

## Capítulo 3

# Projeto de Algoritmos e Indução Matemática

Um *algoritmo* é uma descrição precisa de um método para a resolução de determinado problema. Este capítulo apresenta a relação entre a prova de teoremas por indução matemática e o projeto de algoritmos. Conforme se verá, é possível extrair de uma prova por indução um algoritmo para a resolução do problema associado ao teorema provado. A partir desse princípio, o método pode também ser utilizado para provar que um algoritmo está correto. O método tem ainda a qualidade adicional de permitir que se compreendam as razões pelas quais um algoritmo de fato encontra a solução para o problema proposto.

### 3.1 Provas Indutivas e Algoritmos

#### 3.1.1 Recursividade e Indução Matemática

Um importante conceito, presente tanto na Matemática quanto na Ciência da Computação, é o de *recursividade*. Em termos gerais, ele se refere a uma definição que é feita em termos de si mesma. Mais concretamente, um programa (ou função, ou procedimento) é dito recursivo quanto tem a possibilidade de fazer chamadas a si mesmo para resolver determinada instância. Evidentemente, o programa deve ter também um *critério de parada* para evitar que as chamadas se acumulem indefinidamente. Nem toda chamada deve gerar uma nova chamada, ou o programa jamais terminaria. Programas não-recursivos são chamados de *iterativos*.

Na Matemática, a recursividade aparece normalmente na forma de relações de recorrência. Pode-se definir o fatorial de um inteiro positivo  $n$  da seguinte forma:

$$n! = \begin{cases} n(n-1)! & \text{se } n > 0 \\ 1 & \text{se } n = 0 \end{cases}$$

Observe que os elementos essenciais da recursividade estão presentes na relação de recorrência. A primeira linha (caso  $n > 0$ ) corresponde à chamada recursiva; a segunda ( $n = 0$ ) é o critério de parada. Essa relação de recorrência pode prontamente ser convertida em um algoritmo recursivo para determinar o fatorial de um inteiro  $n$ , como mostra a figura 3.1.

```

01 function fact (n): integer {
02     if (n = 0) return 1; /*criterio de parada*/
03     else return (n·fact(n - 1)); /*chamada recursiva*/
04 }

```

Figura 3.1: Algoritmo recursivo para o cálculo do fatorial de  $n$

O conceito de recursividade está intimamente relacionado a provas por indução matemática. O passo indutivo pode ser interpretado como a tentativa de se resolver um problema de certo tamanho (determinado pelo parâmetro de indução). Para isso, utiliza-se a hipótese de indução, que nada mais é que a solução de um problema do mesmo tipo, mas de tamanho menor; o princípio é exatamente o mesmo de uma chamada recursiva. O caso base da prova, por sua vez, fornece o critério de parada do algoritmo. Note que provas por indução levam naturalmente a uma implementação recursiva do algoritmo. (No entanto, é importante observar que é possível converter implementações recursivas em iterativas. Em alguns casos, essa é uma operação simples; em outros, extremamente trabalhosa.)

Recursivo ou iterativo, o importante é que um algoritmo pode ser elaborado. Assim sendo, se formos capazes de enunciar um teorema que represente o problema que se deseja resolver e, além disso, provarmos esse teorema por indução matemática, seremos também capazes de construir um algoritmo garantidamente correto para o problema.

### Um Exemplo Simples

[MAX] Dado um conjunto  $E = \{e_1, e_2, \dots, e_n\}$  onde  $e_j \in Z$ ,  $j = 1, \dots, n$ , determinar o maior valor presente em  $E$ .

