1. High-level architecture of your optimizer, including the analysis and optimization algorithm(s) implemented, and why you chose that approach.

We chose to use dead code elimination with reaching definitions but without branch removal. We implemented fixed point iteration to solve the reaching definition problem. We then implemented the worklist algorithm by BFS'ing over the IN-set of each instruction, filtering for instructions that defined one of the operands of the current instruction. We chose this approach because it identifies more dead code than without reaching definitions.
We chose not to mark labels as critical and instead traverse to them from branch instructions. This prevents code bloat and unnecessary NOPs.

2. Low-level design decisions you made in selection of implementation language, and their rationale.

We selected Java, because the starter code was written in Java. This eliminated the risk of introducing parsing errors from translating to other languages.
Java is also a great language to use because of its robust object-oriented features that make scaling the code much easier as the scope of our program grows.
Overall, Java's modularity made it easier to implement objects and data structures such as Control Flow Graphs.

3. Software engineering challenges and issues that arose and how you resolved them.

We ran into issues with code readability and reuse. We originally wrote all optimization logic in the main method of the optimizer. However, as the scope grew, we abstracted basic blocks into the block class and put all control flow graph/iteration logic inside. This made it easier to construct the graph, perform iteration, and run mark sweep.
We also ran into issues with print debugging, since it was too slow and verbose. We solved this by setting up JDB and using the debug console.

4. Any known outstanding bugs or deficiencies that you were unable to resolve before the project submission.

N/A

5. Build and usage instructions for your optimizer.

From the root directory:

- Run build.sh (**./build.sh**)
- Run run.sh with the given IR file that you are trying to optimize (**./run.sh path/to/ir/file**)

- The optimized IR is produced in out.ir and can be run on a given input as desired. We created run.py to automate the process of testing our code on the given test cases.

6. A summary of the test results for the public test cases (see Section 1.2).

With our optimizer, all inputs reach the exact lowest dynamic instruction counts in the given csv files as we are doing dead code elimination with reaching definitions.