Projeto e Análise de Algoritmos

Introdução

Atílio G. Luiz

Primeiro Semestre de 2024

Agradecimentos

A maioria dos exemplos dos slides e exercícios adicionais foram ou criados e gentilmente cedidos pelo prof. Cid Carvalho de Souza e pela profa. Cândida Nunes da Silva (particularmente com modificações do prof. Orlando Lee); ou criados e gentilmente cedidos pelos professores Flávio Keidi Miyazawa e Lehilton Pedrosa. Eu recriei ou reestruturei o conteúdo das unidades, possivelmente introduzindo erros, que devem ser reportados a mim.

O conjunto de slides de cada unidade do curso será disponibilizado como guia de estudos e deve ser usado unicamente para revisar as aulas. Para estudar e praticar, leia o livro-texto indicado e resolva os exercícios sugeridos.

Atilio

Introdução à análise de algoritmos

Objetivos

O que veremos nesta disciplina?

1. Demonstrar correção de um algoritmo
   - como ter certeza de que a saída está correta
   - como convencer outras pessoas disso

2. Analisar a complexidade de um algoritmo
   - como estimar a quantidade de recursos utilizados
   - recursos podem ser tempo, memória, acesso a rede etc.

Objetivos

3. Utilizar técnicas conhecidas de projeto de algoritmos
   - divisão e conquista, programação dinâmica etc.
   - utilizar recursão adequadamente

4. Entender a dificuldade intrínseca de alguns problemas
   - inexistência de algoritmos eficientes
   - identificar os problemas intratáveis

Problemas computacionais

Um problema computacional é uma relação entre um conjunto de instâncias e um conjunto de soluções:

- uma instância é um conjunto de valores conhecidos
- uma solução é um conjunto de valores a computar
- cada instância corresponde a uma ou mais soluções

Exemplo de problema: teste de primalidade

Problema: determinar se um dado número é primo

- instâncias: números naturais
- soluções: sim ou não

Exemplo:

- Instância: 9411461
- Solução: sim

Exemplo:

- Instância: 8411461
- Solução: não

Exemplo de problema: ordenação

Problema: ordenar os elementos de um vetor

- instâncias: conjunto de vetores de inteiros
- soluções: conjunto de vetores de inteiros em ordem crescente

Exemplo:

- Instância:

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

- Solução:

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

Algoritmos

Um algoritmo é uma sequência finita de instruções que

- recebe uma instância de um problema computacional
- devolve uma solução correspondente à instância recebida

Observações:

- a instância recebida é chamada de entrada
- a solução devolvida é chamada de saída
- toda instrução deve ser bem definida

Descrição de algoritmos

Podemos escrever um algoritmo de várias maneiras:

- em uma linguagem de programação, como C, Pascal, Java, Python...
- em português, ou outra língua natural
- em pseudocódigo, como no livro de CLRS

Usaremos apenas as duas últimas opções!

Exemplo de pseudocódigo

Um algoritmo para o problema da ordenação:

```
Insertion-Sort(A, n)
1   para j = 2 até n faça
2       chave = A[j]
3       i = j − 1
4       enquanto i ≥ 1 e A[i] > chave faça
5           A[i + 1] = A[i]
6           i = i − 1
7       A[i + 1] = chave
```

Más práticas

- Não misture código com pseudocódigo:

```
for (i = 0; i < n; i++) ...
```

```
if (A[i] >= chave)
    break
```

- Não escreva frases confusas:

```
se A[i] > chave então
    troque as posições dos elementos
```

```
chave = pega o próximo elemento
i = procura posição da chave
```

- Evite algoritmos complicados ou com muitas variáveis:

```
se j = 1 então
    A[2] = A[1]
    A[1] = chave
senão
    enquanto i ≥ 1 e A[i] ≥ chave faça
        ...
```

Modelo computacional

Só podemos escrever instruções bem definidas:

- o resultado de cada instrução é inambíguo e depende somente do estado corrente da execução
- deve ser possível executar cada instrução usando o computador adotado

O conjunto de instruções permitidas é determinado pelo que chamamos de modelo computacional

Máquinas RAM

Usaremos o Modelo Abstrato RAM (Random Access Machine)

- simula máquinas convencionais
- possui um único processador sequencial
- tipos básicos são números inteiros e pontos flutuantes
- cada *palavra de memória* tem tamanho limitado, i.e., valores não podem ser arbitrários

Instruções elementares

- operações aritméticas como soma, subtração, produto...
- acesso direto às posições da memória
- comandos de fluxo de controle (**se**, **enquanto**...)

