**Original Feature Set**

**A.** List all the vendors (vendor) that have a vulnerable product along with a total number of

vendors.

**B.** Display all the CVEs (cve\_id) caused by a particular weakness (cwe\_code):

a. As a simple list

b. Clustered by vulnerability score (cvss)

**C.**

a. For a specified CVE (cve\_id), display the name of vendor (vendor) and the product

(vulnerable\_product).

b. For a specified vendor (vendor), display the CVEs (cve\_id) associated with it along

with the product and the weakness (cwe\_code)

**D.**

a. Find the vendor (vendor) with the highest number of CVEs and display all the CVEs

(cve\_id) and publication dates (pub\_date).

b. Find the vendor (vendor) with the worst cumulative Common Vulnerability Score

(cvss). You will need to add up all the Common Vulnerability Scores associated with

a vendor and display the one with the highest score. Display all the CVEs (cve\_id)

Common Vulnerability Scores (cvss).

**E.** Sort the records in the CVE dataset into ascending order of the ratio between impact and

access complexity (access\_complexity). You will need to allocate numeric value to the

impact categories and then calculate the ratio.

**F.** For a specified record with incomplete data, display all possible records. The user will enter

data for some fields and you must list all possible CVEs that it might refer to.

Task

[ Evaluate the time complexity of the simplistic solution across all its Features; Make use of the large datasets (LargeDataset-\*) and analyse the change in performance, as compared to the sample dataset. Identifying areas that may be potentially problematic.

For every problematic area, include a clear justification of:

a. Why it is problematic (e.g., an exact match search of an array operates in linear time, which can be improved upon by utilising an alternative data structure),

b. Why it is significant (e.g., this operates over the whole list of entries – a large dataset). ]

1. Utilising the structures and algorithmic approaches covered, design a more efficient solution to address as many of the problem areas identified, as is practical.

Must include data structure and algorithm designs, producing UML Class Diagram of the program structure designed.

2. Evaluate the algorithmic (time) and data (space) complexity of the new solution and compare against the previous version. Where possible, compare these with the areas original code problems identified to indicate where you’ll have made a notable improvement to a problematic area (and how you did so), and identify problematic areas that you would be unable to resolve with the chosen approach.

3. Discuss 2 additional approaches that you might have adopted. Justify your choices in (1), comparing it to the 2 additional approaches. You should evaluate the algorithmic (time) and data (space) complexity of these different approaches using a feature from the feature set as an example.

4. **Implement your more efficient solution from (1)**. You can extend the original solution to do so.

5. Utilise profiling tools to highlight areas of poor time performance.

Using a Java Profiling Tool, quantify and explain the performance of the software in real-time terms against the required Feature Set. Details for each Feature profiled should include at a minimum: the time taken to complete a sample operation, where the majority of this time is spent (in your software or the core Java API), and where possible, explain how that relates to the formal quantification in (2 & 3).