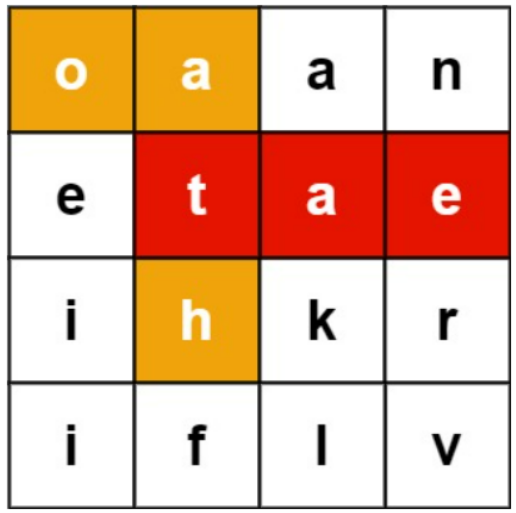
<https://leetcode.com/problems/word-search-ii/>

Given an m x n board of characters and a list of strings words, return *all words on the board*.

Each word must be constructed from letters of sequentially adjacent cells, where **adjacent cells** are horizontally or vertically neighboring. The same letter cell may not be used more than once in a word.

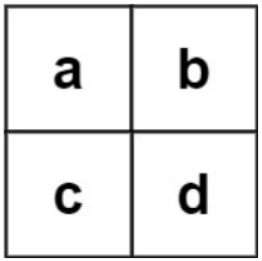
**Example 1:**



Input: board = [["o","a","a","n"],["e","t","a","e"],["i","h","k","r"],["i","f","l","v"]], words = ["oath","pea","eat","rain"]

Output: ["eat","oath"]

**Example 2:**



Input: board = [["a","b"],["c","d"]], words = ["abcb"]

Output: []

**Constraints:**

* m == board.length
* n == board[i].length
* 1 <= m, n <= 12
* board[i][j] is a lowercase English letter.
* 1 <= words.length <= 3 \* 104
* 1 <= words[i].length <= 10
* words[i] consists of lowercase English letters.
* All the strings of words are unique.

**Attempt 1: 2023-02-25**

**Solution 1: Native DFS Backtracking (10 min, TLE)**

class Solution {

public List<String> findWords(char[][] board, String[] words) {

Set<String> set = new HashSet<String>();

// Note: Don't share the usage of 'visited' boolean array for all words,

// each word should have its own 'visited' boolean array

// The error will introduce when the 1st word found, the helper() method

// will return true, and backtracking "visited[x][y] = false" will not

// be called since early return

// e.g

// char[][] board = {{'o','a','a','n'},{'e','t','a','e'},{'i','h','k','r'},{'i','f','l','v'}};

// String[] words = {"oath","pea","eat","rain"};

// Initially the 'visited' boolean array is:

// [false, false, false, false]

// [false, false, false, false]

// [false, false, false, false]

// [false, false, false, false]

// After getting "oath" the 'visited' boolean array will change to below:

// the "o, a, t, h" will persist as 'true' instead of rollback to 'false'

// [true, true, false, false]

// [false, true, false, false]

// [false, true, false, false]

// [false, false, false, false]

//boolean[][] visited = new boolean[board.length][board[0].length];

for(String word : words) {

boolean[][] visited = new boolean[board.length][board[0].length];

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

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

if(helper(board, i, j, word, 0, visited)) {

set.add(word);

}

}

}

}

return new ArrayList<String>(set);

}

int[] dx = new int[]{0,0,1,-1};

int[] dy = new int[]{1,-1,0,0};

private boolean helper(char[][] board, int x, int y, String word, int index, boolean[][] visited) {

if(index == word.length()) {

return true;

}

if(x < 0 || x >= board.length || y < 0 || y >= board[0].length || board[x][y] != word.charAt(index) || visited[x][y]) {

return false;

}

visited[x][y] = true;

for(int k = 0; k < 4; k++) {

if(helper(board, x + dx[k], y + dy[k], word, index + 1, visited)) {

return true;

}

}

visited[x][y] = false;

return false;

}

}

**Note: Don't share the usage of 'visited' boolean array for all words, each word should have its own 'visited' boolean array**

**The error will introduce when the 1st word found, the helper() method will return true, and backtracking "visited[x][y] = false" will not be called since early return**

e.g

char[][] board = {{'o','a','a','n'},{'e','t','a','e'},{'i','h','k','r'},{'i','f','l','v'}};

