Due dates:

Late work should only follow an explanation as to why it will be late, except for medical emergencies, etc. Warn me and the TAs, and we'll discuss the right response.

Assignment 1: **April 19 (deadline is MIDNIGHT of deadline day in all cases)**

Assignment 2: **April 28th**

Assignment 3: **May 12th**

Assignment 4: **May 24th**

Final Project: **June 6 (first day of finals)**

Assignment 1:

1. Create your own copy of the base dominion code, including everything required to compile the code. Check it into your class github repository in a subdirectory called dominion
2. Read the rules of Dominion, and understand the game sufficiently to be comfortable with testing an implementation of it! If you search online, you can find the official rules and multiple web sites allowing you to play the game for free, as well as forums discussing rules questions for most cards. Your first job is to become a "subject expert" in Dominion, since you will be testing an implementation of it. Note that the primary source of information about the Dominion implementation itself is the dominion.c and dominion.h files provided in the class repository. The specification you use will have to combine this information with knowledge about how the game works, discovered by investigation. This is a typical testing experience, where you are not given a complete specification, but must discover one for yourself.
3. Pick 5 cards implemented in dominion.c. **Refactor** the code so that these cards are implemented in their own functions, rather than as part of the *switch* statement in cardEffect. You should call the functions for these cards in the appropriate place in cardEffect. Check in your changes, with appropriate svn commit messages. Document your changes in a text file in your dominion directory, called "refactor.txt," discussing the process of extracting the functions. Your implementation of at least two of these 5 cards should be incorrect in some way. **Do not document the bugs.** By bugs I mean something that does not behave correctly – it may crash, or it may cause incorrect Dominon behavior, but it is not correct. ALL CODE SHOULD COMPILE.

Assignment 2:

1. Write unit tests for four functions (not card implementations, and not cardEffect) in dominion.c. Check these tests in as unittest1.c, unittest2.c, unittest3.c, and unittest4.c. At least two of these functions should be more than 5 lines of code.

2. Write unit tests for four Dominion cards implemented in dominion.c. Do not test more than two of the cards you chose to refactor. Write these tests so that they work whether a card is implemented inside cardEffect or in its own function – go through the public dominon.h API. These tests should be checked in as cardtest1.c, cardtest2.c, cardtest3.c, and cardtest4.c
3. Execute your unit tests and describe any bugs you find in unittestbugs.txt.
4. Use gcov to measure code coverage for all of these tests. Report your findings, and describe their implications for the tests in as part of your "unittesting.txt" file (in step 6).
5. Add a rule that will generate and execute all of these tests, and append complete testing results (including coverage %ages) into a file called "unittestresults.out". The rule should be named "unittestresults.out" (that is what it produces) and it should depend on all your test code as well as the dominion code. Do this by extending the existing Makefile. You will want to test this on a standard unixlike environment (flip, flop, Cygwin or a Mac OS X machine terminal).
6. Discuss your unit testing efforts in a file called "unittesting.txt".

Helpful hint: to avoid making it hard to collect coverage when a test fails, use your own asserttrue function instead of the standard C assert (which basically crashes the code and fails to collect coverage). Your assert can also print out more information, and maybe have an option that controls whether it exits the program or not. In these and other tests, I find it helpful to make all tests print "TEST SUCCESSFULLY COMPLETED" or some other message if and only if the entire test passes, and usually (this isn't always possible for crashing bugs) print "TEST FAILED" for a failure. This makes it easy to process failing and passing tests.

Assignment 3:

1. Write a random test generator for three Dominion cards, one of them being the adventurer card, and at least one being a card you wrote unit tests for in assignment 2. Check these testers in as randomtestcard1.c, randomtestcard2.c, and randomtestadventurer.c. These testers should all take a random seed as an argument when run from the command line.
2. Add rules to the Makefile to produce randomtestcard1.out, randomtestcard2.out, and randomtestadventurer.out, including coverage results. Use some fixed seed for this part (42 works fine).
3. Write up the development of your random testers, including improvements in coverage and efforts to check the correctness of your specification by breaking the code, as randomtesting.txt. Make sure to discuss how much of adventurer and the other cards' code you managed to cover. Was there code you failed to cover? Why? For at least one card, make your tester achieve 100% statement and branch coverage, and document this (and how long the test has to run to achieve this level of coverage). It should not take more than five minutes to achieve the coverage goal (on a reasonable machine, e.g. flip or flop). Compare your coverage to that for your unit tests, and discuss how the tests differ in ability to detect faults. Which tests had higher coverage – unit or random? Which tests had better fault detection capability?

