This is the ReadMe document for BIS634 2021 Fall, HW4, by Olina (Qiuyu) Zhu, NetID: qz258.

All code and their corresponding output are attached after page 8 in this document, but some output are screenshotted and pasted below each individual exercise response.

The dataset used in Exercise 2 was retrieved from https://simplemaps.com/data/world-cities, and code used to calculate distance was retrieved from https://stackoverflow.com/questions/4913349/haversine-formula-in-python-bearing-and-distance-between-two-gps-points and later modified.

Exercise 1:

The optimal choices of a and b as found using the gradient descent algorithm are:
a: 0.7102008000009036
b: 0.16787267500030226

Note that this is also the a and b values for global minimum at 1.00004896388. The local minimum is 1.00005261742 at a = 0.7112031249999144 and b = 0.17089062500010838.

Given that the function is unknown, and one can only retrieve results by querying given API, I estimated the gradient using the function given in class: $\triangle f \approx [(f(a+\triangle a, b) - f(a,b))/\triangle a, (f(a, b+\triangle b) - f(a,b))/\triangle b]$.

I chose the step size to be 0.25, and delta to be 1e-4 as was done in class. I started with an initial guess of a=0.4 and b=0.8, for reasons:

I. They are not at the extremes of possible inputs (0, 1)
II. a and b assume very different values

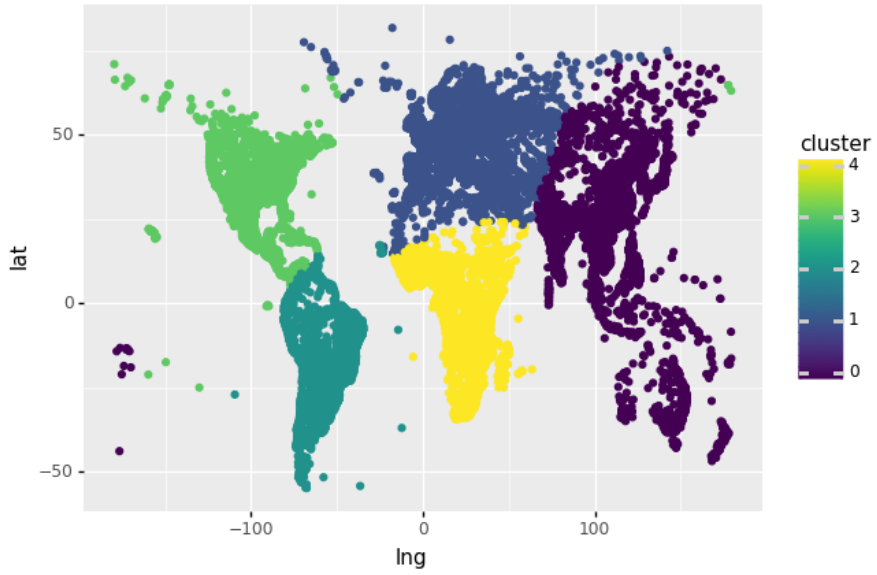
I chose the stopping criteria to be when the function value varies for less than 0.0001 upon each iteration. I chose this because any variation at the 1e-4 level is relatively insignificant and our resulting a and b values are likely extremely close to the minimum point.

Exercise 2:

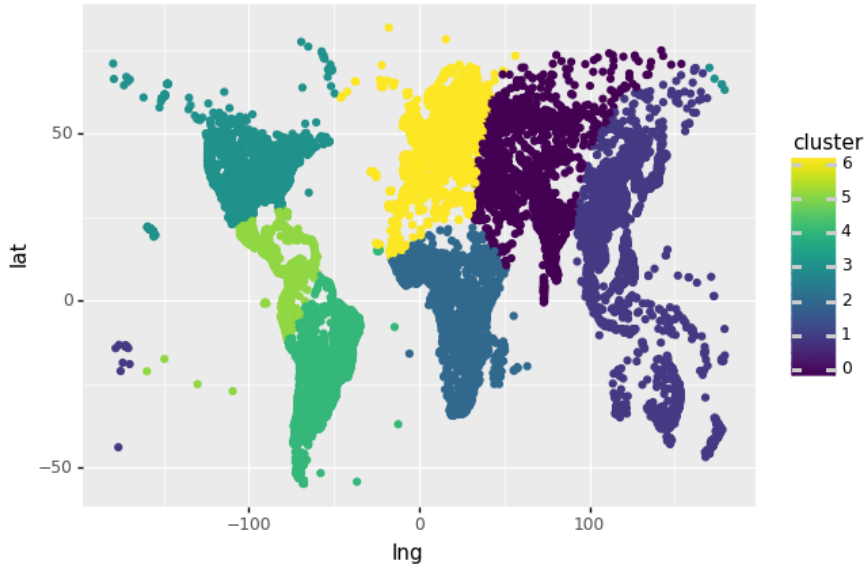
Note: I was not able to get the cartopy library to work on my device, so unfortunately I was not able to do a good projection of the map, although I have provided code for how I would have done it if the library worked (modified from code given in assignment manual). Because of this, my output maps are likely inaccurate due to poor projection.

Diversity of Results with each k: I firstly want to comment on the diversity of results not with each k, but with multiple runs of the same k - it seems that because the cluster centers are chosen pseudo-randomly, the resulting clusters shown on the map are also different. Sometimes Russia and Mongolia were in the same cluster, other times Eurasia was largely a cluster, and other times Europe and Africa were in the same cluster. In terms of diversity of results with k, Europe, Russia, China, and African countries could be distributed among five or more clusters with higher k (k=15), whereas with lower k (k=5) the clusters were mostly defined by continent, but did vary slightly per run for adjacent continents (like Europe and Africa, as mentioned before). For k=7, most of the variation occurred among the Eurasian and African plate, whereas the clusters for the Americas were largely stable every run. Diversity seemed to increase with k size.

