Trabalho sobre Números Primos

Matheus Novais

Outubro 2024

1 Introdução

Neste relatório, será abordado o processo de geração de números pseudo-aleatórios e a verificação de primalidade, seguindo as diretrizes do trabalho individual. Para a primeira etapa foram implementados dois algoritmos de geração de números pseudo-aleatórios, sendo eles Linear congruential generator e Park Miller, analisando sua eficiência na geração de grandes números de até 4096 bits. Em seguida, esses mesmos algoritmos foram utilizados para gerar valores que foram testados pelo Miller-Rabin, juntamente com o de teste de Fermat, para avaliar a primalidade desses números. Durante a execução, foi verificado o tempo necessário para cada operação. O relatório inclui as dificuldades encontradas, além dos resultados obtidos, como tabelas e códigos comentados.

Os códigos e comentários com melhor detalhamento da implementação podem também ser vistos no github, dentro do seguinte repositório:
https://github.com/matheusnovx/prime-number-INE5429.

2 Geração de números pseudo-aleatórios

Para a geração de números pseudo-aleatórios foram escolhidos os algoritmos Linear congruential generator(LCG) e Park Miller random number generator.

2.1 Linear congruential generator(LCG)

2.1.1 Explicação

O **Linear Congruential Generator (LCG)** é um dos métodos mais antigos, tendo sido publicado em 1958[6] [5], e populares para gerar números pseudoaleatórios. Ele se baseia na seguinte fórmula de recorrência:

$$X_{n+1} = (a \cdot X_n + c) \mod m$$

onde:

- X_n é o n-ésimo número aleatório gerado.
- X_{n+1} é o próximo número aleatório.
- a é o multiplicador (um número inteiro).
- c é a constante aditiva (também um número inteiro).
- m é o módulo (um número inteiro positivo).
- X_0 é o valor inicial ou *seed*.

Os parâmetros usados durante a implementação vieram da sugestão dada conformePress [4].

2.1.2 Implementação

```python
from time import time_ns

class LinearCongruentialGenerator:
    def __init__(self, m: int = 2 ** 32, a: int = 1664525, c: int = 1013904223):
        self.m = m  # Modulo
        self.a = a
        self.c = c

    def lcg(self, n_bits, seed=None):
        # Mascara para limitar os bits gerados
        mask = (1 << n_bits) - 1
        m, a, c = 2 ** 32, self.a, self.c

        # Define o seed com base no tempo atual se nao for fornecido
        if seed is None:
            seed = time_ns() % m

        number_generated = (a * seed + c) % m
        final_result = 0
        bits_generated = 0

        # Gera numeros ate alcancar a quantidade desejada de bits
        while bits_generated < n_bits:
            number_generated = (a * number_generated + c) % m
            new_generated = number_generated & ((1 << 32) - 1)

            final_result = (final_result << 32) | new_generated
            bits_generated += 32

        return final_result & mask
```

2.1.3 Resultados

Length (bits)	Tempo de execução (segundos)
40	0.00000122
56	0.00000118
80	0.00000150
128	0.00000175
168	0.00000236
224	0.00000278
256	0.00000693
512	0.00001299
1024	0.00001909
2048	0.00003782
4096	0.00007714

2.2 Park Miller

2.2.1 Explicação

O **Park-Miller Generator**, também conhecido como Lehmer random number generator, é um tipo específico de **Linear Congruential Generator (LCG)** utilizado para gerar números pseudoaleatórios, tendo sido definido em 1969[3]. Ele é definido pela seguinte fórmula de recorrência:

$$X_{n+1} = (a \cdot X_n) \mod m$$

onde os parâmetros são:

- X_n é o n-ésimo número aleatório gerado.
- X_{n+1} é o próximo número aleatório.
- a é o multiplicador, que é um número primo.
- $m = 2^{31} - 1$ é o módulo, normalmente sendo um número primo de Mersenne[7].
- X_0 é o valor inicial ou *seed*, que deve ser um inteiro positivo entre 1 e $m - 1$.

