Números Primos e Geradores de Números Pseudo Aleatórios:

Relatório de Implementação

Alek Frohlich

Departamento de Informática e Estatística
Universidade Federal de Santa Catarina
Brasil
14 de Julho de 2021

1 Introdução

Neste trabalho, implementamos dois geradores de números pseudo-aleatórios - LCG e BBS - e comparamos seus resultados quanto ao tempo médio de geração de números de dada magnitude. Fazemos também uma comparação qualitativa dos PRNGs implementados e discutimos a complexidade algorítmica de suas respectivas implementações. Além disso, implementamos dois testes de primalidade probabilísticos: Miller-Rabin e Fermat. A análise dos testes de primalidade é feita de forma análoga à dos PRNGs. Por fim, utilizamos combinações de PRNGs com testes de primalidade para gerar números primos pseudo-aleatórios de tamanhos variados.

O relatório está estruturado em duas seções: números aleatórios e números primos. Em cada uma dessas seções, discutimos os resultados obtidos por cada método implementado e fazemos comparações. O código fonte pode ser encontrado no seguinte repositório.

2 Números Aleatórios

Os geradores usados foram: Linear Congruential Generator (LCG) e Blum-Blum Shub (BBS), as referências utilizadas estão presentes na documentação do código fonte (arquivo "prng.py"). Os parâmetros utilizados e as motivações por trás de suas escolhas estão justificados no arquivo "driver.py".

2.1 Resultados

A Tabela 1 a seguir resume o tempo tomado por cada um dos dois algoritmos para gerar números de diversas magnitudes. O tamanho das amostras variam entre algoritmos (BBS é mais lento que LCG): para o LCG, utilizamos 1000 amostras; e, para o BBS, utilizamos 10 amostras. Vale ressaltar que as seeds estão fixas. Assim, basta descomentar as funções "gen_prnumbers_LCG"e "gen_prnumbers_BBS"para reproduzir os mesmos resultados (o tempo irá variar um pouco de máquina para máquina).

2.2 Comparação

Os dois algoritmos são similares no sentido que ambos computam o próximo estado do gerador utilizando uma recorrência linear sob o anél $\mathbb{Z}/n\mathbb{Z}$. As diferenças começam na escolha de parâmetros. Para o BBS, utilizamos o produto de dois primos p, q que satisfazem certas condições (vide documentação do fonte) e uma seed coprima à pq. Para o LCG, por sua vez, utilizamos quaisquer a, b, n desde que b e n sejam coprimos e $a - 1$ seja divisível pelos fatores de n (mais sobre essa questão na documentação). Outra diferença entre os dois algoritmos está em suas saídas. Enquanto o LCG utiliza cada estado x_n (um inteiro) como saída, o BBS mantém a sequência dos x_n apenas como estado interno e utiliza o bit menos significativo de cada x_n como saída (há outras possíveis maneiras

Algoritmo	Magnitude	Tempo médio em segundos
LCG	40	1.474e-06
LCG	56	1.306e-06
LCG	80	1.420e-06
LCG	128	1.496e-06
LCG	168	1.491e-06
LCG	224	1.560e-06
LCG	256	1.575e-06
LCG	512	2.104e-06
LCG	1024	3.973e-06
LCG	2048	7.786e-06
LCG	4096	1.098e-05
BBS	40	9.6750e-05
BBS	56	9.3579e-05
BBS	80	0.0001
BBS	128	0.0002
BBS	168	0.0003
BBS	224	0.0004
BBS	256	0.0008
BBS	512	0.0026
BBS	1024	0.00810
BBS	2048	0.05814
BBS	4096	0.3739

Tabela 1: Comparação entre LCG e BBS.

de extrair uma saída a partir do estado, vide referência). Assim, o BBS gera um fluxo de bits, os quais precisam ser convertidos para um número inteiro. O fazemos agrupando cada k bits, onde k é a magnitude do número desejado, em um inteiro (veja o método "_next_int(bits)").

2.3 Análise de complexidade

Ambos os algoritmos são bem simples: O LCG toma uma adição e uma multiplicação modular por iteração, enquanto o BBS toma apenas uma multiplicação modular. Observe que, quando forçados a gerar números de determinada magnitude, é possível que o algoritmo tenha que iterar várias vezes até chegar em um número grande o suficente. Na nossa implementação do LCG, utilizamos $n = 2^{\mathbf{bits}+1}$ para gerar números de magnitude igual a **bits**. Assim, metade do espaço de busca é composto de números grandes o suficiente e também não corremos o risco de gerar números grandes de mais; como com 41 ou mais bits quando $\mathbf{bits} = 40$. Utilizamos da mesma ideia para a escolha dos parâmetros do BBS: para o gerador de números pseudo-aleatórios de magnitude **bits**, escolhemos p, q tais que o produto pq esteja o mais próximo possível, mas sem passar, de $\mathbf{bits}+1$. Por fim, ressaltamos que o "percurso"que ambos os geradores fazem

é altamente dependente dos parâmetros escolhidos, o que dificulta uma análise geral (mas não impede).

3 Números Primos

Os testes utilizados foram: Miller-Rabin (MR) e Fermat, as referências utilizadas estão presentes na documentação do código fonte (arquivo "primes.py"). O número de rodadas para ambos os testes foi fixado em 40 pois é o suficiente para todas as magnitudes do Miller-Rabin utilizadas neste trabalho (veja a seção de comparação). Vale notar que o teste de Fermat está um pouco relaxado; isto é, o número de rodadas deveria ser aumentado para que o teste atinja um grau de confiança similar ao do Miller-Rabin. Não o fizemos porque a combinação BBS e Fermat já estava muito lenta, como pode ser visto na próxima tabela.