Assignment 4:

1. Write a random tester that plays, given a seed as an argument, a complete games of Dominion, with a random number of players (from 2-4) and a random set of kingdom cards. Check this in as testdominion.c, and add appropriate rules to the Makefile to create testdomininon.out by running this tester with a seed of 42. Using the tester should be as easy as typing: testdominion <seed> at a terminal. The goal of this exercise is not to write a "good" AI dominion player, but to write a good test generator that plays valid (or mostly valid – making some disallowed calls the API should catch is a good idea!) games of dominion, but may play in ways no human or sensible robot would ever dream of playing. You may need some attempt to play "well" to avoid never being able to buy/play more expensive cards, however.
2. Writing a specification for full games of Dominion is very difficult! Instead of producing a complete specification (though you can check as much as your wish in the random game generator) we are going to use differential testing. Write a program, diffdominion, in C, Python, or shell script (or something else available on any machine we are likely to test your code on) that takes as input two directory locations and a random number seed. It compiles the dominion implementations in those directory locations, and runs testdominion on both implementations. The output should state "TEST PASSED" if the two dominions behave identically, and "TEST FAILED" if they differ. When they differ, it should produce a diff of the behavior using the unix diff tool. You may need to add more output to testdominion.c to make it possible to compare the games played, and you may need to grep to remove debugging output from classmates' code, or information you random game generator prints that is useful for debugging, but not required to match between implementations. For example, two perfectly valid dominion implementations may not have the same discard piles.
3. Pick a classmate with a working Dominion implementation. Use your diffdominion to compare their dominion with yours, for several seeds. Proceed until you find a difference, and write it up as differential.txt. How easy is it to decide who is correct? Discuss why this is not an ideal case for differential testing. Also measure code coverage on both implementations, and discuss the results. Your random dominion tester should, if run more than 20 times, be able to obtain at least 60% coverage of dominion.c

Project:

In the course of assignments 1-4 you will perform testing on a Dominion implementation. Your project comes in four parts:

1. Use the unit and random tests (both card level and "whole game" level) to test code written or modified by your classmates. Find and document at least three bugs, and communicate them to your classmate, via a bug report you post in your repository, in a directory called "bugreports" (not in the dominion directory). The name of each bug report file should start with the ONID login of the student whose code you are testing.
2. Document the process of identifying and fixing a bug in your own code. You may start with a bug report from a classmate or from your own testing. Show how you used a debugger to understand the problem, and describe how (if) any of Agans' principles applied to the process.

Submit this file in your dominion directory as debugging.txt. If any files or logs are discussed in debugging.txt make sure to also submit those!

3. Pick one of the following:
    a. Use delta debugging tools (downloaded from Zeller's website; not Igor, but the Python tools most easily found by searching for "delta debugging tutorial") to minimize a failing test case for Dominion. Describe the process and what you did in deltadebug.txt, and include any relevant code (modifications of the python scripts) or files. One way to go about this is to take your work from Assignment 4 and modify it so that it produces a standalone C file containing calls to play the randomly generated Dominion game. The items to be delta-debugged will be lines in between curly braces in main.c of that file.
    b. Implement Tarantula using gcov, and describe the process of using it to localize a bug in Dominion. Write this up as tarantula.txt, and include any relevant code or files. This will require 1) generating gcov files for each test somehow (these could be random tests defined by their seeds), 2) attaching information on whether these tests pass or fail to each test somehow and 3) parsing the gcov filesand using the tarantula formula.
    c. Obtain dominion.c mutants of your code or a classmates, and perform mutation testing. Discuss your results as mutation.txt.
4. Write a test report, in pdf format, describing your experience testing Dominion. Document in detail, including code coverage information, the status and your view of the reliability of the Dominion code of at least two of your classmates. This file is submitted as testreport.pdf in your dominion directory. Your test report should have a minimum of 1,000 words of text. This covers the entire class testing process; you can base it on your unit tests, your random tests, your whole game generator, etc.