745. Prefix and Suffix Search
We will use a dict to record every possible (prefix + '.' + suffix, index). Then when we want to find any combination. We could get it immediately. TC is f: O(1) init: O(n*len*len)

```
class WordFilter:
    def __init__(self, words: List[str]):
        self.input = {}
        prefix = ''

        for idx, word in enumerate(words):
            prefix = ''
            for i in [''] + list(word):
                prefix += i
                suffix = ''
                for j in [''] + list(word)[::-1]:
                    suffix = j + suffix
                    self.input[prefix + '.' + suffix] = idx

    def f(self, prefix: str, suffix: str) -> int:
        findWord = prefix + '.' + suffix
        if findWord in self.input:
            return self.input[findWord]
        return -1
```

340. Longest Substring with At Most K Distinct Characters
We will use slide window to get the longest string. We will use a dictionary to memorize our each character's present time. Once their presenting time is larger than k. We will get the max_length, and move left to right until count is equal to k. In the end, we need to move right by 1 to start the next sliding window. TC O(n)

```
from collections import defaultdict
class Solution:
    def lengthOfLongestSubstringKDistinct(self, s: str, k: int) -> int:
        left, right = 0, 0
        memo = defaultdict(int)
        count = 0
        max_length = 0

        if not s:
            return 0

        while right < len(s):
            while count <= k and right < len(s):
                if memo[s[right]] == 0:
```

```
                count += 1
                if count > k:
                    memo[s[right]] += 1
                    break
            memo[s[right]] += 1
            right += 1

        max_length = max(right - left, max_length)
        while count > k:
            if memo[s[left]] == 1:
                count -= 1
            memo[s[left]] -= 1
            left += 1
        right += 1
    return max_length
```

72. Edit Distance

We will use dp to get minmum change times from previous result. If word1[i - 1][j - 1] == word2[i - 1][j - 1], then dp[i][j] = dp[i - 1][j - 1], else we need get the minimum of insert, delete, replace From the previous one. That's dp[i][j] = 1 + min(dp[i - 1][j], dp[i][j], dp[i][j - 1]) TC is O(mn)

```
class Solution:
    def minDistance(self, word1: str, word2: str) -> int:
        length_1 = len(word1) + 1
        length_2 = len(word2) + 1
        max_length = 0
        dp = [[0 for j in range(length_2)] for i in range(length_1)]
        for i in range(length_1):
            dp[i][0] = i
        for j in range(length_2):
            dp[0][j] = j
        for i in range(1, length_1):
            for j in range(1, length_2):
                if word1[i - 1] == word2[j - 1]:
                    dp[i][j] = dp[i - 1][j - 1]
                else:
                    dp[i][j] = 1 + min(dp[i - 1][j], dp[i - 1][j - 1], dp[i][j - 1])
        return dp[-1][-1]
```

98. Validate Binary Search Tree

We will use pre-order traverse to traverse all nodes and always memorize the previous number and compare to the current one, if current one is less than the previous one, we will return False. TC is O(n)

```
class Solution:
    def isValidBST(self, root: TreeNode) -> bool:
```

```
        self.prev = None
        if root == None:
            return True
        if self.traverse(root) == False:
            return False
        return True
    def traverse(self, node):
        if node.left:
            if self.traverse(node.left) == False:
                return False

        if self.prev == None:
            self.prev = node.val
        else:
            if self.prev >= node.val:
                return False
            self.prev = node.val

        if node.right:
            if self.traverse(node.right) == False:
                return False
```

621. Task Scheduler

We will use dict to count each task's number and get the maximum task and its number. Then we will divide it into two situations. One, There is idle, which means k is very large and our task cannot fill it. The length is gonna be (max_value - 1) * (n + 1) + max_count. Another situation is that we could fill each idle very easily. We will use len(tasks). So in the end, we will return maximum of the previous two. TC is O(n)

```
from collections import Counter
class Solution:
    def leastInterval(self, tasks: List[str], n: int) -> int:
        memo = list(Counter(tasks).values())

        max_value = max(memo)
        max_count = memo.count(max_value)
        return max(len(tasks), (max_value - 1) * (n + 1) + max_count)
```