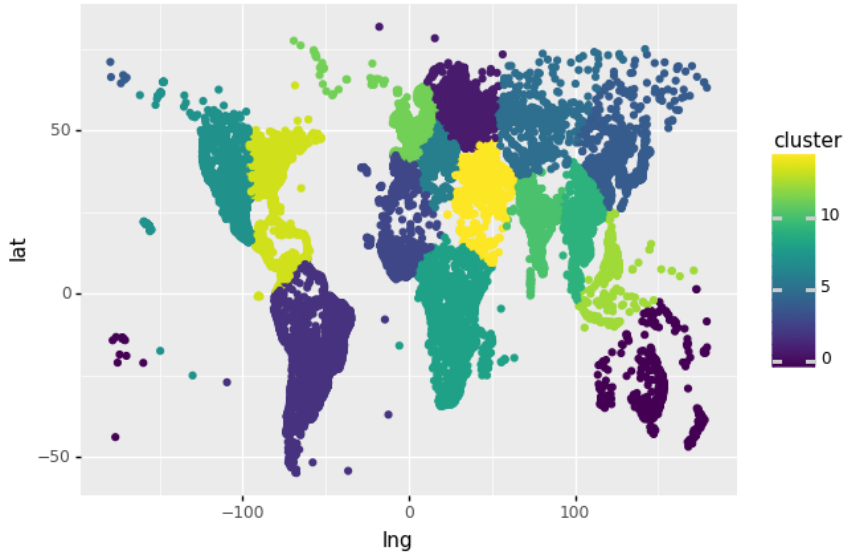
The maps are shown below:



K = 5



K = 7



K = 15

* Note: I tried running with plt.savefig("Q2.png") using cartopy projections, but I could not get the color mapping to work, and it only worked when color is specifically set to a predefined color like "red" or "black", but not when I used the color assignment code as shown in my script. To generate the above results (incorrect but do print), please comment out everything from "# map cluster to colors" to "plt.show()", and uncomment the line underneath.

Exercise 3:

I added the runtime of each n value for both algorithms to dictionaries (for fib and lru_fib) with n as the key and runtime as the value. I then transformed the dictionaries into data frames and merged the two together. Note that I was not able to get runtime for n>45 for Fibonacci naive implementation, because the runtime was simply too great (I tried running fib(100), it ran for over 12 hours and did not complete). I then plotted both runtimes onto separate graphs as well as on a single graph (three graphs in total) to visualize the difference in runtime as a function of n value.

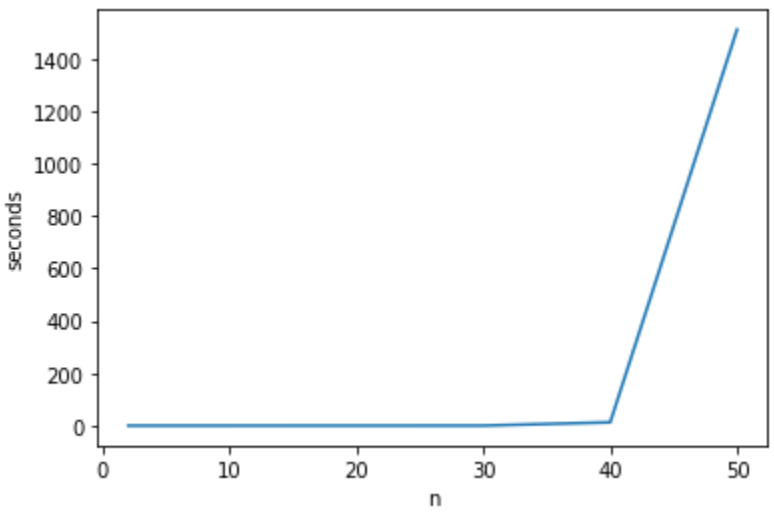
I plotted the data separately first because on the combined graph one could barely tell the impacts of n value on lru_fib gradient, due to the large y-scale used to accommodate runtime values for naive fib. Therefore, plotting on separate graphs enables better visualization of the lru_fib runtime gradient.

Interestingly, for the lru_fib runtime plot, one could see a peak at n=700, and another at n=1000. Please note that I ran this algorithm multiple times and only showed one graph, but other runs showed peaks elsewhere, and this seems to occur randomly. The peaks seem significant but are of magnitude 0.001 seconds, which I believe could just be due to overhead.

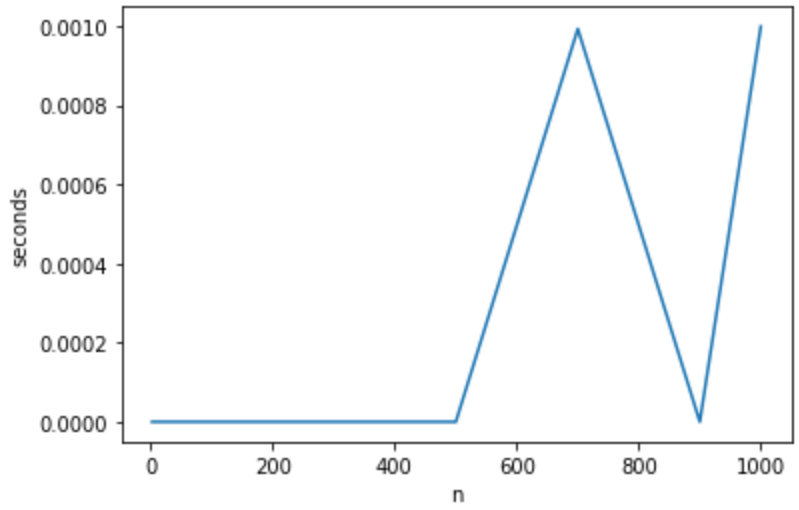
The graphs are as follows:

	n	fib_seconds	lru_seconds
0	2	0.000000	0.000000
1	10	0.000000	0.000000
2	30	0.103721	0.000000
3	50	1510.915282	0.000000
4	40	12.661980	0.000000
5	45	139.336617	0.000000
6	70	NaN	0.000000
7	90	NaN	0.000000
8	100	NaN	0.000000
9	200	NaN	0.000000
10	300	NaN	0.000000
11	500	NaN	0.000000
12	700	NaN	0.000993
13	900	NaN	0.000000
14	1000	NaN	0.000999

