Aerial-Bombing-Operations-and-Weather-Conditions-Data-Analysis-in-World-War-II

In this project, we will use data sources related to aerial bombing operations and weather conditions during World War 2. After this point, we will use the acronym 'WW2' for World War 2. We will begin with data cleaning, followed by data visualization to gain a better understanding. These processes can be referred to as 'EDA' (Exploratory Data Analysis).

Afterward, our focus will shift to time series prediction for forecasting when bombing operations occurred. For time series prediction, we will utilize the 'ARIMA' method.

In [1]:
```
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
import plotly.plotly as py
from plotly.offline import init_notebook_mode, iplot
init_notebook_mode(connected=True)
import plotly.graph_objs as go

import os
print(os.listdir("../input"))

import warnings
warnings.filterwarnings("ignore")
plt.style.use('ggplot')
```

```
['weatherww2', 'world-war-ii']
```

Load the Data

In [2]:
```
# bombing data
aerial = pd.read_csv("../input/world-war-ii/operations.csv")
# first weather data that includes locations like country, latitude and longitude.
weather_station_location = pd.read_csv("../input/weatherww2/Weather Station Locations.csv")
# Second weather data that includes measured min, max and mean temperatures
weather = pd.read_csv("../input/weatherww2/Summary of Weather.csv")
```

Data Cleaning

Aerial Bombing data includes a lot of NaN value.

* Drop countries that are NaN
* Drop if target longitude is NaN
* Drop if takeoff longitude is NaN
* Drop unused features

In [3]:
```
# drop countries that are NaN
aerial = aerial[pd.isna(aerial.Country)==False]
# drop if target longitude is NaN
aerial = aerial[pd.isna(aerial['Target Longitude'])==False]
# Drop if takeoff longitude is NaN
aerial = aerial[pd.isna(aerial['Takeoff Longitude'])==False]
# drop unused features
drop_list = ['Mission ID','Unit ID','Target ID','Altitude (Hundreds of Feet)','Airborne Aircraft',
             'Attacking Aircraft', 'Bombing Aircraft', 'Aircraft Returned',
             'Aircraft Failed', 'Aircraft Damaged', 'Aircraft Lost',
             'High Explosives', 'High Explosives Type','Mission Type',
             'High Explosives Weight (Pounds)', 'High Explosives Weight (Tons)',
             'Incendiary Devices', 'Incendiary Devices Type',
             'Incendiary Devices Weight (Pounds)',
             'Incendiary Devices Weight (Tons)', 'Fragmentation Devices',
             'Fragmentation Devices Type', 'Fragmentation Devices Weight (Pounds)',
             'Fragmentation Devices Weight (Tons)', 'Total Weight (Pounds)',
             'Total Weight (Tons)', 'Time Over Target', 'Bomb Damage Assessment','Source ID']
aerial.drop(drop_list, axis=1,inplace = True)
aerial = aerial[ aerial.iloc[:,8]!="4248"] # drop this takeoff latitude
aerial = aerial[ aerial.iloc[:,9]!=1355]   # drop this takeoff longitude
```

In [4]:
```
aerial.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 2555 entries, 0 to 178080
Data columns (total 17 columns):
Mission Date            2555 non-null object
Theater of Operations   2555 non-null object
Country                 2555 non-null object
Air Force               2505 non-null object
Aircraft Series         2528 non-null object
Callsign                10 non-null object
Takeoff Base            2555 non-null object
Takeoff Location        2555 non-null object
Takeoff Latitude        2555 non-null object
Takeoff Longitude       2555 non-null float64
Target Country          2499 non-null object
Target City             2552 non-null object
Target Type             602 non-null object
Target Industry         81 non-null object
Target Priority         230 non-null object
Target Latitude         2555 non-null float64
Target Longitude        2555 non-null float64
dtypes: float64(3), object(14)
memory usage: 359.3+ KB
```

In [5]:
```
# The features we use
weather_station_location = weather_station_location.loc[:,["WBAN","NAME","STATE/COUNTRY ID","Latitude","Longitude"] ]
weather_station_location.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 161 entries, 0 to 160
Data columns (total 5 columns):
WBAN                161 non-null int64
NAME                161 non-null object
STATE/COUNTRY ID    161 non-null object
Latitude            161 non-null float64
Longitude           161 non-null float64
dtypes: float64(2), int64(1), object(2)
memory usage: 6.4+ KB
```

In [6]:
```
# The features we use
weather = weather.loc[:,["STA","Date","MeanTemp"] ]
weather.info()
```

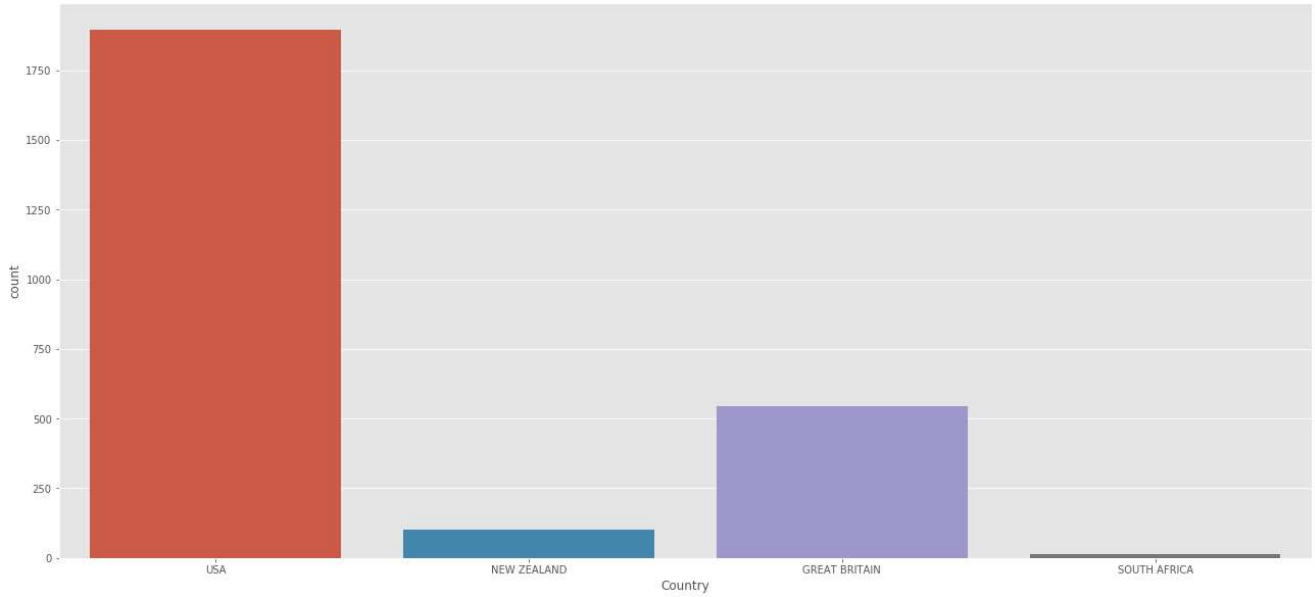
```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 119040 entries, 0 to 119039
Data columns (total 3 columns):
STA         119040 non-null int64
Date        119040 non-null object
MeanTemp    119040 non-null float64
dtypes: float64(1), int64(1), object(1)
memory usage: 2.7+ MB
```

Data Visualization

* How many country which attacks
* Top target countries
* Top 10 aircraft series
* Takeoff base locations (Attacjk countries)
* Target locations
* Bombing paths
* Wheater of Operations
* Weather station locations