3.1 Resultados

A Tabela 2 a seguir resume o tempo tomado por cada uma das combinações - LCG e MR; BBS e Fermat - para gerar números primos pseudo aleatórios de diversas magnitudes. As seeds estão fixas como antes. Assim, basta descomentar as funções "gen_primes_LCG_MR"e "gen_primes_BBS_Fermat"para reproduzir os mesmos resultados (o tempo irá variar um pouco de máquina para máquina).

3.2 Comparação

Ambos os testes são similares no sentido que ambos são probabilísticos e utilizam congruências para testar por pseudo primos. Com probabilísticos, queremos dizer que, dado um ímpar positivo n, os algoritmos só conseguem nos afirmar se n é um pseudo primo com respeito às congruências testadas. Para ambos os algoritmos, os números primos são sempre detectados como pseudo primos. Dessa forma, o único erro que eles podem cometer é classificar um número composto como sendo primo. O trabalho dos algoritmos então é iterativamente diminuir a probabilidade que n seja um número composto. Ambos fazem isso através da busca de testemunhas para a não primalidade de n. Veremos que o número de testemunhas escala melhor para o Miller-Rabin do que para o teste de Fermat. Por isso o primeiro é mais usado na prática.

Façamos a análise para o Miller-Rabin. Seja $n = 2^s d + 1$, onde d é impar. Então n é pseudo primo de MR se satisfaz uma das seguintes s congruências:

$$a^d \equiv 1 \pmod{n} \tag{1}$$

$$a^{2^r d} \equiv -1 \pmod{n},\ 0 \leq r < s \tag{2}$$

Veja que as congruências são parametrizadas pela escolha da base a entre 0 e n. Essa realidade é similar a do teste de Fermat e, portanto, passaremos a chamar os a's de testemunhas ao invés de suas correspondentes congruências.

Combinação	Magnitude	Tempo médio (segundo)	Tamanho amostra
LCG e MR	40	0.0006	100
LCG e MR	56	0.0007	100
LCG e MR	80	0.0015	100
LCG e MR	128	0.0043	100
LCG e MR	168	0.0075	100
LCG e MR	224	0.0171	64
LCG e MR	256	0.0160	32
LCG e MR	512	0.1343	16
LCG e MR	1024	0.9606	8
LCG e MR	2048	5.809	4
LCG e MR	4096	42.8137	2
BBS e Fermat	40	0.0015	100
BBS e Fermat	56	0.0032	100
BBS e Fermat	80	0.0075	100
BBS e Fermat	128	0.0219	100
BBS e Fermat	168	0.0339	100
BBS e Fermat	224	0.0807	64
BBS e Fermat	256	0.1052	32
BBS e Fermat	512	0.8203	16
BBS e Fermat	1024	13.4314	8
BBS e Fermat	2048	93.3022	4
BBS e Fermat	4096	1023.3534	1

Tabela 2: Comparação entre MR e Fermat.

Antes de discutirmos a distribuição de testemunhas para um dado n, note que as escolhas $a = 1$ e $a = -1$ são inúteis, pois nunca serão testemunhas (a primeira sempre satisfaz $a^d \equiv 1 \pmod{n}$ e a segunda sempre satisfaz $a^d \equiv -1 \pmod{n}$). Há um resultado que mostra que, se n é composto, então pelo menos $3/4$ das bases são testemunhas. Assim, a chance de n ser classificado erroneamente após k iterações do MR é de no máximo $1/4^k$ (assumindo que as bases são escolhidas com reposição de forma aleatória e a probabilidade é uniforme). Note que a escolha anterior de $k = 40$ dá uma margem de erro práticamente nula.

Agora analisamos o teste de Fermat. Seja n um ímpar maior que três. Então, n é pseudo primo de Fermat se satisfaz a seguinte congruência:

$$a^{n-1} \equiv 1 \pmod{n}$$

Analogamente ao MR, as escolhas de $a = 1$ e $a = -1$ são inúteis. Outro resultado nos diz que, se n é composto[1], então pelo menos metade das bases a que são coprimas à n são testemunhas. Portanto, mesmo se supusermos que todas as bases são coprimas à n (o que não é verdade, porque n é composto!)

[1]Ignoramos aqui a questão dos números de Carmichael, outra falha do teste de Fermat.

a chance de n ser classificado erroneamente após k iterações do teste de Fermat seria de no máximo $1/2^k$. Assim, duas rodadas de Fermat equivaleriam a uma de MR. Porém, esse cenário se deteriora rapidamente com o aumento do número de fatores. Sob uma lente mais rigorosa (ainda ignorando números de Carmichael), há $\phi(n) - 2$ números coprimos à n em $[2, n-2]$, intervalo que contém $n-3$ números. Assim, sob hipóteses similares à análise do MR, a chance de uma base escolhida aleatóriamente cair no conjunto das testemunhas é $(\phi(n) - 2)/2(n - 3)$. Tirando o complemento, obtemos a probabilidade da base não ser uma testemunha é $1 - (\phi(n) - 2)/2(n - 3)$. Iterando k vezes, obtemos $(1 - (\phi(n) - 2)/2(n-3))^k$ que vai pra zero quando $k \to \infty$. Analizando gráficamente (não deu tempo de por o plot :)), é possível ver que essa probabilidade varia entre $1/2^k$ e $(9/10)^k$, dependendo da escolha de n. Até no pior caso, é possível convergir para zero, porém o número de rodadas necessárias para garantir isso será grande de mais para ser prático.

3.3 Análise de complexidade

O teste de Fermat realiza uma exponenciação modular por rodada, equanto o Miller-Rabin executa no náximo s exponenciações modulares por rodada. Assim, no pior caso, será necessário executar k operações para o Fermat e ks operações para o Miller-Rabin.