Task 1:

Idea: Combine the data column and the time column, then add 8 hours and recalculate the timestamp.

1. **Combining Date and Time Columns**: The script starts by creating a new column '**Data_Time**', which combines the existing '**Date**' and '**Time**' columns into a single string. This step is crucial for treating the date and time as a single entity, facilitating further transformations.
2. **String to Timestamp Conversion**: The combined '**Data_Time**' string is then converted into a timestamp format using the '**to_timestamp**' function. This step is necessary to enable arithmetic operations on the date-time data.
3. **Time Zone Adjustment**: The script adds 8 hours to the '**Data_Time**' column to convert the time from GMT to China time. This is achieved using the expression '**Data_Time + interval 8 hours**'. This step is the core of the task, aligning the time with the China time zone.
4. **Extracting Date and Time**: After adjusting the time zone, the script separates the '**Data_Time**' column back into '**Date**' and '**Time**' columns. This is done by casting the '**Data_Time**' to a date format for the '**Date**' column and formatting the time part for the **Time** column.
5. **Calculating Timestamp Difference**: The script then calculates the difference in timestamps from a start date ('**1899-12-30 00:00:00**').
6. **Cleaning Up**: Unnecessary columns are dropped, and the final DataFrame is shown and returned with one more column ('**Data_Time**') which will be used in next tasks.

Output:

```
+------+---------+----------+-------+----------------+------------------+----------+--------+
|UserID| Latitude| Longitude|AllZero|        Altitude|         Timestamp|      Date|    Time|
+------+---------+----------+-------+----------------+------------------+----------+--------+
|   130|39.975088| 116.33269|      0|492.126135170604| 40001.13300925926|2009-07-07|03:11:32|
|   130| 39.97504|116.332806|      0|491.743412073491| 40001.13302083333|2009-07-07|03:11:33|
|   130|39.975009|116.332997|      0|491.676630577428| 40001.13303240741|2009-07-07|03:11:34|
|   130|39.975048|116.332932|      0|491.403044619423|40001.133043981485|2009-07-07|03:11:35|
|   130|39.974977| 116.33305|      0|491.043900918635| 40001.13306712963|2009-07-07|03:11:37|
|   130|39.974967|116.333116|      0|489.435272309711| 40001.13318287037|2009-07-07|03:11:47|
|   130|39.974931|116.333188|      0|487.797303149606| 40001.13329861111|2009-07-07|03:11:57|
|   130|39.974927|116.333249|      0|489.250744750656|40001.133368055554|2009-07-07|03:12:03|
|   130|39.974953| 116.33331|      0|487.910994094488|40001.133472222224|2009-07-07|03:12:12|
|   130|39.974988|116.333359|      0|486.979045275591| 40001.13353009259|2009-07-07|03:12:17|
|   130|39.975029|116.333311|      0|486.138612204724| 40001.13361111111|2009-07-07|03:12:24|
|   130|39.974993|116.333368|      0|485.249717847769| 40001.13365740741|2009-07-07|03:12:28|
|   130|39.975013|116.333424|      0|484.258648293963|40001.133738425924|2009-07-07|03:12:35|
|   130|39.975008|  116.3335|      0|483.536105643045| 40001.13379629629|2009-07-07|03:12:40|
|   130|39.975018|116.333564|      0|482.802204724409| 40001.13385416667|2009-07-07|03:12:45|
|   130|39.974994|116.333631|      0|482.616423884514| 40001.13386574074|2009-07-07|03:12:46|
|   130|   39.975|116.333703|      0|482.115708661417|40001.133888888886|2009-07-07|03:12:48|
|   130|39.975025|116.333767|      0|481.127742782152| 40001.13396990741|2009-07-07|03:12:55|
|   130|39.974989|116.333882|      0|480.362650918635| 40001.13402777778|2009-07-07|03:13:00|
|   130|39.974964|116.333959|      0|480.077047244094|40001.134050925924|2009-07-07|03:13:02|
+------+---------+----------+-------+----------------+------------------+----------+--------+
only showing top 20 rows
```

Task 2:

Idea: First calculate the data points recorded every day, then filter the ones that are greater than or equal to 5, and finally sort them.

1. **Counting Data Points per Day**: The script starts by grouping the DataFrame by **'UserID'** and **'Date'**, then counting the number of data points for each user on each day. This is achieved using **'df.groupBy('UserID', 'Date').count()'**.

2. **Filtering Days with At Least Five Data Points**: The script then filters these grouped results to retain only those days where a user has five or more data points, using **'filter(col('count') >= 5)'**.

