Florian Neziri

Birkbeck Student ID: 13172387

MSc Data Science

Programming with Data Coursework 2 – Technical Annex

Academic Declaration:

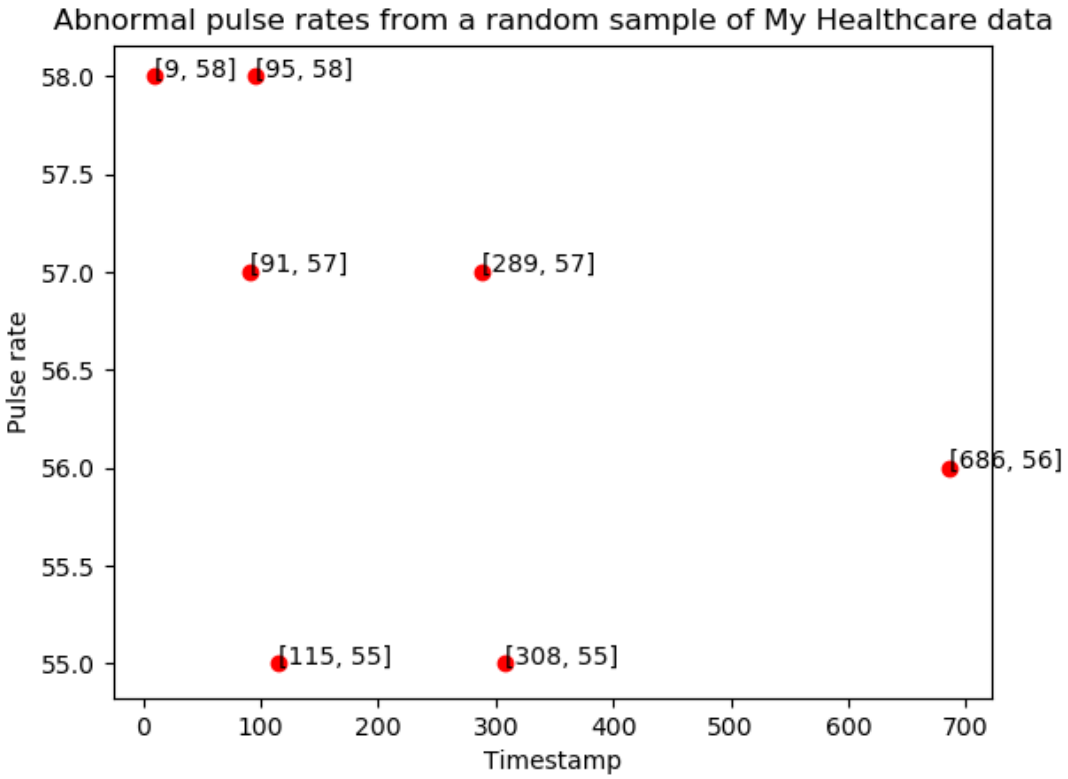
I have read and understood the sections of plagiarism in the College Policy on assessment offences and confirm that the work is my own, with the work of others clearly acknowledged. I give my permission to submit my report to the plagiarism testing database that the College is using and test it using plagiarism detection software, search engines or meta-searching software.

Signed: Florian Neziri Date: 15/03/2020

Phase 2 – Run Analytics

*Phase 2a: Develop a python function for abnormal values for **pulse***

Plot and discussion:



In the random sample that we obtained from the original data size of 100 records, we can see that there is a good spread of abnormal pulse values between 55-58, with 2 records for pulses of 55, 57, and 58, and 1 record for a pulse rate of 56. However, the sample does not contain any records with an abnormal pulse of 59 or 100, which the complete data set does have. So the random sample of abnormal pulse rates is not completely representative of the wider sample for abnormal pulse rates.

