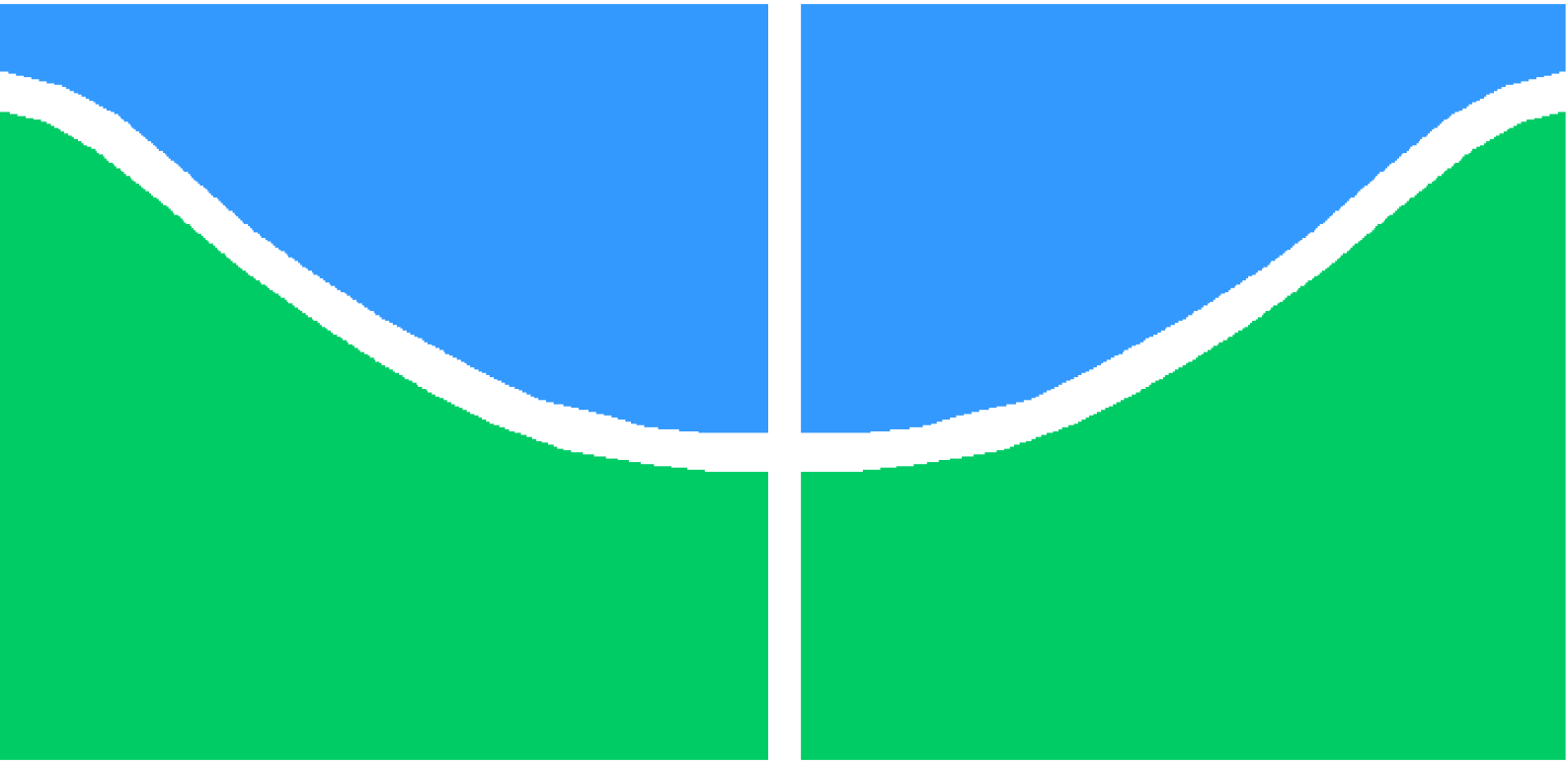

Universidade de Brasília – UnB
Faculdade UnB Gama – FGA
Engenharia Eletrônica

Algoritmos para multiplicação rápida de dois números com mais de 1.000 dígitos cada

Autor: Arthur Luís Komatsu Aroeira
Orientador: Dr. Edson Alves da Costa Júnior

Brasília, DF
2018

Arthur Luís Komatsu Aroeira

Algoritmos para multiplicação rápida de dois números com mais de 1.000 dígitos cada

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

Universidade de Brasília – UnB

Faculdade UnB Gama – FGA

Orientador: Dr. Edson Alves da Costa Júnior

Brasília, DF

2018

Arthur Luís Komatsu Aroeira
Algoritmos para multiplicação rápida de dois números com mais de 1.000 dígitos cada/ Arthur Luís Komatsu Aroeira. – Brasília, DF, 2018-
85 p. : il. (algumas color.) ; 30 cm.

Orientador: Dr. Edson Alves da Costa Júnior

Trabalho de Conclusão de Curso – Universidade de Brasília – UnB
Faculdade UnB Gama – FGA , 2018.

1. FFT. 2. NTT. I. Dr. Edson Alves da Costa Júnior. II. Universidade de Brasília. III. Faculdade UnB Gama. IV. Algoritmos para multiplicação rápida de dois números com mais de 1.000 dígitos cada

CDU 02:141:005.6

Arthur Luís Komatsu Aroeira

Algoritmos para multiplicação rápida de dois números com mais de 1.000 dígitos cada

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

Trabalho aprovado. Brasília, DF, 18 de junho de 2018 – Data da aprovação do trabalho:

Dr. Edson Alves da Costa Júnior
Orientador

Felipe Duerno do Couto Almeida
Convidado 1

Dr. Marcelino Monteiro de Andrade
Convidado 2

Brasília, DF
2018

Agradecimentos

Agradeço primeiramente à minha família que me acompanhou nessa etapa importante da minha vida, sempre me auxiliando e incentivando em todos os momentos, sejam eles bons ou ruins. Agradeço ao meu orientador Edson Alves, com quem aprendi não só diversos tópicos de matemática, mas me fez refletir sobre certos aspectos da vida e isso me fez uma pessoa melhor. Agradeço à minha namorada Victória Gomes por sempre acreditar em mim e me apoiar nas minhas decisões. Por fim, agradeço aos docentes e colegas da Faculdade do Gama, que estiveram presentes em toda minha formação acadêmica.

Resumo

Multiplicação de números grandes possui um papel fundamental na performance de aplicações importantes, como a criptografia. A proposta deste trabalho é apresentar diferentes algoritmos para multiplicação rápida de números inteiros e analisar suas respectivas vantagens e desvantagens. Será visto que é possível encontrar algoritmos mais eficientes do que o método tradicional ensinado em escola, como a Transformada Numérica Teórica (NTT), a qual utiliza conceitos como Transformada Discreta de Fourier, Transformada Rápida de Fourier, Corpos Finitos e Aritmética Modular.

Palavras-chaves: Multiplicação rápida. NTT. FFT. Karatsuba.

Abstract

Multiplication of large integers plays an important role in the performance of certain applications such as cryptography. The purpose of this work is to present different fast multiplication algorithms and analyze their advantages and disadvantages. It will be seen that it is possible to find faster algorithms than the traditional method taught in school, such as the Number Theoretic Transform (NTT), which uses concepts like the Discrete Fourier Transform, Fast Fourier Transform, Finite Fields and Modular Arithmetic.

Key-words: Fast multiplication. NTT. FFT. Karatsuba.

Lista de ilustrações

Lista de tabelas

Índice de algoritmos

Sumário

1 Introdução

A multiplicação de números inteiros é uma das operações matemáticas básicas ensinadas para crianças na escola. O método tradicional comumente utilizado é simples e de fácil aprendizagem, o qual consiste em realizar deslocamentos e somas para multiplicar dois números inteiros. Nos computadores atuais, a Unidade Lógica e Aritmética (ULA) presente no computador é capaz de realizar multiplicações com números de tipicamente 64 bits. Com isso, é possível representar $2^{64} = 18446744073709551616$ valores possíveis, o qual possui 19 dígitos em base decimal. Porém, existem aplicações que demandam multiplicações de números de maior magnitude (i.e. mais de 1000 dígitos).

Uma forma de multiplicar tais números é representar cada dígito dos números a serem multiplicados em posições separadas na memória do computador e aplicar um algoritmo de multiplicação. Será mostrado mais adiante que o algoritmo tradicional ensinado em escola é ineficiente para números com mais de 1000 dígitos e existem algoritmos mais eficazes.

Estes algoritmos de multiplicação podem ser aplicados em *software* ou em um *hardware* matemático dedicado. Vários métodos e suas combinações já foram analisados quanto à adequação em projetos de *hardware*, visando principalmente à plataforma FPGA (*Field-programmable gate array*). Estudos já mostraram que projetos de *hardware* otimizados de multiplicadores, a um custo de uso adicional de recursos de *hardware*, podem oferecer menor latência em comparação com projetos de multiplicadores individuais (RAFFERTY; ONEILL; HANLEY, 2017). Porém, nem toda aplicação embarcada possui *hardware* matemático dedicado. A solução para essas aplicações é implementar o algoritmo em *software*.

Uma aplicação importante de multiplicação de números grandes ocorre na área de segurança da informação. Os sistemas de criptografia atuais são baseados na dificuldade em solucionar problemas matemáticos em aritmética modular, nas quais são utilizadas operações de multiplicação e exponenciação modular de números grandes. Logo, as eficiências dos sistemas criptográficos dependem da eficiência e implementação de algoritmos de multiplicação modular. Na área da Teoria dos Números, multiplicação de números com milhões de dígitos são cruciais para estudar e analisar propriedades especiais de números, como a fatoração de números grandes e o cálculo de constantes matemáticas especiais (por exemplo, o π) com milhões de casas decimais. Estes cálculos e métodos já estão presentes em *softwares* como *Mathematica*, *Maple* e *MATLAB* (Lüders, 2015).

Objetivos

Este trabalho tem como objetivo implementar e analisar algoritmos conhecidos de multiplicação de números inteiros e implementá-los em *software* em um sistema embarcado. Os algoritmos a serem analisados serão:

- algoritmo tradicional de multiplicação;
- algoritmo de Karatsuba;
- algoritmo de multiplicação baseado na Transformada Rápida de Fourier;
- algoritmo de multiplicação baseado na Transformada Numérica Teórica.

Os algoritmos a serem implementados são:

- algoritmo tradicional de multiplicação;
- algoritmo de multiplicação baseado na Transformada Rápida de Fourier;
- algoritmo de multiplicação baseado na Transformada Numérica Teórica.

Os objetivos secundários são as implementações e os testes de alguns destes algoritmos em um notebook com i7-6700HQ de 2.60GHz e 16 GB de memória RAM, um Raspberry Pi 3, um NodeMCU com ESP8266 e um Arduino UNO.

Estrutura do Trabalho

Este trabalho está dividido em 5 capítulos. O Capítulo 2 aborda uma visão geral sobre o tema, como o contexto histórico e a apresentação de alguns algoritmos importantes de multiplicação de números inteiros. O Capítulo 3 descreve a metodologia seguida para a realização do trabalho. O Capítulo 4 apresenta os resultados das análises realizadas para medir as eficiências dos algoritmos. O Capítulo 5 apresenta as considerações finais do trabalho.

2 Fundamentação Teórica

Neste capítulo serão apresentados alguns tópicos essenciais para o entendimento de algoritmos para multiplicação de números inteiros.

2.1 Histórico dos métodos de multiplicação de inteiros

A necessidade de multiplicar números esteve presente em diversas civilizações, como os babilônicos, egípcios, chineses e indianos, com relatos de até 2000 a.C. (BOYER, 1991). Métodos diferentes de multiplicação de números foram desenvolvidos em cada sociedade de forma independente. Influenciados pela presença na escrita em suas civilizações, os babilônicos e os egípcios redigiam tabelas para auxiliar em cálculos de operações aritméticas básicas como adição, subtração, multiplicação e divisão. Em outros momentos e lugares, pessoas usavam técnicas diferentes: seixos, marcas na poeira, cordas amarradas (quipu), tokens em um tabuleiro (Idade Média), além de quadros como o ábaco (ainda usado hoje em países asiáticos) (CHABERT; BARBIN, 1999).

A multiplicação de números grandes com o uso de tabelas possui uma história longa que varia de acordo com o lugar e o período, tendo relatos na China Antiga, no mundo árabe e na Europa medieval. No século XVI, Napier introduziu o conceito de logaritmos que simplificaram multiplicações de números grandes. Apesar de aperfeiçoados no século XIX, tabuleiros (como os ossos de Napier) continuaram a ser fabricados até o século XX, até serem substituídos por réguas de cálculo baseadas em logaritmos. Posteriormente, todos estes métodos se tornaram obsoletos com o advento de calculadoras e computadores (CHABERT; BARBIN, 1999, p. 8).

Após o aperfeiçoamento de técnicas de multiplicação, as pessoas encontraram uma nova forma de entretenimento no século XIX, que consistia em assistir a crianças prodígio que realizavam cálculos matemáticos rápidos em um palco. Desta forma, crianças dotadas eram encorajadas a praticar cálculos mentais para que pudessem ganhar dinheiro. Colburn, Bidder e Dase são exemplos de calculistas famosos que participavam desses eventos e eram capazes de multiplicar números de até 100 dígitos mentalmente (O'CONNOR; ROBERTSON, 1997).

Até 1960, acreditava-se que não era possível encontrar algoritmos para multiplicação de números mais eficientes do que o algoritmo tradicional ensinado em escola, que utiliza cerca de n^2 operações para multiplicar dois números de n dígitos. Andrey Kolmogorov conjecturou em 1956 que o algoritmo tradicional é assintoticamente[1] ótimo. A história mudou em 1960, quando Kolmogorov anunciou sua conjectura em um de seus

[1] assintoticamente no sentido de analisar o comportamento do algoritmo quando o número de dígitos tende ao infinito

seminários e chamou atenção de um espectador: Anatoli Karatsuba. Após pensar ativamente sobre o assunto, Karatsuba desprovou a conjectura em menos de uma semana, encontrando um algoritmo que requer assintoticamente $n^{\log_2 3}$ operações (KARATSUBA, 1995). Curiosamente, esta ideia parece não ter sido descoberta antes de 1960. Nenhum dos calculistas que ficaram famosos por multiplicar grandes números mentalmente relataram o uso deste método, apesar deste algoritmo oferecer uma técnica de multiplicar números de até 8 dígitos mentalmente com prática (KNUTH, 1997, p. 295).

Desde então, algoritmos eficientes para multiplicação de números inteiros foram encontrados, como o algoritmo de Toom–Cook (1966), o algoritmo de Schönhage–Strassen (1971) e o algoritmo de Fürer (2007), nomeados em homenagem aos seus inventores (KNUTH, 1997) (FURER, 2007). Cada um possui vantagens e desvantagens, com suas eficiências relacionadas ao tamanho dos números em dígitos de entrada e à forma como são implementados.

A principal aplicação de algoritmos para multiplicação de números grandes é a criptografia. A teoria da transmissão de informação e a criação de computadores estimulou o avanço da matemática cibernética por volta de 1950 (KARATSUBA, 1995). Em telecomunicações, principalmente com o surgimento da internet, surgiu-se a necessidade de proteger dados transmitidos e garantir que um transmissor seja capaz de enviar uma mensagem a um receptor sem que nenhum intermediário seja capaz de entendê-la. Uma das técnicas para criptografar dados é a criptografia simétrica, a qual se utiliza apenas uma chave para criptografar e descriptografar dados (STALLINGS, 2013, p. 27). Esta técnica é eficiente e torna o trabalho de descriptografar dados muito difícil sem o conhecimento da chave, mas possui um problema notável: o receptor deve conhecer previamente a chave. Logo, o emissor não pode simplesmente enviá-la, pois qualquer intermediário poderia captá-la. Uma solução encontrada foi a criptografia assimétrica, que permitiu a criação de métodos de troca de chaves seguros como a troca de chaves Diffie-Hellman (STALLINGS, 2013, p. 262):

A técnica de criptografia assimétrica revolucionou a história da teoria de segurança da informação e resultou em algoritmos seguros como o RSA. No RSA, as chaves pública e privada dependem das escolhas de dois números primos grandes (tipicamente de 1024 ou 2048 bits) e sua multiplicação. Além disso, o algoritmo faz uso da exponenciação modular, a qual é facilmente computável. Porém, sua inversa (logaritmo discreto) é um problema difícil, o que torna o RSA eficiente. Calcular a exponenciação modular para números grandes é computacionalmente caro e uma forma de melhorar sua eficiência é acelerar o processo de multiplicação de inteiros, o que motivou estudos nessa área (SHAHRAM; AZMAN; KUMBAKONAM, 2014).

2.2 Complexidade assintótica de algoritmos

O objetivo principal deste trabalho é identificar e comparar algoritmos para multiplicação de números inteiros. Mas como medir as suas eficiências? Uma forma de analisar essa eficiência é contar o número total de instruções que o algoritmo realiza, o qual está diretamente ligado ao tempo em que o computador o processa. Porém, a quantidade de instruções pode variar com entradas de tamanhos distintos, o que nos leva à conclusão de que, ao estudar a velocidade de um algoritmo, deve-se analisar seu tempo de execução como uma função de tamanho n (KLEINBERG; TARDOS, 2005, p. 31).

Além disso, entradas de mesmo tamanho podem apresentar desempenhos diferentes em um algoritmo. Como podem haver uma gama enorme de entradas possíveis, expressar a variedade de instâncias para uma entrada como uma função pode se tornar um trabalho complexo. Portanto, convém analisar aquelas entradas que levam o maior tempo de execução de código, denominadas de pior caso. Dessa forma, para uma entrada aleatória de tamanho fixo, é garantido que o seu tempo de execução não ultrapassará o do pior caso. Esse método de análise faz um trabalho suficiente de capturar a eficiência de algoritmos no caso geral (ROUGHGARDEN, 2017).

Para isso, denominaremos a função de complexidade $f(n)$ que mede o tempo necessário para executar um algoritmo para um problema de tamanho n. Para valores pequenos de n, qualquer algoritmo pode ser executado sem muito custo como pode ser visto na Tabela 1. Portanto, a análise de algoritmos é realizada para valores grandes de n, ou seja, o comportamento da função $f(n)$ para n grande[2]. Esse estudo visa entender o comportamento assintótico destas funções, o qual representa o limite do comportamento de $f(n)$ quando n tende ao infinito (ZIVIANI, 2011). Para isso, será utilizada a notação *Big-O* apresentada na Definição 1.

Definição 1. Uma função $f(n)$ é dita $O(g(n))$ se existem duas constantes c_0 e n_0 tais que $f(n) < c_0 g(n)$ para qualquer $n > n_0$ (SEDGEWICK, 1998).

[2] o "grande" depende da aplicação do problema

Exemplo 1. Para exemplificar, vamos analisar a complexidade assintótica do Algoritmo 1.

