Universidade do Estado do Rio de Janeiro
Centro de Tecnologia e Ciências
Instituto de Matemática e Estatística
Departamento de Informática e Ciência da Computação

Anderson Zudio de Moraes
Victor Cracel Messner

O Algoritmo AKS e a Evolução dos Testes de Primalidade

Rio de Janeiro
2016

Anderson Zudio de Moraes
Victor Cracel Messner

O Algoritmo AKS e a Evolução dos Testes de Primalidade

Monografia apresentada, como requisito parcial para obtenção do título de Bacharel, ao Instituto de Matemática e Estatística, da Universidade do Estado do Rio de Janeiro.

Orientador: Prof. Paulo Eustáquio Duarte Pinto

Rio de Janeiro
2016

AGRADECIMENTOS

Queremos agradecer a todo apoio das nossas famílias, por tornar a graduação em uma excelente instituição de ensino superior possível.

Agradecemos ao nosso orientador, Paulo Eustáquio, pelo o incrível tema deste projeto e pela valiosa orientação. Assim como aos demais professores que em muito colaboraram com nossa formação acadêmica, dos quais destacamos: Leandro Marzulo, Fabiano Oliveira, Vera Werneck, Rosa Costa, Igor Machado, Alexandre Rojas, Vitor Hugo Pontes, Rogério Quintino e Clícia Stelling.

Agradecemos também aos nossos colegas de graduação, que passaram pelas mesmas dificuldades e barreiras durante todo o procedimento de aprendizado até aqui: Robson Eduardo, Ronaldo Ferreira, Raquel Marcolino, Giancarlo França, Victor da Cruz e Bruno Masquio.

RESUMO

ZUDIO, A. M.;MESSNER, V. C. *O Algoritmo AKS e a Evolução dos Testes de Primalidade*. 2016. 168 f. Monografia (Bacharelado em Ciência da Computação) – Instituto de Matemática e Estatística, Universidade do Estado do Rio de Janeiro, Rio de Janeiro, 2016.

A determinação de números primos do ponto de vista computacional tornou-se uma tarefa muito importante, pois esses números desempenham um papel destacado em inúmeros contextos da computação, como por exemplo, na criptografia. Neste trabalho apresentamos uma visão da evolução das teorias e algoritmos necessários para tal fim. Este projeto procura detalhar os principais avanços teóricos e computacionais do problema de determinação de primos, mostrando as abordagens históricas e as mais recentes, mais voltadas para a utilização computacional, dentre as quais se encaixam os algoritmos probabilísticos, efetivamente utilizados na prática. Destacamos também o algoritmo *AKS*, de grande importância teórica, pois foi o primeiro teste de primalidade determinístico, polinomial e geral. Procuramos dar uma abordagem que ilustre as teorias e também discuta as questões práticas de implementação, utilizando a linguagem de programação *C++*.

Palavras-chave: AKS. Algoritmo. Primalidade.

ABSTRACT

ZUDIO, A. M.;MESSNER, V. C. 2016. 168 f. Monografia (Bacharelado em Ciência da Computação) – Instituto de Matemática e Estatística, Universidade do Estado do Rio de Janeiro, Rio de Janeiro, 2016.

The determination of prime numbers from a computational point of view has become a very important task, as these numbers play a leading role in numerous contexts of computing, such as in cryptography. We present a vision of the evolution of theories and algorithms necessary for this purpose. This project seeks to detail the main theoretical and computational advances in the problem of determining prime numbers, showing historical approaches, in particular the probabilistic algorithms, effectively used in practice. We also highlight the algorithm AKS, of great theoretical importance, because it was the first primality test wich was deterministic, polynomial and general. We try to give an approach that illustrates the theories and also discuss the practical issues of implementation, using the programming language C++.

Keywords: AKS. Algoritmo. Primalidade.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE ALGORITMOS

LISTA DE LISTAGENS

SUMÁRIO

1 INTRODUÇÃO

Este projeto lida com o problema de determinação computacional de números primos. Um número é primo, se e somente se for um natural maior ou igual a 2 tal que seus únicos divisores inteiros positivos sejam 1 e ele mesmo. À primeira vista, parece que o problema é simples, mas na prática, a determinação de números primos grandes é uma tarefa difícil e por isso utilizamos algoritmos probabilísticos na sua solução. Até recentemente, não era conhecido um algoritmo polinomial, geral e determinístico, para o problema. Isto mudou quando três pesquisadores indianos do Instituto Indiano de Kanpur publicaram a primeira versão do algoritmo *AKS* (Agrawal–Kayal–Saxena primality test), em 2002, causando um grande impacto na comunidade científica, pois até este ponto ninguém tinha sido capaz de elaborar um algoritmo eficiente para o problema. Imediatamente após a primeira versão do *AKS*, o algoritmo foi melhorado por grandes pesquisadores da área, até que em 2004 os autores do algoritmo publicaram o artigo *Primes is in P* (AGRAWAL; KAYAL; SAXENA, 2004) com a terceira versão do algoritmo. Há várias outras versões do algoritmo modificadas por outros pesquisadores que são conhecidas como algoritmos da classe *AKS*. Este projeto mantém o foco na terceira versão do *AKS*, dos autores originais, fornecendo detalhes do algoritmo, incluindo implementações e testes de performance, utilizando a linguagem *C++* e bibliotecas que fornecem ferramentas para aplicações da área da Teoria dos Números.

Apesar da enorme importância teórica do *AKS*, atualmente as aplicações práticas geralmente não utilizam esse algoritmo, optando por enfoques probabilísticos com métodos de alta performance, inspirados em algoritmos anteriores ao *AKS*, baseando-se em resultados teóricos importantes conhecidos, como por exemplo, o Pequeno Teorema de *Fermat*. Esses algoritmos apresentam, em geral, uma margem de erro muito pequena na determinação de primos, o que os tornam viáveis para utilizações práticas.

Neste trabalho apresentamos os marcos históricos na determinação de primos, dentre eles o certificado de *Pratt*, o teste de primalidade de *Solovay-Strassen*, o teste de primalidade de *Miller-Rabin*, o algoritmo *ECPP*, o algoritmo de *Baillie-PSW*, além do algoritmo *AKS*, propriamente dito.

Neste capítulo serão apresentadas a motivação, os objetivos, a estruturação do projeto e, por fim, uma pequena introdução teórica dos conceitos básicos mais utilizados no decorrer do projeto.

1.1 Motivação

A matemática, assim como as outras ciências, sempre despertaram um grande interesse nos homens por revelarem verdades sobre assuntos desconhecidos. Neste trabalho discutiremos aspectos relativos ao tratamento computacional de números primos.

Como os testes de primalidade possuem grande importância para os sistemas de segurança atuais e suas fundamentações matemáticas derivam das mais variadas ideias, a apresentação dos principais testes de primalidade revelariam a evolução do pensamento matemático para a resolução de problemas que são estudados há milênios.

Além da abordagem histórica sobre o problema, é de extrema importância que sejam apresentadas implementações dos testes citados, bem como seus algoritmos e teoremas que os fundamentam para que, além da contextualização histórica, seja dada uma abordagem prática e teórica sobre os mesmos.

Durante nossa graduação sempre tivemos interesse por problemas de computação, participando, inclusive, em maratonas de programação, até que encontramos um problema de criptografia cuja a abordagem não poderia utilizar critérios mais simples estudados durante o curso de graduação.

Tendo isso em vista, decidimos estudar mais profundamente sobre os testes de primalidade e, após uma árdua pesquisa em conteúdos de alto rigor matemático conseguimos solucionar o problema. Portanto uma das motivações desse projeto, já que possuíamos interesse sobre o tema, tornou-se elaborar um conteúdo de fácil entendimento, tornando-o acessível para que alunos do quarto período da graduação da computação ou de outros cursos que possuam um conhecimento matemático básico e noções de computação sejam capazes de aprender o conteúdo. É importante ressaltar que assim como no passado, durante o desenvolvimento deste projeto, abordagens didáticas sobre diversos dos testes citados não foram encontradas na literatura com facilidade, e em alguns casos não foram encontradas de forma alguma.

Tendo em vista todos os pontos descritos, devemos acrescentar uma motivação final que consiste em apresentar um trabalho que contenha, não somente um algoritmo sobre o teste de primalidade, mas sim um conjunto de algoritmos desse tipo que possuam relevância histórica e prática, diferindo assim dos artigos encontrados tanto pelo aspecto didático, quanto pela quantidade de testes descritos.

1.2 Objetivo

O objetivo deste projeto é realizar uma abordagem histórica computacional do problema de determinação de primos, de forma que fique claro a importância, o funcionamento e as propriedades matemáticas que fundamentam os vários algoritmos e teorias

envolvidos, tais como o certificado de *Pratt*, os principais testes de primalidade probabilísticos e o algoritmo *AKS*. Também temos como objetivo apresentar uma abordagem prática dos algoritmos, com implementações na linguagem de programação *C++* visando performance e simplicidade de todos os algoritmos envolvidos. Em especial, queremos também determinar a melhor biblioteca gratuita de *C++* para a implementação do teste de primalidade *AKS* e expor os maiores problemas em utilizar esse teste na prática.

É nossa intenção que este este projeto torne fácil o entendimento das ideias em que se basearam os testes de primalidade abordados e de cada implementação, para que desperte o interesse de alunos de graduação ou mestrado a estudar problemas na área de Algoritmos e Teoria dos Números.

1.3 **Estrutura**

O projeto consiste de 5 capítulos, um apêndice e duas seções de anexos. Cada capítulo é subdivido em seções. Geralmente, uma seção procura separar a ideia principal de um algoritmo de seu pseudocódigo e implementação, de modo que, às vezes, a implementação pode não seguir exatamente os passos descritos pelo algoritmo devido a alguma restrição da linguagem de programação ou alguma otimização feita em um certo passo da aplicação. Detalhamos cuidadosamente essas diferenças na seção em que cada código está implementado. Também mantivemos em seções separadas todos os testes de performance de cada algoritmo, de maneira que resultados fiquem juntos nas mesmas tabelas e gráficos para facilitar a compreensão do efeito prático dos testes.

Os capítulos deste trabalho são:

- Capítulo 2 - Este capítulo descreve as abordagens anteriores ao *AKS*: o algoritmo de força bruta, o crivo de Erastótenes, o pequeno teorema de Fermat, o certificado de *Pratt* e cinco testes de primalidade probabilísticos. Para o certificado de *Pratt* o trabalho detalha sua importância teórica e o seu funcionamento. Para cada teste de primalidade abordado é feita uma pequena introdução histórica do mesmo, revelando o porquê desse teste ter sido escolhido em meio a tantos outros possíveis testes. Para cada abordagem são mostradas as ideias necessárias para a construção do algoritmo bem como as provas matemáticas envolvidas. São também apresentados os algoritmos em pseudocódigo e sua implementação em *C++*, relacionando, o máximo possível, o programa com o pseudocódigo apresentado. No final do capítulo apresentamos os resultados de benchmarks das implementações feitas.

- Capítulo 3 - Consiste na abordagem teórica do algoritmo *AKS*. O capítulo apresenta detalhadamente a ideia do algoritmo, o seu funcionamento, seu pseudocódigo e sua complexidade assintótica.

- Capítulo 4 - Apresenta a abordagem prática do *AKS*. São detalhadas quatro implementações do mesmo algoritmo apresentado no Capítulo 3. Nele também mostramos as limitações de cada implementação e realizamos testes práticos de performance com todas elas.

- Capítulo 5 - Consiste na conclusão do projeto, com as observações dos autores e possíveis trabalhos futuros.

O apêndice procura detalhar alguns algoritmos básicos que são usados durante o trabalho. A primeira seção de anexos do trabalho mostra todos os códigos implementados incluindo comentários detalhados sobre cada teste que não são mostrados durante os pequenos blocos de códigos dos capítulos. A segunda seção de anexo mostra os arquivos utilizados para efetuar os testes práticos.

1.4 **Fundamentos teóricos**

Esta pequena seção descreve alguns dos fundamentos teóricos básicos necessários para o entendimento deste projeto, somente para efeitos de revisão. O trabalho em geral tem sua parte teórica baseada em Álgebra e Complexidade de Algoritmos.

1.4.1 Fundamentos teóricos de Álgebra

Todos os conceitos utilizados estão no livro (COUTINHO, 2004), que sumariza os conhecimentos de grupos, anéis e corpos, envolvidos nas diversas abordagens. Optamos por não repetir esses conceitos por serem muito extensos e por apresentar alguns deles no momento em que são utilizados no texto.

1.4.2 Fundamentos de Complexidade de Algoritmos

A complexidade de um algoritmo está ligada diretamente com sua eficiência, de modo que nos revele o quanto de recurso de máquina o algoritmo requer quando colocado em execução. Esta complexidade geralmente é a medição de tempo de processamento ou de consumo de memória. Quando falamos de complexidade de tempo, estamos nos referindo à quantidade de passos que um algoritmo requer para a sua execução completa, onde cada passo é uma operação primitiva da máquina. Para complexidade de espaço temos que o interesse é saber o número de células de memória utilizadas simultaneamente durante o processamento do algoritmo.

Notação O-grande

A complexidade pode ser classificada de diversas maneiras. Uma das classificações fundamentais é o pior caso, cujo objetivo é o de medir a maior utilização possível do recurso em questão. O processo para medir a complexidade de um algoritmo consiste em associar ao parâmetro de entrada n, indicador do tamanho dessa entrada, com o recurso gasto da máquina, através de uma função $f(n)$. Às vezes é muito difícil ou até mesmo impossível obter esta função. Usa-se uma aproximação $T(f(n))$ para descrever a complexidade, tal que ela se aproxima do valor do termo de maior grau de $f(n)$, à medida que a entrada cresce. Em outras palavras:

$\lim_{N\to\infty} \frac{T(f(n))}{f(n)} = 1$, onde $T(f(n))$ é o termo de maior grau de $f(n)$.

Com isto nós temos uma família de notações que são amplamente utilizadas para evidenciar a ordem de grandeza deste termo de maior grau da função. Uma notação que vamos utilizar durante todo o projeto é a notação *big-O* (O-grande):

Definição 1.4.1. *Dizemos que $f(n) = O(g(n))$ se, para alguma constante C e um valor inicial $n_0, \forall n \geq n_0$ temos $f(n) \leq C \times g(n)$.*

Na prática as funções escolhidas para $g(n)$ normalmente possuem apenas um termo que representa a ordem da função $f(n)$. Por exemplo, para quaisquer inteiros A, B e C, se $f(n) = An^4 + Bn^2 + Cn$, então $f(n) = O(n^4)$. Esta notação é largamente utilizada na literatura e é a mais conhecida de todas.

Notação Ômega-grande

Outra maneira de classificar a complexidade é pelo seu melhor caso, cujo o objetivo é medir a menor utilização possível do recurso em questão. O processo é exatamente o mesmo da notação anterior, só que dessa vez estamos interessado na seguinte definição:

Definição 1.4.2. *Dizemos que $f(n) = \Omega(g(n))$ se, para alguma constante C e um valor inicial $n_0, \forall n \geq n_0$ temos $f(n) \geq C \times g(n)$.*

Utilizando o exemplo anterior, para quaisquer inteiros A, B e C, se $f(n) = An^4 + Bn^2 + Cn$, então $f(n) = \Omega(n)$. Esta notação também será utilizada durante o projeto.

Classes de complexidade de problemas

Vale a pena revisar também alguns conceitos da Teoria da NP-Completude utilizadas neste projeto, relativa às classes de complexidade de problemas. Os problemas computacionais podem ser classificados de acordo com sua dificuldade característica. Algumas dessas classes são:

- Classe P - A classe P de *Polynomial time* é uma das classes de complexidade fundamentais que consiste dos problemas em que são conhecidos pelo menos um algoritmo determinístico e eficiente que o resolve, ou seja, para esse problema há uma máquina de *Turing* determinística que resolve o problema em tempo polinomial.

- Classe NP - A classe NP, *Nondeterministic Polynomial Time* é uma classe fundamental de complexidade que consiste dos problemas de decisão em que a resposta *Sim* pode ser verificada de forma eficiente, ou seja, há uma prova que pode ser verificada por uma máquina determinística de *Turing* em tempo polinomial de que a resposta *Sim* é realmente válida.

- Classe co-NP - Esta classe consiste dos problemas de decisão em que a resposta *Não* pode ser verificada de forma eficiente, ou seja, exatamente como a classe *NP* mas a prova verificada em tempo polinomial pela máquina de *Turing* determinística atesta que a resposta é realmente *Não*.

Algoritmos Pseudo-Polinomiais

Muitos problemas são resolvidos por algoritmos numéricos em que a análise de complexidade assintótica do algoritmo para uma entrada n com valor numérico é polinomial mas, quando se analisa a mesma entrada n em função de seu tamanho na representação binária em *bits*, a complexidade assintótica é exponencial. Estes algoritmos são conhecidos como Pseudo-Polinomiais. Um exemplo é o algoritmo de força bruta para a determinação de primos como será visto a seguir.

Considere que vamos checar se uma entrada n é um número primo ou composto. Uma aproximação intuitiva é utilizar o algoritmo que checa sequencialmente se cada um dos números $\{2,\ 3,\ [...],\ \sqrt{n}\}$ divide n. É fácil checar que essa aproximação tem complexidade assintótica $O(\sqrt{n})$, quando tomamos a análise em função da entrada n com o seu valor numérico. Mas, na verdade, este algoritmo tem complexidade assintótica exponencial quando analisamos a entrada n de forma correta, que é pelo tamanho de sua representação binária, que vale aproximadamente $\lg n$. Dessa forma é fácil verificar que na verdade estamos falando de um algoritmo de complexidade assintótica da forma $O(2^{(\log n)/2})$.

2 A EVOLUÇÃO DOS TESTES DE PRIMALIDADE

O problema de determinação de primos vem cada vez mais recebendo a atenção de pesquisadores, tendo seu ápice com o avanço do uso de maquinas, onde a a segurança de informações, através do uso de criptografia, utiliza fortemente muitos resultados pesquisados sobre números primos.

Este problema era recentemente da classe *co-NP*, até que em 1975 *Vaughan Pratt* demonstrou que o problema está em *NP* (PRATT, 1975), através da utilização de um certificado de primalidade especial, promovendo um importante avanço teórico para o problema. Diversos testes de primalidade probabilísticos foram utilizados no decorrer dos anos para resolver o problema.

Neste capítulo fazemos um breve relato da evolução das abordagens sobre números primos de interesse computacional. Destacamos as abordagens da época de Euclides, o pequeno teorema de Fermat, o certificado de *Pratt* e os algoritmos probabilísticos principais. Contudo, o algoritmo *AKS*, que em 2002 avançou o problema para uma nova era, demonstrando que este está em *P*, vai ser abordado no próximo capítulo, pois nenhum outro teste de primalidade teve tanto impacto.

Todas as implementações apresentadas vão utilizar a linguagem *C++* e podem ser compiladas utilizando o compilador de C++ do *GNU Collection*. Todas elas somente utilizam bibliotecas padrões, exceto uma. Ao final do capítulo apresentamos os resultados de um teste simplificado de performance dessas implementações, utilizando inteiros pequenos.

2.1 As abordagens históricas

Esta seção mostra duas antigas abordagens para a determinação de primos. A primeira abordagem é a força bruta utilizada por programadores que desejam testar a primalidade de um número de forma intuitiva ou desconhecem outro método, talvez até mesmo sendo um bom exercício para um aluno aprendendo a programar. Já a segunda abordagem é o crivo de Eratóstenes, que é um método simples e prático de atacar o problema até um certo limite numérico.

2.1.1 Força bruta

O algoritmo de força bruta consiste em dividir uma entrada n por todos os números naturais menores que ele e maiores que 1. Se alguma dessas divisões for inteira, n é

composto. Se nenhuma divisão é inteira, n é primo. Podemos fazer uma modificação para melhorar tal abordagem, modificando o limite superior sabendo que não é necessária a divisão por todos os números menores que n, mas sim todos os números menores ou iguais $\sqrt{n}$.

Como descrito no Capítulo 1, não podemos cair na armadilha de pensar que a complexidade assintótica desta simples abordagem é $O(\sqrt{n})$. Sabemos que trata-se de um algoritmo pseudo-polinomial de complexidade assintótica $O(2^{(\log n)/2})$.

A seguir temos uma simples forma de implementar este algoritmo. Infelizmente essa implementação perde a viabilidade muito rápido ao aumentarmos o valor de n.

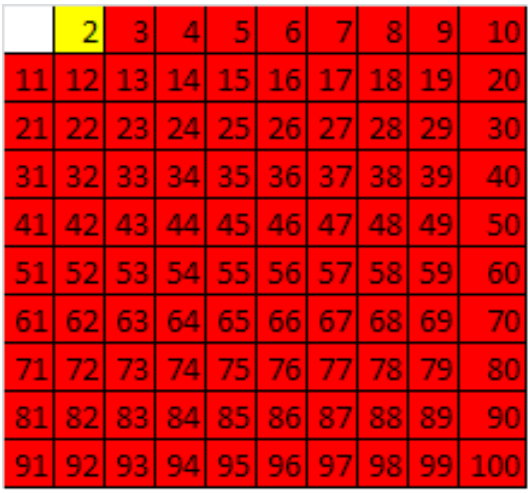
Listagem 2.1 - Determinação de primo por força bruta

```
1 bool Forca_bruta(unsigned int a){
2   for(unsigned int i = 2; i <= floor(sqrt(n)); i++)
3     if(!(n % i)) return false;
4   return true;
5 }
```

2.1.2 Crivo de Eratóstenes

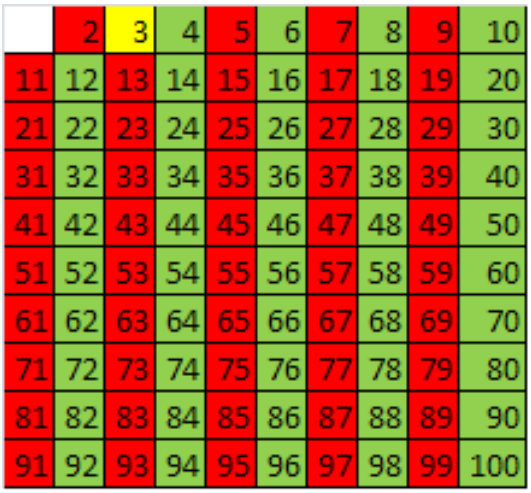
A ideia deste método é muito simples, utilizando uma folha de papel escreva todos os naturais começando de 2 até um certo limite n. Você pode fazer isso computacionalmente utilizando um simples *array*, onde cada índice deste *array* representa um número natural na folha de papel. Logo após, começando do número 2 marque os naturais seguintes sempre somando 2 ao anterior. Assim você vai marcar os números {4, 6, 8, ...} até chegar no limite da folha. Estes números marcados são os múltiplos de 2, por isso são todos números compostos. Computacionalmente basta fazer um *loop* no índice do *array* e utilizar um inteiro qualquer para marcar o natural, coloque este inteiro no índice do *array* que você está iterando a cada duas casas. Após concentre sua atenção agora no primeiro natural depois de 2 que não foi marcado, este tem de ser o número 3, como não há nenhum natural abaixo dele que divide ele na folha, então este número é primo. Digamos que o limite n é tomado como 100, o resultado do crivo será o seguinte:

Figura 1 - Inicio do crivo para 100



	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

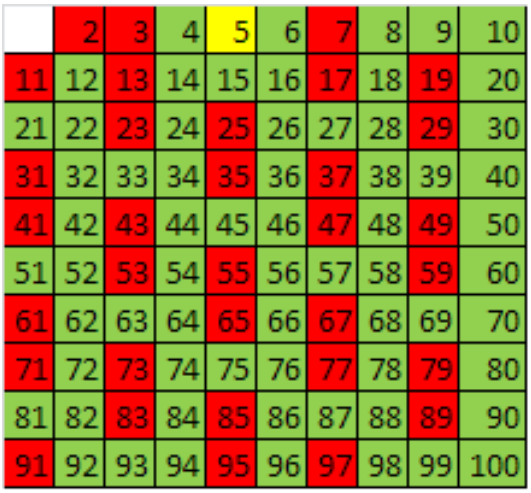
Figura 2 - Marcando os múltiplos de 2
com a cor verde



	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

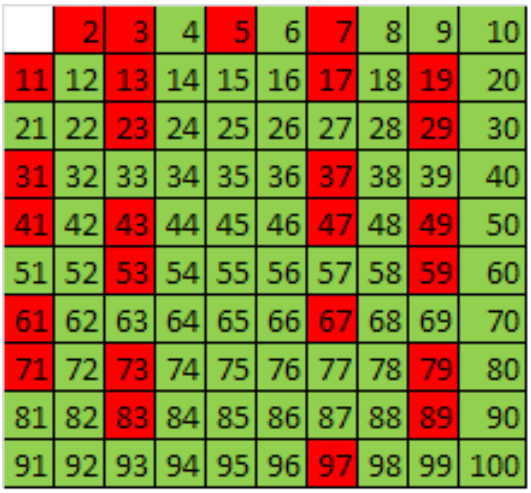
Agora nosso foco é o número 3, faça o mesmo procedimento para ele, marcando os números múltiplos de 3. Para isso basta marcar os naturais sempre somando 3 ao anterior, se o número estiver marcado, ignore-o e some 3, continue até o limite. Desta forma você irá marcar os números {6, 9, 12, ...}. Computacionalmente é o mesmo procedimento anterior, só que desta vez você deve iterar no índice do *array* somando 3 casas. Todos os novos números marcados são múltiplos de 3, por isto estes números são compostos. O próximo natural acima de 3 que não foi marcado ainda tem de ser o 5, como este natural não foi marcado pelos naturais abaixo dele, então ele é primo. Agora você pode concentrar sua atenção nele e o estado do crivo é o seguinte:

Figura 3 - Marcando os múltiplos de 3



	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

Figura 4 - Final do crivo para 100



	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

Daqui em diante você deve seguir o mesmo procedimento, marque todos os múltiplos do primo focado, depois prossiga para o próximo natural não marcado. Este é o seu próximo primo, execute o mesmo procedimento para achar os múltiplos dele até chegar no final da folha, sem números naturais adicionais para focar. No final temos um resultado que é o Crivo de Eratóstenes e pode ser visto na Figura 4, fácil de programar, porém ineficiente para aplicações que exigem determinar primos.

Para saber se um número é primo ou composto com o Crivo basta verificar se este está marcado. Computacionalmente basta percorrer o *array* pelo seu índice, se o conteúdo estiver marcado então o índice é um número composto. Uma ideia para a implementação do Crivo é substituir a marcação por um número primo que divide o índice do composto do *array*.

2.2 O pequeno teorema de Fermat

O Pequeno Teorema de *Fermat* é base para vários testes de primalidade. Foi escrito por *Pierre de Fermat* em 1640, sendo assim batizado para não ser confundido com o Último Teorema de *Fermat*. No decorrer deste capítulo vamos utiliza-lo com frequência.

Teorema de Fermat. *Para qualquer número natural a e um primo p temos que* $a^p \equiv a \mod p$.

Para representar o MDC entre dois números a e p, usaremos neste trabalho a notação (a, p). Uma forma alternativa de escrever a parte final do teorema, quando $(a, p) = 1$ é $a^{p-1} \equiv 1 \mod p$.

Uma consequência do teorema de Fermat é que, se existir um número natural $a > 1$, para um inteiro x, tal que $(a, x) = 1$ e $a^{x-1} \neq 1 \mod x$, então temos uma prova de que x é composto. O teorema conduz, então, a um algoritmo que, quando responde que um número é composto, o resultado é definitivo. Note que, nesse caso, não se obtém nenhuma indicação dos fatores primos desse número composto. Entretanto, se indicar que o número é primo o resultado pode ser falso, mesmo se usarmos inúmeras bases a para o teste. O problema é que existem bases naturais a que não identificam o composto x pelo teorema de Fermat. Se isto acontecer, x é um pseudoprimo de Fermat na base a. Vale mencionar que se x é um pseudoprimo de Fermat para todas as bases naturais a, com $1 < a < x$, então x é um número de *Carmichael*. Provou-se que existem infinitos números de *Carmichael*.

2.3 O Certificado de Pratt

Um certificado de primalidade é uma prova formal de que um certo número é realmente primo. O certificado de Pratt teve extrema importância teórica. Surgiu em 1975, com o trabalho de Vaughan Pratt, que demonstrou que o problema de determinação de primos está em *NP*, com um certificado que pode ser verificado em tempo polinomial. Foi um importante avanço para o estudo do problema, pois influenciou na criação de algoritmos probabilísticos rápidos e criou uma esperança de que os pesquisadores encontrassem uma solução polinomial para o problema.

O funcionamento do certificado descrito nesta seção segue a descrição do trabalho original de Vaughan Pratt (PRATT, 1975). Para um número primo n, o certificado de Pratt apresenta um inteiro x que seja uma raiz primitiva de n, bem como os fatores primos de n - *1* e, recursivamente, para cada um desses fatores primos, o certificado contém as mesmas informações, exceto quando o fator é 2. Devemos ainda ressaltar que a ideia do certificado de Pratt é baseada no método heurístico de Lucas-Lehmer (LEHMER, 1927)

para teste de primalidade, que consiste em checar o Pequeno Teorema de Fermat com um teste adicional:

Teste de Lucas. *Seja $n > 1$. Se a é relativamente primo com n, $a^{n-1} \equiv 1 \mod n$ e $a^{(n-1)/q} \not\equiv 1 \mod n$ para cada fator primo q de $n-1$, então n é primo.*

Antes de explicar o certificado, precisamos definir o que é uma raiz primitiva. Um inteiro x é uma raiz primitiva de n se a cardinalidade de sua ordem multiplicativa módulo n vale $\phi(n)$, onde $\phi(n)$ é a função aritmética Totiente de Euler para o valor n. A cardinalidade da ordem multiplicava para x é dada pelo tamanho do grupo cíclico formado pelas potencias da forma x^i módulo n para $i > 0$. Podemos simplificar ainda o tamanho do grupo cíclico de forma mais conveniente para a computação. Para $1 \leq x < n$ comece com $i = 1$ e execute iterações incrementando i até encontrar a forma $x^i \equiv 1 \mod n$. A quantidade de elementos obtidos é cardinalidade da ordem multiplicativa para x. Vamos exemplificar o cálculo da raiz primitiva de $n = 7$. Teríamos os seguintes conjuntos com os resultados módulo 7:

- $1 = \{1^1\}$
- $2 = \{2^1\ 2^2\ 2^3\}$
- $3 = \{3^1\ 3^2\ 3^3\ 3^4\ 3^5\ 3^6\}$
- $4 = \{4^1\ 4^2\ 4^3\}$
- $5 = \{5^1\ 5^2\ 5^3\}$
- $6 = \{6^1\ 6^2\}$

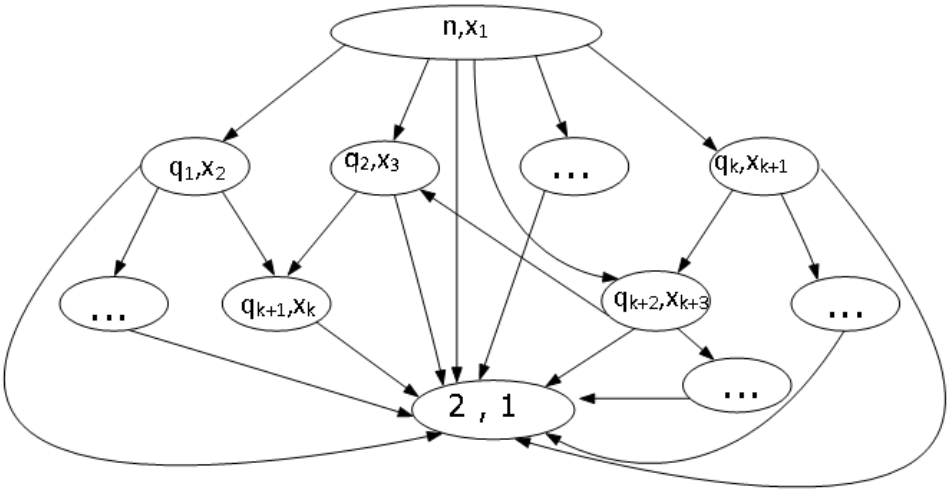
- $1 = \{1\}$
- $2 = \{2\ 4\ 1\}$
- $3 = \{3\ 2\ 6\ 4\ 5\ 1\}$
- $4 = \{4\ 2\ 1\}$
- $5 = \{5\ 3\ 1\}$
- $6 = \{6\ 1\}$

Sabemos que $\phi(7) = 6$. Então, pela tabela acima, temos que o número 3 é uma raiz primitiva de 7 pois última iteração de 3 vale $i = \phi(7) = 6$.

O certificado de Pratt pode ser representado por um conjunto de pares da forma (p, x), onde p é um inteiro provavelmente primo e x uma raiz primitiva de p. Este conjunto é formado primeiramente pelo par (n, x_1) onde n é o próprio número estudado n e x_1 sua raiz primitiva. Os outros pares são formados por todos os fatores primos q de $n-1$ com suas respectivas raízes primitivas e, recursivamente, os pares formados para provar a primalidade de cada q com suas respectivas raízes primitivas.

O autor mostra um modelo gráfico do certificado utilizando uma árvore. Preferimos mostrar um modelo equivalente, na forma de um digrafo acíclico, onde cada vértice é um par *(p, x)* do conjunto, a única fonte é o vértice (n, x_1), o único sumidouro é o vértice $(2, 1)$ e as arestas ligam um vértice (q_k, x_k) com os vértices da forma (q_r, x_r), onde q_r é um fator primo de $q_k - 1$. A Figura 5 ilustra essa representação.

Figura 5 - Um modelo generalizado do certificado de Pratt.



As seguintes regras de inferência são utilizadas para verificar o certificado:

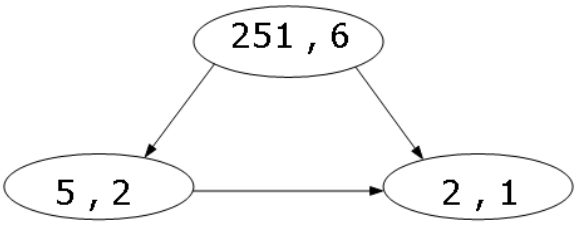
1. Axioma (x,y,1)
2. $(p, x, a) \vdash (p, x, qa)$ tal que $x^{(p-1)/q} \not\equiv 1 (\text{mod } p)$ e $q|(p-1)$
3. $(p, x, p-1) \vdash p$ tal que $x^{(p-1)} \equiv 1 (\text{mod } p)$

Nestas inferências temos que p é o número que desejamos saber se é primo, x uma raiz primitiva do número p, q um fator primo de $p-1$ e o inteiro a é um inteiro que varia de 1 a $p-1$. A segunda e a terceira regra de inferência vêm do Teste de Lucas citado anteriormente. A ideia do sistema de inferências é verificar se p atende o teste.

O método para verificar o certificado é checar se cada vértice do digrafo atende o teste, para isso deve ser feito o seguinte procedimento, começando pela fonte: cada vértice (p, x) deve atender as regras de inferência de Pratt, seus vizinhos devem constituir o conjunto de fatores primos de $p-1$ e, recursivamente, o procedimento deve ser bem sucedido para os seus vizinhos, com exceção do vértice $(2, 1)$.

Note que o vértice *(2, 1)* deve ser vizinho a todos os vértices do digrafo pois cada $p-1$ deve ser par, tendo o primo 2 em sua fatoração. Para compreender melhor o modelo, o método de verificação e o funcionamento do certificado de Pratt, apresentaremos um exemplo para o primo 251.

Figura 6 - Certificado de Pratt para o inteiro 251.



Para verificar o certificado de 251 devemos começar no vértice (251, 6). Temos $p - 1 = 250 = 2 \times 5^3$. Então os vizinhos constituem os fatores primos de 250. A verificação com as regras de inferência pode ser desta forma:

1. (251, 6, 1) Axioma;
2. (251, 6, 2) 1, R_1, $6^{125} \equiv 250 \bmod 251$; $(q = 2)$
3. (251, 6, 10) 2, R_1, $6^{25} \equiv 138 \bmod 251$; $(q = 5)$
4. (251, 6, 50) 3, R_1, $6^{25} \equiv 138 \bmod 251$; $(q = 5)$
5. (251, 6, 250) 4, R_1, $6^{25} \equiv 138 \bmod 251$; $(q = 5)$
6. (251) 5, R_2, $6^{250} \equiv 1 \bmod 251$; (Fermat)

Na linha 6 conseguimos inferir 251, isto quer dizer que podemos continuar a busca no digrafo. Então para provar a primalidade de 251 devemos provar a primalidade dos seus vizinhos recursivamente. Vamos verificar o par (5, 2): a aresta aponta para a fatoração completa do par, pois $4 = 2 \times 2$, então devemos inferir $p = 5$:

1. (5, 2, 1) Axioma;
2. (5, 2, 2) 1, R_1, $2^2 \equiv 4 \bmod 5$; $(q = 2)$
3. (5, 2, 4) 2, R_1, $2^2 \equiv 4 \bmod 5$; $(q = 2)$
4. (5) 3, R_2, $2^4 \equiv 1 \bmod 5$; (Fermat)

Como estamos verificando o digrafo recursivamente, devemos continuar do vértice (5, 2) verificando a primalidade de p do seu vizinho, que é (2, 1). Como este vértice é o sumidouro, podemos checar diretamente as regras de inferência:

1.	(2, 1, 1)	Axioma;
2.	(2)	2, R_2, $1^1 \equiv 1 \bmod 2$; (Fermat)

Assim podemos afirmar que 2 é um fator primo de 4 para o vértice anterior, então o vértice (5, 2) atende o Teste de Lucas. Quando a recursão volta para o vértice (251, 6), temos que o último vizinho é o próprio vértice (2, 1), que já foi verificado. Assim temos que a fonte também atende o teste, provando a primalidade de 251.

Para estudar a complexidade da verificação do certificado para um inteiro n, sabemos que operações de potência modular podem ser executadas com um algoritmo de complexidade assintótica $O(\log p)$ para qualquer p (no apêndice deste trabalho pode ser visto o algoritmo que computa potências modulares). Pratt mostra que o máximo de linhas necessárias para inferir n no certificado completo é $\lceil 4 \lg p \rceil$. Como qualquer inteiro $p \neq n$ de qualquer vértice do digrafo é menor que a entrada n (por ser um fator primo de $n-1$), então a busca no digrafo tem complexidade assintótica $O(\log n)$. Computacionalmente precisamos checar cada vértice uma única vez. Com a potência modular, multiplicação de inteiros ordinária e a verificação de cada vizinho para afirmar que estes formam uma fatoração de $p-1$, temos que a complexidade assintótica para verificar o certificado é $O(\log^4 n)$. Note que se utilizarmos um método de multiplicação de inteiros rápida, por exemplo *Schönhage–Strassen*, então a complexidade assintótica é ainda menor.

Antes de finalizar a seção, provemos que o certificado funciona, para isso enunciaremos o seguinte lema que pode ser visto no trabalho original de Pratt:

Lema 2.3.1. *Não existem números tal que a ordem multiplicativa seja* $(p-1) \bmod p$ *se* p *não for primo.*

Teorema 2.3.1. *O certificado de Pratt prova a primalidade de p.*

Demonstração. Assuma que todos os fatores inteiros q de p - *1* são primos provados. Se p for primo, pelo Lema 2.3.1 este possui ordem multiplicativa de grandeza $p-1$. Para $a = 1$ o axioma das regras de inferência se aplica. Então, podemos induzir que para todas as inferências até $a = p-1$ são possíveis.

Dessa forma tendo inferido todos os anteriores, devemos mostrar que p pode ser inferido. Para tanto tomaremos o termo $(p, x, p-1)$, que ocasiona o pequeno teorema de *Fermat*: $x^{(p-1)} = 1 \bmod p$, logo se p for inferido, este é primo pelo Teste de Lucas. □

2.4 Algoritmos Probabilísticos

Apesar do algoritmo *AKS* ter grande importância teórica, na prática ele não é muito utilizado. Temos preferência pelos algoritmos probabilísticos atuais, que possuem

melhor performance e que nunca respondem que um inteiro composto é primo. Mas quando responde primo, há uma probabilidade, em geral muito baixa, deste ser composto.

Antes de começarmos, de fato, a abordar os algoritmos probabilísticos é importante que entendamos que estes podem ser divididos em dois tipos: os algoritmos probabilísticos *Las Vegas* e os algoritmos probabilísticos *Monte Carlo*.

