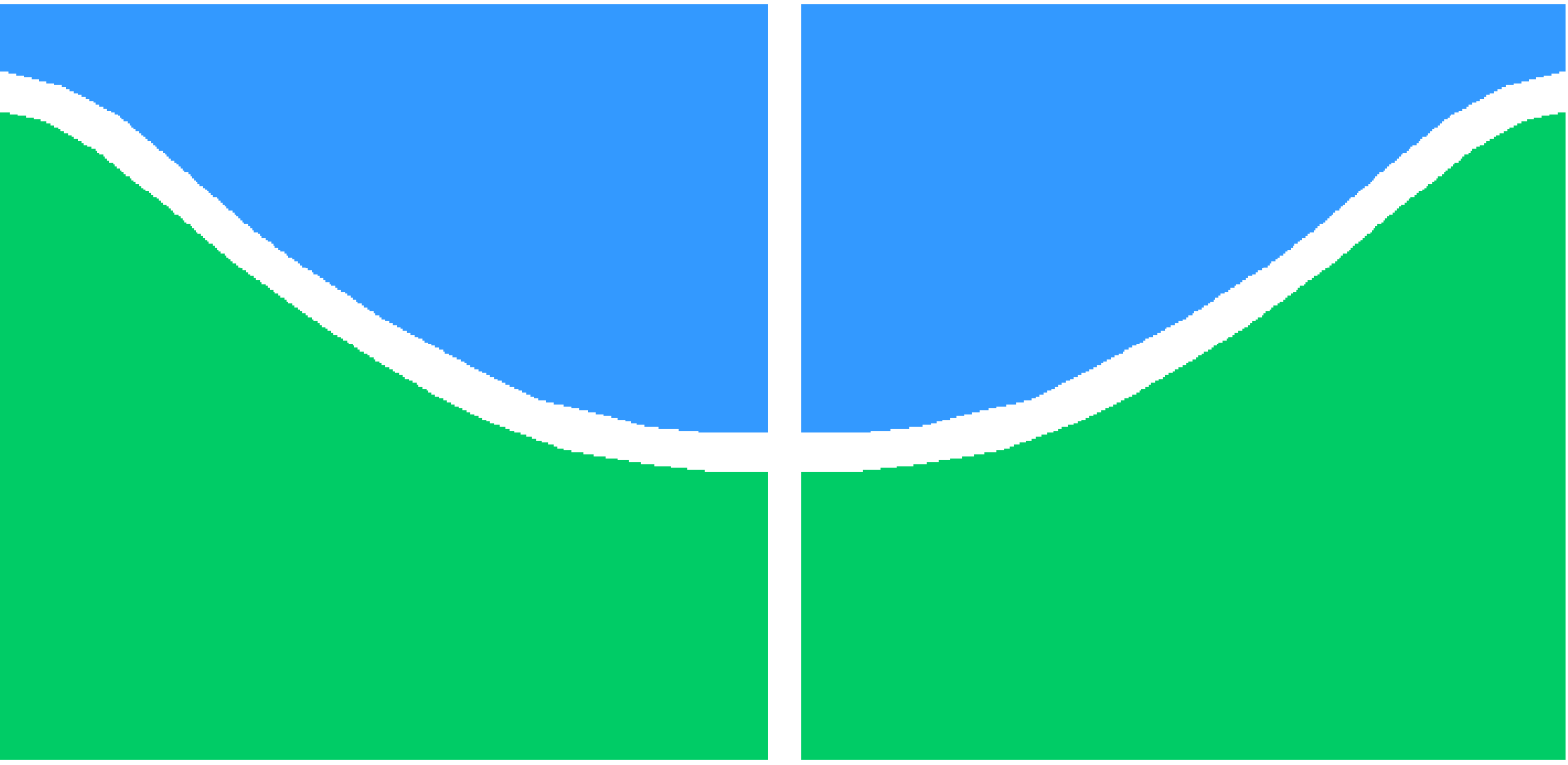

Universidade de Brasília - UnB

Faculdade UnB Gama - FGA

Engenharia Eletrônica

Implementação e análise de algoritmos acerca de números primos para uso em maratonas de programação

Autor: Felipe Duerno do Couto Almeida

Orientador: Prof. Dr. Edson Alves da Costa Júnior

Brasília, DF

2016

Felipe Duerno do Couto Almeida

Implementação e análise de algoritmos acerca de números primos para uso em maratonas de programação

Monografia submetida ao curso de graduação em Engenharia Eletrônicada Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em Engenharia Eletrônica.

Universidade de Brasília - UnB

Faculdade UnB Gama - FGA

Orientador: Prof. Dr. Edson Alves da Costa Júnior

Brasília, DF

2016

Felipe Duerno do Couto Almeida

Implementação e análise de algoritmos acerca de números primos para uso em maratonas de programação

Monografia submetida ao curso de graduação em Engenharia Eletrônicada Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em Engenharia Eletrônica.

Trabalho aprovado. Brasília, DF, 09 de dezembro de 2016:

Prof. Dr. Edson Alves da Costa Júnior
Orientador

Prof. Dr. Fernando William Cruz
Membro 1

Prof. Dr. Tiago Alves
Membro 2

Brasília, DF
2016

Agradecimentos

Agradeço aos meus pais, Ozana Aparecida do Couto Almeida e Weber Jaime de Almeida, por serem tão presentes na minha vida, me ensinando a lutar pelos meus sonhos e sempre estando ao meu lado mesmo nos momentos mais difíceis. Eles são meus maiores exemplos e os principais responsáveis por eu ter chegado onde estou. Sou eternamente grato a eles por isso. Pais, muito obrigado por tudo.

Agradeço ainda às minhas irmãs Julliana do Couto Almeida e Jéssica Karine do Couto Almeida e à todos os meus familiares por sempre me darem apoio e incentivo a continuar nesta batalha em busca dos meus sonhos.

Agradeço a todos os meus amigos, que sempre me apoiaram e estiveram ao meu lado, principalmente nos momentos difíceis. Muito obrigado por me ensinarem e por terem me ajudado a chegar onde estou e ao longo deste TCC.

Agradeço à minha namorada, Natália Santos Mendes, por sempre estar comigo e me apoiar de forma tão ativa e incansável. Muito obrigado por todo o seu apoio.

Agradeço especialmente ao meu orientador Edson Alves da Costa Junior por ser este exemplo de pessoa e humildade e me inspirar a continuar me dedicando cada vez mais ao meu curso e a ajudar ao próximo. Gostaria de ressaltar que ele não só me tornou um melhor programador e aluno, mas graças a ele, hoje sou uma pessoa melhor.

We all use math every day; to predict weather, to tell time, to handle money.
Math is more than formulas or equations; it's logic, it's rationality,
it's using your mind to solve the biggest mysteries we know.
(NUMB3RS)

Resumo

O crescente de iniciativas como o *Google Code Jam*, *ACM ICPC* e diversas competições nacionais e internacionais (muitas vezes denominadas *Hackathons*, fazendo referência às maratonas de programação, do inglês: *Hacking Marathons*), prova que as competições de programação são incentivos realmente eficientes para, no mínimo, o crescimento profissional de qualquer um da área de TI (Tecnologia da Informação). As competições internacionais de programação são comumente divididas em sete grandes tópicos: String, Matemática, Programação Dinâmica, Geometria Computacional, Grafos, Força Bruta e Ad Hoc. Com o foco em Matemática voltada para as maratonas de programação, o presente trabalho tem o objetivo de implementar e analisar os principais algoritmos existentes acerca de números primos, dividindo-os em três grandes conjuntos: Testes de Primalidade, Fatoração e Geração de Prováveis Primos. Para cada conjunto, serão implementados diversos métodos já conhecidos e então estes serão otimizados ao máximo. A análise dos algoritmos será executada seguindo-se o padrão de maratonas de programação: dado um conjunto de entradas, o algoritmo irá gerar um conjunto de saídas que devem ser exatamente igual às saídas de controle, dadas como corretas. Além disso, os algoritmos sempre devem ser executados com um tempo e memória limites. Ao término do trabalho pretende-se obter, para cada algoritmo implementado, suas características de complexidade assintótica de tempo e memória e, por fim, afirmar em quais situações sua utilização é viável.

