Universidade Federal de Goiás – Instituto de Informática

Análise e Projeto de Algoritmos
Introdução à Análise de Algoritmos

Diogo Stelle
com slides do Prof. Fábio H. V. Martinez
diogostelle@inf.ufg.br / diogo.stelle@gmail.com

2019

UFG
INSTITUTO DE INFORMÁTICA
UFG

Conteúdo da Aula

- Roteiro
- Convenções na descrição de algoritmos
- Ordenação por inserção
- Análise de algoritmos
- Projeto de algoritmos
- Exercícios

Visão Geral da Aula

- visão geral da estrutura usada para projeto e análise dos algoritmos
- pseudocódigo usado nos algoritmos
- ordenação por inserção:
  - correção
  - tempo de execução

Convenções na descrição de algoritmos

Linguagem Usada

- Pseudocódigo
  - palavras-chave em português
  - similar a C, C++, Java, Python, Pascal
  - clareza e concisão
  - questões de engenharia de software são negligenciadas arbitrariamente

Linguagem Usada

- Pseudocódigo
    - indentação é usada para indicar blocos de instruções
    - estruturas condicionais: **se, senão**
    - estruturas de repetição: **enquanto, para, repita enquanto**
    - Símbolo **** indica que o restante da linha é um comentário
    - atribuição múltipla: **i = j = e**
    - variáveis são locais aos procedimentos

Linguagem Usada

- Pseudocódigo
    - **A[i]** é usado para indicar o **i-ésimo** elemento do vetor **A**; **A[i..j]** é usado para indicar um intervalo de **j−i+1** elementos do vetor **A**
    - informações são organizadas em objetos compostos de atributos: por exemplo, **A.length**
    - uma variável representando um vetor ou um objeto é, na verdade, um ponteiro para os dados que representam o vetor ou o objeto

Linguagem Usada

- Pseudocódigo
  - parâmetros de um procedimento são sempre passados **por valor**, a menos de vetores e objetos
  - a sentença **devolva** imediatamente termina o procedimento sendo executado e transfere o controle para o ponto onde esse procedimento foi chamado; **devolva** permite devolução de mais que um valor
  - os operadores **E** e **OU** são operadores lógicos
  - sentença **erro** termina a execução do algoritmo sem especificação de como proceder devido a um erro

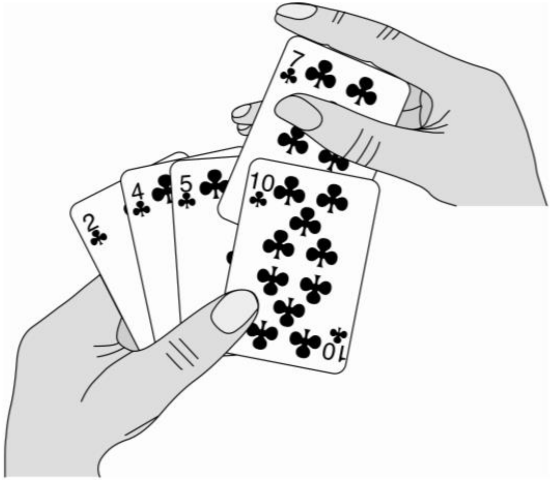
Ordenação por inserção

Problema da Ordenação

- o algoritmo de ordenação por inserção soluciona o problema da ordenação:
- **PROBLEMA DA ORDENAÇÃO**
    - **Entrada:** uma sequência de **n** números $\{a_1; a_2;...; a_n\}$
    - **Saída:** uma permutação $\{a'_1; a'_2;...; a'_n\}$ da sequência de entrada de tal forma que $a'_1 \leq a'_2 \leq ... \leq a'_n$
- os elementos ou chaves da sequência estão armazenados em um vetor

Ideia do algoritmo

- o algoritmo de ordenação por inserção funciona da forma como muitas pessoas ordenam uma mão de cartas de baralho



- usa a técnica incremental de projeto de algoritmos: tendo ordenado o vetor **A[1..j−1]**, o elemento **A[j]** é inserido em seu local correto e obtemos então o vetor **A[1..j]** ordenado

