Problem 1: In these exercises, we are going to be processing some natural linguistic data, the first paragraph of Moby Dick. We will first write some procedures that help us to manipulate this corpus. We will then start analyzing this data using some probabilistic models.

We will start by defining the variable `moby-word-tokens`, which is a list containing all of the words from our corpus.

```
(def moby-word-tokens '(CALL me Ishmael . Some years ago never mind
     how long precisely having little or no money in my purse , and
     nothing particular to interest me on shore , I thought I would
     sail about a little and see the watery part of the world .  It is
     a way I have of driving off the spleen , and regulating the
     circulation . Whenever I find myself growing grim about the mouth
     whenever it is a damp , drizzly November in my soul whenever I
     find myself involuntarily pausing before coffin warehouses , and
     bringing up the rear of every funeral I meet and especially
     whenever my hypos get such an upper hand of me , that it requires
     a strong moral principle to prevent me from deliberately stepping
     into the street , and methodically knocking people's hats off
     then , I account it high time to get to sea as soon as I can .
     This is my substitute for pistol and ball . With a philosophical
     flourish Cato throws himself upon his sword I quietly take to the
     ship .  There is nothing surprising in this . If they but knew it
     , almost all men in their degree , some time or other , cherish
     very nearly the same feelings toward the ocean with me .))
```

Below we have defined the function `member-of-list`, which takes two arguments, `w` and `l`. The function returns `true` if the element `w` is a member of the list `l`, and `false` otherwise. For example, if `w` equals `'the` and `l` equals `(list 'the 'man 'is)`, then the function will return `true`. In contrast, if `w` equals `'the` and `l` equals `(list 'man 'is)`, then the function will return false.

```
(defn member-of-list? [w l]
  (if (empty? l)
    false
    (if (= w (first l))
      true
      (member-of-list? w (rest l)))))
```

Below we have defined the skeleton for the function `get-vocabulary`. This function takes two arguments, `word-tokens` and `vocab`. `word-tokens` is a list of words, and the function should return the list of unique words occuring in word-tokens. This list of unique words is called a vocabulary. For example, if word-tokens is equal to `(list 'the 'man 'is 'man 'the)`, then get-vocabulary should return (list 'the 'man 'is).

Fill in the missing parts of this function. When you call `(get-vocabulary moby-word-tokens`

`'())`, you will get back a list of all of the unique words occuring in moby-word-tokens. Give this the name `moby-vocab`.

```
;;(defn get-vocabulary [word-tokens vocab]
;;  (if (empty? word-tokens)
;;    vocab
;;    (if (member-of-list? ;;finish this line
;;      (get-vocabulary ;;finish this line
;;      (get-vocabulary ;;finish this line
```

Answer 1:

```
(defn get-vocabulary [word-tokens vocab]
  (if (empty? word-tokens)
    vocab
    (if (member-of-list? (first word-tokens) vocab)
      (get-vocabulary (rest word-tokens) vocab)
      (get-vocabulary (rest word-tokens)
        (cons (first word-tokens) vocab)))))
(def moby-vocab (get-vocabulary moby-word-tokens '()))
```

Problem 2: Define a function `get-count-of-word`. This function should take three arguments, `w`, `word-tokens`, and `count`. `w` is a word and `word-tokens` is a list of words. When you call `(get-count-of-word w word-tokens 0)`, the function should return the number of occurrences of the word `w` in the list `word-tokens`. For example, if word-tokens equals `(list 'the 'the 'man)`, and `w` equals `'the`, then the function should return 2. If `word-tokens` is the same, but `w` equals `'man`, then the function should return `1`.

```
;;(defn get-count-of-word [w word-tokens count]
  ;;fill this in
```

Below we have defined the function `get-word-counts`, which takes two arguments, `vocab` and `word-tokens`. `vocab` is assumed to be a list of the unique words that occur in the list `word-tokens`. The function returns the number of times each word in vocab occurs in word-tokens. For example, suppose `vocab` equals `(list 'man 'the 'is)`, and `word-tokens` equals `(list 'the 'man 'is 'is)`. Then the function would return `(list 1 1 2)`, corresponding to the number of times `'man`, `'the`, and `'is` occur in `word-tokens`.

```
(defn get-word-counts [vocab word-tokens]
  (let [count-word
    (fn [w] (get-count-of-word w word-tokens 0))]
    (map count-word vocab)))
```

Answer 2:

```
(defn get-count-of-word [w word-tokens count]
  (if (empty? word-tokens)
    count
    (if (= w (first word-tokens))
      (get-count-of-word w (rest word-tokens) (+ count 1))
      (get-count-of-word w (rest word-tokens) count))))
```

Problem 3: Use the function `get-word-counts`, and the other variables we have defined, to define a variable `moby-word-frequencies`. This variable should contain the number of times each word in `moby-vocab` occurs in `moby-word-tokens`.

