Using RGB Pixel Data to Generate a Stream of True Random Numbers for Encryption

By: Ikey Croog

North Shore Hebrew Academy High School

Introduction:

Random number generators (RNG) have a critical use in the generating of encryption keys. These encryption keys are important in the encryption and protection of data on computers. Most of these RNGs rely on numbers that are generated to be only seemingly random. These are called pseudo random number generators (PRNG) and come with their own challenges. PRNGs are not truly random and depending on the PRNG it can fail many statistical tests if not built correctly [11][12]. PRNGs also have the problem of repetition in the generation of numbers over several iterations and can therefore be predictable. In order to give pseudo-randomness, PRNGs utilize an initial value called a seed. This seed is processed to produce a pseudo-random number and another number put back into the PRNG to produce additional numbers [6]. An example of a simple PRNG is as follows: (As + c) mod X; where X is some number, c is some increment, A is some multiplier, and s is a seed [4][8]. The seed is important in the generation of numbers different from any other. If someone were to find the seed and the initial conditions they would be able to generate the same exact numbers [7]. Most PRNG usually rely on true random numbers (TRNG) to generate the seeds [6]. TRNGs use random processes that generate efficiently random numbers. An example of this would be the generation of random numbers through the random properties of light and quantum mechanics [13]. Most banking and financial institutions rely on this type (or similar types) of PRNG for encryption or use truly random methods to generate random numbers for encryption keys. One such example of a truly random generation of numbers is using lava lamps to generate numbers for encryption [10]

TRNGs come with their own problems. Some of these include: the reliance on existing hardware, the difficulties of implementation, and the speed of TRNG compared to PRNG which can be significantly slower [6]. Because of this TRNG are mainly used for seed generation and not used for generating streams of random numbers. Another problem is even random processes according to statistics will sometimes generate similar seeds which can be problematic [3]. A proposed solution to the speed and hardware restrictions of TRNGs and the lack of randomness of PRNGs is to generate random numbers through individual RGB (red, green, blue) values of pixels in a JPEG image. Although the idea of generating true random numbers through images has existed, it is only a one time generation of a random number either through the hashing of the image or the processing of RGB channels [9]. These methods of generation require a stream of images and therefore requiring existing hardware and take time to generate.

A better method of generation is to combine the RGB values of several individual pixels in an image together instead of using the entire image to generate one random number. The proposed method uses the following equation: (RGB1 * RGB2... + c) mod X; with X being some number, and c being some increment. The data from an individual pixels RGB values in a random image can be averaged or multiplied together to give a random number. The algorithm can then move to a new pixel mod of the first pixels multiplied RGB value. This next pixel's RGB values can also be multiplied together and then multiplied with the original pixels averaged number. This process can be repeated several times and then the mod X of the number can be taken with some increment. To generate another random number, another pair of RGB values are selected and combined. Unlike PRNGs there is no initially seed, instead two or more truly random RGB values are multiplied together to give a random number. Because this method of generation does not rely on a seed, one number outputted from the generator will have little to no correlation to the last number outputted. This method not only gets rid of hardware restrictions but also generates true random numbers with speed and efficiency.

The proposed project is to use this method of generating numbers from the combination and manipulation of pixels to achieve a sequence of numbers that are completely or at least significantly random. Through a python algorithm, one pixel will be selected and combined with several others to produce a random value. Just like a normal RNG another value will be selected and combined to produce another number. This will allow for a stream of generated numbers. Although no seed will be required, an image and various integer inputs will be needed. This is not a problem as most images can be found over the internet or taken manually, and all other numbers can be generated from this image through different ways of processing. Several random images can also be combined to increase the entropy of the algorithm. Images can also be created by the user. Although this would require existing hardware to create images manually, it would be a one time process and a constant stream of images will not be needed. Through this method of generation more chaotic encryption keys can be generated to protect the data on computer systems.

Hypothesis

If random pixels and their RGB values are selected and combined according to an algorithm then a number can be produced that is significantly random enough to be used in the generation of encryption keys.

Research Question

Can the processing of individual pixel of an RGB image lead to a system capable of efficiently generating a string of random numbers without the reliance on initial hardware or a seed?

Engineering Goal/Expected Outcomes

Through the selection and combination of the RGB values of two pixels, a number can be produced with significant entropy to be deemed a chaotic system. The system will be pendulum-esque in that a small change to the initial input will cause a different output. The image will act as the seed providing true random numbers to be processed and outputted by the computer. Due to the largely software aspect of the generation of the numbers, it limits hardware constraints. This leaves speed constraints largely based upon the system of which the numbers are being generated on.

