homework 3, version 7

Homework 3: *Structure and Language*

18.S191 , Fall 2023

This notebook contains *built-in, live answer checks*! In some exercises you will see a coloured box, which runs a test case on your code, and provides feedback based on the result. Simply edit the code, run it, and the check runs again.

Feel free to ask questions!

Initializing packages

When running this notebook for the first time, this could take up to 15 minutes. Hang in there!

```
begin
    using Colors
    using PlutoUI
    using Compose
    using LinearAlgebra
end
```

Introduction

So far in the class the **data** that we have been dealing with has mainly been in the form of images. But, of course, we know that data comes in many other forms too, as we briefly discussed in the first lecture.

In this homework we will look at another very important data source: **written text** in **natural language**. (The word "natural" here is to distinguish human (natural) languages from computer languages.)

We will both analyse actual text and try to generate random text that looks like natural language. Both the analysis and synthesis of natural language are key components of artificial intelligence and are the subject of much current research.

Exercise 1: *Language detection*

In this exercise we will create a very simple *Artificial Intelligence*. Natural language can be quite messy, but hidden in this mess is *structure*, which we will look for today.

Let's start with some obvious structure in English text: the set of characters that we write the language in. If we generate random text by sampling (choosing) random characters (`Char` in Julia), it does not look like English:

```
['\Ue881f', '\U7426f', '\U9721c', '\U14fe0', '듽']
```

```
rand(Char, 5)   # sample 5 random characters
```

```
"\U34472\U8b675\Ubbd68\U7561b\Ue7c85\Ue5f73\U474b7\U4106b\U60106\U60f12\U4815a\U63038\U6c31
```

```
String(rand(Char, 40))   # join into a string
```

[`Char` in Julia is the type for a Unicode character, which includes characters like 日 and ☃.]

Instead, let's define an *alphabet*, and only use those letters to sample from. To keep things simple we will ignore punctuation, capitalization, etc., and use only the following 27 characters (English letters plus "space"):

```
alphabet =
  ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o', 'p', 'q', 'r', 's'
```

```
alphabet = ['a':'z' ; ' ']   # includes the space character
```

Let's sample random characters from our alphabet:

```
kreghqqjprbgnfgtwkvbanbgjz bqwofpaxdqocs
```

```
Text(String(rand(alphabet, 40)))
```

That already looks a lot better than our first attempt! But it still does not look like English text – we can do better.

Frequency table

English words are not well modelled by this random-Latin-characters model. Our first observation is that **some letters are more common than others**. To put this observation into practice, we would like to have the **frequency table** of the Latin alphabet. We could search for it online, but it is actually very simple to calculate ourselves! The only thing we need is a *representative sample* of English text.

The following samples are from Wikipedia, but feel free to type in your own sample! You can also enter a sample of a different language, if that language can be expressed in the Latin alphabet.

Remember that the 👁 button on the left of a cell will show or hide the code.

We also include a sample of Spanish, which we'll use later!

```
samples =
  (English = "Although the word forest is commonly used, there is no universally recognised
```

> Although the word forest is commonly used, there is no universally recognised precise definition, with more than 800 definitions of forest used around the world.[4] Although a forest is usually defined by the presence of trees, under many definitions an area completely lacking trees may still be considered a forest if it grew trees in the past, will grow trees in the future,[9] or was legally designated as a forest regardless of vegetation type.[10][11] The word forest derives from the Old French forest (also forès), denoting "forest, vast expanse covered by trees"; forest was first introduced into English as the word denoting wild land set aside for hunting[14] without the necessity in definition of having trees on the land.[15] Possibly a borrowing, probably via Frankish or Old High German, of the Medieval Latin foresta, denoting "open wood", Carolingian scribes first used foresta in the Capitularies of Charlemagne specifically to denote the royal hunting grounds of the King. The word was not endemic to Romance languages, e. g. native words for forest in the Romance languages derived from the Latin silva, which denoted "forest" and "wood(land)" (confer the English sylva and sylvan); confer the Italian, Spanish, and Portuguese selva; the Romanian silvă; and the Old French selve, and cognates in Romance languages, e. g. the Italian foresta, Spanish and Portuguese floresta, etc., are all ultimately derivations of the French word.