O teorema mais natural (existem outros, conforme veremos mais adiante) a ser enunciado para a resolução de [MAX] é o de que se sabe resolver uma instância de [MAX] de um tamanho determinado, o que pode ser sempre feito para qualquer problema cujas informações sobre a instância possam ser especificadas por um conjunto discreto. Deste modo, o teorema relacionado a [MAX] para uma instância de tamanho  $n$ ,  $T(n)$  é enunciado como se segue:

**Teorema**  $T(n)$ : *Sabe-se resolver [MAX] para  $|E| = n$ , sendo  $n$  um inteiro positivo.*

**Prova** Por indução simples em  $n$ . O caso base,  $n = 1$ , é trivial: o maior elemento de um conjunto unitário  $E = \{e_1\}$  é  $e_1$ , o único elemento do conjunto.  $T(1)$  é verdadeiro, portanto. Como hipótese de indução, suponha que  $T(k - 1)$  ( $k > 1$ ) seja verdadeiro, ou seja, que saibamos encontrar o maior elemento de um conjunto com  $n = k - 1$  elementos. Devemos provar que isso implica que se sabe encontrar o máximo de um conjunto de cardinalidade  $n = k$ . Seja  $E_k = \{e_1, e_2, \dots, e_{k-1}, e_k\}$  um conjunto qualquer com  $k$  elementos.  $E_k$  pode ser particionado em dois subconjuntos:  $E_{k-1} = \{e_1, e_2, \dots, e_{k-1}\}$  e  $\{e_k\}$ . Como  $E_{k-1}$  tem  $k - 1$  elementos, a hipótese de indução garante que é possível encontrar o índice  $m$  ( $1 \leq m < k$ ) tal que  $e_m$  é um máximo de  $E_{k-1}$ . Sendo  $E_{k-1}$  um subconjunto de  $E_k$ ,  $e_m$  é um candidato a máximo de  $E_k$ ; o outro candidato é  $e_k$ , o único elemento não considerado na hipótese de indução. Para determinar o máximo, basta comparar  $e_m$  e  $e_k$  e escolher o maior deles.  $\square$

Apesar de se tratar de uma prova muito simples, ela pode parecer um tanto trabalhosa para um problema tão trivial. No entanto, ela tem uma particularidade interessante: dela se deriva um algoritmo para a resolução de [MAX] para um conjunto  $E$  de cardinalidade  $n$ , sendo  $n$  qualquer inteiro positivo. E o mais importante: da prova deriva-se um algoritmo garantidamente *correto*.

Devido à própria natureza da prova indutiva, a maneira mais natural de se representar o algoritmo gerado a partir dela é em sua versão recursiva. O uso da hipótese de indução pode ser interpretado como uma chamada recursiva. O caso base da prova indutiva, não por acaso, é o caso base do algoritmo recursivo. A correspondência é óbvia nesse caso, conforme mostra o pseudocódigo da figura 3.2.

```

01 function max ( $E, n$ ): integer {
02   if ( $n = 1$ ) return  $E[1]$ ;          /*caso base*/
03    $M \leftarrow \max (E, n - 1)$ ;      /*uso da hipótese de inducao*/
04   if ( $M > E[n]$ ) return  $M$ ;      /*passo indutivo*/
05   else return  $E[n]$ ;
06 }

```

Figura 3.2: Algoritmo recursivo para [MAX]

Com alguma adaptação, a prova indutiva pode ser transformada também em um algoritmo iterativo (figura 3.3). Nesse caso, resolve-se primeiro o caso base,  $T(1)$ . A partir dessa solução, utiliza-se o passo indutivo para a resolução de  $T(2)$ . Da mesma forma, resolve-se  $T(3)$  a partir de  $T(2)$ ,  $T(4)$  a partir de  $T(3)$  e assim sucessivamente, até que o problema completo,  $T(n)$ , esteja resolvido. Na verdade, isso também é feito na versão recursiva; nesse caso, entretanto, parece mais natural a linha de raciocínio “invertida”: “para resolver  $T(n)$  utiliza-se  $T(n - 1)$ ;  $T(n - 1)$ , por sua vez, é resolvido a partir de  $T(n - 2)$ ; e assim sucessivamente, até que se chegue ao caso base”. Conforme já mencionado na seção ??, ambas são perfeitamente equivalentes.

```

01 function max ( $E, n$ ): integer {
02    $M \leftarrow E[1]$ ;                  /*caso base*/
03   for  $i \leftarrow 2$  to  $n$  do {      /*uso da hipótese de inducao*/
04     if ( $E[i] > M$ )  $M \leftarrow E[i]$ ; /*passo indutivo*/
05   }
06   return  $M$ ;
07 }

