

Capítulo 3

Projeto de Algoritmos e Indução Matemática

Este capítulo apresenta a relação entre a prova de teoremas por indução matemática e o projeto de algoritmos. De uma outra perspectiva, esta relação permite também demonstrar a corretude de um algoritmo via a prova de um teorema por indução matemática. Assim, o método aqui apresentado não só tem a importante função de auxiliar efetivamente a concepção de um algoritmo para a resolução de um problema específico como também a de permitir a compreensão das razões pelas quais um algoritmo realmente encontra a solução desejada para o problema em questão.

3.1 Problemas, Teoremas, Provas Indutivas e Algoritmos

3.1.1 Um Exemplo Simples

Considere o problema abaixo:

Problema [MAX] Dado um conjunto $E = \{e_1, e_2, \dots, e_n\}$ onde $e_j \in Z$, $j = 1, \dots, n$, deseja-se encontrar o elemento de E de maior valor. \square

O teorema mais natural (existem outros, conforme veremos mais adiante no curso) 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 específicas 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 $|E| = n$, para todo n , ou seja $n = 1, 2, 3, \dots$

Vamos agora provar este teorema por indução matemática simples:

Base Para $n = 1$, $T(1)$ se resume a saber encontrar o maior elemento de um conjunto unitário $E = \{e_1\}$. Como e_1 é o único elemento, e_1 é também o maior elemento de E .

Passo Indutivo Precisamos provar que se sabemos encontrar o maior elemento de um conjunto com n elementos ($T(n)$, nossa hipótese), então sabemos encontrar o maior elemento de um conjunto

de $n+1$ elementos ($T(n+1)$). Portanto, para $E_n = \{e_1, e_2, \dots, e_n\}$ sabemos, por hipótese, encontrar i_M , tal que e_{i_M} é o maior elemento de E_n . Precisamos, então, demonstrar que a partir deste conhecimento é possível encontrar o maior elemento de E_{n+1} . Para tal, basta neste caso argumentar que tudo que é necessário fazer para se determinar o maior elemento de $E_{n+1} = E_n \cup \{e_{n+1}\}$ é comparar o maior elemento de E_n , e_{i_M} , com o novo elemento incorporado ao conjunto e_{n+1} . Se e_{i_M} for menor que e_{n+1} este é o maior elemento de E_{n+1} , caso contrário e_{i_M} o será.

Agora observe que a prova acima acaba de especificar um algoritmo para a resolução de (MAX) para um conjunto qualquer E de qualquer tamanho n . Este é obtido através da execução dos passos indutivos a partir do caso base até o tamanho de E , i.e. n . Em outras palavras, iremos resolver $T(1)$ de acordo com a prova do caso base e, em seguida, ir aplicando o passo indutivo para obter $T(2)$ a partir de $T(1)$, $T(3)$ a partir de $T(2)$, \dots , até se obter $T(n)$. Este algoritmo pode ser escrito tanto de forma iterativa,

```
#define n 100
int E[n] = { e_1, ..., e_n }
MAX_ITER(n)
{
    int i, Tmax_elem[n], Tmax_index[n];
    // Tmax_elem[i] contém o valor do maior elemento de
    // E[i] = { e_1, ..., e_i }Tmax_index[i] contém o índice
    // deste elemento.

    // Caso Base
    Tmax_elem[1] = E[1]; // Tmax_elem[i] ou Tmax_index[i] representam a
    Tmax_index[1] = 1; // solução de (MAX) para n=i
    // Passos indutivos
    for(i=1; i<=n-1; i++) {
        // Passo indutivo T(i) => T(i+1)
        if( Tmax_elem[i] < E[i+1]) {
            Tmax_elem[i+1] = E[i+1];
            Tmax_index[i+1] = i+1;
        }
        else {
            Tmax_elem[i+1] = Tmax_elem[i];
            Tmax_index[i+1] = Tmax_index[i];
        }
    } // end for
    // Solução do Problema
    print( Valor do Maior Elemento: Tmax_elem[n], Indice: Tmax_index[n] );
}
```

