie structure (n: total number of words; k: average length of each word)

Note:

<https://leetcode.com/problems/palindrome-pairs/solutions/79195/o-n-k-2-java-solution-with-trie-structure/comments/84103>

Each word indeed can have up to O(n) words that can form a palindrome pair with it. If we check these words one by one, it will lead to the O(n^2 \* k) solution. But if we rearrange all the words in a prefix tree structure, it is possible to exclude a whole bunch of words by simply checking each letter in the word. In this way we can reduce the time complexity down to O(n \* k^2). Of course we have assumed that n >> k. If n ~ k, then the complexies of the two solutions are comparable to each other

**Refer to**

<https://leetcode.com/problems/palindrome-pairs/solutions/176205/beats-80-trie-java-with-explanations/>

**Thought**

We want to concatenate string B to string A to make AB palindrome.

How could AB be palindrome?

If B ends with x, then A must starts with x. If the second character of B is y, then the second last character of A is y...

That is,

  **Case 1. A must be prefix of reversed B, and the rest of reversed B should be palindrome.** For example,

(B:oooabc - cbaooo,    A:cba      AB:cba|oooabc)

  **Case 2. Or, reversed B must be prefix of A, and the rest of A should be palindrome.** For example,

(B:abc - cba          A:cbaooo,  AB:cbaooo|abc)

**Each word in words can be B. We put all reversed words in a trie.**

**Each word in words can be A. So we search A in trie.**

In this way,

  Case 1. if we found A in trie, and the branch under the end node is a palindrome, we found it!

  Case 2. if we reach a leaf of trie, and the leaf node represents the last character of current searching word, we found it!

For Case 2, actually its a corner cases of considering empty string as natural palindrome. Both ("", self-palindrome) and (self-palindrome, "") are still palindrome.

  For Case 1, we modify TrieNode data structure by adding an ArrayList - list of word indices such that nodes below can construct a palindrome, and use isPalindrome(str, 0, j) method to check a substring from first char to jth char is a potential palindrome or not, and jth char is next node after end node of A (e.g B:**oooabc** - **cbaooo**, A:**cba**, we found "**cba**" in reversely constructed Trie branch "**cbaooo**", then check palindrome by isPalindrome("**oooabc**", 0, 2), j is 2 means the last 'o' in "**oooabc**", and the last 'o' is the next node after end node of A as "**cba**" when talking about Trie branch "**cbaooo**")

  For Case 2, for each leaf node store current word's index into leaf node's list, and use another node.index property to identify the current word is fully constructed

**Refer to**

<https://leetcode.com/problems/palindrome-pairs/solutions/79195/o-n-k-2-java-solution-with-trie-structure/>

Apparently there is an O(n^2 \* k) naive solution for this problem, with n the total number of words in the words array and k the average length of each word: **for each word, we simply go through the words array and check whether the concatenated string is a palindrome or not.**

Of course this will result in TLE, as expected. To improve the algorithm, we want to reduce the number of words that need to be checked for each word, instead of iterating through the whole array. This prompted me to think if I can extract any useful information out of the process checking whether the concatenated string is a palindrome, so that it can help eliminate as many words as possible for the rest of the words array.

To begin, here is the technique I employed to check for palindromes: maintain two pointers i and j, with i pointing to the start of the string and j to the end of the string. Characters pointed by i and j are compared. If at any time the characters pointed by them are not the same, we conclude the string is not a palindrome. Otherwise we move the two pointers towards each other until they meet in the middle and the string is a palindrome.

By examining the process above, I did find something that we may take advantage of to get rid of words that need to be checked otherwise. For example, let's say we want to append words to w0, which starts with character 'a'. Then we only need to consider words ending with character 'a', i.e., this will single out all words ending with character 'a'. If the second character of w0 is 'b', for instance, we can further reduce our candidate set to words ending with string "ba", etc. Our naive solution throws away all these useful pieces of information and repeats the comparison, which leads to the undesired O(n^2 \* k) time complexity.

In order to exploit the information gathered so far, we obviously need to restructure all the words in the words array. If you are familiar with **Trie** structure (I believe you are, since LeetCode has problems for it. In case you are not, see [Trie](https://en.wikipedia.org/wiki/Trie)), it will come to mind as we need to deal with words with common suffixes. The next step is to design the structure for each Trie node. There are at least two fields that should be covered for each TrieNode: a TrieNode array denoting the next layer of nodes and a boolean (or integer) to signify the end of a word. So our tentative TrieNode will look like this:

class TrieNode {

    TrieNode[] next;

    boolean isWord;

}

One point here is that we assume all the words contain lowercase letters only. This is not specified in the problem statement so you probably need to confirm with the interviewer (here I assume it is the case).

Now we will rearrange each word into this Trie structure: for each word, simply starting from its last character and identify the node at the next layer by indexing into root's

next array with index given by the difference between the ending character and character 'a'. If the indexed node is null, create a new node. Continue to the next layer and towards the beginning of the word in this manner until we are done with the word, at which point we will label the isWord field of the final node as true.

After building up the Trie structure, we can proceed to search for pairs of palindromes for each word in the words array. I will use the following example to explain how it works and make possible modifications of the TrieNode we proposed above.

Let's say we have these words:

["ba", "a", "aaa"], the Trie structure will be as follows:

        root (f)

          | 'a'

          n1 (t)

    ------------

'b' |          | 'a'

    n2 (t)    n3 (f)

                | 'a'

              n4 (t)

The letter in parentheses indicates the value of isWord for each node: f ==> false and t ==> true. The letter beside each vertical line denotes the index into the

next array of the corresponding node. For example, for the first vertical line, 'a' means root.next[0] is not null. Similarly 'b' means n1.next[1] is not null, and so on.

Here is the searching process:

For word "ba", starting from the first character 'b', index into the [root.next](http://root.next) array with index given by 'b' - 'a' = 1. The corresponding node is null, then we know there are no words ending at this character, so the searching process is terminated;

For word "a", again indexing into array [root.next](http://root.next) at index given by 'a' - 'a' = 0 will yield node n1, which is not null. We then check the value of n1.isWord. If it is true, then it is possible to obtain a palindrome by appending this word to the one currently being examined (a.k.a word "a"). Also note that the two words should be distinct from each other, but the n1.isWord field provides no information about the word itself (means no information about what the original whole word is, like here input 3rd string 'aaa' has suffix 'a' exactly the same value as 2nd string 'a'), which makes it impossible to distinguish the two words. So it is necessary to modify the structure of the TrieNode so that we can identify the word it represents. One easy way is to have an integer field to remember the index of the word in the words array. For non-word nodes, this integer will take negative values (-1 for example) while for those representing a word, it will be non-negative values. Suppose we have made this modification, then the two words will be identified to be the same, so we discard this pair combination. Since the word "a" has only one letter, it seems we are done with it. Or do we? Not really. What if there are words with suffix "a" ("aaa" in this case)? We need to continue to check the rest part of these words (such as "aa" for the word "aaa") and see if the rest forms a palindrome. If it is, then appending this word ("aaa" in this case) to the original word ("a") will also form a palindrome ("aaaa"). Here I take another strategy: add an integer list to each TrieNode; the list will record the indices of all words satisfying the following two conditions: ***1. each word has a suffix represented by the current TrieNode; 2. the rest of the word forms a palindrome.***

Before I get to the third word "aaa", let me spell out the new TrieNode and the corresponding Trie structure for the above array.

