212. Word Search II

We will use Trie and Dfs to search every route in board and if it's a word in words, we will append it to our result and remove that word out of our trie. TC is O((n * n) ** 4)

```
class Trie:
    def __init__(self):
        self.trie = {}

    def insert(self, word):
        node = self.trie
        for i in word:
            if i not in node:
                node[i] = {}
            node = node[i]
        node['word'] = True

class Solution:
    def findWords(self, board: List[List[str]], words: List[str]) -> List[str]:
        rows, cols = len(board), len(board[0])
        self.result = []
        seen = set()
        self.word_trie = Trie()
        node = self.word_trie.trie
        for word in words:
            self.word_trie.insert(word)

        for i in range(rows):
            for j in range(cols):
                if board[i][j] in node:
                    seen.add((i, j))
                    self.dfs(board[i][j], board, seen, i, j, node[board[i][j]])
                    seen.remove((i, j))
        return self.result

    def dfs(self, word, board, seen, cur_i, cur_j, node):
        if 'word' in node and node['word']:
            self.result.append(word)
            node['word'] = False
        rows, cols = len(board), len(board[0])
        for d_i, d_j in [[1, 0], [-1, 0], [0, 1], [0, -1]]:
            new_i = cur_i + d_i
            new_j = cur_j + d_j
            if 0 <= new_i < rows and 0 <= new_j < cols and (new_i, new_j) not in seen and
board[new_i][new_j] in node:
```

```
                seen.add((new_i, new_j))
                self.dfs(word + board[new_i][new_j], board, seen, new_i, new_j,
node[board[new_i][new_j]])
                seen.remove((new_i, new_j))
```

126. Word Ladder II

We will use bfs to get shortest path distance and every words one word could transform to, then use dfs get every path and return the result,

```
WORDTABLE = 'abcdefghijklmnopqrstuvwxyz'
from collections import defaultdict
class Solution:
    def findLadders(self, beginWord: str, endWord: str, wordList: List[str]) -> List[List[str]]:
        wordList = set(wordList)
        res = []
        self.len = None
        self.nebor = defaultdict(set)

        self.len = self.bfs([beginWord], set(), endWord, wordList)

        if self.len > 0:
            self.dfs([beginWord], res, endWord)
        return res

    def dfs(self, cur, res, endWord):
        if len(cur) >= self.len:
            return
        for newWord in self.nebor[cur[-1]]:
            cur.append(newWord)
            if newWord == endWord:
                res.append(cur[:])
            self.dfs(cur, res, endWord)
            cur.pop()

    def bfs(self, cur, visited, endWord, wordList):
        count = 1
        mark = False
        while cur:
            next_ite = set()
            for word in cur:
                for i in range(len(word)):
                    for l in WORDTABLE:
                        newWord = word[:i] + l + word[i + 1:]
                        if newWord in wordList and newWord not in visited:
                            self.nebor[word].add(newWord)
```

```
                    if newWord == endWord:
                        mark = True
                    next_ite.add(newWord)
            for w in next_ite:
                visited.add(w)
            if mark:
                return count + 1
            cur = next_ite
            count += 1
        return -1
```

752. Open the Lock
We will use bfs to traverse all possible path until we get to the target, TC is O(1)

```
class Solution:
    def openLock(self, deadends: List[str], target: str) -> int:
        seen = set(['0000'])
        count = 0
        cur = set(['0000'])
        deadends_set = set(deadends)
        if target == '0000':
            return count
        if '0000' in deadends_set:
            return -1
        while cur:
            next_ite = set()
            count += 1
            for e in cur:
                arr = list(e)
                for i in range(4):
                    a = arr[i]
                    acc = (10 + int(a) + 1) % 10
                    dee = (10 + int(a) - 1) % 10
                    arr[i] = str(acc)
                    new_acc = ''.join(arr)
                    if new_acc not in deadends_set and new_acc not in seen:
                        if new_acc == target:
                            return count
                        next_ite.add(new_acc)
                        seen.add(new_acc)

                    arr[i] = str(dee)
                    new_dee = ''.join(arr)
```

```
                if new_dee not in deadends_set and new_dee not in seen:
                    if new_dee == target:
                        return count
                    next_ite.add(new_dee)
                    seen.add(new_dee)
                arr[i] = a
        cur = next_ite
    return -1
```

93. Restore IP Addresses

We will split whole string to any 4 combination of parts and check each part is valid, if so, we will append that combination to our result. TC is (length - 1)!

```
class Solution:
    def restoreIpAddresses(self, s: str) -> List[str]:
        def isValid(s):
            return not (int(s) < 0 or int(s) >= 256 or (len(s) > 1 and s[0] == '0'))
        length = len(s)
        result = []
        if len(s) > 3 * 4:
            return result
        for i in range(1, length - 2):
            for j in range(i + 1, length - 1):
                for k in range(j + 1, length):
                    if isValid(s[:i]) and isValid(s[i:j]) and isValid(s[j:k]) and isValid(s[k:length]):
                        result.append(s[:i] + "." + s[i:j] + "." + s[j:k] + "." + s[k:length])
        return result
```

131. Palindrome Partitioning

We will use dfs to append each likely palindrome until the end of string. TC is O(2 ** (n - 1))

```
class Solution:
    def partition(self, s: str) -> List[List[str]]:
        result = []
        def dfs(start, end, cur):
            if start == end:
                result.append(cur)
                return
            for i in range(start + 1, end + 1):
                if s[start:i] == s[start:i][::-1]:
                    dfs(i, end, cur + [s[start:i]])
        dfs(0, len(s), [])
        return result
```