Procedure

I. Mathematical Algorithm:

R, G, B = respective RGB values

n = some pixel position represented linearly

i = some increment

m = max value

x = some mod value to move by

$$([(R_n + 1) * (G_n + 1) * (B_n + 1)] * [(R_{n+1} + 1) * (G_{n+1} + 1) * (B_{n+1} + 1)] + i) \bmod (m + 1)$$

$$n = ((R_{n-1} + 1) * (G_{n-1} + 1) * (B_{n-1} + 1)) \bmod x$$

II. Pseudocode:

- For testing purposes, the following values will be selected at manually although other modifications of the image can be made to produce these numbers randomly:
    - The starting x and y coordinates
    - The max value
    - The increment which can also act as a minimum value
    - The mod value to move by
    - The amount of which to loop by
- The x and y values will be used to select respective RGB values
- These RGB values will be multiplied together
- The multiplied value will be taken to the mod of the previously selected mod value
- This new value will be used to move the x and y values to a new position
- The new x and y values will be used to get the RGB values at that given position

- The RGB values will then be multiplied together
- This new multiplied RGB value will be multiplied together with the previous multiplied RGB value
- The python script will continue to move and multiply RGB values for the given loop value previously selected
- The final value of multiplied RGB values will be added with the increment
- This final value will be taken to the mod of the max value to produce a new number
- The process can be repeated to produce a string of random numbers

III. Code:

The algorithm for this random number generator was written in python. In order to access the image, the python script relies on the PIL python library [2]. Although the library can be used to edit and transform images, the library is utilized specifically for the access of specific pixels in a given image. The initial image used for testing purposes was this one of Tesla CEO Elon Musk [13]:

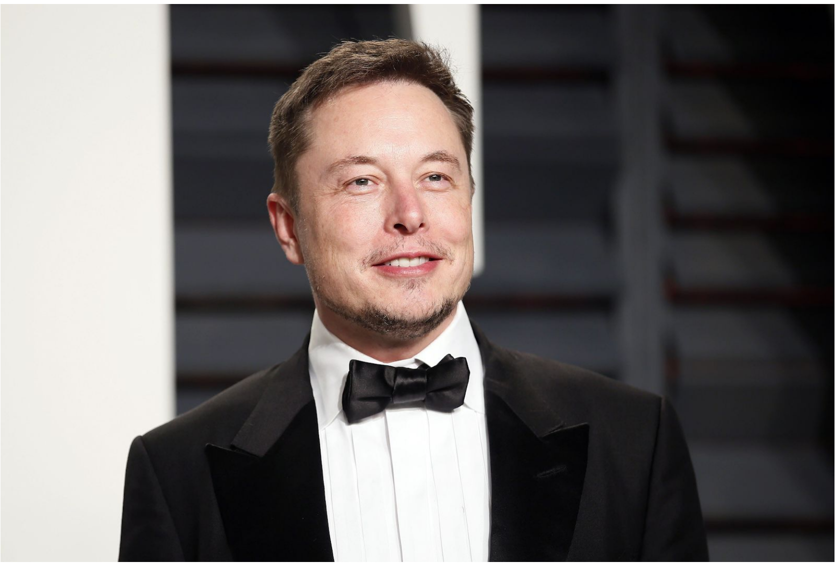
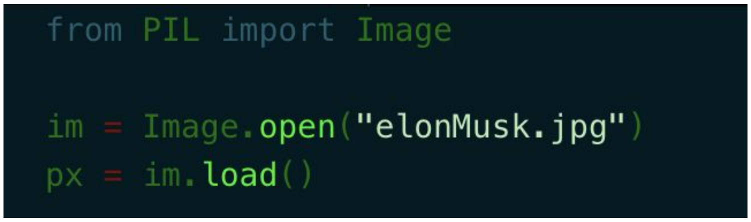


Figure 1: Image of Elon Musk. Image used for testing. [14]



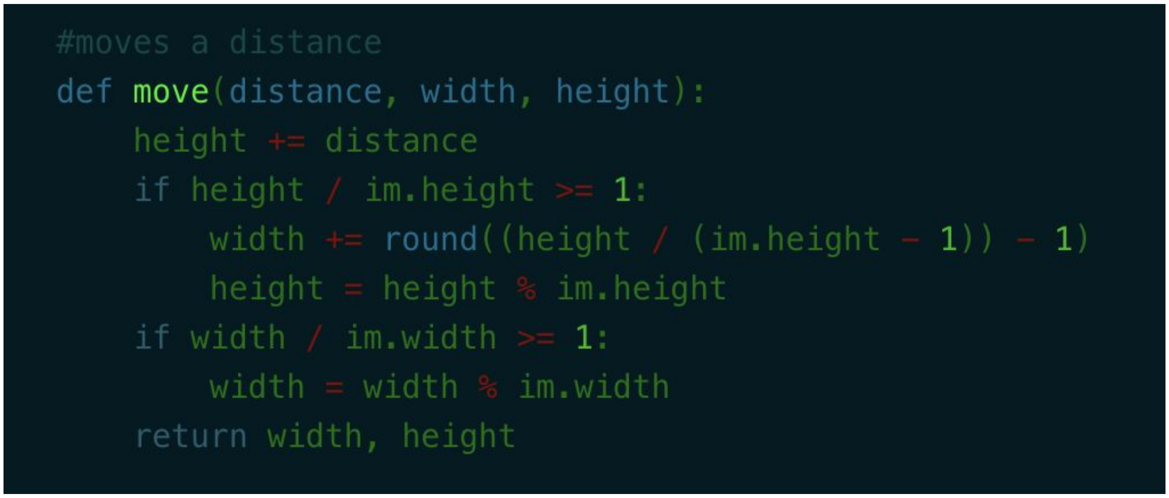
```python
from PIL import Image

im = Image.open("elonMusk.jpg")
px = im.load()
```

Figure 2: Image of PIL import. Illustrates PIL and figure 1 as an import.