Palavras-chaves: números primos. programação para competição. fatoração. testes de primalidade. prováveis primos.

Abstract

The growing of initiatives such as the Google Code Jam, ACM ICPC and several national and international competitions, commonly called hackathons (or *Hacking Marathons*), prove that programming competitions are really efficient incentives for, at least, the professional growth of anyone of the IT field. International programming contests are commonly divided into seven major topics: String, Mathematics, Dynamic Programming, Computational Geometry, Graphs, Brute Force and Ad Hoc. Focusing on Mathematics, this study aims to implement and analyze the main existing algorithms about prime numbers, dividing them into three major sets: Primality Tests, Factorization and Generation Probable Primes. For each set, many well-known algorithms will be implemented and then optimized up to their maximums. The analysis of algorithms will be performed following the standard of the programming contests: given a set of inputs, the algorithm will generate a set of outputs. This set of outputs must be exactly the same as a set of control outputs, given as correct. Moreover, the algorithms should always be executed within time and memory limits. Our purpose is to obtain, for each algorithm implemented, its asymptotic complexity characteristics in time and memory, and finally, the proposal is to state in which situations its use is feasible.

Key-words: prime numbers. competitive programming. factorization. primality tests. probable primes.

Lista de ilustrações

Lista de tabelas

Lista de abreviaturas e siglas

ACM	*Association for Computing Machinery*
FGA	Faculdade UnB Gama
ICPC	*International Collegiate Programming Contest*
IoT	*Internet of Things*
TCC	Trabalho de Conclusão de Curso
TI	Tecnologia da Informação

Lista de símbolos

$\mathcal{O}(n)$	Função Big-O
$\Omega(n)$	Função Big-Omega
$\Theta(n)$	Função Big-Theta
$\mathbb{N}$	Conjunto dos números naturais
$\mathbb{Z}$	Conjunto dos números inteiros
$\mathbb{R}$	Conjunto dos números reais
$\mathbb{C}$	Conjunto dos números complexos
$\mathbb{P}$	Conjunto dos números primos
$\pi(n)$	Quantidade de números primos até um inteiro positivo n
$a \in A$	Elemento a pertence ao conjunto A
$a \nmid b$	a não divide b
$a \mid b$	a divide b

Sumário

1 Introdução

A crescente de iniciativas como o *Google Code Jam*, *ACM ICPC* e diversas competições nacionais e internacionais, muitas vezes denominadas *Hackathons* ou *Hacking Marathons*, fazendo referência às maratonas de programação, prova que as competições de programação são incentivos realmente eficientes para, no mínimo, o crescimento profissional de qualquer Engenheiro de Software ou Cientista da Computação.

Atualmente, é possível encontrar maratonas de programação para os mais diversos públicos:

- *IoT*: A multinacional *Informa PLC* realizou há pouco tempo um *hackathon* para desenvolvimento de soluções com *IoT* (INTERNET..., 2016);
- *Hacking cívico*: A empresa *CotidianO* realizou um *hackathon* em Brasília com enfoque em soluções de problemas sociais (COTIDIANO..., 2016);
- *Variados*: O *Hackathon Globo* acontece anualmente, sempre com temas variados e projetos inovadores (HACKATHON..., 2016);
- *Mundialmente conhecidos*: O *Google Code Jam* é uma das maratonas mais conhecidas do mundo, composta por três fases, tem por objetivo encontrar os maiores maratonistas ao redor do mundo (GOOGLE..., 2016).

Estes são só alguns exemplos de quanto a cultura das maratonas de programação está forte atualmente. Este trabalho, no entanto, não faz referência a nenhuma dessas maratonas, mas sim à tradicional e internacional Maratona de Programação, um evento que acontece no Brasil desde 1996 através da Sociedade Brasileira de Computação e é patrocinado pela IBM. A Maratona é dividida em quatro fases: Regional Brasileira, Final Brasileira, Sulamericana e Final Mundial. O objetivo é simples: resolver o maior número possível dos problemas dados em cinco horas de competição.

Nesta competição, os problemas são divididos em sete grandes tópicos: String, Matemática, Programação Dinâmica, Geometria Computacional, Grafos, Força Bruta e Ad Hoc.

É fato que os competidores precisam se dedicar ao máximo para aprender o maior número possível de algoritmos para as competições, porém, dada a falta de um material tão específico, muitas vezes é difícil decidir qual a melhor abordagem e qual algoritmo melhor se encaixa em cada problema e cada situação. Não existe um material com um estudo experimental acerca do desempenho de um grande número de algoritmos específicos voltados para as maratonas de programação.

1.1 Objetivos

A proposta do trabalho é implementar e analisar algoritmos voltados para a Maratona de Programação referentes apenas ao tópico de Matemática, mais especificamente, pretende-se realizar uma análise computacional acerca dos algoritmos que envolvem números primos. O foco do estudo é analisar três conjuntos de algoritmos: verificação de primalidade, geração de prováveis primos e fatoração de inteiros.

O objetivo principal é implementar e analisar, de forma didática, os principais algoritmos de cada conjunto. O estudo é direcionado para uso em maratonas de programação, reduzindo-se assim o escopo dos algoritmos, pois estes devem ser implementados e executados em um curto período de tempo, ou seja, suas implementações precisam ser curtas e ter um desempenho razoável.

Para que as análises sejam confiáveis e os testes sejam sistemáticos, é preciso ainda:

1. Desenvolver uma ferramenta para medição de tempo de execução dos algoritmos;
2. Desenvolver uma ferramenta de automação e sistematização dos testes;
3. Selecionar e implementar uma série de algoritmos acerca de Testes de Primalidade;
4. Selecionar e implementar uma série de algoritmos acerca de Fatoração de Inteiros.

1.2 Estrutura do Trabalho

A primeira etapa do trabalho está dividida em quatro seções, começando pela Fundamentação Teórica, onde são discutidos os fundamentos dos algoritmos acerca de números primos, otimizações e ferramentas da Ciência da Computação para analisar a ordem de complexidade assintótica de algoritmos.

Em seguida é apresentada a Metodologia, onde são apresentados os componentes, recursos e métodos utilizados para a realização da pesquisa, como: Hardware e Software, Linguagens de Programação, Critério de escolha dos algoritmos, Estratégia de testes e avaliações e Cronograma.

Então são apresentados os resultados do estudo, com comparações e análises entre os algoritmos implementados, diversos testes de desempenho, gasto de memória, veracidade de cada algoritmo e comparativos listando vantagens e desvantagens de cada método. Vale ressaltar que todos os algoritmos implementados podem ser encontrados nos apêndices do trabalho.

Por fim, são apresentadas as considerações finais, onde é discutido o que foi feito e os trabalhos futuros.

2 Fundamentação Teórica

As principais características de um algoritmo implementado em maratonas de programação são: baixo tempo de implementação e baixa ordem de complexidade. Para explorar essas características em algoritmos relacionados aos números primos, é preciso entender conceitos acerca da Ciência da Computação e da Teoria dos Números.

Neste capítulo serão apresentadas as ferramentas da Ciência da Computação necessárias para realizar a análise assintótica de complexidade de algoritmos: Big-O, Big-Theta e Big-Omega.

Também serão abordados os conceitos elementares da Aritmética Modular, Teoria do Números Primos e Teoria dos Crivos[1] para o implementação de algoritmos relacionados aos números primos.

Por fim, serão apresentadas as teorias por trás de cada algoritmo implementado.

2.1 Competições de Programação

A programação competitiva muitas vezes é a maior diversão de estudantes de graduação, mas para se sair bem em competições, os chamados "maratonistas" precisam se dedicar e estudar bastante. O objetivo da programação competitiva é resolver, o mais rápido possível, problemas bem conhecidos da ciência da computação. Os irmãos Steven Halin e Felix Halin, autores do livro *Competitive Programming* (HALIM; HALIM, 2013), explicaram a frase termo a termo. O termo "bem conhecidos" vem do fato de todos os problemas de competições já terem sido solucionados, nenhum deles é um problema em aberto ou sem solução conhecida. O termo "resolver" implica que os competidores precisam estudar ao ponto de serem capazes de solucionar estes problemas, eles não precisam chegar a uma solução ótima, mas seu código precisa estar produzindo as mesmas saídas que o autor do problema dentro de um tempo limite de execução estipulado. Finalmente, o termo "o mais rápido possível" é o cerne da competição, a velocidade é um objetivo natural do comportamento humano.

