907. Sum of Subarray Minimums
We will iterate through all elements and use a stack to store all elements less than current element. We will add all minimum values in all intervals. TC is O(n)

```
class Solution:
    def sumSubarrayMins(self, A: List[int]) -> int:
        stack = [0]
        A = [0] + A
        result = [0] * len(A)
        for i in range(1, len(A)):
            while A[stack[-1]] > A[i]:
                stack.pop()
            j = stack[-1]
            result[i] = A[i] * (i - j) + result[j]
            stack.append(i)
        return sum(result) % (10**9 + 7)
```

22. Generate Parentheses
We will use dfs to append parenthese to current one. TC is O(n)

```
class Solution:
    def generateParenthesis(self, n: int) -> List[str]:
        next_ite = []
        res = []
        def dfs(cur, l, r):
            if l == r and l == n:
                res.append(cur)
                return
            if l > n or r > n:
                return
            if l > r:
                dfs(cur + ')', l, r + 1)
            if r < n:
                dfs(cur + '(', l + 1, r)
        dfs('', 0, 0)
        return res
```

37. Sudoku Solver
We will use set to record all rows, columns and cube’s status and dfs to fill all cells. TC is O(1)

```
class Solution:
    def solveSudoku(self, board: List[List[str]]) -> None:
        """
        Do not return anything, modify board in-place instead.
        """
        sets = [set() for i in range(27)]
```

```
        count = 9 * 9
        for i in range(9):
            for j in range(9):
                if board[i][j] != '.':
                    sets[i].add(board[i][j])
                    sets[9 + j].add(board[i][j])
                    sets[18 + i // 3 * 3 + j // 3].add(board[i][j])
                    count -= 1
        self.fill(board, sets, 0, 0, count)

    def fill(self, board, sets, start_i, start_j, rest):
        if rest == 0:
            return True
        i, j = start_i, start_j
        while i < 9:
            while j < 9:
                if board[i][j] == '.':
                    for k in range(1, 10):
                        ch = str(k)
                        if ch not in sets[i] and ch not in sets[9 + j] and ch not in sets[18 + i // 3 * 3 + j // 3]:
                            board[i][j] = ch
                            sets[i].add(ch)
                            sets[9 + j].add(ch)
                            sets[18 + i // 3 * 3 + j // 3].add(ch)
                            if self.fill(board, sets, i, j, rest-1):
                                return True
                            sets[i].remove(ch)
                            sets[9 + j].remove(ch)
                            sets[18 + i // 3 * 3 + j // 3].remove(ch)
                            board[i][j] = '.'
                    return False
                j += 1
            j = 0
            i += 1
        return True
```

79. Word Search

We will use dfs to search all possible path and return True if there is one existing. TC is O(row * col * len)

```
class Solution:
    def exist(self, board: List[List[str]], word: str) -> bool:
        directions = [[0, 1], [0, -1], [1, 0], [-1, 0]]
        if not board or not board[0]:
```

```
            return False
        if not word:
            return True
        def findNextWord(index, visited, i, j):
            if index == len(word):
                return True
            for d_i, d_j in directions:
                new_i = i + d_i
                new_j = j + d_j
                if 0 <= new_i < rows and 0 <= new_j < cols and (new_i, new_j) not in visited and
word[index] == board[new_i][new_j]:
                    visited.add((new_i, new_j))
                    if findNextWord(index + 1, visited, new_i, new_j):
                        return True
                    visited.remove((new_i, new_j))
            return False
        rows = len(board)
        cols = len(board[0])
        for i in range(rows):
            for j in range(cols):
                if board[i][j] == word[0]:
                    if findNextWord(1, set([(i, j)]), i, j):
                        return True
        return False
```

127. Word Ladder

We will use bfs to search all possible next words we could get until there is no any unvisited words.

```
class Solution:
    def ladderLength(self, beginWord: str, endWord: str, wordList: List[str]) -> int:
        wordSet = set(wordList)
        count = 1
        ite_words = 'abcdefghijklmnopqrstuvwxyz'

        if endWord not in wordList:
            return 0

        cur_ite = set([beginWord])
        if beginWord in wordSet:
            wordSet.remove(beginWord)
        wordSet.remove(endWord)
        other_ite = set([endWord])
        next_ite = set()
```

```
visited = set()
while cur_ite:
    for c in cur_ite:
        for i in range(len(c)):
            for l in ite_words:
                new_word = c[:i] + l + c[i + 1:]
                if new_word in other_ite:
                    return count + 1
                if new_word in wordSet:
                    next_ite.add(new_word)
                    wordSet.remove(new_word)
    count += 1
    if len(cur_ite) > len(other_ite):
        cur_ite = other_ite
        other_ite = next_ite
    else:
        cur_ite = next_ite
    next_ite = set()
return 0
```