In class we defined the functions `normalize`, `flip`, and `sample-categorical`. These functions are very useful for us, and are included below.

```
(defn flip [p]
  (if (< (rand 1) p)
    true
    false))

(defn normalize [params]
  (let [sum (apply + params)]
    (map (fn [x] (/ x sum)) params)))

(defn sample-categorical [outcomes params]
  (if (flip (first params))
    (first outcomes)
    (sample-categorical (rest outcomes)
      (normalize (rest params)))))
```

Let's define a function that returns a particular probability distribution, the uniform distribution. The uniform distribution is the distribution which assigns equal probability to every possible outcome.

The function `uniform-distribution` takes a single argument, `outcomes`, which is a list of length n. The function should return a list containing the number `1/n` repeated n times. For example, if outcomes equals `(list 'the 'a 'every)`, then this function will return `(list 1/3 1/3 1/3)`. This list can be interpreted as a probability distribution over the outcomes, which assigns equal probability to each of them.

```
(defn create-uniform-distribution [outcomes]
  (let [num-outcomes (count outcomes)]
    (map (fn [x] (/ 1 num-outcomes))
      outcomes)))
```

Answer 3:

```
(def moby-word-frequencies (get-word-counts moby-vocab moby-word-tokens))
```

Problem 4: Using `create-uniform-distribution` and `sample-categorical`, write a function `sample-uniform-BOW-sentence` that takes a number `n` and a list `vocab`, and returns a sentence of length `n`. Each word in the sentence should be generated independently from the uniform distribution over vocab. For example, given `n` equal to `4` and `vocab` equal to `(list 'the 'a 'every)`, a possible return value for this function is `(list 'a 'the 'the 'a)`.

Note that this is a bag of words model, as defined in class. That is, we assume every element of the list is generated independently. We will call this the uniform bag of words model.

Answer 4: https://foundations-computational-linguistics.github.io/chapters/11-BagOfWords.html

```
(defn list-unfold [generator n]
  (if (= n 0)
    '()
    (cons (generator)
      (list-unfold generator (- n 1)))))
(defn sample-uniform-BOW-sentence [n vocab]
  (list-unfold
    (fn [] (sample-categorical vocab
      (create-uniform-distribution vocab))) n))
```

Problem 5: Define a function `compute-uniform-BOW-prob`, which takes two arguments, `vocab` and `sentence`. `vocab` is the list of all words in the vocabulary, and sentence is a list of observed words. The function should return the probability of the sentence according to the uniform bag of words model.

For example, if `vocab` equals `(list 'the 'a 'every)`, and sentence equals `(list 'every 'every)`, then the function should return $\frac{1}{9}$.

Answer 5: https://rosettacode.org/wiki/Exponentiation_operator

```
(defn ** [x n] (reduce * (repeat n x)))
(defn compute-uniform-BOW-prob [vocab sentence]
  (** (first (create-uniform-distribution vocab))
```

```
    (count sentence)))
```

Problem 6: Using `sample-uniform-BOW-sentence` and `moby-vocab`, sample a 3-word sentence from the vocabulary of our Moby Dick corpus. This will be a sample from the uniform bag of words model for this vocabulary. Repeat this process a number of times. For each of these 3-word sentences, use `compute-uniform-BOW-prob` to compute the probability of the sentence according to the uniform bag of words model. What do you notice? Why is this true?

Answer 6:

```
(def moby-sentence (sample-uniform-BOW-sentence 3 moby-vocab))
(compute-uniform-BOW-prob moby-vocab moby-sentence)
(warehouses deliberately myself)
1/2744000
(flourish purse strong)
1/2744000
(part upper off)
1/2744000
```

The probability of the sentence is always $\frac{1}{2744000}$. This is because we are using a uniform Bag of Words Model, and each word in the vocab has the same likelihood of appearing.