O funcionamento das competições (também chamadas de *maratonas*) de programação é o seguinte: os maratonistas competem em equipes de três pessoas, todas as equipes recebem um conjunto de problemas (geralmente de 7 a 12) e tem um tempo determinado para resolvê-los. Assim que uma equipe termina a codificação de um problema, ela o envia para o juiz eletrônico[2] (sistema utilizado para avaliar o código enviado), os

[1] Crivos são algoritmos que encontram primos através da exclusão sistemática de compostos presentes em uma listagem consecutiva de n inteiros

[2] O Juiz Eletrônico é um sistema que executa o código recebido com diversos casos de teste e compara

problemas são divididos em sete principais temas: Ad Hoc, String, Matemática, Programação Dinâmica, Geometria Computacional, Grafos e Busca Completa. Neste trabalho serão estudados algoritmos que se encaixam no tema *Matemática*.

A codificação correta de um problema em particular resulta em um *AC* (do inglês, *Accepted*), expressão utilizada para quando o juiz eletrônico aceita a resposta da equipe, ou seja, as saídas de todos os casos de teste do código enviado são idênticas às saídas esperadas. No entanto, uma codificação incorreta pode resultar em diversos erros distintos, como pode ser visto na Tabela 1.

Tabela 1 – Lista de códigos de erro acusados por juizes eletrônicos

Código	Descrição	Tradução
WA	Wrong Answer	Resposta Incorreta
TLE	Time Limit Exceeded	Tempo Limite Excedido
MLE	Memory Limit Exceeded	Memória Limite Excedida
RTE	Runtime Error	Erro de Execução
CE	Compilation Error	Erro de Compilação
PE	Presentation Error	Erro de Apresentação

Uma violação muito comum, causada por algoritmos de ordem de complexidade muito elevada para determinado problema, é a violação do tipo TLE. Este código de erro é recebido quando o algoritmo enviado não consegue finalizar sua execução dentro do tempo limite estabelecido. Por isso, é sempre importante se perguntar se o algoritmo com a ordem de complexidade atual consegue resolver o problema no tempo estabelecido. Problemas voltados para a fatoração de números primos e testes de primalidade podem ser resolvidos com uma abordagem bastante fácil, no entanto, estas implementações geralmente excedem o limite de tempo de execução estabelecido por terem uma ordem de complexidade muito alta. Neste tipo de questões é de extrema importância entender os conceitos da análise assintótica de algoritmos.

2.2 Análise assintótica de algoritmos

A questão que a análise assintótica, ou ordem de complexidade assintótica, de algoritmos busca responder é: *qual a taxa de crescimento de recursos necessários à medida em que a entrada aumenta* (EVANS, 2011). Fazer uma análise assintótica é se preocupar com valores grandes de entrada para o processamento do algoritmo, com o intuito de calcular o tempo de processamento ou a memória necessária para a execução do algoritmo. Com isso é possível saber se é necessária a utilização de outra abordagem ou ferramenta para a realização de uma determinada tarefa. Nesta seção são apresentadas três funções

as saídas com a resposta correta, se a resposta correta tenha sido encontrada, o juiz retorna AC (*Accepted*), valor padrão para questões corretas, caso contrário, o juiz retorna o erro encontrado.

utilizadas na Ciência da Computação para identificar as características de quanto os algoritmos necessitam de recursos à medida em que suas entradas aumentam.

2.2.1 Big-O

Dada uma função f de entrada, a função $\mathcal{O}(f)$ é dada pelo conjunto de todas as funções que crescem, no máximo, tão rápido quanto f. Para definir de forma mais formal o conceito de $\mathcal{O}(f)$, é preciso entender o significado do crescimento assintótico de uma função. O crescimento assintótico é a taxa de crescimento de uma função, quando seu argumento tende ao infinito. Na Figura 1 são ilustradas três funções: uma cresce a uma taxa proporcional a $f(n) = n$, outra a $f(n) = n^2$ e a última, uma implementação por força bruta da série de Fibonnacci, a uma taxa exponencial. No gráfico à esquerda, parece que a função n^2 cresce mais rapidamente que $Fibo(n)$, porém, no gráfico à direita fica claro que, ao aumentarmos o argumento das funções, a de crescimento exponencial se mostra muito mais custosa.

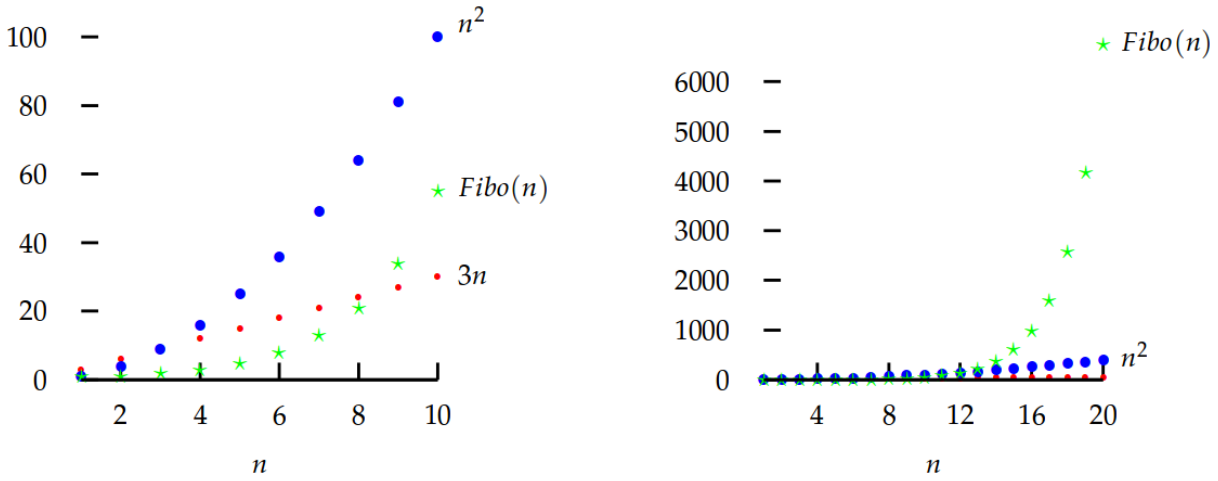

Figura 1 – Ordem de crescimento de funções (EVANS, 2011, p. 131)

A função Big-O, ou $\mathcal{O}(f)$, revela o comportamento assintótico de funções. Mais precisamente:

Definição 1 (Conjunto de funções $\mathcal{O}$) Uma função g pertence ao conjunto de funções de $\mathcal{O}(f)$ se, e somente se, existem constantes c e n_0 tal que, para todo valor $n \leq n_0$,

$$g(n) \leq cf(n). \tag{2.1}$$

A notação $\mathcal{O}(f)$ leva em consideração apenas a proporção de crescimento de um algoritmo, não se preocupando com as constantes que a multiplicam, para facilitar o entendimento, são mostrados alguns exemplos abaixo:

Exemplo 1: $f(n) = \frac{n^3}{10}$

Neste exemplo, f tem a ordem de complexidade $\mathcal{O}(n^3)$. Pois, para qualquer valor de n usado, a diferença entre os valores de $f(n)$ e n^3 não aumenta.

Exemplo 2: $f(n) = 2nlog(n)$

Pelo mesmo motivo do exemplo anterior, f agora tem a ordem de complexidade $\mathcal{O}(nlog(n))$.