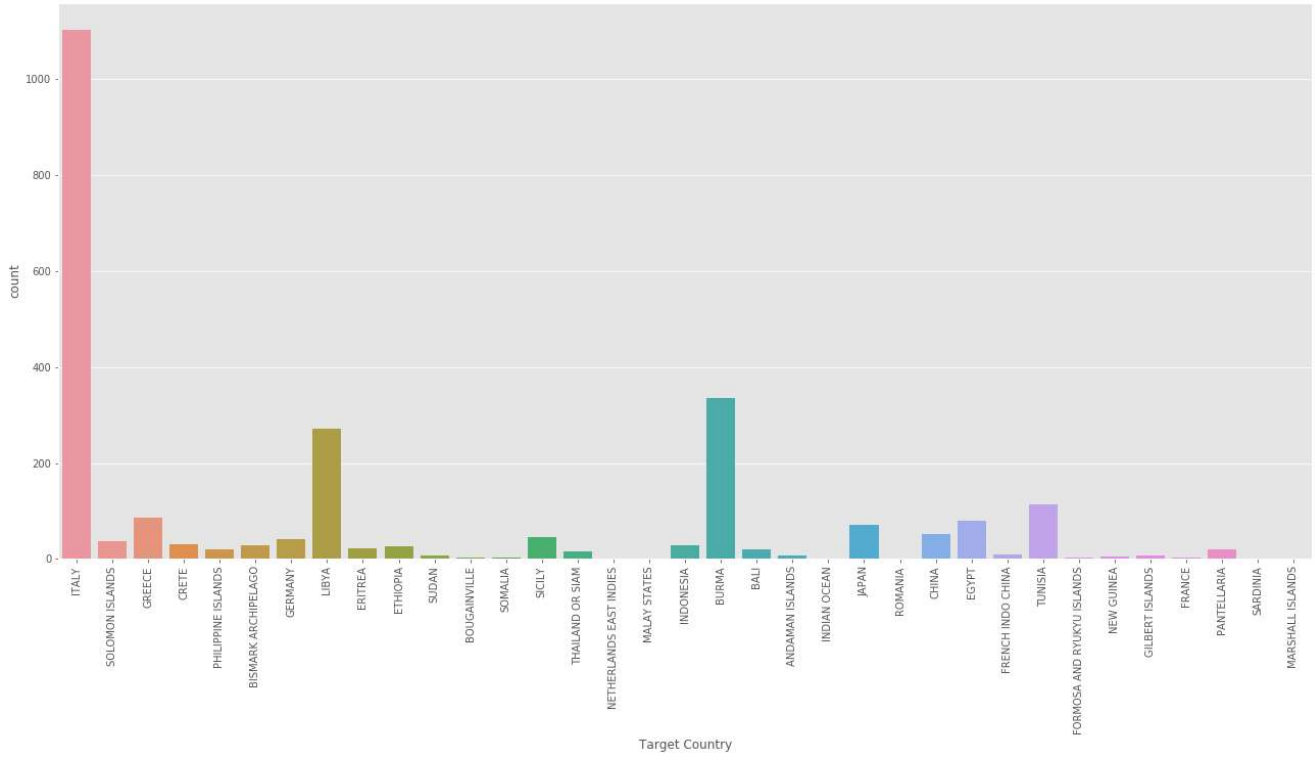
In [7]:
```
# country
print(aerial['Country'].value_counts())
plt.figure(figsize=(22,10))
sns.countplot(aerial['Country'])
plt.show()
```

```
USA             1895
GREAT BRITAIN     544
NEW ZEALAND       102
SOUTH AFRICA       14
Name: Country, dtype: int64
```



In [8]:
```
# Top target countries
print(aerial['Target Country'].value_counts()[:10])
plt.figure(figsize=(22,10))
sns.countplot(aerial['Target Country'])
plt.xticks(rotation=90)
plt.show()
```

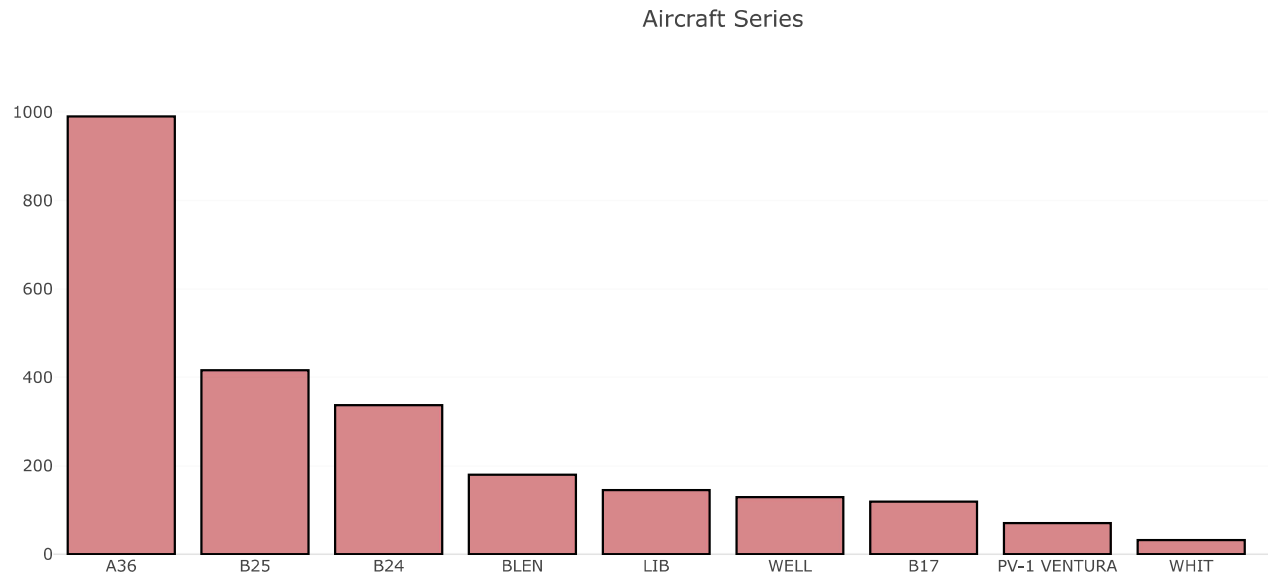
```
ITALY      1104
BURMA       335
LIBYA       272
TUNISIA     113
GREECE       87
EGYPT        80
JAPAN        71
CHINA        52
SICILY       46
GERMANY      41
Name: Target Country, dtype: int64
```



In [9]:
```
# Aircraft Series
data = aerial['Aircraft Series'].value_counts()
print(data[:10])
data = [go.Bar(
            x=data[:10].index,
            y=data[:10].values,
            hoverinfo = 'text',
```

```
            marker = dict(color = 'rgba(177, 14, 22, 0.5)',
                          line=dict(color='rgb(0,0,0)',width=1.5)),
    )]

layout = dict(
    title = 'Aircraft Series',
)
fig = go.Figure(data=data, layout=layout)
iplot(fig)
```

```
A36             990
B25             416
B24             337
BLEN            180
LIB             145
WELL            129
B17             119
PV-1 VENTURA     70
WHIT             32
HALI             18
Name: Aircraft Series, dtype: int64
```



Visualize take off bases of countries who attack

In plot below, blue color is USA and red color is Great Britain

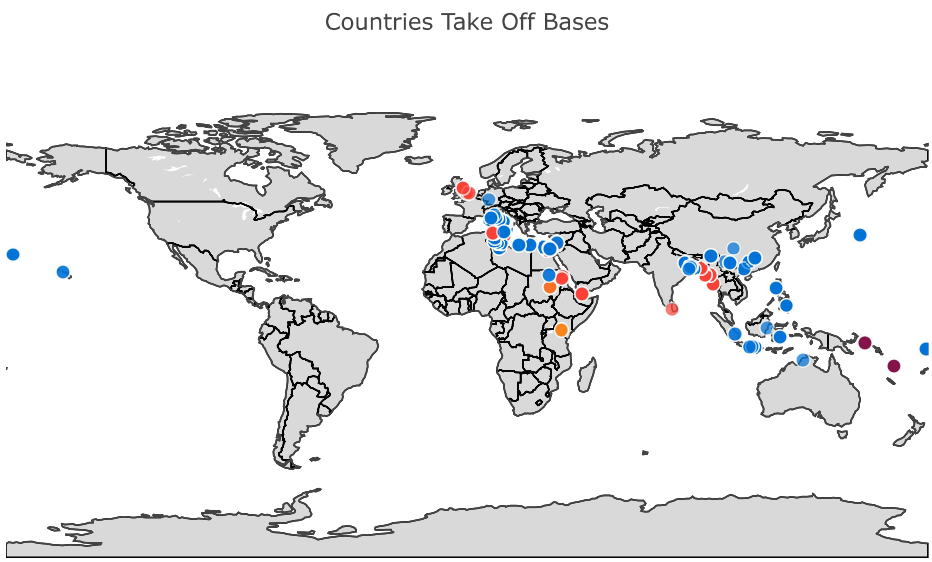
In [10]:
```
aerial.head()
```

Out[10]:

	Mission Date	Theater of Operations	Country	Air Force	Aircraft Series	Callsign	Takeoff Base	Takeoff Location	Takeoff Latitude	Takeoff Longitude	Target Country	Target City	Target Type	Target Industry	Target Priority	Targe Latitud
0	8/15/1943	MTO	USA	12 AF	A36	NaN	PONTE OLIVO AIRFIELD	SICILY	37.131022	14.321464	ITALY	SPADAFORA	NaN	NaN	NaN	38.2
2	8/15/1943	MTO	USA	12 AF	A36	NaN	PONTE OLIVO AIRFIELD	SICILY	37.131022	14.321464	ITALY	COSENZA	NaN	NaN	NaN	39.2
3	8/15/1943	MTO	USA	12 AF	A36	NaN	PONTE OLIVO AIRFIELD	SICILY	37.131022	14.321464	ITALY	GIOJA TAURO	NaN	NaN	NaN	38.4
8	8/15/1943	MTO	USA	12 AF	A36	NaN	PONTE OLIVO AIRFIELD	SICILY	37.131022	14.321464	ITALY	SCILLA	NaN	NaN	NaN	38.2
9	8/15/1943	MTO	USA	12 AF	A36	NaN	PONTE OLIVO AIRFIELD	SICILY	37.131022	14.321464	ITALY	GIOJA TAURO	NaN	ARMAMENT AND ORDNANCE PLANTS	NaN	38.4