Algoritmo

```
INSERTION-SORT(A)
    para j = 2 até A.length
        chave = A[j]
        // insere A[j] na sequência ordenada A[1..j − 1]
        i = j − 1
        enquanto i > 0 E A[i] > chave
            A[i + 1] = A[i]
            i = i − 1
        A[i + 1] = chave
```

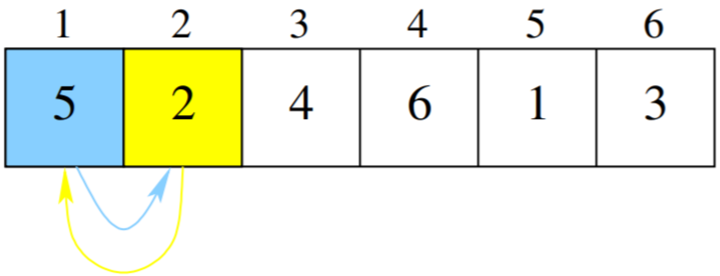
Funcionamento do algoritmo

1	2	3	4	5	6
5	2	4	6	1	3

Funcionamento do algoritmo

1	2	3	4	5	6
5	2	4	6	1	3

Funcionamento do algoritmo



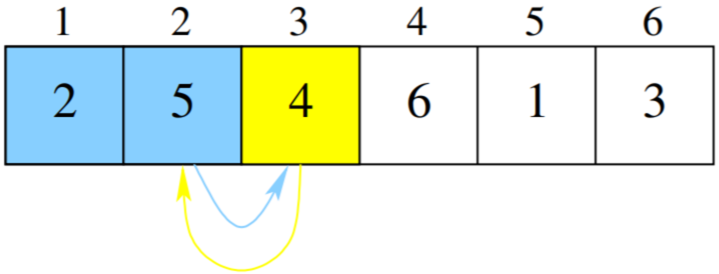
Funcionamento do algoritmo

1	2	3	4	5	6
2	5	4	6	1	3

Funcionamento do algoritmo

1	2	3	4	5	6
2	5	4	6	1	3

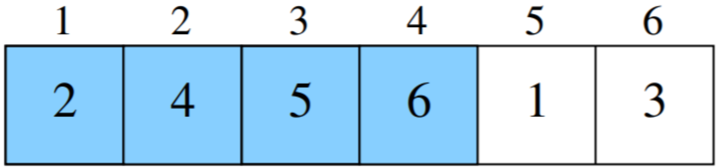
Funcionamento do algoritmo



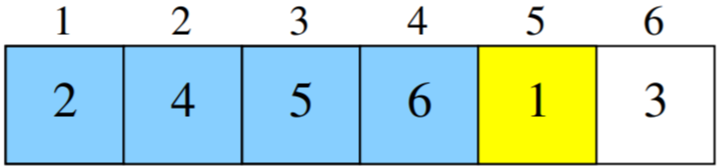
Funcionamento do algoritmo

1	2	3	4	5	6
2	4	5	6	1	3

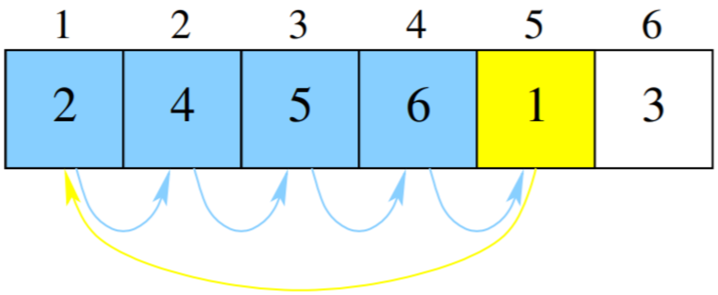
Funcionamento do algoritmo



Funcionamento do algoritmo



Funcionamento do algoritmo



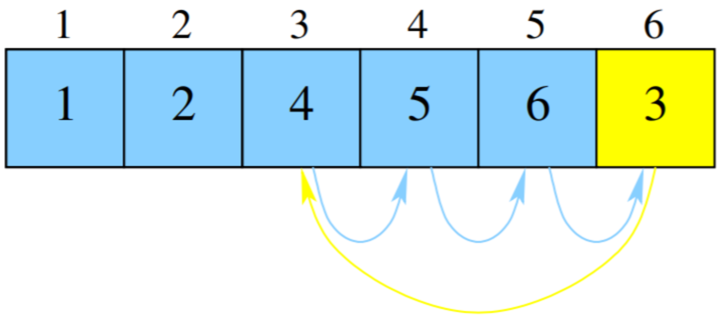
Funcionamento do algoritmo

1	2	3	4	5	6
1	2	4	5	6	3

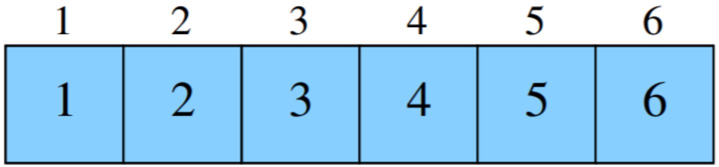
Funcionamento do algoritmo

1	2	3	4	5	6
1	2	4	5	6	3

Funcionamento do algoritmo



Funcionamento do algoritmo



Correção do algoritmo

- Invariantes e a correção do InsertionSort
  - o índice **j** indica a "carta atual" a ser inserida na mão esquerda
  - no início de cada iteração da estrutura de repetição para, o vetor consistindo dos elementos **A[1..j−1]** constitui a atual mão esquerda com as cartas ordenadas
  - o restante do vetor **A[j+1..n]** consiste da pilha de cartas ainda na mesa
  - os elementos em **A[1..j−1]** são aqueles que originalmente estavam armazenados neste intervalo do vetor, mas agora estão ordenados

Correção do algoritmo

- Invariantes e a correção do InsertionSort
  - esta propriedade do vetor **A[1..j−1]** pode ser descrita como um **invariante**:
    - no começo de cada iteração da estrutura de repetição para das linhas **1–8**, o vetor **A[1..j−1]** consiste dos elementos originalmente em **A[1..j−1]**, mas em ordem crescente
  - usamos invariantes para ajudar a compreender por que um algoritmo está correto

Correção do algoritmo

- devemos mostrar três propriedades sobre um invariante:
  - **Inicialização**: é verdadeiro antes da primeira iteração da estrutura de repetição;
  - **Manutenção**: se é verdadeiro antes de uma iteração da estrutura de repetição, permanece verdadeiro antes da próxima iteração;
  - **Término**: quando a estrutura de repetição termina, o invariante nos dá uma propriedade útil que nos permite mostrar que o algoritmo está correto
- similar à indução matemática

Correção do algoritmo

```
INSERTION-SORT(A)
    para j = 2 até A.length
        chave = A[j]
        // insere A[j] na sequência ordenada A[1..j − 1]
        i = j − 1
        enquanto i > 0 E A[i] > chave
            A[i + 1] = A[i]
            i = i − 1
        A[i + 1] = chave
```

Correção do algoritmo

- Invariantes e a correção do InsertionSort
    - **Inicialização**: antes da primeira iteração da estrutura de repetição **para**, temos que **j = 2**; dessa forma, o vetor **A[1..j−1]** consiste de um único elemento **A[1]**, que é de fato o elemento original em **A[1]**; além disso, **A[1]** está ordenado, o que mostra que o invariante é verdadeiro antes da primeira iteração;

Correção do algoritmo

- Invariantes e a correção do InsertionSort
    - **Manutenção**: o corpo de instruções da estrutura de repetição para trabalha para movimentar os elementos **A[j−1]**, **A[j−2]**,... uma posição para direita, até encontrar a posição correta para **A[j] (linhas 4–7)**, momento em que insere no local correto do vetor **A** o valor de **A[j] (linha 8)**; o vetor A[1..j] consiste então dos elementos originalmente em **A[1..j]**, mas rearranjados em ordem crescente; incrementar **j** para a próxima iteração da estrutura de repetição para preserva assim o invariante;

Correção do algoritmo

- Invariantes e a correção do InsertionSort
  - **Término**: a condição de parada da estrutura de repetição para é **j > A.length = n**; como cada iteração incrementa **j** em uma unidade, temos que **j = n+1**; substituindo **n+1** no lugar de **j** no invariante, temos que o vetor **A[1..n]** consiste dos elementos originalmente em **A[1..n]**, mas rearranjados em ordem crescente; o vetor **A[1..n]** é o vetor completo, isto é, o vetor de entrada está rearranjado em ordem crescente; portanto, o algoritmo está correto