```

Figura 3.3: Algoritmo iterativo para [MAX]

No pseudocódigo da figura 3.2, pode-se observar claramente o caso base (linha 2) e a operação básica realizada no passo indutivo (linha 4). O uso da hipótese de indução não é tão claro, mas pode-se considerar que ele está implícito no *loop* iniciado na linha 3, que é o que permite que instâncias progressivamente maiores sejam resolvidas a partir de instâncias menores.

Em geral, para problemas mais simples, é isso o que ocorre quando um algoritmo recursivo é transformado em um algoritmo iterativo: a recursão converte-se em um *loop*. Apesar de o procedimento ser muito simples nesse caso, nem sempre é isso que ocorre. Em particular, a

conversão de algoritmos com mais de uma chamada recursiva normalmente requer a utilização de estruturas de dados especiais para armazenar os resultados parciais obtidos pelo programa. De qualquer forma, fazer essa transformação não é de forma alguma necessário: para resolver um problema, um algoritmo recursivo é tão bom quanto um iterativo. Além disso, tem a vantagem de evidenciar mais claramente o *teorema* a partir do qual foi construído, bem como as suas partes constituintes: caso base, uso da hipótese e passo indutivo.

### 3.1.2 Algoritmo Alternativo

Pode-se dizer que os dois pseudocódigos para [MAX] apresentados até aqui na verdade são diferentes implementações do mesmo algoritmo. Afinal, são derivados da mesma prova para o teorema apresentado. Conforme se verá nesta seção, uma prova diferente do mesmo teorema dá origem a um algoritmo essencialmente distinto dos apresentados.

Consideremos uma prova por indução *forte* do teorema associado a [MAX] (a prova anterior foi por indução simples).

**Prova** Por indução simples em  $n$ . O caso base é o mesmo da prova anterior,  $n = 1$ : o máximo de um conjunto unitário é seu único elemento. Como hipótese de indução, suponha que  $T(n)$  ( $1 \leq n < k$ ) seja verdadeiro, ou seja, que saibamos encontrar o maior elemento de um conjunto com  $n < k$  elementos. Deve-se provar que isso implica que se sabe encontrar o máximo de um conjunto de cardinalidade  $n = k$ . Seja  $E_k = \{e_1, e_2, \dots, e_{k-1}, e_k\}$  um conjunto qualquer com  $k$  elementos.  $E_k$  pode ser particionado em dois subconjuntos menores:  $E_a = \{e_1, e_2, \dots, e_{\lfloor n/2 \rfloor}\}$  e  $E_b = \{e_{\lfloor n/2 \rfloor + 1}, e_{\lfloor n/2 \rfloor + 2}, \dots, e_{k-1}, e_k\}$ . Tanto  $E_a$  quanto  $E_b$  têm menos de  $k$  elementos; portanto, pela hipótese de indução, é possível encontrar os máximos de cada um dos conjuntos ( $M_a$  e  $M_b$ , respectivamente). Como todos os elementos de  $E$  estão distribuídos entre  $E_a$  e  $E_b$ , a máximo de  $E$  estará em  $E_a$  ou em  $E_b$ . Além disso, como ele não é menor que nenhum outro elemento, será o máximo do subconjunto em que estiver. Assim sendo, para determinar o máximo de  $E$ , basta escolher o maior entre  $M_a$  e  $M_b$ .  $\square$

Observe que a divisão do conjunto  $E$  na prova foi feita em duas partes de tamanhos (aproximadamente) iguais. A prova estaria igualmente correta se a divisão não fosse balanceada. Na verdade, qualquer *partição* do conjunto estaria correta: basta que  $E = E_a \cup E_b$ ,  $|E_a| > 0$ ,  $|E_b| > 0$  e  $E_a \cap E_b = \emptyset$ .

