Matemática Discreta: Aula 6

Operações Recursivas

Podemos definir recursivamente certas operações sobre objetos, como nos exemplos abaixo.

Exemplo 1: Uma definição recursiva para a operação de exponenciação a^n, definida para um número real a diferente de zero e n inteiro não-negativo é:

1. $a^0 = 1$

2. $a^n = (a^{n-1})a$ para $n \geq 1$

Exemplo 2: Uma definição recursiva para a multiplicação de dois inteiros positivos m e n é:

1. $m(1) = m$

2. $m(n) = m(n-1) + m$ para $n \geq 2$

Algoritmos Recursivos

A sequência S é definida recursivamente por:

1. $S(1) = 2$
2. $S(n) = 2S(n-1)$ para $n \geq 2$

Pela afirmação 1, $S(1)$, o primeiro objeto em S, é 2. Então pela sentença 2, o segundo objeto em S é $S(2) = 2S(1) = 2(2) = 4$. Novamente por 2, $S(3) = 2S(2) = 2(4) = 8$. Dessa forma podemos deduzir que S é a sequência $2, 4, 8, 16, 32, \ldots$

Suponha que desejamos escrever um programa de computador para obter $S(n)$ para algum inteiro positivo n. Podemos utilizar duas abordagens diferentes. Se desejamos encontrar $S(12)$, por exemplo, podemos começar com $S(1) = 2$ e então computar $S(2)$, $S(3)$, e assim por diante. Esta abordagem, sem dúvida, envolve iterações através de uma espécie de laço. Uma versão desse **algoritmo iterativo** é mostrada abaixo, implementada como uma função em Pascal.

```
function S(n: integer): integer;
{cálculo iterativo do valor S(n) }

var
    i: integer;                    {índice do loop}
    ValorCorrente: integer;        {valor corrente da função}

begin
    if n = 1 then
      S := 2
    else
      begin
        i:= 2;
        ValorCorrente := 2;
        while i <= n do
          begin
            ValorCorrente : = 2 * ValorCorrente:
            i:= i + 1;
          end;

       {o ValorCorrente tem agora o valor S(n)}
       S := ValorCorrente;
     end;
end;
```

A base, quando $n = 1$, é obtida na cláusula **then** da instrução **if**. A cláusula **else**, para $n > 1$, realiza uma inicialização e, a seguir, entra no laço **while**, que calcula os demais valores da sequência, até que o limite superior da mesma é alcançado. Você pode acompanhar a execução passo a passo deste algoritmo para valores pequenos de n, de forma a se convencer de que ele funciona.
A segunda abordagem utilizada para calcular $S(n)$ usa a definição de recursão de S diretamente. Uma versão do **algoritmo recursivo**, implementado novamente em Pascal, é apresentada a seguir.

```
function S(n : integer):integer;
{Calcula recursivamente os valores de S(n)}

begin
      if n = 1 then
         S:= 2
      else
         S:= 2*S(n-1);
end;
```

O corpo desta função consiste em uma única estrutura **if-then-else**. Para entendermos o seu funcionamento, vamos acompanhar a execução da mesma para calcular $S(3)$. A função é inicialmente chamada para o valor de entrada $n = 3$. Como n é diferente de 1, a execução é direcionada para a cláusula **else**. Neste ponto o procedimento para calcular $S(3)$ é momentaneamente suspenso, até que o valor de $S(2)$ seja conhecido. Qualquer informação relevante para a obtenção de $S(3)$ é armazenada na memória do computador numa *pilha*, para ser posteriormente carregada quando o cálculo puder ser completado. (Uma pilha é uma coleção de dados onde qualquer novo item ao entrar é armazenado no topo, e somente o item do topo da pilha pode ser removido ou acessado. A função é novamente chamada para um valor de entrada $n = 2$. Novamente a cláusula **else** é executada e o cálculo de $S(2)$ é momentaneamente suspenso com as informações relevantes armazenadas na pilha, enquanto a função é chamada novamente, agora com $n = 1$ como entrada.

