Computabilidade e Complexidade de Algoritmos

Fabiano Magrin da Costa Vieira
Ciência da Computação – Unifran

>> Computabilidade:

O termo "computável" foi proposto por Alan Turing para designar a totalidade de números reais cuja expansão decimal pode ser calculada em tempo finito, através de recursos finitos. No mesmo artigo, Turing argumenta que este conjunto corresponde ao conjunto de funções ou predicados computáveis. A proposta de Turing consistia em identificar os possíveis processos envolvidos no ato de "computar um número". Para tanto, Turing definiu um artefato teórico, que chamou de "máquina de computar", de maneira que, todo número cuja expansão decimal pudesse ser obtida a partir de operações da máquina seria chamado de "número computado pela máquina".

Ao identificar os processos necessários para computar um número, Turing estaria indiretamente, identificando o conjunto de números (funções, predicados, etc) computáveis e, consequentemente, respondendo a seguinte pergunta: O que pode ser efetuado pela máquina de computar?

Na tentativa de definir o "computável", Turing se deparou com o seguinte problema: os números computados pela máquina realmente formam aquele conjunto que, intuitivamente, consideramos "computável"? Para responder a tal questão com precisão, Turing precisaria provar matematicamente que (→) todos os números que consideramos "computáveis" podem ser obtidos pela máquina. Desta maneira, a máquina computaria necessariamente todos os "números computáveis". Por outro lado seria também necessária a prova matemática de que (←) todos os números que a máquina computa são considerados por nós "números computáveis". Estaria então provado que a máquina não computaria nada além do que "números computáveis". (artigo de 1936).

Ref.: http://www.ic.uff.br/isabel/pdf/Computabilidade,%20Hist%C3%B3ria%20e%20Matem%C3%A1tica.pdf

>> Objetivos:

- Investigar a possibilidade de existir ou não algoritmos que solucionem determinada classe de problemas.

- Verificar os limites da computação e, também, o que pode realmente ser implementado e solucionado por meio de um computador.

- O objetivo está centrado na possibilidade de construção desses algoritmos.

?? Questão:

>> O computador pode resolver quais problemas ?

- **Como verificar essa questão ?**

> **Entendendo "computabilidade" como conceituação de tudo que pode ser realizado por computador a discussão a respeito deste tema é fundamental aos estudantes de Ciência da Computação. A Computabilidade vem mostrar ao profissional em formação dois importantes resultados (negativos) com relação ao computador. O primeiro resultado é consequência do trabalho de Gödel** (filósofo e matemático)**, e consiste na constatação de que nem tudo se pode resolver através do computador. O segundo resultado negativo, abordado por Turing, aponta a impossibilidade de caracterizar a classe de problemas computáveis. Informalmente, poderíamos dizer que sabemos que existem problemas não resolvíveis através de computadores, mas não sabemos exatamente quais são. É através do estudo da Computabilidade que o profissional da área de computação adquire pleno conhecimento da real capacidade dos computadores, e passa a lidar com seu instrumento de trabalho de maneira realística, sem mitos.**
>
> Ref.: http://www.ic.uff.br/isabel/pdf/Computabilidade,%20Hist%C3%B3ria%20e%20Matem%C3%A1tica.pdf

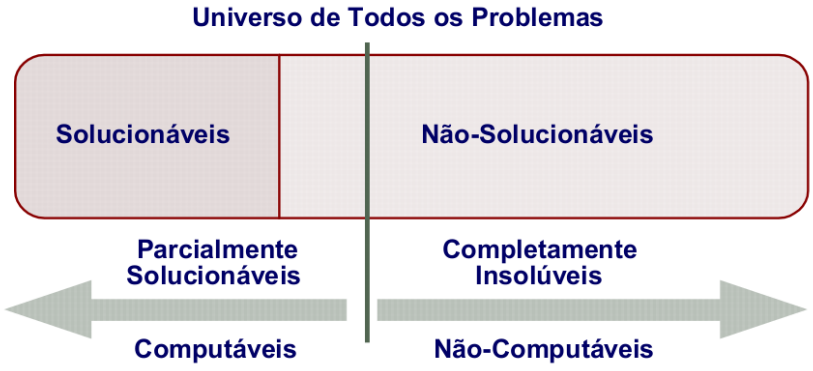
>> Abordagem:

"Focada nos problemas com respostas do tipo Sim/Não (ou Problemas de decisão). A vantagem de tal abordagem é que a verificação da solubilidade de um problema pode ser associada às condições de **aceitação**/**rejeição** de uma Máquina Universal às respostas Sim/Não, respectivamente."

Ref.: http://usuarios.upf.br/~mcpinto/teoria/teoria-computabilidade.pdf

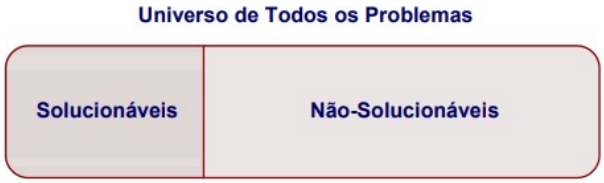
>> Classes de Solucionabilidade de Problemas

Ref.: http://www2.fct.unesp.br/docentes/dmec/olivete/tc/arquivos/Aula9.pdf



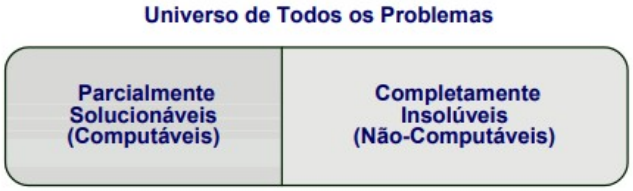
- Um problema é denominado Solucionável se existe um algoritmo (Máquina Universal) que solucione o problema e sempre para (**aceitação** – Sim ou **rejeição** – Não) com qualquer entrada.

