BIS634 Asssignment 4

Question 1

Tahir Manuel D'Mello

In [1]:
```
import requests
import json
import time
```

Implement a two-dimensional version of the gradient descent algorithm to find optimal choices of a and b. **(7 points)**

In [2]:
```
def func(a, b):

    query = f'http://ramcdougal.com/cgi-bin/error_function.py?a={a}&b={b}'
    f = float(requests.get(query, headers={"User-Agent": "MyScript"}).text)

    return f

func(0.4, 0.2)
```

Out[2]:
```
1.294915
```

In [3]:
```
def gradient_descent(a_initial, b_initial, stopping, h, gamma):

    a_old = a_initial
    b_old = b_initial
    f_old = func(a_old, b_old)

    fprime_a = ( func(a_old + h, b_old) - f_old ) / h
    fprime_b = ( func(a_old, b_old + h) - f_old ) / h

    a_next = a_old - gamma*fprime_a
    b_next = b_old - gamma*fprime_b

    f_next = func(a_next, b_next)

    #print("Function is", f_next)

    while abs(f_old -  f_next) > stopping:
        a_old = a_next
        b_old = b_next
        f_old = f_next

        fprime_a = ( func(a_old + h, b_old) - f_old ) / h
        fprime_b = ( func(a_old, b_old + h) - f_old ) / h

        a_next = a_old - gamma*fprime_a
        b_next = b_old - gamma*fprime_b
        f_next = func(a_next, b_next)

        #print("Function is", f_next)

    return a_next, b_next, f_next
```

Explain how you estimate the gradient given that you cannot directly compute the derivative. **(3 points)**

The gradients for both a and b were computed using the following mathematical formulae:

$$\frac{\partial f}{\partial x}(a,b) = \lim_{h \to 0} \frac{f(a+h,b) - f(a,b)}{h}$$

$$\frac{\partial f}{\partial y}(a,b) = \lim_{h \to 0} \frac{f(a,b+h) - f(a,b)}{h}$$

identify any numerical choices -- including but not limited to stopping criteria -- you made **(3 points)**, and justify why you think they were reasonable choices **(3 points)**.

There were 3 numerical choices made:

1. Stopping criteria - Iterations were stopped when the difference between the minima picked in consecutive iterations didn't differ by more than a picked value (0.000001 in this case). This was a reasonable ned point to aim for.
2. Gradient displacement h - This was a small value chosen to compute the mathematical value of the gradient being calculated. Value chosen was 0.0001 as this ensured a reasonable estimate of the gradient.

3. Gamma - This determines the size of the step. An appropriate value of the stepsize (0.1) was chosen to ensure that the minima is reached without overtaking the point while still keeping the number of iterations needed down.

Find both locations (i.e. a, b values) querying the API as needed **(5 points)** and identify which corresponds to which (local minimum and global minimum). **(2 point)**

In [5]:
```
a, b, f = gradient_descent(0.4, 0.2, 0.000001, 0.0001, 0.1)
print('Optimal a is at' , a)
print('Optimal b is at' , b)
print("Minima of function is", f)
```

```
Optimal a is at 0.7117455600003538
Optimal b is at 0.16897035000009702
Minima of function is 1.00000019686
```

This is the global minima.

In [6]:
```
a, b, f = gradient_descent(0.7, 0.8, 0.000001, 0.0001, 0.1)
print('Optimal a is at' , a)
print('Optimal b is at' , b)
print("Minima of function is", f)
```

```
Optimal a is at 0.2171203699999562
Optimal b is at 0.689218500000343
Minima of function is 1.10000130297
```

This is the local minima.

Briefly discuss how you would have tested for local vs global minima if you had not known how many minima there were. **(2 points)**

A fee methods to test for local vs gobal minima in the gradient decent algorithm:

1. Vary initial guesses across the set of input values possible.
2. Vary the learning rate (varying gamma in this case) for various input values to ensure that the step size doesn't overshoot any minima.
3. Other techniques like stochastic gradient descent can be implemented (randomly without replacement selects one direction to perform gradient descent per iteration) which give us more directions to move in and can help escape local minima.

Then at the end, the smallest value found is (probably, if done correctly) the glocal minima and the rest are all local minima.

Question 2

Tahir Manuel D'Mello

In [1]:
```
import pandas as pd
import plotnine as p9
import random
import numpy as np
import time
import warnings
from math import radians, cos, sin, asin, sqrt
import cartopy.crs as ccrs
import matplotlib.pyplot as plt
```

In [2]:
```
#Taken from StackOverflow Link provided in assignment and modified a little

def haversine(lat1, lon1, lat2, lon2):
    """
    Calculate the great circle distance in kilometers between two points
    on the earth (specified in decimal degrees)
    """
    # 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
```

In [3]:
```
data_full = pd.read_csv("worldcities.csv")
```

In [4]:
```
data_full
```

Out[4]:

	city	city_ascii	lat	lng	country	iso2	iso3	admin_name	capital	population	
0	Tokyo	Tokyo	35.6839	139.7744	Japan	JP	JPN	Tōkyō	primary	39105000.0	1
1	Jakarta	Jakarta	-6.2146	106.8451	Indonesia	ID	IDN	Jakarta	primary	35362000.0	1
2	Delhi	Delhi	28.6667	77.2167	India	IN	IND	Delhi	admin	31870000.0	1
3	Manila	Manila	14.6000	120.9833	Philippines	PH	PHL	Manila	primary	23971000.0	1
4	São Paulo	Sao Paulo	-23.5504	-46.6339	Brazil	BR	BRA	São Paulo	admin	22495000.0	1
...	...	...	...	...	...	...	...	...	...	...	
42900	Tukchi	Tukchi	57.3670	139.5000	Russia	RU	RUS	Khabarovskiy Kray	NaN	10.0	1
42901	Numto	Numto	63.6667	71.3333	Russia	RU	RUS	Khanty-Mansiyskiy Avtonomnyy Okrug-Yugra	NaN	10.0	1
42902	Nord	Nord	81.7166	-17.8000	Greenland	GL	GRL	Sermersooq	NaN	10.0	1
42903	Timmiarmiut	Timmiarmiut	62.5333	-42.2167	Greenland	GL	GRL	Kujalleq	NaN	10.0	1
42904	Nordvik	Nordvik	74.0165	111.5100	Russia	RU	RUS	Krasnoyarskiy Kray	NaN	0.0	1

42905 rows × 11 columns

In [5]:
```
df = data_full[['city', 'lat', 'lng']]
df
```

Out[5]:

	city	lat	lng
0	Tokyo	35.6839	139.7744
1	Jakarta	-6.2146	106.8451
2	Delhi	28.6667	77.2167
3	Manila	14.6000	120.9833
4	São Paulo	-23.5504	-46.6339
...	...	...	...
42900	Tukchi	57.3670	139.5000
42901	Numto	63.6667	71.3333
42902	Nord	81.7166	-17.8000
42903	Timmiarmiut	62.5333	-42.2167
42904	Nordvik	74.0165	111.5100

42905 rows × 3 columns

Modify the k-means code (or write your own) from slides8 to use the Haversine metric and work with our dataset. **(5 points)**