In [11]:
```
# ATTACK
aerial["color"] = ""
aerial.color[aerial.Country == "USA"] = "rgb(0,116,217)"
aerial.color[aerial.Country == "GREAT BRITAIN"] = "rgb(255,65,54)"
```

```
aerial.color[aerial.Country == "NEW ZEALAND"] = "rgb(133,20,75)"
aerial.color[aerial.Country == "SOUTH AFRICA"] = "rgb(255,133,27)"

data = [dict(
    type='scattergeo',
    lon = aerial['Takeoff Longitude'],
    lat = aerial['Takeoff Latitude'],
    hoverinfo = 'text',
    text = "Country: " + aerial.Country + " Takeoff Location: "+aerial["Takeoff Location"]+" Takeoff Base: " + aerial['Takeoff Base'],
    mode = 'markers',
    marker=dict(
        sizemode = 'area',
        sizeref = 1,
        size= 10 ,
        line = dict(width=1,color = "white"),
        color = aerial["color"],
        opacity = 0.7),
)]
layout = dict(
    title = 'Countries Take Off Bases ',
    hovermode='closest',
    geo = dict(showframe=False, showland=True, showcoastlines=True, showcountries=True,
               countrywidth=1, projection=dict(type='Mercator'),
              landcolor = 'rgb(217, 217, 217)',
              subunitwidth=1,
              showlakes = True,
              lakecolor = 'rgb(255, 255, 255)',
              countrycolor="rgb(5, 5, 5)")
)
fig = go.Figure(data=data, layout=layout)
iplot(fig)
```

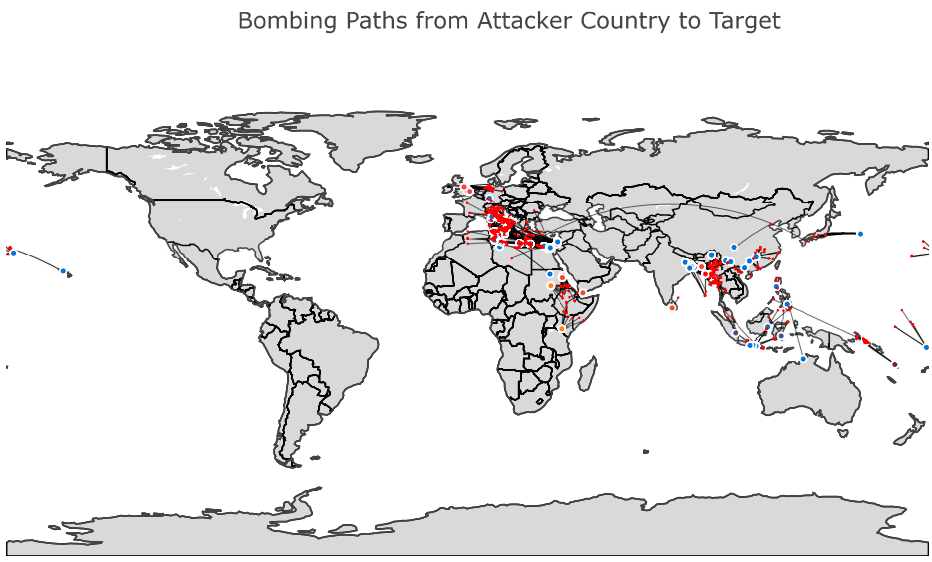


Let's visualize the bombing paths, including which countries the aircraft took off from, which countries they bombed, and which cities were targeted

In [12]:

```
# Bombing paths
# trace1
airports = [ dict(
        type = 'scattergeo',
        lon = aerial['Takeoff Longitude'],
        lat = aerial['Takeoff Latitude'],
        hoverinfo = 'text',
        text = "Country: " + aerial.Country + " Takeoff Location: "+aerial["Takeoff Location"]+" Takeoff Base: " + aerial['Takeoff Base'],
        mode = 'markers',
        marker = dict(
            size=5,
            color = aerial["color"],
            line = dict(
                width=1,
                color = "white"
            )
        ))]

# trace2
targets = [ dict(
        type = 'scattergeo',
        lon = aerial['Target Longitude'],
```

```
        lat = aerial['Target Latitude'],
        hoverinfo = 'text',
        text = "Target Country: "+aerial["Target Country"]+" Target City: "+aerial["Target City"],
        mode = 'markers',
        marker = dict(
            size=1,
            color = "red",
            line = dict(
                width=0.5,
                color = "red"
            )
        ))]

# trace3
flight_paths = []
for i in range( len( aerial['Target Longitude'] ) ):
    flight_paths.append(
        dict(
            type = 'scattergeo',
            lon = [ aerial.iloc[i,9], aerial.iloc[i,16] ],
            lat = [ aerial.iloc[i,8], aerial.iloc[i,15] ],
            mode = 'lines',
            line = dict(
                width = 0.7,
                color = 'black',
            ),
            opacity = 0.6,
        )
    )

layout = dict(
    title = 'Bombing Paths from Attacker Country to Target ',
    hovermode='closest',
    geo = dict(showframe=False, showland=True, showcoastlines=True, showcountries=True,
               countrywidth=1, projection=dict(type='Mercator'),
              landcolor = 'rgb(217, 217, 217)',
              subunitwidth=1,
              showlakes = True,
              lakecolor = 'rgb(255, 255, 255)',
              countrycolor="rgb(5, 5, 5)")
)

fig = dict( data=flight_paths + airports+targets, layout=layout )
iplot( fig )
```



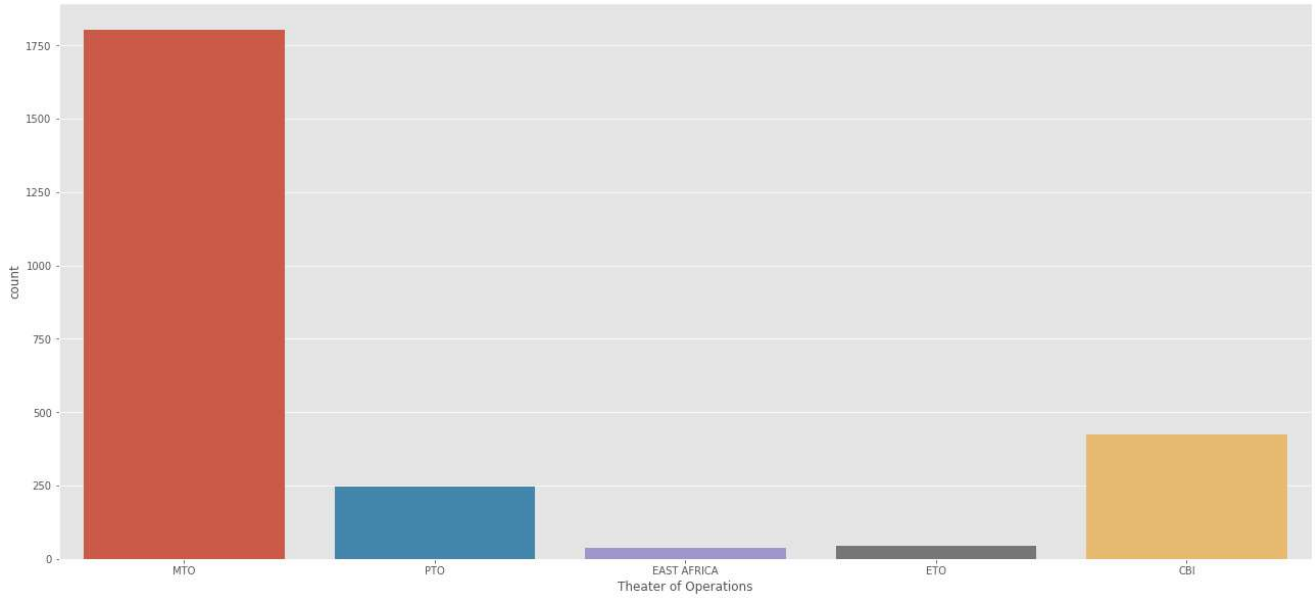
From bombing paths, most of the bombing attack is done in Mediterranean theater of operations.

