MO417 — Complexidade de Algoritmos I

C. C. de Souza C. N. da Silva O. Lee
P. J. de Rezende

5 de agosto de 2010

Antes de mais nada...

- Esses slides são fruto de um trabalho conjunto de Cid C. de Souza, Cândida N. da Silva, Orlando Lee e Pedro J. de Rezende.
- Vários outros professores colaboraram direta ou indiretamente para a preparação do material desses slides. Agradecemos, especialmente (em ordem alfabética):
  - Célia Picinin de Mello
  - José Coelho de Pina
  - Paulo Feofiloff
  - Ricardo Dahab
  - Zanoni Dias
- Mas os erros/imprecisões que persistem devem ser comunicados a mim.

Introdução

O que veremos nesta disciplina?

- Como provar a "corretude" de um algoritmo
- Estimar a quantidade de recursos (temp, 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 $A[1 \ldots n]$ é crescente se $A[1] \leq \ldots \leq A[n]$.

Problema: rearranjar um vetor $A[1 \ldots n]$ 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
  - ...
- Humm, vamos projetar um novo *game* neste curso?
  Não!

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! Interprete isso como um desafio para inteligência humana.
- Esses problemas tem a característica notável de que se um deles admitir um algoritmo "eficiente" então todos admitem algoritmos "eficientes".
- Por que devo me preocupar com problemas $\mathcal{NP}$-sei-lá-o-quê?
  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 indentificar quando estamos lidando com um problema $\mathcal{NP}$-completo!

Algoritmos e tecnologia

- O Éden: 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 **muita** diferença ter um **bom** 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 em 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 "meia-boca". Executa $50n \log n$ instruções.

Algoritmos e tecnologia

- O que acontece quando ordenamos um vetor de um milhão 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 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 $A[1 \ldots n]$ de modo que fique crescente.

```
ORDENA-POR-INSERÇÃO(A, n)
1  para j ← 2 até n faça
2      chave ← A[j]
3      ▷ Insere A[j] no subvetor ordenado A[1 . . . j−1]
4      i ← j − 1
5      enquanto i ≥ 1  e A[i] > chave faça
6          A[i + 1] ← A[i]
7          i ← i − 1
8      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 muito leeeeeeeeeeeento 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 a 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 onde $c \geq 1$ é uma contante.

  Isto é razoável?

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 ficam 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 $A[1 \ldots n]$

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

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?
  Resposta padrão: (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 robustas 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".

Começando a trabalhar

Ordenação

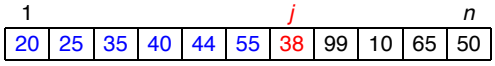
Problema: ordenar um vetor em ordem crescente

Entrada: um vetor $A[1 \ldots n]$

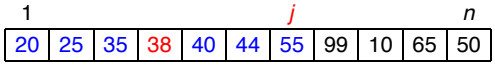
Saída: vetor $A[1 \ldots n]$ rearranjado em ordem crescente

Vamos começar estudando o algoritmo de ordenação baseado no método de inserção.

Inserção em um vetor ordenado

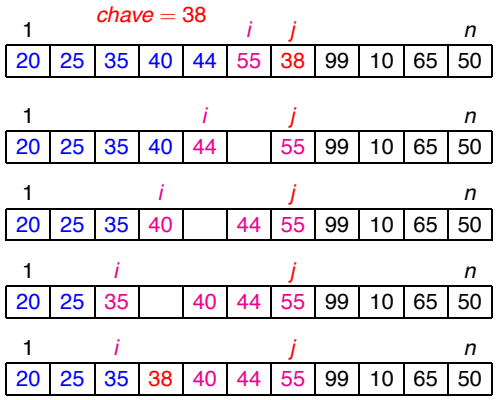


- O subvetor $A[1 \ldots j-1]$ está ordenado.
- Queremos inserir a $chave = 38 = A[j]$ em $A[1 \ldots j-1]$ de modo que no final tenhamos:



- Agora $A[1 \ldots j]$ está ordenado.

Como fazer a inserção



Ordenação por inserção

chave	1							j		n
99	20	25	35	38	40	44	55	99	10	65	50

chave	1							j		n
99	20	25	35	38	40	44	55	99	10	65	50

chave	1								j	n
10	20	25	35	38	40	44	55	99	10	65	50

chave	1								j	n
10	10	20	25	35	38	40	44	55	99	65	50

Ordenação por inserção

chave	1								j	n
65	10	20	25	35	38	40	44	55	99	65	50

chave	1								j	n
65	10	20	25	35	38	40	44	55	65	99	50

chave	1									j
50	10	20	25	35	38	40	44	55	65	99	50

chave	1									j
50	10	20	25	35	38	40	44	50	55	65	99

Ordena-Por-Inserção

Pseudo-código

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


Análise do algoritmo

O que é importante analisar ?

- Finitude: o algoritmo pára?
- Corretude: o algoritmo faz o que promete?
- Complexidade de tempo: quantas intruções são necessárias no pior caso para ordenar os n elementos?

O algoritmo pára

```
ORDENA-POR-INSERÇÃO(A, n)
1  para j ← 2 até n faça
   ...
4      i ← j − 1
5      enquanto i ≥ 1  e A[i] > chave faça
6          ...
7          i ← i − 1
8      ...
```

No laço **enquanto** na linha 5 o valor de i diminui a cada iteração e o valor inicial é $i = j - 1 \geq 1$. Logo, a sua execução pára em algum momento por causa do teste condicional $i \geq 1$.

O laço na linha 1 evidentemente pára (o contador j atingirá o valor $n + 1$ após $n - 1$ iterações).

Portanto, o algoritmo **pára**.

Ordena-Por-Inserção

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

O que falta fazer ?

- Verificar se ele produz uma resposta correta.
- Analisar sua complexidade de tempo.