MC458 — Projeto e Análise de Algoritmos I

Cid Carvalho de Souza   Cândida Nunes da Silva
Orlando Lee

1 de março de 2016

Antes de mais nada. . .

- Uma versão anterior deste conjunto de slides foi preparada por Cid Carvalho de Souza e Cândida Nunes da Silva para uma instância anterior desta disciplina.
- Esses slides são o fruto de um trabalho colaborativo de vários professores.
- Nunca é demais enfatizar que o material é apenas um guia e não deve ser usado como única fonte de estudo. Para isso consultem a bibliografia (em especial, "Cormen" e "Manber").

Orlando Lee

Agradecimentos (Cid e Cândida)

- Várias pessoas contribuíram direta ou indiretamente com a preparação deste material.
- Algumas destas pessoas cederam gentilmente seus arquivos digitais enquanto outras cederam gentilmente o seu tempo fazendo correções e dando sugestões.
- Eis a lista de "colaboradores" (em ordem alfabética):
  - Célia Picinin de Mello
  - José Coelho de Pina
  - Orlando Lee
  - Paulo Feofiloff
  - Pedro Rezende
  - Ricardo Dahab
  - Zanoni Dias

Introdução

O que veremos nesta disciplina?

- Como provar a "corretude" de um algoritmo
- Estimar a quantidade de recursos (tempo, memória) de um algoritmo = análise de complexidade
- Técnicas e idéias gerais de projeto de algoritmos: divisão-e-conquista, programação dinâmica, algoritmos gulosos etc
- Tema recorrente: natureza recursiva de vários problemas
- A dificuldade intrínseca de vários problemas: inexistência de soluções eficientes

Algoritmos

O que é um algoritmo?

Informalmente, um algoritmo é um procedimento computacional bem definido que:

- recebe um conjunto de valores como entrada e
- produz um conjunto de valores como saída.

Equivalentemente, um algoritmo é uma ferramenta para resolver um problema computacional. Este problema define a relação precisa que deve existir entre a entrada e a saída do algoritmo.

Exemplos de problemas: teste de primalidade

Problema: determinar se um dado número é primo.

Exemplo:

Entrada: 9411461

Saída: É primo.

Exemplo:

Entrada: 8411461

Saída: Não é primo.

Exemplos de problemas: ordenação

Definição: um vetor $A[1 \dots n]$ é crescente se $A[1] \leq \dots \leq A[n]$.

Problema: rearranjar um vetor $A[1 \dots n]$ de modo que fique crescente.

Entrada:

1										n
33	55	33	44	33	22	11	99	22	55	77

Saída:

1										n
11	22	22	33	33	33	44	55	55	77	99

Instância de um problema

Uma instância de um problema é um conjunto de valores que serve de entrada para esse.

Exemplo:
Os números 9411461 e 8411461 são instâncias do problema de primalidade.

Exemplo:
O vetor

1										n
33	55	33	44	33	22	11	99	22	55	77

é uma instância do problema de ordenação.

A importância dos algoritmos para a computação

Onde se encontra aplicações para o uso/desenvolvimento de algoritmos "eficientes"?

- projetos de genoma de seres vivos
- rede mundial de computadores
- comércio eletrônico
- planejamento da produção de indústrias
- logística de distribuição
- *games* e filmes
- ...

Dificuldade intrínseca de problemas

- Infelizmente, existem certos problemas para os quais **não se conhece** **algoritmos "eficientes"** capazes de resolvê-los. Eles são chamados problemas $\mathcal{NP}$-completos.

  Curiosamente, **não foi provado** que tais algoritmos não existem!
- Esses problemas tem a característica notável de que se <u>um</u> deles admitir um algoritmo "eficiente" então <u>todos</u> admitem algoritmos "eficientes".
- Por que devo me preocupar com problemas $\mathcal{NP}$-completos?
  Problemas dessa classe surgem em inúmeras situações práticas!