The following five functions were defined in order for the RNG to perform properly:

1. “move” function:

The functions main purpose is to transform the x, y grid of the image into a linear system and move accordingly. The function takes in height (corresponding to a given y coordinate), width (corresponding to a given x value), and distance (some distance to move by) as parameters. The function then adds distance to the inputted height value. The function then checks if the height value exceeds the max y value of the image. If that is the case it will set the height value to height mod of the max y value and increase width by the amount of times height exceeds the max y value. The function then returns a new height and width value.

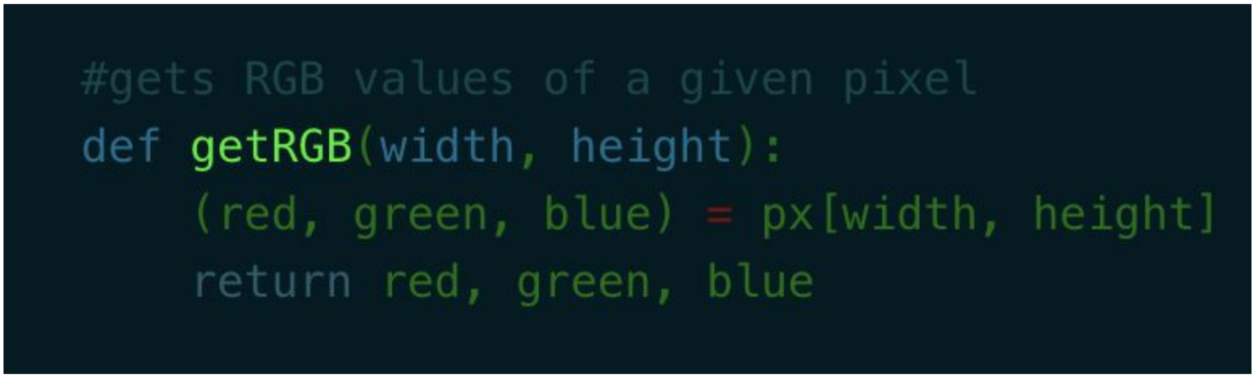


```python
#moves a distance
def move(distance, width, height):
    height += distance
    if height / im.height >= 1:
        width += round((height / (im.height - 1)) - 1)
        height = height % im.height
    if width / im.width >= 1:
        width = width % im.width
    return width, height
```

Figure 3: Image of move function. The code in the move function.

2. “getRGB” function:

This function simply takes in a height and width and using the PIL library returns individual red, green, and blue at the given height and width of the image.

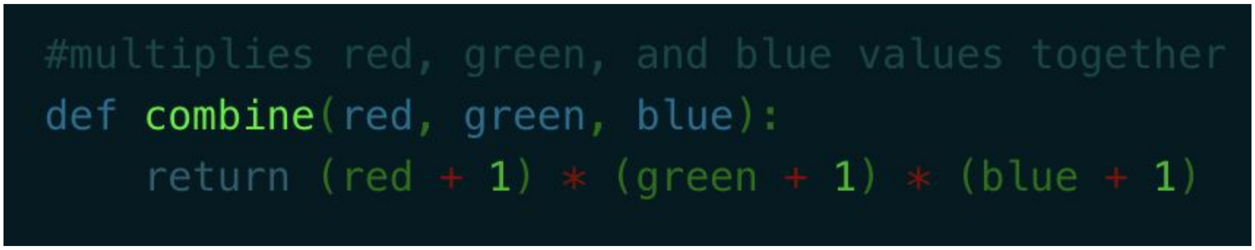


```
#gets RGB values of a given pixel
def getRGB(width, height):
    (red, green, blue) = px[width, height]
    return red, green, blue
```

Figure 4: Image of getRGB function. Uses the PIL library to get RGB values.

3. “combine” function:

This function takes in a red, green, and blue values as parameters. The function then returns the combination of the values according to this equation: (red + 1) * (green + 1) * (blue + 1). The “+ 1” after each value is used to get rid of zero values to avoid a zero output. Despite the “+ 1” the output is still unique.

