

# Máquinas de Turing para Teste de Primalidade

Vinicius Quaresma da Luz<sup>1</sup>

<sup>1</sup>Departamento de Computação – Universidade Federal de São Carlos (UFSCar)  
São Carlos – SP – Brazil

`viniciusluz@estudante.ufscar.br`

**Resumo.** *Em um contexto no qual assuntos nos campos da Inteligência Artificial ganham mais tração do que nunca, é pertinente discutir o modelo teórico que possibilitou o nascimento da ideia formal de algoritmos: a Máquina de Turing. Este artigo busca discutir sobre este modelo teórico e fornecer uma implementação com viés didático utilizando a linguagem de programação Python, permitindo visualizar o funcionamento de uma máquina cujo propósito é determinar se um certo número é ou não um número primo.*

## 1. Introdução

A Máquina de Turing, proposta por Alan Turing em 1936, é um modelo matemático fundamental para a ciência da computação. Sua função é formalizar o conceito de algoritmo e definir os limites teóricos da computabilidade, servindo como base para o estudo do que pode ser resolvido por um processo mecânico [Turing 1937]. Apesar de sua simplicidade conceitual, seu poder computacional é universal, sendo capaz de simular qualquer algoritmo executado por computadores modernos.

Dentre os problemas clássicos da computação, o teste de primalidade ocupa um lugar de destaque. A tarefa de determinar se um número inteiro é primo ou composto é uma questão fundamental da teoria dos números, com raízes que remontam à matemática da Grécia Antiga [Knuth 1997]. Para a ciência da computação, o problema representa um campo de provas para o desenvolvimento e a análise de algoritmos, bem como para a medição de eficiência computacional.

A relevância do teste de primalidade não é apenas teórica, possuindo aplicações práticas críticas, especialmente na área de criptografia. Sistemas de chave pública, como o RSA, baseiam sua segurança na dificuldade computacional de fatorar números grandes que são o produto de dois primos [Rivest et al. 1978]. Além disso, números primos são utilizados em algoritmos de hashing e na geração de números pseudoaleatórios, tornando o problema central para diversas áreas da computação aplicada [Cormen et al. 2009].

Este trabalho tem como objetivo explorar a resolução do teste de primalidade utilizando o modelo da Máquina de Turing. O foco não reside no desenvolvimento de um algoritmo com máxima eficiência, mas na demonstração de como um modelo computacional teórico pode ser aplicado para resolver um problema concreto. Busca-se, com isso, detalhar o processo lógico e mecânico de uma computação em seu nível mais fundamental.

Para atingir este objetivo, foi projetada e implementada uma Máquina de Turing que decide sobre a primalidade de um número. A abordagem utiliza uma representação unária para a entrada e executa um algoritmo baseado em divisões sucessivas. A

implementação, realizada na linguagem Python, permite simular o modelo teórico e validar seu funcionamento, ilustrando a capacidade do modelo de Turing de encapsular e resolver problemas complexos.

## 2. Materiais e Métodos

### 2.1. A Máquina de Turing

A Máquina de Turing constitui um modelo matemático abstrato de computação que formaliza uma máquina teórica com capacidade para simular a lógica de qualquer algoritmo computável. Proposta por Alan Turing em seu artigo seminal de 1936, "*On Computable Numbers, with an Application to the Entscheidungsproblem*", a máquina foi concebida com o propósito de formalizar o conceito de "computabilidade", bem como explorar os limites do que é mecanicamente decidível [Turing 1937]. Historicamente, o trabalho de Turing estabeleceu as fundações da ciência da computação teórica, ao prover uma definição rigorosa para a noção de algoritmo.

Formalmente, uma Máquina de Turing determinística é representada por uma 7-tupla  $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ , onde:

- $Q$  é um conjunto finito de **estados** internos.
- $\Sigma$  é o **alfabeto de entrada**, um conjunto finito de símbolos que não inclui o símbolo branco.
- $\Gamma$  é o **alfabeto da fita**, um superconjunto finito de  $\Sigma$ .
- $\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$  é a **função de transição**, que dita a ação da máquina a cada passo.
- $q_0 \in Q$  é o **estado inicial**, a partir do qual a computação se inicia.
- $B \in \Gamma \setminus \Sigma$  é o **símbolo branco**, que preenche a fita além da entrada.
- $F \subseteq Q$  é o conjunto de **estados de aceitação**.

O processo computacional tem início com a cadeia de entrada disposta na fita, com a máquina no estado  $q_0$  e o cabeçote posicionado sobre o primeiro símbolo. A cada passo, a função  $\delta$  é aplicada. A computação é terminada quando a máquina atinge um estado pertencente a  $F$ , resultando na aceitação da entrada, ou quando atinge uma configuração para a qual  $\delta$  não está definida, resultando na rejeição.

### 2.2. Técnicas para Construção de Máquinas de Turing

Embora o modelo formal da Máquina de Turing seja simples, o projeto de máquinas para resolver problemas complexos pode se tornar uma tarefa árdua. Para facilitar esse processo, foram desenvolvidas diversas técnicas conceituais que, embora não aumentem o poder computacional do modelo (pela tese de Church-Turing), simplificam o projeto e a descrição dos algoritmos. Destacam-se [Souza 2025]:

- **Armazenamento no Controle Finito:** Esta técnica permite que o estado da máquina seja representado por uma tupla de componentes, como  $[q, A]$ , onde  $q$  é um estado de controle e  $A$  é um símbolo armazenado na "memória" do controle finito. Isso é útil para que a máquina possa "lembrar" de um símbolo lido sem a necessidade de escrevê-lo repetidamente na fita.

- **Fitas com Múltiplas Trilhas:** A fita pode ser visualizada como se tivesse um número finito  $k$  de trilhas. Cada célula da fita contém uma tupla de  $k$  símbolos, e o cabeçote lê e escreve os  $k$  símbolos simultaneamente. Essa técnica é particularmente útil para manter diferentes informações alinhadas, como um número de entrada em uma trilha, um divisor em outra e resultados de cálculos em uma terceira, como no caso do algoritmo de teste de primalidade.
- **Marcação de Símbolos:** Consiste em utilizar um alfabeto de fita estendido para marcar símbolos que já foram processados. Frequentemente, isso é implementado com uma fita de duas trilhas, onde a primeira trilha contém a cadeia de entrada e a segunda é usada para marcar posições com um símbolo especial (e.g., *check*), indicando que o símbolo correspondente na primeira trilha já foi verificado.
- **Sub-rotinas:** Uma Máquina de Turing pode ser projetada de forma modular, onde partes do conjunto de estados são designadas para realizar tarefas específicas, funcionando como sub-rotinas. Uma sub-rotina possui um estado inicial próprio e um ou mais estados de retorno, permitindo que a máquina principal "chame" a sub-rotina para executar uma função e depois retome seu processamento.

Algumas destas técnicas são utilizadas na elaboração da máquina que é objeto deste artigo.

