Capítulo 2
Ordenação Por Inserção

Manoel Ribeiro Filho

Introdução

Começaremos com o algoritmo de ordenação por inserção:
Apresentando um “pseudocódigo”
Demonstrando que ele efetua a ordenação corretamente
Analisando seu tempo de execução, essa análise introduzirá uma notação que focaliza o modo como o tempo aumenta com o número de itens a ordenar

Depois estudaremos outros algoritmos, todos muito simples, sempre fazendo:
Apresentando um “pseudocódigo”
Demonstrando que ele funciona corretamente
Analisando seu tempo de execução

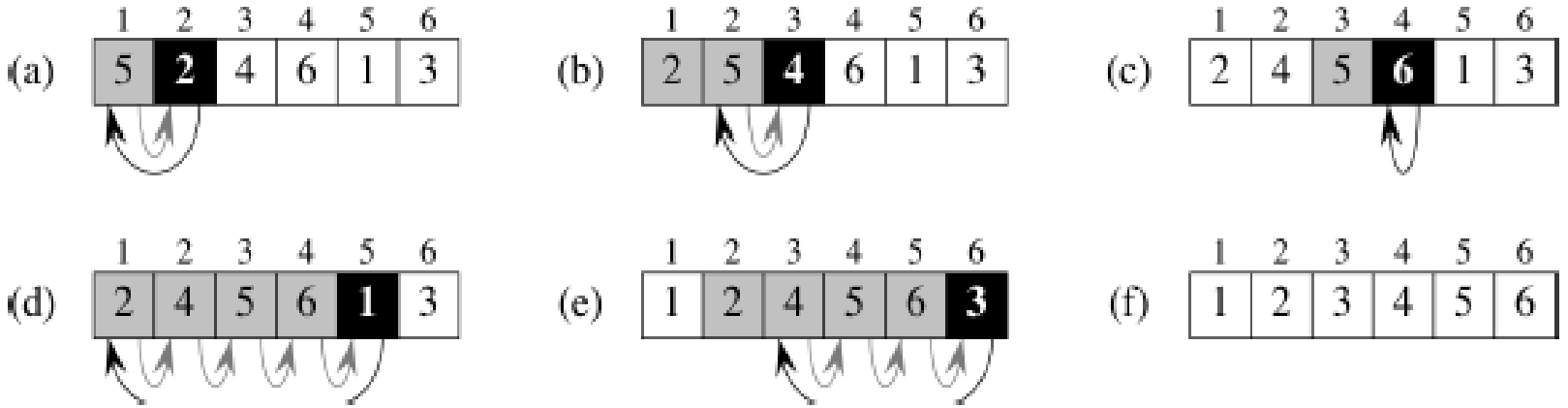
Na primeira prova teremos uma questão, onde a resposta será apresentar os três itens de vermelho citados acima