Algoritmo 1: Algoritmo de exemplo para analisar sua complexidade assintótica

```
1  Function Exemplo(n) is
2      sum ← 0
3      for i ← 1 to n do
4          for j ← 1 to n do
5              sum ← sum + i + j
6          end
7      end
8      for i ← 1 to n do
9          sum ← sum + i
10         sum ← sum/2
11     end
12     return sum
13 end
```

Para analisar a complexidade assintótica do Algoritmo 1, primeiramente deve-se descobrir a quantidade de instruções $f(n)$ que são executados em função de uma entrada n. Inicialmente, a linha 2 faz uma atribuição para a variável *sum*. Além disso, observe que a cada iteração do loop `for` da linha 3, são executados mais n instruções do loop `for` da linha 4, de modo que, no total, são executados n^2 instruções da linha 3 à linha 7. Finalmente, são executados mais $2n$ instruções do loop `for` da linha 8 à linha 11. Então a função de complexidade é $f(n) = n^2 + 2n + 1$.

Vamos mostrar que a função $f(n)$ é $O(n^2)$. Segundo a Definição 1, se $f(n)$ é $O(n^2)$, então existe uma constante c_0 tal que $n^2 + 2n + 1 < c_0 n^2$ para algum $n > n_0$. Vamos checar essa condição: se $n^2 + 2n + 1 < c_0 n^2$, então $1 + 2/n + 1/n^2 < c_0$, o que é verdade para $c_0 = 4$ e $n > n_0 = 1$. Em outras palavras, o termo $2n+1$ de $f(n)$ se torna irrelevante diante do termo n^2 conforme n cresce.

A Tabela 1 mostra o tempo de execução aproximado para algoritmos com as funções de complexidades para determinadas entradas e será utilizada adiante para comparar o ganho de performance em diferentes algoritmos.

Tabela 1 – Tempo aproximado para entradas de tamanho n em um processador rodando 1 milhão de instruções por segundo (KLEINBERG; TARDOS, 2005, p. 34)

	$O(n)$	$O(n \log n)$	$O(n^2)$	$O(n^3)$	$O(1.5^n)$	$O(2^n)$	$O(n!)$
$n = 10$	<1 seg	<1 seg	<1 seg	<1 seg	<1 seg	<1 seg	4 seg
$n = 30$	<1 seg	<1 seg	<1 seg	<1 seg	<1 seg	18 min	10^25 anos
$n = 50$	<1 seg	<1 seg	<1 seg	<1 seg	11 min	36 anos	muito longo
$n = 100$	<1 seg	<1 seg	<1 seg	1 seg	12892 anos	10^17 anos	muito longo
$n = 1000$	<1 seg	<1 seg	1 seg	18 min	muito longo	muito longo	muito longo
$n = 10000$	<1 seg	<1 seg	2 min	12 dias	muito longo	muito longo	muito longo
$n = 100000$	<1 seg	2 seg	3 horas	32 anos	muito longo	muito longo	muito longo
$n = 1000000$	1 seg	20 seg	12 dias	331710 anos	muito longo	muito longo	muito longo

2.3 Algoritmo de multiplicação tradicional ensinada em escola

O primeiro algoritmo de multiplicação a ser estudado é o mais simples, denominado multiplicação ordinária ou multiplicação da escola primária (Lüders, 2015), comumente ensinado para crianças na escola. A Figura 1 ilustra este procedimento para calcular 123×456.

$$
\begin{array}{r}
1\ 2\ 3 \\
\times\ 4\ 5\ 6 \\
\hline
7\ 3\ 8 \\
6\ 1\ 5\ \\
4\ 9\ 2\ \\
\hline
5\ 6\ 0\ 8\ 8
\end{array}
$$

Figura 1 – Método para multiplicação tradicional de escola para 123×456

A ideia por trás deste algoritmo está na multiplicação de polinômios com propagação de *carrys* (Lüders, 2015). Sejam $a = a_{n-1}...a_1a_0$ e $b = b_{m-1}...b_1b_0$ dois números inteiros com n e m dígitos, respectivamente, com $0 \leq a_i, b_i \leq 9$. É possível representar estes números em base 10 como mostrado na Equação 2.1.

$$
\begin{aligned}
a &= \sum_{i=0}^{n-1} a_i \cdot 10^i = a_{n-1} \cdot 10^{n-1} + ... + a_1 \cdot 10^1 + a_0 \cdot 10^0 \\
b &= \sum_{i=0}^{m-1} b_i \cdot 10^i = b_{m-1} \cdot 10^{m-1} + ... + b_1 \cdot 10^1 + b_0 \cdot 10^0
\end{aligned}
\tag{2.1}
$$

Logo, a operação $c = a \cdot b$ pode ser vista como uma multiplicação dos polinômios apresentados na Equação 2.1. O deslocamento realizado serve para alinhar as potências de 10 e facilitar a soma de seus coeficientes. A Figura 2 apresenta a ilustração deste procedimento com o mesmo exemplo da Figura 1.

			$1{\cdot}10^2$	$2{\cdot}10^1$	$3{\cdot}10^0$
$\times$			$4{\cdot}10^2$	$5{\cdot}10^1$	$6{\cdot}10^0$
			$6{\cdot}10^2$	$12{\cdot}10^1$	$18{\cdot}10^0$
		$5{\cdot}10^3$	$10{\cdot}10^2$	$15{\cdot}10^1$	
$+$	$4{\cdot}10^4$	$8{\cdot}10^3$	$12{\cdot}10^2$		
	$4{\cdot}10^4$	$13{\cdot}10^3$	$28{\cdot}10^2$	$27{\cdot}10^1$	$18{\cdot}10^0$

Figura 2 – Mutiplicação 123×456 vista como uma multiplicação de polinômios

Como é necessário que os algarismos c_i do resultado estejam no intervalo $[0, 9]$, deve-se passar o valor excedente (denominado *carry*) para o coeficiente do próximo índice. Este procedimento é chamado de propagação de *carrys*. A técnica consiste em escrever $c_i = c_i{}' + k \cdot 10$ com $0 \leq c_i{}' \leq 9$ e realizar a propagação da forma apresentada na Equação 2.2.

$$\begin{aligned} c_{i+1} \cdot 10^{i+1} + c_i \cdot 10^i &= c_{i+1} \cdot 10^{i+1} + (c_i{}' + k \cdot 10) \cdot 10^i \\ &= c_{i+1} \cdot 10^{i+1} + c_i{}' \cdot 10^i + k \cdot 10^{i+1} \\ &= (c_{i+1} + k) \cdot 10^{i+1} + c_i{}' \cdot 10^i \end{aligned} \tag{2.2}$$

Logo, a propagação de *carrys* do resultado da Figura 2 fica da forma mostrada na Figura 3.

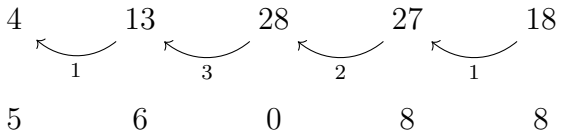

Figura 3 – Procedimento da propagação de *carrys* da Figura 2

Por se tratar de uma multiplicação de polinômios, a multiplicação Tradicional dos números a e b pode ser vista como uma convolução dos vetores $a = [a_{n-1}, ..., a_1, a_0]$ e $b = [b_{n-1}, ..., b_1, b_0]$ seguida pela propagação de *carrys* do resultado. A complexidade do algoritmo da convolução dos vetores a e b pela definição é $O(nm)$ (como será estudado na Seção 2.5) e da propagação de *carrys* é $O(n+m)$. Logo, a complexidade total do algoritmo de multiplicação tradicional é $O(nm+n+m) = O(nm)$ (BRENT; ZIMMERMANN, 2010, p. 4). Caso $m = n$, a complexidade se torna $O(n^2)$. É assintoticamente o algoritmo de multiplicação mais lento presente na literatura, porém o mais simples (KALACH; DAVID; TITTLEY, 2007). Conforme visto na Tabela 1, essa complexidade é eficiente para números com até 1000 dígitos.

2.4 Algoritmo de Karatsuba

O algoritmo de Karatsuba é um algoritmo que utiliza a técnica "dividir e conquistar" para multiplicar números inteiros (ou polinômios) e foi introduzido por Karatsuba em 1960 e publicado em 1962 (SHAHRAM; AZMAN; KUMBAKONAM, 2014) (KNUTH, 1997, p. 295).

O algoritmo de Karatsuba na base 10 está apresentado em pseudo-código no Algoritmo 2.

Algoritmo 2: Algoritmo de Karatsuba em base 10 (BRENT; ZIMMERMANN, 2010)

Entrada: Dois números $A = a_{n-1}...a_1a_0$ e $B = b_{n-1}...b_1b_0$ em base 10 e um número n_0

Saídas : O número $C = A \cdot B = c_{2n-1}...c_1c_0$

```
1  Function KARATSUBA (A, B, n_0) is
2      n ← max(sizeof(A), sizeof(B))
3      if n < n_0 then
4          return MultiplicacaoTradicional(A, B)
5      end
6      k ← ⌈n/2⌉
7      A_0 ← (a_{n-1}, ..., a_k)
8      A_1 ← (a_{k-1}, ..., a_0)
9      B_0 ← (b_{n-1}, ..., b_k)
10     B_1 ← (b_{k-1}, ..., b_0)
11     C_0 ← KARATSUBA(A_0, B_0, n_0)
12     C_1 ← KARATSUBA(A_1, B_1, n_0)
13     C_2 ← KARATSUBA(A_0 + B_0, A_1 + B_1, n_0)
14     return C_0 10^{2k} + (C_2 − C_0 − C_1)10^k + C_1
15 end
```

O Algoritmo 2 pode ser explicado em 4 passos (BRENT; ZIMMERMANN, 2010, p.5):

- **Caso base** (linhas 3, 4, 5 do Algoritmo 2): Caso a quantidade de algarismos seja menor que uma constante n_0 predefinida, retorna-se $A \cdot B$ utilizando a multiplicação ordinária de escola, pois para valores pequenos de n, a multiplicação tradicional acaba sendo mais eficiente. A escolha do valor da constante n_0 depende do processador e seu custo relativo de multiplicação e adição.
- **Dividir** (linhas 6, 7, 8, 9, 10 do Algoritmo 2): Considerando[3] $k = \lceil n/2 \rceil$, os inteiros

[3] $\lceil x \rceil$ é a função teto de x e representa o maior número inteiro maior ou igual a x

A e B são escritos conforme a Equação 2.3.

$$\begin{aligned} A &= A_0 \cdot 10^k + A_1 \\ B &= B_0 \cdot 10^k + B_1 \end{aligned} \tag{2.3}$$

Isso equivale em dividir A e B pela metade de seus algarismos, formando os números apresentados na Equação 2.4.

$$\begin{aligned} (A_0, A_1) &= (a_{n-1}...a_k, a_{k-1}...a_0) \\ (B_0, B_1) &= (b_{n-1}...b_k, b_{k-1}...b_0) \end{aligned} \tag{2.4}$$

- **Conquistar** (linhas 11, 12, 13 do Algoritmo 2): Calculam-se três multiplicações:

$$\begin{aligned} C_0 &= A_0 \cdot B_0 \\ C_1 &= A_1 \cdot B_1 \\ C_2 &= (A_0 + B_0) \cdot (A_1 + B_1) \end{aligned} \tag{2.5}$$

- **Combinar** (linha 14 do Algoritmo 2): Da Equação 2.3, temos que

$$\begin{aligned} C &= A \cdot B = (A_0 \cdot 10^k + A_1)(B_0 \cdot 10^k + B_1) \\ C &= A_0B_0 \cdot 10^{2k} + (A_0B_1 + A_1B_0) \cdot 10^k + A_1B_1 \\ C &= A_0B_0 \cdot 10^{2k} + ((A_0 + B_0)(A_1 + B_1) - A_0A_1 - B_0B_1) \cdot 10^k + A_1B_1 \\ C &= C_0 \cdot 10^{2k} + (C_2 - C_1 - C_0) \cdot 10^k + C_1 \end{aligned} \tag{2.6}$$

Por fim, é importante analisar a complexidade assintótica do Algoritmo de Karatsuba. Seja $T(n)$ a função que representa a quantidade de instruções realizadas no algoritmo para dois números de entrada n. As linhas 11, 12 e 13 do Algoritmo 2 chamam novamente a função com a metade do tamanho atual. Posteriormente, são realizados mais n operações na linha 14. Então a equação de recorrência que relaciona $T(n)$ está apresentada na Equação 2.7 e é possível mostrar que $T(n)$ possui limite assintótico $O(n^{\log_2 3})$ por meio do *Master Theorem* (KNUTH, 1997, p. 295) (CORMEN et al., 2001).

$$T(n) = 3T\left(\frac{n}{2}\right) + O(n) = O(n^{\log_2 3}) \tag{2.7}$$

Como será discutido mais adiante, é possível encontrar algoritmos de multiplicação com menor complexidade computacional.

2.5 Convolução Discreta Circular

A Convolução Discreta Circular é uma das operações mais comuns em processamento digital de sinais e é um conceito que também pode ser utilizado para alguns algoritmos de multiplicação de números inteiros (ver Seção 2.3). Logo, otimizar a operação de convolução permite melhorar a eficiência de algoritmos de multiplicação e isso pode ser feito utilizando o Teorema da Convolução (apresentada na Seção 2.5.2.2) juntamente com a Transformada Rápida de Fourier (apresentada na Seção 2.5.3) (TOLIMIERI; AN; LU, 1997, p. 134).

2.5.1 Definição de Convolução

A definição de convolução está apresentada na Definição 2.

Definição 2. Considere os vetores $a = [a_0, a_1, ..., a_{n-1}]$ e $b = [b_0, b_1, ..., b_{m-1}]$ de tamanhos n e m, respectivamente. A convolução circular de a e b (notação $a * b$) é o vetor c de tamanho $N = m + n - 1$ cujos elementos são definidos por (TOLIMIERI; AN; LU, 1997, p. 134):

$$c_k = \sum_{i=0}^{k} a_i \cdot b_{k-i}, \quad \text{com } k = 0, 1, ..., N-1$$

O Algoritmo 3, escrito em pseudo-código, apresenta uma implementação da operação de convolução de dois vetores a e b usando a Definição 2.

Algoritmo 3: Algoritmo de convolução de 2 vetores pela definição

Entrada: Dois vetores a e b
Saídas : O vetor $c = a * b$

```
1  Function Convolução (a, b) is
2      n ← sizeof(a)
3      m ← sizeof(b)
4      c[n + m − 1] ← (0, 0, ..., 0)
5      for i ← 0 to n − 1 do
6          for j ← 0 to m − 1 do
7              c[i + j] ← c[i + j] + a[i] · b[j]
8          end
9      end
10     return c
11 end
```

A complexidade do Algoritmo 3 é $O(nm)$, pois para cada uma das n iterações do loop `for` da linha 5, há mais m iterações do loop `for` da linha 6.

2.5.2 Transformada Discreta de Fourier (DFT)

A Transformada Discreta de Fourier ou DFT (*Discrete Fourier Transform*) é uma poderosa ferramenta que permite analisar o conteúdo em frequência de uma sequência de números que representam amostras igualmente espaçadas de uma função no tempo (COCHRAN et al., 1967). Essas amostras podem ser representações de sinais contínuos no tempo (i.e. áudio e eletromiografia). Além disso, existem técnicas que permitem utilizar a DFT em multiplicação de matrizes, polinômios e de números inteiros de forma eficiente (AGARWAL; BURRUS, 1975), o qual será o foco dessa Seção. Antes de apresentar a definição da DFT (Seção 2.5.2.2), é fundamental conhecer as raízes complexas da unidade, apresentada na Seção 2.5.2.1.

2.5.2.1 Raízes n-ésimas complexas da unidade

Uma raiz n-ésima complexa da unidade é um número complexo ω tal que $\omega^n = 1$. Como um polinômio de grau n possui exatamente n raízes complexas, o polinômio $\omega^n - 1$ possui exatamente n raízes. Então há exatamente n raízes n-ésimas complexas da unidade (GUIMARAES, 2008, p. 41). É possível encontrar todas as n raízes conforme mostrado no Teorema 1.

Teorema 1. As raízes n-ésimas complexas da unidade são dadas por $\omega_n^0, \omega_n^1, ..., \omega_n^{n-1}$ tais que $\omega_n = e^{2\pi i/n}$.

Demonstração. Queremos encontrar todas as n raízes do polinômio $\omega^n - 1$, ou seja, resolver a equação $\omega^n = 1$. Utilizando a equação de Euler ($e^{ix} = cos(x) + i\ sen(x)$), podemos reescrever $\omega^n = 1$ como:

$$\begin{aligned} \omega^n &= 1 = cos(2\pi k) + i\ sen(2\pi k) \quad \forall k \in \mathbb{Z} \\ \omega^n &= e^{2\pi ik} \\ \omega &= e^{2\pi ik/n} \end{aligned} \tag{2.8}$$

Como $f(k) = e^{2\pi ik/n}$ é uma função periódica de período n, podemos definir um intervalo $0 \leq k < n \mid k \in \mathbb{Z}$ o que fará com que $f(k)$ gere todas as n raízes da unidade (GUIMARAES, 2008, p. 41).

□

Denominaremos $\omega_n = e^{2\pi i/n}$ como sendo a principal n-ésima raiz complexa da unidade. Logo, todas as outras raízes da unidade serão potências inteiras de ω_n (CORMEN et al., 2001, p. 831).

Podemos visualizar as raízes da unidade geometricamente no plano complexo. Elas estão situadas em um círculo de raio unitário com ângulos igualmente espaçados entre si. A Figura 4 ilustra as raízes da unidade para $n = 8$.

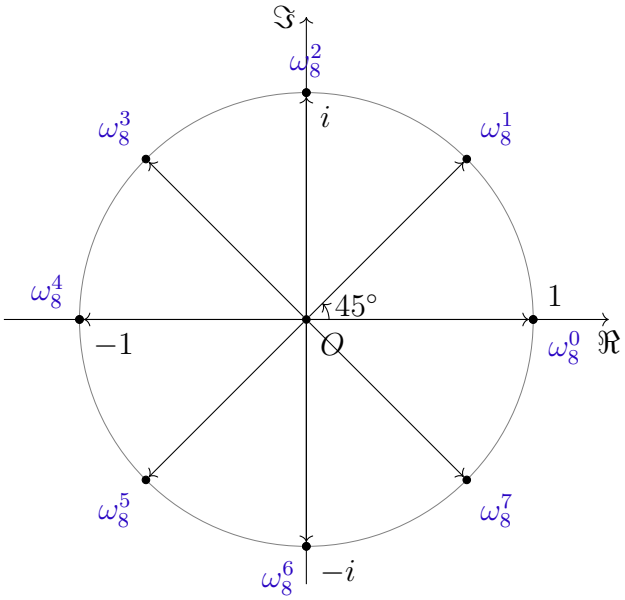

Figura 4 – Representação visual das raízes da unidade para $n = 8$

As raízes da unidade possuem propriedades especiais que serão utilizadas mais adiante, descritas nos Lemas 1 e 2.