Desta vez o valor de retorno, 2, pode ser calculado diretamente pela cláusula **then** da função. Esta última chamada à função retorna este valor à penúltima chamada, que pode recuperar qualquer informação relevante para o caso $n = 2$ da pilha, calcular $S(2)$ e disponibilizar o resultado à chamada anterior (a primeira chamada). Finalmente, esta chamada inicial de S é capaz de esvaziar a pilha e concluir seus cálculos, retornando o valor de $S(3)$.
Quais são as vantagens e desvantagens de algoritmos iterativo e recursivo para executar uma determinada tarefa? Neste exemplo, a versão recursiva é menor, pois dispensa o gerenciamento do laço. Descrever a execução de uma versão recursiva é mais complexo do que descrever a versão iterativa, porém todos os passos são realizados automaticamente. Não precisamos saber o que está acontecendo internamente, é necessário apenas termos conhecimento de que uma quantidade grande de chamadas pode demandar muita memória para o armazenamento na pilha das informações necessárias às chamadas anteriores. Se for necessária mais memória do que o que se tem disponível, ocorre um ”estouro de pilha”.

Além de usar muita memória, algoritmos recursivos podem demandar muitos outros procedimentos computacionais e, por isso, podem ser mais lentos do que os não-recursivos.
De alguma forma, a recursão fornece uma maneira natural de se abordar uma grande variedade de problemas, alguns dos quais poderiam ter soluções não-recursivas extremamente complexas. Problemas onde se desejam calcular valores para uma sequência definida recursivamente são naturalmente escritos em algoritmos recursivos. Muitas linguagens de programação permitem o uso de recursão (infelizmente, nem todas).
Exemplo 3: No Exemplo 2, uma definição recursiva foi dada para multiplicar dois inteiros positivos m e n. Uma versão em Pascal de um algoritmo recursivo para multiplicação baseada nesta definição é dada abaixo:

```
function Produto(m,n:integer):integer;
{calcula recursivamente o produto de m por n}

begin
      if n = 1 then
         Produto := m
      else
         Produto := Produto(m, n− 1) +m;
end;
```

Um algoritmo recursivo chama ele próprio para valores “menores” que o valor de entrada. Suponha um problema que pode ser resolvido pela solução de versões menores do mesmo problema, e que essas versões tornam-se, em algum momento, casos triviais que podem ser manipulados facilmente. Então, um algoritmo recursivo pode ser útil, mesmo que o problema original não seja colocado de forma recursiva. Para nos convencermos de que um algoritmo recursivo funciona não necessitamos começar com um particular valor de entrada, e a partir daí trabalhar com os valores anteriores até alcançarmos o caso trivial e então voltar para calcular o valor inicial.

Acompanhamos este procedimento para computar o valor de $S(3)$ apenas para ilustrar o mecanismo computacional de recursividade. Em vez disso, nós poderíamos verificar o caso trivial (da mesma forma que verificamos o passo básico em uma prova por indução) e verificar que se o algoritmo funciona corretamente quando é chamado para valores de entrada menores, então ele resolve o problema para o valor original de entrada (este fato é semelhante a provar $P(k+1)$ a partir da hipótese $P(k)$ em uma prova por indução).

Resolvendo Relações de Recorrência

Para obtermos o valor $S(n)$ da sequência S, desenvolvemos dois algoritmos, um iterativo e outro recursivo. No entanto, existe ainda uma forma mais fácil para computar $S(n)$. Lembremos que

$$S(1) = 2 \textbf{ (1)}$$
$$S(n) = 2S(n-1) \text{ para } n \geq 2 \textbf{ (2)}$$

Como

$$S(1) = 2 = 2^1$$
$$S(2) = 4 = 2^2$$
$$S(3) = 8 = 2^3$$
$$S(4) = 16 = 2^4$$

e assim por diante, podemos concluir que:

$$S(n) = 2^n \textbf{ (3)}$$

Usando a equação **(3)**, podemos escolher um valor para n e computar $S(n)$ sem a necessidade de computarmos inicialmente S para valores menores que n.