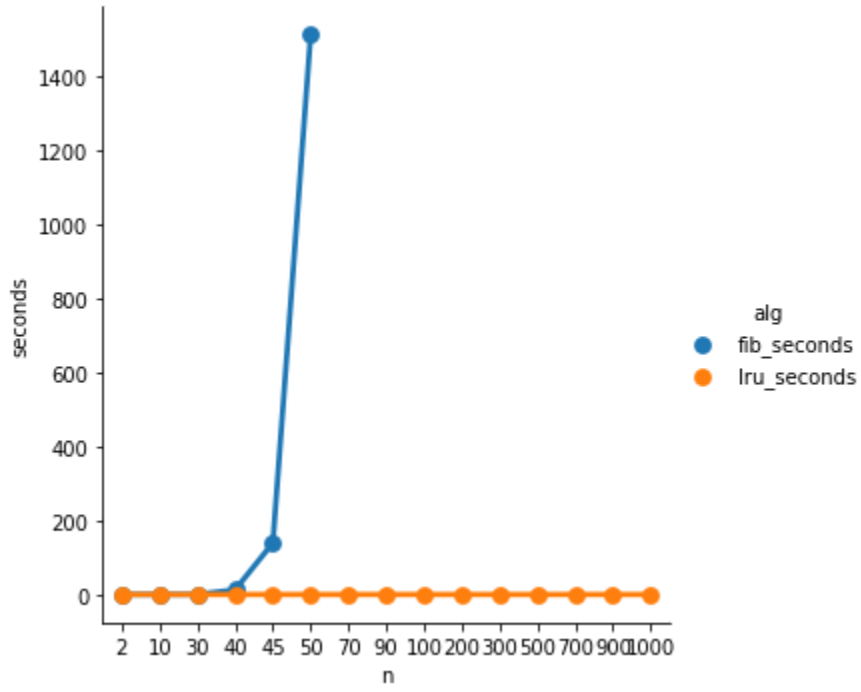
Dataframe



Fib_seconds graph



Lru_seconds graph

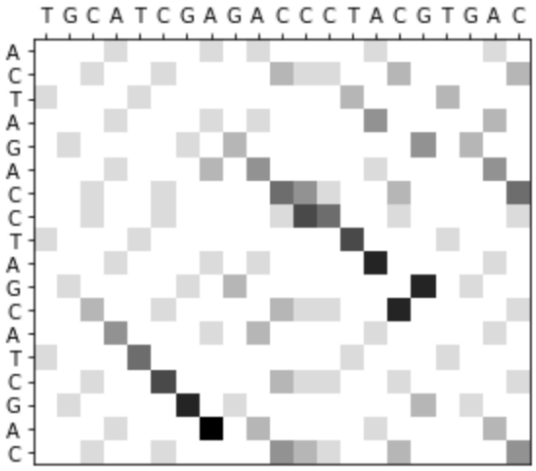


Combined graph

Exercise 4:

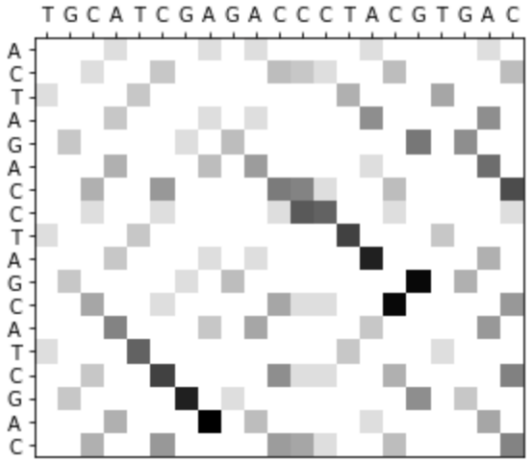
I tested my function using the given test code but with match, gap_penalty, and mismatch_pentalty = 1 (defaults). The score is 7 because there is a gap and the length of the sequence is 8. The output is as follows:

```
match = gap_penalty = mismatch_pentalty = 1:
Longest matching score: 7.0
Matching sequences:
 s1 = GCATCGA
 s2 = GCATCGA
```

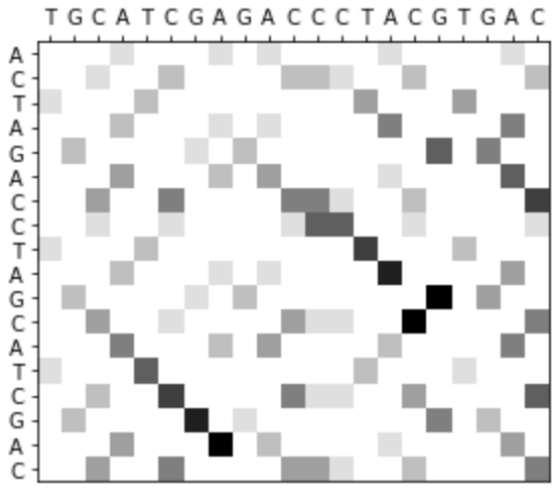


I also tested using other values for match, gap_pentalty, and mismatch_penalty:

```
match = 10, gap_penalty = 3, mismatch_pentalty = 7:
Longest matching score: 77.0
Matching sequences:
 s1 = TAGCATCGA
 s2 = T-GCATCGA
```



```
match = 1, gap_penalty = 0, mismatch_pentalty = 1
Longest matching score: 8.0
Matching sequences:
 s1 = AGA-CCTA-G
 s2 = AGACCCTACG
```



These tests show that my functions works because:

I. The first test (default values) have the same output as example test in assignment manual

II. The second test (match = 10, gap = 3, mismatch = 7) showed a score of 77 which is correct because there are 8 matches so running score = 80, 0 mismatches so running score = 80, and 1 gap so final score = 80 - 3 = 77. This is clearly better than the other choice, which has 8 matches, but two gaps and 0 mismatches, which gives a final score of 80-3*2 = 74.

III. The third test (match = 1, gap = 0, mismatch = 1) showed a score of 8 which is correct because there are again 8 matches (score = 8), 0 mismatches (score=8). Note the presence of gaps does not affect the score. This does not differentiate between the three potential sequences of 8 matches (no mismatches).