- Um problema é denominado Não-Solucionável se não existe um algoritmo (Máquina Universal) que solucione o problema e sempre para com qualquer entrada.



- Um problema é denominado Parcialmente Solucionável ou Computável se existe um algoritmo (Máquina Universal) que sempre para se a resposta for afirmativa (**aceitação** – Sim), porém, se a resposta esperada for negativa, o algoritmo pode parar (**rejeição** – Não) ou permanecer processando indefinidamente (loop infinito).

- Um problema é denominado Completamente Insolúvel ou Não-Computável se não existe um algoritmo (Máquina Universal) que solucione o problema tal que sempre para quando a resposta é afirmativa (**aceitação** – Sim).



>> Relação entre as classes:

- A classe dos problemas Parcialmente Solucionáveis contém a classe dos problemas Solucionáveis e parte da classe dos problemas Não-Solucionáveis.

- Há problemas Não-Solucionáveis que possuem soluções parciais (aceitação/rejeição).

- Os problemas Completamente Insolúveis não possuem solução total nem parcial.

>> Porque estudar problemas não-solucionáveis?

Ref.: http://www2.fct.unesp.br/docentes/dmec/olivete/tc/arquivos/Aula9.pdf

- Verificar as capacidades e limitações dos computadores.

- Evitar a pesquisa de soluções inexistentes – problemas que não podem ser resolvidos computacionalmente (Indecidíveis).

- Para verificar que outros problemas também são insolúveis (princípio da redução).

>> **Objetivo**: VERIFICAR O QUE É QUE UM COMPUTADOR PODE FAZER!

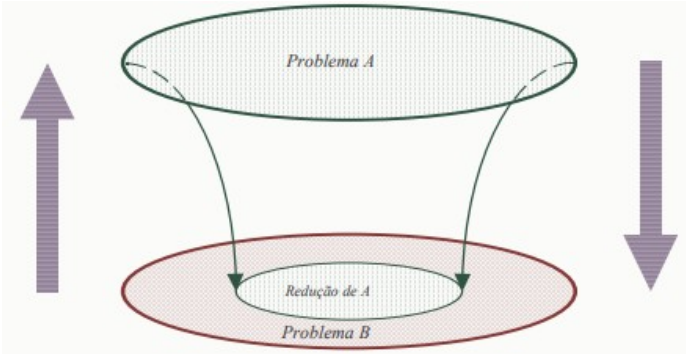
>> Exemplos clássicos de Problemas Não-Solucionáveis:

- Detector Universal de Loops: não existe um algoritmo genérico capaz de verificar se um programa vai parar ou não para uma determinada entrada. Esse problema é conhecido universalmente como **Problema da Parada**.

- Equivalência de Compiladores: não existe algoritmo genérico capaz de comparar (verificar se são equivalentes, se reconhecem a mesma linguagem) dois compiladores de linguagens livres de contexto (LLC) e parar durante o processamento.

>> Princípio da Redução:

Solucionável



Insolúvel

Sejam A e B dois problemas de decisão. Suponha que é possível modificar (reduzir) A de tal forma que ele se porte como um caso de B. Portanto:

- Se A é Não-Solucionável (Não-Computável), então, como A é um caso de B, conclui-se que B também é Não-Solucionável.

- Se B é Solucionável (ou Parcialmente Solucionável), então, como A é um caso de B, conclui-se que A também é Solucionável (ou Parcialmente Solucionável).

>> Máquina Universal:

- Questões que NÃO devem ser consideradas na investigação da computabilidade:
  - Limitações reais de uma implementação, como por exemplo: tamanho de espaço de endereços, tamanho da memória principal, questões de tempo e outros.

- Em estudos iniciados pelo matemático David Hilbert (início do século XX) foram propostos formalismos para a verificação da computabilidade, sendo os mais importantes:

  - Máquina de Post.
  - Máquina com Pilhas. → **Máquinas Universais**
  - Máquina de Turing. ← **Estudado em nossa disciplina.**

- Estas máquinas podem simular computadores reais e não apresentam as limitações citadas anteriormente, logo podem ser utilizadas para verificar o que um dispositivo de computação pode calcular em um determinado tempo (execução em tempo finito) → Decidibilidade.