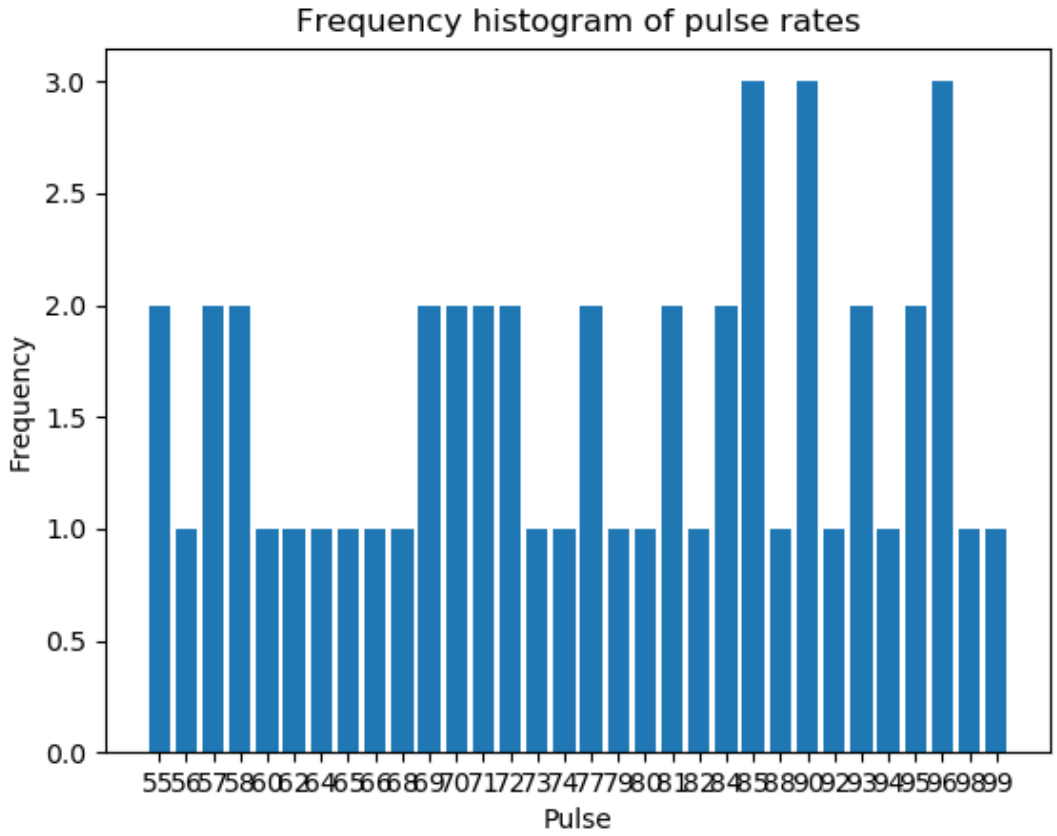
Computational cost: **O(n)**

The for loop has the same complexity regardless of values of n, as it relies on the length of the sample size, so its complexity is constant: O(50) (in this case sample size is 50). However, the whole function has complexity O(n) as, though the sample size stays constant for different values of n records, the time taken to create said sample will increase linearly with increasing n, as the random samples are taken from the whole dataset.

So abnormalPulseAnalytics has computational cost **O(n)** (worst-case)

Phase 2b: Present a frequency histogram of pulse rates.

Plot and discussion:



In the random sample, there doesn't seem to be to clear a pattern in terms of the frequency of each pulse rate. The highest frequency pulse rates are found towards the larger end – 85, 90, and 96 with 3 records. We would expect to see more of a normal distribution in pulse rates, with fewer records at the tail end of the data (the abnormal values of 55-59 and 100) and a higher frequency of pulses in the middle.

Computational cost: **O(n)**

This function is similar to the abnormalPulseAnalytics function in Phase 2a as, again, the sample size stays constant for different values of n records, but the time taken to create said sample will increase linearly with increasing n – so has computational complexity O(n). The first for loop traverses the length of the sample, so this will be constant or O(50), and creates a list of just pulse values which is then used in the next for loop, which traverses this list of pulses (which is the same length as the sample size, so also has complexity O(50)).

So frequencyAnalytics has computational cost **O(n)** (worst-case)

Phase 3: Search for heart rates using the HealthAnalyzer

For this question I created two solutions – one using linear search and the other using binary search. For the linear search, an advantage over the binary search algorithm is that there is no need to sort the data by pulse value as the algorithm will go through every single record in the data set and compare the pulse value with the key we are looking for (in this case a pulse value of 56).

For binary search I needed to sort the data by pulse value as it compares the middle value of the data array with the key we are looking for. I also had to use a modified version of the usual binary search algorithm as I needed to return all the records with a pulse rate of 56 as opposed to just locating one record with pulse value 56. This entailed creating one function called **healthAnalzyerBinLower()** to find the lower bound/first occurrence of a record with pulse value 56, and a second function called **healthAnalzyerBinUpper()** to find the upper bound/last occurrence of a record with pulse value 56.

