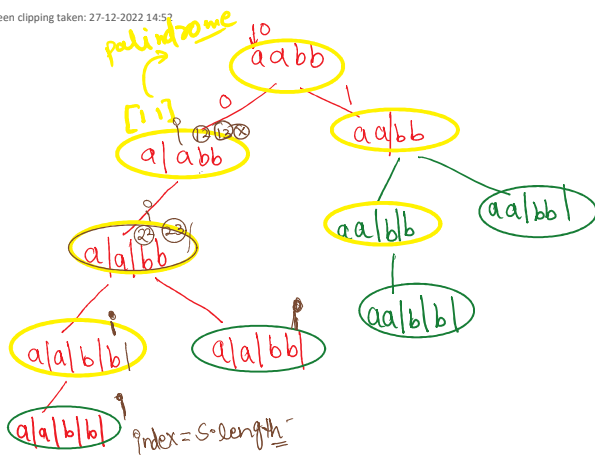


27 December 2022 14:31

Screen clipping taken: 27-12-2022 14:52





②


Screen clipping taken: 27-12-2022 15:39


## 79. Word Search


Medium




 12.4K

 499





 Companies

Given an `m x n` grid of characters `board` and a string `word`, return `true` if `word` exists in the grid.

The word can 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.

**Example 1:**

|   |   |   |   |
|---|---|---|---|
| A | B | C | E |
|---|---|---|---|

## 79. Word Search

Medium

12.4K

499

Companies

Given an  $m \times n$  grid of characters `board` and a string `word`, return `true` if `word` exists in the grid.

The word can 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.

Example 1:

|   |   |   |   |
|---|---|---|---|
| A | B | C | E |
| S | F | C | S |
| A | D | E | E |

Input: `board = [ ["A","B","C","E"], ["S","F","C","S"], ["A","D","E","E"] ]`, `word = "ABCCED"`  
Output: `true`

Screen clipping taken: 27-12-2022 15:39

Approach:-

- ① find the first character of the given string.
- ② start backtracking in all four direction untill we find all the letter of sequentially adjacent cells.
- ③ At the end if we found the result then return true or return false.

Edge cases:-

- ① Stopping Condition :-  
# if we reach the end of the boundaries of the matrix  
# or the letter at which we are making recursive call is not the require letter

#

|   | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| 0 | A | B | C | E |
| 1 | S | F | C | S |
| 2 | A | D | E | E |

Word: "ABCCED"

for (i = 0 → n)

for (j = 0 → m)

if (board[i][j] == word.charAt(index))

{ if (searchNext(board, word, i, j, index, m, n))

{ return true; }

else

return false;

Boundary Condition:-

#

|  |  |  |  |
|--|--|--|--|
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

if (row < 0 || col < 0)

row = m || col = n

|| board[row][col] != word.charAt(index)  
|| board[row][col] == '!' )

return false;

Screen clipping taken: 27-12-2022 17:44

```

1 class Solution {
2     public static boolean searchNext(char[][] board, String word, int row, int col, int index, int m, int n){
3         // if index reaches at the end that means we have found the word
4         if(index == word.length()){
5             return true;
6         }
7         // Checking the boundaries if the character at which we are placed is not
8         // the required character
9         if(row < 0 || col < 0 || row == m || col == n || board[row][col] != word.charAt(index) || board[row][col] == '1'){
10             return false;
11         }
12         // this is to prevent reusing of the same character
13         char c = board[row][col];
14         board[row][col] = '1';
15
16         boolean top = searchNext(board, word, row - 1, col, index + 1, m, n);
17         boolean bottom = searchNext(board, word, row + 1, col, index + 1, m, n);
18         boolean right = searchNext(board, word, row, col + 1, index + 1, m, n);
19         boolean left = searchNext(board, word, row, col - 1, index + 1, m, n);
20
21         board[row][col] = c; // undo change
22
23         return top || bottom || right || left;
24     }
25
26     public boolean exist(char[][] board, String word) {
27         int index = 0;
28         // First search the first character
29         int m = board.length;
30         int n = board[0].length;
31         for(int i = 0; i < m; i++){
32             for(int j = 0; j < n; j++){
33                 if(board[i][j] == word.charAt(index)){
34                     if(searchNext(board, word, i, j, index, m, n)){
35                         return true;
36                     }
37                 }
38             }
39         }
40         return false;
41     }
42 }

```

que 3 :- N Queens :-

Screen clipping taken: 27-12-2022 18:18

**51. N-Queens**

Hard 96 200

The **n-queens** puzzle is the problem of placing **n** queens on an **n x n** chessboard such that no two queens attack each other.