Por mais que seja normalmente visto como um caso particular do LCG, mas com c = 0, ele tem suas restrições e propriedades únicas, principalmente a necessidade de ter um *seed* coprimo de m. Tendo também uma maior restrição na escolha do multiplicador(a) e do módulo(m).

2.2.2 Implementação

```python
from time import time_ns

class ParkMiller:
    def __init__(self, m: int = (2 ** 32) - 1, a: int = 16807):
        self.a = a
        self.m = m  # Modulo

    def pm(self, n_bits, seed=None):
        # Mascara para limitar os bits gerados
        mask = (1 << n_bits) - 1
        m, a = self.m, self.a

        # Define o seed com base no tempo atual se nao for fornecido
        if seed is None:
            seed = time_ns() % m

        number_generated = (a * seed) % m
        result = []
        bits_collected = 0

        # Gera numeros ate alcancar a quantidade desejada de bits
        while bits_collected < n_bits:
            number_generated = (a * number_generated) % m
            new_generated = number_generated & ((1 << 32) - 1)

            result.append(new_generated)
            bits_collected += 32

        # Combina os numeros gerados em um unico resultado
        final_result = 0
        for part in result:
            final_result = (final_result << 32) | part

        return final_result & mask
```

2.2.3 Resultados

Length (bits)	Tempo de execução (segundos)
40	0.00000131
56	0.00000134
80	0.00000347
128	0.00000470
168	0.00000647
224	0.00000701
256	0.00000480
512	0.00000605
1024	0.00001447
2048	0.00003847
4096	0.00008062

2.3 Comparação

Para ser feito essa comparação entre esses dois algoritmos, foi utilizado da implementação para criar uma tabela com o tempo de execução e com o tamanho de número gerado. O tempo de execução, para não ser injusto, foi considerado a partir de uma média de tempo das 1 milhão de vezes que foram gerados números. Podemos então ver pela analise da tabela que o LCG teve um desempenho superior em quase todos os tamanhos de números criados, tendo apenas uma exceção no de 256 bits. A complexidade de ambos é de O(n) pela necessidade de iterar até que a quantidade de bits desejados seja alcançada, provando ainda mais a proximidade de desempenho entre os dois.

3 Verificação de primalidade

3.1 Miller Rabin

3.1.1 Explicação

O **Teste de Primalidade de Miller-Rabin**, definido em 1976 por Gary L. Miller [2], é um algoritmo probabilístico utilizado para determinar se um número inteiro n é primo. O teste é baseado na teoria dos números e na propriedade de que, se n é primo, então $a^{n-1} \equiv 1 \mod n$ para qualquer inteiro a tal que $1 < a < n-1$.

O algoritmo funciona da seguinte maneira:

1. **Fatoração**: Para um número ímpar n, escrevemos $n-1$ como $d \cdot 2^r$, onde d é ímpar e r é um inteiro não negativo. Isso é feito encontrando o maior r tal que 2^r divide $n-1$.

$$n - 1 = d \cdot 2^r$$

2. **Escolha de** a: Escolha um inteiro aleatório a tal que $1 < a < n-1$.

3. **Teste de Primalidade**: - Calcule $x = a^d \mod n$. - Se $x \equiv 1 \mod n$ ou $x \equiv n-1 \mod n$, o teste continua com uma nova escolha de a. - Caso contrário, repita os seguintes passos $r-1$ vezes: - Calcule $x = x^2 \mod n$. - Se $x \equiv n-1 \mod n$, continue com a próxima escolha de a. - Se $x \equiv 1 \mod n$, então n não é primo.

4. **Resultado**: Se nenhuma das condições acima for satisfeita para $r-1$ iterações, então n é considerado composto. Caso contrário, n pode ser primo.

Este teste é geralmente executado várias vezes com diferentes escolhas de a para aumentar a confiabilidade do resultado. O número de iterações determina a probabilidade de erro do teste. Quanto mais iterações, menor a probabilidade de que um número composto seja erroneamente identificado como primo. Durante o desenvolvimento dessas comparações foi utilizado 5 iterações como o número que identificasse um primo mas que não custasse mais tempo do que o necessário.