Operações como exponenciação não são elementares.

Correção de algoritmos

Um algoritmo está correto se:

1. só utiliza instruções do modelo de computação adotado,
2. termina para toda instância do problema e
3. devolve uma solução que correspondente à instância recebida.

Ao escrever um algoritmo, sempre devemos

1. Testar o algoritmo
   - com uma ou mais instâncias de exemplo
   - executando ou simulando o algoritmo
2. Demonstrar que o algoritmo está correto
   - escrever uma prova formal genérica
   - vale para toda instância do problema

Complexidade de algoritmos

- Nem sempre é viável executar um algoritmo
  - pode levar anos ou séculos para terminar
  - pode precisar de mais memória do que há disponível

- Queremos
  1. projetar algoritmos eficientes
  2. comparar algoritmos diferentes

- Para isso, precisamos medir a complexidade do algoritmo
  - independentemente de quem programou,
  - da linguagem em foi escrito
  - e da máquina a ser usada

Modelo computacional e complexidade

Vamos estimar o tempo de execução de um algoritmo

- contamos o número de instruções elementares executadas
- supomos que cada instrução elementar consome um tempo contante

A análise de complexidade depende sempre do modelo computacional adotado!

- o modelo computacional RAM é realista
- seu conjunto de instruções elementares é compatível com os computadores modernos
- é suficientemente genérico para as diferentes arquiteturas

Definição de tamanho da entrada

Um parâmetro importante: tamanho da entrada

- quantidade de bits necessários para codificar a entrada.

Quantos bits k são necessários para codificar um inteiro positivo n?

- Como 2^{k-1} é o menor valor n com k bits,
  $2^{k-1} \leq n < 2^k \quad \Rightarrow k-1 \leq \log_2 n < k \quad \Rightarrow k = \lfloor \log_2 n \rfloor + 1 \approx \log_2 n$
- Ex.: Entrada com valor 1 bilhão tem tamanho
  $k = \lfloor \log_2 10^9 \rfloor + 1 = 30$ bits.

Se algoritmo recebe valor n como entrada e gasta $T(k) = n$ passos, seu tempo de execução é linear?

- Não, pois $T(k) = n \approx 2^k$ (exponencial em k)

Definição de tamanho da entrada

Considere os algoritmos:

- P_1: soma os N inteiros em um array
- P_2: testa se um número N é primo, checando se N é divisível por $2, 3, \ldots, N-1$

Os dois realizam da ordem de $N-1$ operações (adições ou divisões) P_1 é considerado *viável*, enquanto P_2 *inviável*. Por quê?

- Supondo inteiros de 64 bits, o tamanho da entrada para P_1 vale $k = 64N$.
  Concluímos que $T_1(k) = N - 1 = \frac{1}{64}k - 1$ (linear no tamanho da entrada).
- Em P_2, o tamanho da entrada é o número de bits de N, ou seja, $k \approx \log_2 N$.
  Concluímos que $T_2(k) = N - 1 \approx 2^k - 1$. (exponencial no tamanho da entrada)

Definição de tamanho da entrada

Definições equivalentes: produzem aproximadamente uma constante vezes o total de bits

Ex.: Quantidade de dígitos de um inteiro n

- Quantidade de bits $\approx \log_2 n$
- Quantidade de dígitos $\approx \log_{10} n$

$$\log_{10} n = \left(\frac{1}{log_2 10} \right) \cdot log_2 n$$

Definição de tamanho da entrada

Definições equivalentes: produzem aproximadamente uma constante vezes o total de bits

Ex.: Tupla $\langle x_1, x_2, \ldots, x_n \rangle$ com n elementos, cada um com aproximadamente C bits

- Total de bits $= C \cdot n$
- Então podemos usar n como tamanho da entrada
- Ex.: Vetor de inteiros, cada um com 32 bits
- Ex.: strings, cada caractere com 8 bits

Análise assintótica e de pior caso

Consideramos apenas instâncias grandes

- o número de instruções normalmente cresce com o tamanho da entrada n
- instâncias com tamanho limitado por constante gastam tempo constante

Fazemos apenas análise de pior caso

- restringimos a entradas com um dado tamanho n
- consideramos apenas uma instância para a qual o algoritmo executa o maior número de instruções