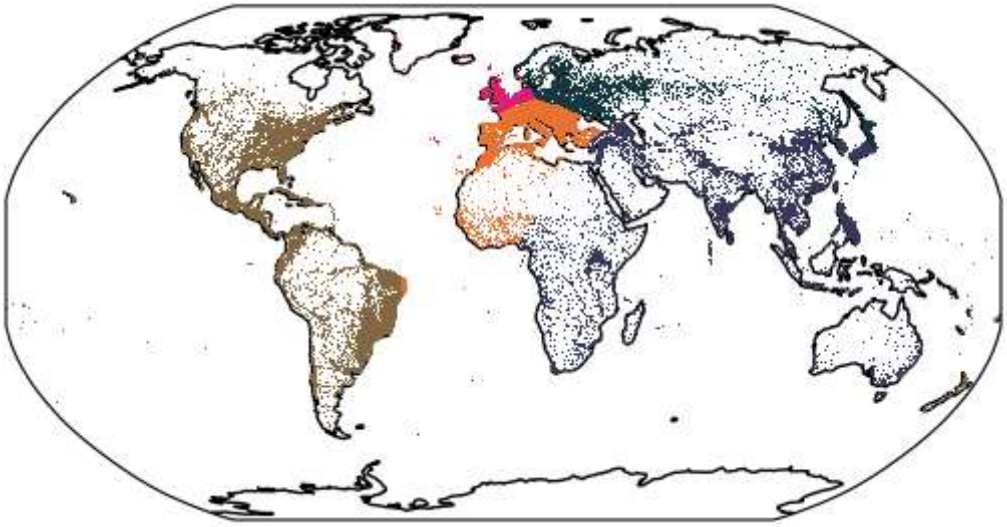
In [6]:
```
def k_means_haversine(k, df, plot_d):
```

```
    pts = [np.array(pt) for pt in zip(df['lat'], df['lng'])]

    centers = random.sample(pts, k)
    old_cluster_ids, cluster_ids = None, [] # arbitrary but different


    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 = haversine(pt[0], pt[1], center[0], center[1])

                if dist < min_dist:
                    min_cluster = i
                    min_dist = dist

            cluster_ids.append(min_cluster)

        cluster_pts = [[pt for pt, cluster in zip(pts, cluster_ids) if cluster == match] for mat

        centroids = [sum(pts)/len(pts) for pts in cluster_pts]

    df_out = df.assign(cluster=cluster_ids)

    if plot_d == True:
        fig = plt.figure()
        ax = fig.add_subplot(1, 1, 1, projection=ccrs.Robinson())
        ax.coastlines()

        for i in range(k):
            lats = df_out[df_out['cluster'] == i]['lat']
            lngs = df_out[df_out['cluster'] == i]['lng']
            col = (np.random.random(), np.random.random(), np.random.random())
            ax.plot(lngs, lats, ",", color=col, transform=ccrs.PlateCarree())

        ax.set_extent([-180, 180, -90, 90], crs=ccrs.PlateCarree())
        plt.show()

    return df_out
```

Visualize your results with a color-coded scatter plot. **(5 points)**
Be sure to use an appropriate map projection (i.e. do not simply make x=longitude and y=latitude). **(5 points)**

In [7]:
```
df_out = k_means_haversine(5, df, True)
df_out
```

```
C:\Users\tahir\anaconda3\lib\site-packages\cartopy\crs.py:245: ShapelyDeprecationWarning: __len_
_ for multi-part geometries is deprecated and will be removed in Shapely 2.0. Check the length o
f the `geoms` property instead to get the  number of parts of a multi-part geometry.
C:\Users\tahir\anaconda3\lib\site-packages\cartopy\crs.py:297: ShapelyDeprecationWarning: Iterat
ion over multi-part geometries is deprecated and will be removed in Shapely 2.0. Use the `geoms`
property to access the constituent parts of a multi-part geometry.
C:\Users\tahir\anaconda3\lib\site-packages\cartopy\crs.py:364: ShapelyDeprecationWarning: __len_
_ for multi-part geometries is deprecated and will be removed in Shapely 2.0. Check the length o
f the `geoms` property instead to get the  number of parts of a multi-part geometry.
```



Out[7]:

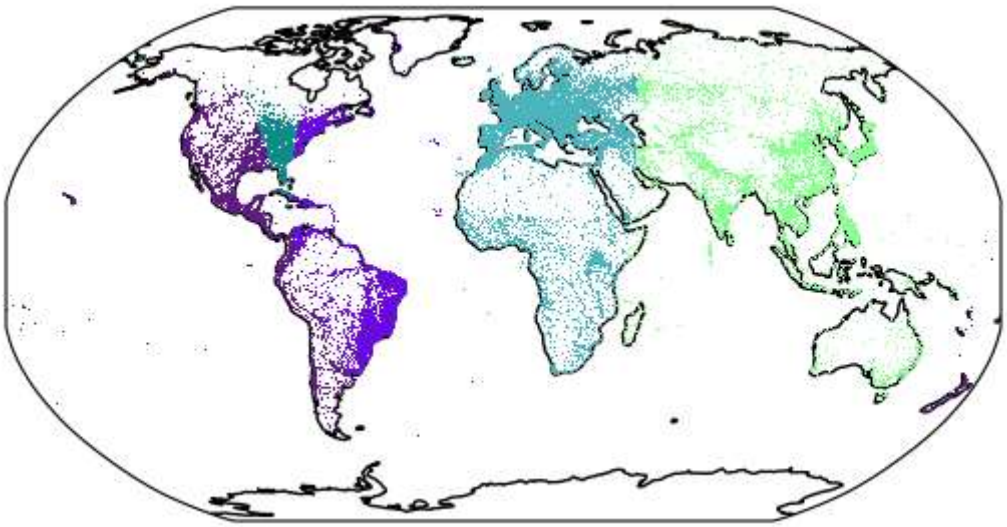
	city	lat	lng	cluster
0	Tokyo	35.6839	139.7744	0
1	Jakarta	-6.2146	106.8451	0
2	Delhi	28.6667	77.2167	0
3	Manila	14.6000	120.9833	0
4	São Paulo	-23.5504	-46.6339	1
...	...	...	...	...
42900	Tukchi	57.3670	139.5000	4
42901	Numto	63.6667	71.3333	4
42902	Nord	81.7166	-17.8000	4
42903	Timmiarmiut	62.5333	-42.2167	2
42904	Nordvik	74.0165	111.5100	4