- Monte Carlo - Consistem em algoritmos probabilísticos cujo tempo de execução é determinístico, porém apresentam uma provável resposta correta com um grau de certeza tipicamente alto o suficiente.

- Las Vegas - Consistem em algoritmos probabilísticos cuja aposta não é baseada na resposta, mas sim nos recursos necessários para a computação do algoritmo. O algoritmo é determinístico e provavelmente termina, a implementação de um algoritmo *Las Vegas* pode exigir tanto tempo ou recuso computacional tal que a aplicação não finaliza.

No decorrer desse capítulo serão abordados dois algoritmos probabilísticos muito utilizados na atualidade, dentre os quais podemos destacar um que é determinístico para primos de até *64* bits e outro que fornece um certificado de primalidade que pode ser examinado rapidamente. Nesta seção serão abordados também alguns algoritmos que têm grande importância teórica, detalhando seu funcionamento e focando na implementação prática.

Vale notar que há um teste de primalidade determinístico de grande importância teórica que já foi utilizado na prática chamado *Adleman–Pomerance–Rumely* ou simplesmente *APR*, posteriormente o algoritmo foi amplamente melhorado, notavelmente por Henri Cohen e H. W. Lenstra dando origem ao teste *APR-CL*. Este teste foi construído com o intuito de evitar o uso de números aleatórios, ele tem complexidade assintótica quase polinomial e sua primeira versão foi introduzida em 1979. Este capítulo não vai detalhar o *APR*, contudo, devido a sua importância teórica o teste vale a pena ser mencionado. Para mais detalhes recomenda-se a leitura de (ADLEMAN; POMERANCE; RUMELY, 1983). Outro teste que vale mencionar é o *Lucas-Lehmer test (LLT)*, utilizado na determinação de números de *Mersenne*, que são primos na forma $2^k - 1$ onde k é um inteiro. Este teste é eficiente e determinístico, porém ele não é geral. Os maiores números primos conhecidos até agora são números de *Mersenne*, o *LLT* é o método mais rápido conhecido até agora na determinação desses números. Detalhes deste algoritmo podem ser visto em (CRANDALL; POMERANCE, 2001).

2.4.1 O Algortimo de Solovay-Strassen

Esse algoritmo não é mais utilizado em larga escala por ter sido superado pelos algoritmos de *Miller-Rabin* e posteriormente pelo Baillie-PSW, ambos detalhados neste capítulo. Contudo, o algoritmo de *Solovay-Strassen* representou um grande avanço na história da pesquisa sobre números primos, foi o primeiro teste de primalidade randômico que teve utilização em larga escala. Esse teste surgiu pouco depois da criação do sistema de criptografia *RSA* que se fundamentava na dificuldade de fatorar números grandes. Imediatamente após o seu surgimento, a criptografia *RSA* não havia sido tão utilizada devido a dificuldade de gerar números primos grandes o suficiente para que tornassem inviáveis as tentativas de quebra de segurança por métodos de fatorações até então conhecidos.

Foi nesse contexto que o algoritmo de Solovay-Strassen surgiu e, devido ao seu alto desempenho e taxa de acerto para a época, tornou-se um marco nos algoritmos de primalidade e viabilizou o uso da criptografia *RSA* que é utilizada para segurança em sistemas computacionais até hoje.

2.4.1.1 A ideia

O Algoritmo de *Solovay-Strassen* é um teste de primalidade *Monte Carlo* que se baseia no seguinte teorema:

Teorema de Euler. *Seja n um inteiro primo. Então* $\forall a \in \mathbb{Z}/n\mathbb{Z}^*$ *temos que:*
$(a/n) \times a^{(n-1)/2} \equiv 1 \bmod n$*, sendo (a/n) o Símbolo de Legendre.*

Apesar da definição original utilizar o Símbolo de *Legendre*, estaremos utilizando o símbolo de Jacobi, que nada mais é do que uma generalização do Símbolo de *Legendre*, que não requer que n seja primo. Chamaremos o Símbolo de Jacobi de (a/n).

Temos as seguintes propriedades para a, b e n inteiros:

1. Se MDC(a, n) $\neq 1$, então (a/n) = 0.
2. (ab/n) = (a/n) $\times$ (b/n)
3. Se a é par temos (a/n) = (2/n)$\times$((a/2)/n)
4. (a/n) = (n/a) $\times$ $(-1^{(a-1)(n-1)/4})$ com a, n ímpares.
5. Se $a \equiv b \bmod n$ então (a/n) = (b/n).

Considere também seus decorrentes desenvolvimentos e fatos:

1. Se MDC(a, n) $\neq 1$, então (a/n) = 0, caso contrário $\pm$ 1.

Demonstração. Parte da própria definição descrida no item 1 anterior □

2. $(ab/n) = (-a/n) = (a/n)(-1/n)$ e $(-1/n) = (-1)^{(n-1)/2} = \begin{cases} 1 & \text{se} \quad n \equiv 1 \bmod 4 \\ -1 & \text{se} \quad n \equiv 3 \bmod 4 \end{cases}$

Demonstração. Para o item 2 tomemos um m na forma $4k+1$ que desejamos provar como verdadeira. Aplicando-a na fórmula $(-1)^{(n-1)/2}$ temos $(-1)^{(4k)/2} = (-1)^{2k}$ que é garantidamente 1 para todo k. Analogamente podemos tomar um $n = 4k + 3$, dessa forma $(-1)^{(n-1)/2}$ □

3. $(ab/n) = (2a/n) = (a/n)(2/n)$ e
$(2/n) = (-1)^{(n^2-1)/8} = \begin{cases} 1 & \text{se} \quad n \equiv 1 \bmod 8 \quad \text{ou} \quad n \equiv 7 \bmod 8 \\ -1 & \text{se} \quad n \equiv 3 \bmod 8 \quad \text{ou} \quad n \equiv 5 \bmod 8 \end{cases}$

Demonstração. Para o item 3 tomemos um m na forma $8k + 1$ que desejamos provar como verdadeira. Aplicando-a na fórmula $(-1)^{(n^2-1)/8}$ temos $(-1)^{((8k+1)^2-1)/8} = (-1)^{(8\times8\times k^2+8\times2\times k+1-1)/8}$ que é garantidamente 1 para todo k. As demais propriedades são demonstradas de forma análoga. □

4. $(a/n) = (n/a) = (-1)^{\frac{(a-1)(n-1)}{4}} = \begin{cases} 1 & \text{caso} \quad n \equiv 1 \bmod 4 \quad \text{ou} \quad a \equiv 1 \bmod 4 \\ -1 & \text{caso} \quad n \equiv 3 \bmod 4 \quad \text{e} \quad a \equiv 3 \bmod 4 \end{cases}$.

Demonstração. Prova análoga aos itens 2 e 3. □

O algoritmo para o cálculo do símbolo de Jacobi de maneira intuitiva, bem como seus detalhes, estão presentes no apêndice deste projeto.

O teste de primalidade *Monte Carlo* de *Solovay-Strassen* consiste de k iterações onde a cada iteração escolhemos um a inteiro aleatoriamente tal que $a \in \mathbb{Z}/n\mathbb{Z}^*$ e testamos se este inteiro atende o Teorema de Euler descrito no começo desta seção. Para n composto sabe-se que existem até no máximo $\frac{n}{2}$ números inteiros que satisfazem a condição do teorema. Caso, em alguma iteração k este teste falhe, temos uma prova de que n é composto; caso o teste passe em todas as iterações temos que n é provavelmente primo com uma probabilidade inversamente proporcional a k de de erro. Escolhendo k suficientemente grande temos que esta probabilidade fica suficientemente pequena.

No algoritmo do teste em pseudocódigo, a seguir, é utilizada a função *pow_mod()* de potência modular. O objetivo desta função é calcular o resultado de $a^b \mod c$. Esta função será utilizada também na computação de outros testes neste projeto e os detalhes do algoritmo correspondente pode ser visto no apêndice.

Vale notar também que a implementação do símbolo de Jacobi apresentada a seguir não é implementada da forma mais eficiente. Uma implementação com melhor performance pode ser encontrada na biblioteca *GMP*. Outra implementação é a que pode ser vista em (BRENT; ZIMMERMANN, 2010), que é mais eficiente que a do *GMP* para números acima de 10000 dígitos apresentando complexidade $O(M(n)\log(n))$ onde *M(n)* é o tempo para computar a multiplicação de um inteiro com n *bits*.

2.4.1.2 O Algoritmo

Algoritmo 1 - Teste de primalidade Solovay-Strassen.

DOCUMENTAÇÃO

TÍTULO

Solovay-Strassen

PROPÓSITO

Teste de primalidade.

MÉTODO

Seleção de um número randômico para verificar o Teroema de Euler

ENTRADAS

p: Número a ser testado

k: Número de iterações do teste

SAÍDAS

Responde Provavelmente primo ou Composto

OBSERVAÇÕES, RESTRIÇÕES, REQUISITOS

A entrada n deverá ser maior que 2.

ALGORITMO SOLOVAY-STRASSEN

declarar a, jac, i, k **numérico**
1. **se** $((p \bmod 2 = 0))$, **então**
2. | **retornar** "Composto"
3. **fim se**
4. **para** i **de** 1 **até** k , **fazer**
5. | $a \leftarrow numero\_randomico(1, p-1)$
6. | $jac \leftarrow p + Calcula\_Jacobiano(a, p) \bmod p$
7. | **se** $((jac = 0)$ ou $((jac \neq pow\_mod((a, (p-1)/2, p)))$, **então**
8. | | **retornar** "Composto"
9. | **fim se**
10. **fim para**
11. **retornar** "Provavelmente primo"

2.4.1.3 Implementação

Inicialmente apresentaremos as implementações das funções auxiliares. A primeira delas é a implementação da potencia modular, que será utilizada no restante do trabalho.

Listagem 2.2 - Potência modular

```
inline unsigned int pow_mod(const unsigned int &a, const unsigned int  &b,
    const unsigned int &n){
    unsigned int base, power, resp;
    resp = 1;
    base = a;
    power = b;
    while (power){
        if (power & 1)
            resp = (resp*base) % n;
        power >>= 1;
        base = base % n;
        base = (base*base) % n;
    }
    return resp;
}
```

Os detalhes desta implementação serão abordados no Capítulo 4. A segunda função auxiliar é a que computa o calculo do Símbolo de Jacobi.

Listagem 2.3 - Símbolo de Jacobi

```
int Jacobi(long long int a,long long int n){
    if(!a) return 0; // (0/n) = 0
    if(a==1) return 1; // (1/n) = 1
    short resp = 1;
    int v0;
    while(a){
        if(!(a&1)){
            v0 = __builtin_ctz (a);
            a >>= v0;
            if(n % 8==3 || n % 8== 5) if(v0&1) resp=-resp;
        }
        long long int aux = a; a = n; n = aux;
        if(a%4==3 && n%4==3) resp=-resp;
        a = a % n;
    }
    if(n==1) return resp;
    return 0;
}
```

O algoritmo que computa o símbolo de Jacobi pode ser implementado utilizando um método de divisões sucessivas, conforme mostrado no apêndice deste trabalho. Entre-

tanto, optamos por usar uma versão equivalente mais eficiente, baseada na representação binária de a. Para isso podemos utilizar a função interna *builtin_ctz()* do compilador de *C++* do *GNU Compiler Collection* para contar o número de zeros à direita de a em sua forma binária. Com isso, removermos todos os zeros à direita através da divisão $a/(2^{v0})$ onde $v_0 = builtinz\_ctz()$. O mesmo critério serve para o sinal que *resp* deverá possuir, pois sabemos que $(-1)^{v_0}$ será 1 caso v_0 seja par e -1 caso contrário.

Apresentamos, a seguir, o código que implementa o teste de primalidade *Monte Carlo* de *Solovay-Strassen*.

Listagem 2.4 - Teste de primalidade Solovay-Strassen

```
bool solovay_strassen(const long long int & p, long &k){
    if (!(p&1)) return false;
    long long int a, jac;
    int i;
    random_device rd; // declarando
    uniform_int_distribution<int> uni; // declarando
    mt19937 rgn(rd()); // inicializando
    for(i = 0;i < k; i++){
        a = uni(rgn); // random inicializado
        a = 1 + (a % (p-1)); //Escolhe [1, p-1]
        jac=(p+Jacobi(a,p))%p;
        if(!jac || pow_mod(a,(p-1)/2,p)!=jac){
            return false;
        }
    }
    return true;
}
```

Note que é utilizada a biblioteca *random* para a geração de números aleatórios do *C++11*. O código inteiro com um exemplo de *int_main* pode ser visto no anexo.

A análise do decrescimento da taxa de erro pode ser vista em (MONIER, 1980), sabemos que a probabilidade de erro é de $(\frac{1}{2})^k$, onde k representa o números de iterações executadas pelo algoritmo. Note que para $k = 10$, a probabilidade de p ser primo é de aproximadamente 99,9%.

O autor do artigo também escreve uma abordagem mais profunda a respeito da fundamentação matemática do teste e de suas capacidades, bem como a demonstração da complexidade assintótica do mesmo.

2.4.2 O Algortimo de Miller-Rabin

Desenvolvido por Gary Miller, publicado em (MILLER, 1975) e posteriormente melhorado por Michael Rabin em (DIVISION; RABIN, 1976), este algoritmo tornou-se um

importante marco nos testes probabilísticos de primalidade e contribuiu para as questões de seguranças em geração de chaves na criptografia. Observe que os números primos descobertos por esse teste são tão grandes que inviabilizam as opções de tabelamentos de números primos e métodos simples como divisões sucessivas.

O teste continua sendo utilizado até hoje devido a simplicidade da implementação, boa performance e baixa probabilidade de erro. Quase todas as bibliotecas que fornecem boas ferramentas de Teoria dos Números para diversas linguagens implementam este teste. Vale notar que o teste original de Miller é determinístico, mas baseado na hipótese não provada de Riemann. O teste passou a ser probabilístico depois que foi modificado por Rabin.

2.4.2.1 A ideia

O algoritmo *Miller-Rabin* é um teste de primalidade *Monte Carlo* baseado em duas ideias principais:

Pequeno Teorema de Fermat. $a^{p-1} \equiv 1 \bmod p$ *com (a, p) = 1 e p primo*

Teorema 2.3.2.1. *Se* $x^{p'} \not\equiv 1 \bmod p$ *e* $a^{p'2^s} \not\equiv -1 \bmod p$ *para todo* $0 \leq s < v_0$ *tal que* $p - 1 = 2^{v_0} \times p'$ *então p não é primo.*

A ideia do segundo teorema vem do desejo de extrair sucessivamente raízes quadradas de a^{p-1} do primeiro teorema até que este seja reduzido para a forma $1 \mod p$ ou $-1 \mod p$.

Assim como no algoritmo *Solovay-Strassen*, a ideia é realizar k iterações tal que a cada iteração temos um inteiro $a \in \mathbb{Z}/n\mathbb{Z}^*$ aleatório que é submetido ao teste do segundo teorema. Para p composto existem muitos inteiros a onde o teorema falha, gerando uma prova de que p é realmente composto. Tais números são chamados de testemunhas. Existe uma probabilidade de não se encontrar uma testemunha nas k iterações para um número p composto, resultando a falsa declaração dele ser primo. Os inteiros a que não são testemunhas de p composto são chamados de fortes mentirosos, e são uma minoria. Com isto, o retorno do algoritmo é que ou p é garantidamente composto ou é provavelmente primo com uma probabilidade de $(\frac{1}{4})^k$ de erro. Dessa forma, com somente 5 iterações, temos uma probabilidade próxima de 99,9% de uma resposta *provavelmente primo* ser na verdade *primo*.

2.4.2.2 O Algoritmo

Algoritmo 2 - Teste de primalidade Miller-Rabin.

DOCUMENTAÇÃO

TÍTULO

Miller-Rabin

PROPÓSITO

Teste de primalidade.

MÉTODO

Seleção de um número randômico para servir de testemunha

ENTRADAS

p: Número a ser testado

k: Número de iterações do teste

SAÍDAS

Responde Provavelmente Primo ou Composto

OBSERVAÇÕES, RESTRIÇÕES, REQUISITOS

A entrada p deverá ser maior que 2.

ALGORITMO MILLER-RABIN

declarar a, p1, v0, i, r, s, resto, k **numérico**
se $(p \bmod 2 \neq 0)$, **então**
  $p1 \leftarrow p - 1$
  $v0 \leftarrow conta\_zeros\_direita(p1)$
  $p2 \leftarrow p1/2^{v0}$
  para i **de** 1 **até** k , **fazer**
    $a \leftarrow numero\_randomico(1, p - 1)$
    $s \leftarrow p2$
    $resto \leftarrow pot\_mod(a, s, p)$
    enquanto $((s \neq p1)$, $(resto \neq p1)$ e $(resto \neq 1))$, **fazer**
      $resto \leftarrow (resto^2) \bmod p$
      $s \leftarrow 2 * s$
    fim enquanto
    se $((resto \neq p1)$ e $((s \bmod 2) = 0))$, **então**
      retornar "Composto"
    fim se
  fim para
senão
  retornar "Composto"
fim se
retornar "Provavelmente Primo"

2.4.2.3 Implementação

Listagem 2.5 - Teste de primalidade Miller-Rabin

```
1  bool miller_rabin (const unsigned int & p, long &k) {
2    if (p % 2 == 0 || p == 1) return false;
3    unsigned int p1 = p-1,a ,s , mod, v0 = __builtin_ctz (p-1), p2;
4    random_device rd;
5    uniform_int_distribution<int> uni;
6    mt19937 rgn(rd());
7    p2 = p1 >> v0;
8    for(int i=0;i < k;i++){
9      a = uni(rgn); a = 1 + (a % (p1));
10     s = p2;
11     mod = pow_mod (a,s,p);
12     while (s!=(p1) && mod!=1 && mod!=(p1)){
13       mod = (mod * mod) % p;
14       s *= 2;
15     }
16     if (mod!=p1 && !(s&1)) return false;
17   }
18   return true;
19 }
```

A implementação segue fielmente o algoritmo. As variáveis declaradas na linha 3 estão com a mesma nomenclatura das variáveis do algoritmo. A linha 2 filtra os termos pares e impede que o valor 1 seja computado. Como sabemos o número de zeros à direita de um número corresponde ao número de multiplicações por 2 que o mesmo possui em seus fatores, dessa forma encontrar o valor de v_0 é simples, podendo ser facilmente obtido utilizando divisões sucessivas ou deslocamentos na representação binária. Esta parte foi escrita da mesma forma que a da implementação do teste de *Solovay-Strassen* na seção anterior, utilizando a função *builtin_ctz*.

A probabilidade de um número composto passar o teste como provavelmente primo está diretamente relacionada ao número de iterações k. Na linha 9 sorteamos um número inteiro randômico a que pode servir como testemunha. Posteriormente é calculada uma potencia modular na linha 11 que crescerá com a potência de 2 até que seja atingido o máximo de $p-1$. Foi utilizada a mesma implementação da função *pow_mod* mostrada na seção do algoritmo de *Solovay-Strassen*.

Como vimos na ideia do algoritmo devemos encontrar um número tal que o resto seja igual a 1 ou -1 $\mod p$ e diferente de $p-1$ em todo o intervalo até v_0 com o próprio não incluso. Caso tal número seja encontrado temos uma testemunha para provar que p é composto. É importante que lembremos que é necessário um critério de parada para caso ele chegue ao valor $p-1$. Dentro do *loop* da linha 12 multiplicamos o expoente s por 2

e consequentemente elevamos o resto ao quadrado, isso decorre de $a^{2p} = a^{p+p} = a^p \times a^p$. Por fim, devemos conferir se o valor que entrou no *loop* terminou com um valor diferente de $p - 1$. É simples vermos que o valor $p - 1$ é conferido com uma variação do primeiro teorema da ideia, o Pequeno Teorema de Fermat, essa variação é $a^p \equiv p - 1 \bmod p$.

Para a geração de números aleatórios utilizamos uma biblioteca do *C++11*. O código completo pode ser visto no anexo, junto com um exemplo de função principal.

Para maiores informações sobre o fundamento matemático deste teste junto com as demonstrações dos teoremas descritos nesta seção, recomendamos a leitura de (MONIER, 1980) e (DONADELLI, 20XX).

2.4.3 Teste de Lucas

O Teste de Lucas que analisaremos nesta seção não será o que inspirou a ideia de *Every Prime has a Succint certificate* de (PRATT, 1975), mas sim aquele relacionado com a sequencia de Lucas, descrito em Lucas Pseudoprimes em (BAILLIE; WAGSTAFF, 1980). Apesar de não ser extremamente útil sozinho, esse teste tornou-se importante pois, ao ser combinado com o teste de *Miller-Rabin* torna-se uma ferramenta poderosa que atualmente conhecemos como Algoritmo de *Baillie-PSW* a ser detalhado em uma seção posterior.

2.4.3.1 A ideia

Considere os inteiros D, P e Q tal que $D = P^2 - 4Q \neq 0$ e $P > 0$. Agora tomemos as sequencias de Lucas U_k e V_k definidas pelas equações a seguir:

$U_0 = 0 \qquad U_1 = 1$

$V_0 = 2 \qquad V_1 = P$

$U_k = PU_{k-1} - QU_{k-2},$

$V_k = PV_{k-1} - QV_{k-2}$

Usamos a notação $U_k(P, Q)$ e $V_k(P, Q)$ para mostrar a dependência dessas sequências por P e Q. Usamos D para denotar o discriminante da sequência. Agora considere $\delta(n) = n - (D/n)$ onde n é um inteiro positivo ímpar e (D/n) o símbolo de Jacobi.

Se n é composto, $(n, Q) = 1$ e a equação $U_{\delta(n)} \equiv 0 \mod n$ é atendida, então n é um pseudoprimo de Lucas.

A equação acima é conhecida como Teste de Lucas. Podemos derivar desta equação

um teste de primalidade onde todo inteiro n positivo ímpar que for rejeitado pelo teste é composto e, se for aceito, é provavelmente primo.

A ideia do algoritmo é tomar uma sequencia S da forma $S_n = (-1)^n(2n + 5)$ e descobrir o primeiro termo da sequencia tal que o Símbolo de Jacobi (S_k/n) seja -1. Tal S_k será nosso discriminante D da sequencia de Lucas e, com base nele, obteremos as sequencias $U_k(P, Q)$ e $V_k(P, Q)$

Sabemos, também, que para quaisquer sequências de Lucas *U(P, Q)* e *V(P, Q)*, as seguintes relações são válidas:

1. $U_0(P, Q) = 0$, $V_0(P, Q) = 2$
2. $U_1(P, Q) = 1$, $V_1(P, Q) = P$
3. $U_{2k}(P, Q) = U_k(P, Q) \times V_k(P, Q)$
4. $V_{2k}(P, Q) = V_k(P, Q) \times V_k(P, Q) - 2 \times Q^k$
5. $D = P^2 - 4Q$, D é chamado de discriminante da sequência de Lucas
6. $U_{m+n}((P, Q) = U_m(P, Q) \times V_n(P, Q) + U_n(P, Q) \times V_m(P, Q)) \times \frac{1}{2}$
7. $V_{m+n}(P, Q) = (V_m(P, Q) \times V_n(P, Q) + D \times U_m(P, Q) \times U_n(P, Q)) \times \frac{1}{2}$

Com bases nessas propriedades podemos elaborar o algoritmo. O primeiro passo consiste da utilização do símbolo de Jacobi sobre a sequencia S para encontrar o discriminante D, através de iterações sobre os termos da sequência, variando até achar um termo tal que o símbolo de Jacobi seja igual a -1.

Após obter o discriminante, é simples obter o valor de Q, basta que fixemos $P = 1$, sabendo que $D = P^2 - 4Q$, para $P = 1$, temos que $Q = (1 - D)/4$. Para obter as sequências *U(P, Q)* e *V(P, Q)*, fixamos $U_0 = 0$, $U_1 = 1$, $V_0 = 2$ e $V_1 = P$ como descrito na definição da sequência de Lucas e iteramos sobre $\frac{p+1}{2}$ usando as propriedades mostradas até chegar ao ponto em que podemos utilizar a equação que consiste no Teste de Lucas para determinar se n é composto ou provavelmente primo.

2.4.3.2 O Algoritmo

Algoritmo 3 - Teste de primalidade de Lucas.

DOCUMENTAÇÃO

TÍTULO

Teste de primalidade de Lucas

PROPÓSITO

Teste de primalidade.

MÉTODO

Aplicação de propriedades das sequencias de Lucas para verificar primalidade

ENTRADAS

n: Número ímpar a ser testado

SAÍDAS

Responde Provavelmente Primo ou Composto

ALGORITMO TESTE DE LUCAS

declarar D, P, Q, U, V, U_2, V_2, Q_2, k, U_{aux} **numérico**

1. $D \leftarrow Encontra\_D(n)$
2. $P \leftarrow 1$
3. $Q \leftarrow (1 - D)/4$
4. $U \leftarrow 0$
5. $U_2 \leftarrow 1$
6. $V \leftarrow 2$
7. $V_2 \leftarrow P$
8. $k \leftarrow (n + 1)/2$
9. **enquanto** $(k \neq 0)$, **fazer**
10. | $U_2 \leftarrow U_2 \times V_2$
11. | $V_2 \leftarrow V_2 \times V_2 - 2 \times Q$
12. | **se** $(k \bmod 2 \neq 0)$, **então**
13. | | $U_{AUX} \leftarrow U$
14. | | $U \leftarrow U_2 \times V + U \times V_2$
15. | | **se** $(U \bmod 2 \neq 0)$, **então**
16. | | | $U \leftarrow U + n$
17. | | **fim se**
18. | | $V \leftarrow V_2 \times V + U_2 + U_{aux} \times D$
19. | | **se** $(V \bmod 2 \neq 0)$, **então**
20. | | | $V \leftarrow V + n$
21. | | **fim se**
22. | | $U \leftarrow U/2$
23. | | $V \leftarrow V/2$
24. | | $U \leftarrow U \bmod n$
25. | | $V \leftarrow V \bmod n$
26. | **fim se**

Algoritmo 3 - Teste de primalidade de Lucas (continuação)

— *continuação* —
27. | $Q \leftarrow Q \times Q$
28. | $k \leftarrow k/2$
29. **fim enquanto**
30. **se** $(U \neq 0)$, **então**
31. | **retornar** "Composto"
32. **fim se**
33. **retornar** "Provavelmente Primo"

2.4.3.3 Implementação

A implementação do Símbolo de Jacobi presente na seção do algoritmo de *Solovay-Strassen* não leva em consideração a existência de entradas negativas. Porém, nesse algoritmo torna-se necessário o uso de uma versão que considere essas entradas. Para tanto, usaremos uma propriedade do símbolo de Jacobi mostrada na seção de *Solovay-Strassen*. Segue a nova implementação do símbolo de Jacobi que será utilizada no restante deste capítulo:

Listagem 2.6 - Símbolo de Jacobi versão 2

```
inline int Jacobi(long long int a,long long int n){
    if(!a) return 0;
    if(a==1) return 1;
    int resp = 1, v0;
    if(a<0){
        a = -a;
        if(n % 4==3) resp = -resp;
    }
    while(a){
        if(!(a&1)){
            v0 = __builtin_ctz (a);
            a >>= v0;
            if(n % 8 == 3|| n % 8 == 5) if(v0&1) resp= -resp;
        }
        long long int aux = a;
        a = n;
        n = aux;
        if(a % 4 == 3 && n % 4 == 3) resp = -resp;
        a = a % n;
    }
    if(n == 1) return resp;
    return 0;
}
```

A seguir será apresentado o código que implementa o Teste de primalidade de Lucas. Vale a pena ressaltar que existem pequenas diferenças na implementação do código ao compararmos com o algoritmo como foi apresentado. Tais diferenças serão explicadas posteriormente.

Listagem 2.7 - Teste de primalidade Lucas

```
bool lucas (long long int n){
    long long int D, ud = 5, P, Q, U, V, U2, V2, aux, uaux, k;
    short sinal = 1, jac;
    while (1){
        D = ud * sinal;
        jac = Jacobi(d, n);
        if (!jac && ud!=n) return false;
        if (jac == -1)     break;
        sinal = -sinal;
        ud+=2;
    }
    P = 1;
    Q = (1-D)/4;
    U = 0; V = 2;
    U2 = 1; V2 = P;
    k = (n+1)/2;
    while (k){
        U2 = (n + (U2*V2)) % n;
        V2 = (n + (V2 * V2 - Q - Q)) % n;
        if (k&1){
            uaux = U;
            U = U2 * V + U * V2;
            if ((U&1))
              U += n;
            U/=2;
            V = V2*V + U2 * uaux * D;
            if ((V&1))
              V += n;
            V /= 2;
            while (U<0)  U=(U+n);
            while (V<0)  V=(V+n);
            U %= n;
            V %= n;
        }
        Q = (n + (Q * Q)) % n;
        k /= 2;
    }
    if (U) return false;
    return true;
}
```

Nas linhas de 4 até 10, temos a implementação de *Encontra_D()* do algoritmo. Esta função consiste em varrer os termos da sequência S em busca do discriminante D. As linhas 12 até 16 implementam as váriaveis fixas da definição da sequência de Lucas. Na linha 17 é simples ver que só devemos avaliar os *bits* do auxiliar k enquanto $k/2$ for diferente de 0. Também é simples ver que divisões sucessivas por 2 eliminam o ultimo *bit* da representação binária do inteiro que pode ser verificado como 0 ou 1 através da operação modular com 2.

As linhas 18 e 19 correspondem à aplicação das propriedade 3 e 4 presentes na seção anterior. Note que o expoente de Q na linha 35 é dobrado a cada iteração conforme sua equivalente no algoritmo. Conseguimos ver dessa forma que as variáveis U_2 e V_2 são da forma exponencial 2^a e que, no algoritmo, não há a operação de modularidade nessas linhas. Já na implementação nas respectivas linhas 18, 19 e 35 existe uma operação de modularidade com a entrada. Elas são executadas para assegurar que as váriaveis não sofrerão *overflow*.

Se o *bit* da representação binária do auxiliar k for 1 devemos adicionar o índice. Para tanto devemos aplicar as propriedades 6 e 7 da sequência de Lucas. É necessário uma variável auxiliar U_{aux} para guardar o valor inicial de U para que seja utilizado no calculo de V.

As clausulas condicionais das linhas 23 e 27 podem parecer estranhas a uma primeira observação. Contudo, nada mais é do que uma forma de manter a congruência módulo n sem ocasionar erros devido ao truncamento nas linhas 25 e 29. Por fim, a linha 38 computa se $U_n + 1 \equiv 0 \bmod n$ verificando o Teste de Lucas. Este resultado vai nos dizer se n é provavelmente primo ou composto. Como já citado anteriormente, caso n composto seja declarado provavelmente primo então temos que n é um pseudoprimo de Lucas.

Podemos notar a existência de dois *loops* que não estavam presentes no algoritmo nas linhas 30 e 31. Eles existem para contornar os casos em que U e V têm o seu resultado negativo na congruência módulo n, caso que é tratado diretamente em algumas linguagens de programação. Contudo, *C++* não é uma delas. É possível substituir cada um desses *loops* por uma clausula condicional, caso a performance seja um critério realmente importante.

Esse algoritmo não possui decrescimento em sua taxa de erro. Entretanto, é possível elaborar um algoritmo que implementa testes adicionais com base nesse algoritmo. Tal teste é chamado de *Strong Lucas Test*, que não será mostrado nesse projeto, pois é muito complicado. Ele pode ser encontrado em (BAILLIE; WAGSTAFF, 1980). A implementação completa do Teste de Lucas pode ser encontrada na parte de anexos no final do projeto com um exemplo de *int_main()* e junto com todas as implementações feitas durante todo este trabalho.

2.4.4 O algoritmo ECPP

Obter fatores primos não é uma tarefa fácil para números grandes, em geral. Um algoritmo para fatoração de inteiros com complexidade polinomial em relação ao tamanho de n não é conhecido. Para superar esse problema, Shafi Goldwasser e Joe Kilian, apresentaram em 1986 um certificado de primalidade com inspiração no certificado de Pratt que trabalha com grupos elípticos módulo p ao invés de trabalhar com o anel $\mathbb{Z}_n$, que pode ser verificado em complexidade assintótica $O(\log^4 n)$. Junto com o certificado no mesmo trabalho, temos também um teste de primalidade *Las Vegas* que gera o certificado utilizando curvas elípticas (GOLDWASSER; KILIAN, 1986). Os testes de primalidade deste tipo ficaram conhecidos como *ECPP* (Elliptic Curve Primality Proving). Atkin A. O. L. e François Morain utilizam o mesmo certificado na versão de Atkin deste teste, com a mesma ideia do teste *Goldwasser-Kilian* (ATKIN; MORAIN, 1993). Esta seção vai descrever como funciona o certificado de *Atkin-Goldwasser-Kilian-Morain*, o algoritmo de *Goldwasser-Kilian*, o algoritmo *ECPP* de Atkin e vai mostrar uma implementação da versão de Atkin do teste de primalidade *Las Vegas ECPP*.

2.4.5 O Certificado Atkin-Goldwasser-Kilian-Morain

O conceito inédito de fatoração de inteiros utilizando curvas elípticas desenvolvido por Lenstra em 1985, que pode ser visto em (JR, 1987), foi rapidamente utilizado para o problema de determinação de primos por Goldwasser e Kilian. No mesmo ano Atkin escreveu um algoritmo de primalidade *Las Vegas* utilizando curvas elípticas de provavelmente melhor performance com base nas mesmas ideias do algoritmo *Goldwasser–Kilian*, este algoritmo foi amplamente melhorado por várias pessoas, particularmente Morain. Atkin e Morain se reuniram pessoalmente para publicar o algoritmo de Atkin em 1993, que gera e verifica o mesmo certificado. Por causa disto, o certificado ficou conhecido como *Atkin–Goldwasser–Kilian–Morain certificate*.

Para provar que n é primo neste sistema, é necessário trabalhar com curvas fornecidas por equações do tipo $y^2 = x^3 + ax + b$ onde $a, b \in \mathbb{Z}/n\mathbb{Z}$. O problema é que não podemos garantir que $\mathbb{Z}/n\mathbb{Z}$ é um corpo, pois não sabemos se n é primo. Então precisamos utilizar a definição de Lenstra para curvas elípticas:

Definição 2.4.1. *Seja o anel $\mathbb{Z}/n\mathbb{Z}$ onde n é relativamente primo com 2 e 3. $P^2(\mathbb{Z}/n\mathbb{Z})$ o plano projetivo em $\mathbb{Z}/n\mathbb{Z}$. Denotando $(x : y : z)$ como a classe de equivalência contendo (x, y, z), isto é, o conjunto de classes $(x : y : z)$ tal que o ideal (x, y, z) é o anel $\mathbb{Z}/n\mathbb{Z}$, ou seja, a igualdade $(x : y : z) = (x_i, y_i, z_i)$ implica que $\exists k$ tal que $k \in \mathbb{Z}/n\mathbb{Z}^*$, $x = kx_i$, $y = ky_i$ e $z = kz_i$.*

A curva elíptica é um par $E = (a, b)$ tal que escrevemos $E(a, b)$ com elementos de

$\mathbb{Z}/n\mathbb{Z}$. O discriminante Δ_E da curva elíptica E, é da forma $\Delta_E = -16(4a^3 + 27b^2) \neq 0$. O conjunto de pontos de E sobre $\mathbb{Z}/n\mathbb{Z}$ é:

$$E(\mathbb{Z}/n\mathbb{Z}) = (x : y : z) \in P^2(\mathbb{Z}/n\mathbb{Z}), \quad y^2z = x^3 + axz^2 + bz^3$$

Com isto, podemos definir $E(\mathbb{Z}/n\mathbb{Z})$ como um grupo abeliano com a operação de adição utilizando o método *tangent-and-chord* para efetuar a adição de dois pontos. A descrição deste método conhecido está no Capítulo 3 de (GOLDWASSER; KILIAN, 1986). Há um ponto especial em E chamado de ponto no infinito descrito como $I = (0 : 1 : 0)$, que é o elemento zero do grupo. Observe que há pontos onde $z = 0$ que fazem parte da curva e que são diferentes de I, por exemplo, se $\exists k$ tal que $k^2 = n$ então (k : 1 : 0) é um desses pontos. Durante todo o desenvolvimento deste capítulo, este trabalho vai utilizar a definição acima para curvas elípticas. O certificado de *Atkin–Goldwasser–Kilian–Morain* vem do seguinte teorema de Goldwasser e Kilian:

Teorema 2.4.1. *Para qualquer inteiro n relativamente primo com 2 e 3, E(a, b) uma curva elíptica em $\mathbb{Z}/n\mathbb{Z}$, P um ponto tal que $P \in E(\mathbb{Z}/n\mathbb{Z})$ e dois inteiros m e q tal que $q|m$. Considere as seguintes condições:*

- $\frac{m}{q}P = (x : y : z) = (x_q : y_q : z_q)$, *$z_q$ relativamente primo com n.*
- $mP = I$
- $q > (\sqrt[4]{n} + 1)^2$

Se estas condições forem atendidas, podemos realizar a seguinte implicação:

Se q é primo então n é primo.

A prova do teorema acima pode ser vista no trabalho original de *Goldwasser-Kilian* citado anteriormente. Para obter o certificado deste teorema, os algoritmos de *Goldwasser-Kilian* e *Atkin* emitem um conjunto com a curva *E(a, b)*, um ponto P desta curva, o inteiro m e um inteiro provavelmente primo q_1, também com as mesmas informações recursivamente para provar a primalidade deste q_1. O algoritmo termina o conjunto de parâmetros que formam o certificado com um q_k suficientemente pequeno que todos sabem que é primo. Goldwasser e Kilian demonstram que este certificado pode ser verificado em $O(\log^4 n)$.

Na seção a seguir detalharemos a versão do *ECPP* de Goldwasser e Kilian, mas o foco da seção é o *ECPP* de Atkin, que foi melhorada por diversos colaboradores, notavelmente Morain. Será apresentada uma implementação do *ECPP* de Atkin utilizando uma biblioteca rápida que fornece ferramentas para lidar com problemas da área de Teoria dos Números. A técnica de utilizar curvas elípticas em testes de primalidade é amplamente

utilizada na prática e está entre as técnicas mais rápidas utilizadas para prova de primalidade hoje em dia. Tenha em mente que o algoritmo probabilístico abaixo é do tipo *Las Vegas*, diferente de todos os outros mostrados neste capítulo.

2.4.5.1 A ideia

A ideia geral do algoritmo de *Goldwasser-Kilian* é fornecer um conjunto de parâmetros que atendam o teorema de *Goldwasser-Kilian* mostrado anteriormente para um inteiro n. Logo após, o algoritmo gera recursivamente o mesmo conjunto para provar a primalidade de q. No final temos o certificado de *Atkin-Goldwasser-Kilian-Morain* para o inteiro de entrada n. Esse algoritmo trabalha com grupos que são fáceis de serem gerados aleatoriamente. A ideia é obter curvas elípticas módulo n até que alguma seja encontrada tal que a quantidade de pontos na curva seja suficientemente grande para ter um fator q provavelmente primo. O grupo gerado pode ser utilizado para provavelmente obter os parâmetros que atendem o teorema. Este método é utilizado até que o inteiro q testado recursivamente seja tão pequeno que este possa ser identificado como primo rapidamente por um algoritmo determinístico qualquer.

O algoritmo de *Goldwasser-Kilian* pode ser resumido em um conjunto de três passos. No primeiro passo é necessário computar o número de pontos da curva elíptica gerada aleatoriamente. Para isso eles utilizaram o algoritmo de *Schoof* que pode ser visto em (SCHOOF, 1985). O algoritmo de *Schoof* computa de forma eficiente a quantidade de pontos em uma curva elíptica com complexidade assintótica de $O(\log^{8+\epsilon} n)$, mas a sua implementação é extremamente difícil. Considere os passos abaixo para uma entrada n provavelmente primo:

- Passo 1 - Escolha uma curva elíptica E aleatoriamente em $\mathbb{Z}/n\mathbb{Z}$ até que o número de pontos m da curva satisfaça $m = 2q$, com q provavelmente primo. O número de pontos m deve ser computado com o algoritmo de *Schoof*. Determine q com um teste de primalidade probabilístico rápido, por exemplo, o algoritmo de *Miller-Rabin*.

- Passo 2 - Se para algum ponto $P \in E$, m e q satisfazem as condições do teorema, então n é primo se o próximo passo gerar os mesmos parâmetros para q, caso contrário n é composto ou o algoritmo não consegue provar a primalidade de n.