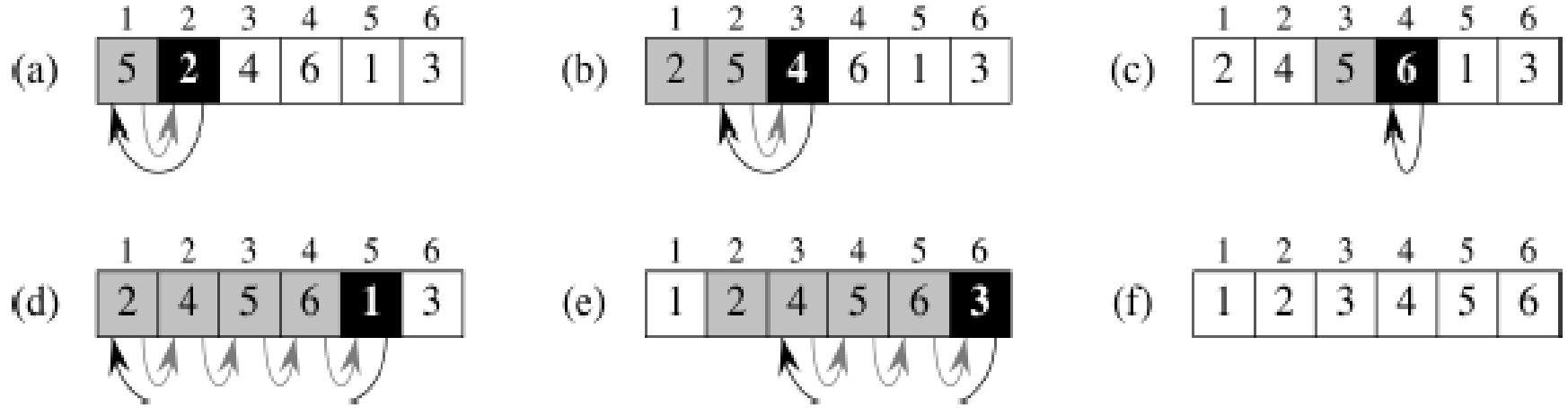
2.1 Projeto do Algoritmo

Eficiente para ordenar um número pequeno de elementos.
Semelhante a ordenar as cartas em um jogo de baralho. Inicialmente a mão esquerda está vazia, e as cartas viradas para baixo, em cima da mesa.
Em seguida, retiramos uma carta de cada vez da mesa e a inserimos na posição correta na mão esquerda. Para encontrar a posição correta para uma carta, nós a comparamos com cada uma das cartas que já estão na mão, da direita para a esquerda. Em todas as vezes, as cartas que seguramos na mão esquerda são ordenadas, e essas cartas eram as que
estavam na parte superior da pilha sobre a mesa.

Nosso pseudocódigo para ordenação por inserção é apresentado como um procedimento denominado Insertion- Sort, que toma como parâmetro um arranjo *A*[1 .. *n*] contendo uma sequência de comprimento *n* que deverá ser ordenada. (No código, o número *n* de elementos em *A* é denotado por *A*·*comprimento*. O arranjo de entrada *A* conterá a sequência de saída ordenada quando Insertion-Sort terminar.

```
INSERTION-SORT(A)
1   for j = 2 to A·comprimento
2       chave = A[j]
3       // Inserir A[j] na sequência ordenada A[1.. j − 1].
4       i = j − 1
5       while i > 0 e A[i] > chave
6           A[i + 1] = A[i]
7           i = i − 1
8       A[i + 1] = chave
```





A operação de Insertlon-sort sobre o arranjo $A = \langle 5, 2, 4, 6, 1, 3 \rangle$. Os índices do arranjo aparecem acima dos retângulos, e os valores armazenados nas posições do arranjo aparecem dentro dos retângulos. **(a)**–**(e)** Iterações do laço **for** das linhas 1 a 8. Em cada iteração, o retângulo preto contém a chave obtida de $A[j]$, que é comparada com os valores contidos nos retângulos sombreados à sua esquerda, no teste da linha 5. Setas sombreadas mostram os valores do arranjo deslocados uma posição para a direita na linha 6, e setas pretas indicam para onde a chave é deslocada na linha 8. (**f**) O arranjo ordenado final.

Análise 1: O algoritmo funciona?

A melhor maneira de mostrar que um algoritmo iterativo faz o que promete é exibir um bom *invariante* do processo iterativo

Invariantes de laço e a correção da ordenação por inserção

No início de cada iteração do laço **for**, indexado por j, o subarranjo que consiste nos elementos $A[1 .. j - 1]$ constitui a mão ordenada atualmente e o subconjunto remanescente

$A[j + 1 .. n]$ corresponde à pilha de cartas que ainda está sobre a mesa.

Na verdade, os elementos $A[1 .. j - 1]$ são os que estavam *originalmente* nas posições 1 a $j - 1$, mas agora em sequência ordenada. Afirmamos essas propriedades de $A[1 .. j - 1]$ formalmente como um de ***invariante de laço***.

No início de cada iteração para o laço **for** das linhas 1–8, o subarranjo $A[1 .. j – 1]$ consiste nos elementos que estavam originalmente em $A[1 .. j – 1]$, porém em sequência ordenada.

Nos passos de (a) a (f), verifique o vetor de A[1...j-1] é crescente, o que é um invariante do processo.

Usamos invariantes de laço para nos ajudar a entender por que um algoritmo é correto.
Devemos mostrar três detalhes sobre um invariante de laço:

Inicialização: Ele é verdadeiro antes da primeira interação do laço

Manutenção: Permanecerá verdadeiro em cada interação do laço

Término: Quando o laço termina, o invariante de laço permanece verdadeiro.