Lema 1. Se n e k forem inteiros não-negativos com n par, então $(\omega_n^k)^2 = (\omega_n^{k+n/2})^2$.

Demonstração.

$$\begin{aligned}(\omega_n^k)^2 = \omega_n^{2k} = e^{\frac{4\pi i}{n}k} &= \cos\left(\tfrac{4\pi k}{n}\right) + i\sin\left(\tfrac{4\pi k}{n}\right)\\ &= \cos\left(\tfrac{4\pi k}{n} + 2\pi\right) + i\sin\left(\tfrac{4\pi k}{n} + 2\pi\right)\\ &= \cos\left(\tfrac{2\pi}{n}(2k+n)\right) + i\sin\left(\tfrac{2\pi}{n}(2k+n)\right)\\ &= e^{\frac{2\pi i}{n}(2k+n)} = \left(e^{\frac{2\pi i}{n}(k+n/2)}\right)^2\\ &= (\omega_n^{k+n/2})^2\end{aligned} \tag{2.9}$$

□

Lema 2. Se n e k forem inteiros não-negativos com n par, então $\omega_n^{k+n/2} = -\omega_n^k$.

Demonstração.

$$\omega_n^{k+n/2} = e^{\frac{2\pi i}{n}(k+n/2)} = e^{\frac{2\pi i}{n}k+\pi i} = e^{\pi i}e^{\frac{2\pi i k}{n}} = -\omega_n^k$$

□

2.5.2.2 Definição da Transformada Discreta de Fourier

A definição da Transformada Discreta de Fourier está apresentada na Definição 3.

Definição 3. Seja $a = \{a_0, a_1, ..., a_{n-1}\}$ uma sequência de n números complexos. A Transformada Discreta de Fourier de a (notação $DFT(a)$) transforma a sequência a na sequência $A = \{A_0, A_1, ..., A_{n-1}\}$ de números complexos cujos elementos são definidos por (COCHRAN et al., 1967):

$$A_k = \sum_{j=0}^{n-1} a_j e^{-2\pi ikj/n}, \qquad \text{com } k = 0, 1, ..., n-1 \text{ e } i = \sqrt{-1}$$

A DFT pode ser vista como uma atribuição dos pontos $\{\omega_n^0, \omega_n^1, ..., \omega_n^{n-1}\}$ no polinômio $A(x) = a_0x^0 + a_1x^1 + ... + a_{n-1}x^{n-1}$ com $\omega_n = e^{-2\pi i/n}$ como sendo a principal raiz n-ésima complexa da unidade (CORMEN et al., 2001, p. 833).

A inversa da DFT existe e é similar à sua definição. O Teorema 2 apresenta a Transformada Inversa Discreta de Fourier.

Teorema 2. A Transformada Inversa Discreta de Fourier da sequência $A = \{A_0, ..., A_{n-1}\}$ (notação $IDFT(A)$) é a sequência $a = \{a_0, a_1, ..., a_{n-1}\}$ cujos elementos são (COCHRAN et al., 1967):

$$a_k = \frac{1}{n} \sum_{j=0}^{n-1} A_j e^{2\pi ijk/n}, \qquad \text{com } k = 0, 1, ..., n-1 \text{ e } i = \sqrt{-1}$$

Uma importante aplicação da DFT é o Teorema da Convolução (apresentado no Teorema 3), que permite o cálculo da Convolução Circular utilizando a Transformada Discreta de Fourier. Isso é conveniente principalmente quando há algoritmos para computação rápida da DFT (TOLIMIERI; AN; LU, 1997, p. 139).

Teorema 3. A DFT da Convolução Circular dos vetores a e b pode ser calculada como (SMITH, 2007):

$$DFT(a * b) = DFT(a) \cdot DFT(b) \tag{2.10}$$

em que a operação $a \cdot b$ representa o produto termo a termo de a e b.

Demonstração.

$$
\begin{aligned}
a_m * b_m &= \sum_{k=0}^{n-1} a_k b_{m-k} \\
&= \sum_{k=0}^{n-1} \left[\frac{1}{n} \sum_{j=0}^{n-1} A_j e^{2\pi i k j/n} \right] \left[\frac{1}{n} \sum_{l=0}^{n-1} B_l e^{2\pi i (m-k) l/n} \right] \\
&= \frac{1}{n} \sum_{j=0}^{n-1} A_j \sum_{l=0}^{n-1} B_l \frac{1}{n} \sum_{k=0}^{n-1} e^{2\pi i k n/n} e^{2\pi i (m-k) l/n} \\
&= \frac{1}{n} \sum_{j=0}^{n-1} A_j \sum_{l=0}^{n-1} B_l e^{2\pi i m l/n} \frac{1}{n} \sum_{k=0}^{n-1} e^{2\pi i k (n-l)/n} \\
&= \frac{1}{n} \sum_{j=0}^{n-1} A_j \sum_{l=0}^{n-1} B_l e^{2\pi i m l/n} \delta[n-l] \\
&= \frac{1}{n} \sum_{j=0}^{n-1} A_j B_j e^{2\pi i m j/n} \\
&= IDFT(A_m \cdot B_m)
\end{aligned}
\tag{2.11}
$$

Logo, $a_m * b_m = IDFT(A_m \cdot B_m)$ implica em $DFT(a_m * b_m) = A_m \cdot B_m \quad \forall m \in \{0, 1, ..., n-1\}$ □

2.5.3 Transformada Rápida de Fourier (FFT)

Como a Transformada de Fourier Discreta (DFT) é essencial para análise de sinais digitais, surgiu-se a necessidade de implementar algoritmos rápidos para sua computação. Calcular a DFT pela definição possui complexidade $O(n^2)$ e é eficiente apenas para sinais com até 1000 amostras, como visto na Tabela 1. Um sinal de áudio de 1 minuto, por exemplo, amostrado a 128 kbps (taxa comum de amostra de sinais de áudio) precisaria de $60 \cdot 128 \cdot 1000 = 7680000$ amostras e tornaria o cálculo de sua DFT pela definição muito lento. Um algoritmo para computação dos coeficientes de Fourier que requer menos esforço computacional foi descoberto por Cooley e Tukey em 1965 e este método é amplamente conhecido como Transformada Rápida de Fourier ou FFT *(Fast Fourier Transform)* (COCHRAN et al., 1967). Com esta técnica, a complexidade para computação da DFT se tornou $O(n \log n)$ e sua eficiência proporcionou as soluções de diversos problemas substancialmente mais rápidas do que no passado.

Considere um vetor $a = [a_0, a_1, ..., a_{n-1}]$ de tamanho n. O algoritmo da Transformada Rápida de Fourier mostrada por Cooley e Tukey para calcular $DFT(a)$ está apresentada na forma de pseudo-código no Algoritmo 4.

Algoritmo 4: Algoritmo da Transformada Rápida de Fourier (CORMEN et al., 2001, p. 835)

Entrada: O vetor $a = [a_0, a_1, ..., a_{n-1}]$
Saídas : O vetor $A = DFT(a) = [A_0, A_1, ..., A_{n-1}]$

1 **Function** $FFT\ (a)$ **is**
2 $n \leftarrow sizeof(a)$; ▷ n é uma potência de 2
3 **if** $n = 1$ **then**
4 **return** a
5 **end**
6 $a^{\text{par}}[\,] \leftarrow (a_0, a_2, ..., a_{n-2})$
7 $a^{\text{impar}}[\,] \leftarrow (a_1, a_3, ..., a_{n-1})$
8 $A^{\text{par}}[\,] \leftarrow FFT(a^{\text{par}})$
9 $A^{\text{impar}}[\,] \leftarrow FFT(a^{\text{impar}})$
10 $\omega \leftarrow 1$
11 **for** $k \leftarrow 0$ **to** $n/2 - 1$ **do**
12 $A_k \leftarrow A_k^{\text{par}} + \omega \cdot A_k^{\text{impar}}$
13 $A_{k+n/2} \leftarrow A_k^{\text{par}} - \omega \cdot A_k^{\text{impar}}$
14 $\omega \leftarrow \omega \cdot e^{-2\pi i/n}$; ▷ $\omega = \omega \cdot \omega$
15 **end**
16 **return** A
17 **end**

Queremos calcular a DFT de a, ou seja, devemos computar o polinômio $A(x) = a_0x^0 + a_1x^1 + ... + a_{n-1}x^{n-1}$ nos pontos $\{\omega_n^0, \omega_n^1, ..., \omega_n^{n-1}\}$ com $\omega_n = e^{-2\pi i/n}$ (ver Seção 2.5.2). A FFT utiliza uma estratégia de dividir e conquistar, podendo ser explicada em 4 passos (CORMEN et al., 2001):

- **Caso base** (linhas 3, 4, 5 do Algoritmo 4): Caso o tamanho do vetor da FFT seja 1, a DFT será ela mesma por definição.

- **Dividir** (linhas 6, 7 do Algoritmo 4): O polinômio $A(x)$ é dividido em dois novos polinômios de tamanho $n/2$ utilizando os índices ímpares e pares de a como mostrado na Equação 2.12. Nota-se que é essencial que n seja par em todas as camadas da recursão, logo assumiremos que o tamanho n de $A(x)$ é uma potência de 2, sem perda de generalidade[4].

$$\begin{aligned} A^{\text{par}}(x) &= a_0x^0 + a_2x^1 + ... + a_{n-2}x^{n/2-1} \\ A^{\text{impar}}(x) &= a_1x^0 + a_3x^1 + ... + a_{n-1}x^{n/2-1} \end{aligned} \tag{2.12}$$

[4] sempre é possível tornar o tamanho do vetor como uma potência de 2 adicionando-se novos elementos com valor 0 ao final

- **Conquistar** (linhas 8, 9 do Algoritmo 4): Após a divisão de a, calcula-se recursivamente a DFT de a^{par} e a^{impar}.

- **Combinar** (linhas 10, 11, 12, 13, 14, 15 do Algoritmo 4): O polinômio $A(x)$ pode ser reconstruído como:

$$A(x) = A^{\text{par}}(x^2) + xA^{\text{impar}}(x^2) \tag{2.13}$$

Então o problema de computar $A(x)$ nos pontos $\{\omega_n^0, \omega_n^1, ..., \omega_n^{n-1}\}$ se reduz a:

1. computar $A^{\text{par}}(x)$ e $A^{\text{impar}}(x)$ nos pontos $\{\omega_n^0, \omega_n^1, ..., \omega_n^{n-1}\}$
2. combinar os resultados de acordo com a Equação 2.13.

A razão pela qual a FFT é eficiente é porque não é necessário passar por todas as n raízes da unidade para reconstruir $A(x)$, mas apenas metade delas. Isso se deve ao fato de os quadrados das raízes da unidade formarem pares contendo os mesmos valores (Lema 1). Portanto, a combinação é feita separando A_k em dois intervalos:

1. $0 \leq k < n/2$ (linha 12):

$$\begin{aligned} A_k &= A(\omega_n^k) \\ &= A^{\text{par}}((\omega_n^k)^2) + \omega_n^k A^{\text{impar}}((\omega_n^k)^2) \\ &= A_k^{\text{par}} + \omega_n^k \cdot A_k^{\text{impar}} \end{aligned} \tag{2.14}$$

2. $n/2 \leq k < n$ (linha 13):

$$\begin{aligned} A_{k+n/2} &= A(\omega_n^{k+n/2}) \\ &= A^{\text{par}}((\omega_n^{k+n/2})^2) + \omega_n^{k+n/2} A^{\text{impar}}((\omega_n^{k+n/2})^2) \\ &= A^{\text{par}}((\omega_n^k)^2) - \omega_n^k A^{\text{impar}}((\omega_n^k)^2) \qquad \text{(Lema 1 e 2)} \\ &= A_k^{\text{par}} - \omega_n^k \cdot A_k^{\text{impar}} \end{aligned} \tag{2.15}$$

Por fim, a linha 14 atualiza o valor de ω para cada iteração para obter ω_n^k, já que $\omega_n^k = \omega \cdot \omega_n^{k-1}$.

Observa-se que, para calcular a DFT inversa utilizando a FFT, basta utilizar $\omega_n = e^{2\pi i/n}$ e depois dividir todos os elementos por N como apresentado no Teorema 2.

Por fim, é importante analisar a complexidade assintótica do Algoritmo da FFT. Seja $T(n)$ a função que representa a quantidade de instruções realizadas no algoritmo para dois números de entrada n. As linhas 8 e 9 do Algoritmo 2 chamam novamente a função com a metade do tamanho atual. Posteriormente, são realizados mais n operações das linhas 11 à 15. Então a equação de recorrência que relaciona $T(n)$ está apresentada

na Equação 2.16 e é possível mostrar que $T(n)$ possui limite assintótico $O(n \log n)$ por meio do *Master Theorem* (KNUTH, 1997, p. 295) (CORMEN et al., 2001).

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n) = O(n \log n) \tag{2.16}$$

Exemplo 2. Vamos calcular a DFT do vetor $v = [1, 2, 3, 0, 0, 0, 0, 0]$ utilizando a FFT. Lembrando que a FFT divide o vetor em coeficientes pares e ímpares, a Figura 5 apresenta uma árvore de recursão na qual cada nível representa uma divisão do vetor acima.

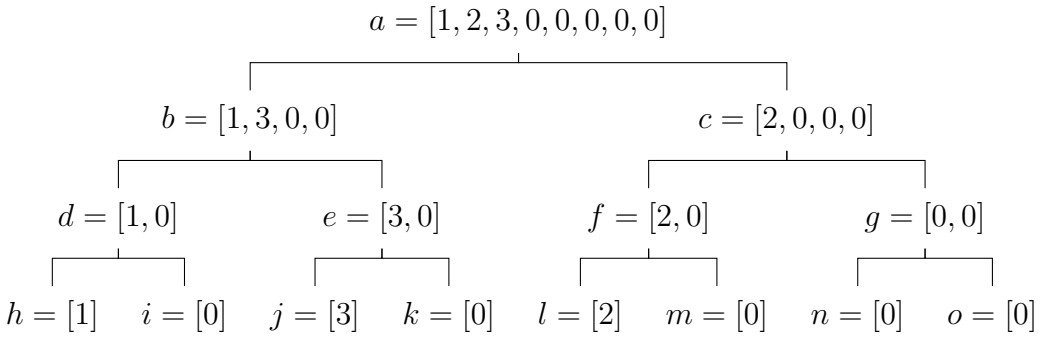

Figura 5 – Árvore de recursão da FFT do Exemplo 2

A seguir, foi calculado a FFT dos vetores dos níveis da árvore da Figura 5 utilizando o Algoritmo 4. Considere os vetores com letras maiúsculas as DFTs dos vetores com letras minúsculas:

1. Os vetores h, i, j, k, l, m, n e o possuem apenas um elemento, logo representam o caso base do Algoritmo 4, já que a FFT de um vetor de apenas um elemento é ele mesmo.

2. Os vetores d, e, f e g possuem 2 elementos. Logo, como apresentado na técnica de combinação do Algoritmo 4, a FFT é calculada como sendo:

   - Vetor d:

     $$D_0 = H_0 + \omega_2^0 \cdot I_0 = 1 + 1 \cdot 0 = 1$$
     $$D_1 = H_0 - \omega_2^0 \cdot I_0 = 1 - 1 \cdot 0 = 1$$

     Então $D = FFT(d) = [1, 1]$.

   - Vetor e:

     $$E_0 = J_0 + \omega_2^0 \cdot K_0 = 3 + 1 \cdot 0 = 3$$
     $$E_1 = J_0 - \omega_2^0 \cdot K_0 = 3 - 1 \cdot 0 = 3$$

     Então $E = FFT(e) = [3, 3]$.

- Vetor f:

  $$F_0 = L_0 + \omega_2^0 \cdot M_0 = 2 + 1 \cdot 0 = 2$$
  $$F_1 = L_0 - \omega_2^0 \cdot M_0 = 2 - 1 \cdot 0 = 2$$

  Então $F = FFT(f) = [2, 2]$.

- Vetor g:

  $$G_0 = N_0 + \omega_2^0 \cdot O_0 = 0 + 1 \cdot 0 = 0$$
  $$G_1 = N_0 - \omega_2^0 \cdot O_0 = 0 - 1 \cdot 0 = 0$$

  Então $G = FFT(g) = [0, 0]$.

3. Os vetores b e c possuem 4 elementos. Logo, a combinação é realizada da seguinte forma:

   - Cálculo das raízes da unidade:

     $$\omega_4^0 = 1$$
     $$\omega_4^1 = \omega_4^0 \cdot e^{-2\pi i/4} = -i$$

   - Vetor b:

     $$B_0 = D_0 + \omega_4^0 \cdot E_0 = 1 + 1 \cdot 3 = 4$$
     $$B_2 = D_0 - \omega_4^0 \cdot E_0 = 1 - 1 \cdot 3 = -2$$
     $$B_1 = D_1 + \omega_4^1 \cdot E_1 = 1 + (-i) \cdot 3 = 1 - 3i$$
     $$B_3 = D_1 - \omega_4^1 \cdot E_1 = 1 - (-i) \cdot 3 = 1 + 3i$$

     Então $B = FFT(b) = [4, 1 - 3i, -2, 1 + 3i]$.

   - Vetor c:

     $$C_0 = F_0 + \omega_4^0 \cdot G_0 = 2 + 1 \cdot 0 = 2$$
     $$C_2 = F_0 - \omega_4^0 \cdot G_0 = 2 - 1 \cdot 0 = 2$$
     $$C_1 = F_1 + \omega_4^1 \cdot G_1 = 2 + (-i) \cdot 0 = 2$$
     $$C_3 = F_1 - \omega_4^1 \cdot G_1 = 2 - (-i) \cdot 0 = 2$$

     Então $C = FFT(c) = [2, 2, 2, 2]$.

4. Por fim, o vetor a possui 8 elementos. Logo, a combinação é realizada da seguinte forma:

   - Cálculo das raízes da unidade:

     $$\omega_8^0 = 1$$
     $$\omega_8^1 = \omega_8^0 \cdot e^{-2\pi i/8} = \frac{\sqrt{2}}{2} - \frac{\sqrt{2}}{2}i$$
     $$\omega_8^2 = \omega_8^1 \cdot e^{-2\pi i/8} = -i$$
     $$\omega_8^3 = \omega_8^2 \cdot e^{-2\pi i/8} = -\frac{\sqrt{2}}{2} - \frac{\sqrt{2}}{2}i$$

- Vetor A:

$$A_0 = B_0 + \omega_8^0 \cdot C_0 = 4 + 1 \cdot 2 = 6$$
$$A_4 = B_0 - \omega_8^0 \cdot C_0 = 4 - 1 \cdot 2 = 2$$
$$A_1 = B_1 + \omega_8^1 \cdot C_1 = 1 - 3i + (\tfrac{\sqrt{2}}{2} - \tfrac{\sqrt{2}}{2}i) \cdot 2 = 1 + \sqrt{2} + (-3 - \sqrt{2})i$$
$$A_5 = B_1 - \omega_8^1 \cdot C_1 = 1 - 3i - (\tfrac{\sqrt{2}}{2} - \tfrac{\sqrt{2}}{2}i) \cdot 2 = 1 - \sqrt{2} + (-3 + \sqrt{2})i$$
$$A_2 = B_2 + \omega_8^2 \cdot C_2 = -2 + (-i) \cdot 2 = -2 - 2i$$
$$A_6 = B_2 - \omega_8^2 \cdot C_2 = -2 - (-i) \cdot 2 = -2 + 2i$$
$$A_3 = B_3 + \omega_8^3 \cdot C_3 = 1 + 3i + (-\tfrac{\sqrt{2}}{2} - \tfrac{\sqrt{2}}{2}i) \cdot 2 = 1 - \sqrt{2} + (3 - \sqrt{2})i$$
$$A_7 = B_3 - \omega_8^3 \cdot C_3 = 1 + 3i - (-\tfrac{\sqrt{2}}{2} - \tfrac{\sqrt{2}}{2}i) \cdot 2 = 1 + \sqrt{2} + (3 + \sqrt{2})i$$

Então $A = FFT(a) = [6, 1+\sqrt{2}+(-3-\sqrt{2})i, -2-2i, 1-\sqrt{2}+(3-\sqrt{2})i, 2, 1-\sqrt{2}+(\sqrt{2}-3)i, -2+2i, 1+\sqrt{2}+(3+\sqrt{2})i]$.