- Passo 3 - Se q for grande, tente provar a primalidade de q recursivamente utilizando os passos 1 e 2. Caso contrário, utilize um algoritmo determinístico qualquer para determinar a primalidade de q.

Com este pequeno resumo temos o algoritmo de *Goldwasser-Kilian* visto em (GOLDWASSER; KILIAN, 1986). A ideia do *ECPP* de Atkin é oposta, com a intenção de não ter

que utilizar o algoritmo de *Schoof*, que é realmente difícil de ser implementado e causa uma provável perda de performance no algoritmo. O algoritmo de *Atkin* constrói uma curva elíptica E com o método de multiplicação complexa por uma ordem de um corpo quadrático complexo $\mathbb{Q}(\sqrt{D})$, na qual a entrada n pode ser dividida em um produto de dois elementos. Assim, o autor do algoritmo foi capaz de determinar a cardinalidade imediatamente da curva elíptica $E(\mathbb{Z}/n\mathbb{Z})$. O problema é que ninguém sabe quando isto é possível sem se basear em conjeturas. Então o teste probabilístico é do tipo *Las Vegas*, assim como o algoritmo de *Goldwasser-Kilian*.

Esta seção vai apresentar uma versão do *ECPP* de Atkin com o pseudocódigo simplificado do algoritmo, com o intuito de mostrar somente a ideia fundamental. Todas as propriedades exibidas aqui que os autores utilizaram para a construção do algoritmo seguem resultados de extensos teoremas que vão além do escopo desse trabalho. Todos os detalhes que não são exibidos podem ser vistos no trabalho original do autor em (ATKIN; MORAIN, 1993) ou na dissertação de mestrado de Uzunkol (UZUNKOL, 2004) que também exibe o algoritmo com um pseudocódigo mais detalhado.

A ideia inicial do *ECPP* de Atkin é a mesma do algoritmo de *Goldwasser-Kilian*. O algoritmo de Atkin gera o conjunto de parâmetros do teorema para n com um q provavelmente primo e tenta recursivamente gerar o mesmo conjunto de parâmetros para provar a primalidade de q, até que este fique pequeno o suficientemente para termos certeza de que ele é primo. Para determinar que um inteiro pequeno é primo, podemos utilizar um teste de primalidade determinístico mesmo que este não seja polinomial. Nesta seção, considere que um inteiro pequeno é qualquer inteiro abaixo do primo $2^{16} + 1$. Podemos, ainda, fornecer para o algoritmo uma lista com todos os primos pequenos pré-computados por um algoritmo determinístico.

O algoritmo começa computando um discriminante $-D$ negativo tal que $D \neq 0$ e tem de ser fundamental. Um discriminante fundamental é aquele em que $D \equiv 3 \mod 4$ ou $D \equiv 4 \mod 16$ ou $D \equiv 8 \mod 16$ e $|D|$ é livre de quadrados, ou seja, nenhum inteiro diferente de 1, elevado ao quadrado divide $|D|$. Além do discriminante ter de ser fundamental, este também tem que ser um bom discriminante. Se isto não acontecer, o algoritmo continua buscando até encontrar um discriminante fundamental e bom, ou desiste de procurar quando D for grande.

Um discriminante fundamental $-D$ é definido como um bom discriminante no *ECPP* de Atkin para o inteiro de entrada n se ele pode ser representado pela forma principal de $-D$, isto é, se n pode ser dividido em dois elementos $n = \pi\pi'$ no corpo quadrático complexo $\mathbb{Q}(\sqrt{-D})$. O inteiro testado n só pode ser representado pela forma principal de $-D$, se e somente se n é representado pela forma principal de $K(-D)$ onde $K(-D)$ é o conjunto de quadráticos primitivos reduzidos do discriminante $-D$ e $h(-D)$ sua ordem. Para saber se $-D$ é um bom discriminante, primeiro verifique se o símbolo de Legendre $(-D/n)$ vale 1, se isto acontecer então obter a divisão de n em dois elementos

em $\mathbb{Q}(\sqrt{-D})$, é equivalente a computar a seguinte equação para $x, y \in \mathbb{Z}$, tal que π e π' podem ser escritos em função de x, yeD:

$4n = x^2 + |D|y^2$

Esta é uma equação que pode ser resolvida utilizando o algoritmo de Cornacchia, que computa soluções de equações Diofantinas da forma $A = x^2 + By^2$ onde A e B são relativamente primos, tal que $1 \leq B < A$. Há uma ótima descrição do algoritmo na dissertação de mestrado de Osmanbey Uzunkol citado anteriormente. Após obter os valores de x e y, o algoritmo de Atkin tenta obter o número de pontos m da curva elíptica que vai ser construída em um passo posterior, utilizando as seguintes equações sabendo que π e π' podem ser escritos em função de D, x e y:

- $m = n + 1 - x$ ou $m = n + 1 + x$

Caso $-D = -3$:

- $m = n + 1 + (x + 3y)/2$ ou $m = n + 1 - (x + 3y)/2$
- $m = n + 1 + (x - 3y)/2$ ou $m = n + 1 - (x - 3y)/2$

Caso $-D = -4$:

- $m = n + 1 + 2y$ ou $m = n + 1 - 2y$

Toda vez que m é atualizado por uma das equações acima, um teste deve ser feito para descobrir se m é provavelmente fatorado. Definimos um inteiro como provavelmente fatorado verificando se existe um fator q inteiro que é provavelmente primo tal que este divida o inteiro. Podemos testar a primalidade de q com um teste de primalidade probabilístico de boa performance que acerte com uma alta probabilidade, por exemplo, o algoritmo de *Miller-Rabin*. Se o algoritmo declarar um composto q como provavelmente primo (o que deve ser muito raro com algumas iterações), não há problemas pois no último passo o algoritmo de Atkin tenta provar a primalidade de n verificando se q é realmente primo. Caso este não seja, basta voltar um passo e achar outro q. Note que no certificado há a seguinte condição: $q > (\sqrt[4]{n} + 1)^2$. Na prática, é neste passo que garantimos que q vai aceitar esta condição. Se, depois de confirmar que m é provavelmente fatorado com q, mas este não atender à restrição, então ignoramos estes inteiros m e q e prosseguimos tentando achar outros valores.

O próximo passo é criar uma curva elíptica E sobre $\mathbb{Q}$ utilizando o método de multiplicação complexa pelo anel de inteiros quadráticos de $\mathbb{Q}(\sqrt{-D})$, para saber detalhes sobre este método veja (UZUNKOL, 2004) e (ATKIN; MORAIN, 1993). Depois de obter E, precisamos reduzir a curva elíptica para uma curva em modularidade com n, a curva $E(\mathbb{Z}/n\mathbb{Z})$ da definição 2.3.1. Para isso precisamos computar o Corpo Classe *Hilbert*

de $\mathbb{Q}(\sqrt{-D})$, que é a extensão abeliana maximal sem ramificação do corpo quadrático complexo $\mathbb{Q}(\sqrt{-D})$. Então é necessário encontrar o Polinômio Classe *Hilbert* que pode ser feito utilizando um algoritmo de Henri Cohen que pode ser visto em também na dissertação de mestrado de Uzunkol em pseudocódigo.

Finalmente após obter a curva elíptica $E(\mathbb{Z}/n\mathbb{Z})$, temos que escolher um ponto P aleatório dessa curva. Além disso precisamos também de métodos rápidos para multiplicar e somar pontos desse tipo de curva, este trabalho vai utilizar os algoritmos descritos em (BLAKE; SEROUSSI; SMART, 1999) para isso. A ideia do algoritmo que escolhe um ponto aleatório de $E(\mathbb{Z}/n\mathbb{Z})$, é escolher um $x \in \mathbb{Z}/n\mathbb{Z}$ e substituir por um x' da forma de *Weierstrass* da curva $E(\mathbb{Z}/n\mathbb{Z})$, que é dada pela equação $y^2 = x^3 + ax + b$, onde $a, b \in \mathbb{Z}/n\mathbb{Z}$ e o discriminante é da forma $4a^3 + 27b^2 \in \mathbb{Z}/n\mathbb{Z}$. A forma geral da equação de *Weierstrass* não é utilizada pois sabemos que n é relativamente primo com 6. Logo após o algoritmo utiliza x' para achar y', se tal y' for encontrado então temos o nosso ponto P com (x', y').

Para realizar a operação de multiplicação com o ponto P (precisamos disso para o certificado), o algoritmo assume que n é primo. Se não for primo, pode acontecer uma impossibilidade em alguma computação no momento de gerar o ponto P ou em alguma computação de multiplicação com P. Se isto acontecer temos que n é imediatamente composto. O algoritmo utilizado para realizar a computação de $a \times P$ onde a é um inteiro, utiliza um método binário que requer a adição rápida de pontos na curva $\mathbb{Z}/n\mathbb{Z}$, que pode ser visto com detalhes na dissertação de Uzunkol e no artigo citado anteriormente de Blake. A ideia para computar a multiplicação é começar com um ponto $Q = I = (0, 1)$, depois iterar sobre a representação binária de a, onde a cada iteração dobramos o valor de Q e se o *bit* de a da iteração atual for 1, então computamos $Q = P + Q$. No final das iterações temos que $Q = a \times P$.

Com a curva $E(\mathbb{Z}/n\mathbb{Z})$, o ponto $P \in E(\mathbb{Z}/n\mathbb{Z})$ e os inteiros $\{m, q\}$ podemos começar a emitir o certificado *Atkin-Goldwasser-Kilian-Morain*. O passo final do algoritmo é fornecer todos os conjuntos de valores necessários para formar o certificado realizando os passos anteriores de forma recursiva ou iterativa para q, até que o inteiro testado seja pequeno o suficiente para provar a primalidade deste com um teste determinístico. Considerando um inteiro $\epsilon > 0$ temos que a complexidade assíntótica do algoritmo de Atkin utilizando conjeturas é $O(\log^{5+\epsilon})$ e a complexidade assintótica do algoritmo de Goldwasser-Kilian é $O(\log^{8+\epsilon} n)$. Vale notar que existe uma versão do ECPP de Atkin que F. Morain chama de *Fast ECPP*, que tem provavelmente complexidade assintótica $O(\log^{4+\epsilon})$ e pode ser vista no artigo dele em (MORAIN, 2007).

2.4.5.2 O Algoritmo

Algoritmo 4 - Algoritmo ECPP

DOCUMENTAÇÃO

TÍTULO

Algoritmo ECPP de Atkin

PROPÓSITO

Teste de primalidade.

MÉTODO

Utiliza curvas elípticas.

ENTRADAS

n: provavelmente primo

SAÍDAS

Responde Primo ou Composto, gera um certificado para a entrada.

ALGORITMO ECPP DE ATAKIN

1. $i \leftarrow 0$
2. $n_0 = n$
3. $Lista \leftarrow$ Lista ordenada de primos pequenos
4. $n_p \leftarrow$ Maior elemento da Lista
5. **enquanto** $(n_i > n_p)$, **fazer**
6. | Execute um teste de primalidade probabilístico para n_i com uma baixa probabilidade de um composto ser declarado como "Provavelmente Primo". Se o resultado for composto vá para o passo 13, caso contrário continue para o próximo passo.
7. | Ache um discriminante $-D_i$ fundamental que é bom para n_i ($n_i = \pi\pi'$ em $\mathbb{Q}(\sqrt{-D_i})$)
8. | Verifique as equações que constroem m, a cada equação verifique se m é provavelmente fatorado com $q > (\sqrt[4]{n_i} + 1)^2$. Se tal m não for possível, volte ao passo anterior e determine outro discriminante.
9. | Construa a curva elíptica E com o método de multiplicação complexa.
10. | Reduza a curva elíptica E para $E(\mathbb{Z}_n)$
11. | Determine um ponto P aleatoriamente em $E(\mathbb{Z}_n)$
12. | Verifique se $\frac{m}{q}P \neq I$ e $mP = I$, se a condição for falsa vá para o passo anterior e determine outro ponto P, se a condição for verdadeira vá para o passo 14 e se alguma computação não foi possível vá para o passo 13.
13. | A execução deste passo diz que n_i é composto, isto deve ser raro. Se i vale 0 retorne "Composto"e pare o algoritmo, caso contrário faça $n_i = n_{i-1}$, $i = i - 1$ e volte para o passo 6.
14. | **escrever** $E(a, b)$, P, m e q como parte do certificado
15. | $i = i + 1$
16. | $n_i = q$
17. **fim enquanto**
18. Se n_i está na Lista retorne "Primo", caso contrário vá para o passo 13.

2.4.5.3 Implementação

Esta implementação utiliza uma biblioteca fora das padrões que o *C++* oferece, diferente das outras deste capítulo. A biblioteca *PARI* oferece ferramentas para a implementação de códigos que necessitam de soluções de Teoria dos Números. Ela está sob a licença *GPL (GNU Public License)*. A biblioteca fornece suporte para inteiros e reais de precisão arbitrária, assim como suporte para estruturas algébricas como anéis, corpos e grupos. Também oferece suporte para a utilização de polinômios sobre essas estruturas algébricas e muitas funções que tornam a implementação do *ECPP* mais fácil. No Capítulo 4 deste trabalho, que mostra várias implementações do *AKS*, utilizamos esta biblioteca e outras que fornecem as mesmas funcionalidades. O motivo pelo qual o *PARI* foi escolhida entre elas, foi a de que esta oferece algumas funções que as outras utilizadas no Capítulo 4 não oferecem. A versão da biblioteca utilizada para esta implementação foi a 2.7.5 lançada em 9 de novembro de 2015. Sua documentação e seu código para instalação podem ser vistos em ⟨http://pari.math.u-bordeaux.fr/⟩. Mais detalhes sobre as opções de instalações utilizadas neste projeto para a biblioteca estão na Seção 4.2.3, que trata da implementação do *AKS* com ela. Nesta mesma seção são explicados detalhes para a utilização da mesma. Alguns desses detalhes também valem para esta implementação. Esses detalhes são importantes para a compreensão dos códigos usados.

O primeiro detalhe é de que a biblioteca utiliza um único tipo para a implementação de todas as estruturas da linguagem, o tipo *GEN*, que nada mais é do que um ponteiro para *long long int* da linguagem *C*. O tipo *GEN* pode ser um inteiro de precisão arbitrária, um real de precisão arbitrária, um polinômio, um complexo, qualquer estrutura algébrica oferecida pela biblioteca ou até mesmo uma matriz. Quando utilizamos a declaração de variável da forma *GEN var*, a variável *var* pode ser qualquer coisa, cabe ao programador ter em mente o tipo verdadeiro dela dentro do sistema ou algum erro pode ocorrer se não for utilizada a função correta da biblioteca para o determinado *GEN*. A construção de um *GEN* requer acessos de baixo nível sob a estrutura interna que a biblioteca usa para a construção da variável. Não é recomendado construir essas váriaveis desta maneira se o programador não tiver total domínio sob a estrutura desta dentro da biblioteca. Nesta implementação isto só é feito na construção de um polinômio e algumas váriaveis que internamente representam números complexos.

O segundo detalhe é que a biblioteca utiliza um *heap* e *stack* em uma área de memória própria que o programador tem total acesso. Todas as operações da biblioteca colocam seus valores nesse *stack*, que o programador tem de inicializar com um certo tamanho na função principal do código com a chamada *pari_init()*. Infelizmente essa área de memória não coleta o seu próprio lixo automaticamente. Cabe ao programador a fazer uma coleta de lixo manual eficiente em seu código dessa área ou o programa pode não funcionar devido a um *overflow* de memória. Um código eficiente faz esta coleta de

lixo em menos de 1% do tempo de execução total do programa. A implementação desta seção tem isso em mente, bem como que a coleta de lixo deve ser eficiente com o uso da memória. Omitiremos todos esses detalhes durante a descrição da implementação. Deve-se ter em mente de que as funções do tipo *gerepile*, *gerepileupto* e ponteiros do tipo *pari_sp* estão realizando estas operações extremamente necessárias de coleta de lixo. A biblioteca recomenda que o programador inicialize sua área de memória com pelo menos *500KB* para uma aplicação qualquer. Recomendamos o uso de *10MB* nesta implementação, para inteiros primos menores que 30 dígitos, sendo bem pessimista devido a quantidade de computações de inteiros extremamente grandes que o código pode fazer.

Como o código desta implementação é extenso, não entraremos em grandes detalhes linha a linha, assim como feito nas outras implementações deste trabalho. A ideia nesta seção é mostrar a implementação fornecendo uma visão geral sobre o código e as diferenças que este tem comparado com o algoritmo apresentado anteriormente. O primeiro passo foi implementar a lista de primos pequenos. Consideramos o primo $2^{16}+1$ como um número pequeno e este será o nosso limite. Determinamos em um pequeno arquivo chamado *small_primes.h* todos os primos pequenos, utilizando a implementação simples do *AKS* deste trabalho mostrada no Capítulo 4. O arquivo contém um *array* normal chamado *primos* com todos os primos no intervalo $[2, 2^{16}+1]$ ordenados. Definimos *SMALL_PRIME* nesse arquivo como o valor do último primo para o critério de parada do *loop* principal do algoritmo e *LAST_PRIME* como o índice no *array* deste primo.

Esta implementação não segue fielmente o algoritmo apresentado na seção anterior. De fato, a maior diferença é que esta implementação não utiliza o passo 13. Nesse caso a ideia é que a implementação é melhor utilizada em aplicações práticas onde é necessário encontrar um primo qualquer com um certo número de *bits*. A implementação tem como base o código aberto que pode ser visto em ⟨https://sourceforge.net/projects/gmp-ecpp/⟩ do *GMP* de vários autores, incluindo Henri Cohen, no qual este código utiliza alguns algoritmos apresentados por ele em seu livro (COHEN, 2013).

Primeiro apresentaremos as funções auxiliares mais importantes para a implementação, começando pela função auxiliar que calcula um fator provavelmente primo de m para determinar se este é provavelmente fatorado. As funções que não são mostradas aqui têm o código muito extenso, podem ser vistas na parte de anexos deste projeto com a implementação completa.

Listagem 2.8 - Provavelmente Fatorado

```
GEN provavelmente_fatorado(GEN m, GEN limite){
    pari_sp ltop = avma;
    GEN q = gcopy(m);

    GEN gen_3 = addii(gen_2, gen_1), gen_5 = addii(gen_3, gen_2);
    while(gequal0(modii(q, gen_2))) q = diviiexact(q, gen_2);
    while(gequal0(modii(q, gen_3))) q = diviiexact(q, gen_3);
    while(gequal0(modii(q, gen_5))) q = diviiexact(q, gen_5);
    if(gcmp(q, limite) != 1 || gequal(q, m)){
            avma = ltop;
            return NULL;
  }
    if(BPSW_psp(q)) return gerepileupto(ltop, q);

    avma = ltop;
    return NULL;
}
```

O código acima retorna um fator q provavelmente primo de m ou retorna nulo. Não se sabe qual é a chance que fator q tem de ser primo devido ao teste utilizado *Baillie-PSW*, que é detalhado na próxima seção. Morain em seu artigo (MORAIN, 2007), sugere primeiro testar se $m = 2q$ com q provavelmente primo na explicação teórica, o grande problema é que este teste é restrito. Então o próprio Morain no mesmo trabalho admite que na prática a melhor maneira de achar q é dividir m por pequenos primos conhecidos até encontrar q provavelmente primo. Já a implementação base que utilizamos, faz esta função com a fatoração com curvas elípticas de Lenstra conhecido como *ECM (Elliptic Curve Factorization)*. Este trabalho optou pela solução em que dividimos o inteiro m por 2, 3 e 5 continuamente enquanto a divisão for exata, logo após verificamos apenas uma vez se q é provavelmente primo, isto nos livra da segunda chamada de uma segunda chamada ao *Baillie-PSW*, que o código base com *GMP* citado anteriormente faz. Apesar de restrito, a solução apresentada aqui funciona bem na prática e é suficiente para encontrar m em poucas iterações.

A segunda função auxiliar implementa a multiplicação de pontos na curva elíptica $E(\mathbb{Z}/n\mathbb{Z})$ com um escalar. O algoritmo utiliza o método explicado na ideia do algoritmo, o método binário, que consiste em tomar o ponto $Q = I$ e iterar sobre a representação binária do escalar a, onde a cada iteração dobramos o valor de Q e caso o *bit* da iteração atual da representação binária de a valer 1, somamos o ponto Q com P, no final temos que $Q = a \times P$. O código abaixo utiliza alguns resultado vistos na dissertação de Mestrado de Uzunkol, em (COHEN, 2013) e em (BLAKE; SEROUSSI; SMART, 1999) para otimizar o processo, também utiliza a soma de pontos em curva elípticas com método rápido que pode ser encontrada nos mesmo trabalhos.

Listagem 2.9 - Multiplicação de um ponto por um escalar em $E(\mathbb{Z}/n\mathbb{Z})$

```
int computa_P(pair<GEN, GEN> P, GEN k, GEN ni, GEN a, pair<GEN, GEN> *Pr){
  pari_sp ltop = avma, lbot; int resposta = 1;
  pair<GEN, GEN> A, O, C; GEN d, d_x, d_y, m, inv;
  A.first = gcopy(P.first); A.second = gcopy(P.second);
  O.first = gen_0; O.second = gen_1;
  while(resposta && (cmpis(k, 0) == 1)){
    if(!gequal0(Fp_red(k, gen_2))){
      d = gcdii(Fp_red(subii(O.first, A.first), ni), ni);
      k = subis(k, 1);
      if(gequal1(d) || gequal(d, ni)) resposta = 1;
      else resposta = 0;
      if(!(gequal0(A.first) && gequal1(A.second))){
        if(gequal0(O.first) && gequal1(O.second))
          O.first = gcopy(A.first); O.second = gcopy(A.second);
        else if(resposta){
          d_x = Fp_red(subii(O.first, A.first), ni);
          d_y = Fp_red(subii(O.second, A.second), ni);
          if(gequal(A.first, O.first) &&
       gequal0(Fp_red(addii(A.second, O.second), ni)))
            C.first = gen_0; C.second = gen_1;
          else{
            inv = Fp_invsafe(d_x, ni);
            if(inv == NULL) inv = gen_0;
            m = Fp_red(mulii(d_y, inv), ni);
            C.first=Fp_red(subii(mulii(m,m),addii(A.first,O.first)),ni);
            C.second=Fp_red(subii(mulii(m,
                                  subii(A.first,C.first)), A.second), ni);}
          O.first = gcopy(C.first); O.second = gcopy(C.second);}}}
    else{
      d = gcdii(Fp_red(mulis(A.second, 2), ni), ni);
      k = diviuexact(k, 2);
      if(gequal1(d) || gequal(d, ni)) resposta = 1;
      else resposta = 0;
      if(resposta){
        inv = Fp_invsafe(mulsi(2, A.second), ni);
        if(inv == NULL) inv = gen_0;
        m =Fp_red(mulii(addii(mulis(mulii(A.first,A.first),3),a),inv),ni);
        C.first = Fp_red(subii(mulii(m, m), mulis(A.first, 2)), ni);
        C.second=modii(subii(mulii(m,subii(A.first,C.first)),A.second),ni);
        A.first = gcopy(C.first); A.second = gcopy(C.second);}}}
  lbot = avma;
  Pr->first = Fp_red(O.first, ni);
  Pr->second = gerepile(ltop, lbot, Fp_red(O.second, ni));
  if(gequal0(Pr->first) && gequal1(Pr->second)) return -1;
  return !resposta;}
```

A implementação acima retorna o valor a -1 caso o resultado seja o ponto $I = (0, 1)$, 0 se o ponto é diferente de I e 1 se o ponto não pode ser computado. A última função auxiliar que vamos apresentar nesta seção, é a que auxilia na computação da curva elíptica $E(\mathbb{Z}/n\mathbb{Z})$, computando as raízes do Polinômio Classe *Hilbert*. O algoritmo detalhado pode ser visto no trabalho de Uzunkol, assim como detalhes do algoritmo que computa o polinômio. A função *FpX_roots()* da biblioteca *PARI* computa as raízes do polinômio e a função *RgX_to_Fpx()* reduz os coeficientes do polinômio para módulo n da iteração atual do *ECPP*.

Listagem 2.10 - Encontrar curva ideal

```
1  bool curva_ideal(long D, GEN n, GEN *raiz, long *qtd_raiz, GEN *a, GEN *b){
2    GEN hpoly, menor_coeficiente;
3    *qtd_raiz = 0;
4    pari_sp av = avma;
5    hpoly = hilbert(D);
6    menor_coeficiente = gel(hpoly, 2);
7    if(!Z_ispower(menor_coeficiente, 3)){
8      avma = av;
9      return false;
10   }
11   hpoly = RgX_to_FpX(hpoly, n);
12   *raiz = gerepileupto(av, FpX_roots(hpoly, n));
13   *qtd_raiz = (lg(*raiz)-1);
14
15   if(*qtd_raiz > 0) return true;
16   else return false;
```

O programador pode filtrar os casos especiais quando $-D = -3$ ou $-D = -4$ onde temos o valor do par *(a, b)* de forma imediata, esta aplicação faz isso antes de utilizar a função acima. Nesta função também podem ser utilizados resultados que envolve o Polinômio de *Weber*, para esta aplicação não utilizamos estes resultados. Como exemplo de uma implementação que utiliza estes resultados, cheque a implementação citada anteriormente do *GMP*, esta te oferece a opção de utilizar o método só com o Polinômio de *Hilbert*, ou só com o Polinômio de *Weber*, ou os dois ao mesmo tempo. As outras funções auxiliares que não são mostradas nessa seção, somente computam passos intermediários das funções acima. Elas podem ser vistas na implementação completa na parte de anexos desse trabalho. Como descrito anteriormente, todas as funções utilizam ferramentas da biblioteca *PARI* para efetuar a coleta de lixo na *stack*, no código principal a seguir isto também é feito. A variável n de entrada foi implementada como uma *string*, para o programador que utiliza este *ECPP* não ter de utilizar nenhuma ferramenta de *Input/Output* da biblioteca *PARI*.

Listagem 2.11 - ECPP de Atkin

```
bool ecpp(const string n){
  pari_sp av = avma, av2, av3, av4;
  GEN n_i = gp_read_str(n.c_str()),i = gen_0, q, m, a, b, raiz, invariante;
  GEN c, g,   x, y, n_i1, teste, U, V;
  pair<GEN, GEN> P, P1, P2;
  long D, qtd_raiz, k; bool achei;
  while(gcmpgs(n_i, SMALL_PRIME) == 1){
    av2 = avma;
    if(!BPSW_psp(n_i)){
      av = avma;
      return false;
    }
    for(D = -3; D > -D_MAX; D--){
      achei = false;
      if(D%4 != 0 && D%4 != -3 && D%4 != 1) continue;
      av3 = avma;
      GEN D_MOD_Ni = addsi(D, n_i);
      if(kronecker(D_MOD_Ni, n_i) != 1){
        avma = av3; continue;
      }
      GEN GEN_d = stoi(D);
      if(!cornacchia2(negi(GEN_d), n_i, &U, &V)){
        avma = av3;
        continue;
      }
      teste = itor(n_i, PRECISION);
      teste = sqrtnr(teste, 4); teste = addrs(teste, 1);
      teste = powru(teste, 2);
      n_i1 = addii(n_i, gen_1);
      m = gcopy(n_i1);
      m = addii(m, U);
      q = provavelmente_fatorado(m, teste);
      if(q == NULL){
        affii(n_i1, m); m = subii(m, U);
        q = provavelmente_fatorado(m, teste);
      }
      if(q == NULL){
        if(D == -4){
          affii(n_i1, m); m = addii(m, mulis(V, 2));
          q = provavelmente_fatorado(m, teste);
          if(q == NULL){
            affii(n_i1, m); m = subii(m, mulis(V, 2));
            q = provavelmente_fatorado(m, teste);
          }
        }

```

```
47          else if(D == -3){
48            affii(n_i1, m); m = addii(m,divii(addii(U,mulis(V, 3)),gen_2));
49            q = provavelmente_fatorado(m, teste);
50            if(q == NULL){
51              affii(n_i1,m); m = subii(m,divii(addii(U, mulis(V, 3)),gen_2));
52              q = provavelmente_fatorado(m, teste);
53            }
54            if(q == NULL){
55              affii(n_i1,m); m = addii(m,divii(subii(U,mulis(V, 3)),gen_2));
56              q = provavelmente_fatorado(m, teste);
57            }
58            if(q == NULL){
59              affii(n_i1,m); m = subii(m,divii(subii(U,mulis(V, 3)),gen_2));
60              q = provavelmente_fatorado(m, teste);
61            }
62          }
63        }
64        if(q == NULL){
65            avma = av3;
66            continue;
67        }
68        qtd_raiz = 0;
69        for(int tipo = 0; tipo < 2; tipo++){
70          if(!tipo){
71            if (D == -3) {
72              a = Fp_red(gen_0, n_i);
73              b = Fp_red(gen_m1, n_i);
74              qtd_raiz = 1;
75            }
76            else if (D == -4) {
77              a = Fp_red(gen_m1, n_i);
78              b = Fp_red(gen_0, n_i);
79              qtd_raiz = 1;
80            }
81          }
82          else{
83            if(!curva_ideal(D, n_i, &raiz, &qtd_raiz, &a, &b)) continue;
84          }
85          for(int tentativa = 0; tentativa < qtd_raiz; tentativa++){
86            if(tipo){
87              invariante = Fp_red(gel(raiz, tentativa+1), n_i);
88              c=modii(mulii(invariante,
89                            ginvmod(subis(invariante, 1728),n_i)),n_i);
90              a = Fp_red(mulis(c, -3), n_i);
91              b = Fp_red(mulis(c, 2), n_i);
92            }
93
```

```
94            while(true){
95              do
96                g = randomi(n_i);
97              while(gequal0(g));
98              if(kronecker(g, n_i) == -1){
99                if(D == -3)
100               if(gequal1(Fp_pow(g, diviuexact(subiu(n_i, 1), 3), n_i)))
101                 continue;
102               break;
103             }
104           }
105           for(int t_ponto = 0; t_ponto < 100; t_ponto++){
106             y = gen_0;
107             while(gequal0(y)){
108               do {
109                 do
110                   x = randomi(n_i);
111                 while (gequal0(x));
112                 y = powiu(x, 3);
113                 y = addii(y, mulii(x, a));
114                 y = addii(y, b);
115                 y = Fp_red(y, n_i);
116               } while (kronecker(y, n_i) == -1);
117               y = Fp_sqrt(y, n_i);
118               if(y == NULL) y = gen_0;
119             }
120             P.first = gcopy(x); P.second = gcopy(y);
121             k = 0;
122             while(true){
123               int respP1 = computa_P(P, m, n_i, a, &P1);
124               if(respP1 == 1){
125                 avma = av; return false;
126               }
127               int respP2 = computa_P(P, divii(m, q), n_i, a, &P2);
128                 if(respP2 == 1){
129                   avma = av; return false;
130                 }
131                 if(respP1 == -1 && respP2 == 0){
132                   achei = true;
133                   break;
134                 }
135                 ++k;
136                 if(D == -3){
137                   if(k >= 6) break;
138                   b = gmul(b, g);
139                 }
140
```

```
141                 else if(D == -4){
142                   if(k >= 4) break;
143                   a = gmul(a, g);
144                 }
145                 else{
146                   if(k >= 2) break;
147                   a = gmul(gmul(g, g), a);
148                   b = gmul(gpowgs(g, 3), b);
149                 }
150                 a = Fp_red(a, n_i);
151                 b = Fp_red(b, n_i);
152               }
153               if(achei) break;
154             }
155             if(achei) break;
156           }
157           if(achei) break;
158         }
159         if(achei) break;
160         else avma = av3;
161       }
162       if(-D == D_MAX){
163         avma = av; return false;
164       }
165       i++;
166       n_i = gerepileupto(av2, gcopy(q));
167     }
168     long n_long = itos(n_i);
169     long low = 0, high = LAST_PRIME, mid;
170     while(low <= high){
171       mid = (low + high)/2;
172       if(primos[mid] == n_long){
173                 av = avma;
174         return true;
175       }
176       else if(primos[mid] < n_long) low = mid + 1;
177       else high = mid - 1;
178     }
179         av = avma;
180     return false;
181 }
```

Como citado anteriormente nesta seção, não entraremos em grandes detalhes sobre a implementação devido ao extenso tamanho do código. A maior diferença entre a estrutura da implementação e o algoritmo apresentado na seção anterior, é o fato de que a implementação não faz o passo 13 do algoritmo. Esta ideia faz com o que esta aplicação

seja melhor utilizada em códigos que precisam encontrar algum primo aleatório em poucas iterações. Nesse código foi utilizado o algoritmo de *Baillie-PSW* implementado pelo *PARI*, que será detalhado na seção seguinte. Não utilizamos outro teste, pois sabemos que este tem uma performance melhor na prática.

Na linha 13 podemos ver que um critério de parada utilizado para buscar o discriminante *-D* é não ultrapassar o valor de *DMAX*. Este critério é um parâmetro arbitrário do algoritmo, que na prática, deve ser calibrado para otimizar o método. Não devemos verificar discriminantes acima de 10^6 de acordo com os autores do algoritmo (ATKIN; MORAIN, 1993). No código completo deste trabalho, *DMAX* vale 38000. Se necessário o programador pode facilmente mudar este parâmetro para efeito de calibração da implementação. Note que *DMAX* pequeno, faz com que a aplicação desista facilmente de provar a primalidade de n, e se for muito grande, a aplicação pode demorar muito em uma entrada só.

Sabemos que o símbolo de Jacobi é uma generalização do símbolo de Legendre para números ímpares, todos os passos do algoritmo em que precisamos computar o símbolo de Legendre, podemos computar o símbolo de Jacobi em seu lugar. Mas a biblioteca *PARI* somente fornece funções para computar o símbolo de Kronecker, que nada mais é do que uma generalização do símbolo de Jacobi para qualquer inteiro. Quando se trata de números ímpares, o símbolo de Jacobi e o símbolo de Kronecker são equivalentes. Na linha 18 podemos ver um desses passos, em que o símbolo Kronecker foi utilizado no lugar do símbolo de Legendre do algoritmo original.

Para saber se o discriminante D é um bom discriminante e obter os valores de x, y para calcular o inteiro m, devemos resolver uma equação Diofantina como explicado na ideia do algoritmo. Para esta computação temos o algoritmo de *Cornacchia*. O *PARI* oferece funções para executar este algoritmo, uma delas é o *cornacchia2()*, que resolve a exata equação que precisamos para o *ECPP*, a utilização dela pode ser vista na linha 22.

As linhas 32 até 67 computam o inteiro q provavelmente primo para cada valor de m, de acordo com as equações desta variável apresentadas na ideia do algoritmo. Se a variável q do tipo *GEN* for diferente de nulo em algum teste intermediário, nós temos que m é provavelmente fatorado e o algoritmo pode prosseguir para o próximo passo. O *loop* da linha 69 tenta encontrar a curva elíptica E, antes filtrando os casos especiais na hora computar as raízes do Polinômio Classe *Hilbert*. Se a aplicação obtém com sucesso alguma raiz do polinômio, prosseguimos com a execução dos passos 9 e 10 do algoritmo para determinar a curva $E(\mathbb{Z}/n\mathbb{Z})$.

Na linha 105 temos um *loop* que controla as tentativas de encontrar um ponto P adequado para o certificado. A quantidade de pontos que atendem as restrições do teorema de *Goldwasser-Kilian* é elevada. Na prática, de acordo com algumas conjeturas mostradas em (ATKIN; MORAIN, 1993), o algoritmo não deve levar mais do que algumas iterações para encontrar um ponto. O limite que esta aplicação deu para encontrar um

ponto, é testar até 100 pontos por padrão. Caso não encontre, a aplicação prossegue para a próxima iteração do discriminante $-D$. Note que este é outro parâmetro arbitrário para a calibração do algoritmo, quanto maior for este parâmetro, mais chances de achar o ponto que certifique a primalidade da entrada (que pode não existir para a curva atual), mas também é maior o tempo de execução. Se o parâmetro for muito pequeno, a implementação pode desistir muito rápido de encontrar tal ponto.

As linhas 107 até 119, executam o método rápido de determinação de um ponto aleatório da curva elíptica que foi explicado anteriormente. Nas linhas 123 até 134 temos a computação e a verificação das restrições do teorema de *Goldwasser-Kilian* para escrever o certificado, utilizando o método rápido de multiplicação e adição de pontos na curva. No código desta seção omitimos a impressão do certificado para manter a implementação menor, mas o código completo da aplicação no final deste projeto, fornece a opção para o programador realizar a impressão do certificado como parte do *Output* da função.

Na linha 131, temos a clausula condicional que identifica que as duas restrições com o ponto P foram atendidas do teorema: quando a variável *booleana* dentro desta clausula é determina como verdadeira, a aplicação continua imediatamente para a próxima iteração de n_i com este valendo q.

Finalmente as linhas 168 até 178, identificam se o valor da última iteração de n_i é realmente primo. O código utiliza uma pesquisa binária no *array primos*, que deve conter todos os primos pequenos ordenados, determinados por um algoritmo determinístico. A definição de primo pequeno para aplicação vem das definições de *SMALL_PRIME* e *LAST_PRIME*, que devem valer o último primo da lista e o seu índice no *array* nesta ordem. Para todos os testes executados, determinamos os primos desta lista utilizando o *AKS*.

A aplicação pode ser compilada com o parâmetro *-Ofast* do compilador de *C++* do *GNU Compiler Collection*. O programador que utiliza esta implementação não pode esquecer de inicializar a biblioteca *PARI*. O código completo com todas as funções utilizadas, inclusive com as que não foram exibidas aqui, com um exemplo de função principal podem ser vistas na parte de anexos deste projeto.

2.4.6 O Algoritmo Baillie-PSW

O Algoritmo *Baillie-PSW* é um teste de primalidade *Monte Carlo* construído na década de 1980, que pode ser considerado determinístico para números até 2^{64}. Atualmente é muito utilizado na prática, perdendo apenas para o algoritmo de *Miller-Rabin*, uma vez que todo *Baillie-PSW* o implementa. O algoritmo pode ser encontrado em quase todas as bibliotecas que possuem ferramentas de Teoria dos Números. Vale ressaltar que ainda não foi descoberto um inteiro composto que seja identificado como primo nesse

teste, contudo não existem provas que que tal número não exista. Em (POMERANCE, 1984), *Carl Pomerance*, mostra um argumento heurístico em que o teste tem infinitos contra-exemplos. Existe também um prêmio em dinheiro para recompensar alguém que aponte um composto que passe o teste.

2.4.6.1 A ideia

A ideia consiste em executar o Teste de Lucas mostrado na seção 2.3.3 e o teste de *Miller-Rabin* da seção 2.3.2 com 1 iteração. A razão é que não se tem conhecimento sobre uma interceção entre o conjunto de pseudoprimos de Lucas e o conjunto de pseudoprimos que passam no teorema 2.3.2.1 que o *Miller-Rabin* faz a cada iteração. Existem variantes do algoritmo que utilizam outros testes de primalidade do mesmo tipo, por exemplo, uma variação seria utilizar o Teste de Fibonnacci no lugar do Teste de Lucas. Há diversas sugestões para a implementação ter melhor performance na prática como por exemplo realizar divisões sucessivas por pelos mil primeiros primos antes de realizar o teste.

2.4.6.2 O Algoritmo

Algoritmo 5 - Teste de primalidade Baillie-PSW.

DOCUMENTAÇÃO
TÍTULO
Baillie-PSW

PROPÓSITO
Teste de primalidade.

MÉTODO
Executar o Miller-Rabin com Tesde de Lucas.

ENTRADAS
n: Número a ser testado.
SAÍDAS
Responde Provavelmente Primo ou Composto
OBSERVAÇÕES, RESTRIÇÕES, REQUISITOS
A entrada n deverá ser maior que 2.
ALGORITMO ALGORITMO BAILLIE-PSW
1. **se** $((miller(n, 1) \neq Verdade)$, **então**
2. | **retornar** Composto
3. **fim se**
4. **se** $((Lucas(n) \neq Verdade)$, **então**
5. | **retornar** Composto
6. **fim se**
7. **retornar** Provavelmente Primo

2.4.6.3 Implementação

Listagem 2.12 - Teste de primalidade Baillie-PSW

```
1 bool baillie_psw (long long int n){
2     int i;
3     if (!miller_rabin(n, 1)) return false;
4     if (!lucas(n)) return false;
5     return true;
6 }
```

O simples código conta com as funções *miller_rabin()* e *lucas()*, que são respectivamente as implementações mostradas nas seções 2.3.2 e 2.3.3. É interessante executar uma pré-computação com um *array* com os 1000 primeiros inteiros primos antes de executar os testes, isto faz com o que a aplicação fique com melhor performance em geral. Este

teste pode ser visto em inúmeras bibliotecas para *C*, inclusive algumas das bibliotecas utilizadas no Capítulo 4 deste projeto, que fornecem ferramentas para lidar com problemas de Teoria dos Números. Elas implementam este teste com uma ótima performance, visto que utilizam poderosas ferramentas de pré-computação.

