# Bingo Bonanza

## Testing Documentation

| | |
|---|---|
| **Name(s):** | Brendan Simms , Shane Lennon |
| **Student Number(s):** | 19500949 , 17496766 |
| **Module Code:** | CA400 |
| **Supervisor:** | Mark Humphries |
| **Date Of Completion:** | 06/05/2023 |

## Table Of Contents

## 1. Testing Types

Due to our project consisting of a large amount of both server and client side code there was a need to research all the possible testing that would be fruitful when testing our application to ensure that both the backend and front end were tested appropriately.

In terms of our backend we determined we would use Unit testing to test our Lambda functions, this would allow us to ensure that the functions execute as intended. To do this we used the unit testing library in Python as we both had experience with this framework and it would ensure our code was of a sufficiently high level.

In terms of our front end the main way we tested this implementation was using User testing and Google chromes built in Lighthouse for testing our webpages accessibility and performance. The User Testing was an event hosted by the Louth Mavericks where we got 20 participants to play a game of Bingo using testing accounts we created. This testing event was very successful in our opinion and we created google forms for the testers to rate their experience and give us accurate feedback on the event. Google Lighthouse provides metrics with a core out of 100 based on how your webpage preforms this was incredibly useful as we could run these tests on both Mobile and Desktop devices and contrast and differences.

## 2. Adhoc Testing

We made sure to use Adhoc testing throughout the development of our Application to ensure that the components worked as intended before we introduced any unit testing. We would use dummy data we generated to ensure that our feature worked as intended. This was a frequent strategy we would use before implementing the component into our overall application. Obviously this is not a sufficient high level of testing so we would not just use this testing on its own but would rather use it to confirm that we believed the component was working as intended before we developed further tests to ensure it. This was extremely useful as there were several instances where we found code not always preforming as intended on occasion when ran. We would then be able to fix these bugs before carrying out our Unit testing and could ensure our unit testing made sure the bug did not occur again. This allowed us to save a large amount of time while ensuring that the integrity of our overall application had not been breached.

## 3. Unit Testing

As aforementioned, we used Python unit Testing library to test the Lambda functions of our application. In this section we will go through each component pertaininent to this testing type and outline what the tests do and the purpose of the tests. We split the functionality of our testing as much as possible to ensure that the code preformed as reliably as possible. To achieve this we tested the individual functions within our code rather than the entire program as a whole.

### 3.1 Book Generation Unit Testing

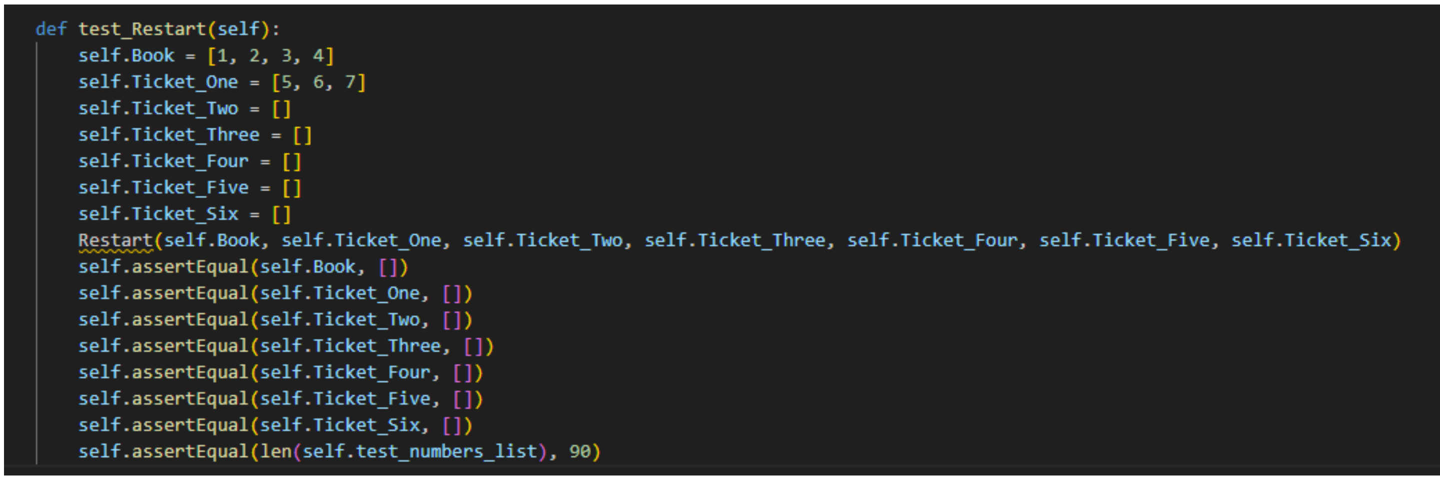
In Book generation there was several vital functions that needed to be tested they are as follows:

- Restart Function
- Clear Lines Of Ticket
- Calculate Limits Function
- CheckPossibility function