Generalidades

- **analisar** um algoritmo significa predizer os recursos que o algoritmo requer
- em geral, queremos medir o tempo computacional de um algortimo, mas eventualmente também podemos nos preocupar com memória, largura de banda de comunicação, etc
- antes de analisar um algoritmo, devemos ter um modelo da tecnologia em que o mesmo será implementado, com seus custos associados

Generalidades

- fixaremos um modelo de implementação genérico de um processador, chamado **máquina de acesso aleatório (RAM)**, onde os algoritmos serão transformados em programas
- cada instrução (aritmética, de movimentação de dados e de controle) no modelo RAM tem custo **constante**
- os tipos de dados são números inteiros e números de ponto flutuante, com limite no número de bytes de armazenamento
- analisar um algoritmo no modelo RAM pode ser um desafio

Generalidades

- ferramentas matemáticas necessárias podem incluir combinatória, teoria das probabilidades, destreza algébrica e habilidade de identificar os termos mais significativos em fórmulas
- como o comportamento de um algoritmo pode ser diferente para cada entrada possível, precisamos de uma maneira de resumir seu comportamento em uma fórmula simples e compreensível

Análise de algoritmos

Análise do InsertionSort

- o tempo gasto pelo algoritmo InsertionSort depende da entrada: ordenar mil números é mais demorado que ordenar três números
- além disso, o algoritmo pode gastar diferentes quantidades de tempo para ordenar duas sequências de entrada de mesmo tamanho, dependendo de quão ordenadas elas já estão
- em geral, o tempo gasto por um algoritmo cresce com o tamanho de sua entrada e, por isso, é usual descrever o tempo de execução de um programa como uma função do tamanho de sua entrada

Análise do InsertionSort

- **tamanho da entrada** depende do problema: número de itens de entrada ou número total de bits necessários para representar a entrada; às vezes, mais de um número representa a entrada
- **tempo de execução** de um algoritmo sobre uma entrada particular é o número de operações primitivas ou passos executados
- uma quantidade de tempo constante é necessária para executar cada linha de um algoritmo
- isto é, a execução da *i*-ésima linha do algoritmo gasta tempo $\mathbf{c}_i$, onde $\mathbf{c}_i$ é uma constante

Análise do InsertionSort

INSERTION-SORT(A)	custo	vezes
para $j = 2$ **até** $A.length$	c_1	n
chave = $A[j]$	c_2	$n-1$
// insere $A[j]$ na sequência ordenada $A[1..j-1]$	0	$n-1$
$i = j - 1$	c_4	$n-1$
enquanto $i > 0$ **E** $A[i] >$ chave	c_5	$\sum_{j=2}^{n} t_j$
$A[i+1] = A[i]$	c_6	$\sum_{j=2}^{n}(t_j - 1)$
$i = i - 1$	c_7	$\sum_{j=2}^{n}(t_j - 1)$
$A[i+1]$ = chave	c_8	$n-1$

Análise do InsertionSort

- o tempo de execução do InsertionSort é então a soma dos tempos de execução de cada sentença executada:

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

$$+ c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1) .$$

Análise do InsertionSort

- mesmo para entradas de um dado tamanho, o tempo de execução de um algoritmo pode depender de quais entradas deste tamanho são fornecidas
- no InsertionSort, o melhor caso ocorre quando o vetor de entrada é fornecido em ordem crescente
- neste caso, para cada $j = 2,3,\ldots,n$, temos que $A[i] \leq chave$ na linha 5, quando i foi inicializado com $j-1$

Análise do InsertionSort

- assim, t_j = 1 para todo j = 2,3,...,n e o tempo de execução do melhor caso do InsertionSort é:

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) \\ &= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8) \end{aligned}$$

- podemos expressar este tempo de execução como **an+b**, para constantes a e b que dependem exclusivamente dos custos das instruções c_i ; ou seja, uma **função linear de n**

Análise do InsertionSort

- o pior caso para o InsertionSort ocorre quando o vetor de entrada é fornecido em ordem decrescente
- neste caso, todo elemento A[j] deve ser comparado com cada elemento do vetor ordenado A[1..j−1]
- assim, $t_j = j$ para todo $j = 2,3,\ldots,n$

