EX1

```
There is one global min:  1.00004401976  at  [0.7127202500004529,  0.17030597999984032]
There is one local min:  1.00007797336  at  [0.713875639999905,  0.16824395999993627]
```

As the function is unknown, but we can query result by API, gradient descent can be used for estimate derivative.

$$\nabla f(a,b) \approx \frac{f(a+h,b)-f(a,b)}{h}\hat{i} + \frac{f(a,b+h)-f(a,b)}{h}\hat{j},$$

I chose the step size to be 0.1, and h to be 1e-4. The gradient descent method requires the parameters to be small enough to reduce the bias. The stopping criterion I chose is when the function value changes by less than 0.0001 in each iteration. I chose this criterion because any change at the 1e-4 level is relatively insignificant, and our generated a and b values are likely to be very close to the minimum point.

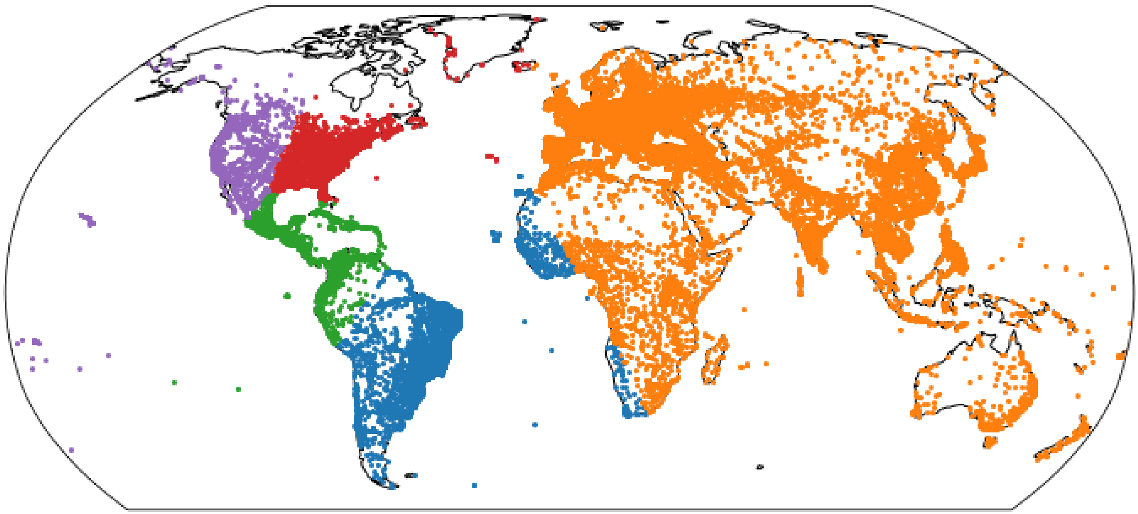
```python
def opt_min(a,b,prev_error):
    h=1e-4
    ss=0.1
    stop=1e-4
    while(abs(query_api(a,b)-prev_error)>stop):
        prev_error=query_api(a,b)
        d_a=(query_api(a+h, b) - query_api(a, b))/h
        d_b=(query_api(a, b+h) - query_api(a, b))/h
        next_a=(a-ss*d_a)
        next_b=(b-ss*d_b)
        a,b=next_a,next_b
    return a,b,prev_error
```

