Aluno: Daniel de Souza Baulé (16200639)

Disciplina: INE5429 - Segurança em Computação

Trabalho Individual 2 - Número Primos

Este trabalho tem como objetivo a geração de números aleatórios primos de 40, 56, 80, 128, 168, 224, 256, 512, 1024, 2048 e 4096 bits. A geração destes números primos ocorre por meio da geração de números aleatórios utilizando os seguintes algoritmos:

- Linear Congruential Generator
- Xorshift

Uma vez gerados os números aleatórios, são utilizados os seguntes algoritmos para a execução dos testes de primalidade, para verificar se os números gerados são de fato primos:

- Miller-Rabin
- Teste de Primalidade de Fermat

As implementações dos algoritmos foram realizadas na linguagem de programação Python, devido à familiaridade do aluno com a linguagem, além da facilidade de lidar com números nas ordens de grandeza necessárias para a realização da atividade. A principal desvantagem da implementação em Python é a pior performance quando comparada a implementações similares em C ou Java, por exemplo, mas como o foco do trabalho é compreender a implementação destes algoritmos e não a posterior utilização destas implementações, a performance não foi considerada um fator crucial.

Números Aleatórios

O ponto inicial do trabalho foi a implementação de 2 algoritmos de geração de números aleatórios, sendo escolhidos:

- Linear Congruential Generator
- Xorshift

Linear Congruential Generator

O primeiro algoritmo para geração de números aleatórios foi o `Linear Congruential Generator`. Pesquisando sobre o algoritmo foi possivel obter uma implementação básica em Python, sendo realizadas adaptações para garantir a geração de números com o número de bits requisitados.

A implementação foi como mostrado abaixo:

```
import time
'''
Linear Congruential Generation
From Wikipedia:
    def lcg(modulus, a, c, seed):
        while True:
            seed = (a * seed + c) % modulus
            yield seed
'''
class LinearCongruentialGenerator:
    def __init__(self, seed=None, numMinBits = 32, m=None, a=1664525, c=1013904223, ):
        self.m = m if m is not None else 2**numMinBits
        self.a = a
        self.c = c
        self.numMinBits = numMinBits;
        # If no seed is provided, use time
        self.state = seed % self.m if seed is not None else int(time.time())
    # Change bit count
    def setMinBits(self, numMinBits):
        self.numMinBits = numMinBits
        self.m = 2 ** numMinBits
    # Generate next random number
    def next(self):
        # Generate next number
        self.state = (self.a * self.state + self.c) % self.m
        # While number doesn't meet bit count requiremente, keep generating
        while self.state < (2**(self.numMinBits - 1)):
            self.state = (self.a * self.state + self.c) % self.m
        # Return random number
        return self.state
```

Analisando a complexidade do algoritmo, desconsiderando as alterações realizadas para obtenção de um número em uma certa faixa, obtemos complexidade linear.

Xorshift

O outro algoritmo implementado para geração de números aleatórios foi o Xorshift, escolhido devido a sua implementação simples, porém extremamente diferente do algoritmo anterior.

No caso do Xorshift, a implementação foi baseada em uma implementação em linguagem C disponível na pagina da Wikipedia. Foi realizada apenas a implementação direta do algoritmo Xorshift 32 bits, com numeros aleatórios com maior contagem de bits sendo gerados por meio da concatenação de números menores.

A implementação foi como mostrado abaixo:

```
'''
Xorshift
From Wikipedia (C implementation):
    struct xorshift32_state {
      uint32_t a;
    };
    /* The state word must be initialized to non-zero */
    uint32_t xorshift32(struct xorshift32_state *state)
    {
        /* Algorithm "xor" from p. 4 of Marsaglia, "Xorshift RNGs" */
        uint32_t x = state->a;
        x ^= x << 13;
        x ^= x >> 17;
        x ^= x << 5;
        return state->a = x;
    }
'''
class XorShiftGenerator:
    def __init__(self, seed=None):
        self.state = (seed & 0xffffffff if seed is
            not None else (int(time.time()) & 0xffffffff ))
    def next(self):
        self.state ^= (self.state << 13) & 0xffffffff
        self.state ^= (self.state >> 17) & 0xffffffff
        self.state ^= (self.state << 5) & 0xffffffff
        self.state &= 0xffffffff
        return self.state
    def getNumber(self, numBits):
        # Make sure number has specified number of bits (Most significant bit 1)
        numBitsFirstNumber = numBits % 32
        if numBitsFirstNumber == 0:
            numBitsFirstNumber = 32
        # Force number to have specified bit count
        next = self.next()
        number = next & (0xffffffff >> (32 - numBitsFirstNumber))
        number |= (0x80000000 >> (32 - numBitsFirstNumber))
        numBits -= numBitsFirstNumber
        # Concatenate remaining bits
        while numBits > 0:
            number = number << 32
            next = self.next()
            number |= next
```

```
            numBits -= 32
        return number
```

Assim como anteriormente, analisando a complexidade do algoritmo, desconsiderando as alterações realizadas para obtenção de um número em uma certa faixa, obtemos complexidade linear.