O resultado encontrado pode ser conferido calculando-se a DFT de v pela Definição 3:

$$v_0 = \sum_{j=0}^{7} v_j e^{-2\pi i j \cdot 0/8} = 6$$
$$v_1 = \sum_{j=0}^{7} v_j e^{-2\pi i j \cdot 1/8} = 1 + \sqrt{2} + (-3 - \sqrt{2})i$$
$$v_2 = \sum_{j=0}^{7} v_j e^{-2\pi i j \cdot 2/8} = -2 - 2i$$
$$v_3 = \sum_{j=0}^{7} v_j e^{-2\pi i j \cdot 3/8} = 1 - \sqrt{2} + (3 - \sqrt{2})i$$
$$v_4 = \sum_{j=0}^{7} v_j e^{-2\pi i j \cdot 4/8} = 2$$
$$v_5 = \sum_{j=0}^{7} v_j e^{-2\pi i j \cdot 5/8} = 1 - \sqrt{2} + (-3 + \sqrt{2})i$$
$$v_6 = \sum_{j=0}^{7} v_j e^{-2\pi i j \cdot 6/8} = -2 + 2i$$
$$v_7 = \sum_{j=0}^{7} v_j e^{-2\pi i j \cdot 7/8} = 1 + \sqrt{2} + (3 + \sqrt{2})i$$

É interessante observar que, utilizando a FFT para este exemplo, foram realizadas 76 operações de soma e multiplicação. Já utilizando a definição de DFT, foram utilizadas 112 operações.

2.5.4 Algoritmo de multiplicação de números inteiros utilizando FFT

Sejam dois números inteiros $a = \sum_{i=0}^{n-1} a_i \cdot 10^i$ e $b = \sum_{i=0}^{m-1} b_i \cdot 10^i$. Da Seção 2.3, foi analisado que a multiplicação $a \cdot b$ pode ser vista como uma convolução dos vetores $a = [a_0, a_1, ..., a_{n-1}]$ e $b = [b_0, b_1, ..., b_{m-1}]$ seguida pela propagação de *carrys*. Utilizando a FFT, é possível computar a convolução $a * b$ em tempo $O(n \log n)$ seguindo os passos:

1. Para o algoritmo da convolução com a FFT funcionar corretamente, os tamanhos dos vetores devem ser uma potência de 2 maior que o tamanho do vetor do resultado da convolução entre eles. Para dois vetores a e b de tamanhos n e m respectivamente, o tamanho de $a * b$ é $n + m - 1$ (ver Seção 2.5). Logo, deve-se preencher os vetores a e b com elementos nulos ao final de modo que o tamanho novo N seja uma potência de 2 maior ou igual a $n + m - 1$.

2. Calcular a DFT de a e b com a FFT em $O(n \log n)$ (ver Seção 2.5.3):

$$\begin{aligned} A &= DFT(a) = [A_0, A_1, ..., A_{N-1}] \\ B &= DFT(b) = [B_0, B_1, ..., B_{N-1}] \end{aligned} \tag{2.17}$$

3. Calcular o produto termo a termo de A e B em $O(n)$:

$$C = A \cdot B = [A_0 B_0,\ A_1 B_1, ...,\ A_{N-1} B_{N-1}] \tag{2.18}$$

4. Pelo Teorema da Convolução (Teorema 3), é possível obter a convolução de $a * b$ a partir da Transformada de Fourier Inversa de $A \cdot B$ em $O(n \log n)$:

$$a * b = IDFT(A \cdot B) = IDFT(C) \tag{2.19}$$

Após o cálculo da convolução, realizar a propagação de *carrys* (ver Seção 2.3) em $O(n)$. A complexidade total do algoritmo é $O(2n \log n + n + n \log n + n) = O(n \log n)$.

Exemplo 3. Considere novamente o exemplo 123×456. A seguir, os passos utilizados para calcular esta multiplicação utilizando a FFT:

1. Queremos encontrar o resultado da convolução dos vetores $a = [1, 2, 3]$ e $b = [4, 5, 6]$. O tamanho dos vetores deve ser uma potência de 2 maior que $n+m-1 = 3+3-1 = 5$. A próxima potência de 2 maior que 5 é 8. Logo, os novos vetores a serem analisados são:

$$\begin{aligned} a &= [1, 2, 3, 0, 0, 0, 0, 0] \\ b &= [4, 5, 6, 0, 0, 0, 0, 0] \end{aligned} \tag{2.20}$$

2. Calcular $A = DFT(a)$ e $B = DFT(b)$ com FFT. O vetor A já foi calculado no Exemplo 2. O vetor B pode ser calculado de forma análoga:

$$\begin{aligned} A = DTF\,(a) = \Big[\ & 6, \quad 1+\sqrt{2}+\left(-3-\sqrt{2}\right)i, \quad -2-2i, \quad 1-\sqrt{2}+\left(3-\sqrt{2}\right)i, \\ & 2, \quad 1-\sqrt{2}+\left(-3+\sqrt{2}\right)i, \quad -2+2i, \quad 1+\sqrt{2}+\left(3+\sqrt{2}\right)i\Big] \\ B = DFT\,(b) = \Bigg[& 15, \quad 4+\frac{5}{\sqrt{2}}+\left(-6-\frac{5}{\sqrt{2}}\right)i, -2-5i, \quad 4-\frac{5}{\sqrt{2}}+\left(6-\frac{5}{\sqrt{2}}\right)i, \\ & 5, \quad 4-\frac{5}{\sqrt{2}}+\left(-6+\frac{5}{\sqrt{2}}\right)i, -2+5i, \quad 4+\frac{5}{\sqrt{2}}+\left(6+\frac{5}{\sqrt{2}}\right)i\Bigg] \end{aligned}$$

3. Calcular o produto C termo a termo de A e B:

$$\begin{aligned} c_0 &= a_0 \cdot b_0 = 90 \\ c_1 &= a_1 \cdot b_1 = -14 - 7\sqrt{2} + (-28 - 20\sqrt{2})i \\ c_2 &= a_2 \cdot b_2 = -6 + 14i \\ c_3 &= a_3 \cdot b_3 = -14 + 7\sqrt{2} + (28 - 20\sqrt{2})i \\ c_4 &= a_4 \cdot b_4 = 10 \\ c_5 &= a_5 \cdot b_5 = -14 + 7\sqrt{2} + (-28 + 20\sqrt{2})i \\ c_6 &= a_6 \cdot b_6 = -6 - 14i \\ c_7 &= a_7 \cdot b_7 = -14 - 7\sqrt{2} + (28 + 20\sqrt{2})i \end{aligned} \tag{2.21}$$

4. Por fim, calcula-se $a * b$ utilizando a FFT inversa de C:

$$a * b = IDFT(A \cdot B) = IDFT(C) = [4, 13, 28, 27, 18, 0, 0, 0]$$

Observa-se que é o mesmo resultado do encontrado na Figura 2. Realizando a propagação de *carrys*, encontra-se o resultado $123 \times 456 = 56088$.

Na implementação prática da FFT, nota-se que se deve utilizar algum tipo de dado básico para tratar pontos flutuantes (i.e. `double` da linguagem C) para trabalhar com os números complexos. Isso leva ao aparecimento de erros numéricos durante a computação da FFT, os quais podem ser significativos quando o vetor de entrada possuir muitos números inteiros. Uma alternativa para evitar erros numéricos é modificar a FFT para utilizar um corpo finito na implementação da DFT (GOURDON, 2001). Esta técnica é denominada de *Number Theoretic Transform* e será estudada na Seção 2.6.

2.6 Transformada de Fourier Discreta em um Corpo Finito

O *Number Theoretic Transform* (NTT) é uma Transformada Discreta de Fourier aplicada em um conjunto de números inteiros. Dessa forma, evita-se o uso de pontos

flutuantes e apenas operações com números inteiros são realizadas. Antes de apresentar a definição da NTT (Seção 2.6.3), é necessário entender alguns conceitos importantes na área da Teoria dos Números, como Grupos, Anéis e Corpos Finitos (Seção 2.6.1) e raízes da unidade módulo n (Seção 2.6.2).

2.6.1 Grupos, Anéis, Corpos Finitos

Considere um conjunto A e uma operação denotada por $\oplus$ que relaciona seus elementos. Então o conjunto A que satisfaz as seguintes condições é denominado de grupo (notação $(A, \oplus)$) (NUSSBAUMER, 1981, p. 24):

1. Propriedade associativa: $a \oplus (b \oplus c) = (a \oplus b) \oplus c$ para qualquer $a, b, c \in A$;
2. Elemento neutro: existe um elemento $e \in A$ denominado elemento neutro tal que $e \oplus a = a$ para qualquer $a \in A$;
3. Elemento simétrico: cada elemento $a \in A$ possui seu elemento simétrico $\bar{a}$ tal que $a \oplus \bar{a} = \bar{a} \oplus a = e$.

Quando a operação $\oplus$ também é comutativa ($a \oplus b = b \oplus a \ \forall \ a, b \in A$), o grupo é chamado de *Abeliano*. A ordem do grupo é a quantidade de elementos deste grupo.

Um conjunto A é um anel com respeito às operações $\oplus$ e $\otimes$ caso as seguintes condições são satisfeitas (NUSSBAUMER, 1981, p.25):

1. $(A, \oplus)$ é um grupo Abeliano;
2. Se $c = a \otimes b$, para $a, b \in A$, então $c \in A$;
3. Propriedade associativa: $a \otimes (b \otimes c) = (a \otimes b) \otimes c$;
4. Propriedade distributiva: $a \otimes (b \oplus c) = a \otimes b \oplus a \otimes c$ e $(b \oplus c) \otimes a = b \otimes a \oplus c \otimes a$.

Um anel é comutativo caso a operação $\otimes$ seja comutativa. Adicionalmente, um anel é unitário caso exista um (e apenas um) elemento neutro para a operação $\otimes$. É possível mostrar, por exemplo, que o conjunto de inteiros formam um anel comutativo e unitário com respeito às operações de adição e multiplicação (COHEN, 1993).

Um anel comutativo A unitário é chamado de corpo se todo elemento $a \in A$ possuir um (e apenas um) elemento simétrico $a \otimes \bar{a} = u$. Um corpo com um número finito de elementos é denominado de corpo finito. Por exemplo, é possível mostrar que, para todo primo p, o conjunto $\{0, 1, ..., p-1\}$ forma um corpo finito com as operações de adição e multiplicação no conjunto dos inteiros ($\mathbb{Z}$) módulo p. Este corpo é denominado de corpo de Galois e é denotado por $GF(p)$ ou $\mathbb{Z}/p\mathbb{Z}$ (NUSSBAUMER, 1981, p.25).

2.6.2 Raízes da unidade módulo n

Assim como os números complexos, é possível definir raízes da unidade para um anel com o uso de aritmética modular. Uma raiz k-ésima módulo n da unidade é um número inteiro positivo a tal que $a^k \equiv 1 \pmod{n}$.

Um conceito fundamental na análise de raízes da unidade módulo n são as raízes primitivas módulo n. Introduzidas por Gauss em 1801, as raízes primitivas têm aplicações importantes como na criptografia e em algoritmos de teste de primalidade (COHEN, 1993, p. 84). A definição de raiz primitiva módulo n está apresentada na Definição 4.

Definição 4. Um número a é chamado de raiz primitiva módulo n se $n-1$ é o menor inteiro positivo tal que $a^{n-1} \equiv 1 \pmod{n}$ (IRELAND; ROSEN, 1982, p. 41).

Uma propriedade importante das raízes primitivas é que elas geram um grupo multiplicativo de inteiros módulo n, como apresentada no Teorema 4.

Teorema 4. Seja MDC[5]$(a, n) = 1$ e $a_1, a_2, ..., a_{\phi(n)}$[6] um conjunto de inteiros positivos menores que n e relativamente primos a n. Se a é uma raiz primitiva de n, então

$$a, a^2, ..., a^{\phi(n)} \tag{2.22}$$

são congruentes módulo n a $a_1, a_2, ..., a_{\phi(n)}$ em alguma ordem (BURTON, 1976, p. 150).

Um caso especial do Teorema 4 é quando n é primo, o que implica $\phi(n) = n - 1$. Escolhendo-se a sequência $\{1, 2, ..., n-1\}$, as potências módulo n da raiz primitiva de n irão gerar todos os elementos dessa sequência. Por exemplo, 2 é uma raiz primitiva de 13, então todos os elementos gerados por $2^i \pmod{13}$ serão distintos para $1 \leq i \leq 12$ (COHEN, 1993):

$$\{2^1, 2^2, 2^3, 2^4, 2^5, 2^6, 2^7, 2^8, 2^9, 2^{10}, 2^{11}, 2^{12}\} \equiv \{2, 4, 8, 3, 6, 12, 11, 9, 5, 10, 7, 1\} \pmod{13}$$

Já o número 3 não é uma raiz primitiva de 13, então haverá pelo menos dois elementos iguais gerados por $3^i \pmod{13}$ para $1 \leq i \leq 12$:

$$\{3^1, 3^2, 3^3, 3^4, 3^5, 3^6, 3^7, 3^8, 3^9, 3^{10}, 3^{11}, 3^{12}\} \equiv \{3, 9, 1, 3, 9, 1, 3, 9, 1, 3, 9, 1\} \pmod{13}$$

[5] $MDC(a, b)$ é o máximo divisor comum de a e b.

[6] $\phi(n)$ é a função totiente de Euler e conta a quantidade de inteiros positivos menos que n são relativamente primos a n.

2.6.3 Transformada Numérica Teórica (NTT)

A DFT pode ser generalizada conforme apresentada na Definição 5.

Definição 5. Seja R um anel, $n \geq 2$ e ω_n a principal raiz n-ésima da unidade. A Transformada de Fourier Discreta do vetor $a = [a_0, a_1, ..., a_{n-1}]$ de n elementos do anel R é o vetor $A = [A_0, A_1, ..., A_{n-1}]$ tal que (BRENT; ZIMMERMANN, 2010):

$$A_k = \sum_{j=0}^{n-1} \omega_n^{kj} a_j \tag{2.23}$$

As mesmas propriedades descritas na Seção 2.5.2.2 são mantidas, como a transformada inversa e o teorema da convolução.

Uma Transformada Numérica Teórica ou *Number Theoretic Transform* (NTT) é uma Transformada de Fourier Discreta escolhendo-se o corpo finito $\mathbb{Z}/p\mathbb{Z}$ (os números inteiros módulos a um primo p) (ver Seção 2.6.1) (SMITH, 2007). Na Seção 2.5.2.2, foi utilizado $\omega_n = e^{-2\pi i/n}$ para a raiz primitiva complexa e satisfaz $\omega_n^n = 1$ por ser uma raiz da unidade. Além disso, ω_n^m gera todas as raízes n-ésimas complexas da unidade para $0 \leq m \leq n-1$ (ver Teorema 1). Da mesma forma, se encontrarmos um número primo da forma $p = cn + 1$ e definirmos $\omega_p = r^c \pmod{n}$ em que r é uma raiz primitiva de p, nota-se que as mesmas propriedades são satisfeitas:

- $\omega_p^n = r^{cn} = r^{p-1} \equiv 1 \pmod{p}$ (Pequeno Teorema de Fermat)
- do Teorema 4, $\omega_p^m \equiv r^{cm} \pmod{p}$ gera todos os pontos de $\mathbb{Z}/p\mathbb{Z}$ em alguma ordem para $0 \leq m \leq n-1$.

É possível construir uma "Transformada Rápida Numérica Teórica" (FNTT) a partir da NTT da mesma forma que a FFT é construída a partir da DFT. O Algoritmo 5 apresenta a versão modificada do Algoritmo 4, ou seja, apresenta o algoritmo que calcula a NTT com as mesmas técnicas que a FFT, porém com a escolha de um anel diferente.

Algoritmo 5: Algoritmo da Transformada Rápida Numérica Teórica

Entrada: O vetor $a = [a_0, a_1, ..., a_{n-1}]$, um primo $P = 2^b c + 1$ e sua raiz primitiva r

Saídas : O vetor $A = NTT(a) = [A_0, A_1, ..., A_{n-1}]$

Function FNTT (a, P, r) **is**
  $n \leftarrow sizeof(a)$; ▷ n é uma potência de 2
  if $n = 1$ **then**
    return a
  end
  $a^{\text{par}}[\,] \leftarrow (a_0, a_2, ..., a_{n-2}) \pmod P$
  $a^{\text{impar}}[\,] \leftarrow (a_1, a_3, ..., a_{n-1}) \pmod P$
  $A^{\text{par}}[\,] \leftarrow \text{NTT}(a^{\text{par}})$
  $A^{\text{impar}}[\,] \leftarrow \text{NTT}(a^{\text{impar}})$
  $c = (P-1)/n$
  $\omega = 1$
  for $k \leftarrow 0$ **to** $n/2 - 1$ **do**
    $A_k \leftarrow A_k^{\text{par}} + \omega \cdot A_k^{\text{impar}} \pmod P$
    $A_{k+n/2} \leftarrow A_k^{\text{par}} - \omega \cdot A_k^{\text{impar}} \pmod P$
    $\omega \leftarrow \omega \cdot r^c \pmod P$
  end
  return A
end

Exemplo 4. Vamos calcular a NTT do vetor $a = [1, 2, 3, 0, 0, 0, 0, 0]$ utilizando a FNTT com um número primo $p = 641 = 2^7 \cdot 5 + 1$ e sua raiz primitiva $r = 3$. O vetor é dividido da mesma forma da Figura 5. A seguir, calcula-se a FNTT dos vetores dos níveis da árvore da Figura 5 utilizando o Algoritmo 5:

1. Os vetores h, i, j, k, l, m, n e o possuem apenas um elemento, logo representam o caso base do Algoritmo 5, já que a FNTT de um vetor de apenas um elemento é ele mesmo.