3.1.2 Implementação

```python
import random

def is_prime(n, quantidade_iteracoes):
    # Trata os numeros de 1 a 4 e verifica se sao pares
    if n == 2 or n == 3:
        return True
    if n == 1 or n % 2 == 0:
        return False

    # m = (n - 1) / 2^k
    m = n - 1
    # Divide por 2 ate chegar em um numero impar
    while m % 2 == 0:
        m //= 2

    # "a" e um numero aleatorio gerado e 1 < a < n-1
    a = random.randint(2, n - 1)

    # b = a^m mod n
    b = pow(a, m, n)

    # b equivale 1 (mod n)
    if b % n == 1:
        return True

    # Realiza esse teste quantidade_iteracoes vezes
    for _ in range(quantidade_iteracoes):
        # b equivale -1 (mod n)
        if b % n == n - 1:
            return True
        else:
            # b = b^2 mod n
            b = pow(b, 2, n)

    return False
```

3.1.3 Resultados

Length (bits)	Tempo de execução (segundos)
40	0.00026534
56	0.00022435
80	0.00088720
128	0.00205767
168	0.00268602
224	0.01293516
256	0.01094379
512	0.09748211
1024	3.37255161
2048	32.88018889
4096	675.27115960

- 175395211111
- 29009937820800089
- 6605669929738572510240 59
- 2850326279670241261597734842564838 89807
- 853503296052442411672918355601861812934125234782 93
- 211866724692363649152260418906167148778284719406180330 94
896791142851
- 1100115345522493707577937677542916213646180285756974236 97
58632437100566134 7731
- 29197306649465045977336765786553959828230683932333342677
44681933241017393763632472628031602405872086229227220034
045504717635352271230374647547415967052 19
- 13791335073619356197860466361065561959916939707057042842
76659399584351366475607844951155094942926509237890787044
93709494748294151103220281278080887216577436468235787611
90709323187470508716147295874252299743323483861716085241

896301235659444635548163694987425131476415141362547547431421752626385088068361714045 1

- 19556801993149096375867621475846935367733621992928217614797269655366608552344897792613090457878598109586447626759511664852843729946555429026732346006392656368249373711310554809150133426203017375535856169894356855922457718170544188688797807233722138314258984620341146671684260288401878951677085363426569033233649798178215496484474404889541353195399375174063165834767033266609775074600527130564091807816428783673574645358253535522803992764248753893677960779052252263058313787938676461578183038453118449303856804057038937941432761111771872471643498740848359668668324369409343500220986381088657925116495475120279735931199

- 99866431907576507809551993249877552464138753074264885320701695833096696014879948731235046207948337990450589325977093049378997768546079017961782927023512656777424272536283147372868054169571309503185734350037163807848732246175078783956578248232231700965464491400353279191639981349069711927077052679605915755226254640894551499300285891237785923734338274104929390407715177989217684220622123266999416079215063532614489158601000835477047579540699687874243760107200041582694177787997105028801137273418294522305138860977571024259672660485920859633364436041819040946860254252778571779955923858540685387371140620689342429218760171273002315453835171278549436238866751220137335687376874875262486476315414132413926952511433681715859398070792042298837417176776635335927349853942066892364670196255588898045324026220971520212586965874397456175613082132174519121392727482513658137501384703317054228477591055091021915490066154067971337847132005819700866380683500051947334098872205480268625101919993059006236820401551012880167790272957716716744238873400695768219168649011021757228038598301915431733570495950593454459107909802986781244670407020874015554389857934401820479243914249381656034162834552883598572359158645375878735443799309310214633212389339

3.2 Fermat

3.2.1 Explicação

O **Teste de Primalidade de Fermat** é um algoritmo probabilístico utilizado para verificar se um número inteiro n é primo. Baseia-se no pequeno teorema de Fermat, que afirma que se p é um número primo e a é um inteiro tal que $1 < a < p - 1$, então:

$$a^{p-1} \equiv 1 \mod p$$

O algoritmo funciona da seguinte maneira:

1. **Escolha de** a: Escolha um inteiro aleatório a tal que $1 < a < n - 1$.

