COMPLEXIDADE COMPUTACIONAL

Algoritmos e Estruturas de Dados

Prof.ª Hélida Salles

Introdução e conceitos básicos

- Um algoritmo é um procedimento que consiste em um conjunto de regras não ambíguas as quais especificam, para cada entrada, uma sequência finita de operações, terminando com uma saída correspondente.
- Um algoritmo resolve um problema quando, para qualquer entrada, produz uma resposta correta, se forem concedidos tempo e memória suficientes para a sua execução.

Problemas indecidíveis

- Existem problemas para os quais não existe um algoritmo
  - São conhecidos como problemas indecidíveis.

Exemplo:

- Problema da parada: dado um programa e um arquivo de entrada para ele, decidir se tal programa pára ou não, quando executado com este arquivo de entrada
  - Não existe algoritmo que resolva o problema da parada.

Introdução e conceitos básicos

Teoria x Complexidade X Computabilidade

- Na teoria da computação perguntamos: "Quais problemas são efetivamente computáveis?"
    - Propósito da TC = desenvolver modelos matemáticos formais de computação que reflitam os computadores do mundo real
- Na teoria da complexidade perguntamos: "O que torna alguns problemas computacionalmente difíceis e outros fáceis?"

Introdução e conceitos básicos

Teoria x Complexidade X Computabilidade

- Na teoria da computabilidade a preocupação é classificar problemas entre tratáveis e intratáveis

No projeto de algoritmos, a pergunta é mais específica:
"Quais problemas são eficientemente computáveis?"
O que significa eficiente?

Introdução e conceitos básicos

- Custos de algoritmos
  - Definir qual tipo de custo interessa.
  - Uma execução tem vários custos associados:
    - Tempo de execução,
    - Uso de espaço (cache, memória, disco),
    - Energia consumida,
    - Energia dissipada, etc.

Introdução e conceitos básicos

- Custos de algoritmos
  - Existem características e medidas que são importantes em contextos diferentes
    - Linhas de código fonte (LOC), legibilidade, manutenabilidade, corretude, custo de implementação, robustez, extensibilidade, etc.
  - A medida mais importante: **tempo de execução**.
  - A complexidade pode ser vista como uma propriedade do problema

Introdução e conceitos básicos

- Mesmo um problema sendo computável, não significa que existe um algoritmo que vale a pena aplicar.

- Qual o melhor algoritmo?
  - Para um dado problema, existem diversos algoritmos com desempenhos diferentes.
- Com um algoritmo ineficiente, um computador rápido não ajuda!

Introdução e conceitos básicos

- Suponha que uma máquina resolva um problema de tamanho N em um dado tempo t.
  - Qual tamanho de problema uma máquina 10 vezes mais rápida resolve no mesmo tempo?

Como comparar eficiências?

- Uma medida concreta do tempo depende:
  - do tipo da máquina usada (arquitetura, cache, memória, ...)
  - da qualidade e das opções do compilador ou ambiente de execução
  - do tamanho do problema (da entrada)
- Portanto, foram inventadas as máquinas abstratas.
- A análise da complexidade de um algoritmo consiste em determinar o número de operações básicas (atribuição, soma, comparação, ...) em relação ao tamanho da entrada.
- Observe que nessa medida o tempo é "discreto" (não contínuo).

Análise assintótica

- Em geral, o número de operações fornece um nível de detalhamento grande.
- Portanto, analisamos somente a taxa ou ordem de crescimento, substituindo funções exatas com cotas mais simples.
- Duas medidas são de interesse particular:
  - A complexidade pessimista
  - A complexidade média
  - Também podemos pensar em considerar a complexidade otimista (no caso melhor): mas essa medida faz pouco sentido