2.2.2 Big-Omega

O conjunto $\Omega(f)$ é composto por todas as funções g com ordem de complexidade pelo menos igual a f, este conjunto engloba todas as funções g que crescem com a mesma razão ou mais rapidamente que f, conforme a definição a seguir:

Definição 2 (Conjunto de funções Ω) Uma função g pertence ao conjunto de funções de $\Omega(f)$ se, e somente se, existem constantes c e n_0 tal que, para todo valor $n \leq n_0$,

$$cf(n) \leq g(n). \tag{2.2}$$

2.2.3 Big-Theta

Assim como o conjunto $\mathcal{O}(f)$ é composto por todas as funções g com complexidade menor ou igual a f, o conjunto $\Theta(f)$ denota o conjunto de funções g com ordem de complexidade iguais a f. Uma definição mais formal de $\Theta(f)$ combina as desigualdades das definições de $\mathcal{O}(f)$ e $\Omega(f)$:

Definição 3 (Conjunto de funções Θ) Uma função g pertence ao conjunto de funções de $\Theta(f)$ se, e somente se, existem constantes c_1, c_2 e n_0 tal que, para todo valor $n \leq n_0$, a seguinte desigualdade é verdadeira:

$$c_1 f(n) \leq g(n) \leq c_2 f(n). \tag{2.3}$$

2.3 Teoria dos Números Primos

Os números primos começaram a ser estudados por volta de 300 a.C., época em que Euclides de Alexandria escreveu sua grande obra *Os Elementos* (HEATH C.B., 1908). Desde então, por mais de 2000 anos os números primos foram estudados pelos mais brilhantes matemáticos, desenvolvendo teorias cada vez mais elaboradas, encontrando e utilizando os números primos nas mais diversas áreas da ciência. No entanto, uma pergunta permanece em aberto até os dias de hoje: *qual o seu padrão?*

Para definir números primos, antes é preciso conhecer o conceito de divisibilidade.

Definição 4 (Divisibilidade) Um número inteiro não nulo a divide um inteiro b se existe um inteiro c tal que $b = ac$.

Se a divide b, b é chamado múltiplo de a e a é chamado divisor de b. Se a divide b usa-se a notação $a \mid b$, caso contrário, usa-se $a \nmid b$.

Definição 5 (Divisor próprio) Seja d um inteiro positivo. Dizemos que d é um divisor próprio de um inteiro a se d divide a e $d \neq a$.

As Definições 4 e 5 permitem definir números primos de forma concisa e precisa, evitando o erro comum de interpretar o número 1 como primo[3].

Definição 6 (Número primo) Um inteiro positivo p é primo se possui apenas um único divisor próprio.

Os inteiros positivos diferentes de 1 que não são primos são chamados de números compostos.

De acordo com a Definição 6, o número 2 é o menor número primo, e o único par.

Os primos são considerados os "átomos" da matemática, pois podem ser utilizados para gerar todos os números positivos maiores ou iguais a 2, fato considerado o Teorema Fundamental da Aritmética, enunciado a seguir.

Teorema 7 (Teorema Fundamental da Aritmética) Seja n um número inteiro maior ou igual a 2. Então n pode ser representado como um produto de potências de primos, e esta representação é única (desconsiderada a ordem dos fatores). Mais formalmente, todo número $n \in \mathbb{N}$ maior que 1 pode ser expresso da forma:

$$n = \{{p_1}^{k_1}.{p_2}^{k_2}...{p_N}^{k_N} : k_i \in \mathbb{N}, p_i \in \mathbb{P}, 1 \leq i \leq N\} \tag{2.4}$$

Uma prova para o Teorema 7 pode ser encontrada em (CRANDALL; POMERANCE, 2005), por Euclides. Além de provar que todo número tem uma forma única de fatoração, Euclides também demonstrou a infinitude dos números primos.

Teorema 8 (Infinitude dos números primos) existem infinitos números primos.

Demonstração: seja $n \in \mathbb{N}$ um número formado por N fatores primos distintos e $p_i \in \mathbb{P}$ um de seus fatores primos, o próximo múltiplo de p_i é $n + p_i$, como não existe

[3] A definição comum "Um número primo é um inteiro positivo com apenas dois divisores inteiros positivos, o número 1 e ele mesmo" pode levar a este erro conceito devido ao próprio texto.

um $p \in \mathbb{P}$, tal que $p \mid 1$, então o número $n+1$ não é divisível por nenhum dos primos p_i, gerando assim, pelo menos mais um número primo sempre que este método é aplicado.

Indo um pouco mais a fundo em relação aos números primos, define-se a função $\pi(x)$:

Definição 9 (Contagem de primos) Seja x um inteiro positivo. Define-se a função $\pi(x)$ como a quantidade de números de primos entre 1 e x.

$$\pi(x) = \#\{p \leq x : p \in \mathbb{P}\}. \tag{2.5}$$

A distribuição exata dos números primos não é conhecida, porém, diversas aproximações assintóticas foram encontradas ao longo da história. No século XIX, P. Chebyshev provou o seguinte teorema:

Teorema 10 (Distribuição de $\pi(x)$ por Chebyshev) Existem $a, b \in \mathbb{N}^*$ tal que, para todo $x \geq 3$:

$$\frac{ax}{\ln x} < \pi(x) < \frac{bx}{\ln x}. \tag{2.6}$$

Em 1791, Gauss, aos quatorze anos, propôs uma conjectura acerca do comportamento assintótico dos números primos, mais tarde conhecida como PNT, do inglês, *Prime Number Theorem*. Cerca de 50 anos depois, Chebyshev teve uma grande evolução com o Teorema 10, mas apenas em 1896, *Hadamard* e *de la Vall ée Poussin* provaram, independentemente, sua veracidade.

Teorema 11 (Prime Number Theorem (PNT)) Quando $x \to \infty$, a função $\pi(x)$ tende a $\frac{x}{\ln x}$:

$$\lim_{x \to \infty} \frac{\pi(x)}{\frac{x}{\ln x}} = 1. \tag{2.7}$$

Um estudo mais aprofundado acerca da distribuição dos números primos, relacionada à Função Zeta de Riemann e a Hipótese de Riemann, foge ao escopo deste trabalho, mas pode ser encontrado no Capítulo 3.2 do livro *The Development of Prime Numbers Theory*, de W. Narkiewicz (NARKIEWICZ, 2000).

2.4 Conceitos de Aritmética Modular

Diversos algoritmos relacionados a testes de primalidade e fatoração de inteiros utilizam ferramentas da aritmética modular, alguns mais avançados ainda utilizam álgebra polinomial e curvas elípticas, no entanto, estes estão fora do escopo deste trabalho. Nesta seção serão apresentados conceitos elementares da aritmética modular.

O princípio da aritmética modular é considerar que todos os números inteiros estão dentro de um conjunto de N resíduos, onde N é um número finito, contradizendo a idéia da infinitude dos inteiros.

Definição 12 (Módulo) A operação a módulo N resulta no resto da divisão de a por N. Notação:

$$a \pmod{N} \tag{2.8}$$

Definição 13 (Resíduo) É o resultado da operação módulo.

Definição 14 (Congruência) Dados dois números $a, b \in \mathbb{N}$, a é dito congruente a b módulo N se ambos tem o mesmo resíduo módulo N. Notação:

$$a \equiv b \pmod{N} \tag{2.9}$$

Uma das mais antigas operações da teoria de números é o cálculo do maior divisor comum. Esta operação é fundamental para a implementação dos algoritmos de Pollard-Rho(2.7.5), Pollard $p-1$ (2.7.6) e Lehman (2.7.4), além disso, esta operação é utilizada na construção de vários dos outros algoritmos presentes neste estudo.

Definição 15 (Maior Divisor Comum (GCD)) Dados dois números $x, y \in \mathbb{N}$, o maior divisor comum de x, y é o maior número inteiro que divide x e y.

$$gcd(x, y) = max(n \in \mathbb{N} : n|x, n|y). \tag{2.10}$$

A implementação proposta do Euclides e utilizada no estudo é mostrada abaixo.

Algoritmo 1: Implementação proposta por Euclides para o GCD

input : two integers $x, y > 1$.
output: $gcd(x, y)$

```
begin
    if y is 0 then
        return x;
    return gcd(y, x (mod y));
```

Assim como o *gcd*, outra função utilizada no estudo é o menor múltiplo comum, definido abaixo.

Definição 16 (Menor Múltiplo Comum (LCM)) Dados dois números $x, y \in \mathbb{N}$, o menor múltiplo comum de x, y é o menor número inteiro divisível por x e y.

$$lcm[x, y] = min(n \in \mathbb{N} : x|n, y|n). \tag{2.11}$$

Na aritmética módulo N, todas as operações entre números estão reduzidas ao intervalo [0, N-1]. Por exemplo, a operação $x^y \pmod N$ resulta na y-ésima potência de x, reduzida ao intervalo [0, N-1], a operação $x^{-1} \pmod N$ resulta em um número y, tal que $xy \equiv 1 \pmod N$, e assim por diante.