Dificuldade intrínseca de problemas

Exemplos:

- calcular as rotas dos caminhões de entrega de uma distribuidora de bebidas em São Paulo, minimizando a distância percorrida. (vehicle routing)
- calcular o número mínimo de *containers* para transportar um conjunto de caixas com produtos. (bin packing 3D)
- calcular a localização e o número mínimo de antenas de celulares para garantir a cobertura de uma certa região geográfica. (facility location)
- e muito mais...

É importante saber identificar quando estamos lidando com um problema $\mathcal{NP}$-completo!

Algoritmos e tecnologia

- O mundo ideal: os computadores têm velocidade de processamento e memória infinita. Neste caso, qualquer algoritmo é igualmente bom e esta disciplina é inútil! Porém...
- O mundo real: computadores têm velocidade de processamento e memória limitadas.

Neste caso faz <u>**muita**</u> diferença ter um <u>**bom**</u> algoritmo.

Algoritmos e tecnologia

Exemplo: **ordenação de um vetor de n elementos**

- Suponha que os computadores A e B executam $1G$ e $10M$ instruções por segundo, respectivamente. Ou seja, A é **100 vezes mais rápido** que B.
- Algoritmo 1: implementado na máquina A por um excelente programador em linguagem de máquina (ultra-rápida). Executa $2n^2$ instruções.
- Algoritmo 2: implementado na máquina B por um programador mediano em linguagem de alto nível dispondo de um compilador "mais-ou-menos". Executa $50n \log n$ instruções.

Algoritmos e tecnologia

- O que acontece quando ordenamos um vetor de um milhão (10^6) de elementos? **Qual algoritmo é mais rápido?**
- Algoritmo 1 na máquina A:
  $\frac{2.(10^6)^2 \text{ instruções}}{10^9 \text{ instruções/segundo}} \approx 2000$ segundos
- Algoritmo 2 na máquina B:
  $\frac{50.(10^6 \log 10^6) \text{ instruções}}{10^7 \text{ instruções/segundo}} \approx 100$ segundos
- Ou seja, B foi **VINTE VEZES** mais rápido do que A!
- Se o vetor tiver 10 milhões (10^7) de elementos, esta razão será de 2.3 dias para 20 minutos!

Algoritmos e tecnologia – Conclusões

- O uso de um algoritmo adequado pode levar a ganhos extraordinários de desempenho.
- Isso pode ser tão importante quanto o projeto de *hardware*.
- A melhora obtida pode ser tão significativa que não poderia ser obtida simplesmente com o avanço da tecnologia.
- As melhorias nos algoritmos produzem avanços em outras componentes básicas das aplicações (pense nos compiladores, buscadores na internet, etc).

Descrição de algoritmos

Podemos descrever um algoritmo de várias maneiras:

- usando uma linguagem de programação de alto nível: `C`, `Pascal`, `Java` etc
- implementando-o em linguagem de máquina diretamente executável em *hardware*
- em português
- em um pseudo-código de alto nível, como no livro do CLRS

Usaremos essencialmente as duas últimas alternativas nesta disciplina.

Exemplo de pseudo-código

Algoritmo ORDENA-POR-INSERÇÃO: rearranja um vetor $A[1\dots n]$ de modo que fique crescente.

```
ORDENA-POR-INSERÇÃO(A, n)
para j ← 2 até n faça
    chave ← A[j]
    ▷ Insere A[j] no subvetor ordenado A[1 . . . j−1]
    i ← j − 1
    enquanto i ≥ 1 e A[i] > chave faça
        A[i + 1] ← A[i]
        i ← i − 1
    A[i + 1] ← chave
```


Corretude de algoritmos

- Um algoritmo (que resolve um determinado problema) está correto se, para toda instância do problema, ele pára e devolve uma resposta correta.
- Algoritmos incorretos também têm sua utilidade, se soubermos prever a sua probabilidade de erro.
- Neste curso vamos trabalhar apenas com algoritmos corretos.