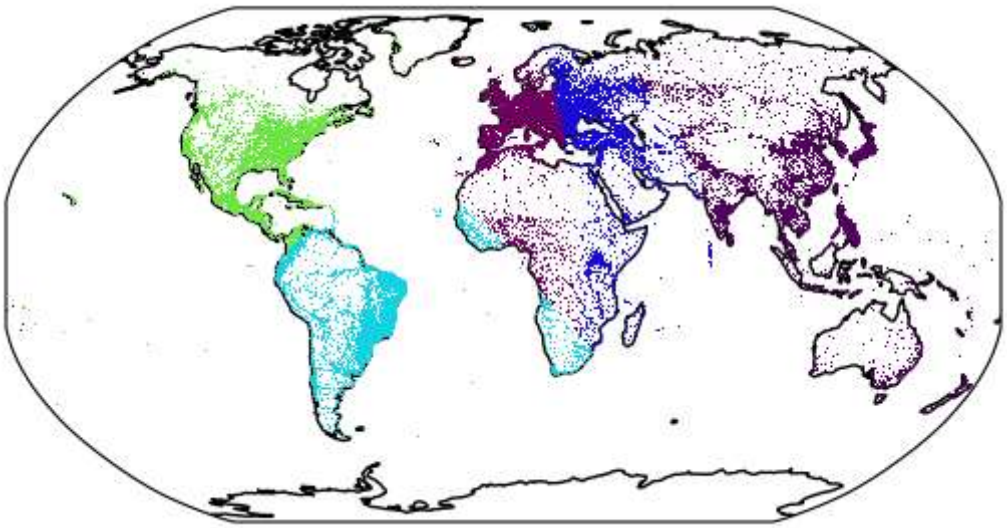
42905 rows × 4 columns

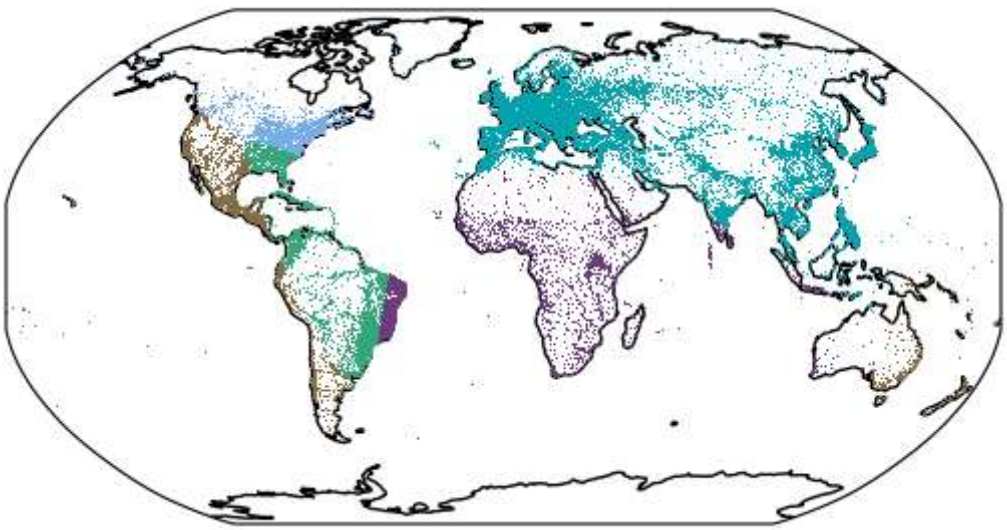
Use this algorithm to cluster the cities data for k=5, 7, and 15. Run it several times to get a sense of the variation of clusters for each k (share your plots). **(5 points)**

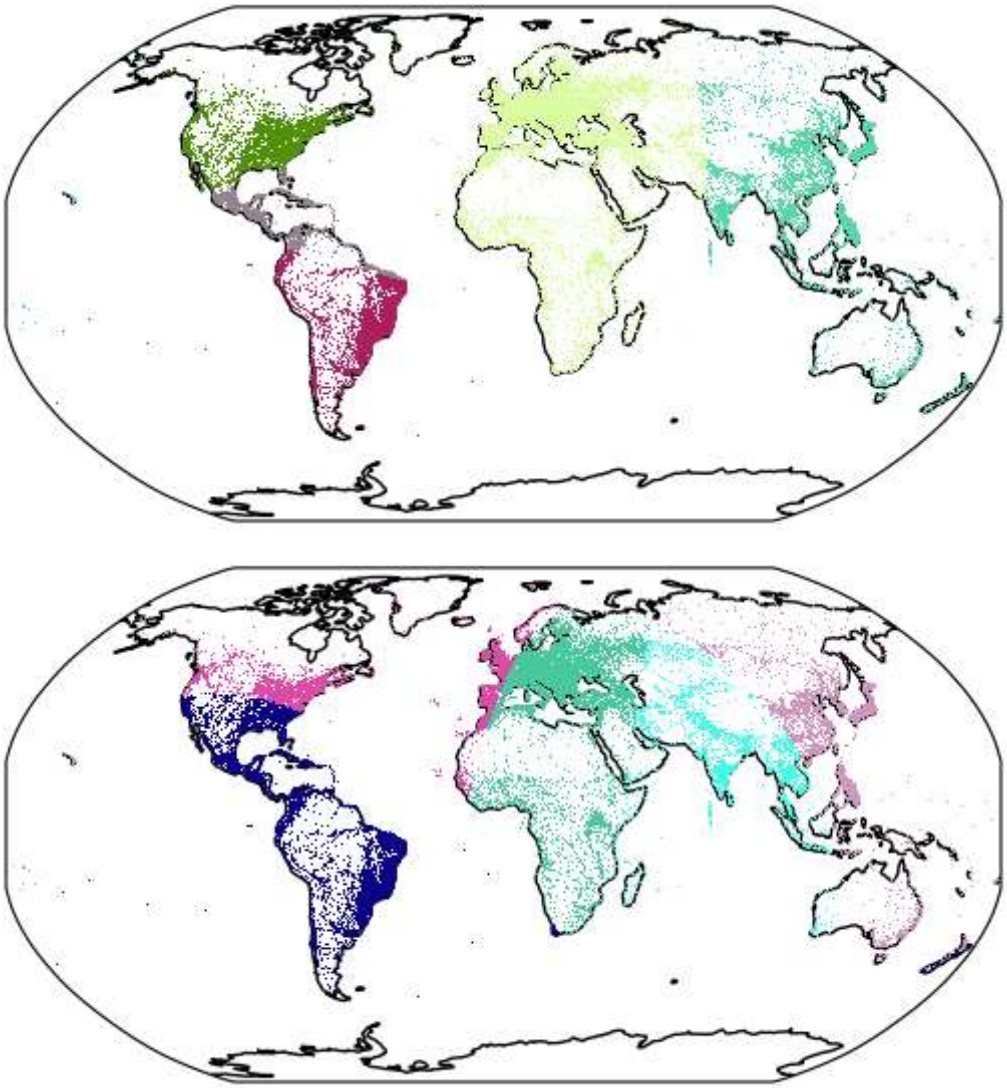
In [8]:
```
time_set_5 = []
warnings.filterwarnings("ignore")

for i in range(5):
    t_start = time.perf_counter()

    df_out = k_means_haversine(5, df, True)

    t_end = time.perf_counter()
```

```
    time_set_5.append(t_end-t_start)

print("The average run time for k = 5 clustering and plotting is", np.mean(np.array(time_set_5)))
```



