About the project

The task was to create a random number generator. That generator had to be created following linear-same_remainder with given parameters method. In the end, the generator's quality had to be determined by commenting on the results of specific tests on it.

Problem and Solution

The problem with generators is that many methods - including linear-same_remainder - use modulo calculus with big numbers that are Integers (big Z set). This is so there is a big period until you encounter the same random number again.
Programming and making computations with big integers is problematic due to the nature of computers (that won't be analyzed in this paper).

This can be solved with many tricks that some programming languages are dependent on. The chosen language that the project was based on was JAVA, due to being able to create a structure that holds the generator and a future simulation model with ease.

JAVA, due to its nature of easy compatibility between machines, can only do so much with Integers up to 2^31 - we need more than that. The trick to solve this problem was to use a library called BigInteger and BigDecimal. Using these special Objects, calculus complexity is negated. The rest of the project - tests, functions, statistics - was about translating the theoretical requirements into code.

linear-same_remainder method -> Zk = (A* Zk-1 + C)mod m.
With A = 1664525 , C = 1013904223 , m = 2^32, Random_number = Zk/m

Methodology

So, the production of the code had the following steps.

With the help of BigInteger, express the linear-same_remainder methodology and take your current Integer.

With the help of BigDecimal, perform the division of BigIntegers to get the random number.

For the quality of the generated numbers save the sum of the numbers and divide - in the end- them with how many have been computed. That will give the median.
median = sum / numbers

With that number calculate the variance assuming deviationMedian = 0.5.
variance = SUM((random_number - 0.5)^2)

That will give the standard deviation in the end.
deviation = SQUARE_ROOT(variance/numbers)

For further quality analysis two tests were tasked to be done - Runns , Areas.

Runns test takes the current random number and sees whether it is above or below the 0.5 mark. It then calculates the distance of consecutive numbers in the same region. If the region changes the distance goes to zero and repeats from that random number. Then it sees how many times we have seen this distance.

Areas test divides the space (0,1) based on accuracy. For the project the accuracy was 0.1 decimals. This means that the space was (0.0-0.1, 0.1-0.2, 0.2-0.3, … , 0.9-1). (0.1 accuracy means 10 regions of a space). After that it counts how many numbers a region has.
To mark a region at a number, multiply by accuracy and take the non-decimal part and increase the counter in a named region.
For example -> 0.95*10 = 9.5 -> 9 = non_decimal -> increase the counter of region 9.

Code

```
//Use of library in JAVA
//The seed, previous = 0
BigInteger current;
BigInteger previous = BigInteger.valueOf(0);

//The rest of variables needed
double deviationMedian = 0.5;
double randomTotal = 0;
double variance = 0;
double randomNumber = 0;
int counter = 0;
boolean top = true;
runs = new int[numberCount+1];
areas = new int[area_division];

for(int i =0; i< numbers_you_want; i++) {

//This is linear-same_remainder
current = a.multiply(previous);
current = current.add(c);
current = current.mod(mod);

//This is current random number
BigDecimal decimal = BigDecimal.valueOf(current.longValue());
decimal = decimal.divide(BigDecimal.valueOf(mod.longValue()));
randomNumber = decimal.doubleValue();
```

```
//Increase total for median
randomTotal += randomNumber;


//Calculate variance
double distance = randomNumber - deviationMedian;
variance += Math.pow(distance, 2);

//Runns Test
if(randomNumber > deviationMedian && !top) {
        top = true;
        runs[counter]++;
        counter =0 ;
}else if(randomNumber < deviationMedian && top) {
        top = false;
        runs[counter]++;
        counter =0 ;
}
counter++;


//Areas Test
double temp = randomNumber*area_division;
int area =(int) temp;
areas[area]++;

//Next iteration
previous = current; }

//At the end

//this is needed to save the last run of Runns since it may not have changed region for
it to be saved.
runs[counter]++;

deviation = Math.sqrt(variance/ numberCount);
median = randomTotal / numberCount;

//from main to initiate the generator with the required numbers

a = 1664525, c = 1013904223, m = 32

generateNumbers(1.000.000,10);
// create 1000000 numbers and areas test accuracy with 10(0.1)
```

In the end some printing functions were created for the output.