Denotamos por $T(n)$ o número de instruções executadas no pior caso para entradas de tamanho n

Características e limitações

Esse tipo de análise de complexidade:

- normalmente estima bem tempo de execução real
- permite comparar diversos algoritmos para um problema
- continua relevante mesmo com evoluções tecnológicas

Limitações:

- é uma análise pessimista do tempo de execução
- em certas aplicações, certas instâncias ocorrem mais frequentemente que um pior caso
- não fornece informação sobre tempo de execução médio

Começando a trabalhar

Ordenação

Problema: ordenar os elementos de um vetor

- Entrada: vetor $A[1 \ldots n]$
- Saída: rearranjo de $A[1 \ldots n]$ em ordem crescente

Vamos começar revendo a ordenação por inserção.

Inserção em um vetor ordenado

Ideia do algoritmo

- suponha que o subvetor $A[1 \ldots j-1]$ já está ordenado.
- vamos inserir o $A[j]$ para que $A[1 \ldots j]$ fique ordenado
- o valor do elemento a ser inserido é chamado de *chave*

Antes de inserir:

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

Após inserir:

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

Inserindo uma chave

$chave = 38$

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

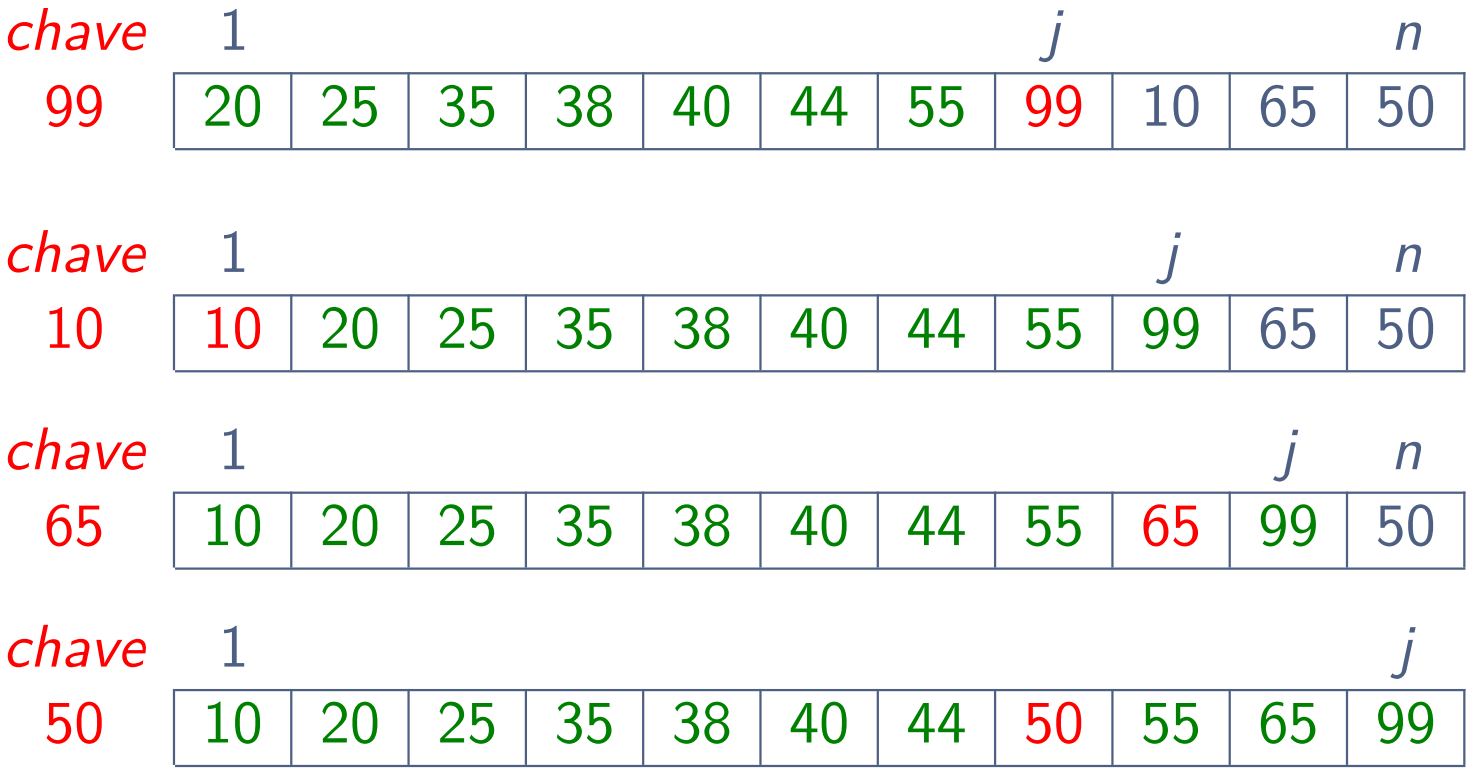
1				i		j				n
20	25	35	40	44		55	99	10	65	50

