Números Primos – Um Algoritmo para Geração da Seqüência

Vilson L. Dalle Mole[1], Jefferson G. Martins[2]

***Resumo*--Este artigo faz uma breve discussão sobre a verificação da primalidade de um dado número n pertencente a IN*[3]. Primeiramente discute-se um algoritmo de geração da seqüência de números primos baseado na verificação da primalidade pela existência de um divisor maior que um e menor que um número n qualquer, através da operação mod (resto de divisão). Na seqüência é apresentado o algoritmo desenvolvido e sua implementação com base no crivo de Erastothenes. Os resultados apresentados indicam ser este um ótimo algoritmo dentro de seus objetivos de geração da seqüência de números primos.**

***Palavras-Chave*—Primalidade, Números Primos, Prime.**

I. Considerações Iniciais

A ciência matemática tem despertado a atenção do ser humano, desde os primórdios da raça humana, seja pela simples necessidade de contabilizar algo ou pela curiosidade de algumas características um tanto incomuns, dentre as quais está o número primo.

Embora pesquisas voltadas para a descoberta de novos e maiores números primos não sejam algo recente, elas têm recebido atenção especial da área computacional, principalmente quando esta se volta para a utilização de assinaturas digitais e criptografia [2][5].

Pela definição clássica “um número é primo, se e somente se, for divisível apenas por um e por ele mesmo” [1][2][3][4][5][6]. Embora existam outras definições quanto à primalidade dos números, tal como “primos entre si” e “primos gêmeos” etc., a definição anterior foi assumida neste trabalho e, segundo ela, para determinar a seqüência dos números primos, é necessário verificar cada um dos valores pertencentes a IN^*, determinando se o mesmo é ou não primo, conforme ilustrado no algoritmo do Quadro 1, a seguir.

Esta implementação consegue eliminar verificações redundantes por assumir um valor k ($k=\sqrt{n}$) como limite superior para a busca dos divisores de n. Neste contexto, tem-se o exemplo a seguir, ilustrado na Figura 1, no qual n=36.

1 2 3 4 6 9 12 18 36

Figura 1. Exemplo Ilustrativo para o Algoritmo do Quadro I

Dispondo-se os divisores de n seqüencialmente e ordem crescente, verifica-se que k é o ponto médio dos divisores de n, sendo $k=\sqrt{n}=6$.

Quadro I

Um Algoritmo para Geração da Seqüência de Números Primos

```
void geraSeqPrimo()
{
    Para ( n=1; n< infinito; n++)
    {
        se (primo (n))
            printPrimo(n);
    }
}

boolean primo(n | n ∈ IN* )
{
        se ( n ≤ 3 )
                return true;
        se ( n mod 2 == 0 )
            return false;
        k = √n;
        para ( i=3; i ≤ k; i += 2 )
        {
            se ( n mod i == 0 )
                return false;
        }
        return true;
}
```

Assim, é simples observar a correspondência entre os divisores e justificar o limite k para a referida busca, pois a cada par (d_i, d_j) de divisores de n tal que $d_i \leq \sqrt{n} \leq d_j$, ao se testar a divisibilidade de n por d_i encontra-se como quociente d_j e vice-versa.

Apesar deste algoritmo ser eficaz, ele não é eficiente uma vez que varre todo o conjunto dos números naturais realizando testes desnecessários sobre os números pares, mesmo considerando que a rotina primo(n) os elimine com apenas um passo, podendo o algoritmo ser melhorado, conforme ilustrado no Quadro 2, a seguir.

A nova versão do algoritmo eliminou todos os pares não primos sem a necessidade de operações computacionais.

[1] Docente junto ao Centro Federal de Educação Tecnológica do Paraná (CEFET-PR), unidade de Medianeira, Curso Tecnologia em Informática (e-mail: vilson@md.cefetpr.br).

[2] Docente junto ao Centro Federal de Educação Tecnológica do Paraná (CEFET-PR), unidade de Medianeira, Curso Tecnologia em Informática (email: jmartins@md.cefetpr.br).

