



# Project 2A: NGordnet (NGrams)

[FAQ](#)[Introduction](#)[Getting Started](#)[Building An NGrams Viewer](#)[TimeSeries](#)[TimeSeries Tips](#)[NGramMap](#)[Input File Formats](#)[NGramMap Tips](#)[HistoryTextHandler](#)[HistoryTextHandler Tips](#)[HistoryHandler](#)[HistoryHandler Tips](#)[Deliverables and Scoring](#)[Acknowledgements](#)

---

## FAQ

Each assignment will have an FAQ linked at the top. You can also access it by adding “/faq” to the end of the URL. The FAQ for Project 2A is located [here](#).

---

## Introduction

In this project, we will build a browser based tool for exploring the history of word usage in English texts. We have provided the front end code (in Javascript and HTML) that collects user inputs and displays outputs. Your Java code will be the back end for this tool, accepting input and generating appropriate output for display.

A video introduction to this project can be found below (or [at this link](#)).



To support this tool, you will write a series of Java packages that will allow for data analysis. Along the way we'll get lots of experience with different useful data structures. The early part of the project (proj2a) will start by telling you exactly what functions to write and classes to create. The later part (proj2b) will be more open to your own design.

---

## Getting Started

To get started, use `git pull skeleton main` as usual.

You'll also need to download the Project 2 data files (not provided via GitHub for space reasons).

 **Task:** Download the data files [at this link](#).

You should unzip this file into the proj2 directory such that the `data` folder is at the same level as the `ngordnet` and `static` folders.

- [How to unzip folders on Windows](#)
- [How to unzip folders on Mac](#)

Once you are done with this step, your `proj2a` directory should look like this:

```
proj2a
├── data
│   ├── ngrams
│   └── wordnet
├── ngordnet
└── static
```

Note that we've set up hidden `.gitignore` files in the skeleton code so that Git will avoid uploading these data files. This is intentional. Uploading the data files to GitHub will result in a lot of headaches for everybody, so please don't mess with any file called `.gitignore`. If you need to work on multiple machines, you should download the zip file once for each machine.

If `NgordnetQuery` doesn't compile, make sure you are using Java version 15 (preview) or higher (preferably 17+).

Check that your style checker plugin is up-to-date. The version should be between 2.0.8-2.0.10! See [this Ed post](#) for more details.

A video guide to setting up your computer for this project can be found [at this link](#). Note that some files/filenames may be slightly different; in particular, the `hugbrowsermagic` directory in the video is now just called `browser` in your skeleton files.

---

## Building An NGrams Viewer

The [Google Ngram dataset](#) provides many terabytes of information about the historical frequencies of all observed words and phrases in English (or more precisely all observed [ngrams](#)). Google provides the [Google Ngram Viewer on the web](#), allowing users to visualize the relative historical popularity of words and phrases. For example, the link above plots the **relative popularity** of the phrases “global warming” (a 2gram) and “to the moon” (a 3gram).

In Project 2A, you will be build a version of this tool that only handles 1grams. In other words, you’ll only be able to handle individual words. We’ll only use a small subset (around 300 megabytes) of the full 1grams dataset, as larger datasets will require more sophisticated techniques that are out of scope for this class.

Most of your work for Project 2A will be in the `ngordnet.ngrams` package, though some work will also be in `ngordnet.main`, as well as potentially the `ngordnet.plotter` package.

---

## TimeSeries

A `TimeSeries` is a special purpose extension of the existing `TreeMap` class where the key type parameter is always `Integer`, and the value type parameter is always `Double`. Each key will correspond to a year, and each value a numerical data point for that year.

For example, the following code would create a `TimeSeries` and associate the number 3.6 with 1992 and 9.2 with 1993.

```
TimeSeries ts=new TimeSeries();  
ts.put(1992,3.6);  
ts.put(1993,9.2);
```

The `TimeSeries` class provides some additional utility methods to the `TreeMap` class, which it extends.

 **Task:** Fill out the `TimeSeries` class (located in the `ngordnet/ngrams/TimeSeries.java` file) according to the API provided in the file. Be sure to read the comments above each method.

**Note:** there are two constructors for this class, and you must complete them both.

! For an example of how `TimeSeries` objects are used, check out the test named `testFromSpec()` in the `TimeSeriesTest.java` file that we've provided. This test creates a `TimeSeries` of cat and dog populations and then computes their sum. Note that there is no value for 1993 because that year does not appear in either `TimeSeries`.