3. **Counting Eligible Days per User**: Next, it groups the filtered data by **'UserID'** and aggregates the count of days where the condition (at least five data points) is met. This step calculates the total number of such days for each user.

4. **Sorting and Selecting Top Users**: The script sorts the users based on the count of days with at least five data points in descending order. In case of a tie, it sorts by **'UserID'** in ascending order. This sorting determines the top 5 users as per the requirement.

5. **Displaying and Returning Results**: The top 5 users are displayed using **'show(5)'** and then returned with **'limit(5)'**. This ensures that only the top 5 users, as per the defined criteria, are included in the final output.

Output:

```
+------+---------+
|UserID|DaysCount|
+------+---------+
|   144|      312|
|   140|      237|
|   142|      108|
|   134|       57|
|   141|       26|
+------+---------+
only showing top 5 rows
```

Task 3:

Idea: First calculate the week number since December 30, 1899 for each day, then calculate the weekly data points, and finally filter out the ones greater than 100

1. **Converting Timestamp to Week Numbers**: The script starts by converting the **'Timestamp'** to days and then to week numbers. This is crucial to group data points by weeks. The week number is calculated by subtracting 2 from the day number (to align with the week starting on Monday) and dividing by 7, then flooring the result.

2. **Counting Data Points Per Week for Each User**: The DataFrame is grouped by **'UserID'** and **'Weeks'**, and the number of data points for each user per week is counted using '**groupBy**' and '**agg(count('*').alias('DataPointsCount'))**'.

3. **Filtering 'Weeks' with More Than 100 Data Points**: The script filters out the weeks where a user has more than 100 data points using **'filter(col('DataPointsCount') > 100)'**.

4. **Counting Eligible Weeks per User**: After filtering, it aggregates the data by **'UserID'** to

count the number of weeks each user has more than 100 data points. This is done through ‘**groupBy**’ and ‘**agg(count('*').alias('WeeksCount'))**’.

5. **Ordering Results by ‘UserID’**: The results are ordered by **‘UserID’**, ensuring that the output is structured and easy to understand.
6. **Displaying and Returning Results**: The script displays the results using **show()** and then returns the DataFrame. This ensures that all user IDs and their respective counts of weeks with more than 100 data points are included.

Output:

```
+------+----------+
|UserID|WeeksCount|
+------+----------+
|   130|         9|
|   131|         9|
|   132|         2|
|   133|         4|
|   134|        15|
|   135|         6|
|   136|         4|
|   137|         1|
|   138|         6|
|   139|         4|
|   140|        52|
|   141|         8|
|   142|        31|
|   143|         1|
|   144|        63|
|   145|         5|
|   146|         2|
|   147|        12|
|   148|         3|
|   149|         1|
+------+----------+
only showing top 20 rows
```

Task 4:

Idea: First find the minimum latitude of each user, then attach the date, and finally select the largest 5, and remove duplicate values to retain the date of first arrival.

1. **Computing Southernmost Latitude**: The script begins by grouping the DataFrame by **‘UserID’** and calculating the minimum latitude for each user using ‘**min('Latitude').alias('MinLatitude')**’. This step identifies the southernmost point for each user.
2. **Renaming UserID Column for Joining**: To facilitate a subsequent join operation, the ‘**UserID**’ column in the aggregated DataFrame is renamed to **‘ID’**.
3. **Joining DataFrames**: The original DataFrame is joined with the aggregated DataFrame containing the southernmost latitudes. This join is based on matching **‘UserID’** and the latitude being equal to the minimum latitude. This step associates each user's southernmost latitude with the corresponding date(s).

4. **Removing Duplicate Entries**: To ensure only one record per user, the script applies **‘orderBy('UserID’,'Date')’** and **‘dropDuplicates(['UserID'])’**. This ordering helps identify the first occurrence of the southernmost point for each user, in case of ties. This step is crucial as there could be multiple instances where the user reached their southernmost point.
5. **Ordering by Latitude**: The results are then ordered by latitude (to prioritize southernmost locations).
6. **Selecting Relevant Columns and Limiting Entries**: The script selects only the relevant columns (‘**UserID**’, ‘**MinLatitude**’, and ‘**Date**’) and limits the output to the first 5 entries using **‘limit(5)’**. This step addresses the requirement to output the top 5 user IDs.
7. **Displaying and Returning Results**: Finally, the results are displayed and the modified DataFrame is returned. This step ensures the output includes the top 5 user IDs, the southernmost latitude they reached, and the date of this occurrence.