Results

Below are the results of the generator for 1.000.000 numbers.

===>**Median** is: 0.5005121144118533
!!!>**Standar deviation** is: 0.28874040292970876
##############RUNNS TEST###############
Distance: 0.->Counter: 1.
Distance: 1.->Counter: 251065.
Distance: 2.->Counter: 125256.
Distance: 3.->Counter: 62481.
Distance: 4.->Counter: 31075.
Distance: 5.->Counter: 15624.
Distance: 6.->Counter: 7711.
Distance: 7.->Counter: 3844.
Distance: 8.->Counter: 2009.
Distance: 9.->Counter: 973.
Distance: 10.->Counter: 491.
Distance: 11.->Counter: 238.
Distance: 12.->Counter: 121.
Distance: 13.->Counter: 59.
Distance: 14.->Counter: 34.
Distance: 15.->Counter: 11.
Distance: 16.->Counter: 7.
Distance: 17.->Counter: 2.
Distance: 23.->Counter: 1.
%%%%%%%%%%%%%AREAS TEST%%%%%%%%%%%%%%%%%%%%
Area: 0.->Counter: 99736.
Area: 1.->Counter: 99850.
Area: 2.->Counter: 99946.
Area: 3.->Counter: 100148.
Area: 4.->Counter: 99428.
Area: 5.->Counter: 99505.
Area: 6.->Counter: 100581.
Area: 7.->Counter: 100243.
Area: 8.->Counter: 100411.
Area: 9.->Counter: 100152.

And some random numbers while they are generated.

Current Z : 1013904223
Current random number: 0.23606797284446657. AREA[2]
Current distance: -0.26393202715553343
Current variance: 0.06966011495842923
@@
Current Z : 1196435762
Current random number: 0.278566908556968. AREA[2]
Current distance: -0.22143309144303203
Current variance: 0.11869272894444742
@@
Current Z : 3519870697
Current random number: 0.8195337599609047. AREA[8]
Current distance: 0.3195337599609047
Current variance: 0.2207945526992005
@@ .
.
.
Current Z : 492536926
Current random number: 0.11467768950387836. AREA[1]
Current distance: -0.38532231049612164
Current variance: 83370.67243135397
@@
Current Z : 503324709
Current random number: 0.11718941596336663. AREA[1]
Current distance: -0.3828105840366334
Current variance: 83370.81897529722
@@
Current Z : 4074525504
Current random number: 0.9486743956804276. AREA[9]
Current distance: 0.44867439568042755
Current variance: 83371.02028401056
@@

Conclusions

Commenting on the above results.
Random numbers have to follow homomorphic distribution. This means that the median is around 0.5 (the median is 0.5005… which is good) with a standard deviation as big as possible (deviation is 0.28…) so that the numbers are not around 0.5 but scattered around the (0,1).

Areas test highlights that since all areas are having many numbers around 100.000.

Runns test highlights that - with so many numbers - it is highly likely that sequential numbers that are on the same region (above/below 0.5) must have small distance. As it can be seen the most probable distance is 1. This means that random numbers most likely change region from current to next. While the biggest distance that was only encountered once was 23 with 1.000.000 numbers.

In the end ,with the little tests that were made, someone can say a generator like this one is of an acceptable quality - for periods of 2^32.

The project demanded a generation of 100 numbers with Runns test for only 10 numbers. Of course the Runns test for 10 numbers will not give any meaningful results…
About the 100 numbers we can see that the generation was not as good with
median = 0.46809412553440777
deviation = 0.2717992019507281

having the quality of the median dropped.

Areas Test.
Area: 0.->Counter: 10.
Area: 1.->Counter: 11.
Area: 2.->Counter: 11.
Area: 3.->Counter: 10.
Area: 4.->Counter: 13.
Area: 5.->Counter: 13.
Area: 6.->Counter: 11.
Area: 7.->Counter: 5.
Area: 8.->Counter: 9.
Area: 9.->Counter: 7.

It seems that it kept the same pattern of each area having the same amount of numbers(this time around 10). But of course its quality dropped with the small sample.