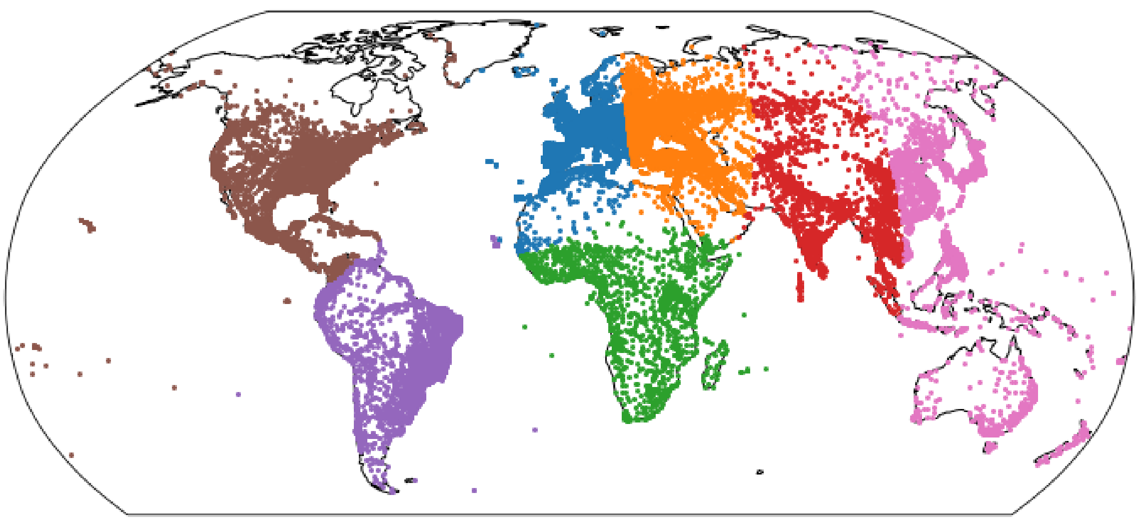
When we don't know how many minima there are, if the estimated global minima are very close to the local minima, then we have probably found the right answer

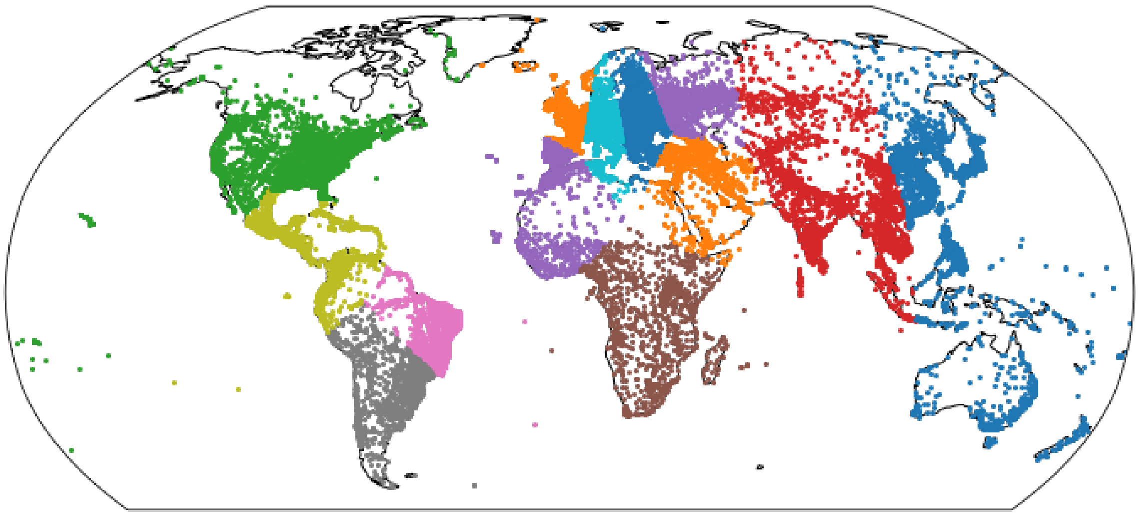
```
There is one global min:  1.00004401976  at  [0.7127202500004529,  0.17030597999984032]
There is one local min:  1.00007797336  at  [0.713875639999905,  0.16824395999993627]
{1.00004401976:  [0.7127202500004529,  0.17030597999984032],
 1.00007797336:  [0.713875639999905,  0.16824395999993627],
 1.00009984108:  [0.7097796300002397,  0.1682439500001575],
 1.10018428952:  [0.213861230000456,  0.6783404199999342],
 1.10022471098:  [0.2259441199999319,  0.6823019100000156],
 1.10023418814:  [0.2095755899989115,  0.6785623599992459],
 1.10024005109:  [0.22594412999948865,  0.6963756699999039],
 1.10025784087:  [0.22827274999969094,  0.6927519299991683],
 1.10027318479:  [0.20957558000024293,  0.7005525900004465]}
```