2. **Cálculo**: Calcule $a^{n-1} \mod n$ utilizando a exponenciação modular.

3. **Verificação**: - Se $a^{n-1} \equiv 1 \mod n$, então n pode ser primo, e o teste continua com uma nova escolha de a. - Se $a^{n-1} \not\equiv 1 \mod n$, então n é composto.

4. **Repetição**: Para aumentar a confiança no resultado, repita o teste com diferentes escolhas de a. O número de iterações determina a probabilidade de erro do teste. Quanto mais escolhas de a você testar, menor será a chance de que um número composto seja erroneamente identificado como primo.

Embora o teste de Fermat seja rápido e fácil de implementar, ele não é completamente confiável, pois existem números compostos, conhecidos como *falsos primos de Fermat, ou, Fermat liar*, que podem passar no teste. Portanto, ele é frequentemente usado em combinação com outros testes de primalidade para garantir resultados mais precisos.

3.2.2 Implementação

```python
import random

def is_prime(n, quantidade_iteracoes):
    # Trata os numeros de 1 a 4 e verifica se e par
    if n == 2 or n == 3:
        return True
    if n == 1 or n % 2 == 0:
        return False

    # m = n - 1
    m = n - 1

    # Quanto maior a quantidadeIteracoes, maior a precisao.
    for _ in range(quantidade_iteracoes):
        # Gera um "a" aleatorio.
        a = random.randint(2, n - 2)

        # Teorema de Fermat: a^(n-1) (equivalente) 1 (mod n)
        # Se essa congruencia nao for verdadeira, n e composto.
        if pow(a, m, n) != 1:
            return False

    # Depois de fazer o teste para quantidadeIteracoes "a"s diferentes, "n" e provavelmente primo.
    return True
```

3.2.3 Resultados

Length (bits)	Tempo de execução (segundos)
40	0.00009999
56	0.00025744
80	0.00074520
128	0.00171838
168	0.00511315
224	0.00711520
256	0.01313107
512	0.12032855
1024	3.49889076
2048	13.26827607
4096	144.65822992

- 1068532403959
- 37567503780507689
- 229510335224864221903411
- 280556882133598050280672357178277080617
- 33581159757896435364136340095029358976651411008769
- 1983214967850407643206005985762195474698363307280400800498
  3630089593
- 7521997492021921620124269143471384552191136731378401872898
  3707402499451936877
- 8479990057479439997742904671676160302860682640910580803086
  0182510294787159423045140484279301405648564691442897046225
  88868077423615335166993471880497514819
- 1327468267944444470000918083709160745041983049933092068629
  9266412815906489510784514513583259907247211699773096306949
  3084598020668320365685631733575205797579514237043604121964
  0029486741918924669463701925825326767166810888998069530920

696559591359721074140442157894947957050252545366030201286885558720442034425849

- 598531316098149758495831519088402056914638992449822266887782311571060800044701995895781736783477975453141461742662747962194877137966281791364361469012681043657222542870816828741050100318033066478048665894534445437156982193059958991064411668209203636339803231900809782831783005710653570235177641878149242296918728453671584529313014881490711887396322376042238054841858287826627872576162668439982282733490956631315671313271125889668788934642261015825952682033325836079931035667805373864601338795436207878445636863245475547708062805721179306705655695194560626022354291102471444823724357531956831846890751781357155352197