2. Os vetores d, e, f e g possuem 2 elementos. Logo, como apresentado na técnica de combinação do Algoritmo 5, a FNTT é calculada como sendo:

   - Vetor d:

     $$D_0 = H_0 + \omega_2^0 \cdot I_0 = 1 + 1 \cdot 0 \equiv 1 \pmod{641}$$
     $$D_1 = H_0 - \omega_2^0 \cdot I_0 = 1 - 1 \cdot 0 \equiv 1 \pmod{641}$$

     Então $D = FNTT(d) = [1, 1]$.

- Vetor e:

  $$E_0 = J_0 + \omega_2^0 \cdot K_0 = 3 + 1 \cdot 0 \equiv 3 \pmod{641}$$
  $$E_1 = J_0 - \omega_2^0 \cdot K_0 = 3 - 1 \cdot 0 \equiv 3 \pmod{641}$$

  Então $E = FNTT(e) = [3, 3]$.

- Vetor f:

  $$F_0 = L_0 + \omega_2^0 \cdot M_0 = 2 + 1 \cdot 0 \equiv 2 \pmod{641}$$
  $$F_1 = L_0 - \omega_2^0 \cdot M_0 = 2 - 1 \cdot 0 \equiv 2 \pmod{641}$$

  Então $F = FNTT(f) = [2, 2]$.

- Vetor g:

  $$G_0 = N_0 + \omega_2^0 \cdot O_0 = 0 + 1 \cdot 0 \equiv 0 \pmod{641}$$
  $$G_1 = N_0 - \omega_2^0 \cdot O_0 = 0 - 1 \cdot 0 \equiv 0 \pmod{641}$$

  Então $G = FNTT(g) = [0, 0]$.

3. Os vetores b e c possuem 4 elementos. Logo, a combinação é realizada da seguinte forma:

   - Cálculo das raízes da unidade:

     $$c = (P - 1)/n = (641 - 1)/4 = 160$$
     $$\omega^0 \equiv 1 \pmod{641}$$
     $$\omega^1 = \omega^0 \cdot r^c \pmod{P} \equiv 487 \pmod{641}$$

   - Vetor b:

     $$B_0 = D_0 + \omega_4^0 \cdot E_0 = 1 + 1 \cdot 3 \equiv 4 \pmod{641}$$
     $$B_2 = D_0 - \omega_4^0 \cdot E_0 = 1 - 1 \cdot 3 \equiv 639 \pmod{641}$$
     $$B_1 = D_1 + \omega_4^1 \cdot E_1 = 1 + 487 \cdot 3 \equiv 180 \pmod{641}$$
     $$B_3 = D_1 - \omega_4^1 \cdot E_1 = 1 - 487 \cdot 3 \equiv 463 \pmod{641}$$

     Então $B = FNTT(b) = [4, 180, 639, 463]$.

   - Vetor c:

     $$C_0 = F_0 + \omega_4^0 \cdot G_0 = 2 + 1 \cdot 0 \equiv 2 \pmod{641}$$
     $$C_2 = F_0 - \omega_4^0 \cdot G_0 = 2 - 1 \cdot 0 \equiv 2 \pmod{641}$$
     $$C_1 = F_1 + \omega_4^1 \cdot G_1 = 2 + 487 \cdot 0 \equiv 2 \pmod{641}$$
     $$C_3 = F_1 - \omega_4^1 \cdot G_1 = 2 - 487 \cdot 0 \equiv 2 \pmod{641}$$

     Então $C = FNTT(c) = [2, 2, 2, 2]$.

4. Por fim, o vetor A possui 8 elementos. Logo, a combinação é realizada da seguinte forma:

- Cálculo das raízes da unidade:

$$c = (P-1)/n = (641-1)/8 = 80$$
$$\omega^0 \equiv 1 \pmod{641}$$
$$\omega^1 = \omega^0 \cdot r^c \pmod{P} \equiv 318 \pmod{641}$$
$$\omega^2 = \omega^1 \cdot r^c \pmod{P} \equiv 487 \pmod{641}$$
$$\omega^3 = \omega^2 \cdot r^c \pmod{P} \equiv 385 \pmod{641}$$

- Vetor a:

$$A_0 = B_0 + \omega_8^0 \cdot C_0 = 4 + 1 \cdot 2 \equiv 6 \pmod{641}$$
$$A_4 = B_0 - \omega_8^0 \cdot C_0 = 4 - 1 \cdot 2 \equiv 2 \pmod{641}$$
$$A_1 = B_1 + \omega_8^1 \cdot C_1 = 180 + 318 \cdot 2 \equiv 175 \pmod{641}$$
$$A_5 = B_1 - \omega_8^1 \cdot C_1 = 180 - 318 \cdot 2 \equiv 185 \pmod{641}$$
$$A_2 = B_2 + \omega_8^2 \cdot C_2 = 639 + 487 \cdot 2 \equiv 331 \pmod{641}$$
$$A_6 = B_2 - \omega_8^2 \cdot C_2 = 639 - 487 \cdot 2 \equiv 306 \pmod{641}$$
$$A_3 = B_3 + \omega_8^3 \cdot C_3 = 463 + 385 \cdot 2 \equiv 592 \pmod{641}$$
$$A_7 = B_3 - \omega_8^3 \cdot C_3 = 463 + 385 \cdot 2 \equiv 334 \pmod{641}$$

Então $A = FNTT(a) = [6, 175, 331, 592, 2, 185, 306, 334]$.

O resultado encontrado pode ser conferido calculando-se a NTT de v pela Definição 5:

$$A_0 = \sum_{j=0}^{7} a_j (r^c)^{0 \cdot j} \equiv 6 \pmod{641} \qquad A_4 = \sum_{j=0}^{7} a_j (r^c)^{4 \cdot j} \equiv 2 \pmod{641}$$
$$A_1 = \sum_{j=0}^{7} a_j (r^c)^{1 \cdot j} \equiv 175 \pmod{641} \qquad A_5 = \sum_{j=0}^{7} a_j (r^c)^{5 \cdot j} \equiv 185 \pmod{641}$$
$$A_2 = \sum_{j=0}^{7} a_j (r^c)^{2 \cdot j} \equiv 331 \pmod{641} \qquad A_6 = \sum_{j=0}^{7} a_j (r^c)^{6 \cdot j} \equiv 306 \pmod{641}$$
$$A_3 = \sum_{j=0}^{7} a_j (r^c)^{3 \cdot j} \equiv 592 \pmod{641} \qquad A_7 = \sum_{j=0}^{7} a_j (r^c)^{7 \cdot j} \equiv 334 \pmod{641}$$

2.6.4 Algoritmo de multiplicação de números inteiros utilizando NTT

O procedimento realizado para multiplicar números inteiros com a NTT é o mesmo do mostrado na Seção 2.5.4. É importante analisar a escolha do primo p a ser utilizado na NTT. Nota-se que, por ser um algoritmo de dividir e conquistar, convém a quantidade de elementos da entrada ser uma potência de 2. Logo, para a linha 10 do Algoritmo 5 ser sempre inteira, convém $p-1$ ser também uma potência de 2. Além disso, foi analisado na Seção 2.6.3 que p deve ser da forma $p = cn + 1$. Então uma escolha apropriada para

o primo P deve ser da forma $P = c2^n + 1$. Isso faz com que o algoritmo da NTT para multiplicação de inteiros funcione corretamente com números de até 2^n algarismos.

Exemplo 5. Considere novamente o exemplo 123×456. A seguir, os passos utilizados para calcular esta multiplicação utilizando a NTT:

1. Queremos encontrar o resultado da convolução dos vetores $a = [1, 2, 3]$ e $b = [4, 5, 6]$. O tamanho dos vetores deve ser uma potência de 2 maior que $n+m-1 = 3+3-1 = 5$. A próxima potência de 2 maior que 5 é 8. Logo, os novos vetores a serem analisados são:

$$\begin{aligned} a &= [1, 2, 3, 0, 0, 0, 0, 0] \\ b &= [4, 5, 6, 0, 0, 0, 0, 0] \end{aligned} \tag{2.24}$$

2. Calcular $A = NTT(a)$ e $B = NTT(b)$ com FNTT. O vetor A já foi calculado no Exemplo 4. O vetor B pode ser calculado de forma análoga:

$$\begin{aligned} A &= [6, 175, 331, 592, 2, 185, 306, 334] \\ B &= [15, 29, 510, 289, 5, 54, 127, 285] \end{aligned}$$

3. Calcular o produto C termo a termo de A e B módulo P:

$$\begin{aligned} c_0 &= a_0 \cdot b_0 \equiv 90 \pmod{641} \\ c_1 &= a_1 \cdot b_1 \equiv 588 \pmod{641} \\ c_2 &= a_2 \cdot b_2 \equiv 227 \pmod{641} \\ c_3 &= a_3 \cdot b_3 \equiv 582 \pmod{641} \\ c_4 &= a_4 \cdot b_4 \equiv 10 \pmod{641} \\ c_5 &= a_5 \cdot b_5 \equiv 375 \pmod{641} \\ c_6 &= a_6 \cdot b_6 \equiv 402 \pmod{641} \\ c_7 &= a_7 \cdot b_7 \equiv 322 \pmod{641} \end{aligned}$$

4. Por fim, calcula-se $a * b$ utilizando a FNTT inversa de C:

$$a * b = IDFT(A \cdot B) = IDFT(C) = [4, 13, 28, 27, 18, 0, 0, 0]$$

Observa-se que é o mesmo resultado do encontrado na Figura 2. Realizando a propagação de *carrys*, encontra-se o resultado $123 \times 456 = 56088$.

Como visto na Seção 2.6.3, a FFT pode resultar em erros numéricos decorrente da aritmética de ponto flutuante. Com o FNTT, não há erro de precisão, pois todas as operações são feitas com números inteiros (SMITH, 2007). Adicionalmente, o espaço requisitado na implementação da FNTT é menor, então multiplicar números grandes num sistema de memória limitada pode ser ideal com FNTT (TOMMILA, 1997).

A Transformada Numérica Teórica foi pela primeira vez implementada no algoritmo de Schönhage–Strassen, o qual possui complexidade $O(n \log n \log\log n)$ (HASTAD, 2000, p. 79). Schönhage e Strassen conjecturaram que o algoritmo de multiplicação mais rápido possível possui complexidade $O(n \log n)$. O algoritmo criado por eles permaneceu como o mais rápido assintoticamente por mais de 35 anos, até que Fürer publicou um algoritmo em 2007 ainda mais eficiente, baseado na mesma ideia do NTT. O algoritmo de Fürer possui complexidade $O(n \log n \ 2^{O(\log^* n)})$, o que representou um passo significativo em se aproximar de $O(n \log n)$ (FURER, 2007).

3 Metodologia

Neste capítulo, será explicado a metodologia seguida para a realização deste trabalho. A Seção 3.1 descreve como foi organizado o fluxo do trabalho. A Seção 3.2 explica como as fontes bibliográficas e referências para a escrita da Fundamentação Teórica e para a implementação dos algoritmos foram buscadas. A Seção 3.3 apresenta as ferramentas utilizadas para a escrita do trabalho e implementação dos algoritmos. A Seção 3.4 mostra os métodos utilizados para implementar e otimizar os algoritmos. A Seção 3.5 explica qual estratégia foi seguida para analisar a eficiência dos algoritmos implementados. Finalmente, a Seção 3.6 apresenta os *hardwares* utilizados para analisar os algoritmos e o porquê de suas escolhas.

3.1 Fluxo do Trabalho

O andamento do trabalho foi montado a partir de reuniões com o orientador, os quais foram realizados uma vez por semana por cerca de uma hora. Os principais pontos discutidos nas reuniões eram como implementar os algoritmos de forma eficiente e como o trabalho escrito seria organizado.

O trabalho teve início primeiramente na busca de artigos científicos no Portal de Periódicos Capes. A busca de artigos sobre algoritmos de multiplicação revelou uma ideia de como está o andamento da pesquisa nessa área e quais eram os algoritmos mais utilizados nos *softwares* atuais. Após o estudo e análise dos artigos, implementou-se o algoritmo tradicional de escola, o algoritmo da Transformada Rápida de Fourier e o algoritmo da Transformada Numérica Teórica na linguagem de programação C++. Após a validação dos testes, a escrita do trabalho foi iniciada. Posteriormente, os algoritmos foram otimizados e implementados nas placas de desenvolvimento apresentadas na Seção 3.6.

3.2 Fontes Bibliográficas

As principais fontes bibliográficas buscadas estão apresentadas nos itens a seguir:

- Portal de Periódicos Capes[1] (Coordenação de Aperfeiçoamento de Pessoal de Nível Superior): fundada pelo MEC, a Capes desempenha papel fundamental na expansão e consolidação do ensino superior em todos os estados do Brasil. Instituições cadastradas (como a Universidade de Brasília) possuem acesso a uma vasta biblioteca de artigos e materiais de produção científica.

[1] http://www.periodicos.capes.gov.br/

- Bibliotecas[2]: livros e materiais foram pesquisados, em sua maior parte, na Biblioteca da Faculdade do Gama, que é voltada para as áreas de Engenharia e possui cerca de 4.800 exemplares, entre livros, normas técnicas e CD ROMs. Outra biblioteca consultada foi a Biblioteca Central da UnB, localizada no campus Darcy Ribeiro.

- Google Acadêmico[3]: o Google Acadêmico (ou *Google Scholar*) é uma ferramenta de pesquisa gratuita criada pelo Google e permite pesquisar jornais acadêmicos, livros, teses, dissertações, papéis de conferências e artigos variados.

- Livros da Editora Vestseller[4]: a Editora VestSeller é uma editora brasileira que tem como foco distribuir livros na área de ciências exatas (como Matemática, Física e Química). Possui diversos materiais de preparação para os vestibulares de engenharia mais díficeis do Brasil (como ITA, IME, AFA).

- *websites:* dois websites que contiveram informações relevantes e que foram citados ao longo deste trabalho:

  i) *Apfloat arbitrary precision library (TOMMILA, 1997)*: *Apfloat* é uma biblioteca de precisão aritmética arbitrária para C++ e seu *website* possui explicações de seu funcionamento, como a Transformada Numérica Teórica, o que auxiliou no entendimento do assunto e na implementação do algoritmo.

  ii) *MacTutor History of Mathematics archive* (O'CONNOR; ROBERTSON, 1997): criado por John J. O'Connor e Edmund F. Robertson pela Universidade da Escócia, apresenta vários tópicos sobre a história da matemática.

3.3 Ferramentas de Trabalho

O computador utilizado para a realização de todo o trabalho foi um notebook do modelo G1511 da marca Avell, o qual é equipado com um processador Intel i7-6700HQ de 2.60GHz e 16 GB de memória RAM. Os seguintes *softwares* foram utilizados para a implementação dos algoritmos, organização e escrita do trabalho:

- sistema operacional: o sistema operacional utilizado foi o Ubuntu Linux (versão 17.04) com ambiente de trabalho Gnome. O Linux foi escolhido porque oferece diversas ferramentas de desenvolvimento de software, o que facilitou a realização do trabalho;

- ambiente de desenvolvimento integrado (IDE): os textos e algoritmos foram escritos no editor de texto *Atom* (versão 1.26.1), o qual oferece diversos *plugins* para muitas

[2] https://www.bce.unb.br/
[3] https://scholar.google.com.br/
[4] http://www.vestseller.com.br/

linguagens de programação. Além disso, possui uma interface gráfica customizável e estável;

- linguagens de programação: as seguintes linguagens de programação foram utilizadas no desenvolvimento do trabalho:

  i) escrita da monografia: a produção de texto do trabalho foi escrito em linguagem *Latex*, o qual é uma linguagem de diagramação de texto de alta qualidade e facilita a escrita de expressões matemáticas complexas (KNUTH, 1986);

  ii) implementação dos algoritmos: os algoritmos de multiplicação de números inteiros foram escritos em linguagem de programação C++ (STROUSTRUP, 2013), conhecida por ser rápida e possuir estruturas de dados úteis já embutidas nas bibliotecas padrão. A compilação dos programas foi realizada com o compilador *GNU GCC* (versão 7.2.0);

  iii) comparação de performances: para comparar a performance dos algoritmos de multiplicação para números de precisão arbitrária, escolheu-se a biblioteca GMP[5] (versão 6.1.1) (*GNU Multiple Precision Arithmetic Library*) para a linguagem C++, a biblioteca *BigInteger* para a linguagem Java[6] (com compilador *javac* versão 1.8.0_151) e a linguagem interpretada Pytho[7] (versão 3.6.3) que possui precisão de números inteiros arbitrária por padrão;

  iv) automação dos comandos: para compilar, executar, medir o tempo de execução, organizar os arquivos e analisar as saídas, foi utilizada a linguagem de programação de *shell script* (versão 4.4.12) com o interpretador de comandos *bash*;

  v) programação das placas de desenvolvimento: foi utilizado o programa *PlatformIO*[8], o qual é um *software* livre destinado à facilitar o desenvolvimento, compilação e programação de sistemas embarcados.

- Armazenamento de arquivos: o trabalho escrito e os algoritmos foram armazenados no *Gitlab*[9], o qual é um repositório de *software* gratuito. Com isso, foi possível adicionar o orientador como colaborador, o que facilitou o compartilhamento de arquivos.

[5] https://gmplib.org/
[6] https://java.com/
[7] https://www.python.org/
[8] https://platformio.org/
[9] https://gitlab.com/

3.4 Implementação dos algoritmos

Para este trabalho, foi decidido trabalhar com o Algoritmo Tradicional da Escola, Algoritmo de Multiplicação com FFT e com NTT. As subseções a seguir apresentam como cada um dos 3 algoritmos foram implementados e otimizados.

3.4.1 Algoritmo Tradicional da Escola

O algoritmo tradicional ensinado em escola foi implementado seguindo o Algoritmo 3 conforme a definição de convolução (Definição 2). Sua implementação em C++ está apresentada no Apêndice A. O algoritmo já está na sua forma mais simples e nenhuma otimização pode ser aplicada neste caso.

3.4.2 Algoritmo de Multiplicação com FFT

Primeiramente, a FFT foi implementada seguindo o Algoritmo 4. Esta implementação é lenta, já que usa uma estratégia recursiva para calcular os termos da DFT e logo uma pilha de recursão vai sendo formada. Além disso, o retorno das funções recursivas é um vetor, o que faz com que vários elementos sejam copiados a cada retorno da função. Sua implementação em C++ está apresentada no Apêndice B.

Uma forma de otimizar o algoritmo é utilizando o método iterativo *in-place*[10]. Dessa forma, as variáveis podem ser alocadas estaticamente e não há a criação de pilhas de recursão. Isso é fundamental para implementar o algoritmo nas placas de desenvolvimento, já que possuem memória limitada.

A primeira implementação iterativa fazia uso da estrutura padrão `vector` do C++. Uma outra versão foi feita substituindo o `vector` por um vetor estático. O algoritmo da FFT iterativa com vetores estáticos em C++ está apresentada no Apêndice C.