Resultados

Para a comparação dos algoritmos, foi realizada a implementação de um simples teste, com a configuração destes para gerar números com os diferentes números de bits requisitados no enunciado, além da contagem do tempo para geração de cada número:

```
from RandomNumberGeneration import LinearCongruentialGenerator, XorShiftGenerator
import time
def bitLen(int_type):
    length = 0
    while (int_type):
        int_type >>= 1
        length += 1
    return(length)
numBitsList = [40, 56, 80, 128, 168, 224, 256, 512, 1024, 2048, 4096]
sampleSize = 50;
seed = int(time.time())
lcg = LinearCongruentialGenerator(seed)
xsg = XorShiftGenerator(seed)
print('---------- Linear Congruential Generator ----------')
for numBits in numBitsList:
    lcg.setMinBits(numBits)
    averageTime = 0.0;
    print('\tGenerating ' + str(numBits) + ' bits numbers:')
    for _ in range(sampleSize):
        startTime = time.time()
        next = lcg.next()
        curTime = (time.time() - startTime)
        len = bitLen(next)
        averageTime += curTime/sampleSize
        print('\t\t' + str(len) + ' - ' + str(curTime))
    print('\t\tAverage Time: ' + str(averageTime))
print('---------- Xorshift ----------')
for numBits in numBitsList:
    averageTime = 0.0;
    print('\tGenerating ' + str(numBits) + ' bits numbers:')
    for _ in range(sampleSize):
        startTime = time.time()
        next = xsg.getNumber(numBits)
        curTime = (time.time() - startTime)
        len = bitLen(next)
```

```
            averageTime += curTime/sampleSize
            print('\t\t' + str(len) + ' - ' + str(curTime))
        print('\t\tAverage Time: ' + str(averageTime))
```

É necessário lembrar que a contagem de tempo como implementada pode ser afetada pelo escalonamento de processos do sistema operacional. Para amenizar este erro, o código gera 50 números de cada tamanho, calculando o tempo médio para a geração.

Algorítmo	Tamanho do Número	Tempo para Gerar (s)
Linear Congruente	40	7.410049438476564e-06
	56	7.2622299194335965e-06
	80	6.308555603027342e-06
	128	6.079673767089844e-06
	168	6.198883056640625e-06
	224	6.561279296874999e-06
	256	6.742477416992189e-06
	512	8.258819580078123e-06
	1024	1.255512237548828e-05
	2048	1.5039443969726562e-05
	4096	2.843379974365234e-05
Xorshift	40	7.848739624023435e-06
	56	8.530616760253905e-06
	80	8.740425109863282e-06
	128	9.39369201660156e-06
	168	1.1439323425292973e-05
	224	1.1687278747558594e-05
	256	1.453876495361328e-05

Algorítmo	Tamanho do Número	Tempo para Gerar (s)
	512	2.2592544555664067e-05
	1024	3.9763450622558594e-05
	2048	8.925437927246094e-05
	4096	1.5981674194335936e-04

Nos resultados é possível observar que a variação é mínima, mas como esperado números maiores tendem a demorar mais para gerar. É possível observar também que o Xorshift apresenta performance um pouco pior, provavelmente devido a utilização de concatenação para geração de números maiores que 32 bits.

Números Primos

Após finalizada a implementação dos algoritmos para geração de números aleatórios, foram implementados algoritmos para teste de primalidade de números:

- Miller-Rabin
- Teste de Primalidade de Fermat

O teste de primalidade de Miller-Rabin era de implementação obrigatória, enquanto a implementação do Teste de Primalidade de Fermat foi escolhida devido ao funcionamento diferente do algoritmo, e alta disposição de material na Internet.