**TrieNode:**

class TrieNode {

    TrieNode[] next;

    int index;

    List<Integer> list;

    TrieNode() {

        next = new TrieNode[26];

        index = -1;

        list = new ArrayList<>();

    }

}

**Trie:**

          root (-1,[1,2])

            | 'a'

          n1 (1,[0,1,2])

    ---------------------

'b' |                | 'a'

  n2 (0,[0])    n3 (-1,[2])

                      | 'a'

                n4 (2,[2])

**The first integer in the parentheses is the index of the word in the words" array (defaulted to -1). The integers in the square bracket are the indices of words satisfying the two conditions mentioned above.**

Let's continue with the third word "aaa" with this new structure. Indexing into array [root.next](http://root.next) at index given by 'a' - 'a' = 0 will yield node n1 and n1.index = 1 >= 0, which means we have a valid word now. The index of this word (which is 1) is also different from the index of the word currently being visited, a.k.a "aaa" (which is 2). So pair (2,1) is a possible concatenation to form a palindrome. But still we need to check the rest of "aaa" (excluding the substring represented by current node n1 which is "a" from the beginning of "aaa") to see if it is a palindrome. If so, (2,1) will be a valid combination. We continue in this fashion until we reach the end of "aaa". Lastly we will check n4.list to see if there are any words satisfying the two conditions specified in step 2 which are different from current word, and add the corresponding valid pairs.

**Both building and searching the Trie structure take O(n \* k^2), (n: total number of words; k: average length of each word) which sets the total time complexity of the solution.**

Here is the complete Java program:

private static class TrieNode {

    TrieNode[] next;

    int index;

    List<Integer> list;

    TrieNode() {

    next = new TrieNode[26];

    index = -1;

    list = new ArrayList<>();

    }

}

public List<List<Integer>> palindromePairs(String[] words) {

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

    TrieNode root = new TrieNode();

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

        addWord(root, words[i], i);

    }

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

        search(words, i, root, res);

    }

    return res;

}

private void addWord(TrieNode root, String word, int index) {

    for (int i = word.length() - 1; i >= 0; i--) {

        int j = word.charAt(i) - 'a';

        if (root.next[j] == null) {

            root.next[j] = new TrieNode();

        }

        if (isPalindrome(word, 0, i)) {

            root.list.add(index);

        }

        root = root.next[j];

    }

    root.list.add(index);

    root.index = index;

}

private void search(String[] words, int i, TrieNode root, List<List<Integer>> res) {

    for (int j = 0; j < words[i].length(); j++) {

    if (root.index >= 0 && root.index != i && isPalindrome(words[i], j, words[i].length() - 1)) {

        res.add(Arrays.asList(i, root.index));

    }

    root = root.next[words[i].charAt(j) - 'a'];

      if (root == null) return;

    }

    for (int j : root.list) {

    if (i == j) continue;

    res.add(Arrays.asList(i, j));

    }

}

private boolean isPalindrome(String word, int i, int j) {

    while (i < j) {

    if (word.charAt(i++) != word.charAt(j--)) return false;

    }

    return true;

}

We have the TrieNode structure at the top. In the palindromePairs function, we build up the Trie by adding each word, then search for valid pairs for each word and record the results in the res list. The last isPalindrome function checks if the substring [i, j] (both inclusive) of the given word is a palindrome.

**Previously Self Commented Version**

class Solution {

    public void addWord(TrieNode root, String word, int index) {

        // Important tips: The corner stone here is loop a word in backward

        // e.g given word = "dcba", adding sequence on TrieNode root is

        // 'a' -> 'b' -> 'c' -> 'd'

        for(int i = word.length() - 1; i >= 0; i--) {

            int j = word.charAt(i) - 'a';

            if(root.next[j] == null) {

                root.next[j] = new TrieNode();

            }

            if(isPalindrome(word, 0, i)) {

                root.list.add(index);

            }

            // Move on from current level to next level

            root = root.next[j];

        }

        // This line is for handling empty string case ?

        // As already reach the first character of string,

        // if recording its index means left substring is

        // "" empty string, and actually "" is palindrome

        // which satisfy two conditions set up for this list:

        // (1) each word has a suffix represented by the current Trie node;

        //    --> current full string, e.g "dcba" has suffix Trie node "dcba"

        // (2) the rest of the word forms a palindrome.

        //    --> rest of the word of "abcd" is "" forms a palindrome

        // E.g if comment out this line, will error out as

        // 51/134 passed, and fail at

        // Input:["abcd","dcba","lls","s","sssll"]

        // Output:[[2,4],[3,2]]

        // Expected:[[0,1],[1,0],[2,4],[3,2]]

        root.list.add(index);

        // Update the index when a word scan finished

        root.index = index;

    }

    public void search(String[] words, int i, TrieNode root, List<List<Integer>> result) {

        for(int j = 0; j < words[i].length(); j++) {

            // (1) root.index >= 0 --> finished construct of current word as an existing word of in given words, locate at index, not tag index as -1 anymore

            // (2) root.index != i --> not the word itself(The index of this word is also different from the index of the word currently being visited)

            // (3) isPalindrome(...)

            // --> E.g But still we need to check the rest of "aaa" (excluding the substring

            //        represented by current node n1 which is "a" from the beginning of "aaa")

            //        to see if it is a palindrome. If so, (2,1) will be a valid combination.

            //        We continue in this fashion until we reach the end of "aaa"

            //        We start with index 'j' to 'words[i].length() - 1', no need to reverse the

            //        index checking order (e.g 'words[i].length() - 1' to 'j'), because we re-construct

            //        the word such as "aaa" from top root node to leaves node. The in order checking

            //        sequence here exactly implement check palindrome from top root node to leaves

            //        node (e.g first check palindrome from 3rd 'a' to 1st 'a' in "aaa", then check

            //        palindrome from 2nd 'a' to 1st 'a' in "aaa", then check 1st 'a' in "aaa")

            if(root.index >= 0 && root.index != i && isPalindrome(words[i], j, words[i].length() - 1)) {

                // Based on word at position i of words array and find a matching palindrome word

                // which at position root.index of words array

                result.add(Arrays.asList(i, root.index));

            }

            // Update to next level and try to find if that match conditions in next loop,

            // end with no more characters

            root = root.next[words[i].charAt(j) - 'a'];

            if(root == null) {

                return;

            }

        }

        // Lastly we will check current word's last node's list to see if there are any words

        // satisfying the two conditions specified which are different from current word,

        // and add the corresponding valid pairs.

        // E.g If miss this part, will cause 16 / 134 test cases passed.

        // Input: ["abcd","dcba","lls","s","sssll"]

        // Output: []

        // Expected: [[0,1],[1,0],[2,4],[3,2]]

        /\*\*

        \* There is a special analysis for this, initially, don't understand why we

        \* already loop the full word, still need to check one more node 'root'? And

        \* how is this one more node 'root' come from ?

        \* Explaination:

        \*  words:    abcd   dcba  lls  s  sssll

            position: 0 1 2  3    4

        The digital number after n means in when we construct this Trie, which

        step we reach which character, e.g (n5 means at step 5 we add last

        character of "dcba" as 'a' onto Trie, n6 means at step 6 we add second last

        character of "dcba" as 'b' onto Trie after step 5)

        abcd -> end at n8

        dcba -> end at n4 -> when construct Trie, "abcd" added onto Trie as "dcba" first

        lls  -> end at n11

        s    -> end at n9

        sssll-> end at n16

                          root (-1, [3])

        / 'a'        | 'd'        | 'l'            \ 's'

        n5 (-1, [])    n1 (-1, [])    n12 (-1, [])    n9 (3, [2,3])

        | 'b'              | 'c'        | 'l'            | 'l'

        n6 (-1, [])        n2 (-1, [])    n13 (-1, [4])  n10 (-1, [2])

        | 'c'      | 'b'        | 's'            | 'l'

        n7 (-1, [1])    n3 (-1, [0])    n14 (-1, [4])  n11 (2, [2])

        | 'd'      | 'a'        | 's'

        n8 (1, [1])    n4 (0, [0])    n15 (-1, [4])

  | 's'

        n16 (4, [4])

        \* In this example, we first will search for "abcd" which locate at

        \* position 0 of words array, the special thing is we not directly

        \* reach the first character 'a' of "abcd", we have an additional

        \* "root" at the beginning before "abcd", so, we will loop 4 times

        \* to change the "root" (by root = root.next[words[i].charAt(j) - 'a'];)

        \* to reach the last character of current word (e.g when j increase

        \* from 0 to 3, "root" varies as "root" -> "a" -> "b" -> "c" -> "d"),

        \* and stay on the last character constructed node (e.g n8).