- Append Tickets
- Generate Book

## Restart Function

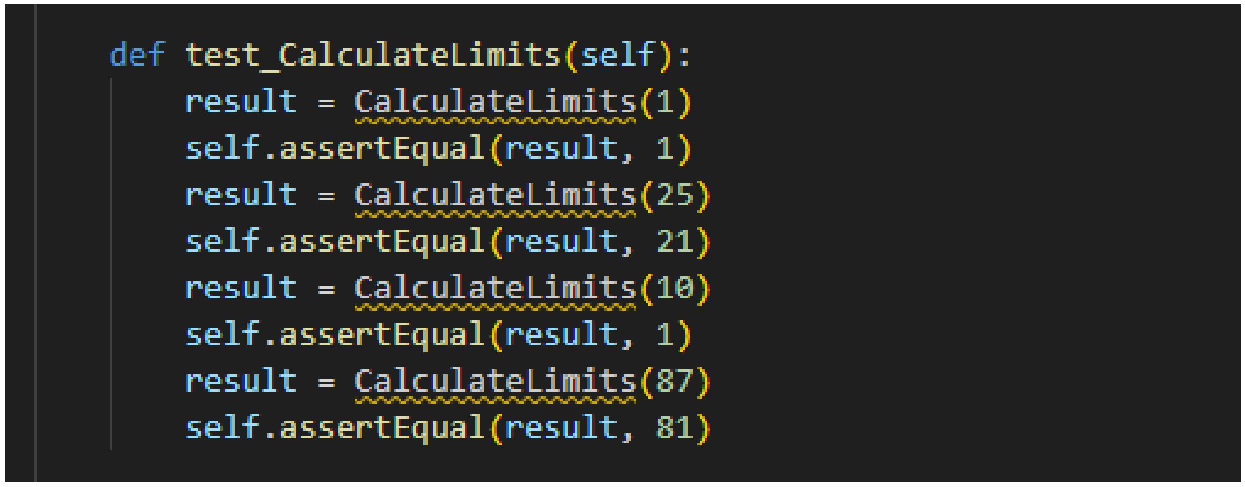
The restart function is vital as it was used to prevent an infinite loop issue we had early when developing our code, it essentially reset the program so that the process can restart if the rule of no line containing two numbers in the same range (1-10, 11-20, … so on) cannot be proven to be true. The Restart function works by resetting our data structures to their default state of an empty list. The code is as follows:



```
def test_Restart(self):
    self.Book = [1, 2, 3, 4]
    self.Ticket_One = [5, 6, 7]
    self.Ticket_Two = []
    self.Ticket_Three = []
    self.Ticket_Four = []
    self.Ticket_Five = []
    self.Ticket_Six = []
    Restart(self.Book, self.Ticket_One, self.Ticket_Two, self.Ticket_Three, self.Ticket_Four, self.Ticket_Five, self.Ticket_Six)
    self.assertEqual(self.Book, [])
    self.assertEqual(self.Ticket_One, [])
    self.assertEqual(self.Ticket_Two, [])
    self.assertEqual(self.Ticket_Three, [])
    self.assertEqual(self.Ticket_Four, [])
    self.assertEqual(self.Ticket_Five, [])
    self.assertEqual(self.Ticket_Six, [])
    self.assertEqual(len(self.test_numbers_list), 90)
```

## Calculate Limits Function

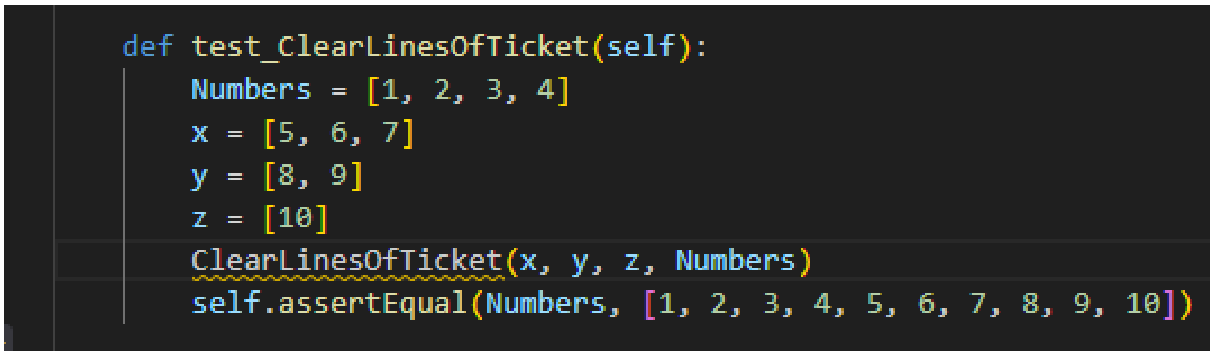
This function is used to calculate the lower range of the integer given to it as a parameter, the lower limit should be: the closest multiple of 10 less than the number + 1. This is used to ensure that the golden rule mentioned in the Restart function section is adhered too. Without this code working we cannot tell what the limits for a particular number are and therefore cannot calculate if the rule will be broken. The code to test this function is as follows:



```
def test_CalculateLimits(self):
    result = CalculateLimits(1)
    self.assertEqual(result, 1)
    result = CalculateLimits(25)
    self.assertEqual(result, 21)
    result = CalculateLimits(10)
    self.assertEqual(result, 1)
    result = CalculateLimits(87)
    self.assertEqual(result, 81)
```

## Clear Lines of Tickets

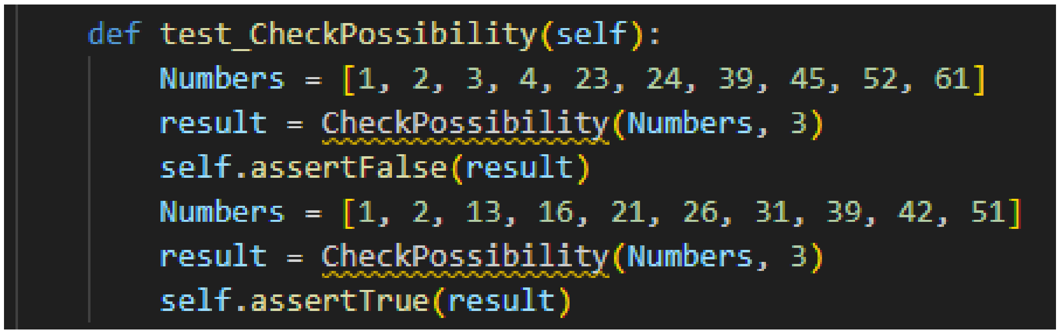
This function is used to ensure that the individual lines within a ticket are reset when needed, thai is essential as we had an issue when resetting the overall data strictures that the lines would not reset causing some interesting issues when the application determined it needed to re run the entire process from scratch. This function simply resets the lists to empty, and that the numbers removed form the List are put back into the overall Numbers List containing one instance of each number from 1-90. The test for this section of code is as follows:



```
def test_ClearLinesOfTicket(self):
    Numbers = [1, 2, 3, 4]
    x = [5, 6, 7]
    y = [8, 9]
    z = [10]
    ClearLinesOfTicket(x, y, z, Numbers)
    self.assertEqual(Numbers, [1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
```