Exercise 1.1 - *Data cleaning*

Looking at the sample, we see that it is quite *messy*: it contains punctuation, accented letters and numbers. For our analysis, we are only interested in our 27-character alphabet (i.e. `'a'` through `'z'` plus `' '`). We are going to clean the data using the Julia function `filter`.

```
[7, 9, -5]
```

```
filter(isodd, [6, 7, 8, 9, -5])
```

`filter` takes two arguments: a **function** and a **collection**. The function is applied to each element of the collection, and it must return either `true` or `false` for each element. [Such a function is often called a **predicate**.] If the result is `true`, then that element is included in the final collection.

Did you notice something cool? Functions are also just *objects* in Julia, and you can use them as arguments to other functions! *(Fons thinks this is super cool.)*

We have written a function `isinalphabet`, which takes a character and returns a boolean:

```
isinalphabet (generic function with 1 method)
```

```
(true, false)
```

```
isinalphabet('a'), isinalphabet('+')
```

👉 Use `filter` to extract just the characters from our alphabet out of `messy_sentence_1`:

```
messy_sentence_1 = "#wow 2020 ¥500 (blingbling!)"
```

```
messy_sentence_1 = "#wow 2020 ¥500 (blingbling!)"
```

```
cleaned_sentence_1 = "wow   blingbling"
```

```
1 cleaned_sentence_1 = filter(isinalphabet,messy_sentence_1)
```

> **Got it!**
>
> Great!

We are not interested in the capitalization of letters (i.e. `'A'` vs `'a'`), so we want to *map* these to lower case *before* we apply our filter. If we don't, all upper case letters would get deleted.

Julia has a `map` function to do exactly this. Like `filter`, its first argument is the function we want to map over the vector in the second argument.

👉 Use the function `lowercase` to convert `messy_sentence_2` into a lower case string, and then use `filter` to extract only the characters from our alphabet.

```
messy_sentence_2 = "Awesome! 😍"
```

```
1 messy_sentence_2 = "Awesome! 😍"
```

```
cleaned_sentence_2 = "awesome "
```

```
1 cleaned_sentence_2 = filter(isinalphabet,lowercase(messy_sentence_2))
```

> **Got it!**
>
> Great!

Finally, we need to deal with **accents**: simply deleting accented characters from the source text might deform it too much. We can add accented letters to our alphabet, but a simpler solution is to *replace* accented letters with the corresponding unaccented base character. We have written a function `unaccent` that does just that.

```
french_word = "Égalité!"
```

```
1 french_word = "Égalité!"
```

```
"Egalite!"
```

```
1 unaccent(french_word)
```

`unaccent`

Turn `"áéíóúüñ asdf"` into `"aeiouun asdf"`.

👉 Let's put everything together. Write a function `clean` that takes a string, and returns a *cleaned* version, where:

- accented letters are replaced by their base characters;
- upper-case letters are converted to lower case;
- it is filtered to only contain characters from `alphabet`

```
clean (generic function with 1 method)
```

```
function clean(text)
    text=lowercase(text)
    text=unaccent(text)
    return filter(isinalphabet,text)
end
```

```
"creme brulee est mon plat prefere"
```

```
clean("Crème brûlée est mon plat préféré.")
```

> **Got it!**
>
> Great! 🥕

Exercise 1.2 - *Letter frequencies*

We are going to count the *frequency* of each letter in this sample, after applying your `clean` function. Can you guess which character is most frequent?

```
first_sample =
"although the word forest is commonly used there is no universally recognised precise defin
```

```
first_sample = clean(first(samples))
```

```
letter_frequencies (generic function with 1 method)
```

```
['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o', 'p', 'q', 'r', 's'
```

```
alphabet
```

```
sample_freqs =
[0.0630892, 0.00580131, 0.0203046, 0.0406091, 0.106599, 0.0319072, 0.0261059, 0.0340827, 0.0
```

```
sample_freqs = letter_frequencies(first_sample)
```

The result is a 27-element array, with values between `0.0` and `1.0`. These values correspond to the *frequency* of each letter.

`sample_freqs[i] == 0.0` means that the ith letter did not occur in your sample, and
`sample_freqs[i] == 0.1` means that 10% of the letters in the sample are the ith letter.