A figura 3.4 mostra o algoritmo resultante da prova por indução forte. Repare que, para resolver uma instância, o algoritmo a divide em duas instâncias menores do mesmo problema. Essa estratégia de resolução, muito comum para diversos tipos de problemas, é chamada de *divisão-e-conquista*. A fase da *divisão* corresponde a criação das instâncias que serão passadas como parâmetros nas chamadas recursivas; a *conquista* é a combinação dos resultados fornecidos por essas chamadas.

No caso do problema *max*, tanto a etapa da divisão (cálculo do elemento “do meio” para a separação do vetor em duas partes) quanto a de conquista (escolha do máximo entre as duas respostas) são muito simples. Frequentemente, contudo, é necessário realizar algum esforço computacional em cada uma dessas etapas.

```

01 function max (E, left, right): integer {
02   if (left = right) return E[left];      /* caso base */
03   middle ← (right - left) / 2;           /* divisao */
04   Ma ← max (E, left, middle);         /* uso da hipotese de inducao */
05   Mb ← max (E, middle + 1, right);     /* uso da hipotese de inducao */
06   if (Ma > Mb) return Ma;           /* conquista */
07   else return Mb;
08 }

```

Figura 3.4: Algoritmo recursivo para [MAX]

## 3.2 Exemplos

Nesta seção, apresentam-se alguns outros problemas para os quais se podem criar algoritmos a partir das provas por indução dos teoremas correspondentes. Conforme se verá, as transformações são essencialmente as mesmas.

### 3.2.1 Classificação em um Campeonato

[CLASS] Ao final um campeonato sem empates em que todos os  $n$  participantes jogam entre si uma única vez, determinar uma classificação (ordem) dos adversários tal que, para todo  $i \in \{1, 2, \dots, n-1\}$ , o participante  $p_i$  vence  $p_{i+1}$ .

O teorema correspondente a esse problema já foi provado no capítulo 2. A transformação da prova apresentada em algoritmo é imediata, conforme mostra a figura 3.5. No código, a função *vence* ( $p_i, p_j$ ) tem valor verdadeiro se  $p_i$  tiver vencido a partida contra  $p_j$ ; caso contrário, o valor é falso.

```

01 procedure class (p, n) {
02   if (n = 2) {
03     if vence (p[2], p[1]) troca (p[1], p[2]);
04     return;
05   }
06   class (p, n - 1);
07   tmp ← p[n];
08   i ← n;
09   while ((vence (tmp, p[i - 1])) and (i > 1)) do {
10     p[i] ← p[i - 1];
11     i ← i - 1;
12   }
13   p[i] ← tmp;
14 }

```

Figura 3.5: Determinação da Classificação dos Participantes de um Campeonato

### 3.2.2 Diagonais de um Polígono Convexo

[DIAGS] Dado um polígono convexo de  $n$  vértices (sempre numerados no sentido horário), enumere todas as suas diagonais, identificadas pelos vértices em suas extremidades.

O teorema correspondente a esse problema é o seguinte:

**Teorema** *É possível enumerar todas as diagonais de um polígono com  $k$  vértices numerados no sentido horário, para todo  $k \geq 3$ .*

**Prova** Por indução simples. O caso base é o triângulo ( $n = 3$ ). Como triângulos não possuem diagonais, esse caso é trivial. Como hipótese de indução, suponha que seja possível enumerar as diagonais de um polígono convexo com  $n = k - 1$  vértices. Deseja-se provar que é possível enumerar as arestas de um polígono convexo  $P_k$  com  $k$  vértices. Sejam  $v_1, v_2, v_3, \dots, v_{k-1}, v_k$  os vértices desse polígono, de acordo com a numeração horária. Crie um novo polígono  $P_{k-1}$  composto pelos pontos  $v_1, v_2, v_3, \dots, v_{k-1}$  (em outras palavras  $P_{k-1}$  é criado a partir de  $P_k$  pela substituição das arestas  $v_{k-1}v_k$  e  $v_kv_1$  pela aresta  $v_{k-1}v_1$ ).  $P_{k-1}$  é um polígono convexo com  $k - 1$  vértices e todas as suas diagonais desse polígono são também diagonais de  $P_k$ . Pela hipótese de indução, é possível fazer a enumeração dessas diagonais. Para completar o conjunto, basta incluir explicitamente as diagonais de  $P_k$  que não fazem parte de  $P_{k-1}$ :  $v_{k-1}v_1$  (que é um lado de  $P_{k-1}$ , não uma diagonal) e as diagonais que têm  $v_k$  (que não pertence a  $P_{k-1}$ ) como extremidade ( $v_2v_k, v_3v_k, v_4v_k, \dots, v_{k-3}v_k, v_{k-2}v_k$ ).  $\square$