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

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

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

Ordenando por inserção



Pseudocódigo de Insertion-Sort

```
Insertion-Sort(A, n)
1   para j = 2 até n faça
2       chave = A[j]
3       i = j − 1
4       enquanto i ≥ 1 e A[i] > chave faça
5           A[i + 1] = A[i]
6           i = i − 1
7       A[i + 1] = chave
```

Análise do algoritmo

O que é importante analisar?

- Correção
  - já testamos o algoritmo com um exemplo
  - por enquanto, suponha que o algoritmo está correto
  - vamos mostrar que ele está correto depois

- Complexidade de tempo
  - considere vetores com n elementos
  - quantas instruções são executadas?

Contando o número de instruções

Insertion-Sort(A, n)	Custo	Qnts vezes?
1 **para** $j = 2$ **até** n **faça**	c_1	n
2 $\quad chave = A[j]$	c_2	$n-1$
3 $\quad i = j-1$	c_3	$n-1$
4 $\quad$ **enquanto** $i \geq 1$ **e** $A[i] > chave$ **faça**	c_4	$\sum_{j=2}^{n} t_j$
5 $\quad\quad A[i+1] = A[i]$	c_5	$\sum_{j=2}^{n}(t_j - 1)$
6 $\quad\quad i = i-1$	c_6	$\sum_{j=2}^{n}(t_j - 1)$
7 $\quad A[i+1] = chave$	c_7	$n-1$

- a linha k executa um número constante de instruções c_k
- cada linha executa uma ou mais vezes
- quantas vezes a linha 4 executa depende da entrada
  - seja t_j quantas vezes **enquanto** executa para um certo j

Tempo de execução total

- Considere uma instância de tamanho n
- Seja $T(n)$ o número de instruções executadas para ela
- Basta somar para todas as linhas

$$T(n) = c_1 \cdot n + c_2 \cdot (n-1) + c_3 \cdot (n-1) + c_4 \cdot \sum_{j=2}^{n} t_j + c_5 \cdot \sum_{j=2}^{n} (t_j - 1) + c_6 \cdot \sum_{j=2}^{n} (t_j - 1) + c_7 \cdot (n-1)$$

Observações:

- entradas do mesmo tamanho têm tempos diferentes
- vamos considerar diferentes instâncias
  - melhor caso: quando $T(n)$ é o menor possível
  - pior caso: quando $T(n)$ é o maior possível

Melhor caso

Um melhor caso ocorre quando $t_j = 1$ para cada j

- basta que a condição do **enquanto** sempre falhe
- ocorre se a entrada A já vem ordenada

Nesse caso:

$$\begin{aligned} T(n) &= c_1 \cdot n + c_2 \cdot (n-1) + c_3 \cdot (n-1) + c_4 \cdot (n-1) + c_7 \cdot (n-1) \\ &= (c_1 + c_2 + c_3 + c_4 + c_7) \cdot n - (c_2 + c_3 + c_4 + c_7) \\ &= a \cdot n + b \end{aligned}$$

- os valores de a e b são constantes
- o tempo de execução no melhor caso é linear em n

Pior caso

Um pior caso ocorre quando t_j é máximo para cada j

- basta que a condição do **enquanto** só falha quando $i = 0$
- nessa situação, teremos $t_j = j$
- ocorre se a entrada A vem ordenada decrescentemente

Relembre que

- $\sum_{j=2}^{n} j = n(n+1)/2 - 1$ e $\sum_{j=2}^{n} (j-1) = n(n-1)/2$

Pior caso (cont)