### 3. Implementação

Para a implementação da Máquina de Turing, foi utilizada a linguagem de programação Python. A estrutura adotada tenta emular o funcionamento de uma Máquina de Turing, gerenciando elementos de seu estado como a posição do controle interno, estado das fitas, dentre outros. O programa aceita uma entrada binária  $w_1 \geq 2_{(10)}$  e fornece um resultado  $r \in \{q_{\text{accept}}, q_{\text{reject}}\}$ .

O código-fonte completo da máquina de Turing em Python, discutido neste artigo, está disponível publicamente no repositório GitHub abaixo.

- **Repositório GitHub:** <https://github.com/Lvght/turing-machine>

Abaixo, buscando elucidar como o código funciona e melhor explicar o processo de testes na seção subsequente, estão alguns trechos adaptados do código.

```

1 class TuringMachineDefinition:
2     Q: Set[State]
3     Sigma: Set[str]
4     Gamma: Set[str]
5     delta: Delta
6     q0: State
7     BlankSymbol: str
8     F: Set[State]
9
10    def __init__(
11        self,
12        Q: Set[State],
13        Sigma: Set[str],
14        Gamma: Set[str],
15        delta: Delta,
16        q0: State,
17        BlankSymbol: str,

```

```

18     F: Set[State],
19 ):
20     self.Q = Q
21     self.Sigma = Sigma
22     self.Gamma = Gamma
23     self.delta = delta
24     self.q0 = q0
25     self.BlankSymbol = BlankSymbol
26     self.F = F

```

### Código 1. Definição da Máquina de Turing

A classe que consome essa definição é a classe `TURINGMACHINE`, exposta abaixo. Note que algumas omissões foram feitas para fins de brevidade.

```

1 class TuringMachine:
2     _definition: TuringMachineDefinition
3     _tape: Dict[int, TapeBlock]
4     _head_position: int
5     _current_state: State
6     _verbose: bool
7     _step_count: int
8
9     def __init__(
10         self,
11         definition: TuringMachineDefinition,
12         w_1: str,
13         w_2: Optional[str] = None,
14         w_3: Optional[str] = None,
15         verbose: bool = False,
16     ):
17         self._verbose = verbose
18         self._definition = definition
19         self._current_state = self._definition.q0
20         self._head_position = 0
21         self._step_count = 0
22         # ...
23         self._tape = {
24             i: (w_1[i], w_2[i], w_3[i]) for i in range(max_len)
25         }
26
27     def run(self) -> TuringMachineExcecutionResult:
28         while self._current_state not in self._definition.F:
29             self.step()
30         # ...
31         return TuringMachineExcecutionResult(
32             step_count=self._step_count,
33             final_state=self._current_state,
34         )
35
36     def step(self):
37         self._step_count += 1
38         # ...
39         next_state, write_template, direction = self._fd_transition()
40         current_symbols = self.current_tape_block
41         new_symbols_to_write = tuple(
42             current_symbols[i]

```

```

43         if write_template[i] == "*"
44         else write_template[i] for i in range(3)
45     )
46     # ...
47     self._current_state = next_state
48     self._tape[self._head_position] = new_symbols_to_write
49     self._head_position += direction.value
50     # ...

```

## Código 2. Executor para a definição da Máquina de Turing

A lógica principal da simulação ocorre no método STEP e no método FIND\_TRANSITION. No método STEP, o programa avalia seu estado atual e o conteúdo da fita sob o controle da Máquina de Turing. Com isso, ele avalia a função  $\delta$  e determina qual será o próximo passo. A função FIND\_TRANSITION é a responsável por encontrar o  $\delta$ , ou lançar um MISSING\_TRANSITION\_ERROR caso não encontre. Caso a função lance a exceção, a máquina tem o seu funcionamento interrompido.

Um ponto digno de menção é que esta implementação oferece suporte a *wildcards* nas regras de definição de  $\delta$ , simplificando expressivamente a definição das regras.

```

1 def _find_transition(self) -> Tuple[State, TapeBlock, Direction]:
2     current_symbols = self.current_tape_block
3
4     for (
5         state,
6         read_symbols,
7     ), transition_output in self._definition.delta.items():
8         if state == self._current_state:
9             is_match = all(
10                 read_symbols[i] == current_symbols[i]
11                 or read_symbols[i] == "*"
12                 for i in range(3)
13             )
14             if is_match:
15                 return transition_output
16
17     raise MissingTransitionError(
18         f"Não existe \delta({self._current_state}, {current_symbols})"
19     )

```

## Código 3. Método para encontrar $\delta$ com base no estado atual da máquina

Além disso, a cada passo da máquina, caso a opção *verbose* esteja habilitada, ela exibe em sua saída padrão as informações no seguinte formato:

```

1 -> delta(q_seek_end, ('1', '1', '0'))
2 Sub-rotina: compare_ge
3 F1:      | B | 1 | 0 | 1 | 1 | B | B ...
4 F2:      | B | 0 | 0 | 1 | 1 | B | B ...
5 F3:      | B | 1 | 0 | 0 | 0 | B | B ...
6 CTRL:      ^      compare_ge__q_seek_end
7
8 Próximo estado: q_seek_end (Parte da sub-rotina: compare_ge)
9 Escrever:      ('1', '1', '0')
10 Direção:      RIGHT

```

## Código 4. Informações exibidas pelo método step

A linha CTRL mostra uma seta apontando para onde está o controle da máquina naquele instante, e isso é atualizado visualmente a cada iteração da máquina.

### 3.1. Algoritmo utilizado

```
1 ALGORITMO TestePrimalidade
2 INICIO
3     FITA_1 <- ENTRADA
4
5     # Dividir
6     FITA_2 <- (2)_10
7
8     # Loop principal, responsável por aumentar o divisor
9     # Fica em execução até aceitar, rejeitar ou até a máquina parar.
10    ENQUANTO VERDADEIRO FAÇA
11        SE FITA_1 = FITA_2 ENTÃO
12            ACEITA
13            FIMALGORITMO
14        SENÃO
15            FITA_3 <- FITA_1
16
17            # Loop de subtração
18            ENQUANTO FITA_3 >= FITA_2 FAÇA
19                FITA_3 <- FITA_3 - FITA_2
20            FIMENQUANTO
21
22            SE FITA_3 = (0)_10 ENTÃO
23                REJEITA
24                FIMALGORITMO
25            SENÃO
26                FITA_2 <- FITA_2 + (1)_10
27            FIMSE
28        FIMSE
29    FIMENQUANTO
30 FIMALGORITMO
```

**Código 5. Pseudocódigo utilizado para guiar o projeto e implementação da Máquina de Turing**

### 3.2. Definição da Máquina de Turing

Utilizando as demais classes definidas para a execução deste projeto, foram criadas as definições e regras necessárias para expressar o algoritmo desejado utilizando a máquina de Turing.

Para criar esta máquina, foi adotada a estratégia de quebrá-la em sub-rotinas. A notação utilizada foi a de que o nome da sub-rotina é seguido por 2 *underlines* e então pelo nome do estado. Isto é utilizado pela visualização completa posteriormente para facilitar a compreensão de qual sub-rotina está em execução a cada passo.

