Relatório Parcial - Testes de Primalidade

Gervásio Protásio dos Santos Neto *

18 de setembro de 2013

Iniciação Científica em Criptografia
Orientador: Prof. Routo Terada

Instituto de Matemática e Estatística - IME USP
Rua do Matão 1010
05311-970 Cidade Universitária, São Paulo - SP

*Número USP: 7990996 e-mail: gervasio.neto@usp.br

1 Primeiros Passos

Comecei a implementação de diversos teste de primalidade. Tive cuidado de fazê-lo de forma a facilitar a realização do teste múltiplas vezes para colher dados para testes de hipótese.

O primeiro a ser implementado foi o de Miller-Rabin. A implementação foi feita em linguagem C e modularizada para que fosse possível aproveitar funções e esqueletos na implementação de outros testes.

O algoritmo de exponenciação modular foi o método binário, baseado principalmente no pseudo-código apresentado em Terada, R. "Criptografia e Segurança de Redes" e tem complexidade O($\log e$), onde e é o expoente. Para descobrir os inteiros t, c tais que $n - 1 = 2^t c$, foi utilizada uma pequena sub-rotina com complexidade O($\log n$).

A implementacão tem se mostrado eficiente, levando em média 61,7 nanossegundos para testar primos da ordem de 10^9, com probabilidade de erro Pr = $o(\frac{1}{4^{30}})$. Uma análise mais rigorosa da complexidade ainda é necessária, contudo, parece algo pseudo-polinomial, linear em s (o expoente da probabilidade de erro) e logarítmico ou no pior caso $O((nlog(n))$ quanto ao tamanho da entrada.

O segundo teste implementado foi o Teste de Wilson, baseado no teorema de Wilson, que afirma que se p é primo, então $p - 1 \equiv -1 \bmod p$.

Apesar de significantemente mais eficiente que testes triviais, como o Crivo de Erastónes, ainda se mostrou menos eficiente que o teste de Miller-Rabin, chegando a levar 30 segundos para testar a primalidade de $2^{31} - 1$ (um primo de Mersenne que também é o maior número que pode ser representado por um int em C).

Contudo, diferentemente do teste Miller-Rabin, o teste de Wilson é determinístico e pode ser usado para validar resultados obtidos com o teste de Miller-Rabin.

O teste de Lucas, por requerer a fatoração de $n - 1$ (sendo n o número cuja primalidade está em teste), não foi implementado. Não se conhece algoritmos polinomiais para fatoração numérica e o problema parece bem mais difícil que simplesmente decidir a primalidade.

De fato, enquanto o algoritmo AKS mostra que o problema de decisão da primalidade está em P, não se sabe se fatoração numérica também pertence a essa classe de complexidade. Contudo, sabe-se que fatoração é um problema de decisão que está na intersecção de NP e co-NP, o que é um forte indício de que não é um problema NP-completo.

O teste baseado na Soma de Dois Quadrados (que é influenciado pelo Teorema Natalino de Fermat [1]) também parecia ter uma implementação complicada e pouco eficiente. Contudo, os Lemas e Teoremas desenvolvidos para provar sua

[1] Um primo p pode ser escrito como soma de dois quadrados $x^2 + y^2$ se, e somente se, $p \equiv 1 \bmod 4$.

correção serem valiosos do ponto de vista teórico, eles não vingam uma implementação algorítmica prática. Mesmo assim, tais resultados são bastante profundos e ainda preciso estudá-los e entendê-los melhor.

O teste de Pratt, enquanto não algoritmicamente viável, leva a conclusões interessantes. Um dos resultados do teste é uma upper-bound para o número de operações necessárias para verificar a primalidade de um número. O resultado final é no máximo $O(log\ n)$ divisões e exponenciações necessárias. O resultado exato é $3log(n)$ - 2.

Acredito que uma análise mais profunda desse resultado possa ajudar na análise de outros testes, bem como ajudar a decidir um número de vezes para rodar o algoritmo de Miller, de tal forma que mais iterações seriam redundante. Contudo, os teoremas necessários para atingir tal resultado são complicados e mais tempo é necessário para entendê-los.