Complexidade de algoritmos

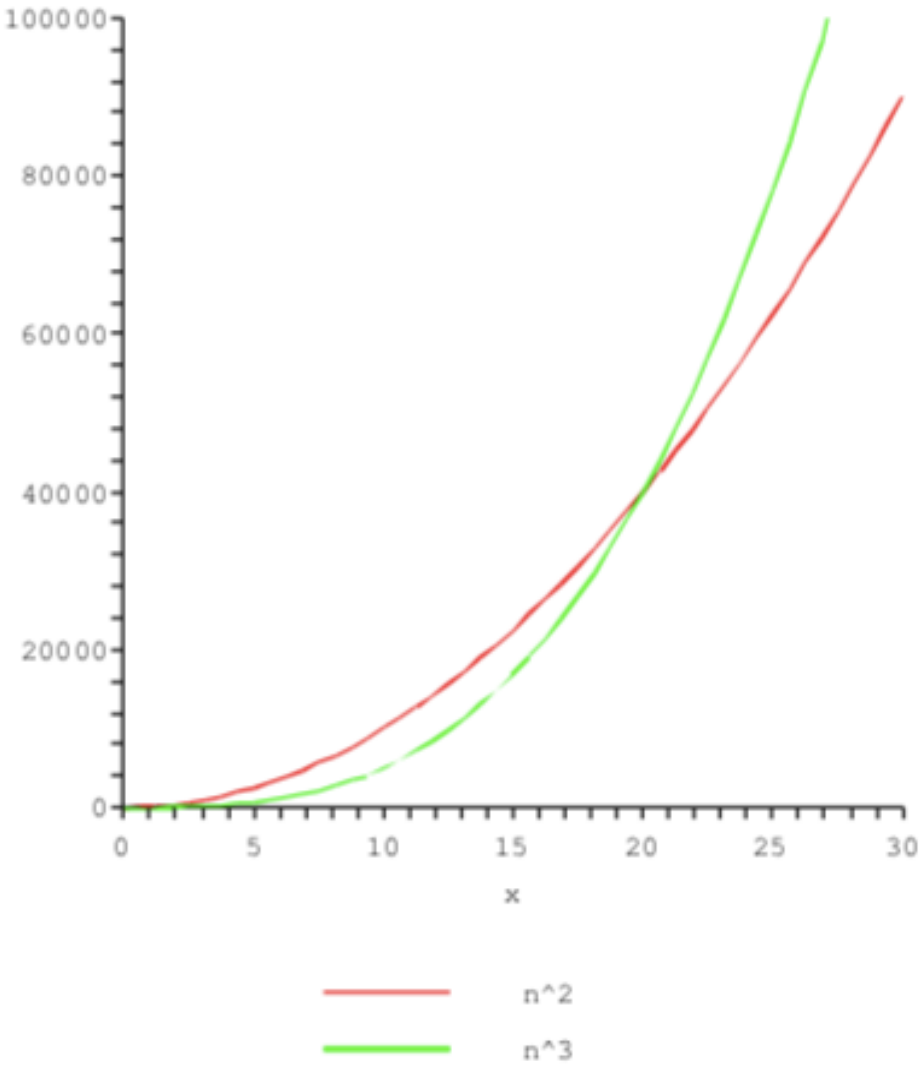
- Considere dois algoritmos A e B com tempo de execução $O(n^2)$ e $O(n^3)$, respectivamente. Qual deles é o mais eficiente?

- Considere dois programas A e B com tempos de execução $100n^2$ milisegundos, e $5n^3$ milisegundos, respectivamente, qual é o mais eficiente?

Complexidade de algoritmos

- Considerando dois algoritmos com tempo de execução $O(n^2)$ e $O(n^3)$ esperamos que o primeiro seja mais eficiente que o segundo.
- Para n grande, isso é verdadeiro, mas o tempo de execução atual pode ser $100n^2$ no primeiro e $5n^3$ no segundo caso.
    - Logo para $n < 20$ o segundo algoritmo é mais rápido.

Complexidade de algoritmos



- Assintoticamente consideramos um algoritmo com complexidade $O(n^2)$ melhor que um algoritmo com $O(n^3)$.
- De fato, para n suficientemente grande $O(n^2)$ sempre é melhor.
- Mas na prática, não podemos esquecer o tamanho do problema real.

Exercício

1. Considere dois computadores C1 e C2 que executam 10^7 e 10^9 operações por segundo (OP/s) e dois algoritmos de ordenação A e B que necessitam $2n^2$ e $50n\log_{10}n$ operações com entrada de tamanho n, respectivamente. Qual o tempo de execução de cada combinação para ordenar 10^6 elementos?