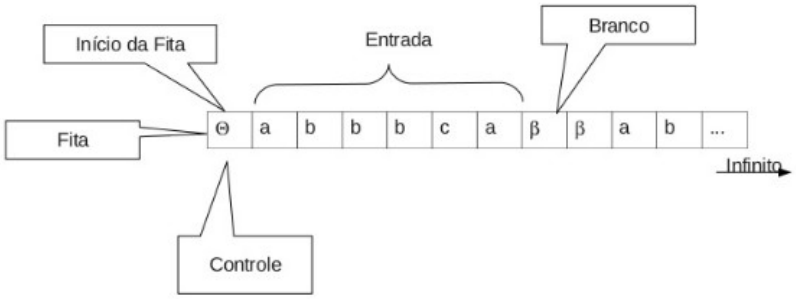
>> Tese de Church-Turing:

Ref.: http://www2.fct.unesp.br/docentes/dmec/olivete/tc/arquivos/Aula9.pdf

- **Tese**: "Qualquer computação que pode ser executada por meios mecânicos pode ser executada por uma Máquina de Turing".

- Principais razões pelas quais a maioria dos investigadores demonstram que esta tese é verdadeira:
  - As MT's são tão gerais que podem simular qualquer computação.
  - Nunca ninguém descobriu um modelo algorítmico mais geral que as MT's.

>> Máquina de Turing (Turing Machine):



- **Definição formal:**

MT = (E, A, V, f, q_0, F, β), onde:

E: conjunto finito de estados.
A: conjunto finito de símbolos (alfabeto da fita).
V: conjunto finito de símbolos (alfabeto de entrada - $V \in A$).
f: são as funções de transição de estados.

f: E, A → A, {R, L}, E

q_0: estado inicial ($q_0 \in E$).
F: conjunto finito de estados finais ($F \in E$).
β: símbolo do branco (depois da entrada, há brancos infinitamente).
No simulador é o caractere _.

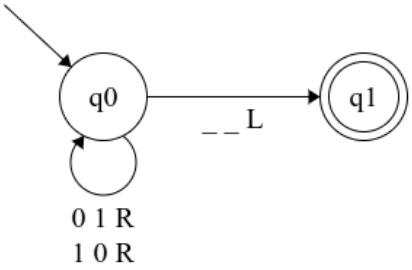
- **Exemplo**: NOT binário.

  E: 000111...
  S: 111000...

  - Representação por funções:

    f (q0, 0) = (1, R, q0)
    f (q0, 1) = (0, R, q0)
    f (q0, _) = (_, L, q1) OBS.: _ é o branco (β).

  - Representação por dígrafo:



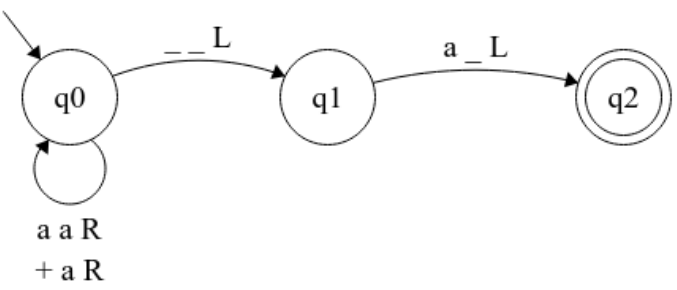
q1 é um estado final (pertence a F).

- **OBS.:** a MT para quando não houver função a ser executada no estado atual ou quando fizer um movimento à esquerda do símbolo inicial.

>> Exercícios:

1. Construa uma MT para somar dois números no formato unário, seguindo o exemplo abaixo:

E: aaa+aaaa . . . Onde, . . . são brancos infinitos à direita.
S: aaaaaaa . . .



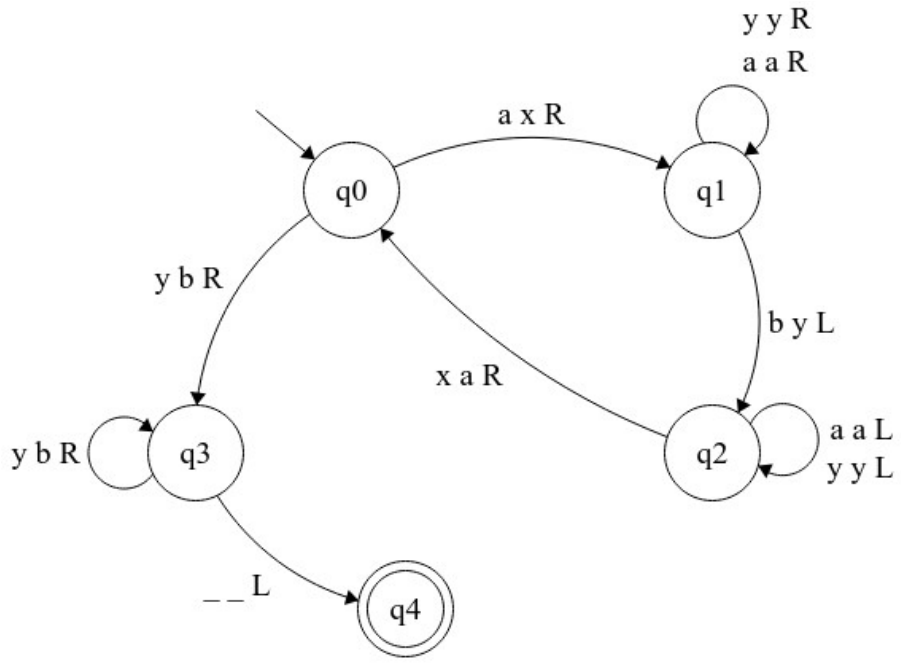
f (q0, a) = (a, R, q0)
f (q0, +) = (a, R, q0)
f (q0, _) = (_, L, q1)
f (q1, a) = (_, L, q2)

