Assignment 4

Exercise 1

Suppose we have a neural network for predicting the time between hospital readmissions for a certain subset of patients. Such a network would depends on many parameters, but for simplicity, let's assume there's only two parameters: a and b. Each parameter lies between 0 and 1 and represents the weights of two "synapses" in our model.

Using the API at e.g.

http://ramcdougal.com/cgi-bin/error_function.py?a=0.4&b=0.2 (http://ramcdougal.com/cgi-bin/error_function.py?a=0.4&b=0.2) (Links to an external site.)

one can find the prediction errors in arbitrary units of running our model with specified parameters (here a=0.4 and b=0.2, but you can change that).

1.1 Implement a two-dimensional version of the gradient descent algorithm to find optimal choices of a and b. (7 points) (Optimal here means values that minimize the error.) See slides 12 for a 1-dimensional implementation of the algorithm.

In [5]:
```
import time
import random
import requests
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

In [5]:
```
def f(a, b):
    return float(requests.get(f"http://ramcdougal.com/cgi-bin/error_function.py?a={a}&b={b}", headers={"User-Agent": "MyScript"}).text)
```

In [6]:
```
def faprime(a,b):
  h = 1e-4
  return (f(a+h,b) - f(a,b))/h

def fbprime(a,b):
  h = 1e-4
  return (f(a,b+h) - f(a,b))/h
```

In [22]:
```
def gradient_descent(a0,b0,steps,gamma,limit):
  f0 = f(a0,b0)
  for i in range(steps):
    fap = faprime(a0,b0)
    fbp = fbprime(a0,b0)
    ax = a0 - gamma * fap
    bx = b0 - gamma * fbp
    fx = f(ax,bx)
    print(ax,bx,fx)
    if abs(fx-f0) <= limit:
      print(fx-f0)
      break
    #UPDATE
    a0, b0 = ax, bx
    f0 = fx
```

In [29]:
```
gradient_descent(0.4,0.2,30,0.2,1e-7)
```

```
0.7743400000000286 0.16273999999996108 1.0117763896
0.6994720000000229 0.1701920000001465 1.00047511494
0.7144455999999352 0.16870160000028706 1.00001821001
0.7114508799999527 0.1689996800000813 1.0000009046
0.7120498199997712 0.1689400600000333 1.00000001822
0.7119300199996289 0.1689519800001314 1.00000002161
3.3900000584452528e-09
```

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

The way the calculation is completed is following the algorithm of Explicit Eulers:

f'(a) = [f(a+h,b) - f(a,b)]/ h

f'(b) = [f(a,b+h) - f(a,b)]/ h

updated f: f(a,b) = f(a0-gamma *f'(a), b0- gamma*f'(b))

The way the gradient implemented was 1) find f'(a) and f'(b), which is the derivative of f for a and b; 2) Then for each step of the iterative algorithm, we use the algorithm to calculate the new f based on the estimated a and b based on updated a, b, f; 3) if the calculation does not reaches the limit, namely, the difference between gradient change not as small as the set limit, we continue until completed all steps defined in the function are completed; if the difference is small enough before completing all defined steps, we stop too.

numerical choices:

stepsize h = 1e-4 a = 0.4 b = 0.2 learning rate = 0.2 limit of gradient change is: 1e-7

a, b were set to be 0.4, 0.2 as suggested in the question. I tested learning rate of 0.1, 0.2, 0.3 and found that learning rate of 0.2 is the most optimal learning rate that converges the fastest. Then I used a step size of 1e-4 because it should be small but not too small that the derivative would not be moving. The limit of gradient change is set to be 1e-7. I tested 1e-3, 1e-5, 1e-7, and found that the limit of 1e-7 is closest to the global minimum.

1.3 It so happens that this error function has a local minimum and a global minimum. Find both locations (i.e. a, b values) querying the API as needed (5 points) and identify which corresponds to which (2 point). Briefly discuss how you would have tested for local vs global minima if you had not known how many minima there were. (2 points)

In [25]:
```
gradient_descent(0.1,0.1,20,0.3,1e-8)
```