A figura 3.6 mostra o pseudocódigo do algoritmo correspondente a essa prova por indução. Observe que a retirada de vértices do polígono é feita implicitamente: basta utilizar como parâmetro da função o número de vértices do polígono que devem ser considerados. A função *imprima* é responsável pela impressão no formato apropriado das arestas determinadas pelos vértices cujos rótulos são passados como parâmetro.

```

01 procedure diags (P, n) {
02     if (n = 3) return; /*caso base*/
03     diags (P, n - 1); /*uso da hipotese de inducao*/
05     imprima (n - 1, 1); /*aresta v_{n-1}v_1*/
06     for i ← 2 to n - 2 do {
07         imprima (i, n); /*aresta v_iv_n*/
08     }
09 }
```

Figura 3.6: Algoritmo para [DIAGS]

### 3.2.3 Interseção de Diagonais de um Polígono Convexo

[INTER] Dado um polígono convexo com  $n$  vértices (numerados no sentido horário), enumere todas as interseções de pares de diagonais, identificadas pelos rótulos dos vértices que determinam as diagonais que se interceptam.

À primeira vista, pode parecer que esse é um problema difícil. Afinal, dadas duas diagonais, não há garantias de que se interceptem. É possível, por exemplo, que as duas diagonais partam de um mesmo vértice do polígono (nesse caso, considera-se que não há interseção; estamos interessadas apenas nas interações que ocorrem no interior do polígono). Mesmo que as diagonais sejam disjuntas nos vértices, é possível que não se interceptem. Assim, para enumerar os cruzamentos de diagonais, não é suficiente enumerar todos os pares de diagonais.

No entanto, uma observação mais cuidadosa do problema mostra que há um método relativamente simples de fazer enumeração. Em primeiro lugar, três vértices distintos jamais determinarão duas diagonais que se interceptam: qualquer par de diagonais determinado por eles terá um vértice em comum. Portanto, serão necessários pelo menos quatro vértices para determinar duas diagonais que se cruzam. De fato, é fácil perceber que, de todos os pares de diagonais determinados por quatro pontos, em apenas um deles as diagonais se interceptam. Assim, o problema de enumerar todos os pares de diagonais que se interceptam em um polígono é equivalente a enumerar todas as quádruplas de vértices do polígono.

**Teorema** *Dado um polígono convexo  $P_n$ , sendo  $n$  o número de lados (ou vértices), é possível enumerar todas as quádruplas de vértices.*

**Prova** Por indução simples em  $n$ . O caso base é  $n = 4$  (para que haja uma *quádrupla*, é necessário haver pelo menos quatro pontos). Esse caso é trivial: todos os quatro vértices ( $v_1, v_2, v_3$  e  $v_4$ ) de um quadrilátero convexo formam a única quádrupla. Como hipótese de indução, suponha que seja possível enumerar todas as quádruplas de um polígono convexo com  $k$  vértices. Seja  $P_{k+1} = (v_1, v_2, v_3, \dots, v_k, v_{k+1})$  um polígono convexo com  $k + 1$  vértices. Retire o vértice  $v_{k+1}$  desse polígono e construa, ligando  $v_1$  a  $v_k$ , o polígono  $P_k = (v_1, v_2, \dots, v_k)$ . Como  $P_k$  tem  $k$  vértices, a hipótese de indução garante que é possível enumerar todas as quádruplas de vértices desse polígono. Claramente, elas são também quádruplas de  $P_{k+1}$ . Para completar a lista de todas as quádruplas desse polígono, resta considerar as que contêm o vértice  $v_{k+1}$ , o único que não pertence a  $P_k$ . Isso é simples: basta adicionar  $v_{k+1}$  a todas as triplas de vértices de  $P_k$ .  $\square$

O pseudocódigo da figura 3.7 representa o algoritmo recursivo que pode ser derivado da prova por indução.