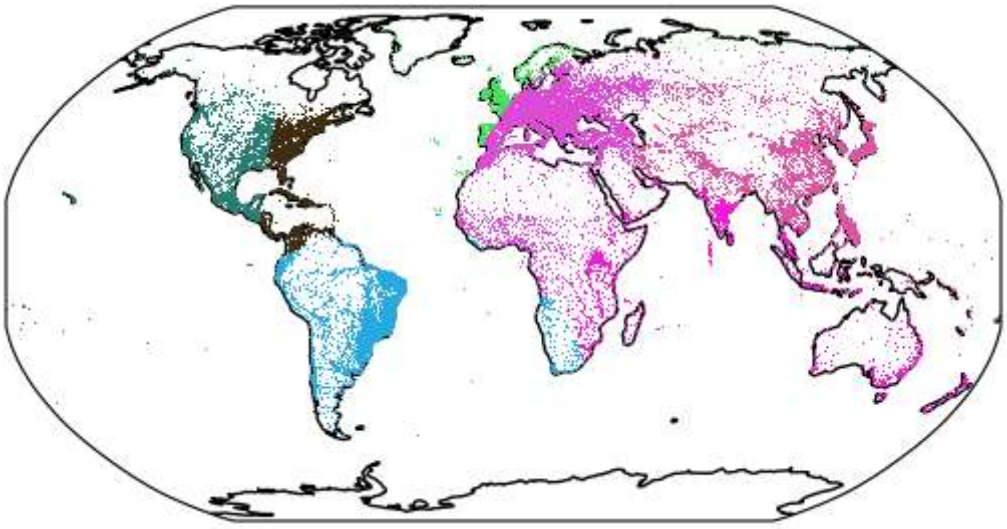


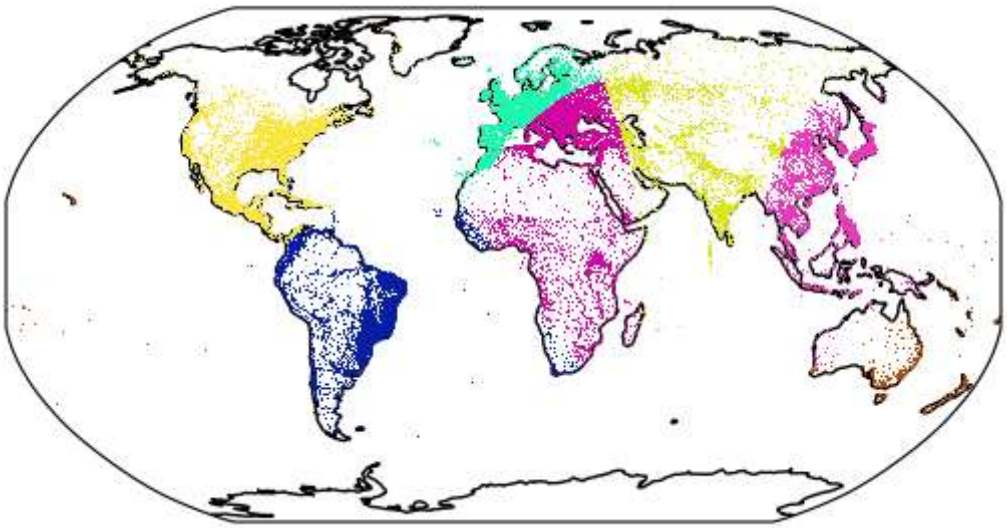


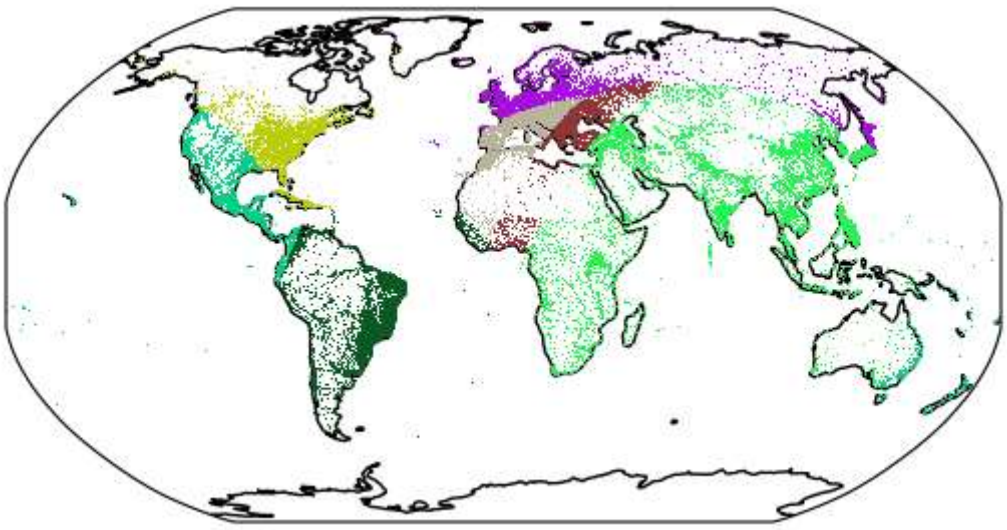


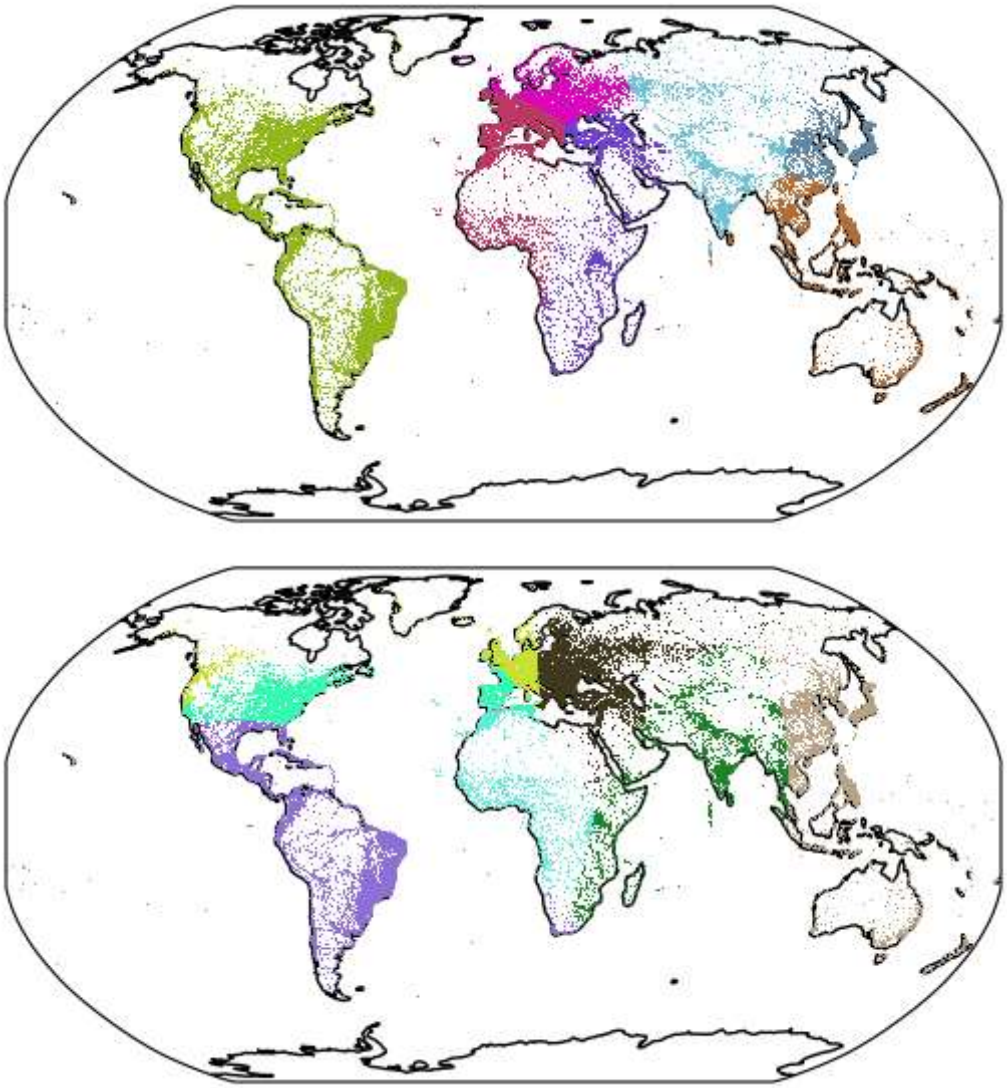
The average run time for k = 5 clustering and plotting is 2.0968576799999994 seconds.