```
0.16957000000018532 0.4533700000000195 1.1576772418
0.1973980000005259 0.5947180000005602 1.10923512993
0.2085292000009286 0.6512572000010429 1.1014803318
0.21298168000095644 0.6738728800011028 1.10023794002
0.2147626900011023 0.6829191700008618 1.10003850743
0.2154750700014251 0.6865376800011646 1.10000633857
0.21576001000168646 0.6879850600016834 1.1000010877
0.215874010001793 0.6885640300016259 1.10000020594
0.21591958000156666 0.6887956000014682 1.10000004825
0.21593785000174606 0.6888882400018058 1.10000001635
0.21594514000234924 0.6889252900022068 1.10000000859
-7.760000197976069e-09
```

Based on the initial guess of a as 0.1, b as 0.1, we find that the gradient descent produces an error at the local minimum instead of a global minimum.

local minimum: a = 0.21594514000234924; b = 0.6889252900022068; point = 1.10000000859;

global minimum: a = 0.7083534400010191; b = 0.16930796000049747; point = 1.00004017672

To see whether an error like this would occur, I would set my learning rate high and have a high enough limit of gradient stop, large steps defined, to loop through the whole data and see if there is any change in decreasing and increasing trend inside the data. If there is spotted ones, I will try to start the gradient descent at different location of a and b and see if there is different local minimas, and determine which one is the global minimum.

Exercise 2

As briefly introduced in slides13, k-means is an algorithm for automatically identifying clusters in data. Lloyd's algorithm (there are others) for k-means is simple: for a given k, pick k points at pseudo-random from your data set (this is called the Forgy method for initialization, there are other other strategies). These will be "seeds" for forming k clusters. Assign each data point to a cluster based on whichever seed point is closest. Iterate, using the centroid of each cluster as the seeds for the next round of point assignments. Repeat until convergence. (Feel free to Google or ask questions if you need clarification on the algorithm.)

2.1 Modify the k-means code (or write your own) from slides8 to use the Haversine metric and work with our dataset (5 points). Note: since this algorithm uses (pseudo)randomness, you'll have to run it multiple times to get a sense of expected runtime.

In [6]:
```
df = pd.read_csv('worldcities.csv')
```

In [21]:
```
#import this function from stack overflow
#cite: https://stackoverflow.com/questions/4913349/haversine-formula-in-python
-bearing-and-distance-between-two-gps-points

from math import radians, cos, sin, asin, sqrt

def haversine(lon1, lat1, lon2, lat2):
    """
    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 r
eturn value units.
    return c * r
```

In [22]:
```
#from cheatsheet
import numpy as np
import numpy.linalg as lin
from math import sin, cos, sqrt, atan2, acos, radians, degrees, pi

#takes 2-d coords as numpy array, returns numpy array representing vector
def T(geocoords):
    #unpack coords from input
    long, lat = geocoords

    #convert long and lat to spherical coords
    theta = radians(long)
    phi = pi/2 - radians(lat)

    #convert spherical coords to 3-d cartesian coords
    x = sin(phi)*cos(theta)
    y = sin(phi)*sin(theta)
    z = cos(phi)

    return np.array([x,y,z])
```

In [23]:
```
#rewrite vectorize function
def vT(vecs):
    vecs = np.array(vecs)
    res = []
    for i in range(vecs.shape[0]):
        res.append(T(vecs[i]))
    return np.array(res)
```

In [24]:
```
# takes 3 x 1 numpy array representing unit vector,
# returns numpy array representing long and lat coords
def inverse_T(vec):
    #unpack coords of input
    x, y, z = vec

    theta = atan2(y, x)
    phi = acos(z/sqrt(x * x + y * y + z * z))

    long = degrees(theta)
    lat = degrees(pi/2 - phi)