2 Próximos Estágios

Um problema que apresentou-se e no qual começarei a trabalhar imediatamente, antes de outras implementações, é achar uma forma para lidar como números superiores a 2^{31}, visto que do ponto de vista prático, esse valor ainda é considerado pequeno. Um bom valor limite parece ser da ordem de 4096 bits.

Uma solução para esse este problema pode ser abandonar o uso da linguagem C e dar preferência para Perl ou Java, que contém módulos capazes de lidar com números maiores que esse limiar (os bigInts). O módulo bigInt de Perl, por exemplo, não estabelece limites ao tamanho do inteiro, podendo este ser tão grande quanto a mémoria do computador permitir. A biblioteca *Relic*-toolkit, uma API em C, também permite inteiros de precisão arbitrária.

Em seguida, deveremos implementar o teste AKS e comparar seu desempenho com os demais testes implementados. Não se espera que o AKS possua desempenho melhor que o Teste de Miller-Rabin, contudo é um teste conceitualmente importante e sua implementação aparenta ser relevante no contexto da pesquisa.

Contudo, alguns problemas se apresentam imediatamente em uma tentavia de implementação do AKS. Em sua execução, além de ser necessário calcular ordem multiplicativa em um grupo, é necessário calcular $\varphi(\mathrm{r})$, um problema equivalente a fatoração de inteiros. Irá-se procurar alguma função $f(n)$ ou alguma heurística que nos livre do fardo de realizar este cálculo.

Também serão implementados testes que testam primos específicos, como o teste de Lucas-Lehmer para primos de Mersenne[2], o teste de Pépin para primos de Fermat[3] e o teste de Proth para primos da forma $k2^n + 1$. Cada um desses testes será comparado com os já implementados e de propósito geral com entradas compostas apenas de seu respectivo tipo específico de primo.

Uma outro ideia é também verificar se ocorre algo especial quando testando primos de Sophie Germain[4]. Um dos motivos disso é que a complexidade do algoritmo AKS depende de uma conjectura sobre a distribuição desses primos.

Uma vez terminadas as implementações e colhidos dados suficientes, serão realizados testes de hipótese para formalizar os resultados de qual teste é mais eficiente e em que situação. As "amostras" dos testes serão médias de 120 análises prévias. Dessa forma, pelo Teorema Central do Limite, teremos uma "população" aproximadamente normal, o que tornará os testes mais confiáveis e permitirá conclusões mais profundas.

Os resultados colhidos tanto nas análises quanto nos teste de hipótese serão então plotados para permitir melhor entendimento visual.

Terminada esse fase, começarei a estudar o teste sobre curvas elípticas, que por ser o ponto principal do projeto, foi deixado para depois que o entendimento

[2]Primos da forma $p = 2^q - 1$, onde q também é um número primo.
[3]Primos da forma $p = 2^{2n} + 1$, $n \in \mathrm{N}$.
[4]Primos p tais que $2p + 1$ também são primos.

sobre testes em geral estivesse mais consolidado. Serão estudados algoritmos para contagem de pontos sobre curvas geradas aleatóriamente (possivelmente heurísticas, pois acredita-se que esse problema possa ser NP-completo).

Então, procurar-se-á a melhor forma de combinar este algoritmo com os anteriormente estudados para desenvolver um que tenta resolver o problema de decisão no menor tempo possível.

A ideia de combinação dos algoritmos veio da linguagem python, onde o algoritmo padrão de ordenação funciona de forma semelhante, tentando se comportar de forma a ter eficiência máxima. É uma idéia que vem crescendo, sobre tudo sob a forma dos chamados algoritmos genéticos. Contudo não se pretende investigar muito a fundo esse tipo de algoritmo por fugir ao escopo da pesquisa.

2.1 Resumo das Perspectivas

- Implementar o teste AKS
- Testar a implementação em linguagens com bigInt
- Implementar testes pra primos específicos
- Realizar testes de hipótese para melhor entender a eficiência dos testes
- Aprofundar a análise dos algorítmos
- Desenvolver heurísticas para porções potencialmente custosas dos algorítmos
- Obter vizualização gráfica dos resultados
- Estudo do teste sobre curvas elípticas
- Integração da pesquisa

3 Soluções sendo estudadas

Ainda estou tentando desenvolver uma heurística para evitar a ordem do grupo multiplicativo no algoritmo AKS.