To make it easier to convert between a character from the alphabet and its index, we have the following function:

```
index_of_letter (generic function with 1 method)
```

```
(1, 2, 27)
```

```
1 index_of_letter('a'), index_of_letter('b'), index_of_letter(' ')
```

```
indices =  [10, 17, 26]
```

```
1 indices=[i for i in 1:27 if sample_freqs[i]==0]
```

👉 Which letters from the alphabet did not occur in the sample?

```
unused_letters =  ['j', 'q', 'z']
```

```
1 unused_letters = alphabet[indices]# replace with your answer
```

Got it!

Well done!

Hint

[illegible]

Now that we know the frequencies of letters in English, we can generate random text that already looks closer to English!

Random letters from the alphabet:

> vvxcvweulqnjnyjmktenswjxtedfsftesypo
> qwbbiqfamxtmyqdlhhxvzbdpamuueegpdkwanvtiibenuo ifasvjldlgnquyrdz

> ysnqlgjkzlqkwlsrhsrpypcalnplze lxwdhnfnuxszfpxmxdejruheegr
> sgifceaehebydjjxhjqozeizgwnsvjjijjkdqhyijrjltklkpolfrokdgzegyceakgaylduplkjrkwy jjk
> vpjkzbgzknfnpszkwfmsxpbtwgllhmwdtfqrvonirkfamajihkhjmbbtdejqti
> arnebhyhxrwtgzjpbipzozpuoirnvuqzeqhjxjtqkwtxqaupqh
> ljklitpkahvgsoihpkgwixhkzoitvucqiejivuh x

Random letters at the correct frequencies:

> n iooreaea hnans tonhoigayvevt srrboggarm sy rno nin d iee rgs ooreioidnieph tnn paiwslerx
> sna n ftrurr sn gelgheik erherfuwrf go nhfinrenneual adnsa e ewrrdl stwl winocolenna
> nonmavrinhuinonsstts lituecnihs erwrtsh ilrtscigtdeatui efsie g csct fcvlirsrc l o e e ehty
> windleynr e rgnetcaenlmhhperftc taovhprtddhasnlr s ninra sisodeel nnolsyedarhety y
> ehitdeihnaoenosiefiahe neeo

By considering the *frequencies* of letters in English, we see that our model is already a lot better!

Our next observation is that **some letter *combinations* are more common than others**. Our current model thinks that `potato` is just as 'English' as `ooaptt`. In the next section, we will quantify these *transition frequencies*, and use it to improve our model.

```
rand_sample (generic function with 1 method)
```

```
rand_sample_letter (generic function with 1 method)
```

Exercise 1.3 - *Transition frequencies*

In the previous exercise we computed the frequency of each letter in the sample by *counting* their occurences, and then dividing by the total number of counts.

In this exercise, we are going to count *letter transitions*, such as `aa`, `as`, `rt`, `yy`. Two letters might both be common, like `a` and `e`, but their combination, `ae`, is uncommon in English.

To quantify this observation, we will do the same as in our last exercise: we count occurences in a *sample text*, to create the **transition frequency matrix**.

```
transition_counts (generic function with 1 method)
```

```
function transition_counts(cleaned_sample)
    [count(string(a, b), cleaned_sample)
        for a in alphabet,
            b in alphabet]
end
```

```
normalize_array (generic function with 1 method)
```

```
normalize_array(x) = x ./ sum(x)
```

```
transition_frequencies = normalize_array ∘ transition_counts;
```

```
27×27 Matrix{Float64}:
 0.0          0.000725689  0.000725689  0.0          …  0.000725689  0.0  0.0101597
 0.000725689  0.0          0.0          0.0             0.00145138   0.0  0.0
 0.00217707   0.0          0.0          0.0             0.0          0.0  0.00145138
 0.0          0.0          0.0          0.0             0.0          0.0  0.0232221
 0.000725689  0.0          0.00290276   0.00870827      0.0          0.0  0.0312046
 0.0          0.0          0.0          0.0          …  0.0          0.0  0.0065312
 0.00145138   0.0          0.0          0.0             0.0          0.0  0.00798258
 ⋮                                                   ⋱               ⋮
 0.00580552   0.0          0.0          0.0             0.0          0.0  0.0
 0.00217707   0.0          0.0          0.0             0.0          0.0  0.00145138
 0.0          0.0          0.0          0.0             0.0          0.0  0.0
 0.000725689  0.0          0.0          0.0             0.0          0.0  0.0101597
 0.0          0.0          0.0          0.0          …  0.0          0.0  0.0
 0.0145138    0.00290276   0.00725689   0.0101597       0.0          0.0  0.000725689
```

```
transition_frequencies(first_sample)
```

What we get is a **27 by 27 matrix**. Each entry corresponds to a character pair. The *row* corresponds to the first character, the *column* is the second character. Let's visualize this:


```
show_pair_frequencies(transition_frequencies(first_sample))
```

The brightness of each letter pair indicates how frequent that pair is; here space is indicated as `_`.