    return np.array([long, lat])
```

In [11]:
```
# takes a n x 3 numpy array representing a list of our 3d vectors,
# returns their normalized sum
def cartesian_centroid(vecs):
    summed = np.sum(vecs, axis=0)
    normalized = summed / lin.norm(summed)
    return normalized
```

In [25]:
```
# takes a n x 2 numpy array representing a list of our 2d geographical coordin
ate pairs, returns their centroid as a 2 x 1 numpy array.
def centroid(vecs):
    cart_vecs = vT(vecs)
    cart_center = cartesian_centroid(cart_vecs)
    return inverse_T(cart_center)
```

In [13]:
```
#!conda install -c conda-forge cartopy -y
```

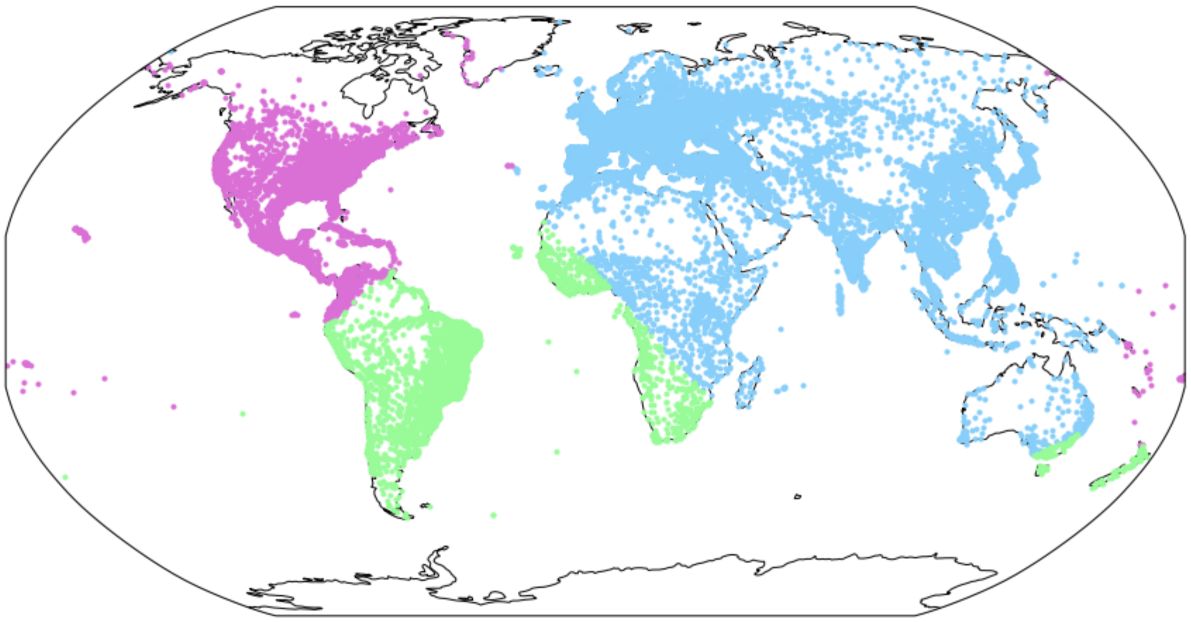
In [26]:
```
import cartopy.crs as ccrs
from time import perf_counter
```

In [41]:
```python
def geo_kmeans(k=3):
    df2 = df.copy() #create a copy so that modification not take place at df
    pts = [np.array(pt) for pt in zip(df2['lng'], df2['lat'])]
    checkpoint1 = perf_counter()
    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 = haversine(pt[0], pt[1], center[0], center[1])
                if dist < min_dist:
                    min_cluster = i
                    min_dist = dist
            cluster_ids.append(min_cluster)
        df2['cluster'] = cluster_ids
        df2['cluster'] = df2['cluster'].astype('category')
        cluster_pts = [[pt for pt, cluster in zip(pts, cluster_ids) if cluster
== match]
                        for match in range(k)]
        centers = [centroid(pts) for pts in cluster_pts]
    checkpoint2 = perf_counter()
    print("Runtime for Clustering is ", checkpoint2 - checkpoint1)