EX2

The order is k=5,7,15







We can see that as the k value becomes larger, the cluster increases and more color blocks can be seen on the legend. When k = 5, the entire Asian-African-European plate as well as the oceanic region is almost connected. In multiple attempts, sometimes sub-Saharan Africa separates out. I think the possible reason is that at low k values, the main influencing factors are natural geographical divisions, such as oceans and vast uninhabited areas. As the value of k rises, we can see some typical urban cluster divisions emerge. At k = 7, we see the distribution of the Far East, the South Asian subcontinent, Eastern Europe, and Western Europe. This is quite a bit more detailed than at k=5. And at k=15, the level of detail increases further. The Americas appear as a smaller block of North America, Central America, and South America separated by the Amazon rainforest. In addition, a more detailed zoning of urban zones has emerged in Europe.

EX3

In [2]:
```
# naive solution
def naive_fib(n):
    if n <2:
        return n
    return naive_fib(n - 1) + naive_fib(n - 2)
```

In [3]:
```
# test
print(naive_fib(4))
print(naive_fib(7))
print(naive_fib(15))
print(naive_fib(18))
```

```
3
13
610
2584
```

In [4]:
```
@lru_cache()
def lru_cache_fib(n):
    if n < 2:
        return n
    else:
        return lru_cache_fib(n-1) + lru_cache_fib(n-2)
```

In [5]:
```
# test
print(lru_cache_fib(4))
print(lru_cache_fib(7))
print(lru_cache_fib(15))
print(lru_cache_fib(18))
```

```
3
13
610
2584
```

First, I completed two solutions of the Fibonacci series.

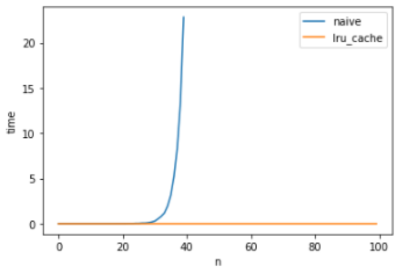
In [97]:
```python
time_naive = []

for n in tqdm(range(40)):
    start=time.time()
    runtime=naive_fib(n)
    time_naive.append(time.time()-start)
```

```
100%|████████████████████████████████████████████████████████████
██████████████████████| 40/40 [00:58<00:00,  1.45s/it]
```

In [6]:
```python
time_lru_cache = []

for n in tqdm(range(100)):
    start=time.time()
    runtime=lru_cache_fib(n)
    time_lru_cache.append(time.time()-start)
```

```
100%|████████████████████████████████████████████████████████████
████████████████| 100/100 [00:00<00:00, 100294.21it/s]
```

In [123]:
```python
plt.plot(range(40),time_naive, label="naive")
plt.plot(range(100),time_lru_cache, label="lru_cache")
plt.legend()
plt.xlabel("n")
plt.ylabel("time")
plt.show()
```



By comparing the two methods, we find that recursion but using the cache file replacement mechanism is much faster, with a very short, elapsed time at least up to 100 positions. The naive recursive method, on the other hand, consumes exponentially more time when computing up to about the 30th position. In contrast, the cache modification method consumes essentially the same amount of time. Thus, I think lru_cache method is better.

EX4

```
In [1]: import numpy as np
        import matplotlib.pyplot as plt
```

```
In [29]: def smith_waterman(seq1,seq2,match=1,gap_penalty=1,mismatch_penalty=1):
             matrix=np.zeros((len(seq1)+1, len(seq2)+1))
             for i in range(1, len(seq1) + 1):
                 for j in range(1, len(seq2) + 1):
                     if seq1[i-1]==seq2[j-1]:
                         score = match
                         matrix[i][j]=score
                     else:
                         score =-mismatch_penalty
                         matrix[i][j]=score
                     matrix[i][j] = max(0, matrix[i-1][j-1] + score, matrix[i][j-1] - gap_penalty, matrix[i-1][j] - gap_penalty)

             for i in range(1, len(seq1) + 1):
                 for j in range(1, len(seq2) + 1):
                     if matrix[i][j] == matrix.max():
                         max_i,max_j=i,j

             subseq1 = ''
             subseq2 = ''
             while matrix[max_i][max_j] != 0:
                 score_max = matrix[max_i][max_j]
                 score_up_left = matrix[max_i-1][max_j-1]
                 score_up = matrix[max_i][max_j-1]
                 score_left = matrix[max_i-1][max_j]
```