In [9]:
```
time_set_7 = []

for i in range(5):
    t_start = time.perf_counter()

    df_out = k_means_haversine(7, df, True)

    t_end = time.perf_counter()

    time_set_7.append(t_end-t_start)

print("The average run time for k = 7 clustering and plotting is", np.mean(np.array(time_set_7))
```



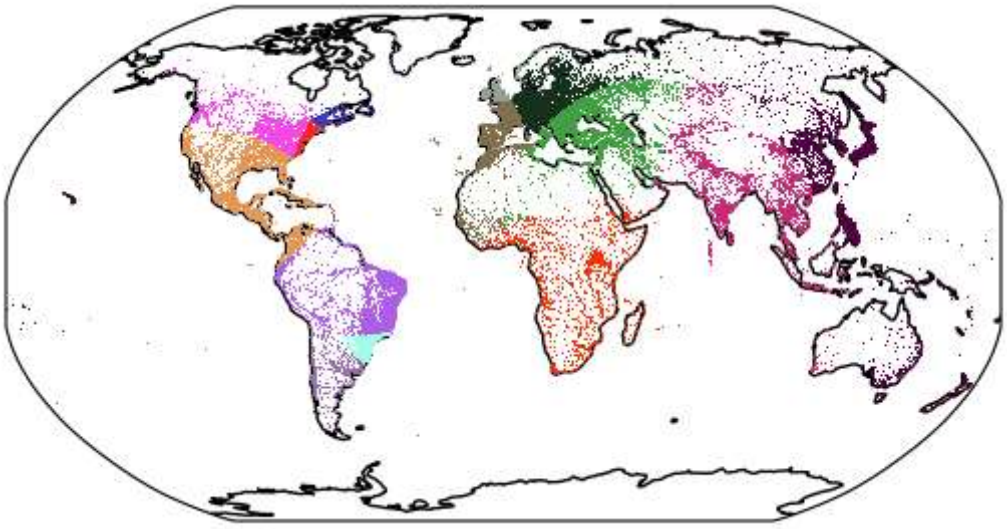


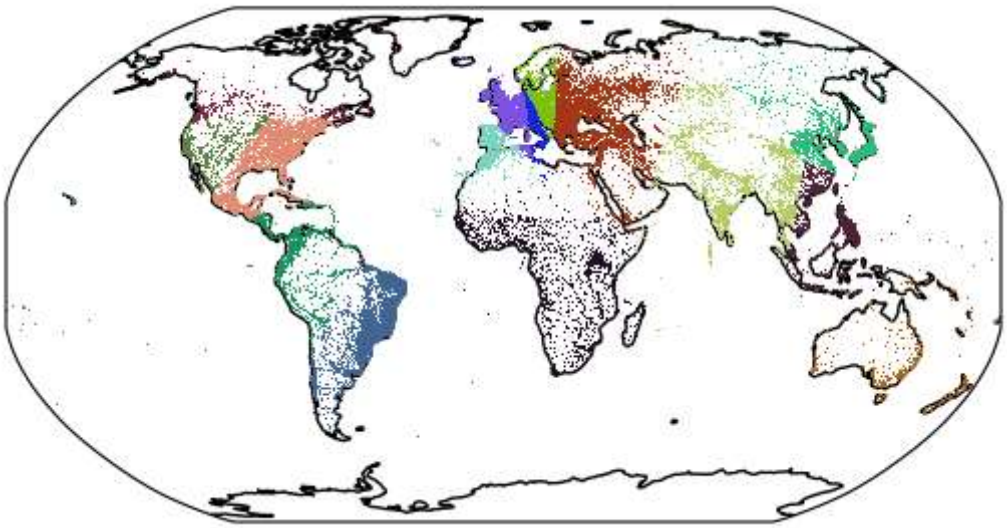


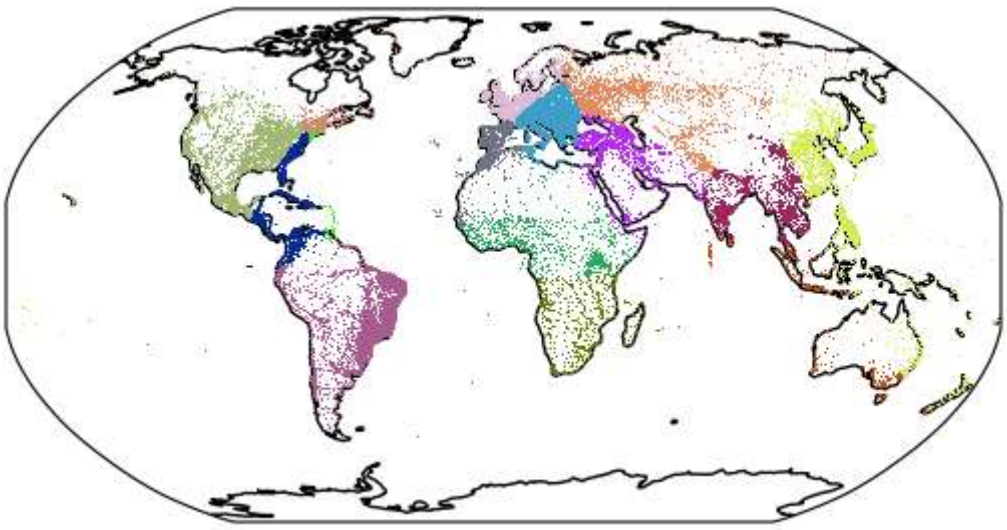


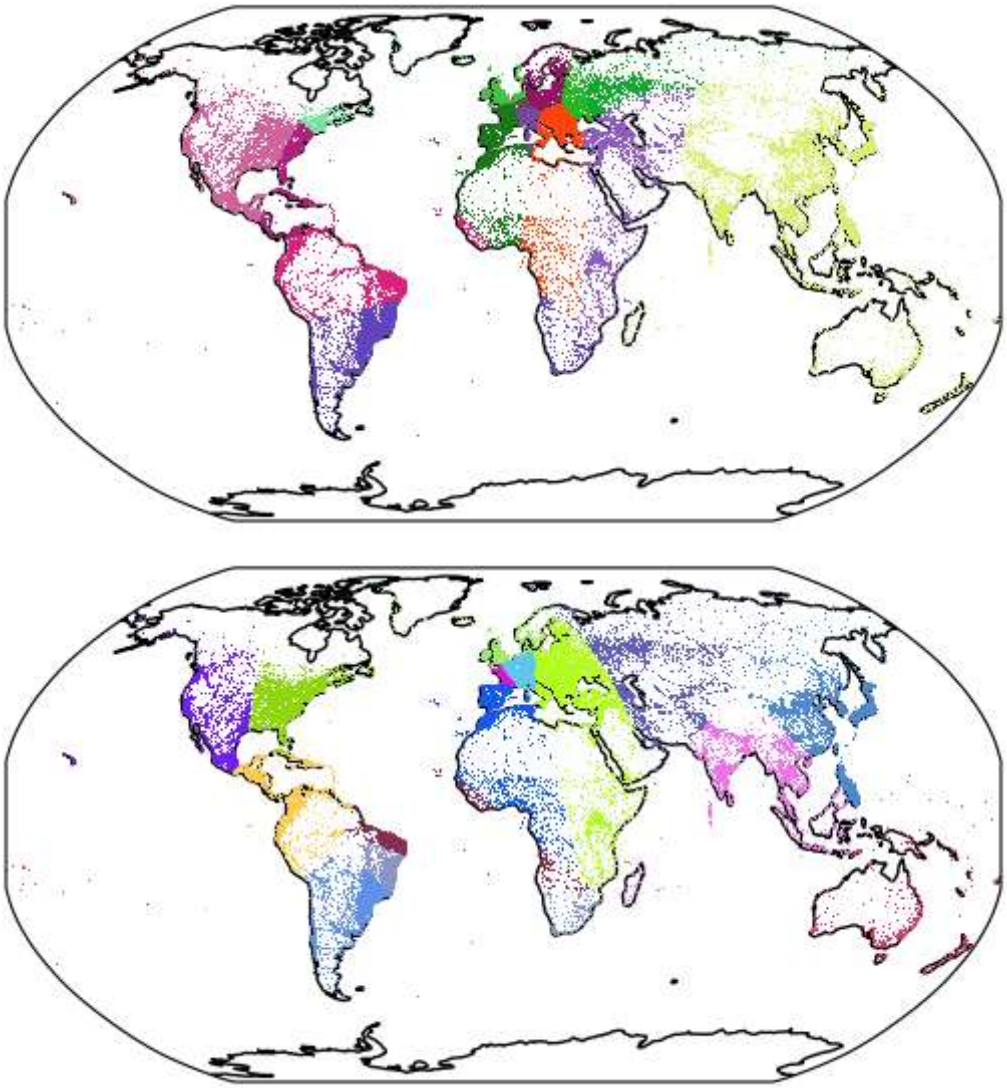

```
The average run time for k = 7 clustering and plotting is 2.377253739999998 seconds.
```