Miller-Rabin

O primeiro algoritmo para teste de primalidade implementado foi o de Miller-Rabin. Baseando-se em implementações em pseudo-código disponibilizadas na internet, foi implementado o seguinte código:

```
from random import randint
'''
Miller-Rabin primality test
From wikipedia:
    Input #1: n > 3, an odd integer to be tested for primality
    Input #2: k, the number of rounds of testing to perform
    Output: “composite” if n is found to be composite,
    “probably prime” otherwise
    write n as 2^r·d + 1 with d odd (by factoring out powers of 2 from n − 1)
```

```
    WitnessLoop: repeat k times:
       pick a random integer a in the range [2, n − 2]
       x ← ad mod n
       if x = 1 or x = n − 1 then
          continue WitnessLoop
       repeat r − 1 times:
          x ← x^2 mod n
          if x = n − 1 then
             continue WitnessLoop
       return “composite”
    return “probably prime”
'''
class MillerRabin:
    def decompose(n):
        r = 0
        while n % 2 == 0:
            r += 1
            n = n >> 1
        return r, n
    def testPrime(n, k=200):
        # Known small prime numbers
        if n in (2,3):
            return True
        # No number smaller than 2 is prime
        if n < 2:
            return False
        # No even number other than 2 is prime
        if n % 2 == 0:
            return False
        # Write n as 2^r·d + 1 with d odd (by factoring out powers of 2 from n − 1)
        r, d = MillerRabin.decompose(n - 1)
        # WitnessLoop: repeat k times:
        for _ in range(k):
            # Pick a random integer a in the range [2, n − 2]
            a = randint(2, n - 2)
            # x ← a^d mod n
            x = pow(a, d, n)
            # if x = 1 or x = n − 1 then continue WitnessLoop
            if x in (1, n - 1):
                continue
            try:
                # repeat r − 1 times:
                for _ in range(r - 1):
                    # x ← x^2 mod n
                    x = pow(x, 2, n)
                    # if x = n − 1 then continue WitnessLoop
                    if x == n - 1:
                        raise Exeption()
            except Exception as e:
                continue
            return False
```

```
        return True
```

Comparando resultados com testes de primalidade disponíveis online, foi concluído que a implementação está correta.

Quanto a complexidade do algoritmo, todas as fontes encontradas concordam em O(k log3n), sendo n o número testado e k o número de rodadas.

Teste de Primalidade de Fermat

O segundo algoritmo para teste de primalidade implementado foi o Teste de Primalidade de Fermat. Assim como anteriormente, baseando-se em implementações em pseudo-código disponibilizadas na internet, foi implementado o seguinte código:

```
from random import randint
'''
Fermat primality test
From wikipedia:
    Inputs: n: a value to test for primality, n>3; k: a parameter that
        determines the number of times to test for primality
    Output: composite if n is composite, otherwise probably prime
    Repeat k times:
        Pick a randomly in the range [2, n − 2]
        If a^(n-1) % n != 1, then return composite
        If composite is never returned: return probably prime
'''
class Fermat:
    def testPrime(n, k=200):
        # Known small prime numbers
        if n in (2, 3):
            return True
        # No number smaller than 2 is prime
        if n < 2:
            return False
        # No even number other than 2 is prime
        if n % 2 == 0:
            return False
        # Repeat k times:
        for _ in range(k):
            # Pick a randomly in the range [2, n − 2]
            a = randint(2, n - 2)
            # If a^(n-1) % n != 1, then return composite
            if pow(a, n - 1, n) != 1:
                return False
            # If composite is never returned: return probably prime
        return True
```

Comparando resultados com testes de primalidade disponíveis online, foi concluído que a implementação está correta.

Quanto a complexidade do algoritmo, todas as fontes encontradas concordam em O(k log2n), sendo n o número testado e k o número de rodadas.

Resultados

Primeiramente, foi implementado um código de teste para os algoritmos de teste de primalidade, com números conhecidos primos/não primos, como mostrado abaixo:

```
from PrimeTests import MillerRabin, Fermat
primes = {
    40 : [701886890821,145833959483],
    80 : [627039700318015972526563, 17258016887375724832991],
    128 : [2818236695937908372537526441471694837 21,13014949767276061219450657191161981 6
    168 : [5357723680234491985768045469787937530750457608892 7,8763868424809381827710419
    224 : [9555570476784365604183370437996185501407868060964191196593302762711, 1192904
    256 : [1145217625417365729421758312928588471159780222975336433663365144733982966812
    512 : [1168798893473061508508875279728225037963956089980928675900416417776827783528
    1024 :
    [4295990300262175286569706942624603289030853170013305621487189320909290520964802545
    3857539893494415263925530463526684509054452473252476311593844494140599449192427108 9
}
notPrimes = {
    40 : [701886690821,145834959483],
    80 : [627039700318015972526565, 17258016887375724832992],
    128 : [2818236695937908372537526441471694837 25,13014949767276061219450657191161981 6
    168 : [5357723680234491985768045469787937530750457608892 5,8763868424809381827710419
    224 : [9555570476784365604183370437996185501407868060964191196593302762717, 1192904
    256 : [1145217625417365729421758312928588471159780222975336433663365144733982966812
    512 : [1168798893473061508508875279728225037963956089980928675900416417776827783528
    1024 :
    [4295990300262175286569706942624603289030853170013305621487189320909290520964802545
    3857539893494415263925530463526684509054452473252476311593844494140599449192427108 9

}
numBitsList = list(primes.keys())
numBitsList.sort()
for numBits in numBitsList:
    print('Number of bits: ' + str(numBits))
    print('\tTesting known primes:')
    for prime in primes[numBits]:
        print('\t\tMiller-Rabin: - ' + str(MillerRabin.testPrime(prime)))
    for prime in primes[numBits]:
        print('\t\tFermat: - ' + str(Fermat.testPrime(prime)))
    print('\tTesting known not primes:')
    for notPrime in notPrimes[numBits]:
        print('\t\tMiller-Rabin: - ' + str(MillerRabin.testPrime(notPrime)))
```

```
    for notPrime in notPrimes[numBits]:
        print('\t\tFermat: - ' + str(Fermat.testPrime(notPrime)))
```