Potências são operações elementares na aritmética modular. As definições de raízes primitivas, ordem e função totiente de Euler, sobre potências sobre um módulo, são essenciais em vários dos algoritmos apresentados neste estudo.

Definição 17 (Raízes Primitivas) Um número é dito raíz primitiva módulo N, se suas potências formam todos os elementos não nulos módulo N.

Definição 18 (Ordem) A ordem de um número $a \pmod N$ é o menor valor de k, tal que:

$$a^k \equiv 1 \pmod N \tag{2.12}$$

A notação para este valor é:

$$k = ord_n(a) \tag{2.13}$$

Definição 19 (Função totiente de Euler) É a quantidade de números inteiros positivos menores ou iguais a um limiar x, tal que estes números sejam relativamente primos com x. Notação:

$$\varphi(x) = \#\{n \in \mathbb{N} | n \leq x : gcd(n, x) = 1\} \tag{2.14}$$

2.5 Teoria dos Crivos

A Teoria de Crivos é um conjunto de técnicas gerais em teoria dos números, dedicada a contar ou a estimar o tamanho dos conjuntos de números inteiros que passarão por um crivo. Neste capítulo serão apresentados crivos para encontrar o conjunto de números primos existente em um subconjunto de números inteiros. Com os métodos de crivo, o números primos são encontrados através da exclusão sistemática de compostos presentes em uma listagem consecutiva de n inteiros. Os crivos realizam um pré-processamento para encontrar e salvar todos os números primos até um limite n em uma estrutura. Em seguida, realizar um teste de primalidade se reduz à checar se um número está ou não nesta estrutura. Os crivos, por sua natureza, são métodos bastante eficientes para realizar testes de primalidade quando se sabe *a priori* o valor limite dos números a serem testados. Os crivos mais conhecidos são: Crivo de Eratóstenes (HORSLEY, 1772), Crivo de Atkin (ATKIN; BERNSTEIN, 2003) e Crivo de Sundaram (AIYAR, 1934).

Estes são crivos elementares utilizados para contar ou encontrar todos os números primos até um certo valor. Estas técnicas são extremamente rápidas, quando se deseja

encontrar uma quantidade grandes de números primos até um dado limite, visto que todos são computados ao mesmo tempo, porém, por utilizarem uma quantidade muito grande de memória, só são úteis quando se deseja encontrar números primos até um um limite relativamente pequeno, não muito além de aproximadamente 10^8.

Em maratonas de programação, no entanto, este é um ótimo limite de trabalho, o que torna o conhecimento de tais técnicas fundamental para qualquer maratonista.

2.5.1 Crivo de Eratóstenes

O Crivo de Eratóstenes é um algoritmo elementar utilizado para encontrar todos os números primos até um valor limite. A idéia básica é marcar iterativamente como compostos os múltiplos dos números primos encontrados até o momento, partindo dos múltiplos de dois.

Este crivo recebeu o nome de Crivo de Eratóstenes em homenagem ao matemático grego, *Eratosthenes de Cyrene*, por Nicomachus em seu livro Introdução à Aritmética.

2.5.2 Crivo de Atkin

O Crivo de Atkin é um crivo moderno utilizado para encontrar todos os números primos até um valor limite. Este algoritmo foi proposto em 2003 pelos matemáticos A. O. L. Atkin e e D. J. Bernstein. A idéia do algoritmo é utilizar formas quadráticas binárias para encontrar os números primos. O algoritmo verifica a primalidade de um número de entrada $n \in \mathbb{N}$ da seguinte forma:

1. Se n tem raíz quadrada inteira, então n é composto;
2. Se n é múltiplo de 2, 3, ou 5 $\pmod{60}$, então n é composto;
3. Se n tem resto 1, 13, 17, 29, 37, 41, 49, ou 53 $\pmod{60}$, então: se a equação $4x^2 + y^2 = n$ tem um número ímpar de soluções, n é primo, caso contrário, n é composto;
4. Se n tem resto 7, 19, 31, ou 43 $\pmod{60}$, então: se a equação $3x^2 + y^2 = n$ tem um número ímpar de soluções, n é primo, caso contrário, n é composto;
5. Se n tem resto 11, 23, 47, ou 59 $\pmod{60}$, então: se a equação $3x^2 - y^2 = n$ tem um número ímpar de soluções, n é primo, caso contrário, n é composto;

2.5.3 Crivo de Sundaram

O Crivo de Sundaram foi descoberto pelo matemático S. P. Sundaram em 1934. O algoritmo é simples: inicie com uma lista de inteiros de 1 a n, então remova os números

da forma $i+j+2ij \leq n$, onde $i, j \in \mathbb{N}, 1 \leq i \leq j$, por fim, multiplique por dois e some um a cada número restante. Este processo resulta no conjunto de todos os números primos até $2n+1$.

2.6 Testes de Primalidade

Um teste de primalidade é um algoritmo para determinar se um número inteiro é primo ou não. Este tipo de teste é usado em diversas áreas da matemática como, por exemplo, a criptografia. Diferentemente da fatoração de inteiros, os testes de primalidade geralmente não fornecem os fatores primos, indicando apenas se o número fornecido é ou não primo. Nesta seção serão apresentados os testes de primalidade implementados neste estudo.

2.6.1 Divisão Experimental

O modo mais simples de se checar a primalidade de um número n consiste na busca completa: tentar dividir n por todos os números naturais entre 1 e n. Caso todas as divisões resultem em resto positivo, n é um número primo; caso contrário será um número composto.

Dois fatos são decorrentes da definição de divisibilidade (Definições 4 e 5):

1. Não existe um número natural k, com $k > n$, tal que $k \mid n$;
2. Todo número natural n é divisível por, pelo menos, 1 e ele mesmo.

Por estes dois motivos, os testes de primalidade só precisam checar os números entre estes valores.

Checar um número natural n dividindo-o por todos os números naturais entre 1 e n é bem intuitivo. Porém, este algoritmo tem um custo computacional bastante elevado (complexidade $\mathcal{O}(n)$), pois contém muitas redundâncias e cálculos desnecessários. Uma série de fatos e observações podem ser utilizados para otimizar este algoritmo, como visto abaixo.

2.6.1.1 Otimização 1: checar se n é divisível por 2

Se um número n é par e maior que dois, então o número 2 é um divisor próprio de n e, pela Definição 6, n não é primo, de modo que a busca completa pode ser interrompida neste ponto.

Se n é ímpar, então o número 2 não é um divisor de n e ele só tem divisores ímpares. Desta forma, para checar se n é primo, basta checar se ele não é divisível pelos ímpares positivos menores que ele mesmo.

Embora esta otimização diminua o tempo de execução da busca completa, ela não reduz a sua ordem de complexidade, que continua sendo $\mathcal{O}(n)$.

2.6.1.2 Otimização 2: checar apenas se n é divisível por primos

O Teorema 7 diz que, caso um número seja composto, ele terá ao menos um divisor primo e, desta forma, os únicos números que precisam ser testados para afirmar que um número inteiro positivo n é primo são os primos menores que n. Esta otimização reduz não somente o tempo de execução do código, mas também sua ordem de complexidade para $\mathcal{O}(\pi(n))$.

2.6.1.3 Otimização 3: checar apenas os primos até $\sqrt{n}$

Dado um inteiro positivo n, se n é composto, então pelo menos um de seus fatores é menor ou igual a $\sqrt{n}$. Desta forma, o teste de primalidade só precisa checar os números primos até $\sqrt{n}$, o que reduz a ordem de complexidade analítica do algoritmo para $\mathcal{O}(\pi(\sqrt{n}))$.

Este importante fato é enunciado e demonstrado a seguir.

Proposição 20 (Menor divisor próprio) Seja n um inteiro positivo composto. Então n possui um divisor próprio d tal que $d \leq \sqrt{n}$.

Demonstração: Como, por hipótese, n é composto, a Definição 5 e o Teorema 7 garantem que existem dois inteiros positivos a e b tais que $n = ab$.