Theater of Operations:

- ETO: European Theater of Operations
- PTO: Pasific Theater of Operations
- MTO: Mediterranean Theater of Operations
- CBI: China-Burma-India Theater of Operations
- EAST AFRICA: East Africa Theater of Operations

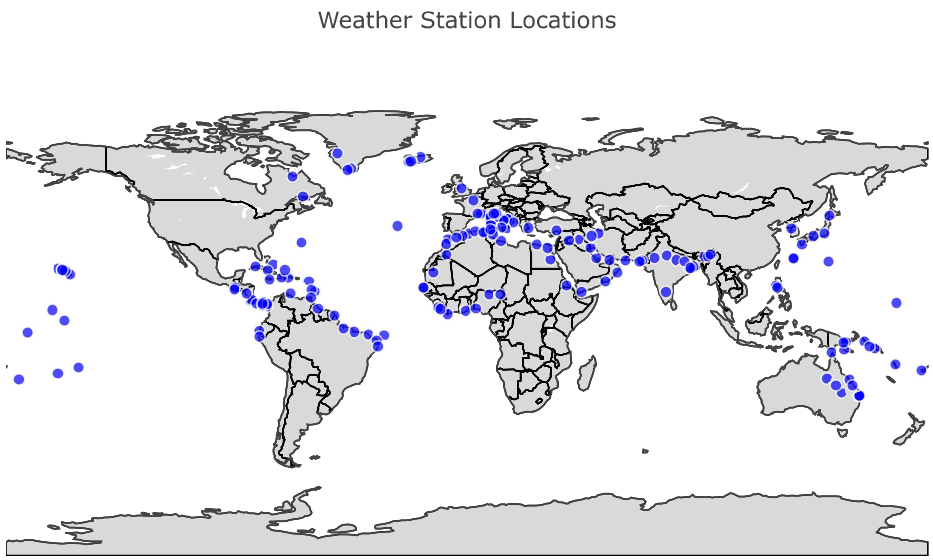
In [13]:
```
#Theater of Operations
print(aerial['Theater of Operations'].value_counts())
plt.figure(figsize=(22,10))
sns.countplot(aerial['Theater of Operations'])
plt.show()
```

```
MTO            1802
CBI             425
PTO             247
ETO              44
EAST AFRICA      37
Name: Theater of Operations, dtype: int64
```



Weather station locations

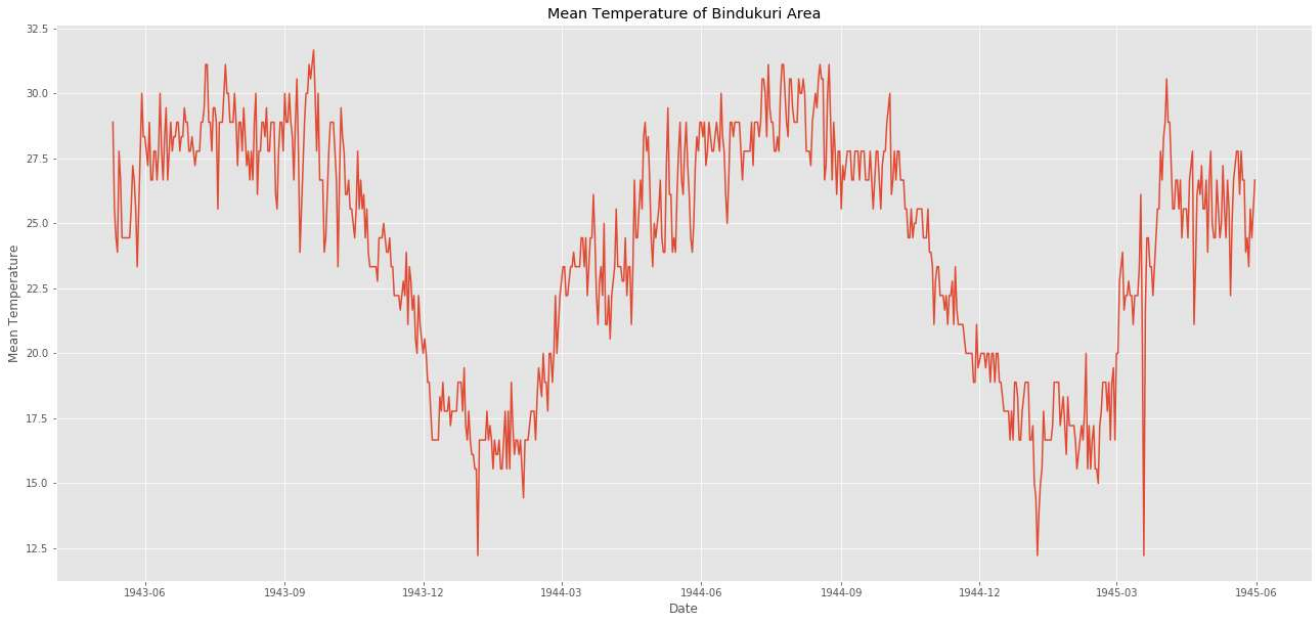
In [14]:
```
data = [dict(
    type='scattergeo',
    lon = weather_station_location.Longitude,
    lat = weather_station_location.Latitude,
    hoverinfo = 'text',
    text = "Name: " + weather_station_location.NAME + " Country: " + weather_station_location["STATE/COUNTRY ID"],
    mode = 'markers',
    marker=dict(
        sizemode = 'area',
        sizeref = 1,
        size= 8 ,
        line = dict(width=1,color = "white"),
        color = "blue",
        opacity = 0.7),
)]

layout = dict(
    title = 'Weather Station Locations ',
    hovermode='closest',
    geo = dict(showframe=False, showland=True, showcoastlines=True, showcountries=True,
               countrywidth=1, projection=dict(type='Mercator'),
              landcolor = 'rgb(217, 217, 217)',
              subunitwidth=1,
              showlakes = True,
              lakecolor = 'rgb(255, 255, 255)',
              countrycolor="rgb(5, 5, 5)")
)

fig = go.Figure(data=data, layout=layout)
iplot(fig)
```



"Let's focus on the USA and Burma War."

"In this conflict, the USA bombed Burma (specifically the city of Katha) from 1942 to 1945." "The closest weather station to this war is Bindukuri, which has temperature records from 1943 to 1945." "Now, let's visualize this situation. However, before visualization, we need to convert the date features into datetime objects."

In [15]:
```
weather_station_id = weather_station_location[weather_station_location.NAME == "BINDUKURI"].WBAN
weather_bin = weather[weather.STA == 32907]
weather_bin["Date"] = pd.to_datetime(weather_bin["Date"])
plt.figure(figsize=(22,10))
plt.plot(weather_bin.Date,weather_bin.MeanTemp)
plt.title("Mean Temperature of Bindukuri Area")
plt.xlabel("Date")
plt.ylabel("Mean Temperature")
plt.show()
```



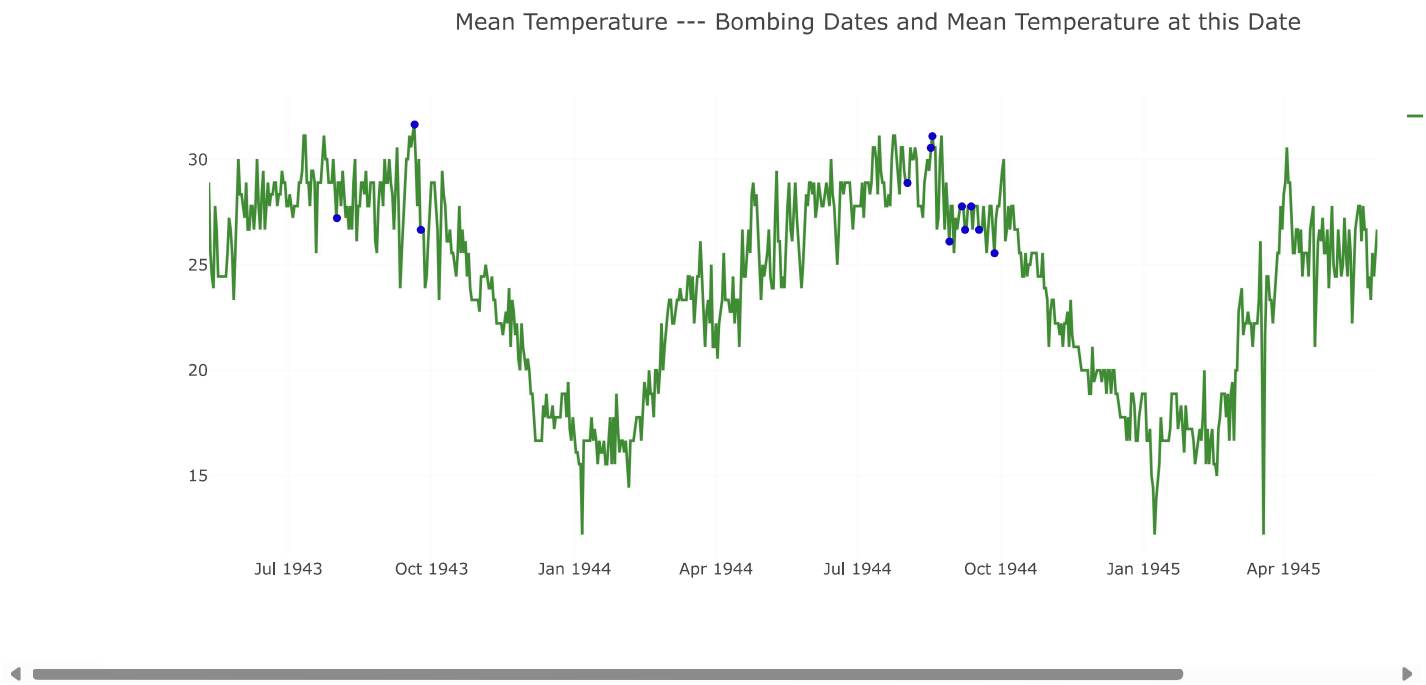
"We have temperature measurements from 1943 to 1945."