References:

https://simplemaps.com/data/world-cities
https://stackoverflow.com/questions/4913349/haversine-formula-in-python-bearing-and-distance-between-two-gps-points\
https://kanoki.org/2020/08/30/matplotlib-scatter-plot-color-by-category-in-python/

```
# import libraries
import pandas as pd
import numpy as np
import requests
```

```
# function to query API given in homework
def query_api(a, b):
    return float(requests.get(f"http://ramcdougal.com/cgi-bin/error_function.py?a={a}&b={b}", headers={"U
```

```
# 2D gradient descent algorithm

def gradient_descent(a, b, ss, stop_thresh, delta, prev):
    # iterate and increment a, b values to find minimum
    while(abs(query_api(a, b) - prev) > stop_thresh):
        # save current function value
        prev = query_api(a, b)
        # calculate gradient
        delta_a = (query_api(a+delta, b) - query_api(a, b))/delta
        delta_b = (query_api(a, b+delta) - query_api(a, b))/delta
        grad_vec = np.array([delta_a, delta_b])
        # increment a and b in the direction of the gradient
        a -= ss*delta_a
        b -= ss*delta_b
    return [a,b, prev]
```

```
# test function
# define delta of a and b
delta = 1e-4
# if function value varies < stop_thresh, then stop iterations and consider minimum found
stop_thresh = 0.0001
# define step size
ss = 0.25
# initialize values
a = 0.4
b = 0.8
prev = 9999
print("Testing gradient descent algorithm, a = 0.4, b = 0.8...")
print(gradient_descent(a, b, ss, stop_thresh, delta, prev))
```

```
Testing gradient descent algorithm, a = 0.4, b = 0.8...
[0.21882577500036382, 0.6906851500002358, 1.10004420658]
```

```
# find local minimum and global minimum using gradient descent function above
# doing do by separating a, b into ranges for testing
def minima(a_inputs, b_inputs):
    results = {}
    for i in range(len(a_inputs)):
         gd_ret = gradient_descent(float(a_inputs[i]), float(b_inputs[i]), 0.25, 0.0001, 1e-4, 9999)
         func_val = gd_ret[2]
         a_val = gd_ret[0]
         b_val = gd_ret[1]
         results[func_val] = [a_val, b_val]
    sorted_func_vals = sorted(results.keys())
    global_min = sorted_func_vals[0]
    global_min_vals = results.get(global_min)
    local_min = sorted_func_vals[1]
    local_min_vals = results.get(local_min)
```

```
        print("Global minimum is:", global_min, "at", global_min_vals)
        print("Local minimum is:", local_min, "at", local_min_vals)


# test function
a_inputs = [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9]
b_inputs = [0.5, 0.6, 0.3, 0.8, 0.7, 0.1, 0.2, 0.9, 0.4]
print(minima(a_inputs, b_inputs))
```

```
Global minimum is: 1.00004896388 at [0.7102008000009036, 0.16787267500030226]
Local minimum is: 1.00005261742 at [0.7112031249999144, 0.17089062500010838]
None
```

3
```
# import libraries
import pandas as pd
import cartopy.crs as ccrs
import matplotlib.pyplot as plt
from matplotlib.pyplot import cm
import plotnine as p9
import random
import numpy as np
from math import radians, cos, sin, asin, sqrt
```

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)

~\AppData\Local\Temp/ipykernel_7400/31984956.py in <module>
      1 # import libraries
      2 import pandas as pd
----> 3 import cartopy.crs as ccrs
      4 import matplotlib.pyplot as plt
      5 from matplotlib.pyplot import cm

ModuleNotFoundError: No module named 'cartopy'
```

7
```
# import data
df_cities = pd.read_csv("worldcities.csv")
df_cities = df_cities[['city', 'lat', 'lng']]
df_cities.head()
```

7

	city	lat	lng
0	Tokyo	35.6897	139.6922
1	Jakarta	-6.2146	106.8451
2	Delhi	28.6600	77.2300
3	Mumbai	18.9667	72.8333
4	Manila	14.6000	120.9833

4
```
# function as given in class but modified
# helper function to normalize
def normalize(series):
    return (series - series.mean()) / series.std()

# helper function to calculate distance, using haversine function link given in assignment manual
def haversine(lon1, lat1, lon2, lat2):
    # convert decimal degrees to radians
    lon1, lat1, lon2, lat2 = map(radians, [lon1, lat1, lon2, lat2])

    # haversine formula
    dlon = lon2 - lon1
    dlat = lat2 - lat1
    a = sin(dlat/2)**2 + cos(lat1) * cos(lat2) * sin(dlon/2)**2
    c = 2 * asin(sqrt(a))
    r = 6371 # Radius of earth in kilometers. Use 3956 for miles. Determines return value units.
    return c * r

# k-means function and plot
```

```
def k_means(k, df):
    pts = [np.array(pt) for pt in zip(df['lng'], df['lat'])]
    centers = random.sample(pts, k)
    old_cluster_ids, cluster_ids = None, []
    while cluster_ids != old_cluster_ids:
        old_cluster_ids = list(cluster_ids)
        cluster_ids = []
        for pt in pts:
            min_cluster = -1
            min_dist = float('inf')
            for i, center in enumerate(centers):
                dist = np.linalg.norm(haversine(pt[0], pt[1], center[0], center[1]))
                if dist < min_dist:
                    min_cluster = i
                    min_dist = dist
            cluster_ids.append(min_cluster)
        df['cluster'] = cluster_ids
        cluster_pts = [[pt for pt, cluster in zip(pts, cluster_ids) if cluster == match]
                       for match in range(k)]
        centers = [sum(pts)/len(pts) for pts in cluster_pts]

    # map cluster to color
    cluster_set = set(df["cluster"])
    colors = {}
    color = ['black', 'red', 'white', 'green', 'tan', 'orange', 'brown', 'blue', 'cyan', 'aqua', 'indigo'
    c = 0
    for i in cluster_set:
        colors[i] = color[c]
        c += 1

    # plot figure with code given in assignment manual
    fig = plt.figure()
    ax = fig.add_subplot(1, 1, 1, projection=ccrs.Robinson())
    lats = df['lat']
    lngs = df['lng']
    ax.coastlines()
    ax.plot(lngs, lats, "o", color=df['cluster'].map(colors), transform=ccrs.PlateCarree())
    ax.set_extent([-180, 180, -90, 90], crs=ccrs.PlateCarree())
    plt.savefig("Q2.png")
    plt.show()

    # (p9.ggplot(df, p9.aes(x="lng", y="lat", color="cluster")) + p9.geom_point()).draw()
```

```
30 # test function
   print("Testing k_means function with k = 5...")
   print(k_means(5, df_cities))
```

```
Testing k_means function with k = 5...