F = {q2} (est. final)

2. Exemplo de MT para reconhecer a linguagem:

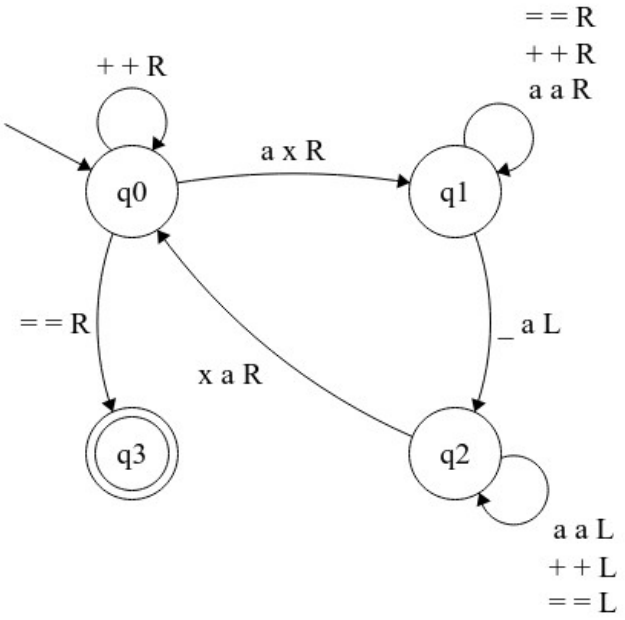
L = { $\mathbf{a}^n \mathbf{b}^n$, n >= 1 }