In [10]:
```
time_set_15 = []

for i in range(5):
    t_start = time.perf_counter()

    df_out = k_means_haversine(15, df, True)

    t_end = time.perf_counter()

    time_set_15.append(t_end-t_start)

print("The average run time for k = 15 clustering and plotting is", np.mean(np.array(time_set_15
```










```
The average run time for k = 15 clustering and plotting is 3.4952057800000005 seconds.
```

Comment briefly on the diversity of results for each k. **(5 points)**

At the start of each clustering process, the first set of centroids are picked at random from the points to cluster.
This introduces diversty between every run of the k-means algorithm for the same k.

Additoinally, having differnt k values means that there are different number of clusters.
This introduces diversity in results between clusters of different k values.

Smaller values of k result in bigger clusters and vice-versa.
These bigger clusters fill bigger spaces on the map and appear distinctly different as compared to smaller clusters made by larger values of k.

Question 3

Tahir Manuel D'Mello

In [1]:
```
from functools import lru_cache
import time
import numpy as np
import matplotlib.pyplot as plt
```

Implement Fibonacci recursively and recursively with caching. **(5 points)**

In [2]:
```
def rec_fibonacci(n):
    if n in (1,2):
        return 1
    return rec_fibonacci(n-1) + rec_fibonacci(n-2)
```

In [3]:
```
@lru_cache()
def rec_fibonacci_cache(n):
    if n in (1,2):
        return 1
    return rec_fibonacci_cache(n-1) + rec_fibonacci_cache(n-2)
```

Time them as functions of n. **(5 points)**

In [4]:
```
plain_time = []

for n in range(1,51):
    t_start = time.perf_counter()

    rec_fibonacci(n)

    t_end = time.perf_counter()

    plain_time.append(t_end-t_start)
```

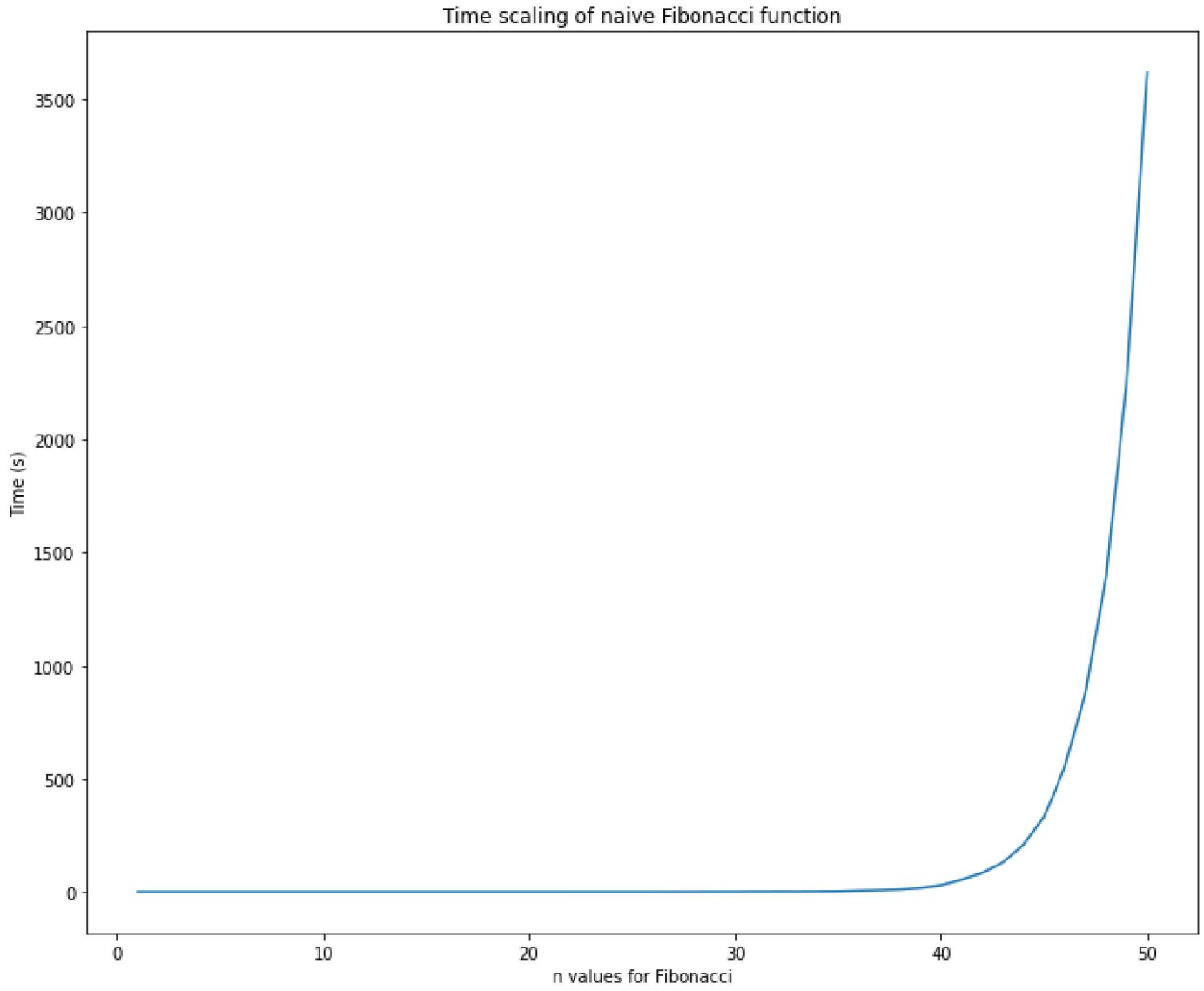
In [10]:
```
cache_time = []

for n in range(1,10000):
    t_start = time.perf_counter()

    rec_fibonacci_cache(n)

    t_end = time.perf_counter()

    cache_time.append(t_end-t_start)
```