        \* Since we already pass the for loop because of 'j' already = 3, and

        \* increase to 4 which over length of "abcd" to terminate the for loop,

        \* but we only check lists from "root" -> "a" -> "b" -> "c", not yet

        \* checking list of node constructed by "d"(e.g here as n8), now we check

        \* its list as [1], which point out the word at position of 1 of words

        \* array as "dcba" satisfy a counter part of current word "abcd" to

        \* make up a palindrome, finally add this pair (i = 0, j = 1) to result.

        \*

        \* Remind:

        \* How is n8 (1, [1]) generated ?

        \* Two conditions:

        \* (1) Word "dcba" has a suffix represented by the current Trie node as "dcba"

        \* (2) Current word "abcd", left part empty string "" forms a palindrome

        \*/

        for (int j : root.list) {

            // If miss this check, can be test with

            // Input:["abcd","dcba","lls","s","sssll"]

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

            // Expected:[[0,1],[1,0],[3,2],[2,4]]

            if(i == j) {

                continue;

            }

            result.add(Arrays.asList(i, j));

        }

    }

    private boolean isPalindrome(String word, int i, int j) {

        while (i < j) {

            if (word.charAt(i++) != word.charAt(j--)) {

                return false;

            }

        }

        return true;

    }

    public List<List<Integer>> palindromePairs(String[] words) {

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

        TrieNode root = new TrieNode();

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

            addWord(root, words[i], i);

        }

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

            search(words, i, root, result);

        }

        return result;

    }

}

class TrieNode {

    TrieNode[] next;

    int index;

    // Since the word "a" has only one letter, it seems we are done with it. Or do we? Not really. What if there are words

    // with suffix "a" ("aaa" in this case)? We need to continue to check the rest part of these words (such as "aa" for the

    // word "aaa") and see if the rest forms a palindrome. If it is, then appending this word ("aaa" in this case) to the

    // original word ("a") will also form a palindrome ("aaaa").

    // Here I take another strategy: add an integer list to each TrieNode;

    // the list will record the indices of all words satisfying the following two conditions:

    // (1) each word has a suffix represented by the current Trie node;

    // (2) the rest of the word forms a palindrome.

    List<Integer> list;

    public TrieNode() {

        next = new TrieNode[26];

        index = -1;

        list = new ArrayList<Integer>();

    }

}

**Example 1:**

1. words = {"cbaaa", "bc", "abc"}

When it tries to match cbaaa, when it arrives cb, its pos is 1, then we should check if aaa is a palindrome. When it arrives cba, we should check if rest of string aa is a palindrome. In this way, we can know that [“cbaaa”, “bc”], [“cbaaa”, “abc”] are pairs.

################################################################################

Explain on addWord logic:

    public void addWord(TrieNode root, String word, int index) {

        for(int i = word.length() - 1; i >= 0; i--) {

            int j = word.charAt(i) - 'a';

            if(root.next[j] == null) {

                root.next[j] = new TrieNode();

            }

            if(isPalindrome(word, 0, i)) {

                root.list.add(index);

            }

            root = root.next[j];

        }

        root.list.add(index);

        root.index = index;

    }

        Round 1:                            Round 2:                              Round 3:

        Trie add "cbaaa"                    Trie add "bc"                        Trie add "abc"

              root (index=-1,list=[])              root (index=-1,list=[])              root (index=-1,list=[])

        /                                    /            \                        /            \

      / 'a'                                / 'a'          \ 'c'                  / 'a'          \ 'c'

      n1 (-1,[])                          n1 (-1,[])      n6 (-1,[1])          n1 (-1,[])      n6 (-1,[1])

      |                                    |              |                    |              |

      | 'a'                                | 'a'          | 'b'                | 'a'          | 'b'

      n2 (-1,[])                ==>      n2 (-1,[])      n7 (1,[1])  ==>      n2 (-1,[])      n7 (1,[1,2])

      |                                    |                                    |              |

      | 'a'                                | 'a'                                | 'a'          | 'a'

      n3 (-1,[])                          n3 (-1,[])                            n3 (-1,[])      n8 (2,[2])

      |                                    |                                    |

      | 'b'                                | 'b'                                | 'b'

      n4 (-1,[0])                          n4 (-1,[0])                          n4 (-1,[0])

      |                                    |                                    |

      | 'c'                                | 'c'                                | 'c'

      n5 (0,[0])                          n5 (0,[0])                            n5 (0,[0])

Each node index and list statistics:

node.index -> if a word finished, update its index in words array on last character's corresponding node, default as -1 means not a word in words array

node.list -> isPalindrome(word, 0, i)

Round 1:

Trie add word = "cbaaa"

--------------------------------------------------------------------------------

root (index=-1,list=[])

create n1

root.index -> -1

root.list -> isPalindrome("cbaaa", 0, 4) -> "cbaaa" not a palindrome -> list = []

move from root to n1

--------------------------------------------------------------------------------

n1 (-1,[])

create n2

n1.index -> -1

n1.list -> isPalindrome("cbaaa", 0, 3) -> "cbaa" not a palindrome -> list = []

move from n1 to n2

--------------------------------------------------------------------------------

n2 (-1,[])

create n3

n2.index -> -1

n2.list -> isPalindrome("cbaaa", 0, 2) -> "cba" not a palindrome -> list = []

move from n2 to n3

--------------------------------------------------------------------------------

n3 (-1,[])

create n4

n3.index -> -1

n3.list -> isPalindrome("cbaaa", 0, 1) -> "cb" not a palindrome -> list = []

move from n3 to n4

--------------------------------------------------------------------------------

n4 (-1,[0])

create n5

n4.index -> -1

n4.list -> isPalindrome("cbaaa", 0, 0) -> "c" is a palindrome -> list = [0]

move from n4 to n5

--------------------------------------------------------------------------------

n5 (0,[0])

n5.index -> 0 (word "cbaaa" reverse sequence as "aaabc" finished at n5, last character('c')'s next level has no further character, additionally we use "root.index" to record current word "cbaaa" index in words array as 0 for empty string)

n5.list -> use "root.list.add(index)" to record current word "cbaaa" index in words array as 0 for empty string

================================================================================

Round 2:

Trie add word = "bc"

--------------------------------------------------------------------------------

root (index=-1,list=[])

create n6

root.index -> -1

root.list -> isPalindrome("bc", 0, 1) -> "bc" not a palindrome -> list = []

move from root to n6

--------------------------------------------------------------------------------

n6 (-1,[1])

create n7

n6.index -> -1

n6.list -> isPalindrome("bc", 0, 0) -> "b" is a palindrome -> list = [1]

move from n6 to n7

--------------------------------------------------------------------------------

n7 (1,[1])

n7.index -> 1 (word "bc" reverse sequence as "cb" finished at n7, last character('b')'s next level has no further character, additionally we use "root.index" to record current word "bc" index in words array as 1 for empty string)

n7.list -> use "root.list.add(index)" to record current word "bc" index in words array as 1 for empty string

================================================================================

Round 3:

Trie add word = "abc"

--------------------------------------------------------------------------------

root (index=-1,list=[])

no create node since already have n6

root.index -> -1

root.list -> isPalindrome("abc", 0, 2) -> "abc" not a palindrome -> list = []

move from root to n6

--------------------------------------------------------------------------------

n6 (-1,[1])

no create node since already have n7

n6.index -> -1

n6.list -> isPalindrome("abc", 0, 1) -> "ab" not a palindrome -> list keep as [1]

move from n6 to n7

--------------------------------------------------------------------------------

n7 (1,[1,2])

create n8

n7.index -> 1 (keep as 1 since "bc" finished at n7 in Round 2, n7 hold its finished status tag with "bc" index in words array as 1)

n7.list -> isPalindrome("abc", 0, 0) -> "a" is a palindrome -> list update from [1] to [1,2]

move from n7 to n8

--------------------------------------------------------------------------------

n8 (2,[2])

n8.index -> 2 (word "abc" reverse sequence as "cba" finished at n8, last character('a')'s next level has no further character, additionally we use "root.index" to record current word "abc" index in words array as 2 for empty string)

n8.list -> use "root.list.add(index)" to record current word "abc" index in words array as 2 for empty string