- 633984729258250956406442414064260887679544765757447093766679753145701869733061558948690801885790792038395919195937388068643575907740296740170458739903748399180618873370858350564501932519851063118502871721031226241581361762689180343224626380174309016062670672721283080518623237465639164515677251263947346696906835502930074334514617535625836847841710916068274407402417303325505598221807586720474649669088117891966465222032296979017187941327090684757463782352098957043688604017711826853997360773068471464932823716739608504912245629078117772024637291776553839493490693129673926983819116567917610669215970916744562961564863419296638401130218642353497223412891297350769456466784917118848272627348233325640336665170496339050694115473196129241781086755685528039603848163822127641191795790523906094898467368703368853831612952654606001332736926783275609702859856438103321569689084472300765983126073881909009400537783397999180465163526377317697957587989044374468977115313109226192228440393652685185974601297851468365736617435418376546712681696081255334062069629224092085566336955301271290639129555791878237674055024160884620506730381397977641088606879487061702367849025480522978580843050306993816460624637898730125215235397778297578683270161343

3.3 Comparação

Para podermos fazer a comparação dos resultados, foi primeiro desenvolvido um programa de teste que gerava números aleatórios a partir do LCG, visto anteriormente na seção 2.1, com ele foram gerados 100 números primos com tamanhos entre 40 e 1024 bits, conforme a lista passada para o trabalho, e 5 números primos com 2048 e 4096 bits, por questão de tempo e poder computacional. O programa se baseia em um loop que enquanto não for gerado um número primo não é liberado, e assim que for gerado a quantidade necessário é falado o tempo médio entre essas gerações. Os números gerados foram também testados, de forma manual, pelo comando da biblioteca *openssl prime* que verifica se o número é de fato primo. Nessa comparação podemos ver que por mais que não exista uma diferença tão significativa até a geração de números com 1024 bits, mas que após esse tamanho ocorre um discrepância entre o tempo de execução, podendo ser ainda mais compreensível quando visto o de 4096 bits. Para ambos os algoritmos é dado como a complexidade de O(k $\log^2$ n)[8][1], onde k é a quantidade de vezes que testamos um número e n é o valor que queremos verificar a primalidade. Já a complexidade do programa de teste é a O(n * k), sendo k a complexidade dos algoritmos e n a quantidade de números que são precisos antes de gerar um primo.

Referências

[1] Joe Hurd. "Verification of the Miller–Rabin probabilistic primality test". Em: *The Journal of Logic and Algebraic Programming* 56.1 (2003). Probabilistic Techniques for the Design and Analysis of Systems, pp. 3–21. ISSN: 1567-8326. DOI: `https://doi.org/10.1016/S1567-8326(02)00065-6`. URL: `https://www.sciencedirect.com/science/article/pii/S1567832602000656`.

[2] Gary L. Miller. "Riemann's Hypothesis and tests for primality". Em: *Proceedings of the Seventh Annual ACM Symposium on Theory of Computing*. STOC '75. Albuquerque, New Mexico, USA: Association for Computing Machinery, 1975, pp. 234–239. ISBN: 9781450374194. DOI: `10.1145/800116.803773`. URL: `https://doi.org/10.1145/800116.803773`.

[3] W. H. Payne, J. R. Rabung e T. P. Bogyo. “Coding the Lehmer pseudo-random number generator”. Em: *Commun. ACM* 12.2 (fev. de 1969), pp. 85–86. ISSN: 0001-0782. DOI: 10.1145/362848.362860. URL: https://doi.org/10.1145/362848.362860.

[4] William H. Press. *Numerical Recipes in Fortran 77: The Art of Scientific Computing*. Cambridge University Press, 1992.

[5] A. Rotenberg. “A New Pseudo-Random Number Generator”. Em: *Journal of the ACM* (1960).

[6] W. E. Thomson. “A Modified Congruence Method of Generating Pseudo-random Numbers”. Em: *The Computer Journal* (1958).

[7] Wikipédia. *Primo de Mersenne — Wikipédia, a enciclopédia livre*. [Online; accessed 30-julho-2024]. 2024. URL: https://pt.wikipedia.org/w/index.php?title=Primo_de_Mersenne&oldid=68357525.

[8] Wikipedia contributors. *Fermat primality test — Wikipedia, The Free Encyclopedia*. [Online; accessed 15-October-2024]. 2024. URL: https://en.wikipedia.org/w/index.php?title=Fermat_primality_test&oldid=1227031378.