Vamos ver que as três propriedades são válidas para o algoritmo de inserção apresentado

Inicialização: Começamos mostrando que o invariante de laço é válido antes da primeira iteração do laço, quando j = 2. Nesse momento, o subarranjo A[1...j-1] consiste apenas do único elemento A[1], portanto está trivialmente ordenado.

Manutenção: Em seguida, abordamos a segunda propriedade: mostrar que cada interação mantém o invariante do laço. São os passos de (a) a (e).

Término: Finalmente examinamos o que ocorre quando o laço termina. A condição de término do laço é j> n. Como em cada interação do laço j aumenta de 1, isso ocorre quando j=n+1, que é o passo (f), onde o subarranjo A[1...n] é o arranjo inteiro, que está ordenado. Portanto o algoritmo está correto.

Empregaremos esse método de invariantes de laço para mostrar que um algoritmo está correto. É uma técnica muito importante, pois mostra teoricamente a correção de um algoritmo.

Convenções do pseudocódigo

* O recuo indica estrutura de bloco. Por exemplo, o corpo do laço for que começa na linha 1 consiste nas linhas 2 a 8, e o corpo do laço **while** que começa na linha 5 contém as linhas 6 e 7, mas não a linha 8. Nosso estilo de recuo também se aplica a instruções **if-else.**2 O uso de recuo em vez de indicadores convencionais de estrutura de bloco, como instruções **begin** e **end**, reduz bastante a confusão, ao mesmo tempo que preserva ou até mesmo aumenta a clareza.
* As interpretações das construções de laço **while**, **for** e **repeat-until** e das construções condicionais **if-else** são semelhantes às das linguagens C, C++, Java, Python e Pascal. O contador do laço mantém seu valor após sair do laço, ao contrário de algumas situações que surgem em C++, Java e Pascal. Desse modo, logo depois de um laço **for**, o valor do contador de laço é o valor que primeiro excedeu o limite do laço **for**. Usamos essa propriedade em nosso argumento de correção para a ordenação por inserção. O cabeçalho do laço **for** na linha 1 é **for** $j = 2$ **to** $A{\cdot}comprimento$ e, assim, quando esse laço termina, $j = A{\cdot}comprimento + 1$ (ou, o que é equivalente, $j = n + 1$, visto que $n = A{\cdot}comprimento$). Usamos a palavra-chave **to** quando um laço **for** incrementa seu contador do laço a cada iteração, e usamos a palavra-chave **downto** quando um laço **for** decrementa seu contador de laço. Quando o contador do laço mudar por uma quantidade maior do que 1, essa quantidade virá após a palavra-chave opcional **by**

Convenções do pseudocódigo

- O símbolo “//” indica que o restante da linha é um comentário.
- Uma atribuição múltipla da forma i=j=e atribui as variáveis i e j o valor da expressão e; ela deve ser tratada como equivalente à atribuição j=e seguida pela atribuição i=j.
- Elementos de arranjos (vetores) são acessados especificando-se o nome do arranjo seguido pelo índice entre colchetes. Por exemplo, A[i] indica o i-ésimo elemento do arranjo A. A notação “..” é usada para indicar uma faixa de valores dentro de um arranjo. Desse modo, A[1..j] indica o subarranjo de A que consiste nos j elementos A[1],A[2],...A[j]
- Os operadores booleanos “e” e “ou” são operadores com curto-circuito. Isto é, quando avaliamos a expressão “x e y”, avaliamos primeiro x. Se x for avaliada como FALSO, a expressão inteira não poderá ser avaliada como VERDADEIRO, e assim não avaliamos y. Se, por outro lado, x for avaliado como VERDADEIRO, teremos que avaliar y para determinar o valor da expressão inteira. De modo semelhante, na expressão “x ou y”, avaliamos a expressão y somente se x for avaliado como FALSO.

Lista de exercícios 1

Da página 16, do livro clrs, fazer
2.1-1
2.1-2
2.1-3

2.2 Análise do Algoritmo

Analisar um algoritmo significa prever os recursos de que o algoritmo necessita, ocasionalmente memória, largura de banda de comunicação ou hardware de computador são a principal preocupação, porém mais frequentemente é o **tempo de computação** que desejamos medir. Então podemos declarar...

Análise 2: Quanto tempo consome?

A pergunta fundamental da análise de algoritmos é *Quanto tempo o algoritmo consome?* À primeira vista, a pergunta não faz sentido porque o tempo depende

1. da instância do problema, ou seja, do particular vetor $A[1..n]$ sendo ordenado e
2. da máquina (computador) sendo usada.