---------------------------------------------------------------------------

NameError                                 Traceback (most recent call last)

~\AppData\Local\Temp/ipykernel_8124/1381800187.py in <module>
      1 # test function
      2 print("Testing k_means function with k = 5...")
----> 3 print(k_means(5, df_cities))
      4

~\AppData\Local\Temp/ipykernel_8124/2691505285.py in k_means(k, df)
     41     # plot figure with code given in assignment manual
     42     fig = plt.figure()
---> 43     ax = fig.add_subplot(1, 1, 1, projection=ccrs.Robinson())
```

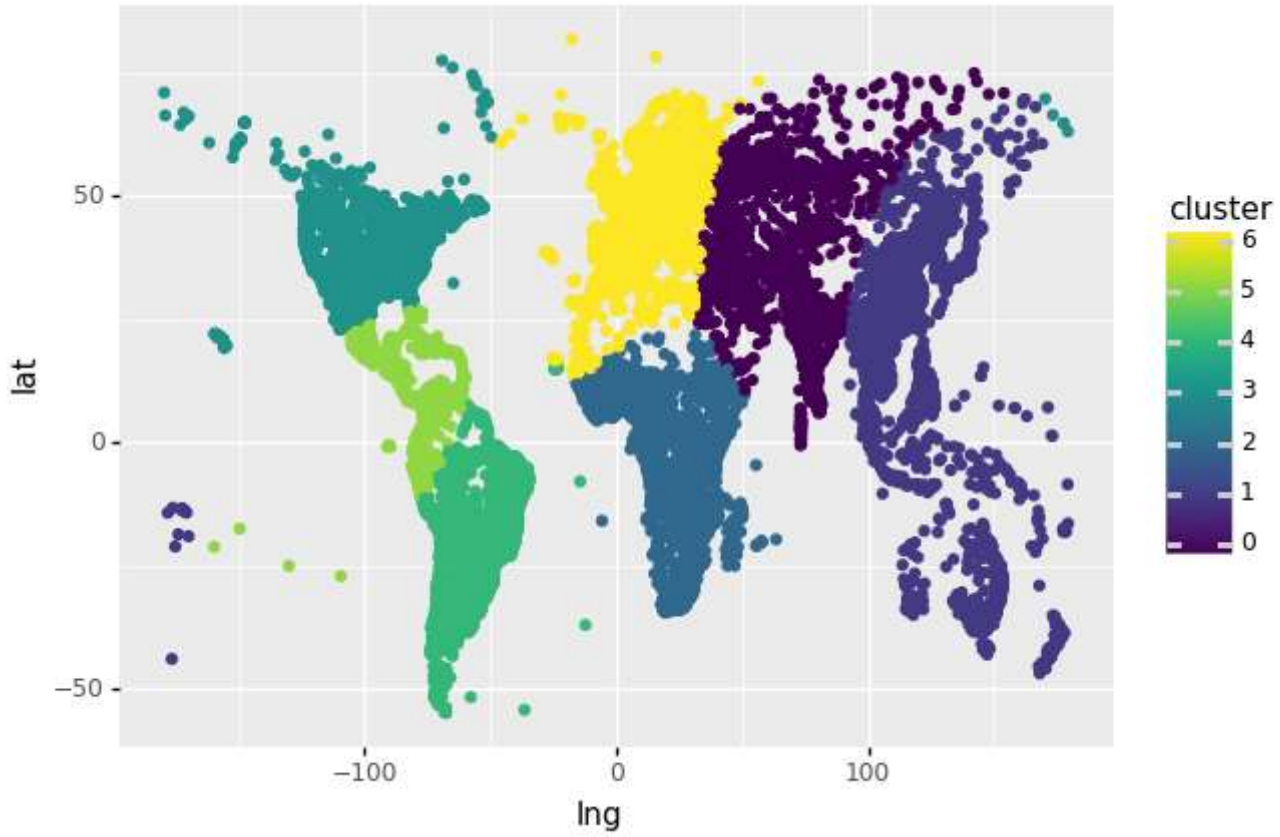
```
44      lats = df['lat']
45      lngs = df['lng']
```

```
NameError: name 'ccrs' is not defined
```

```
<Figure size 432x288 with 0 Axes>
```

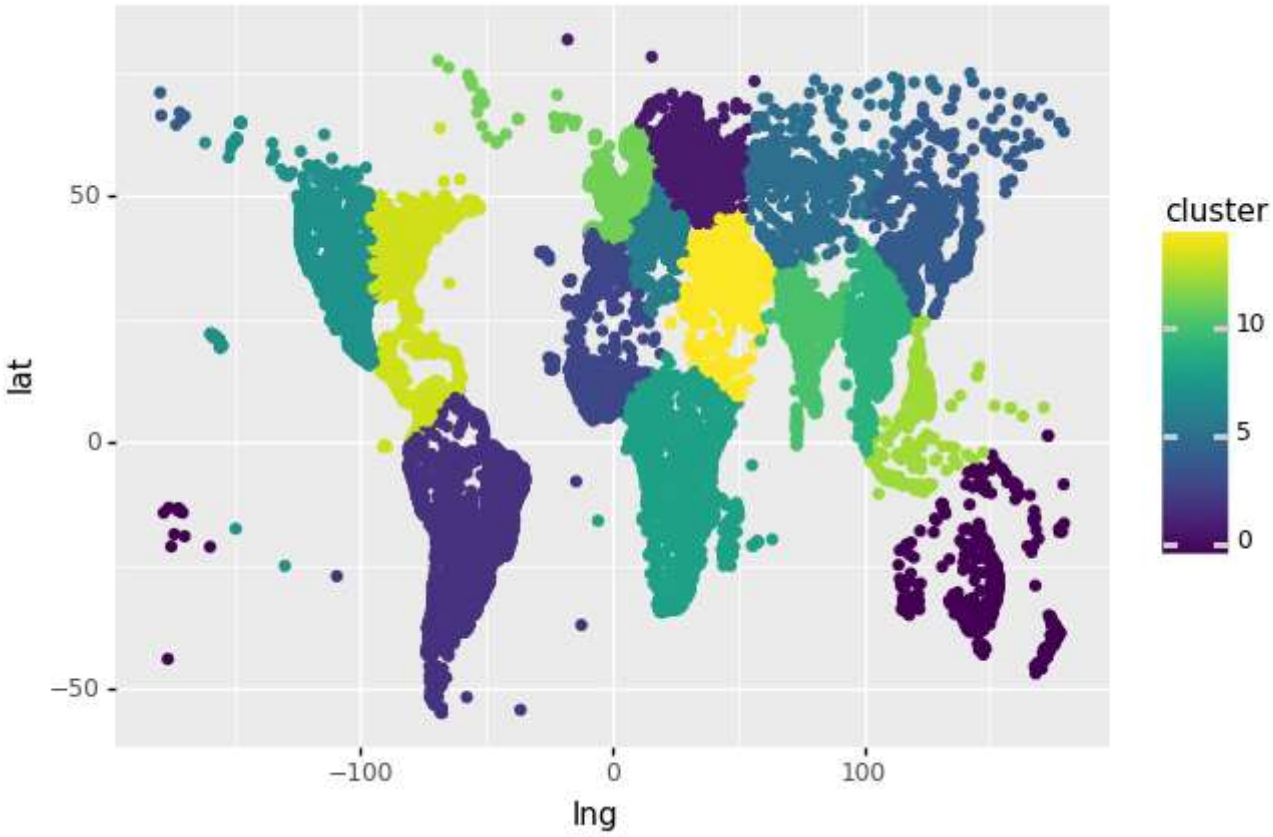
10
```
print("Testing k_means function with k = 7...")
print(k_means(7, df_cities))
```