================================================================================

################################################################################

################################################################################

Explain on search logic:

    public void search(String[] words, int i, TrieNode root, List<List<Integer>> result) {

        for(int j = 0; j < words[i].length(); j++) {

            if(root.index >= 0 && root.index != i && isPalindrome(words[i], j, words[i].length() - 1)) {

                result.add(Arrays.asList(i, root.index));

            }

            root = root.next[words[i].charAt(j) - 'a'];

            if(root == null) {

                return;

            }

        }

        for (int j : root.list) {

            if(i == j) {

                continue;

            }

            result.add(Arrays.asList(i, j));

        }

    }

Search based on full constructed Trie tree after Round 3

      After Round 3:

              root (index=-1,list=[])

        /            \

      /'a'          \ 'c'

      n1 (-1,[])      n6 (-1,[1])

      |              |

      | 'a'          | 'b'

      n2 (-1,[])      n7 (1,[1,2])

      |              |

      | 'a'          | 'a'

      n3 (-1,[])      n8 (2,[2])

      |

      | 'b'

      n4 (-1,[0])

      |

      | 'c'

      n5 (0,[0])

Round 1:

search word = "cbaaa", i = 0

--------------------------------------------------------------------------------

root (index=-1,list=[]), j = 0

check conditions -> if(root.index >= 0 && root.index != i && isPalindrome(words[i], j, words[i].length() - 1)) {result.add(Arrays.asList(i, root.index));}

root.index = -1 < 0 -> check failure: current node not the last character of an existing word which means not an option for further check

root = root.next[words[i].charAt(j) - 'a'] -> move from root to n6

--------------------------------------------------------------------------------

n6 (-1,[1]), j = 1

check conditions -> if(root.index >= 0 && root.index != i && isPalindrome(words[i], j, words[i].length() - 1)) {result.add(Arrays.asList(i, root.index));}

n6.index = -1 < 0 -> check failure: current node not the last character of an existing word which means not an option for further check

n6 = n6.next[words[i].charAt(j) - 'a'] -> move from n6 to n7

--------------------------------------------------------------------------------

n7 (1,[1,2]), j = 2

check conditions -> if(root.index >= 0 && root.index != i && isPalindrome(words[i], j, words[i].length() - 1)) {result.add(Arrays.asList(i, root.index));}

n7.index = 1 >= 0 -> n7.index = 1 means word("bc") finished construction at n7

n7.index = 1 != i(0) -> n7.index != i means founded word("bc") not the current searching word("cbaaa")

isPalindrome(words[i], j, words[i].length() - 1) = isPalindrome("cbaaa", 2, 4) -> "aaa" is a palindrome

-> check success: result.add(Arrays.asList(i, root.index)) -> {{0, 1}} means {"cbaaa", "bc"} is a pair

n7 = n7.next[words[i].charAt(j) - 'a'] -> move from n7 to n8

--------------------------------------------------------------------------------

n8 (2,[2]), j = 3

check conditions -> if(root.index >= 0 && root.index != i && isPalindrome(words[i], j, words[i].length() - 1)) {result.add(Arrays.asList(i, root.index));}

n8.index = 2 >= 0 -> n8.index = 2 means word("abc") finished construction at n8

n8.index = 2 != i(0) -> n8.index != i means founded word("abc") not the current searching word("cbaaa")

isPalindrome(words[i], j, words[i].length() - 1) = isPalindrome("cbaaa", 3, 4) -> "aa" is a palindrome

-> check success: result.add(Arrays.asList(i, root.index)) -> {{0, 2}} means {"cbaaa", "abc"} is a pair

no more node to check

--------------------------------------------------------------------------------

result = {{0, 1}, {0, 2}}

================================================================================

Round 2:

search word = "bc", i = 1

--------------------------------------------------------------------------------

root (index=-1,list=[]), j = 0

check conditions -> if(root.index >= 0 && root.index != i && isPalindrome(words[i], j, words[i].length() - 1)) {result.add(Arrays.asList(i, root.index));}

root.index = -1 < 0 -> check failure: current node not the last character of an existing word which means not an option for further check

root = root.next[words[i].charAt(j) - 'a'] -> null -> no more node to check

================================================================================

Round 3:

Search word = "abc", i = 2

--------------------------------------------------------------------------------

root (index=-1,list=[]), j = 0

check conditions -> if(root.index >= 0 && root.index != i && isPalindrome(words[i], j, words[i].length() - 1)) {result.add(Arrays.asList(i, root.index));}

root.index = -1 < 0 -> check failure: current node not the last character of an existing word which means not an option for further check

root = root.next[words[i].charAt(j) - 'a'] -> move from root to n1

--------------------------------------------------------------------------------

n1 (-1,[]), j = 1

check conditions -> if(root.index >= 0 && root.index != i && isPalindrome(words[i], j, words[i].length() - 1)) {result.add(Arrays.asList(i, root.index));}

root.index = -1 < 0 -> check failure: current node not the last character of an existing word which means not an option for further check

root = root.next[words[i].charAt(j) - 'a'] -> null -> no more node to check

================================================================================

Done, result keep as {{0, 1}, {0, 2}}

################################################################################

**Example 2:**

2. words = {"abc", "xyxcba", "xcba"}

When we check abc and it arrives c in trie. We should know that the rest of string after abc has palindrome. So we modify trie like below:

c has a list palins. It tells string below c, what rest of word is palindrome. When we scan abc, it will finally stops at c in xyxabc. When it ends, check if palins list of trie c is not empty. If not, it may be able to concatenate into palindrome.

################################################################################

Explain on addWord logic:

    public void addWord(TrieNode root, String word, int index) {

        for(int i = word.length() - 1; i >= 0; i--) {

            int j = word.charAt(i) - 'a';

            if(root.next[j] == null) {

                root.next[j] = new TrieNode();

            }

            if(isPalindrome(word, 0, i)) {

                root.list.add(index);

            }

            root = root.next[j];

        }

        root.list.add(index);

        root.index = index;

    }

        Round 1:                          Round 2:                                  Round 3:

        Trie add "abc"                    Trie add "xyxcba"                        Trie add "xcba"

              root (index=-1,list=[])              root (index=-1,list=[])                root (index=-1,list=[])

                  \                          /          \                          /          \

                    \ 'c'                    / 'a'        \ 'c'                    / 'a'        \ 'c'

                    n1 (-1,[])              n4 (-1,[])    n1 (-1,[])              n4 (-1,[])    n1 (-1,[])

                    |                        |            |                        |              |

                    | 'b'                    | 'b'        | 'b'                    | 'b'          | 'b'

                    n2 (-1,[0])              n5 (-1,[])    n2 (-1,[0])              n5 (-1,[])    n2 (-1,[0])

                    |              ==>      |            |              ==>      |              |

                    | 'a'                    | 'c'        | 'a'                    | 'c'          | 'a'

                    n3 (0,[0])              n6 (-1,[1])  n3 (0,[0])              n6 (-1,[1,2])  n3 (0,[0])

                                            |                                      |

                                            | 'x'                                  | 'x'

                                            n7 (-1,[])                            n7 (2,[2])

                                            |                                      |

                                            | 'y'                                  | 'y'

                                            n8 (-1,[1])                            n8 (-1,[1])

                                            |                                      |

                                            | 'x'                                  | 'x'

                                            n9 (1,[1])                            n9 (1,[1])

Each node index and list statistics:

node.index -> if a word finished, update its index in words array on last character's corresponding node, default as -1 means not a word in words array

node.list -> isPalindrome(word, 0, i)

Round 1:

Trie add word = "abc"

