Q1 [Python] The probability of rain on a given calendar day in Vancouver is p[i], where i is the day's index. For example, p[0] is the probability of rain on January 1 , and p[10] is the probability of precipitation on January 11 . Assume the year has 365 days (i.e. p has 365 elements). What is the chance it rains more than n (e.g. 100) days in Vancouver? Write a function that accepts p (probabilities of rain on a given calendar day) and n as input arguments and returns the possibility of raining at least n days.

ANSWER: if we want 365>=n>= 100 days to be rainy, we have to choose n days from the 365 days to be rainy. The other (365 -n) days are going to be not rainy or (1-p[i]). We then multiply all these probabilities together. We repeat this step for all the values of 365>=n>= 100 and all the combinations of choosing n from 365 and add them all together:

```python
from itertools import combinations
from typing import Sequence

def prob_rain_more_than_n(p: Sequence[float], n: int) -> float:
    # Number of days in a year
    num_days = len(p)

    # Initialize the probability to 0
    prob = 0

    # find all the combinations of 100+ days
    for comb in combinations(range(num_days), n):
        # Calculate the probability of this combination of days with rain(we use * because it's an AND case)
        prob_rain = 1
        for i in range(num_days):
            if i in comb: # if i is one of the 100+ days that rains
                prob_rain *= p[i]
            else:
                prob_rain *= (1 - p[i]) # it does not rain on that day

        # Add the probability of this combination to the total probability(because it's an OR case)
        prob += prob_rain

    return prob
```

The above method is going to be VERY slow and impractical so another method that I used was to get the average of the probability of rain for all 365 probabilities to fix p. Then I used binomial distribution:

```python
from scipy.stats import binom
from typing import Sequence
import random

def prob_rain_more_than_n(p: Sequence[float], n: int) -> float:
    # Calculate the average probability of rain
    avg_p = sum(p) / len(p)
    print(avg_p)
    # Calculate the probability of having at least n days of rain
    prob = 1 - binom.cdf(n - 1, 365, avg_p) # 1 - (probability of rain in 0<=n<=99 days)

    return prob
```

Q2 [Python] A phoneme is a sound unit (similar to a character for text). We have an extensive pronunciation dictionary (think millions of words). Below is a snippet:

ABACUS AE B AH K AH S

BOOK B UH K

THEIR DH EH R

THERE DH EH R

TOMATO T AH M AA T OW

TOMATO T AH M EY T OW

Given a sequence of phonemes as input (e.g. ["DH", "EH", "R", "DH", "EH", "R"]), find all the combinations of the words that can produce this sequence (e.g. [["THEIR", "THEIR"], ["THEIR", "THERE"], ["THERE", "THEIR"], ["THERE", "THERE"]]). You can preprocess the dictionary into a different data structure if needed.

ANSWER: Since in the input list we only have access to the phonemes, I made a library that we can access the corresponding words for every phenom:

```python
from itertools import product
def preprocess_pronunciation_dict(pronunciation_dict):
    phoneme_to_words = {}
    for word, phonemes in pronunciation_dict:
        phoneme_key = tuple(phonemes)
        if phoneme_key not in phoneme_to_words:
            phoneme_to_words[phoneme_key] = []
        phoneme_to_words[phoneme_key].append(word)
    return phoneme_to_words
```

Then I will have a phoneme window that slides through all the phonemes as its start point and it considers all the phonemes that come after it. We then extract a tuple representing the current phoneme window from the input list and check if the phonemes exist in our dictionary, if yes, we add it our list of all_combos and start from the end of the current phonemes. Since some phonemes can refer to multiple words and we

```python
def  find_word_combos_with_pronunciation(phonemes_input):
    start = 0
    all_combos = []
    while start < len(phonemes_input):
        found_word = False
        for j in range(start + 1, len(phonemes_input) + 1):
            curr_phoneme = tuple(phonemes_input[start:j]) # window of current phenomes
            if curr_phoneme in phoneme_to_words:
                all_combos.append(phoneme_to_words[curr_phoneme])
                start = j # start from the end of current phenomes
                found_word = True
                break
        if not found_word:
            start += 1
    return list(product(*all_combos)) # some phenomes can refer to multiple words and we need all combonations of them
```

need all combinations of them.

Q3 [C] Find the n most frequent words in the TensorFlow Shakespeare dataset.

ANSWER: For every buffer word that we read from the file, we call a function named clean_buffer. This function converts the word to lowercase using the tolower function and removes punctuation characters from the word. Then we add the word to word_table which stores information about distinct words and their counts. Then we sort the word_table array in descending order of occurrence counts using the bubble sort algorithm. The most frequent words (top n words) are extracted from the sorted word_table array and stored in a dynamically allocated 2D array of characters named result.