Didn't quite finish

Code

EX1

```
In [94]: import requests
         import numpy as np
         import pprint
```

```
In [55]: # query API
         def query_api(a, b):
             return float(requests.get(f"http://ramcdougal.com/cgi-bin/error_function.py?a={a}&b={b}", headers={"User-Agent": "MyScript"
```

```
In [56]: def opt_min(a,b,prev_error):
             h=1e-4
             ss=0.1
             stop=1e-4
             while(abs(query_api(a,b)-prev_error)>stop):
                 prev_error=query_api(a,b)
                 d_a=(query_api(a+h, b) - query_api(a, b))/h
                 d_b=(query_api(a, b+h) - query_api(a, b))/h
                 next_a=(a-ss*d_a)
                 next_b=(b-ss*d_b)
                 a,b=next_a,next_b
             return a,b,prev_error
```

```
In [57]: opt_min(a=0.4,b=0.2,prev_error=100)
Out[57]: (0.7106722500001436, 0.1690771800000544, 1.0000317932)
```

```
In [96]: output={}
         range_a = [0.1,0.5,0.9]
         range_b = [0.1,0.5,0.9]
         for a in range_a:
             for b in range_a:
                 optimal=opt_min(a,b,100)
                 avalue=optimal[0]
                 bvalue=optimal[1]
                 mininum=optimal[2]
                 output[mininum]=[avalue,bvalue]
         mininum_list=sorted(output.keys())
         global_=mininum_list[0]
         local_=mininum_list[1]
         global_ab=output.get(global_)
         local_ab=output.get(local_)

         print("There is one global min:", global_, "at", global_ab)
         print("There is one local min:", local_, "at", local_ab)
         pprint.pprint(output)
```

```
There is one global min: 1.00004401976 at [0.7127202500004529, 0.17030597999984032]
There is one local min: 1.00007797336 at [0.713875639999905, 0.16824395999993627]
{1.00004401976: [0.7127202500004529, 0.17030597999984032],
 1.00007797336: [0.713875639999905, 0.16824395999993627],
 1.00009984108: [0.7097796300002397, 0.1682439500001575],
 1.10018428952: [0.213861230000456, 0.6783404199999342],
 1.10022471098: [0.2259441199999319, 0.6823019100000156],
 1.10023418814: [0.2095755899989115, 0.6785623599992459],
 1.10024005109: [0.22594412999948865, 0.6963756699999039],
 1.10025784087: [0.22827274999969094, 0.6927519299991683],
 1.10027318479: [0.20957558000024293, 0.7005525900004465]}
```

```
In [ ]:
```

EX2

In [6]:
```
import pandas as pd
import cartopy.crs as ccrs
import matplotlib.pyplot as plt
import random
import numpy as np
from math import radians, cos, sin, asin, sqrt
```

In [7]:
```
df = pd.read_csv("worldcities.csv")
df = df[['city', 'lat', 'lng']]
df.head()
```

Out[7]:

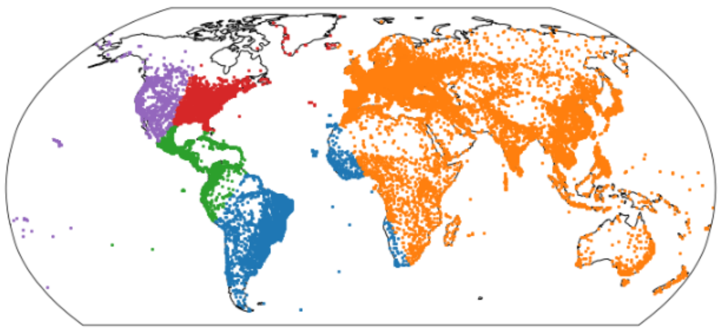
	city	lat	lng
0	Tokyo	35.6839	139.7744
1	Jakarta	-6.2146	106.8451
2	Delhi	28.6667	77.2167
3	Manila	14.6000	120.9833
4	São Paulo	-23.5504	-46.6339