```
Testing k_means function with k = 7...
None
```



11
```
print("Testing k_means function with k = 15...")
print(k_means(15, df_cities))
```

```
Testing k_means function with k = 15...
None
```


lat
50
0
−50
−100
0
100
lng
cluster
10
5
0

```
import pandas as pd
import time
import seaborn as sns
from plotnine import ggplot
```

```
# non-lru_cache fibonacci implementation
def fibonacci(n):
    if n == 1 or n == 2: return 1
    return fibonacci(n-1) + fibonacci(n-2)
```

```
# test function
print("Testing...")
print("n = 2, should be 1:", fibonacci(2))
print("n = 3, should be 2:", fibonacci(3))
print("n = 10, should be 55:", fibonacci(10))
```

```
Testing...
n = 2, should be 1: 1
n = 3, should be 2: 2
n = 10, should be 55: 55
```

```
# lru_cache fibonacci implementation
def lru_fibonacci(n, d):
    if n == 1 or n == 2: return 1
    if n not in d: d[n] = lru_fibonacci(n-1, d) + lru_fibonacci(n-2, d)
    return d[n]
```

```
# test function
# add realization about could do d={} after consulting slides in readme
print("Testing...")
d = {}
print("n = 2, should be 1:", lru_fibonacci(2, d))
print("n = 3, should be 2:", lru_fibonacci(3, d))
print("n = 10, should be 55:", lru_fibonacci(10, d))
```

```
Testing...
n = 2, should be 1: 1
n = 3, should be 2: 2
n = 10, should be 55: 55
```

```
# reimplementing above two functions to reflect runtime as a function of n
time_fibo = {}
time_lru = {}

def timed_fibonacci(n):
    start = time.time()
    ret = fibonacci(n)
    time_fibo[n] = time.time() - start
    return ret

def timed_lru_fibonacci(n, d):
    start = time.time()
    ret = lru_fibonacci(n, d)
    time_lru[n] = time.time() - start
    return ret
```

```
# getting some data from both algorithms
```

```
timed_fibonacci(2)
timed_fibonacci(10)
timed_fibonacci(30)
timed_fibonacci(40)
timed_fibonacci(50)
# timed_fibonacci(70)
# timed_fibonacci(90)
# timed_fibonacci(100)
print(time_fibo)
```

```
{2: 0.0, 10: 0.0, 30: 0.10372114181518555, 50: 1510.9152822494507, 40: 12.661979675292969}
```

81
```
# add one more entry to fib
timed_fibonacci(45)
print(time_fibo)
```

```
{2: 0.0, 10: 0.0, 30: 0.10372114181518555, 50: 1510.9152822494507, 40: 12.661979675292969, 45: 139.336616
```

112
```
timed_lru_fibonacci(2, d={})
timed_lru_fibonacci(10, d={})
timed_lru_fibonacci(30, d={})
timed_lru_fibonacci(40, d={})
timed_lru_fibonacci(45, d={})
timed_lru_fibonacci(50, d={})
timed_lru_fibonacci(70, d={})
timed_lru_fibonacci(90, d={})
timed_lru_fibonacci(100, d={})
timed_lru_fibonacci(200, d={})
timed_lru_fibonacci(300, d={})
timed_lru_fibonacci(500, d={})
timed_lru_fibonacci(700, d={})
timed_lru_fibonacci(900, d={})
timed_lru_fibonacci(1000, d={})
print(time_lru)
```

```
{2: 0.0, 10: 0.0, 30: 0.0, 50: 0.0, 70: 0.0, 90: 0.0, 100: 0.0, 200: 0.0, 300: 0.0, 500: 0.0, 700: 0.0, 9
```

85
```
# organize dictionaries of runtime into dataframes
df_fib = pd.DataFrame.from_dict(data=time_fibo, orient='index').reset_index()
df_fib.columns=['n', 'seconds']
df_fib.head(6)
```

85

	n	seconds
0	2	0.000000
1	10	0.000000
2	30	0.103721
3	50	1510.915282
4	40	12.661980
5	45	139.336617

106
```
df_lru = pd.DataFrame.from_dict(data=time_lru, orient='index').reset_index()
df_lru.columns=['n', 'seconds']
df_lru.head(15)
```

106

	n	seconds
0	2	0.000000
1	10	0.000000
2	30	0.000000
3	50	0.000000
4	70	0.000000
5	90	0.000000
6	100	0.000000
7	200	0.000000
8	300	0.000000
9	500	0.000000
10	700	0.000993
11	900	0.000000
12	1000	0.000999
13	40	0.000000
14	45	0.000000

107

```
# merge dataframes
df_runtime = df_fib.merge(df_lru, on='n', how='outer')
df_runtime.columns = ['n', 'fib_seconds', 'lru_seconds']