    #draw the plot
    color_list = ['lightSkyBlue','orchid','PaleGreen','yellow','pink',
                    'plum','MistyRose','MediumPurple','LightSteelBlue','PeachP
uff',
                    'PaleTurquoise','PaleTurquoise','PaleVioletRed','oliveDra
b','MidnightBlue']

    fig = plt.figure(figsize = (15, 15))
    ax = fig.add_subplot(1, 1, 1, projection=ccrs.Robinson())
    ax.coastlines()
    for i in range(k):
        lngs = df2[df2["cluster"] == i]["lng"]
        lats = df2[df2["cluster"] == i]["lat"]
        ax.plot(lngs, lats, ".", c=color_list[i], transform=ccrs.PlateCarree
())
    ax.set_extent([-180, 180, -90, 90], crs=ccrs.PlateCarree())
    plt.show()
```

2.2 Visualize your results with a color-coded scatter plot (5 points)

```
In [37]: geo_kmeans()
```

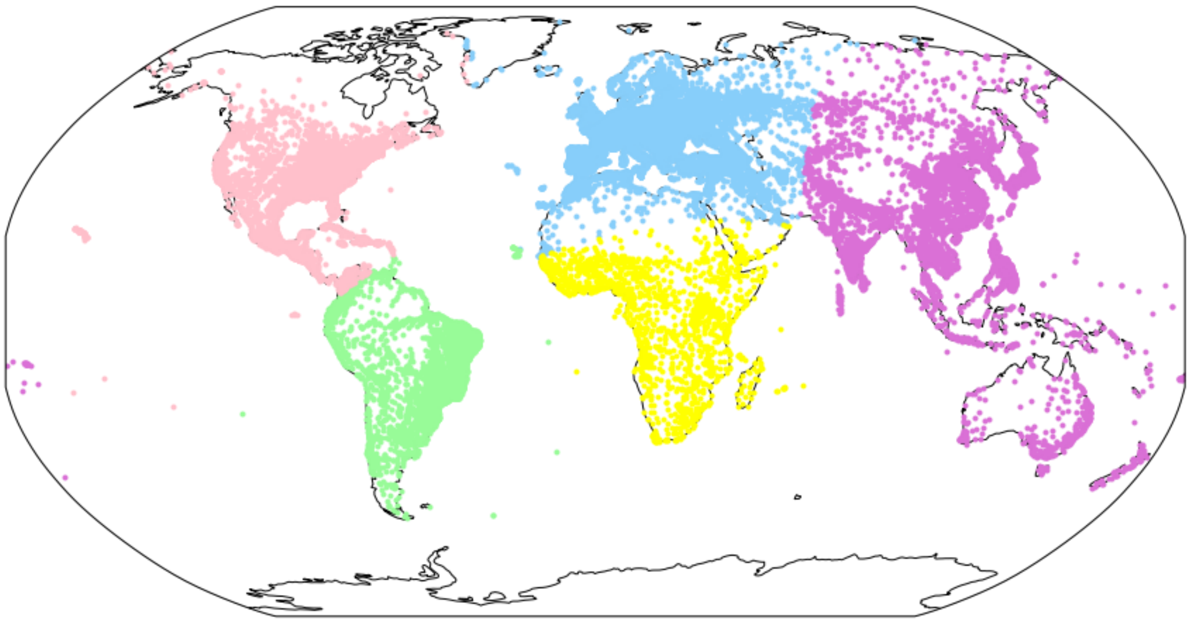
```
Runtime for Clustering is  12.902549500000077