[3] Conjunto dos números naturais com exclusão do zero

Considerando o intuito de gerar a seqüência dos números primos até um limite L, pode-se afirmar que o novo algoritmo obtém um ganho significativo em performance em relação ao anterior.

Outro ponto importante a considerar está na rotina primo(n). Esta toma o valor de n e procura um divisor entre 3 e √n, percorrendo a seqüência de números ímpares no intervalo [3, √n] até encontrar um divisor ou chegar ao fim da seqüência. Como a rotina retorna ao encontrar o primeiro divisor, parece perfeitamente plausível dizer que ela trata corretamente todos os casos não incorrendo em testes desnecessários. No entanto, uma análise mais apurada revela que, à medida que n cresce, o algoritmo passa a fazer cálculos desnecessários o que pode ser observado no exemplo a seguir - Quadro 2.

QUADRO II
ALGORITMO MELHORADO

```
void geraSeqPrimo()
{
printPrimo(1);
printPrimo(2);
printPrimo(3);

Para ( n=5; n< infinito; n += 2)
{
se (primo (n))
        printPrimo(n);
}
}

boolean primo(n | n ∈ IN* & n é impar )
{
se ( n ≤ 3 )
        return true;
k = √n;
para ( i=3; i ≤ k; i += 2 )
{
        se ( n mod i == 0 )
                return false;
}
return true;
```

Sendo n=169, seus divisores menores ou iguais a √n são 1 3 5 7 9 11 13

Notadamente, o teste do mod para i=9 é desnecessário, pois 9 é múltiplo de 3 e 9 < √169, o que é suficiente para dispensar o teste, pois, se n (169) for múltiplo de 3, também o será de 9. Assim, conclui-se que é necessário apenas efetuar o teste do resto sobre o conjunto de números primos menores ou iguais à √n.

O ponto central aqui é que para dispensar o teste do resto para os múltiplos (tal como o 9, no exemplo anterior), se faz necessário conhecer a seqüência dos números primos anteriores à √n. Assim, a seqüência deve ser armazenada de forma e deve-se poder recuperar seus elementos na mesma ordem em que foram gerados, dando um certo caráter estocástico ao algoritmo.

Todos esses melhoramentos, no entanto, ainda são insuficientes para tornar o algoritmo eficiente a ponto de poder ser utilizado para gerar a seqüência de números primos de 1 até um limite L grande o bastante, pois o crescimento do número de operações necessárias para a validação/negação de n como primo cresce também em função de n.

Abordando este contexto, o presente artigo está estruturado em 4 seções. Inicialmente é realizada uma introdução sobre a temática discutida. A seção II apresenta o Algoritmo implementado, suas vantagens e os fundamentos sobre o qual o mesmo está embasado. Os resultados são discutidos na seção III e na última seção (IV) são apresentadas as considerações finais.

II. O ALGORITMO IMPLEMENTADO

As origens do algoritmo implementado remontam a 200 AC com o crivo de Erastothenes. Segundo essa metodologia, para encontrar a seqüência dos números primos até um limite L, dispomos todos os números naturais de 1 até L em seqüência e então, a começar pelo numero 2, descartamos todos os seus múltiplos da seqüência. O primeiro número após o dois que ainda permanecer na seqüência é primo, no caso o 3. Repete-se o processo agora com os múltiplos de 3, encontrando-se o número primo 5 e assim sucessivamente até atingir o limite L [1][2][3][5]. A Tabela , a seguir, apresenta a aplicação do crivo de Erastothenes.

TABELA I
APLICAÇÃO DO CRIVO DE ERASTOTHENES

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Ciclo 0	1	2	3	x	5	x	7	x	9	x	11	x	13	x	15	x	17	x	19	x
Ciclo 1	1	2	3	x	5	x	7	x	x	x	11	x	13	x	x	x	17	x	19	x

A metodologia exposta acima, define a seqüência de números primos como sendo o conjunto das lacunas não preenchidas por múltiplos de 2 ou mais. Os múltiplos de 2 compõem a seqüência de números pares, no entanto a seqüência de números impares nunca será completamente preenchida por múltiplos de 3 que seria a primeira lacuna deixada pelos múltiplos de 2 ou por qualquer outra seqüência de múltiplos de primos subsequente. Cada nova lacuna identifica um número primo e gera uma nova seqüência de múltiplos.

