Palindrome Pairs

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

Example 1:
Given `words` = `["bat", "tab", "cat"]`
Return `[[0, 1], [1, 0]]`
The palindromes are `["battab", "tabbat"]`

Example 2:
Given `words` = `["abcd", "dcba", "lls", "s", "sssll"]`
Return `[[0, 1], [1, 0], [3, 2], [2, 4]]`
The palindromes are `["dcbaabcd", "abcddcba", "slls", "llssssll"]`

Credits:
Special thanks to @dietpepsi for adding this problem and creating all test cases.

Solution 1

```
public List<List<Integer>> palindromePairs(String[] words) {
    List<List<Integer>> pairs = new LinkedList<>();
    if (words == null) return pairs;
    HashMap<String, Integer> map = new HashMap<>();
    for (int i = 0; i < words.length; ++ i) map.put(words[i], i);
    for (int i = 0; i < words.length; ++ i) {
        int l = 0, r = 0;
        while (l <= r) {
            String s = words[i].substring(l, r);
            Integer j = map.get(new StringBuilder(s).reverse().toString());
            if (j != null && i != j && isPalindrome(words[i].substring(l == 0 ? r
: 0, l == 0 ? words[i].length() : l)))
                pairs.add(Arrays.asList(l == 0 ? new Integer[]{i, j} : new Intege
r[]{j, i}));
            if (r < words[i].length()) ++r;
            else ++l;
        }
    }
    return pairs;
}

private boolean isPalindrome(String s) {
    for (int i = 0; i < s.length()/2; ++ i)
        if (s.charAt(i) != s.charAt(s.length()-1-i))
            return false;
    return true;
}
```

written by Nursike original link here

Solution 2

Apparently there is a 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 need to reduce the number of words that is needed to check 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 of 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.

First here is the technique I employed to check palindrome: maintain two pointers, i and j, with i pointing to the beginning of the string, j pointing 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 above process, I do find something that we may take advantage of to get rid of words that need to be checked. For example, let's say we want to append words to wo, 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 wo is 'b' for instance, we can further reduce our candidate set to words ending with string "ba", etc. Our naive solution throws all the information away and repeats the comparison, which leads to the undesired O(n^2*k) 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), it will come to mind as we need to deal with words with common suffixes. The next step is to design fields for each node in the Trie. 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: simply starting from the

root of the Trie and the ending character of each word, 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; t ==> true. The character at 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:

1. For word "ba", starting from the first character 'b', index into the root.next array with index ('b' - 'a' = 1). The corresponding node is null, then we know there are no words ending at this character, so the search is terminated;
2. For word "a", again indexing into array root.next at index ('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 different, but the n1.isWord field provides no information about the word itself, which makes it impossible to distinguish the two words. So we need to modify the fields of the TrieNode so 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 we know the two words are 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 we have words with suffix "a" ("aaa" in this case)? We need to continue 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: each word has a suffix represented by the current Trie node; 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.

```
class TrieNode {
     TrieNode[] next;
     int index;
     List<Integer> list;

    TrieNode() {
       next = new TrieNode[26];
       index = -1;
       list = new ArrayList<>();
    }
}
```

Trie structure:

```
          root (-1,[1])
            | 'a'
          n1 (1,[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 at index ('a' - 'a' = 0) will yield node n1 and n1.index = 1 >= 0, which is also different from the index of "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 string represented by current n1 which is "a" from the beginning of "aaa") to see if it is a palindrome. If it is, then (2,1) is 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), which set the total time complexity of the solution. See the complete Java program in the answer.

written by fun4LeetCode original link here

Solution 3

There is no requirement that states i and j should be different in (i, j) pairs. So in example 2 (3, 3) should lead to 'ss' which is a palindrome. Problem statement should be corrected as:

Find all pairs of ***distinct*** indices (i, j) in the given list, so that

written by smirzai original link here