Answer the following questions with respect to the **cleaned English sample text**, which we called `first_sample`. Let's also give the transition matrix a name:

```
sample_freq_matrix = transition_frequencies(first_sample);
```

👉 What is the frequency of the combination `"th"`?

```
th_frequency = 0.02104499274310595
```

```
th_frequency = transition_frequencies(first_sample)[20,8]
```

👉 What about `"ht"`?

```
ht_frequency = 0.0
```

```
ht_frequency = transition_frequencies(first_sample)[8,20]
```

> **Got it!**
>
> Great! 🎉

👉 Write code to find which le**tt**ers appeared doubled in our sample.

```
double_letters =  ['e', 'l', 'm', 'o', 'r', 's', ' ']
```

```
double_letters = vec([alphabet[i] for i in 1:27 if
transition_frequencies(first_sample)[i,i] != 0])
```

Got it!

Great!

```
if !@isdefined(double_letters)
    not_defined(:double_letters)
else
    let
        result = double_letters
        if result isa Missing
            still_missing()
        elseif result isa String
            keep_working(md"Use `collect` to turn a string into an array of characters.")
        elseif !(result isa AbstractVector{Char} || result isa AbstractSet{Char})
            keep_working(md"Make sure that `double_letters` is a `Vector`.")
        elseif Set(result) == Set(alphabet[diag(sample_freq_matrix) .!= 0])
            correct()
        elseif push!(Set(result), ' ') == Set(alphabet[diag(sample_freq_matrix) .!= 0])
            almost(md"We also consider the space (`' '`) as one of the letters in our `alphabet`.")
        else
            keep_working()
        end
    end
end
```

Hint

First answer this question by looking at the pair frequency image

👉 Which letter is most likely to follow a **W**?

You are free to do this partially by hand, partially using code, whatever is easiest!

```
most_likely_to_follow_w = 'o': ASCII/Unicode U+006F (category Ll: Letter, lowercase)
```

```
most_likely_to_follow_w = alphabet[argmax([transition_frequencies(first_sample)[23,i] for i in 1:27])]
```

Got it!

You got the right answer!

👉 Which letter is most likely to precede a **W**?

You are free to do this partially by hand, partially using code, whatever is easiest!

most_likely_to_precede_w = ' ': ASCII/Unicode U+0020 (category Zs: Separator, space)

```
most_likely_to_precede_w =  alphabet[argmax([transition_frequencies(first_sample)
[i,23] for i in 1:27])]
```

> Got it!
>
> Splendid!

👉 What is the sum of each row? What is the sum of each column? What is the sum of the matrix? How can we interpret these values?"

row_sums =

```
[0.0624093, 0.00580552, 0.0203193, 0.0406386, 0.106676, 0.0319303, 0.0261248, 0.0341074, 0.0
```

```
row_sums= vec(sum(transition_frequencies(first_sample), dims=1))
```

col_sums =

```
[0.063135, 0.00580552, 0.0203193, 0.0406386, 0.106676, 0.0319303, 0.0261248, 0.0341074, 0.06
```

```
col_sums = vec(sum(transition_frequencies(first_sample), dims=2))
```

row_col_answer =

Blablabla

```
row_col_answer = md"""

Blablabla
"""
```

We can use the measured transition frequencies to generate text in a way that it has **the same transition frequencies** as our original sample. Our generated text is starting to look like real language!