Para responder à primeira objeção, digo que estou interessado *no pior caso*: para cada valor de n, considero a instância $A[1..n]$ para a qual o algoritmo consome mais tempo. Vamos denotar esse tempo por $T(n)$.

Para responder à segunda objeção, observo que o tempo não depende tanto assim do computador. Ao mudar de um computador para outro, o consumo de tempo do algoritmo é apenas multiplicado por uma constante. Por exemplo, se $T(n) = 250n^2$ em um computador, então $T(n) = 500n^2$ em um computador duas vezes mais lento e $T(n) = 25n^2$ em um computador dez vezes mais rápido.

Modelo Computacional

Antes de podermos analisar um algoritmo, devemos ter um modelo da tecnologia de implementação que será usada, inclusive um modelo para os recursos dessa tecnologia e seus custos. Consideraremos um modelo de computação genérico de máquina de acesso aleatório (***random-access machine***, ***RAM*****)** com um único processador como nossa tecnologia de implementação e entenderemos que nossos algoritmos serão implementados como programas de computador. No modelo de RAM, as instruções são executadas uma após outra, sem operações concorrentes.

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.
- Note que não podemos representar números reais, a menos de aproximações. Assim, não poderemos representar π, $\sqrt{2}$ etc, de maneira exata.

  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 ficam em uma **zona cinza**, por exemplo, exponenciação,
- veja maiores detalhes do modelo RAM no CLRS.

Computadores reais contêm instruções que não citamos, e tais instruções representam uma área cinzenta no modelo de RAM. Por exemplo, a exponenciação é uma instrução de tempo constante? No caso geral, não; são necessárias várias instruções para calcular x^y quando *x* e *y* são números reais. Porém, em situações restritas, a exponenciação é uma operação de tempo constante.....

No modelo de RAM, não tentamos modelar a hierarquia da memória que é comum em computadores contemporâneos. Isto é, não modelamos caches ou memória virtual.
Os modelos que incluem a hierarquia de memória são bem mais complexos que o modelo de RAM, portanto pode ser difícil utilizá-los.

Porém, as análises do modelo de RAM em geral permitem previsões excelentes do desempenho em máquinas reais.

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 complicada e depende do conhecimento da distribuição das instâncias.

Até mesmo a análise de um algoritmo simples no modelo de RAM pode ser um desafio. As ferramentas matemáticas exigidas podem incluir análise combinatória, teoria das probabilidades, destreza em álgebra e a capacidade de identificar os termos mais significativos em uma fórmula.

Tendo em vista que o comportamento de um algoritmo pode ser diferente para cada entrada possível, precisamos de um meio para resumir esse comportamento em fórmulas simples, de fácil compreensão

Em geral, o tempo gasto por um algoritmo cresce com o tamanho da entrada; assim, é tradicional descrever o tempo de execução de um programa em função do tamanho de sua entrada.

Para isso, precisamos definir os termos “**tempo de execução**” e “**tamanho da entrada**” com mais cuidado.

A melhor noção para ***tamanho da entrada*** depende do problema que está sendo estudado. No caso de muitos problemas, como a ordenação ou o cálculo de transformações discretas de Fourier, a medida mais natural é o *número de itens na entrada* — por exemplo, o tamanho *n* do arranjo para ordenação.

Para muitos outros problemas, como a multiplicação de dois inteiros, a melhor medida do tamanho da entrada é o *número total de bits* necessários para representar a entrada em notação binária comum.

Às vezes, é mais apropriado descrever o tamanho da entrada com dois números em vez de um. Por exemplo, se a entrada para um algoritmo é um grafo, o tamanho da entrada pode ser descrito pelos números de vértices e arestas no grafo. Indicaremos qual medida de tamanho da entrada está sendo usada com cada problema que estudarmos.

O ***tempo de execução*** de um algoritmo em determinada entrada é o número de operações primitivas ou “passos” executados. É conveniente definir a noção de passo de modo que ela seja tão independente de máquina quanto possível. Por enquanto, vamos adotar a visão a seguir.