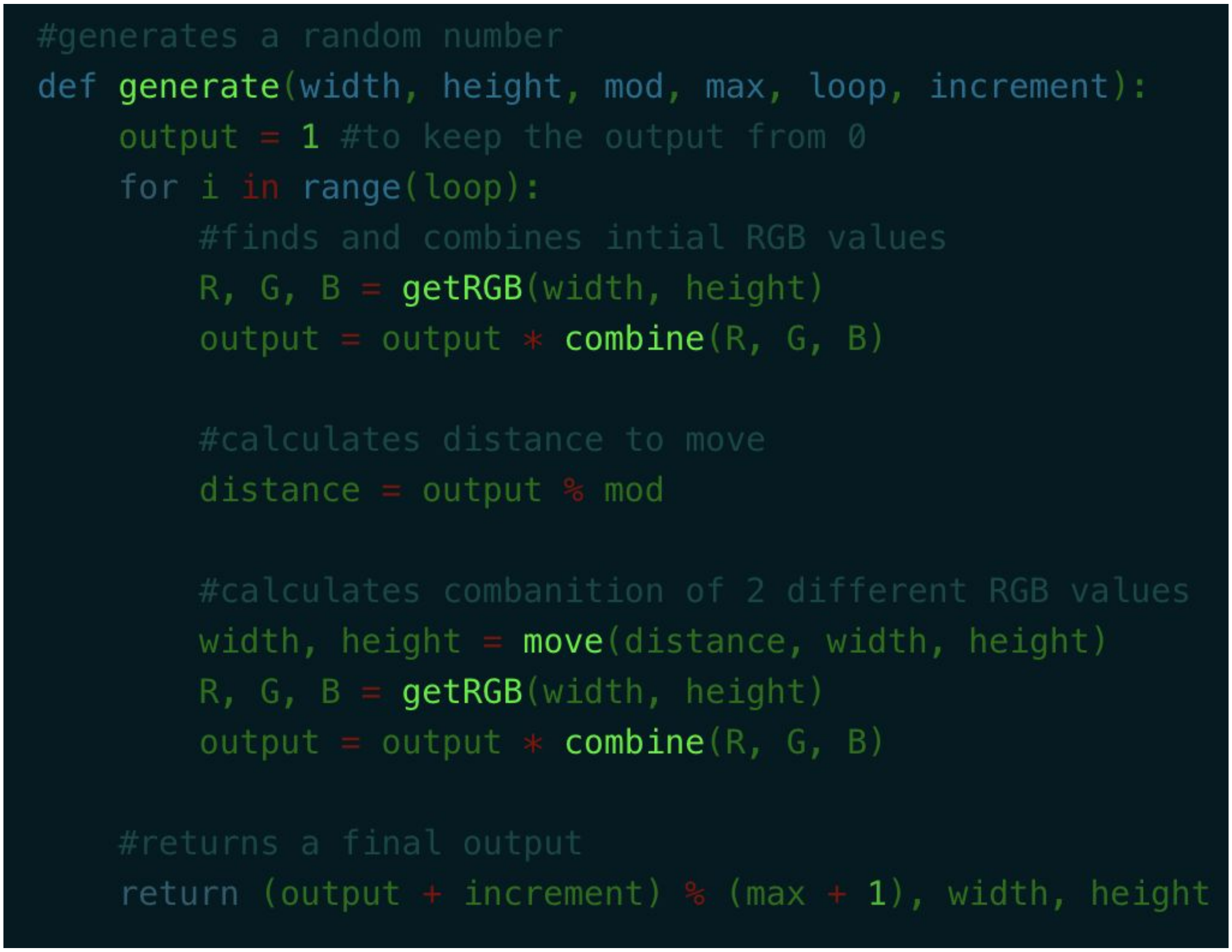


```
#multiplies red, green, and blue values together
def combine(red, green, blue):
    return (red + 1) * (green + 1) * (blue + 1)
```

Figure 5: Image of combine function. Shows combination of RGB values.

4. “generate” function

The actual generation of the random numbers is handled by a function called “generate”. This function takes width, height, mod (the mod value to move by), max (maximum number to generate), loop (amount of times to combine RGB values), and increment as parameters. The function starts by getting the RGB values at the inputted width and height. The function then sets a variable called “output” to the output of the combine function passing through the RGB values mentioned previously. The “output” variable will be initially set to one in order to avoid an output of zero. Another variable, “distance” will be calculated by taking the “output” variable to the mod of the “mod” parameter. The move function will then be called with the distance width, and height variables as its inputs. The output of the move function will be assigned to the width and height parameters. The function then finds and combines the RGB values of the new height and width. This is then multiplied with “output” and the value is set to “output”. The function will then loop for loop plus 1 amount of times and continue to multiply the RGB with “output” to get larger and more random values of “output”. Once the loop is finished, the function adds “output” with the increment parameter and takes it to the mod of max plus 1. The function will return this final number and the final height and width to be used to produce a new number.

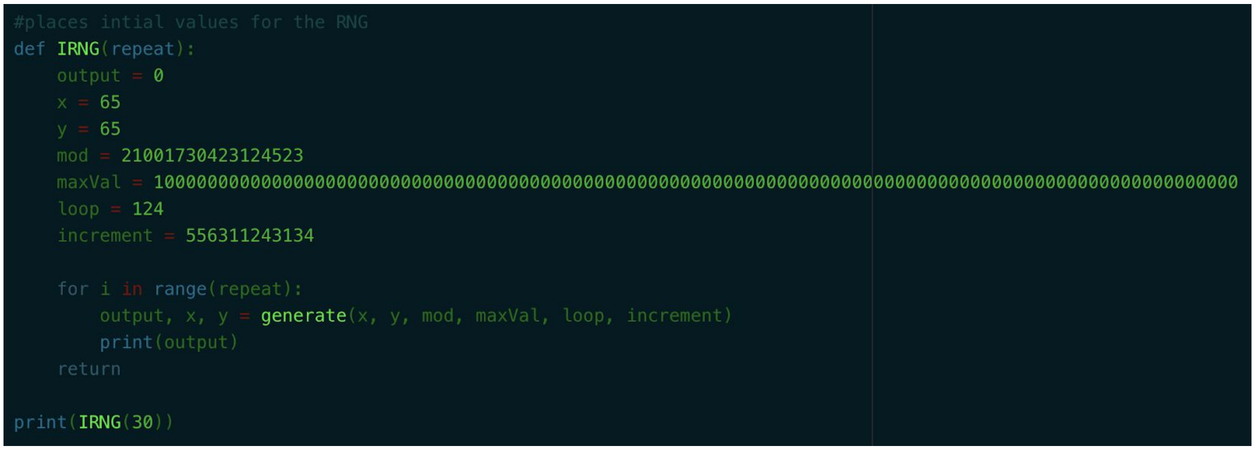


```
#generates a random number
def generate(width, height, mod, max, loop, increment):
    output = 1 #to keep the output from 0
    for i in range(loop):
        #finds and combines intial RGB values
        R, G, B = getRGB(width, height)
        output = output * combine(R, G, B)

        #calculates distance to move
        distance = output % mod

        #calculates combanition of 2 different RGB values
        width, height = move(distance, width, height)
        R, G, B = getRGB(width, height)
        output = output * combine(R, G, B)

    #returns a final output
    return (output + increment) % (max + 1), width, height
```