Como citado anteriormente, este teste é considerado probabilístico pois não há provas de que não existe interseção nos conjuntos de *pseudoprimos* de Lucas e números compostos que passam no *Miller-Rabin*. Especula-se de que não há compostos abaixo de 1000 dígitos que passam o teste, mas isto não nunca foi comprovado. Assim como todos os testes de primalidade mostrados nesse capítulo, a implementação do *Baillie-PSW* com função principal encontra-se nos anexos deste projeto.

2.5 **Testes de performance**

Esta seção é dedicada a mostrar o tempo de execução de todos os testes de primalidade detalhados neste capítulo, menos do Teste de Lucas que é utilizado no *Baillie-PSW*. Note que o objetivo é somente ter uma noção da performance prática de cada teste de primalidade, pois os testes de performance efetuados aqui utilizam entradas muito pequenas quando comparadas as entradas que são utilizadas na prática. Também note que não podemos comparar o *ECPP* com os outros testes, devido a natureza do *ECPP* ser diferente de todos os outros algoritmos apresentados aqui. Como descrito nas seções 2.3.1 e 2.3.2, o *Solovay-Strassen* e o *Miller-Rabin*, recebem como parâmetro de entrada um número de iterações k, quanto maior for este k, maior a probabilidade da resposta "provavelmente primo"ser realmente "primo"e obviamente maior o tempo de execução do algoritmo. Os parâmetros k escolhidos foram 10 para o *Solovay-Strassen* e 5 para o *Miller-Rabin*, então temos uma probabilidade de aproximadamente 99,9% de um número provavelmente primo ser primo em ambos os testes.

O recurso computacional do projeto para os testes de performance utiliza um processador AMD FX-8350 4.00 GHz, 16 GB de memória RAM no sistema operacional Ubuntu 15.10, onde cada aplicação foi executada com exatamente um núcleo dedicado deste recurso computacional. Todos as implementações foram compiladas utilizando o compilador de *C++* do *GNU Compiler Collection 4.8.4*, utilizando o parâmetro -Ofast. Em especial vale notar que o *PARI* foi configurado para utilizar a biblioteca *GMP*, na hora de implementar as variáveis de precisão arbitrária, você pode ver com mais detalhes essas configurações utilizadas no Capítulo 4. A contagem de tempo foi feita utilizando a biblioteca *Chrono* do *C++11* com *Steady Time*, somente contando o tempo de execução em que a função principal do teste de primalidade esteve no processador, excluindo o tempo de qualquer inicialização da aplicação e qualquer operação de entrada ou saída. A execução da inicialização do *PARI* não é cronometrada para o algoritmo *ECPP*, pois o

tempo desta inicialização é desprezível quando comparado ao tempo total da execução do algoritmo.

Os testes foram feitos com a execução de três arquivos, a aplicação executou cada algoritmo do começo ao fim, sequencialmente sem parar no recurso computacional do projeto. Os tempos apresentados nas tabelas abaixo são a média aritmética de 10 execuções de cada arquivo. Estas entradas também vão ser utilizados no Capítulo 4, para testar as implementações do *AKS*. Os arquivos utilizados podem ser vistos na parte de anexos deste projeto.

- Arquivo 1 - Consiste como entrada, o conjunto de inteiros [0, 10000], sabe-se que dentre esses inteiros existem 1229 primos. Não executamos este teste para o *ECPP*, pois a pré-computação contido no mesmo consiste de primos maiores que a entrada do teste.

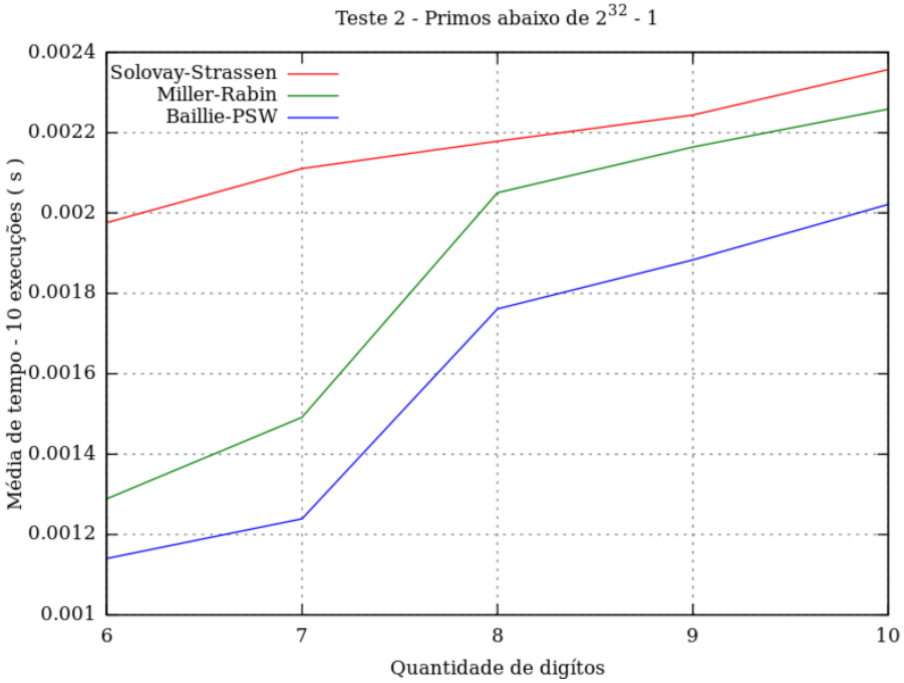
- Arquivo 2 - Este teste consiste na execução sequencial de determinados inteiros primos distintos. Foram utilizados primos de 6 até 10 dígitos, todos abaixo de 2^{31}. A entrada é formatada em grupos de 5 primos para cada dígito, com um total de 25 primos testados a cada execução.

- Arquivo 3 - Assim como no teste 2, este arquivo executa sequencialmente determinados inteiros primos de 10 a 16 dígitos, maioes que 2^31. Para cada dígito, há 5 primos distintos, formando um total de 35 primos na execução sequencial do teste.

Tabela 1 - Testes de performance - Probabilísticos

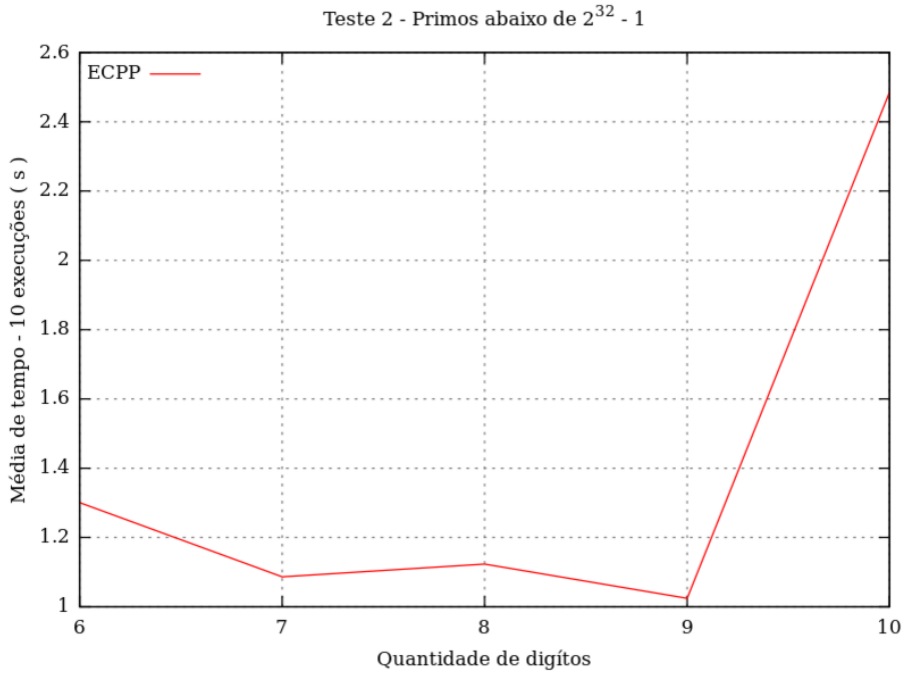
Teste 1 - 10000 Inteiros	
Implementação	Média 10 execuções (s)
Solovay-Strassen	1.01513
Miller-Rabin	1.00789
Baillie-PSW	0.540026
Teste 2 - Primos abaixo de $2^{31} - 1$	
Implementação	Média de 10 execuções
Solovay-Strassen	0.0108654
Miller-Rabin	0.00925344
Baillie-PSW	0.00804556
ECPP	7.0204
Teste 3 - Primos acima de $2^{31} - 1$	
Implementação	Média de 10 Execuções
Solovay-Strassen	0.02203471
Miller-Rabin	0.01865224
Baillie-PSW	0.01958281
ECPP	40.56864

Figura 7 - Teste 2 - Solovay-Strassen, Miller-Rabin e Baillie-PSW



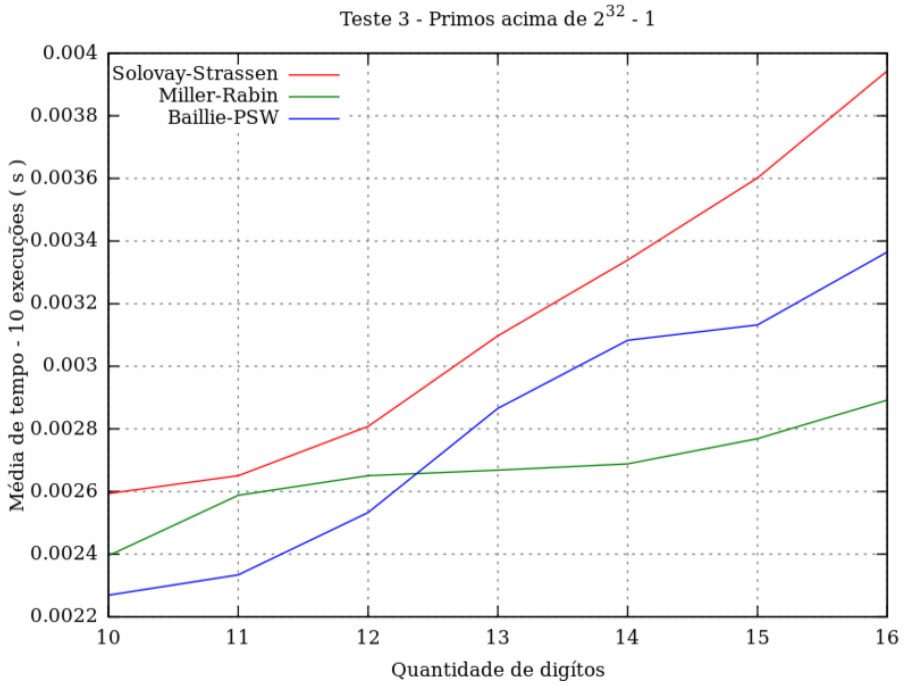
Legenda: Execução sequencial para 5 primos de cada dígito.

Figura 8 - Teste 2 - ECPP.



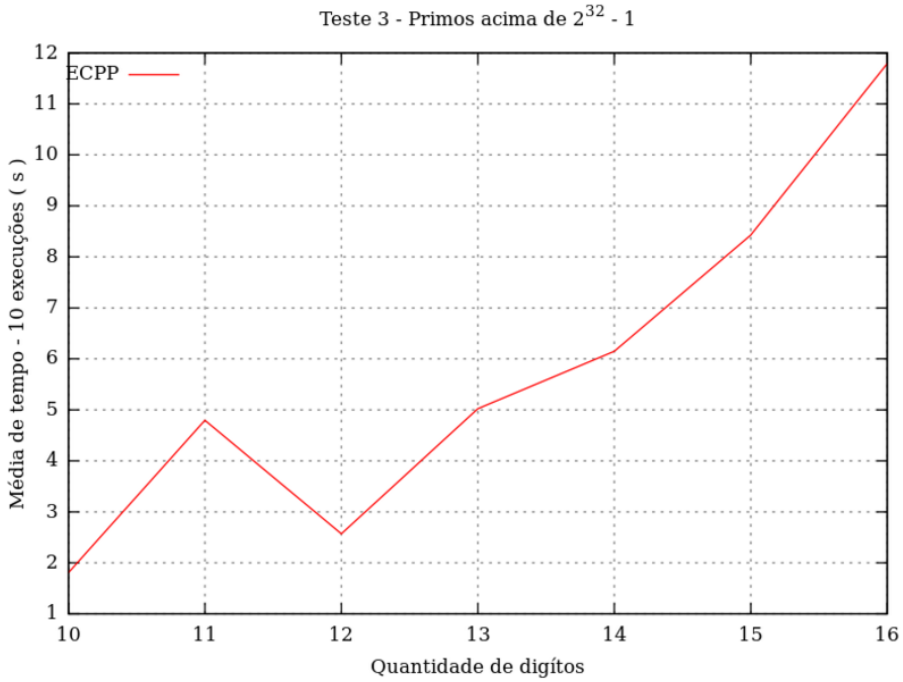
Legenda: Execução sequencial para 5 primos de cada dígito.

Figura 9 - Teste 3 - Solovay-Strassen, Miller-Rabin e Baillie-PSW.



Legenda: Execução sequencial para 5 primos de cada dígito.

Figura 10 - Teste 3 - ECPP.



Legenda: Execução sequencial para 5 primos de cada dígito.

3 TESTE PRIMALIDADE AKS

3.1 Introdução

O algoritmo AKS foi desenvolvido por três cientistas indianos do Indian Institute of Technology Kanpur, Manindra Agrawal, Neeraj Kayal e Nitin Saxena, que batizaram o algoritmo com suas iniciais. Pode ser considerado um marco de grande importância teórica, pois foi o primeiro teste de primalidade que é polinomial, determinístico e geral, enquanto os testes conhecidos previamente possuíam somente uma ou duas das características anteriores.

O AKS foi concebido através de uma ideia que surgiu do projeto de bacharelado feito por Neeraj Kayal e Nitin Saxena orientado por Manindra Agrawal. Recebeu em 2006 o prêmio Gödel, que reconhece grandes artigos na área da Teoria da Computação. Recebeu, também, o prêmio Fulkerson que reconhece artigos da área de Matemática Discreta.

A ideia inicial do AKS vem de uma generalização do Pequeno Teorema de Fermat para polinômios. Se dois naturais a e n são primos entre si, então n é primo se e somente se a equação 1 vale no anel de polinômios $\mathbb{Z}$[x].

$$(x-a)^n \equiv (x^n - a) \bmod n \tag{1}$$

Como o tempo para computar o teste acima é extremamente alto, esta solução não é muito útil, o cálculo de $(x-a)^n$ leva mais tempo que o Crivo de Eratóstenes. Manindra Agrawal e seu supervisor de doutorado, Somenath Biwas, trabalham em um teste de primalidade *Monte Carlo* que contornava totalmente o problema de expandir o polinômio. Infelizmente o resultado prático do teste não é competitivo com o algoritmo de *Miller-Rabin*.

Uma nova ideia surgiu desse resultado, mas inicialmente só era interessante como uma nota histórica para o problema de determinação de primos. Mesmo assim, Manindra Agrawal tinha uma grande fé no seu teste de primalidade, com os seus estudantes, Rajat Bhattacharjee e Prashant Pandey, decidiu examina-lo a fundo. Dessa pesquisa outra ideia surgiu: examinar o resto da divisão de $(x-a)^n$ por $x^r - 1$, ao invés de examinar somente o polinômio $(x-a)^n$. Se a computação de r se mantiver em tempo logarítmico em relação à n, então essa divisão pode ser computada por algoritmos com tempo polinomial.

Agrawal apresenta uma notação que utilizaremos daqui em diante. A notação $f(x) = g(x) \bmod (h(x), n)$ representa a equação $f(x) = g(x)$ no anel quociente $\mathbb{Z}_n[x]/(h(x))$, ou seja, o resto do polinômio depois da divisão por $h(x)$, com os coeficientes inteiros re-

duzidos em n.

Temos que se n é primo então, certamente a equação 1 pode ser reescrita como a equação 2 para todo r e n relativamente primos a a.

$$(x-a)^n \equiv (x^n - a) \bmod (x^r - 1, n) \tag{2}$$

Basta saber, agora, quais valores de a e r permitem saber se n é composto. No projeto de bacharelado conjunto de Bhattacharjee e Pandey (BHATTACHARJEE; PANDEY, 2001), os dois estudantes fixaram $a = 1$ e examinaram os requerimentos de r. Através da análise de experimentos com $r \leq 100$ e $n \leq 10^{10}$, eles chegaram na seguinte conjectura: quando r é relativamente primo a n e a equação 3 vale, então n é primo ou $n^2 \equiv 1 \bmod r$.

$$(x-1)^n \equiv (x^n - 1) \bmod (x^r - 1, n) \tag{3}$$

Então para um dos primeiros números primos r em $\log n$, quando $n^2 \not\equiv 1 \bmod r$, teríamos um teste de primalidade polinomial de complexidade assintótica $O(\log^{3+\varepsilon} n)$.

Com a orientação de Agrawal, os estudantes, Neeraj Kayal e Nitin Saxena, continuaram examinando como projeto de bacharelado, a relação do teste da equação 3 para descobrir outros testes de primalidade *Monte Carlo*. O resultado foi um grande sucesso.

Assumindo que a hipótese de Riemann é verdadeira, eles conseguiram mostrar que o teste da Equação 3, pode ser restringido para r com valores de 2 até $4 \lg^2 n$ para uma prova de primalidade. Desse modo, podiam obter um teste de primalidade com complexidade $O(\log^{6+\varepsilon})$. Além de descobrir que a conjectura de Bhattacharjee e Pandey, seguia uma conjectura antiga de *Carl Pomerance*, um dos pesquisadores que descobriu o teste de primalidade *APR*. Eles concluíram o projeto de bacharelado que trouxe a expectativa de conseguir um teste de primalidade polinomial, em abril de 2002, que pode ser visto em (KAYAL; SAXENA et al., 2002).

No próximo verão, os dois não foram para casa. Eles começaram os estudos para o doutorado, o projeto de bacharelado deles precisava somente de uma pequena mudança de ponto de vista para se chegar ao algoritmo pretendido. Eles analisaram o teste da Equação 2, novamente com a fixo em 1 e r variado. Logo se perguntaram sobre o efeito de manter r fixo e variar a. O resultado não demorou, foi uma caracterização das potências de números primos. (BORNEMANN, 2003).

O teorema que eles encontraram, agora é base do algoritmo *AKS*. As principais ideias, a análise de complexidade e o próprio algoritmo AKS serão abordados com mais detalhes nas seções posteriores deste capítulo.

3.2 A ideia do algoritmo AKS

Antes de explicarmos o funcionamento do algoritmo, devemos entender o fundamento matemático que o constitui. Para tanto apresentaremos alguns lemas.

Lema 3.2.1. *Seja $a \in \mathbb{Z}, n \in \mathbb{N}, n \geq 2$ e $(a, n) = 1$, então n é primo se e somente se $(x-a)^n \equiv (x^n - a) \bmod n$.*

Demonstração. Para provar este lema basta provar que

$$(x-a)^n - (x^n - a) \equiv 0 \bmod n \tag{1}$$

Pelo binômio de Newton, temos que os coeficientes de x^i para $0 < i < n$ são da forma $\binom{n}{i}\ a^{n-i}$.

$$(x+a)^n = \sum_{i=0}^{n} \binom{n}{i} x^{n-i}a^i = x^n + a^n + \sum_{i=1}^{n-1} \binom{n}{i} x^{n-i}a^i \qquad \text{(Binômio de Newton)}$$

Então utilizando estes coeficientes, para demonstrar este lema, basta provar que a Equação 3 vale para todo n primo e não vale para valores de n compostos

$$\binom{n}{i} = \frac{n!}{i!(n-i)!} \equiv 0 \bmod n \tag{2}$$

Se n é primo, então na Equação 2 sempre existirá um fator n disponível, devido ao fato de que n não pode ser escrito como um produto de dois ou mais números menores do que n. Portanto $i!$ e $(n-i)!$ não divide n, o que conclui que todos os coeficientes são 0 na modularidade com n.

Agora suponha que n é composto, tomemos um primo q que é um fator de n, tal que $q^k || n$. Como consequência desta suposição temos que

1. q^k não divide $\binom{n}{q}$

$$\binom{n}{q} = \frac{n!}{q!(n-q)!} = q^{k-1}\alpha\frac{(n-1)!}{(q-1)!(n-q)!} = q^{k-1}\alpha\frac{(n-1)(n-2)...(n-q+1)}{(q-1)!} \tag{3}$$

Para que a Equação 3 seja 0 na modularidade com q^k, é necessário que os fatores do produto $(n-1)(n-2)...(n-q+1)$ possuam pelo menos um fator q. Contudo uma vez que n é múltiplo de q, sabemos que o próximo número menor do que n, que contém o fator q será n-q. Portanto como produtório não contém o fator q e a equação 3 acima não será divisível por q^k.

2. q^k é relativamente primo com a^{n-q}
   Suponha que não sejam primos entre si. Então como q^k é uma potência de primo, a deve ser da forma q^t com $t \in \mathbb{N}$. Porém, dada a hipótese $(a, n) = 1$ e q fator de n, se a for da forma q^k teríamos uma contradição. Portanto a^{n-q} deve ser primo com q^k, caso contrário teríamos um absurdo.

Pelos itens acima, temos que alguns coeficientes de $x^q \bmod n$ não valem zero e, portanto, a equação 2 não vale para todos os valores de n compostos, concluindo a prova. □

O Lema 3.2.1 sugere um teste de primalidade, cuja complexidade é da forma Ω(n), pois temos que avaliar pelo menos n coeficientes no lado esquerdo da equação. No artigo *Primes is in P* (AGRAWAL; KAYAL; SAXENA, 2004), os autores sugerem uma modificação para reduzir o número de coeficientes que são avaliados nos dois lados, através da equação 4. Escolhendo um r suficientemente pequeno:

$$(x + a)^n \equiv x^n + a \bmod (x^r - 1, n) \tag{4}$$

A Equação 4 é satisfeita para todos os n primos e todos os valores de a e r. Contudo, alguns n compostos podem também satisfazer a equação para alguns valores de a e r. Apesar disso, será apresentada uma forma de gerar um r apropriado para que, quando a Equação 4 for satisfeita para alguns valores de a, n deverá ser primo. Se esta forma de geração e o número de iterações dos valores de a tiverem complexidade polinomial na ordem de $\log n$ então temos um teste de primalidade determinístico em tempo polinomial.

Antes de demonstrar o limite para a geração de r, há uma notação que precisamos estabelecer, ela será utilizada posteriormente no decorrer do projeto. Para $r \in \mathbb{N}$, $a \in \mathbb{Z}$, tal que $(a, r) = 1$, dizemos que a ordem de um módulo r é o menor número k, tal que $a^k \equiv 1 \mod r$. A notação utilizada para isto será $o_r(a) = k$.

Agora com os seguintes lemas podemos mostrar o limite superior para a geração de r:

Lema 3.2.2. *Seja o MMC(m) o mínimo múltiplo comum dos primeiros m números para* $m \geq 7$ *então* $MMC(m) \geq 2^m$

Demonstração. (ARVIND; SAPTHARISHI, 20XX) Considere a seguinte integral:

$$\int_0^1 x^n (1 - x)^n \, \mathrm{d}x \tag{5}$$

Podemos observar que $x(1-x) < \frac{1}{4}$, então a integral acima tem um limite superior de 2^{2n}. Se utilizarmos o teorema binomial para expandir $(1 - x)^n$, temos que:

$$\frac{1}{2^{2n}} \leq \int_0^1 x^n(1-x)^n \,\mathrm{d}x = \int_0^1 \sum_{k=0}^{n}(-1)^k \binom{n}{k} x^{n+k} = \sum_{k=0}^{n}(-1)^k \binom{n}{k} \frac{1}{n+k+1} = \frac{M}{N} \qquad (6)$$

Na equação acima, podemos observar que N é, no máximo, o MMC(L) dos números 1, 2, ..., $2n$, $2n+1$ e M é pelo menos 1. Então $\frac{1}{2^{2n}}L > \frac{1}{2^{2n}}N \geq 1$ e assim temos que $L > 2^{2n}$.

Considere m par e maior que 7 na forma de $m = 2n$, temos que o MMC(m) é da forma $MMC(m) > 2^m$. Para $m \geq 7$ na forma ímpar podemos observar que $m = 2n+1$ então o MMC(m) fica como $MMC(m) > 2^{m-1}$.

□

Lema 3.2.3. *Seja $B = \lceil \lg^5 n \rceil$. Existe um $r \leq$ máximo$\{3, B\}$ tal que $o_r(n) > \lg^2 n$.*

Demonstração. (AGRAWAL; KAYAL; SAXENA, 2004) Podemos checar que para $n = 2$ o lema é verdadeiro quando $r = 3$:

$$o_3(2) = 2 > \lg^2 2 = 1 \qquad (7)$$

Agora assuma que $n > 2$ e $B = \lceil \lg^5 n \rceil$. Então $B > 10$ e o Lema 3.2.2 se aplica. Seja $m^k \leq B, m \geq 2$. O maior valor que k pode assumir é $\lfloor \lg B \rfloor$, considere o menor r que não divide a seguinte equação:

$$n^{\lfloor \lg B \rfloor} \times \prod_{i=1}^{\lfloor \lg n \rfloor} (n^i - 1) \qquad (8)$$

Como r não divide $n^{\lfloor \lg B \rfloor}$, então *(r, n)* não pode ser dividido por todos os divisores primos de r, assim $\frac{r}{(r,n)}$ não divide a equação acima. Com o fato de que r é o menor número que não divide a equação, temos que *(r, n) = 1*. Como r não divide nenhum $n^i - 1$ para $1 \leq i \leq \lfloor \lg n \rfloor$ temos que $o_r(n) > \lfloor \lg n \rfloor$. Para finalizar:

$$n^{\lfloor \lg B \rfloor} \times \prod_{i=1}^{\lfloor \lg n \rfloor} (n^i - 1) < n^{\lfloor \lg B \rfloor + \frac{1}{2}\lg^2 n \times (\lg^2 n - 1)} \leq n^{\lg^4 n} \leq 2^{\lg^5 n} \leq 2^B \qquad (9)$$

Pelo Lema 3.2.2, o mínimo múltiplo comum dos primeiros B números é pelo menos 2^B então $r \leq B$, concluindo a prova. □

Sabendo agora que podemos descobrir r em tempo polinomial, no artigo dos autores do AKS podemos ver uma prova para a Equação 4 que demonstra que a deve ser testado através de iterações até um limite de $\lfloor \sqrt{\phi(n)} \lg n \rfloor$ onde ϕ é a função aritmética

Totiente de Euler. Com esta ideia em mente, podemos apresentar agora o algoritmo completo para o teste de primalidade polinomial e analisar sua complexidade nas seções posteriores.

3.3 O Algoritmo

(AGRAWAL; KAYAL; SAXENA, 2004)

Algoritmo 6 - Teste de primalidade AKS.

DOCUMENTAÇÃO
TÍTULO
AKS

PROPÓSITO
Teste de primalidade.

MÉTODO
Verifica se a equação 4 é atendida.

ENTRADAS
n: Número a ser testado
SAÍDAS
Responde Primo ou Composto
OBSERVAÇÕES, RESTRIÇÕES, REQUISITOS
A entrada n deverá ser maior que 1.
ALGORITMO AKS

1. **se** ($n = a^b$, $a \in \mathbb{N}$ e $b > 1$), **então** {*Etapa 1*}
2. | **retornar** "Composto"
3. **fim se**
   declarar r **numérico**
4. $r \leftarrow$ menor número tal que $o_r(n) > \lg^2(\mathrm{n})$ {*Etapa 2*}
5. **se** ($n > (a, n) > 1$ para algum a $\leq$ r), **então** {*Etapa 3*}
6. | **escrever** "Composto"
7. **fim se**
8. **se** (r $\geq$ n), **então** {*Etapa 4*}
9. | **retornar** "Primo"
10. **fim se**
11. **para** a **de** 1 **até** $\lfloor \sqrt{\phi(r)} \lg n \rfloor$, **fazer** {*Etapa 5*}
12. | **se** $((x + a)^n \neq x^n + a \pmod{x^r - 1, n})$), **então**
13. | | **retornar** "Composto"
14. | **fim se**
15. **fim para**
16. **retornar** "Primo"

Nesta seção vamos detalhar cada etapa do algoritmo separadamente, deixando a

análise de complexidade na seção seguinte. O algoritmo mostrado é a última versão do teste de primalidade *AKS*. Note que o *AKS* no artigo original possui 6 etapas. Assumimos que as etapas 5 e 6 são uma só, por simplicidade.

- Etapa 1: A primeira etapa consiste em determinar se o número n da entrada é uma potência perfeita, isto é, verificar se podemos escrever n como a^b para qualquer $a \in \mathbb{N}$ e $b > 1$. Se este teste for positivo, o número é obviamente composto. Esta etapa preliminar é opcional e pode ser implementada em tempo quase linear como descrito em (BERNSTEIN; JR; PILA, 2007). A sua execução vale a pena, pois identifica muitos compostos óbvios antes de entrar em uma etapa que exige mais tempo de computação.

- Etapa 2: Como descrito anteriormente, a ordem de n em um módulo r escrita como $o_r(n)$ é o menor número k tal que $n^k \equiv 1 \mod r$. Esta etapa consiste em determinar a constante r, a mesma potência r da equação 4 da seção anterior, que é fundamental para a execução das etapas posteriores. O Lema 3.2.3 demonstra que ela vale, no máximo, $\lceil \lg^5 n \rceil$, uma propriedade fundamental para este algoritmo funcionar em tempo polinomial.

- Etapa 3: Esta etapa consiste de um pequeno teste, que busca um $a \leq r$, tal que este seja fator de n. Se este teste encontrar tal fator, n é composto. Caso r seja maior que n, esta etapa vai verificar todos os números abaixo de n, mas a condição $(a, n) < n$ não permite que o n primo seja identificado como composto, pois iterações acima de n devem retornar que $(a, n) = n$ ou $(a, n) = 1$.

- Etapa 4: A quarta etapa consiste de um simples teste para verificar se o valor r é maior ou igual a n. Se for, n é primo pois caso não fosse a terceira etapa iria encontrar um fator não trivial de n. Esta etapa só é relevante para valores de n abaixo de *5 690 034* pois o Lema 3.2.3 demonstra que r tem um limite superior em $\lceil \lg^5 n \rceil$.

- Etapa 5 Esta é a etapa principal do algoritmo que consiste em verificar o teste da Equação 4 da seção anterior em que checamos se $(x + a)^n$ no anel quociente $Z_n[x]/(x^r - 1)$ é congruente a $(x^n + a)$ no mesmo anel quociente. O limite superior para a é demonstrado com detalhes em (AGRAWAL; KAYAL; SAXENA, 2004) e r é a constante encontrada na Etapa 2. Caso o teste seja bem sucedido, então n é composto pois vai contra o teste estabelecido na Equação 4. Finalmente, caso n passe integralmente por todas as iterações então podemos dizer que n é primo.

Vamos mostrar na seção seguinte que este algoritmo tem complexidade de tempo polinomial.

3.4 **Análise de complexidade**

Antes de começar a análise de complexidade, precisamos estabelecer uma notação que será utilizada nesta seção: o símbolo $\tilde{O}(f(x))$ será utilizado para substituir a notação de complexidade assintótica O onde $\tilde{O}(f(x)) = O(f(n)\times\log^{k+\epsilon}(f(n))$ para qualquer $\epsilon > 0$. Essencialmente, esta notação ignora os fatores logarítmicos de n pois estes são irrelevantes em determinar se o algoritmo é ou não de complexidade polinomial pois $\lg n < n$.

Com esta notação, podemos considerar que a multiplicação de dois inteiros de m *bits* é da forma $\tilde{O}(m)$ pelo algoritmo de *Schönhage–Strassen* (SCHÖNHAGE; STRASSEN, 1971). Também podemos definir a complexidade da multiplicação de polinômios de grau d e coeficientes de m *bits* como $\tilde{O}(d \times m)$ utilizando *FFT* (Fast Fourier Transform) e a versão do *Schönhage–Strassen* para polinômios (CANTOR; KALTOFEN, 1991).

Teorema 3.4.1. *A complexidade assintótica do teste de primalidade AKS é* $\tilde{O}(\log^{10.5} n)$

Demonstração. A Etapa 1 consiste em determinar se n é uma potência perfeita na forma $n = a^b$. Isto pode ser feito utilizando pesquisa binária na base da potência no intervalo de]1, 2, ..., n[. Iterando sobre o expoente b temos que o máximo de pesquisas binárias que precisamos realizar é da forma $2^b \leq n$ (ou $b \leq \lg n$). A cada pesquisa binária uma potência precisa ser computada. Utilizando a multiplicação rápida temos que a complexidade da Etapa 1 é de $\tilde{O}(\log^3 n)$. Um pseudo-código deste algoritmo pode ser visto em (DIETZFELBINGER, 2004). Vale citar também que há um algoritmo que computa esta etapa em praticamente tempo linear em (BERNSTEIN; JR; PILA, 2007).

Na Etapa 2 temos que computar o valor de r, isto pode ser feito com iterações sobre r e verificando se $n^k \neq 1 \mod r$ para todo k menor ou igual a $\log^2 n$, então temos $O(\log^2 n)$ multiplicações módulo r. Com a multiplicação rápida de inteiros temos que a complexidade de cara iteração é $\tilde{O}(\log r \times \log^2 n)$. Como foi demonstrado no lema 3.2.3 que o valor máximo de r é $\lceil \lg^5 n \rceil$, a complexidade final desta etapa é $\tilde{O}(\log^7 n)$.

A Etapa 3 consiste em realizar computações de MDC (máximo divisor comum) r vezes. Cada computação de MDC tem complexidade de $O(\lg n)$ como descrito em (AGRAWAL; KAYAL; SAXENA, 2004). Então esta etapa tem complexidade $O(r \lg n) = O(log^6 n)$.

É fácil verificar que a Etapa 4 tem complexidade $O(\log n)$.

Na Etapa 5 precisamos verificar $\lfloor \sqrt{\phi(r)} \lg n \rfloor$ onde ϕ é a função aritmética Totiente de Euler. Como sabemos que o $\phi(r)$ é no máximo $r - 1$ então podemos considerar que a complexidade é da forma $O(\sqrt{\phi(r)} \lg n) = O(\sqrt{r} \log n)$. Como a cada iteração temos que computar a multiplicação de $\lg n$ polinômios de grau r pelo algoritmo de exponenciação binária, temos que cada equação pode ser verificada em complexidade $\tilde{O}(r \log^2 n)$. Então a complexidade final da Etapa 5 é $\tilde{O}(r\sqrt{r} \log^3 n) = \tilde{O}(r^{1.5} \log^3 n) = \tilde{O}(\log^{10.5} n)$, que também é a complexidade final do algoritmo pois esta domina todas as anteriores. □

Vale notar que a complexidade do AKS pode melhorar se o limite superior de r for menor. O melhor cenário possível seria onde a complexidade assintótica na determinação de r fosse $O(\log^2 n)$. Há duas conjecturas que suportam essa ideia:

Conjectura 3.4.1. *Conjectura de Artin: Para qualquer número natural que não é um quadrado perfeito, o número de primos $q \leq m$ em que $o_q(n) = q - 1$ é assintóticamente $C_a \times \frac{m}{\ln m}$ onde C_a é a constante de Artin que tem um limite de $C_a > 0.35$.*

Se a conjectura de Artin se efetivar para m em complexidade assintótica $O(\log^2 n)$ é imediato mostrar que existe um r em complexidade assintótica em $O(log^2 n)$ com as propriedades que atendem a Equação 4 da Seção 3.2. Sabe-se que a conjectura é verdadeira na hipóstese generalizada de Riemann. (AGRAWAL; KAYAL; SAXENA, 2004)

Conjectura 3.4.2. *A conjectura Sophie-Germain de densidade de primos: O número de primos $q \leq m$ que atendem a condição de $2q + 1$ ser primo também é assintóticamente $\frac{C_2 m}{\ln^2 m}$ onde C_2 é a constante primo gêmeo que é estimada ser aproximadamente 0.66. Esses primos também são chamados de primos de Sophie-Germain.*

Se esta conjectura for verdadeira, com a densidade de primos Sophie-Germain existe pelo menos $\lg^2 n$ primos entre $8 \lg^2 n$ e $c \lg^2 n \times \lg^2(\lg n)$ para uma constante c apropriada. Para qualquer primo q, vale que $o_q(n) \leq 2$ ou que $o_q(n) \geq \frac{q-1}{2}$. Para q da forma $o_q(n) \leq 2$ temos que q divide $n^2 - 1$ e que tal número é limitado pela complexidade assintótica de $O(\log n)$. Isso implica que existe um primo r tal que a complexidade assintótica para determinar o mesmo é $\tilde{O}(\log^2 n)$ e que $o_r > \log^2 n$. Tal r trará um algoritmo com complexidade assintótica $\tilde{O}(\log^6 n)$. (AGRAWAL; KAYAL; SAXENA, 2004)

Há um certo progresso na prova da Conjectura 3.4.2. (GOLDFELD, 1969) demonstrou que certos primos q ocorrem com uma densidade positiva e (FOUVRY, 1985) com isto provou o seguinte lema que é verdade para expoentes até 0.6683 como visto em (BAKER; HARMAN, 1996):

Lema 3.4.1. *Seja P(m) o maior divisor primo de m. Existe constantes c maiores que zero e n_0 que para todo $x \geq n_0$:*

$$|\{q \mid q \, primo, q \leq x \, e \, P(q-1) > q^{\frac{2}{3}}\}| \geq \frac{cx}{\ln x} \tag{7}$$

Com o lema acima podemos revisitar o Teorema 3.4.1 e analisar novamente a complexidade do AKS com o seguinte teorema:

Teorema 3.4.2. *A complexidade assintótica do AKS é $\tilde{O}(\log^{7.5} n)$*

Demonstração. A densidade de primos do lema 3.4.1 implica que a Etapa 2 do Algoritmo AKS vai encontrar um r em complexidade assintótica $O(\log^3 n)$ com $o_r(n) > \lg^2 n$. Então na etapa 5 temos que $\tilde{O}(r^{1.5} \log^3 n) = \tilde{O}(\log^{7.5} n)$. □

4 IMPLEMENTAÇÃO DO ALGORITMO AKS

Neste capítulo vamos apresentar uma implementação em C++ da versão do algoritmo AKS da Seção 3.3. Com o objetivo de ter o código simplificado e eficiente, somente utilizando os recursos que a própria linguagem oferece independendo da preocupação em instalar qualquer pacote adicional para compilar o código. Logo após, apresentaremos uma versão otimizada do código com o objetivo de diminuir o tempo de execução para entradas muito grandes e diminuir a restrição para entradas n, permitindo assim maiores valores através da utilização de bibliotecas externas que auxiliam no uso de inteiros de precisão arbitrária. As implementações foram amplamente testadas. Neste capítulo apresentarmos uma análise delas em função do tempo para várias entradas, assim como apresentaremos também uma análise de restrição máxima da entrada que elas têm. Os testes foram feitos com o mesmo recurso computacional mencionado no Capítulo 2: o processador AMD FX-8350 4.00 GHz, 16 GB de memória RAM no sistema operacional Ubuntu 15.10. Compilado com C++ do *GNU Compiler Collection* 4.8.4 utilizando o parâmetro -Ofast.

A maior dificuldade em escrever a primeira implementação inicial simples foi definitivamente testar a congruência polinomial na Etapa 5. Inicialmente, foram utilizados outros trabalhos para buscar a maneira mais eficiente de implementar o AKS de acordo com a visão dos autores, mas infelizmente múltiplos erros foram encontrados em várias implementações. Foi verificado que havia múltiplos primos entre os 100 primeiros que não eram detectados, algumas ainda tinham erros extremamente grosseiros que identificavam múltiplos compostos como primos e vice-versa. Dentre outras implementações que não tinham erros, algumas eram extremamente confusas e não aceitavam entradas n maiores que 10^4, ou implementavam versões alternativas do AKS que eram muito lentas para se realizar qualquer teste com números acima de 10^4. Vale a pena citar alguns trabalhos com código limpo e claro, que implementaram versões recentes do *AKS* como os trabalhos de (ROTELLA, 2005) e (JIN, 2005). Já na segunda implementação realizada, não encontramos muitas dificuldades, pois a própria primeira implementação validava os testes feitos. Nessas, foram utilizadas bibliotecas externas que são otimizadas a nível de código de máquina.

