UNIVERSIDADE ESTADUAL DE MONTES CLAROS – UNIMONTES

CENTRO DE CIÊNCIAS EXATAS E TECNOLÓGICAS – CCET

DEPARTAMENTO DE CIÊNCIAS DA COMPUTAÇÃO – DCC

EXERCÍCIO VI: TÓPICOS ESPECIAIS

RAMON LOPES DE QUEIROZ

MONTES CLAROS – MG

NOVEMBRO / 2025

RAMON LOPES DE QUEIROZ

EXERCÍCIO VI: TÓPICOS ESPECIAIS

Atividade avaliativa apresentada para atendimento de requisito parcial para aprovação na disciplina Matemática Computacional do Curso de Graduação em Bacharelado em Sistemas de Informação – 1º período

Professor: Dr. Reginaldo Morais de Macedo

MONTES CLAROS – MG

NOVEMBRO / 2025

Atividade VI:

1. Máquina de Turing e o Problema da Parada:

Máquina de Turing (MT): É um modelo matemático abstrato de computação. Não é uma máquina física, mas um conceito. Ela consiste em uma **fita infinita** (dividida em células), uma **cabeça** de leitura/escrita e um conjunto de **regras**. A máquina lê um símbolo na fita, consulta suas regras (seu "programa") e, com base no símbolo lido e em seu "estado" atual, ela (1) escreve um novo símbolo na fita, (2) move a cabeça para a esquerda ou direita, e (3) muda para um novo estado. A Máquina de Turing, apesar de sua simplicidade, é universal: acredita-se que *qualquer* algoritmo que pode ser executado por um computador real pode ser executado por uma Máquina de Turing (esta é a Tese de Church-Turing).

Problema da Parada (The Halting Problem): É a pergunta: "Existe um algoritmo (um programa) que possa analisar *qualquer* outro programa e sua *qualquer* entrada e determinar se esse programa eventualmente irá parar (halt) ou se ele entrará em um loop infinito?"

Alan Turing, em 1936, provou que **não**, tal algoritmo universal não pode existir. O Problema da Parada é **indecidível** (ou incomputável).

Exemplos:

1. **Loop Infinito Simples:**

Programa: while (true) { continue; }

Comentário: É trivial para um humano ver que este programa nunca para. Um "verificador de parada" simples poderia detectar este caso.

2. **Programa Complexo (Conjectura de Collatz):**

Programa: Pegue um número n. Se for par, divida por 2. Se for ímpar, multiplique por 3 e some 1. Repita até n ser 1. O programa *para* quando n chega a 1.

Comentário: Embora acreditemos que este programa pare para *qualquer* n inicial, ninguém jamais conseguiu provar isso. Um "verificador de parada" teria que,

essencialmente, resolver esta conjectura matemática aberta para garantir que o programa para para todos os inputs.

3. **O Paradoxo do Verificador (A Prova de Turing):**

Programa: Vamos supor que criamos o programa VerificaParada(P, I) (que diz se o programa P para com a entrada I). Agora, criamos um programa paradoxal Paradoxo(P):

1. Paradoxo(P) chama VerificaParada(P, P). (Ele se auto-analisa).
2. Se VerificaParada disser "Sim, vai parar", o Paradoxo entra em um loop infinito (while(true)).
3. Se VerificaParada disser "Não, vai entrar em loop", o Paradoxo para imediatamente.

Comentário: O que acontece se rodarmos Paradoxo(Paradoxo)? Se ele parar, ele deveria entrar em loop. Se ele entrar em loop, ele deveria parar. Isso é uma contradição lógica, provando que o VerificaParada perfeito não pode existir.

---

2. O Problema do Programa 0:

O "Problema do Programa 0" não é um termo padrão universal como o "Problema da Parada". No entanto, provavelmente se refere a uma variação indecidível clássica, que é:

"É possível criar um programa que determine se qualquer outro programa arbitrário, ao ser executado, eventualmente imprimirá o símbolo '0'?"

Assim como o Problema da Parada, este problema também é **indecidível**. Isso é provado por "redução": se pudéssemos resolver o "Problema do Programa 0", poderíamos usá-lo para resolver o Problema da Parada (que sabemos ser impossível).

(Como? Poderíamos pegar um programa P qualquer, modificar todo comando de "Parar" nele por um comando "Imprimir 0", e então usar nosso "Verificador de 0" para saber se o programa modificado imprime 0. Se ele imprimir, saberíamos que o

P original parou. Como isso resolveria o Problema da Parada, e o Problema da Parada é insolúvel, o "Problema do Programa 0" também deve ser insolúvel.)

Exemplos:

1. **Programa que Imprime Dígitos de Pi:**

Programa: Um algoritmo que calcula e imprime os dígitos de π (3.14159265...0...).

Comentário: Um "Verificador de 0" deveria responder "Sim", pois sabemos que o dígito '0' aparece em π (embora demore).

2. **Programa de Potências de 2:**

Programa: Um programa que imprime 2, 4, 8, 16, 32, 64, ...

