8

Recursividade

A recursividade pode ser considerada um método eficaz para resolver um problema originalmente complexo, reduzindo-o em pequenas ocorrências do problema principal. Assim, segue a idéia do dividir para conquistar (*divide and conquer*), ou seja, para resolver um problema complexo, nós o dividimos em partes menores, nas quais a complexidade pode ser considerada menor em relação ao problema original. Resolvendo, isoladamente, cada uma das pequenas partes, podemos obter a solução do problema original como um todo.

Um algoritmo é dito recursivo quando chama a si mesmo ou chama uma seqüência de outros algoritmos, e um deles chama novamente o primeiro algoritmo. Observamos, dessa forma, que muitas vezes um algoritmo recursivo representa de forma mais natural a solução de um determinado problema.

A grande dificuldade no uso da técnica recursiva está no reconhecimento das situações, nas quais a técnica não compromete as questões de espaço (memória ocupada) e de tempo (tempo de execução), mantendo-se, ao mesmo tempo, uma boa compreensão da solução do problema, ou seja, o algoritmo obtido representa a solução do problema de forma simplificada.

A função fatorial

Utilizaremos o fatorial para apresentar os conceitos iniciais da recursividade. Posteriormente, veremos que a recursividade é um método inadequado para resolver o fatorial. Na matemática, a função fatorial de um número positivo inteiro é geralmente definida pela seguinte fórmula:

$$n! = n \times (n-) \times \ldots \times 1$$

Porém, para calcular o fatorial, precisamos de uma definição mais precisa, tal como a seguinte:

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

Calculando 4! usando a última definição, faríamos:

$$
\begin{aligned}
4! &= 4 \times 3! \\
&= 4 \times (3 \times 2!) \\
&= 4 \times (3 \times (2 \times 1)) \\
&= 4 \times (3 \times (2 \times (1 \times 0!))) \\
&= 4 \times (3 \times (2 \times (1 \times 1))) \\
&= 4 \times (3 \times (2 \times 1)) \\
&= 4 \times (3 \times 2) \\
&= 4 \times 6 \\
&= 24
\end{aligned}
$$

Os passos anteriores ilustram a essência da forma que a recursividade trabalha. Para obter a resposta do problema original, primeiramente um método geral é usado para reduzir o problema original em um ou mais problemas de natureza semelhante, porém de tamanho menor. O mesmo método é, então, usado para esses subproblemas, e a recursividade continua até que o tamanho dos subproblemas seja reduzido à solução trivial, ponto em que a solução é dada diretamente sem mais precisar da recursividade.

Em outras palavras, todo processo recursivo consiste em duas partes:

a. *Solução trivial*: É conseguida por definição, ou seja, não é necessário fazer uso da recursividade para obtê-la;

b. *Solução geral*: É a solução genérica que funciona em uma parte menor do problema original, mas que pode ser aplicada integralmente ao problema original.

Para exercitar o processo de criação de um algoritmo recursivo citado anteriormente, apresentamos uma versão recursiva para calcular o fatorial de *n*.

```
int fat(int n)
/* fat :versão recursiva
n é um inteiro não negativo
a função retorna o valor do fatorial de n */
{
     if (n==0)
          return (1);
     else
          return (n*fatorial(n-1));
}
```


Para compreendermos o funcionamento da versão recursiva sugerida para calcular o fatorial de um número *n*, podemos usar como apoio as pilhas para ilustrar a evolução do conteúdo da função fatorial.

Como já comentamos, a área de memória principal denominada por pilhas é usada para o armazenamento das variáveis locais. Toda vez que uma função é chamada, suas variáveis locais são criadas na pilha de memória. À medida que a função é encerrada, essas variáveis locais são desempilhadas.