## Check Possibility

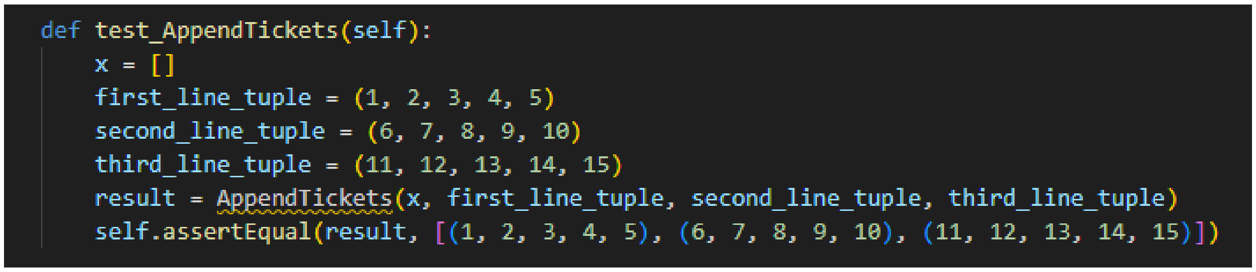
This function is arguably our most vital function within the Book Generation as this function is used to signal if the process will be impossible to complete and then runs the functions to reset our data structures if it is. The Check Possibility function works by comparining each item in the Numbers List lower limit with every other item in the lists lower limit, if these values are the same the counter is incremented and if the counter is greater than the second parameter given to the function than it return false otherwise it returns True. The code to test it was as follows:



```
def test_CheckPossibility(self):
    Numbers = [1, 2, 3, 4, 23, 24, 39, 45, 52, 61]
    result = CheckPossibility(Numbers, 3)
    self.assertFalse(result)
    Numbers = [1, 2, 13, 16, 21, 26, 31, 39, 42, 51]
    result = CheckPossibility(Numbers, 3)
    self.assertTrue(result)
```

## Append Tickets

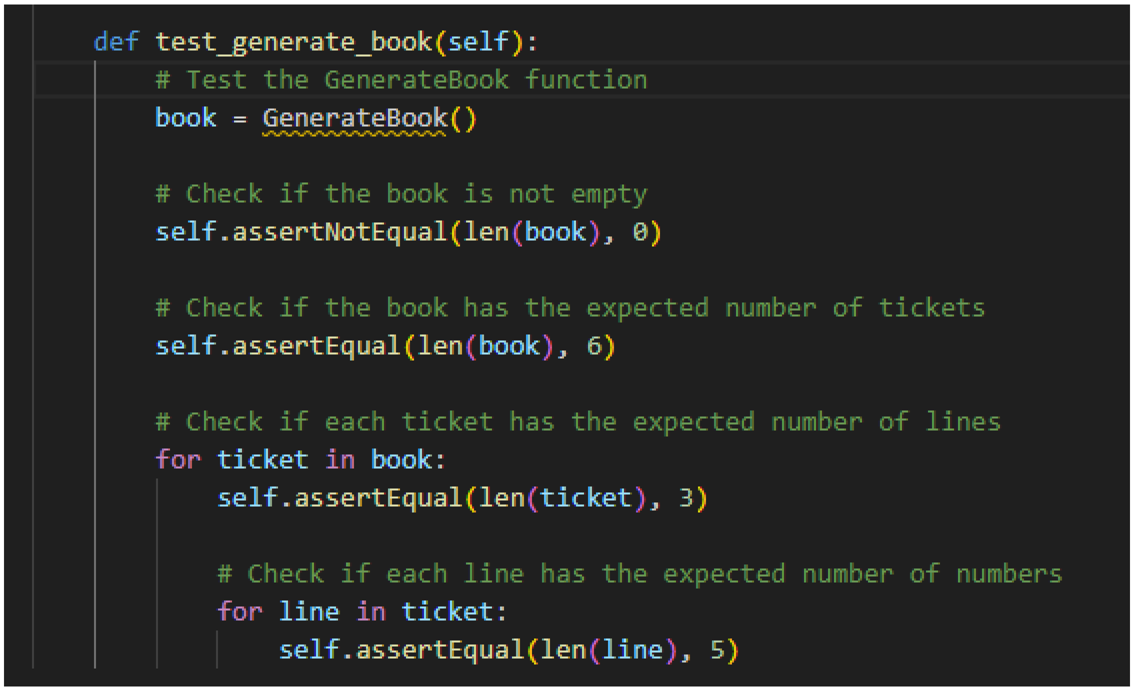
This function is used to determine which Ticket the current Lines belong too and then appends each of the three line tuples into the List for that Ticket, this is important as it verifies the tuples are able to be appened into the List without any issue. The code to test this is as follows:



```
def test_AppendTickets(self):
    x = []
    first_line_tuple = (1, 2, 3, 4, 5)
    second_line_tuple = (6, 7, 8, 9, 10)
    third_line_tuple = (11, 12, 13, 14, 15)
    result = AppendTickets(x, first_line_tuple, second_line_tuple, third_line_tuple)
    self.assertEqual(result, [(1, 2, 3, 4, 5), (6, 7, 8, 9, 10), (11, 12, 13, 14, 15)])
```

## Generate Books

This is the overall runner function that generates the entire Book for use by the application, This is verified to be working by ensuring that each Data structure within the book has the expected length of data within it. The code to test this is as follows:


```python
    def test_generate_book(self):
        # Test the GenerateBook function
        book = GenerateBook()

        # Check if the book is not empty
        self.assertNotEqual(len(book), 0)

        # Check if the book has the expected number of tickets
        self.assertEqual(len(book), 6)

        # Check if each ticket has the expected number of lines
        for ticket in book:
            self.assertEqual(len(ticket), 3)

            # Check if each line has the expected number of numbers
            for line in ticket:
                self.assertEqual(len(line), 5)
```

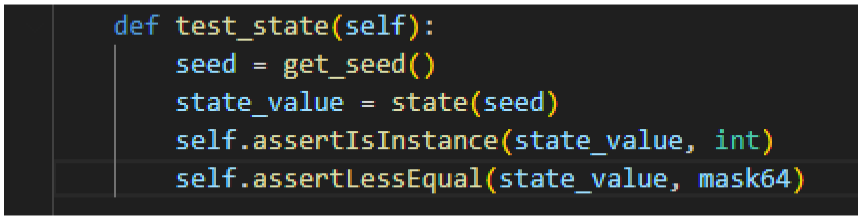
## 3.2 XORShift* Unit Testing

XORShift star has a few functions that needed testing they are

- State
- Next Number
- Decimal to binary
- XORStarMain

### State Function

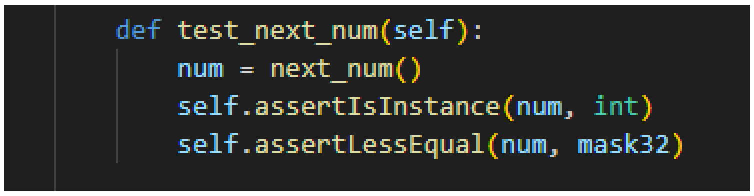
The state function takes a seed and It initializes a variable named state to 0 bitwise ANDed with mask64. It assigns the seed value bitwise ANDed with mask64 to the state variable. This operation ensures that the state value is within the range of mask64.


```python
    def test_state(self):
        seed = get_seed()
        state_value = state(seed)
        self.assertIsInstance(state_value, int)
        self.assertLessEqual(state_value, mask64)
```

### Next Number Function

This function generates a pseudorandom number based on the current state value.