Given an integer **n**, return *all distinct solutions to the n-queens puzzle*. You may return the answer in *any order*.

Each solution contains a distinct board configuration of the **n-queens' placement**, where **"Q"** and **"."** both indicate a queen and an empty space, respectively.

**Example 1:**

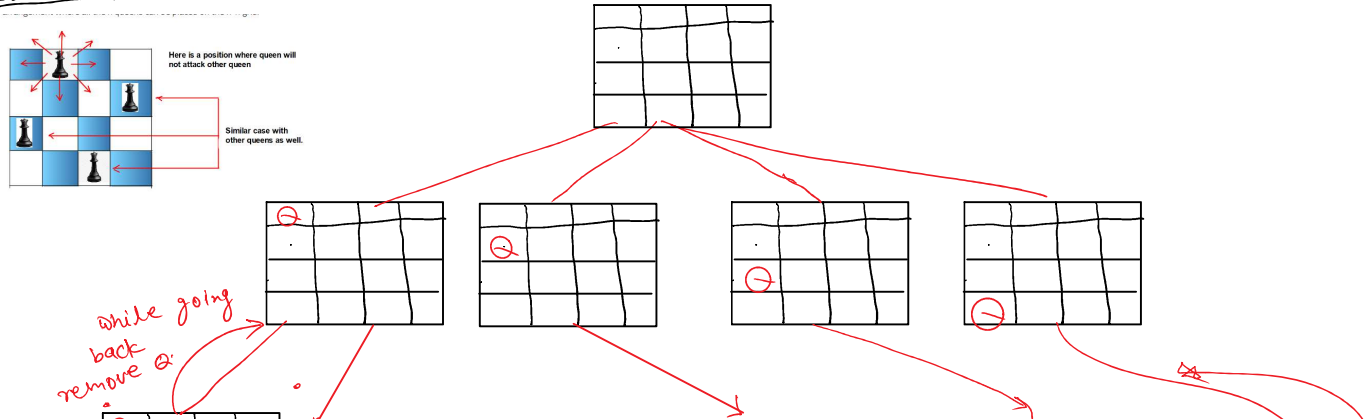
**Input:** n = 4  
**Output:** [["Q",".",".","Q"],[".","Q",".","Q"],["Q",".","Q","."],[".","Q","Q","."]]  
**Explanation:** There exist two distinct solutions to the 4-queens puzzle as shown above

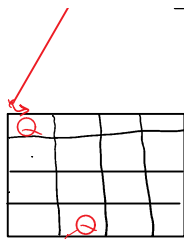
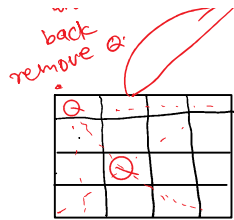
**Rules for n Queen in chessboard:**

- ① Every row should have one Queen.
- ② Every Column should have one Queen.
- ③ No two Queen can Attack each other.

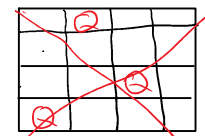
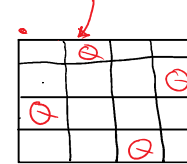
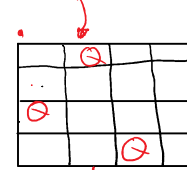
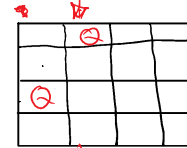
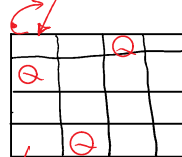
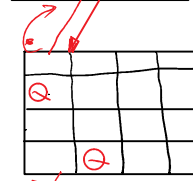
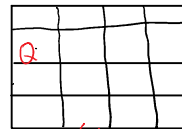
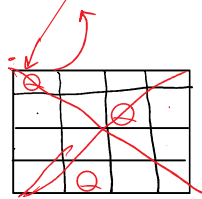
Screen clipping taken: 27-12-2022 18:21

**Solution 1:-**



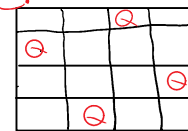


in third column  
we can see that no  
Queen can be placed.



now  
we  
will  
backtrack

now we  
will back  
track.



this will be  
stored as our  
answer.

this will  
be stored  
as our  
answer

bool isSafe :-

diagonal condition

① while(row >= 0 && col >= 0)  
if(board[row][col] == 'Q')  
row--  
col--

② any ~~have~~ have Queen (col >= 0) col--

③ while(row < n && col >= 0) →  
row++;  
col--;

if (col == n)

{ ans.add(board)

} return;