Contudo, para limitar φ(r) comecei a utilizar a seguinte abordagem: calculei os valores da função Totiente até 10^4 e então plotei os resultos para melhor entender o comportamento da função. As Figuras 1, 2 e 3[5] contém alguns dos gŕaficos obtidos.

Enão foram ultizadas funções de curve fitting tanto do GNUPlot[6] quanto do Wolfram|Alpha[7]. Ambos retornaram resultados pouco consistentes quando se tentou encaixar os dados em funções não lineares.

Contudo, ao aplicar-se regressão linear no conjunto de dados, obtivemos um aproximação razoável, uma reta da forma $0.61x \pm 11$. Tal reta está ressaltada na Figura 3.

Ainda não testamos o algoritmo AKS com essa upper-bound.

Além disso, realizamos mais alguns experimentos com o Teste de Miller-Rabin e plotamos os resultados tamanho da entrada vs. tempo de execução médio. O gŕafico está na Figura 4. Ainda realizaremos um teste de curve fitting para melhor entendet seu comportamento.

[5] Ver Apêndice.
[6] Aplicativo standard de plotting do sistema Linux
[7] www.wolframalpha.com

4 Apêndices

4.1 Códigos

Exponencição Modular

```
unsigned long modExp(unsigned long m, unsigned long e,
unsigned long n){

  unsigned long temp = 1;
  int j;
  int b;
  for(j = log2(e); j >= 0; j--){
    temp = (temp*temp)%n;
    b = (e & (1 << j)) >> j;
    if(b == 1)
      temp = (temp*m)%n;
  }
  return temp;
}
```

Adiquirir parametros t e c

```
void getParameters(int* t, long* c,unsigned long n){
  int x = n-1, aux = 0;
  while(x%2 == 0){
    x/=2;
    aux++;
  }
  *t = aux;
  *c = (n-1)/(pow(2,aux));
}
```

Teste de Miller-Rabin

```
Boolean millerRabinTest(unsigned long n, int s){
  int i,j, t=0, aux;
  unsigned long c = 0, a = 0, r0, r1;

  if((n % 2 == 0 && n != 2) || n == 1 )
    return FALSE;
  if(n == 2) return TRUE;

  getParameters(&t, &c, n);

  srand(time(NULL));
  for(i = 0; i < s; i++){
    a = rand()%(n-4) + 2;
    r0 = modExp(a, c, n);
    r1 = (r0 * r0)%n;
    for(j= 0; j < t; j++){
      if(r1 == 1 && r0 != 1 && r0 != n-1)
        return FALSE;
      aux = r1;
      r1 = (r1*r1)%n;
      r0 = aux;
    }
    if(r1 != 1) return FALSE;
  }
  return TRUE;
}
```

Fatorial Modular

```
unsigned long factModN(unsigned long n, unsigned long m){
  unsigned long i, x = 1;
  for(i = 1; i <= n; i++)
    x = (x*i)%m;
  return x;
}
```

Teste de Wilson

```
Boolean wilsonTest(unsigned long n){
  unsigned long f = factorialModN(n-1,n);
  if(f == n-1) return TRUE;
  return FALSE;
}
```

Código para avaliação de desempenho

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <time.h>
#include "rabinFunctions.h"

void help();

int main(int argc, char* argv[]){
  char c, *outputName, *inputName;
  FILE* input;
  FILE* output;
  Boolean runtime = TRUE, single = FALSE, result;
  unsigned long p = 2;
  int runs = 30, primes = 0, composites = 0;
  clock_t start, end;
  double elapsed = 0;

  if(argc < 2){
    help();
  }

  while( (c = getopt(argc, argv, "t:f:o:r:y")) != -1){
    switch(c){
    case 'f':
      inputName = optarg;
      break;
    case 't':
      p = atoi(optarg);
      single = TRUE;
      break;
    case 'o':
      outputName = optarg;
      break;
    case 'y':
      runtime = FALSE;
      break;
    case 'r':
      runs = atoi(optarg);
      break;
    }
  }

  output = fopen(outputName, "w");
  input = fopen(inputName, "r");