Por fim, as implementações anteriores faziam uso da biblioteca `complex` do C++, que é uma classe que guarda as partes reais e complexas dos números e realiza operações básicas com eles (como soma e multiplicação). Uma última tentativa de otimizar o código foi abandonar o `complex` e utilizar apenas dois vetores estáticos de pontos flutuantes que representam as partes reais e complexas. Algumas funções foram escritas para realizar operações básicas entre eles.

3.4.3 Algoritmo de Multiplicação com NTT

Primeiramente, a NTT faz uso de uma função auxiliar que calcula $a^b \pmod{P}$ (linha 13 do Algoritmo 5). Como b pode ser grande, convém utilizar o algoritmo da

[10] *in-place* significa que as operações são realizadas dentro do próprio vetor de entrada, não necessitando de espaço auxiliar adicional

Exponenciação Rápida, que permite o cálculo de $a^b \pmod P$ em $O(\log b)$. A técnica consiste em calcular o quadrado de a repetidas vezes para formar o resultado a^b, por meio da representação binária de b.

A primeira implementação da NTT foi utilizando o Algoritmo 5. Como explicado na Subseção 3.4.2, esta implementação é lenta, já que usa o método recursivo. Sua implementação em C++ está apresentada no Apêndice D.

A segunda forma utilizada foi o método iterativo *in-place*, porém com a estrutura `vector` do C++. Por fim, retirou-se o `vector` e substituiu-se *arrays* estáticos em seu lugar. Esta implementação está no Apêndice E.

3.5 Análise dos Algoritmos

Foram realizadas dois tipos de análise nos algoritmos implementados: performance (Seção 3.5.1) e memória (Seção 3.5.2).

3.5.1 Análise de Performance

Foram implementados 3 algoritmos de multiplicação de números inteiros para estudar suas eficiências: o algoritmo da escola tradicional, o algoritmo de multiplicação utilizando a Transformada Rápida de Fourier e o algoritmo de multiplicação utilizando a Transformada Numérica Teórica. Além disso, utilizou-se a biblioteca GMP do C++, a biblioteca *BigInteger* do Java e a biblioteca padrão do Python para comparar com os tempos de execução dos algoritmos implementados. A estratégia para testar as eficiências dos códigos foi gerar diversos casos de teste e medir o tempo de execução real de cada algoritmo. Cada caso de teste possui dois números a serem multiplicados. Para isso, os códigos foram submetidos a dois tipos de entradas de tamanho n:

- Entrada de números ordenados: o primeiro tipo de entrada analisado foi gerar todas as combinações possíveis de pares ordenados (x, y) tais que $0 \leq x, y \leq n$. Por exemplo, se $n = 2$, as entradas testadas foram $[(0,0), (0,1), (0,2), (1,0), (1,1), (1,2), (2,0), (2,1), (2,2)]$. Isso permite avaliar casos que podem gerar conflitos (i.e. multiplicação com 0 e *corner cases*) e avaliar a performance dos algoritmos para diversas entradas.

- Entrada de números aleatórios: o segundo tipo de entrada foi gerar números aleatórios de n dígitos na base decimal. Foram gerados exatamente 10 pares de números com n algarismos para cada caso de teste.

Após escrever os códigos que geram os casos de teste em C++, variou-se o parâmetro n em potências de 2 devido à natureza logarítmica das complexidades dos algoritmos implementados. Os valores de n utilizados para a entrada de números ordenados no com-

putador e no Raspberry Pi 3 foram $n = \{2^i\}_{i=1}^{10} = [2, 4, 8, 16, 32, 64, 128, 256, 512, 1024]$. Para as entradas de números aleatórios, os valores de n utilizados no computador e no Raspberry Pi 3 foram $n = \{2^i\}_{i=1}^{15} = [2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32768]$.

Após registrado o resultado da multiplicação para cada um dos casos de teste e o tempo de execução de cada código, foi implementado uma rotina de verificação, o qual consistiu em comparar todas as saídas, pois como todos os algoritmos computam o mesmo procedimento, todos os resultados devem ser idênticos. Uma vez validado os resultados, montou-se um gráfico com os tempos de execução para os dois tipos de entrada. O gráfico foi gerado em Python utilizando-se a biblioteca *matplotlib*[11].

3.5.2 Análise de Memória

É necessário conhecer a memória gasta pelos algoritmos implementados, principalmente ao realizar testes em *hardwares* com pouca memória disponível. Convém estudar a memória gasta pelos vetores utilizados, ou seja, a memória contígua alocada na memória RAM, já que estes são os que mais impactam na memória final usada. Sejam dois números a e b com n e m dígitos cada, respectivamente. A seguir, apresenta-se uma análise para calcular a memória gasta nos 3 algoritmos:

- **Tradicional:** Usa-se $n + m$ *bytes* para guardar os dígitos dos números a e b. Além disso, é necessário utilizar um vetor auxiliar do tipo inteiro de $n + m - 1$ posições, totalizando $4(n+m-1)$ *bytes* para guardar o resultado. Observa-se que não é possível aproveitar os próprios vetores a e b para guadar o resultado, pois é necessário acessar constantemente os dígitos de a e b para realizar as operações. Logo, são necessários, no total, $n + m + 4n + 4m - 4 = 5n + 5m - 4$ *bytes*.

- **FFT:** Uma das vantagens de utilizar a FFT Iterativa é que as operações podem ser feitas $in - place$, ou seja, o resultado do algoritmo pode ser guardado no próprio vetor de entrada. Porém, para a FFT funcionar, é necessário o vetor de entrada possuir pelo menos a próxima potência de 2 da soma dos dígitos de a e b, ou seja, são necessários 2 vetores de $2^{\lceil \log_2 (n+m) \rceil}$ posições cada, totalizando $2^{\lceil \log_2 (n+m) \rceil + 1}$ *bytes*. Como cada vetor precisa guardar a parte imaginária e real do números complexo, é preciso guardar mais outra posição no vetor, totalizando $2^{\lceil \log_2 (n+m) \rceil + 2}$ posições. Finalmente, como o tipo de dados utilizado é `float`, a memória total será de $4 \cdot 2^{\lceil \log_2 (n+m) \rceil + 2} = 2^{\lceil \log_2 (n+m) \rceil + 4}$ *bytes*.

- **NTT:** A memória gasta pela NTT é similar à memória gasta pela FFT, mas toda as operações são realizadas com números inteiros, não necessitando guardar as partes

[11] https://matplotlib.org/

reais e imaginárias dos números. Logo, a memória gasta pela NTT é a metade da gasta pela FFT, totalizando $2^{\lceil \log_2 (n+m) \rceil + 3}$ *bytes*.

A Tabela 2 apresenta o resumo das memórias gastas pelos 3 algoritmos.

Tabela 2 – Memória gasta pelos 3 algoritmos de multiplicação implementados

Algoritmos	**Memória (*bytes*)**
Tradicional	$5n + 5m - 4$
FFT	$2^{\lceil \log_2 (n+m) \rceil + 4}$
NTT	$2^{\lceil \log_2 (n+m) \rceil + 3}$

Para saber a memória máxima que as placas de desenvolvimento conseguem alocar, foi utilizada o método de tentativa e erro para encontrar o maior n possível para a memória RAM de cada *hardware* (foi-se aumentando o tamanho da alocação até haver falha de segmentação).

3.6 Placas de Desenvolvimento

Para testar os algoritmos em *hardwares* diferentes, foram utilizadas as chamadas placas de desenvolvimento, que consistem em placas de circuito impresso com um processador embutido e ferramentas que facilitam o desenvolvimento de projetos embarcados para criação de protótipos, como reguladores de tensão e pinos eletrônicos de entrada/saída. A ideia é verificar o comportamento dos algoritmos implementados para diferentes processadores para avaliar sua performance em termos de tempo de execução e limites de memória.

Para isso, foram escolhidas 3 placas de desenvolvimento: Arduino UNO, NodeMCU e Raspberry Pi 3 modelo B, os quais são detalhadas a seguir:

1. Arduino UNO: placa de desenvolvimento popular que faz uso do microcontrolador ATmega328P. O ATmega328P possui arquitetura RISC de 8 bits, velocidade de *clock* máxima de 20 MHz, 32 kB de memória *flash*, 1 kB de memória EEPROM e 2 kB de memória RAM. Além disso, não possui um coprocessador aritmético capaz de realizar operações com números flutuantes (ATMEL, 2015).

2. NodeMCU: utiliza o microcontrolador ESP8266 da *Espressif Systems* e é muito utilizada em projetos na área Internet das Coisas (IOT). O ESP8266 possui arquitetura RISC de 32 bits, velocidade de *clock* máxima de 160 MHz, suporta 4 MB de memória EEPROM e cerca de 50 kB de memória RAM. Assim como o ATmega328P, também não possui um coprocessador aritmético (ESPRESSIF, 2018).

3. Raspberry Pi 3B: pequeno computador de baixo custo muito utilizado em projetos embarcados. O Raspberry Pi 3B possui arquitetura ARM, velocidade de *clock* máxima de 1.2 GHz, 1 GB de memória RAM e memória expansível via cartão SD. Além disso, possui um coprocessador capaz de realizar operações com números em ponto flutuante (HALFACREE; UPTON, 2012).

4 Resultados

Neste capítulo, serão apresentados os ganhos de performance das otimizações realizadas para melhorar o tempo de execução dos algoritmos da FFT e NTT (Seção 4.1) e a execução de todos os algoritmos nos *hardwares* (Seção 4.2).

4.1 Otimizações dos algoritmos da FFT e NTT

Conforme apresentado na Seção 3.4, o algoritmo da FFT foi implementada de 4 formas distintas. As Figuras 6 e 7 apresentam a diferença de performance das 4 implementações:

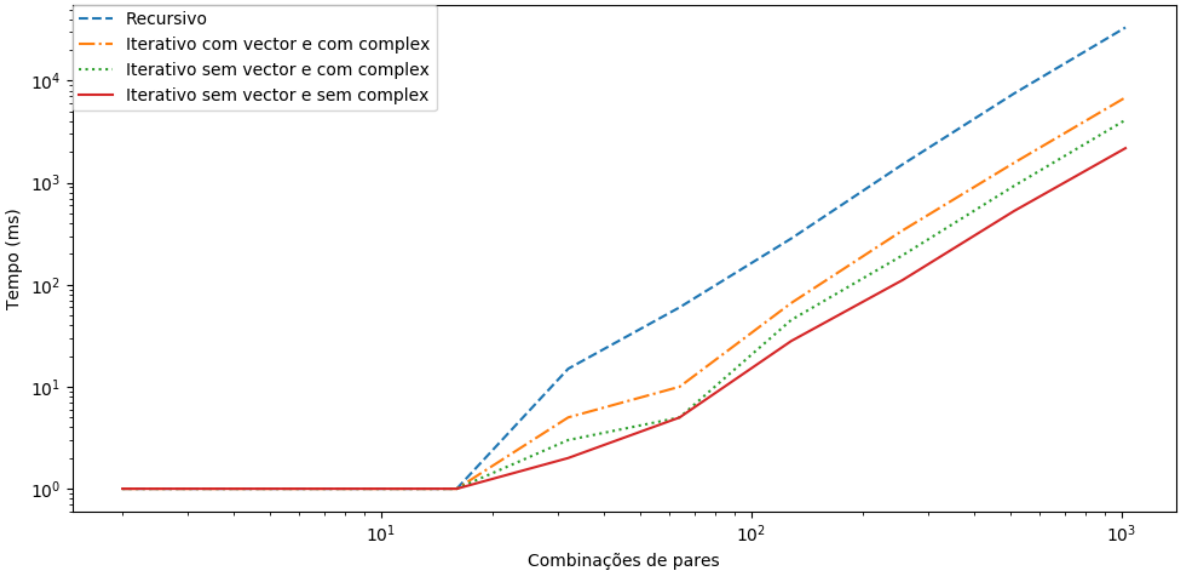

Figura 6 – Gráfico dos tempos de execução da FFT, em escala logarítmica, para todas as combinações de pares de números menores ou iguais a n

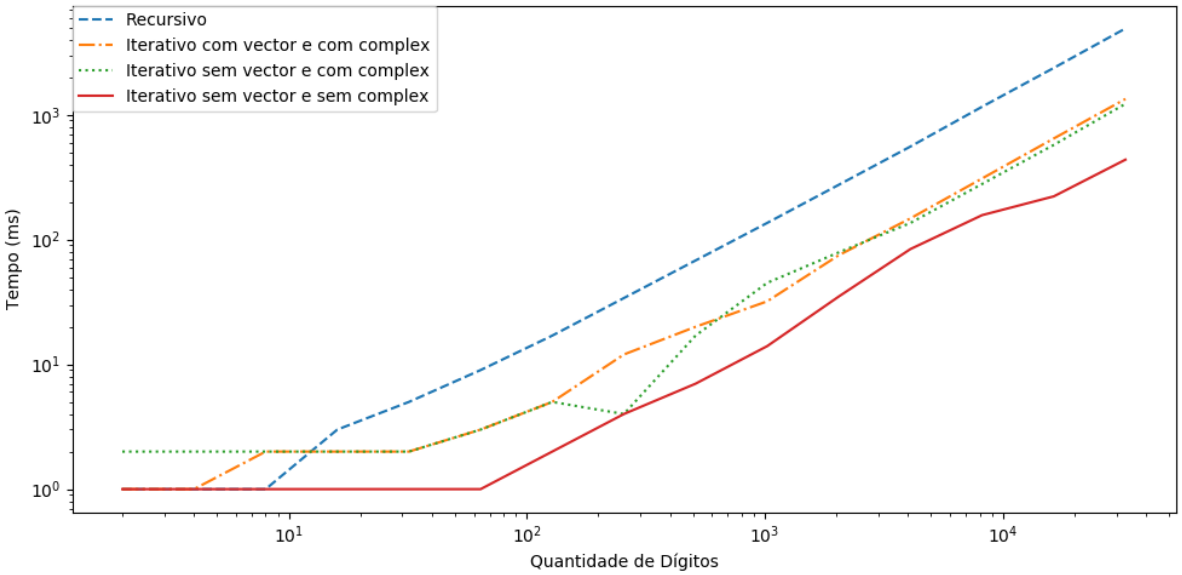

Figura 7 – Gráfico dos tempos de execução da FFT em escala logarítmica para quantidade de dígitos n

Já o algoritmo da NTT foi implementada de 3 formas distintas (ver Seção 3.4.3). As Figuras 8 e 9 apresentam a diferença de performance das 3 implementações:

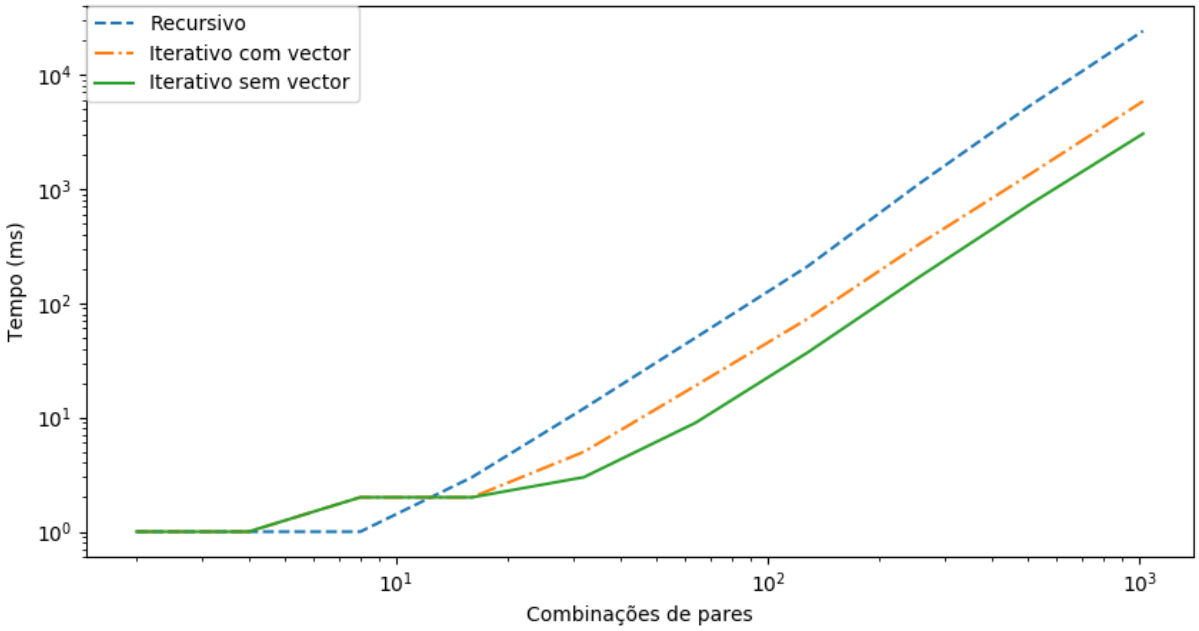

Figura 8 – Gráfico dos tempos de execução da NTT, em escala logarítmica, para todas as combinações de pares de números menores ou iguais a n

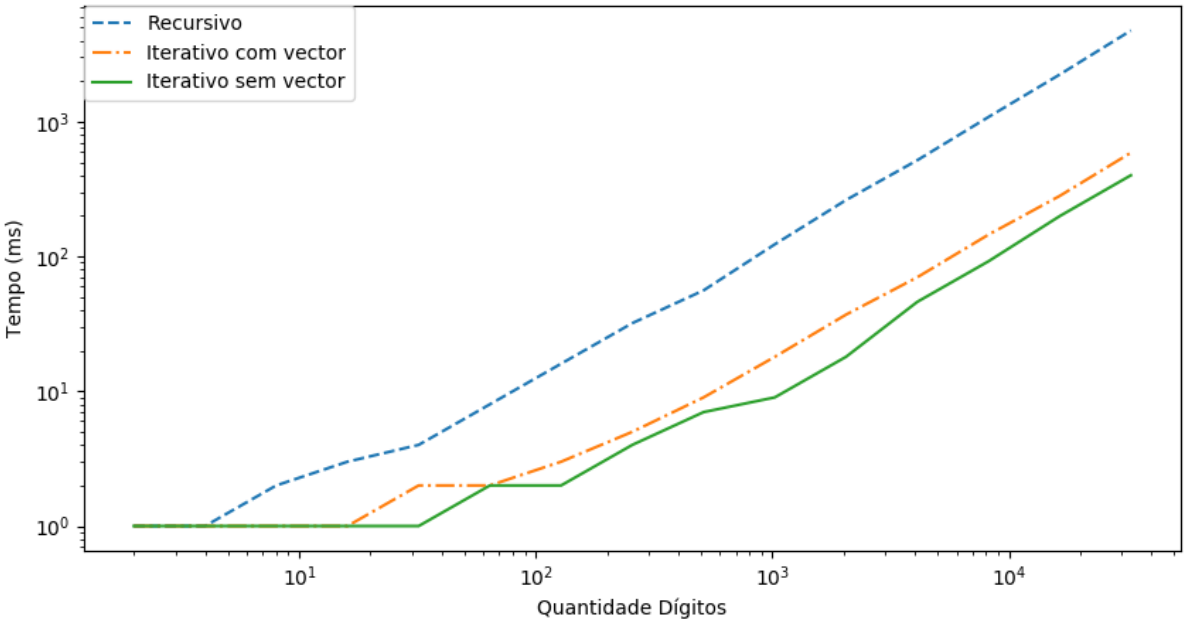

Figura 9 – Gráfico dos tempos de execução da NTT em escala logarítmica para quantidade de dígitos n

De forma esperada, é possível observar que:

- o método recursivo é o mais lento, pois faz uso de chamadas recursivas e, consequentemente, uma pilha de recursão é formada;
- ao passar para a forma iterativa *in-place*, há um ganho de performance e memória, já que se utiliza apenas um vetor e não há a criação de uma pilha de recursão;
- ao utilizar um vetor estático no lugar da estrutura padrão `vector` do C++, há uma melhora na performance, pois não há alocação dinâmica e as operações são realizadas diretamente em memória.

