Projeto e Análise de Algoritmos

Professor Anderson Namen

Baseado nos slides de apoio disponíveis no site da Editora McGraw-Hill e em material da Unicamp elaborado pelo professor Flávio Keidi Miyazawa

Bibliografia

Algoritmos – Teoria e Prática (tradução da 2ª. Edição americana)
Cormen, Leiserson, Rivest & Stein
Ed. Campus

Projeto de Algoritmos – Fundamentos, análise e exemplos da Internet
Goodrich & Tamassia
Bookman

Eficiência de Algoritmos

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.

Eficiência de Algoritmos

- O que acontece quando ordenamos um vetor de um milhão de elementos? **Qual algoritmo é mais rápido?**
- Algoritmo 1 na máquina *A*:
$$\frac{2.(10^6)^2 \text{ instruções}}{10^9 \text{ instruções/segundo}} \approx 2000 \text{ segundos}$$
- Algoritmo 2 na máquina *B*:
$$\frac{50.(10^6 \log 10^6) \text{ instruções}}{10^7 \text{ instruções/segundo}} \approx 100 \text{ segundos}$$
- 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!

Eficiência de Algoritmos

- 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).

Eficiência de Algoritmos

- Modelo usado – RAM (random-access machine)

- Não confundir com memória RAM

- Simula um único processador que executa instruções sequencialmente (não há concorrência)

- Executa operações aritméticas, comparações, movimentações de dados e fluxo de controle (teste if/else e chamadas de rotinas) em tempo constante