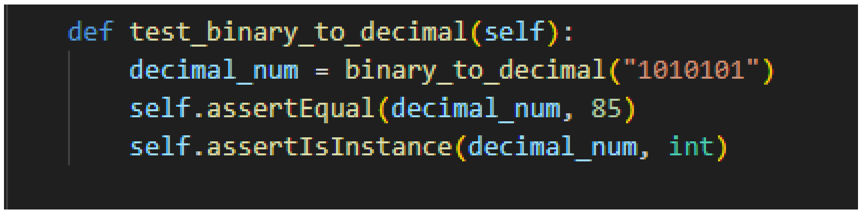
It performs a series of bitwise operations on x to generate a modified state value:
(x ^ (x >> 12)) performs a bitwise XOR between x and the result of right-shifting x by 12 bits.
(x ^ (x << 25)) performs a bitwise XOR between x and the result of left-shifting x by 25 bits.
(x ^ (x >> 27)) performs a bitwise XOR between x and the result of right-shifting x by 27 bits.
These bitwise XOR operations introduce randomness and help generate a new state value.
After each bitwise operation, the result is bitwise ANDed with mask64 to ensure it falls within the desired range. We then preform a serious of operations to ensure a pseudo random number is outputted in the range of mask64.



```
def test_next_num(self):
    num = next_num()
    self.assertIsInstance(num, int)
    self.assertLessEqual(num, mask32)
```

## Decimal to binary Function

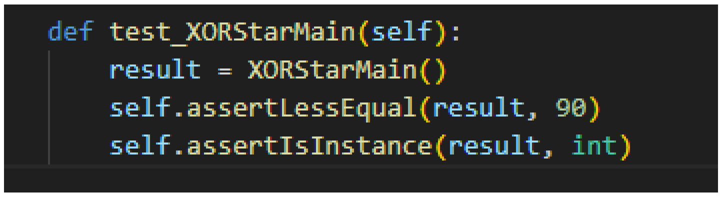
This function is quite important as the XORStar will give us binary numbers that are too large to use in our bingo game as the numbers needed are from (1-90). So to fix this issue we decided to split the binary into binary bits of 7 as all the numbers from 1-90 can be represented by this. However numbers greater than 90 can also be given so we developed our next function to handle this.



```
def test_binary_to_decimal(self):
    decimal_num = binary_to_decimal("1010101")
    self.assertEqual(decimal_num, 85)
    self.assertIsInstance(decimal_num, int)
```

## XORStarMain Function

This function Gets our Number and converts it from binary to decimal calling the function above it then checks using the while loop to make sure the number length is less than 7. It then calls the next num function and then it returns the Final Number.



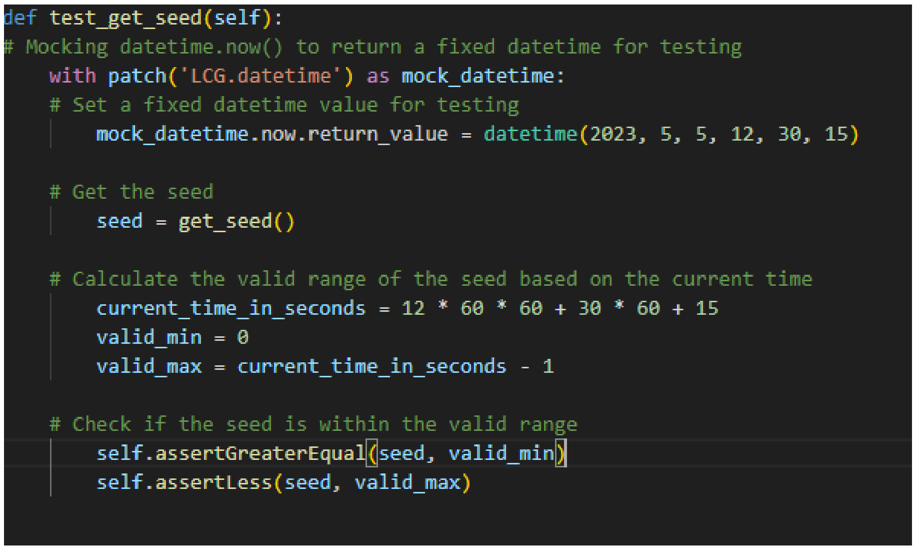
```
def test_XORStarMain(self):
    result = XORStarMain()
    self.assertLessEqual(result, 90)
    self.assertIsInstance(result, int)
```

## 3.3 LCG Unit Testing

There isn't a lot of functions to test in the LCG but we will talk about each of them they include

- LCGMain
- get_seed

Our get_seed function generates a seed for us to use for our pseudo random number generation. It uses current time it gets the current hours minutes and converts them into seconds and adds them all up with current seconds to be our seed.



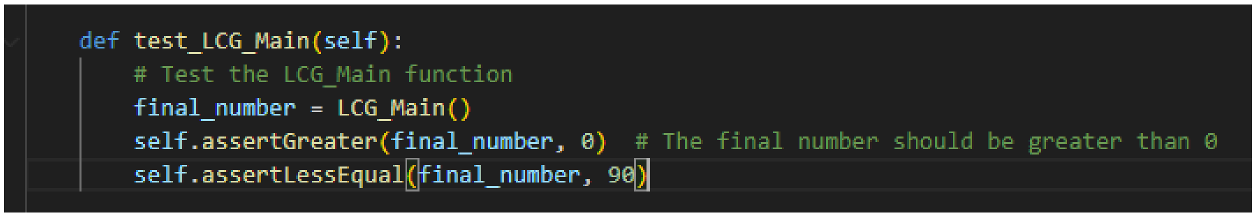
```
def test_get_seed(self):
# Mocking datetime.now() to return a fixed datetime for testing
    with patch('LCG.datetime') as mock_datetime:
    # Set a fixed datetime value for testing
        mock_datetime.now.return_value = datetime(2023, 5, 5, 12, 30, 15)

    # Get the seed
        seed = get_seed()

    # Calculate the valid range of the seed based on the current time
        current_time_in_seconds = 12 * 60 * 60 + 30 * 60 + 15
        valid_min = 0
        valid_max = current_time_in_seconds - 1

    # Check if the seed is within the valid range
        self.assertGreaterEqual(seed, valid_min)
        self.assertLess(seed, valid_max)
```

