Alunos: Vitor Enzo e Vinicius de Oliveira

① Sabe-se que o problema da parada (the halting problem) é incomputável. Explique esse problema discutindo porque ele é incomputável.

Considerando o problema da parada, temos: Dado um determinado programa P e uma entrada I, devemos saber se o programa com a entrada ficará executando infinitamente ou não. Entretanto sabe-se que não pode haver um algoritmo que receba esse programa P e saiba resolver esse problema.

Isso é dado uma vez que, considerando uma função computável halts(f) que retorna TRUE se a subrotina f termina e falso caso nunca terminar, mostrada abaixo, teremos uma contradição:

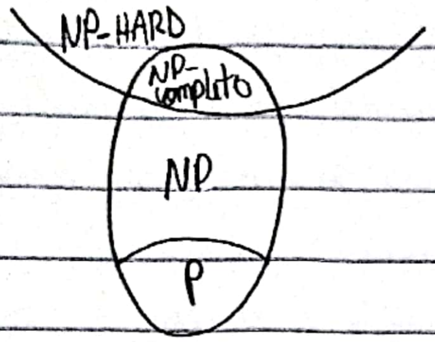
```
def g():
    if halts(g):
        loop_forever
```

Ou seja, observando o código, se halts(g) retornar TRUE, sabemos que o código termina, mas logo após isso, entra em um loop infinito. Da mesma forma, se halts(g) retornar FALSE, então significa que o código não irá terminar, mas logo em seguida o conteúdo dentro do if não será executado e o programa é finalizado.

Por isso, a suposição de que existe função computável halts() é FALSA.

② a) O que são problemas polinomiais, ou seja, pertencentes a classe P. Cite um exemplo de problema P.

Dada a seguinte estrutura de Classes:




temos que os problemas polinomiais são aqueles em que existe um determinado algoritmo A com complexidade $O(n^k)$ (polinomial) que soluciona P. Ou seja, de forma matemática, podemos dizer que um problema qualquer p pode ser resolvido utilizando uma máquina de turing determinística

Pode-se considerar como exemplo algoritmos de ordenação: Quick-sort, merge sort e vários outros.

b) Explique o que são problemas não determinísticos polinomiais (NP). Cite um exemplo

Dado um problema p, dizemos que $p \in NP$ caso seja facilmente verificável, mas não facilmente solucionável. Em outras palavras, não existe algoritmo eficiente que solucione p, porém, dada uma solução, é fácil verificar se ela é válida. De forma matemática podemos dizer que é um problema que consegue ser resolvido utilizando uma máquina de turing não determinística.

Como exemplo temos verificar se um número é primo. Dado apenas um número, é difícil calcular se ele é primo ou não, porém, dado a sua lista de divisores, é fácil verificar se ele é primo.

③ a) Explique o que são problemas np-completos? Cite um exemplo

Pode-se dizer que problemas $p \in NP$ para os quais $\forall q \in NP$ conseguem ser reduzidos a p de maneira eficiente (tempo polinomial) são chamados de NP-completos. Ou seja, se houver um algoritmo eficiente para resolver um problema NP-completo, então há um algoritmo eficiente para resolver todos os problemas NP

Assim, sabe-se que os problemas NP-completos são bem difíceis de serem resolvidos, uma vez que necessitam de uma quantidade exponencial de tempo à medida que o tamanho do problema aumenta

Como exemplo, podemos citar: O problema do roteamento de veículos, problema da torre de hanói e vários outros.

b) Explique o que são problemas NP-HARD. Cite um exemplo.

São problemas $p \notin NP$ onde verificar se uma dada solução é válida não é viável, ou seja, são problemas que não podem ser resolvidos em tempo polinomial, mesmo utilizando uma máquina de turing não-determinística. Assim os problemas NP-HARD incluem problemas de otimização e decisão que necessitam da análise de todas as possibilidades antes de determinar qual a melhor solução, podendo levar um tempo extremamente grande.

Como exemplo de problema NP-HARD temos o problema do caixeiro viajante (TSP)

S T Q Q S S D

__ / __ / __

4) Sejam $L_1$ e $L_2$, dois problemas de decisão e suponha que o algoritmo $A_2$ resolva $L_2$. Em outras palavras, se $y$ é uma entrada para $L_2$, então $A_2$ retornará True ou False, dependendo se $y \in L_2$ ou $y \notin L_2$.

A ideia da redução é encontrar uma transformação de $L_1$ para $L_2$ de modo que o algoritmo $A_2$ possa ser parte de um algoritmo $A_1$ para resolver $L_1$.

Dessa forma, a redução é um algoritmo que transforme um problema em outro problema. Assim, $L_1$ é redutível ao problema $L_2$, se um algoritmo $A_2$ utilizado para resolver $L_2$ de forma eficiente, também poderá ser usado como uma subrotina para resolver $L_1$ eficientemente. Caso isso seja verdade, então resolver $L_1$ não deverá necessitar de mais recursos para sua solução que $L_2$.

5) O Teorema de Cook-Levin estabelece que o problema de satisfabilidade booleana (SAT) é NP-completo. Em outras palavras, ele prova que qualquer problema em NP pode ser reduzido ao problema de SAT em tempo polinomial.

Isso significa que se pudermos resolver SAT em tempo polinomial, então poderíamos resolver qualquer problema em NP em tempo polinomial. No entanto, não se sabe se SAT pode ser resolvido em tempo polinomial ou não.

A importância desse teorema se deve ao incentivo do estudo da NP-Completude, tornando-a um dos principais temas de pesquisa na área de algoritmos e complexidade computacional.

Dessa forma, o teorema tem implicações na prática da resolução de problemas computacionais; na classificação de problemas, pois ajuda a identificar quais problemas são mais difíceis e onde o foco da pesquisa deve ser direcionado; na segurança criptográfica, pois muitos sistemas criptográficos são baseados na dificuldade de resolver problemas NP-completos, como o SAT. Resumindo, o teorema ajuda a identificar limitações fundamentais dos algoritmos e a motivar o desenvolvimento de algoritmos aproximados para resolver problemas intratáveis.