Substituindo, temos

$$
\begin{aligned}
T(n) &= c_1 \cdot n + c_2 \cdot (n-1) + c_3 \cdot (n-1) + c_4 \cdot (n(n+1)/2 - 1) + \\
&\quad c_5 \cdot n(n-1)/2 + c_6 \cdot n(n-1)/2 + c_7 \cdot (n-1) \\
&= (c_4/2 + c_5/2 + c_6/2) \cdot n^2 + \\
&\quad (c_1 + c_2 + c_3 + c_4/2 - c_5/2 - c_6/2 + c_7) \cdot n - \\
&\quad (c_2 + c_3 + c_4 + c_7) \\
&= a \cdot n^2 + b \cdot n + c
\end{aligned}
$$

- os valores de a, b, c são constantes
- o tempo de execução no pior caso é quadrático em n

Complexidade assintótica

Estamos interessados principalmente

- na análise de pior caso
- no tempo de execução para instâncias grandes

Comportamento assintótico

- no pior caso temos $T(n) = an^2 + bn + c$
  - o termo dominante é o que contém n^2
  - o tempo de execução é uma função quadrática
  - as constantes a, b, c só dependem da implementação
- não nos preocupamos com os valores de a, b, c

Por que isso é razoável?

Um exemplo de função quadrática

Considere a função $3n^2 + 10n + 50$

n	$3n^2 + 10n + 50$	$3n^2$
64	12978	12288
128	50482	49152
512	791602	786432
1024	3156018	3145728
2048	12603442	12582912
4096	50372658	50331648
8192	201408562	201326592
16384	805470258	805306368
32768	3221553202	3221225472

- quando n é grande, o termo $3n^2$ é uma boa estimativa
- podemos nos concentrar nos termos dominantes

Notação assintótica

Como simplificar o tempo de pior caso de INSERTION-SORT?

- ao invés de escrever $T(n) = an^2 + bn + c$,
- escrevemos somente $T(n) = \Theta(n^2)$

Essa notação significa que, para n suficientemente grande,

1. $T(n)$ é limitada superiormente por $c \cdot n^2$, para algum $c > 0$
2. $T(n)$ é limitada inferiormente por $d \cdot n^2$, para algum $d > 0$

Vamos formalizar essa notação assintótica mais adiante!

Começando a trabalhar

- Conclusões

Conclusões

Algumas conclusões:

- Projetar algoritmos melhores pode levar a ganhos extraordinários de desempenho.
- Isso é tão importante quanto o projeto de hardware.
- O ganho obtido ao melhorar a complexidade de um algoritmo não poderia ser obtida simplesmente com o avanço da tecnologia.
- Queremos estudar principalmente algoritmos fundamentais, que produzem avanços em outras componentes básicas das aplicações (pense nos compiladores, buscadores na internet, classificadores, etc).

Começando a trabalhar

- Exercícios

Exercícios

1. Reescreva o procedimento INSERTION-SORT para ordenar em ordem não crescente, em vez da ordem não decrescente.

2. Considere o problema de busca:
   - **Entrada:** Uma sequência de n números $A = \langle a_1, a_2, \ldots, a_n \rangle$ e um valor v.
   - **Saída:** Um índice i tal que $v = A[i]$ ou o valor especial NIL, se v não aparecer em A.

   Escreva o pseudocódigo para busca linear, que faça a varredura da sequência, procurando por v.

Exercícios

3. Considerando a busca linear do exercício anterior, quantos elementos da sequência de entrada precisam ser verificados em média, considerando que o elemento que está sendo procurado tenha a mesma probabilidade de ser qualquer elemento no arranjo? E no pior caso? Quais são os tempos de execução do caso médio e do pior caso da busca linear em notação Θ?

Exercícios

4. Considere a ordenação de n números armazenados no arranjo A, localizando primeiro o menor elemento de A e permutando esse elemento com o elemento contido em $A[1]$. Em seguida, determine o segundo menor elemento de A e permute-o com $A[2]$. Continue dessa maneira para os primeiros $n-1$ elementos de A.

   Escreva o pseudocódigo para esse algoritmo, conhecido como **ordenação por seleção**. Forneça os tempos de execução do melhor caso e do pior caso da ordenação por seleção em notação Θ.

Exercícios

5. Considere o problema de somar dois inteiros binários de n bits, armazenados em dois arranjos de n elementos A e B. A soma dos dois inteiros deve ser armazenada em forma binária em um arranjo de $(n+1)$ elementos C. Enuncie o problema formalmente e escreva o pseudocódigo para somar os dois inteiros.