Error message from Main

```
Failed to show value:
MethodError: no method matching length(::Nothing)
The function `length` exists, but no method is defined for this combination of
argument types.

Closest candidates are:
  length(::Cmd)
   @ Base process.jl:716
  length(::Base.EnvDict)
   @ Base env.jl:232
  length(::Base.AsyncGenerator)
   @ Base asyncmap.jl:390
  ...
```

Show stack trace...

```
1 @bind ex23_sample Select([v => String(k) for (k, v) in
  zip(fieldnames(typeof(samples)), samples)])
```

Random letters from the alphabet:

jogfyni snpohqsmgodbvkwrmawkkhdpexxoxnppjhkdaqrahquysihzcupcifbtvxnlttla ulsvt af wgqjbncrtphxlqbbnfdcjwht ynanvvzetixghukrcshcjjlcowevxyuhurtl agvrinjaq hymblilpjmxjnoiyglzjdsgcnbujxzlfmdpbktltplpbfaxvwuhqotbalzqiqrlrhjvxvvwfrwuvvwyrfyolvtewj jcubyzqeabwpiuq deibixwacacitgihzwkiao gnicroamzmeyfklkgfmwouawv fvtxkztp ngpcmjuqeuhrkovzxxgthhgzoe dbpyxhcncgemfxxvcgemshlo cesqsznmhcxhwohvkg mbvefa

Random letters at the correct frequencies:

agrrntr rroe earstnehn ledititanya fdved tooug nose tateite ergseye umoeel arirrls liunpedvpreobhtersre r tedyoesyfaethego dls ntfr hg r nnaluvetodulreuelstsaiiowntodrnoyna etter r esm o t doiieee hsr ogoothssvdne lnusagulfro te alt siaoolewhegcrcfghr ctl lvni vrn tngmttmervilaf e eus r lcltetvutdouhdnod gle ias eces atli a niro rtt ll o d natcfoefluht tdchtaheaytgcbtlpho ghy llaefgcr

Random letters at the correct transition frequencies:

aghorath pe l t t ing hedesidunly o tingungnsioly prdef lvath bengagesthom dith asinis ayanst fofesew degesh tiondse ty fobely w wo mal espothexparedes ces care f a f

> vatiborорewa ongndeshobes ieg win d sies ororen s g sthedsifistuanfe pr ofor woforoninge sth lvitren in fis lifio t ord wicanorore ropondstues tin kigewor bl s mil tesesese d w or wiresencewod womparores an t thoro tugh le ithan gies

```
sample_text (generic function with 1 method)
```

Exercise 1.4 - *Language detection*

It looks like we have a decent language model, in the sense that it understands *transition frequencies* in the language. In the demo above, try switching the language between English and Spanish – the generated text clearly looks more like one or the other, demonstrating that the model can capture differences between the two languages. What's remarkable is that our "training data" was just a single paragraph per language.

In this exercise, we will use our model to write a **classifier**: a program that automatically classifies a text as either English or Spanish.

This is not a difficult task – you can download dictionaries for both languages, and count matches – but we are doing something much more exciting: we only use a single paragraph of each language, and we use a *language model* as classifier.

Mystery sample

Enter some text here -- we will detect in which language it is written!

La publicación de este programa está considerada como el hecho que dio origen al nacimiento de la Industria del videojuego en España, ya que tras el éxito de su lanzamiento en el Reino Unido abrió el mercado europeo al software made in spain.

```
@bind mystery_sample TextField((70, 8), default="""
La publicación de este programa está considerada como el hecho que dio origen al nacimiento de la Industria del videojuego en España, ya que tras el éxito de su lanzamiento en el Reino Unido abrió el mercado europeo al software made in spain.
""")
```

It looks like this text is *Spanish*!

```
"La publicación de este programa está considerada como el hecho que dio origen al nacimient
```

```
mystery_sample
```

Let's compute the transition frequencies of our mystery sample! Type some text in the box above, and check whether the frequency matrix updates.

```
27×27 Matrix{Float64}:
 0.0         0.00456621  0.00913242  …  0.0  0.0  0.0         0.0  0.0273973
 0.0         0.0         0.0            0.0  0.0  0.0         0.0  0.0
 0.00913242  0.0         0.0            0.0  0.0  0.0         0.0  0.0
 0.00456621  0.0         0.0            0.0  0.0  0.0         0.0  0.0
 0.0         0.0         0.00456621     0.0  0.0  0.0         0.0  0.0365297
 0.0         0.0         0.0         …  0.0  0.0  0.0         0.0  0.0
 0.0         0.0         0.0            0.0  0.0  0.0         0.0  0.0
 ⋮                                   ⋱                       ⋮
 0.0         0.0         0.0            0.0  0.0  0.0         0.0  0.0
 0.00456621  0.0         0.0            0.0  0.0  0.0         0.0  0.0
 0.0         0.0         0.0            0.0  0.0  0.0         0.0  0.0
 0.00456621  0.0         0.0            0.0  0.0  0.0         0.0  0.0
 0.00456621  0.0         0.0         …  0.0  0.0  0.0         0.0  0.0
 0.0136986   0.0         0.00913242     0.0  0.0  0.00456621  0.0  0.0
```

```
transition_frequencies(mystery_sample)
```

Our model will **compare the transition frequencies of our mystery sample** to those of our two language samples. The closest match will be our detected language.