Comentário: Um "Verificador de 0" deveria responder "Não", pois nenhum número nessa sequência termina em 0.

3. **Programa da Conjectura de Goldbach:**

Programa: Um programa que testa todos os números pares, um por um, procurando um que *não* seja a soma de dois números primos. Se encontrar um contra-exemplo, ele imprime "0" e para.

Comentário: Não sabemos se este programa *jamais* imprimirá "0". Se ele imprimir, a Conjectura de Goldbach (um famoso problema matemático não resolvido) é falsa. Um "Verificador de 0" funcional seria capaz de resolver problemas matemáticos abertos.

---

3. Decidibilidade, Computabilidade e Complexidade:

Computabilidade (O que podemos resolver?): É o ramo que estuda quais problemas *podem* ser resolvidos por um algoritmo (ou uma Máquina de Turing), dado tempo e memória infinitos. Se um problema não pode ser resolvido por uma Máquina de Turing, ele é **incomputável** (ou **não-computável**).

Decidibilidade (Podemos obter um "Sim" ou "Não"?): É um subconjunto da computabilidade focado em "problemas de decisão" (perguntas com resposta "sim" ou "não"). Um problema é **decidível** se existe um algoritmo (uma MT) que o resolve e que **sempre para** (halta) com a resposta correta ("sim" ou "não") em um tempo finito. Se o algoritmo pode não parar (entrar em loop) para uma das respostas, o problema é **indecidível**.

Complexidade (Quão rápido podemos resolver?): Este campo lida apenas com problemas que *sabemos* que são computáveis e decidíveis. A questão aqui não é *se* podemos resolver, mas *quão eficientemente* podemos resolvê-lo. Ela mede os recursos (normalmente **tempo** ou **espaço/memória**) que um algoritmo necessita, em função do tamanho da entrada (normalmente "n").

Exemplos:

Computabilidade:

1. **Computável:** Somar dois números. Existe um algoritmo claro para isso.
2. **Computável:** Ordenar uma lista de nomes. Vários algoritmos (como Quicksort ou Mergesort) fazem isso.
3. **Incomputável:** O Problema da Parada. Como provado por Turing, não existe um algoritmo que resolva este problema para todos os casos.

Decidibilidade:

1. **Decidível:** "O número *n* é primo?" Sim, podemos testar todos os divisores até a raiz quadrada de *n*. O algoritmo sempre para com "sim" ou "não".
2. **Decidível:** "Esta lista de nomes está em ordem alfabética?" Sim, podemos verificar a lista item por item. O algoritmo sempre para.
3. **Indecidível:** "O programa *P* irá parar com a entrada *I*?" (O Problema da Parada). Não há algoritmo que *sempre pare* com a resposta correta.

Complexidade:

1. **Tempo Polinomial (Rápido/Tratável):** Ordenar uma lista (ex: Quicksort) tem complexidade média $O(n \log n)$. Se a entrada (lista) dobrar, o tempo não dobra exponencialmente.

2. **Tempo Polinomial (Lento):** Multiplicação de matrizes padrão tem complexidade $O(n^3)$. É tratável, mas fica lento rapidamente.
3. **Tempo Exponencial (Intratável):** Resolver o "Caixeiro Viajante" (achar a menor rota visitando N cidades) por força bruta. A complexidade é $O(n!)$. Para 20 cidades, o número de cálculos é astronômico.

---

4. O Problema P vs NP:

Classe P (Polynomial time): É o conjunto de problemas de decisão que podemos **resolver** (achar a solução) rapidamente. "Rapidamente" significa em "tempo polinomial" (como $O(n)$, $O(n^2)$, $O(n^3)$). São considerados os problemas "fáceis" ou "tratáveis".

Classe NP (Nondeterministic Polynomial time): É o conjunto de problemas de decisão onde, se alguém lhe der uma *possível solução*, você pode **verificar** se ela está correta rapidamente (em tempo polinomial).

A grande questão (**P vs NP**) é: **Será que P = NP?**

Em outras palavras: Se um problema é fácil de *verificar*, isso significa que ele também é fácil de *resolver*?

Quase todos os cientistas da computação acreditam que **P ≠ NP** (ou seja, "verificar" é fundamentalmente mais fácil que "resolver"), mas ninguém jamais conseguiu provar isso.

Impactos

Se P = NP (se alguém provar que é possível resolver problemas NP rapidamente), o impacto seria revolucionário:

1. **Quebra da Criptografia:** Toda a segurança da internet (bancos, senhas, criptomoedas) é baseada em problemas NP que *acreditamos* serem difíceis de resolver (como a fatoração de números gigantes). Se P=NP, tudo isso seria quebrável instantaneamente.

2. **Avanços Científicos:** Problemas complexos em biologia (como prever o enovelamento de proteínas), logística (Caixeiro Viajante), e design de novos materiais poderiam ser resolvidos de forma otimizada.

Exemplos:

1. **Exemplo de P:**

Problema: Ordenar uma lista.

Comentário: Este problema está em P. Podemos *resolver* (ordenar a lista) rapidamente usando Quicksort (em tempo $O(n \log n)$).