Figure 6: Image of generate function. The code described above.

5. “IRNG” function

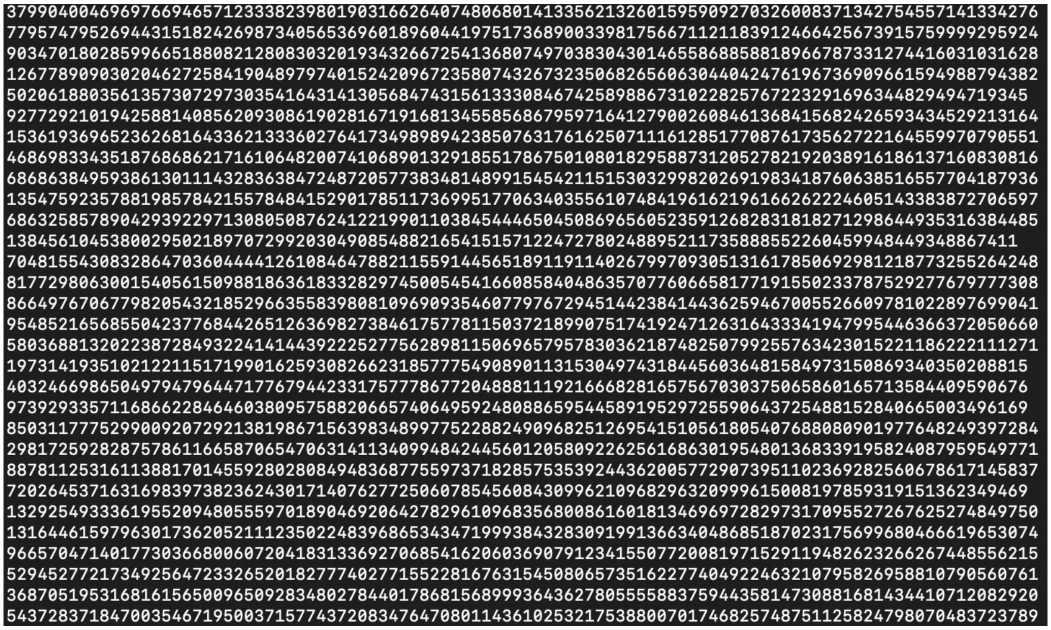
The final function is called “IRNG” which stands for image random number generator. This function takes in one parameter called repeat. The main purpose of this function is to set initial values for the generate function and allow for a stream of numbers to be generated. In this case the inputs are manually selected for testing purposes. The generation of these initial values can be generated randomly in a variety of ways such as: through the hashing and processing of the image, processing of the average color values or RGB channels, processing of existing user input, or any way the user of the algorithm deems acceptable. In the case below x and y are initially set to 65 and correspond to the width and height in the generate function. The mod value is set to “21001730423124523”. The max value is set to 10^100 in order to generate a large number. The loop value is set to 124 making the function loop that many times before giving a final output. The increment is set to “556311243134”. The function simply loops “repeat” number of times and calls the generate function. In the test case the function is set to output 30 random numbers based on the initial conditions.



```
#places intial values for the RNG
def IRNG(repeat):
    output = 0
    x = 65
    y = 65
    mod = 21001730423124523
    maxVal = 10000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
    loop = 124
    increment = 556311243134

    for i in range(repeat):
        output, x, y = generate(x, y, mod, maxVal, loop, increment)
        print(output)
    return

print(IRNG(30))
```

Figure 7: Image of IRNG function. Shows the initial values set for the generate function and the IRNG function call.