Essas otimizações foram fundamentais para a execução dos algoritmos em *hardwares* com pouca memória, como o Arduino Uno e o NodeMCU.

4.2 Análise dos algoritmos implementados nos *hardwares*

Esta Seção apresenta a análise da execução dos algoritmos implementados no Computador (Seção 4.2.1), no Arduino Uno (Seção 4.2.2), no NodeMCU (Seção 4.2.3) e no Raspberry Pi 3B (Seção 4.2.4). Para o Computador, os algoritmos foram testados números com entradas de números ordenados e entradas de números aleatórios. Já para o Arduino Uno, NodeMCU e Raspberry Pi 3, os algoritmos foram testados somente com entradas de números aleatórios, com o objetivo de avaliar a performance para números arbitrariamente grandes.

4.2.1 Computador

Como apresentado na Seção 3.5, os algoritmos foram submetidos a dois tipos de entradas: entradas de números ordenados e entradas de números aleatórios. Os algoritmos são classificados em algoritmos implementados (tradicional, FFT, NTT) e algoritmos de bibliotecas (GMP, Java, Python). A seguir, estão apresentadas duas tabelas (Tabelas 3 e 4) e dois gráficos (Figuras 10 e 11) dos tempos de execução para cada uma das entradas.

Tabela 3 – Tempo de execução dos algoritmos para entradas de pares ordenados de números menores ou iguais a n (ver Seção 3.5), em ms

n / **alg.**	**2**	**4**	**8**	**16**	**32**	**64**	**128**	**256**	**512**	**1024**
Tradicional	1	1	1	3	1	8	65	204	958	3839
FFT	1	1	1	1	2	6	21	107	534	2187
NTT	1	1	1	2	3	9	32	178	718	3033
GMP	0	1	1	2	2	3	4	21	83	337
Java	55	51	55	76	102	148	545	664	1082	2187
Python	19	18	23	24	23	39	79	287	987	3829

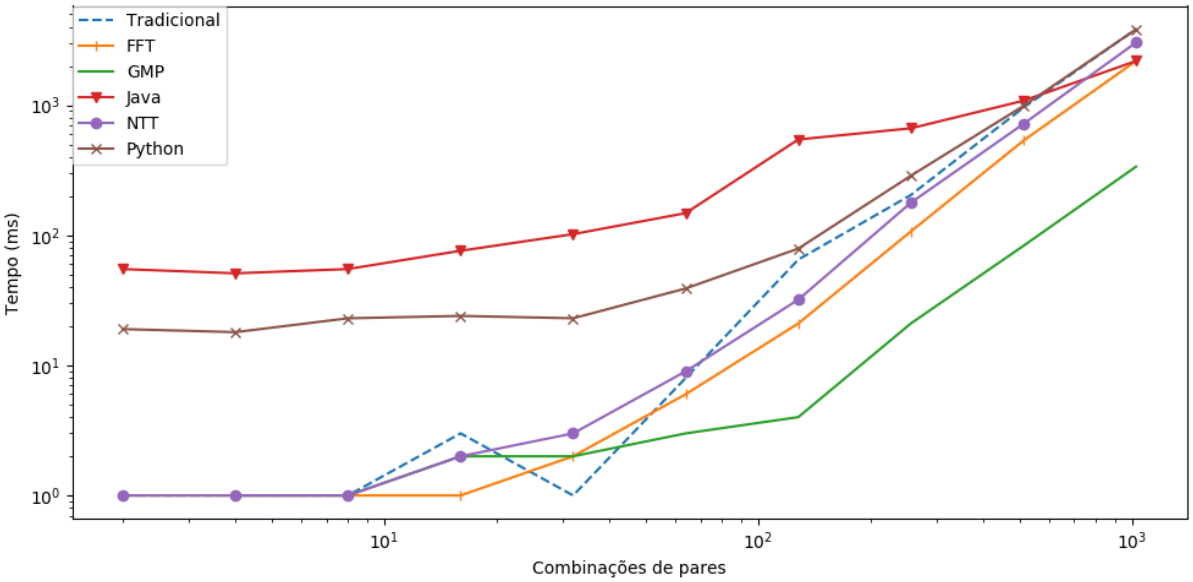

Figura 10 – Gráficos dos tempos de execução dos algoritmos em escala logarítmica para todas as combinações até n

Tabela 4 – Tempo de execução dos algoritmos para entradas de pares de números aleatórios de n dígitos (ver Seção 3.5), em ms

alg. \ n	**2**	**4**	**8**	**16**	**32**	**64**	**128**	**256**	**512**	**1024**	**2048**	**4096**	**8192**	**16384**	**32768**
Tradicional	1	1	1	1	1	2	3	7	22	81	312	1178	4692	19908	74207
FFT	2	2	2	2	2	2	3	1	7	14	27	51	108	225	445
NTT	1	1	1	2	1	2	2	4	6	12	25	46	97	191	402
GMP	0	2	1	2	1	1	1	1	2	2	2	3	6	14	34
Java	41	41	55	54	57	65	54	78	87	114	165	328	533	1268	2475
Python	16	19	20	18	22	17	22	19	22	23	21	27	70	201	734

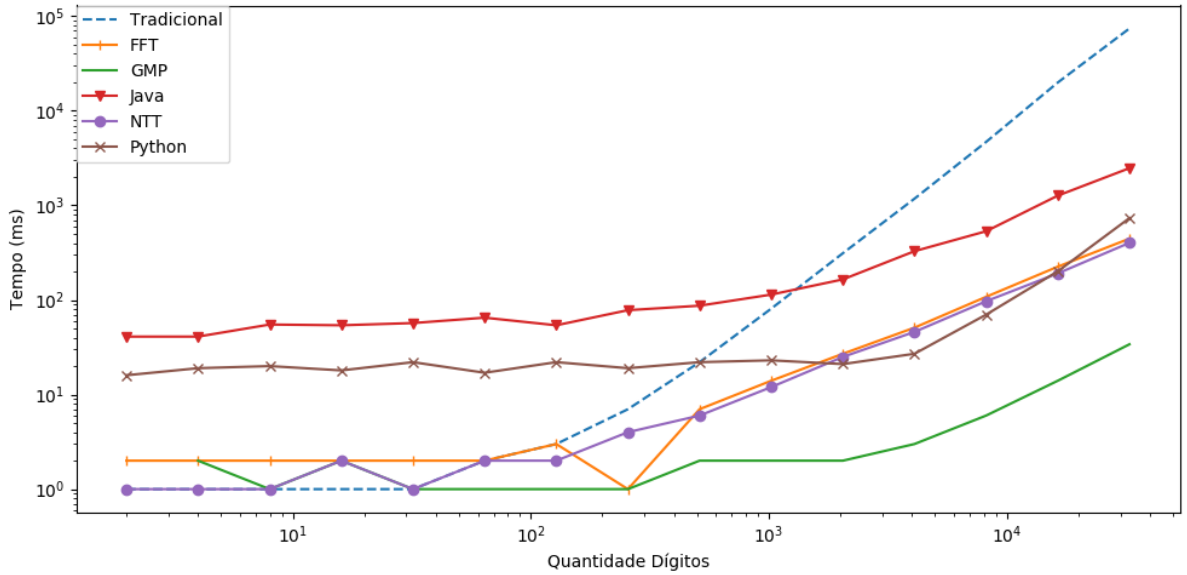

Figura 11 – Gráficos dos tempos de execução dos algoritmos em escala logarítmica o qual n é a quantidade de dígitos dos números que foram multiplicados

Como estudado na Seção 2.3, o algoritmo tradicional possui a pior complexidade assintótica conhecida para um algoritmo de multiplicação ($O(n^2)$). Porém, vale ressaltar que só a complexidade assintótica não é suficiente para comparar eficiência de algoritmos, já que a constante que acompanha $f(n)$ em $O(f(n))$ influencia seu comportamento para valores pequenos de n (ver Seção 2.2). O algoritmo tradicional para valores pequenos de n acaba sendo mais eficiente do que a FFT e a NTT, apesar de terem complexidades assintóticas melhores. Isso ocorre devido à simplicidade de sua implementação, que constitui apenas de dois loops realizando simples operações de somas e e multiplicações. Em contraste ao tradicional, a FFT e a NTT possuem etapas adicionais que impactam em suas performances, como preenchimento de zeros nos vetores de entrada, operações de resto de divisão e operações com pontos flutuantes.

Para entradas de números ordenados, os algoritmos possuíram performances similares, já que a quantidade de dígitos das entradas foram pequenas (máximo de 4 dígitos).

Para entradas de números aleatórios grandes, a FFT a NTT apresentaram tempos similares, com a NTT ligeiramente mais rápida que a FFT. Além disso, a FFT e a NTT possuíram tempos de execução melhores em relação ao algoritmo tradicional. Analisando o gráfico da Figura 10, nota-se que o desempenho da FFT e da NTT supera o do algoritmo tradicional para números de aproximadamente 150 dígitos.

Já os algoritmos com bibliotecas os algoritmos com bibliotecas, no geral o mais rápido foi o do GMP, seguido pelo o do Python e por fim o do Java. É interessante observar que os algoritmos da NTT e FFT implementados em C++ foram mais rápidos que o Java e passaram do Python no final do gráfico da Figura 10.

4.2.2 Arduino Uno

Foi possível alocar no máximo 2048 *bytes* de memória contígua no Arduino Uno. Logo, foram testados números com a seguinte quantidade de dígitos utilizando os resultados da Tabela 2 (considerando $m = n$):

- **Tradicional:** $5n + 5m - 4 = 2048 \Rightarrow n \approx 200$ dígitos
- **NTT:** $2^{\lceil \log_2 (n+m) \rceil + 4} = 2048 \Rightarrow n \approx 128$ dígitos
- **FFT:** $2^{\lceil \log_2 (n+m) \rceil + 3} = 2048 \Rightarrow n \approx 64$ dígitos

O gráfico da Figura 12 apresenta as diferentes performances dos 3 algoritmos implementados no Arduino Uno.

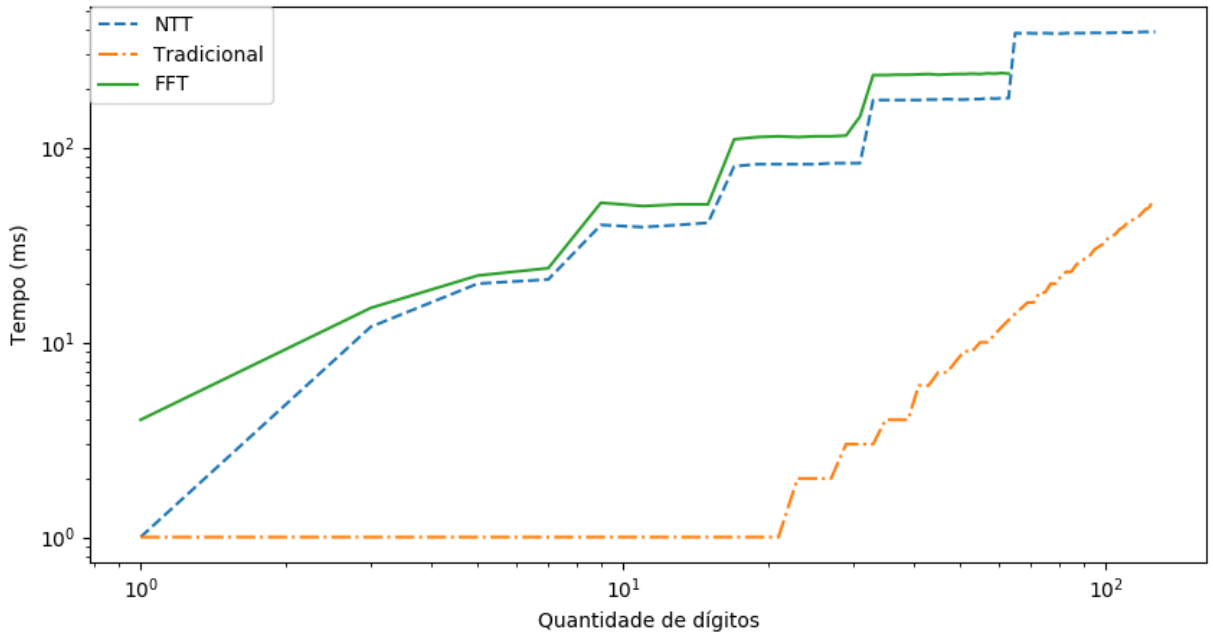

Figura 12 – Gráficos dos tempos de execução dos algoritmos implementados no Arduino em escala logarítmica

Do gráfico da Figura 12, observa-se que o tamanho das entradas não foram grandes o suficiente para os algoritmos da FFT e NTT serem mais rápidas que o tradicional. Além disso, o algoritmo da NTT foi mais rápido que o da FFT, já que o ATmega328P não possui coprocessador aritmético capaz de realizar operações com pontos flutuantes.

4.2.3 NodeMCU

Foi possível alocar no máximo 32768 *bytes* de memória contígua no NodeMCU. Logo, foram testados números com a seguinte quantidade de dígitos utilizando os resultados da Tabela 2 (considerando $m = n$):

- **Tradicional:** $5n + 5m - 4 = 32768 \Rightarrow n \approx 3270$ dígitos
- **NTT:** $2^{\lceil \log_2 (n+m) \rceil + 4} = 32768 \Rightarrow n \approx 2048$ dígitos
- **FFT:** $2^{\lceil \log_2 (n+m) \rceil + 3} = 32768 \Rightarrow n \approx 1024$ dígitos

O gráfico da Figura 13 apresenta as diferentes performances dos 3 algoritmos implementados no NodeMCU.

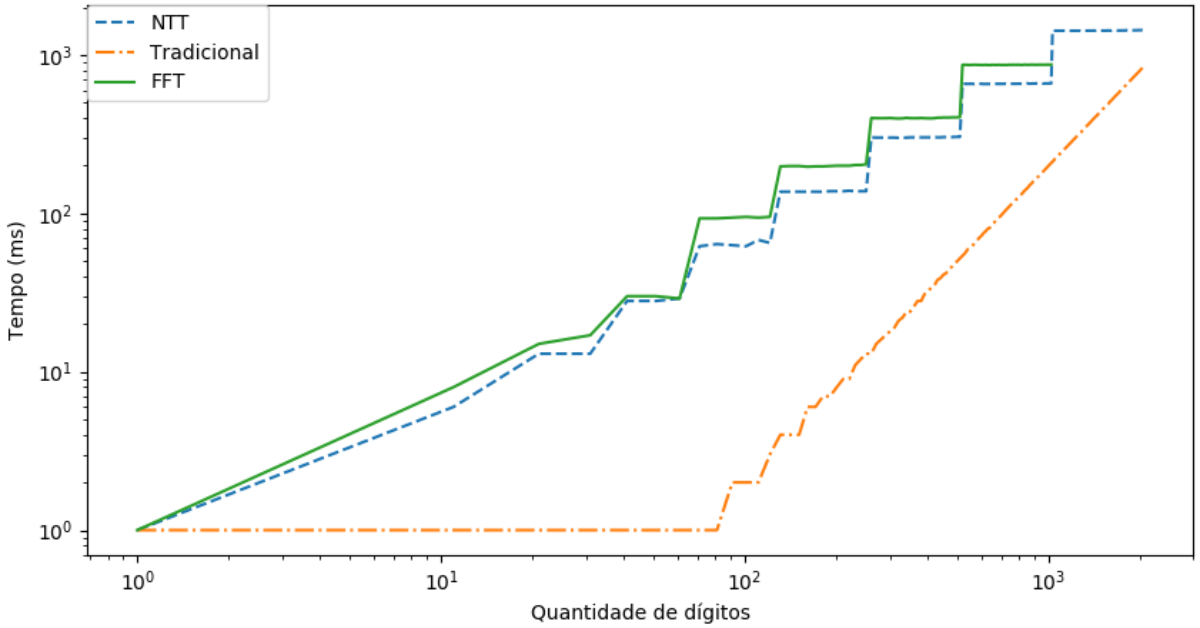

Figura 13 – Gráficos dos tempos de execução dos algoritmos implementados no NodeMCU em escala logarítmica

Os resultados encontrados no NodeMCU foram semelhantes dos encontrados no Arduino. Do gráfico da Figura 13, nota-se que, apesar do algoritmo tradicional ainda ser mais rápido que a FFT e NTT, há a tendência do tempo de execução do algoritmo tradicional ultrapassar os tempos da FFT e NTT.

4.2.4 Raspberry Pi 3

O gráfico da Figura 14 apresenta as diferentes performances dos 3 algoritmos implementados no Raspberry Pi 3.

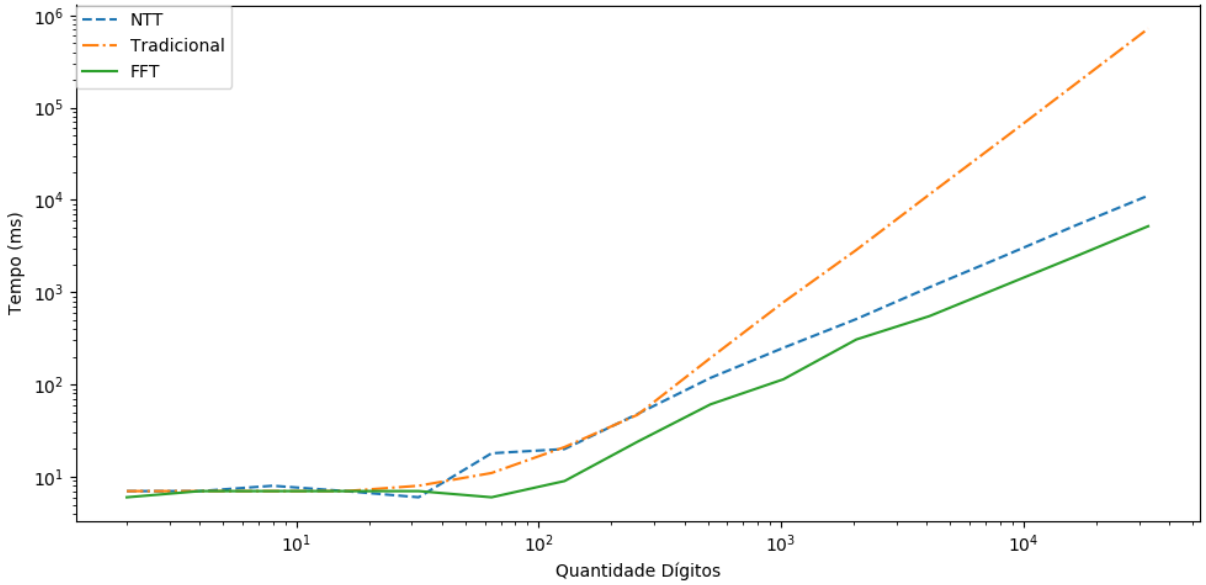

Figura 14 – Gráficos dos tempos de execução dos algoritmos implementados no Raspberry Pi em escala logarítmica

Como o Raspberry Pi 3 possui 1 GB de memória RAM, foi possível utilizar os tamanhos apresentados na Seção 3.5.1. Nota-se pelo gráfico da Figura 14 que o algoritmo tradicional ultrapassou os algoritmos da FFT e NTT com números de cerca de 200 dígitos. Além disso, o algoritmo da FFT foi mais rápida do que a NTT, já que o Raspberry Pi coprocessador aritmético capaz de realizar operações com pontos flutuantes.