4.1 Implementação simples

4.1.1 Funções e estruturas auxiliares

A partir da segunda etapa são necessárias algumas funções auxiliares para a implementação do AKS. Como o código não utiliza bibliotecas externas precisamos definir essas funções. As Etapas 2 e 3 necessitam de um algoritmo capaz de computar o MDC de dois números inteiros, uma implementação do algoritmo de Euclides seria o suficiente para resolver este problema. Neste trabalho foi utilizado o algoritmo de Stein (também conhecido como o algoritmo de MDC binário detalhado no apêndice) que tem uma performance um pouco melhor na prática, apesar de ter a mesma complexidade do Algoritmo Euclidiano. Vale a pena citar que existem algoritmos mais rápidos para computar o MDC, como o algoritmo de Lehmer utilizado em muitas bibliotecas, outro exemplo é o *Accelerated integer GCD algorithm*, no qual uma versão pode ser vista em (SEDJELMACI; LAVAULT, 2014).

Listagem 4.1 - MDC de Stein

```
1  inline unsigned int mdc(const unsigned int &a, const unsigned int &b){
2      unsigned int _a = a, _b = b, numero_z_dir;
3
4      if(_a == 0) return b;
5      if(_b == 0) return a;
6
7      numero_z_dir = __builtin_ctz(_a | _b); //0's a direita do menor
8      _a >>= __builtin_ctz(_a); //Desloca e atribui
9
10     do{
11         _b >>= __builtin_ctz(_b);
12
13         if(_a > _b){
14           int troca = _a; _a = _b; _b = troca;
15         }
16
17         _b = _b - _a;
18     } while(_b != 0);
19
20     return _a << numero_z_dir;
21 }
```

Por causa da própria definição de um passo da etapa 2, é necessário computar várias potências modulares com três inteiros. O algoritmo utilizado para esta tarefa pode ser visto com mais detalhes em (BRUEN et al., 1996) ou no apêndice deste trabalho. O algoritmo utiliza um método conhecido como exponenciação binária da direita para esquerda que utiliza a representação binária da base para efetuar a potência.

Listagem 4.2 - Potência modular

```
inline unsigned int pow_mod(const unsigned int &a, const unsigned int  &b,
    const unsigned int &n){
    unsigned int base=a, power = b, resp = 1;

    while (power){

        if (power&1)
            resp = (resp*base) % n;

        power >>= 1;
        base = base % n;
        base = (base*base) % n;
    }
    return resp;
}
```

Na etapa 5 precisamos computar a constante que define o limite superior de a utilizando a função aritmética Totiente de Euler, que computa a quantidade de números relativamente primos com a entrada que podem ser encontrados utilizando o MDC.

Listagem 4.3 - Totiente de Euler

```
inline unsigned int totiente_euler(unsigned int n){
    unsigned int resultado;

    resultado = n;
    for (unsigned long long int i=2; i*i<=n; ++i){
        if (n % i == 0){
            while (n % i == 0)
                n /= i;

            resultado -= resultado / i;
        }
    }
    if (n > 1)
      resultado -= resultado / n;

    return resultado;
}
```

A etapa 5 lida com polinômios, uma simples classe auxiliar foi definida para tratar esses polinômios. A classe conta com um *array*, auxiliar para armazenar os coeficientes do polinômio assim como dois operadores básicos para facilitar a visualização da etapa sem comprometer sua performance. O *array* foi declarado com tamanho de 64 *bits* para diminuir a restrição da entrada n, que será analisada em uma próxima seção deste capítulo.

Listagem 4.4 - Classe polinômio e suas funções

```
1  class polinomio{
2      public:
3          unsigned long long int *coef;
4          unsigned int grau = 0;
5          polinomio(const unsigned int &tam);
6          bool operator == (const polinomio &a);
7          polinomio& operator = (const polinomio &a);
8          ~polinomio();
9  };
10 polinomio::polinomio(const unsigned int &tam){
11     coef = new unsigned long long int[tam+1];
12     for(unsigned int i = 0; i <= tam; ++i) coef[i] = 0;
13 }
14 bool polinomio::operator == (const polinomio &a){
15     if(grau != a.grau)
16     return false;
17     for(unsigned int i = 0; i <= grau; ++i)
18         if(coef[i] != a.coef[i]) return false;
19     return true;
20 }
21 polinomio& polinomio::operator = (const polinomio &a){
22     grau = a.grau;
23     for(unsigned int i = 0; i <= grau; ++i) coef[i] = a.coef[i]; //Teste
24     return *this;
25 }
26 polinomio::~polinomio(){ //Destrutor
27     delete[] coef;
28 }
```

4.1.2 Etapa 1

A primeira etapa consiste em determinar todos os números compostos que são potências perfeitas. Isto foi feito utilizando uma estratégia simples com pesquisa binária na base da potência no intervalo de $]1, 2, ..., n[$. Sabemos que, para $n = a^b$ precisamos executar no máximo $\lg n$ pesquisas binárias: se $a^b < n$, então limitamos a parte de baixo do intervalo pela metade, caso contrário limitamos a parte de cima. Caso a igualdade seja satisfeita, encontramos um composto que é uma potência perfeita. Um pseudocódigo pode ser encontrado no Capítulo 2 de (DIETZFELBINGER, 2004). Como executamos $\lg n$ pesquisas binárias nas linhas 7-15 que fazem $\lg n$ potências na linha 9, esta implementação tem complexidade $O(\log^3 n)$. Vale a pena citar que esta etapa pode ser feita em essencialmente em tempo linear como descrito em (BERNSTEIN; JR; PILA, 2007).

Listagem 4.5 - AKS Simples etapa 1

```
unsigned int menor, maior, meio;
double potencia;
double logaritmo_n = log2(n);
for(int b = 2; b <= logaritmo_n; ++b){
  menor = 1;
    maior = n;
    while(maior - menor >= 2){
      meio = (menor + maior)/2;
        potencia = pow((double) meio, b);
        if(potencia < n)
          menor = meio;
        else
          if(potencia > n)  maior = meio;
        else
          return false;
    }
}
```

4.1.3 Etapas 2, 3 e 4

Na segunda etapa, o algoritmo de Stein foi utilizado para determinar se a iteração de r é relativamente primo com n, e a função *pow_mod()* foi utilizada para determinar se a iteração de k está correta, de acordo com a definição de ordem onde $n^k \equiv 1 \bmod r$. Na implementação abaixo da etapa 2, uma solução intuitiva foi utilizada onde encontramos o valor exato de k (a ordem de n em relação a r). O critério de parada é verificar se o valor k encontrado é realmente maior que $\lg^2 n$.

Listagem 4.6 - AKS Simples etapa 2 intuitiva

```
unsigned int r = 0;
double logaritmo_n_2 = logaritmo_n * logaritmo_n;
unsigned int q = 2; //r temporario, iteracao atual de r

while(!r){
  //Verificando se n e relativamente primo com q
  if(mdc(n, q) == 1){
    int k = 1;
    while(pow_mod(n, k, q) != 1) k++;
    if(k > logaritmo_n_2) r = q;
    ++q;
  }
}
```

Contudo podemos melhorar um pouco a implementação acima sabendo que o AKS não utiliza o valor de k nas etapas seguintes: podemos modificar o *loop* da linha 8 para realizar um simples teste, que requer uma quantidade menor de iterações para determinar r utilizando a própria definição de ordem como artifício. O *loop* foi modificado para ter no máximo $\lfloor \lg^2 n \rfloor$ iterações onde, se em alguma iteração o teste da ordem é aceito, então sabemos que k é menor que $\lfloor \lg^2 n \rfloor$, portanto a iteração atual de r não é a que esperamos. Caso o *loop* termine sem que o teste seja aceito, temos o valor certo de r sem a determinação exata de k economizando várias iterações.

Listagem 4.7 - AKS Simples etapa 2 melhorada

```
1  unsigned int r = 0;
2  double logaritmo_n_2 = logaritmo_n * logaritmo_n;
3  unsigned int q = 2; //r temporario, iteracao atual de r
4  while(!r){
5    if(mdc(n, q) == 1){ //Verificando se n e relativamente primo com q
6          unsigned int k; //Procurando a ordem n^k mod r
7          for(k = 1; k <= floor(logaritmo_n_2);++k){
8            //Ordem menor que logaritmo
9            if(pow_mod(n, k, q) == 1){
10               break;
11             }
12         }
13         if(k > logaritmo_n_2)
14           r = q;
15   }
16   ++q;
17 }
```

A terceira etapa utiliza somente a função auxiliar de MDC. Com um simples *loop* verificamos se algum valor a é relativamente primo com n. Caso o teste do *loop* seja bem sucedido, determinamos que o número é composto parando a execução do *AKS*.

Listagem 4.8 - AKS Simples etapa 3

```
1 for(unsigned int a = 2; a <= r; ++a){
2   //Descobrindo se a e relativamente primo com n
3   unsigned int este_mdc = mdc(a, n);
4   if(este_mdc > 1 && este_mdc < n) return false;
5 }
```

Tão simples quanto a etapa 3, a implementação da etapa 4 consiste em um condicional para finalizar a execução do *AKS*, caso o teste seja bem sucedido.

Listagem 4.9 - AKS Simples etapa 4

```
1 if(r >= n) return true;
```

As etapas 2, 3 e 4 podem ser executadas juntas utilizando o *loop* principal da implementação da etapa 2 melhorada. Para a etapa 4, é imediato que se a iteração r for maior ou igual a n, podemos parar imediatamente e responder primo. Para a etapa 3, é simples notar que se trata de um *loop* similar ao da etapa 2, pois todos os números que vieram antes de r são as mesmas iterações da etapa 2. Com uma nova cláusula para o condicional da linha 5, podemos economizar várias chamadas da função MDC não sendo necessária uma nova verificação.

Listagem 4.10 - AKS Simples etapa 2, 3 e 4

```
1  unsigned int r = 0, k;
2  double logaritmo_n_2 = logaritmo_n * logaritmo_n;
3  double q = 2; //r temporario, iteracao atual de r
4  while(!r){
5    if(q == n) return true; //Etapa 4
6      // o mesmo mdc pode ser usado para realizar as etapas 2 e 3 e o valor
       deve ser verificado pela etapa 4
7      if(mdc(n, q) == 1){ //Verificando se n e relativamente primo com q
8        for(k = 1; k <= floor(logaritmo_n_2);++k){
9          if(pow_mod(n, k, q) == 1){ //Ordem menor que logaritmo
10             break;
11         }
12       }
13       if(k > logaritmo_n_2)  r = q;
14     }
15     else{ //Etapa 3
16       //existe a (q) menor do que r tal que a e n nao sao primos entre si
17       return false;
18     }
19     ++q;
20 }
```

4.1.4 Etapa 5

A etapa 5 é a mais difícil de ser implementada. Inicialmente, precisamos calcular o limite superior da iteração a que é dado por $\lfloor\sqrt{\phi \lg n}\rfloor$, onde ϕ é a função aritmética Totiente de Euler de n que pode ser vista na seção 4.1 e no apêndice deste trabalho com mais detalhes. Precisamos então computar o lado esquerdo e o lado direito da equação 4 da seção 3.2:

$$(X + a)^n \equiv X^n + a \bmod (X^r - 1, n) \tag{4}$$

O resultado do lado direito desta equação é sempre o polinômio $X^{n \mod r} + a$, então podemos instanciar direto um objeto de polinômio sem precisar computa-lo somente atualizando o coeficiente de grau zero a cada iteração de a. O lado esquerdo precisa ser computado a cada iteração, isto é feito pela função *potencia_modular_polinomial()* que retorna exatamente o resultado $(X+a)^n$ no anel $\mathbb{Z}_n[X]/(X+r)$, os detalhes desta função vão ser explicados nesta seção no parágrafo posterior. Esta função auxiliar deixa o código da etapa mais simples, decidimos especificar a função de potência modular polinomial nesta seção, e não em 4.4.1 para deixar a ideia da etapa mais clara.

Listagem 4.11 - AKS Simples etapa 5

```
1  //Funcao totiente de Euler
2  unsigned int phi;
3
4  phi = totiente_euler(r);
5
6  unsigned int constante = floor(sqrt(phi)*logaritmo_n);
7
8  unsigned int grau_maximo = n % r;
9
10 //Polinomio da direita x^(n % r)
11 polinomio lado_dir(grau_maximo);
12 lado_dir.coef[grau_maximo] = 1;
13 lado_dir.grau = grau_maximo;
14
15 //Polinomio do lado esquerdo
16 polinomio lado_esq(grau_maximo);
17
18 for(unsigned int a = 1; a <= constante; ++a){
19   //Polinomio direito x^(n % r) + a
20   lado_dir.coef[0] = a;
21   potencia_modular_polinomial(lado_esq, n, r, a, floor(logaritmo_n));
22
23   if(!(lado_esq == lado_dir))
24     return false;
25 }
26 return true;
```

Algoritmo 7 - Potência Modular Polinomial

DOCUMENTAÇÃO

TÍTULO

Potência Modular Polinomial

PROPÓSITO

Retorna $Q(X)^n$ no anel $\mathbb{Z}[X]/Z(X)$.

MÉTODO

Iteração sobre a representação binária de n.

ENTRADAS

Z(X): Polinômio do módulo
Q(X): Polinômio Base

SAÍDAS

O polinômio $Q(X)^n$ no anel $\mathbb{Z}[X]/Z(X)$

OBSERVAÇÕES, RESTRIÇÕES, REQUISITOS

Nenhuma

ALGORITMO CONGRUÊNCIA POLINOMIAL

declarar P(X) **polinômio**
declarar B **representação binária**
1. P(X) ← 1
2. B ← representação binária de n
3. **para** i **de** 0 **até** $\lg n$, **fazer**
4. | P(X) ← $P(X) \times P(X)$ computado no anel $\mathbb{Z}[X]/Z(X)$
5. | **se** (B(i) é 1), **então**
6. | | P(X) ← $P(X) \times P(X)$ computado no anel $\mathbb{Z}[X]/Z(X)$
7. | **fim se**
8. **fim para**
9. **retornar** P(X)

Este algoritmo funciona exatamente da mesma maneira que a potência modular de inteiros detalhada no apêndice. A função implementada difere um pouco do algoritmo, a diferença é que cada coeficiente mantido dos polinômios estão módulo n, assim a saída da função se torna $Q(X)^n$ no anel $\mathbb{Z}_n[X]/Z(X)$, a ideia pode ser vista em (ROTELLA, 2005). Esta implementação não é genérica e se aproveita de diversas características e propriedades da congruência polinomial. A ideia para computar o quadrado de *P(X)* é cortar o polinômio pela metade e somar suas partes sabendo que o grau máximo do polinômio resultado vale até r. As operações de modularidade com n são feitas para não haver estouro nas variáveis. O conjunto de *loops* do lado de fora estão em uma pequena cláusula condicional para evitar muitas multiplicações por zero nas primeiras iterações quando o grau do polinômio corrente resultado ainda é abaixo de r. Para a segunda multiplicação do polinômio dentro da condicional que verifica o *bit* i de n outro *loop* específico é utilizado sabendo que o grau máximo é r.

Listagem 4.12 - Potência modular polinomial

```
inline potencia_modular_polinomial(polinomio &resto, const unsigned int &n,
    const unsigned int &r, const unsigned int &a, const int log_n){
polinomio resultado(r); //auxiliar
unsigned long long int novo_coeficiente;
unsigned int anterior;
bool bit;
resto.grau = 0;   resto.coef[0] = 1; //P(x) = 1;
for(int i = log_n; i >= 0; --i){
  resultado.grau = resto.grau * 2;
  if(resultado.grau >= r){
    resultado.grau = r - 1;
    for(unsigned int j = 0; j <= resultado.grau; ++j){
      novo_coeficiente = 0;
      for(unsigned int k = 0, l = j; k <= j; ++k, --l){
        novo_coeficiente += resto.coef[k]*resto.coef[l];
        novo_coeficiente %= n;}
      for(unsigned int k = j+1, l = r-1; k < r; ++k, --l){
        novo_coeficiente += resto.coef[k]*resto.coef[l];
        novo_coeficiente %= n;}
        resultado.coef[j] = novo_coeficiente;}}
  else{ //Evita muitas iteracoes com mult zero
    for(unsigned int j = 0; j <= resultado.grau; ++j){
      novo_coeficiente = 0;
      for(unsigned int k = 0, l = j; k <= j; ++k, --l){
        novo_coeficiente += resto.coef[k]*resto.coef[l];
        novo_coeficiente %= n;}
        resultado.coef[j] = novo_coeficiente;}}
    resto = resultado;
    bit = (n & ( 1 << i )) >> i; //Pega o i'th bit de n
    if(bit){
      if(resto.grau + 1 == r) anterior = resto.coef[r - 1];
      else{
        anterior = 0;
        resto.grau += 1;}
      for(unsigned int j = 0; j <= resto.grau; ++j){
        novo_coeficiente = (resto.coef[j]*a) + anterior;
        anterior = resto.coef[j];
        resto.coef[j] = novo_coeficiente % n;}}}
  for(unsigned int i = resto.grau; i >= 0; --i){
    if(resto.coef[i] != 0){
      resto.grau = i;
      break;}}
return resto;}
```

Como já citado anteriormente, o *array* que guarda os coeficientes do polinômio tem tamanho de 64 *bits* para evitar mais um *overflow* que será explicada na seção de testes e

limites da implementação neste capítulo. Vale a pena citar que esta etapa pode ser mais rápida, a implementação acima utiliza uma multiplicação de polinômios simples de complexidade $O(n^2)$, esta poderia ser reduzida para $O(n \log n)$ caso a implementação utilize um algoritmo de multiplicação de polinômios que utilize FFT (Fast Fourier Transform) para computar o polinômio do lado esquerdo, o método pode ser visto em (CANTOR; KALTOFEN, 1991). O código completo incluindo comentário e um exemplo de função *main()* podem ser visto no anexo deste projeto.

4.2 **Implementação otimizada**

Esta seção irá mostrar três versões otimizadas da implementação do teste de primalidade *AKS* simples. Todas as implementações, assim como a primeira, serão analisadas em função do tempo na seção seguinte. Também será analisado em mais detalhes as limitações na entrada de cada implementação, nesta seção só mencionaremos em quais linhas de código causam essas limitações. As implementações utilizaram as versões mais recentes das bibliotecas mais recomendadas para se lidar com problemas de Teoria dos Números, elas oferecem ferramentas apropriadas para a fácil implementação do algoritmo. Estas são: *NTL* (Number Theory Library), *FLINT* (Fast Library for Number Theory) e *PARI* (Pascal Arithmetic). Todas as três bibliotecas estão sob a licença *GPL* (GNU General Public License), ou seja, gratuitas para uso, modificação e estudo desde que qualquer trabalho derivado das mesmas também mantenha esta licença.

Junto com as três implementações, também utilizamos a biblioteca *GMP* (GNU Multiple Precision Arithmetic Library), pois todas as três bibliotecas de Teoria dos Números utilizam ela. O *GMP* é uma biblioteca para *C/C++* que da suporte para inteiros e reais de precisão arbitrária, ou seja, váriaveis limitadas somente pela quantidade de memória que a máquina tem. Esta biblioteca é muito utilizada, conta com um enorme suporte e é otimizada a nível de código de máquina. A versão do *GMP* utilizada nas três versões é a 6.1.0 lançada em 01 de novembro de 2015, a página oficial da biblioteca que disponibiliza toda a documentação e códigos para download é ⟨https://gmplib.org⟩.

Foi considerado para este projeto o uso de outras bibliotecas, como o *LiDIA* (A library for computational number theory) e o *Miracl* (Multiprecision Integer and Rational Arithmetic C/C++ Library), mas estas não tem o suporte que as três acima têm e foram descontinuadas. Muitas implementações do *AKS* podem ser encontradas utilizando estas bibliotecas, onde algumas infelizmente não funcionam ou só implementam a etapa 5. As implementações completas das três versões, com comentários detalhados e com um exemplo de *int main()* podem ser vistas na parte de anexos deste projeto.

4.2.1 Versão utilizando o NTL

A biblioteca *NTL* (Number Theory Library) conta com uma ótima documentação e vários exemplos. Sua página oficial ⟨http://www.shoup.net/ntl/⟩ fornece o download de toda a documentação e código da biblioteca. Ela possui uma excelente interface para *C++* deixando o código consideravelmente simples, conta com módulos que oferecem suporte para inteiros de precisão arbitrária, também conta com diversos módulos que podem facilitar a implementação do *AKS*. Os módulos utilizados foram: *ZZ* (pela função de MDC e inteiros de precisão arbitrária) e *ZZ_pX* (polinômios de coeficientes *ZZ_p* com precisão arbitrária no anel $\mathbb{Z}_n$). Outros módulos como *ZZ_pEX* (polinômios com coeficientes de precisão arbitrária no anel quociente $\mathbb{Z}_n[X]/\mathbb{Z}[x]$) podem ser utilizados, mas verificamos com alguns testes que a diferença de tempo de execução é mínima. A versão utilizada neste trabalho do NTL é a 9.6.4 lançada em 30 de janeiro de 2016.

Listagem 4.13 - Teste de primalidade AKS com NTL e GMP

```
bool aks_ntl(const string n){
  mpz_t mpz_n; //Etapa 1
  mpz_init(mpz_n);
  mpz_set_str(mpz_n, n.c_str(), 10);
  if(mpz_perfect_power_p(mpz_n))
    return false;
  //Fim da Etapa 1

  //Etapas 2, 3 e 4
  ZZ ZZ_n = to_ZZ(conv<ZZ>(n.c_str()));
  bool teste;
  long k;
  ZZ r(2), n_mod_r;
  double logaritmo_n = log(ZZ_n)/log(2);
  while(1){
    if(r == ZZ_n) return true; //Etapa 4
    if(IsOne(GCD(ZZ_n, r))){
      teste = true;
      rem(n_mod_r, ZZ_n, r);
      for(k = 1; k <= logaritmo_n_2; ++k){
        if(IsOne(PowerMod(n_mod_r, k, r))){
          teste = false;
          break;
        }
      }
      if(teste) break;
    }
    else return false; //Etapa 3
    ++r;
  } //Fim da Etapa 2, Etapa 3 e Etapa 4
```

```
31    //Etapa 5
32    long phi;
33    long r_long = conv<long>(r);
34    totiente_euler(phi, r_long);
35    long int constante = floor(sqrt(phi)*logaritmo_n);
36    ZZ_p::init(ZZ_n);
37    ZZ_pX f(r_long, 1); f -= 1;
38    const ZZ_pXModulus modulo(f);
39    ZZ_pX lado_direito(conv<long>(n_mod_r), 1);
40    ZZ_pX lado_esquerdo(1, 1);
41    for(long a = 1; a <= constante; ++a){
42      SetCoeff(lado_esquerdo, 1);
43      lado_esquerdo += a;
44      PowerMod(lado_esquerdo, lado_esquerdo, ZZ_n, modulo);
45      lado_esquerdo -= a;
46      if(lado_esquerdo != lado_direito) return false;
47      clear(lado_esquerdo);
48    }
49    return true;
50    //Fim Etapa 5
51  }
```

A etapa 1 utiliza uma variável do tipo *mpz_t* que vem da biblioteca *GMP*. Este é o tipo de inteiro de precisão arbitrário da biblioteca, o *GMP* oferece uma função chamada *mpz_perfect_power()* que aceita um *mpz_t* como entrada, respondendo se a entrada é ou não uma potência perfeita de algum número. Esta função tem uma performance muito boa e é perfeita para executar a primeira etapa do *AKS*.

Nas etapas 2, 3 e 4 declaramos o primeiro tipo de variável que vamos utilizar do *NTL*, a classe *ZZ*, que implementa inteiros de precisão arbitrária fornecendo diversas funções e operações aritméticas com interface de *C++* (opcional). Na linha 12 declaramos k com o tipo *long* que gera uma limitação na entrada n junto com as variáveis declaradas do tipo *double*, todas essas limitações vão ser analisadas na próxima seção. Logo após, utilizamos mais quatro funções do módulo *ZZ* nas linhas 17, 19 e 21. A função *IsOne()* verifica se o objeto da classe *ZZ* é igual a um (equivalente a um simples teste $ZZ == 1$) e pode ser substituída pela sua equivalente utilizando a interface de C++ que o *NTL* oferece. A função *gcd()* retorna um objeto da classe *ZZ* que contém o valor do MMC de dois outros objetos, sabe-se que a biblioteca utiliza o algoritmo de MDC binário. Já a função *rem()* é uma simples operação de modularidade com os parâmetros sendo objetos da classe *ZZ*. Sua equivalente que pode ser utilizada é $ZZ_1 = ZZ_2 \% ZZ_3$, onde o ZZ_i é cada um dos i objetos. Finalmente, a função *PowerMod()* é basicamente a potência modular binária que resulta em um objeto da classe *ZZ*, a função está disponível na biblioteca em duas versões através de polimorfismo: uma que utiliza todos os argumentos com inteiros de precisão arbitrária e a segunda utilizada na implementação em que o

argumento que recebe o valor do expoente é do tipo *long*.

Na etapa 5, utilizamos a mesma função *totiente_euler()* mostrada na implementação anterior, mas dessa vez com ϕ passado como parâmetro por referência. O *conv* da linha 33 converte um objeto *ZZ* para um valor do tipo *long*. O principal motivo desta conversão é a limitação da biblioteca nas funções seguintes que utilizam o módulo *ZZ_pX*. Apesar dos coeficientes de *ZZ_pX* serem do tipo *ZZ_p* de precisão arbitrária, o grau máximo que a classe suporta é do tipo *long*. Como o grau máximo que estes polinômios podem atingir no *AKS* são no máximo do valor de r, este tem de ser um *long*.

Na linha 36, utilizamos a função *init* da classe *ZZ_p* para informar que todos os coeficientes dos polinômios da classe *ZZ_pX* vão ser calculados módulo n. Na linha 38 inicializamos um polinômio da classe *ZZ_pXModulus* para o *NTL* realizar uma pré-computação e acelerar as contas que são feitas pela função da linha 44 *PowerMod()*, como pode ser visto o polinômio inicializado é o $x^r - 1$. A função *PowerMod()* responde no primeiro parâmetro passado por referência, um objeto da classe *ZZ_pX* que vale a exponenciação do polinômio *ZZ_pX* do segundo argumento com o *ZZ* do terceiro argumento módulo o quarto argumento da classe *ZZ_pXModulus*.

Sabe-se que todas as multiplicações feitas por este módulo utilizam um método com *FFT* (Fast Fourier Transform) e *CRT* (Chinese Remainder Theorem), ou o algoritmo clássico de multiplicação de polinômio, ou utiliza a versão para polinômios do algoritmo de *Karatsuba*, essa escolha é determinada por uma função heurística. Vale notar que, se este módulo utiliza-se a versão de multiplicação de polinômios do algoritmo de *Schönhage–Strassen*, este poderia ser mais rápido. Finalmente a função *SetCoeff* recebe dois argumentos: o primeiro argumento é o polinômio, o segundo argumento é o grau no qual o seu coeficiente vai receber o valor 1. A última função *clear()* basicamente transforma o polinômio passado como parâmetro por referência em zero.

4.2.2 Versão utilizando o FLINT

O *FLINT* (Fast Library for Number Theory) não fornece tantas facilidades quanto a última biblioteca e sua documentação deixa um pouco a desejar, mas a biblioteca oferece algumas funcionalidades que não há na anterior, por exemplo, a função *fmpz_euler_phi()* e uma conversão direta entre os tipos de precisão arbitrária utilizados no FLINT e no *GMP*. Assim como o NTL, esta biblioteca possui módulos bem estruturados para a implementação do *AKS* que fornecem suporte variáveis de precisão arbitrária e polinômios com coeficientes de precisão arbitrária. Também vale notar que a biblioteca conta com uma interface opcional para *C++* que deixa o código mais simples. Sua página oficial ⟨http://www.flintlib.org/⟩ disponibiliza toda a sua documentação e seu código oficial para *download* que conta com alguns bons exemplos de sua utilização.

Os módulos utilizados para esta versão do *AKS* foram: *fmpz* (fornece inteiros de precisão arbitrária) e *fmpz_mod_poly* (fornece suporte a polinômios de coeficientes de precisão arbitrária no anel $\mathbb{Z}_n$), os módulos equivalentes da biblioteca que possuem uma versão com interface opcional para *C++* são: flintxx, *fmpzxx* e *fmpz_mod_polyxx*. Neste trabalho a variante com *C++* foi utilizada na versão 2.5.2 lançada em 7 de agosto de 2015.

Listagem 4.14 - Teste de primalidade AKS com FLINT e GMP

```
bool aks_flint(const fmpz &n){
  //Etapa 1
  mpz_t mpz_n;
  mpz_init(mpz_n);
  fmpz_get_mpz(mpz_n, &n);
  if(mpz_perfect_power_p(mpz_n)) return false;
  //Fim da Etapa 1
  //Etapas 2, 3 e 4
  bool teste;
  long k;
  fmpz r = 2, pot_mod, gcd;
  double logaritmo_n =  fmpz_dlog(&n)/fmpz_dlog(&r);
  double logaritmo_n_2 = logaritmo_n*logaritmo_n;
  while(1){
    if(r == n) return true;  //Etapa 4
    fmpz_gcd(&gcd, &n, &r);
    if(fmpz_is_one(&gcd)){
      teste = true;
      for(k = 1; k <= logaritmo_n_2; ++k){
        fmpz_powm_ui(&pot_mod, &n, k, &r);
          if(fmpz_is_one(&pot_mod)){
            teste = false;
            break;
          }
      }
      if(teste) break;
    }
    else return false; //Etapa 3
    ++r;
  } //Fim da Etapa 2, Etapa 3 e Etapa 4
  //Etapa 5
  fmpz phi, coeficiente;
  fmpz_euler_phi(&phi, &r);
  double phi_d = fmpz_get_d(&phi);
  long constante = floor(sqrt(phi_d)*logaritmo_n);
  slong r_long = fmpz_get_si(&r);
  fmpz_mod_poly_t lado_direito, lado_esquerdo, modulo;
  fmpz_mod_poly_init(lado_direito, &n);
```

```
39    fmpz_mod_poly_init(lado_esquerdo, &n);
40    fmpz_mod_poly_init(modulo, &n);
41    fmpz n_1 = n - 1;
42    fmpz_mod_poly_set_coeff_ui(modulo, r_long, 1);
43    fmpz_mod_poly_set_coeff_fmpz(modulo, 0, &n_1);
44    fmpz n_mod_r = n % r;
45    slong n_mod_r_long = fmpz_get_si(&n_mod_r);
46    fmpz_mod_poly_set_coeff_ui(lado_direito, fmpz_get_si(&n_mod_r), 1);
47    for(slong a = 1; a <= constante; ++a){
48      fmpz_mod_poly_set_coeff_ui(lado_esquerdo, 1, 1);
49      fmpz_mod_poly_set_coeff_ui(lado_esquerdo, 0, a);
50      fmpz_mod_poly_powmod_fmpz_binexp(lado_esquerdo, lado_esquerdo, &n,
      modulo);
51      fmpz_mod_poly_get_coeff_fmpz(&coeficiente, lado_esquerdo, 0);
52      coeficiente -= a;
53      fmpz_mod_poly_set_coeff_fmpz(lado_esquerdo, 0, &coeficiente);
54      if(!fmpz_mod_poly_equal(lado_esquerdo, lado_direito)) return false;
55      fmpz_mod_poly_realloc(lado_esquerdo, 0);
56    }
57    return true; //Fim Etapa 5
58  }
```

A etapa 1 desta implementação é exatamente igual a etapa 1 da implementação anterior. O *FLINT*, assim como o *NTL*, não oferece uma função generalizada para qualquer inteiro para esta etapa e a performance da função *mpz_perfect_power()* do *GMP* é excelente.

Nas etapas 2, 3 e 4 podemos ver variáveis com o tipo *fmpz*, elas são do módulo de inteiros de precisão arbitrária do *FLINT* (que utilizam como base os tipos de *C* ou do tipo *mpz* do *GMP*). Assim como no *NTL*, algumas variáveis foram declaradas com o tipo *long* e *double*, a causa disso vem de uma restrição da etapa 5 que a biblioteca impõe e será explicada em detalhes na próxima seção (junto com a análise da restrição do valor de entrada n).

Na linha 16, podemos ver a função *fmpz_gcd* que responde o valor do MDC dos dois últimos parâmetros e coloca a resposta no primeiro parâmetro que é passado por referência, internamente a biblioteca utiliza a função de MDC do *GMP*, ou em caso dos números serem pequenos, a biblioteca utiliza o MDC Euclidiano. A função *fmpz_is_one()* é uma simples verificação para saber se o *fpmz* de entrada é 1. Na linha 20 foi utilizado outra função *fmpz_powm_ui()* que é basicamente a potência modular binária apresentada. Note que o módulo *fmpzxx* da biblioteca fornece uma interface opcional para substituir algumas funções com operadores da própria, por exemplo, substituir a função *fmpz_is_one()* pelo operador $==$ do *C*.

Na etapa 5, utilizamos o segundo módulo *fmpz_mod_poly* para lidar com polinômios de coeficientes inteiros de precisão arbitrária no anel $\mathbb{Z}_n$. Na linha 33 utilizamos a con-

veniente função *fmpz_euler_phi()* que a biblioteca nos fornece para calcular o Totiente de Euler de r. Observe que r é convertido para o tipo *long* na linha 36, a limitação presente no *NTL* também existe no *FLINT*: o grau máximo do polinômio do módulo deve ser uma variável do tipo *long*. As funções *fmpz_mod_poly_init()* utilizada nas linhas 38, 39 e 40 inicializam o polinômio para os coeficientes serem calculados em modularidade n.

As funções *fmpz_mod_poly_set_coeff_ui()* e *fmpz_mod_poly_set_coeff_mpz()* atribuem valores nos coeficientes do polinômio. Durante as iterações da etapa 5, na linha 50, é utilizada uma função chamada *fmpz_mod_poly_powmod_fmpz_binexp()*, esta função recebe como primeiro parâmetro passado por referência um *fmpz_mod_poly_t* para receber o resultado de $Z[x]^k \mod Q[x]$, onde $Z[x]$ é o segundo parâmetro do tipo *fmpz_mod_poly_t*, k o terceiro do tipo *fmpz_t* e $Q[x]$ o quarto do tipo *fmpz_mod_poly_t*. A função utiliza o mesmo algoritmo de exponenciação binária apresentado neste projeto e as multiplicações são feitas utilizando a versão de multiplicação de polinômios do algoritmo de *Karatsuba*. Assim como no *NTL*, essas multiplicações poderiam ser mais rápidas se a biblioteca utiliza-se o algoritmo de *Schönhage–Strassen* para multiplicação de polinômios. Para finalizar a função *fmpz_mod_poly_realloc()* faz com o que todos os coeficientes do polinômio tenham o valor zero.

4.2.3 Versão utilizando PARI

PARI/GP é um sistema especializado com funções úteis para Teoria dos Números, é capaz de realizar computações de tipo recursivas com uma alta performance utilizando uma linguagem de script com tipos de dados familiares para matemáticos. Neste projeto vamos utilizar o PARI que é a versão para *C* em biblioteca do sistema *PARI/GP*. A biblioteca foi feita com a intenção de fornecer as mesmas funcionalidades do *PARI/GP*, mas com uma performance mais alta através da linguagem *C* e com recursos para a programação em baixo nível.

A documentação da biblioteca deixa muito a desejar, poucos exemplos de códigos são fornecidos e em geral a utilização da biblioteca é pouco intuitiva. O ponto forte da biblioteca é que ela oferece módulos úteis para a implementação do *AKS*, como inteiros de precisão arbitrária e polinômios com coeficientes de precisão arbitrária. Vale notar que o *PARI* implementa inteiros de precisão arbitrária usando código próprio, mas durante sua instalação o usuário pode optar por utilizar a biblioteca *GMP* para melhorar sua performance. Se o usuário utilizar esta opção, a biblioteca altera sua estrutura interna para substituir automaticamente as operações de precisão arbitrária do *GMP*. De acordo com a documentação da biblioteca a performance geral se torna aproximadamente 10% mais eficiente. Durante sua instalação, o usuário também pode optar pela opção de *"Tunar"* a biblioteca que essencialmente a torna mais rápida transformando parte do

código interno em algo especializado para a arquitetura da máquina que está utilizando ela. A documentação recomenda que esta opção seja utilizada somente caso o usuário também tenha uma instalação do *GMP* com sua opção *"Tunar"* ativada. Os autores do *PARI* dizem que esta opção aumenta a performance do sistema em aproximadamente 15%.

A página oficial do *PARI* com sua documentação e código é: ⟨http://pari.math.u-bordeaux.fr/⟩. A versão instalada no recurso computacional deste projeto não utiliza a opção *"Tunar"* do *PARI* e nem a do *GMP*. A opção de utilizar os recursos do *GMP* no *PARI* foi ativada. A implementação abaixo foi feita utilizando a versão 2.7.5 do *PARI* lançada em 9 de novembro de 2015.

Listagem 4.15 - Teste de primalidade AKS com PARI e GMP

```
bool aks_pari(const string _n){
  pari_sp av = avma, av2;
  GEN n = gp_read_str(_n.c_str());
  if(Z_isanypower(n, NULL)){  //Etapa 1
    avma = av;
    return false;
  } //Fim Etapa 1
  //Etapas 2, 3 e 4
  bool teste;
  long k;
  GEN r = gen_2, n_REAL = itor(n, 3);
  double logaritmo_n =  dbllog2r(n_REAL);
  double logaritmo_n_2 = logaritmo_n*logaritmo_n;
  while(1){
    if(gequal(r, n)){  //Etapa 4
      avma = av;
      return true;
    }
    if(gequal1(gcdii(n, r))){
      teste = true;
      for(k = 1; k <= logaritmo_n_2; k++){
        if(gequal1(Fp_powu(n, k, r))){
          teste = false;
          break;
        }
      }
      if(teste) break;
    }
    else{ //Etapa 3
      avma = av;
      return false;
    }
    r = gaddgs(r, 1);
```

```
34    } //Fim da Etapa 2, Etapa 3 e Etapa 4
35    //Etapa 5
36    long r_long = gtolong(r);
37    long phi;
38    phi = eulerphiu(r_long);
39    long constante = floor(sqrt(phi)*logaritmo_n); //piso de raiz(phi(n))
       log2 (n)
40    GEN n_mod_r = Fp_red(n, r);
41    long n_mod_r_long = gtolong(n_mod_r);
42    GEN lado_direito, lado_esquerdo;
43    lado_direito = cgetg(n_mod_r_long + 3, t_POL);
44    lado_direito[1] = evalvarn(0);
45    for(ulong x = 2; x <= n_mod_r_long+2; ++x) gel(lado_direito, x) = gen_0;
46    gel(lado_direito, n_mod_r_long + 2) = gen_1; //x^(r%n)
47    lado_direito = normalizepol(lado_direito);
48
49    for(long a = 1; a <= constante; ++a){
50      av2 = avma;
51      lado_esquerdo = mkpoln(2, gen_1, gen_0);
52      gel(lado_esquerdo, 2) = stoi(a); //x + a
53
54      lado_esquerdo = potencia_modular_polinomial(lado_esquerdo, n, r_long,
       ceil(logaritmo_n));
55      gel(lado_esquerdo, 2) = subis(gel(lado_esquerdo, 2), a);
56
57      if(degree(lado_esquerdo) != degree(lado_direito)){
58        avma = av;
59        return false;
60      }
61      for(ulong x = 0; x <= degree(lado_esquerdo); ++x){
62        if(!gequal(gel(lado_esquerdo, x+2), gel(lado_direito, x+2))){
63          avma = av;
64          return false;
65        }
66      }
67
68      avma = av2;
69    }
70
71    avma = av;
72    return true;
73    //Fim Etapa 5
74
75  }
```

Antes de tudo, o programador deve saber que o *PARI* utiliza um espaço de memória iniciado pela função *pari_init()* que cria os seguintes componentes da biblioteca: stack,

heap, váriaveis globais e uma tabela de primos. Esse componentes podem ser utilizados livremente pelo programador (fortemente recomendado utilizar), o código da própria biblioteca utiliza esses componentes. Se na função principal do programa esta área não for inicializada, então o código acima não irá funcionar.