c:\Users\qiuxi\Anaconda3_2\lib\site-packages\cartopy\mpl\geoaxes.py:388: Matp
lotlibDeprecationWarning:
The 'inframe' parameter of draw() was deprecated in Matplotlib 3.3 and will b
e removed two minor releases later. Use Axes.redraw_in_frame() instead. If an
y parameter follows 'inframe', they should be passed as keyword, not position
ally.
  inframe=inframe)
```



2.3 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); be sure to use an appropriate map projection (i.e. do not simply make x=longitude and y=latitude; 5 points).

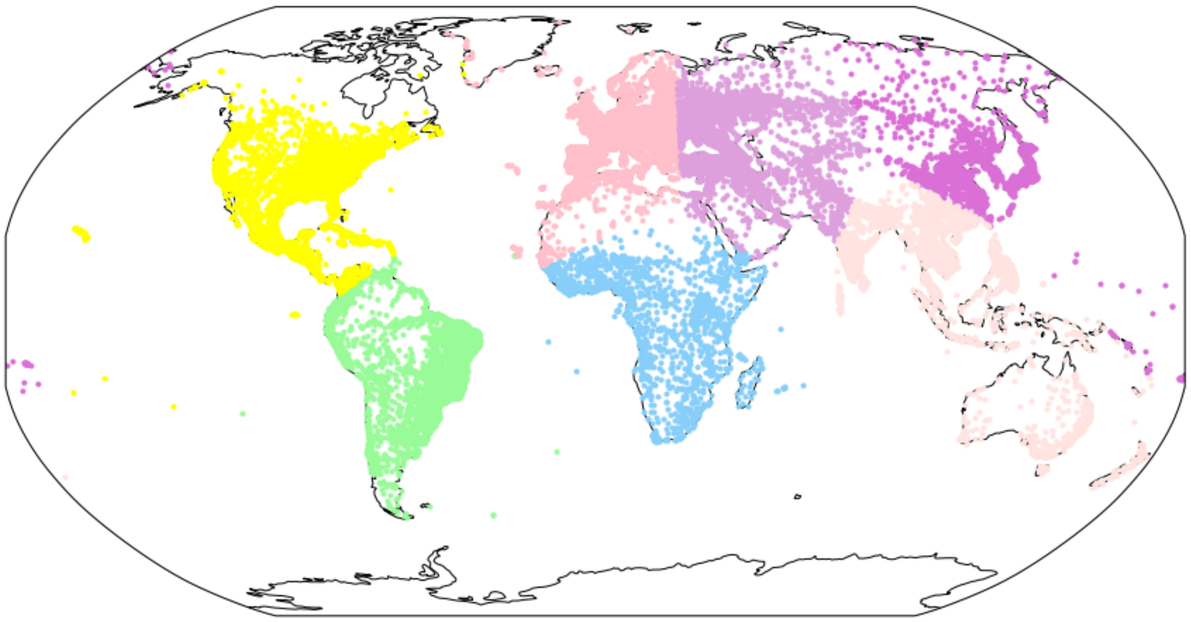
```
In [38]: geo_kmeans(5)
```

```
Runtime for Clustering is  16.022558500000287

c:\Users\qiuxi\Anaconda3_2\lib\site-packages\cartopy\mpl\geoaxes.py:388: Matp
lotlibDeprecationWarning:
The 'inframe' parameter of draw() was deprecated in Matplotlib 3.3 and will b
e removed two minor releases later. Use Axes.redraw_in_frame() instead. If an
y parameter follows 'inframe', they should be passed as keyword, not position
ally.
  inframe=inframe)
```



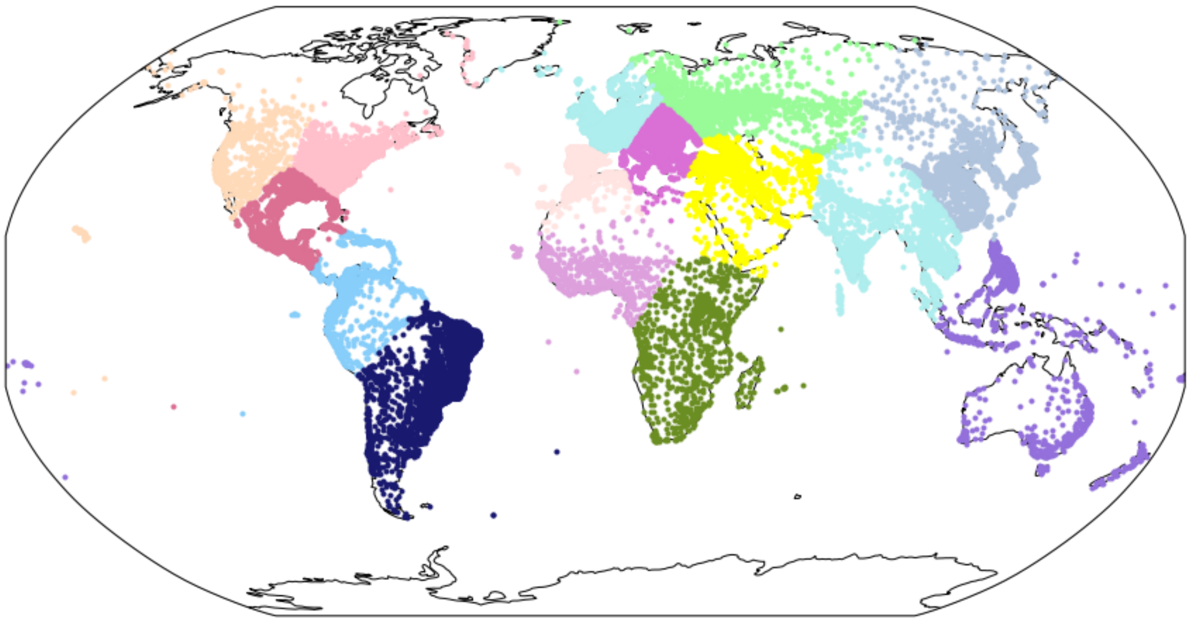
In [39]:
```
geo_kmeans(7)
```

