	4	3	2	1	0
Coding Style					
White Spacing	Coding solution adheres to all stylistic best practices; code employs white space to enhance readability throughout, operators and conditional expressions can be identified easily.	Logical blocks are indented consistently and spacing of blocks enhances readability. Complex conditional expressions and use of operators are mostly separated with blank space to enhance readability. Effective use of best stylistic practices.	Indentation and spacing make the code mostly readable. Some application of best stylistic practices for the programming language is in evidence.	Spacing and indentation inconsistencies hinder readability. Minimal attempt to use best stylistic practices for programming language.	No attempt to use indentation or spacing to enhance readability. Best practices for stylistic programming language conventions are not in evidence.
Variable Naming	Variable names clearly demonstrate their purpose. Abbreviations are used sparingly and appropriate for the domain. Single letter variables are restricted to for loop indexing. Proper and consistent use of name styling (e.g. under_score, camelCase, PascalCase). All constant values are associated with a constant variables name.	Effective variable names are used to give indication of purpose. Occasional abbreviations used only to shorten variable names. Rare use of non-named constants and single letter variable names. Effective use of consistent naming styles.	Adequate naming conventions. Variables many be shortened for brevity, moderate use of non-named constants. Single letter variables are found outside of normal use cases. Instances of inconsistent or inappropriate use of naming styles.	Frequent use of abbreviations for brevity, single letter variables, or mnemonics for variable names. Non-constant values are rarely assigned a name. Regularly inconsistent variable naming style.	Arbitrary or non-descriptive naming of variables. No names for constant values. Arbitrary variable naming style.
Function Naming	Function names clearly demonstrate their purpose. Names utilize verb phrases to describe functions action. Abbreviations are used sparingly and appropriate for the domain. Proper and consistent use of name styling (e.g. under_score, camelCase, PascalCase).	Effective function names are used to give indication of purpose. Occasional abbreviations used only to shorten names. Effective use of consistent naming styles.	Adequate function naming. Naming tends to be too general or have its meaning obscured by artificial shortening for brevity. Instances of inconsistent or inappropriate use of naming styles.	Frequent use of abbreviations for brevity, noun or verb phrases used. Inconsistent and irregular terminology. Regularly inconsistent variable naming style.	Arbitrary or non-descriptive function naming and arbitrary naming style.
Logical Blocks	Logical blocks are clearly delimited and consistently positioned using the standard for the language.	Logical blocks are positioned consistently.	Logical blocks are occasionally positioned consistently.	Logical blocks are regularly positioned inconsistently.	Logical blocks are positioned arbitrarily yet syntactically valid.
Solution Design					
Imperative Problem Solving	Functions are used to encourage code reuse and eliminate duplication. Global variable use is only used when essential. Each function has a single and well-defined responsibility or purpose.	Effective use of functions for code reuse and mitigate duplicate code. Global variables may be present as perceived optimizations, but not essential. Some functions contain dual purpose code.	Functions are used with occasional instances of duplicate code. Global variables are used to solve design issues. Functions generally have multiple responsibilities.	Functions are infrequently used with reliance on duplicate code. Global variables are used often. Functions are used to group blocks of code regardless of functionality. Evidence of logical issues/misunderstandings present in solution.	Functions are not used. Global variables are used as a primary means of maintaining state. Code is frequently duplicated. Logical constructs are frequently misused resulting in redundant, incorrect, or unreachable code.
OOP Concepts	Classes are used encapsulation to isolate data and behavior. Each class has a well-defined responsibility in the system. Best practice software design principles and OOP techniques are used to promote high cohesion within a class and low coupling.	Classes demonstrate effective encapsulation. Classes occasionally have more than one responsibility. OOP techniques are mostly applied for high cohesion and low coupling between classes.	Adequate class design. Encapsulation is present, but classes have multiple responsibilities. OOP techniques are used occasionally resulting in lower cohesion and higher coupling. Instances of exposing private members as public present.	Classes are regularly designed to incorporate functionality and state for convenience rather than for proper design. Global variables are used to compensate for design issues. OOP techniques are not used resulting in low cohesion and high coupling. Public members of often use for the sake of ease or misunderstanding.	Code does not follow any OOP principles. If classes are present, they are simply a container for arbitrary state and functional behavior. Result is code that would be unmaintainable outside of the present assignment.

Documentation					
Source Code Comments	Classes have header comments detailing the role and responsibility of the class in the given system. Instances of complex algorithms or difficult sections of code are clearly explained in documentation. Line comments appear near the lines they reference in a consistent position. Function comments are used appropriately give the target development language. Block comments are only used when appropriate.	Effective documentation is used to formally explain the purpose of functions and classes. Difficult lines of code are also provided explanation. Line and block comments are used interchangeably. Line comment placement can be inconsistent.	Adequate documentation is used to explain the purpose of functions. Classes are documented less frequently. Complex lines are not guaranteed to have any comment explanation. Arbitrary comment style and position.	Sporadic use documentation in the program. Relegated to seemingly arbitrary lines.	Functions, classes, and complex algorithmic components are not explained through documentation.
External Documentation	Language documentation standards and documentation tools are used correctly and to a high standard throughout the solution.	Language documentation standards are used in most cases, with effective use of documentation tools.	Some attempt has been made to follow language documentation standards and to use documentation tools.	No attempt has been made to follow language documentation standards or to use language specific documentation tools.	The solution contains no documentation to indicate its purpose.
Correctness and Testing					
Correctness	The solution produces correct results and gracefully handles exceptional cases.	The solution produces correct results in most use cases, but fails under some exceptional cases.	The solution produces correct results in the most common use cases, but produces incorrect results in some exceptional cases.	The solution runs, but crashes or produces incorrect results in many or all cases.	The solution does not compile, or it always crashes when run. Significant modifications would be necessary to bring the solution to a correct functioning state.
Testing	The codebase is rigorously tested.	The core functionality is thoroughly tested.	Basic functionality is thoroughly tested.	Basic functionality is only minimally tested.	The correctness is not tested in any capacity.