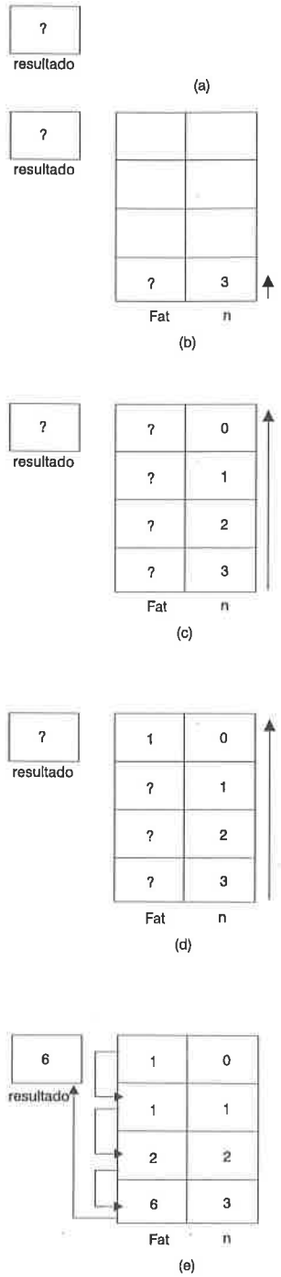
Suponha, então, que em um certo ponto do *main ( )* de um programa C houve a seguinte chamada: *resultado = fat (3);*. Estamos considerando que a variável *resultado* é global e ocupa um certo endereço de memória. Isso é ilustrado na Figura 8.1 (a). Observe que, no momento da chamada da função *fat* (3), o conteúdo de *resultado* está indefinido.

Após a chamada de *fat*(3), podemos visualizar, na Figura 8.1 (b), que são criadas na pilha as variáveis *fat* e *n*. Note que o conteúdo de *fat* está indefinido e o de *n* tem o valor de 3.

Como *n* é diferente de 0, ocorre uma chamada de *fat* (2). Isso está ilustrado na Figura 8.1 (c). Depois, nessa mesma figura, podemos encontrar a representação de *fat*(2) chamando *fat*(1) e *fat*(1) chamando *fat*(0). Observe que até esse momento tanto o conteúdo de *resultado* como de *fat* está indefinido.

A Figura 8.1 (d) ilustra o momento da chamada *fat*(0) que tem a *solução trivial*, ou seja, o valor de 1, ocorrendo, dessa forma, a primeira finalização das chamadas recursivas. Na Figura 8.1 (e), encontramos o processo de desempilhamento das variáveis locais *n* e *fat*. Note que, ao término do último desempilhamento de *fat*, teremos o valor de 6 que é o fatorial de 3; esse valor será atribuído à variável *resultado*. Observe todo esse processo:

Figura 8.1
Uma possível ilustração da chamada recursiva fat (3).



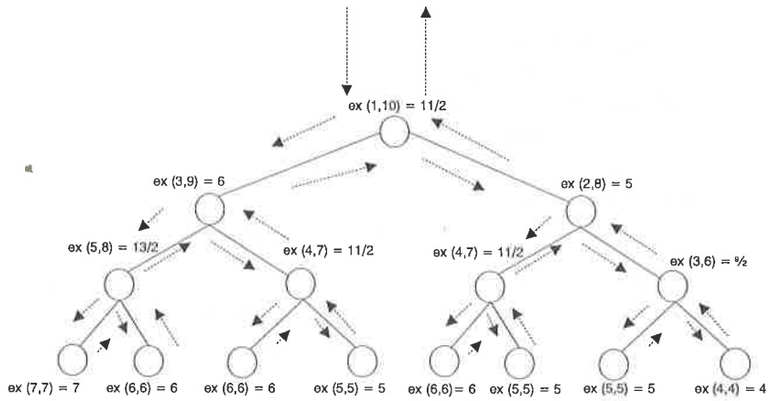
Outra possibilidade muito interessante para analisarmos o comportamento de um algoritmo recursivo é construir a chamada árvore de recursão. Para exemplificar a construção dessa árvore, vamos considerar a seguinte função recursiva:

```
float ex( float a, float b)
{
      if (a >= b)
            return ( (a+b)/2);
      else
            return ( ex(ex(a+2,b-1),ex(a+1, b-2));
}
```

A função *ex()*, de fato, não realiza nada de útil. Ela será usada apenas para ilustrar a construção da árvore de recursão, que é considerada uma boa ferramenta para acompanhar a evolução de um algoritmo recursivo.

