37. Sudoku Solver
We will use set array to record which number has existed on each row, column and small block. And we will use an array to manage our ‘.’. For each empty cell, we will use every possible number and try other cells until our empty array is empty, we will return True. TC is O(9 ^ 27)

```
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)]
        empty_set = []
        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])
                else:
                    empty_set.append((i, j))
        self.fill(board, sets, empty_set)
    def fill(self, board, sets, empty_set):
        if not empty_set:
            return True
        i , j = empty_set.pop()
        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(board[i][j])
                sets[9 + j].add(board[i][j])
                sets[18 + i // 3 * 3 + j // 3].add(board[i][j])
                if self.fill(board, sets, empty_set):
                    return True
                sets[i].remove(board[i][j])
                sets[9 + j].remove(board[i][j])
                sets[18 + i // 3 * 3 + j // 3].remove(board[i][j])
                board[i][j] = '.'
        empty_set.append((i,j))
```

150. Evaluate Reverse Polish Notation
We will iterate through the whole array from left to right. We will use a stack to store data. If it’s an operator, we will pop two numbers from stack and append the result to the stack. We should leave div first when the div result is negative, which is - (abs(a) // abs(b)). TC is O(n)

```
class Solution:
    def evalRPN(self, tokens: List[str]) -> int:
        stack = []
        for token in tokens:
            if token not in '+-*/':
                stack.append(int(token))
            else:
                num2 = stack.pop()
                num1 = stack.pop()
                if token == '+':
                    stack.append(num1 + num2)
                elif token == '-':
                    stack.append(num1 - num2)
                elif token == '*':
                    stack.append(num1 * num2)
                elif token == '/':
                    if num1 // num2 >= 0:
                        stack.append(num1 // num2)
                    else:
                        stack.append(-(abs(num1) // abs(num2)))
        return stack[0]
```

1169. Invalid Transactions
We will iterate through all transactions and use a dict to store all people’s transaction by time order. When the current transaction’s time belong to somebody who had transaction, we will start from the tail of those transactions until time difference is larger than 60. TC is O(n * n)

```
from collections import defaultdict
class Solution:
    def invalidTransactions(self, transactions: List[str]) -> List[str]:
        time_memo = defaultdict(list)
        result = set()
        transactions = list(map(lambda t: t.split(','), transactions))
        transactions.sort(key=lambda a: (int(a[1]), a[0], a[2], a[3]))
        for t in transactions:
            name, time, amount, city = t

            if name in time_memo:
                for prev_time, prev_city, tra in time_memo[name][::-1]:
                    if int(time) - prev_time <= 60:
                        if city != prev_city:
                            result.add(','.join(t))
                            result.add(tra)
                    else:
```

```
                    break

            time_memo[name].append((int(time), city, ','.join(t)))
            if int(amount) > 1000:
                result.add(','.join(t))
        return list(result)
```

1170. Compare Strings by Frequency of the Smallest Character
We will words frequency and sort them. Then go through all words in queries and get number in words whose number is larger than it. And append it to our result. TC is O(nlogn + m)

```
from bisect import *
from collections import defaultdict
class Solution:
    def numSmallerByFrequency(self, queries: List[str], words: List[str]) -> List[int]:
        result = []
        length = len(words)
        words = list(map(lambda a: self.getFre(a), words))
        words.sort()
        for word in queries:
            result.append(length - bisect_right(words, self.getFre(word)))
        return result

    def getFre(self, word):
        memo = defaultdict(int)
        min_word = min(list(word))
        for w in word:
            memo[w] += 1
        return memo[min_word]
```

1171. Remove Zero Sum Consecutive Nodes from Linked List
We will use a map to record all sum from beginning to current node values and its node. When later we encounter some node of that sum is equal to some value in our map. We will let node in that map connect to current node’s next node. TC is O(n)

```
class Solution:
    def removeZeroSumSublists(self, head: ListNode) -> ListNode:
        memo = {}
        dummy = ListNode(0)
        dummy.next = head
        memo[0] = dummy
        cur_sum = 0

        while dummy:
```

```
    cur_sum += dummy.val
    if cur_sum in memo:
        memo[cur_sum].next = dummy.next
    else:
        memo[cur_sum] = dummy
    dummy = dummy.next

return memo[0].next
```