1. Waitlist

   We will set our queue as a linked list. Each time we have new people, we will append it at the end of linked list. When there is an empty table, We will compare from the head, once we get a number less or equal to our empty table number, we will delete it from our linked list and return this number. TC is O(n)

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

class Solution:
    def __init__(self):
        self.dummy = Node(0)

    def addQueue(self, number):
        head = self.dummy
        while head.next:
            head = head.next
        head.next = Node(number)

    def emptyTable(self, number):
        head = self.dummy
        ret = None
        while head.next:
            if head.next.number <= number:
                ret = head.next.number
                head.next = head.next.next
                return ret
            else:
                head = head.next
```

159. Longest Substring with At Most Two Distinct Characters

We will use slide window to get the window where there are only two distinct words. Frist we will move right until count >= 3 or end. Then we will record the max length. Then we will move left bound until there are less than 3 distinct letters. Then we will move right to next index and repeat the previous steps. O(n)

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

        while right < length:
            while right < length:
                if memo[s[right]] == 0:
                    count += 1
                memo[s[right]] += 1
                if count == 3:
                    break
                right += 1

            max_length = max(max_length, right - left)
            while left < right:
                memo[s[left]] -= 1
                if memo[s[left]] == 0:
                    count -= 1
                    left += 1
                    break
                left += 1
            right += 1
        return max_length
```

252. Meeting Rooms

We only need to sort the intervals and compare adjacent intervals in the array. If the latter one's start is smaller than the previous one, then we should return False. In the end, we will return True. TC is O(nlogn)

```
class Solution:
    def canAttendMeetings(self, intervals: List[List[int]]) -> bool:
        if len(intervals) < 2:
          return True
        intervals.sort()
        cur = intervals[0]
        for i in range(1, len(intervals)):
          if not cur[1] <= intervals[i][0]:
            return False
          cur = intervals[i]
```

```
        return True
```

253. Meeting Room II

I will use add all start time to an array starts, and so does ends. We will sort each array, and compare the first one. If starts[0] < ends[0]: popleft starts, room += 1, else popleft ends, room -= 1. Each time we add room, we will compare it with our current room and keep the maximum. TC is O(nlogn)

```
class Solution:
    def minMeetingRooms(self, intervals: List[List[int]]) -> int:
      starts, ends = [], []
      count = 0
      max_room = 0
      for interval in intervals:
        starts.append(interval[0])
        ends.append(interval[1])
      starts.sort(reverse=True)
      ends.sort(reverse=True)
      while starts and ends:
        if starts[-1] < ends[-1]:
          count += 1
          starts.pop()
          max_room = max(max_room, count)
        else:
          count -= 1
          ends.pop()
      return max_room
```

208. Implement Trie

We could use dict to store letters. When it comes to the end of a word, we would add a mark '$', then when we search that word, we could differentiate between it's part of a word or the end of a word. In the end, we will return the result. TC is O(len(word)).

```
class Trie:

    def __init__(self):
        """
        Initialize your data structure here.
        """
        self.trie = {}

    def insert(self, word: str) -> None:
        """
        Inserts a word into the trie.
        """
```

```
        node = self.trie
        for w in word:
            if w not in node:
                node[w] = {}
            node = node[w]
        node['$'] = True


    def search(self, word: str) -> bool:
        """
        Returns if the word is in the trie.
        """
        node = self.trie
        for w in word:
            if w not in node:
                return False
            node = node[w]
        return '$' in node

    def startsWith(self, prefix: str) -> bool:
        """
        Returns if there is any word in the trie that starts with the given prefix.
        """
        node = self.trie
        for w in prefix:
            if w not in node:
                return False
            node = node[w]
        return True
```