Capítulo 2

Recursão

> "Ao tentar resolver o problema, encontrei obstáculos dentro de obstáculos. Por isso, adotei uma solução recursiva."
>
> — um aluno

> "To understand recursion, we must first understand recursion."
>
> — folclore

O conceito de recursão é de fundamental importância em computação. Este capítulo introduz o conceito por meio de um exemplo muito simples.

2.1 Algoritmos recursivos

Muitos problemas computacionais têm a seguinte propriedade: cada instância[1] do problema contém uma instância menor do mesmo problema. Dizemos que esses problemas têm *estrutura recursiva*. Para resolver um tal problema é natural aplicar o seguinte método:

se a instância em questão é pequena,
    resolva-a diretamente (use força bruta se necessário);
senão,
    reduza-a a uma instância menor do mesmo problema,
    aplique o método à instância menor
    e volte à instância original.

A aplicação deste método produz um *algoritmo recursivo*.

---

[1] Uma **instância** de um problema é um exemplo do problema. Cada conjunto de dados de um problema define uma instância. (A palavra *instância* é um neologismo importado do inglês. Ela está sendo empregada aqui no sentido de *exemplo*, *espécime*, *amostra*.)

2.2 Um exemplo: o problema do máximo

Considere o problema de determinar *o valor de um*[2] *elemento máximo de um vetor* $v[0\,..\,n{-}1]$. O tamanho de uma instância do problema é n. É claro que o problema só faz sentido se o vetor não for vazio, ou seja, se $n \geq 1$. Se $n = 1$,[3] então $v[0]$ é o único elemento do nosso vetor e portanto $v[0]$ é o máximo. Se $n > 1$, o valor que procuramos é o maior dentre o máximo do vetor $v[0\,..\,n{-}2]$ e o número $v[n{-}1]$. Assim, a instância $v[0\,..\,n{-}1]$ do problema fica reduzida à instância $v[0\,..\,n{-}2]$. Estas observações levam à seguinte função recursiva:

```
/* Ao receber v e n >= 1, esta função devolve o valor de
 * um elemento máximo do vetor v[0..n-1]. */
int MáximoR (int v[], int n) { 4
   if (n == 1)
      return v[0];
   else {
      int x;
      x = MáximoR (v, n - 1);
      if (x > v[n-1])
         return x;
      else
         return v[n-1];
   }
}
```

Para verificar que uma função recursiva está correta, use o seguinte roteiro. Passo 1: Escreva *o que* a função deve fazer (veja Capítulo 1). Passo 2: Verifique se a função de fato faz o que deveria quando n é pequeno ($n = 1$, no nosso exemplo). Passo 3: Imagine que n é grande ($n > 1$, no nosso exemplo) e suponha que a função fará a coisa certa se no lugar de n tivermos algo menor que n. Sob esta hipótese, verifique que a função faz o que dela se espera.

Como o computador executa uma função recursiva? Embora relevante, esta pergunta será ignorada por enquanto. Veja o conceito de pilha de execução na Seção 6.5.

---

[2] Eu não disse "*do* elemento máximo" porque o vetor pode ter vários elementos máximos.

[3] Embora sejam tipograficamente semelhantes, os sinais $=$ e `=` têm significados diferentes. O primeiro é o sinal de igualdade da matemática: "$x = y$" significa "x é igual a y". O segundo é o operador de atribuição na linguagem C: "`x = y`" significa "atribua à variável `x` o valor da variável `y`". O "$=$" da matemática corresponde ao "`==`" da linguagem C.

[4] Veja Seção A.4.

Exercícios

2.2.1 Escreva uma versão iterativa da função `MáximoR`.

2.2.2 Critique a função abaixo. Ela promete encontrar o valor de um elemento máximo de `v[0..n-1]`.

```
int máximoR1 (int v[], int n) {
   int x;
   if (n == 1) return v[0];
   if (n == 2) {
      if (v[0] < v[1]) return v[1];
      else return v[0]; }
   x = máximoR1 (v, n - 1);
   if (x < v[n-1]) return v[n-1];
   else return x; }
```

2.2.3 Critique a seguinte função recursiva que promete encontrar o valor de um elemento máximo do vetor `v[0..n-1]`.

```
int máximoR2 (int v[], int n) {
   if (n == 1) return v[0];
   if (máximoR2 (v, n - 1) < v[n-1])
      return v[n-1];
   else
      return máximoR2 (v, n - 1); }
```

2.2.4 Se `X` é a função recursiva abaixo, qual o valor de `X(4)`?

```
int X (int n) {
   if (n == 1 || n == 2) return n;
   else return X (n - 1) + n * X (n - 2); }
```

2.2.5 O que há de errado com a seguinte função recursiva?

```
int XX (int n) {
   if (n == 0) return 0;
   else return XX (n/3 + 1) + n; }
```

2.2.6 Programa de teste. Escreva um pequeno programa para testar a função recursiva `MáximoR`. O seu programa deve pedir ao usuário que digite uma sequência de números ou gerar um vetor aleatório (veja Apêndice I).