5 Considerações Finais

Neste trabalho, foram apresentados algoritmos para multiplicação de números complexidade assintótica melhor do que o método tradicional apresentado na escola, assim como a análise de suas eficiências em *software*. Foi visto que, na prática, o algoritmo Tradicional é eficiente para números de até 150 dígitos e a FFT e a NTT para números de mais de 150 dígitos. Além disso, ficou claro que uma simples operação matemática básica como a multiplicação pode possuir uma implementação sofisticada, envolvendo números complexos e aritmética modular.

O algoritmo de Karatsuba, FFT e NTT são conhecidos e implementados principalmente em *software*, mas raramente em *hardware* devido à natureza recursiva (para o algoritmo de Karatsuba) e de uso de números muito grandes (para a FFT e NTT) (KALACH; DAVID; TITTLEY, 2007). Caso não haja um *hardware* especializado para uma aplicação, a solução é implementar o algoritmo em *software*. Estes métodos eficazes de multiplicação podem se tornar ferramentas que otimizam outros algoritmos, como o RSA.

O estudo realizado no Capítulo 2 e os resultados apresentados no Capítulo 4 mostram que se deve analisar os seguintes tópicos na escolha de um algoritmo de multiplicação para uma aplicação específica: quantidade de dígitos dos números de entrada, memória disponível no *hardware* local, precisão e existência da aritmética de ponto-flutuante implementada em *hardware*.

Trabalhos Futuros

Algoritmos para multiplicação de números inteiros é uma área recente, com novas ideias sendo exploradas atualmente. Apesar de diversos estudos com grandes progressos realizados até os dias atuais, não se conhece ainda qual o algoritmo com a menor complexidade assintótica para multiplicar números inteiros (ver Seção 2.6.4). Como sugestão de trabalhos futuros, pode-se apontar:

- Implementação e análise do algoritmo de Karatsuba;
- Otimizar ainda mais os algoritmos (como aproveitar melhor a memória utilizada para guardar os números);
- Estudo da implementação dos algoritmos utilizado-se programação paralela. Em placas de vídeo, por exemplo, é possível acelerar a velocidade do algoritmo da NTT e FFT;

- Estudo da implementação dos algoritmos em *hardware*, como na plataforma FPGA;
- Analisar os ganhos de performance do algoritmo RSA utilizando diferentes algoritmos de multiplicação;
- Implementação dos algoritmos em Assembly, permitindo adicionar a instrução de multiplicação em hardwares sem multiplicadores;
- Criar uma implementação híbrida como foi feita na biblioteca GMP, a qual combina vários algoritmos de multiplicação e decide o uso de cada um de acordo com suas características;

Referências

AGARWAL, R. C.; BURRUS, C. S. Number theoretic transforms to implement fast digital convolution. *Proceedings of the IEEE*, v. 63, n. 4, p. 550–560, April 1975. ISSN 0018-9219. Citado na página 30.

ATMEL. *ATMEL 8-BIT MICROCONTROLLER WITH 4/8/16/32KBYTES IN-SYSTEM PROGRAMMABLE FLASH DATASHEET*. 2015. Rev.: Atmel-8271J-AVR-ATmega48A/48PA/88A/88PA/168A/168PA/328/328P-Datasheet_11/2015. Citado na página 55.

BOYER, C. *A History of Mathematics*. Wiley, 1991. ISBN 9780471543978. Disponível em: <https://books.google.com.br/books?id=OtsY845tywQC>. Citado na página 21.

BRENT, R.; ZIMMERMANN, P. *Modern Computer Arithmetic*. New York, NY, USA: Cambridge University Press, 2010. ISBN 0521194695, 9780521194693. Citado 4 vezes nas páginas 15, 26, 27 e 43.

BURTON, D. *Elementary number theory*. Allyn & Bacon, Incorporated, 1976. ISBN 9780205048144. Disponível em: <https://books.google.com.br/books?id=PPvuAAAAMAAJ>. Citado na página 42.

CHABERT, J.; BARBIN, E. *A History of Algorithms: From the Pebble to the Microchip*. Springer Berlin Heidelberg, 1999. (chabert: A History of Algorithms). ISBN 9783540633693. Disponível em: <https://books.google.com.br/books?id=B1c0s3ffN_0C>. Citado na página 21.

COCHRAN, W. T. et al. What is the fast fourier transform? *Proceedings of the IEEE*, v. 55, n. 10, p. 1664–1674, Oct 1967. ISSN 0018-9219. Citado 3 vezes nas páginas 30, 32 e 33.

COHEN, H. *A Course in Computational Algebraic Number Theory*. New York, NY, USA: Springer-Verlag New York, Inc., 1993. ISBN 0-387-55640-0. Citado 2 vezes nas páginas 41 e 42.

CORMEN, T. H. et al. *Introduction to Algorithms*. 2nd. ed. Cambridge, MA, USA: McGraw-Hill Higher Education, 2001. ISBN 0070131511. Citado 6 vezes nas páginas 15, 28, 30, 32, 34 e 35.

ESPRESSIF. *ESP8266EX Datasheet*. 2018. Version 5.8. Citado na página 55.

FURER, M. Faster integer multiplication. In: *Proceedings of the Thirty-ninth Annual ACM Symposium on Theory of Computing*. New York, NY, USA: ACM, 2007. (STOC '07), p. 57–66. ISBN 978-1-59593-631-8. Disponível em: <http://doi.acm.org/10.1145/1250790.1250800>. Citado 2 vezes nas páginas 22 e 48.

GOURDON, P. S. X. Fft based multiplication of large numbers. 2001. Disponível em: <xavier.gourdon.free.fr/Constants/Algorithms/fft.ps>. Citado na página 40.

GUIMARAES, C. d. S. *Números complexos e polinômios*. : Vestseller, 2008. v. 1. (Matemática em Nível IME ITA, v. 1). Citado na página 30.

HALFACREE, G.; UPTON, E. *Raspberry Pi User Guide*. 1st. ed. : Wiley Publishing, 2012. ISBN 111846446X, 9781118464465. Citado na página 56.

HASTAD, J. *Notes for the course advanced algorithms*. 2000. Disponível em: <https://www.nada.kth.se/~johanh/algnotes.pdf>. Citado na página 48.

IRELAND, K.; ROSEN, M. *A classical introduction to modern number theory*. Springer-Verlag, 1982. (Graduate texts in mathematics). ISBN 9783540906254. Disponível em: <https://books.google.com.br/books?id=WvjuAAAAMAAJ>. Citado na página 42.

KALACH, K.; DAVID, J. P.; TITTLEY, N. Hardware complexity of modular multiplication and exponentiation. *IEEE Transactions on Computers*, IEEE Computer Society, Los Alamitos, CA, USA, v. 56, p. 1308–1319, 2007. ISSN 0018-9340. Citado 2 vezes nas páginas 26 e 65.

KARATSUBA, A. The complexity of computations. v. 211, p. 169–, 01 1995. Citado na página 22.

KLEINBERG, J.; TARDOS, E. *Algorithm Design*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2005. ISBN 0321295358. Citado 3 vezes nas páginas 13, 23 e 25.

KNUTH, D. E. *The TeXbook*. : Addison-Wesley Professional, 1986. ISBN 0201134470. Citado na página 51.

KNUTH, D. E. *The Art of Computer Programming, Volume 2 (3rd Ed.): Seminumerical Algorithms*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1997. ISBN 0-201-89684-2. Citado 4 vezes nas páginas 22, 27, 28 e 35.

Lüders, C. Fast Multiplication of Large Integers: Implementation and Analysis of the DKSS Algorithm. *ArXiv e-prints*, mar. 2015. Citado 2 vezes nas páginas 19 e 25.

NUSSBAUMER, H. *Fast Fourier Transform and Convolution Algorithms*. Springer Berlin Heidelberg, 1981. (Springer Series in Information Sciences). ISBN 9783662005514. Disponível em: <https://books.google.fr/books?id=tnjpCAAAQBAJ>. Citado na página 41.

O'CONNOR, J. J.; ROBERTSON, E. F. *Memory, mental arithmetic and mathematics*. 1997. <http://www-history.mcs.st-and.ac.uk/HistTopics/Mental_arithmetic.html>. Accessed: 24-11-2017. Citado 2 vezes nas páginas 21 e 50.

RAFFERTY, C.; ONEILL, M.; HANLEY, N. Evaluation of large integer multiplication methods on hardware. *IEEE Transactions on Computers*, IEEE Computer Society, Los Alamitos, CA, USA, v. 66, n. 8, p. 1369–1382, 2017. ISSN 0018-9340. Citado na página 19.

ROUGHGARDEN, T. *Algorithms Illuminated - Part 1*. : Soundlikeyourself Publishing, 2017. (Algorithms Illuminated). ISBN 0999282905. Citado na página 23.

SEDGEWICK, R. *Algorithms in C++ - Parts 1-4: Fundamentals, Data Structures, Sorting, Searching*. : Addison-Wesley-Longman, 1998. ISBN 978-0-201-35088-3. Citado na página 23.

SHAHRAM; AZMAN; KUMBAKONAM. Efficient big integer multiplication and squaring algorithms for cryptographic applications. Penang 11800, Malaysia, p. 10, 07 2014. Citado 2 vezes nas páginas 22 e 27.

SMITH, J. O. *Introduction to Digital Filters with Audio Applications*. http://www.w3k.org/books/: W3K Publishing, 2007. ISBN 978-0-9745607-1-7. Citado 3 vezes nas páginas 32, 43 e 48.

STALLINGS, W. *Cryptography and Network Security: Principles and Practice*. 6th. ed. Upper Saddle River, NJ, USA: Prentice Hall Press, 2013. ISBN 0133354695, 9780133354690. Citado na página 22.

STROUSTRUP, B. *The C++ Programming Language*. 4th. ed. : Addison-Wesley Professional, 2013. ISBN 0321563840, 9780321563842. Citado na página 51.

TOLIMIERI, R.; AN, M.; LU, C. *Algorithms for Discrete Fourier Transform and Convolution*. 2nd. ed. : Springer-Verlag New York, 1997. ISBN 978-1-4757-2767-8. Citado 3 vezes nas páginas 28, 29 e 32.

TOMMILA, M. *Number theoretic transforms*. 1997. <http://www.apfloat.org/ntt.html>. Accessed: 26-11-2017. Citado 2 vezes nas páginas 48 e 50.

ZIVIANI, N. *Projeto de Algoritmos: com implementações em Java e C++ de Fabiano Cupertino*. 2nd. ed. : Cengage Learninig Edições Ltda., 2011. Citado na página 23.

Apêndices

APÊNDICE A – Algoritmo da Convolução Tradicional

```
1 vector<int> ConvolucaoTradicional(vector<int> a, vector<int> b)
2 {
3   int A = a.size(), B = b.size();
4   vector<int> v(A + B - 1, 0);
5   for(int i = 0 ; i < A ; i++)
6     for(int j = 0 ; j < B ; j++)
7         v[i + j] += a[i] * b[j];
8   return v;
9 }
```

APÊNDICE B – Algoritmo da FFT Recursivo

```
1  vector<complex<float>> FFT(vector<complex<float>> a)
2  {
3    int n = a.size();
4    if(n == 1)
5      return a;
6
7    vector<complex<float>> a_par(n >> 1), a_impar(n >> 1);
8    for(int i = 0; i < (n >> 1); i++)
9    {
10     a_par[i] = a[i << 1];
11     a_impar[i] = a[(i << 1)+1];
12   }
13
14   vector<complex<float>> A_par = FFT(a_par);
15   vector<complex<float>> A_impar = FFT(a_impar);
16   vector<complex<float>> A(n);
17
18   complex<float> w(1.0, 0);
19   complex<float> omega((float)cos(2*M_PI/n), (float)sin(2*M_PI/n));
20   n >>= 1;
21
22   for(int k = 0; k < n; k++)
23   {
24     complex<float> u = A_par[k];
25     complex<float> v = w * A_impar[k];
26     A[k]   = u + v;
27     A[k + n] = u - v;
28     w = w * omega;
29   }
30
31   return A;
32 }
```

APÊNDICE C – Algoritmo da FFT Iterativo

```
1  void FFT(complex<float> a[], int size_a)
2  {
3    for(int j = 1, i = 0; j <= size_a-1; j++)
4    {
5      for(int k = size_a >> 1; k > (i ^= k); k >>= 1);
6      if(j < i)
7        swap(a[i], a[j]);
8    }
9    for(int len = 1; len < size_a; len <<= 1)
10   {
11     const int unityStep = len<<1;
12     const float theta = -2 * M_PI / unityStep;
13     const complex<float> unityRoot(cos(theta), sin(theta));
14     for(int i = 0; i < size_a; i += len << 1)
15     {
16       complex<float> omega = 1;
17       for(int j = 0; j < len; j++)
18       {
19         complex<float> u = a[i + j];
20         complex<float> v = a[i + j + len] * omega;
21         omega *= unityRoot;
22
23         a[i + j] = u + v;
24         a[i + j + len] = u - v;
25       }
26     }
27   }
28 }
```

APÊNDICE D – Algoritmo da NTT Recursivo

```
1  const int md = 998244353; //119*2^23 + 1
2  const int gen = 3;      //primitive root of md
3  //exponenciacao rapida
4  int mod_pow(int a, int b)
5  {
6    int r = 1;
7    while(b)
8    {
9      if(b & 1)
10       r = (long long) r * a % md;
11     a = (long long) a * a % md;
12     b >>= 1;
13   }
14   return r;
15 }
16
17 vector<int> NTT(vector<int> a)
18 {
19   int n = a.size();
20   if(n == 1)
21     return a;
22
23   vector<int> a_par(n/2), a_impar(n/2);
24   for(int i = 0 ; i < n/2 ; i++)
25   {
26     a_par[i] = a[2*i];
27     a_impar[i] = a[2*i+1];
28   }
29
30   vector<int> y_par = NTT(a_par);
31   vector<int> y_impar = NTT(a_impar);
32   vector<int> y(n);
33
34   int c = (md - 1) / n; //P = c * n + 1 -> c = (P - 1) / n
35   int omega = mod_pow(gen, c); // omega = r ^ c mod P
36   int w = 1;
37   n >>= 1;
38   for(int k = 0; k < n; k++)
39   {
40     int u = y_par[k];
```

```
41        int v = (long long) w * y_impar[k] % md;
42        y[k] = (u + v) % md;
43        y[k + n] = (u - v + md) % md;
44        w = (long long) w * omega % md;
45      }
46      return y;
47  }
```

APÊNDICE E – Algoritmo da NTT Iterativo

```
1  const int md = 998244353; //119*2^23 + 1
2  const int gen = 3;      //primitive root of md
3
4  //exponenciacao rapida
5  int mod_pow(int a, int b)
6  {
7    int r = 1;
8    while(b)
9    {
10     if(b & 1)
11       r = (long long) r * a % md;
12     a = (long long) a * a % md;
13     b >>= 1;
14   }
15   return r;
16 }
17
18 void ntt(int a[], int size_a)
19 {
20   for(int j = 1, i = 0; j <= size_a-1; j++)
21   {
22     for(int k = size_a >> 1; k > (i ^= k); k >>= 1);
23     if(j < i)
24       swap(a[i], a[j]);
25   }
26   for(int len = 1; len < size_a; len <<= 1)
27   {
28     int root = mod_pow(gen, (md - 1) / (len << 1));
29     for(int i = 0; i < size_a; i += len << 1)
30     {
31       int w = 1;
32       for(int j = 0; j < len; j++)
33       {
34         int u = a[i + j];
35         int v = (long long) a[i + j + len] * w % md;
36         a[k]   = (u + v)   % md;
37         a[k + n] = (u - v + md) % md;
38         w = (long long) w * root % md;
39       }
40     }
```

```
41        }
42    }
```

APÊNDICE F – Algoritmo de multiplicação com FFT

```
1  void multiply(vc &a, vc &b)
2  {
3    int size_a = a.size(), size_b = b.size();
4
5    //complete arrays with 0 to get the size of the next power of 2
6    int size = 1, largest = size_a + size_b - 1;
7    while(size < largest)
8      size <<= 1;
9    for(int i = size_a; i < size; i++)
10     a.push_back(complex<float>(0, 0));
11   for(int i = size_b; i < size; i++)
12     b.push_back(complex<float>(0, 0));
13
14   //convolution with fft
15   a = fft(a);
16   b = fft(b);
17   for(int i = 0; i < size; i++)
18     a[i] = a[i] * b[i];
19
20   //inverse fft
21   for(int i = 1; i < size >> 1; i++)
22     swap(a[i], a[size-i]);
23   a = fft(a);
24   for (int i = 0; i < size; i++)
25     a[i] = round(a[i].real() / (float) size);
26 }
```

APÊNDICE G – Algoritmo de multiplicação com NTT

```
1  void multiply(vector<int> &a, vector<int> &b)
2  {
3    int size_a = a.size(), size_b = b.size();
4
5    //complete arrays with 0 to get the size of the next power of 2
6    int size = 1, largest = size_a + size_b - 1;
7    while(size < largest)
8      size <<= 1;
9    for(int i = size_a; i < size; i++)
10     a.push_back(0);
11   for(int i = size_b; i < size; i++)
12     b.push_back(0);
13
14   //convolution with ntt
15   a = ntt(a);
16   b = ntt(b);
17   for(int i = 0; i < size; i++)
18     a[i] = (long long) a[i] * b[i] % md;
19
20   //inverse ntt
21   for(int i = 1; i < size >> 1; i++)
22     swap(a[i], a[size-i]);
23   a = ntt(a);
24   int inv = mod_pow(size, md - 2);
25   for (int i = 0; i < size; i++)
26     a[i] = (long long) a[i] * inv % md;
27 }
```