como recursiva,

```
#define n 100
struct tmax {
    int elem;
    int index;
}
struct tmax MAX_REC( int ); // prototyte: para retornar dois valores
int E[n] = { e_1, ..., e_n };
```

```

main()
{
    struct tmax T;
    T = MAX_REC ( n );
    // Solução do Problema
    print( Valor do Maior Elemento: T.elem, Indice: T.index );
}

struct tmax MAX_REC(int i)
{
    struct tmax Ti, T;
    if ( i == 1 ){
        // Caso Base
        T.elem = E[1]; // T.elem ou T.index representam a solução de (MAX)
        T.index = 1; // para n=i
        return( T );
    }
    else {
        // Passo indutivo T(i) => T(i+1)
        Ti = MAX_REC (i-1);
        if( Ti.elem < E[i] ) {
            T.elem = E[i];
            T.index = i;
        }
        else {
            T.elem = Ti.elem;
            T.index = Ti.index;
        }
        return(T);
    }
}

```

neste texto, iremos priorizar a forma recursiva. Uma razão para esta escolha é a de que a forma recursiva permite evidenciar mais claramente o *teorema* que está sendo provado. Também, esta forma, permite uma representação mais clara no caso em que a prova por indução forte é usada. Vejamos então a prova por indução forte do teorema de que sabemos resolver (MAX) para todo n . O caso base é o mesmo.

Passo Indutivo Precisamos provar que se sabemos encontrar o maior elemento de um conjunto com menos de n elementos, ($T(n_0)$, $n_0 < n$, nossa hipótese), então sabemos encontrar o maior elemento de um conjunto de n elementos ($T(n)$). Portanto, para $E_{n_1} = \{e_1, e_2, \dots, e_{n_1}\}$ e $E_{n_2} = \{e_1, e_2, \dots, e_{n_2}\}$, $n_1 < n$, $n_2 < n$ e $n_1 + n_2 = n$, sabemos, por hipótese, encontrar i_{M_1} e i_{M_2} respectivamente o maior elemento de E_{n_1} e E_{n_2} .

Precisamos, então, demonstrar que a partir deste conhecimento é possível encontrar o maior elemento de E_n . Como o maior dos maiores elementos de dois conjuntos é o maior elemento da união dos conjuntos, para se determinar o maior elemento de $E_n = E_{n_1} \cup E_{n_2}$ compara-se o maior elemento de E_{n_1} com o de E_{n_2} , o maior elemento de E_n será o maior entre $e_{i_{M_1}}$ e $e_{i_{M_2}}$.

O algoritmo resultante desta prova particiona sucessivamente o conjunto corrente E_n (aquele para o qual queremos resolver o problema (MAX)) em dois subconjuntos E_{n_1} e E_{n_2} , tais que

$E_n = E_{n1} \cup E_{n2}$, $E_{n1} \cap E_{n2} = \emptyset$, $E_{n1} \neq \emptyset$ e $E_{n2} \neq \emptyset$ (caso contrário, não seria uma partição), garantindo que os subconjuntos tenham menos de n elementos. Este procedimento é repetido até que um (ou os dois) subconjunto tenha apenas 1 elemento, i.e., quando se chega ao caso base.

```
#define n 100
struct tmax {
    int elem;
    int index;
}

struct tmax MAX_IND_FORTE( int , int );
int E[n] = { e_1, ..., e_n };

main ()
{
    struct tmax T;
    T = MAX_REC ( 1, n );
    // Solução do Problema
    print( Valor do Maior Elemento: T.elem, Indice: T.index );
}

struct tmax MAX_IND_FORTE(int inicio, int fim)
{
    struct tmax T1, T2, T;
    int meio;
    if ( inicio == fim ){
        // Caso Base
        T.elem = E[inicio]; // T.elem ou T.index representam a solução de
        T.index = inicio; // (MAX) para n=i
        return( T );
    }
    else {
        // Passo indutivo T(n1), T(n2) => T(n)
        meio = floor((inicio + fim)/2.);
        T1 = MAX_IND_FORTE (inicio, meio);
        T2 = MAX_IND_FORTE (meio + 1, fim);
        if( T1.elem > T2.elem ) {
            T.elem = T1.elem;
            T.index = T1.index;
        }
        else {
            T.elem = T2.elem;
            T.index = T2.index;
        }
        return(T);
    }
}
```

O método a ser seguido para o projeto de um algoritmo para um problema dado consiste em seguir esta sequência de passos. O exemplos que seguem aplicam este método para diferentes problemas.