### 3.2.4 Bolas Distintas em Urnas Distintas

[UdBd] Dadas  $n$  bolas distintas e  $m$  urnas distintas, enumere todas as configurações que as bolas nas urnas podem formar.

**Teorema** *É possível enumerar todas as configurações possíveis para a distribuição de  $n$  bolas distintas entre  $m$  urnas distintas.*

**Prova** Por indução simples em  $n$ . O caso base é  $n = 0$ . Existe uma única maneira de distribuir zero bola em  $m$  urnas: deixar todas vazias. Considere, como hipótese de indução, que seja possível obter a enumeração de todas as configurações possíveis para a distribuição de  $n = k - 1$  bolas distintas ( $k \geq 1$ ) em  $m$  urnas distintas. Deseja-se provar que o mesmo é válido para  $n = k$  bolas e  $m$  urnas. Seja  $b$  uma bola qualquer do conjunto de  $n$  bolas (a primeira delas, por exemplo). Para cada urna  $u_i$  ( $1 \leq i \leq m$ ), haverá pelo menos uma configuração em que  $b$  é colocada em

```

01 procedure inter (P, n) {
02   if (n = 4) { /*caso base*/
03     imprima (1, 2, 3, 4);
04     return;
05   };
06   inter (P, n - 1); /*uso da hipotese de inducao*/
07   for i ← 1 to n - 3 do {
08     for j ← (i + 1) to n - 2 do {
09       for k ← (j + 1) to n - 1 do {
10         imprima (i, j, k, n); /*tripla (v_i, v_j, v_k) + vertice v_n*/
11       }
12     }
13   }
14 }

```

Figura 3.7: Algoritmo para [DIAGS]

$u_i$ . Portanto, para garantir que todas as configurações sejam listadas, é necessário listar todas as configurações em que  $b$  está em  $u_1$ , todas em que  $b$  está em  $u_2$  e assim por diante. Considere o caso genérico  $u_i$ : desejamos listar todas as configurações em que  $b$  está em  $u_i$ . Isso pode ser feito pela determinação de todas as configurações possíveis para as  $k - 1$  bolas restantes (nas  $m$  urnas) e pela adição, em cada uma dessas configurações, da bola  $b$  à urna  $u_i$ . A enumeração de configurações com  $k - 1$  bolas em  $m$  urnas é possível graças à hipótese de indução. Como é necessário analisar todas as  $m$  posições possíveis para  $b$ , é necessário aplicar a hipótese de indução  $m$  vezes.  $\square$

Dessa prova deriva imediatamente um algoritmo recursivo para enumerar todas as configurações possíveis para  $n$  bolas distintas em  $m$  urnas distintas, mostrado na figura 3.8. Na primeira chamada a essa rotina, a configuração  $C$  (o segundo parâmetro de entrada) deve corresponder a  $m$  urnas vazias.

```

01 procedure udbd (b, C, n, m) {
02   if (b > n) {
03     imprima a configuração C;
04     return;
05   }
06   for i ← 1 to m do {
07     insira a bola b na urna i da configuração C;
08     udbd (b + 1, C, n, m);
09     retire a bola b da urna i da configuração C;
10   }
11 }

```

Figura 3.8: Enumeração das configurações para  $n$  bolas distintas em  $m$  urnas distintas

### 3.2.5 Ordenação

[ORD] Dada uma seqüência  $\sigma$  de  $n$  números reais, encontrar uma permutação  $\Pi$  de  $\sigma$  em que os elementos estejam em ordem não-decrescente, ou seja,  $\Pi_1 \leq \Pi_2 \leq \dots \leq \Pi_n$ .

