1. Random Testing

a. Overview

Each random test program tests one refactored function in the dominion game. The programs cover the baron, minion, and tribute cards, respectively. Every program initializes and begins a timer that will record the run time of all the random tests done. Each test defines a maximum number of tests, MAX_TEST, which defines how many repeated tests will be done. Each test also initializes function and game variables and initializes the game. Finally all the tests call their corresponding function using the cardEffect function. Also, the assert code used in all the random test programs follow this format:

```
printf("\n>>> Testing if number of buys increase… ");

if(teststate.numBuys == state.numBuys + buys){
   printf("PASSED… number of buys increased correctly\n");
   pass++;
}else{
   printf("FAILED… number of buys did not increase correctly\n");
   fail++;
}
```

Each random testing program also uses two gameState structs to compare a manual, or expected value, with the value that is produced by the function. These values were first initialized randomly and copied into each other which allows them to start the same. The teststate struct is then put into the function and its outputs are compared to the original state struct that was manually augmented based on the expected value of that variable.

After running the program, each test outputs the results of each individual test in a format identical to the following:

```
=== RANDOM TEST: Minion ===

|| TEST NUMBER 1 ||

>>> Testing program... PASSED... program runs

>>> Testing that cardDeck decreases...FAILED... cardDeck does not
decrease correctly

>>> Testing action points... FAILED... action points did not
increase by 1
```

```
.
.
.
|| TEST NUMBER 5 ||

>>> Testing program... PASSED... program runs

>>> Testing that cardDeck decreases...FAILED... cardDeck does not
decrease correctly

>>> Testing action points... FAILED... action points did not
increase by 1

=== END OF TESTING: Minion ===
```

This output is followed by the time it took to run the program in seconds and a count of how many passes and fails each test run had, shown below:

```
--> Testing time: 0.200000 sec
--> Passed: 1882/4000
--> Failed: 2118/4000
```

b. Baron - randomtestcard1.c

The first random test program covers the refactored baron function. The function first tests to see if the cardEffect function properly calls the refactored baron function and runs the game. The tests run in this program looks at whether or not the function executes properly (looked at the return value of the refactbaron function stored in `testret`), the number of cards in hand (`state.handCount[currentPlayer]`), the number of estates in the supply (`state.supplyCount[estate]`), and if the number of buys increase (`state.numBuys`). These tests cover the baron card's function of affecting game play depending on the players' number of estate cards and how that changes buys and deck values. Each test was repeated 1000 times. One of the requirements of this random test program is to have 100% branch and statement coverage for the refactored baron function which was achieved by randomizing the values that were changed in the struct but also randomizing the choice variables since they are used in conditional statements inside the refactored baron function.

c. *Minion - randomtestcard2.c*

The second random test program looked at the refactored minion code. Since the beginning this code has been extremely buggy making it difficult to run the code to test. My random test for the minion code faced many problems of running into segmentation faults when I set MAX_TEST any value of 10 or above, unfortunately I couldn't figure out why this was happening so I had to set the number of tests to 5. Despite this, this program tests that the program runs (`testret == 0`), the deck of cards decreased after cards are drawn (`state.deckCount[currentPlayer]`), and that action points increase (`state.numActions`). These tests make sure that the function executes and runs by asserting that the return value of the function was 0. Next, I tested that the number in the deck decreased since a card was drawn out of the deck in the function. Finally, I tested that action points increased the correct amount by 1 after a minion card was played. These tests were repeated 5 times. The assignment requires that this random test needed to cover at least 70% of the minion function and I obtained about 80% statement and branch coverage.

d. *Tribute - randomtestcard3.c*

The final random test program took a look at the refactored tribute function. I tested if the code runs (`testret == 0)`, as I did for the other two functions, and I tested if the action points (`state.numActions)` and buys increased (`state.numBuys)`, and if the card deck decreases (`state.deckCount[currentPlayer]`). I expected the action points to increase by 2 when the tribute card is played so I manually changed the action points by adding 2 to the initialized value and compared it to the tested value. I did the same thing with the buys. Finally, I tested the card deck number the same way I did in randomtestcard2.c for the minion function. Each test was repeated 1000 times.