Análise do InsertionSort

- assim, no pior caso, o tempo de execução do InsertionSort é:

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

Análise do InsertionSort

- podemos expressar este tempo de execução de pior caso como **$an^2 + bn + c$** para constantes a, b e c que dependem dos custos das instruções c_i
- uma função **quadrática de n**

Casos na análise de algoritmos

- Melhor caso, pior caso ou caso médio?
  - usualmente nos concentramos apenas no tempo de execução de pior caso
  - é o maior tempo gasto para qualquer entrada de tamanho n, isto é, fornece um limitante superior do tempo de execução para qualquer entrada de tamanho n
  - o pior caso ocorre muito frequentemente para muitos algoritmos
  - o “caso médio” é frequentemente tão ruim quanto o pior caso

Caso médio

- Análise de caso médio
    - **análise probabilística**
    - o escopo da análise de caso médio é limitado, porque muitas vezes não sabemos o que significa a entrada “média” para um problema particular
    - consideramos então que todas as entradas de um determinado tamanho são igualmente prováveis
    - na prática esta condição pode ser violada, mas em alguns casos podemos usar um **algoritmo aleatorizado**, que faz escolhas aleatórias, para permitir uma análise probabilística que fornece um tempo de execução **esperado**

Funções que representam tempo de execução

Funções que representam tempo de execução

- Ordem de crescimento
  - usamos algumas abstrações simplificadas para facilitar a análise de um algoritmo
  - por exemplo, ignoramos o custo real de cada instrução e usamos constantes $\mathbf{c_i}$ para representar esses custos
  - além dos custos reais de cada instrução, acabamos por ignorar também os custos abstratos $\mathbf{c_i}$: expressamos o tempo de execução de pior caso do InsertionSort como $\mathbf{an^2 + bn + c}$, para constantes **a,b** e **c** que dependem dos custos das instruções $\mathbf{c_i}$
  - a **taxa de crescimento** ou **ordem de crescimento** do tempo de execução é o que realmente nos interessa, o que nos leva a fazer mais uma simplificação

Funções que representam tempo de execução

- Ordem de crescimento
  - consideramos apenas o termo dominante de uma fórmula, por exemplo an^2 , já que os outros termos menores são praticamente insignificantes para grandes valores de **n**
  - também ignoramos o coeficiente constante do termo dominante, já que fatores constantes são menos significativos que a taxa de crescimento na determinação da eficiência computacional para grandes entradas
  - no InsertionSort, quando ignoramos termos de menor ordem e o coeficiente constante do termo dominante, sobra o termo n^2

Funções que representam tempo de execução

- Ordem de crescimento
  - dizemos então que o InsertionSort tem tempo de execução de pior caso **$\Theta(n^2)$**
  - consideramos um algoritmo mais eficiente que outros se seu tempo de execução de pior caso tem a menor taxa de crescimento

Exercícios

Exercícios

- 2.1-1 Ilustre o funcionamento do InsertionSort sobre o vetor A = {31, 41, 59, 26, 41, 58}
- 2.1-2 Reescreva o algoritmo InsertionSort para rearranjar uma sequência de entrada em ordem descrescente
- 2.1-3 Considere o seguinte problema:

problema da busca

- Entrada: uma sequência de n números $A = \{a_1, a_2, \ldots, a_n\}$ e um valor v
- Saída: um índice i tal que v = A[i] ou um valor especial $k \notin \{1,\ldots,n\}$ indicando que v não ocorre em A

Escreva um pseudocódigo para a busca linear, que percorre a sequência em busca de v. Usando um invariante, prove que o algoritmo está correto.

Exercícios

- Considere a ordenação de n números armazenados em um vetor A encontrando primeiro o menor elemento de A e trocando-o com o elemento A[1]. Então encontre o segundo menor elemento de A e troque-o com A[2]. Continue desta maneira para os primeiros $n-1$ elementos de A. Escreva um pseudocódigo para este algoritmo, que é conhecido como **ordenação por seleção**. Qual invariante o algoritmo mantém? Por que o algoritmo precisa ser executado somente para os primeiros $n-1$ elementos, ao invés de para todos os n elementos?