3.1.2 Classificação em um Campeonato

Problema [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} . \square

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 3.1. 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 3.1: Determinação da Classificação dos Participantes de um Campeonato

3.1.3 Diagonais de um Polígono Convexo

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

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

3.1.4 Interseção de Diagonais de um Polígono Convexo

Problema [INTER] Dado um polígono convexo com n vértices (numerados no sentido horário), enumere todas as interseções de pares de diagonais, identificadas pelas coordenadas dos vértices que determinam as diagonais que se interceptam. \square

3.1.5 Bolas Distintas em Urnas Distintas

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

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 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 3.2. Na primeira chamada a essa rotina, a configuração C (o segundo parâmetro de entrada) deve corresponder a m urnas vazias.

3.1.6 Ordenação

A criação de um algoritmo para um determinado problema P é feita em duas etapas, de acordo com a técnica vista até aqui: num primeiro momento, elabora-se um teorema T_P a partir de P ; em seguida, a partir de uma prova indutiva de T_P cria-se um algoritmo (normalmente recursivo). Assim sendo, o algoritmo resultante desse processo depende não só do teorema enunciado, mas também da prova fornecida.

O propósito desta seção é justamente ilustrar como diferentes provas para o mesmo teorema podem dar origem a algoritmos completamente distintos. Considere, por exemplo, o problema da *ordenação*:

```

01 procedure udbd ( $b, C, n, m$ ) {
02   if ( $b > n$ ) {
03     imprima a configuração  $C$ ;
04     return;
05   }
06   for  $i \leftarrow 1$  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 3.2: Enumeração das configurações para n bolas distintas em m urnas distintas

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

O teorema que naturalmente deriva desse enunciado é o seguinte:

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

A seguir, apresentam-se três possíveis provas para esse teorema e os três algoritmos que delas derivam. Para a apresentação dos algoritmos, suponha que a seqüência de entrada seja representada como um vetor v de n posições; a notação $v[i]$ representa o elemento na i -ésima posição do vetor, sendo $1 \leq i \leq k$.

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 vista disso, o algoritmo derivado a partir dessa prova é conhecido como *Insertion Sort*. Sua versão recursiva é apresentada na figura 3.3.

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. Isso pode ser feito da seguinte forma: dada a seqüência S_k , encontre o elemento M de maior valor (isso pode ser feito utilizando o algoritmo descrito na seção ??). 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

```

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

```

Figura 3.3: Insertion Sort

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. Como a atividade básica executada no passo indutivo é a *seleção* do maior elemento, o algoritmo resultante da segunda versão da prova recebe o nome de *Selection Sort*. A versão recursiva do algoritmo é apresentada na figura 3.4.

```

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 3.4: 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 subsequências têm tamanho menor que k , a hipótese de indução garante que é possível ordenar cada uma delas separadamente. Uma vez feito isso, a sequência original não estará ordenada, mas estará dividida em duas subsequências ordenadas. Para ordená-la completamente, basta intercalar as duas subsequências. \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, conforme se verá no capítulo ??, 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  $\lfloor (right + left)/2 \rfloor$ ;
04   mergeSort (v, left, middle);
05   mergeSort (v, middle + 1, right);
06   intercala (v, left, middle, right);
07 }
```

Figura 3.5: Mergesort

Exercícios

1. Um dos métodos de ordenação mais utilizados, o *quicksort*, não foi apresentado no texto. Descubra como funciona esse algoritmo e elabore uma prova indutiva do teorema apresentado na seção 3.1.6 que leva naturalmente ao *quicksort*.
2. Apresente as versões iterativas dos algoritmos de ordenação por inserção (*insertion sort*) e seleção (*selection sort*).
3. Apresente uma versão iterativa do algoritmo *mergesort*. Certifique-se de que ela funciona para vetores de qualquer tamanho.