The only question left is: How do we compare two matrices? When two matrices are almost equal, but not exactly, we want to quantify the *distance* between them.

👉 Write a function called `matrix_distance` which takes 2 matrices of the same size and finds the distance between them by:

1. Subtracting corresponding elements
2. Finding the absolute value of the difference
3. Summing the differences

```
matrix_distance (generic function with 1 method)
```

```
function matrix_distance(A, B)

    return √sum(((A .- B).^2))
end
```

```
distances = (English = 0.120172, Spanish = 0.0931628)
```

> **Got it!**
>
> Keep it up!

We have written a cell that selects the language with the *smallest distance* to the mystery language. Make sure sure that `matrix_distance` is working correctly, and scroll up to the mystery text to see it in action!

Further reading

It turns out that the SVD of the transition matrix can mysteriously group the alphabet into vowels and consonants, without any extra information. See this paper if you want to try it yourself! We found that removing the space from `alphabet` (to match the paper) gave better results.

Exercise 2 - *Language generation*

Our model from Exercise 1 has the property that it can easily be 'reversed' to *generate* text. While this is useful to demonstrate its structure, the produced text is mostly meaningless: it fails to model words, let alone sentence structure.

To take our model one step further, we are going to *generalize* what we have done so far. Instead of looking at *letter combinations*, we will model *word combinations*. And instead of analyzing the frequencies of bigrams (combinations of two letters), we are going to analyze n*-grams*.

Dataset

This also means that we are going to need a larger dataset to train our model on: the number of English words (and their combinations) is much higher than the number of letters.

We will train our model on the novel *Emma* (1815), by Jane Austen. This work is in the public domain, which means that we can download the whole book as a text file from `archive.org`. We've done the process of downloading and cleaning already, and we have split the text into word and punctuation tokens.

```
emma = let
    raw_text =
    read(download("https://ia800303.us.archive.org/24/items/EmmaJaneAusten_753/emma_pd
    f_djvu.txt"), String)

    first_words = "Emma Woodhouse"
    last_words = "THE END"
    start_index = findfirst(first_words, raw_text)[1]
    stop_index = findlast(last_words, raw_text)[end]

    raw_text[start_index:stop_index]
end;
```

```
splitwords (generic function with 1 method)
```

```
emma_words =
 ["Emma", "Woodhouse", ",", "handsome", ",", "clever", ",", "and", "rich", ",", "with", "a", "
```

```
emma_words = splitwords(emma)
```

```
forest_words =
 ["Although", "the", "word", "forest", "is", "commonly", "used", ",", "there", "is", "no", "ur
```

```
forest_words = splitwords(samples.English)
```

Exercise 2.1 - *bigrams revisited*

The goal of the upcoming exercises is to **generalize** what we have done in Exercise 1. To keep things simple, we *split up our problem* into smaller problems. (The solution to any computational problem.)

First, here is a function that takes an array, and returns the array of all **neighbour pairs** from the original. For example,

```
bigrams([1, 2, 3, 42])
```

gives

```
[[1,2], [2,3], [3,42]]
```

(We used integers as "words" in this example, but our function works with any type of word.)

```
bigrams (generic function with 1 method)
```

```
function bigrams(words)
    starting_positions = 1:length(words)-1

    map(starting_positions) do i
        words[i:i+1]
    end
end
```

```
[[1, 2], [2, 3], [3, 42]]
```

```
bigrams([1, 2, 3, 42])
```

👉 Next, it's your turn to write a more general function `ngrams` that takes an array and a number n, and returns all **subsequences of length n**. For example:

```
ngrams([1, 2, 3, 42], 3)
```

should give

```
[[1,2,3], [2,3,42]]
```

and

```
ngrams([1, 2, 3, 42], 2) == bigrams([1, 2, 3, 42])
```

```
ngrams (generic function with 1 method)
```

```
function ngrams(words, n)
    starting_positions = 1:length(words)-n+1
    map(starting_positions) do i
        words[i:i+n-1]
    end
end
```

```
[[1, 2, 3], [2, 3, 42]]
```

```
ngrams([1, 2, 3, 42], 3)
```

```
[["Although", "the", "word", "forest"], ["the", "word", "forest", "is"], ["word", "forest", "
```

```
ngrams(forest_words, 4)
```

Got it!

