355. Design Twitter
We will use a dict set to record all following relationships and a dict list to record each user's twitter with timestamp before it so that we could sort them. TC: postTweet: O(1), getNewsFeed(userId): O(mn), follow: O(1), unfollow: O(1)

```
from collections import defaultdict
class Twitter:

    def __init__(self):
        """
        Initialize your data structure here.
        """
        self.count = 0
        self.follows = defaultdict(set)
        self.tweets = defaultdict(list)

    def postTweet(self, userId: int, tweetId: int) -> None:
        """
        Compose a new tweet.
        """
        self.tweets[userId].append((self.count, tweetId))
        self.count += 1

    def getNewsFeed(self, userId: int) -> List[int]:
        """
        Retrieve the 10 most recent tweet ids in the user's news feed. Each item in the news feed
must be posted by users who the user followed or by the user herself. Tweets must be ordered
from most recent to least recent.
        """
        feeds = self.tweets[userId][:]
        for i in self.follows[userId]:
            feeds += self.tweets[i]
        feeds.sort(reverse=True)
        return list(map(lambda a: a[1], feeds))[:10]

    def follow(self, followerId: int, followeeId: int) -> None:
        """
        Follower follows a followee. If the operation is invalid, it should be a no-op.
        """
        if followeeId != followerId:
            self.follows[followerId].add(followeeId)

    def unfollow(self, followerId: int, followeeId: int) -> None:
        """
        Follower unfollows a followee. If the operation is invalid, it should be a no-op.
        """
        if followeeId in self.follows[followerId]:
            self.follows[followerId].remove(followeeId)
```

35. Search Insert Position
We will use binary search to get index that less or equal to target, TC is O(logn)

```
class Solution:
    def searchInsert(self, nums: List[int], target: int) -> int:
        l, r = 0, len(nums) - 1
        while l <= r:
            mid = (l + r) // 2
            if nums[mid] == target:
                return mid
            elif nums[mid] < target:
                l = mid + 1
            else:
                r = mid - 1
        return l
```

Also, we could remove first if statement and get the left insertion index and right insertion index:

```
def searchInsert(nums, target):
   l, r = 0, len(nums) - 1
   while l <= r:
       mid = (l + r) // 2
       if nums[mid] < target:
           l = mid + 1
       else:
           r = mid - 1
   return l
```

```
def searchInsert(nums, target):
   l, r = 0, len(nums) - 1
   while l <= r:
       mid = (l + r) // 2
       if nums[mid] <= target:
           l = mid + 1
       else:
           r = mid - 1
   return l
```

47. Permutations II

We will dfs to traverse all possible solutions. We will sort our list first and when its same element used previous, then this element could be used. TC is O(n * n)

```
import itertools
class Solution:
    def permuteUnique(self, nums: List[int]) -> List[List[int]]:
        nums.sort()
        self.result = []
        used = [False] * len(nums)
        self.dfs(used, nums, [])
        return self.result

    def dfs(self, used, nums, cur):
        if len(cur) == len(nums):
            self.result.append(cur[:])
            return
        for i in range(len(used)):
            if used[i]:
                continue
            if i > 0 and nums[i - 1] == nums[i] and not used[i - 1]:
                continue
            used[i] = True
            cur.append(nums[i])
            self.dfs(used, nums, cur)
            cur.pop()
            used[i] = False
```

```
from collections import defaultdict
def findLastPage(clicks, startPoint):
 children = defaultdict(list)
 visited = set()
 node = startPoint
 visited.add(node)
 for start, end in clicks:
   children[start].append(end)
 if not children[node]:
   return node
 cur = set([node])
 while cur:
   next_ite = set()
   for n in cur:
     if not children[n]:
       return n
     else:
       for child in children[n]:
         if child not in visited:
           next_ite.add(child)
           visited.add(child)
   cur = next_ite
 return False

print(findLastPage(arr, 'A'))


def searchInsert(nums, target):
   l, r = 0, len(nums) - 1
   while l <= r:
       mid = (l + r) // 2
       if nums[mid] < target:
           l = mid + 1
       else:
           r = mid - 1
   return l

print searchInsert([1,3,3,3,5,6], 3)
```

4. Random pair users
We will use randint to pick random index from users, and then we will replace the index with the lastIndex and return it. After each picking, we will deduct lastIndex by 1. TC is O(n)

```
from random import *
def matchUsers(users):
    lastIndex = len(users) - 1
    result = []
    for i in range(len(users) // 2):
        user1 = randomPickUser(users, lastIndex)
        lastIndex -= 1
        user2 = randomPickUser(users, lastIndex)
        lastIndex -= 1
        result.append((user1, user2))
    return result

def randomPickUser(user, lastIndex):
    idx = randint(0, lastIndex)
    users[idx], users[lastIndex] = users[lastIndex], users[idx]
    return users[lastIndex]['id']
users = [{'id': 'A'}, {'id': 'B'}, {'id': 'C'}, {'id': 'D'}]
```

Then we will pair uses from different groups, every time, we will pick users from group which has the largest number of users. And then pick another from other teams. TC is O(n)

```
from random import *
from collections import defaultdict
def matchUsers(users):
    memo = defaultdict(set)
    result = []
    for user in users:
        memo[user['team']].add(user['id'])
    while True:
        user_pair = randomPickUser(memo)
        if not user_pair:
            return result
        else:
            result.append(user_pair)


def randomPickUser(user_memo):
    if len(user_memo.keys()) < 2:
        return False
    v, k = max(map(lambda a: (len(a[1]), a[0]), user_memo.items()))
    user1 = pickOne(user_memo, k)
    user2_key = k
    keys = list(user_memo.keys())
    while user2_key == k:
        user2_key = choice(keys)
    user2 = pickOne(user_memo, user2_key)
    return (user1, user2)

def pickOne(user_memo, key):
    user = choice(list(user_memo[key]))
    user_memo[key].remove(user)
    if len(user_memo[key]) == 0:
        del user_memo[key]
    return user
```

5. waitList

We will use a linked list. When a new party comes in, we add it to the end of the linked list, when we want to move waiting list, we will find from first node and find the exact one of which val = capacity, then delete this node from our linked list and return its key. TC is O(n)

```
class Node:
    def __init__(self, key, val):
        self.val = val
        self.key = key
        self.next = None

class WaitingList:
    def __init__(self, parties):
        self.dummy = Node(0, 0)
        self.tail = self.dummy
        for party in parties:
            self.addpartyToWaitingList(party)

    def addpartyToWaitingList(self, party):
        node = Node(party['label'], party['number'])
        self.tail.next = node
        self.tail = node

    def moveWaitingList(self, capacity):
        node = self.dummy
```

```
while node.next and node.next.val != capacity:
    node = node.next
ret = node.next
if ret:
    node.next = ret.next
    if self.tail == ret:
        self.tail = node
    return ret.key
else:
    return None
```