Supondo que seja realizada uma chamada do tipo *ex (1,10)*, podemos visualizar, na Figura 8.2, o desenho da respectiva árvore de recursão. Note que o caminho percorrido durante a execução da função está indicado pelas setas pontilhadas. Observe:

Figura 8.2
Árvore de recursão para a chamada de ex (1,10).



Rotinas recursivas e pilhas

Como já apresentamos, o controle de chamadas e de retornos de rotinas pode ser efetuado por uma pilha (criada e mantida automaticamente pelo sistema). De fato, quando uma rotina é chamada, empilha-se o endereço de retorno e também todas as variáveis locais são recriadas na pilha, porque, para todo efeito, são outras variáveis locais. Quando ocorre o retorno da rotina, as variáveis locais que foram criadas deixam de existir.

O cálculo do MDC (Máximo Divisor Comum) pode ser elaborado por meio de uma versão de algoritmo recursivos, como é apresentado a seguir:

```
int mdc(int p1, int p2)
{
      if (p2==0)
            mdc = p1;
      else
            mdc = mdc(p2, p1 % p2);
}
```

Com o exemplo anterior, podemos perceber que um algoritmo recursivo pode ser conciso e elegante, porém a compreensão dos detalhes de seu funcionamento computacional pode exigir um exaustivo acompanhamento do programa.

Os computadores podem usar pilhas para controlar a evolução da execução de um programa. A mente humana não é tão boa para realizar essas tarefas. Parece ser difícil para uma pessoa guardar uma longa seqüência de resultados e, então, voltar para terminar um trabalho.

Para demonstrar esse fato, considere o exemplo a seguir, que é uma rima infantil norte-americana:

As I was going to St. Ives,
I met a man with seven wives.
Each wife had seven sacks,
Each sack had seven cats,
Each cat had seven kits:
Kits, cats, sacks and wives,
How many were there going to St. Ives?

Responder de forma rápida a pergunta final pode ser mais difícil do que parece. Isso acontece porque parece existir uma certa dificuldade para uma pessoa acompanhar simultaneamente muitos cálculos computacionais parciais; torna-se, portanto, necessário para nós pensarmos em outras possibilidades para fazer tais acompanhamentos, como, por exemplo, ilustrações especiais e árvores de recursividade. Veremos como essas dificuldades ficam mais evidentes apresentando um jogo chamado Torres de Hanói.

As Torres de Hanói

No século XIX, um jogo chamado Torres de Hanói apareceu na Europa com um material promocional, explicando que o jogo representava uma tarefa realizada no Templo de Brahma (montanhas de Hanói – sacerdotes brâmanes).

Dizia, ainda, que na criação do mundo aos sacerdotes foi dada uma plataforma de metal na qual existiam três hastes de diamante.

Na primeira haste existiam 64 discos de ouro, cada um ligeiramente menor em relação ao que estava por cima. (A versão mais exótica vendida na Europa da época possuía oito discos e três hastes de madeira.)

Os sacerdotes foram encarregados de mover todos os discos de ouro do primeiro disco para o terceiro, porém com as seguintes condições:

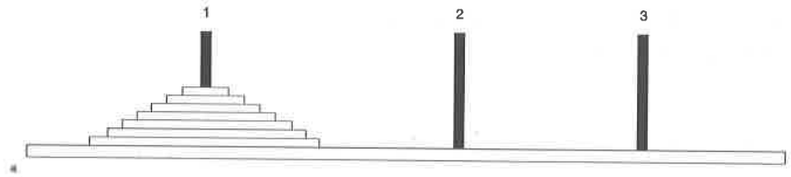
a. Mover apenas um disco por vez;

b. E nenhum disco poderia ficar sobre um disco menor.

