687. Longest Univalue Path

We will traverse our trees post-orderly and get left maximum same nodes as the left node and right maximum same nodes as the right node. If left and right nodes could connect, we will compare max_path with 1 + left + right and return 1 + max(left, right), if not, we will return 1 + left or right which has the same value as the current node. TC is O(n)

```
class Solution:
    def longestUnivaluePath(self, root: TreeNode) -> int:
      if not root:
        return 0
      self.max_path = 0
      def helper(node):
        if not node:
          return 0
        left = helper(node.left)
        right = helper(node.right)
        val = 1
        if node.left and node.val == node.left.val and node.right and node.val == node.right.val:
          self.max_path = max(self.max_path, val + left + right)
          val += max(left, right)
          return val
        if node.right and node.val == node.right.val:
          val += right
        elif node.left and node.val == node.left.val:
          val += left
        self.max_path = max(self.max_path, val)
        return val
      helper(root)
      return self.max_path - 1
```

299. Bulls and Cows

We will iterate our source and target and calculate all letters' number, set bull as same letter's number and cow as rest same letter not in the same position. TC is O(n)

```
from collections import defaultdict
class Solution:
    def getHint(self, secret: str, guess: str) -> str:
        s_memo = defaultdict(int)
        g_memo = defaultdict(int)
        length = len(secret)
        bulls, cows = 0, 0
        for i in range(length):
          if secret[i] == guess[i]:
            bulls += 1
          else:
```

```
            s_memo[secret[i]] += 1
            g_memo[guess[i]] += 1
        for k, v in s_memo.items():
          if k in g_memo:
            cows += min(g_memo[k], v)
        return str(bulls) + 'A' + str(cows) + 'B'
```

939. Minimum Area Rectangle

We will store all points according to their x. Then compare every two points on the same row and store every row paris in a hashmap, if there is already one their, we will get area and compare with min_area to get the min one. TC is O(n**2)

```
from collections import defaultdict
class Solution:
    def minAreaRect(self, points: List[List[int]]) -> int:
        memo = defaultdict(list)
        seen = defaultdict(int)
        n = len(points)
        nx = len(set(x for x, y in points))
        ny = len(set(y for x, y in points))
        if nx == n or ny == n:
            return 0
        min_area = float('inf')
        points.sort()
        for x, y in points:
            memo[y].append(x)

        keys = list(memo.keys())
        keys.sort()
        for key in keys:
            for i in range(1, len(memo[key])):
                for j in range(i):
                    if memo[key][j] != memo[key][i]:
                        if (memo[key][j], memo[key][i]) in seen:
                            min_area = min(min_area, (key - seen[(memo[key][j], memo[key][i])]) *
(memo[key][i] - memo[key][j]))
                        seen[(memo[key][j], memo[key][i])] = key
        return 0 if min_area == float('inf') else min_area
```

1087. Brace Expansion

We will split S by ‘{’ or ‘}’ and split each substring by comma. Then we only need to combine each letter in the string to our existed substrings. TC is O(n**2)

```
import re
class Solution:
```

```
    def expand(self, S: str) -> List[str]:
        res = []
        for arr in filter(lambda a: a, re.split('{|}', S)):
            if not res:
                res = arr.split(',')
            else:
                next_ite = []
                words = arr.split(',')
                for s in res:
                    for e in words:
                        next_ite.append(s + e)
                res = next_ite
        return sorted(res)
```

753. Cracking the Safe

We will take advantage of De Bruijn sequence: and dfs to add all digits that haven't been seen. We always could find that number. TC is O(k * n)\

```
class Solution:
    def crackSafe(self, n: int, k: int) -> str:
        seen = set()
        res = []
        def dfs(prefix):
            for i in map(str, reversed(range(k))):
                s = prefix + i
                if s not in seen:
                    seen.add(s)
                    res.append(i)
                    dfs(s[1:])
        dfs('0' * (n - 1))
        return '0' * (n - 1) + ''.join(res)
```