The other function LCGMain is the main part of the LCG and handles all calculations, this function assigns our constants and then multiplies the seed by one of the constants the result is then added to another variable constant and finally the result is modulo of another variable constant. This results in our final number generated.



```python
def test_LCG_Main(self):
    # Test the LCG_Main function
    final_number = LCG_Main()
    self.assertGreater(final_number, 0)  # The final number should be greater than 0
    self.assertLessEqual(final_number, 90)
```

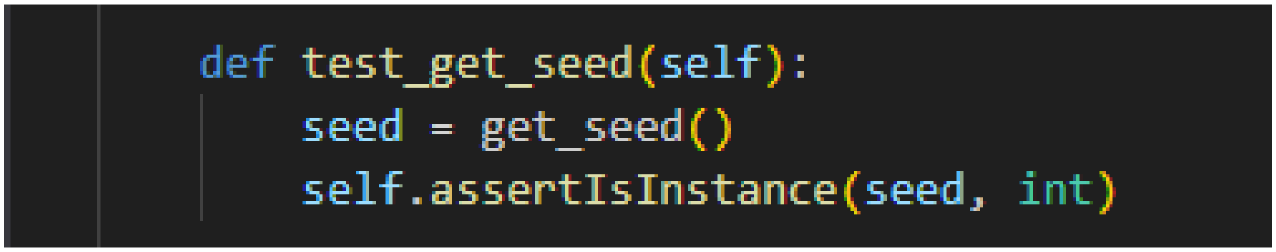
## 3.4 Blum Blum Shub Unit Testing

In Blum Blum Shub there are several functions that needed to be tested they are as follows:

- Get_Seed Function
- Is_Prime Function
- get_Primes Function
- Co_Primes Function
- get_M Function
- BBS_Main Function

### Get_Seed

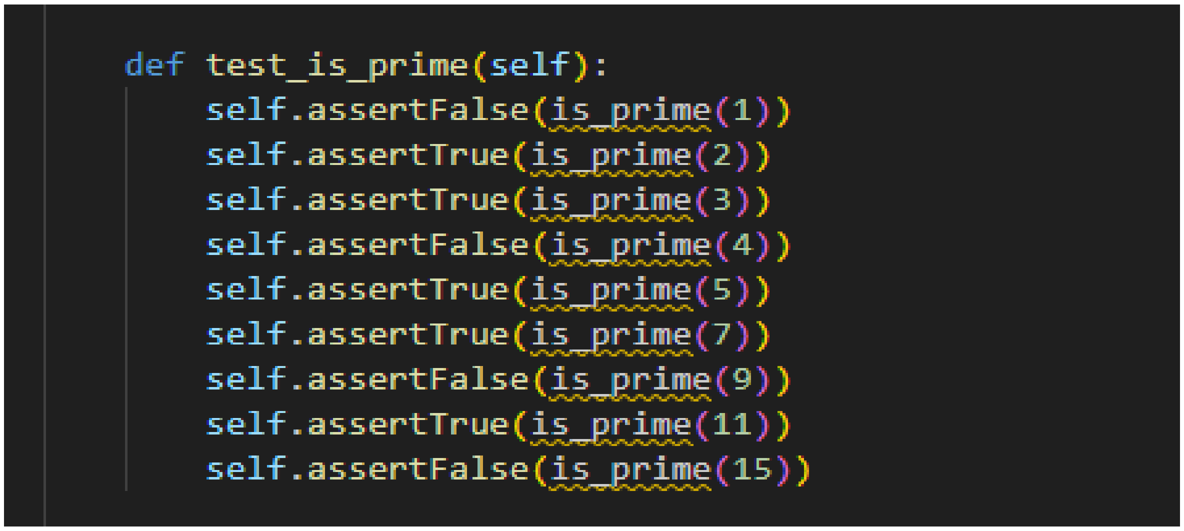
This function is quite a simple one to test the whole purpose of this function is to generate a random number below the system time in Seconds. To ensure that this function works as intended we just need to ensure that the Return of the function is indeed an integer value. This code is tested as follows:



```python
def test_get_seed(self):
    seed = get_seed()
    self.assertIsInstance(seed, int)
```

### Is_Prime

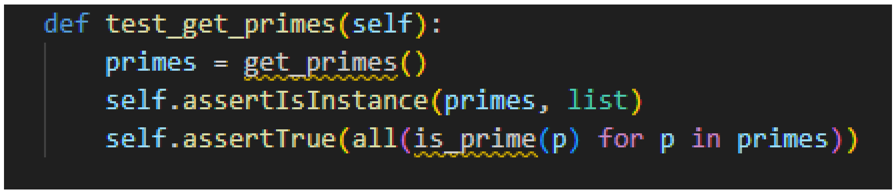
This function is used within our code to ensure that the numbers generated for our Blum Blum Shub alogirhtim are primes. This is necessary due to the way Blum Blum Shub works it relies on two large co-prime numbers to generate an integer M for the PRNG. The test of this code is as follows:



```
def test_is_prime(self):
    self.assertFalse(is_prime(1))
    self.assertTrue(is_prime(2))
    self.assertTrue(is_prime(3))
    self.assertFalse(is_prime(4))
    self.assertTrue(is_prime(5))
    self.assertTrue(is_prime(7))
    self.assertFalse(is_prime(9))
    self.assertTrue(is_prime(11))
    self.assertFalse(is_prime(15))
```

## Get_Prime

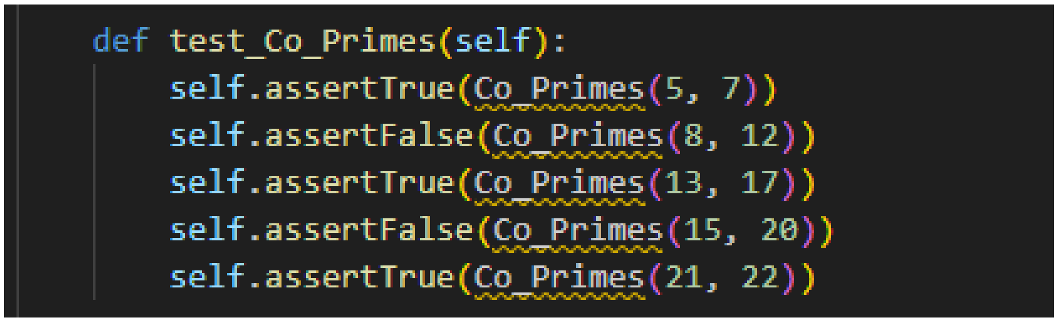
This function creates a list of numbers where the max is 10,000. It contains a loop that starts at 2 ie. the first prime and increments by 1 up to 10,000 and calls the is_prime() function to check if they are primes if so it adds them to the list so we have a big list of prime numbers. The large list of primes is need to generate our M for BBS formula.



```
def test_get_primes(self):
    primes = get_primes()
    self.assertIsInstance(primes, list)
    self.assertTrue(all(is_prime(p) for p in primes))
```