```
Runtime for Clustering is  63.79687139999987

c:\Users\qiuxi\Anaconda3_2\lib\site-packages\cartopy\mpl\geoaxes.py:388: Matp
lotlibDeprecationWarning:
The 'inframe' parameter of draw() was deprecated in Matplotlib 3.3 and will b
e removed two minor releases later. Use Axes.redraw_in_frame() instead. If an
y parameter follows 'inframe', they should be passed as keyword, not position
ally.
  inframe=inframe)
```



In [42]:
```
geo_kmeans(15)
```

```
Runtime for Clustering is  112.29810090000001

c:\Users\qiuxi\Anaconda3_2\lib\site-packages\cartopy\mpl\geoaxes.py:388: Matp
lotlibDeprecationWarning:
The 'inframe' parameter of draw() was deprecated in Matplotlib 3.3 and will b
e removed two minor releases later. Use Axes.redraw_in_frame() instead. If an
y parameter follows 'inframe', they should be passed as keyword, not position
ally.
  inframe=inframe)
```



2.5 comment briefly on the diversity of results for each k. (5 points)

Runtime-wise, the larger the k is, the longer the runtime is. On k = 3, we are seeing a high randomness in the resulting clusters: if you run the couple times on the function, we see a diversity of clusters on the map; on k = 5 and 7, we see the randomness taking less control on the plotted clusters; on k = 15, we are seeing that all plotted maps looks relatively the same.

Exercise 3

3.1 In class, we discussed two different strategies for computing the Fibonacci sequence: directly with the recursive strategy, and recursive but modified using lru_cache. Implement both (yes, I know, I gave you implementations on the slides, but try to do this exercise from scratch as much as possible) (5 points)

In [32]:
```
# recursive strategy
def r(n):
  if n in (1,2):
    return 1
  return r(n-1) + r(n-2)
```

In [33]:
```
# recursive but modified using lru_cache
from functools import lru_cache