Entretanto, poderiam usar qualquer haste de diamante como descanso para os discos.

Aos sacerdotes, foi dito também que quando eles terminassem de movimentar os 64 discos de ouro, isso significaria o fim do mundo!

Figura 8.3
Ilustração do jogo Torres de Hanói.



Nossa tarefa, naturalmente, é escrever um programa que imprima uma lista de instruções para os sacerdotes. Podemos resumir nossa tarefa por meio da seguinte instrução:

Move (64,1,3,2)

A instrução acima tem o seguinte significado:

'Mover 64 discos da torre 1 para a torre 3, usando a torre 2 como armazenamento temporário'.

Desenvolvendo uma solução para o jogo Torres de Hanói

A idéia que pode nos dar uma solução é concentrar nossa atenção não no primeiro passo (que é justamente mover o primeiro disco para algum lugar), mas sim no passo mais difícil, que é mover o último disco.

Não há outra forma de chegar ao último disco até que os 63 discos anteriores tenham sido removidos; ainda mais, eles devem estar na torre 2 para que possamos movimentar o último disco da torre 1 para a torre 3. Lembre-se de que somente um disco pode ser movido por vez, e o último (o maior) não pode nunca estar sobre os demais. Então, quando movemos o disco 1, não pode existir nenhum disco nas torres 1 e 3.

Assim, podemos resumir os passos do nosso algoritmo para as Torres de Hanói como:

Move (63, 1,2,3) // Move 63 discos da torre 1 para 2 (torre 3 temporária)
Imprima 'Mova o disco nr. 64 da torre 1 para torre 3'
Move (63,2,3,1) // Move 63 discos da torre 2 para torre 3 (torre 1 temporária)

Demos um pequeno passo em direção à solução, aliás pequeníssimo passo, se considerarmos que temos de descrever ainda como movimentar 63 discos. Entretanto, é um passo significativo, porque não existe nenhuma razão de não utilizarmos o mesmo passo para movimentar os 63 discos remanescentes.

É essa a idéia da recursividade. Descrevemos como realizar o 'passo-chave' e consideramos que o restante do problema é feito — essencialmente — da mesma forma. Essa é a idéia do dividir para conquistar (*divide and conquer*). Para resolver um problema, dividimos o problema original em partes cada vez menores, cada uma das quais mais fáceis de serem resolvidas do que o problema original.

Realizando um refinamento nos passos anteriores

Para escrever o algoritmo na sua forma final, devemos saber, em cada passo, qual torre será usada como armazenamento temporário; assim, poderemos construir a função 'move' com as seguintes especificações:

`move (int qtd, int inicio, int fim, int aux)`

Condição inicial: Existe, no mínimo, *qtd* discos na torre *inicio*. O disco de topo (se existir) nas torres *aux* e *fim* é maior do que qualquer outros *qtd* discos presentes na torre *inicio*.

Condição final: Os *qtd* discos que estavam na torre *inicio* foram movidos para a torre *fim*. A torre *aux* (usada como armazenamento temporário) retornou para sua posição inicial.

Supondo que nossa tarefa deve terminar em um número finito de passos (mesmo que isso signifique o fim do mundo), temos de achar uma forma de parar a recursividade. Uma regra óbvia é que, quando não existirem mais discos para serem movidos, não haverá mais nada a se fazer . . .