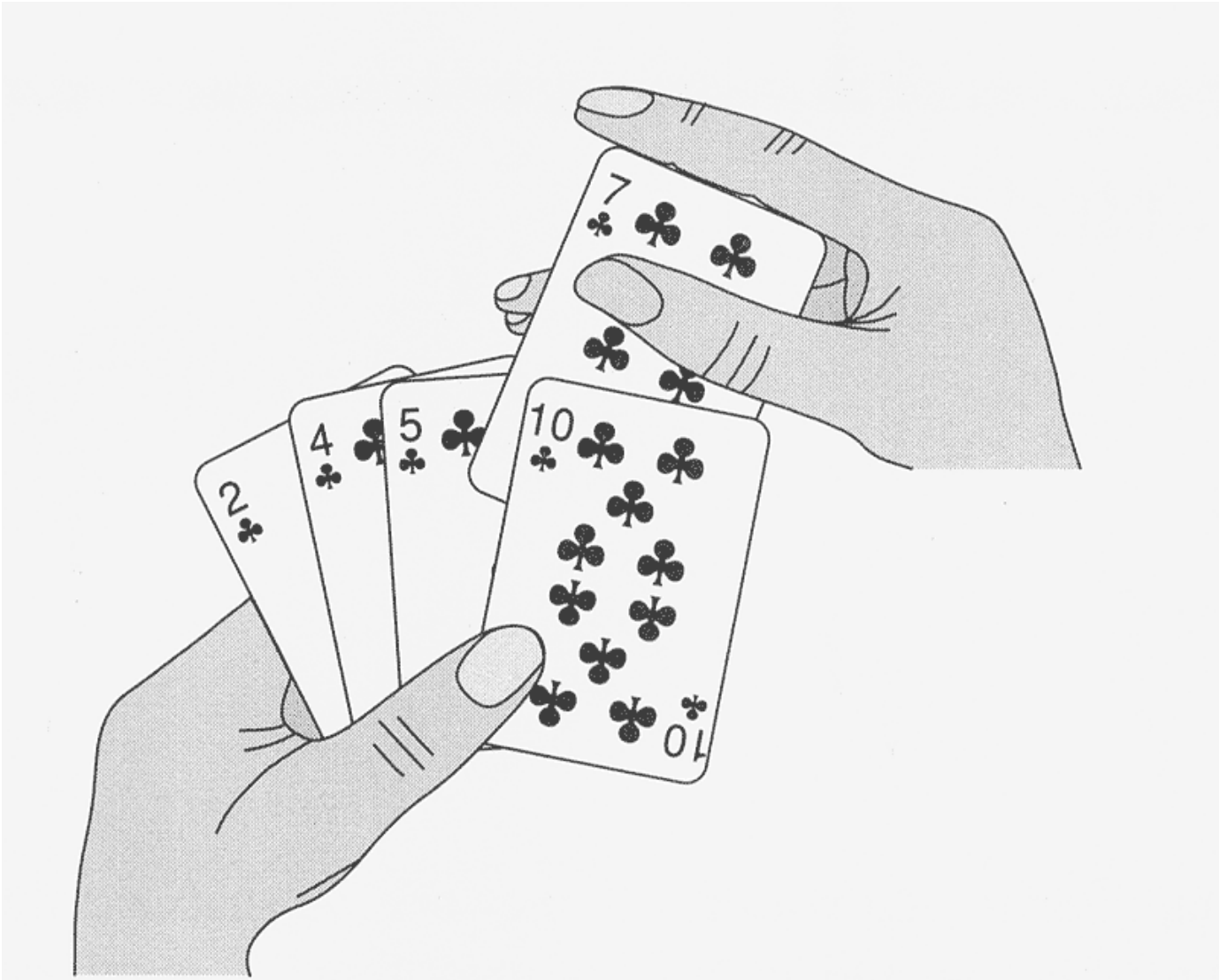

Figure 2.1 Sorting a hand of cards using insertion sort.

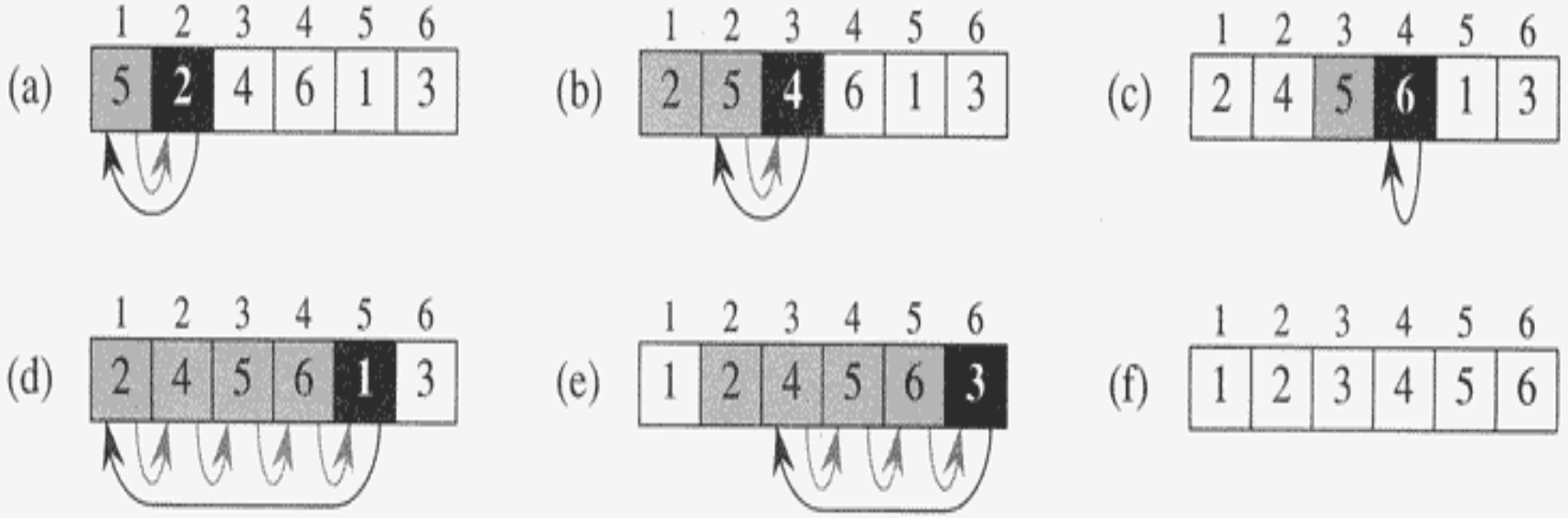

Figure 2.2 The operation of INSERTION-SORT on the array $A = \langle 5, 2, 4, 6, 1, 3 \rangle$. Array indices appear above the rectangles, and values stored in the array positions appear within the rectangles. **(a)–(e)** The iterations of the **for** loop of lines 1–8. In each iteration, the black rectangle holds the key taken from $A[j]$, which is compared with the values in shaded rectangles to its left in the test of line 5. Shaded arrows show array values moved one position to the right in line 6, and black arrows indicate where the key is moved to in line 8. **(f)** The final sorted array.

```
INSERTION-SORT(A)
1  for j ← 2 to length[A]
2       do key ← A[j]
3          ▷ Insert A[j] into the sorted sequence A[1 .. j − 1].
4          i ← j − 1
5          while i > 0 and A[i] > key
6               do A[i + 1] ← A[i]
7                  i ← i − 1
8          A[i + 1] ← key
```

