Phase 1 – myHealthCare function

The myHealthCare function uses Pandas to create a data frame via a dictionary. The data is generated using the random library, the ranges specified, rand.seed(109) is used for reproducibility. The ranges and the abnormal values are set as global variables outside the myHealthCare() function. The index generated by the pandas data frame is set to be the same as the timestamp. The function returns a data frame of n records and the 8 values. Initially n is set to 1000 but the function can be called for any positive integer value of n.

By calling the function via print with the .head() function, the first 5 rows output as:

```
     Timestamp  Temperature  Heart Rate  Pulse  Blood Pressure  Respiratory Rate  Oxygen Saturation   pH
101        101           38          61     70             121                13                 93  7.5
102        102           37          93     79             121                16                 95  7.6
103        103           39          60     91             120                13                 94  7.5
104        104           39          76     65             120                13                 94  7.1
105        105           36          55     99             121                16                 97  7.1
```

Phase 2 – abnormalSignAnalytics and frequencyAnalytics functions

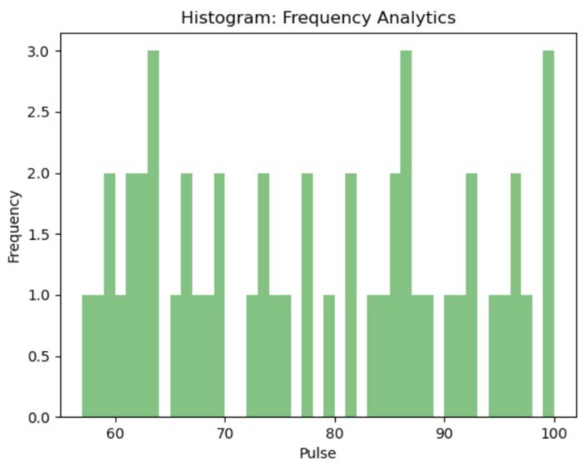
To generate a sample of 50 records (not 50 specified records), a data frame is created (outside the functions) from the myHealthCare output, smallSampleDF. It uses a 5% fraction of the myhealthcare data frame to allow for different sizes of n for my healthcare, to scale with the size of n. random_state is also passed as an argument to set the seed for reproducibility.

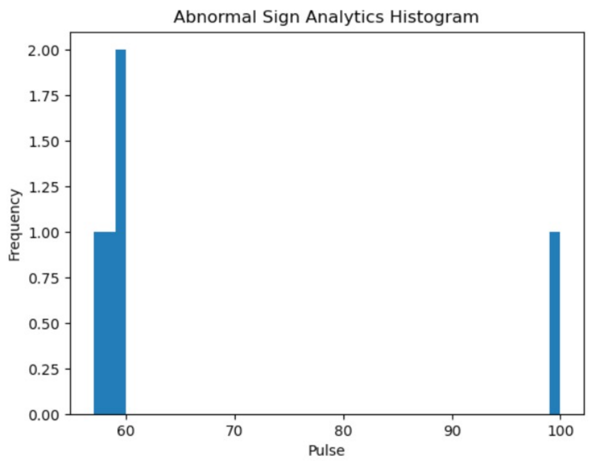
abnormalSignAnalytics takes two arguments, the abnormal sign measurement which is set to use the information in the abnormal sign global variables, for pulse here; plus which values column from the data frame is of interest. (Pulse has been chosen rather than blood pressure as pulse has a greater range, so potentially more interesting to explore. Blood pressure only has a range of 2 possible values.) The function selects only the “Timestamp” column and the column of interest, “Pulse” from the selected data frame; smallSampleDF here, then creates a list by iterating though the data frame. The function returns the list as output. Print statements have been added to the code to display the column/sign of interest, the count and the data:

```
Abnormal sign values for Pulse counted 5 records; [(976, 100), (249, 58), (486, 59), (301, 57), (765, 59)]
```

Iterating through the data frame to create a list results in complexity of O(n) as the whole of the size of the smallSampleDF has to be iterated through to check for the required information. The sample is set to be 5% of the total data frame, so this will grow with the size of n.

frequencyAnalytics takes the required column/value of interest as an argument, “Pulse” here. It also uses the smallSampleDF. Matplotlib is used to generate a histogram plot by passing the column required as an argument. The number of bins is set to show a bin for each value. The function outputs the plot. As plt.show() must be used to show a plot, this is not entered into the function output to prevent the graph being shown every time the function is called, but as required. The output histogram is shown side by side with the histogram for abnormalSignAnalytics:





abnormalSignAnalytics is a subset of the data in **frequencyAnalytics** so simply shows a subset of the abnormal values which are then duplicated in **frequencyAnalytics** pulse is used in both. The frequency analytics output shows the random nature of the generated dataset. This isn't necessarily reflective of a real life situation where perhaps the data would appear more normally distributed. This could be simulated in the random generation of data by using random.normal and choosing a standard deviation and a mean.