Suponha, por contradição, que $a, b > \sqrt{n}$. Então podemos escrever:

$$a = \sqrt{n} + c_1, \tag{2.15a}$$
$$b = \sqrt{n} + c_2. \tag{2.15b}$$

Onde $c1, c2 \in \mathcal{R}_+$. Da expressão inicial $n = ab$, temos:

$$n = n + (c_1 + c_2)\sqrt{n} + c_1 c_2. \tag{2.16}$$

Como c_1 e c_2 são reais positivos, o lado direito da Equação 2.16 é sempre maior que o lado esquerdo, chegando assim a uma contradição. Portanto, a suposição $a, b > \sqrt{n}$ não é verdadeira e um dos dois fatores sempre é menor ou igual a $\sqrt{n}$.

Tendo conhecimento de tais otimizações, é possível implementar um algoritmo simples para checar a primalidade de um número. A implementação do teste de primalidade por divisão experimental realizada neste estudo é ilustrada no Algoritmo 2.

Algoritmo 2: Implementação otimizada da Divisão Experimental

input : an integer $n > 1$.
output: *true* if n is prime, and $false$ otherwise.

```
begin
    if n is divisible by 2 then
        return false;
    for i ← 3 to √n, i ∈ ℙ do
        if n is divisible by i then
            return false;
    return true;
```

2.6.2 Crivo de Eratóstenes

O Crivo de Eratóstenes, como enunciado na Seção 2.5.1, é utilizado para encontrar todos os primos até um dado limite n, não sendo eficiente para checar se um único número é primo ou não. Utilizando a definição da Seção 2.5.1 desenvolveu-se uma versão não otimizada do Crivo no Algoritmo 3.

Algoritmo 3: Implementação não otimizada do Crivo de Eratóstenes

input : an integer $n > 1$.
output: all primes in range 2 to n.

```
begin
    A ← boolean array indexed by integers 2 to n, initially all set to true;
    for i ← 2 to n do
        if A[i] is true then
            for j ← 2i, 3i, 4i, ..., not exceeding n do
                A[j] ← false;
    return all i such that A[i] is true;
```

Este algoritmo percorre todos os números inteiros de 2 a n (ordem de complexidade: $\mathcal{O}(n)$) e, para cada primo p encontrado, marcam-se todos os múltiplos p como compostos (ordem de complexidade: $\mathcal{O}(\frac{n}{p})$), o algoritmo completo tem então uma ordem de complexidade de: $\mathcal{O}(n \sum_{p \leq n} \frac{1}{p})$.

Utilizando as otimizações 2.6.1.1, 2.6.1.2 e 2.6.1.3 implementou-se uma versão otimizada do Crivo no Algoritmo 4.

Algoritmo 4: Implementação otimizada do Crivo de Eratóstenes

input : an integer $n > 1$.
output: all primes in range 2 to n.

begin
  $A \leftarrow$ boolean array indexed by integers 2 to n, initially with the odds and 2 set to *true*, and others to $false$;
  for $i \leftarrow 3, 5, 7, ...,$ *not exceeding* $\sqrt{n}$ **do**
    if $A[i]$ *is true* **then**
      for $j \leftarrow i^2, i^2 + 2i, i^2 + 4i, ...,$ *not exceeding* n **do**
        $A[j] \leftarrow false$;
  return all i such that $A[i]$ is *true*;

Este algoritmo percorre apenas os números ímpares de 3 a $\sqrt{n}$ (ordem de complexidade: $\mathcal{O}(\sqrt{n})$) e, para cada primo p encontrado, marcam-se todos os múltiplos, a partir de p^2, como compostos (ordem de complexidade: $\mathcal{O}(\frac{n-p^2}{p})$). O algoritmo completo tem uma ordem de complexidade de: $\mathcal{O}(\sum_{p \leq \sqrt{n}} \frac{n-p^2}{p})$.

2.6.3 Crivo de Atkin

Baseando-se o enunciado da Seção 2.5.2, o Algoritmo 5 implementa o recentemente descoberto Crivo de Atkin.

Algoritmo 5: Implementação do Crivo de Atkin

input : an integer $n > 1$.
output: all primes up to n.

begin
  if n *has integer square root* **then**
    for $j \leftarrow i^2, i^2 + 2i, i^2 + 4i, ...,$ *not exceeding* n **do**
      $A[j] \leftarrow false$;
  $A \leftarrow$ boolean array indexed by integers 2 to n, initially with the odds and 2 set to *true*, and others to $false$;
  for $i \leftarrow 3, 5, 7, ...,$ *not exceeding* $\sqrt{n}$ **do**
    if $A[i]$ *is true* **then**
      for $j \leftarrow i^2, i^2 + 2i, i^2 + 4i, ...,$ *not exceeding* n **do**
        $A[j] \leftarrow false$;
  return all i such that $A[i]$ is *true*;

Algoritmo 6: Implementação do Crivo de Sundaram

input : an integer $n > 1$.
output: all primes up to n.

```
begin
    A ← boolean array indexed by integers 1 to n, initially with all set to true;
    for i ← [1, n] do
        for j ← [i, n] do
            if i + j + 2ij > n then
                break;
            A[i + j + 2ij] ← false;
    return all 2x + 1 such that A[x] is true;
```

2.6.4 Crivo de Sundaram

2.6.5 Pseudoprimos de Fermat

(RIESEL, 2012)

2.6.6 Pseudoprimos Fortes

2.6.7 Miller-Rabin modificado

2.6.8 Método de Miller

2.7 Fatoração de Inteiros

2.7.1 Divisão Experimental

2.7.2 Crivo de Eratóstenes

2.7.3 Método de Fermat

2.7.4 Método de Lehman

2.7.5 Pollard rho

2.7.6 Pollard $p - 1$

3 Metodologia

Neste trabalho serão implementados diversos algoritmos relacionados aos números primos, voltados para o uso em maratonas de programação. Os algoritmos serão divididos em três grupos: Teste de primalidade, Fatoração de inteiros, Geração de pseudo-primos. Todos os algoritmos utilizados ao decorrer deste trabalho foram retirados de bibliografia e não têm objetivo exploratório, visto que a pesquisa em questão é de caráter bibliográfico.

Após a implementação de cada algoritmo, este será testado até o seu limite e então avaliado. Por fim, será feito um comparativo entre os algoritmos, levando em consideração o desempenho, gasto de memória e nível de dificuldade de implementação.

Os testes de desempenho dos algoritmos serão realizados de acordo com o seu grupo.

Os algoritmos de teste de primalidade têm o seguinte fluxo de execução: recebem um inteiro positivo $n > 1$ e retornam todos os números primos de 1 a n. Para este grupo de algoritmos, os testes consistem em identificar o valor máximo de n em que cada algoritmo ainda é capaz de executar dentro do tempo limite estipulado.

Os algoritmos de fatoração de inteiros têm o seguinte fluxo de execução: recebem um inteiro $k > 0$ e uma sequência de k inteiros positivos $n_i \leq L$, onde i representa o i-ésimo número de entrada e L é um valor limite estipulado, então retornam todos os fatores primos de cada número n_i. Neste grupo, os testes consistem em identificar os maiores valores de k e L em que cada algoritmo ainda é capaz de ser executado dentro do tempo limite. A avaliação ainda irá apresentar o menor valor de k para o qual é vantajoso utilizar crivos ao invés de algoritmos de fatoração de um único inteiro positivo por vez.

O fluxo de execução dos algoritmos de geração de prováveis primos será decidido na segunda etapa do trabalho.

Os limites de tempo e memória a serem utilizados, irão seguir os padrões da *ACM International Collegiate Programming Contest* (colegiado internacional de competições de programação, sigla: ACM-ICPC). O tempo limite de execução dos algoritmos será de **três segundos**, já o uso de memória limite disponível para a execução do algoritmo será de 1GB (HALIM; HALIM, 2013).

3.1 Ferramentas

3.1.1 *Hardware* e *Software*

Todos os testes e avaliações serão realizadas utilizando-se apenas um *notebook*, a descrição do seu *hardware* é mostrada na Tabela 2.