Exercício

Algoritmo	Comp. C_1	Comp. C_2
A	$\frac{2\times(10^6)^2 OP}{10^7 OP/s} = 2 \times 10^5 s$	$\frac{2\times(10^6)^2 OP}{10^9 OP/s} = 2 \times 10^3 s$
B	$\frac{50\times10^6 \log 10^6 OP}{10^7 OP/s} = 30s$	$\frac{50\times10^6 \log 10^6 OP}{10^9 OP/s} = 0.3s$

Um panorama de tempo de execução

- Tempo constante: O(1) (raro).
- Tempo sublinear (log(n), log(log(n)), etc): Rápido.
- Tempo linear: Número de operações proporcional à entrada.
- Tempo n log n: Comum em algoritmos de divisão e conquista.
- Tempo polinomial n^k: Frequentemente de baixa ordem ($k \leq 10$), considerado eficiente.
- Tempo exponencial: 2^n, $n!$, n^n considerado intratável.

Exemplos de algoritmos

- Tempo constante: Determinar se uma sequência de números começa com 1.
- Tempo sublinear: Busca binária.
- Tempo linear: Buscar o máximo de uma sequência.
- Tempo n log n: Mergesort.
- Tempo polinomial: Multiplicação de matrizes.
- Tempo exponencial: Busca exaustiva de todos subconjuntos de um conjunto, de todas permutações de uma sequência, etc.

Classes de problemas

- P
- NP
- NP-completos

- Pesquisar e realizar tarefa no moodle

Classes P e NP

Algoritmos e Estruturas de Dados

Prof.ª Hélida Salles

Problemas intratáveis

- Problemas que podem ser resolvidos em tempo polinomial em um computador típico são exatamente os mesmos problemas que podem ser resolvidos em tempo polinomial em uma máquina de Turing
- Problemas práticos = exigem tempo polinomial = resolvidos em um período em tempo tolerável
- Problemas que exigem tempo exponencial = intratáveis!

Tipos de problemas

- Decisão
  - Problema que exige resposta (SIM/NÃO)
  - Mais simples
- Localização
  - Procura de um elemento que satisfaça a condição de solução
  - Tem um correspondente de decisão
- Otimização
  - Além de exibir uma solução, requer uma solução de qualidade ótima
  - Pode-se definir problemas de decisão perguntando se existe uma solução melhor que um certo parâmetro.

Máquina de Turing

- Uma máquina de Turing (MT) é uma máquina de computação abstrata com o poder dos computadores reais e de outras definições matemáticas do que pode ser calculado
    - A MT consiste em um controle de estados finitos e uma fita infinita dividida em células
    - Cada célula contém um dentre um número finito de de símbolos de fita
    - A MT executa movimentos com base em seu estado atual e no símbolo de fita presente na célula varrida pela cabeça de fita
    - Em um movimento ela muda de estado, sobregrava a célula varrida com algum símbolo e move a cabeça uma célula para a direita ou para a esquerda.

As classes p e np

- Formalmente:
  - As classes de problemas P são aquelas que podem ser resolvidos em tempo polinomial por MT determinísticas
  - As classes de problemas NP são aquelas que podem ser resolvidos em tempo polinomial por MT não-determinísticas

Classe P

- Uma máquina de Turing M é dita de complexidade de tempo T(n), [que tem tempo de execução T(n)], se sempre que M recebe uma entrada w, de tamanho n, M pára depois de efetuar no máximo T(n) movimentos, independente do fato de M aceitar ou não.
- Uma linguagem L está na classe P se existe algum polinômio T(n) tal que L = L(M) para alguma MT determinística M de complexidade de tempo T(n)

Classe NP

- Uma linguagem L está na classe NP se existe uma MT não-determinística M e uma complexidade de tempo polinomial T(n) tais que L=L(M) e, quando M recebe uma entrada de comprimento n, não existe nenhuma sequência de mais de T(n) movimentos de M.

Classe NP completa

- Seja uma linguagem L na classe NP. L é NP-completa se:
  1. L está em NP
  2. Para toda linguagem L' em NP, existe uma redução de tempo polinomial de L' a L