@lru_cache()
def r_lru(n):
  if n in (1,2):
    return 1
  return r_lru(n-1) + r_lru(n-2)
```

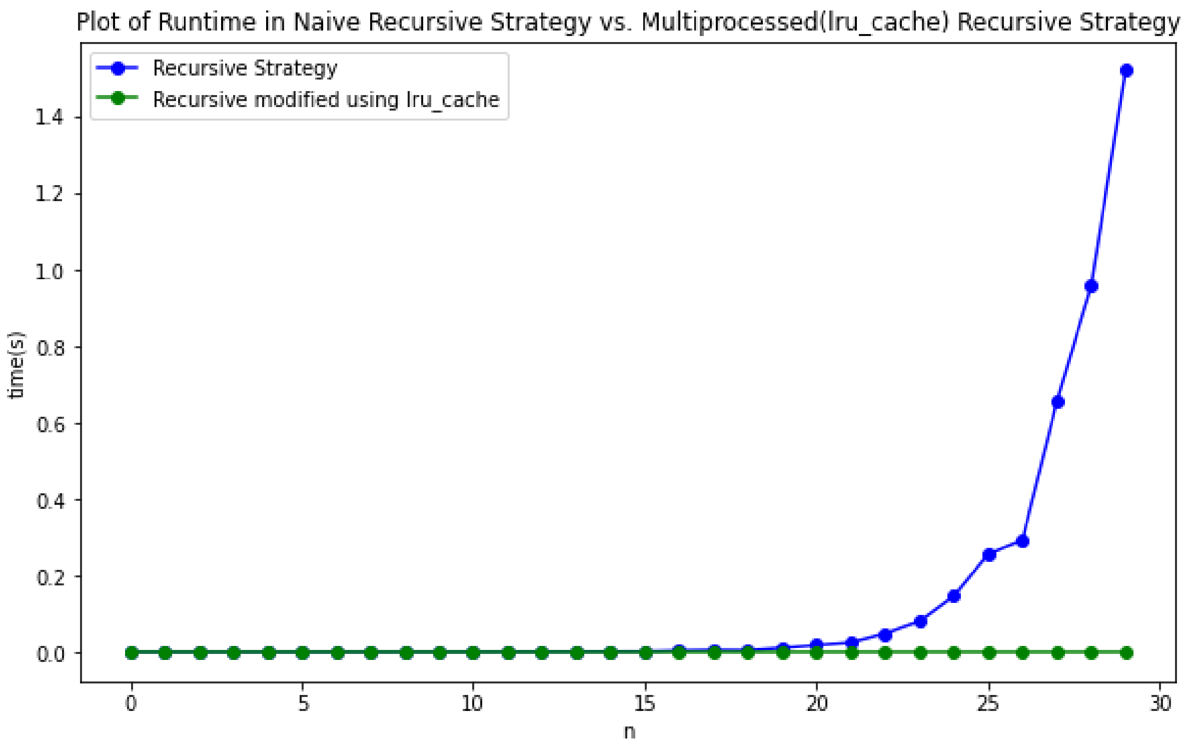
3.2 time them as functions of n (5 points)

3.3 display this in the way you think is best (5 points).

In [43]:
```
from time import perf_counter
n = np.arange(30)
t_no_lru = np.zeros(len(n))
t_lru = np.zeros(len(n))
```

In [44]:
```
for i in n:
  #first no lru cache time count
  checkpoint1 = perf_counter()
  r(i+1)
  checkpoint2 = perf_counter()
  t_no_lru[i] = (checkpoint2-checkpoint1)

  #Second record with lru cache
  checkpoint3 = perf_counter()
  r_lru(i+1)
  checkpoint4 = perf_counter()
  t_lru[i] = (checkpoint4 - checkpoint3)
```

In [50]:
```
fig, ax = plt.subplots(figsize=(10, 6))
plt.plot(n, t_no_lru, '-ob', label = "Recursive Strategy")
plt.plot(n, t_lru, '-og', label = "Recursive modified using lru_cache")
plt.legend()
plt.xlabel('n')
plt.ylabel('time(s)')
plt.title('Plot of Runtime in Naive Recursive Strategy vs. Multiprocessed(lru_
cache) Recursive Strategy')
plt.show()
```



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

From the lecture we know that the time complexity of naive implementation is proportional to φ^n , while the time complexity of lru_cache implemented strategy is O(n). Thus I chose n to be 30 because when n gets very large, the runtime of naive implementation will be too long. I display the graph of runtime for recursive strategy vs. with lru_cache recursive strategy using scatterplot, which does successfully shows the difference in time complexity.

Exercise 4

4.1 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 [56]:
```python
def align(seq1,seq2, match = 1, gap_penalty = 1, mismatch_penalty = 1):
  seq1 = '-' + seq1
  seq2 = '-' + seq2
  a = len(seq1)
  b = len(seq2)
  grid = np.zeros((a,b))


  #set up grids
  for i in range(1,a):
    for j in range(1,b):
      if seq1[i] == seq2[j]:
         s = match
      else:
        s = -mismatch_penalty
        #mij
      grid[i][j] = max(grid[i-1][j-1]+s, grid[i-1][j] - gap_penalty, grid[i][j
-1]-gap_penalty,0)

  l_score = 0
  l_coord = (0,0)