Abaixo está sua definição. Por conta de sua extensão, a definição de  $\delta$  é exibida em uma figura separada logo abaixo.

```
1 primality_test_definition = TuringMachineDefinition(
2     Q={
3         # Estados do Orquestrador
```

```

4      "q_start",
5      "q_accept",
6      "q_reject",
7      # Sub-rotina: INIT_DIVISOR
8      "init__q0",
9      "init__q_write0",
10     "init__q_writel",
11     "init__q_fill_zeros",
12     "init__q_rewind",
13     # Sub-rotina: COMPARE_EQUALITY (N == D)
14     "compare_eq__q_seek_end",
15     "compare_eq__q_compare",
16     "compare_eq__q_notequal_rewind",
17     # Sub-rotina: COPY_TAPE_1_TO_3
18     "copy__q_copy",
19     "copy__q_rewind",
20     # Sub-rotina: COMPARE_GREATER_EQUAL (T3 >= T2)
21     "compare_ge__q_seek_end",
22     "compare_ge__q_compare_len",
23     "compare_ge__q_rewind_msb",
24     "compare_ge__q_compare_msb",
25     "compare_ge__q_rewind_false",
26     "compare_ge__q_rewind_true",
27     # Sub-rotina: SUBTRACT
28     "subtract__q0",
29     "subtract__q1",
30     "subtract__q2",
31     "subtract__q_rewind",
32     # Sub-rotina: CHECK_REMAINDER_IS_ZERO
33     "check_remainder__q_check",
34     "check_remainder__q_rewind",
35     # Sub-rotina: CLEANUP_TAPE_3
36     "cleanup__q_0",
37     "cleanup__q_rewind",
38     # Sub-rotina: INCREMENT_DIVISOR
39     "increment__q_seek_end",
40     "increment__q_add_carry",
41     "increment__q_rewind",
42 },
43 Sigma={"0", "1"},
44 Gamma={"0", "1", "B"},
45 BlankSymbol="B",
46 q0="q_start",
47 F={"q_accept", "q_reject"},
48 delta=# Incluído na figura abaixo.
49 )

```

**Código 6. Definição da Máquina de Turing para teste de primalidade**

```

1  # =====
2  # Ponto de Entrada
3  # =====
4  ("q_start", ("*", "*", "*")): (
5      "init__q0",
6      ("*", "*", "*"),
7      Direction.RIGHT,
8  ),

```

```

9 # =====
10 # SUB-ROTINA 1: INIT_DIVISOR (FITA_2 <- 2)
11 # =====
12 ("init__q0", ("0", "*", "*")): (
13     "init__q0",
14     ("*", "0", "*"),
15     Direction.RIGHT,
16 ),
17 ("init__q0", ("1", "*", "*")): (
18     "init__q0",
19     ("*", "0", "*"),
20     Direction.RIGHT,
21 ),
22 ("init__q0", ("B", "B", "B")): (
23     "init__q_write0",
24     ("B", "B", "B"),
25     Direction.LEFT,
26 ),
27 ("init__q_write0", ("*", "*", "*")): (
28     "init__q_writel",
29     ("*", "0", "*"),
30     Direction.LEFT,
31 ),
32 ("init__q_writel", ("*", "*", "*")): (
33     "init__q_fill_zeros",
34     ("*", "1", "*"),
35     Direction.LEFT,
36 ),
37 ("init__q_fill_zeros", ("0", "*", "*")): (
38     "init__q_fill_zeros",
39     ("*", "0", "*"),
40     Direction.LEFT,
41 ),
42 ("init__q_fill_zeros", ("1", "*", "*")): (
43     "init__q_fill_zeros",
44     ("*", "0", "*"),
45     Direction.LEFT,
46 ),
47 ("init__q_fill_zeros", ("B", "B", "B")): (
48     "init__q_rewind",
49     ("B", "B", "B"),
50     Direction.RIGHT,
51 ),
52 ("init__q_rewind", ("0", "*", "*")): (
53     "init__q_rewind",
54     ("*", "*", "*"),
55     Direction.LEFT,
56 ),
57 ("init__q_rewind", ("1", "*", "*")): (
58     "init__q_rewind",
59     ("*", "*", "*"),
60     Direction.LEFT,
61 ),
62 ("init__q_rewind", ("B", "B", "B")): (
63     "compare_eq__q_seek_end",
64     ("B", "B", "B"),

```



```

65     Direction.RIGHT,
66 ),
67 # =====
68 # SUB-ROTINA 2: COMPARE_EQUALITY (SE FITA_1 == FITA_2)
69 # =====
70 ("compare_eq__q_seek_end", ("0", "*", "*")): (
71     "compare_eq__q_seek_end",
72     ("0", "*", "*"),
73     Direction.RIGHT,
74 ),
75 ("compare_eq__q_seek_end", ("1", "*", "*")): (
76     "compare_eq__q_seek_end",
77     ("1", "*", "*"),
78     Direction.RIGHT,
79 ),
80 ("compare_eq__q_seek_end", ("B", "B", "B")): (
81     "compare_eq__q_compare",
82     ("B", "B", "B"),
83     Direction.LEFT,
84 ),
85 ("compare_eq__q_compare", ("0", "0", "*")): (
86     "compare_eq__q_compare",
87     ("0", "0", "*"),
88     Direction.LEFT,
89 ),
90 ("compare_eq__q_compare", ("1", "1", "*")): (
91     "compare_eq__q_compare",
92     ("1", "1", "*"),
93     Direction.LEFT,
94 ),
95 ("compare_eq__q_compare", ("B", "B", "B")): (
96     "q_accept",
97     ("B", "B", "B"),
98     Direction.RIGHT,
99 ),
100 ("compare_eq__q_compare", ("0", "1", "*")): (
101     "compare_eq__q_notequal_rewind",
102     ("*", "*", "*"),
103     Direction.LEFT,
104 ),
105 ("compare_eq__q_compare", ("1", "0", "*")): (
106     "compare_eq__q_notequal_rewind",
107     ("*", "*", "*"),
108     Direction.LEFT,
109 ),
110 ("compare_eq__q_notequal_rewind", ("0", "*", "*")): (
111     "compare_eq__q_notequal_rewind",
112     ("*", "*", "*"),
113     Direction.LEFT,
114 ),
115 ("compare_eq__q_notequal_rewind", ("1", "*", "*")): (
116     "compare_eq__q_notequal_rewind",
117     ("*", "*", "*"),
118     Direction.LEFT,
119 ),
120 ("compare_eq__q_notequal_rewind", ("B", "B", "B")): (