Problem 7: In class we looked at a more general version of the bag of words model, in which different words in the vocabulary can be assigned different probabilities. We defined a function `sample-BOW-sentence`, which returns a sentence sampled from the bag of words model that we have specified. Below we have included a slight variant of the function which we defined in class. Previously the variables vocabulary and probabilities were defined outside of the function. In the current version, they are passed in as arguments. The function is identical otherwise.

```
(defn sample-BOW-sentence [len vocabulary probabilities]
  (if (= len 0)
    '()
    (cons (sample-categorical vocabulary probabilities)
    (sample-BOW-sentence (- len 1) vocabulary probabilities))))
```

The function `sample-BOW-sentence` allows us to sample a sentence given arbitrary probabilities for the words in our vocabulary. Let's make use of this power and define a distribution over the vocabulary which is better than the uniform distribution. We will use the word frequencies for our Moby Dick corpus to *estimate* a better distribution.

Above we defined the variable `moby-word-frequencies`, which contains the frequency of every word that occurs in our Moby Dick corpus. Using `normalize` and `moby-word-frequencies`, define a variable `moby-word-probabilities`. This variable should contain probabilities for every word in `moby-vocab`, in proportion to its frequency in the text. A word which occurs 2 times should receive twice as much probability as a word which occurs 1 time.

Answer 7:

```
(def moby-word-probabilities (normalize moby-word-frequencies))
```

Problem 8: Using `sample-BOW-sentence`, sample a 3-word sentence from a bag of words model, in which the probabilities are set to be those in `moby-word-probabilities`. Repeat this process a number of times, and write down the sentences that you collect through this process.

Answer 8:

```
(sample-BOW-sentence 3 moby-vocab moby-word-probabilities)
(CALL degree whenever)
(to with account)
(interest other nothing)
```

Problem 9: Define a function `lookup-probability`, which takes three arguments, `w`, `outcomes`, and `probs`. `probs` represents a probability distribution over the elements of `outcomes`. For example, if outcomes is `(list 'the 'a 'every)`, then `probs` may be equal to `(list 0.2 0.5 0.3)`. The first number in `probs` is the probability of the first element of outcomes, the second number in probs is the probability of the second element of outcomes, and so on.

`lookup-probability` should look up the probability of the element `w`. For example, if `w` equals `'the`, then look-up probability should return `0.2`. If `w` equals `'a`, then `lookup-probability` should return `0.5`.

Below we have defined the function `product`. This function takes a list of numbers as its argument, and returns the product of these numbers.

```
(defn product [l]
  (apply * l))
```

Answer 9:

```
(defn lookup-probability [w outcomes probs]
  (if (= (first outcomes) w)
   (first probs)
   (lookup-probability w (rest outcomes) (rest probs))))
```

Problem 10: Using `lookup-probability` and `product`, define a function `compute-BOW-prob` which takes three arguments, `sentence`, `vocabulary`, and `probabilities`. The arguments `vocabulary` and `probabilities` are used to define a bag of words model with the associated probability distribution over vocabulary words. The function should compute the probability of the sentence (which is a list of words) according to the bag of words model.

This function is a generalization of the function `compute-uniform-BOW-prob` that you defined above.

Answer 10:

```
(defn get-prob [sentence vocabulary probabilities]
  (if (empty? sentence)
    '()
    (cons (lookup-probability (first sentence) vocabulary probabilities)
    (get-prob (rest sentence) vocabulary probabilities))))
(defn compute-BOW-prob [sentence vocabulary probabilities]
  (product (get-prob sentence vocabulary probabilities)))
```

Problem 11: In problem 8, you collected a number of 3-word sentences. These sentences were generated from a bag of words model in which the probabilities were set to those in `moby-word-probabilities`, which reflect the relative frequency of the words in the Moby Dick corpus. Use `compute-BOW-prob` to compute the probability of these sentences according to the bag of words model. How does your answer differ from problem 6?

Choose one of the 3-word sentences that you have generated. Can you construct a different sentence which has the same probability according to the bag of words model? When computing the probability of a sentence under a bag of words model, what information about the sentence suffices to compute this probability?

Answer 11:

```
(compute-BOW-prob moby-sentence moby-vocab moby-word-probabilities)
(CALL degree whenever)
3/9129329
(to with account)
5/9129329
(interest other nothing)
2/9129329
```

I will use $C(list)$ to denote the count or the size of a list. The probability that `(to with account)` is the given sentence is $\frac{5}{9129329}$. The sentence `(grim to way)` also has a probability of $\frac{5}{9129329}$. In q6, we assumed that each word had the same frequency. Therefore the denominator was $(C(vocab))^3 = 140^3 = 2744000$. In q11, we took the frequencies of the words into consideration, so $(C(tokens))^3 = 209^3 = 9129329$. As stated in this paragraph, the frequencies of the words are necessary for the probability of a sentence under a bag of words model.