"The temperature oscillates between 12 and 32 degrees." "The temperature in winter months is colder than the temperature in summer months."

In [16]:
```
aerial = pd.read_csv("../input/world-war-ii/operations.csv")
aerial["year"] = [ each.split("/")[2] for each in aerial["Mission Date"]]
aerial["month"] = [ each.split("/")[0] for each in aerial["Mission Date"]]
aerial = aerial[aerial["year"]>="1943"]
aerial = aerial[aerial["month"]>="8"]
```

```
aerial["Mission Date"] = pd.to_datetime(aerial["Mission Date"])

attack = "USA"
target = "BURMA"
city = "KATHA"

aerial_war = aerial[aerial.Country == attack]
aerial_war = aerial_war[aerial_war["Target Country"] == target]
aerial_war = aerial_war[aerial_war["Target City"] == city]
```

In [17]:
```
liste = []
aa = []
for each in aerial_war["Mission Date"]:
    dummy = weather_bin[weather_bin.Date == each]
    liste.append(dummy["MeanTemp"].values)
aerial_war["dene"] = liste
for each in aerial_war.dene.values:
    aa.append(each[0])

# Create a trace
trace = go.Scatter(
    x = weather_bin.Date,
    mode = "lines",
    y = weather_bin.MeanTemp,
    marker = dict(color = 'rgba(16, 112, 2, 0.8)'),
    name = "Mean Temperature"
)
trace1 = go.Scatter(
    x = aerial_war["Mission Date"],
    mode = "markers",
    y = aa,
    marker = dict(color = 'rgba(16, 0, 200, 1)'),
    name = "Bombing temperature"
)
layout = dict(title = 'Mean Temperature --- Bombing Dates and Mean Temperature at this Date')
data = [trace,trace1]

fig = dict(data = data, layout = layout)
iplot(fig)
```



"The green line represents the mean temperature measured in Bindukuri."

"The blue markers indicate bombing dates and the corresponding temperature on those dates." "As seen from the plot, the USA carried out bombing missions during high-temperature periods." "The question is, can we predict future weather, and based on this prediction, determine whether bombings will occur or not." "To address this question, let's start with time series prediction."

Time Series Prediction with ARIMA

ARIMA : AutoRegressive Integrated Moving Average.

The way that we will follow:

* Time Series.
* Stationarity of a Time Series
* Forecasting a Time Series

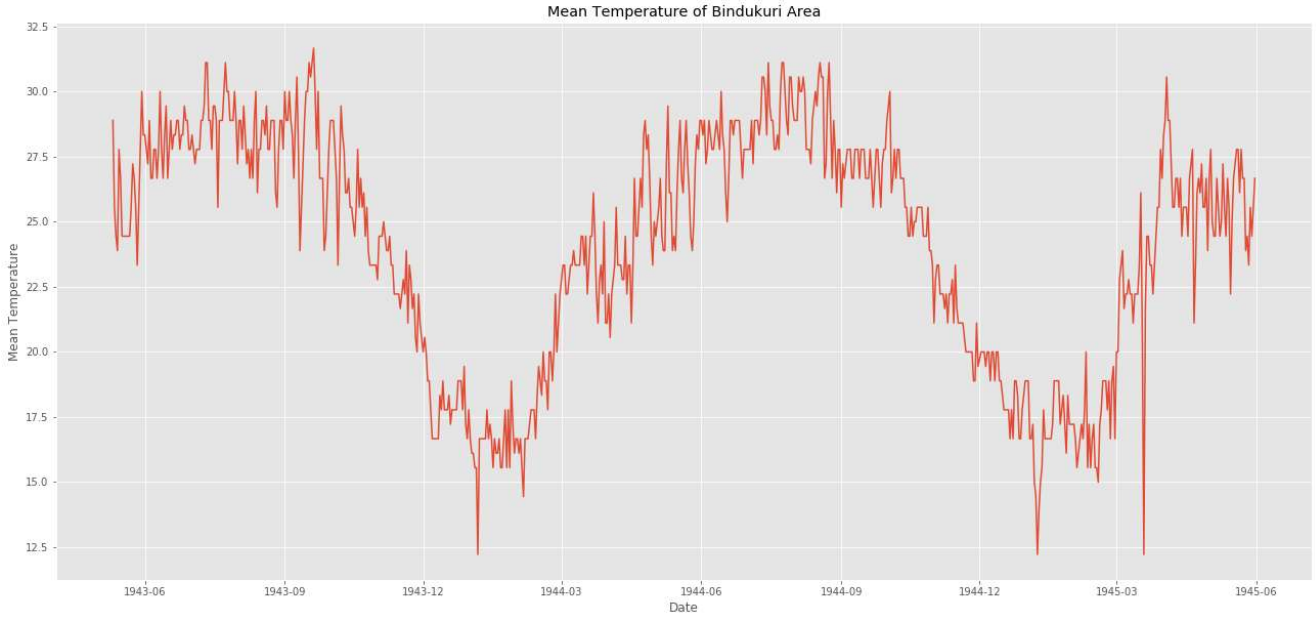
What is time series?

"A time series is a collection of data points that are recorded at constant time intervals." "It is time-dependent." "Most time series data exhibit some form of seasonal trends. For instance, if we sell ice cream, it's highly likely that there will be higher sales during the summer seasons. Therefore, this time series displays seasonal trends."

Stationarity of a Time Series

- There are three basic criterion for a time series to understand whether it is stationary series or not.
  - Statistical properties of time series such as mean, variance should remain constant over time to call **time series is stationary**

In [18]:
```
# Mean temperature of Bindikuri area
plt.figure(figsize=(22,10))
plt.plot(weather_bin.Date,weather_bin.MeanTemp)
plt.title("Mean Temperature of Bindukuri Area")
plt.xlabel("Date")
plt.ylabel("Mean Temperature")
plt.show()

# Create time series from weather
timeSeries = weather_bin.loc[:, ["Date","MeanTemp"]]
timeSeries.index = timeSeries.Date
ts = timeSeries.drop("Date",axis=1)
```

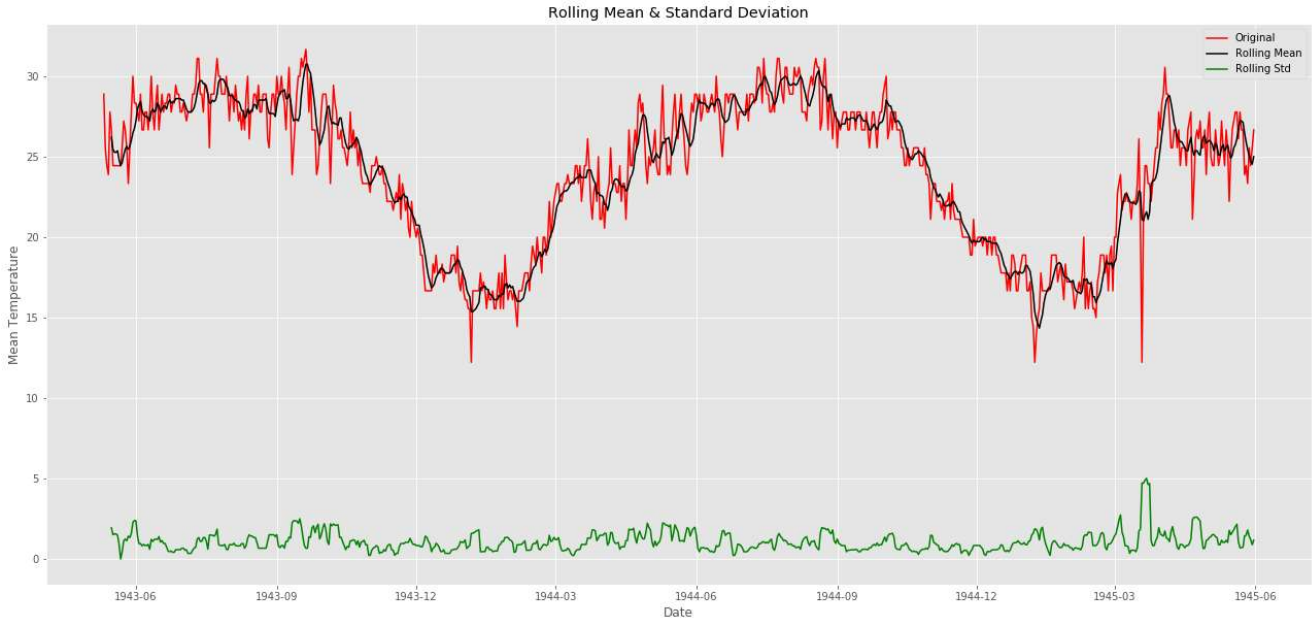


"As you can see from the plot above, our time series exhibits seasonal variation. In summer, the mean temperature is higher, and in winter, the mean temperature is lower for each year."