Output:

```
+------+----------------+----------+
|UserID|     MinLatitude|      Date|
+------+----------------+----------+
|   144|        1.044024|2009-03-25|
|   142|       1.2724579|2008-07-26|
|   160|       13.359522|2010-12-27|
|   153|    22.230673974|2012-05-08|
|   134|22.3335333333333|2007-07-18|
+------+----------------+----------+
```

Task 5:

Idea: First remove the invalid data and convert the unit to meters, then find the maximum and minimum altitudes every day, calculate the span, and retain the 5 users with the largest data

1. **Converting Altitude and Filtering Invalid Values**: The script first converts the altitude from feet to meters, considering -777 as an invalid value. This is done using a **‘when’** condition and multiplication by 0.3048 (conversion factor from feet to meters). Invalid values are replaced with **‘None’** and then dropped from the DataFrame.
2. **Ensuring Altitude as a Float**: The altitude column is cast to a **‘DoubleType’** to ensure proper handling of floating-point calculations.
3. **Computing Max and Min Altitude Per Day**: The script groups the data by **‘UserID’** and **‘Date’**, then calculates the maximum and minimum altitude for each user on each day. This is crucial for determining the altitude span.
4. **Calculating Altitude Span Per Day**: For each day and user, the script calculates the altitude span by subtracting the minimum altitude from the maximum altitude.
5. **Aggregating Maximum Altitude Span Per User**: The script then groups the data by **‘UserID’** and finds the maximum altitude span across all days for each user. This step identifies the largest altitude span that each user experienced in a single day.

6. **Ordering and Selecting Top Users**: The results are ordered by the altitude span in descending order, and the top 5 users are selected using **‘limit(5)’**. This prioritizes users who had the largest altitude spans.

7. **Displaying and Returning Results**: The script displays the results and returns the DataFrame, ensuring that the output includes the top 5 user IDs and their corresponding maximum altitude spans.

Output:

```
+------+------------------+
|UserID|      AltitudeSpan|
+------+------------------+
|   144|       26217.00624|
|   140|16710.995280000003|
|   153| 7723.200042999985|
|   147|       2142.249395|
|   142|         1979.9808|
+------+------------------+
```

Preparation of task6 and task7:

Idea: Add a function to calculate the distance between two data points.

1. **Window Specification Setup**: The script begins by defining a window partitioned by **‘UserID’** and **‘Date’**, ordered by **‘Timestamp’**. This window specification is critical for applying the **‘lag’** function correctly, allowing the calculation of distances between consecutive points for each user on each given day.

2. **Applying the Lag Function for Previous Coordinates**: The script employs the **‘lag’** function to obtain the previous latitude (‘**PrevLatitude**’) and longitude (‘**PrevLongitude**’) for each data point. Additionally, it captures the previous timestamp (‘**PrevData_Time**’). This step is fundamental for calculating the distance and speed between consecutive points, as it provides the previous geographical coordinates and time for each point.

3. **Distance Calculation Using Haversine Formula**: The Haversine formula is implemented to calculate the distance between two points on the Earth's surface. This involves calculating the difference in latitudes and longitudes in radians and applying the Haversine formula, which involves trigonometric functions to compute the distance based on the Earth’s radius (assumed here as 6373.0 km).

   The calculated distance (‘**Distance**’) is the spherical distance between each pair of consecutive points for a user on a given day.

4. **Data Cleaning**: The script removes rows with null values, which are likely present for the first record of each user on each day due to the use of the lag function. It also drops the columns holding previous latitude and longitude values, as they are no longer needed after calculating the distance.

Task 6:

Idea: Calculate the distance between two points first, then calculate the sum of the daily distances, and the sum of the left and right distances.

1. **Calculating Distances Between Consecutive Points**: The function ‘**dis_two_point**’ presumably calculates the distance between consecutive data points. This function is applied to the DataFrame **‘df’** to create a new DataFrame **‘daily_distance_df’** containing these distances.
2. **Aggregating Daily Distances**: The distances are then aggregated by **‘UserID’** and **‘Date’** to calculate the total daily distance for each user. This is done using **‘groupBy’** and **‘sum('Distance').alias('DailyDistance')’**.
3. **Window Specification for Ranking Users**: A window specification (**‘windowSpecUser’**) is defined to rank users based on their maximum daily distance. This specification orders the data by daily distance in descending order and then by date, which is essential for identifying the day each user travelled the most.
4. **Identifying Maximum Daily Distance for Each User**: The script uses the window specification to rank the daily distances for each user and filters out the top rank (i.e., the maximum daily distance) using **‘filter(col('Rank') == 1)’**. The result is a DataFrame ‘‘**max_distance_by_user**’ containing each user's date of maximum daily distance.
5. **Displaying Date of Maximum Daily Distance**: The date of maximum daily distance for each user is displayed using **‘show()’**.
6. **Calculating Total Distance Travelled by All Users**: The script also calculates the total distance travelled by all users across all days. This is done by aggregating the **‘daily_distance_df’** without any grouping and summing up the **‘DailyDistance’**.
7. **Displaying Total Distance**: The total distance travelled by all users is displayed.
8. **Returning Results**: The function returns two DataFrames: **‘max_distance_by_user’**, containing the date of maximum daily distance for each user, and **‘total_distance’**, representing the total distance travelled by all users.

