5. Longest Palindromic Substring

```
class Solution:
    def longestPalindrome(self, s: str) -> str:
        result = ''
        if not s or len(s) <= 1:
            return s

        length = len(s)

        for idx in range(0, length - 1):
            tmp = self.helper(idx, idx, s)
            if len(tmp) > len(result):
                result = tmp

            tmp = self.helper(idx, idx + 1, s)
            if len(tmp) > len(result):
                result = tmp

        return result

    def helper(self, l, r, s):
        while l >= 0 and r < len(s) and s[l] == s[r]:
            l -= 1
            r += 1
        return s[l + 1:r]
```

22. Generate Parentheses

We will use dfs to get all possible combinations. TC is O(n^2)

```
class Solution:
    def generateParenthesis(self, n: int) -> List[str]:
        result = []
        def helper(left_num, cur, target):
            if target == 0:
                if left_num == 0:
                    result.append(cur)
                return
            if left_num > 0:
                helper(left_num - 1, cur + ')', target - 1)
            helper(left_num + 1, cur + '(', target - 1)
        helper(0, '', n * 2)
        return result
```

146. LRU Cache

```
class Node:
  def __init__(self, key, val, prev, next):
    self.key = key
    self.val = val
    self.prev = prev
    self.next = next

class LRUCache:

    def __init__(self, capacity: int):
        self.map = {}
        self.capacity = capacity
        node = Node(0, 0, None, None)
        self.dummy = node
        self.dummy.next = node
        self.dummy.prev = node

    def get(self, key: int) -> int:
        if key not in self.map:
          return -1
        node = self.map[key]
        self.delete_node(node)
        self.insert_head(node)
        return node.val

    def put(self, key: int, value: int) -> None:
        if key in self.map:
          node = self.map[key]
          node.val = value
          self.delete_node(node)
          self.insert_head(node)
        else:
          if self.capacity == 0:
            node = self.dummy.prev
            self.delete_node(node)
            del self.map[node.key]
            self.capacity += 1
          node = Node(key, value, None, None)
          self.map[key] = node
          self.insert_head(node)
          self.capacity -= 1
```

```
    def delete_node(self, node):
        node.prev.next = node.next
        node.next.prev = node.prev

    def insert_head(self, node):
        node.next = self.dummy.next
        node.prev = self.dummy
        node.next.prev = node
        self.dummy.next = node


# Your LRUCache object will be instantiated and called as such:
# obj = LRUCache(capacity)
# param_1 = obj.get(key)
# obj.put(key,value)
```

794. Valid Tic-Tac-Toe State

Straightforward. Handle all cases directly.

```
class Solution:
    def validTicTacToe(self, board: List[str]) -> bool:
        num_X = 0
        num_O = 0
        success_num = 0
        for s in board:
          num_X += s.count('X')
          num_O += s.count('O')
        if num_X < num_O or num_X - num_O > 1:
          return False

        for i in range(3):
            if board[i][0] != ' ' and board[i][0] == board[i][1] == board[i][2]:
              if num_X == num_O and board[i][0] == 'X' or (num_X > num_O and board[i][0] == 'O'):
                return False
              success_num += 1
        if success_num > 1:
          return False

        success_num = 0
        for j in range(3):
            if board[0][j] != ' ' and board[0][j] == board[1][j] == board[2][j]:
              if num_X == num_O and board[0][j] == 'X' or (num_X > num_O and board[0][j] == 'O'):
```

```
            return False
          success_num += 1
      if success_num > 1:
        return False

      success_num = 0
      if board[1][1] != ' ' and (board[0][0] == board[1][1] == board[2][2] or board[2][0] ==
board[1][1] == board[0][2]):
        if num_X == num_O and board[1][1] == 'X' or (num_X > num_O and board[1][1] == 'O'):
          return False
      return True
```

445. Add Two Numbers II

We will use stack to solve this question. TC is O(n), SC is O(n)

```
# Definition for singly-linked list.
# class ListNode:
#     def __init__(self, x):
#         self.val = x
#         self.next = None

class Solution:
    def addTwoNumbers(self, l1: ListNode, l2: ListNode) -> ListNode:
        stack1 = []
        stack2 = []
        stack3 = []
        carry = 0
        dummy = ListNode(0)
        dummy_mem = dummy
        while l1:
          stack1.append(l1.val)
          l1 = l1.next

        while l2:
          stack2.append(l2.val)
          l2 = l2.next

        while stack1 and stack2:
          carry, rest = divmod(stack1.pop() + stack2.pop() + carry, 10)
          stack3.append(rest)

        stack1 = stack1 or stack2
        while stack1:
          carry, rest = divmod(stack1.pop() + carry, 10)
```

```
  stack3.append(rest)
if carry:
  stack3.append(carry)
while stack3:
  dummy.next = ListNode(stack3.pop())
  dummy = dummy.next
return dummy_mem.next
```