Sendo obtidos os seguintes resultados:

```
Number of bits: 40
        Testing known primes:
                Miller-Rabin:  - True
                Miller-Rabin:  - True
                Fermat:  - True
                Fermat:  - True
        Testing known not primes:
                Miller-Rabin:  - False
                Miller-Rabin:  - False
                Fermat:  - False
                Fermat:  - False
Number of bits: 80
        Testing known primes:
                Miller-Rabin:  - True
                Miller-Rabin:  - True
                Fermat:  - True
                Fermat:  - True
        Testing known not primes:
                Miller-Rabin:  - False
                Miller-Rabin:  - False
                Fermat:  - False
                Fermat:  - False
Number of bits: 128
        Testing known primes:
                Miller-Rabin:  - True
                Miller-Rabin:  - True
                Fermat:  - True
                Fermat:  - True
        Testing known not primes:
                Miller-Rabin:  - False
                Miller-Rabin:  - False
                Fermat:  - False
                Fermat:  - False
Number of bits: 168
        Testing known primes:
                Miller-Rabin:  - True
                Miller-Rabin:  - True
                Fermat:  - True
                Fermat:  - True
        Testing known not primes:
                Miller-Rabin:  - False
                Miller-Rabin:  - False
                Fermat:  - False
                Fermat:  - False
```

```
Number of bits: 224
        Testing known primes:
                Miller-Rabin:  - True
                Miller-Rabin:  - True
                Fermat:  - True
                Fermat:  - True
        Testing known not primes:
                Miller-Rabin:  - False
                Miller-Rabin:  - False
                Fermat:  - False
                Fermat:  - False
Number of bits: 256
        Testing known primes:
                Miller-Rabin:  - True
                Miller-Rabin:  - True
                Fermat:  - True
                Fermat:  - True
        Testing known not primes:
                Miller-Rabin:  - False
                Miller-Rabin:  - False
                Fermat:  - False
                Fermat:  - False
Number of bits: 512
        Testing known primes:
                Miller-Rabin:  - True
                Miller-Rabin:  - True
                Fermat:  - True
                Fermat:  - True
        Testing known not primes:
                Miller-Rabin:  - False
                Miller-Rabin:  - False
                Fermat:  - False
                Fermat:  - False
Number of bits: 1024
        Testing known primes:
                Miller-Rabin:  - True
                Miller-Rabin:  - True
                Fermat:  - True
                Fermat:  - True
        Testing known not primes:
                Miller-Rabin:  - False
                Miller-Rabin:  - False
                Fermat:  - False
                Fermat:  - False
```

Uma vez concluído que os testes apresentam funcionamento correto, foi implementada a geração de números aleatórios primos:

```
from RandomNumberGeneration import LinearCongruentialGenerator
from PrimeTests import MillerRabin
```

```
import time
def bitLen(int_type):
    length = 0
    while (int_type):
        int_type >>= 1
        length += 1
    return(length)
numBitsList = [40, 56, 80, 128, 168, 224, 256, 512, 1024, 2048, 4096]
sampleSize = 10;
seed = int(time.time())
lcg = LinearCongruentialGenerator(seed)
for numBits in numBitsList:
    lcg.setMinBits(numBits)
    averageTime = 0.0;
    print('Generating ' + str(numBits) + ' bits numbers:')
    for _ in range(sampleSize):
        startTime = time.time()
        next = lcg.next()
        while not MillerRabin.testPrime(next):
            next = lcg.next()
        len = bitLen(next)
        curTime = (time.time() - startTime)
        averageTime += curTime/sampleSize
        print('\t' + str(len) + ' - ' + str(curTime))
        #print('\t' + str(next))
    print('\tAverage Time: ' + str(averageTime))
```