--------------------------------------------------------------------------------

root (index=-1,list=[])

create n1

root.index -> -1

root.list -> isPalindrome("abc", 0, 2) -> "abc" not a palindrome -> list = []

move from root to n1

--------------------------------------------------------------------------------

n1 (-1,[])

create n2

n1.index -> -1

n1.list -> isPalindrome("abc", 0, 1) -> "ab" not a palindrome -> list = []

move from n1 to n2

--------------------------------------------------------------------------------

n2 (-1,[0])

create n3

n2.index -> -1

n2.list -> isPalindrome("abc", 0, 0) -> "a" is a palindrome -> list = [0]

move from n2 to n3

--------------------------------------------------------------------------------

n3 (0,[0])

n3.index -> 0 (word "abc" reverse sequence as "cba" finished at n3, last character('a')'s next level has no further character, additionally we use "root.index" to record current word "abc" index in words array as 0 for empty string)

n3.list -> use "root.list.add(index)" to record current word "abc" index in words array as 0 for empty string

================================================================================

Round 2:

Trie add word = "xyxcba"

--------------------------------------------------------------------------------

root (index=-1,list=[])

create n4

root.index -> -1

root.list -> isPalindrome("xyxcba", 0, 5) -> "xyxcba" not a palindrome -> list = []

move from root to n4

--------------------------------------------------------------------------------

n4 (-1,[])

create n5

n4.index -> -1

n4.list -> isPalindrome("xyxcba", 0, 4) -> "xyxcb" not a palindrome -> list = []

move from n4 to n5

--------------------------------------------------------------------------------

n5 (-1,[])

create n6

n5.index -> -1

n5.list -> isPalindrome("xyxcba", 0, 3) -> "xyxc" not a palindrome -> list = []

move from n5 to n6

--------------------------------------------------------------------------------

n6 (-1,[1])

create n7

n6.index -> -1

n6.list -> isPalindrome("xyxcba", 0, 2) -> "xyx" is a palindrome -> list = [1]

move from n6 to n7

--------------------------------------------------------------------------------

n7 (-1,[])

create n8

n7.index -> -1

n7.list -> isPalindrome("xyxcba", 0, 1) -> "xy" not a palindrome -> list = []

move from n7 to n8

--------------------------------------------------------------------------------

n8 (-1,[1])

create n9

n8.index -> -1

n8.list -> isPalindrome("xyxcba", 0, 0) -> "x" is a palindrome -> list = [1]

move from n8 to n9

--------------------------------------------------------------------------------

n9 (1,[1])

n9.index -> 1 (word "xyxcba" reverse sequence as "abcxyx" finished at n9, last character('x')'s next level has no further character, additionally we use "root.index" to record current word "xyxcba" index in words array as 1 for empty string)

n9.list -> use "root.list.add(index)" to record current word "xyxcba" index in words array as 1 for empty string

================================================================================

Round 3:

Trie add word = "xcba"

--------------------------------------------------------------------------------

root (index=-1,list=[])

no create node since already have n4

root.index -> -1

root.list -> isPalindrome("xcba", 0, 3) -> "xcba" not a palindrome -> list = []

move from root to n4

--------------------------------------------------------------------------------

n4 (-1,[])

no create node since already have n5

n4.index -> -1

n4.list -> isPalindrome("xcba", 0, 2) -> "xcb" not a palindrome -> list = []

move from n4 to n5

--------------------------------------------------------------------------------

n5 (-1,[])

no create node since already have n6

n5.index -> -1

n5.list -> isPalindrome("xcba", 0, 1) -> "xc" not a palindrome -> list = []

move from n5 to n6

--------------------------------------------------------------------------------

n6 (-1,[1]) update to (-1,[1,2])

no create node since already have n7

n6.index -> -1

n6.list -> isPalindrome("xcba", 0, 0) -> "x" is a palindrome -> list update from [1] to [1,2]

move from n6 to n7

--------------------------------------------------------------------------------

n7 (-1,[]) update to (2,[2])

n7.index -> 2 (word "xcba" reverse sequence as "abcx" finished at n7, last character('x')'s next level has no further character, additionally we use "root.index" to record current word "xcba" index in words array as 2 for empty string)

n7.list -> use "root.list.add(index)" to record current word "xcba" index in words array as 2 for empty string

================================================================================

################################################################################

################################################################################

Explain on search logic:

    public void search(String[] words, int i, TrieNode root, List<List<Integer>> result) {

        for(int j = 0; j < words[i].length(); j++) {

            if(root.index >= 0 && root.index != i && isPalindrome(words[i], j, words[i].length() - 1)) {

                result.add(Arrays.asList(i, root.index));

            }

            root = root.next[words[i].charAt(j) - 'a'];

            if(root == null) {

                return;

            }

        }

        for (int j : root.list) {

            if(i == j) {

                continue;

            }

            result.add(Arrays.asList(i, j));

        }

    }

Search based on full constructed Trie tree after Round 3

        After Round 3:

                root (index=-1,list=[])

          /          \

        / 'a'        \ 'c'

        n4 (-1,[])    n1 (-1,[])

        |              |

        | 'b'          | 'b'

        n5 (-1,[])    n2 (-1,[0])

        |              |

        | 'c'          | 'a'

        n6 (-1,[1,2])  n3 (0,[0])

        |

        | 'x'

        n7 (2,[2])

        |

        | 'y'

        n8 (-1,[1])

        |

        | 'x'

        n9 (1,[1])

Round 1:

search word = "abc", i = 0

--------------------------------------------------------------------------------

root (index=-1,list=[]), j = 0

check conditions -> if(root.index >= 0 && root.index != i && isPalindrome(words[i], j, words[i].length() - 1)) {result.add(Arrays.asList(i, root.index));}

root.index = -1 < 0 -> check failure: current node not the last character of an existing word which means not an option for further check

root = root.next[words[i].charAt(j) - 'a'] -> move from root to n4

--------------------------------------------------------------------------------

n4 (-1,[1]), j = 1

check conditions -> if(root.index >= 0 && root.index != i && isPalindrome(words[i], j, words[i].length() - 1)) {result.add(Arrays.asList(i, root.index));}

n4.index = -1 < 0 -> check failure: current node not the last character of an existing word which means not an option for further check

n4 = n4.next[words[i].charAt(j) - 'a'] -> move from n4 to n5

--------------------------------------------------------------------------------

n5 (-1,[1]), j = 2

check conditions -> if(root.index >= 0 && root.index != i && isPalindrome(words[i], j, words[i].length() - 1)) {result.add(Arrays.asList(i, root.index));}

n5.index = -1 < 0 -> check failure: current node not the last character of an existing word which means not an option for further check

n5 = n5.next[words[i].charAt(j) - 'a'] -> move from n5 to n6

--------------------------------------------------------------------------------

n6 (-1,[1,2]), j = 3 break out loop

check conditions -> if(root.index >= 0 && root.index != i && isPalindrome(words[i], j, words[i].length() - 1)) {result.add(Arrays.asList(i, root.index));}

n6.index = -1 < 0 -> check failure: current node not the last character of an existing word which means not an option for further check

Spaeical case here:

Rewind the meaning of n6 (-1,[1,2]) => n6.index = -1 means no word during trie construstion finished at n6, [1,2] means even no word construction finished at n6, but during trie construction we also record a word's index in the words array into node.list property if remain part of word is a palindrome, in another word, the prefix substring of the original word ending at next node of n6 as traversal from bottom node in trie is a palindrome, e.g the prefix substring as "xyx" of "xyxcba" is a palindrome as traversal from bottom node n9 -> n8 -> n7, ending at n7 as next node of n6, then we record word "xyxcba" index in words array as 1 into n6.list property, another prefix subtring as "x" of "xcba" is also a palindrome as traversal from intermediate node n7, also ending at n7 itself as next node of n6, then we record word "xcba" index in words array as 2 into n6.list property)

First for(int j = 0; j < words[i].length(); j++) loop ending when search word = "abc" as j looped over length when j > 2