Tabela 2 – Configurações de *Hardware* do computador utilizado para o trabalho

Fabricante	Dell Inc.
Modelo	XPS 15 L502X
Processador	Intel Core i7-2670QM CPU 2.20GHz x 8
Memória RAM	7,7 GiB
Hard Disk	147,5 GB
Placa de Vídeo	2GB NVIDIA Corporation GeForce GT 540M

As maratonas de programação são realizadas no mundo inteiro e não seguem um padrão de ambiente para compilação e execução dos códigos dos maratonistas, no entanto, tais configurações não divergem muito das configurações citadas acima. As únicas restrições são: o ambiente do *ejudge* deve ser capaz de compilar e executar os códigos em todas as linguagens de programação permitidas na maratona em questão e deve ser justo ao compilar e executar os códigos de todos os participantes.

O sistema operacional utilizado é o *Elementary OS 0.3.2 Freya* de 64 bits. Sistema operacional baseado no *Ubuntu 14.04*, ainda está em sua versão beta, porém, se mostra bastante estável. O sistema operacional em si não faz diferença no resultado final da pesquisa, o único requisito é que este precisa ser capaz de compilar e executar os códigos desenvolvidos em C/C++ com os compiladores *gcc* e *g++*, e o sistema operacional escolhido atende este requisito.

Todos os códigos foram desenvolvidos utilizando-se os editores de texto *open source* *Atom* (versão 1.3.2) e *Vim* (versão 7.4.52). A ferramenta *Git* foi utilizada para realizar o versionamento do repositório dos códigos e textos produzidos e a plataforma *GitHub* foi utilizada para armazenar este repositório em nuvem.

3.1.2 Linguagens de Programação

Mantendo a tradição das maratonas de programação, todos os algoritmos serão implementados usando o C/C++. Neste trabalho, estão sendo utilizados os compiladores *gcc* e *g++*, ambos na versão (Ubuntu 4.8.4-2ubuntu1 14.04.3) 4.8.4.

Para realizar os testes, será desenvolvido um *script* de compilação que irá funcionar de forma semelhante a um juiz eletrônico, indicando violação do tipo TLE (Seção 2, Tabela 1) e gravando o tempo de execução de todas as implementações. Este *script* será implementado em linguagem C, por utilizar funções de mais baixo nível, como o trata-

mento de sinais e interrupções. No entanto, os algoritmos analisados serão implementados apenas em C++, utilizando o a versão de padronização de C++11, versão aprovada pela Organização Internacional de Padronização (ISO) no dia 12 de agosto de 2011. O trabalho fará uso da biblioteca *Standard Template Library* (STL), biblioteca que fornece quatro importantes componentes para a linguagem C++: *algorithms*, *containers*, *functional* e *iterators* (JOSUTTIS, 2012).

3.2 Critério de Escolha dos Algoritmos

Para escolher os algoritmos a serem implementados de forma eficaz, é preciso analisar o escopo do trabalho. Este trabalho pretende mostrar os melhores algoritmos relacionados aos números primos que podem ser utilizados em maratonas de programação. Um algoritmo bom para maratonas de programação satisfaz os seguintes requisitos: baixas ordens de complexidade temporal e espacial e rápida implementação. Desta forma, os critérios de escolha dos algoritmos implementados neste trabalho, são:

- Veloz (baixa ordem de complexidade temporal);
- Pouco gasto de memória (baixa ordem de complexidade espacial);
- Simples (factível de ser implementado durante uma maratona ≈5h).

3.3 Metodologia de Desenvolvimento

O estudo foi dividido em duas grandes etapas: Algoritmos e Análises. A etapa de Algoritmos consiste na busca por referências, implementação, testes e validação de cada algoritmo e documentação, a etapa de Análises consiste em realizar comparações de desempenho, gasto de memória e simplicidade dos algoritmos implementados e então a documentação destes resultados.

As etapas foram, inevitavelmente, desenvolvidas em cascata, no entanto, cada etapa foi desenvolvida em pequenas iterações. Na etapa de Algoritmos, cada iteração representa a busca, implementação, testes/validação e documentação de um ou mais novos algoritmos. Na etapa de Análises, cada iteração representa o desenvolvimento de uma nova ferramenta, seja de testes automatizados, modelos de apresentação de resultados ou métodos de execução de algoritmos com contagem de tempo e memória, contudo, para que esta nova ferramenta seja dada como pronta, ela deve ser testada e validada de acordo com sua funcionalidade.

Para realizar o gerenciamento das tarefas e entregas, foi utilizado o *ZenHub*, um *KanBan* integrado ao *GitHub*, a figura 2 mostra a utilização desta ferramenta durante o desenvolvimento.

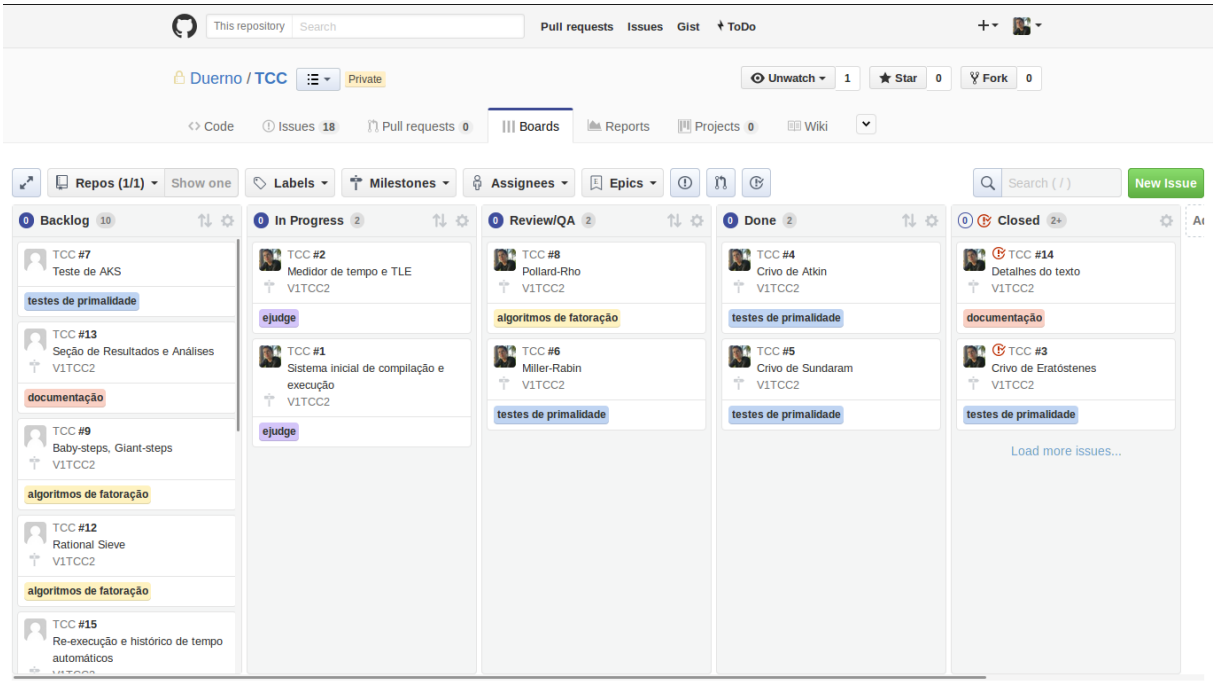

Figura 2 – Exemplo de utilização da ferramenta ágil *ZenHub* integrada ao *GitHub*

3.4 Arquitetura

O ambiente de trabalho foi dividido em três grandes componentes: códigos (*codes*), manuscrito (*paper*) e resultados (*results*). O componente *codes* é responsável por armazenar os códigos fonte dos algoritmos implementados, os testes e análises realizadas e os arquivos de entrada. Todos os arquivos LaTeX referentes ao manuscrito do trabalho estão no componente *paper* e todas as figuras e tabelas de resultados estão armazenadas no componente *results*.

Todos os componentes foram salvos no mesmo repositório no *GitHub*, cada um deles em sua respectiva pasta. Na figura 3 é ilustrada uma versão simplificada da estrutura de arquivos utilizada no trabalho.