>> Exemplos de sentenças (ou cadeias): **ab, aabb, aaabbb, aaaabbbb, …**



3. Construa uma MT para somar dois números no formato unário, seguindo o exemplo abaixo:

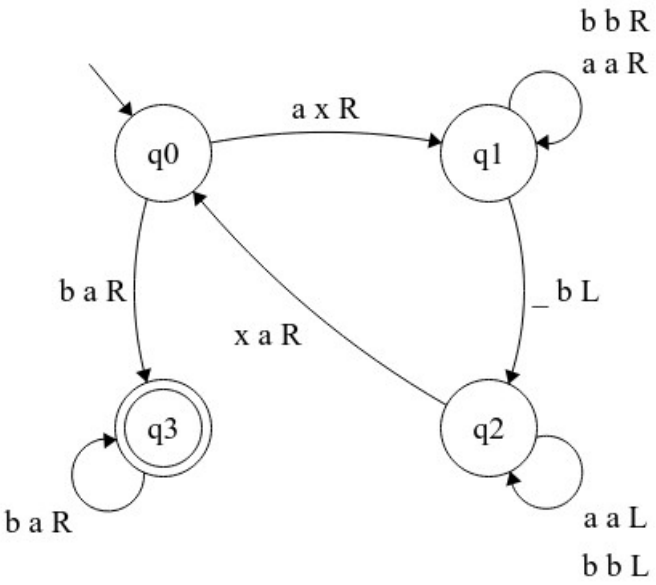
E: aaa+aa=
S: aaa+aa=aaaaa



4. Construa uma MT para dobrar uma cadeia de a's na entrada (fita), ou seja, multiplicar por 2.

E: aaa

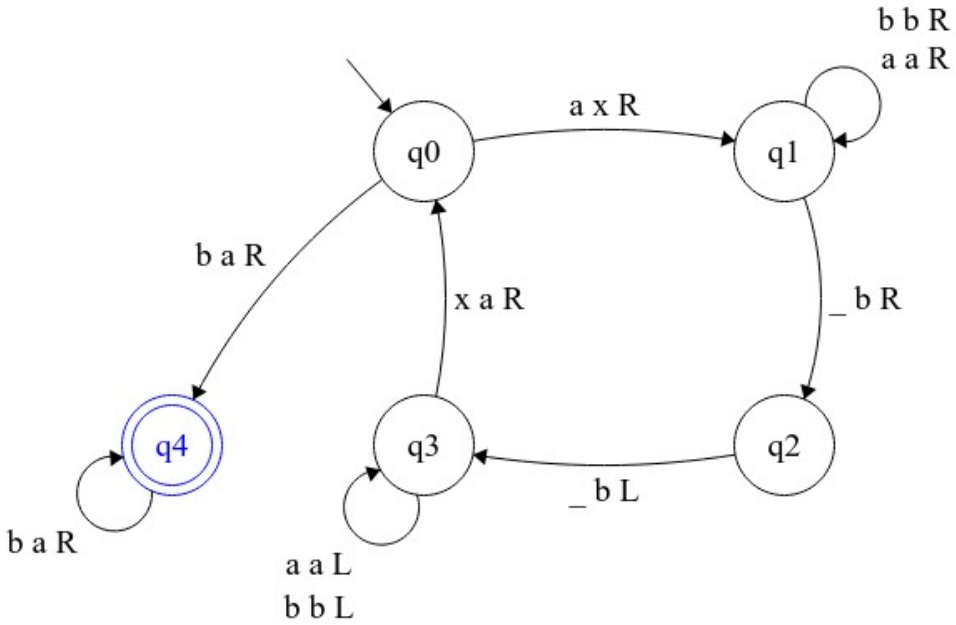
S: aaaaaa



5. Construa uma MT para triplicar uma cadeia de a's na entrada (fita), ou seja, multiplicar por 3.

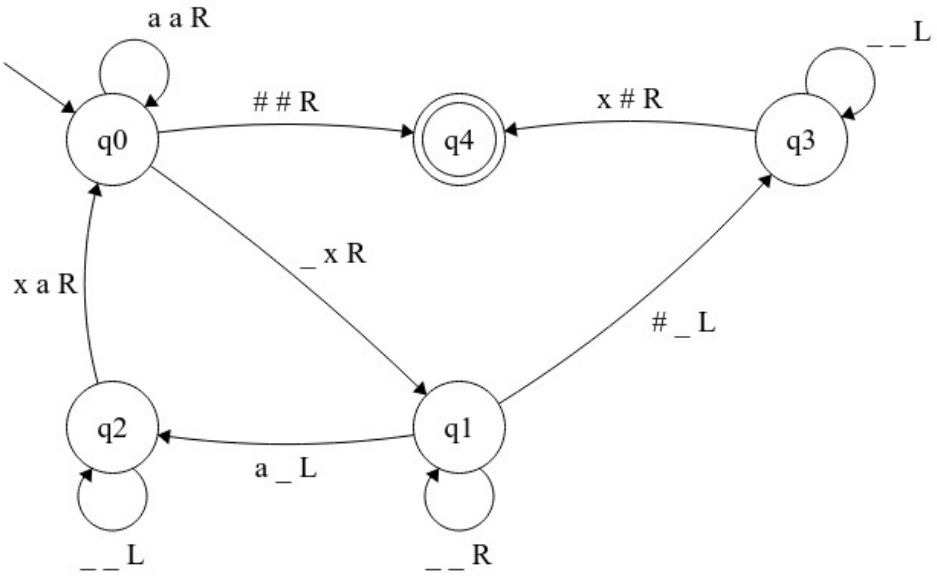
E: aaa

S: aaaaaaaaa



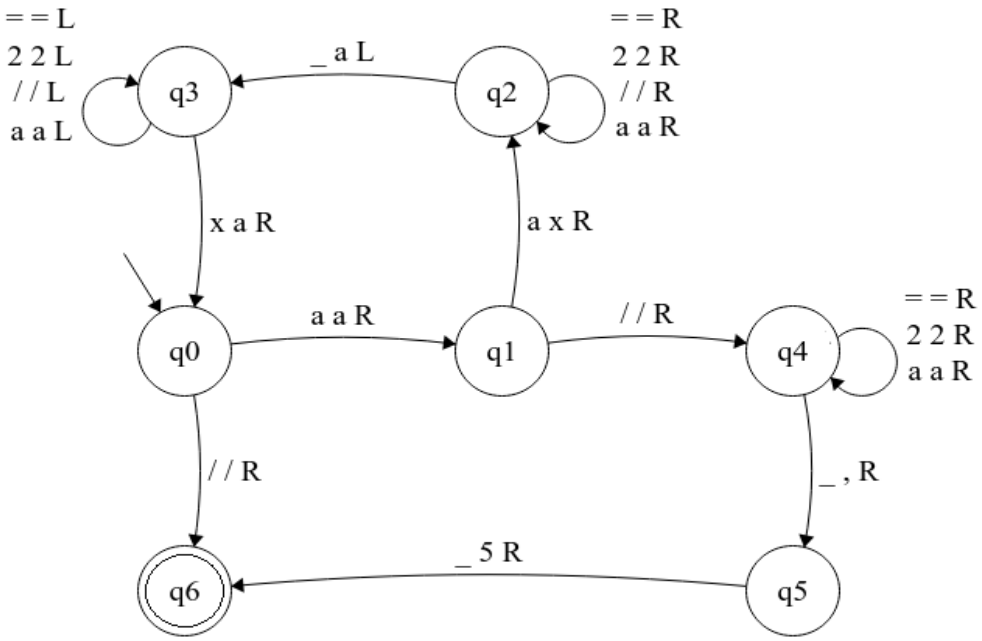
6. Construa uma MT para desfragmentar uma cadeia de a's com espaços entre eles e finalizada com #.

E: _ _ _ a a a _ a _ a a a _ _ _ a a _ _ _ a a _ a _ #
S: a a a a a a a a a a a a #