O problema de ordenação é um dos que aparecem com mais freqüência na literatura. Isso é evidenciado pela grande quantidade de algoritmo existentes. Nesta seção, serão apresentados três diferentes algoritmos, cada um deles baseado em uma prova indutiva para o mesmo teorema:

**Teorema  $T(n)$ :** *É possível resolver [ORD] para  $n = k$ , sendo  $k$  um inteiro positivo.*

Uma primeira versão para a prova desse teorema é apresentada a seguir:

**Prova (primeira versão)** Por indução simples. O caso base é trivial: para  $n = 1$ , não há o que fazer; uma seqüência de um elemento já está ordenada. Suponha, por hipótese de indução, que seja possível ordenar uma seqüência de  $n = k - 1$  números. Deseja-se provar que é possível ordenar uma seqüência de  $k$  números. Seja  $S_k$  a seqüência a ser ordenada e  $x$  o valor do último elemento. Se removermos  $x$  de  $S_k$ , teremos uma nova seqüência,  $S_{k-1}$ , de tamanho  $k - 1$ . Pela hipótese de indução, é possível ordenar  $S_{k-1}$ . Resta agora apenas inserir  $x$  na posição correta. Mas isso é muito simples: basta percorrer a seqüência  $S_{k-1}$  (já ordenada) do maior para o menor elemento e inserir  $x$  imediatamente após o primeiro elemento visitado cujo valor seja menor ou igual a  $x$ . Se todos os elementos forem maiores que  $x$ , a inserção deverá ser feita antes da primeira posição de  $S_{k-1}$ . Em qualquer dos casos, a seqüência resultante terá os  $n$  da seqüência original em ordem.  $\square$

Nessa prova, a operação básica realizada no passo indutivo é a *inserção* de um elemento em sua posição correta. Em outras palavras, demonstrou-se que o problema de ordenação pode ser resolvido a partir do problema de *inserção*. Em vista disso, o algoritmo derivado a partir dessa prova é conhecido como *insertion sort*, apresntado (em sua versão recursiva) na figura 3.9. Note que, nesse algoritmo (e nos dois que se seguem), considera-se que a seqüência de entrada esteja representada sobre um vetor  $v$  de  $n$  posições; a notação  $v[i]$  representa o elemento na  $i$ -ésima posição do vetor, sendo  $1 \leq i \leq k$ .

```

01 procedure insertionSort (v, n) {
02   if (n = 1) return;
03   insertionSort (v, n - 1);
04   tmp ← v[n];
05   i ← n;
06   while ((v[i - 1] > tmp) and (i > 1)) do {
07     v[i] ← v[i - 1];
08     i ← i - 1;
09   }
10   v[i] ← tmp;
11 }

```

Figura 3.9: Insertion Sort

**Prova (segunda versão)** Por indução simples. O caso base ( $n = 1$ ) é trivial e idêntico ao da primeira versão. Como hipótese de indução, suponha que seja possível ordenar uma seqüência de

$n = k - 1$  números,  $k > 1$ . Deseja-se provar que ordenar uma seqüência  $S_k$  de  $n = k$  números é possível. Para isso, dada a seqüência  $S_k$ , encontre inicialmente o elemento  $M$  de maior valor (isso pode ser feito utilizando o algoritmo descrito na seção 3.1: trata-se de uma versão do problema [MAX]). Seja  $i$  a posição na seqüência em que  $M$  se encontra ( $1 \leq i \leq k$ ). Se  $i \neq k$ , troque  $M$  com o elemento da  $k$ -ésima posição do vetor. Feito isso, garante-se que  $M$  estará em sua posição definitiva. Para garantir a ordenação de toda a seqüência, resta apenas ordenar seus  $k - 1$  primeiros elementos. Isso pode ser feito aplicando-se a hipótese de indução.  $\square$