## Co_Primes

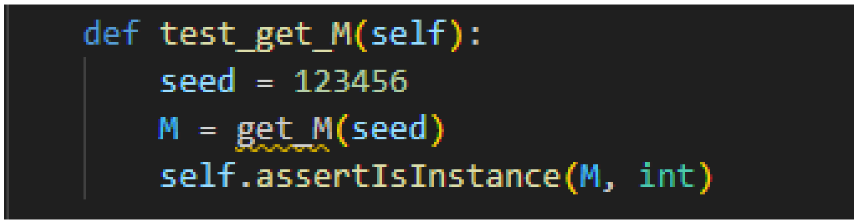
This function is udes within our code to determine if the two primes generated for P and Q are indeed CoPrimes, a co prime is defined as a pair of two numbers that do not share any factor other than one, the function determines this by calculating the greatest common factor between two numbers and if it is one the function returns True otherwise it returns False. We tested this function as follows:



```
def test_Co_Primes(self):
    self.assertTrue(Co_Primes(5, 7))
    self.assertFalse(Co_Primes(8, 12))
    self.assertTrue(Co_Primes(13, 17))
    self.assertFalse(Co_Primes(15, 20))
    self.assertTrue(Co_Primes(21, 22))
```

## Get_M

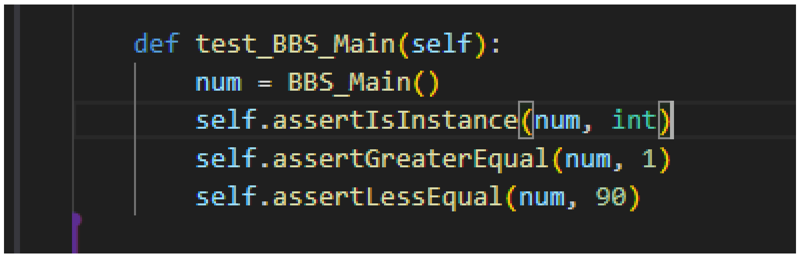
M is defined as the product of multiplying our two primes P and Q by each other, in terms of testing this function all we can truly do is ensure that the value returned by it is always an integer. The test of this code is as follows:



```
def test_get_M(self):
    seed = 123456
    M = get_M(seed)
    self.assertIsInstance(M, int)
```

## BBS_Main

BBS_Main function essentially ensure that the number being returned is a number between (1-90) it calls get_num() on our seed in a while loop with the conditions of final number being equal to 0 or final number being less than 90.
Testing this we just ensure that the number is an integer, greater than or equal to 1 and less than or equal to 90.



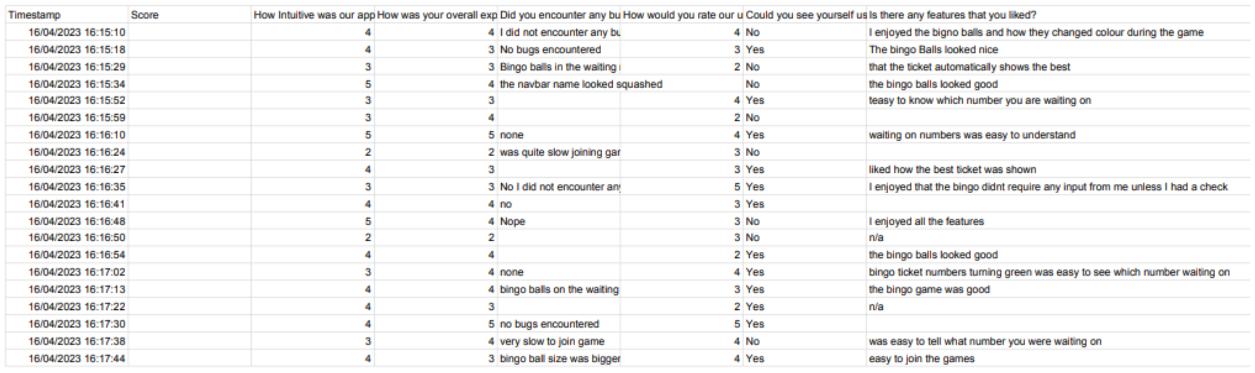
```
def test_BBS_Main(self):
    num = BBS_Main()
    self.assertIsInstance(num, int)
    self.assertGreaterEqual(num, 1)
    self.assertLessEqual(num, 90)
```

## 4. User Testing

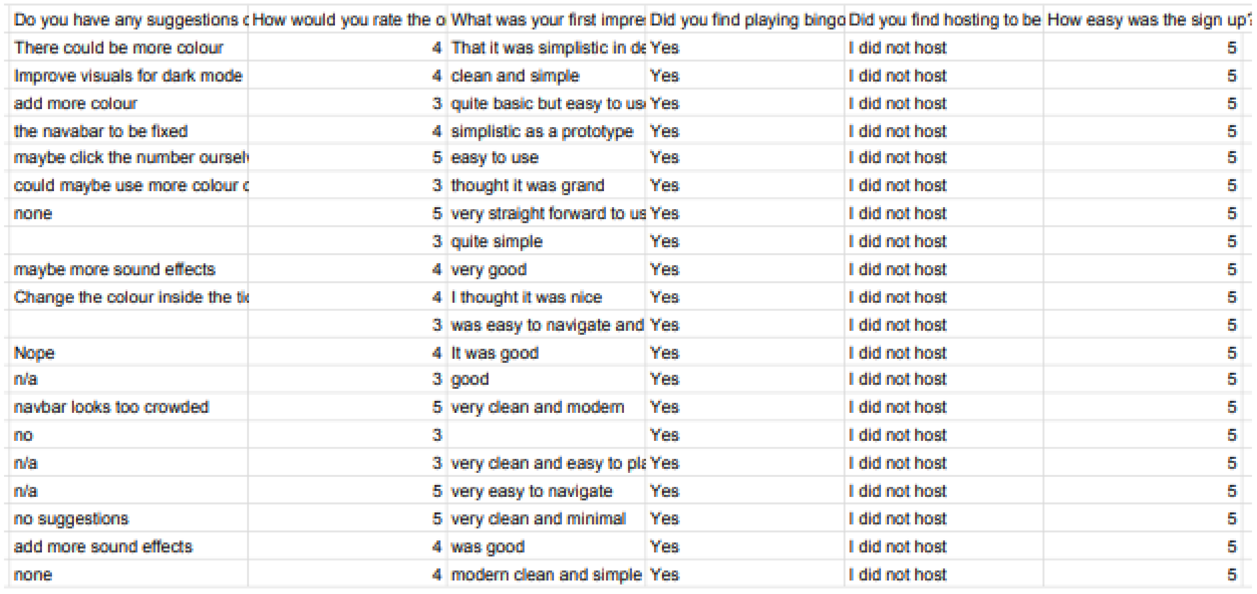
We were fortunate to be able to host a real world event for the user testing of our application. This is thanks to the local American Football club in Dundalk, after one of their games we were able to get 20 people to play a game of Bingo and fill in a form relating to the user experience of the testing event. The survey focused on what the users thought about the application, if they experienced any bugs and their overall experience while playing the game. The form and the questions asked can be seen by accessing this link:

https://docs.google.com/forms/d/e/1FAIpQLSfyrvQveC_8RryQjDpxdb3Kj27pFqM9d6Io9ztOtiK5Kyuwxg/viewform?usp=sf_link

The results of these forms will be concluded here, overall we believe that the testing event was very successful. Most of the responses collaborate that information although the testing event was very useful as the testers pointed out UI details that could be improved upon. These UI issues were mostly caused as the code focused on the web application rather than the mobile version which we later fixed down the road to ensure that our application was fully responsive and appeared as intended regardless of what size screen the application was being used on.



| Timestamp | Score | How Intuitive was our app | How was your overall exp | Did you encounter any bu | How would you rate our u | Could you see yourself us | Is there any features that you liked? |
|---|---|---|---|---|---|---|---|
| 16/04/2023 16:15:10 | | 4 | 4 | I did not encounter any bu | 4 | No | I enjoyed the bigno balls and how they changed colour during the game |
| 16/04/2023 16:15:18 | | 4 | 3 | No bugs encountered | 3 | Yes | The bingo Balls looked nice |
| 16/04/2023 16:15:29 | | 3 | 3 | Bingo balls in the waiting | 2 | No | that the ticket automatically shows the best |
| 16/04/2023 16:15:34 | | 5 | 4 | the navbar name looked squashed | | No | the bingo balls looked good |
| 16/04/2023 16:15:52 | | 3 | 3 | | 4 | Yes | teasy to know which number you are waiting on |
| 16/04/2023 16:15:59 | | 3 | 4 | | 2 | No | |
| 16/04/2023 16:16:10 | | 5 | 5 | none | 4 | Yes | waiting on numbers was easy to understand |
| 16/04/2023 16:16:24 | | 2 | 2 | was quite slow joining gar | 3 | No | |
| 16/04/2023 16:16:27 | | 4 | 3 | | 3 | Yes | liked how the best ticket was shown |
| 16/04/2023 16:16:35 | | 3 | 3 | No I did not encounter an | 5 | Yes | I enjoyed that the bingo didnt require any input from me unless I had a check |
| 16/04/2023 16:16:41 | | 4 | 4 | no | 3 | Yes | |
| 16/04/2023 16:16:48 | | 5 | 4 | Nope | 3 | No | I enjoyed all the features |
| 16/04/2023 16:16:50 | | 2 | 2 | | 3 | No | n/a |
| 16/04/2023 16:16:54 | | 4 | 4 | | 2 | Yes | the bingo balls looked good |
| 16/04/2023 16:17:02 | | 3 | 4 | none | 4 | Yes | bingo ticket numbers turning green was easy to see which number waiting on |
| 16/04/2023 16:17:13 | | 4 | 4 | bingo balls on the waiting | 3 | Yes | the bingo game was good |
| 16/04/2023 16:17:22 | | 4 | 3 | | 2 | Yes | n/a |
| 16/04/2023 16:17:30 | | 4 | 5 | no bugs encountered | 5 | Yes | |
| 16/04/2023 16:17:38 | | 3 | 4 | very slow to join game | 4 | No | was easy to tell what number you were waiting on |
| 16/04/2023 16:17:44 | | 4 | 3 | bingo ball size was bigger | 4 | Yes | easy to join the games |



| Do you have any suggestions c | How would you rate the o | What was your first impre | Did you find playing bingo | Did you find hosting to be | How easy was the sign up |
|---|---|---|---|---|---|
| There could be more colour | 4 | That it was simplistic in de | Yes | I did not host | 5 |
| Improve visuals for dark mode | 4 | clean and simple | Yes | I did not host | 5 |
| add more colour | 3 | quite basic but easy to us | Yes | I did not host | 5 |
| the navabar to be fixed | 4 | simplistic as a prototype | Yes | I did not host | 5 |
| maybe click the number oursel | 5 | easy to use | Yes | I did not host | 5 |
| could maybe use more colour c | 3 | thought it was grand | Yes | I did not host | 5 |
| none | 5 | very straight forward to us | Yes | I did not host | 5 |
| | 3 | quite simple | Yes | I did not host | 5 |
| maybe more sound effects | 4 | very good | Yes | I did not host | 5 |
| Change the colour inside the ti | 4 | I thought it was nice | Yes | I did not host | 5 |
| | 3 | was easy to navigate and | Yes | I did not host | 5 |
| Nope | 4 | It was good | Yes | I did not host | 5 |
| n/a | 3 | good | Yes | I did not host | 5 |
| navbar looks too crowded | 5 | very clean and modern | Yes | I did not host | 5 |
| no | 3 | | Yes | I did not host | 5 |
| n/a | 3 | very clean and easy to pla | Yes | I did not host | 5 |
| n/a | 5 | very easy to navigate | Yes | I did not host | 5 |
| no suggestions | 5 | very clean and minimal | Yes | I did not host | 5 |
| add more sound effects | 4 | was good | Yes | I did not host | 5 |
| none | 4 | modern clean and simple | Yes | I did not host | 5 |