Display this in the way you think is best. **(5 points)**

In [6]:
```
a = np.arange(1,51)

plt.figure(figsize=(12,10))
plt.plot(a, np.array(plain_time))

plt.xlabel('n values for Fibonacci')
plt.ylabel('Time (s)')
plt.title('Time scaling of naive Fibonacci function')
```

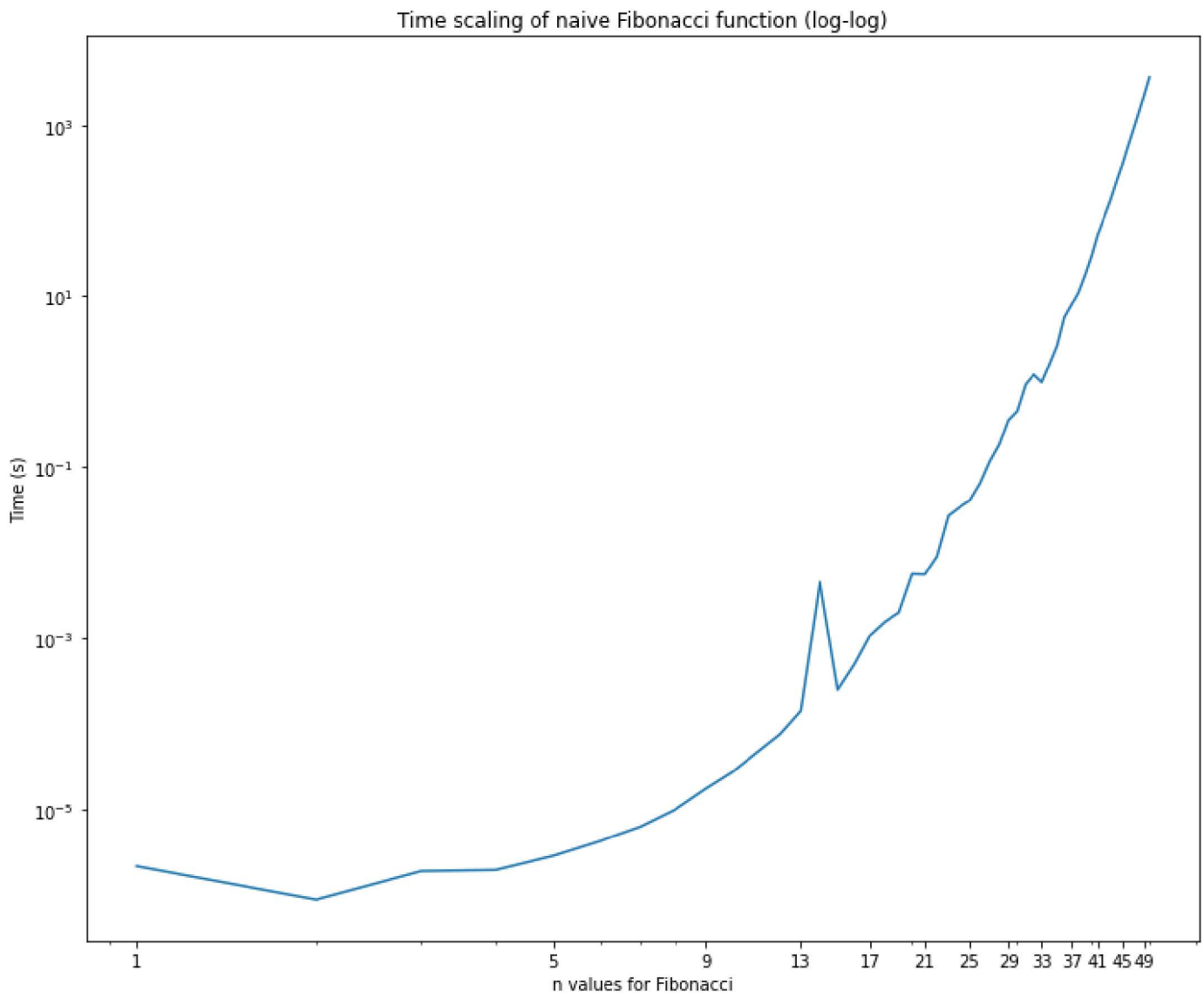
```
plt.show
```

Out[6]:
```
<function matplotlib.pyplot.show(*args, **kw)>
```



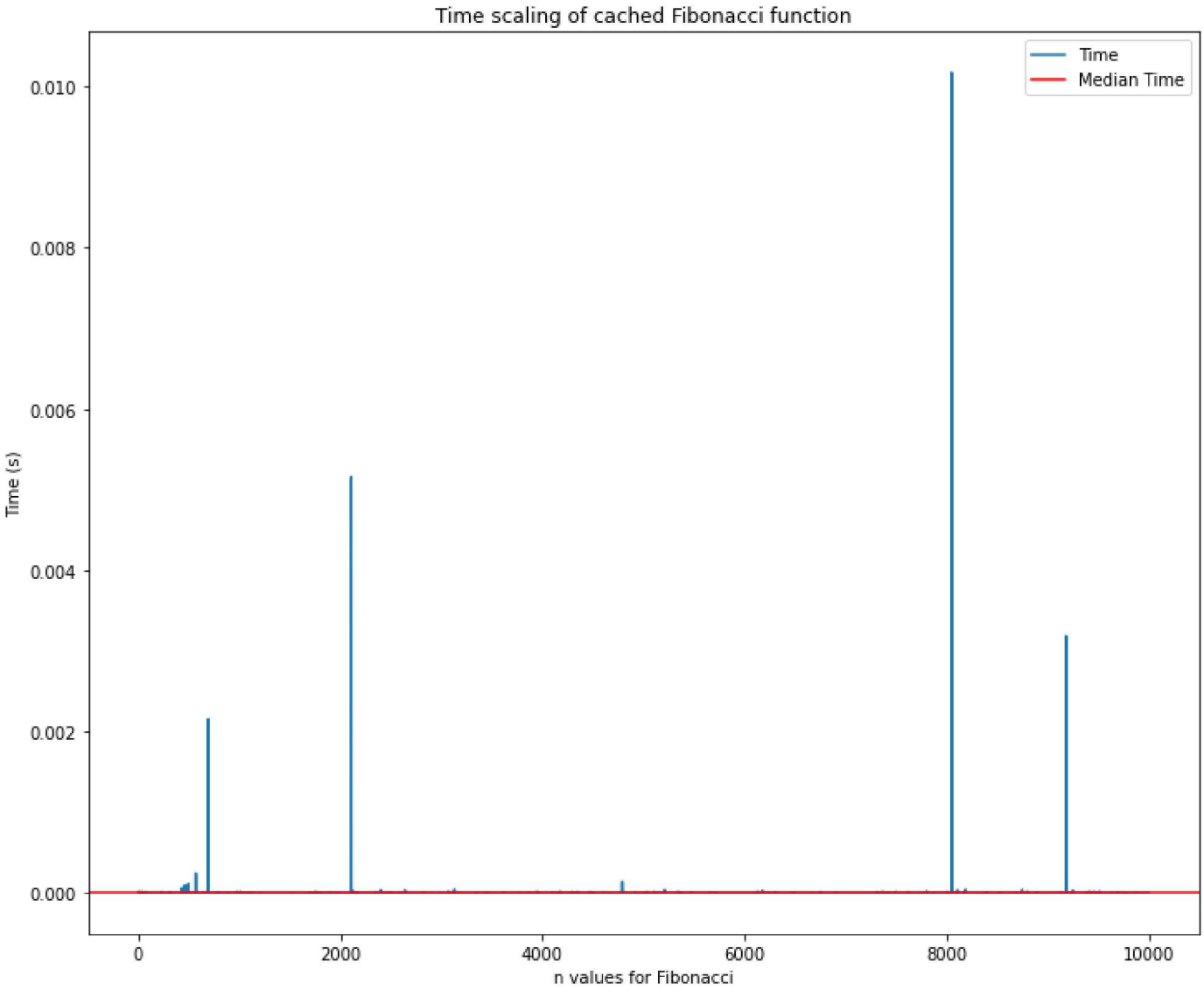
In [8]:
```
plt.figure(figsize=(12,10))
plt.plot(a, np.array(plain_time))

plt.yscale('log')
plt.xscale('log')
plt.xticks(np.arange(1,51, 4), np.arange(1,51, 4))
plt.xlabel('n values for Fibonacci')
plt.ylabel('Time (s)')
plt.title('Time scaling of naive Fibonacci function (log-log)')

plt.show
```

