1. Unique word ratio
   This is the question for us to practice set operation. We would transform two lists to sets, and the unique ratio is `len(list1_set & list2_set) / len(list1_set | list2_set). TC is O(n)`

```
def getUniqRatio(list1, list2):
   list1_set = set(list1)
   list2_set = set(list2)
   return len(list1_set & list2_set) / len(list1_set | list2_set)
```

3. Longest Substring Without Repeating Characters
We will use memo to store each element’s index, then every time we will check if this letter exists in memo and its index equal or larger than our start index, we will update our start index. Or will compare the current length from start to current index’s length,and store the maximum one. TC is O(n)

```
from collections import defaultdict
class Solution:
    def lengthOfLongestSubstring(self, s: str) -> int:
        memo = {}
        max_length = 0
        start = 0

        for i, l in enumerate(s):
            if l in memo and memo[l] >= start:
                start = memo[l] + 1
            else:
                max_length = max(max_length, i - start + 1)
            memo[l] = i
        return max_length
```

14. Longest Common Prefix
We will compare one by one until the end. Each time, we will set our result as the common prefix of the previous. So will get the longest common prefix. TC is O(nlogn + kn)

```
class Solution:
    def longestCommonPrefix(self, strs: List[str]) -> str:
        if not strs:
            return ''

        strs.sort()
        result = strs[0]
        for s in strs[1:]:
            if not s or not result:
                return ''
            length = min(len(result), len(s))
            for i in range(length):
                if result[i] != s[i]:
                    result = result[:i]
                    break
                if i == length - 1:
                    result = result[:length]
        return result
```

207. Course Schedule

We will use parents and children to store every course’s prerequisites and every course’s after courses. We will use nodes to store all nodes we will check in the end. We will check every parent which doesn’t have prerequisites. Then we will remove it in every child’s set. If child has not parent, we will remove its child. Repeat this process until we cannot. In the end, we will compare nodes length and nodes we remove. If they are equal, we will return True, or False. TC is O(n)

```
from collections import defaultdict
class Solution:
    def canFinish(self, numCourses: int, prerequisites: List[List[int]]) -> bool:
        parents = defaultdict(set)
        childs = defaultdict(set)
        nodes = set()
        count = 0

        for p in prerequisites:
            parents[p[0]].add(p[1])
            childs[p[1]].add(p[0])
            nodes.add(p[0])
            nodes.add(p[1])

        cur = nodes - set(parents.keys())
        count = len(cur)
        while len(cur) > 0:
            next_ite = set()
            for node in cur:
                for child_node in childs[node]:
                    parents[child_node].remove(node)
                    if len(parents[child_node]) == 0:
                        count += 1
                        next_ite.add(child_node)
            cur = next_ite
        return count == len(nodes)
```

528. Random Pick with Weight

We will accumulate every previous element to current element, so in the end will pick one element randomly from 1 to last element’s number, then return the binary search left index. TC is O(logn)

```
from bisect import *
from random import *
class Solution:

    def __init__(self, w: List[int]):
        self.w = w
        self.length = len(w)
        for i in range(1, len(w)):
            self.w[i] += self.w[i - 1]

    def pickIndex(self) -> int:
        num = choice(range(1, self.w[-1] + 1))
        return bisect_left(self.w, num)
```