7. Construa uma MT para dividir por 2 um número no formato unário.

E: aaaa/2= ou E: aaaaa/2=
S: aaaa/2=aa S: aaaaa/2=aa,5

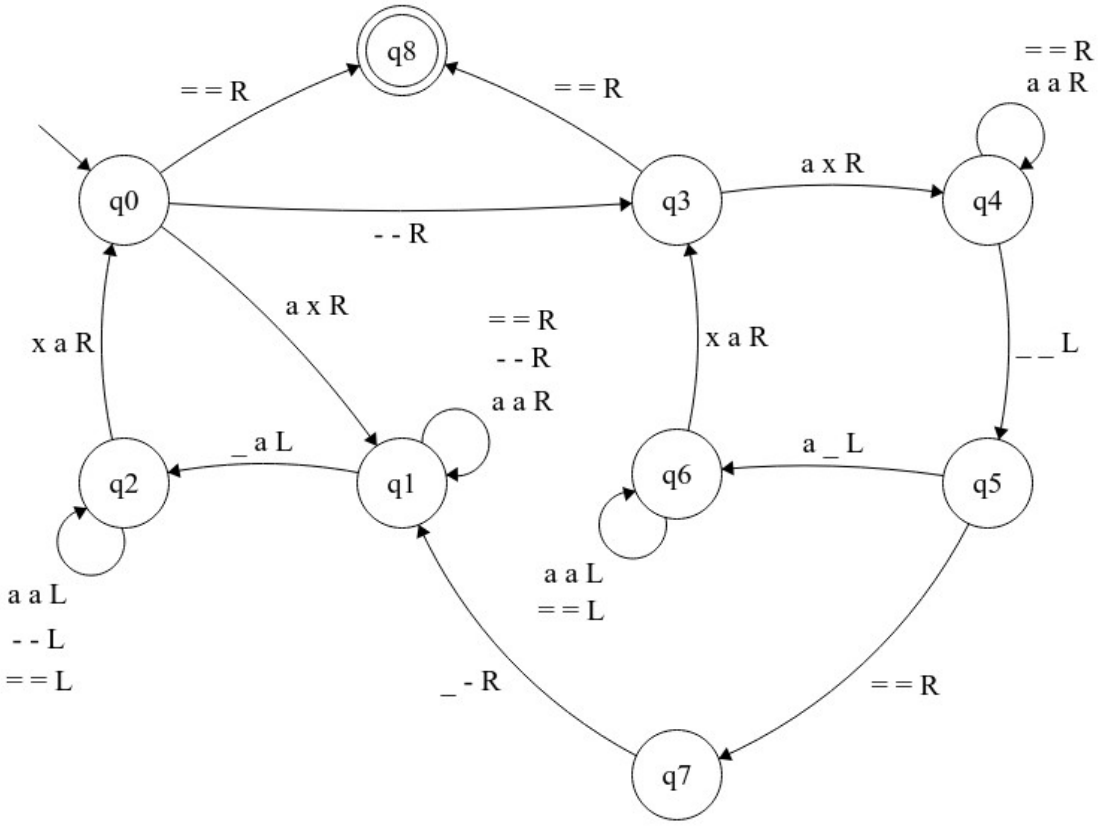


8. Construa uma MT para subtrair (maior – menor) dois números no formato unário.

E: aaaaa-aa=
S: aaaaa-aa=aaa

Desafio:

E: aa-aaaaa=
S: aa-aaaaa=-aaa



9. Construa uma MT para multiplicar 2 números no formato unário.

E: aaxaaaa=
S: aaxaaaa=aaaaaaaa

0 a b R 1
1 a a R 1
1 x x R 2
2 a b R 3
2 = = L 5
3 a a R 3
3 = = R 3
3 _ a L 4
4 a a L 4
4 = = L 4
4 b a R 2
5 a a L 5
5 x x L 5
5 b a R 0

Exercício: construir o dígrafo da MT.