! You may not add additional public methods to this class. You're welcome to add additional private methods.

---


## TimeSeries Tips

- `TimeSeries` objects should have no instance variables. A `TimeSeries` is-a `TreeMap`. That means your `TimeSeries` class also has access to all methods that a `TreeMap` has; see [the TreeMap API](#).
- **You should never impute any zeroes.** In other words, nowhere should you have any code which fills in a zero if a value is unavailable.
- The provided `TimeSeriesTest` class provides a simple test of the `TimeSeries` class. Feel free to add your own tests.
  - Note that the unit tests we gave you **do not** evaluate the correctness of the `dividedBy` method.
- You'll notice in `testFromSpec()` that we did not directly compare `expectedTotal` with `totalPopulation.data()`. This is because doubles are prone to rounding errors, especially after division operations (for reasons that you will learn in 61C). Thus, `assertThat(x).isEqualTo(y)` may unexpectedly return false when `x` and `y` are doubles. Instead, you should use `assertThat(x).isWithin(1E-10).of(y)`, which returns true as long as `x` and `y` are within 1E-10 of each other.
- You may assume that the `dividedBy` operation never divides by zero.

---

## NGramMap

The `NGramMap` class will provide various convenient methods for interacting with Google's NGrams dataset. This task is more open-ended and challenging than the creation of the `TimeSeries` class. As with `TimeSeries`, you'll be filling in the methods of an existing `NGramMap.java` file. **NGramMap should not extend any class.**

 **Task:** Fill out the `NGramMap` class (located in the `ngordnet/ngrams/NGramMap.java` file) according to the API provided in the file. Once again, be sure to read the comments above each method.

! For an example of an `NGramMap` at work, the `testOnLargeFile()` in `NGramMapTest` creates an `NGramMap` from the `top_14377_words.csv` and `total_counts.csv` files (described below). It then performs various operations related to the occurrences of the words "fish" and "dog" in the period between 1850 and 1933.

EDIT (2/27): If you call a method that returns a `TimeSeries`, and there is no available data for the given method call, you should return an empty `TimeSeries`. For example:  
`TimeSeries ts = ngm.countHistory("adopt", 1400, 1410)` should return a `TimeSeries` with nothing in it.

⚠ You may not add additional public methods to this class. You're welcome to add additional private methods.

---

## Input File Formats

The NGram dataset comes in two different file types. The first type is a "words file". Each line of a words file provides tab separated information about the history of a particular word in English during a given year.

airport	2007	175702	32788
airport	2008	173294	31271
request	2005	646179	81592
request	2006	677820	86967
request	2007	697645	92342
request	2008	795265	125775
wandered	2005	83769	32682

wandered	2006	87688	34647
wandered	2007	108634	40101
wandered	2008	171015	64395

The first entry in each row is the word. The second entry is the year. The third entry is the number of times that the word appeared in any book that year. The fourth entry is the number of distinct sources that contain that word. **Your program should ignore this fourth column.** For example, from the text file above, we can observe that the word “wandered” appeared 171,015 times during the year 2008, and these appearances were spread across 64,395 distinct texts. For this project, we never care about the fourth entry (total number of volumes).

The other type of file is a “counts file”. Each line of a counts file provides comma separated information about the total corpus of data available for each calendar year.

1470	984	10	1
1472	117652	902	2
1475	328918	1162	1
1476	20502	186	2
1477	376341	2479	2

The first entry in each row is the year. The second is the total number of words recorded from all texts that year. The third number is the total number of pages of text from that year. The fourth is the total number of distinct sources from that year. Your program should ignore the third and fourth columns. For example, we see that Google has exactly one English language text from the year 1470, and that it contains 984 words and 10 pages. For the purposes of our project the 10 and the 1 are irrelevant.

You may wonder why one file is tab separated and the other is comma separated. I didn’t do it, Google did. Luckily, this difference won’t be too hard to handle.

---

## NGramMap Tips

There is a lot to think about for this part of the project. We’re trying to mimic the situation in the real world where you have some big open-ended

problem and have to figure out the approach from scratch. This can be intimidating! It will likely take some time and a lot of experimentation to figure out how to proceed. To help keep things from being too difficult, we've at least provided a list of methods to implement. Keep in mind that in the real world (and in proj2b and proj3), even the list of methods will be your choice.

Your code should be fast enough that you can create an `NGramMap` using `top_14377_words.csv`. Loading should take less than 60 seconds (maybe a bit longer on an older computer). If your computer has enough memory, you should also be able to load `top_49887_words.csv`.