"Now, let's assess the stationarity of the time series. We can check stationarity using the following methods:" "Plotting Rolling Statistics: We use a window, let's say with a size of 6, and then calculate the rolling mean and variance to check for stationarity." "Dickey-Fuller Test: The test results include a Test Statistic and some Critical Values for different confidence levels. If the test statistic is less than the critical value, we can conclude that the time series is stationary."

In [19]:
```
# adfuller library
from statsmodels.tsa.stattools import adfuller
# check_adfuller
def check_adfuller(ts):
    # Dickey-Fuller test
    result = adfuller(ts, autolag='AIC')
    print('Test statistic: ' , result[0])
    print('p-value: '  ,result[1])
    print('Critical Values:' ,result[4])
# check_mean_std
def check_mean_std(ts):
    #Rolling statistics
    rolmean = pd.rolling_mean(ts, window=6)
    rolstd = pd.rolling_std(ts, window=6)
    plt.figure(figsize=(22,10))
    orig = plt.plot(ts, color='red',label='Original')
    mean = plt.plot(rolmean, color='black', label='Rolling Mean')
    std = plt.plot(rolstd, color='green', label = 'Rolling Std')
    plt.xlabel("Date")
    plt.ylabel("Mean Temperature")
    plt.title('Rolling Mean & Standard Deviation')
```

```
    plt.legend()
    plt.show()

# check stationary: mean, variance(std)and adfuller test
check_mean_std(ts)
check_adfuller(ts.MeanTemp)
```




```
Test statistic:  -1.409596674588756
p-value:  0.5776668028526452
Critical Values: {'1%': -3.439229783394421, '5%': -2.86545894814762, '10%': -2.5688568756191392}
```

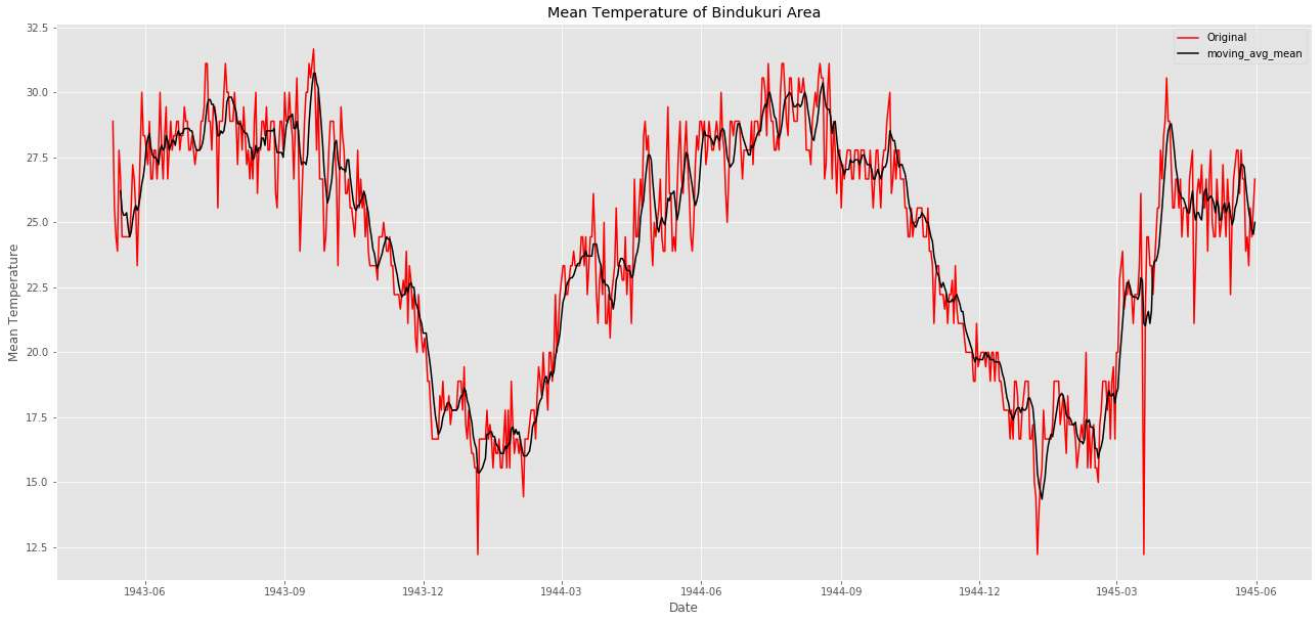
"Our first criterion for stationarity is a constant mean, which we fail because, as you can see from the plot (black line) above, the mean is not constant (not stationary)."

"The second criterion is a constant variance, and it appears to be constant (stationary)." "The third criterion is whether the test statistic is less than the critical value. Let's take a look:" "Test statistic = -1.4, and the critical values are {'1%': -3.439229783394421, '5%': -2.86545894814762, '10%': -2.5688568756191392}. The test statistic is greater than the critical values (not stationary)." "In conclusion, our time series is not stationary."

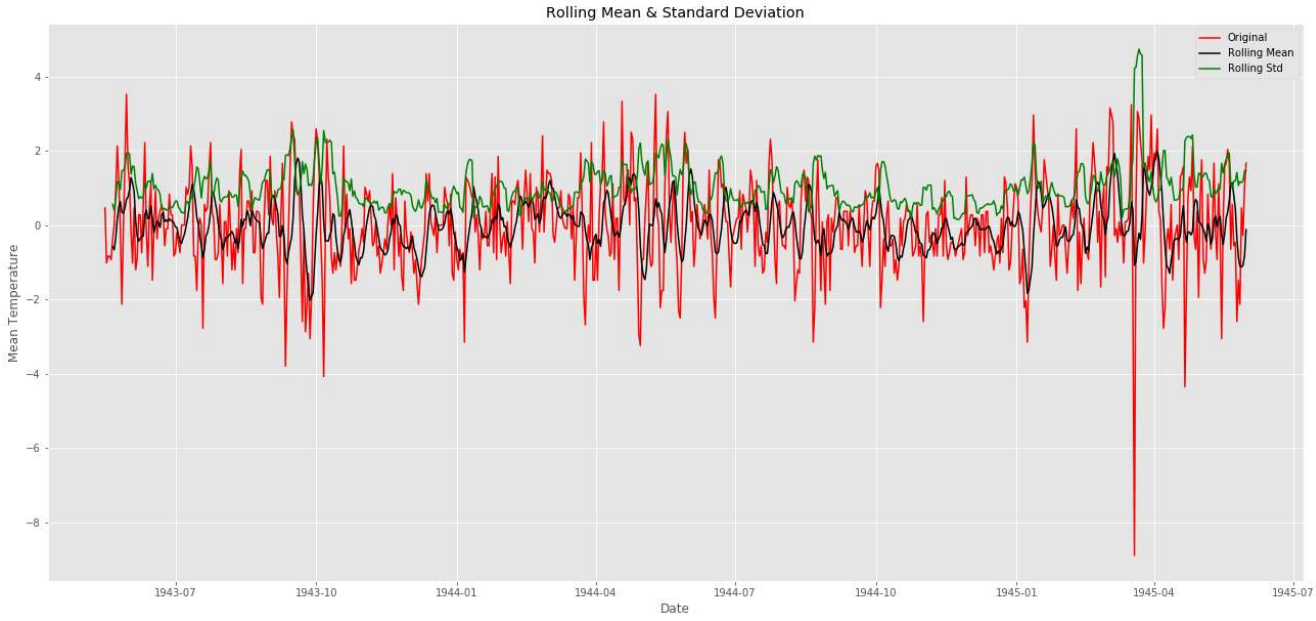
"To Make a Time Series Stationary"

"As mentioned before, there are two reasons behind the non-stationarity of a time series:" "Trend: varying mean over time. We need a constant mean for the stationarity of a time series." "Seasonality: variations at specific times. We need constant variations for the stationarity of a time series." "First, solve the trend (constant mean) problem." "The most popular method is the moving average." "Moving average: We have a window that takes the average over the past 'n' samples. 'n' is the window size."