10. Construa uma MT para calcular a potência de base 2.

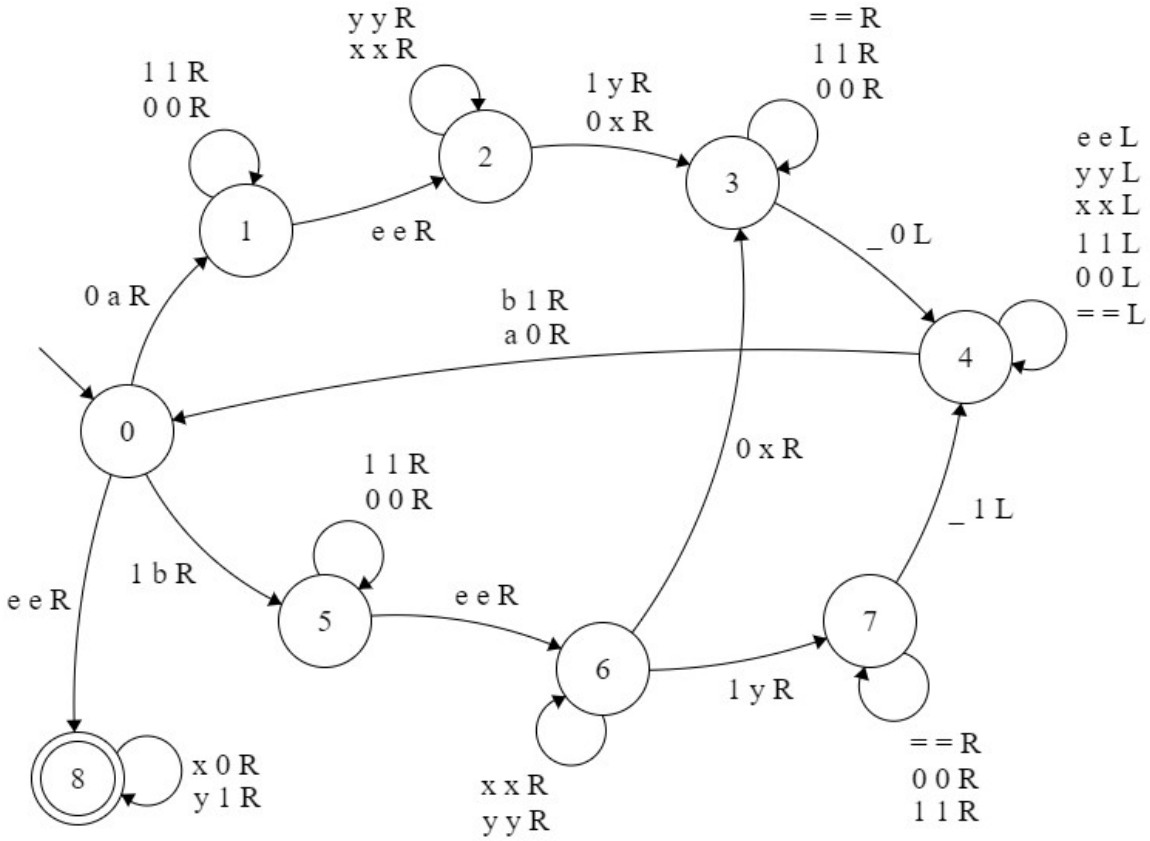
E: 2eaaa=
S: 2eaaa=aaaaaaaa

0 2 2 R 0
0 e e R 0
0 a a R 0
0 = = R 0
0 _ a L 1
1 = = L 1
1 a a L 1
1 e e R 2
2 a x R 3
3 a a R 3
3 = = R 4
4 a x R 5
4 b a R 7
5 a a R 5
5 b b R 5
5 _ b L 6
6 a a L 6
6 b b L 6
6 x a R 4
7 b a R 7
7 _ _ L 8
8 a a L 8
8 = = L 8
8 x a R 2

11. Construa uma MT para efetuar o AND bit-a-bit entre dois números binários de tamanhos iguais (1 ou mais bits).

E: 0011e1001=
S: 0011e1001=0001

AND: 0 e 0 = 0 ; 0 e 1 = 0 ; 1 e 0 = 0 ; 1 e 1 = 1



>> Complexidade de Algoritmos

>> Objetivos:

- Análise de um algoritmo já construído para verificar sua eficiência de computação (**objetivo principal da disciplina**).

- Projetar um algoritmo eficiente tendo, desde o início, a preocupação com sua complexidade (eficiência).

>> Critérios para medir a qualidade de um software:

- **Usuário:**

  - Interface.
  - Segurança e robustez.
  - Compatibilidade.
  - **Desempenho (velocidade de processamento). ****
  - **Consumo de recursos (ex.: memória RAM, espaço em disco etc.). ****

- **Desenvolvedor (programação):**

  - Portabilidade do código.
  - Facilidade de entendimento do código.
  - Reutilização de código.

**** A análise da complexidade de algoritmos é um processo para entender e avaliar um algoritmo em relação a esses critérios e, além disso, como aplicá-los em problemas práticos.**

➢ **Para pensar...** 

Um problema que possui solução conhecida, geralmente, pode ser resolvido através de diversos algoritmos. Porém, o fato de um algoritmo solucionar um determinado problema não significa que, na prática, essa solução seja aceitável.

Por que???

>> Medidas de Complexidade:

http://www.inf.ufrgs.br/~prestes/Courses/Complexity/aula1.pdf
https://www.ime.usp.br/~song/mac5710/slides/01complex.pdf

>> Tempo:

Podemos considerar o tempo absoluto em horas, minutos, segundos etc.. Porém, medir o tempo absoluto não é o melhor, pois depende da velocidade da máquina.

 **O tempo de execução de um algoritmo não depende somente do algoritmo, mas também do conjunto de instruções e capacidades do computador, da qualidade do compilador e da habilidade do programador.**

- Na **Análise da Complexidade de Algoritmos** conta-se o número de operações relevantes realizadas pelo algoritmo e esse número é expressado como uma função de **n**, onde **n** é um parâmetro que caracteriza o tamanho da entrada fornecida ao algoritmo. Essas operações podem ser: repetições, comparações, operações aritméticas, atribuições, chamadas de funções etc.

- O **número de operações** realizadas por um determinado algoritmo pode depender da particular instância da entrada. Em geral interessa-nos o **pior caso**, ou seja, o maior número de operações usadas para qualquer entrada de tamanho n.

  - Um caso relevante é quando **n** tem valor muito grande ($n \to \infty$) e, para esse caso, atribui-se o nome de comportamento assintótico.

- Análises também podem ser feitas para o **melhor caso** e o **caso médio**. Para o último, supõe-se conhecida uma certa distribuição da entrada (como a entrada se comporta na maioria das vezes).

- **Complexidade para o pior caso**: é analisada a entrada que faz o algoritmo funcionar mais lentamente.

- **Complexidade média**: todas as possíveis entradas são analisadas e o tempo médio é calculado.

💭 **O esforço realizado por um algoritmo é calculado a partir da quantidade de vezes que a operação fundamental é executada.**

http://www.inf.ufrgs.br/~prestes/Courses/Complexity/aula1.pdf

>> Comportamento Assintótico

A análise de todas as instruções de um algoritmo pode se tornar uma tarefa árdua e sem grandes efeitos para o cálculo da complexidade desse algoritmo.