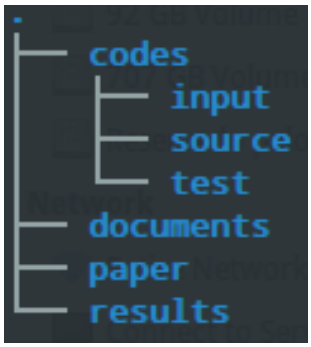

Figura 3 – Ilustração simplificada da estrutura de arquivos do trabalho

Além dos três componentes principais, criou-se uma pasta (*documents*) para ar-

mazenar a documentação referente ao trabalho.

O componente relacionado aos códigos foi dividido em três partes: entradas (*input*), código fonte (*source*) e testes (*test*). Na pasta de entradas estão arquivos contendo os primeiros 1 milhão de primos, retirado da fonte confiável (CALDWELL, 2016). Todos os algoritmos implementados estão organizados dentro da pasta dos códigos fonte, os algoritmos foram divididos em testes de primalidade, fatoração de inteiros e encontrar primos até um dado limite, cada um em seu respectivo arquivo, além disso, foi criado um arquivo para a implementação de funções matemáticas úteis para a implementação dos algoritmos. Por fim, todos os testes estão armazenados na pasta de testes, nesta pasta estão os *scripts* para analisar a corretude dos algoritmos, contagem de tempo de um dado algoritmo, análise automatizada de todos os algoritmos de cada conjunto e geração de arquivos de resultados, como tabelas e gráficos.

3.5 Estratégia de Testes e Avaliações

Após devidamente implementados, os algoritmos precisam ser testados, avaliados e comparados. Como mencionado no início deste capítulo, os testes serão realizados de acordo com a função de cada algoritmo, os algoritmos foram divididos em três grupos, separados pelas seguintes funções: testar primalidade, fatorar inteiros positivos e gerar prováveis primos (também conhecidos como pseudo-primos).

3.5.1 Teste de Primalidade

Todos os algoritmos de teste de primalidade tem o mesmo padrão de execução, recebem um inteiro positivo $n > 1$ e retornam todos os números primos de 1 a n. Os testes e avaliações realizadas neste grupo de algoritmos seguirá o seguinte procedimento:

1. Identificar o valor máximo de n em que é possível executar o código dentro do tempo limite;
2. Identificar a quantidade de memória gasta para executar o código;
3. Mensurar o nível de dificuldade de implementação do algoritmo;
4. Comparar os resultados com os outros algoritmos já testados.

3.5.2 Fatoração de Inteiros

Os algoritmos de fatoração de inteiros funcionam da seguinte maneira: primeiramente é definido um valor inteiro positivo L, este é o valor limite, o maior inteiro positivo que se deseja fatorar, utilizado para todos os algoritmos de fatoração de inteiros. A entrada

destes algoritmos consiste em um inteiro $k > 0$ e uma sequência de k inteiros positivos $n_i \leq L$, onde i é o índice do i-ésimo número de entrada. A saída é o conjunto de fatores primos de cada número n_i de entrada. Para testar e avaliar estes algoritmos, seguir-se-á o seguinte procedimento:

Neste grupo, os testes consistem em identificar os maiores valores de k e L em que cada algoritmo ainda é capaz de ser executado dentro do tempo limite. A avaliação ainda irá apresentar o menor valor de k para o qual é vantajoso utilizar crivos em vez de algoritmos de fatoração de um único inteiro positivo por vez.

1. Identificar os valores máximos do par k e L em que é possível executar o código dentro do tempo limite;
2. Identificar a quantidade de memória gasta para executar o código;
3. Mensurar o nível de dificuldade de implementação do algoritmo;
4. Identificar o menor valor de k para o qual é vantajoso utilizar crivos ao invés de algoritmos de fatoração de um único inteiro positivo por vez;
5. Comparar os resultados com os outros algoritmos já testados.

4 Resultados e Discussão

Algoritmos já implementados; Desempenho de cada algoritmo (tempo e limite de inputs).

5 Considerações Finais

Dada a importância das maratonas de programação em geral, é possível compreender a necessidade de estudos como este. Tal estudo serve como guia para entusiastas à maratonistas competitivos. Por outro lado, pretende-se reunir e comparar diversos algoritmos da área da teoria dos números, revisão dificilmente encontrada na literatura.

A proposta do trabalho é realizar um comparativo entre os principais algoritmos, acerca de números primos, voltados para maratona de programação. Um grande desafio do trabalho é conseguir reunir uma grande quantidade de algoritmos sobre o tema que atendam aos critérios das maratonas de programação, visto que não existe um material específico e completo para tal assunto.

Nesta primeira etapa, foram levantadas as ferramentas matemáticas a serem utilizadas nas duas principais partes do trabalho:

- Implementação e otimização de algoritmos: aritmética modular, definição e propriedades de números primos, otimizações em crivos;
- Análise de algoritmos: análise de complexidade assintótica de algoritmos.

Além disso, o sistema de compilação, execução e análise dos algoritmos, similar a um juiz eletrônico, já se mostra funcional.

5.1 Trabalhos Futuros

Na segunda etapa do trabalho, propõe-se realizar o estudo principal: a implementação, análise e comparativo entre os algoritmos levantados.

Pretende-se realizar três estudos comparativos, um para cada uma das três classes de algoritmos estudadas: teste de primalidade, fatoração e geração de prováveis primos.

A dificuldade em realizar uma análise assintótica precisa dos algoritmos traz um grande risco à conclusão do comparativo teórico de desempenho entre os algoritmos. No entanto, um comparativo baseado no desempenho experimental também será realizado, de forma que todos os algoritmos possam ser comparados de forma justa.

Outra dificuldade é o tratamento do desempenho experimental. Como os algoritmos serão executados sobre um sistema multiprocessado, o erro na medida de desempenho é inevitável. Com este fator, algoritmos com desempenho similar serão colocados em um mesmo grupo e os destaques serão dados aos algoritmos de desempenho claramente superior.

Referências

AIYAR, V. R. Sundaram's sieve for prime numbers. The Mathematics Student, v. 73, 1934. Citado na página 30.

ATKIN, A. O. L.; BERNSTEIN, D. J. Prime sieves using binary quadratic forms. Mathematics of Computation, v. 73, n. 246, p. 1023–1030, 2003. Citado na página 30.

CALDWELL, P. C. *The Prime Pages*. 2016. Disponível em: <http://primes.utm.edu/>. Citado na página 41.

COTIDIANO Hackathon. 2016. Disponível em: <http://www.cotidiano.com.br/hackathon-campcoti1/>. Citado na página 21.

CRANDALL, R.; POMERANCE, C. *Prime Numbers: A Computational Perspetive*. 2. ed. [S.l.]: Springer, 2005. Citado na página 27.

EVANS, D. *Introduction to Computing: Explorations in Language, Logic, and Machines*. 1. ed. [S.l.]: Creative Commons, 2011. Citado 3 vezes nas páginas 11, 24 e 25.

GOOGLE Code Jam. 2016. Disponível em: <https://code.google.com/codejam/>. Citado na página 21.

HACKATHON Globo. 2016. Disponível em: <http://hackathonglobo.com/>. Citado na página 21.

HALIM, S.; HALIM, F. *Competitive Programming 3: The New Lower Bound of Programming Contests*. 3. ed. [S.l.]: Steven & Felix, 2013. Citado 2 vezes nas páginas 23 e 37.

HEATH C.B., S. T. L. *The Thirteen Books of Euclid's Elements*. 2. ed. [S.l.]: Cambridge: at the University Press, 1908. Citado na página 26.

HORSLEY, S. The sieve of eratosthenes. Royal Society, 1772. Disponível em: <http://www.jstor.org/stable/106053>. Citado na página 30.

INTERNET of Things WORLD. 2016. Disponível em: <https://iotworldevent.com/iot-hackathon/>. Citado na página 21.

JOSUTTIS, N. M. *The C++ Standard Template Library*. 2. ed. [S.l.]: Addison-Wesley Professional, 2012. Citado na página 39.

NARKIEWICZ, W. *The Development of Prime Number Theory*. 1. ed. [S.l.]: Springer, 2000. Citado na página 28.

RIESEL, H. *Prime Numbers and Computer Methods for Factorization*. 2. ed. [S.l.]: Birkhauser, 2012. Citado na página 36.

Apêndices

APÊNDICE A – Crivo de Eratóstenes

Texto do apêndice.