```

```

121     "copy__q_copy",
122     ("B", "B", "B"),
123     Direction.RIGHT,
124 ),
125 # =====
126 # SUB-ROTINA 3: COPY_TAPE_1_TO_3 (FITA_3 <- FITA_1)
127 # =====
128 ("copy__q_copy", ("0", "*", "*")): (
129     "copy__q_copy",
130     ("*", "*", "0"),
131     Direction.RIGHT,
132 ),
133 ("copy__q_copy", ("1", "*", "*")): (
134     "copy__q_copy",
135     ("*", "*", "1"),
136     Direction.RIGHT,
137 ),
138 ("copy__q_copy", ("B", "B", "B")): (
139     "copy__q_rewind",
140     ("B", "B", "B"),
141     Direction.LEFT,
142 ),
143 ("copy__q_rewind", ("0", "*", "*")): (
144     "copy__q_rewind",
145     ("*", "*", "*"),
146     Direction.LEFT,
147 ),
148 ("copy__q_rewind", ("1", "*", "*")): (
149     "copy__q_rewind",
150     ("*", "*", "*"),
151     Direction.LEFT,
152 ),
153 ("copy__q_rewind", ("B", "B", "B")): (
154     "compare_ge__q_seek_end",
155     ("B", "B", "B"),
156     Direction.RIGHT,
157 ),
158 # =====
159 # SUB-ROTINA 4: COMPARE_GREATER_EQUAL (SE FITA_3 >= FITA_2)
160 # =====
161 ("compare_ge__q_seek_end", ("0", "*", "*")): (
162     "compare_ge__q_seek_end",
163     ("0", "*", "*"),
164     Direction.RIGHT,
165 ),
166 ("compare_ge__q_seek_end", ("1", "*", "*")): (
167     "compare_ge__q_seek_end",
168     ("1", "*", "*"),
169     Direction.RIGHT,
170 ),
171 ("compare_ge__q_seek_end", ("B", "B", "B")): (
172     "compare_ge__q_compare_len",
173     ("B", "B", "B"),
174     Direction.LEFT,
175 ),
176 ("compare_ge__q_compare_len", ("*", "0", "0")): (

```

```

177     "compare_ge__q_compare_len",
178     ("*", "0", "0"),
179     Direction.LEFT,
180 ),
181 ("compare_ge__q_compare_len", ("*", "1", "1")): (
182     "compare_ge__q_compare_len",
183     ("*", "1", "1"),
184     Direction.LEFT,
185 ),
186 ("compare_ge__q_compare_len", ("*", "0", "1")): (
187     "compare_ge__q_compare_len",
188     ("*", "0", "1"),
189     Direction.LEFT,
190 ),
191 ("compare_ge__q_compare_len", ("*", "1", "0")): (
192     "compare_ge__q_compare_len",
193     ("*", "1", "0"),
194     Direction.LEFT,
195 ),
196 ("compare_ge__q_compare_len", ("*", "B", "0")): (
197     "compare_ge__q_rewind_true",
198     ("*", "B", "0"),
199     Direction.LEFT,
200 ),
201 ("compare_ge__q_compare_len", ("*", "B", "1")): (
202     "compare_ge__q_rewind_true",
203     ("*", "B", "1"),
204     Direction.LEFT,
205 ),
206 ("compare_ge__q_compare_len", ("*", "0", "B")): (
207     "compare_ge__q_rewind_false",
208     ("*", "0", "B"),
209     Direction.LEFT,
210 ),
211 ("compare_ge__q_compare_len", ("*", "1", "B")): (
212     "compare_ge__q_rewind_false",
213     ("*", "1", "B"),
214     Direction.LEFT,
215 ),
216 ("compare_ge__q_compare_len", ("B", "B", "B")): (
217     "compare_ge__q_rewind_msb",
218     ("B", "B", "B"),
219     Direction.RIGHT,
220 ),
221 ("compare_ge__q_rewind_msb", ("0", "*", "*")): (
222     "compare_ge__q_rewind_msb",
223     ("0", "*", "*"),
224     Direction.LEFT,
225 ),
226 ("compare_ge__q_rewind_msb", ("1", "*", "*")): (
227     "compare_ge__q_rewind_msb",
228     ("1", "*", "*"),
229     Direction.LEFT,
230 ),
231 ("compare_ge__q_rewind_msb", ("B", "B", "B")): (
232     "compare_ge__q_compare_msb",

```

```

233     ("B", "B", "B"),
234     Direction.RIGHT,
235 ),
236 ("compare_ge__q_compare_msb", ("*", "0", "0")): (
237     "compare_ge__q_compare_msb",
238     ("*", "0", "0"),
239     Direction.RIGHT,
240 ),
241 ("compare_ge__q_compare_msb", ("*", "1", "1")): (
242     "compare_ge__q_compare_msb",
243     ("*", "1", "1"),
244     Direction.RIGHT,
245 ),
246 ("compare_ge__q_compare_msb", ("*", "0", "1")): (
247     "compare_ge__q_rewind_true",
248     ("*", "0", "1"),
249     Direction.LEFT,
250 ),
251 ("compare_ge__q_compare_msb", ("*", "1", "0")): (
252     "compare_ge__q_rewind_false",
253     ("*", "1", "0"),
254     Direction.LEFT,
255 ),
256 ("compare_ge__q_compare_msb", ("B", "B", "B")): (
257     "compare_ge__q_rewind_true",
258     ("B", "B", "B"),
259     Direction.LEFT,
260 ),
261 ("compare_ge__q_rewind_true", ("0", "*", "*")): (
262     "compare_ge__q_rewind_true",
263     ("*", "*", "*"),
264     Direction.LEFT,
265 ),
266 ("compare_ge__q_rewind_true", ("1", "*", "*")): (
267     "compare_ge__q_rewind_true",
268     ("*", "*", "*"),
269     Direction.LEFT,
270 ),
271 ("compare_ge__q_rewind_true", ("B", "B", "B")): (
272     "subtract__q0",
273     ("B", "B", "B"),
274     Direction.RIGHT,
275 ),
276 ("compare_ge__q_rewind_false", ("0", "*", "*")): (
277     "compare_ge__q_rewind_false",
278     ("*", "*", "*"),
279     Direction.LEFT,
280 ),
281 ("compare_ge__q_rewind_false", ("1", "*", "*")): (
282     "compare_ge__q_rewind_false",
283     ("*", "*", "*"),
284     Direction.LEFT,
285 ),
286 ("compare_ge__q_rewind_false", ("B", "B", "B")): (
287     "check_remainder__q_check",
288     ("B", "B", "B"),

```

```

289     Direction.RIGHT,
290 ),
291 # =====
292 # SUB-ROTINA 5: SUBTRACT (FITA_3 <- FITA_3 - FITA_2)
293 # =====
294 ("subtract__q0", ("0", "*", "*")): (
295     "subtract__q0",
296     ("*", "*", "*"),
297     Direction.RIGHT,
298 ),
299 ("subtract__q0", ("1", "*", "*")): (
300     "subtract__q0",
301     ("*", "*", "*"),
302     Direction.RIGHT,
303 ),
304 ("subtract__q0", ("B", "B", "B")): (
305     "subtract__q1",
306     ("B", "B", "B"),
307     Direction.LEFT,
308 ),
309 ("subtract__q1", ("*", "0", "0")): (
310     "subtract__q1",
311     ("*", "*", "0"),
312     Direction.LEFT,
313 ),
314 ("subtract__q1", ("*", "0", "1")): (
315     "subtract__q1",
316     ("*", "*", "1"),
317     Direction.LEFT,
318 ),
319 ("subtract__q1", ("*", "1", "1")): (
320     "subtract__q1",
321     ("*", "*", "0"),
322     Direction.LEFT,
323 ),
324 ("subtract__q1", ("*", "1", "0")): (
325     "subtract__q2",
326     ("*", "*", "1"),
327     Direction.LEFT,
328 ),
329 ("subtract__q2", ("*", "0", "1")): (
330     "subtract__q1",
331     ("*", "*", "0"),
332     Direction.LEFT,
333 ),
334 ("subtract__q2", ("*", "0", "0")): (
335     "subtract__q2",
336     ("*", "*", "1"),
337     Direction.LEFT,
338 ),
339 ("subtract__q2", ("*", "1", "1")): (
340     "subtract__q2",
341     ("*", "*", "1"),
342     Direction.LEFT,
343 ),
344 ("subtract__q2", ("*", "1", "0")): (