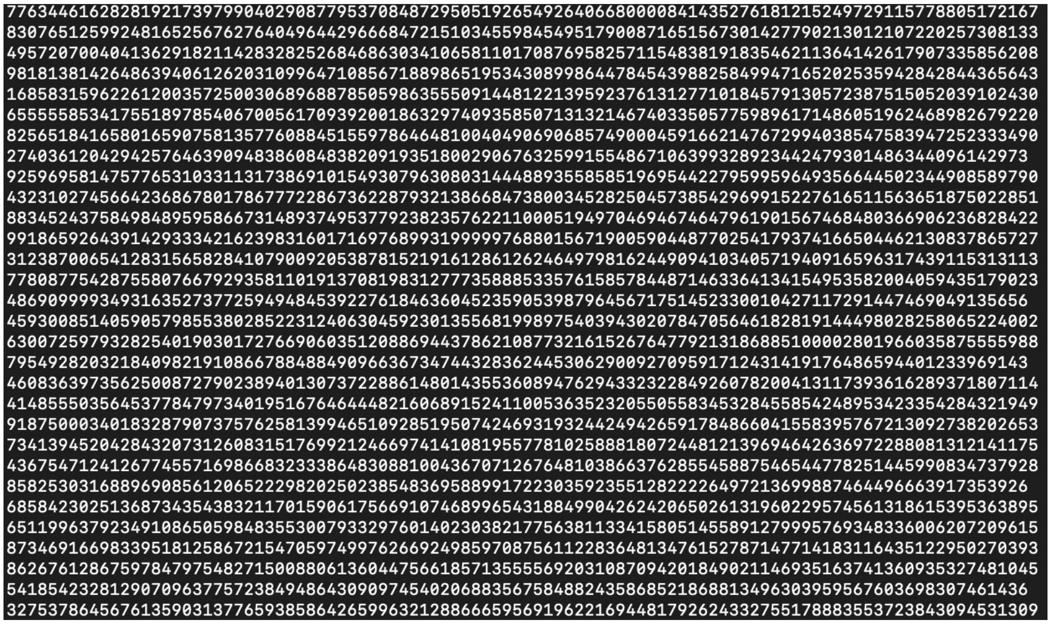
Output based on test case:



```
3799040046969766946571233382398019031662640748068014133562132601595909270326008371342754557141334276
7795747952694431518242698734056536960189604419751736890033981756671121183912466425673915759999295924
9034701802859966518808212808303201934326672541368074970383043014655868858818966787331274416031031628
1267789090302046272584190489797401524209672358074326732350682656063044042476196736909661594988794382
502061880356135730729730354164314130568474315613330846742589886731022825767223291696344829494719345
9277292101942588140856209308619028167191681345585686795971641279002608461368415682426593434529213164
1536193696523626816433621333602764173498989423850763176162507111612851770876173562722164559970790551
4686983343518768686217161064820074106890132918551786750108018295887312052782192038916186137160830816
6868638495938613011143283638472487205773834814899154542115153032998202691983418760638516557704187936
1354759235788198578421557848415290178511736995177063403556107484196162196166262224605143383872706597
6863258578904293922971308050876241221990110384544465045086965605235912682831818271298644935316384485
13845610453800295021897072992030490854882165415157122472780248895211735888552260459948449348867411
7048155430832864703604444126108464788211559144565189119114026799709305131617850692981218773255264248
8177298063001540561509881863618332829745005454166085840486357077606658177191550233787529277679777308
8664976706779820543218529663558398081096909354607797672945144238414436259467005526609781022897699041
9548521656855042377684426512636982738461757781150372189907517419247126316433341947995446366372050660
5803688132022387284932241414439222527756289811506965795783036218748250799255763423015221186222111271
197314193510212211517199016259308266231857775490890113153049743184456036481584973150869340350208815
403246698650497947964471776794423317577786772048881119216668281657567030375065860165713584409590676
973929335711686622846460380957588206657406495924808865954458919529725590643725488152840665003496169
8503117775299009207292138198671563983489977522882490968251269541510561805407688080901977648249397284
2981725928287578611665870654706314113409948424456012058092262561686301954801368339195824087959549771
8878112531611388170145592802808494836877559737182857535392443620057729073951102369282560678617145837
720264537163169839738236243017140762772506078545608430996210968296320999615008197859319151362349469
1329254933361955209480555970189046920642782961096835680086160181346969728297317095527267625274849750
1316446159796301736205211123502248396865343471999384328309199136634048685187023175699680466619653074
9665704714017730366800607204183133692706854162060369079123415507720081971529119482623266267448556215
5294527721734925647233265201827774027715522816763154508065735162277404922463210795826958810790560761
3687051953168161565009650928348027844017868156899936436278055558837594435814730881681434410712082920
5437283718470035467195003715774372083476470801143610253217538800701746825748751125824798070483723789
```

Figure 8: Image of a set 30 of numbers. Numbers generated in the console from the conditions of Figure 7.

To demonstrate the algorithms chaotic nature, the initial x value was changed from 65 to 66:

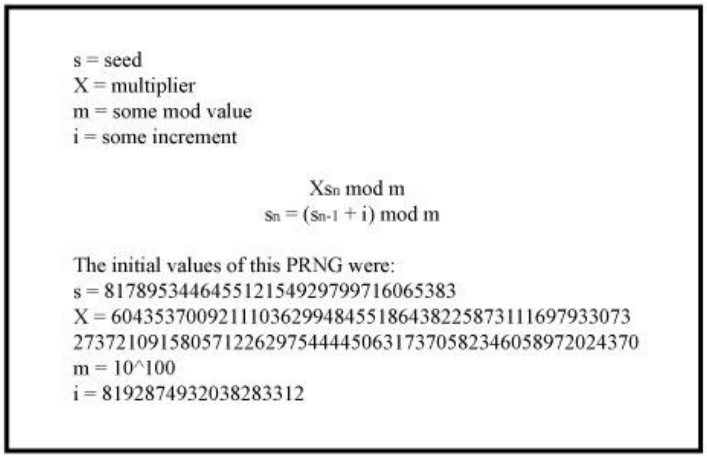


```
7763446162828192173979904029087795370848729505192654926406680000841435276181215249729115778805172167
8307651259924816525676276404964429666847215103455984549517900871651567301427790213012107220257308133
4957207004041362918211428328252684686303410658110170876958257115483819183546211364142617907335856208
9818138142648639406126203109964710856718898651953430899864478454398825849947165202535942842844365643
1685831596226120035725003068968878505986355509144812213959237613127710184579130572387515052039102430
6555558534175518978540670056170939200186329740935850713132146740335057759896171486051962468982679220
8256518416580165907581357760884515597864648100404906906857490004591662147672994038547583947252333490
274036120429425764639094838608483820919351800290676325991554867106399328923442479301486344096142973
9259695814757765310331131738691015493079630803144488935585851969544227959959649356644502344908589790
4323102745664236867801786777228673622879321386684738003452825045738542969915227616511563651875022851
8834524375849848959586673148937495377923823576221100051949704694674647961901567468480366906236828422
9918659264391429333421623983160171697689931999997688015671900590448770254179374166504462130837865727
3123870065412831565828410790092053878152191612861262464979816244909410340571940916596317439115313113
7780877542875580766792935811019137081983127773588853357615857844871463364134154953582004059435179023
486909999349316352737725949484539227618463604523590539879645671751452330010427117291447469049135656
4593008514059057985538028522312406304592301355681998975403943020784705646182819144498028258065224002
6300725979328254019030172766906035120886944378621087732161526764779213186885100002801966035875555988
795492820321840982191086678848849096636734744328362445306290092709591712431419176486594401233969143
4608363973562500872790238940130737228861480143553608947629433232284926078200413117393616289371807114
4148555035645377847973401951676464448216068915241100536352320550558345328455854248953423354284321949
9187500034018328790737576258139946510928519507424693193244249426591784866041558395767213092738202653
7341394520428432073126083151769921246697414108195577810258881807244812139694642636972288081312141175
4367547124126774557169866832333864830881004367071267648103866376285545887546544778251445990834737928
858253031688969085612065222982025023854836958899172230359235512822226497213699887464496663917353926
6858423025136873435438321170159061756691074689965431884990426242065026131960229574561318615395363895
6511996379234910865059848355300793329760140230382177563811334158051455891279995769348336006207209615
8734691669833951812586721547059749976266924985970875611228364813476152787147714183116435122950270393
8626761286759784797548271500880613604475661857135555692031087094201849021146935163741360935327481045
541854232812907096377572384948643090974540206883567584882435868521868813496303959567603698307461436
3275378645676135903137765938586426599632128866659569196221694481792624332755178883553723843094531309
```

Figure 9: Image of 30 different numbers. Numbers generated with the same initial conditions as figure 7 but x was changed to 66.

Data Analysis

Although randomness can never fully be tested, some tests can be done on a set of numbers to begin to determine its “randomness”. The test used for the analysis of these sets was one by the National Institute of Standards and Technology (NIST) [4]. The specific tool for analysis of the was a Github program that runs on python and performs NIST randomness test on sets of integer [1]. For analysis, three different methods of number generation were used: the IRNG, a simple PRNG, a TRNG from Random.org [5]. The simple PRNG written in python uses the following equation to generate pseudo random numbers:



s = seed
X = multiplier
m = some mod value
i = some increment

$$Xs_n \text{ mod } m$$
$$s_n = (s_{n-1} + i) \text{ mod } m$$

The initial values of this PRNG were:
s = 8178953446455121549297997160653838
X = 6043537009211103629948455186438225873111697933073
27372109158057122629754444506317370582346058972024370
m = 10^100
i = 8192874932038283312

Figure 10: Image of PRNG with initial conditions. The math behind the simple PRNG used for testing.

The initial values for the image based random number generator were the ones used in the example before were x was 65. The Random.org generator is based on atmospheric noise and does not include a PRNG [5]. The set of numbers from Random.org was made by stringing together smaller sets of true random number in order to produce a number large enough to be tested. All three sets of numbers used 10 integers with a max value of 100 digits. The following NIST test were used on all 3 sets and showed different results: Binary Matrix Rank Test, Non-overlapping Template Matching Test, Overlapping Template Matching Test, Frequency Test within a Block, Discrete Fourier Transform

(Spectral) Test, Linear Complexity Test, Random Excursions Test, Run Test, Approximate Entropy Test.