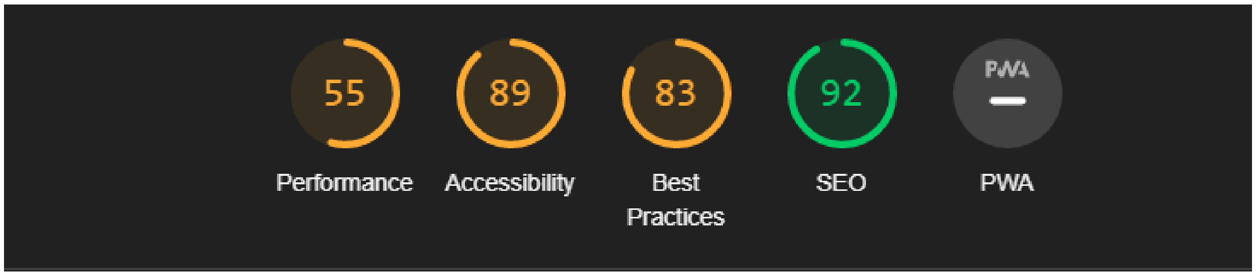
The results of the questionnaire can be found via the link:
https://drive.google.com/file/d/1ENedkHbHNJWWbIQLKOT6YcKh5PK62Edx/view?usp=share_link

We used this form to improve upon our UI after the testing event and make the application more refined. We still ensured that the basic concepts of minimalistic design and simplicity were there but just added more refined UI choices to really make important aspects of the Application stand out and be more apparent to the users. One of the refinements we made was on the Navbar on a mobile screen. The mobile screen caused our NavBar to look distorted with most of the application name being removed from being visible. We decided that in the Mobile version we would simply just use our Icon instead of using a text of our Projects Name.

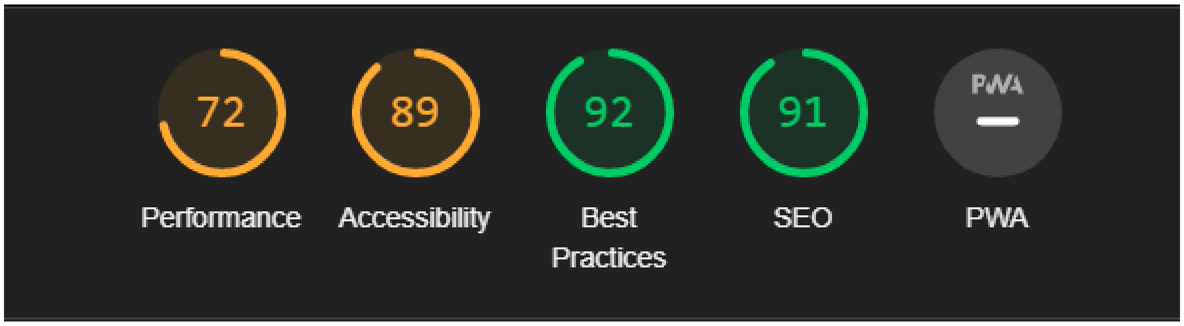
## 5. UI Testing

As aforementioned we used google Lighthouse to compare our Front end performance between Mobile and Desktop versions of our application, this was of high importance to us as we wanted to give users the ability to play on mobile devices or desktop applications.

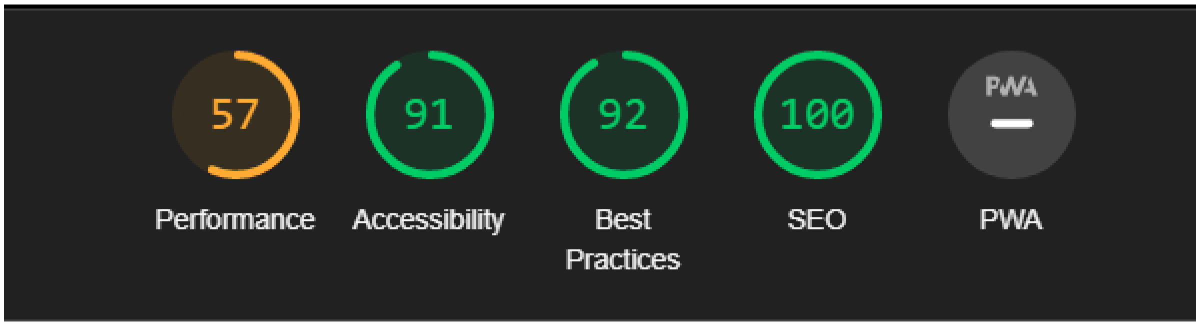
*HostGame Mobile*



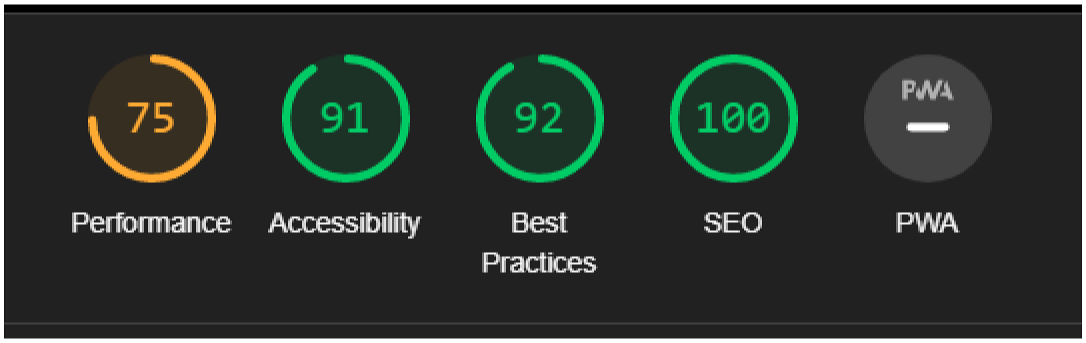
*HostGame Desktop*



*Authentication Mobile*



*Authentication Desktop*



In the early stages of this testing between mobile and desktop versions of the app the results would dramatically decrease when switching to the mobile version of the application we were able to improve upon this to a high level but were not able to get this perfectly balanced. This allowed us to highlight parts of our application that were redundant and could be remove while also suggesting other potential ways in which the webpages could be improved for each metric.

## 6. Conclussion

We used a multitude of different testing techniques to ensure that our application was validated and consistent with the results we expected it to get. While the previous statement is true there are some further refinements we could made upon our testing to get this validation to an even higher level. We could've introduced integration testing for our backend to ensure that the components that use this use this as expected and the results returned by our backend to the frontend components is accurate to the results they return when tested individually. On top of the UI testing we carried out we could've used the Jest Javascript framework or snapshot testing to further refine our testing of the front end components of the application.