- The bulk of your work in this class will be implementing the constructor. You'll need to parse through the provided data files and store this data in a data structure (or structures) of your choice.
  - This choice is important, since picking the right data structure(s) can make your life a lot easier when implementing the rest of the methods. Thus, we recommend taking a look at the rest of the methods first to help you decide what data structure might be best; then, begin implementing the constructor.
- Avoid using a `HashMap` or `TreeMap` as an **actual type argument** for your maps. This is usually a sign that what you actually want is a custom defined type. In other words, if your instance variables include a nested mapping that looks like `HashMap<blah, HashMap<blah, blah>>`, then a `TimeSeries` or some other class you come up with might be useful to keep in mind instead.
- We have not taught you how to read files in Java. We recommend using the `In` class. The official documentation can be found [here](#). However, you're welcome to use whatever technique you'd like that you learn about online. We provide an example class `FileReaderDemo.java` that gives examples of how to use `In`.
- If you use `In`, don't use `readAllLines` or `readAllStrings`. These methods are slow. Instead, read inputs one chunk at a time. See `ngordnet/main/FileReaderDemo.java` for an example.
  - Additionally, to check if there are any lines left in a file, you should use `hasNextLine` (and not `isEmpty`).



- Our provided tests only cover some methods, but some methods are only tested on a very large file. You will need to write additional tests.
    - Rather than using one of the large input files (e.g. `top_14377_words.csv`), we recommend starting with one of the smaller input files, either `very_short.csv` or `words_that_start_with_q.csv`.
  - **You should never impute any zeroes.** In other words, nowhere should you have any code which fills in a zero if a value is unavailable.
  - If it helps speed up your code, you can assume year arguments are between 1400 and 2100.
  - `NGramMap` should not extend any other class.
  - Your methods should be simple! If you pick the right data structures, the methods should be relatively short.
- 

## HistoryTextHandler

In this final part of Project 2A, we'll do a bit of software engineering to set up a web server that can handle NgordnetQueries. While this content isn't strictly related to data structures, it is incredibly important to be able to take projects and deploy them for real world use.

**Note:** You should only begin this part when you are fairly confident that `TimeSeries` and `NGramMap` are working properly.

1. In your web browser, open up the `ngordnet_2a.html` file in the `static` folder. You can do this from your finder menu in your operating system, or by right-clicking on the `ngordnet_2a.html` in IntelliJ, clicking "Open in", then "Browser". You can use whatever browser you want, though TAs will be most familiar with Chrome. You'll see a web browser based interface that will ultimately (when you're done with the project) allow a user to enter a list of words and display a visualization.
2. Try entering "cat, dog" into the "words" box, then click `History (Text)`. You'll see that nothing useful shows up. Optional: If you open the developer tools in your web browser (see Google for how

to do this), you'll see an error that looks like either "CONNECTION\_REFUSED" or "INVALID\_URL". The problem is that the Javascript tries to access a server to generate the results, but there is no web server running that can handle the request to see the history of cat and dog.

3. Open the `ngordnet.main.Main` class. This class's `main` method first creates a `NgordnetServer` object. The API for this class is as follows: First, we call `startUp` on the `NgordnetServer` object, then we "register" one or more `NgordnetQueryHandler` using the `register` command. The precise details here are beyond the scope of our class.

The basic idea is that when you call

```
hns.register("historytext", new DummyHistoryTextHandler(ngm)),
```

an object of type `DummyHistoryTextHandler` is created that will handle any clicks to the `History (Text)` button.

4. Try running the `ngordnet.main.Main` class. The terminal output in IntelliJ might be red, but as long as you see the line: `INFO org.eclipse.jetty.server.Server - Started...`, the server started correctly. Now open the `ngordnet_2a.html` file again, enter "cat, dog" again, then click `History (Text)`. This time, you should see a message that says:

```
You entered the following info into the browser:
Words: [cat, dog]
Start Year: 2000
End Year: 2020
```

5. Now open `ngordnet.main.DummyHistoryTextHandler`, you'll see a `handle` method. This is called whenever the user clicks the `History (Text)` button. The expected behavior should instead be that when the user clicks `History (Text)` for the prompt above, the following text should be displayed:

```
cat: {2000=1.71568475416827E-5, 2001=1.6120939684412677E-5, 2002=1.
dog: {2000=3.127517699525712E-5, 2001=2.99511426723737E-5, 2002=3.0
```




To pass on the autograder, the formatting of the output must match exactly.