The following results were for the image based random number generator:

- Binary Matrix Rank Test: Random
- Non-overlapping Template Matching Test: Random
- Overlapping Template Matching Test: Random
- Frequency Test within a Block: Random
- Discrete Fourier Transform (Spectral) Test: Non-random
- Linear Complexity Test: Random
- Random Excursions Test: Random
- Run Test: Non-random
- Approximate Entropy Test: Non-random

The following results were for the PRNG:

- Binary Matrix Rank Test: Non-random
- Non-overlapping Template Matching Test: Random
- Overlapping Template Matching Test: Random
- Frequency Test within a Block: Non-random
- Discrete Fourier Transform (Spectral) Test: Non-random
- Linear Complexity Test: Random
- Random Excursions Test: Random
- Run Test: Non-random
- Approximate Entropy Test: Non-random

The following were the results of the TRNG:

- Binary Matrix Rank Test: Non-random
- Non-overlapping Template Matching Test: Random

- Overlapping Template Matching Test: Non-Random
- Frequency Test within a Block: Non-random
- Discrete Fourier Transform (Spectral) Test: Random
- Linear Complexity Test: Non-random
- Random Excursions Test: Random
- Run Test: Random
- Approximate Entropy Test: Random

Conclusion

Both the TRNG and the image based random number generator got a “random” result on 6 out of 9 of the above tests. This is in comparison with the PRNG which got a “random” result on 4 out of the 9 tests above. Although the TRNG got a “random” result the same amount of times as the image based generator, it got a “random” on different tests. Both the TRNG and the image based generator passed the Random Excursion Test and the Non-overlapping Template Matching Test. The TRNG passed the Run Test, the Discrete Fourier Transform Test, and the Approximate Entropy Test while the image based generator failed. The image based generator passed the Binary Matrix Rank Test, the Overlapping Template Matching Test, Frequency Test within a Block, and Linear Complexity Test while the TRNG failed. The image based RNG also shared similar results with the PRNG with the Discrete Fourier Transform (Spectral) Test, the Run Test, and Approximate Entropy Test.

Through the NIST test results, it can be concluded that the image based RNG is significantly random enough to be used to generate encryption keys. Because some of the tests are shared with the TRNG and it passes more tests than the PRNG, the image RNG is more similar to the TRNG than the PRNG. Although it fails some of the test that the TRNG passed, the image RNG still passes other tests the TRNG did not. Because the processing of the image in the image based RNG is significantly reliant on software, the speed of generation is mostly client side and depends on the user's processor speed. This is unlike the TRNG which is server side, relying on existing hardware on the Random.org servers. This means that the TRNG does not vary with the users processor speed and therefore generation speeds are fixed. The image base RNG also does not rely on a seed like the PRNG. Although the image based generator does rely on existing inputs, the image itself can be processed to get these numbers and no outside TRNG is needed.

Bibliography

Ang, Steven Kho. “Stevenang/randomness_testsuite.” *GitHub*, 12 Dec. 2019, github.com/stevenang/randomness_testsuite. [1]

Clark, Alex. “Pillow (PIL Fork).” *Pillow (PIL Fork)*, 2010, pillow.readthedocs.io/en/stable/#. [2]

Cook, John D. “Random Number Generator Seed Mistakes.” *John D Cook*, 29 Jan. 2016, www.johndcook.com/blog/2016/01/29/random-number-generator-seed-mistakes/. [3]

Downham, D. Y. “Multiplicative Congruential Pseudo-Random Number Generators.” *The Computer Journal*, vol. 10, no. 1, 1967, pp. 74–77., doi:10.1093/comjnl/10.1.74. [4]

Haahr, Mads. “True Random Number Service.” *RANDOM.ORG - Integer Set Generator*, 1998, www.random.org/integer-sets/. [5]

Jun, Benjamin, and Paul Kocher. “THE INTEL® RANDOM NUMBER GENERATOR.” *Http://www.cryptography.com/*, 22 Apr. 1999, pdfs.semanticscholar.org/10be/4f31b2b5bd7c51dd38b7339543ff46d51a2a.pdf. [6]

Lee, Kyungroul, et al. “TRNG (True Random Number Generator) Method Using Visible Spectrum for Secure Communication on 5G Network.” *IEEE Access*, vol. 6, 2018, pp. 12838–12847., doi:10.1109/access.2018.2799682. [7]

Lewis, P. A. W., et al. “A Pseudo-Random Number Generator for the System/360.” *IBM Systems Journal*, vol. 8, no. 2, 1969, pp. 136–146., doi:10.1147/sj.82.0136. [8]

Li, Rongzhong. “A True Random Number Generator Algorithm from Digital Camera Image Noise for Varying Lighting Conditions.” *SoutheastCon 2015*, 2015, doi:10.1109/secon.2015.7132901. [9]

Limer, Eric. “This Wall of Lava Lamps Is Crucial for Encryption.” *Popular Mechanics*, Popular Mechanics, 14 Nov. 2017, www.popularmechanics.com/technology/security/news/a28921/lava-lamp-security-cloudflare/. [10]

O'Neill, M.E. “Specific Problems with Other RNGs.” *PCG, A Better Random Number Generator*, 31 Aug. 2014, www.pcg-random.org/other-rngs.html. [11]

Rukhin, Andrew, et al. “A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications.” 2000, doi:10.6028/nist.sp.800-22. [12]

Stefanov, André, et al. “Optical Quantum Random Number Generator.” *Journal of Modern Optics*, vol. 47, no. 4, 2000, pp. 595–598., doi:10.1080/09500340008233380. [13]

Yuan, Li. “Why China Can't Get Enough of Elon Musk.” *The Wall Street Journal*, Dow Jones & Company, 1 Apr. 2017, www.wsj.com/articles/elon-musk-inspires-a-china-in-need-of-original-ideas-1490871589 [14]