Once the first and last occurrence of records with pulse value 56 are located, since the data is sorted by pulse value we know that every record with pulse value 56 will be located between these two bounds. So a final function is needed which returns all the values between these two indices in a list, which is what the for loop in the function **healthAnalzyerBin()** does.

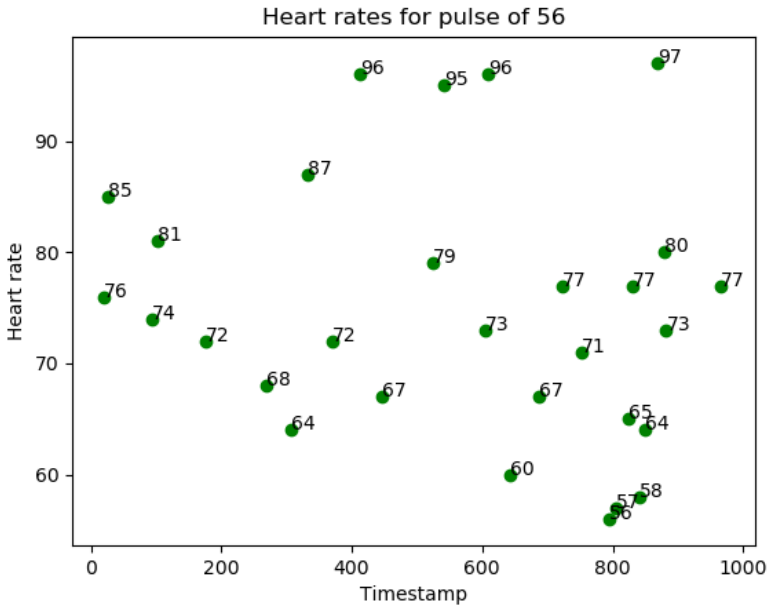
Complexity: linear search: **O(n)**, binary search: **O(log(n))**

For **linear** search, the algorithm will compare the key to every pulse value in the data set. This means that the complexity will directly increase with the number of records in the data set, hence **O(n)**.

For **binary** search, the algorithm will locate the middle value and then halve the data set (should the middle not be equal to the key) and keep searching and halving the dataset, and so on until it finds the key it is looking for or reaches the last element in the set. As we keep halving the data set after each search, the complexity is **O(log(n))** with the logarithm being of base 2. This is the case for both the *Lower* and *Upper* versions of the binary search algorithm I have implemented.

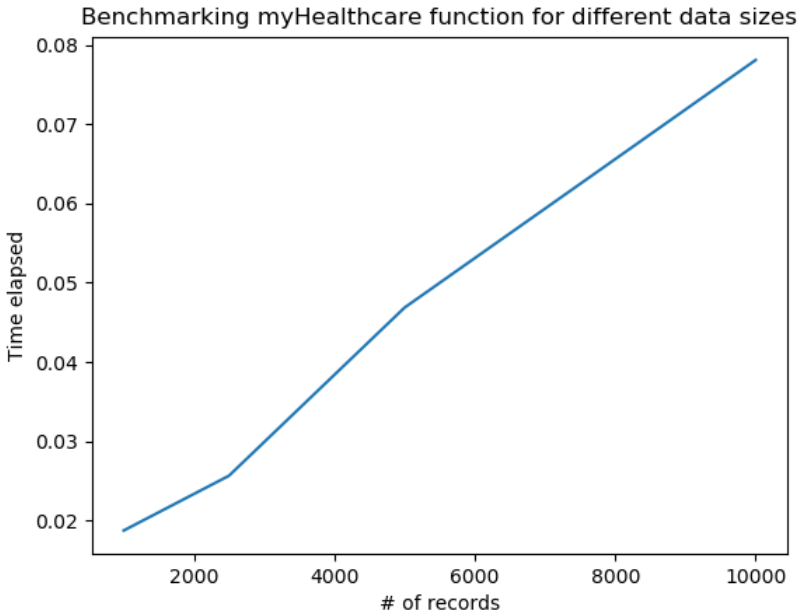
If the binary search algorithm has complexity O(log(n)), it is a more efficient algorithm to use than linear search which has complexity O(n).