Uma quantidade de tempo constante é exigida para executar cada linha do nosso pseudo código. Uma linha pode demorar uma quantidade de tempo diferente de outra linha, mas consideraremos que cada execução da *i*-ésima linha leva um tempo *ci*, onde *ci* é uma constante. Esse ponto de vista está de acordo com o modelo de RAM e também reflete o modo como o pseudo-código seria implementado na maioria dos computadores reais

Na discussão a seguir, nossa expressão para o tempo de execução do algoritmo de inserção evoluirá de uma fórmula confusa que utiliza todos os custos de instrução *ci* até uma notação muito mais simples, que também é mais concisa e mais fácil de manipular.

Essa notação mais simples também facilitará a tarefa de determinar se um algoritmo é mais eficiente que outro.

Começaremos apresentando o procedimento Insertion-Sort com o "custo" de tempo de cada instrução e o número de vezes que cada instrução é executada. Para cada $j = 2, 3, ..., n$, onde $n = A \cdot comprimento$, seja t_j o número de vezes que o teste do laço **while** na linha 5 é executado para aquele valor de j. Quando um laço **for** ou **while** termina da maneira usual (isto é, devido ao teste no cabeçalho do laço), o teste é executado uma vez mais do que o corpo do laço. Consideramos que comentários não são instruções executáveis e, portanto, não demandam nenhum tempo.

INSERTION-SORT(A)	*custo*	*vezes*
1 **for** $j = 2$ **to** A·*comprimento*	c_1	n
2 $chave = A[j]$	c_2	$n-1$
3 //Inserir $A[j]$ na sequência ordenada $A[1..j-1]$.	0	$n-1$
4 $i = j-1$	c_4	$n-1$
5 **while** $i > 0$ e $A[i] > chave$	c_5	$\sum_{j=2}^{n} t_j$
6 $A[i+1] = A[i]$	c_6	$\sum_{j=2}^{n} (t_j - 1)$
7 $i = i-1$	c_7	$\sum_{j=2}^{n} (t_j - 1)$
8 $A[i+1] = chave$	c_8	$n-1$

O tempo de execução do algoritmo é a soma dos tempos de execução para cada instrução executada; uma instrução que demanda c_i passos para ser executada e é executada *n* vezes contribuirá com $c_i n$ para o tempo de execução total. Para calcular $T(n)$, o tempo de execução de Insertion-Sort de uma entrada de *n* valores, somamos os produtos das *colunas custo* e *vezes*, obtendo

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

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

Tempo de Execução do Melhor Caso

O melhor caso ocorre se o arranjo já está ordenado. Então, para cada $j = 2, 3, ..., n$, descobrimos que $A[i] \leq chave$ na linha 5, que significa que o algoritmo nunca vai entrar no loop while, resultando que o termo $c_5 \sum_{j=2}^{n} tj$.

Seja igual a $c_5(n-1)$

E que $C_6 = C_7 = 0$

E o tempo de execução do melhor caso 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 esse tempo de execução como an + b, para constantes a e b que dependem dos custos de cada ci . Portanto o tempo de execução no melhor caso é uma função linear de n.

Tempo de Execução do Pior Caso

Se o arranjo estiver ordenado em ordem inversa — ou seja, em ordem decrescente —, resulta **o pior caso**.

Devemos comparar cada elemento $A[j]$ com cada elemento do subarranjo ordenado inteiro, $A[1 .. j - 1]$, e então $tj = j$ para 2, 3, ..., n. Observando que

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

e

$$\sum_{j=2}^{n} (j-1) = \frac{n(n-1)}{2}$$

Apêndice A, do clrs, apresenta modos de resolver esses somatórios), descobrimos que, no pior caso, o tempo de execução de Insertion-Sort é....

$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5(n(n+1)/2 -1) + c_6(n(n-1)/2) + c_7(n(n-1)/2) + c_8(n-1)$

T(n) =

$$\left(\frac{c_5}{2}+\frac{c_6}{2}+\frac{c_7}{2}\right)n^2+\left(c_1+c_2+c_4+\frac{c_5}{2}-\frac{c_6}{2}-\frac{c_7}{2}+c_8\right)n$$

$$-(c_2+c_4+c_5+c_8)$$

Podemos expressar esse tempo de execução do pior caso como $an^2 + bn + c$, portanto, ele é uma ***função quadrática*** de *n*.

Análise do pior caso e do caso médio

Em nossa análise da ordenação por inserção, examinamos tanto o melhor caso, no qual o arranjo de entrada já estava ordenado, quanto o pior caso, no qual o arranjo de entrada estava ordenado em ordem inversa.