O mais **importante** durante a análise é verificar qual o **termo que mais cresce** dentro da equação que descreve o número (quantidade) de operações realizadas pelo algoritmo. Esse termo denota o comportamento assintótico do algoritmo.

- Exemplo:

```
lista = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
maior = lista[0]
for v in lista:
    if v > maior:
        maior = v
print(maior)
```

Função f(n) que descreve o número de instruções que o programa executa para uma entrada com n valores. No exemplo, n é igual a 10.

- Cálculos para o pior caso:

  $f(n) = 1 + 1 + n + n + n - 1 + 1 = 3n + 2$

  Complexidade: **f(n) = n**

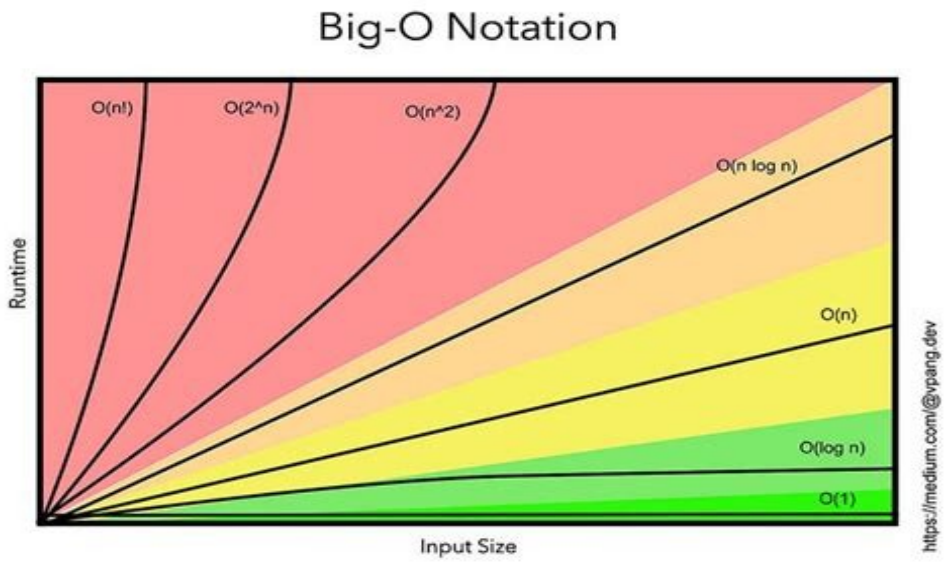
>> Notação Big-O:

Expressa, matematicamente, o comportamento assintótico de um algoritmo com grande número de dados na entrada (**n** com valores altos – **Pior caso**).

>> Tabela de Complexidade de Tempo:

Constante	O(1)	O mais rápido!
Logarítmica	O(log n)	Rápido!
Linear	O(n)	Sem analisar a entrada é o melhor que se espera.
Linearítmica	O(n log n)	Ótimo! Limite de muitos problemas práticos.
Quadrática	$O(n^2)$	Razoável para n pequenos.
Cúbica	$O(n^3)$	Razoável para n pequenos.
Exponencial	$O(2^n)$, $O(n^n)$	Não utilizar, se possível!!
Fatorial	O(n!)	Não utilizar, se possível!!

>> Gráfico de Complexidade de Tempo:



>> Tabela de Tempo - Complexidade vs Tamanho da entrada (n):

Tamanho da entrada

complexidade	10	100	10^3	10^4	10^5	10^6
$\log_2 n$	3	6	9	13	16	19
n	10	100	1000	10^4	10^5	10^6
$n \log_2 n$	30	664	9965	10^5	10^6	10^7
n^2	100	10^4	10^6	10^8	10^{10}	10^{12}
n^3	10^3	10^6	10^9	10^{12}	10^{15}	10^{18}
2^n	10^3	10^{30}	10^{300}	10^{300}	10^{3000}	10^{300000}

1 ano = $365 \times 24 \times 60 \times 60 \approx 3 \times 10^7$ segundos

1 século $\approx 3 \times 10^9$ segundos

1 milénio $\approx 3 \times 10^{10}$ segundos

http://www.inf.ufrgs.br/~prestes/Courses/Complexity/aula1.pdf

>> Cota ou limite superior (*upper bound*):

Exemplo: para multiplicar duas matrizes quadradas (n x n) há um algoritmo conhecido que utiliza 3 estruturas de repetição. A complexidade desse algoritmo é classificada como sendo $O(n^3)$. Sendo assim, pode-se afirmar que a complexidade desse problema não é superior a $O(n^3)$ **(*upper bound*)**.

A cota ou limite superior de um problema pode cair, caso alguém crie um novo e melhor algoritmo que solucione esse problema.

>> Exemplos no Notebook Jupyter (Google Colab):

- Analisar a complexidade:
  - $f(n) = n^2 + 1000\ n + 2000$

- Analisar as complexidades (gráfico):
  - $f(n) = 100\ n^2 + 5000\ n + 3000$
  - $g(n) = 5\ n^3 + 100\ n + 1000$
  - $h(n) = n \log_2 n$