10. Regular Expression Matching

This question is quite difficult for me. We could use two dimension DP for this question. For s[i] and p[j], if s[i] == p[j], we will set match[i + 1][j + 1] = match[i][j]. The same as when p[j] == '.'. When p[j] == '*', we will discuss it into two situations: 1, when p[j - 1] != '.' and s[i] != p[j - 1] and p[j - 1] != '.', we only could remove p[j] and p[j -1]. Otherwise, we should set 'a*' as single 'a', multiple 'a' or empty. So if any match[i + 1][j] or match[i][j + 1] or match[i + 1][j - 1] is True, we will set match[i + 1][j + 1] True. TC is O(n**2)

```
class Solution:
    def isMatch(self, s: str, p: str) -> bool:
        rows, cols = len(s), len(p)
        match = [[False for _ in range(cols + 1)] for _ in range(rows + 1)]
        match[0][0] = True

        for j in range(cols):
            if p[j] == '*':
                match[0][j + 1] =  match[0][j] or match[0][j - 1]

        for i in range(0, rows):
            for j in range(0, cols):
                if s[i] == p[j] or p[j] == '.':
                    match[i + 1][j + 1] = match[i][j]
                elif p[j] == '*':
                    if p[j - 1] != '.' and s[i] != p[j - 1]:
                        match[i + 1][j + 1] = match[i + 1][j - 1]
                    else:
                        match[i + 1][j + 1] = match[i + 1][j] or match[i][j + 1] or match[i + 1][j - 1]
        return match[-1][-1]
```

295. Find Median from Data Stream

For this question, we could use two heap, one is to store large part, another one to store small part. Each time we add new number, if two parts have the same amount of number, we should heappush num to max part and pop out a min number, which is pushed to min part. When, min part has 1 more element than max part, we should push new number to min part and pop out max one(we use - to get the maximum). TC for addNum is Log(n), for findMedian is O(1).

```
from heapq import *
class MedianFinder:

    def __init__(self):
        """
        initialize your data structure here.
        """
        self.max_heap = []
        self.min_heap = []
```

```
    def addNum(self, num: int) -> None:
        if len(self.min_heap) == len(self.max_heap):
            heappush(self.min_heap, -heappushpop(self.max_heap, num))
        else:
            heappush(self.max_heap, -heappushpop(self.min_heap, -num))

    def findMedian(self) -> float:
        if len(self.min_heap) - len(self.max_heap) == 1:
            return -self.min_heap[0]
        else:
            return (-self.min_heap[0] + self.max_heap[0]) / 2
```

981. Time Based Key-Value Store

For this question, we could store our key value in a nested map, which is [key][timestamp], and use another map to store all timestamp for that key. We could binary search timestamp and find the prev_stamp in our timestamp map and get the final value in our nested map. Time Complexity is O(1) for set, O(logn) for get.

```
from collections import defaultdict
from bisect import bisect_right
class TimeMap:

    def __init__(self):
        """
        Initialize your data structure here.
        """
        self.map = defaultdict(dict)
        self.timestamp = defaultdict(list)

    def set(self, key: str, value: str, timestamp: int) -> None:
        self.map[key][timestamp] = value
        self.timestamp[key].append(timestamp)

    def get(self, key: str, timestamp: int) -> str:
        if key not in self.map:
            return ''
        new_index = bisect_right(self.timestamp[key], timestamp)
        if new_index == 0:
            return ''
        return self.map[key][self.timestamp[key][new_index - 1]]
```

31. Next Permutation

It’s a math question.

```
class Solution:
    def nextPermutation(self, nums: List[int]) -> None:
        """
        Do not return anything, modify nums in-place instead.
        """
        length = len(nums)
        k = None
        l = None

        for i in range(length - 1, 0, -1):
            if nums[i - 1] < nums[i]:
                k = i - 1
                break
        if k == None:
            return nums.reverse()

        for i in range(length - 1, k, -1):
            if nums[i] > nums[k]:
                nums[i], nums[k] = nums[k], nums[i]
                break

        l, r = k + 1, length - 1
        while l < r:
            nums[l], nums[r] = nums[r], nums[l]
            l += 1
            r -= 1
```

46. Permutations

This is very simple. We could take advantage of python’s built-in methods to solve this question.

O(n*n)

```
from itertools import permutations
class Solution:
    def permute(self, nums: List[int]) -> List[List[int]]:
        return list(permutations(nums))
```