Porém, em geral, nos concentraremos em determinar apenas o ***tempo de execução do pior caso***; ou seja, o tempo de execução mais longo para *qualquer* entrada de tamanho *n*.

Apresentamos três razões para essa orientação.

1. O tempo de execução do pior caso de um algoritmo estabelece um limite superior para o tempo de execução para qualquer entrada. Conhecê-lo nos dá uma garantia de que o algoritmo nunca demorará mais do que esse tempo. Não precisamos fazer nenhuma suposição sobre o tempo de execução esperando que ele nunca seja muito pior.
2. Para alguns algoritmos, o pior caso ocorre com bastante frequência. Por exemplo, na pesquisa de um banco de dados em busca de determinada informação, o pior caso do algoritmo de busca frequentemente ocorre quando a informação não está presente no banco de dados. Em algumas aplicações, a busca de informações ausentes pode ser frequente.

3. Muitas vezes, o “caso médio” é quase tão ruim quanto o pior caso. Suponha que escolhemos n números aleatoriamente e aplicamos ordenação por inserção. Quanto tempo transcorrerá até que o algoritmo determine o lugar no subarranjo $A[1 .. j - 1]$ em que deve ser inserido o elemento $A[j]$? Em média, metade dos elementos em $A[1 .. j - 1]$ é menor que $A[j]$ e metade dos elementos é maior. Portanto, em média, verificamos metade do subarranjo $A[1 .. j - 1]$ e, portanto, $tj = j/2$. Resulta que o tempo de execução obtido para o caso médio é uma função quadrática do tamanho da entrada, exatamente o que ocorre com o tempo de execução do pior caso.

Em alguns casos particulares, estaremos interessados no tempo de execução do ***caso médio*** de um algoritmo; nesse caso deve-se usar a técnica da ***análise probabilística*** aplicada a vários algoritmos. O escopo da análise do caso médio é limitado porque pode não ser evidente o que constitui uma entrada “média” para determinado problema.

Ordem de Crescimento

Usamos algumas abstrações simplificadoras para facilitar nossa análise do procedimento Insertion-Sort. Primeiro, ignoramos o custo real de cada instrução, usando as constantes *ci* para representar esses custos. Então, observamos que até mesmo essas constantes nos dão mais detalhes do que realmente necessitamos: expressamos o tempo de execução do pior caso como $an^2 + bn + c$ para algumas constantes *a*, *b* e *c* que dependem dos custos de instrução *ci*. Desse modo, ignoramos não apenas os custos reais de instrução, mas também os custos abstratos *ci*.

Agora, faremos mais uma abstração simplificadora.

Ordem de crescimento

É a ***taxa de crescimento***, ou ***ordem de crescimento***, do tempo de execução que realmente nos interessa. Portanto, consideramos apenas o termo inicial de uma fórmula (por exemplo, an^2), já que os termos de ordem mais baixa são relativamente insignificantes para grandes valores de *n*.

Também ignoramos o coeficiente constante do termo inicial, visto que fatores constantes são menos significativos que a taxa de crescimento na determinação da eficiência computacional para grandes entradas. No caso da ordenação por inserção, quando ignoramos os termos de ordem mais baixa e o coeficiente constante do termo inicial, resta apenas o fator de n^2 do termo inicial.

Afirmamos que a ordenação por inserção tem um tempo de execução do pior caso igual a $\Theta(n^2)$ (lido como “teta de *n* ao quadrado”).

Neste capítulo usaremos informalmente a notação Θ e a definiremos com precisão no Capítulo 3.

Em geral, consideramos que um algoritmo é mais eficiente que outro se seu tempo de execução do pior caso apresentar uma ordem de crescimento mais baixa. Devido a fatores constantes e termos de ordem mais baixa, um algoritmo cujo tempo de execução tenha uma ordem de crescimento mais alta pode demorar menos tempo para pequenas entradas do que um algoritmo cuja ordem de crescimento seja mais baixa.
Porém, para entradas suficientemente grandes, um algoritmo $\Theta(n^2)$, por exemplo, será executado mais rapidamente no pior caso que um algoritmo $\Theta(n^3)$.

Lista de Exercícios 2

Da páginas 20 e 21 do clrs fazer
2.2-1
2.2-2
2.2-3