- All lines of text, including the last line, should end in a new line character.
- All whitespace and punctuation (commas, braces, colons) should follow the example above.

These numbers represent the **relative popularity** of the words cat and dog in the given years. Due to rounding errors, your numbers may not be exactly the same as shown above. Your format should be exactly as shown above: specifically the word, followed by a colon, followed by a space, followed by a string representation of the appropriate `TimeSeries` where key-value pairs are given as a comma-separated list inside curly braces, with an equals sign between the key and values. Note that you don't need to write any code to generate the string representation of each `TimeSeries`, you can just use the `toString()` method.

Now it's time to implement the HistoryText button!

 **Task:** Create a new file called `HistoryTextHandler.java` that takes the given `NgordnetQuery` and returns a `String` in the same format as above.

Then, modify `Main.java` so that your `HistoryTextHandler` is used when someone clicks `History (Text)`. In other words, instead of registering `DummyHistoryTextHandler`, you should register your `HistoryTextHandler` class instead.

---

## HistoryTextHandler Tips

- The constructor for `HistoryTextHandler` should be of the following form: `public HistoryTextHandler(NGramMap map)`.
- Use the `DummyHistoryTextHandler.java` as a guide, pattern matching where appropriate. Being able to tinker with example code and bend it to your will is an incredibly important real world skill. Experiment away, don't be afraid to break something!
- **For Project 2A**, you can ignore the `k` instance variable of `NgordnetQuery`.

- Use the `.toString()` method built into the `TimeSeries` class that gets inherited from `TreeMap`.
  - For your `HistoryTextHandler` to be able to do something useful, it's going to need to be able to access the data stored in your `NGramMap`. **DO NOT MAKE THE NGRAM MAP INTO A STATIC VARIABLE!** This is known as a “global variable” and is rarely the appropriate solution for any problem. Hint: Your `HistoryTextHandler` class can have a constructor.
- 