Devido a melhor performance obtida, foi utilizado o algoritmo de `Linear Congruential Generation` em conjunto com o teste de primalidade de `Miller-Rabin` para geração dos números primos aleatórios mostrados abaixo:

Tamanho do Número	Tempo para Gerar (s)
40	0.0019159555435180665
56	0.0025369644165039065
80	0.00484907627105713
128	0.011374306678771972
168	0.018348765373229978
224	0.029324126243591313
256	0.03960294723510742
512	0.18643634319305422

Tamanho do Número	Tempo para Gerar (s)
1024	1.8157038688659664
2048	29.737854599952698
4096	184.57118501663206

Para se obter estes valores foi realizada a média de tempo a partir da geração de 10 números de cada tamanho. A seguir, também se gerou alguns exemplos de números aleatórios primos:

- 40 bits: 984881580169
- 56 bits: 59110269851068091
- 80 bits: 907671903864918607805683
- 128 bits: 308844230555655163188126921457452195287
- 168 bits: 366155287549482999304639720205531388563920662944771
- 224 bits: 605123814014491763867008858832571136012038705893811433105104470868167198728873
- 256 bits: 605123814014491763867008858832571136012038705893811433105104470868167198728873
- 512 bits: 101433960124322264855145471543253432388560891756408490597551182358884086406939724708796772829309461127328228741030266360740939297523814909363630352394972 19
- 1024 bits: 112114064601907357497793143440333629927372279837757161618227310052599969548139600227879943431334745671042615186892259525004751128497258591306259571194882760337267757244490779871035297587726375829432552987987215174801409487489561214840348812970361419952146176294111948545279674983119692551923717904052082066849
- 2048 bits: 236244257590910950269667767768716223176716134722528147592776227264998405102934554968119095644556993383759371363031213178644486261151056626671390797213888175846482215042730848587027707525260645134778750106997717913554566488939230542024122481000065935146423364369531355321674056565781767914000280603271662678367733172227192896434634966936748420233210281725490767765127780717114331376525177189167828073976598366914644905752765684

87392570796090995447870044111958228744379717082838049471352828458943859
46361885080666241790990652631866589571052784568133922959254288411709335
35477888130252666673601081007126383354316580488 23

- 4096 bits:
  71476385273390183740495605619325906882493018033607671275651209865841230
  51353970141814493481360764732022783681154941025392658241871525502713574
  36631244188805680657050775084867492526585712595320879921263817272318415
  54984721393022789620152129093140351596138584384934152802785881383099830
  36684191893238068676132444408095238139054425869019978814381940014624943
  05848306204609595632101785099101835599302836857898715944889700146764619
  24075952410832994924257866391807951314217510492579598547828251667258181
  28726589626340043448667930078140647375510045925516096471540291936056293
  47501957158671458599531081952919933991591354045049287514204516242426214
  07701280178484443755142037466155830925741000879004689825298079643262204
  90101227510614393940465245753452243871994732356525687285000030886503095
  35734974683119978089857486192241000635842051807353689875723559489803166
  20693985387097418238395768409133825963353071931244500420993585386082951
  07721450178722449052376411789134768103421814643326301553796663571891536
  57017840349986750858043645966297404049897537209721776107723567709026500
  44683792511162444368284583027857268165223045682056276836510784279950526
  99067346883796423432127698877208740885850276049303462091718818414659581
  09077609275069337177073943

Dificuldades Encontradas

Dentre as principais dificuldades encontradas estão o tempo para geração de números aleatórios primos grandes, o que dificultou os testes, além de a comparação de resultados com testes de primalidade online, visto que a maioria não suporta números nesta ordem de grandeza.

Outra dificuldade foi a geração de números aleatórios grandes com algoritmos limitados, como Xorshift, sendo necessária a concatenação de resultados.

Referências

Miller-Rabin

https://en.wikipedia.org/wiki/Miller%E2%80%93Rabin_primality_test

https://crypto.stanford.edu/pbc/notes/numbertheory/millerrabin.html

Fermat

https://en.wikipedia.org/wiki/Fermat_primality_test

https://www.geeksforgeeks.org/primality-test-set-2-fermet-method/

Linear Congruential Generator

https://en.wikipedia.org/wiki/Linear_congruential_generator

https://rosettacode.org/wiki/Linear_congruential_generator

Xorshift

https://en.wikipedia.org/wiki/Xorshift

https://www.javamex.com/tutorials/random_numbers/xorshift.shtml