String[] words = {"oath","pea","eat","rain"};

**Initially the 'visited' boolean array is:**

[false, false, false, false]

[false, false, false, false]

[false, false, false, false]

[false, false, false, false]

**After getting "oath" the 'visited' boolean array will change to below: the "o, a, t, h" will persist as 'true' instead of rollback to 'false'**

[true, true, false, false]

[false, true, false, false]

[false, true, false, false]

[false, false, false, false]

**The Solution 2 working with 'Trie' no need to use individual 'visited' boolean array for each word is because DFS helper() method in Solution 2 is a void return style, the judgement is transferred on 'Trie' itself, since void return style helper() method, it won't cause early return and no backtracking afterwards.**

**Solution 2: Trie + DFS Backtracking (30 min, TLE)**

class Solution {

public List<String> findWords(char[][] board, String[] words) {

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

boolean[][] visited = new boolean[board.length][board[0].length];

Trie t = new Trie();

for(String word : words) {

t.insert(word);

}

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

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

helper(board, result, i, j, "", t, visited);

}

}

return result;

}

int[] dx = new int[] {0,0,1,-1};

int[] dy = new int[] {1,-1,0,0};

private void helper(char[][] board, List<String> result, int x, int y, String s, Trie t, boolean[][] visited) {

if(x < 0 || x >= board.length || y < 0 || y >= board[0].length || visited[x][y]) {

return;

}

s += board[x][y];

if(!t.startWith(s)) {

return;

}

if(t.search(s) && !result.contains(s)) {

result.add(s);

}

visited[x][y] = true;

for(int k = 0; k < 4; k++) {

helper(board, result, x + dx[k], y + dy[k], s, t, visited);

}

visited[x][y] = false;

}

}

class Trie {

TrieNode root;

public Trie() {

root = new TrieNode();

}

public TrieNode insert(String word) {

TrieNode p = root;

for(char c : word.toCharArray()) {

int index = c - 'a';

if(p.children[index] == null) {

p.children[index] = new TrieNode();

}

p = p.children[index];

}

p.isEnd = true;

return p;

}

public TrieNode get(String word) {

TrieNode p = root;

for(char c : word.toCharArray()) {

int index = c - 'a';

if(p.children[index] == null) {

return null;

} else {

p = p.children[index];

}

}

return p;

}

public boolean startWith(String word) {

return get(word) == null ? false : true;

}

public boolean search(String word) {

return get(word).isEnd;

}

}

class TrieNode {

TrieNode[] children;

boolean isEnd;

public TrieNode() {

this.children = new TrieNode[26];

this.isEnd = false;

}

}

**Solution 3: Promotion on Trie + DFS Backtracking (30 min)**

class Solution {

public List<String> findWords(char[][] board, String[] words) {

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

Trie t = new Trie();

for(String word : words) {

t.insert(word);

}

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

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

helper(board, result, i, j, t.root);

}

}

return result;

}

int[] dx = new int[] {0,0,1,-1};

int[] dy = new int[] {1,-1,0,0};

private void helper(char[][] board, List<String> result, int x, int y, TrieNode p) {

if(x < 0 || x >= board.length || y < 0 || y >= board[0].length) {

return;

}

char c = board[x][y];

if(c == '#' || p.children[c - 'a'] == null) {

return;

}

p = p.children[c - 'a'];

if(p.word != null) {

result.add(p.word);

p.word = null; // de-duplicate

}

board[x][y] = '#';

for(int k = 0; k < 4; k++) {

helper(board, result, x + dx[k], y + dy[k], p);

}

board[x][y] = c;

}

}

class Trie {

TrieNode root;

public Trie() {

root = new TrieNode();

}

public void insert(String word) {

TrieNode p = root;

for(char c : word.toCharArray()) {

int index = c - 'a';

if(p.children[index] == null) {

p.children[index] = new TrieNode();

}

p = p.children[index];

}

p.word = word;

}

}

class TrieNode {

TrieNode[] children = new TrieNode[26];

String word;

}

**Refer to**

<https://leetcode.com/problems/word-search-ii/solutions/59780/java-15ms-easiest-solution-100-00/>

## **Backtracking + Trie**

Intuitively, start from every cell and try to build a word in the dictionary. Backtracking (dfs) is the powerful way to exhaust every possible ways. Apparently, we need to do pruning when current character is not in any word.