```

```

345     "subtract__q2",
346     ("*", "*", "0"),
347     Direction.LEFT,
348 ),
349 ("subtract__q1", ("B", "B", "B")): (
350     "subtract__q_rewind",
351     ("B", "B", "B"),
352     Direction.RIGHT,
353 ),
354 ("subtract__q2", ("B", "B", "B")): (
355     "subtract__q_rewind",
356     ("B", "B", "B"),
357     Direction.RIGHT,
358 ),
359 ("subtract__q_rewind", ("0", "*", "*")): (
360     "subtract__q_rewind",
361     ("*", "*", "*"),
362     Direction.LEFT,
363 ),
364 ("subtract__q_rewind", ("1", "*", "*")): (
365     "subtract__q_rewind",
366     ("*", "*", "*"),
367     Direction.LEFT,
368 ),
369 ("subtract__q_rewind", ("B", "B", "B")): (
370     "compare_ge__q_seek_end",
371     ("B", "B", "B"),
372     Direction.RIGHT,
373 ),
374 # =====
375 # SUB-ROTINA 6: CHECK_REMAINDER_IS_ZERO (SE FITA_3 == 0)
376 # =====
377 ("check_remainder__q_check", ("*", "*", "0")): (
378     "check_remainder__q_check",
379     ("*", "*", "*"),
380     Direction.RIGHT,
381 ),
382 ("check_remainder__q_check", ("*", "*", "1")): (
383     "check_remainder__q_rewind",
384     ("*", "*", "*"),
385     Direction.LEFT,
386 ),
387 ("check_remainder__q_check", ("B", "B", "B")): (
388     "q_reject",
389     ("B", "B", "B"),
390     Direction.RIGHT,
391 ),
392 ("check_remainder__q_rewind", ("0", "*", "*")): (
393     "check_remainder__q_rewind",
394     ("*", "*", "*"),
395     Direction.LEFT,
396 ),
397 ("check_remainder__q_rewind", ("1", "*", "*")): (
398     "check_remainder__q_rewind",
399     ("*", "*", "*"),
400     Direction.LEFT,

```

```

401 ),
402 ("check_remainder__q_rewind", ("B", "B", "B")): (
403     "cleanup__q_0",
404     ("B", "B", "B"),
405     Direction.RIGHT,
406 ),
407 # =====
408 # SUB-ROTINA 7: CLEANUP_TAPE_3 (Limpa Fita 3)
409 # =====
410 ("cleanup__q_0", ("*", "*", "0")): (
411     "cleanup__q_0",
412     ("*", "*", "B"),
413     Direction.RIGHT,
414 ),
415 ("cleanup__q_0", ("*", "*", "1")): (
416     "cleanup__q_0",
417     ("*", "*", "B"),
418     Direction.RIGHT,
419 ),
420 ("cleanup__q_0", ("B", "B", "B")): (
421     "cleanup__q_rewind",
422     ("B", "B", "B"),
423     Direction.LEFT,
424 ),
425 ("cleanup__q_rewind", ("0", "*", "*")): (
426     "cleanup__q_rewind",
427     ("*", "*", "*"),
428     Direction.LEFT,
429 ),
430 ("cleanup__q_rewind", ("1", "*", "*")): (
431     "cleanup__q_rewind",
432     ("*", "*", "*"),
433     Direction.LEFT,
434 ),
435 ("cleanup__q_rewind", ("B", "B", "B")): (
436     "increment__q_seek_end",
437     ("B", "B", "B"),
438     Direction.RIGHT,
439 ),
440 # =====
441 # SUB-ROTINA 8: INCREMENT_DIVISOR (FITA_2 <- FITA_2 + 1)
442 # =====
443 ("increment__q_seek_end", ("0", "*", "*")): (
444     "increment__q_seek_end",
445     ("*", "*", "*"),
446     Direction.RIGHT,
447 ),
448 ("increment__q_seek_end", ("1", "*", "*")): (
449     "increment__q_seek_end",
450     ("*", "*", "*"),
451     Direction.RIGHT,
452 ),
453 ("increment__q_seek_end", ("B", "B", "B")): (
454     "increment__q_add_carry",
455     ("B", "B", "B"),
456     Direction.LEFT,

```

```

457 ),
458 ("increment__q_add_carry", ("*", "0", "*")): (
459     "increment__q_rewind",
460     ("*", "1", "*"),
461     Direction.LEFT,
462 ),
463 ("increment__q_add_carry", ("*", "1", "*")): (
464     "increment__q_add_carry",
465     ("*", "0", "*"),
466     Direction.LEFT,
467 ),
468 ("increment__q_add_carry", ("*", "B", "*")): (
469     "increment__q_rewind",
470     ("*", "1", "*"),
471     Direction.LEFT,
472 ),
473 ("increment__q_rewind", ("0", "*", "*")): (
474     "increment__q_rewind",
475     ("*", "*", "*"),
476     Direction.LEFT,
477 ),
478 ("increment__q_rewind", ("1", "*", "*")): (
479     "increment__q_rewind",
480     ("*", "*", "*"),
481     Direction.LEFT,
482 ),
483 ("increment__q_rewind", ("B", "B", "B")): (
484     "compare_eq__q_seek_end",
485     ("B", "B", "B"),
486     Direction.RIGHT,
487 )

```

**Código 7. Definição da função  $\delta$  da Máquina de Turing para teste de primalidade**

Nestas próximas subseções, as sub-rotinas são explicadas em maior detalhe. Junto a cada uma delas, é disponibilizada a definição da função  $\delta$  para aquela sub-rotina em questão. Note que a máquina objeto deste relatório é composta pelo funcionamento em conjunto de todas estas sub-rotinas.