Complexidade de algoritmos

- Em geral, não basta saber que um dado algoritmo pára. Se ele for muuuuito leeeeeeeento, terá pouca utilidade.
- Queremos projetar/desenvolver algoritmos eficientes (rápidos).
- Mas o que seria uma boa medida de eficiência de um algoritmo?
- Não estamos interessados em quem programou, em que linguagem foi escrito e nem qual máquina foi usada!
- Queremos um **critério uniforme** para comparar algoritmos.

Modelo Computacional

- Uma possibilidade é definir um modelo computacional de um máquina.
- O modelo computacional estabelece quais os recursos disponíveis, as instruções básicas e quanto elas custam (= tempo).
- Dentre desse modelo, podemos estimar através de uma análise matemática o tempo que um algoritmo gasta em função do tamanho da entrada (= análise de complexidade).
- A análise de complexidade depende sempre do modelo computacional adotado.

Máquinas RAM

Salvo mencionado o contrário, usaremos o Modelo Abstrato RAM (Random Access Machine):

- simula máquinas convencionais (de verdade),
- possui um único processador que executa instruções seqüencialmente,
- tipos básicos são números inteiros e reais,
- há um limite no tamanho de cada *palavra de memória*: se a entrada tem "tamanho" n, então cada inteiro/real é representado por $c \log n$ bits para alguma constante $c \geq 1$. (Isto garante que é possível *guardar* o valor de n e *indexar* os elementos individualmente.)

Máquinas RAM

- executa operações aritméticas (soma, subtração, multiplicação, divisão, piso, teto), comparações, movimentação de dados de tipo básico e fluxo de controle (teste *if/else*, chamada e retorno de rotinas) em **tempo constante**,
- Certas operações caem em uma **zona cinza**, por exemplo, exponenciação,
- veja maiores detalhes do modelo RAM no CLRS.

Tamanho da entrada

Problema: Primalidade

Entrada: inteiro n

Tamanho: número de bits de $n \approx \lg n = \log_2 n$

Problema: Ordenação

Entrada: vetor $A[1 \dots n]$

Tamanho: $n \lg U$ onde U é o maior número em $A[1 \dots n]$

Informalmente, o tamanho da entrada é o número de bits necessário para especificar a entrada.

Medida de complexidade e eficiência de algoritmos

- A complexidade de tempo ($=$ eficiência) de um algoritmo é o número de instruções básicas que ele executa em função do tamanho da entrada.
- Adota-se uma "atitude pessimista" e faz-se uma análise de pior caso.
  Determina-se o tempo máximo necessário para resolver uma instância de um certo tamanho.
- Além disso, a análise concentra-se no comportamento do algoritmo para entradas de tamanho GRANDE $=$ análise assintótica.

Medida de complexidade e eficiência de algoritmos

- Um algoritmo é chamado eficiente se a função que mede sua complexidade de tempo é limitada por um **polinômio** no tamanho da entrada.
  Por exemplo: n, $3n - 7$, $4n^2$, $143n^2 - 4n + 2$, n^5.
- Mas por que polinômios?
  (Polinômios são funções bem "comportadas").

Vantagens do método de análise proposto

- O modelo RAM é robusto e permite prever o comportamento de um algoritmo para instâncias GRANDES.
- O modelo permite comparar algoritmos que resolvem um mesmo problema.
- A análise é mais robusta em relação às evoluções tecnológicas .

Desvantagens do método de análise proposto

- Fornece um limite de complexidade pessimista sempre considerando o pior caso.
- Em uma aplicação real, nem todas as instâncias ocorrem com a mesma freqüência e é possível que as "instâncias ruins" ocorram raramente.
- Não fornece nenhuma informação sobre o comportamento do algoritmo no caso médio.
- A análise de complexidade de algoritmos no caso médio é bastante difícil, principalmente, porque muitas vezes não é claro o que é o "caso médio".