  if(!single){
    if(input == NULL){
```

```
      puts("Unable to open file for reading.");
      help();
    }
    while(fscanf(input, "%lud", &p) != EOF){\

      start = clock();

      result = millerRabinTest(p, runs);

      end = clock();

      elapsed += (double)(end - start)/CLOCKS_PER_SEC;

      if(result){
        if(!output) fprintf(stdout,
        "%ld is a prime.\n", p);

        else fprintf(output, "%ld is a prime.\n", p);
        primes++;
      }
      else{
        if(!output) fprintf(stdout,
        "%ld is composite.\n", p);

        else fprintf(output, "%ld is composite.\n", p);
        composites++;
      }
    }

    fprintf(stdout, "There were %d composites and %d
         primes.\n"composites, primes);

    if(runtime) fprintf(stdout, "TIME: %f (reading
    and printing times not included)\n", elapsed);

  }

  else{
    start = clock();
    result = millerRabinTest(p, runs);

    end = clock();

    elapsed = (double)(end - start)/CLOCKS_PER_SEC;

    if(result){
      if(!output)fprintf(stdout,"%ld is a prime.\n",p);

      else fprintf(output, "%ld is a prime.\n", p);
    }
```

```
    else{
      if(!output) fprintf(stdout,"%ld is composite.\n",p);
      else fprintf(output, "%ld is composite.\n", p);
    }

    if(runtime) fprintf(stdout, "TIME: %f (reading
    and printing times not included)\n", elapsed);

  }

  fclose(input);
  fclose(output);

  return 0;
}

void help(){
  fprintf(stdout, "Use -t<number> for single number.\n"
          "Use -f<input file name> to read from a file.\n"
          "The output will the STDOUT.\n"
          "To redirect outuput, -o<output file name>.\n"
          "To hide teh running time, use -y.\n"
          "-r<nRuns> to change the argument s.\n"
          );
  exit(-1);
}
```

4.2 Figuras

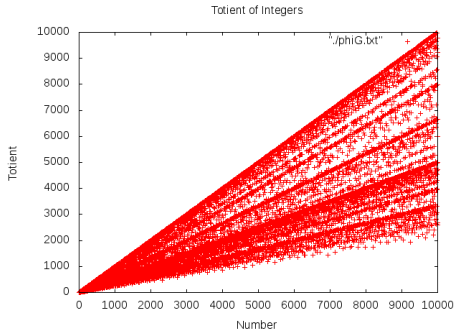

Figura 1: Distribuição de $\varphi(n)$, $n < 10^4$.

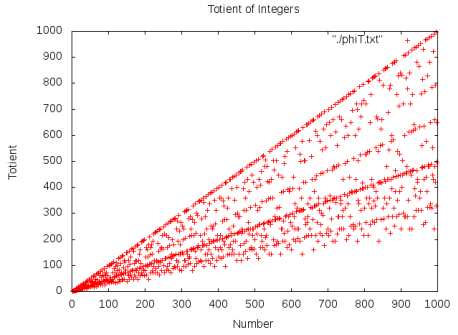

Figura 2: Distribuição de $\varphi(n)$, $n < 10^3$.

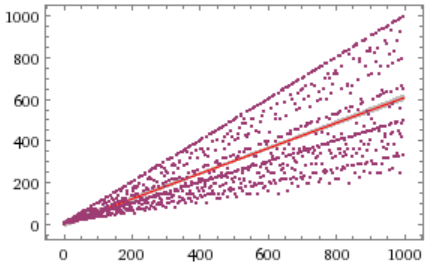

Figura 3: Distribuição de $\varphi(n)$, $n < 10^3$. A linha ressaltada em vermelha representa uma reta obtida por Regressão Linear.

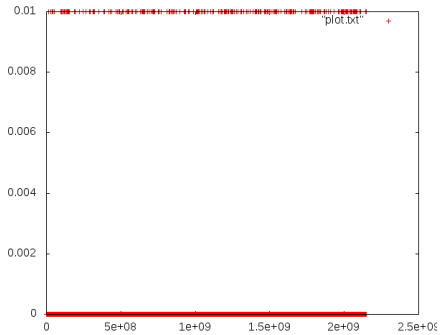

Figura 4: Tempos do Algoritmo Miller-Rabin para 10^4 entrada aleatórias.