### 3.2.1. Sub-rotina de inicialização

A sub-rotina de inicialização copia o valor  $2_{10}$  para a fita 2. Isso é o ponto de partida para que possamos, nos próximos passos, ir incrementando o divisor. Além disso, essa e as demais sub-rotinas possuem sempre uma **etapa de rebobinação**, para que o controle seja devolvido para a próxima sub-rotina sempre estando na posição de índice zero. Por isso observamos o estado  $q_{rewind}$ , que volta a fita até a extremidade esquerda e então passa o fluxo de controle para a sub-rotina seguinte.

$$\delta(\text{init\_q0}, (0, *, *)) = (\text{init\_q0}, (*, 0, *), R)$$

$$\delta(\text{init\_q0}, (1, *, *)) = (\text{init\_q0}, (*, 0, *), R)$$

$$\delta(\text{init\_q0}, (B, B, B)) = (\text{init\_q\_write0}, (B, B, B), L)$$



$$\begin{aligned}
\delta(\text{init\_q\_write0}, (*, *, *)) &= (\text{init\_q\_write1}, (*, 0, *), L) \\
\delta(\text{init\_q\_write1}, (*, *, *)) &= (\text{init\_q\_fill\_zeros}, (*, 1, *), L) \\
\delta(\text{init\_q\_fill\_zeros}, (0, *, *)) &= (\text{init\_q\_fill\_zeros}, (*, 0, *), L) \\
\delta(\text{init\_q\_fill\_zeros}, (1, *, *)) &= (\text{init\_q\_fill\_zeros}, (*, 0, *), L) \\
\delta(\text{init\_q\_fill\_zeros}, (B, B, B)) &= (\text{init\_q\_rewind}, (B, B, B), R) \\
\delta(\text{init\_q\_rewind}, (0, *, *)) &= (\text{init\_q\_rewind}, (*, *, *), L) \\
\delta(\text{init\_q\_rewind}, (1, *, *)) &= (\text{init\_q\_rewind}, (*, *, *), L) \\
\delta(\text{init\_q\_rewind}, (B, B, B)) &= (\text{compare\_eq\_q\_seek\_end}, (B, B, B), R)
\end{aligned}$$

### 3.2.2. Sub-rotina de comparação de igualdade

A próxima sub-rotina fica responsável por comparar a fita 1 com a fita 2. Se o divisor (fita 2) for igual ao número de entrada (fita 1), significa que o número não foi divisível por nenhum valor menor que ele, caracterizando-o como primo. Nesse caso, a máquina entra no estado de aceitação. Caso contrário, o fluxo segue para a próxima sub-rotina.

$$\begin{aligned}
\delta(\text{compare\_eq\_q\_seek\_end}, (0, *, *)) &= (\text{compare\_eq\_q\_seek\_end}, (0, *, *), R) \\
\delta(\text{compare\_eq\_q\_seek\_end}, (1, *, *)) &= (\text{compare\_eq\_q\_seek\_end}, (1, *, *), R) \\
\delta(\text{compare\_eq\_q\_seek\_end}, (B, B, B)) &= (\text{compare\_eq\_q\_compare}, (B, B, B), L) \\
\delta(\text{compare\_eq\_q\_compare}, (0, 0, *)) &= (\text{compare\_eq\_q\_compare}, (0, 0, *), L) \\
\delta(\text{compare\_eq\_q\_compare}, (1, 1, *)) &= (\text{compare\_eq\_q\_compare}, (1, 1, *), L) \\
\delta(\text{compare\_eq\_q\_compare}, (B, B, B)) &= (\text{q\_accept}, (B, B, B), R) \\
\delta(\text{compare\_eq\_q\_compare}, (0, 1, *)) &= (\text{compare\_eq\_q\_notequal\_rewind}, (*, *, *), L) \\
\delta(\text{compare\_eq\_q\_compare}, (1, 0, *)) &= (\text{compare\_eq\_q\_notequal\_rewind}, (*, *, *), L) \\
\delta(\text{compare\_eq\_q\_notequal\_rewind}, (0, *, *)) &= (\text{compare\_eq\_q\_notequal\_rewind}, (*, *, *), L) \\
\delta(\text{compare\_eq\_q\_notequal\_rewind}, (1, *, *)) &= (\text{compare\_eq\_q\_notequal\_rewind}, (*, *, *), L) \\
\delta(\text{compare\_eq\_q\_notequal\_rewind}, (B, B, B)) &= (\text{copy\_q\_copy}, (B, B, B), R)
\end{aligned}$$

### 3.2.3. Sub-rotina de cópia

A sub-rotina de cópia é responsável por transcrever o conteúdo da Fita 1 para a Fita 3. Este passo é crucial para preservar o valor original do dividendo (o número de entrada), permitindo que as operações de subtração sejam realizadas em uma cópia, sem alterar o valor que será testado nas iterações seguintes.

$$\begin{aligned}\delta(\text{copy\_q\_copy}, (0, *, *)) &= (\text{copy\_q\_copy}, (*, *, 0), R) \\ \delta(\text{copy\_q\_copy}, (1, *, *)) &= (\text{copy\_q\_copy}, (*, *, 1), R) \\ \delta(\text{copy\_q\_copy}, (B, B, B)) &= (\text{copy\_q\_rewind}, (B, B, B), L) \\ \delta(\text{copy\_q\_rewind}, (0, *, *)) &= (\text{copy\_q\_rewind}, (*, *, *), L) \\ \delta(\text{copy\_q\_rewind}, (1, *, *)) &= (\text{copy\_q\_rewind}, (*, *, *), L) \\ \delta(\text{copy\_q\_rewind}, (B, B, B)) &= (\text{compare\_ge\_q\_seek\_end}, (B, B, B), R)\end{aligned}$$

### 3.2.4. Sub-rotina de comparação "maior ou igual que"

Esta sub-rotina implementa a comparação  $T_3 \geq T_2$ . Ela compara os bits do mais significativo para o menos significativo. O resultado desta comparação determina se o loop de subtrações sucessivas deve continuar. Se  $T_3 < T_2$ , a divisão pelo divisor atual termina.

$$\begin{aligned}\delta(\text{compare\_ge\_q\_seek\_end}, (0, *, *)) &= (\text{compare\_ge\_q\_seek\_end}, (0, *, *), R) \\ \delta(\text{compare\_ge\_q\_seek\_end}, (1, *, *)) &= (\text{compare\_ge\_q\_seek\_end}, (1, *, *), R) \\ \delta(\text{compare\_ge\_q\_seek\_end}, (B, B, B)) &= (\text{compare\_ge\_q\_compare\_len}, (B, B, B), L) \\ \delta(\text{compare\_ge\_q\_compare\_len}, (*, 0, 0)) &= (\text{compare\_ge\_q\_compare\_len}, (*, 0, 0), L) \\ \delta(\text{compare\_ge\_q\_compare\_len}, (*, 1, 1)) &= (\text{compare\_ge\_q\_compare\_len}, (*, 1, 1), L) \\ \delta(\text{compare\_ge\_q\_compare\_len}, (*, 0, 1)) &= (\text{compare\_ge\_q\_compare\_len}, (*, 0, 1), L) \\ \delta(\text{compare\_ge\_q\_compare\_len}, (*, 1, 0)) &= (\text{compare\_ge\_q\_compare\_len}, (*, 1, 0), L) \\ \delta(\text{compare\_ge\_q\_compare\_len}, (*, B, 0)) &= (\text{compare\_ge\_q\_rewind\_true}, (*, B, 0), L) \\ \delta(\text{compare\_ge\_q\_compare\_len}, (*, B, 1)) &= (\text{compare\_ge\_q\_rewind\_true}, (*, B, 1), L) \\ \delta(\text{compare\_ge\_q\_compare\_len}, (*, 0, B)) &= (\text{compare\_ge\_q\_rewind\_false}, (*, 0, B), L) \\ \delta(\text{compare\_ge\_q\_compare\_len}, (*, 1, B)) &= (\text{compare\_ge\_q\_rewind\_false}, (*, 1, B), L) \\ \delta(\text{compare\_ge\_q\_compare\_len}, (B, B, B)) &= (\text{compare\_ge\_q\_rewind\_msb}, (B, B, B), R) \\ \delta(\text{compare\_ge\_q\_rewind\_msb}, (0, *, *)) &= (\text{compare\_ge\_q\_rewind\_msb}, (0, *, *), L) \\ \delta(\text{compare\_ge\_q\_rewind\_msb}, (1, *, *)) &= (\text{compare\_ge\_q\_rewind\_msb}, (1, *, *), L) \\ \delta(\text{compare\_ge\_q\_rewind\_msb}, (B, B, B)) &= (\text{compare\_ge\_q\_compare\_msb}, (B, B, B), R) \\ \delta(\text{compare\_ge\_q\_compare\_msb}, (*, 0, 0)) &= (\text{compare\_ge\_q\_compare\_msb}, (*, 0, 0), R)\end{aligned}$$