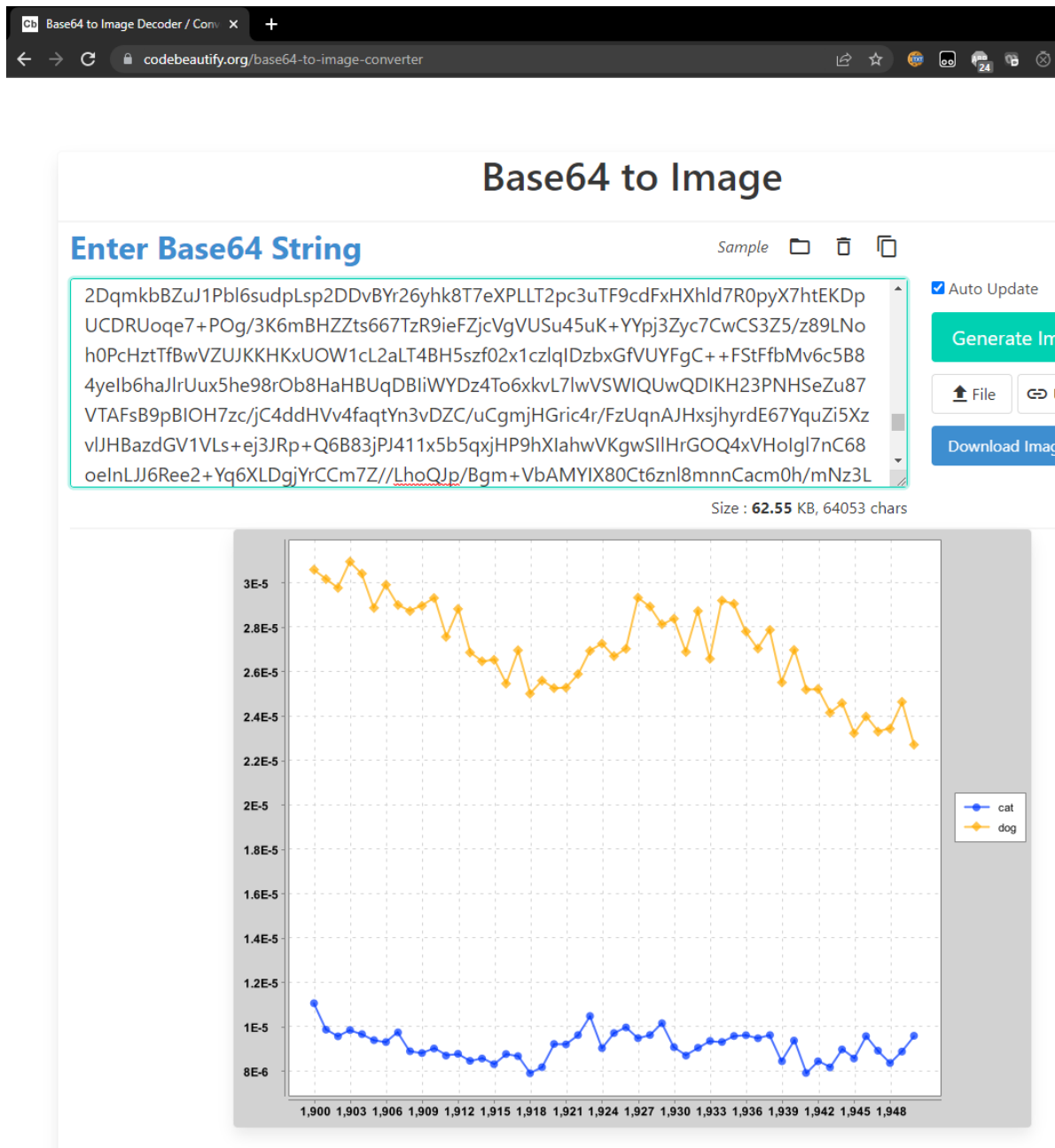
## HistoryHandler

The text based history from the previous section is not useful for much other than auto-grading your work. Actually using our tool to discover interesting things will require visualization.

The `ngordnet.main.PlotDemo` provides example code that uses your `NGramMap` to generate a visual plot showing the relative frequency of the words cat and dog between 1900 and 1950. Try running it. If your `NGramMap` class is correct, you should see a very long string printed to your console that might look something like:

```
iVBORw0KGg...
```

This string is a base 64 encoding of an image file. To visualize it, go to [codebeautify.org](https://codebeautify.org). Copy and paste this entire string into the website, and you should see a plot similar to the one shown below:

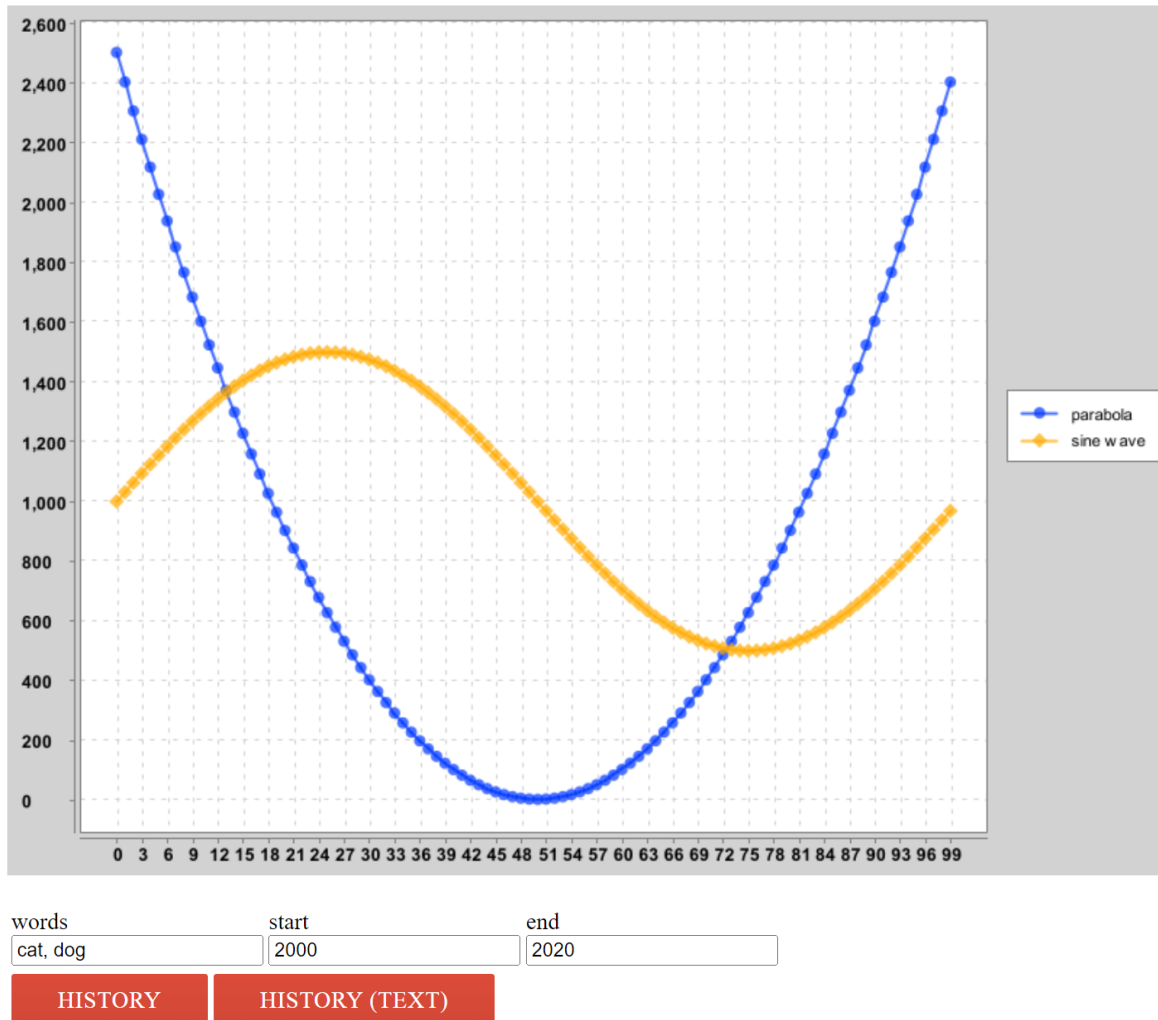


What's going on here? The string your code printed IS THE IMAGE. Keep in mind that any data can be represented as a string of bits. This website knows how to decode this string into the corresponding image, using a predefined standard.


If you look at the plotting library, this code relies on the `ngordnet.Plotter.generateTimeSeriesChart` method, which takes two arguments. The first is a list of strings, and the second is a `List<TimeSeries>`. The `TimeSeries` are all plotted in a different color, and each is assigned the label given in the list of strings. Both lists must be of the same length (since the  $i$ th string is the label for the  $i$ th time series).

The `ngordnet.Plotter.generateTimeSeriesChart` method returns an object of type `XYChart`. This object can in turn either be converted into base 64 by the `ngordnet.Plotter.encodeChartAsString` method, or can be displayed to the screen directly by `ngordnet.Plotter.displayChart`.

In your web browser, again open up the `ngordnet_2a.html` file in the `static` folder. With your `ngordnet.main.Main` class running, enter “cat, dog” into the “words” box, then click “history”. You’ll see the strange image below:



You’ll note that the code is not plotting the history of cat and dog, but rather a parabola and a sinusoid. If you open `DummyHistoryHandler`, you’ll see why.

 **Task:** Create a new file called `HistoryHandler.java` that takes the given `NgordnetQuery` and returns a `String` that contains a base-64 encoded image of the appropriate plot.

Then, modify the `Main.java` so that your `HistoryHandler` is called when someone clicks the `History` button.

---

## HistoryHandler Tips

- The constructor for `HistoryHandler` should be of the following form:  
`public HistoryHandler(NGramMap map)`.
  - Just like before, use `DummyHistoryHandler.java` as a guide. As mentioned in the previous section, we really want you to learn the skill of tinkering with complex library code to get the behavior you want.
- 

## Deliverables and Scoring

You are responsible for implementing four classes:

- **TimeSeries (30%):** Correctly implement `TimeSeries.java`.
- **NGramMap Count (20%):** Correctly implement `countHistory()` and `totalCountHistory()` in `NGramMap.java`.
- **NGramMap Weight (30%):** Correctly implement `weightHistory()` and `summedWeightHistory()` in `NGramMap.java`.
- **HistoryTextHandler (10%):** Correctly implement `HistoryTextHandler.java`.
- **HistoryHandler (10%):** Correctly implement `HistoryHandler.java`.

Below is the velocity limiting policy for this assignment:

1. You will start with **8 tokens, each of which have a 24 hour refresh time**.
2. At **9:00PM on March 8th** (3 hours before the deadline), you will be reset to **4 tokens, each of which have a 15 minute refresh time**.
3. At **12:00AM on March 9th**, you will again be reset to **8 tokens with a 24 hour refresh**. This policy will remain in place for the remainder of the semester.

# Acknowledgements

The WordNet part of this assignment is loosely adapted from Alina Ene and Kevin Wayne's [Wordnet assignment](#) at Princeton University.

*Last built: 2023-10-26 18:40 UTC*