In [20]:
```
# Moving average method

window_size = 6
moving_avg = pd.rolling_mean(ts,window_size)
plt.figure(figsize=(22,10))
plt.plot(ts, color = "red",label = "Original")
plt.plot(moving_avg, color='black', label = "moving_avg_mean")
plt.title("Mean Temperature of Bindukuri Area")
plt.xlabel("Date")
plt.ylabel("Mean Temperature")
plt.legend()
plt.show()
```



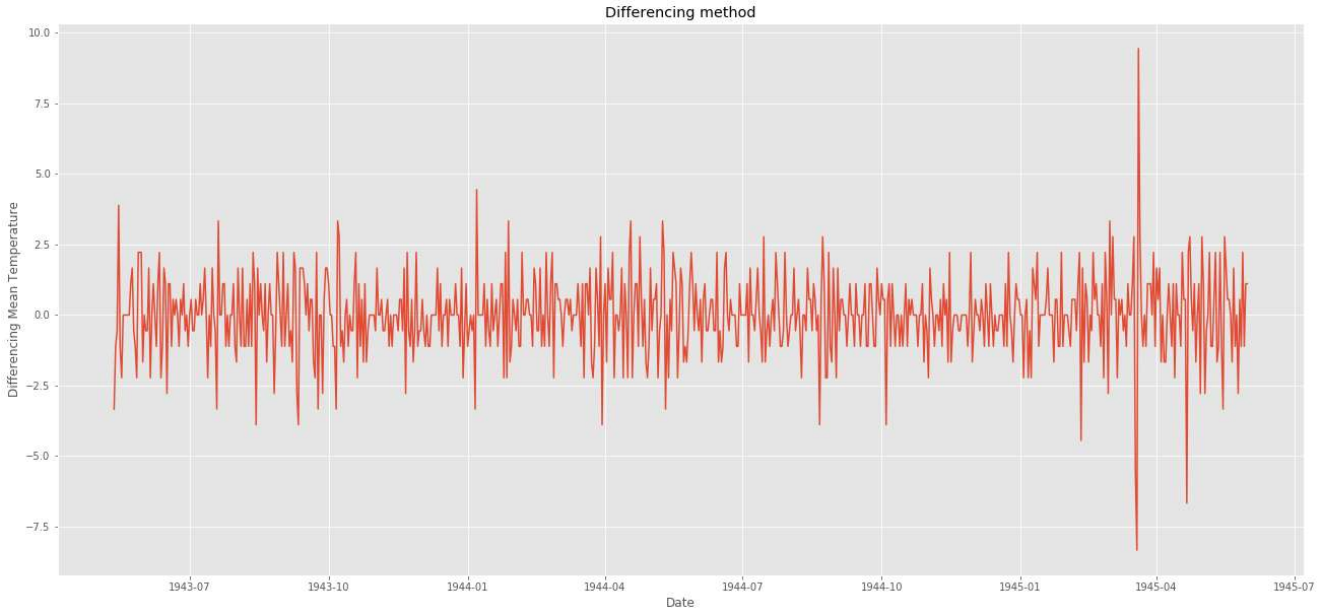
In [21]:
```
ts_moving_avg_diff = ts - moving_avg
ts_moving_avg_diff.dropna(inplace=True) # first 6 is nan value due to window size

# check stationary: mean, variance(std)and adfuller test
check_mean_std(ts_moving_avg_diff)
check_adfuller(ts_moving_avg_diff.MeanTemp)
```



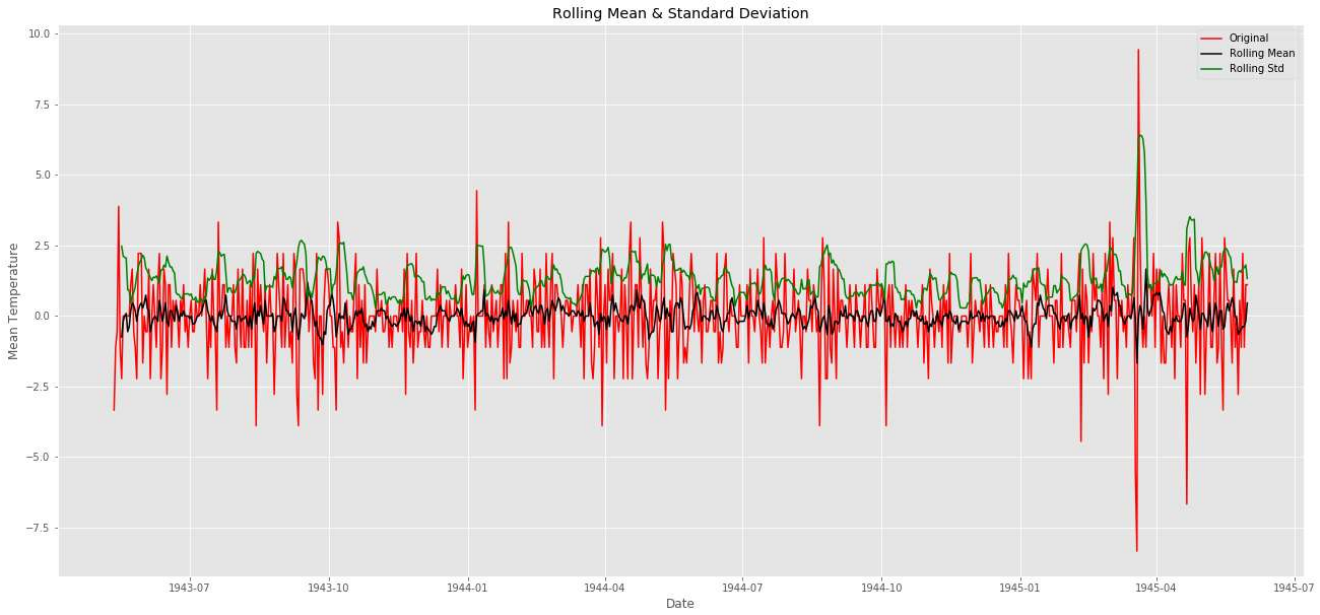

```
Test statistic:  -11.13851433513848
p-value:  3.150868563164539e-20
Critical Values: {'1%': -3.4392539652094154, '5%': -2.86546960465041, '10%': -2.5688625527782327}
```

"For the constant mean criterion: The mean appears to be constant, as you can see from the plot (black line) above." "The second criterion is a constant variance, and it appears to be constant." "The test statistic is smaller than the 1% critical values, so we can confidently say that this is a stationary series with 99% confidence." "We have achieved a stationary time series. However, let's explore one more method to address trend and seasonality." "Differencing method: This is one of the most common methods. The idea is to take the difference between the time series and a shifted version of the time series."

In [22]:
```
# differencing method
ts_diff = ts - ts.shift()
plt.figure(figsize=(22,10))
plt.plot(ts_diff)
plt.title("Differencing method")
plt.xlabel("Date")
plt.ylabel("Differencing Mean Temperature")
plt.show()
```



In [23]:
```
ts_diff.dropna(inplace=True) # due to shifting there is nan values
# check stationary: mean, variance(std)and adfuller test
check_mean_std(ts_diff)
check_adfuller(ts_diff.MeanTemp)
```




```
Test statistic:  -11.678955575105375
p-value:  1.7602075693558953e-21
Critical Values: {'1%': -3.439229783394421, '5%': -2.86545894814762, '10%': -2.5688568756191392}
```

"Constant mean criterion: The mean appears constant, as evident from the plot (black line) above."