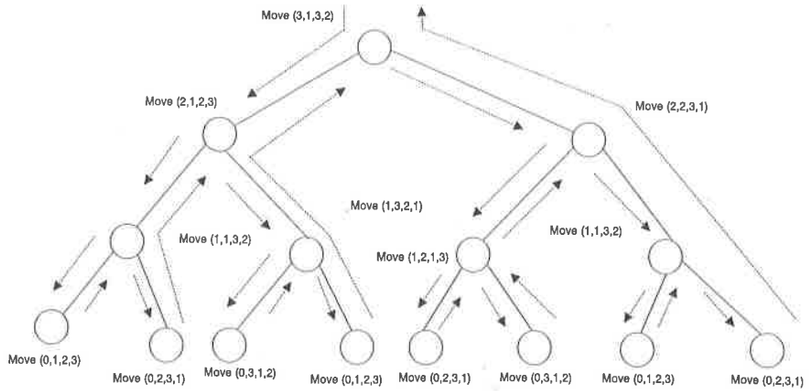
Agora, podemos completar o programa que considera essas regras. Observe:

```
#define discos = 64 // faça essa constante ser menor para testar o
programa
void move(int qtd, int inicio, int fim, int aux);

main( )
{
     move (discos,1,3,2);
}

void move (int qtd, int inicio, int fim, int aux)
{
     if (qtd > 0) {
          move(qtd -1, inicio, aux, fim);
          printf("Mova disco %i de %i para %i", qtd, inicio, fim);
          move(qtd-1, aux, fim, inicio);
     }
}
```

Acompanhando e analisando o programa recursivo de Torres de Hanói

Na Figura 8.4, podemos encontrar a árvore de recursividade das Torres de Hanói com três discos. A evolução da execução segue seu caminho, o qual é mostrado em pontilhado. Lembre-se de que *Move (3,1,3,2)* significa que é para movimentar o disco de número 3 da haste 1 para a haste 3, usando a haste 2 como auxílio; *Move (2,1,2,3)* é para movimentar o disco de número 2 da haste 1 para a haste 2 utilizando a haste 3 como auxílio; com *Move (0,2,3,1)* não é movimentado nenhum disco, e assim por diante.

Figura 8.4
Árvore de recursividade das Torres de Hanói.



Logo a seguir, podemos visualizar outra versão de programa recursivo para solucionar o jogo Torres de Hanói. Sugerimos que o leitor faça a árvore de recursividade para esta nova versão e realize uma comparação com a árvore de recursividade do programa anterior.

```
hanoi (int n, char início, char fim, char aux)
{
      /* um só disco a solução é trivial, faça o movimento e retorne */
      if (n= =1) {
      printf("\nmover disco 1 %c %c", início, fim);
      return;
      /* Mover os primeiros n-1 discos de A p/ B, usando C como auxiliar
*/
      hanoi(n-1, início, aux, fim);
      /* Move último disco de A p/ C */
      printf("\nmover disco %c %c %c", n, início, fim);
      hanoi(n-1,aux, fim, início);
} /* fim hanoi*/
```

Analisando a árvore de recursividade do primeiro programa recursivo, tentaremos realizar algumas análises de desempenho da solução recursiva proposta para as Torres de Hanói. Pela árvore de recursividade obtida para três discos, notamos que desenhar essa árvore para 64 discos pode ser uma tarefa praticamente impossível, analisando o ponto de vista prático. Entretanto, usando a nossa imaginação e algumas suposições, torna-se possível calcular quantas instruções serão necessárias para movimentar todos os 64 discos.

Sabemos que uma instrução é exibida em cada vértice (ou nó), exceto para as folhas (vértices sem filhos, também chamados de vértices externos), porque representam chamadas com *qtd == 0*.