# sort by the n values
df_runtime.sort_values(by=['n'])
df_runtime.head(15)
```

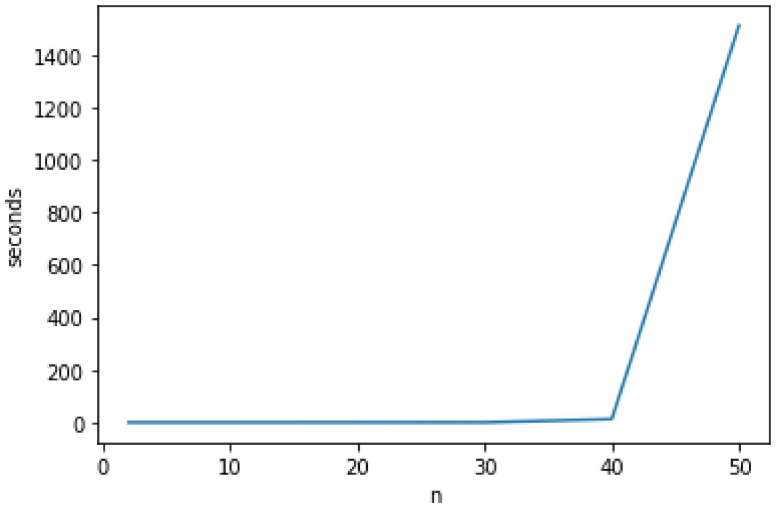
107

	n	fib_seconds	lru_seconds
0	2	0.000000	0.000000
1	10	0.000000	0.000000
2	30	0.103721	0.000000
3	50	1510.915282	0.000000
4	40	12.661980	0.000000
5	45	139.336617	0.000000
6	70	NaN	0.000000
7	90	NaN	0.000000
8	100	NaN	0.000000

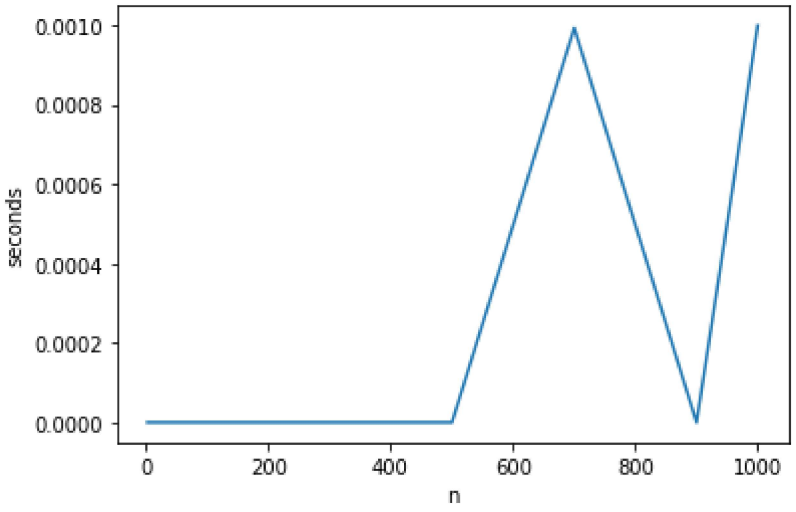
	n	fib_seconds	lru_seconds
9	200	NaN	0.000000
10	300	NaN	0.000000
11	500	NaN	0.000000
12	700	NaN	0.000993
13	900	NaN	0.000000
14	1000	NaN	0.000999

80
```
# plot runtime graph for fib
time_fib_plt = sns.lineplot(data=df_fib, x='n', y='seconds')
time_fib_plt
```

80
```
<AxesSubplot:xlabel='n', ylabel='seconds'>
```



115
```
# plot runtime graph for lru_fib
time_lru_plot = sns.lineplot(data=df_lru, x='n', y='seconds')
```

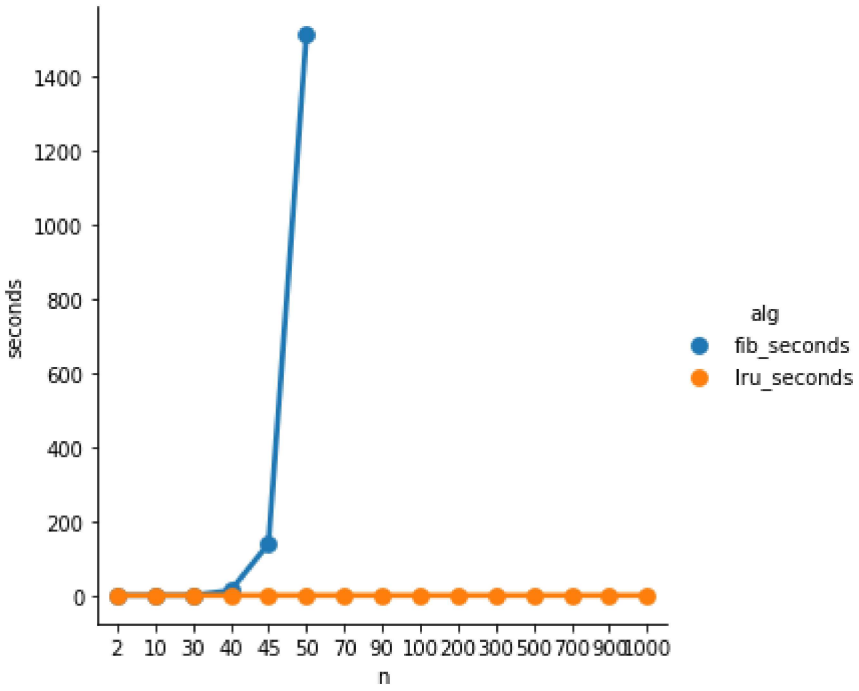


116
```
# plot two graphs together using same scale for y axis
```

```
df_melted = df_runtime.melt('n', var_name='alg',  value_name='seconds')
time_plt = sns.factorplot(x="n", y="seconds", hue='alg', data=df_melted)
time_plt
```

```
C:\Users\olina\AppData\Local\Programs\Python\Python39\lib\site-packages\seaborn\categorical.py:3717: User
```

116 `<seaborn.axisgrid.FacetGrid at 0x1ed33f25c40>`