$$\begin{aligned}
\delta(\text{compare\_ge\_q\_compare\_msb}, (*, 1, 1)) &= (\text{compare\_ge\_q\_compare\_msb}, (*, 1, 1), R) \\
\delta(\text{compare\_ge\_q\_compare\_msb}, (*, 0, 1)) &= (\text{compare\_ge\_q\_rewind\_true}, (*, 0, 1), L) \\
\delta(\text{compare\_ge\_q\_compare\_msb}, (*, 1, 0)) &= (\text{compare\_ge\_q\_rewind\_false}, (*, 1, 0), L) \\
\delta(\text{compare\_ge\_q\_compare\_msb}, (B, B, B)) &= (\text{compare\_ge\_q\_rewind\_true}, (B, B, B), L) \\
\delta(\text{compare\_ge\_q\_rewind\_true}, (0, *, *)) &= (\text{compare\_ge\_q\_rewind\_true}, (*, *, *), L) \\
\delta(\text{compare\_ge\_q\_rewind\_true}, (1, *, *)) &= (\text{compare\_ge\_q\_rewind\_true}, (*, *, *), L) \\
\delta(\text{compare\_ge\_q\_rewind\_true}, (B, B, B)) &= (\text{subtract\_q0}, (B, B, B), R) \\
\delta(\text{compare\_ge\_q\_rewind\_false}, (0, *, *)) &= (\text{compare\_ge\_q\_rewind\_false}, (*, *, *), L) \\
\delta(\text{compare\_ge\_q\_rewind\_false}, (1, *, *)) &= (\text{compare\_ge\_q\_rewind\_false}, (*, *, *), L) \\
\delta(\text{compare\_ge\_q\_rewind\_false}, (B, B, B)) &= (\text{check\_remainder\_q\_check}, (B, B, B), R)
\end{aligned}$$

### 3.2.5. Sub-rotina de subtração

A sub-rotina de subtração realiza a operação  $T_3 \leftarrow T_3 - T_2$  utilizando o método de complemento de dois. Este é o núcleo do laço de divisão, sendo executada repetidamente enquanto o valor na Fita 3 for maior ou igual ao da Fita 2.

$$\begin{aligned}
\delta(\text{subtract\_q0}, (0, *, *)) &= (\text{subtract\_q0}, (*, *, *), R) \\
\delta(\text{subtract\_q0}, (1, *, *)) &= (\text{subtract\_q0}, (*, *, *), R) \\
\delta(\text{subtract\_q0}, (B, B, B)) &= (\text{subtract\_q1}, (B, B, B), L) \\
\delta(\text{subtract\_q1}, (*, 0, 0)) &= (\text{subtract\_q1}, (*, *, 0), L) \\
\delta(\text{subtract\_q1}, (*, 0, 1)) &= (\text{subtract\_q1}, (*, *, 1), L) \\
\delta(\text{subtract\_q1}, (*, 1, 1)) &= (\text{subtract\_q1}, (*, *, 0), L) \\
\delta(\text{subtract\_q1}, (*, 1, 0)) &= (\text{subtract\_q2}, (*, *, 1), L) \\
\delta(\text{subtract\_q2}, (*, 0, 1)) &= (\text{subtract\_q1}, (*, *, 0), L) \\
\delta(\text{subtract\_q2}, (*, 0, 0)) &= (\text{subtract\_q2}, (*, *, 1), L) \\
\delta(\text{subtract\_q2}, (*, 1, 1)) &= (\text{subtract\_q2}, (*, *, 1), L) \\
\delta(\text{subtract\_q2}, (*, 1, 0)) &= (\text{subtract\_q2}, (*, *, 0), L) \\
\delta(\text{subtract\_q1}, (B, B, B)) &= (\text{subtract\_q\_rewind}, (B, B, B), R) \\
\delta(\text{subtract\_q2}, (B, B, B)) &= (\text{subtract\_q\_rewind}, (B, B, B), R) \\
\delta(\text{subtract\_q\_rewind}, (0, *, *)) &= (\text{subtract\_q\_rewind}, (*, *, *), L) \\
\delta(\text{subtract\_q\_rewind}, (1, *, *)) &= (\text{subtract\_q\_rewind}, (*, *, *), L) \\
\delta(\text{subtract\_q\_rewind}, (B, B, B)) &= (\text{compare\_ge\_q\_seek\_end}, (B, B, B), R)
\end{aligned}$$

### 3.2.6. Sub-rotina de verificação do resto

Após a conclusão do loop de subtrações, esta sub-rotina verifica se o valor restante na Fita 3 é zero. Para isso, ela percorre a fita e, se encontrar apenas símbolos '0', conclui que a divisão foi exata. Neste caso, o número de entrada não é primo, e a máquina transita para o estado de rejeição.

$$\delta(\text{check\_remainder\_q\_check}, (*, *, 0)) = (\text{check\_remainder\_q\_check}, (*, *, *), R)$$

$$\delta(\text{check\_remainder\_q\_check}, (*, *, 1)) = (\text{check\_remainder\_q\_rewind}, (*, *, *), L)$$

$$\delta(\text{check\_remainder\_q\_check}, (B, B, B)) = (\text{q\_reject}, (B, B, B), R)$$

$$\delta(\text{check\_remainder\_q\_rewind}, (0, *, *)) = (\text{check\_remainder\_q\_rewind}, (*, *, *), L)$$

$$\delta(\text{check\_remainder\_q\_rewind}, (1, *, *)) = (\text{check\_remainder\_q\_rewind}, (*, *, *), L)$$

$$\delta(\text{check\_remainder\_q\_rewind}, (B, B, B)) = (\text{cleanup\_q\_0}, (B, B, B), R)$$