Plot:



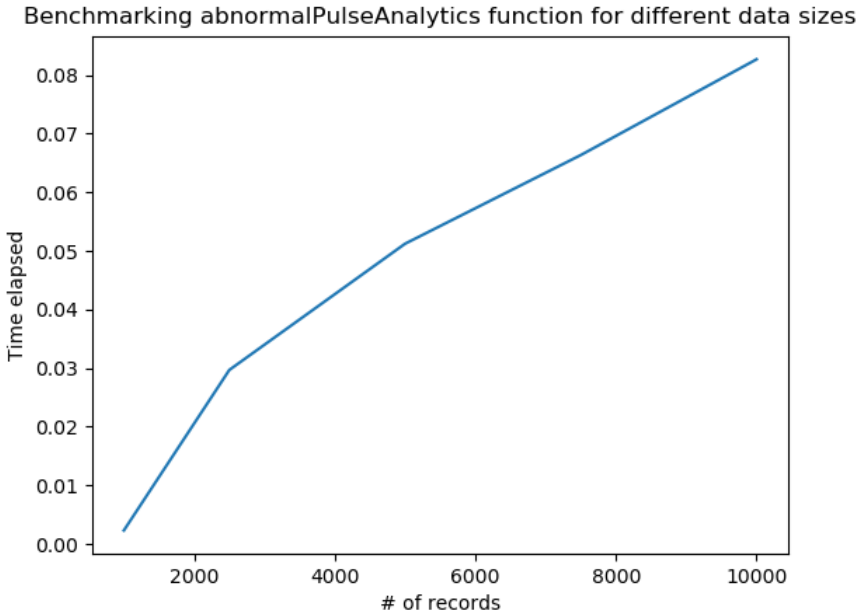
Phase 4: Testing scalability of your algorithm (20 marks)

myHealthcare function (Phase 1): **O(n)**



As we can see from the graph above, there is a linear positive relationship between the number of records created in the myHealthcare function and the time taken for the function to execute. This is what we expected as the myHealthcare function has computational complexity **O(n)** due to the for loop, which creates the records, being dependent on the length (size) of n.

Python function for abnormal values for pulse (Phase 2a): **O(n)**

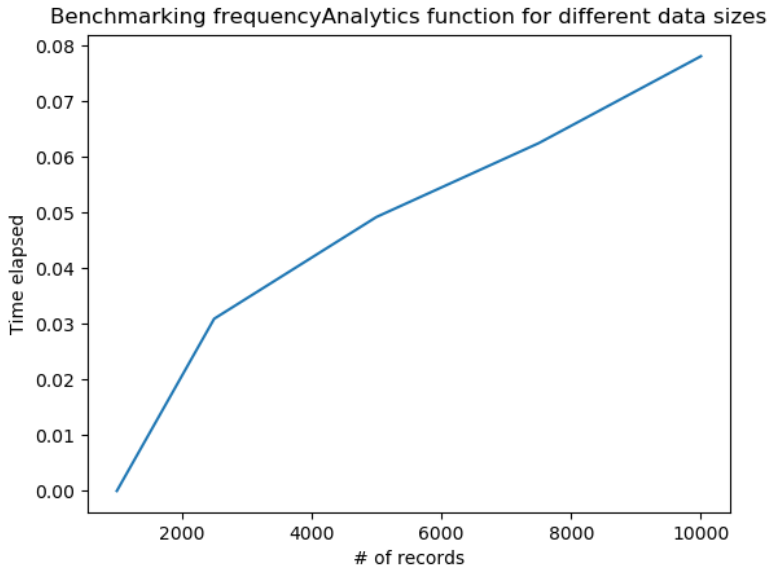


As we can see from the graph above, there is a linear positive relationship between the number of records used in the abnormalPulseAnalytics function and the time taken for the function to execute (create random samples and analyse the frequency of different abnormal pulse values.

The for loop has the same complexity regardless of values of n, as it relies on the length of the sample size, so its complexity is constant: O(50) (in this case sample size is 50). However, the whole function has complexity O(n) as, though the sample size stays constant for different values of n records, the time taken to create said sample will increase linearly with increasing n, as the random samples are taken from the whole dataset.