In [8]:
```
# code from https://stackoverflow.com/questions/4913349/haversine-formula-in-python-bearing-and-distance-between-two-gps-points

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 return value units.
    return c * r
```

In [9]:
```
def kmeans(k):
    pts = [np.array(pt) for pt in zip(df['lng'], df['lat'])]
    centers = random.sample(pts, k)
    old_cluster_ids, cluster_ids = None, [] # arbitrary but different
    while cluster_ids != old_cluster_ids: #### change to do while?
        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(pt - center)
                if dist < min_dist:
                    min_cluster = i
                    min_dist = dist
            cluster_ids.append(min_cluster)
        df['cluster'] = cluster_ids
#     df['cluster'] = df['cluster'].astype('category')
        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]

    fig = plt.figure(figsize=(12, 8))
    ax = fig.add_subplot(1, 1, 1, projection=ccrs.Robinson())
    ax.coastlines()
    for i in range (k):
        ax.plot(df[df['cluster'] == i]['lng'], df[df['cluster'] == i]['lat'], "o", transform=ccrs.PlateCarree(),markersize=2)
```

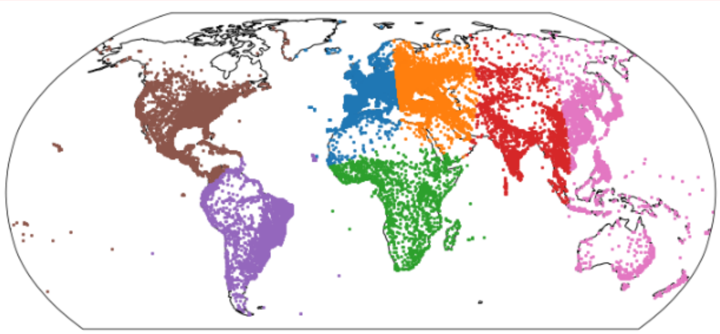
In [10]:
```
kmeans(5)
```

```
D:\Anaconda\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 of the `geoms` property instead to get the  number of parts of a multi-part geometry.
  if len(multi_line_string) > 1:
D:\Anaconda\lib\site-packages\cartopy\crs.py:297: ShapelyDeprecationWarning: Iteration 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.
  for line in multi_line_string:
D:\Anaconda\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 of the `geoms` property instead to get the  number of parts of a multi-part geometry.
  if len(p_mline) > 0:
```



In [11]:
```
kmeans(7)
```

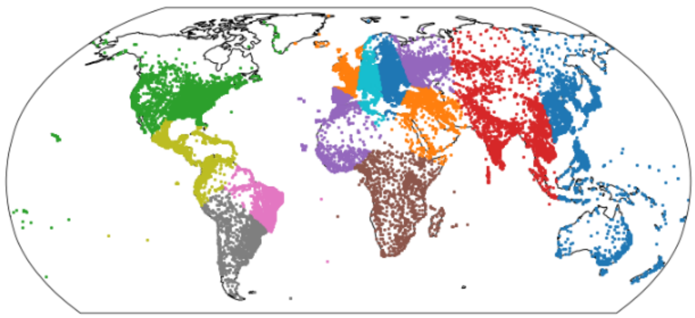
```
D:\Anaconda\lib\site-packages\cartopy\crs.py:245: ShapelyDeprecationWarning: __len__ for multi-part geometries is deprecated and will be removed in Shapely 2.0. Check the length of the `geoms` property instead to get the  number of parts of a multi-part geometry.
  if len(multi_line_string) > 1:
D:\Anaconda\lib\site-packages\cartopy\crs.py:297: ShapelyDeprecationWarning: Iteration 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.
  for line in multi_line_string:
D:\Anaconda\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 of the `geoms` property instead to get the  number of parts of a multi-part geometry.
  if len(p_mline) > 0:
```