  #backward: find max score and end points
  for i in range(a):
      for j in range(b):
        if l_score < grid[i,j]:
          l_score = grid[i,j]
          l_coord = (i,j)

  #return matched seq1 seq2
  seq1_m = ''
  seq2_m = ''
  l_score_copy = l_score
  while l_score > 0: #loop till the score counts to be zero
    if seq1[l_coord[0]] == seq2[l_coord[1]]:#when exact match
      seq1_m = seq1[l_coord[0]] + seq1_m
      seq2_m = seq2[l_coord[1]] + seq2_m
      l_coord =(l_coord[0]-1, l_coord[1]-1)
      l_score -= match
    elif grid[l_coord[0]][l_coord[1]-1] == l_score + gap_penalty:
      seq1_m = '-' + seq1_m
      seq2_m = seq2[l_coord[1]] + seq2_m
      l_coord =(l_coord[0], l_coord[1]-1)
      l_score += gap_penalty
    elif grid[l_coord[0]-1][l_coord[1]] == l_score + gap_penalty:
      seq2_m = '-' + seq2_m
      seq1_m = seq1[l_coord[0]] + seq1_m
      l_coord =(l_coord[0]-1, l_coord[1])
      l_score += gap_penalty
    else:
      seq1_m = seq1[l_coord[0]] + seq1_m
      seq2_m = seq2[l_coord[1]] + seq2_m
      l_coord =(l_coord[0]-1, l_coord[1]-1)
      l_score += mismatch_penalty

  return(seq1_m,seq2_m,l_score_copy)
```

4.2 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 [67]: align('tgcatcgagaccctacgtgac', 'actagacctagcatcgac')
Out[67]: ('agacccta-cgt-gac', 'aga-cctagcatcgac', 8.0)
```

```
In [68]: align('tgcatcgagaccctacgtgac', 'actagacctagcatcgac', gap_penalty=2)
Out[68]: ('gcatcga', 'gcatcga', 7.0)
```

First I tested the two sequence alignment provided in the question. Apparently, these two reports the score that is defined in the problem, so the function does work for aligning sequences and score for changed mismatch penalty. Next I tested for the value of scores based on diferent match numbers.

```
In [53]: def find_l_score(seq1,seq2, match =1, gap_penalty = 1, mismatch_penalty=1,):
           l_score = 0
           for i in range(len(seq1)):
             if seq1[i] == seq2[i]:
               l_score += match
             elif seq1[i] == '-' or seq2[i] == '-':
               l_score -= mismatch_penalty
             else:
               l_score -= gap_penalty


           return l_score
```

```
In [72]: align('tgcatcgagaccctacgtgac', 'actagacctagcatcgac', match = 2)
Out[72]: ('atcgagacccta-cgt-gac', 'a-ctaga-cctagcatcgac', 22.0)
```

```
In [48]: find_l_score('atcgagacccta-cgt-gac', 'a-ctaga-cctagcatcgac', match = 2)
Out[48]: 22
```

I used a function called find_l_score to calculate the score of matched sequences. The returned scores are calculated score of the matched sequences. The above codes shows that the returned score from align function matched the score from the find_l_score function.

```
In [70]: align('tgcatcgagaccctacgtgac', 'actagacctagcatcgac', gap_penalty=3)
Out[70]: ('gcatcga', 'gcatcga', 7.0)
```

```
In [52]: find_l_score('gcatcga', 'gcatcga', gap_penalty=3)
Out[52]: 7
```

These two functions shows the same returned scores based on the alignments, which shows the gap_penalty is correctly calculated as well.

```
In [57]: align('tgcatcgagaccctacgtgac', 'actagacctagcatcgac', mismatch_penalty=3)

Out[57]: ('gcatcga', 'gcatcga', 7.0)
```

```
In [58]: find_l_score('gcatcga', 'gcatcga', mismatch_penalty=3)

Out[58]: 7
```

These two functions shows the same returned scores based on the alignments, which shows the mismatch_penalty is correctly calculated as well.