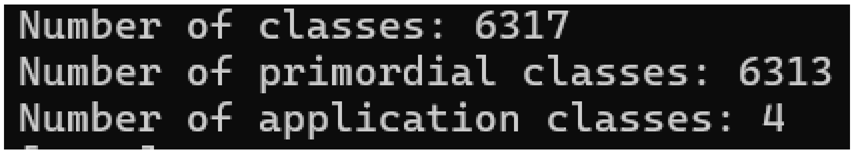
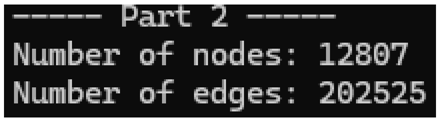
- **Q1:** Compute the class hierarchy for the Sample.jar compiled before and print out the following metrics:
    - # of Classes in the **application scope**
    - # of Classes in the **primordial scope**



```
Number of classes: 6317
Number of primordial classes: 6313
Number of application classes: 4
```

- **Q2:** Compute a **1-CFA Call Graph** for the Sample.jar file and then print out the following metrics:
    - # of Nodes in the Call Graph
    - # of Edges in the Call Graph



```
----- Part 2 -----
Number of nodes: 12807
Number of edges: 202525
```

- **Q3:** Implement a taint analyzer that detects the CWE-78 in the Sample.jar. That is, your code will identify instances of tainted flows that reach the sink: "`java.lang.Runtime.exec(...)`". Notice that not all flows that lead to the sink in the Sample.jar are vulnerable. The code will save the found vulnerability(ies) in a file (tainted_flows.txt), explicitly indicating the statements involved in the tainted flow.

1) **Summarize how JoanAudit identifies tainted flows in a Java program.**
Answer will be roughly 1 page in length.

JoanAudit is a static analysis tool for detecting common injection vulnerabilities, such as SQL or XML injection, in Java web applications. The tool first computes a System Dependence Graph (SDG). Using this graph, they create security slices by identifying sources, sinks, and declassifiers and extracting a slice for each sink. The SDG is constructed by taking the application's bytecode and using a tool such as WALA to create a system dependence graph. This tool also prunes out irrelevant functions based on a list of known irrelevant libraries. The SDG is then annotated with sources (points where untrustworthy input enters the program) and sinks (security-sensitive operations such as SQL code). Declassifiers, which are methods that validate user inputs, are also identified in this process. With this information, JoanAudit creates backward slices from each sink. This means they identify all lines that influence a sink. Normally, this is done naively, which means there are many irrelevant lines included in these slices. However, JoanAudit uses security slicing, which removes irrelevant code with several techniques. It first removes any sinks that are never reachable from a source. Then, it uses data and control dependency graphs to reduce the size of slices.

Once a security slice is extracted, JoanAudit runs an Information Flow Control (IFC) check, where it inspects whether tainted input can reach a sink without being declassified. The tool follows a slice to identify if there are declassifiers along the slice that can protect the sink from untrustworthy input. If the IFC analysis does not find a sanitizer, the path is flagged as potentially vulnerable. Additionally, this takes into account the context of the data used to avoid false positives when there is actually safe sanitization used.

After going through each slice, JoanAudit reports an HTML file with a listing for each flagged path. Each path consisted of the source and sink locations in terms of class, method, and line number. They also include the path connecting them, the vulnerability type, and the size of the slice. The HTML report allows the auditor to click into a detailed view and see only relevant statements in the slice. This is done to reduce the required manual effort.

2) **What are the main limitations of JoanAudit?**

Answer will include a bulleted list of limitations. You are expected to identify at least one limitation of the tool.

- This tool relies on a pre-configured list of input sources, sinks, and declassifiers. However, if a web application uses custom inputs or sinks that are not included in this list, then JoanAudit can miss vulnerabilities.
- This is not a general-use tool as it only focuses on injection-type vulnerabilities and Java Web Applications. This makes a very narrow use-case tool as it does not account for other frameworks or vulnerabilities.