Considerando que o número 2 é o único par primo, é perfeitamente correto desconsiderar todos os demais números pares, reduzindo assim em 50% as operações de descartes. A marcação recai apenas sobre os múltiplos em seqüência de cada número primo encontrado, essa operação requer apenas operações de incremento, uma para cada valor descartado. Além disso, uma análise mais profunda mostra que a marcação de cada primo inicia sempre a partir de seu quadrado, pois os múltiplos inferiores já foram marcados como múltiplos dos primos inferiores. Como pode ser observado na Tabela , foram necessários apenas os descartes

dos múltiplos de 2 e 3, observando que a marcação a partir do 2 começa no 4 que é seu quadrado e a marcação do 3 começa no 9 que também é seu quadrado. Expandindo essa tabela poderá ser visto que a marcação do 5 começará no 25 que corresponde a 5^2 e assim sucessivamente.

O algoritmo implementado faz uso de um vetor de 100.000 *booleans* onde são feitas as marcações, considerando-se apenas os números ímpares. Um sistema de janela deslizante promove que esse vetor desloca-se pela seqüência dos números naturais, sem limite de valor máximo. Devido às limitações de memória RAM, foi usado um vetor de 2.000.000 de elementos para armazenar a seqüência dos números primos. Com esse vetor é possível gerar a seqüência até um limite de 1.168.685.946.230.401.

A seqüência gerada é armazenada em arquivos de texto com 100.000 números primos cada. Também foi gerado um arquivo de resultados resumidos para fins de geração de gráficos.

III. Resultados Obtidos

O algoritmo implementado apresentou desempenho surpreendente, encontrando os 100.000 primeiros números primos em 0.530 segundos e os últimos 100.000, em 6.828 segundos, sendo que o último número primo gerado foi 12.942.466.003.

A evolução do tempo computacional medido apenas como função do tempo relógio, é dada pelo gráfico da Figura 2. Confrontando com o gráfico da Figura 3, a seguir, deduz-se que o tempo inicial é mínimo devido à grande concentração de número primos no intervalo. No entanto, após 100.000.000 de números gerados ocorrem poucas oscilações e acredita-se que estas são conseqüência de a máquina não ter sido dedicada exclusivamente à tarefa.

A Figura 3 apresenta a curva que representa o percentual de números primos encontrados no conjunto IN*. Essa curva foi gerada a partir da seqüência de primos versus o total de números.

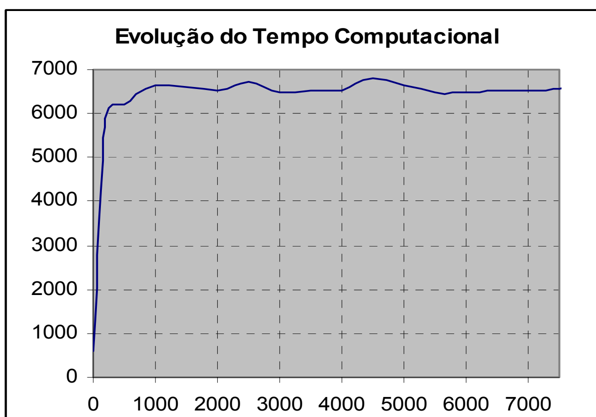

Figura 2. Evolução do Tempo Computacional (relógio)

Uma análise crítica dos resultados mostrou que a dispersão dos números primos aumenta significativamente na seqüência, conforme gráfico da Figura 3. Esta característica já era esperada, uma vez que os números primos podem ser vistos como as lacunas não preenchidas por múltiplos de 2 e seus sucessores primos. Assim, quanto maior a posição do número em IN mais distantes estão as lacunas.

O tempo relógio total foi de 10h 41min 20seg 712m para a geração dos 594.500.000 primeiros números primos, tendo o algoritmo sido executado em um microcomputador Pentium IV de 1.5Ghz com 256 MB de Ram/400MHz, SO Windows 2000 Professional e Linguagem JAVA .