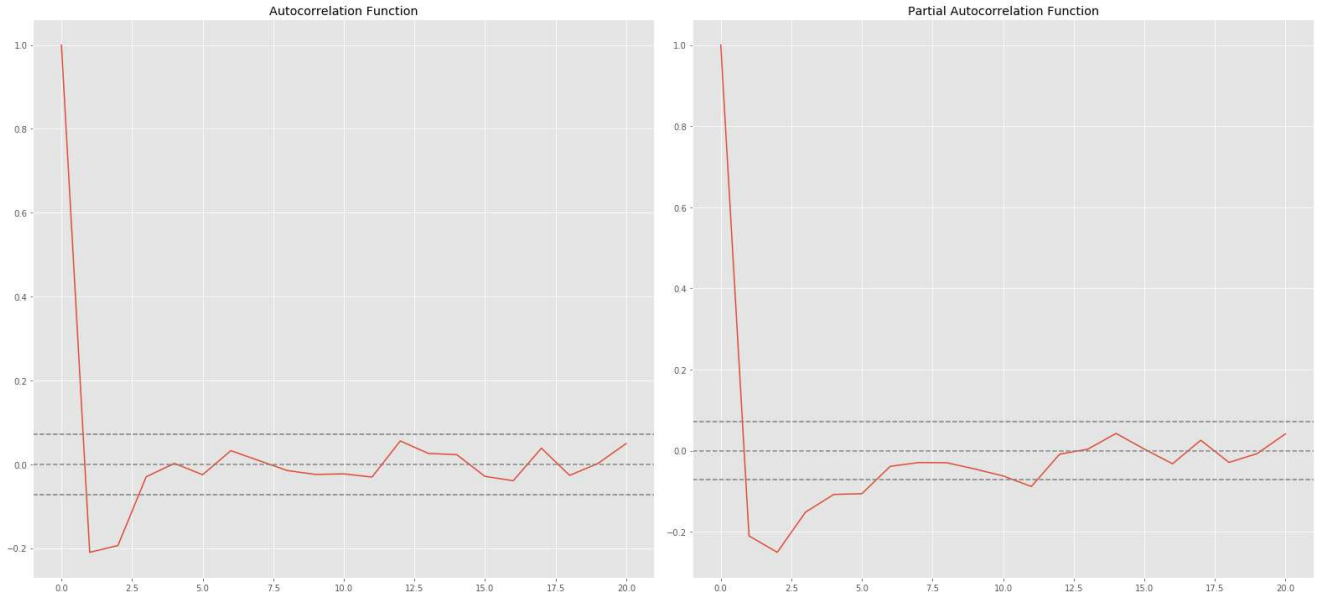
"The second criterion is constant variance, and it appears constant." "The test statistic is smaller than the 1% critical values, so we can confidently say this is a stationary series with 99% confidence."

"For prediction (forecasting), we will use the ts_diff time series, which is the result of the differencing method. There is no specific reason for choosing only it."

"The prediction method we'll use is ARIMA, which stands for Auto-Regressive Integrated Moving Averages." "AR: Auto-Regressive (p): AR terms are simply lags of the dependent variable. For example, if p is 3, we will use x(t-1), x(t-2), and x(t-3) to predict x(t)." "I: Integrated (d): This represents the number of nonseasonal differences. In our case, we took the first-order difference, so we use d=0." "MA: Moving Averages (q): MA terms are lagged forecast errors in the prediction equation." "(p, d, q) are the parameters of the ARIMA model." "To choose the p, d, q parameters, we will use two different plots:" "Autocorrelation Function (ACF): This measures the correlation between the time series and lagged versions of the time series." "Partial Autocorrelation Function (PACF): This measures the correlation between the time series and lagged versions of the time series, while eliminating the variations already explained by intervening comparisons."

In [24]:
```
# ACF and PACF
from statsmodels.tsa.stattools import acf, pacf
lag_acf = acf(ts_diff, nlags=20)
lag_pacf = pacf(ts_diff, nlags=20, method='ols')
# ACF
```

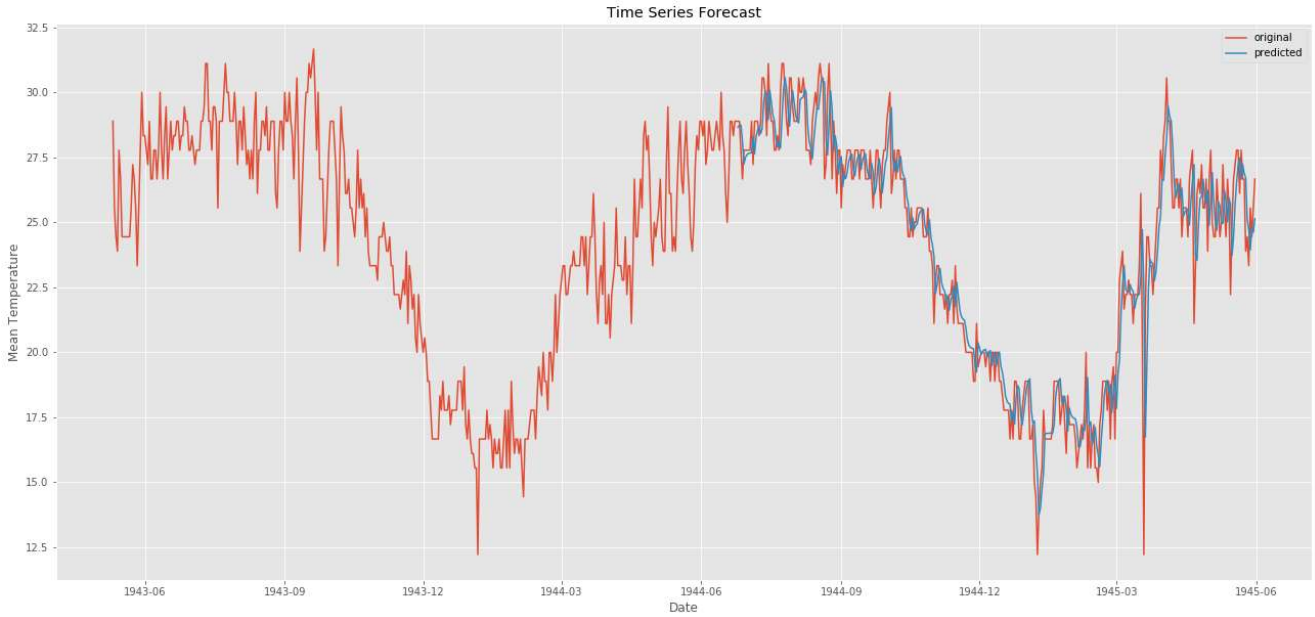
```
plt.figure(figsize=(22,10))

plt.subplot(121)
plt.plot(lag_acf)
plt.axhline(y=0,linestyle='--',color='gray')
plt.axhline(y=-1.96/np.sqrt(len(ts_diff)),linestyle='--',color='gray')
plt.axhline(y=1.96/np.sqrt(len(ts_diff)),linestyle='--',color='gray')
plt.title('Autocorrelation Function')

# PACF
plt.subplot(122)
plt.plot(lag_pacf)
plt.axhline(y=0,linestyle='--',color='gray')
plt.axhline(y=-1.96/np.sqrt(len(ts_diff)),linestyle='--',color='gray')
plt.axhline(y=1.96/np.sqrt(len(ts_diff)),linestyle='--',color='gray')
plt.title('Partial Autocorrelation Function')
plt.tight_layout()
```



"Two dotted lines represent the confidence intervals. We use these lines to determine the values of 'p' and 'q'."

"Choosing 'p': It's the lag value where the PACF chart first crosses the upper confidence interval. In this case, p=1." "Choosing 'q': It's the lag value where the ACF chart first crosses the upper confidence interval. Here, q=1." "Now, let's use (1,0,1) as the parameters for the ARIMA model and make predictions." "We will use the ARIMA implementation from the statsmodels library." "For datetime, we will use it as the start and end indexes for the prediction method."

In [25]:
```
# ARIMA LİBRARY
from statsmodels.tsa.arima_model import ARIMA
from pandas import datetime

# fit model
model = ARIMA(ts, order=(1,0,1)) # (ARMA) = (1,0,1)
model_fit = model.fit(disp=0)

# predict
start_index = datetime(1944, 6, 25)
end_index = datetime(1945, 5, 31)
forecast = model_fit.predict(start=start_index, end=end_index)

# visualization
plt.figure(figsize=(22,10))
plt.plot(weather_bin.Date,weather_bin.MeanTemp,label = "original")
plt.plot(forecast,label = "predicted")
plt.title("Time Series Forecast")
plt.xlabel("Date")
plt.ylabel("Mean Temperature")
plt.legend()
plt.show()
```



lets predict and visualize all path and find mean squared error

In [26]:
```
# predict all path
from sklearn.metrics import mean_squared_error
# fit model
model2 = ARIMA(ts, order=(1,0,1)) # (ARMA) = (1,0,1)
model_fit2 = model2.fit(disp=0)
forecast2 = model_fit2.predict()
error = mean_squared_error(ts, forecast2)
print("error: " ,error)
# visualization
plt.figure(figsize=(22,10))
plt.plot(weather_bin.Date,weather_bin.MeanTemp,label = "original")
plt.plot(forecast2,label = "predicted")
plt.title("Time Series Forecast")
plt.xlabel("Date")
plt.ylabel("Mean Temperature")
plt.legend()
plt.savefig('graph.png')

plt.show()
```

```
error:  1.8625819236007615
```