Importante: Para efeito de testes, acrescente ao seu programa uma função auxiliar que *confira* a resposta produzida por `MáximoR`.

2.3 Outra solução recursiva do problema

A função `MáximoR` discutida acima aplica a recursão ao subvetor $v[0\,..\,n{-}2]$. É possível escrever uma versão que aplique a recursão ao subvetor $v[1\,..\,n{-}1]$:

```
/* Ao receber v e n >= 1, esta função devolve o valor de
 * um elemento máximo do vetor v[0..n-1]. */
int Máximo (int v[], int n) {
   return MaxR (v, 0, n);
}

/* Esta função recebe v, i e n tais que i < n e devolve
 * o valor de um elemento máximo do vetor v[i..n-1]. */
int MaxR (int v[], int i, int n) {
   if (i == n-1) return v[i];
   else {
      int x;
      x = MaxR (v, i + 1, n);
      if (x > v[i]) return x;
      else return v[i];
   }
}
```

A função `Máximo` é apenas uma "embalagem"; o serviço pesado é executado pela função recursiva `MaxR`, que resolve um problema mais geral, com mais parâmetros que o original.

A necessidade de generalizar o problema ocorre com frequência na construção de algoritmos recursivos. O papel dos novos parâmetros (como `i` no exemplo acima) deve ser devidamente explicado na documentação da função,[5] o que nem sempre é fácil (veja mais exemplos nas Seções 7.7 e 12.3).

Exercícios

2.3.1 Verifique que a seguinte função é equivalente à função `Máximo`. Ela usa a aritmética de endereços mencionada no Seção D.4.

```
int máximo2r (int v[], int n) {
   int x;
   if (n == 1) return v[0];
   x = máximo2r (v + 1, n - 1);
   if (x > v[0]) return x;
   return v[0]; }
```

2.3.2 Max-Min. Escreva uma função recursiva que calcule a diferença entre o valor de um elemento máximo e o valor de um elemento mínimo do vetor `v[0..n-1]`.

[5] Explicações do tipo "a primeira chamada da função deve ser feita com `i = 0`" não explicam nada e devem ser evitadas a todo o custo.

2.3.3 SOMA. Escreva uma função recursiva que calcule a soma dos elementos positivos do vetor de inteiros $v[0\,..\,n{-}1]$. O problema faz sentido quando $n = 0$? Quanto deve valer a soma neste caso?

2.3.4 SOMA DE DÍGITOS. Escreva uma função recursiva que calcule a soma dos dígitos decimais de um inteiro positivo. A soma dos dígitos de `132`, por exemplo, é 6.

2.3.5 PISO DE LOGARITMO. Escreva uma função recursiva que calcule $\lfloor \log_2 n \rfloor$, ou seja, o piso do logaritmo de n na base 2. (Veja Exercício 1.2.4.)

2.3.6 FIBONACCI. A sequência de Fibonacci é definida assim: $F_0 = 0$, $F_1 = 1$ e $F_n = F_{n-1} + F_{n-2}$ para $n > 1$. Escreva uma função recursiva que receba n e devolva F_n. Escreva uma versão iterativa da função. Sua função recursiva é tão eficiente quanto a iterativa? Por quê?

2.3.7 Seja `F` a versão recursiva da função de Fibonacci (veja Exercício 2.3.6). O cálculo de `F(3)` provoca a sequência de invocações da função dada abaixo (note a indentação). Dê a sequência de invocações da função provocada pelo cálculo de `F(5)`.

```
F(3)
  F(2)
    F(1)
    F(0)
  F(1)
```

2.3.8 Execute a função `ff` abaixo com argumentos `7` e `0`.

```
int ff (int n, int ind) {
   int i;
   for (i = 0; i < ind; i++)
      printf ("  ");
   printf ("ff (%d,%d)\n", n, ind);
   if (n = 1)
      return 1;
   if (n % 2 == 0)
      return ff (n/2, ind + 1);
   return ff ((n-1)/2, ind + 1) + ff ((n+1)/2, ind + 1); }
```

2.3.9 EUCLIDES. A seguinte função calcula o maior divisor comum dos inteiros positivos `m` e `n`. Escreva uma função recursiva equivalente.

```
int Euclides (int m, int n) {
   int r;
   do {
      r = m % n;
      m = n; n = r;
   } while (r != 0);
   return m; }
```

2.3.10 EXPONENCIAÇÃO. Escreva uma função recursiva eficiente que receba inteiros positivos k e n e calcule o valor de k^n. Suponha que k^n cabe em um `int` (veja Seção C.2). Quantas multiplicações sua função executa aproximadamente?

2.3.11 Leia o verbete *Recursion* na Wikipedia [21].