So abnormalPulseAnalytics has computational complexity **O(n)** (worst-case)

Present a frequency histogram of pulse rates (Phase 2b): **O(n)**

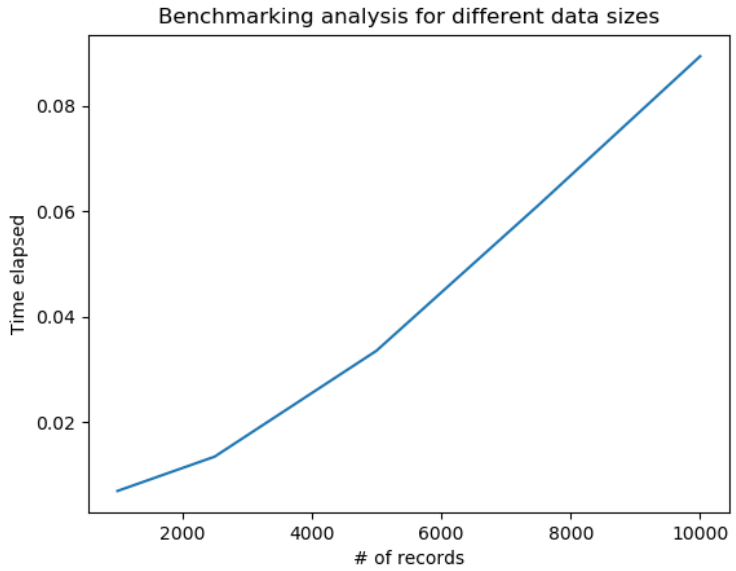


As we can see from the graph above, there is a linear positive relationship between the number of records used in the frequencyAnalytics function and the time taken for the function to execute (create random samples and analyse the frequency of different pulse values.

This function is similar to the abnormalPulseAnalytics function in Phase 2a as, again, the sample size stays constant for different values of n records, but the time taken to create said sample will increase linearly with increasing n – so has computational complexity O(n). The first for loop traverses the length of the sample, so this will be constant or O(50), and creates a list of just pulse values which is then used in the next for loop, which traverses this list of pulses (which is the same length as the sample size, so also has complexity O(50)).

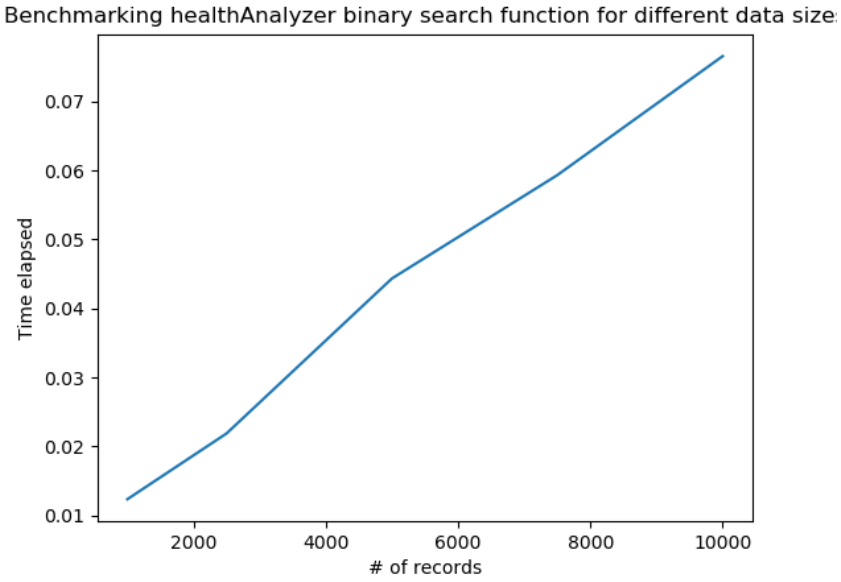
So frequencyAnalytics has computational complexity **O(n)** (worst-case)

Search for heart rates using the HealthAnalyzer (Phase 3 – Linear search algorithm): **O(n)**



For linear search, the algorithm will compare the key to every pulse value in the data set. This means that the computational cost/complexity will directly increase with the number of records in the data set, hence O(n) complexity.

Search for heart rates using the HealthAnalyzer (Phase 3 - Binary search algorithm): **O(log(n))**



For binary search, the algorithm will locate the middle value and then halve the data set (should the middle not be equal to the key) and keep searching and halving the dataset, and so on until it finds the key it is looking for or reaches the last element in the set. As we keep halving the data set aftereach search, the complexity is O(log(n)) with the logarithm being of base 2. This is the case for both the *Lower* and *Upper* versions of the binary search algorithm I have implemented.

If the binary search algorithm has complexity O(log(n)), it is a more efficient algorithm to use than linear search which has complexity O(n).