As funções internas do *PARI* realizam automaticamente a coleta do próprio lixo de memória, mas esta coleta de lixo não é automática para o programador. A biblioteca fornece algumas funções para realizar a tarefa (que é extremamente necessária), porém o código acima só utiliza uma dessas ferramentas por simplicidade, por este motivo a implementação acima não é necessariamente eficiente em termos de uso de memória (apesar de ser rápida). A função acima libera normalmente a memória utilizada após a sua execução. A biblioteca recomenda que a área de memória adicional seja inicializada com pelo menos *500KB*, nós recomendamos *5MB* no mínimo para o *AKS*. vale notar também que a tabela de primos acelera a computação interna do *PARI*, é recomendado manter o mesmo padrão da documentação que consiste em iniciar esta tabela com todos os primos até 2^{16}.

Como pode ser visto na linha 4, há um ponteiro que recebe o endereço do começo da *stack* do *PARI*, toda vez que a função *aks_pari()* retorna o seu valor o endereço é recuperado e todas as variáveis que não são mais utilizadas podem ter seu espaço de memoria reutilizado dentro da *stack* do *PARI*. O código acima não utiliza a *heap* do sistema, mas alguma função interna pode utilizar (veja a documentação para mais detalhes sobre como alocar variáveis na *heap* do *PARI*).

O único tipo de várivel que o programador utiliza é o *GEN*, este é outro ponto importante que o programador deve saber sobre a utilização da biblioteca. Este tipo é um ponteiro para *long long int* utilizado pelo *PARI* que implementa todos os subtipos que a biblioteca fornece. O subtipo mais utilizado na implementação é o inteiro de precisão arbitrária, mas na etapa 5 utilizamos o subtipo que implementa polinômios com coeficientes inteiros de precisão arbitrária. Mantenha em mente que a estrutura interna da área de memória que começa a partir de um ponteiro *GEN* vária bastante de acordo com o subtipo da mesma, a biblioteca não foi construída para o programador utilizar com frequência acessos de baixo nível a essa área. Recomendamos sempre verificar a documentação da biblioteca para saber como esta área funciona para cada subtipo, e sempre tentar utilizar as funções que a biblioteca fornece para montar cada variável do programa, somente utilizando macros de acesso de baixo nível caso não exista outra opção. Também note que em algumas partes dessa implementação utilizamos alguns valores chamados de *gen_*, estes valores são constantes do tipo *GEN* que implementam algum subtipo com valores bases comuns para serem utilizados. Eles estão alocados na área de memória inicializada do *PARI* e podem ser vistos analogamente como váriaveis globais. Algum desses valores são *gen_1* (um GEN de valor 1) e *gen_m1* (GEN de valor -1), a lista de todas as constantes disponíveis na memória pode ser vista na documentação da biblioteca.

As linhas 2 e 3 evitam que o programador que escreve a função *int_main()* tenha de usar as funções de entrada e saída específicas da biblioteca para passar o valor de n para o *AKS*, a execução dessas linhas transformam a *string* com o número de entrada para um *GEN* de subtipo inteiro com precisão arbitrária sem validação de entrada. A linha 4 já foi explicada anteriormente, a função dela é realizar a coleta de lixo na *stack* da biblioteca. Na linha 6, podemos ver que a biblioteca *PARI* fornece uma implementação para determinar se um número é uma potência perfeita, esta função é ideal para o *AKS* e torna o código da etapa 1 simples. O argumento *NULL* desta função pode ser substituído por um *GEN* para receber o valor da base que forma n como uma potência perfeita (ou receber 0 caso este valor não exista).

Nas etapas 2, 3 e 4, algumas váriaveis são inicializadas com o tipo *long* e *double*. O efeito de todas essas váriaveis sobre a limitação máxima de n vão ser discutidas na próxima seção. Nas linhas 13 e 14, as funções *itor()* e *dbllog2r()* são simples conversões entre tipos. As funções *gequal()* e *gequal1()* utilizadas dentro do *loop* são comparações onde a primeira responde se dois valores *GEN* são iguais e a segunda responde se o *GEN* tem valor igual a 1. Nas linhas 18 e 32, o ponteiro *avma* sempre aponta para o topo da *stack* e deve retornar para o local anterior da execução da função pela coleta de lixo. A função *gaddgs()* utilizada na linha 35 responde um *GEN* que é o resultado da soma dos elementos recebidos como argumento. A função *gcdii()* retorna um *GEN* com o valor do MDC dos argumentos que também são do tipo *GEN*, o algoritmo utilizado para realizar esta operação não foi especificado. Finalmente a função da linha 24, *Fp_powu*, é a potência modular que recebe parâmetros do tipo *GEN* e responde um *GEN*.

Na etapa 5, temos uma conversão de *GEN* para *long* na linha 38, o *PARI*, assim como as duas últimas bibliotecas, impõe uma limitação no grau de seu polinômio, então pelo mesmo motivo citado anteriormente para as duas bibliotecas o r tem de ser do tipo *long*. Na linha 40, utilizamos uma função para calcular o Totiente de Euler fornecida pela biblioteca. A função *Fp_red()* retorna o primeiro *GEN* módulo o segundo. Na linha 43, temos outra simples conversão para o tipo *long* que não afeta a limitação da entrada (detalhes constam na próxima seção). A função utilizada na linha 45, *cgetg()*, é uma chamada de baixo nível para criar um *GEN* com um determinado subtipo com uma determinada área de memória pré-alocada, o *GEN* é alocado na *stack* da biblioteca.

Na linha seguinte, definidos uma letra qualquer para representar o polinômio internamente, a letra é irrelevante nesta implementação, isto é feito com a chamada *evalvarn(0) que faz o polinômio ser representado pela letra x*. A macro *gel()* é uma chamada de baixo nível para acessar a estrutura do *GEN* e definir valores para este. Finalmente, a chamada *normalizepol()* garante que a estrutura está pronta para ser utilizada pelas funções internas da biblioteca. Lembrando que este tipo de utilização de baixo nível da biblioteca é altamente não recomendado, mas este é um caso em que esta utilização foi necessária, para detalhes da estrutura veja a documentação do *PARI*.

Logo após, começamos as iterações da etapa 5, nas linhas 52 e 70 temos uma simples coleta de lixo na *stack* da biblioteca para não haver estouro de memória, pois a função da linha 56 programada para esta aplicação não faz sua própria coleta. A função *mkpol()*, da linha 53, inicializa o polinômio (de acordo com a documentação a função inicializa de maneira "suja"a variável, mas se esta utilizar somente constantes guardadas na memória da biblioteca, então é seguro utilizar ela). Na linha 54 podemos ver novamente a utilização da macro de baixo nível com uma conversão simples fornecida pela biblioteca que arruma o coeficiente do polinômio também definido como *GEN*. A macro *degree* retorna o grau do polinômio e o código das linhas 59 até 68, basicamente verificam se os polinômios são iguais utilizando macros de acesso de baixo nível da biblioteca. Finalmente a função *potencia_modular_polinomial()* realiza a computação de potência polinomial no anel quociente $\mathbb{Z}_n[X]/X^r - 1$ com o exato algoritmo apresentado neste capítulo na implementação simples, só que desta vez utilizando o código da biblioteca *PARI*. A seguir temos o código desta função:

Listagem 4.16 - Potência modular polinomial com PARI

```
1  GEN potencia_modular_polinomial(const GEN &base, const GEN &mod, const long
       &poly_mod, const ulong logaritmo_n){
2      GEN resto = pol_1(0);
3      GEN pow2i, ndivpow, truncate, bit;
4      for(ulong i = logaritmo_n; i > 0; i--){
5          resto = FpX_sqr(resto, mod);
6          resto =  ZX_mod_Xnm1(resto, poly_mod);
7          pow2i = powuu(2, i-1);
8          ndivpow = rdivii(mod, pow2i, 3);
9          truncate = floorr(ndivpow);
10         bit = Fp_red(truncate, gen_2);
11         if(gequal1(bit)){
12             resto = FpX_mul(resto, base, mod);
13             resto = ZX_mod_Xnm1(resto, poly_mod);
14         }
15     }
16   resto = RgX_to_FpX(resto, mod); //Reduz o polinomio para Zn[x]
17     return resto;
18 }
```

Na linha 2, utilizamos uma função que constrói de forma limpa um polinômio de valor x^0 utilizando a letra x na representação do polinômio (a escolha da letra é irrelevante para a implementação desde que seja a mesma para todos os polinômios). Todas as variáveis do tipo *GEN* declaradas na linha 3 são auxiliares para obter o *i'ésimo bit* do inteiro de precisão arbitrária n presentes nas linhas 7 a 10.

Na linha 5, a função *FpX_sqr()* realiza a computação do polinômio elevado ao quadrado no anel $\mathbb{Z}_n$. A função da linha seguinte computa o módulo do polinômio an-

terior com $X^r - 1$. A última função utilizada não citada anteriormente é a da linha 12, o *FpX_mul*, que computa a multiplicação do polinômio *resto* com o polinômio base no anel $\mathbb{Z}_n$. O método especificado pela documentação da biblioteca para multiplicar os polinômios é o algoritmo de *Karatsuba*, este é outro sistema que poderia utilizar uma implementação do *Schönhage–Strassen* para multiplicação de polinômios. O código completo da implementação acima do *AKS* pode ser vista no anexo desse trabalho, junto com todas as outras. Estas implementações apresentam comentários detalhados e um exemplo de *int_main()*.

4.3 Limite das implementações

Decidimos colocar o estudo da limitação de cada código nesta pequena seção separada, pois estas implementações têm limites similares. Como pode ser visto claramente, nenhuma das quatro implementações conta com validações de entrada. Deixamos a validação de entrada por conta da aplicação que utilizar as implementações caso necessário. Todas as implementações, com exceção da simples, comportam inteiros de precisão arbitrária como parâmetro, mas estas não suportam qualquer inteiro pela limitação que as bibliotecas impõe no grau dos polinômios para computação da etapa 5.

Teorema 4.3.1. *A implementação simples do AKS responde qualquer entrada* $n \leq 2^{32} - 1$

Demonstração. A implementação recebe como parâmetro de entrada n inteiro do tipo *unsigned int*, váriaveis deste tipo comportam qualquer valor de 0 até $2^{32} - 1$. A etapa 1 desta implementação não impõe nenhuma restrição adicional, pois ela utiliza váriaveis do tipo *double* que comportam reais de precisão dupla, elas podem armazenar números extremamente grandes perdendo somente precisão da mantissa que não afeta o resultado, as outras váriaveis são de precisão única do mesmo tipo da entrada e nunca devem ser maior que a mesma, pois o limite superior da pesquisa binária é o próprio n.

O mesmo acontece nas etapas 2, 3 e 4: as váriaveis declaradas com precisão simples nunca devem ser maiores que a entrada, pois a implementação da etapa 4 garante que a iteração de r corrente seja menor que n, caso contrário a implementação responde primo imediatamente. As váriaveis de precisão dupla não podem ter valores maiores que $\lg^2 n$. Todas as funções auxiliares nessas etapas não retornam valores maiores que n, e internamente não utilizam valores maiores que n, com exceção da potência modular. Mesmo assim, a potência modular consegue realizar todas as computações até $2^{32} - 1$, pois elas utilizam internamente váriaveis do tipo *unsigned long long int*.

O Totiente de Euler de um número retorna no máximo o próprio valor menos um, portanto o valor da variável que guarda o Totiente de Euler de r vale no máximo $r - 1$, então esta variável não oferece nenhuma restrição adicional para a entrada. É fácil verificar

que a constante limite da iteração de a para qualquer entrada dentro da restrição atual nunca é maior que 2^{18} com uma simples substituição no limite $\sqrt{\phi(r)}\lg n \cong \sqrt{\lg^5 n}\lg n = \lg^{3.5} n$ baseada no lema 3.2.3.

A implementação da classe polinômio assume um grau máximo do tipo *unsigned int* que é o suficiente para executar a etapa, pois r é o grau máximo que estes polinômios vão atingir, devido a natureza do algoritmo em computar os polinômios no anel quociente $\mathbb{Z}_n/X^r - 1$. Finalmente para decidir a restrição final só faltou verificar as computações feitas na função *potencia_modular_polinomial()*.

Cada polinômio da classe comporta coeficientes inteiros de precisão dupla, como o polinômio é sempre escrito no anel $\mathbb{Z}_n$, esses coeficientes não devem ter valores maiores que n assim a restrição inicial continua válida. A função conta com dois auxiliares, um deles é o auxiliar de precisão simples que guarda o valor de um dos coeficientes do polinômio computado durante uma iteração deste algoritmo, como estes coeficientes não devem ter valores acima de n, a restrição inicial permanece. A segunda variável auxiliar, que é de precisão dupla, comporta algumas computações durante o algoritmo, essas computações envolvem uma multiplicação de dois coeficientes do polinômio corrente e uma soma com o valor anterior que estava anteriormente nele. Como já sabemos, os coeficientes são certamente menores que um número de *32 bits*, portanto quando realizamos a multiplicação o resultado vale no máximo um certo número menor que 2^{64}. Após esta operação, realizamos uma soma com o número anterior armazenado dentro variável auxiliar que também é certamente menor que *32 bits*, por causa da modularidade com n. Depois desta computação, a variável auxiliar deve guardar um valor menor que *64 bits*, uma operação de módulo com n é realizada antes de uma nova multiplicação, portanto nenhum $overflow$ pode acontecer. Concluindo a análise da função que também não limita a restrição, portanto a restrição de n está em $2^{32} - 1$. Um pequeno teste prático com a implementação com o número primo $2^{32} - 5$ leva 240 segundos no recurso computacional deste projeto. □

Teorema 4.3.2. *As três implementações otimizadas do AKS respondem qualquer entrada* $n \leq 2^{\sqrt[5]{2^{31}-1}}$

Demonstração. As três implementações comportam entradas n inteiras de precisão arbitrária somente limitada pela memória do recurso computacional. A etapa 1 nas implementações feitas com *NTL* e *FLINT*, utilizam uma função específica do *GMP* que comporta inteiros de precisão arbitrária e não causa nenhum efeito na restrição inicial. O mesmo acontece na implementação feita com *PARI*, pois este utiliza uma função da própria biblioteca.

Nas etapas 2, 3 e 4 de todas as implementações, a variável que guarda o valor da iteração r é um inteiro de precisão arbitrária. O retorno das funções das três bibliotecas que computam o MDC e a potência modular comportam saídas de precisão arbitrária.

Os auxiliares que guardam os valores de saída temporariamente dessas funções estão declaradas como inteiros de precisão arbitrária. As funções de comparação utilizadas dentro dos condicionais comportam inteiros de precisão arbitrária. Portando nenhuma das funções e váriaveis auxiliares citadas até agora impõe uma restrição na entrada.

As duas variáveis auxiliares do tipo *double* na etapa 2 impõe o primeiro limite até agora. A primeira variável guarda o valor de $\lg n$ e a segunda guarda o valor de $\lg^2 n$ criando a seguinte restrição: $\lg^2 n \leq lim$ onde *lim* é o limite máximo da variável *double* que é especificada na maioria dos sistemas como 1.79769×10^{308}. A variável que guarda o valor corrente de k, o resultado da ordem módulo r, cria uma segunda restrição para a entrada. Em todas as três implementações esta variável está declarada com o tipo *long* que comporta o valor máximo de $2^{31}-1$, no qual impõe a seguinte restrição: $\lg^2 n \leq 2^{31}-1$ que pode ser escrita como $n \leq 2^{\sqrt{2^{31}-1}}$, pois a iteração k vale no máximo $\lg^2 n$ durante toda a execução do algoritmo.

Na etapa 5, as três bibliotecas suportam polinômios de coeficientes inteiros de precisão arbitrária, estes coeficientes não afetam a última restrição, mas devido a restrição no grau máximo que as três bibliotecas têm, a variável r deve valer no no máximo $2^{31}-1$, pois esta variável é o grau máximo que estes polinômios podem atingir. Como demonstrado no capítulo anterior pelo Lema 3.2.3, o valor de r é no máximo $\lg^5 n$, então temos uma nova restrição na entrada: $\lg^5 n \leq 2^{31}-1$ que pode ser escrita como $n \leq 2^{\sqrt[5]{2^{31}-1}}$.

A implementação que utiliza *FLINT* conta com sua própria função para determinar o Totiente de Euler de r, como a função só trabalha com inteiros de precisão arbitrária, ela não afeta a restrição. Já as outras duas implementações trabalham com precisão única, mas estas também não afetam a restrição inicial, pois o Totiente de Euler retorna no máximo o valor de r menos um.

A variável que guarda o limite de a é do tipo *long*, com o Lema 3.2.3 temos a seguinte restrição: $\sqrt{\lg^5 n - 1} \times \lg n \leq 2^{31}-1$ para a entrada que não afeta a restrição anterior delimitada por r. As três implementações têm uma variável auxiliar que guarda o valor de $n \mod r$ do tipo *long*, ela claramente não afeta a restrição anterior. Para finalizar, no restante do código, as três implementações utilizam somente funções para construção de variáveis para comportar os polinômios necessários para a computação da etapa 5 que não criam nenhuma restrição em n, também utilizam funções para computar a potência polinomial modular que não afetam a restrição anterior, pois as três bibliotecas implementam polinômios com coeficientes de precisão arbitrária. Então temos que a restrição final é $n \leq 2^{\sqrt[5]{2^{31}-1}}$, aproximadamente 23 dígitos. □

4.4 **Análise da performance**

Durante a implementação das versões do teste de primalidade *AKS*, vários testes foram executados para verificar que as implementações não continham erros. Com a implementação simples tivemos sucesso em determinar a primalidade dos 3220000 primeiros números inteiros sem que nenhum erro fosse detectado. Os testes que serão apresentados nesta seção tem por objetivo analisar a performance do algoritmo na prática, comparando a performance de cada uma das bibliotecas utilizadas. Vale notar que estes testes são os mesmos do Capítulo 2 com números considerados pequenos quando comparados aos utilizados em sistemas reais. Nesta análise também estamos utilizando a melhor implementação que encontramos do *AKS* vindo de outros autores, escolhemos uma implementação correta que utiliza a mesma versão do *AKS* apresentada neste projeto, ela pode ser vista no projeto de bacharelado de seu autor em (JIN, 2005), a implementação utiliza a biblioteca *NTL*. O código utilizado do projeto é a função *AKSv3()*, removemos a função extra que utiliza um grande *array* de primos pré-computado escrito diretamente no código que impede a maioria das entradas executem a etapa 5.

O recurso computacional usado para nos testes é munido de um processador *AMD FX-8350 4.00 GHz* com 16 GB de memória *RAM* executando o sistema operacional *Ubunto* 15.10. O compilador *C++* utilizado em todos os testes faz parte do *GNU Compiler Collection* 4.8.4. Todas as bibliotecas utilizadas foram construídas sem argumentos de otimização, em especial o *PARI* foi configurado para utilizar internamente as funções e a arquitetura de variáveis de precisão arbitrária do *GMP*. Sabe-se que as outras bibliotecas sempre utilizam o *GMP* em seu código normal. Todas as implementações foram compiladas utilizando o parâmetro de otimização *-Ofast* para os testes de performance.

Sabe-se que o *GMP* utiliza o algoritmo de *Karatsuba* para multiplicação de inteiros muito grandes, a biblioteca *NTL* utiliza o *FFT* (Fast Fourier Transform) com *CRT* (Chinese Ramainder Theorem), ou o algoritmo de *Karatsuba*, ou o clássico algoritmo quadrático para a aritmética com polinômios, a escolha é feita com uma fraca heurística. As bibliotecas *FLINT* e *PARI* utilizam o algoritmo de *Karatsuba* para multiplicação de polinômios.

Vamos apresentar os testes divididos em três arquivos, exatamente os mesmos do Capítulo 2. Todas as implementações foram executadas dez vezes para cada arquivo em um teste sequencial, o tempo apresentado é a média aritmética dessas dez execuções. O tempo foi cronometrado utilizando a biblioteca *Chrono* do *C++11* com *Steady Time*. Para cada execução foi disponibilizado exatamente um núcleo do processador do recurso computacional totalmente dedicado ao teste, a quantidade de memória não foi limitada. O tempo de execução cronometrado consiste no tempo em que a função *AKS* de cada implementação estava no processador, excluindo qualquer outra inicialização em *int_main()*. Vale notar que a inicialização do *PARI* para a alocação de sua *stack* e *heap* não foi

cronometrado, o tempo de inicialização é desprezível.

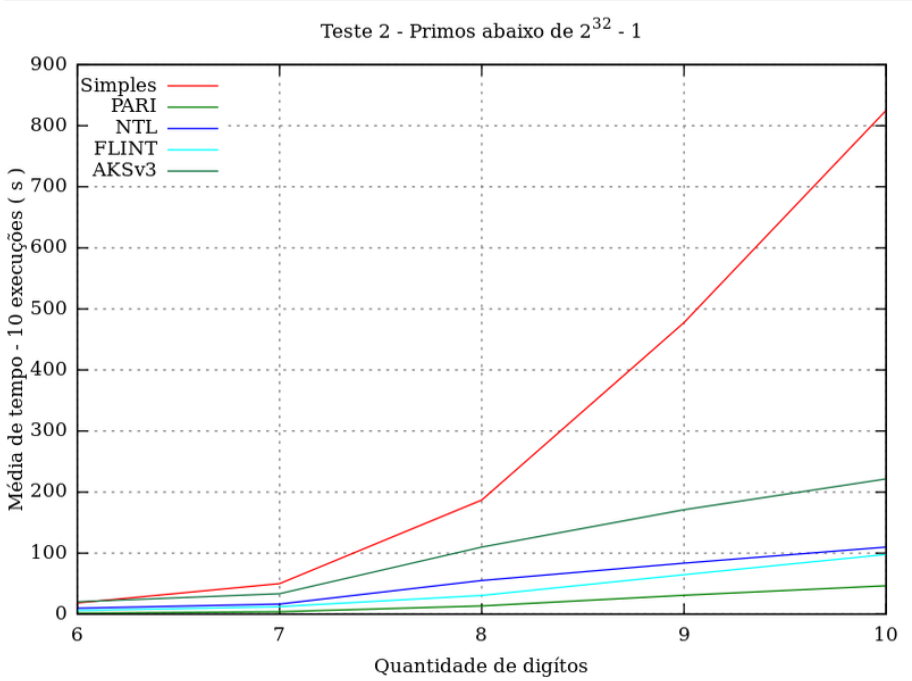
- Arquivo 1 - Consiste dos dez mil primeiros naturais, dentre esses inteiros sabe-se que existem 1229 primos.
- Arquivo 2 - Consiste em determinados primos escolhidos aleatoriamente. Foram utilizados primos de 6 até 10 dígitos abaixo de 2^{31}, formando grupos de cinco primos para cada digito com um total de 25 primos no teste.
- Arquivo 3 - Consiste de primos aleatórios acima de 2^{31} de 10 até 16 dígitos, formando um grupo de 5 primos para cada dígito com um total de 35 primos no arquivo. A entrada foi executada somente com as implementações otimizadas deste projeto, pois as outras não comportam entradas tão grandes. Como o tempo de execução desse arquivo é alto, somente 5 execuções foram feitas.

Todos os arquivos acima podem ser vistos no apêndice deste projeto.

Tabela 2 - Testes de performance - AKS

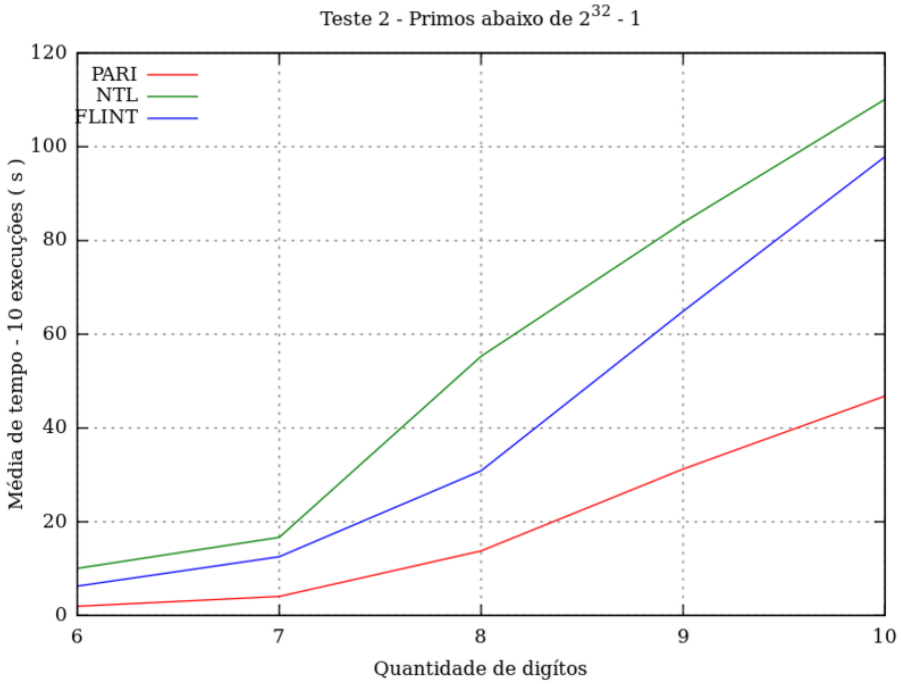
Teste 1 - 10000 Inteiros	
Implementação	Média 10 execuções (s)
AKS Simples	132.852
AKS NTL	118.2
AKS FLINT	125.376
AKS PARI	**37.2695**
Tong Jim AKSv3	233.059
Teste 2 - Primos abaixo de $2^{32} - 1$	
Implementação	Média de 10 execuções (s)
AKS Simples	1558.5753
AKS NTL	276.0183
AKS FLINT	212.52101
AKS PARI	**97.98024**
Tong Jim AKSv3	557.0958
Teste 3 - Primos acima de $2^{32} - 1$	
Implementação	Média de 5 Execuções (s)
AKS NTL	8634.224
AKS FLINT	5969.239
AKS PARI	**2776.0021**

Figura 11 - Teste 2 - Todos



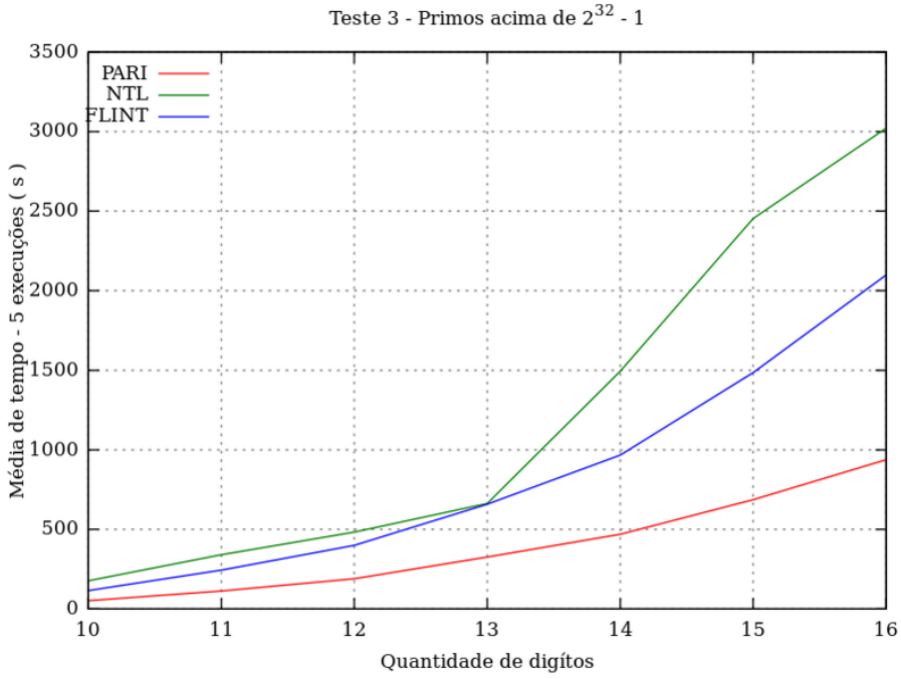
Legenda: Execução sequencial para 5 primos de cada dígito.

Figura 12 - Teste 2 - Somente otimizados.



Legenda: Execução sequencial para 5 primos de cada dígito.

Figura 13 - Teste 3 - Somente otimizados.



Legenda: Execução sequencial para 5 primos de cada dígito.

5 CONCLUSÃO

5.1 O problema de determinação de primos na prática

5.1.1 Resolvendo o problema

Através dos testes de performance concluímos que, se a aplicação somente trabalha com números primos pequenos (até 2^{32-1} por exemplo), a abordagem do tipo força bruta é o suficiente para a determinação da primalidade de alguns poucos inteiros. O Crivo de Eratóstenes torna-se a melhor opção caso seja interessante a determinação de todos os primos em um intervalo pequeno de números.

Quando há a necessidade de trabalhar com a primalidade de inteiros de precisão arbitrária, como é o de aplicações que lidam com primos de 512 *bits* (aproximadamente 154 dígitos), 1024 ou até mesmo de 2048 *bits*, as abordagens de força bruta e o Crivo de Eratóstenes não são viáveis. Uma estratégia viável é a combinação dos testes de primalidade abordados neste projeto.

Geralmente a estratégia utilizada quando se trata de gerar um número primo, é realizar um teste de primalidade probabilístico em entradas aleatórias até que o algoritmo encontre uma entrada na qual a saída é provavelmente primo. Após determinar um inteiro provavelmente primo, deve-se atestar a primalidade deste número com um teste determinístico. Através dos resultados deste projeto conseguimos garantir que esta solução é suficientemente boa, uma vez que os testes de primalidade probabilísticos identificam números compostos rapidamente e em geral executam com melhor performance que os determinísticos, portanto são ótimos filtros para outros testes.

Os sistemas que oferecem soluções de determinação de primos com precisão arbitrária, geralmente têm uma performance muito boa com ajuda de poderosas pré-computações e otimizações, mas o maior problema é que muitas delas não oferecem a computação do problema com os melhores algoritmos conhecidos por causa da falta de implementações eficientes desses testes. Esse é o caso da biblioteca *GMP*, utilizada neste projeto e também em diversas aplicações no mundo todo.

A biblioteca *GMP* fornece funções para a determinação de prováveis primos através da utilização do algoritmos de *Miller-Rabin*, tal como a biblioteca *NTL*. Contudo, essas duas bibliotecas possuem padrões diferentes de sugestões no que diz respeito à quantidade de vezes que deve-se iterar no algoritmo a fim de obter um resultado com uma probabilidade desejada. É importante destacarmos que a utilização do algoritmo de *Baillie-PSW* tem performance superior a 5 iterações do algoritmo de Miller-Rabin, enquanto as sugestões das bibliotecas são de 10 iterações para o *NTL* e de 15 a 50 iterações para o *GMP*.

Já os sistemas que oferecem a solução não probabilística do problema, isto é, oferecem testes onde a saída é um número primo provado, geralmente utilizam o algoritmo *APR-CL*. Este algoritmo é determinístico, porém não tem complexidade assintótica polinomial sendo ultrapassado pelo *AKS* e pelo método *ECPP* para provar primalidade.

Um exemplo de sistema que poderia utilizar uma implementação eficiente do *ECPP* (ou até mesmo do *AKS*), é o próprio *PARI*. A biblioteca implementa sua função de determinação de primos utilizando o algoritmo *Baillie-PSW*. Se o inteiro não for detectado como composto, a função prova a primalidade do número utilizando o *APR-CL*. Outra biblioteca que também faz isso é o *FLINT*, que tem uma implementação do *Miller-Rabin*, do *Baillie-PSW* e uma implementação que tenta provar a primalidade do número por divisões sucessivas. Os autores da biblioteca sugerem a utilização da função que prova a primalidade da entrada, somente mediante a execução prévia do *Miller-Rabin* ou do *Baillie-PSW* como filtro. O problema é que mesmo filtrando vários números compostos aleatórios com os testes probabilísticos, o método de divisões sucessivas está extremamente ultrapassado para provar a primalidade da entrada.

O maior problema de utilizar os métodos mencionados para sistemas de prova de primalidade, está na dificuldade de implementa-los, essas dificuldades vão ser o foco da próxima seção.

5.1.2 O AKS e o ECPP na prática

Computar o *AKS* não é uma tarefa fácil, como demonstrado nas implementações do Capítulo 4. As três implementações otimizadas tem restrições que não permitem a entrada ter mais de 23 dígitos, construir um código sem essa restrição requer o possível uso de outra linguagem de programação, outro sistema ou uma implementação eficiente das funções que computam as operações no anel de polinômios $\mathbb{Z}_n[x]$ de grau arbitrário. Os autores deste trabalho não conseguiram encontrar tal implementação, nem mesmo em artigos científicos, foram utilizadas as ferramentas de busca mais comuns da *Internet* para este fim.

Além disso, o *AKS* realiza operações que precisam de uma certa precisão com números reais, caso essas operações falhem, elas podem alterar o resultado sobre a primalidade da entrada. Vários exemplos podem ser facilmente encontrados em implementações de outros autores, sendo comum encontrar implementações que erram a primalidade de números pequenos por causa de truncamentos ou o uso insuficiente de casas decimais na representação do número real. Contornar o problema da precisão não é uma tarefa fácil e realizar operações com precisão excessiva prejudica a performance da implementação severamente.

É fácil verificar que *ECPP* sofre com o mesmo problema de precisão, pois o método

realiza operações com valores complexos e reais durante toda sua execução. Inúmeros foram os erros de precisão que precisaram ser verificados durante a implementação deste código para o projeto, definitivamente foi um dos maiores desafios de se contornar. Além disso, há uma pequena probabilidade deste não terminar por exigir muita quantidade de recurso da máquina ou de tempo.

Realizar uma implementação do *ECPP* eficiente, é equivalente a determinar de maneira correta os parâmetros que influenciam no uso de recursos da máquina para a execução otimizada da aplicação. Por exemplo, realizar operações com valores enormes para o discriminante pode demorar muito, já calcular com poucos discriminantes pode causar o sistema a desistir de provar a primalidade do número facilmente. O mesmo é análogo quando se trata do limite imposto para verificar pontos na curva elíptica que satisfaçam o teorema de *Goldwasser-Kilian*.

5.2 Contribuição

A principal contribuição desse projeto é a explicação de forma didática sobre uma grande quantidade de testes de primalidade que sumariza grande parte das abordagens mais importantes para o problema de determinação de primos, bem como as suas respectivas implementações que possuem uma performance próxima, e no caso do *AKS*, superior a de algumas encontradas em artigos científicos. Os resultados obtidos dos testes de performance são satisfatórios e nos fornece uma boa ideia do que deve ser utilizado na prática. Vale a pena citar que não encontramos nenhum trabalho científico com este mesmo objetivo.

Foram expostas os maiores problemas na utilização prática do *AKS* e do método *ECPP*. Também pode ser visto que a melhor biblioteca gratuita e atualizada no momento para implementar o *AKS* em *C++* é a biblioteca *PARI*. Mesmo sem todas as suas opções de otimização configuradas, a biblioteca teve uma performance superior a todas as outras.

5.3 Funcionalidades não implementadas

A versão simples implementada neste projeto do *AKS* teve como objetivo expor as ideias do algoritmo de forma didática, nós sentimos que este objetivo foi cumprido. Porém como podemos observar pelos testes de performance, a solução quadrática implementada na Etapa 5 para a multiplicação de polinômios não é boa. Uma funcionalidade interessante para a implementação seria o uso do *FFT* (Fast Fourier Transform) para a computação desta multiplicação, talvez pela implementação da versão do algoritmo de *Schönhage–Strassen* para polinômios.

Outra implementação interessante seria o teste de *Strong Lucas* para o *Baillie-PSW*, que tem um conjunto de pseudoprimos de Lucas menor que o Teste de Lucas mostrado no Capítulo 2.

Por último, o *ECPP* não implementa as soluções com os Polinômios de *Weber*, apesar de opcional, esta implementação poderia diminuir a probabilidade do recurso computacional esgotar. Também não implementamos soluções melhores para a função que encontra um inteiro provavelmente fatorado, poderia ser utilizado o ECM (Elliptic Curve Factorization Method) que é um método sub-exponencial para encontrar a fatoração de um inteiro.

5.4 **Trabalhos Futuros**

É interessante falar de implementações do *AKS* que utilizam paralelismo, apesar da Etapa 5 apresentar muitas dependências é possível paralelizar o algoritmo. Não há muitos trabalhos que estudam essa possibilidade. Outra ideia é combinar o *AKS* com outros testes de primalidade probabilísticos de forma paralela, para verificar em testes de performance se existe uma versão com uma combinação interessante na prática. Há trabalhos que combinam iterações do *ECPP* com *AKS*, até mesmo trabalhos que combinam testes de primalidade probabilísticos com determinísticos, mas a grande maioria não fala de paralelismo, solução que pode oferecer um ganho de desempenho abismal no problema de determinação de primos.

Outro trabalho interessante, seria realizar as implementações dos testes probabilísticos (com exceção do *ECPP*) presentes nesse trabalho utilizando inteiros e reais de precisão arbitrária, a fim de possibilitar os testes de performance com valores acima de 154 dígitos e verificar até que ponto podemos melhorar os testes utilizando pré-computações.

REFERÊNCIAS

ADLEMAN, L. M.; POMERANCE, C.; RUMELY, R. S. On distinguishing prime numbers from composite numbers. v. 117, n. 1, p. 173–206, 1983.

AGRAWAL, M.; KAYAL, N.; SAXENA, N. Primes is in p. *Annals of mathematics*, JSTOR, p. 781–793, 2004.

ARVIND, V.; SAPTHARISHI, R. *Lecture 12: The AKS Primality Test*. 20XX. Disponível em: ⟨http://www.cmi.ac.in/~ramprasad/lecturenotes/algcomp/lecture12.pdf⟩.

ATKIN, A. O. L.; MORAIN, F. Elliptic curves and primality proving. *Mathematics of computation*, v. 61, n. 203, p. 29–68, 1993.

BAILLIE, R.; WAGSTAFF, S. S. Lucas pseudoprimes. *Mathematics of Computation*, v. 35, n. 152, p. 1391–1417, 1980.

BAKER, R. C.; HARMAN, G. The brun-titchmarsh theorem on average. In: *Analytic number theory*. [S.l.]: Springer, 1996. p. 39–103.

BERNSTEIN, D.; JR, H. L.; PILA, J. Detecting perfect powers by factoring into coprimes. *Mathematics of Computation*, v. 76, n. 257, p. 385–388, 2007.

BHATTACHARJEE, R.; PANDEY, P. *Primality testing*. [S.l.], 2001.

BLAKE, I. F.; SEROUSSI, G.; SMART, N. *Elliptic curves in cryptography*. [S.l.]: Cambridge university press, 1999. v. 265.

BORNEMANN, F. Primes is in p: A breakthrough for"everyman". *NOTICES-AMERICAN MATHEMATICAL SOCIETY*, AMERICAN MATHEMATICAL SOCIETY, v. 50, n. 5, p. 545–553, 2003.

BRENT, R. P.; ZIMMERMANN, P. An o (m (n) logn) algorithm for the jacobi symbol. In: *Algorithmic Number Theory*. [S.l.]: Springer, 2010. p. 83–95.

BRUEN, A. A. et al. Applied cryptography: protocols, algorithms, and source code in c. 1996.

CANTOR, D. G.; KALTOFEN, E. On fast multiplication of polynomials over arbitrary algebras. *Acta Informatica*, Springer, v. 28, n. 7, p. 693–701, 1991.

COHEN, H. *A course in computational algebraic number theory*. [S.l.]: Springer Science & Business Media, 2013. v. 138.

COUTINHO, S. *Primalidade em Tempo Polinomial*: Uma introdução ao algoritmo aks. 1. ed. Rio de Janeiro: Sociedade Brasileira de Matemática, 2004. p. 61–71 p.

CRANDALL, R.; POMERANCE, C. Section 4.2. 1: The lucas-lehmer test. *Prime Numbers: A Computational Perspective (1st ed.), Berlin: Springer*, p. 167–170, 2001.

DIETZFELBINGER, M. *Primality testing in polynomial time: from randomized algorithms to"PRIMES is in P"*. [S.l.]: Springer Science & Business Media, 2004. v. 3000.

DIVISION, T. J. W. I. R. C. R.; RABIN, M. *Probabilistic algorithms*. [S.l.: s.n.], 1976.

DONADELLI, J. *Algortmos em teoria dos números e Criptografia de Chave pública*. 20XX. Disponível em: ⟨http://professor.ufabc.edu.br/~jair.donadelli/disciplinas-ufpr/cripto-html/#x1-3700017⟩.