Repare que a única diferença entre essa prova e a anterior está no passo indutivo. Na primeira versão, retira-se da seqüência de tamanho  $k$  um elemento cuja posição final é desconhecida. Após a aplicação da hipótese de indução, insere-se o elemento no trecho ordenado da seqüência. Na segunda versão, o elemento descartado é cuidadosamente escolhido: trata-se do maior elemento da seqüência, destinado a estar na última posição do vetor. Após a aplicação da hipótese de indução, nada mais há a fazer. Portanto, essa versão da prova mostra que o problema da ordenação é pode ser resolvido se o problema da *seleção do máximo* (já estudado) também puder sê-lo. O algoritmo resultante da segunda versão da prova recebe o nome de *selection sort*. A versão recursiva do algoritmo é apresentada na figura 3.10.

```

01 procedure selectionSort ( $v, n$ ) {
02   if ( $n = 1$ ) return;
03    $max \leftarrow 1$ ;
04   for  $i \leftarrow 2$  to  $n$  do {
05     if ( $v[i] > v[max]$ )  $max \leftarrow i$ ;
06   }
07    $tmp \leftarrow v[max]$ ;
08    $v[max] \leftarrow v[n]$ ;
09    $v[n] \leftarrow tmp$ ;
10   selectionSort ( $v, n - 1$ );
11 }
```

Figura 3.10: Selection Sort

**Prova (terceira versão)** Por indução forte. Como nos casos anteriores, o caso base é trivial: uma seqüência com um único elemento ( $n = 1$ ) já está ordenada. Considere, como hipótese de indução, que seja possível ordenar uma seqüência com  $n$  elementos, sendo  $n < k$  e  $k > 1$ . Deseja-se provar que é possível ordenar uma seqüência de  $k$  elementos. Considere a seguinte estratégia: particione a seqüência em duas subseqüências, uma formada pelos  $\lceil k/2 \rceil$  primeiros elementos e a outra pelos  $\lfloor k/2 \rfloor$  últimos. Como as duas subseqüências têm tamanho menor que  $k$ , a hipótese de indução garante que é possível ordenar cada uma delas separadamente. Uma vez feito isso, a seqüência original não estará ordenada, mas estará dividida em duas subseqüências ordenadas. Para ordená-la completamente, basta intercalar as duas subseqüências.  $\square$

Esse prova corresponde a um algoritmo um tanto mais complexo que os anteriormente apresentados. Em lugar de resolver um subproblema de menor tamanho, são resolvidos dois. O passo indutivo envolve uma operação que parece mais complexa que a simples seleção ou inserção de

um elemento: a intercalação de duas subsequências. Entretanto, esse método de ordenação (que recebe o nome de *mergesort*) é em geral mais eficiente que *insertion sort* e *selection sort*.

```
01 procedure mergesort (v, left, right) {
02   if (left = right) return;
03   middle [(right + left)/2];
04   mergesort (v, left, middle);
05   mergesort (v, middle + 1, right);
06   intercala (v, left, middle, right);
07 }
```

Figura 3.11: Mergesort

## Exercícios

1. Na prova por indução apresentada na seção 3.2.3, não foi provado que é possível enumerar todas as triplas de vértices de um polígono convexo. Prove que isso é de fato possível e construa o algoritmo recursivo derivado dessa prova.
2. Um dos métodos de ordenação mais utilizados, *quicksort*, não foi apresentado no texto. Descubra como funciona esse algoritmo e elabore uma prova indutiva do teorema apresentado na seção 3.2.5 que leva naturalmente ao *quicksort*.
3. Apresente as versões iterativas dos algoritmos de ordenação por inserção (*insertion sort*) e seleção (*selection sort*).
4. Construa um algoritmo para intercalar dois vetores ordenados de tamanho  $n_1$  e  $n_2$ . (Dica: utilize um vetor auxiliar de tamanho  $n_1 + n_2$ .)
5. Apresente uma versão iterativa do algoritmo *mergesort*. Certifique-se de que ela ordena corretamente vetores de qualquer tamanho.