Loop invariants and the correctness of insertion sort

	INSERTION-SORT(A)	*cost*	*times*
1	**for** $j \leftarrow 2$ **to** $length[A]$	c_1	n
2	**do** $key \leftarrow A[j]$	c_2	$n-1$
3	▷ Insert $A[j]$ into the sorted sequence $A[1..j-1]$.	0	$n-1$
4	$i \leftarrow j-1$	c_4	$n-1$
5	**while** $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
6	**do** $A[i+1] \leftarrow A[i]$	c_6	$\sum_{j=2}^{n}(t_j-1)$
7	$i \leftarrow i-1$	c_7	$\sum_{j=2}^{n}(t_j-1)$
8	$A[i+1] \leftarrow key$	c_8	$n-1$

$$
\begin{aligned}
T(n) = {} & c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j \\
& + c_6 \sum_{j=2}^{n}(t_j - 1) + c_7 \sum_{j=2}^{n}(t_j - 1) \\
& + c_8(n-1)
\end{aligned}
$$

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j \\ &\quad + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) \\ &\quad + c_8(n-1) \end{aligned}$$

Temos então que

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5 \left(\frac{n(n+1)}{2} - 1 \right) \\ &\quad + c_6 \left(\frac{n(n-1)}{2} \right) + c_7 \left(\frac{n(n-1)}{2} \right) + c_8(n-1) \\ &= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2} \right) n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8 \right) n \\ &\quad - (c_2 + c_4 + c_5 + c_8) \end{aligned}$$

O tempo de execução no pior caso é da forma $an^2 + bn + c$ onde a, b, c são constantes que dependem apenas dos c_i.

Portanto, **no pior caso**, o tempo de execução é uma **função quadrática** no **tamanho da entrada**.

- O algoritmo Ordena-Por-Inserção tem como complexidade (de pior caso) uma função quadrática $an^2 + bn + c$, onde a, b, c são constantes absolutas que dependem apenas dos custos c_i.
- O estudo assintótico nos permite "jogar para debaixo do tapete" os valores destas constantes, i.e., aquilo que independe do tamanho da entrada (neste caso os valores de a, b e c).

Considere a função quadrática $3n^2 + 10n + 50$:

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

Como se vê, $3n^2$ é o termo dominante quando n é grande.

De um modo geral, podemos nos concentrar nos termos dominantes e esquecer os demais.

Eficiência de Algoritmos

• A notação O é muito usada para caracterizar o tempo de execução em função de um parâmetro n.

• Usando a notação O, podemos descrever o tempo de execução de um algoritmo apenas inspecionando a estrutura global do algoritmo.

• Insertion sort é da ordem $O(n^2)$

• Pensa-se sempre no pior caso (limite superior de tempo de execução)

Eficiência de Algoritmos

Algoritmo vetorMax(A,n)

Entrada: um vetor A com n>=1 elementos inteiros

Saída: o maior elemento em A

currentMax ← A[0]

para i ← 1 até n-1 faça

se currentMax < A[i} então

currentMax ← A[i]

retorne currentMax

Eficiência de Algoritmos

currentMax ← A[0] // (n° constante de operações)

para i ← 1 até n-1 faça

// (i inicializado com 1)

// (teste de i realizado n vezes)

    se currentMax < A[i} então // (teste feito n-1 vezes)

        currentMax ← A[i] // (no máximo n-1 vezes)

retorne currentMax // (1 única operação)

Portanto é da ordem O(n)

Eficiência de Algoritmos

Algoritmo recursiveMax(A,n)

Entrada: um vetor A com n>=1 elementos inteiros

Saída: o maior elemento em A

se n = 1 então

retorne A[0]

3 operações

retorne max{recursiveMax(A,n-1),A[n-1]}

Chamado n-1 vezes

Executa n° constante de operações a cada chamada

Conclusão: O(n)

Eficiência de Algoritmos

Algoritmo mediaPrefixada(X)

Entrada: um vetor X com n>=1 elementos

Saída: um vetor A com n elementos tal que A[i] é a média de X[0], X[1],…,X[i]

```
para i ← 0 até n-1 faça
        a ← 0
        para j ← 0 até i faça
                a ← a + X[j]
        A[i] ← a/(i+1)
retorne vetor A
```

O tempo de execução tem qual ordem?

Eficiência de Algoritmos

Algoritmo mediaPrefixada2(X)

Entrada: um vetor X com n>=1 elementos

Saída: um vetor A com n elementos tal que A[i] é a média de X[0], X[1],…,X[i]

```
s ← 0
para i ← 0 até n-1 faça
        s ← s + X[i]
        A[i] ← s/(i+1)
retorne vetor A
```

O tempo de execução tem qual ordem?