1. WaitList
   We will use double linked list to record our wait list. When we want to add party to waiting list, We will append it to the end of our list, also we will update our tail. When we want to move waiting list, we will traverse from the first node and until we check current table's capacity is larger than party's number, we will remove node from linked list. We will return node's label. TC is moveWL: O(n), addToWL:O(1)

```
# waitlist
class Node:
   def __init__(self, key, value):
       self.key = key
       self.value = value
       self.next = None
       self.prev = None

class Waitlist:
   def __init__(self, parties):
       self.dummy = Node(0, 0)
       self.tail = None
       for party in parties:
           self.addPartyToWL(party["label"], party["number"])

   def moveWL(self, party_capacity):
       node = self.dummy
       while node.next and node.next.value > party_capacity:
           node = node.next
       temp = node.next
       if temp:
           node.next = temp.next
           node.next.prev = node
           if temp == self.tail:
               self.tail = node
           return temp.key
       return None

   def strictMoveWL(self, party_capacity):
       node = self.dummy
       while node.next and node.next.value != party_capacity:
```

```
            node = node.next
        temp = node.next
        if temp:
            node.next = temp.next
            return temp.key
        return None

    def addPartyToWL(self, key, value):
        if not self.tail:
            self.tail = Node(key, value)
            self.dummy.next = self.tail
            self.tail.prev = self.dummy
        else:
            self.tail.next = Node(key, value)
            self.tail.next.prev = self.tail
            self.tail = self.tail.next
```

2. Random Match Users
Every time, we will check whether rest users are in the same team, if they are, we will return None. The whole game is over. Then we will pick two users and check whether they are in the same team, if not, we will replace two people with last two people, and return these two people. Each time we get two users, we will deduct end_index by 2.

```
import random

def get_users(users, end_index):
    if len(set(map(lambda user: user["team"], users))) < 2:
        return None
    while True:
        idx1, idx2 = random.sample(range(0, end_index + 1), 2)
        if users[idx1]["team"] != users[idx2]["team"]:
            break
    users[idx1], users[end_index] = users[end_index], users[idx1]
    users[idx2], users[end_index - 1] = users[end_index - 1],
users[idx2]
    return (users[end_index]["id"], users[end_index - 1]["id"])

def match_beans(users):
    if not users:
```

```
        return []

    length = len(users)
    result = []
    end_index = length - 1
    pairs_number = length // 2

    for _ in range(pairs_number):
        temp = get_users(users, end_index)
        if temp:
            result.append(temp)
        else:
            break
        end_index -= 2
    return result
```

3. Love Message
We will iterate each message and use a hashmap to store each receiver’s message and sender to prevent spam. In the end, we will use heapq to get top k frequent receiver. TC is O(klogn)

```
from collections import defaultdict
from heapq import *
def getMaxiMessage(messages, k):
    memo = defaultdict(set)
    result = []
    h = []
    for message in messages:
        memo[message['receiver']].add((message['sender'],
message["content"]))

    for key, v in memo.items():
        heappush(h, (-len(v), key))
    for _ in range(k):
        result.append(heappop(h)[1])
    return result

messages = [
    {'receiver': 'Jack', 'sender': 'Mark', 'content': 'Hello'},
```

```
    {'receiver': 'Jack', 'sender': 'Tom', 'content': 'Hello'},
    {'receiver': 'Jack', 'sender': 'Ave', 'content': 'Hello'},
    {'receiver': 'Mark', 'sender': 'Ave', 'content': 'Hello'}
]

print(getMaxiMessage(messages, 1))
```

4. Find Prefix
We will iterate each string and split them by “ “, compare each word with prefix, once they are equal, we will append (idx, s) to result and break out of this iteration. We will return our list using map.

```
def findPrefix(strs, prefix):
    length = len(prefix)
    result = []

    for s in strs:
        for idx, word in enumerate(s.split(' ')):
            if word[:length].lower() == prefix:
                result.append((idx, s))
                break
    result.sort()
    return list(map(lambda k: k[1], result))
```

5. Find Prefix using Trie
We will trie to store every word and find words by prefix. We will use traverse by layer to traverse all words starting with the prefix. So that we could take advantage of this dictionary for the future use.

```
class Trie:
    def __init__(self):
        self.trie = {}

    def addWord(self, word, dest):
        node = self.trie
        for w in word:
            if w not in node:
```

```
                node[w] = {}
            node = node[w]
        node["WORD"] = dest

    def findByPrefix(self, prefix):
        node = self.trie
        for w in prefix:
            if w not in node:
                return []
            node = node[w]
        next_ite = []
        cur = [node]
        result = set()
        if "WORD" in node:
            result.add(node["WORD"])
        while cur:
            for my_node in cur:
                for key, value in my_node.items():
                    if key == "WORD":
                        result.add(value)
                    else:
                        next_ite.append(value)
            cur = next_ite
            next_ite = []
        return list(result)

def findPrefix2(strs, prefix):
    trie = Trie()
    for s in strs:
        trie.addWord(s.lower(), s)
    return trie.findByPrefix(prefix)
```

6. KMP

We use KMP to check whether string contains our match string. If it contains, we will append it to our result, else check the next one. TC is O(n * m)

```
def getLPS(match):
    d = 0
```

```python
    length = len(match)
    lps = [0] * length
    i = 1

    while i < length:
        if match[i] == match[d]:
            d += 1
            lps[i] = d
            i += 1
        else:
            if d > 0:
                d = lps[d - 1]
            else:
                lps[i] = 0
                i += 1
    return lps

def kmpMatch(dest_strs, match):
    result = []
    lps = getLPS(match)
    for dest_str in dest_strs:
        dest = dest_str.lower()
        i, m = 0, 0
        length = len(dest_str)
        while i < length:
            if dest[i] == match[m]:
                i += 1
                m += 1
                if m == len(match):
                    result.append(dest_str)
                    break
            else:
                if m > 0:
                    m = lps[m - 1]
                else:
                    i += 1
    return result
```