O número de vértices do tipo não-folhas é:

$$1 + 2 + 4 + \ldots + 2^{63} = 2^0 + 2^1 + 2^2 + \ldots + 2^{63} = 2^{64} - 1$$

A partir disso, sabemos que o número de movimentos exigidos para movimentar todos os 64 discos é $2^{64} - 1$. Podemos, então, estimar a magnitude desse valor usando a seguinte aproximação:

$$10^3 = 1000 \approx 1024 = 2^{10}$$

Dessa forma, o número aproximado de movimentos é:

$$2^{64} = 2^4 \times 2^{60} \approx 2^4 \times 10^{18} = 1{,}6 \times 10^{19}$$

Vamos tratar agora sobre o fim do mundo. Existem aproximadamente $3{,}2 \times 10^7$ segundos em um ano. Suponha que as instruções pudessem ser conduzidas em um ritmo alucinante de um movimento por segundo (os sacerdotes têm folga para praticar!). A tarefa total consumirá quase 5×10^{11} anos. Os astrônomos estimam que o universo tem, no mínimo, 20 bilhões (2×10^{10}) de anos; portanto, de acordo com essa lenda, o mundo durará ainda um longo tempo: cerca de 25 vezes mais do que já tem!!!

Questões sobre tempo e espaço

Nós devemos, cuidadosamente, notar que embora não exista um computador que possa rodar o programa de Torres de Hanói, ele falharia por falta de tempo, mas certamente não por falta de espaço. O espaço de memória necessário é usado somente para fazer o controle das 64 chamadas recursivas, mas o tempo necessário exigido é aquele para fazer 2^{64} cálculos. Com esse exemplo, percebemos que em um algoritmo recursivo pode ser necessário fazer uma boa avaliação sobre as questões de tempo (tempo de execução do algoritmo) e de espaço (memória usada para executar o algoritmo).

Desenvolvendo algoritmos recursivos

Recursividade é uma ferramenta que permite ao programador concentrar-se no 'passo-chave' de um algoritmo, sem, inicialmente, preocupar-se em integrar esse passo com os outros. Como é comum na solução de problemas, a primeira abordagem geralmente considera vários exemplos simples. Depois que estes são compreendidos, tentamos formular um método que trabalhe de forma genérica.

Podemos considerar alguns aspectos importantes no desenvolvimento de algoritmos recursivos. Entre eles:

a. Ache o passo-chave. Pergunte-se: "Como esse problema pode ser dividido?" ou "Como posso desenvolver o passo-chave da solução?". Tenha o cuidado de manter sua resposta o mais simples possível, mas genericamente aplicável. Uma vez que tenha conseguido um passo simples que conduza à solução, pergunte se o restante do problema pode ser feito da mesma forma ou de forma semelhante. Modifique o método, se necessário, para que seja suficientemente genérico;

b. Ache uma regra de parada; ela indica que o problema ou uma parte razoável dele foi feito. Essa regra é, no geral, pequena e dá a solução trivial sem a necessidade da recursividade;

c. Monte seu algoritmo: Combine a regra de parada e o 'passo-chave' usando uma estrutura condicional, como, por exemplo, *if . . . then . . . else*.

d. Verifique o término: Um passo de grande importância é certificar-se de que a recursividade sempre terminará. Comece com uma situação geral e verifique se, em um número finito de passos, a regra de parada será satisfeita e a recursividade irá terminar. Certifique-se de que seu algoritmo manipula corretamente os casos extremos. Quando chamado para nada fazer, um algoritmo deve retornar de forma elegante, mas isso é mais importante em um algoritmo recursivo, porque uma chamada para não fazer nada, normalmente, é a regra de parada. Observe, então, que uma chamada para nada fazer, geralmente, não é um erro para um algoritmo recursivo. Portanto, não é muito apropriado para uma função recursiva gerar uma mensagem quando ela realiza uma chamada vazia; em vez disso, ela deve retornar silenciosamente.

e. Desenhe uma árvore de recursividade; isso é a ferramenta-chave para analisar algoritmos recursivos. Como foi visto nas Torres de Hanói, a altura da árvore de recursão está intimamente relacionada à quantidade de memória de que o programa necessitará, e o total do tamanho da árvore reflete o número de vezes que o 'passo-chave' será feito, e daí o tempo total que o programa utilizará. É útil desenhar a árvore de recursividade para um ou dois exemplos simples e apropriados ao seu problema.