Second for(int j : root.list) {if(i == j) {continue;} result.add(Arrays.asList(i, j));} loop

result = {{0, 1}, {0, 2}}

================================================================================

Round 2:

search word = "xyxcba", i = 1

--------------------------------------------------------------------------------

root (index=-1,list=[]), j = 0

check conditions -> if(root.index >= 0 && root.index != i && isPalindrome(words[i], j, words[i].length() - 1)) {result.add(Arrays.asList(i, root.index));}

root.index = -1 < 0 -> check failure: current node not the last character of an existing word which means not an option for further check

root = root.next[words[i].charAt(j) - 'a'] -> null -> no more node to check

================================================================================

Round 3:

Search word = "xcba", i = 2

--------------------------------------------------------------------------------

root (index=-1,list=[]), j = 0

check conditions -> if(root.index >= 0 && root.index != i && isPalindrome(words[i], j, words[i].length() - 1)) {result.add(Arrays.asList(i, root.index));}

root.index = -1 < 0 -> check failure: current node not the last character of an existing word which means not an option for further check

root = root.next[words[i].charAt(j) - 'a'] -> null -> no more node to check

================================================================================

Done, result keep as {{0, 1}, {0, 2}}

################################################################################

**Example 3:**

3. words = {"abcd", "dcba", "lls", "s", "sssll"}

################################################################################

Explain on addWord logic:

    public void addWord(TrieNode root, String word, int index) {

        for(int i = word.length() - 1; i >= 0; i--) {

            int j = word.charAt(i) - 'a';

            if(root.next[j] == null) {

                root.next[j] = new TrieNode();

            }

            if(isPalindrome(word, 0, i)) {

                root.list.add(index);

            }

            root = root.next[j];

        }

        root.list.add(index);

        root.index = index;

    }

Round 1:              Round 2:                Round 3:                  Round 4:                    Round 5:

Trie add "abcd"        Trie add "dcba"          Trie add "lls"            Trie add "s"                Trie add "sssll"

root                  root                    root                      root                        root

(index=-1,list=[])    (index=-1,list=[])    (index=-1,list=[])        (index=-1,list=[3])        (index=-1,list=[3])

  |                  /    |              /    |          \          /    |          \          /    |      |    \

  | 'd'            | 'a'  | 'd'          | 'a'  | 'd'      | 's'    | 'a'  | 'd'      | 's'    | 'a'  | 'd'  | 'l'  | 's'

  n1 (-1,[])        n5    n1 (-1,[])    n5    n1 (-1,[])  n9      n5    n1 (-1,[])  n9        n5    n1    n12    n9

  |              (-1,[])  |            (-1,[])  |        (-1,[2]) (-1,[])  |        (3,[2,3]) (-1,[])(-1,[])(-1,[])(3,[2,3])

  | 'c'            | 'b'  | 'c'          | 'b'  | 'c'      | 'l'    | 'b'  | 'c'      | 'l'    | 'b'  | 'c'  | 'l'  | 'l'

  n2 (-1,[])        n6    n2 (-1,[])    n6    n2 (-1,[])  n10      n6    n2 (-1,[])  n10      n6    n2    n13    n10

  |              (-1,[])  |            (-1,[])  |        (-1,[2]) (-1,[])  |        (-1,[2])  (-1,[])(-1,[])(-1,[4])(-1,[2])

  | 'b'            | 'c'  | 'b'          | 'c'  | 'b'      | 'l'    | 'c'  | 'b'      | 'l'    | 'c'  | 'b'  | 's'  | 'l'

  n3 (-1,[0])      n7    n3 (-1,[0])    n7    n3 (-1,[0]) n11      n7    n3 (-1,[0]) n11      n7    n3    n14    n11

  |              (-1,[1]) |            (-1,[1]) |        (2,[2])  (-1,[1]) |        (2,[2])  (-1,[1])(-1,[0])(-1,[4])(2,[2])

  | 'a'            | 'd'  | 'a'          | 'd'  | 'a'                | 'd'  | 'a'                | 'd'  | 'a'  | 's'

  n4 (0,[0])        n8    n4 (0,[0])    n8    n4 (0,[0])          n8    n4 (0,[0])            n8    n4    n15

                  (1,[1])              (1,[1])                    (1,[1])                      (1,[1])(0,[0])(-1,[4])

                                                                                                                | 's'

                                                                                                                n16

                                                                                                              (4,[4])

Each node index and list statistics:

node.index -> if a word finished, update its index in words array on last character's corresponding node, default as -1 means not a word in words array

node.list -> isPalindrome(word, 0, i)

Round 1:

Trie add word = "abcd"

--------------------------------------------------------------------------------

root (index=-1,list=[])

create n1

root.index -> -1

root.list -> isPalindrome("abcd", 0, 3) -> "abcd" not a palindrome -> list = []

move from root to n1

--------------------------------------------------------------------------------

n1 (-1,[])

create n2

n1.index -> -1

n1.list -> isPalindrome("abcd", 0, 2) -> "abc" not a palindrome -> list = []

move from n1 to n2

--------------------------------------------------------------------------------

n2 (-1,[])

create n3

n2.index -> -1

n2.list -> isPalindrome("abcd", 0, 1) -> "ab" not a palindrome -> list = []

move from n2 to n3

--------------------------------------------------------------------------------

n3 (-1,[0])

create n4

n3.index -> -1

n3.list -> isPalindrome("abcd", 0, 0) -> "a" is a palindrome -> list = [0]

move from n3 to n4

--------------------------------------------------------------------------------

n4 (0,[0])

n4.index -> 0 (word "abcd" reverse sequence as "dcba" finished at n4, last character('a')'s next level has no further character, additionally we use "root.index" to record current word "abcd" index in words array as 0 for empty string)

n4.list -> use "root.list.add(index)" to record current word "abcd" index in words array as 0 for empty string

================================================================================

Round 2:

Trie add word = "dcba"

--------------------------------------------------------------------------------

root (index=-1,list=[])

create n5

root.index -> -1

root.list -> isPalindrome("dcba", 0, 3) -> "dcba" not a palindrome -> list = []

move from root to n5

--------------------------------------------------------------------------------

n5 (-1,[])

create n6

n5.index -> -1

n5.list -> isPalindrome("dcba", 0, 2) -> "dcb" not a palindrome -> list = []

move from n5 to n6

--------------------------------------------------------------------------------

n6 (-1,[])

create n6

n6.index -> -1

n6.list -> isPalindrome("dcba", 0, 1) -> "dc" not a palindrome -> list = []

move from n6 to n7

--------------------------------------------------------------------------------

n7 (-1,[1])

create n8

n7.index -> -1

n7.list -> isPalindrome("dcba", 0, 0) -> "d" is a palindrome -> list = [1]

move from n7 to n8

--------------------------------------------------------------------------------

n8 (1,[1])

n8.index -> 0 (word "dcba" reverse sequence as "abcd" finished at n8, last character('d')'s next level has no further character, additionally we use "root.index" to record current word "dcba" index in words array as 1 for empty string)

n8.list -> use "root.list.add(index)" to record current word "dcba" index in words array as 1 for empty string

================================================================================

Round 3:

Trie add word = "lls"

--------------------------------------------------------------------------------

root (index=-1,list=[])

create n9

root.index -> -1

root.list -> isPalindrome("lls", 0, 2) -> "lls" not a palindrome -> list = []

move from root to n9

--------------------------------------------------------------------------------

n9 (-1,[2])

create n10

n9.index -> -1

n9.list -> isPalindrome("lls", 0, 1) -> "ll" is a palindrome -> list = [2]

move from n9 to n10

--------------------------------------------------------------------------------

n10 (-1,[2])

create n11

n10.index -> -1

n10.list -> isPalindrome("lls", 0, 0) -> "l" is a palindrome -> list = [2]

move from n10 to n11

--------------------------------------------------------------------------------

n11 (2,[2])

n11.index -> 2 (word "lls" reverse sequence as "sll" finished at n11, last character('l')'s next level has no further character, additionally we use "root.index" to record current word "lls" index in words array as 2 for empty string)

n11.list -> use "root.list.add(index)" to record current word "lls" index in words array as 2 for empty string