FOUVRY, É. Theoreme de brun-titchmarsh; application au theoreme de fermat. *Inventiones Mathematicae*, Springer, v. 79, n. 2, p. 383–407, 1985.

GOLDFELD, M. On the number of primes p for which p+ a has a large prime factor. *Mathematika*, Cambridge Univ Press, v. 16, n. 01, p. 23–27, 1969.

GOLDWASSER, S.; KILIAN, J. Almost all primes can be quickly certified. In: ACM. *Proceedings of the eighteenth annual ACM symposium on Theory of computing*. [S.l.], 1986. p. 316–329.

JIN, T. Researching and implementing on aks algorithm. *University of Bath*, 2005.

JR, H. W. L. Factoring integers with elliptic curves. *Annals of mathematics*, JSTOR, p. 649–673, 1987.

KAYAL, N.; SAXENA, N. et al. Towards a deterministic polynomial-time primality test. Citeseer, 2002.

LEHMER, D. H. Tests for primality by the converse of fermat's theorem. *Bulletin of the American Mathematical Society*, v. 33, n. 3, p. 327–340, 1927.

MILLER, G. L. Riemann's hypothesis and tests for primality. In: ACM. *Proceedings of seventh annual ACM symposium on Theory of computing*. [S.l.], 1975. p. 234–239.

MONIER, L. Evaluation and comparison of two efficient probabilistic primality testing algorithms. *Theoretical Computer Science*, Elsevier, v. 12, n. 1, p. 97–108, 1980.

MORAIN, F. Implementing the asymptotically fast version of the elliptic curve primality proving algorithm. *Mathematics of computation*, v. 76, n. 257, p. 493–505, 2007.

POMERANCE, C. Are there counter-examples to the baillie-psw primality test. *Dopo Le Parole aangeboden aan Dr. AK Lenstra. Privately published Amsterdam*, 1984.

PRATT, V. R. Every prime has a succinct certificate. *SIAM Journal on Computing*, SIAM, v. 4, n. 3, p. 214–220, 1975.

ROTELLA, C. An efficient implementation of the aks polynomial-time primality proving algorithm. *School of Computer Science-Carnegie Mellon University. Pittsburgh-Pennsylvania-USA*, 2005.

SCHÖNHAGE, D. D. A.; STRASSEN, V. Schnelle multiplikation grosser zahlen. *Computing*, Springer, v. 7, n. 3-4, p. 281–292, 1971.

SCHOOF, R. Elliptic curves over finite fields and the computation of square roots mod. *Mathematics of computation*, v. 44, n. 170, p. 483–494, 1985.

SEDJELMACI, S. M.; LAVAULT, C. Improvements on the accelerated integer gcd algorithm. *arXiv preprint arXiv:1402.2266*, 2014.

STEIN, J. Computational problems associated with racah algebra. *Journal of Computational Physics*, Elsevier, v. 1, n. 3, p. 397–405, 1967.

UZUNKOL, O. *Atkin's ECPP Algorithm*. Dissertação (Mestrado) — M. Sc. Thesis TU-Kaiserslautern, 2004.

APÊNDICE A – Algoritmos relevantes

A.1 Algoritmos máximo divisor comum

A.1.1 Algoritmo Euclidiano

Algoritmo 8 - MDC Euclidiano

DOCUMENTAÇÃO
TÍTULO
MDC Euclidiano

PROPÓSITO
Retornar o MDC entre dois números.

MÉTODO
Recursivo

ENTRADAS
a: Primeiro número a ser computado
b: Segundo número a ser computado
SAÍDAS
Responde o MDC entre os dois números dados pela entrada
ALGORITMO MDC EUCLIDIANO
declarar z **numérico**
1. **se** (b = 0), **então**
2. | **retornar** a
3. **senão**
4. | MDC(b,a mod b);
5. **fim se**

O algoritmo é bem simples consistindo de divisões sucessivas onde é avaliado o resto da divisão. Se o resto for zero, isso significa que temos o MDC do número. Essa ideia parte do principio de que se subtrairmos o maior número do menor, o MDC entre eles não muda, dessa forma podemos pensar nos números como sendo $ka = b$, onde k é um inteiro qualquer e a e b são as entradas do algoritmo. Ao utilizarmos a propriedade modularidade, as k subtrações que seriam necessárias durante a recursão serão trocadas por uma simples operação de divisão.

Algoritmo 9 - Algoritmo de Stein

DOCUMENTAÇÃO

TÍTULO

Algoritmo de Stein

PROPÓSITO

Retornar o MDC entre dois números.

MÉTODO

Iterações binárias sob um par de números.

ENTRADAS

a: Primeiro número a ser computada b: Segundo número a ser computada

SAÍDAS

Responde o MDC entre os dois núemros dados pela entrada

ALGORITMO MDC STEIN

```
declarar z numérico
se (a = 0), então
|  retornar b
fim se
se (b = 0), então
|  retornar a
fim se
z ← conta.zeros.a.direita(a|b)
a ← a >> conta.zeros.a.direita(a)
fazer
|  b ← b >> conta.zeros.a.direita(b)
|  se (a > b), então
|  |  Troca(a,b);
|  fim se
|  b = b − a
enquanto (b ≠ 0)
retornar a << z
```

A.1.2 Algoritmo de Stein

O algoritmo de stein (STEIN, 1967) possui dois critérios de parada: o menor valor tornou-se 0 ou os dois números são iguais. Agora que sabemos os critérios de parada do algoritmo, devemos ter em mente que este itera sobre a representação binária dos números dados por entrada avaliando seus *bits* mais a direita. Em decorrencia dessa avaliação podemos ter somente três possíveis casos para a sua avaliação:

- Ambos últimos *bits* são 0 - Nesse caso devemos dividir ambos por 2.
- Um *bit* é 0 e o outro é 1 - Nesse caso devemos dividir aquele que possui bit 0 por 2.

- Ambos os bits são 1 - nesse caso devemos realizar uma subtração de ambos. Como sabemos a subtração de dois números ímpares resulta em um número par, logo podemos dividir o resultado da subtração por 2 imediatamente após.

A.2 Potência Modular

Algoritmo 10 - Potência Modular

DOCUMENTAÇÃO

TÍTULO

Potência Modular

PROPÓSITO

Retornar $a^b \bmod n$.

MÉTODO

Iterações binárias sob um número.

ENTRADAS

a: base b: expoente n: Módulo

SAÍDAS

Retornar $a^b \bmod n$.

ALGORITMO POT MOD BINÁRIO

declarar s **numérico**

1. $s \leftarrow 1$
2. **enquanto** (b $\neq$ 0), **fazer**
3. | **se** (b & 1), **então**
4. | | $s \leftarrow (sa) \bmod n$
5. | **fim se**
6. | $b \leftarrow b >> 1$
7. | $a \leftarrow a^2 \bmod n$
8. **fim enquanto**
9. **retornar** s

O algoritmo supracitado faz uso da propriedade de multiplicação nas operações de modularidade que já conhecemos, contudo para fins didáticos será enunciada todas as propriedades básicas dessa operação. São elas:

- $(a + b) \bmod n = ((a \bmod n) + (b \bmod n)) \bmod n$
- $(a - b) \bmod n = ((a \bmod n) - (b \bmod n)) \bmod n$
- $(ab) \bmod n = ((a \bmod n) \times (b \bmod n)) \bmod n$
- $(a(b + c) \bmod n = (((ab) \bmod n) + ((ac) \bmod n)) \bmod n$

Tendo em vista o *item 3* podemos facilmente perceber que uma operação $a^{32} \bmod n$ não necessita de 32 multiplicações consecutivas de a, podendo ser escrito na forma $((((a^2 \bmod n)^2 \bmod n)^2 \bmod n)^2 \bmod n)^2 \bmod n)$.

É também simples perceber que caso o expoente não seja potencia de 2 podemos separá-lo de forma conveniente em somatórios de potencias de 2. Por exemplo em a^{25}:

$25 = 11001 = 2^0 + 2^3 + 2^4$

$a^{25} = a \times a^8 \times a^{16} = (((a^2 \times a)^2)^2)^2 \times a \bmod n$

A.3 **Totiente de Euler**

Algoritmo 11 - Totiente de Euler Trivial

DOCUMENTAÇÃO

TÍTULO

Totiente de Euler 1

PROPÓSITO

Computar a quantidade de inteiros relativamente primos com a entrada, menores que a entrada.

MÉTODO

Verificar o MDC de todos os números menores que aquele número

ENTRADAS

b: número que desejamos saber o ϕ

SAÍDAS

Retornar a quantidade de todos os números primos com número dado por entrada, menores que a entrada.

OBSERVAÇÕES, RESTRIÇÕES, REQUISITOS

Nenhum

ALGORITMO TOTIENTE DE EULER

```
declarar phi,i numérico
1. phi ← 1
2. para i de 2 até n , fazer
3.  |  se (mdc(i,n)=1), então
4.  |  |  phi ← phi + 1
5.  |  fim se
6. fim para
7. retornar phi
```

A função aritmética Totiente de Euler retorna a quantidade de números menores que a entrada que são relativamente primos com ela.

No algoritmo intuitivo acima foi utilizado uma função chamada *mdc()* que faz o máximo divisor comum entre dois números para verificar se são primos entre si. Logo

abaixo é apresentado o pseudocódigo da versão implementada neste trabalho.

Algoritmo 12 - Totiente de Euler

DOCUMENTAÇÃO
TÍTULO
Totiente de Euler 2

PROPÓSITO
Computar a quantidade de inteiros relativamente primos com a entrada, menores que a entrada.

MÉTODO
Verificar todos os números que contribuem com o $\phi(b)$ através de divisões sucessivas

ENTRADAS
b: número que desejamos saber o ϕ
SAÍDAS
Retornar a quantidade de todos os números primos com número dado por entrada, menores que a entrada.
OBSERVAÇÕES, RESTRIÇÕES, REQUISITOS
Nenhum
ALGORITMO TOTIENTE DE EULER 2

```
declarar n,resultado numérico
1.  n ← b
2.  para i de 2 até i² < n , fazer
3.  |   se (n mod i = 0), então
4.  |   |   enquanto (n mod i = 0), fazer
5.  |   |   |   n ← n/i
6.  |   |   fim enquanto
7.  |   |   resultado ← resultado/i
8.  |   fim se
9.  fim para
10. se (n > 1), então
11. |   resultado ← resultado − resultado/n
12. fim se
13. retornar resultado
```

Este algoritmo não segue de forma tão intuitiva quanto o algoritmo anterior. Analisemos então primeiramente o primeiro *loop*. Nele descobrimos quais os números menores que n o dividem, caso seja encontrado algum, retiramos toda a contribuição desse valor de n, ou seja n não possui mais o número encontrado como fator primo, e retiramos todos os números múltiplos desse número do resultado. Avaliemos agora o critério de parada, caso i^2 seja menor do que n, temos que n é o ultimo fator primo de do número dado por entrada. É fácil notarmos que caso não seja menor do que n existem dois números maiores ou iguais a i, pois todos os fatores menores que i já foram retirados de n, cujo produto é menor do que i^2, o que é absurdo. E por fim devemos analisar a última cláusula condi-

cional, no qual caso exista um fator primo que é diferente de 1 a ser analisado devemos retirar suas colaborações do resultado.

A.4 **Símbolo de Jacobi**

Dadas as ideias e proposições exibidas durante a ideia do algoritmo de Solovay-Strassen que demonstram propriedades do símbolo de Jacobi podemos elaborar um algoritmo que calcula dadas as entradas a e n o valor do símbolo de Jacobi (a/n). Devemos lembrar ainda que o símbolo apresenta somente três possiveis valores:

- 0 se $a \equiv 0 \bmod n$
- 1 se $a \not\equiv 0 \bmod n$, mas existe x tal que $a \equiv x^2 \bmod n$
- -1 se $a \not\equiv 0 \bmod n$ e não existe um x que satisfaça a condição anterior.

Analisemos agora os blocos de codigo que constituem o algoritmo apresentado. Quando a = 0 ou a = 1 parte da definição os retornos.

Enquanto a for diferente de 0 podemos aplicar propriedades e calcular o valor. Assim o primeiro loop faz uso do seguinte teorema:

- (AB/N) = (2A/N) = (A/N) * (2/N) e (2/N) $(-1)^{(n^2-1)/8} = 1$ caso $N \equiv 1 \bmod 8$ ou $N \equiv 7 \bmod 8$. -1 caso $N \equiv 3 \bmod 8$ ou $N \equiv 5 \bmod 8$

Dessa forma essa propriedade é aplicada múltiplas vezes resultando em um a impar, uma vez que não exieste mais fator dois entre os fatores primos de a e a alteração em resp equivalente a essa mudança.

Após essa alteração faremos uso da seguinte propriedade para o cálculo do símbolo:

- (A/N) = (N/A) = (-1)((A-1)*(N-1))/4 = 1 caso $N \equiv 1 \bmod 4$ ou $A \equiv 1 \bmod 4$. -1 caso $N \equiv 3 \bmod 4$ e $A \equiv 3 \bmod 4$.

Como n e a são impares agora essa propriedade pode ser aplicada.

Por fim aplicaremos a seguinte propriedade, que irá alterar bastante o valor de nosso novo a, diminuindo-o:

- $A \equiv B \bmod N$ então (A/N) = (B/N).

Finalmente quando encontrarmos a=0 precisamos saber e existiu um n ¿1 que o dividiu ou se ele retornou 0 pois foi dividido por 1 para que dessa forma retornemos um valor condizente.

Algoritmo 13 - Símbolo de Jacobi

DOCUMENTAÇÃO

TÍTULO

Símbolo de Jacobi

PROPÓSITO

Retornar o Símbolo de Jacobi de n com a

MÉTODO

Calculo iterativo utilizando propriedades conhecidas do Simbolo

ENTRADAS

a : parte de cima do simbolo
n : parte de baixo do simbolo

SAÍDAS

Número de Jacobi dadas as entradas

OBSERVAÇÕES, RESTRIÇÕES, REQUISITOS

n impar maior que 3.

ALGORITMO JACOBI

declarar resp,aux **numérico**
1. **se** $(a = 0)$, **então**
2. | **retornar** 0
3. **fim se**
4. **se** $(a = 1)$, **então**
5. | **retornar** 1
6. **fim se**
7. **enquanto** $(a \neq 0)$, **fazer**
8. | **enquanto** $(a \bmod 2 = 0)$, **fazer**
9. | | $a \leftarrow a/2$
10. | | **se** $(n \bmod 8 = 3 \| \| \| n \bmod 8 = 5)$, **então**
11. | | | $resp \leftarrow -resp$
12. | | **fim se**
13. | **fim enquanto**
14. | $aux \leftarrow a$
15. | $a \leftarrow n$
16. | $n \leftarrow a$
17. | **se** $(a \bmod 4 = 3 \&\& n \bmod 4 = 3)$, **então**
18. | | $resp \leftarrow -resp$
19. | **fim se**
20. | $a \leftarrow a \bmod n$
21. **fim enquanto**
22. **se** $(n = 1)$, **então**
23. | **retornar** resp
24. **fim se**
25. **retornar** 0

ANEXO A – Implementações completas

Este trabalho disponibiliza o conjunto de códigos apresentados no projeto, a fim de mostrar a implementação completa de uma aplicação que utiliza os testes de primalidade implementados e de fornecer comentários detalhados no código. Alterações no âmbito de tipos de variáveis foram necessárias para o obtenção de respostas corretas para entradas grandes.

É também de interesse dos autores que os interessados possuam em mãos os códigos desenvolvidos pelo projeto, a fim de que estes possam utiliza-los se necessário. Para a obtenção dos códigos fontes basta acessar a página: ⟨https://github.com/ildosrel/Projeto-Final⟩.

A.1 O algoritmo AKS

A.1.1 AKS.hpp

```
/* Esta classe e suas funções são utilizadas
*  para a etapa 5 do algoritmo, a classe representa
*  um simples polinômio em uma variável arbitrária
*  e oferece as operações de atribuição e igualdade */
class polinomio{
    public:
        unsigned long long int *coef;
        unsigned int grau = 0;
        polinomio(const unsigned int &tam);
        bool operator == (const polinomio &a);
        polinomio& operator = (const polinomio &a);
        ~polinomio();
};

//Construtor da classe
polinomio::polinomio(const unsigned int &tam){
    coef = new unsigned long long int[tam+1];
    for(unsigned int i = 0; i <= tam; ++i) coef[i] = 0;
}

//Operador ==
```

```
bool polinomio::operator == (const polinomio &a){
    if(grau != a.grau) return false;
    for(unsigned int i = 0; i <= grau; ++i){ //Teste
        if(coef[i] != a.coef[i]) return false;
    }
    return true;
}

//Operador de atribuição
polinomio& polinomio::operator = (const polinomio &a){
    grau = a.grau;
    for(unsigned int i = 0; i <= grau; ++i) coef[i] = a.coef[i]; //Teste
    return *this;
}

//Destrutor
polinomio::~polinomio(){
    delete[] coef;
}

//Declaração das funções auxiliares
inline unsigned int pow_mod(const unsigned int &a, const unsigned int &b, const uns
inline unsigned int mdc(const unsigned int &a, const unsigned int &b);
inline unsigned int totiente_euler(unsigned int n);
inline void potencia_modular_polinomial(polinomio &resto, const unsigned int &n, co

/* Teste de primalide AKS: Recebe como entrada um inteiro n positivo
*  retorna sua primalidade:  True  => Primo
*                            False => Composto
*  Para detalhes veja: < https://www.cs.auckland.ac.nz/~msta039/primality_v6.pdf >
bool aks(unsigned int n){
    if(n < 2) return false;

    //ETAPA 1: Utilizando pessquisa binária
    unsigned int menor, maior, meio;
    double potencia;
    double logaritmo_n = log2(n);
    for(int b = 2; b <= logaritmo_n; ++b){
          menor = 1;
```

```
      maior = n;
      while(maior - menor >= 2){
             meio = (menor + maior)/2;
             potencia = pow((double) meio, b); //Função fornecida pelo cmath
             if(potencia < n)  menor = meio;
             else if(potencia > n)  maior = meio;
             else{
                return false;
             }
      }
}
//Fim da Etapa 1

/* Etapas 2, 3 e Etapa 4
*  Neste passo, efetuamos a computação das 3
*  etapas ao mesmo tempo por efeitos de otimização. */
unsigned int r = 0, k; //'r' e a váriavel 'k' da ordem
double logaritmo_n_2 = logaritmo_n * logaritmo_n;
unsigned int q = 2; //Váriavel que representa iteração atual de r

while(!r){
    if(q == n) return true; //Etapa 4

    /* Só precisamo utilizar uma chamada de mdc
    *  para as etapas 2 e 3, o valor deve ser
    *  verificado antes para etapa 4 */
    if(mdc(n, q) == 1){ //Verificando se n é relativamente primo com r atual (q
        for(k = 1; k <= floor(logaritmo_n_2);++k){
            if(pow_mod(n, k, q) == 1) //A ordem 'k' é menor que o logaritmo
                break;
        }
        if(k > logaritmo_n_2) //Caso contrário a ordem é algum número
            r = q;            //abaixo do logaritmo
    }
    else{ //Etapa 3
       //A iteração atual de r (q) é um fator de 'n'
        return false;
    }
    ++q;
```

```
    }
    //Fim das Etapas 2, 3 e 4

    //Etapa 5
    /*  Esta é a etapa que implementa o teorema fundamental
    *   do AKS, o método utilizado é a exponenciação binária
    *   com multiplicação quadrática. O objetivo deste código
    *   é mostrar o AKS de forma simples, qualquer operação
    *   de pré-computação para acelerar esta implementação deve
    *   ser feita aqui. Você pode tornar a implementação abaixo
    *   mais rápida pela utilização de um método rápido
    *   de multiplicação de polinômios sub-quadrática. */

    unsigned int phi; //Guarda o retorno do Totiente Euler
    phi = totiente_euler(r);

    unsigned int constante = floor(sqrt(phi)*logaritmo_n); //Limite de 'a'
    unsigned int grau_maximo = n % r; //Auxiliar para a exponenciação

    polinomio lado_dir(grau_maximo); //Construção do polinômio direito
    lado_dir.coef[grau_maximo] = 1; //P(x) = x^(n%r)
    lado_dir.grau = grau_maximo;

    polinomio lado_esq(r); //Polinomio que vai recer a computação do lado esquerdo
    for(unsigned int a = 1; a <= constante; ++a){
        lado_dir.coef[0] = a; //Polinomio direito x^(n%r) + a

        //Polinomio esq recebe o resto de ((X + a)^n / (X^r - 1)) mod n
        potencia_modular_polinomial(lado_esq, n, r, a, floor(logaritmo_n));

        //Se lado_esq <> lado_dir imprime composto
        if(!(lado_esq == lado_dir)) return false;
    }
    return true;
    //Fim Etapa 5
}

//Funções Auxiliares
```

```
//Potência modular com método binário
inline unsigned int pow_mod(const unsigned int &a, const unsigned int  &b, const un
    unsigned int base, power, resp;

   resp = 1;
   base = a;
   power = b;
   while (power){
       if (power&1)
            resp = (resp*base) % n;
        power >>= 1;
        base = base%n;
        base = (base*base) % n;
   }
   return resp;
}


//Implementação do MDC de Stein
inline unsigned int mdc(const unsigned int &a, const unsigned int &b){
    unsigned int _a = a, _b = b;
    unsigned int numero_z_dir;
    if(_a == 0) return b;
    if(_b == 0) return a;
    numero_z_dir = __builtin_ctz(_a | _b); //Conta quantos 0 tem a direita do menor
    _a >>= __builtin_ctz(_a); //Desloca e atribui
    do{
        _b >>= __builtin_ctz(_b);
        if(_a > _b){
                int troca = _a;
                _a = _b;
                _b = troca;
        }
        _b = _b - _a;
    } while(_b != 0);
    return _a << numero_z_dir;
}

//Calcula a função aritmética Totiente Euler: quantidade de inteiros menores
```

```
//que a entrada e que são relativamente primos com ela
inline unsigned int totiente_euler(unsigned int n){
    unsigned long long int i;
    unsigned int resultado;
    resultado = n;

    for (i=2; i*i<=n; ++i){
        if (n % i == 0){
            while (n % i == 0){
                n /= i;
            }
            resultado -= resultado / i;
        }
    }
    if (n > 1) {
        resultado -= resultado / n;
     }
    return resultado;
}

//Pricipal cálculo da etapa 5 utilizando exponenciação binária
//e multiplicação de polinômios quadrática.
inline void potencia_modular_polinomial(polinomio &resto, const unsigned int &n, co
    polinomio resultado(r); //Auxiliar, guarda resultado de muliplicacoes
    unsigned long long int novo_coeficiente;
    unsigned int anterior;
    bool bit;

    //Limpando o polinomio
    for(int i = 0; i <= r; ++i)
        resto.coef[i] = 0;

    resto.coef[0] = 1; //P(x) = 1;
    resto.grau = 0;

    for(int i = log_n; i >= 0; --i){ //Iterando sobre n binário
        //Multiplicação quadrática
        //resto = resto * resto mod (x^r - 1, n)
        resultado.grau = resto.grau * 2;
```

```
if(resultado.grau >= r){
    resultado.grau = r - 1;
    for(unsigned int j = 0; j <= resultado.grau; ++j){
        novo_coeficiente = 0;
        for(unsigned int k = 0, l = j; k <= j; ++k, --l){ //Primeira parte
            novo_coeficiente += resto.coef[k]*resto.coef[l];
            novo_coeficiente %= n;
        }
        for(unsigned int k = j+1, l = r-1; k < r; ++k, --l){//Segunda parte
            novo_coeficiente += resto.coef[k]*resto.coef[l];
            novo_coeficiente %= n;
        }
        resultado.coef[j] = novo_coeficiente;
    }
}
else{ //Evita muitas multiplicações por zero
    for(unsigned int j = 0; j <= resultado.grau; ++j){
        novo_coeficiente = 0;
        for(unsigned int k = 0, l = j; k <= j; ++k, --l){ //Primeira parte
            novo_coeficiente += resto.coef[k]*resto.coef[l];
            novo_coeficiente %= n;
        }
        resultado.coef[j] = novo_coeficiente;
    }
}
resto = resultado;
//Fim da multiplicação quadrática

bit = (n & ( 1 << i )) >> i; //Pega o i'th bit de n
if(bit){
    //Multiplicação por um polinômio de grau 1
    //resto = resto * (x + a) mod (x^r - 1, n)
    if(resto.grau + 1 == r) anterior = resto.coef[r - 1];
    else{
        anterior = 0;
        resto.grau += 1;
    }
    for(unsigned int j = 0; j <= resto.grau; ++j){
        novo_coeficiente = (resto.coef[j]*a) + anterior;
```

```
                anterior = resto.coef[j];
                resto.coef[j] = novo_coeficiente % n;
            }
        }
    }

    //Corrige o grau do polinômio para prox iteração
    for(unsigned int i = resto.grau; i >= 0; --i){
        if(resto.coef[i] != 0){
            resto.grau = i;
            break;
        }
    }
}
```

A.1.2 NTL.hpp

```
#include <gmp.h>
#include <NTL/ZZ.h>
#include <NTL/RR.h>
#include <NTL/ZZ_pX.h>
/* ----------------------------------------
--Header Default do NTL inclui por default:
--<cstdlib>, <cmath> e <iostream>
---------------------------------------- */

inline void totiente_euler(long &resultado, long n);

/* Observe que a função abaixo recebe a entrada
*  em forma de STRING. Ela está esperando um string
*  que contenha somente um número inteiro natural,
*  válido para a execução do AKS. Não há validação
*  de entrada no código. */
bool aks_ntl(const std::string n){ //Evita o programdor a utilizar I/O do NTL
//Etapa 1
    mpz_t mpz_n; // tipo utilizado pelo GMP
    mpz_init(mpz_n);
    mpz_set_str(mpz_n, n.c_str(), 10);
```

```
    //O NTL nao oferece uma função para
    //computar a etapa 1. O GMP oferece:
    if(mpz_perfect_power_p(mpz_n))
            return false;
//Fim da Etapa 1

//Etapas 2, 3 e 4
NTL::ZZ ZZ_n;  // tipo utilizado pelo NTL, inteiro prec arbitrária
ZZ_n = to_ZZ(NTL::conv<NTL::ZZ>(n.c_str())); //Não há validação de entrada
bool teste;
    long k; //Overflow para n > 2^(32768*raiz(2))
NTL::ZZ r(2), n_mod_r; //N muito grande pode causar overflow no r na etapa 5

double logaritmo_n = log(ZZ_n)/log(2); //Precisa ser real, se nao for causa erros d
    double logaritmo_n_2 = logaritmo_n*logaritmo_n;

    while(1){
        if(r == ZZ_n){  //Etapa 4
            return true;
        }
        if(IsOne(GCD(ZZ_n, r))){ //Verificando se n é relativamente primo com q
            teste = true;
            rem(n_mod_r, ZZ_n, r);
            for(k = 1; k <= logaritmo_n_2; ++k){
                if(IsOne(PowerMod(n_mod_r, k, r))){ // (n^k)%r==1 --> n%r passado p
                    teste = false;                  // resolve um problema da bibli
                    break;
                }
            }
            if(teste) break; //Ordem menor que logaritmo_n_2
        }
        else //Etapa 3
            return false;
        ++r;
    }
    //Fim da Etapa 2, Etapa 3 e Etapa 4

//Etapa 5
//-- Cuidado: N muito grande pode causar overflow no grau do polinomio ZZ_pX --
```

```
//-- Limitacao da biblioteca! Grau max (2^31-1) ... r vale no maximo log2(N)^5 ...
//WARNING! Possivel Overflow para n > 2^(raiz[5](2^32 - 1)) (~23 dígitos)
    long phi; //Funcao totiente de Euler
    long r_long = NTL::conv<long>(r);
    totiente_euler(phi, r_long);
    long int constante = floor(sqrt(phi)*logaritmo_n); //piso de raiz(phi(n)) log2

    NTL::ZZ_p::init(ZZ_n);//Ativa a modularidade com n para o NTL
    NTL::ZZ_pX f(r_long, 1); f -= 1; //x^r - 1, polinomio para acelerar as contas
    const NTL::ZZ_pXModulus modulo(f);  //internas do NTL com pre-computacao
    NTL::ZZ_pX lado_direito(NTL::conv<long>(n_mod_r), 1); //p(x) = x^(n mod r)
    NTL::ZZ_pX lado_esquerdo(1, 1); //p(x) = x

    for(long a; a <= constante; ++a){
        SetCoeff(lado_esquerdo, 1); lado_esquerdo += a;//p(x) = x + a
        PowerMod(lado_esquerdo, lado_esquerdo, ZZ_n, modulo); //(lhs = (x + a)^n mo
        lado_esquerdo -= a;

        if(lado_esquerdo != lado_direito) return false;
        clear(lado_esquerdo);
    }
    return true;
//Fim Etapa 5

}

//Esta função é exatamente a mesma utilizada
//no AKS simples, ela é bem rápida
inline void totiente_euler(long &resultado, long n){
    resultado = n;

    for (long i = 2; i*i<=n; ++i){
        if ((n % i) == 0){
            while ((n % i) == 0){
                n /= i;
            }
            resultado -= resultado / i;
        }
    }
```

```
    if (n > 1) {
        resultado -= resultado / n;
     }
}
```

A.1.3 FLINT.hpp

```
#include <flintxx.h>
#include <fmpzxx.h>
#include <fmpz_mod_polyxx.h>
/* ----------------------------------------
--Header Default do FLINT inclui por default:
--<stdlib.h>, <stdio.h>, <flint.h>, <gmp.h>
---------------------------------------- */


using namespace flint;


/* Dessa vez a função recebe um fmpz
*  que é o inteiro de precisão arbitrária do flint,
*   o input é simples com interface de c++ do flintxx:
*   cin >> n;                                          */
bool aks_flint(const fmpz &n){
    //Etapa 1
    mpz_t mpz_n; // tipo utilizado pelo GMP, utilizado na primeira etapa
    mpz_init(mpz_n);
    fmpz_get_mpz(mpz_n, &n);
    //O FLINT nao oferece uma função para
    //computar a etapa 1. O GMP oferece:
    if(mpz_perfect_power_p(mpz_n)){
        return false;
    }

//Fim da Etapa 1

//Etapas 2, 3 e 4
bool teste;
```

```
    long k; //Overflow para n > 2^(32768*raiz(2))
fmpz r = 2, pot_mod, gcd; //N muito grande pode causar overflow no r na etapa 5
double logaritmo_n =  fmpz_dlog(&n)/fmpz_dlog(&r); //Precia ser real, se nao for ca
    double logaritmo_n_2 = logaritmo_n*logaritmo_n;

    while(1){
        if(r == n){  //Etapa 4
            return true;
        }
        fmpz_gcd(&gcd, &n, &r);
        if(fmpz_is_one(&gcd)){ //Verificando se n é relativamente primo com q
            teste = true;
            for(k = 1; k <= logaritmo_n_2; ++k){
                fmpz_powm_ui(&pot_mod, &n, k, &r);
                if(fmpz_is_one(&pot_mod)){
                    teste = false;
                    break;
                }
            }
            if(teste) break; //Ordem menor que logaritmo_n_2
        }
        else //Etapa 3
            return false;
        ++r;
    }
    //Fim da Etapa 2, Etapa 3 e Etapa 4

    //Etapa 5
//-- Cuidado: N muito grande pode causar overflow no grau do polinomio  fmpz_mod_po
//-- Limitacao da biblioteca! Grau max (2^31-1) ... r vale no maximo log2(N)^5 ...
//WARNING! Possivel Overflow para n > 2^(raiz[5](2^32 - 1)) (~23 dígitos)
    fmpz phi, coeficiente; //Funcao totiente de Euler
    fmpz_euler_phi(&phi, &r);
    double phi_d = fmpz_get_d(&phi);
    long constante = floor(sqrt(phi_d)*logaritmo_n); //piso de raiz(phi(n)) log2 (n

    //Aqui nós montamos o polinomio do lado dir e esq com as ferramentas do flint
    slong r_long = fmpz_get_si(&r);
    fmpz_mod_poly_t lado_direito, lado_esquerdo, modulo;
```

```
    fmpz_mod_poly_init(lado_direito, &n);
    fmpz_mod_poly_init(lado_esquerdo, &n);

    //Aqui nós montamos o polinomio do modulo
    fmpz_mod_poly_init(modulo, &n);
    fmpz n_1 = n - 1;
    fmpz_mod_poly_set_coeff_ui(modulo, r_long, 1);
    fmpz_mod_poly_set_coeff_fmpz(modulo, 0, &n_1); //p(x) = x^r + (n-1)
                                                   //computar ocm este p(x) é equiv
                                                   //a computar com p'(x) = x^r - 1
    fmpz n_mod_r = n % r;
    slong n_mod_r_long = fmpz_get_si(&n_mod_r);
    fmpz_mod_poly_set_coeff_ui(lado_direito, fmpz_get_si(&n_mod_r), 1); //x^(n%r)

    for(slong a; a <= constante; ++a){
        fmpz_mod_poly_set_coeff_ui(lado_esquerdo, 1, 1);
        fmpz_mod_poly_set_coeff_ui(lado_esquerdo, 0, a);
        //A função abaixo computa o p(x) = (x + a)^n mod (x^r + n-1, n)
        fmpz_mod_poly_powmod_fmpz_binexp(lado_esquerdo, lado_esquerdo, &n, modulo);

        fmpz_mod_poly_get_coeff_fmpz(&coeficiente, lado_esquerdo, 0);
        coeficiente -= a;
fmpz_mod_poly_set_coeff_fmpz(lado_esquerdo, 0,  &coeficiente); //lado_esquerdo -= a

//if(lado_esquerdo != lado_direito)
if(!fmpz_mod_poly_equal(lado_esquerdo, lado_direito)) return false;
         fmpz_mod_poly_realloc(lado_esquerdo, 0); //Clear lado_esquerdo
    }
    return true;
//Fim Etapa 5
}
```

A.1.4 PARI.hpp

```
#include <pari/pari.h> //código da biblioteca principal

GEN potencia_modular_polinomial(const GEN &base, const GEN &mod, const long &poly_m
```

```
/* Observe que a função abaixo recebe a entrada
*  em forma de STRING. Ela está esperando um string
*  que contenha somente um número inteiro natural,
*  válido para a execução do AKS. Não há validação
*  de entrada no código. */
bool aks_pari(const std::string _n){ //Evita o programdor a utilizar I/O do PARI
    /*-----------------------------------------------------------*
    | Nao esqueca de iniciar a STACK PARI para as funcoes       |
    | PARI trabalha com uma stack propria que exige que         |
    | o programador faca garbage collection por conta propria   |
    | o ponteiro abaixo serve para retornar a stack para        |
    | onde ela estava depois da execucao de alguma funcao       |
    *-----------------------------------------------------------*/
    pari_sp av = avma, av2; //Guarda o ponteiro da stack, recupera depois
    GEN n; //variável do tipo long long int *
           //implmenta TODOS os tipos internos do PARI
    n = gp_read_str(_n.c_str()); //Não faz validação de entrada

    //Etapa 1
    if(Z_isanypower(n, NULL)){
        avma = av; //Garbage collection, restaura o topo da stack
        return false;
    }
    //Fim Etapa 1

    //Etapas 2, 3 e 4
bool teste;
    long k; //WARNING! Possivel overflow para n > 2^(32768*raiz(2))
GEN r = gen_2;
GEN n_REAL = itor(n, 3); //Variavel que guarda o valor de n como t_REAL "para o log
double logaritmo_n =  dbllog2r(n_REAL); //Precia ser real, se nao for causa erros d
    double logaritmo_n_2 = logaritmo_n*logaritmo_n;

    while(1){
        if(gequal(r, n)){  //Etapa 4
            avma = av; //Garbage collection, restaura o topo da stack
            return true;
        }
        if(gequal1(gcdii(n, r))){ //Verificando se n é relativamente primo com q
```

```
            teste = true;
            for(k = 1; k <= logaritmo_n_2; k++){
                if(gequal1(Fp_powu(n, k, r))){
                    teste = false;
                    break;
                }
            }
            if(teste) break; //Ordem menor que logaritmo_n_2
        }
        else{ //Etapa 3
            avma = av; //Restaura o ponteiro
            return false;

        }
        r = gaddgs(r, 1); //r++
    }
    //Fim da Etapa 2, Etapa 3 e Etapa 4*/

    //Etapa 5
//-- Cuidado: N muito grande pode causar overflow no grau do polinomio --
//-- Limitacao da biblioteca! Grau max (2^31-1)  ... r vale no maximo log2(N)^5 ...
    //WARNING! Possivel Overflow para n > 2^(raiz[5](2^32 - 1)) (~23 dígitos)
long r_long = gtolong(r);

    //Limite dessas variaveis dependem de (r), se r nao estiver em Overflow as cont
    long phi; //Funcao totiente de Euler
    phi = eulerphiu(r_long);
    long constante = floor(sqrt(phi)*logaritmo_n); //piso de raiz(phi(n)) log2 (n)
    GEN n_mod_r = Fp_red(n, r);
    long n_mod_r_long = gtolong(n_mod_r);
    GEN lado_direito, lado_esquerdo;

    //Aqui temos o polinomio do lado direito sendo construido
    //com funções de baixo nível da biblioteca, cuidado
    //ao alterar o código abaixo. Possível point exception
    lado_direito = cgetg(n_mod_r_long + 3, t_POL);
    lado_direito[1] = evalvarn(0);
    for(ulong x = 2; x <= n_mod_r_long+2; ++x) gel(lado_direito, x) = gen_0;
    gel(lado_direito, n_mod_r_long + 2) = gen_1; //x^(r%n)
```

```
    lado_direito = normalizepol(lado_direito);

    for(long a; a <= constante; ++a){
        av2 = avma; //Guarda o ponteiro da stack
        lado_esquerdo = mkpoln(2, gen_1, gen_0); //cria o lado esquerdo, safe
        gel(lado_esquerdo, 2) = stoi(a); //x + a

        //Ceil falha só quando n é uma potencia de 2
        lado_esquerdo = potencia_modular_polinomial(lado_esquerdo, n, r_long, ceil(
        gel(lado_esquerdo, 2) = subis(gel(lado_esquerdo, 2), a); //lado_esquerdo -=

        //O código abaixo é uma verificação em baixo nível equivalente a
        //lado_direto != lado_esquerdo ? composto : continua;
        if(degree(lado_esquerdo) != degree(lado_direito)){
            avma = av; //Garbage collection
            return false;
        }
        for(ulong x = 0; x <= degree(lado_esquerdo); ++x){
            if(!gequal(gel(lado_esquerdo, x+2), gel(lado_direito, x+2))){
                avma = av; //Garbage collection
                return false;
            }
        }

        avma = av2; //Garbage collection, restaura o topo da stack
    }
    avma = av; //Garbage collection, restaura o topo da stack
    return true;
    //Fim Etapa 5
}

/* Esta função é uma tradução da função de mesmo nome do AKS simples.
*  Nós somente traduzimos o código em C para código em PARI.
*  O PARI é extremamente eficiente aqui, ele utiliza uma
*  uma multiplicação de polinômios sub-quadrática. */
GEN potencia_modular_polinomial(const GEN &base, const GEN &mod, const long &poly_m
    GEN resto = pol_1(0);
    GEN pow2i, ndivpow, truncate, bit;
```

```
    for(ulong i = logaritmo_n; i > 0; i--){
        resto = FpX_sqr(resto, mod); //resto = resto*resto mod n
        resto =  ZX_mod_Xnm1(resto, poly_mod); //resto em Zn[x]/x^r - 1

        //Pegando o i-ésimo bit de n
        pow2i = powuu(2, i-1);
        ndivpow = rdivii(mod, pow2i, 3);
        truncate = floorr(ndivpow);
        bit = Fp_red(truncate, gen_2);

        if(gequal1(bit)){
            resto = FpX_mul(resto, base, mod); //resto = resto*(x + a)
            resto = ZX_mod_Xnm1(resto, poly_mod); //resto em Zn[x]/x^r - 1
        }

    }
    resto = RgX_to_FpX(resto, mod); //Reduz o polinomio para Zn[x]
    return resto;
}
```