Out[8]:
```
<function matplotlib.pyplot.show(*args, **kw)>
```



In [12]:
```
a = np.arange(1,10000)

plt.figure(figsize=(12,10))
plt.plot(a, np.array(cache_time), label='Time')
plt.axhline(np.median(np.array(cache_time)), color='r', label='Median Time') # horizontal

plt.xlabel('n values for Fibonacci')
plt.ylabel('Time (s)')
plt.title('Time scaling of cached Fibonacci function')
plt.legend()

#plt.yscale('log')
#plt.xscale('log')
plt.show
```

Out[12]:
```
<function matplotlib.pyplot.show(*args, **kw)>
```



Discuss your choices (e.g. why those n and why you're displaying it that way); **(5 points)** and your results. **(5 points)**

For the naive implementation:
n = 50 takes about 3500 seconds. Thus the function was limited to this n for the sake of time convenience.
The first graph displayed was with the original time in seconds vs n. This was not very instructive as the time starts climbing exponentially at n = 35. Thus, a log-log plot was created to display the results more efficiently.
The result makes sense since this is not a cached implementation. For larger values of n, a large number of individual computations need to be made to calculate the answer.

For the cached implementation:
Fibonnaci was computed till n = 10_000. This was because there are no time limitations for the implementation of the cached function. The graph displayed was a plot of time in seconds vs n. There were minor natural variations in the time for various values of n (These may look large on the plot but the y-axis has a scale of 10^-3 seconds).
A horizontal line for the median time of every n was added to show that the time didn't vary much for n.
This result makes sense since this is a cached function. It remembers past values that it calculated so the time for all values of n remains more or less constant.

Question 4

Tahir Manuel D'Mello

Implement a function that takes two strings and returns an optimal local alignment **(6 points)** and score **(6 points)** using the Smith-Waterman algorithm. Insert "-" as needed to indicate a gap (this is part of the alignment points). Your function should also take and correctly use three keyword arguments with defaults as follows: match=1, gap_penalty=1, mismatch_penalty=1. **(6 points)**

In [1]:
```
import numpy as np
```

In [2]:
```
def smith_waterman(seq1, seq2, match_score = 1, gap_penalty = 1, mismatch_penalty = 1):

    gap_penalty = -gap_penalty
    mismatch_penalty = -mismatch_penalty

    m, n = len(seq1), len(seq2)

    p = np.zeros((m + 1, n + 1))

    max_p = 0
    max_i = 0
    max_j = 0

    for i in range(1, m + 1):
        for j in range(1, n + 1):
            ver_gap = p[i][j-1] + gap_penalty
            hor_gap = p[i-1][j] + gap_penalty

            if seq1[i - 1] == seq2[j - 1]:
                match = p[i - 1][j - 1] + match_score
            else:
                match = p[i - 1][j - 1] + mismatch_penalty

            p[i][j] = max(0, hor_gap, ver_gap, match)

            if( p[i][j] > max_p):
                max_i = i
                max_j = j
                max_p = p[i][j]

    i, j = max_i, max_j
    seq_1_out = ""
    seq_2_out = ""

    while i > 0 and j > 0:
        #print(i)
        #print(j)

        if p[i][j-1] == p[i][j] - gap_penalty:
            seq_1_out = seq_1_out + '-'
            seq_2_out = seq_2_out + seq2[j-1]
            j -= 1

        elif p[i-1][j] == p[i][j] - gap_penalty:
            seq_1_out = seq_1_out + seq1[i-1]
            seq_2_out = seq_2_out + '-'
            i -= 1

        elif p[i-1][j-1] == p[i][j] - match_score:
            seq_1_out = seq_1_out + seq1[i-1]
            seq_2_out = seq_2_out + seq2[j-1]
            i -= 1
            j -= 1

        elif p[i-1][j-1] == p[i][j] - mismatch_penalty:
```

```
            seq_1_out = seq_1_out + seq1[i-1]
            seq_2_out = seq_2_out + seq2[j-1]
            i -= 1
            j -= 1

        else:
            break

    seq_1_out = seq_1_out[::-1]
    seq_2_out = seq_2_out[::-1]

    print("Sequence 1 is", seq_1_out)
    print("Sequence 2 is", seq_2_out)
    print("Score is", max_p)

    return seq_1_out, seq_2_out, max_p
```

In [3]:
```
seq1, seq2, score = smith_waterman('tgcatcgagaccctacgtgac', 'actagacctagcatcgac')
```

```
Sequence 1 is atcgagacccta-cgt-gac
Sequence 2 is a-ctagacc-tagcatcgac
Score is 8.0
```

In [4]:
```
seq1, seq2, score = smith_waterman('tgcatcgagaccctacgtgac', 'actagacctagcatcgac', gap_penalty = 2)
```

```
Sequence 1 is gcatcga
Sequence 2 is gcatcga
Score is 7.0
```

Test it, and explain how your tests show that your function works. Be sure to test other values of match, gap_penalty, and mismatch_penalty. **(7 points)**

In [5]:
```
seq1, seq2, score = smith_waterman('qwertychickenasdfg', 'zxcvbchickenmkopl', gap_penalty = 5,  mismatch_pena
```

```
Sequence 1 is chicken
Sequence 2 is chicken
Score is 7.0
```

This implementation heavily penalizes mismatches and gaps so only exact matches with no breaks are given the highest priority.
The word 'chicken' is thus picked out of the entire sequence.

In [6]:
```
seq1, seq2, score = smith_waterman('apblciduetfqg', 'aabgcmdxekfng', match_score = 3, gap_penalty = 1,  misma
```

```
Sequence 1 is apbl-ci-du-et-fq-g
Sequence 2 is a-b-gc-md-xe-kf-ng
Score is 10.0
```

This implementation heavily penalizes mismatches, lightly penalizes gaps and heavily rewards matches.
Thus, the a-b-c-d-e-f-g are matched strictly with appropriate gaps added in between with no mismatches.

In [7]:
```
seq1, seq2, score = smith_waterman('rsapblciduetfqg', 'paabgcmdxekfnga', match_score = 3, gap_penalty = 4,  m
```

```
Sequence 1 is apblciduetfqg
Sequence 2 is aabgcmdxekfng
Score is 15.0
```

This implementation heavily penalizes gaps, lightly penalizes mismatches and heavily rewards matches.
Thus, the a-b-c-d-e-f-g are matched strictly with appropriate mismatches added in between with no gaps.

With these 3 test cases, the function is demonstrated to work for varying penalties.
It functions in a logical manner as expected and thus shows that the function works.