In [12]:

```
kmeans(15)
```

```
D:\Anaconda\lib\site-packages\cartopy\crs.py:245: ShapelyDeprecationWarning: __len__ for multi-part geometries is deprecated a
nd will be removed in Shapely 2.0. Check the length of the `geoms` property instead to get the  number of parts of a multi-par
t geometry.
  if len(multi_line_string) > 1:
D:\Anaconda\lib\site-packages\cartopy\crs.py:297: ShapelyDeprecationWarning: Iteration over multi-part geometries is deprecate
d and will be removed in Shapely 2.0. Use the `geoms` property to access the constituent parts of a multi-part geometry.
  for line in multi_line_string:
D:\Anaconda\lib\site-packages\cartopy\crs.py:364: ShapelyDeprecationWarning: __len__ for multi-part geometries is deprecated a
nd will be removed in Shapely 2.0. Check the length of the `geoms` property instead to get the  number of parts of a multi-par
t geometry.
  if len(p_mline) > 0:
```



EX3

In [1]:
```
import time
import plotnine as p9
import pandas as pd
from tqdm import tqdm
import matplotlib.pyplot as plt
from functools import lru_cache
```

In [2]:
```
# naive solution
def naive_fib(n):
    if n <2:
        return n
    return naive_fib(n - 1) + naive_fib(n - 2)
```

In [3]:
```
# test
print(naive_fib(4))
print(naive_fib(7))
print(naive_fib(15))
print(naive_fib(18))
```

```
3
13
610
2584
```

In [4]:
```
@lru_cache()
def lru_cache_fib(n):
    if n < 2:
        return n
    else:
        return lru_cache_fib(n-1) + lru_cache_fib(n-2)
```

In [5]:
```
# test
print(lru_cache_fib(4))
print(lru_cache_fib(7))
print(lru_cache_fib(15))
print(lru_cache_fib(18))
```

```
3
13
610
2584
```

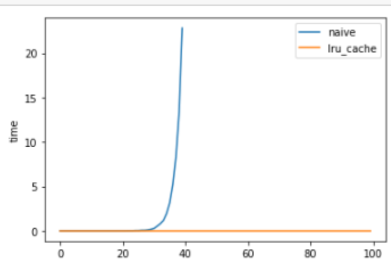
In [97]:
```
time_naive = []

for n in tqdm(range(40)):
    start=time.time()
    runtime=naive_fib(n)
    time_naive.append(time.time()-start)
```

```
100%|████████████████████████████████████████████████████████████████████████████| 40/40 [00:58<00:00,  1.45s/it]
```

In [6]:
```
time_lru_cache = []

for n in tqdm(range(100)):
    start=time.time()
    runtime=lru_cache_fib(n)
    time_lru_cache.append(time.time()-start)
```

```
100%|████████████████████████████████████████████████████████████████| 100/100 [00:00<00:00, 100294.21it/s]
```

In [123]:
```
plt.plot(range(40),time_naive, label="naive")
plt.plot(range(100),time_lru_cache, label="lru_cache")
plt.legend()
plt.xlabel("n")
plt.ylabel("time")
plt.show()
```



EX4

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
```

In [29]:
```
def smith_waterman(seq1,seq2,match=1,gap_penalty=1,mismatch_penalty=1):
    matrix=np.zeros((len(seq1)+1, len(seq2)+1))
    for i in range(1, len(seq1) + 1):
        for j in range(1, len(seq2) + 1):
            if seq1[i-1]==seq2[j-1]:
                score = match
                matrix[i][j]=score
            else:
                score =-mismatch_penalty
                matrix[i][j]=score
            matrix[i][j] = max(0, matrix[i-1][j-1] + score, matrix[i][j-1] - gap_penalty, matrix[i-1][j] - gap_penalty)

    for i in range(1, len(seq1) + 1):
        for j in range(1, len(seq2) + 1):
            if matrix[i][j] == matrix.max():
                max_i,max_j=i,j

    subseq1 = ''
    subseq2 = ''
    while matrix[max_i][max_j] != 0:
        score_max = matrix[max_i][max_j]
        score_up_left = matrix[max_i-1][max_j-1]
        score_up = matrix[max_i][max_j-1]
        score_left = matrix[max_i-1][max_j]
```