================================================================================

Round 4:

Trie add word = "s"

--------------------------------------------------------------------------------

root (index=-1,list=[]) update to (index=-1,list=[3])

no create node since already have n9

root.index -> -1

root.list -> isPalindrome("s", 0, 0) -> "s" is a palindrome -> list = [3]

move from root to n9

--------------------------------------------------------------------------------

n9 (-1,[2]) update to (3,[2,3])

n9.index -> 3 (word "s" reverse sequence as "s" finished at n9, last character('s')'s next level has no further character, additionally we use "root.index" to record current word "s" index in words array as 3 for empty string)

n9.list -> use "root.list.add(index)" to record current word "s" index in words array as 3 for empty string

================================================================================

Round 5:

Trie add word = "sssll"

--------------------------------------------------------------------------------

root (index=-1,list=[3])

create node n12

root.index -> -1

root.list -> isPalindrome("sssll", 0, 4) -> "sssll" not a palindrome -> list keep as [3]

move from root to n12

--------------------------------------------------------------------------------

n12 (-1,[])

create n13

n12.index -> -1

n12.list -> isPalindrome("sssll", 0, 3) -> "sssl" not a palindrome -> list = []

move from n12 to n13

--------------------------------------------------------------------------------

n13 (-1,[4])

create n14

n13.index -> -1

n13.list -> isPalindrome("sssll", 0, 2) -> "sss" is a palindrome -> list = [4]

move from n13 to n14

--------------------------------------------------------------------------------

n14 (-1,[4])

create n15

n14.index -> -1

n14.list -> isPalindrome("sssll", 0, 1) -> "ss" is a palindrome -> list = [4]

move from n14 to n15

--------------------------------------------------------------------------------

n15 (-1,[4])

create n16

n15.index -> -1

n15.list -> isPalindrome("sssll", 0, 0) -> "s" is a palindrome -> list = [4]

move from n15 to n16

--------------------------------------------------------------------------------

n16 (4,[4])

n16.index -> 4 (word "sssll" reverse sequence as "llsss" finished at n16, last character('s')'s next level has no further character, additionally we use "root.index" to record current word "sssll" index in words array as 4 for empty string)

n16.list -> use "root.list.add(index)" to record current word "sssll" index in words array as 4 for empty string

================================================================================

################################################################################

################################################################################

Explain on search logic:

    public void search(String[] words, int i, TrieNode root, List<List<Integer>> result) {

        for(int j = 0; j < words[i].length(); j++) {

            if(root.index >= 0 && root.index != i && isPalindrome(words[i], j, words[i].length() - 1)) {

                result.add(Arrays.asList(i, root.index));

            }

            root = root.next[words[i].charAt(j) - 'a'];

            if(root == null) {

                return;

            }

        }

        for (int j : root.list) {

            if(i == j) {

                continue;

            }

            result.add(Arrays.asList(i, j));

        }

    }

Search based on full constructed Trie tree after Round 5

        After Round 5:

              root

              (index=-1,list=[3])

          /    |      |    \

        | 'a'  | 'd'  | 'l'  | 's'

        n5    n1    n12    n9

      (-1,[])(-1,[])(-1,[])(3,[2,3])

        | 'b'  | 'c'  | 'l'  | 'l'

        n6    n2    n13    n10

      (-1,[])(-1,[])(-1,[4])(-1,[2])

        | 'c'  | 'b'  | 's'  | 'l'

        n7    n3    n14    n11

      (-1,[1])(-1,[0])(-1,[4])(2,[2])

        | 'd'  | 'a'  | 's'

        n8    n4    n15

      (1,[1])(0,[0])(-1,[4])

                      | 's'

                      n16

                    (4,[4])

Round 1:

search word = "abcd", i = 0

--------------------------------------------------------------------------------

root (index=-1,list=[3]), j = 0

check conditions -> if(root.index >= 0 && root.index != i && isPalindrome(words[i], j, words[i].length() - 1)) {result.add(Arrays.asList(i, root.index));}

root.index = -1 < 0 -> check failure: current node not the last character of an existing word which means not an option for further check

root = root.next[words[i].charAt(j) - 'a'] -> move from root to n5

--------------------------------------------------------------------------------

n5 (-1,[]), j = 1

check conditions -> if(root.index >= 0 && root.index != i && isPalindrome(words[i], j, words[i].length() - 1)) {result.add(Arrays.asList(i, root.index));}

n5.index = -1 < 0 -> check failure: current node not the last character of an existing word which means not an option for further check

n5 = n5.next[words[i].charAt(j) - 'a'] -> move from n5 to n6

--------------------------------------------------------------------------------

n6 (-1,[]), j = 2

check conditions -> if(root.index >= 0 && root.index != i && isPalindrome(words[i], j, words[i].length() - 1)) {result.add(Arrays.asList(i, root.index));}

n6.index = -1 < 0 -> check failure: current node not the last character of an existing word which means not an option for further check

n6 = n5.next[words[i].charAt(j) - 'a'] -> move from n6 to n7

--------------------------------------------------------------------------------

n7 (-1,[1]), j = 3

check conditions -> if(root.index >= 0 && root.index != i && isPalindrome(words[i], j, words[i].length() - 1)) {result.add(Arrays.asList(i, root.index));}

n7.index = -1 < 0 -> check failure: current node not the last character of an existing word which means not an option for further check

n7 = n7.next[words[i].charAt(j) - 'a'] -> move from n7 to n8

--------------------------------------------------------------------------------

n8 (1,[1]), j = 4 break out loop

Spaeical case here:

Rewind the meaning of n8 (1,[1]) => n8.index = 1 means a word during trie construstion finished at n8, n8.list = [1] means during trie construction we also record a word's index in the words array into node.list property if remain part of word is a palindrome, in another word, the prefix substring of the original word ending at next node of n8 as traversal from bottom node in trie is a palindrome, in 2nd example {"abc", "xyxcba", "xcba"} we have node n6 encounter this logic, but for now n8 is a bit differant than n6's logic, because n6 in 2nd example is an intermidiate node, which always have REAL remain part of word, but n8 is the end of a word, NO REAL remain part! Then how to continue use logic as if remain part of a word is a palindrome or not to determine if any candidate palindrome pair ? The tricky point is considering empty string "" as remain part of a full constructed word, e.g the remain part of a full constructed word "abcd" is "" when talking about n8, and that's the background how we add word "dcba"'s index = 1 into n8.list when we construct trie, it will help us avoid missing another full constructed word when search in the words array, e.g to search "abcd" palindrome in words array, after this procedure, we won't miss "dcba")

First for(int j = 0; j < words[i].length(); j++) loop ending when search word = "abcd" as j looped over length when j > 3

Second for(int j : root.list) {if(i == j) {continue;} result.add(Arrays.asList(i, j));} loop

result = {{0, 1}}

================================================================================

Round 2:

search word = "dcba", i = 1

--------------------------------------------------------------------------------

root (index=-1,list=[3]), j = 0

check conditions -> if(root.index >= 0 && root.index != i && isPalindrome(words[i], j, words[i].length() - 1)) {result.add(Arrays.asList(i, root.index));}

root.index = -1 < 0 -> check failure: current node not the last character of an existing word which means not an option for further check

root = root.next[words[i].charAt(j) - 'a'] -> move from root to n1

--------------------------------------------------------------------------------

n1 (-1,[]), j = 1

check conditions -> if(root.index >= 0 && root.index != i && isPalindrome(words[i], j, words[i].length() - 1)) {result.add(Arrays.asList(i, root.index));}

n1.index = -1 < 0 -> check failure: current node not the last character of an existing word which means not an option for further check

n1 = n1.next[words[i].charAt(j) - 'a'] -> move from n1 to n2

--------------------------------------------------------------------------------

n2 (-1,[]), j = 2

check conditions -> if(root.index >= 0 && root.index != i && isPalindrome(words[i], j, words[i].length() - 1)) {result.add(Arrays.asList(i, root.index));}

n2.index = -1 < 0 -> check failure: current node not the last character of an existing word which means not an option for further check

n2 = n2.next[words[i].charAt(j) - 'a'] -> move from n2 to n3

--------------------------------------------------------------------------------

n3 (-1,[0]), j = 3

check conditions -> if(root.index >= 0 && root.index != i && isPalindrome(words[i], j, words[i].length() - 1)) {result.add(Arrays.asList(i, root.index));}

n3.index = -1 < 0 -> check failure: current node not the last character of an existing word which means not an option for further check

n3 = n3.next[words[i].charAt(j) - 'a'] -> move from n3 to n4

--------------------------------------------------------------------------------

n4 (0,[0]), j = 4 break out loop

Spaeical case here:

Rewind the meaning of n4 (0,[0]) => n4.index = 0 means a word during trie construstion finished at n4, n4.list = [0] means during trie construction we also record a word's index in the words array into node.list property if remain part of word is a palindrome, in another word, the prefix substring of the original word ending at next node of n4 as traversal from bottom node in trie is a palindrome, in 2nd example {"abc", "xyxcba", "xcba"} we have node n6 encounter this logic, but for now n4 is a bit differant than n6's logic, because n6 in 2nd example is an intermidiate node, which always have REAL remain part of word, but n4 is the end of a word, NO REAL remain part! Then how to continue use logic as if remain part of a word is a palindrome or not to determine if any candidate palindrome pair ? The tricky point is considering empty string "" as remain part of a full constructed word, e.g the remain part of a full constructed word "abcd" is "" when talking about n4, and that's the background how we add word "abcd"'s index = 0 into n4.list when we construct trie, it will help us avoid missing another full constructed word when search in the words array, e.g to search "dcba" palindrome in words array, after this procedure, we won't miss "abcd")

First for(int j = 0; j < words[i].length(); j++) loop ending when search word = "dcba" as j looped over length when j > 3

Second for(int j : root.list) {if(i == j) {continue;} result.add(Arrays.asList(i, j));} loop

new found pair {1, 0}

result update from {{0, 1}} to {{0, 1}, {1, 0}}

================================================================================

Round 3:

search word = "lls", i = 2

--------------------------------------------------------------------------------

root (index=-1,list=[3]), j = 0

check conditions -> if(root.index >= 0 && root.index != i && isPalindrome(words[i], j, words[i].length() - 1)) {result.add(Arrays.asList(i, root.index));}

root.index = 3 < 0 -> check failure: current node not the last character of an existing word which means not an option for further check

root = root.next[words[i].charAt(j) - 'a'] -> move from root to n12

--------------------------------------------------------------------------------

n12 (-1,[]), j = 1

check conditions -> if(root.index >= 0 && root.index != i && isPalindrome(words[i], j, words[i].length() - 1)) {result.add(Arrays.asList(i, root.index));}

n12.index = -1 < 0 -> check failure: current node not the last character of an existing word which means not an option for further check

n12 = n12.next[words[i].charAt(j) - 'a'] -> move from n12 to n13

--------------------------------------------------------------------------------

n13 (-1,[4]), j = 2

check conditions -> if(root.index >= 0 && root.index != i && isPalindrome(words[i], j, words[i].length() - 1)) {result.add(Arrays.asList(i, root.index));}

n13.index = -1 < 0 -> check failure: current node not the last character of an existing word which means not an option for further check

n13 = n13.next[words[i].charAt(j) - 'a'] -> move from n13 to n14

--------------------------------------------------------------------------------

n14 (-1,[4]), j = 3 break out loop

check conditions -> if(root.index >= 0 && root.index != i && isPalindrome(words[i], j, words[i].length() - 1)) {result.add(Arrays.asList(i, root.index));}

n14.index = -1 < 0 -> check failure: current node not the last character of an existing word which means not an option for further check

Spaeical case here:

Rewind the meaning of n14 (-1,[4]) => n14.index = -1 means no word during trie construstion finished at n14, [4] means even no word construction finished at n14, but during trie construction we also record a word's index in the words array into node.list property if remain part of word is a palindrome, in another word, the prefix substring of the original word ending at next node of n14 as traversal from bottom node in trie is a palindrome, e.g the prefix substring as "ss" of "sssll" is a palindrome as traversal from bottom node n16 -> n15, ending at n15 as next node of n14, then we record word "sssll" index in words array as 4 into n14.list property)

First for(int j = 0; j < words[i].length(); j++) loop ending when search word = "lls" as j looped over length when j > 2

Second for(int j : root.list) {if(i == j) {continue;} result.add(Arrays.asList(i, j));} loop

new found pair {2, 4}

result update from {{0, 1}, {1, 0}} to {{0, 1}, {1, 0}, {2, 4}}

================================================================================

Round 4:

search word = "s", i = 3

--------------------------------------------------------------------------------

root (index=-1,list=[3]), j = 0

check conditions -> if(root.index >= 0 && root.index != i && isPalindrome(words[i], j, words[i].length() - 1)) {result.add(Arrays.asList(i, root.index));}

root.index = -1 < 0 -> check failure: current node not the last character of an existing word which means not an option for further check

root = root.next[words[i].charAt(j) - 'a'] -> move from root to n9

--------------------------------------------------------------------------------

n9 (3,[2,3]), j = 1 break out loop

Spaeical case here:

Rewind the meaning of n9 (3,[2,3]) => n9.index = 3 means a word ("s" itself as index = 3 in words array) during trie construstion finished at n9, [2,3] means during trie construction we also record a word's index in the words array into node.list property if remain part of word is a palindrome, in another word, the prefix substring of the original word ending at next node of n9 as traversal from bottom node in trie is a palindrome, e.g the prefix substring as empty string "" of "s" is a palindrome, then we record word "s" index in words array as 3 into n9.list property, another prefix subtring as "ll" of "lls" is also a palindrome as traversal from bottom node n11 -> n10, ending at n10 as next node of n9, then we record word "lls" index in words array as 2 into n9.list property)

First for(int j = 0; j < words[i].length(); j++) loop ending when search word = "s" as j looped over length when j > 0

Second for(int j : root.list) {if(i == j) {continue;} result.add(Arrays.asList(i, j));} loop -> remove the pair (i = 3 == j = 3) pair since its word itself

new found pair {3, 2}

result update from {{0, 1}, {1, 0}, {2, 4}} to {{0, 1}, {1, 0}, {2, 4}, {3, 2}}

================================================================================

Round 5:

search word = "sssll", i = 4

--------------------------------------------------------------------------------

root (index=-1,list=[3]), j = 0

check conditions -> if(root.index >= 0 && root.index != i && isPalindrome(words[i], j, words[i].length() - 1)) {result.add(Arrays.asList(i, root.index));}

root.index = -1 < 0 -> check failure: current node not the last character of an existing word which means not an option for further check

root = root.next[words[i].charAt(j) - 'a'] -> move from root to n9

--------------------------------------------------------------------------------

n9 (3,[2,3]), j = 1

check conditions -> if(root.index >= 0 && root.index != i && isPalindrome(words[i], j, words[i].length() - 1)) {result.add(Arrays.asList(i, root.index));}

n9.index = 3 > 0

n9.index = 3 != i = 4

isPalindrome(words[i], j, words[i].length() - 1) = isPalindrome("sssll", 1, 4) -> check failure: "ssll" not a palindrome

n9 = n9.next[words[i].charAt(j) - 'a'] -> null -> no more node to check

================================================================================

Done, result keep as {{0, 1}, {1, 0}, {2, 4}, {3, 2}}

################################################################################

**Refer to**

[L5.Longest Palindromic Substring](note://47F974F3D2E94D1CABDAC997F4551DB4)

[L214.Shortest Palindrome (Ref.L1044,L1698)](note://C98FF24B295744F888F2B8C18AD71093)