Yay ♥

Hint

[blurred]

If you are stuck, you can write `ngrams(words, n) = bigrams(words)` (ignoring the true value of n), and continue with the other exercises.

Exercise 2.2 - *frequency matrix revisisted*

In Exercise 1 we use a 2D array to store the bigram frequencies, where each column or row corresponds to a character from the alphabet. To use trigrams we could store the frequencies in a 3D array, and so on.

However, when counting words instead of letters we run into a problem: A 3D array with one row, column and layer per word has too many elements to store on our computer!

Emma consists of 8465 unique words. This means that there are 606 billion possible trigrams - that's too many!

Although the frequency array would be very large, *most entries are zero*. For example, *"Emma"* is a common word, but the sequence of words *"Emma Emma Emma"* never occurs in the novel. We about the *sparsity* of the non-zero entries in a matrix. When a matrix is sparse in this way, we can **store the same information in a more efficient structure**.

Julia's `SparseArrays.jl` package might sound like a logical choice, but the arrays from that package support only 1D and 2D types, and we also want to directly index using strings, not just integers. So instead we will use a **dictionary**, or `Dict` in Julia.

Take a close look at the next example. Note that you can click on the output to expand the data viewer.

```
healthy =  Dict("vegetables" => ["🌽", "🎃", "🍕"], "fruits" => ["🍎", "🍊"])
```

```
healthy = Dict("fruits" => ["🍎", "🍊"], "vegetables" => ["🌽", "🎃", "🍕"])
```

```
["🍎", "🍊"]
```

```
healthy["fruits"]
```

(Did you notice something funny? The dictionary is *unordered*, this is why the entries were printed in reverse from the definition.)

You can dynamically add or change values of a `Dict` by assigning to `my_dict[key]`. You can check whether a key already exists using `haskey(my_dict, key)`.

👉 Use these two techniques to write a function `word_counts` that takes an array of words, and returns a `Dict` with entries `word => number_of_occurences`.

For example:

```
word_counts(["to", "be", "or", "not", "to", "be"])
```

should return

```
Dict(
    "to" => 2,
    "be" => 2,
    "or" => 1,
    "not" => 1
)
```

```
word_counts (generic function with 1 method)
```

```
function word_counts(words::Vector)
    counts = Dict()
    # your code here
    for word in words
        if !haskey(counts,word)
            counts[word]=1
        else
            counts[word] += 1
        end
    end
    return counts
end
```

```
Dict("or" => 1, "not" => 1, "to" => 2, "be" => 2)
```

```
1 word_counts(["to", "be", "or", "not", "to", "be"])
```

Got it!

Good job!

👉 Write code to find how many times `"Emma"` occurs in the book.

```
emma_count = 865
```

```
1 emma_count = (word_counts(emma_words))["Emma"]
```

Got it!

Let's move on to the next section.

Great! Let's get back to our n-grams. For the purpose of generating text, we are going to store a *completion cache*. This is a dictionary where each key is an $(n-1)$-gram, and the corresponding value is the vector of all those words which can complete it to an n-gram. Let's look at an example:

```
let
    trigrams = ngrams(split("to be or not to be that is the question", " "), 3)
    cache = completions_cache(trigrams)
    cache == Dict(
        ["to", "be"] => ["or", "that"],
        ["be", "or"] => ["not"],
        ["or", "not"] => ["to"],
        ...
    )
end
```

So for trigrams the keys are the first 2 words of each trigram, and the values are arrays containing every third word of those trigrams.

If the same n-gram occurs multiple times (e.g. "said Emma laughing"), then the last word ("laughing") should also be stored multiple times. This will allow us to generate trigrams with the same frequencies as the original text.