Output:

```
+------+----------+
|UserID|      Date|
+------+----------+
|   130|2009-09-12|
|   131|2009-04-21|
|   132|2010-05-01|
|   133|2011-04-21|
|   134|2007-07-07|
|   135|2009-01-25|
|   136|2008-05-30|
|   137|2011-01-28|
|   138|2007-06-27|
|   139|2007-10-04|
|   140|2009-01-15|
|   141|2011-10-23|
|   142|2008-05-07|
|   143|2009-09-12|
|   144|2009-03-26|
|   145|2008-04-30|
|   146|2007-08-01|
|   147|2011-03-06|
|   148|2011-05-15|
|   149|2009-09-12|
+------+----------+
only showing top 20 rows

+------------------+
|     TotalDistance|
+------------------+
|195649.95924555542|
+------------------+
```

Task 7:

Idea: First calculate the distance between the two data points, then use the timestamp in seconds retained in task1 to calculate the time difference, then calculate the speed, and finally find the maximum speed of each user.

1. **Calculating Distance and Time Between Consecutive Points**: The script starts by using a function **'dis_two_point'**, presumably to calculate the distance between consecutive data points. The DataFrame is ordered by **'UserID'** and **'Timestamp'** before applying this function. A new column **'Second'** is added to store the time difference in seconds between consecutive data points.
2. **Calculating Speed**: The speed is calculated by dividing the distance (in kilometers) by the time difference (in seconds) and then converting it to km/h by multiplying by 3600 (number of seconds in an hour). This results in a new column **'Speed'** in the DataFrame.
3. **Aggregating Maximum Speed Per Day for Each User**: The script groups the data by **'UserID'** and **'Date'**, then uses **'agg(max('Speed').alias('MaxSpeed'))'** to find the maximum speed for each user on each day.
4. **Window Specification for Ranking Users**: A window specification (**'windowSpecUser'**) is defined to rank the users based on their maximum speed. It orders the data by maximum speed in descending order and then by date.
5. **Identifying Maximum Speed for Each User**: The script uses the window specification to rank the speeds for each user and filters out the top rank, i.e., the maximum speed, using **'filter(col('Rank') == 1)'**. This results in a DataFrame **'max_speed_by_user'** containing each user's maximum speed and the corresponding date.
6. **Displaying Maximum Speed**: The maximum speed for each user is displayed using **'show()'**.
7. **Returning the DataFrame**: The function returns the DataFrame **'max_speed_by_user'**, which includes each user's maximum speed and the earliest day they achieved this speed

Output:

```
+------+----------+------------------+
|UserID|      Date|          MaxSpeed|
+------+----------+------------------+
|   130|2009-09-05|487.19918283334795|
|   131|2009-07-16| 234.5585103185148|
|   132|2010-05-01|212.00049212958348|
|   133|2011-01-31| 2090.687565496368|
|   134|2007-07-07|1255.7688100808653|
|   135|2009-01-25| 411.7869112615427|
|   136|2008-05-12|324.67795145987736|
|   137|2011-01-28|287.51411006680325|
|   138|2007-06-27| 7642.134197514664|
|   139|2007-09-04| 1469.867466140526|
|   140|2008-09-03| 5694.102578995255|
|   141|2011-08-31|1634.3571923962322|
|   142|2007-06-13|2020.9519774283237|
|   143|2009-09-13|122.07184128413593|
|   144|2009-03-25|1171521.7275436502|
|   145|2008-05-24| 203.3233004193169|
|   146|2007-08-01|240.32549329256722|
|   147|2011-03-01|420.74409184718235|
|   148|2011-05-15| 513.0581717875818|
|   149|2009-09-13| 480.1485714661732|
+------+----------+------------------+
only showing top 20 rows
```