### 3.2.7. Sub-rotina de limpeza

A sub-rotina de limpeza é executada quando a divisão não é exata. Sua única função é apagar o conteúdo da Fita 3, preenchendo-a com o símbolo branco. Isso prepara a fita para a próxima iteração do loop principal, onde ela receberá uma nova cópia do número de entrada.

$$\delta(\text{cleanup\_q\_0}, (*, *, 0)) = (\text{cleanup\_q\_0}, (*, *, B), R)$$

$$\delta(\text{cleanup\_q\_0}, (*, *, 1)) = (\text{cleanup\_q\_0}, (*, *, B), R)$$

$$\delta(\text{cleanup\_q\_0}, (B, B, B)) = (\text{cleanup\_q\_rewind}, (B, B, B), L)$$

$$\delta(\text{cleanup\_q\_rewind}, (0, *, *)) = (\text{cleanup\_q\_rewind}, (*, *, *), L)$$

$$\delta(\text{cleanup\_q\_rewind}, (1, *, *)) = (\text{cleanup\_q\_rewind}, (*, *, *), L)$$

$$\delta(\text{cleanup\_q\_rewind}, (B, B, B)) = (\text{increment\_q\_seek\_end}, (B, B, B), R)$$

### 3.2.8. Sub-rotina de incremento

Por fim, a sub-rotina de incremento adiciona 1 ao valor do divisor na Fita 2. Após o incremento, o controle retorna para a sub-rotina de comparação de igualdade, reiniciando o ciclo de verificação com o próximo divisor.

$$\delta(\text{increment\_q\_seek\_end}, (0, *, *)) = (\text{increment\_q\_seek\_end}, (*, *, *), R)$$

$$\delta(\text{increment\_q\_seek\_end}, (1, *, *)) = (\text{increment\_q\_seek\_end}, (*, *, *), R)$$

$$\delta(\text{increment\_q\_seek\_end}, (B, B, B)) = (\text{increment\_q\_add\_carry}, (B, B, B), L)$$

$$\begin{aligned}
\delta(\text{increment\_q\_add\_carry}, (*, 0, *)) &= (\text{increment\_q\_rewind}, (*, 1, *), L) \\
\delta(\text{increment\_q\_add\_carry}, (*, 1, *)) &= (\text{increment\_q\_add\_carry}, (*, 0, *), L) \\
\delta(\text{increment\_q\_add\_carry}, (*, B, *)) &= (\text{increment\_q\_rewind}, (*, 1, *), L) \\
\delta(\text{increment\_q\_rewind}, (0, *, *)) &= (\text{increment\_q\_rewind}, (*, *, *), L) \\
\delta(\text{increment\_q\_rewind}, (1, *, *)) &= (\text{increment\_q\_rewind}, (*, *, *), L) \\
\delta(\text{increment\_q\_rewind}, (B, B, B)) &= (\text{compare\_eq\_q\_seek\_end}, (B, B, B), R)
\end{aligned}$$

#### 4. Testes

A máquina foi submetida a um conjunto de valores primos conhecidos e a um conjunto de números que sabe-se não terem esta propriedade. A cada execução, foi mensurado o tempo de execução e a quantidade de passos tomada pela máquina. Além disso, os resultados foram comparados com a função `IS_PRIME_FN`, que é uma função simples para determinar se um valor é ou não primo.

Nos testes, disponíveis por completo no repositório da implementação [da Luz 2025], a Máquina de Turing ofereceu uma performance muito pior. Todavia, em nenhum dos cenários de teste, sua saída diferiu da função `IS_PRIME_FN`. Isso mostra que o comportamento da Máquina de Turing está alinhado com as expectativas, ao menos detre os valores que foram testados.

A fim de exemplificação, abaixo está o final do resultado da execução para o valor  $7_{10}$ .

```

1 A máquina encerrou sua execução.
2 Estado final: q_accept
3 Conteúdo da fita 1: 111
4 Conteúdo da fita 2: 111
5 Conteúdo da fita 3: (vazio)
6 Posição da cabeça: 0
7 Contagem de passos: 456
8 ==| Resultado da Máquina de Turing |==
9 Resultado: q_accept, Passos: 456
10 Tempo gasto: 0.011638 segundos
11
12 Para fins de comparação, a função is_prime_fn também será executada.
13
14 ==| Resultado da função is_prime_fn |==
15 Resultado da função: True, Passos: 2
16 Tempo gasto na função: 0.000010 segundos
17
18 ==| Comparação de Desempenho |==
19 A máquina de Turing levou 0.011638 segundos, enquanto a função levou
    0.000010 segundos.
20 A máquina de Turing foi 116280.00% mais lenta que a função.
21 A máquina de Turing executou 456 passos, enquanto a função executou 2
    passos.
22 ==| Fim da execução |==

```

#### Código 8. Resultado para a entrada 7

Como outro exemplo, aqui está o resultado para  $281_{10}$ .

```

1 A máquina encerrou sua execução.
2 Estado final: q_accept
3 Conteúdo da fita 1: 100011001
4 Conteúdo da fita 2: 100011001
5 Conteúdo da fita 3: (vazio)
6 Posição da cabeça: 0
7 Contagem de passos: 98850
8 ==| Resultado da Máquina de Turing |==
9 Resultado: q_accept, Passos: 98850
10 Tempo gasto: 0.985857 segundos
11
12 Para fins de comparação, a função is_prime_fn também será executada.
13
14 ==| Resultado da função is_prime_fn |==
15 Resultado da função: True, Passos: 16
16 Tempo gasto na função: 0.000009 segundos
17
18 ==| Comparação de Desempenho |==
19 A máquina de Turing levou 0.985857 segundos, enquanto a função levou
    0.000009 segundos.
20 A máquina de Turing foi 10953866.67% mais lenta que a função.
21 A máquina de Turing executou 98850 passos, enquanto a função executou
    16 passos.
22 ==| Fim da execução |==

```

#### **Código 9. Resultado para a entrada 281**

## **5. Conclusão**

Este artigo mostrou o processo de implementação de uma Máquina de Turing em Python para avaliar se um determinado valor é ou não primo. Os testes conduzidos ao final mostraram o correto comportamento da máquina, e mecanismos visuais foram implantados para tornar seu resultado mais didático e fácil de compreender.

Em termos de performance, o comportamento da Máquina de Turing foi muito inferior ao da função simples, que já possuía as primitivas necessárias para testar a primalidade de maneira simples. Todavia, isso não descarta de maneira alguma a importância deste modelo teórico, que segue sendo extremamente relevante para a computação mesmo depois de quase um século de sua concepção.

## **References**

- Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2009). *Introduction to Algorithms*. The MIT Press.
- da Luz, V. Q. (2025). Implementação de uma máquina de turing em python. Repositório de código. Disponível em: <https://github.com/Lvght/turing-machine>.
- Knuth, D. E. (1997). *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. Addison-Wesley Professional.
- Rivest, R. L., Shamir, A., and Adleman, L. (1978). A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126.
- Souza, W. L. d. (2025). Aula 16: Máquinas de turing (mt) - 9.1 definição formal de mt. Material de aula, Disciplina de Teoria da Computação.

Turing, A. M. (1937). On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, s2-42(1):230–265.