Embora a base gerada permitisse verificar – através do algoritmo do Quadro 2 – valores maiores que os listados na Tabela 2, o tipo long da linguagem JAVA apresenta limite superior em 2^{63} – 64bits. Os tempos apresentados na Tabela 2 incluem o tempo necessário para carregar do disco o conjunto de números primos menores ou iguais à raiz quadrada do número que está sendo testado.

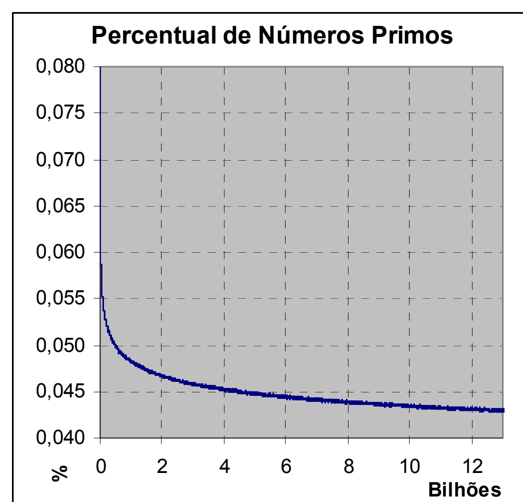

Figura 3. Dispersão dos números primos em %

Tabela II
Tempo Gasto para Verificar a Primalidade

Número	Primo	Time
9.223.372.036.854.775.643	sim	308.500 ms
9.223.372.036.854.775.601	não	105.234 ms
9.223.372.036.854.775.549	Sim	307.250 ms
9.223.372.036.854.775.531	Não	174.609 ms
9.223.372.036.854.775.507	Sim	312.421 ms
9.223.372.036.854.775.433	Sim	307.703 ms
9.223.372.036.854.775.421	Sim	312.359 ms
9.223.372.036.854.775.417	Sim	307.235 ms
9.223.372.036.854.775.399	Sim	325.484 ms
9.223.372.036.854.775.351	Sim	319.265 ms
9.223.372.036.854.775.337	Sim	305.703 ms
9.223.372.036.854.775.309	Não	236.531 ms
9.223.372.036.854.775.291	Sim	310.328 ms

A base gerada totaliza 6,57GB de disco e permite aplicar o algoritmo discutido inicialmente neste artigo para números na ordem de 10^{21}. Substituindo o tipo primitivo long por objetos do tipo BigInteger é possível executar o algoritmo do Quadro

2 para números dessa ordem.

IV. Considerações Finais

O estudo relatado neste artigo demonstrou que o algoritmo proposto e implementado se apresenta com ótimas perspectivas de perfomance. Além disso, a construção de uma base com a seqüência dos números primos permite validar ou negar um número como primo, com baixo custo de tempo computacional, permitindo inclusive a paralelização dos testes.

O algoritmo proposto para gerar a seqüência pode ser facilmente seqüencializado em um conjunto de máquinas rodando em paralelo e realizando um trabalho cooperativo de marcação, o que permite incrementar o tamanho da seqüência gerada, ao mesmo tempo em que reduz o tempo total de processamento.

Computação paralela distribuída baseada em trabalho cooperativo também poderia ser utilizada na validação ou negação de um número primo através do algoritmo do Quadro 2, podendo dessa forma reduzir de forma incremental o tempo necessário.

V. Referências

[1] H. Fonseca, "Números Primos", apresentado em Seminário Temático, Lisboa, Portual, 2000. Disponível: http://www.educ.fc.ul.pt/semtem/semtem99/sem24/.

[2] J.M. Pollard, "Theorems on Factorization and Primality Testing" in Proc. 1974 Cambridge Philosophical Society, 76, pp 512-528.

[3] O. Smith, (1995) Three Simple Prime Number Generators, Seven Seas Software Inc. Disponível: http://www.olympus.net/personal/7seas/primes.html.

[4] P. Ribenboim, The Little Book of Big Primes, Springer Verlag 1991.

[5] R.L. Rivest, "Finding Four Million Large Random Primes" in Crypto 1990 CRYPTO90, LNCS 537, pp. 625-626.

[6] W. J. Ellison, F. Ellison, Prime Numbers. New York: Wiley, 1985.