```
# import libraries
import matplotlib.pyplot as plt
import numpy as np
```

```
# modifying code given in class
def smith_waterman(seq1, seq2, match, gap_penalty, mismatch_penalty):
    # initialize grid
    data = np.zeros((len(seq1), len(seq2)))
    # store max length and coordinates
    max_len = 0
    max_i = 0
    max_j = 0
    # enumerate over sequences
    for i, base1 in enumerate(seq1):
        for j, base2 in enumerate(seq2):
            # if the two chars are identical
            if base1 == base2:
                # if square is already filled in due to gaps in previous enumerations
                # keep the larger number
                # this also takes care of the horizontal gap scenario
                # because every char in seq1 are compared with all chars in seq2 sequentially
                # and max match length is taken
                # data[i, j] = current square value
                # data[i-1, j-1] + match = value diagonally before
                # data[i-1, j-2] + match - gap_penalty= value with horizontal gap (down 1, right 2)
                if i > 0 and j > 1:
                    data[i,j] = max(data[i,j], data[i-1,j-1] + match, data[i-1, j-2]+match-gap_penalty)
                elif i > 0 and j > 0:
                    data[i,j] = max(data[i,j], data[i-1,j-1] + match)
                else:
                    data[i,j] = match
            # if there is a gap vertically
            elif i < len(seq1)-1:
                if seq1[i+1] == base2:
                    if i > 0 and j > 0:
                        # take maximum between its current value and (match-gap_penalty + (value at up 2,
                        data[i+1,j] = max(data[i+1,j], data[i-1,j-1]+match-gap_penalty)
                    else:
                        data[i+1,j] += match-gap_penalty
            # if mismatch
            else: data[i,j] = max(0, data[i,j] - mismatch_penalty)

            # update longest sequence
            if data[i,j] > max_len:
                max_len = data[i,j]
                max_i = i
                max_j = j

    # printing longest matching sequence
    ret_1 = ""
    ret_2 = ""
    # defining the number of iterations
    # cannot use max_len because it actually refers to score, not length
    max_iter = 0
    if max_len % match == 0: max_iter = max_len/match
    else: max_iter = max_len//match + 1

    for i in range(int(max_iter)):
      # match
      if seq1[max_i] == seq2[max_j]:
        ret_1 += seq1[max_i]
        ret_2 += seq2[max_j]
```

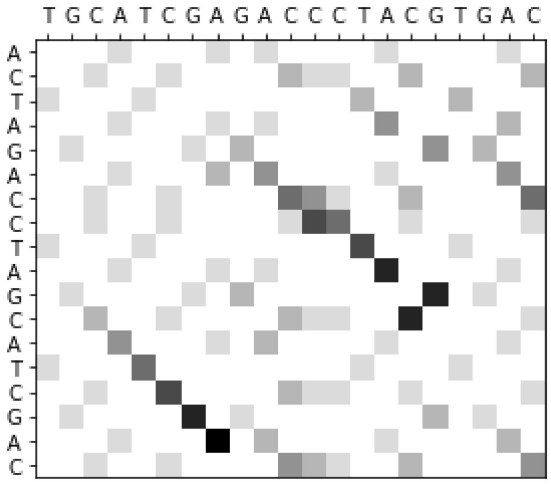
```
      # horitzontal gap
      elif seq1[max_i] == seq2[max_j-1]:
        ret_1 += "-" + seq1[max_i]
        ret_2 += seq2[max_j] + seq2[max_j-1]
        max_j -= 1
      # vertical gap
      elif seq1[max_i-1] == seq2[max_j]:
        ret_1 += seq1[max_i] + seq1[max_i-1]
        ret_2 += "-" + seq2[max_j]
        max_i -= 1
      max_i -= 1
      max_j -= 1

    # output
    print("Longest matching score:", max_len)
    # reverse strings and print
    print("Matching sequences: \n s1 =", ret_1[::-1], "\n s2 =", ret_2[::-1])

    # plot graph
    plt.xticks(range(len(seq2)), labels=seq2)
    plt.yticks(range(len(seq1)), labels=seq1)
    plt.imshow(data, interpolation='nearest',
               cmap='binary')
    plt.gca().xaxis.tick_top()
```

163
```
# test function
print("match = gap_penalty = mismatch_pentalty = 1:")
smith_waterman('ACTAGACCTAGCATCGAC', 'TGCATCGAGACCCTACGTGAC', 1, 1, 1)
plt.show()
```

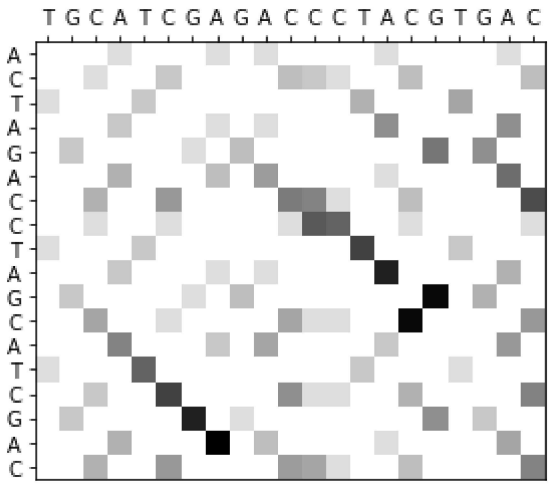
```
match = gap_penalty = mismatch_pentalty = 1:
Longest matching score: 7.0
Matching sequences:
 s1 = GCATCGA
 s2 = GCATCGA
```



164
```
# test function
print("match = 10, gap_penalty = 3, mismatch_pentalty = 7:")
smith_waterman('ACTAGACCTAGCATCGAC', 'TGCATCGAGACCCTACGTGAC', 10, 3, 7)
plt.show()
```

```
match = 10, gap_penalty = 3, mismatch_pentalty = 7:
Longest matching score: 77.0
Matching sequences:
```

```
s1 = TAGCATCGA
s2 = T-GCATCGA
```




```
# test function
print("match = 1, gap_penalty = 0, mismatch_pentalty = 1")
smith_waterman('ACTAGACCTAGCATCGAC', 'TGCATCGAGACCCTACGTGAC', 1, 0, 1)
plt.show()
```

```
match = 1, gap_penalty = 0, mismatch_pentalty = 1
Longest matching score: 8.0
Matching sequences:
 s1 = AGA-CCTA-G
 s2 = AGACCCTACG
```