53. Maximum Subarray

For this question, we could iterate through the whole array and add previous number and current one. We would store the current maximum result for contiguous subarray including the current one. Every time we store the maximum number.

```
class Solution:
    def maxSubArray(self, nums: List[int]) -> int:
        result = 0
        max_res = -float('inf')
        for num in nums:
            result = max(result + num, num)
            max_res = max(result, max_res)
        return max_res
```

21. Merge Two Sorted Lists

We will compare the first element of two arrays. And connect the smaller one to the new linked list. When either one is done. We will connect the rest linkedlist.

```
class Solution:
    def mergeTwoLists(self, l1: ListNode, l2: ListNode) -> ListNode:
        dummy_head = ListNode(0)
        dummy_head_memo = dummy_head
        while l1 and l2:
            if l1.val < l2.val:
                dummy_head.next = l1
                l1 = l1.next
            else:
                dummy_head.next = l2
                l2 = l2.next
            dummy_head = dummy_head.next

        if l1:
            dummy_head.next = l1
        elif l2:
            dummy_head.next = l2

        return dummy_head_memo.next
```

56. Merge Intervals

This question is very easy. We just sort it by the first element of each interval. Then we only need to compare the previous one's last element and current interval. If there is an overlap, we could merge this one to the previous one, or we need to push this one to the final array we store our intervals.

```
class Solution:
    def merge(self, intervals: List[List[int]]) -> List[List[int]]:
        intervals.sort(key=lambda a: a[0])
        pre = None
        result = []
        for interval in intervals:
            if pre and interval[0] <= pre[1]:
                pre[1] = max(interval[1], pre[1])
            else:
                pre = interval
                result.append(pre)
        return result
```

23. Merge k Sorted Lists

For this question, we would push all first node's value and node as a tuple to a priority queue. Then pop out the minimum node, push next one if there is. So in this case, TC is O(logn)

```
from Queue import PriorityQueue
class Solution(object):
    def mergeKLists(self, lists):
        """
        :type lists: List[ListNode]
        :rtype: ListNode
        """
        head = ListNode(0)
        head_mem = head
        h = PriorityQueue()
        for l in lists:
            if l:
                h.put((l.val, l))
        while not h.empty():
            _, node = h.get()
            head.next = node
            head = head.next
            if node.next:
                node = node.next
                h.put((node.val, node))
        return head_mem.next
```

20. Valid Parentheses

This question is very classic. We should use a stack to store either of left parentheses, when we encounter right ones, we should pop out last one in the stack and check whether they are pairs.

If not, we should return False. In the end, we will check whether our stack is empty to confirm every parenthes is paired. The TC is O(n).

```
class Solution(object):
    def isValid(self, s):
        """
        :type s: str
        :rtype: bool
        """
        table = {'(': ')', '{': '}', '[': ']'}
        length = len(s)
        stack = []
        for i in range(length):
            if s[i] in '({[':
                stack.append(s[i])
            else:
                if not stack or table[stack.pop()] != s[i]:
                    return False
        return len(stack) == 0
```