2. **Exemplo de NP (e não se sabe se P):**

Problema: Sudoku.

Comentário: Verificar uma grade de Sudoku preenchida é muito fácil (Classe P para verificação): basta checar as 9 linhas, 9 colunas e 9 blocos. No entanto, *resolver* um Sudoku difícil do zero pode ser muito demorado (exponencial). Como é fácil verificar, ele está em NP. Não sabemos se existe um algoritmo em P para resolvê-lo rapidamente sempre.

3. **Exemplo de NP (e não se sabe se P):**

Problema: Fatoração de Inteiros (a base da criptografia RSA).

Comentário: Se eu lhe der dois números (ex: 3.607 e 3.803) e pedir para multiplicá-los, é fácil (P). Mas se eu lhe der o resultado (13.717.421) e pedir para *achar* os fatores, é extremamente difícil. Porém, *verificar* a resposta (uma simples multiplicação) é fácil. Portanto, a fatoração está em NP.

---

5. O Paradoxo de Gödel (Teoremas da Incompletude):

O "Paradoxo de Gödel" refere-se aos **Teoremas da Incompletude de Gödel** (1931), que abalaram os fundamentos da matemática. Não é um paradoxo, mas sim um teorema *sobre* paradoxos (como o "Paradoxo do Mentiroso").

O **Primeiro Teorema da Incompletude** afirma que:

Em qualquer sistema axiomático formal consistente (como a aritmética que usamos) que seja poderoso o suficiente para descrever a si mesmo, **sempre existirão declarações verdadeiras que não podem ser provadas** *dentro* desse sistema.

Em termos simples: **Nenhum sistema formal pode provar *todas* as verdades sobre si mesmo sem ser contraditório.**

Gödel provou isso construindo uma fórmula matemática (a "sentença G") que, essencialmente, significava: "Esta sentença não pode ser provada".

Se ela fosse provada (falsa), o sistema seria inconsistente.

Se ela não pudesse ser provada (verdadeira), o sistema seria incompleto.
Gödel mostrou que ela é verdadeira, mas não provável, logo, o sistema é incompleto.

Impactos para a Computação
O impacto é profundo e filosófico, estabelecendo os limites do que é "conhecível":

1. **Fundação para Turing:** O trabalho de Gödel (1931) é o precursor matemático direto do trabalho de Turing (1936). O **Problema da Parada é a versão computacional do Teorema da Incompletude**.
   - *Gödel:* "Nem tudo que é verdadeiro pode ser provado."
   - *Turing:* "Nem tudo que é computável é decidível." (Ou: "Nem toda pergunta sobre programas pode ser respondida por um programa.")
2. **Limites da IA:** Os teoremas de Gödel sugerem que uma "Inteligência Artificial Geral" baseada puramente em lógica formal (como um programa de computador) também seria inerentemente incompleta. Ela encontraria "verdades" (talvez sobre seu próprio código ou o mundo) que não poderia provar formalmente.
3. **Verificação de Programas:** O trabalho de Gödel implica que nunca poderemos criar um programa de computador que possa provar *automaticamente* que *qualquer* outro programa está 100% correto e livre de bugs (pois isso seria equivalente a resolver o Problema da Parada).

Referências Bibliográficas:

CIÊNCIA TODO DIA. **P vs NP: O problema matemático que pode MUDAR O MUNDO**. [S. l.: s. n.], 4 out. 2019. 1 vídeo (17 min 19 s). Publicado pelo canal Ciência Todo Dia. Disponível em: https://www.youtube.com/watch?v=O-OLFG0WJpc. Acesso em: 17 nov. 2025.

MÁQUINA de Turing. In: WIKIPÉDIA: a enciclopédia livre. Flórida: Wikimedia Foundation, 2025. Disponível em: https://pt.wikipedia.org/wiki/M%C3%A1quina_de_Turing. Acesso em: 17 nov. 2025.

P versus NP. In: WIKIPÉDIA: a enciclopédia livre. Flórida: Wikimedia Foundation, 2025. Disponível em: https://pt.wikipedia.org/wiki/P_versus_NP. Acesso em: 17 nov. 2025.

PEGORARO, Heitor de Oliveira. O Teorema de Gödel e a Indecidibilidade. **Revista Brasileira de Ensino de Física**, v. 43, 2021. Disponível em: https://www.scielo.br/j/rbef/a/3pbPHWQ6BLHSfWcH6PBY7Rk/?lang=pt. Acesso em: 17 nov. 2025.

PROBLEMA da parada. In: WIKIPÉDIA: a enciclopédia livre. Flórida: Wikimedia Foundation, 2025. Disponível em: https://pt.wikipedia.org/wiki/Problema_da_parada. Acesso em: 17 nov. 2025.

TEORIA da computabilidade. In: WIKIPÉDIA: a enciclopédia livre. Flórida: Wikimedia Foundation, 2025. Disponível em: https://pt.wikipedia.org/wiki/Teoria_da_computabilidade. Acesso em: 17 nov. 2025.