Uma equação como a **(3)**, onde podemos substituir um valor e obter diretamente o seu resultado, é chamada **solução de forma fechada** para a relação de recorrência **(2)** sujeita à hipótese básica (base de recorrência) estabelecida em **(1)**. Encontrar uma solução de forma fechada chama-se **resolver** a relação de recorrência. Obviamente, é interessante encontrar uma solução de forma fechada sempre que for possível.
Uma técnica utilizada para resolver relações de recorrência é uma abordagem do tipo “expanda, suponha e verifique”, que usa repetidamente a relação de recorrência para expandir a expressão para o n-ésimo termo, até que um padrão geral de comportamento da expressão possa ser deduzido. Finalmente, a suposição é verificada (demonstrada) por indução matemática.

Exemplo 4: Consideremos novamente a hipótese básica e a relação de recorrência para a sequência S:

$S(1) = 2$ **(4)**

$S(n) = 2S(n-1)$ para $n \geq 2$ **(5)**

Vamos supor que ainda não conhecemos a solução de forma fechada, e usar a abordagem de expandir, supor e verificar para encontrá-la. Começando com $S(n)$, expandiremos pelo uso repetido da relação de recorrência. Lembrando que a relação de recorrência é a fórmula que diz que qualquer valor de S pode ser substituído por duas vezes o valor anterior anterior de S. Aplicaremos os esta fórmula a S para os valores n, $n-1$, $n-2$, e assim por diante:

$$\begin{aligned} S(n) &= 2S(n-1) \\ &= 2[2S(n-2)] = 2^2 S(n-2) \\ &= 2^2[2S(n-3)] = 2^3 S(n-3) \\ &\vdots \end{aligned}$$

Observando a evolução do padrão, podemos supor que após k expansões sucessivas, a equação terá a forma:

$$S(n) = 2^k S(n-k)$$

Esta expansão dos valores de S em termos dos valores inferiores de S deve parar quando $n - k = 1$, ou seja, quando $k = n - 1$. Neste ponto, teremos

$$\begin{aligned} S(n) &= 2^{n-1} S[n - (n-1)] \\ &= 2^{n-1} S(1) = 2^{n-1}(2) = 2^n \end{aligned}$$

que é a expressão da solução de forma fechada.
Ainda não terminamos, pois o que fizemos até agora foi supor encontrar um padrão geral. Devemos confirmar agora nossa solução de forma fechada utilizando indução em n. A afirmação que desejamos demonstrar é, portanto, $S(n) = 2^n$ para $n \geq 1$.
Para a hipótese básica, $S(1) = 2^1$. Este resultado é verdadeiro pela equação **(4)**. Suponhamos então que $S(k) = 2^k$. Temos então:
$S(k+1) = 2S(k)$ (pela equação **(5)**)
$= 2(2^k)$ (pela hipótese indutiva)
$= 2^{k+1}$
Este resultado prova que a nossa solução de forma fechada está correta.

Exercício 1 Encontre uma solução de forma fechada para a relação de recorrência, sujeita à hipótese básica para a sequência T:

1. $T(1) = 1$

2. $T(n) = T(n-1) + 3$ para $n \geq 2$

Dica: Expanda, suponha e verifique.

$T(n) = T(n-1) + 3$

$= [T(n-2) + 3] + 3 = T(n-2) + 2.3$

$= [T(n-3) + 3] + 2.3 = T(n-3) + 3.3$

Em geral, parece que

$$T(n) = T(n-k) + k.3$$

Quando $n - k = 1$, isto é, $k = n - 1$,

$T(n) = T(1) + (n-1).3 = 1 + (n-1).3$

Agora demonstre por indução que $T(n) = 1 + (n-1).3$.
$T(1) : T(1) = 1 + (1-1).3 = 1$, verdadeiro
Admita que $T(k) : T(k) = 1 + (k-1).3$
Mostre $T(k+1) : T(k+1) \stackrel{?}{=} 1 + k.3$
$T(k+1) = T(k) + 3$
$= 1 + (k-1).3 + 3$
$= 1 + k.3$

Bibliografia

[1] GERSTING, J. L. *Fundamentos Matemáticos para a Ciência da Computação*, LTC, 5a. ed., Rio de Janeiro, 2004.
[2] SCHEINERMAN, E. R. *Matemática discreta. Uma introdução*, Thomson Pioneira, São Paulo, 2003.

Araraquara, 28 de outubro de 2021.