2. Code Coverage

a. *Baron Card*

Output of time ran for random testing for refactored baron function which can be found in randomtestresults.out:

```
=== END OF TESTING: refactBaron ===

--> Testing time: 0.200000 sec
```

Timing the tests using time.h, the results to go through 1000 random tests took 0.2 seconds (±0.02 seconds) which allowed a 100% statement and branch coverage according to gcov results shown below:

```
Function 'refactBaron'
Lines executed:100.00% of 32
Branches executed:100.00% of 20
Taken at least once:100.00% of 20
Calls executed:100.00% of 9
.
.
.
File 'dominion.c'
Lines executed:63.78% of 577
Branches executed:74.24% of 427
Taken at least once:57.38% of 427
Calls executed:51.52% of 99
```

Using randomization of choice variables, it allowed me to increase the branch coverage which allowed me to reach 100% statement and branch coverage on the refactBaron function. The code that I failed to cover in the dominion source code was mainly the other refactored code that was never used during the randomized gameplay. For example, the refactored mine card was never called and therefore never executed. Other parts of the code that wasn't used were other game mechanic functions like newGame and getCost which happened to not be used completely during the randomized runs. One thing that was suspicious and may indicate that the program failed to run a full game is that getWinners was never called. This may indicate that the path from cardEffect that leads to getWinners needs to be further tested as there may be a bug preventing that function from being called.

b. Minion Card

Output of time running the random test for minion:

```
=== END OF TESTING: Minion ===

--> Testing time: 0.000000 sec
```

This test was really fast mainly because the refactored minion code would not accept high numbers of testing which required me to limit the maximum number of tests to 5.

```
Function 'refactMinion'
Lines executed:81.82% of 22
Branches executed:80.00% of 20
Taken at least once:75.00% of 20
Calls executed:71.43% of 7
.
.
.
File 'dominion.c'
Lines executed:55.11% of 577
Branches executed:67.68% of 427
Taken at least once:47.07% of 427
Calls executed:40.40% of 99
```

I achieved 81.82% statement coverage and 80% branch coverage which is above the 70% requirement for the assignment. This was easily achieved by increasing the number of random variables and increasing the test cases that I looked at. Originally I did not test for the card deck amount but when I added a randomized initialized variable for the cardDeck amount I was able to increase the coverage to above 70%. Code not covered was again similar to the ones not covered. Again, getWinners did not get called which confirms my suspicions that there is an issue when calling that function and finishing the game.

c. *Tribute Card*

Output time of randomtestcard3:

```
=== END OF TESTING: Tribute ===

--> Testing time: 0.070000 sec
```

This random test was very quick considering 1000 tests were run, but something that I was really suspicious about was that the number of passes and fails were perfectly 1:3 which is shown in the output here:

```
--> Passed: 1000/4000
--> Failed: 3000/4000
```

This indicates that there may be something wrong when randomizing the values in the test. This also may indicate that there is a really blatant bug when action points and buys are increased and when the card deck decreases since they consistently fail on every test run.

Below is the gcov coverage report for the refactored tribute function and the dominion source code:

```
Function 'refactTribute'
Lines executed:73.68% of 38
Branches executed:93.75% of 32
Taken at least once:65.63% of 32
Calls executed:75.00% of 4
.
.
.
File 'dominion.c'
Lines executed:52.51% of 577
Branches executed:65.34% of 427
Taken at least once:45.20% of 427
Calls executed:34.34% of 99
```

The coverage of the refactored tribute was above the required 70% at 73.68% statement coverage and 93.75% branch coverage. Again, code not covered were mostly unused card functions or other functions that aren't required during game play.

d. Overall

When looking at the code that wasn't covered during testing, many were just functions of cards that were never played. This prevented the dominion source code to reach coverage over % at any given test run. Something that I will check out is the newGame and the getWinner function which both never called and was never run. Other functions like fullDeckCount was never run because my randomizer never runs when deck is full since players will always have some cards out of the deck.

In order to improve this random test suite, I would definitely try to examine these functions and try to create some paths that will allow these functions to be called and used. I could also test more cases that don't necessarily happen every run which I didn't really do for this test suite. I focused on a lot more straightforward cases that didn't require several criteria to be fulfilled before the variable was changed. I would also improve the test result output to be more informative. I would want to better summarize how many time specific cases passed or failed since the way I outputted it now makes it difficult to tell which test may indicate a bug exists. Doing these improvements would also increase coverage of the source code which will make the test suite more thorough.

3. Unit vs. Random Testing

When comparing the coverage results of my unit tests and my random tests, my random tests had significantly higher coverage. I addressed about the same number of tests for my unit tests and my random tests but since my random tests had a greater capability for iteration and testing for various cases (with the help of automation!) which consequently, helped increase branch coverage which also increases statement coverage. I think that my unit tests were not exhaustive enough and never really exceeded 30% source code coverage since each test was very (and maybe too) specific and my random testing covered over 60%.

In terms of fault detection, I think the unit testing gave more specific results and was more telling on what exactly in the code is behaving unexpectedly. Since the inputs of the random testing is varied and changing, it can be hard to retrace steps to find where the issue of the fault is and where the bug is located.

I think a combination of the two would be ideal and most effective since both the specificity of the unit testing will compliment the exposure of the random testing. Using unit testing after random testing can also be helpful to hone in on the fault and find the root of the bug by finding the broad areas where faults are generally coming from.