149. Max points in a line

We will compare point by point and get their slope. If they have the same slope and pass through a same point. Then they will be in the same line. We also need to consider line horizontally and same points. TC is O(n * n)

```
from collections import defaultdict
class Solution:
    def maxPoints(self, points: List[List[int]]) -> int:
        memo = defaultdict(int)
        length  = len(points)
        if not points:
            return 0

        max_count = 1
        for i in range(length - 1):
            memo.clear()
            same_point = 1
            for j in range(i + 1, length):
                if points[i][1] == points[j][1]:
                    if points[i][0] == points[j][0]:
                        same_point += 1
                    else:
                        memo[float('inf')] += 1
                else:
                    memo[(points[i][0] - points[j][0]) / (points[i][1] - points[j][1])] += 1
            values = list(memo.values())
            max_count = max(max_count, (max(values) if values else 0) + same_point)
        return max_count
```

239. Sliding Window Maximum

We will use a deque to maintain slide window’s max value. Every time when we append current’s num’s index to deque, we will pop out all smaller numbers’ index, so our d[0] always stores the largest number’s index of the slide window. When window’s size is equal to k, we will start to add our maximum number of window to our result. TC is O(m)

```
from collections import deque
class Solution:
    def maxSlidingWindow(self, nums: List[int], k: int) -> List[int]:
        d = deque()
        out = []

        for i, num in enumerate(nums):
            while d and nums[d[-1]] < num:
                d.pop()
            d.append(i)
            if d[0] == i - k:
                d.popleft()
            if i >= k - 1:
                out.append(nums[d[0]])
        return out
```

529. Minesweeper

We will use dfs to reveal ‘E’ cell from click position. If the click one is mine. We will reassign it and return board. Game is over. If not, we will dfs all adjacent cells. For each cell, we will check all its adjacent cells to count how many mines are there. If it’s 0, we will change cell’s value to ‘B’ and keep dfs its adjacent cells. If it’s larger than 0, we will change it’s value to str(value). TC is O(n).

```
class Solution:
    def updateBoard(self, board: List[List[str]], click: List[int]) -> List[List[str]]:
        if not board or not board[0]:
            return board

        if board[click[0]][click[1]] == 'M':
            board[click[0]][click[1]] = 'X'
            return board

        rows = len(board)
        cols = len(board[0])
        self.dfs(click, board, rows, cols)
        return board

    def dfs(self, cur, board, rows, cols):
        i, j = cur
        directions = [[0, 1], [0, -1], [1, 0], [-1, 0], [-1, -1], [1, 1], [-1, 1], [1, -1]]
        if board[i][j] != 'E':
          return
        count = 0
        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:
                if board[new_i][new_j] == 'M':
                  count += 1
        if count == 0:
          board[i][j] = 'B'
          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:
                self.dfs([new_i, new_j], board, rows, cols)
        else:
          board[i][j] = str(count)
```

57. Insert Interval

We will add it into intervals and sort it. Then it downgrades to merge interval question. TC is O(nlogn). But I just insert the interval into before the interval that has overlap with it. And start merging since that point. So my TC is O(n)

```
class Solution:
    def insert(self, intervals: List[List[int]], newInterval: List[int]) -> List[List[int]]:
        result = []
        index = -1
        for i, interval in enumerate(intervals):
          if newInterval[0] > interval[1]:
            result.append(interval)
          else:
            index = i
            break
        if index == -1:
          result.append(newInterval)
          return result
        else:
          result.append(newInterval)
          for interval in intervals[index:]:
            if result[-1][0] > interval[1] or result[-1][1] < interval[0]:
              result.append(interval)
            else:
              result[-1][0] = min(result[-1][0], interval[0])
              result[-1][1] = max(result[-1][1], interval[1])

        return result
```

24. Swap Nodes in Pairs

We will use dummy as the first node and appending head behind it.

We will first check next node and next next node both exist. Then we will swap these two nodes. Then repeat this process to the end. Then return dummy_memo.next. TC is O(n)

```
class Solution:
    def swapPairs(self, head: ListNode) -> ListNode:
        dummy = ListNode(0)
        dummy.next = head
        dummy_memo = dummy
```

```
while dummy.next and dummy.next.next:
  temp = dummy.next
  second_next = dummy.next.next.next
  dummy.next = dummy.next.next
  dummy.next.next = temp
  temp.next = second_next
  dummy = dummy.next.next
return dummy_memo.next
```