Algumas observações

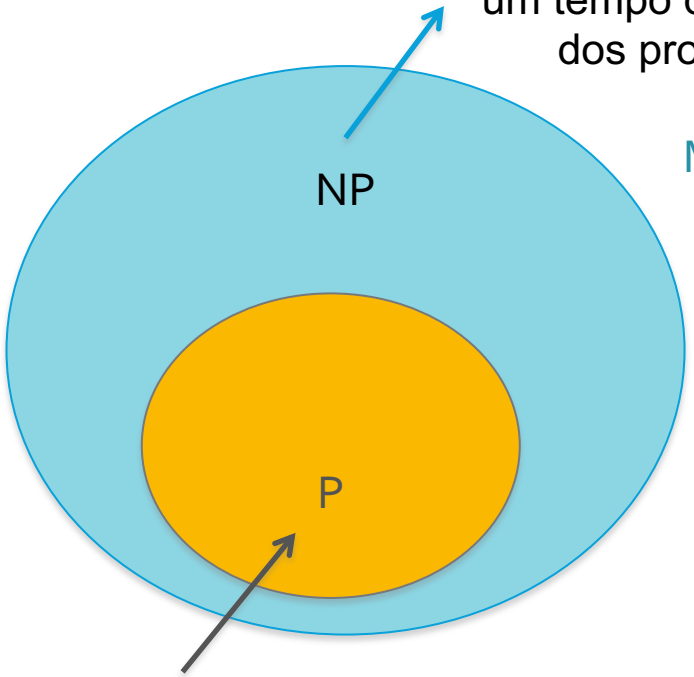
- Como toda MT determinística é uma MT não-determinística que nunca tem opções de movimentos, $P \subseteq NP$.
- Parece que NP contém muitos problemas que não estão em P
- A razão intuitiva é que uma NMT funcionando em tempo polinomial tem a habilidade de pressupor um número exponencial de soluções possíveis para um problema e verificar cada uma delas em um tempo polinomial, "em paralelo". Contudo:
    - Questão em aberto (desde 1971):

      P = NP? De fato, tudo que pode ser feito em tempo polinomial por uma NMT pode realmente ser feito por uma DMT em tempo polinomial?

Exemplo

NP-completo é a classe dos problemas mais difíceis de NP

Inclui os problemas que são (P) e para os problemas da classe NP, os melhores algoritmos conhecidos têm um tempo de trabalho parecido ao dos problemas intratáveis.



Mas ninguém conseguiu demonstrar que não existem algoritmos polinomiais para eles. Ninguém conseguiu provar que são intratáveis!

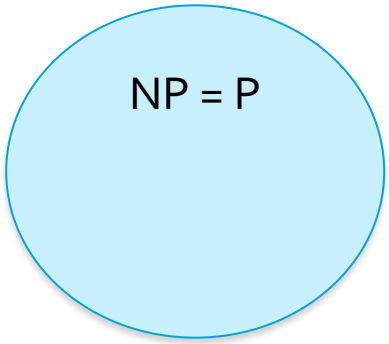
Os problemas que são resolvidos pelos computadores em um tempo razoável são chamados de polinomiais (P)

Hipótese 1

A teoria desenvolvida nos últimos anos chegou, entretanto, a alguma conclusão útil:

•Definiu uma subclasse da classe NP, a subclasse dos problemas NP-completos: problemas mais difíceis da classe NP

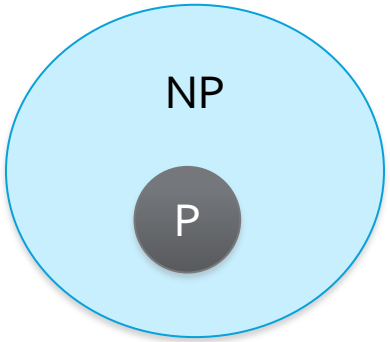
- Se para qualquer um dos tais problemas NP-completos se encontrar um algoritmo polinomial, então todos eles seriam resolvidos em tempo polinomial
- A classe NP seria rebaixada a P,
- P=NP.



Hipótese 2

Se fosse demonstrado que um só dos problemas NP-completos é intratável, então:

- Todos eles seriam intratáveis
- P≠NP.



Consequências

- P≠NP
  - Não haveria mais sentido buscar algoritmos polinomiais para uma série de problemas interessantes, porque saberíamos com segurança que tais algoritmos são intratáveis
- P=NP
  - Teríamos encontrado algoritmos polinomiais para todos esses problemas.
  - Poderíamos resolver, em bem pouco tempo, problemas como o do caixeiro-viajante (NP)
  - Senhas criptográficas seriam decifradas facilmente, pois a criptografia atual depende de um problema da classe NP

Referências

- Cap. 10, Introdução à teoria de autômatos, linguagens e computação. HOPCROFT, ULLMAN & MOTWANI.
- Cap. 6, Complexidade de Algoritmos. TOSCANI & VELOSO.