1. How do we instantly know the current character is invalid? HashMap?
2. How do we instantly know what's the next valid character? LinkedList?
3. But the next character can be chosen from a list of characters. "Mutil-LinkedList"?

Combing them, Trie is the natural choice. Notice that:

1. TrieNode is all we need. search and startsWith are useless.
2. No need to store character at TrieNode. [c.next[i]](http://c.next[i]) != null is enough.
3. Never use c1 + c2 + c3. Use StringBuilder.
4. No need to use O(n^2) extra space visited[m][n].
5. No need to use StringBuilder. Storing word itself at leaf node is enough.
6. No need to use HashSet to de-duplicate. Use "one time search" trie.

For more explanations, check out [dietpepsi's blog](http://algobox.org/word-search-ii/).

## **Code Optimization**

UPDATE: Thanks to @dietpepsi we further improved from 17ms to 15ms.

* 59ms: Use search and startsWith in Trie class like [this popular solution.](https://leetcode.com/discuss/36337/my-simple-and-clean-java-code-using-dfs-and-trie)
* 33ms: Remove Trie class which unnecessarily starts from root in every dfs call.
* 30ms: Use w.toCharArray() instead of w.charAt(i).
* 22ms: Use StringBuilder instead of c1 + c2 + c3.
* 20ms: Remove StringBuilder completely by storing word instead of boolean in TrieNode.
* 20ms: Remove visited[m][n] completely by modifying board[i][j] = '#' directly.
* 18ms: check validity, e.g., if(i > 0) dfs(...), before going to the next dfs.
* 17ms: De-duplicate c - a with one variable i.
* 15ms: Remove HashSet completely. dietpepsi's idea is awesome.

The final run time is 15ms. Hope it helps!

public List<String> findWords(char[][] board, String[] words) {

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

TrieNode root = buildTrie(words);

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

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

dfs (board, i, j, root, res);

}

}

return res;

}

public void dfs(char[][] board, int i, int j, TrieNode p, List<String> res) {

char c = board[i][j];

if (c == '#' || p.next[c - 'a'] == null) return;

p = p.next[c - 'a'];

if (p.word != null) { // found one

res.add(p.word);

p.word = null; // de-duplicate

}

board[i][j] = '#';

if (i > 0) dfs(board, i - 1, j ,p, res);

if (j > 0) dfs(board, i, j - 1, p, res);

if (i < board.length - 1) dfs(board, i + 1, j, p, res);

if (j < board[0].length - 1) dfs(board, i, j + 1, p, res);

board[i][j] = c;

}

public TrieNode buildTrie(String[] words) {

TrieNode root = new TrieNode();

for (String w : words) {

TrieNode p = root;

for (char c : w.toCharArray()) {

int i = c - 'a';

if (p.next[i] == null) p.next[i] = new TrieNode();

p = p.next[i];

}

p.word = w;

}

return root;

}

class TrieNode {

TrieNode[] next = new TrieNode[26];

String word;

}

**Time Complexity analysis:**

<https://leetcode.com/problems/word-search-ii/solutions/59780/java-15ms-easiest-solution-100-00/comments/161749>

Naive way is to search for every word in the dictionary directly by DFS all cells for every word.

The time complexity will be O(m \* n \* l \* wl) where

n is board.length,

m is board[0].length,

l is words.length

wl is the average of length of words in 'words'.

With a Trie to check multiple words at the same time when DFS from a certain cell,

Time: O(m \* n \* wl) = max(O(l \* wl), O(m \* n \* l \* wl)) where

O(l \* wl) - Build the trie

O(m \* n \* l \* wl) - In the worst case where all words start with different chracters, and there is a word starting with a character in the cell board[m - 1][n - 1], we have O(m \* n \* l \* wl). However, if there are words starting with same characters and paths sharing cells, Trie can check multiple words when DFS from a certain cell, rather than check only one word when DFS from a certain cell like the naive way.

Space: O(l \* wl) = max(O(wl), O(l \* wl)) where

O(wl) - The recursive stack can grow at most to wl layers.

O(l \* wl) - In the worst case when all words start with different characters, the trie has l \* wl nodes. Also, since each

word is stored in a leaf node, all the leaf nodes require l \* wl memory.