👉 Write the function `completion_cache`, which takes an array of ngrams (i.e. an array of arrays of words, like the result of your `ngram` function), and returns a dictionary like described above.

```
completion_cache (generic function with 1 method)
```

```
function completion_cache(ngrams)
    cache = Dict()
    for ng in ngrams
        prefix = ng[1:end-1]
        completion = ng[end]
        if haskey(cache, prefix)
            push!(cache[prefix], completion)
        else
            cache[prefix] = [completion]
        end
    end

    return cache
end
```

```
Dict(["or", "not"] => ["to"], ["to", "be"] => ["or", "that"], ["be", "or"] => ["not"], ["not"
```

```
let
    trigrams = ngrams(split("to be or not to be that is the question", " "), 3)
    completion_cache(trigrams)
end
```

What is this cache telling us? In our sample text, the words "to be" were followed by "or" and by "that". So if we are generating text, and the last two words we wrote are "to be", we can look at the cache, and see that the next word can be either "or" or "that".

```
Dict(["\"", "and"] => ["\""], ["15", "]"] => ["Possibly"], ["cognates", "in"] => ["Romance"],
```

```
completion_cache(ngrams_circular(forest_words, 3))
```

Exercise 2.4 - *write a novel*

We have everything we need to generate our own novel! The final step is to sample random ngrams, in a way that each next ngram overlaps with the previous one. We've done this in the function `generate_from_ngrams` below - feel free to look through the code, or to implement your own version.

`generate_from_ngrams`

```
generate_from_ngrams(grams, num_words)
```

Given an array of ngrams (i.e. an array of arrays of words), generate a sequence of `num_words` words by sampling random ngrams.

`ngrams_circular`

Compute the ngrams of an array of words, but add the first n-1 at the end, to ensure that every ngram ends in the the beginning of another ngram.

generate

```
generate(source_text::AbstractString, num_token; n=3, use_words=true)
```

Given a source text, generate a `String` that "looks like" the original text by satisfying the same ngram frequency distribution as the original.

Interactive demo

Enter your own text in the box below, and use that as training data to generate anything!

```
Although the word forest is commonly used, there
is no universally recognised precise definition,
with more than 800 definitions of forest used
around the world.[4] Although a forest is usually
defined by the presence of trees, under many
```

Using 3 grams for characters

magnation, used the coman thavia ard st (alish arobably typenot ing groduced if harive,[9] was forld pasilva for hund.[4] ons und first ing wine areest und for for mortuguest, ulackinitypen trest ish ariveriva; cognated forès), dest use Lation and thouguagnity sta, demagestion trew thougualso frow th der ty the Ita, withavian the Romatine wilvă; cognian flord derece se Carobably the Engliand Portu

```
generate(
    generate_demo_sample, 400;
    n=generate_sample_n_letters,
    use_words=false
) |> Quote
```

Using 3 grams for words

, there is no universally recognised precise definition , with more than 800 definitions of forest used around the world .[4] Although a forest if it grew trees in the future ,[9] or was legally designated as a forest regardless of vegetation type .[10][11] The word was not endemic to Romance languages , e . g . the Italian foresta , denoting " open wood ", Carolingian scribes first used foresta in the past , will grow trees in the past , will grow trees in the Capitularies of Charlemagne specifically to

Automatic Jane Austen

Uncomment the cell below to generate some Jane Austen text:

, the same Mr . Knightley had a cheerful manner , which always interested her . She was obliged to play ; and the seeing him so , with the utmost delight . But what shall I say ? 1 shall be very well off as he was sleeping on the sofa , removing to a seat by her sister , and giving her all her attention . ' My brother and sister when they come to us next Friday or Saturday , and the devil of a temper .' Emma liked the subject so well that she began upon

```
generate(emma, 100; n=4) |> Quote
```

Function library

Just some helper functions used in the notebook.

```
Quote (generic function with 1 method)
```

```
show_pair_frequencies (generic function with 1 method)
```

```
compimg (generic function with 2 methods)
```

```
hint (generic function with 1 method)
```

```
almost (generic function with 1 method)
```

```
still_missing (generic function with 2 methods)
```

```
keep_working (generic function with 2 methods)
```

```
yays =
  [Fantastic!, Splendid!, Great!, Yay ♥, Great! 🥕, Well done!, Keep it up!, Good job!, Awesome!,
```

```
correct (generic function with 2 methods)
```

```
not_defined (generic function with 1 method)
```

```
todo (generic function with 1 method)
```