Recursividade de cauda

Suponha que a última ação de uma função seja fazer uma chamada recursiva para ela própria. Na implementação da recursividade, como já foi mostrado, as variáveis locais da função são empilhadas para dentro da pilha quando a chamada recursiva é iniciada. Quando esta termina, essas variáveis locais serão desempilhadas da pilha e, com isso, serão restaurados seus valores. Mas essa restauração não tem importância, porque a chamada recursiva foi a última ação da função; portanto, assim que a função termina, as variáveis locais recém-restauradas são

imediatamente descartadas. Quando a última ação de uma função é uma chamada recursiva para ela própria, é desnecessário usar a pilha porque, como foi visto, as variáveis locais não necessitam ser preservadas.

Essa situação, na qual a chamada recursiva é o último comando da função, é denominada recursividade de cauda. Isso forma um '*looping*' que será repetido até que uma condição de parada seja satisfeita. A recursividade de cauda pode sofrer transformações até que seja obtida uma estrutura de interação, quando observaremos ganhos em tempo e em espaço.

Com o objetivo de mostrarmos que muitas vezes não devemos usar uma versão recursiva quando é possível obtermos uma versão interativa equivalente, vamos considerar dois exemplos: o primeiro já foi visto e calcula o fatorial de um número de modo recursivo, e o segundo faz o mesmo de forma interativa e está apresentado logo após a versão recursiva. Observe:

```
int fat(int n)
/* fat : versão recursiva
n é um inteiro não negativo
a função retorna o valor do fatorial de n */
{
     if (n==0)  return (1);
     else       return (n*fatorial(n-1));
}

int fatorial (int n)
/* fatorial     : versão interativa
n é um inteiro não negativo
retorna o valor do fatorial de n*/
{
     int k, produto = 1;
     for (k = 1; k<= n; k++)
          produto= produto*k;
     return produto;
}
```

Vamos colocar a seguinte pergunta: Qual dos programas usa menos espaço de memória?

À primeira vista, pode parecer que a versão recursiva o faz. Ela não tem variáveis locais e o programa interativo tem duas. Entretanto, o programa recursivo utilizará uma pilha e a preencherá com *n* números, n, n-1, n-2, . . ., 2,1, que são os parâmetros passados para cada chamada recursiva. Depois do término da última chamada de recursividade, serão multiplicados na mesma ordem, tal como faz o segundo programa. Acompanhe a execução recursiva para n = 5:

```
fatorial (5) = 5*fatorial (4)
             = 5*(4*fatorial (3))
             = 5*(4*(3*fatoriall (2)))
             = 5*(4*(3*(2*fatorial (1))))
             = 5*(4*(3*(2*(1*(fatorial (0))))))
             = 5*(4*(3*(2*(1*))))
             = 5*(4*(3*(2*1)))
             = 5*(4*(3*2))
             = 5*(4*6)
             = 5*24
             = 120
```

Sugerimos que você, amigo leitor, faça, como exercício de fixação, o acompanhamento do funcionamento do programa fatorial na versão recursiva, assumindo para tanto *fat (5)*. Recomendamos que sejam usadas duas pilhas, uma para o valor de *n* e outra para *fat (5)*. Essa tarefa já foi realizada anteriormente para fat (3).

Podemos observar que a versão recursiva consume mais espaço de memória e levará muito mais tempo para executar, pois ela armazena e recupera todos os números, além de multiplicá-los.

Outro exemplo, que representa um uso inadequado (devido ao desperdício de tempo e de espaço) da recursividade, é o algoritmo usado para obter a série de *Fibonacci* (citado, tanto quanto o fatorial, como exemplos de aplicação de recursividade). No caso do algoritmo para exibir a série de *Fibonacci*, pode ser possível também mostrar que é melhor utilizar um algoritmo com interação do que um algoritmo recursivo.