A.1.5 AKS.cpp

```
#include <iostream> //cin; cout
#include <ratio> //duration
#include <chrono> //contagem de tempo
#include <getopt.h> //opções
#include <string> //Algumas funções para lidar com string
#include <cmath> //Utilizado pelas funções de logaritmo, exponenciação e raiz

#include "AKS.hpp" //AKS simples
#include "NTL.hpp" //AKS NTL
#include "FLINT.hpp" //AKS FLINT
#include "PARI.hpp"  //AKS PARI

std::chrono::steady_clock::time_point inicio, fim;
std::chrono::duration<double> tempo_decorrido;
unsigned int n_int; //Entrada do AKS simples
std::string n_string; //Entrada do NTL e do PARI
```

```
fmpz n_fmpz; //Entrada do FLINT
bool teste, simples, ntl, pari, flin, tempo, quiet;
int prox_opcao;
const char* const opcoes = "spnftq";
const char* nome_programa;

void inline configura(int, char**);
void inline uso();

const struct option argumentos[] = {
{"simples", 0, NULL, 's'},
{"pari", 0, NULL, 'p'},
{"ntl", 0, NULL, 'n'},
{"flint", 0, NULL, 'f'},
{"tempo", 0, NULL, 't'},
{"quiet", 0, NULL, 'q'},
{NULL, 0,NULL, 0}
};

int main(int argc, char* argv[]){
    nome_programa = argv[0];
    if(argc == 1) uso();
    else configura(argc, argv);

    /*-------------
      AKS SIMPLES
    --------------*/
    if(simples && !pari && !ntl && !flin){
        if(!quiet){
            std::cout << "AKS Simples -- Contagem de Tempo: ";
            tempo ? std::cout << "ativada.\n" : std::cout << "desativada.\n";
        }

        while(std::cin >> n_int){
            tempo_decorrido = std::chrono::steady_clock::duration::zero();
            inicio = std::chrono::steady_clock::now();
            teste = aks(n_int);
            fim = std::chrono::steady_clock::now();
            if(teste) std::cout << n_int << " e primo.";
```

```
        else std::cout << n_int << " e composto.";
        if(tempo){
            tempo_decorrido = std::chrono::duration_cast<std::chrono::duration<
            std::cout << " Tempo de execucao: " << tempo_decorrido.count();
        }
        std::cout << std::endl;
    }
}

/*-------------
    AKS PARI
--------------*/
else if(!simples && pari && !ntl && !flin){
    if(!quiet){
        std::cout << "AKS PARI -- Contagem de Tempo: ";
        tempo ? std::cout << "ativada.\n" : std::cout << "desativada.\n";
    }
    /*------------------------------------------------
 Inicia o stack para pre computacoes do PARI.
     Primeiro argumento eh o numero de bytes que o pari
     tem para trabalhar -- 5mb utilizado,
     o segundo eh o primo maximo que ele pode
     pre-computar -- 2^16 utilizado
------------------------------------------------*/
pari_init(100000000, 65536);

while(std::cin >> n_string){
        tempo_decorrido = std::chrono::steady_clock::duration::zero();
        inicio = std::chrono::steady_clock::now();
        teste = aks_pari(n_string);
        fim = std::chrono::steady_clock::now();
        if(teste) std::cout << n_string << " e primo.";
        else std::cout << n_string << " e composto.";
        if(tempo){
            tempo_decorrido = std::chrono::duration_cast<std::chrono::duration<
            std::cout << " Tempo de execucao: " << tempo_decorrido.count();
        }
        std::cout << std::endl;
}
```

```
    //Retorna a memoria utilizada pelo PARI STACK para o sistema operacional
    pari_close();
}

/*-------------
    AKS NTL
--------------*/
else if(!simples && !pari && ntl && !flin){
    if(!quiet){
        std::cout << "AKS NTL -- Contagem de Tempo: ";
        tempo ? std::cout << "ativada.\n" : std::cout << "desativada.\n";
    }

    while(std::cin >> n_string){
        tempo_decorrido = std::chrono::steady_clock::duration::zero();
        inicio = std::chrono::steady_clock::now();
        teste = aks_ntl(n_string);
        fim = std::chrono::steady_clock::now();
        if(teste) std::cout << n_string << " e primo.";
        else std::cout << n_string << " e composto.";
        if(tempo){
            tempo_decorrido = std::chrono::duration_cast<std::chrono::duration<
            std::cout << " Tempo de execucao: " << tempo_decorrido.count();
        }
        std::cout << std::endl;
    }
}

/*-------------
    AKS FLINT
--------------*/
else if(!simples && !pari && !ntl && flin){
    if(!quiet){
        std::cout << "AKS FLINT -- Contagem de Tempo: ";
        tempo ? std::cout << "ativada.\n" : std::cout << "desativada.\n";
    }

    while(std::cin >> n_fmpz){ //interface do flint para c++
        tempo_decorrido = std::chrono::steady_clock::duration::zero();
```

```
            inicio = std::chrono::steady_clock::now();
            teste = aks_flint(n_fmpz);
            fim = std::chrono::steady_clock::now();
            if(teste) std::cout << n_fmpz << " e primo.";
            else std::cout << n_fmpz << " e composto.";
            if(tempo){
                tempo_decorrido = std::chrono::duration_cast<std::chrono::duration<
                std::cout << " Tempo de execucao: " << tempo_decorrido.count();
            }
            std::cout << std::endl;
        }
    flint_cleanup(); //Libera a memória utilizada pelo FLINT
    }
    else uso();

    return 0;
}

/*-----------------------------------------
O código abaixo é somente para configuração
-----------------------------------------*/

void inline uso(){
    std::cout << "Utilize assim: " << nome_programa << " opcao [-t] [-q] [ < arquiv
    std::cout <<
" -s --simples Utiliza o AKS Simples\n"
" -p --pari   Utiliza o AKS com PARI.\n"
" -n --ntl Utiliza o AKS com NTL.\n"
" -f --flint  Utiliza o AKS com FLINT.\n"
" -t --tempo Ativa a contagem de tempo.\n"
" -q --quiet So imprime o resultado, minimalista.\n" ;
exit(0);
}

void inline configura(int argc, char** argv){
    quiet = tempo = ntl = simples = flin = pari = false;
    do {
prox_opcao = getopt_long(argc, argv, opcoes, argumentos, NULL);
switch (prox_opcao)
```

```
{
case 's':
simples = true;
break;
case 'p':
pari = true;
break;

case 'n':
ntl = true;
break;
            case 'f':
flin = true;
break;
case 't':
tempo = true;
break;
case 'q':
quiet = true;
break;
case -1:
break;

default:
uso();
}
}
while (prox_opcao != -1);
}
```

A.2 **Os testes de primalidade probabilísticos**

A.2.1 POW_MOD.hpp

```
inline unsigned long long int pow_mod(const unsigned long long int &a, const unsign
    __int128 base, power, resp, aux; //utilize long long para sistemas que nao supo

   resp = 1;
```

```
    base = a;
    power = b;
    while (power){
        if (power&1){

            aux = (resp%n)*(base%n);
            resp = aux % n;
        }
         power >>= 1;
         base = base%n;
         aux = base*base;
         base = aux % n;
    }

    return resp;
}
```

A.2.2 SS.hpp

```
//Não aceita entradas negativas. Veja Jacobi2 no Lucas.hpp
//para um exemplo que aceita
int Jacobi(unsigned long long int a,unsigned long long int n){
    if(!a) return 0; // (0/n) = 0
    if(a==1) return 1; // (1/n) = 1
    short resp;
    int v0;
    resp=1;
    while(a){
        if(!(a&1)){
            v0 = __builtin_ctz (a);
            a>>=v0;
            if(n%8==3||n%8==5) if(v0&1) resp=-resp;
        }
        long long int aux = a;
        a = n;
        n = aux;
        if(a%4==3 && n%4==3) resp=-resp;
        a=a%n;
```

```
    }
    if(n==1) return resp;
    return 0;
}

bool solovay_strassen(const unsigned long long int & p, const int & k){
    if (!(p&1)) return false;
    unsigned long long int r, jac;
    int i;
    std::random_device rd; //Engine de randomização c++11
    std::uniform_int_distribution<unsigned long long int> uni; //distribuição unifo
    std::mt19937 rgn(rd());
    for(i=0;i<k;i++){
        r = uni(rgn); //Escolhe um random
        r = 1 + (r % (p-1)); //Coloca o random no range [1, p-1]
        jac=(p+Jacobi(r,p))%p;
        if(!jac || pow_mod(r,(p-1)/2,p)!=jac){
            return false;
        }
    }
    return true;
}
```

A.2.3 MR.hpp

```
bool miller_rabin(const unsigned long long int & p, const int &k) {
    if (p&1){ //
        unsigned long long int p1 = p-1;
        unsigned long long int v0 = __builtin_ctz (p1), p2; // número de 0 a direit
        unsigned long long int r,s,mod;
        std::random_device rd; //Engine de randomização c++11
        std::uniform_int_distribution<unsigned long long int> uni; //distribuição u
        std::mt19937 rgn(rd());

        // removando potencias de 2 de p1
        p2 = p1 >> v0; // removi todos os 0 a direita
        for(int i=0;i < k; ++i){
            r = uni(rgn); //Escolhe um random
```

```
            r = 1 + (r % (p-1)); //Coloca o random no range [1, p-1]
            s = p2;
            mod = pow_mod (r,s,p); // mod = (r^s) mod p
            while (s!=(p1) && mod!=1 && mod!=(p1)){
                // voltou para a condição original ou
                //encontrou uma das duas condições de paradas do item
                mod = pow_mod (mod,2,p);
                s *= 2;
            }
            if (mod!=p1 && !(s&1)){
                return false;
            }
        }
    }
    else return false;
    return true;
}
```

A.2.4 BPSW.hpp

```
bool baillie_psw(const long long int & p){
    int i;
    /*A precomputação torna o teste muito rapido
      Nós desativamos ela por padrão pois
      para não influenciar nos testes de performance */
    //for (i=0;i<1000;++i)
        //if (!(p%primos[i]))
            //return !(p-primos[i]);
    if (!miller_rabin(p, 1)) return false;
    if (!lucas(p)) return false;
    return true;
}
```

A.2.5 Lucas.hpp

```
//Esta função aceita entradas negativas
inline int Jacobi2(long long int a,long long int n){
```

```
    if(!a) return 0;
    if(a==1) return 1;
    short resp;
    int v0;
    resp=1;
     if(a<0){
        a=-a;
        if(n%4==3) resp=-resp;
    }
    while(a){
        if(!(a&1)){
            v0 = __builtin_ctz (a);
            a>>=v0;
            if(n%8==3||n%8==5) if(v0&1) resp=-resp;
        }
        long long int aux = a;
        a = n;
        n = aux;
        if(a%4==3 && n%4==3) resp=-resp;
        a=a%n;
    }
    if(n==1) return resp;
    return 0;
}

bool lucas (long long int p){
    __int128 d,ud,P,Q,U,V,U2,V2,aux,uaux,n;
    short sinal,jac;
    ud=5; sinal = 1;
    while (1){
        d = ud * sinal;
        jac = Jacobi2(d,p);
        if (!jac && ud!=p){
            return false;
        }
        if (jac == -1){
            break;}
        sinal = -sinal;
        ud+=2;
```

```
    }
    P=1;
    Q= (1-d)/4;
    U=0; V=2;
    U2=1; V2=P;
    n = (p+1)/2;
    while (n){
        U2 = (p+(U2*V2))%p;
        V2 = (p+(V2 * V2 - Q - Q))%p;
        if (n&1){
            uaux = U;
            U = U2 * V + U * V2;
            if ((U&1))
            U+=p;
            U/=2;
            V = V2*V + U2 * uaux * d;
            if ((V&1))
            V+=p;
            V/=2;
            if (U<0) U-=((U/p)+1)*p;
            if (V<0) V-=((V/p)+1)*p;
            U%=p;
            V%=p;
        }
        Q = (p + (Q * Q)) % p;
        n/=2;
    }
    if (U) return false;
    return true;
}
```

A.2.6 Prob.cpp

```
/*-------------------------------------------------------------------
    Universidade do Estado do Rio de Janeiro
    Autores: Anderson Zudio de Moraes
             Victor Cracel Messner
*   Testes de Primalidade Monte Carlo - Miller-Rabin, Solovay-Strassen
```

```
*   e Baillie-PSW
*   OBS: Para as entradas possíveis desse código, o Baillie-PSW é
*   determinístico.

*   O código apresentado faz parte do projeto de conclusão de curso
*   dos autores pelo título de Bacharel, apresentado ao
*   Intituto de Matemática e Estatística,
*   da Universidade do Estado do Rio de Janeiro.
*   Você pode modificar e distribuir este código livremente
*   através dos termos do GNU General Public como publicado
*   pelo Free Software Foundation através da versão 3 da licença,
*   ou qualquer versão subsequente por opção sua.

*   Este é um exemplo de função principal para executar
*   os testes de primalidade probabilsticos apresentados no projeto.
*   Não esqueça que este código utiliza a biblioteca Chrono para
*   a contagem de tempo. Compile com a opção -std=c++11,
*   recomendamos o uso de -Ofast (opcional).

------------------------------------------------------------------------*/

#include <iostream> //cin; cout
#include <ratio> //duration
#include <chrono> //contagem de tempo
#include <getopt.h> //opções
#include <string> //Algumas funções para lidar com string
#include <cmath> //Utilizado pelas funções de logaritmo, exponenciação e raiz
#include <random> //Random engine do c++

#include "POW_MOD.hpp" //pow_mod(), utilizado por todos
#include "SS.hpp" //Solovay-Strassen
#include "MR.hpp" //Miller-Rabin
#include "Lucas.hpp" //Lucas do Baillie-PSW
#include "BPSW.hpp" //Baillie-PSW

std::chrono::steady_clock::time_point inicio, fim;
std::chrono::duration<double> tempo_decorrido;
unsigned long long int n; //Entrada
bool teste, SS, MR, BPSW, tempo, quiet;
```

```
int prox_opcao, k;
const char* const opcoes = "smbtq";
const char* nome_programa;

void inline configura(int, char**);
void inline uso();

const struct option argumentos[] = {
{"solovay-strassen", 0, NULL, 's'},
{"miller-rabin", 0, NULL, 'm'},
{"baillie-psw", 0, NULL, 'b'},
{"tempo", 0, NULL, 't'},
{"quiet", 0, NULL, 'q'},
{NULL, 0,NULL, 0}
};

int main(int argc, char* argv[]){
    nome_programa = argv[0];
    if(argc == 1) uso();
    else configura(argc, argv);

    /*-------------------
      Solovay-Strassen
    -------------------*/
    if(SS && !MR && !BPSW){
        if(!quiet){
            std::cout << "Solovay-Strassen -- Contagem de Tempo: ";
            tempo ? std::cout << "ativada.\n" : std::cout << "desativada.\n";
        }

        while(std::cin >> n >> k){
            tempo_decorrido = std::chrono::steady_clock::duration::zero();
            inicio = std::chrono::steady_clock::now();
            teste = solovay_strassen(n, k);
            fim = std::chrono::steady_clock::now();
            if(teste) std::cout << n << " e provavelmente primo.";
            else std::cout << n << " e composto.";
            if(tempo){
                tempo_decorrido = std::chrono::duration_cast<std::chrono::duration<
```

```
            std::cout << " Tempo de execucao: " << tempo_decorrido.count();
        }
        std::cout << std::endl;
    }
}

/*----------------
    Miller-Rabin
----------------*/
else if(!SS && MR && !BPSW){
    if(!quiet){
        std::cout << "Miller-Rabin -- Contagem de Tempo: ";
        tempo ? std::cout << "ativada.\n" : std::cout << "desativada.\n";
    }

while(std::cin >> n >> k){
        tempo_decorrido = std::chrono::steady_clock::duration::zero();
        inicio = std::chrono::steady_clock::now();
        teste = miller_rabin(n, k);
        fim = std::chrono::steady_clock::now();
        if(teste) std::cout << n << " e provavelmente primo.";
        else std::cout << n << " e composto.";
        if(tempo){
            tempo_decorrido = std::chrono::duration_cast<std::chrono::duration<
            std::cout << " Tempo de execucao: " << tempo_decorrido.count();
        }
        std::cout << std::endl;
}
}

/*--------------
   Baillie-PSW
---------------*/
else if(!SS && !MR && BPSW){
    if(!quiet){
        std::cout << "Baillie-PSW -- Contagem de Tempo: ";
        tempo ? std::cout << "ativada.\n" : std::cout << "desativada.\n";
    }
```

```
        while(std::cin >> n){
            tempo_decorrido = std::chrono::steady_clock::duration::zero();
            inicio = std::chrono::steady_clock::now();
            teste = baillie_psw(n);
            fim = std::chrono::steady_clock::now();
            if(teste) std::cout << n << " e primo."; //deterministico até 2^64
            else std::cout << n << " e composto.";
            if(tempo){
                tempo_decorrido = std::chrono::duration_cast<std::chrono::duration<
                std::cout << " Tempo de execucao: " << tempo_decorrido.count();
            }
            std::cout << std::endl;
        }
    }
    else uso();

    return 0;
}

/*-----------------------------------------
O código abaixo é somente para configuração
-----------------------------------------*/

void inline uso(){
    std::cout << "Utilize assim: " << nome_programa << " opcao [-t] [-q] [ < arquiv
    std::cout <<
" -s --solovay-strassen Utiliza o teste Solovay-Strassen\n"
" -m --miller-rabin  Utiliza o teste Miller-Rabin.\n"
" -b --baillie-psw     Utiliza o teste Baillie-PSW.\n"
" -t --tempo Ativa a contagem de tempo.\n"
" -q --quiet So imprime o resultado, minimalista.\n" ;
exit(0);
}

void inline configura(int argc, char** argv){
    quiet = tempo = MR = SS = BPSW = false;
    do {
prox_opcao = getopt_long(argc, argv, opcoes, argumentos, NULL);
switch (prox_opcao)
```

```
{
case 'm':
MR = true;
break;
case 's':
SS = true;
break;
case 'b':
BPSW = true;
break;
case 't':
tempo = true;
break;
case 'q':
quiet = true;
break;
case -1:
break;

default:
uso();
}
}
while (prox_opcao != -1);
}
```

A.3 O ECPP

A.3.1 small_primes.h

```
#define  SMALL_PRIME 65537   //Ultimo primo da lista
#define  LAST_PRIME 6542 //Indice do ultimo primo
#define D_MAX 380000    //Discriminante maximo
#define PRECISION 1000  //Numero de casas decimais das computações
#define PTRIES 100 //Número de tentativas de pontos na curva para provar o primo

unsigned int primos[] = { ... };
```

A.3.2 ECPP.cpp

```
#include <iostream> //I/O
#include <pari/pari.h> //biblioteca PARI
#include <cmath> //Operações de matematica
#include <random> //Engine de random do c++11
#include <chrono>   //Contagem de tempo
#include <getopt.h> //opções
#include <random> //Random engine do c++

/* Contém informações básicas de parametros de calibramento
*  Você deve calibrar o DMAX, a pre-computação e o número de
*  pontos a serem testados. Também a precisão é um argumento a ser calibrado
*  recomendamos no mínimo 1000 casas decimais. */
#include "small_primes.h"

bool certificado;

//Retorna um fator provavelmente primo de m maior que limite ou NULO
GEN provavelmente_fatorado(GEN m, GEN limite);

/* Algoritmo que computa o polinomio de classe Hilbert,
   Veja See "A Course in Computational Algebraic Number Theory"
   de Henri Cohen pagina 415. Dado um discriminante negativo D
   o algoritmo computa o polinômio de grau h(D) em Zn[X]
   tal que j((D+sqrt(D))/2) é uma raiz. */
 GEN somatorio(GEN q);
 GEN hilbert(long D);


/* Verifica se tem como encontrar a curva elíptica  E(a, b)
   pela definição de Lenstra, sabendo que #E(a, b) = m. */
bool curva_ideal(long D, GEN n, GEN *raiz, long *qtd_raiz, GEN *a, GEN *b);

/* Computa P' = P*k na curva elítpica E(a, b) pela definição de Lenstra */
int computa_P(std::pair<GEN, GEN> P, GEN k, GEN n_i, GEN a, std::pair<GEN, GEN> *Pr

/* Não há validação de entrada em n.
   A função aguarda um intiero positivo de
```

```
   precisão arbitrária.
   Se o sistema retornar FALSE, há uma pequena chance do sistema
   ter desistido de provar a primalidade do número. */
bool ecpp(const std::string n){
    pari_sp av = avma, av2, av3, av4;   //Ponteiros para garbage collection
    GEN n_i = gp_read_str(n.c_str()), q, m, a, b, raiz, invariante, c, g, x, y, n_i
    std::pair<GEN, GEN> P, P1, P2;
    GEN U, V; //Recebe as solucoes de cornacchia
    long D, qtd_raiz, k, i = 0;
    bool achei;

    while(gcmpgs(n_i, SMALL_PRIME) == 1){
        av2 = avma;

        /* Se a execução for um número abaixo de 2^64 - 1
           então o Baillie-psw é determinístico e você pode parar de computar.
           Retorne o resultado de Baillie-PSW.
           Este código não para aqui pois o objetivo é demonstrar o sistema ECPP*/
        if(!BPSW_psp(n_i)){ //Provavelmente primo
                av = avma; //Garbage collection
                return false;
        }

        for(D = -3; D >= -D_MAX; D--){
            achei = false;

            //Testando discriminante fundamental
            if(D%4 != 0 && D%4 != -3 && D%4 != 1) continue;

            av3 = avma; //Garbage collection


            GEN D_MOD_Ni = addsi(D, n_i);
            //Computa o simbolo de Jacobi, a entrada  provavelmente primo
            if(kronecker(D_MOD_Ni, n_i) != 1){
                    avma = av3;
                    continue;
            }
```

```
//Retorna a solucao da eq diogfantina para as variaveis A e B
GEN GEN_d = stoi(D);
if(!cornacchia2(negi(GEN_d), n_i, &U, &V)){
        avma = av3;
        continue;
}

//Determinando m com fator q bom
teste = itor(n_i, PRECISION);
teste = sqrtnr(teste, 4); teste = addrs(teste, 1);
teste = powru(teste, 2);

//m = n_i + 1 + U
n_i1 = addii(n_i, gen_1); //Auxiliar que guarda N_i + 1
m = gcopy(n_i1);
m = addii(m, U);
q = provavelmente_fatorado(m, teste);

if(q == NULL){
    //m = n_i + 1 - U
    affii(n_i1, m); m = subii(m, U);
    q = provavelmente_fatorado(m, teste);
}

//Especiais
if(q == NULL){
    if(D == -4){
        //m = n_i + 1 + 2*V
        affii(n_i1, m); m = addii(m, mulis(V, 2));
        q = provavelmente_fatorado(m, teste);

        if(q == NULL){
            //m = n_i + 1 - 2*V
            affii(n_i1, m); m = subii(m, mulis(V, 2));
            q = provavelmente_fatorado(m, teste);
       }

    }
    else if(D == -3){
```

```
        //m = n_i + 1 + (U + 3*V)/2
        affii(n_i1, m); m = addii(m, divii(addii(U, mulis(V, 3)),gen_2)
        q = provavelmente_fatorado(m, teste);

       if(q == NULL){
            //m = n_i + 1 - (U + 3*V)/2
            affii(n_i1, m); m = subii(m, divii(addii(U, mulis(V, 3)),ge
            q = provavelmente_fatorado(m, teste);
        }

        if(q == NULL){
            //m = n_i + 1 + (U - 3*V)/2
            affii(n_i1, m); m = addii(m, divii(subii(U, mulis(V, 3)),ge
            q = provavelmente_fatorado(m, teste);
        }

        if(q == NULL){
            //m = n_i + 1 + (U - 3*V)/2
            affii(n_i1, m); m = subii(m, divii(subii(U, mulis(V, 3)),ge
            q = provavelmente_fatorado(m, teste);
        }
    }
}

//Nao consegui fatorar m
if(q == NULL){
        avma = av3; //Garbage Collection
        continue;
}

//Eu tenho o #E(a, b), vou tentar construir a curva
qtd_raiz = 0;
for(int tipo = 0; tipo < 2; tipo++){
    if(!tipo){ //Casos especiais
        if (D == -3) {
            a = Fp_red(gen_0, n_i);
            b = Fp_red(gen_m1, n_i);
            qtd_raiz = 1;
        }
```

```
    else if (D == -4) {
        a = Fp_red(gen_m1, n_i);
        b = Fp_red(gen_0, n_i);
        qtd_raiz = 1;
    }
}
else if(!curva_ideal(D, n_i, &raiz, &qtd_raiz, &a, &b)) continue;

for(int tentativa = 0; tentativa < qtd_raiz; tentativa++){
    if(tipo){
        invariante = Fp_red(gel(raiz, tentativa+1), n_i);
        c = Fp_red( mulii(invariante, ginvmod(subis(invariante, 172
        a = Fp_red(mulis(c, -3), n_i);
        b = Fp_red(mulis(c, 2), n_i);
    }

    //Consegui achar uma curva, vou computar g para a multiplicação
    while(true){
        do{
            g = randomi(n_i);
        }
        while(gequal0(g));
        if(kronecker(g, n_i) == -1){

            if(D == -3) //Caso especial
                if(gequal1(Fp_pow(g, diviuexact(subiu(n_i, 1), 3),
                    continue;
                break;
        }
    }

    //Vou tentar encontrar um ponto para satisfazer o teorema
    for(int t_ponto = 0; t_ponto < PTRIES; t_ponto++){

        //Encontrando um ponto aleatório
        y = gen_0;
        while(gequal0(y)){
            do {
                do
```

```
                                    x = randomi(n_i);
                                while (gequal0(x));
                                y = powiu(x, 3);
                                y = addii(y, mulii(x, a));
                                y = addii(y, b);
                                y = Fp_red(y, n_i);
} while (kronecker(y, n_i) == -1);
y = Fp_sqrt(y, n_i);
if(y == NULL) y = gen_0;
}

                        P.first = gcopy(x); P.second = gcopy(y);
k = 0;
                        while(true){
                            //computando P1 e P2
                            int respP1 = computa_P(P, m, n_i, a, &P1);
                            if(respP1 == 1){
                                //Nao foi possivel realizar a computaçao
                                //n_i é composto, desistindo
                                //de provar a primalidade de n_0
                                avma = av;
                                return false;
                            }

                            int respP2 = computa_P(P, divii(m, q), n_i, a, &P2);
                            if(respP2 == 1){
                                //Nao foi possivel realizar a computaçao
                                //n_i é composto, desistindo
                                //de provar a primalidade de n_0
                                avma = av;
                                return false;
                            }

                            if(respP1 == -1 && respP2 == 0){
                                //Achei um ponto que atende o teorema
                                achei = true;
                                break;
                            }
                            ++k;
```

```
                    //Vou tentar denovo com outra curva
                    if(D == -3){
                        if(k >= 6) break;
                        b = gmul(b, g);
                    }
                    else if(D == -4){
                        if(k >= 4) break;
                        a = gmul(a, g);
                    }
                    else{
                        if(k >= 2) break;
                        a = gmul(gmul(g, g), a);
                        b = gmul(gpowgs(g, 3), b);
                    }
                    a = Fp_red(a, n_i);
                    b = Fp_red(b, n_i);
                }
                if(achei) break;
            }
            if(achei) break;
        }
        if(achei) break;
    }
    if(achei) break;
    else avma = av3; //Garbage collection
}

if(-D == D_MAX+1){
        //Tentei muito, melhor desistir
        //Não quer dizer que n_0 é composto,
        //eu estou aqui porque cheguei no meu limite
        //e não consigo provar a primalidade de n_i
        avma = av;
        return false;
}

//Imprimidno o certificado
if(certificado){
```

```
            std::cout << "n_" << i;
            pari_printf(" = %Ps\n", n_i);
            std::cout << "E(a, b), a = ";
            pari_printf("%Ps, b = %Ps\n", a, b);
            std::cout << "m = ";
            pari_printf("%Ps, q = %Ps\n", m, q);
            std::cout << "P = (";
            pari_printf("%Ps, %Ps)\n", P.first, P.second);
            std::cout << "P1 = (";
            pari_printf("%Ps, %Ps)\n", P1.first, P1.second);
            std::cout << "P2 = (";
            pari_printf("%Ps, %Ps)\n", P2.first, P2.second);
        }

        i++;
        n_i = gerepileupto(av2, gcopy(q));  //Garbage Collection
    }

    //n_I é pequeno, vou fazer uma pesquisa binária nos primos que conheço
    long n_long = itos(n_i);

    long low = 0, high = LAST_PRIME, mid;
    while(low <= high){
        mid = (low + high)/2;
        if(primos[mid] == n_long){
                av = avma;
                if(certificado)
                    std::cout << n_long << " e um primo determinado pelo AKS." << s
                return true; //Consegui provar que é primo
        }
        else if(primos[mid] < n_long) low = mid + 1;
        else high = mid - 1;
    }

    //Restaurando
    av = avma;
    return false; //Pela natureza do baillie-psw, eu não devo executar isto
}
```

```
GEN provavelmente_fatorado(GEN m, GEN limite){
    pari_sp ltop = avma;
    GEN q = gcopy(m);
    GEN gen_3 = addii(gen_2, gen_1), gen_5 = addii(gen_3, gen_2);

    while(gequal0(modii(q, gen_2))) q = diviiexact(q, gen_2);
    while(gequal0(modii(q, gen_3))) q = diviiexact(q, gen_3);
    while(gequal0(modii(q, gen_5))) q = diviiexact(q, gen_5);


    if(gcmp(q, limite) != 1 || gequal(q, m)){
            avma = ltop;
            return NULL;
    }


    if(BPSW_psp(q)) return gerepileupto(ltop, q);

    avma = ltop;
    return NULL;
}

GEN somatorio(GEN q){
    pari_sp av = avma;
    GEN erro = gen_1;
long e1, e2, n = 1, sinal = -1;
GEN termo1, termo2, soma = gen_0;

    for (long i = 0; i < 1000; i++)
erro = gdivgs(erro, 2);

do {
e1 = n * (3 * n - 1) / 2;
e2 = n * (3 * n + 1) / 2;
termo1 = gpowgs(q, e1);
termo2 = gpowgs(q, e2);
termo1 = gadd(termo1, termo2);
termo1 = gmulgs(termo1, sinal);
soma = gadd(soma, termo1);
```

```
sinal = -sinal;
n++;
} while (gcmp(gabs(termo1, PRECISION), erro) == 1);

return gerepileupto(av, gadd(gen_1, soma));
}


GEN hilbert(long D){
    pari_sp av = avma;
    GEN gen_D, c_poly, q_poly, ca, cb, tau, J, c, pi2, q, q2, deltaq, deltaq2, f, f
    long b, B, t, a;
    gen_D = gsqrt(stor(D, PRECISION), PRECISION);

    c_poly = cgetg(3, t_POL);
    c_poly[1] = evalvarn(0);
    gel(c_poly, 2) = gen_1; //as contas acabam gerando t_COMPLEX
    c_poly = normalizepol(c_poly);

    b = D % 2;
    if(b < 0) b += 2;
B = (long) sqrt(labs(D) / 3.0);

    do{
        t = (b * b - D) / 4;
        a = (b > 1) ? b : 1;

        do{
            if (t % a == 0){
                ca = mulss(2, a); ca = itor(ca, PRECISION);
                cb = stor(-b, PRECISION);
                tau = gdiv(gadd(cb, gen_D), ca);
                pi2 = constpi(PRECISION);
                pi2 = mulrs(pi2, 2);
                c = cgetc(PRECISION);
                gel(c, 1) = itor(gen_0, PRECISION);
                gel(c, 2) = pi2;

                q = gexp(gmul(c, tau), PRECISION);
```

```
                q2 = gmul(q, q);

                deltaq = gmul(q, gpowgs(somatorio(q), 24));
                deltaq2 = gmul(q2, gpowgs(somatorio(q2), 24));

                f = gdiv(deltaq2, deltaq);
                f1 = gmulsg(256, f);
                f1 = gadd(gen_1, f1);
                f1 = gpowgs(f1, 3);
                J = gdiv(f1, f);

                if (a == b || a * a == t || b == 0)
                    q_poly = mkpoln(2, gen_1, gneg(J));
                else{
                    JJ = gadd(gmul(greal(J), greal(J)), gmul(gimag(J), gimag(J)));
                    q_poly = mkpoln(3, gen_1, gmulsg(-2, greal(J)), JJ);
                }

                c_poly = RgX_mul(c_poly, q_poly);
            }
            a++;
        } while(a * a <= t);

        b += 2;
    } while(b <= B);

    //O codigo abaixo não é muityo bom, mas resolve vários bugs
    for(long i = 2; i <= degree(c_poly)+2; i++){
        if(typ(gel(c_poly, i)) == 6) gel(c_poly, i) = ground(greal(gel(c_poly, i)))
        else gel(c_poly, i) = ground(gel(c_poly, i));
    }

    return gerepileupto(av, gcopy(c_poly));
}

bool curva_ideal(long D, GEN n, GEN *raiz, long *qtd_raiz, GEN *a, GEN *b){
    GEN hpoly, menor_coeficiente;
    *qtd_raiz = 0;
    pari_sp av = avma;
```

```
    hpoly = hilbert(D);

    menor_coeficiente = gel(hpoly, 2);
    if(!Z_ispower(menor_coeficiente, 3)){
        avma = av;
        return false;
    }
    hpoly = RgX_to_FpX(hpoly, n);
    *raiz = gerepileupto(av, FpX_roots(hpoly, n));
    *qtd_raiz = (lg(*raiz)-1);

    //Se eu nao tenho raízes, deve retornar falso
    if(*qtd_raiz > 0) return true;
    else return false;
}

int computa_P(std::pair<GEN, GEN> P, GEN k, GEN n_i, GEN a, std::pair<GEN, GEN> *Pr
    pari_sp ltop = avma, lbot;
    int resposta = 1; //-1 se infinito, 0 se nao tem divisor e 1 se tem
    std::pair<GEN, GEN> A, O, C;
    GEN d, d_x, d_y, m, inv;


    A.first = gcopy(P.first); A.second = gcopy(P.second);
    O.first = gen_0; O.second = gen_1;

    while(resposta && (cmpis(k, 0) == 1)){
        if(!gequal0(Fp_red(k, gen_2))){
            //Fast Point mult & addition
            d = gcdii(Fp_red(subii(O.first, A.first), n_i), n_i);
            k = subis(k, 1);
            if(gequal1(d) || gequal(d, n_i)) resposta = 1;
            else resposta = 0;
            if(!(gequal0(A.first) && gequal1(A.second))){
                if(gequal0(O.first) && gequal1(O.second)){
                    O.first = gcopy(A.first); O.second = gcopy(A.second);
                }
                else if(resposta){
                    d_x = Fp_red(subii(O.first, A.first) , n_i);
```

```
                d_y = Fp_red(subii(O.second, A.second), n_i);
                if(gequal(A.first, O.first) && gequal0(Fp_red(addii(A.second, O
                    C.first = gen_0; C.second = gen_1;
                }
                else{
                    inv = Fp_invsafe(d_x, n_i);
                    if(inv == NULL) inv = gen_0;
                    m = Fp_red(mulii(d_y, inv), n_i);
                    C.first = Fp_red(subii(mulii(m, m), addii(A.first, O.first)
                    C.second = Fp_red(subii(mulii(m, subii(A.first, C.first)),
                }
                O.first = gcopy(C.first); O.second = gcopy(C.second);
            }
        }


    }
    else{
        d = gcdii(Fp_red(mulis(A.second, 2), n_i), n_i);
        k = diviuexact(k, 2);
        if(gequal1(d) || gequal(d, n_i)) resposta = 1;
        else resposta = 0;

        if(resposta){
            inv = Fp_invsafe(mulsi(2, A.second), n_i);
            if(inv == NULL) inv = gen_0;
            m = Fp_red(mulii(addii(mulis(mulii(A.first, A.first), 3), a), inv),
            C.first = Fp_red(subii(mulii(m, m), mulis(A.first, 2)), n_i);
            C.second = Fp_red(subii(mulii(m, subii(A.first, C.first)), A.second
            A.first = gcopy(C.first); A.second = gcopy(C.second);
        }
    }
}

lbot = avma;
Pr->first = Fp_red(O.first, n_i);
Pr->second = gerepile(ltop, lbot, Fp_red(O.second, n_i));

if(gequal0(Pr->first) && gequal1(Pr->second)) return -1;
```

```
    return !resposta;
}

//Declarações de variáveis para a função main
std::chrono::steady_clock::time_point inicio, fim;
std::chrono::duration<double> tempo_decorrido;
std::string n; //Entrada
bool teste, tempo, individual, encontra;
int prox_opcao, k;
const char* const opcoes = "ietc";
const char* nome_programa;

void inline configura(int, char**);
void inline uso();

const struct option argumentos[] = {
{"individual", 0, NULL, 'i'},
{"encontra", 0, NULL, 'e'},
{"tempo", 0, NULL, 't'},
{"certificado", 0, NULL, 'c'},
{NULL, 0,NULL, 0}
};

int main(int argc, char **argv){
    nome_programa = argv[0];
    if(argc == 1) uso();
    else configura(argc, argv);

    pari_init(100000000, 65536); //Aumente para entradas maiores

    /*-------------------------------------
    Tenta provar a primalidade da entrada
    -------------------------------------*/
    if(individual && !encontra){
        while(std::cin >> n){
            tempo_decorrido = std::chrono::steady_clock::duration::zero();
            inicio = std::chrono::steady_clock::now();
            teste = ecpp(n);
            fim = std::chrono::steady_clock::now();
```

```
        if(teste) std::cout << "Provei que e primo " << n << ". ";
        else std::cout <<"Nao consegui provar " << n << ". ";
        if(tempo){
            tempo_decorrido = std::chrono::duration_cast<std::chrono::duration<
            std::cout << " Tempo de execucao: " << tempo_decorrido.count();
        }
        std::cout << std::endl;
    }
}

/*-------------------------------------------------
Me dê o número de bits, eu te dou um inteiro primo
-------------------------------------------------*/
else if(!individual && encontra){
    while(std::cin >> k){
        std::random_device rd;
        std::uniform_int_distribution<short> uni;
        std::mt19937 rgn(rd());
        pari_sp av = avma;
        GEN primo = gen_0, gen_R;
        gen_R = powis(gen_2, k-1);

        inicio = std::chrono::steady_clock::now();
        do{
            av = avma;
            setrand(stoi(uni(rgn)));
            primo = addii(gen_R, randomi(gen_R));
            if(!mpodd(primo)) continue;
            n = std::string(itostr(primo));
            if(ecpp(n)) break;
            avma = av;
        }while(true);
        fim = std::chrono::steady_clock::now();
        pari_printf("%Ps", primo);
        if(tempo){
            tempo_decorrido = std::chrono::duration_cast<std::chrono::duration<
            std::cout << " Tempo de execucao: " << tempo_decorrido.count();
        }
        std::cout << std::endl;
```

```
        }
    }
    else uso();

    pari_close();
    return 0;
}


/*-----------------------------------------
O código abaixo é somente para configuração
-----------------------------------------*/

void inline uso(){
    std::cout << "Utilize assim: " << nome_programa << " opcao [-t] [-c] [ < arquiv
    std::cout <<
" -i --individual Prova a primalidade de primos individuais\n"
" -e --encontra   Encontra um primo de n bits.\n"
" -t --tempo Ativa a contagem de tempo.\n"
" -c --certificado   Imprime o certificado.\n" ;
exit(0);
}

void inline configura(int argc, char** argv){
    certificado = individual = tempo = encontra = false;
    do {
prox_opcao = getopt_long(argc, argv, opcoes, argumentos, NULL);
switch (prox_opcao)
{
case 'i':
individual = true;
break;
case 'c':
certificado = true;
break;
case 'e':
encontra = true;
break;
case 't':
```

```
tempo = true;
break;
case -1:
break;

default:
uso();
}
}
while (prox_opcao != -1);
}
```

ANEXO B – Arquivos utilizados para o teste de performance

B.1 **Arquivo 2**

680327
670303
490271
221227
840823
2980121
2985677
3200203
3061067
2705047
10000993
95000999
75040997
52041001
22141013
221410117
714191069
514191371
124191373
624191377
1247914607
1344914203
1944914221
1744914217
1774914221

B.2 **Arquivo 3**

6241913614
4241913617
8247914627
9247914607
2247914611

17749142149
24749142139
54749142193
84749142151
83449242131
834492421909
234492421903
733252445911
543296445919
143296445921
1432964459191
7432964459183
7834964459183
7834924459207
7131924459169
54321552156217
34321552156211
14321552156209
74321552156209
44321552156207
513215521562137
913215521562119
413215521562141
313215521562157
213215521562129
9132155215621397
2132155215621413
5132155215621407
3132155215621379
1132155215621411