<u>Phase 3 – healthAnalyser function</u>

Pandas functionality allows for easy filtering of the data frame using boolean arguments. Shown in healthAnalyser1. Here the data frame produced in myHealthCare is turned into a list of values containing the value 56 for pulse. The print statement incorporates the count of the number of records, 22 in this case with n=1000.

```
[[174.0, 37.0, 71.0, 56.0, 121.0, 16.0, 97.0, 7.1], [222.0, 37.0, 94.0, 56.0, 121.0, 15.0, 99.0, 7.2], [254.0, 38.0, 93.0, 56.0, 120.0, 12.0, 94.0, 7.5], [304.0, 36.0, 92.0,
 56.0, 121.0, 14.0, 99.0, 7.4], [475.0, 38.0, 97.0, 56.0, 120.0, 11.0, 95.0, 7.1], [480.0, 39.0, 78.0, 56.0, 120.0, 13.0, 99.0, 7.1], [521.0, 36.0, 65.0, 56.0, 121.0, 16.0,
 97.0, 7.4], [574.0, 36.0, 58.0, 56.0, 120.0, 17.0, 96.0, 7.2], [581.0, 38.0, 75.0, 56.0, 120.0, 12.0, 95.0, 7.2], [613.0, 39.0, 57.0, 56.0, 120.0, 15.0, 99.0, 7.4], [624.0,
 39.0, 86.0, 56.0, 121.0, 15.0, 94.0, 7.2], [664.0, 39.0, 66.0, 56.0, 121.0, 13.0, 98.0, 7.4], [667.0, 39.0, 59.0, 56.0, 120.0, 12.0, 99.0, 7.5], [681.0, 39.0, 90.0, 56.0,
 120.0, 17.0, 100.0, 7.2], [714.0, 37.0, 86.0, 56.0, 120.0, 17.0, 99.0, 7.4], [729.0, 37.0, 61.0, 56.0, 120.0, 15.0, 97.0, 7.3], [805.0, 39.0, 61.0, 56.0, 120.0, 15.0, 93.0,
 7.3], [818.0, 38.0, 82.0, 56.0, 121.0, 14.0, 98.0, 7.3], [847.0, 39.0, 93.0, 56.0, 121.0, 13.0, 98.0, 7.5], [879.0, 39.0, 96.0, 56.0, 120.0, 11.0, 97.0, 7.3], [888.0, 37.0,
 55.0, 56.0, 120.0, 13.0, 94.0, 7.5], [1019.0, 39.0, 99.0, 56.0, 121.0, 13.0, 94.0, 7.1]]
Number of records: 22
```

If not utilising the built in functionality of pandas, an alternative method is for the function to perform a linear search. This has complexity of O(n) as the whole list must be traversed in order to check all the elements which may be problematic with large data sets. The output is the same as above, see healthAnalyser2.

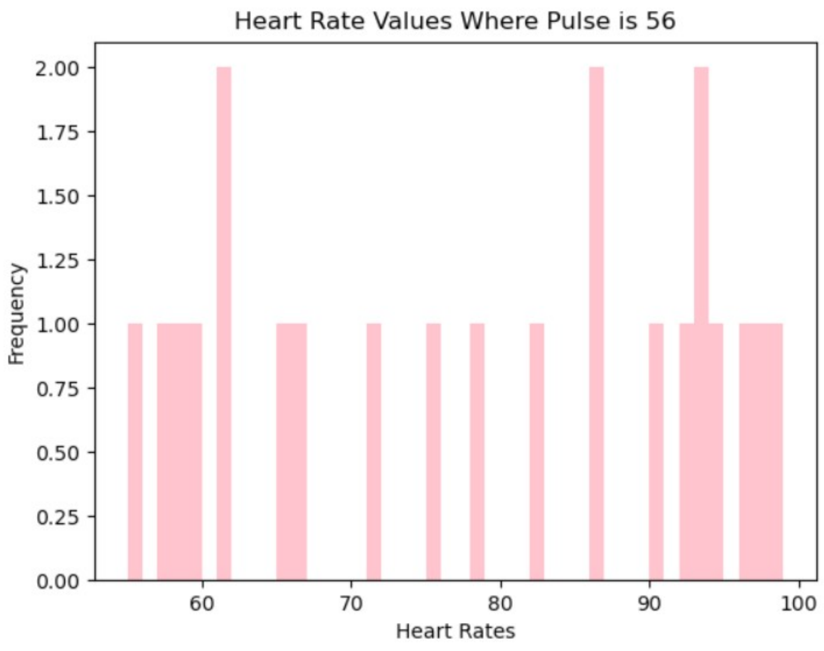
Another method would be to sort the data first, using a merge sort which has the complexity of O(nlogn), then perform either a binary search which needs O(logn) time, or an interpolation search which potentially can find data in O(log log n) time, to find the positions of the desired data. In the case of pulse = 56, in the range for pulse of 55-100, the value 56 would be near the start of any sorted data, so the entire list would <u>not</u> need to be traversed to find it. Binary search splits from the centre of the data, whereas interpolation search estimates the location of the data so would be optimal in this example. See healthAnalyser3.

Comparing the 3 methods delivers the following time to execute:

```
Time for healthAnalyser1:  0.03 seconds
Time for healthAnalyser2:  0.03 seconds
Time for healthAnalyser3:  0.04 seconds
```

The run time for version 3, which uses merge sort, is likely longer as there are more operations executed. Values are for n = 1000.

Heart rate values plot where pulse is 56:



<u>**Phase 4 – Benchmarking**</u>

Running time of the phases are as follows:

```
For n=1,000; Time to run for myHealthCare: 0.05 . Time for all phases: 1.52
For n=2,500; Time to run for myHealthCare: 0.05 . Time for all phases: 0.14
For n=5,000; Time to run for myHealthCare: 0.11 . Time for all phases: 0.17
For n=7,500; Time to run for myHealthCare: 0.15 . Time for all phases: 0.22
For n=10,000; Time to run for myHealthCare: 0.19 . Time for all phases: 0.29
```

As (mostly) expected as the value of n increases as does the time to run. The time to run for all phases vs just the initial myHealthCare function is greater than the one function alone as the number of operations is higher.