Só para recordar, a série de *Fibonacci* pode ser definida da seguinte maneira:

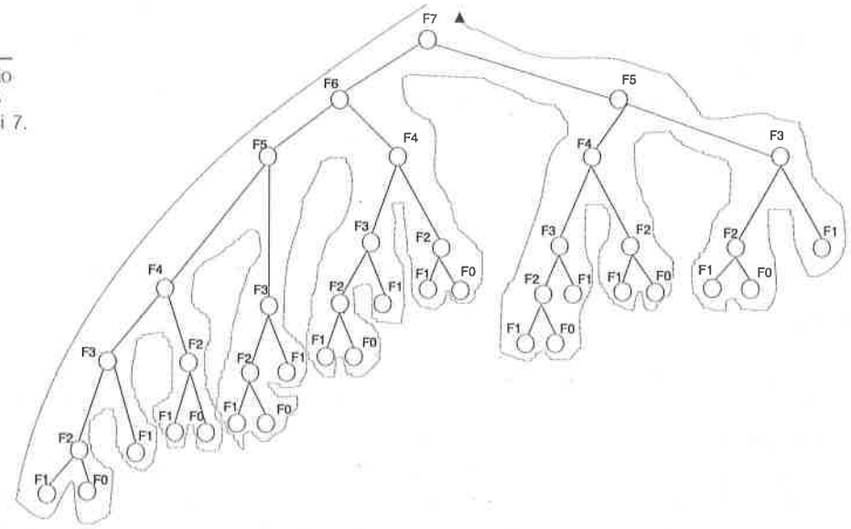
f(0) = 0, f(1) = 1, f(n) = f(n-1) + f(n-2) para n >= 2

Uma versão recursiva seria:

```
int fibonacci(int n)
/*fibonacci      : versão recursiva
o parâmetro n é um inteiro não negativo
a função retorna o número Fibonacci */
{
     if (n < = 0)
               return 0;
     else
          if (n == 1)
          return 1;
     else
          return ficonacci(n - 1) + fibonacci (n - 2);
}
```


Na Figura 8.5, construiu-se a árvore de recursividade para a versão de programa recursiva do cálculo *Fibonacci* para o número 7. Analisando a árvore de recursão, notamos que o programa repete sempre o mesmo cálculo seguidas vezes. Isso faz com que o tempo usado pela versão recursiva para calcular um *F(n)* cresça exponencialmente com o valor de *n*. Vamos agora construir uma versão interativa para vermos se algo muda no tempo de execução. Observe:

Figura 8.5
Árvore de recursão para o cálculo do número Fibonacci 7.



A versão interativa para calcular um número *Fibonacci* pode ficar como é exibido abaixo:

```
int fibonacci(int n)
/* fibonacci : versão interativa
O parâmetro n é inteiro positivo
A função retorna o enésimo número Fibonacci */
{
      int ultimo_menos_um;  // número Fibonacci prévio segundo Fi -2
      int ultimo_valor; // número Fibonacci prévio Fi-1
      int atual; // número Fibonacci atual Fi
      if (n < = 0)
            return 0;
      else
      if (n = = 1)
            return 1;
      else {
            ultimo_menos_um = 0;
            ultimo_valor = 1
            for (int i =2; i < = n; i++){
                  atual=ultimo_menos_um + ultimo_valor;
```

```
                ultimo_menos_um = ultimo_valor;
                ultimo_valor = atual;
            }
            return atual;
        }
    }
```

Podemos notar que a função interativa consome um tempo que aumenta linearmente, isto é, diretamente proporcional a *n*. Considerando que a versão recursiva apresenta um consumo de tempo na forma exponencial e a versão interativa um consumo de tempo na forma linear, concluímos que é preferível utilizarmos a versão interativa para obtermos o valor *Fibonacci* para um *n* elevado, porque a diferença do tempo de execução dos dois programas será considerável.

Comparando os algoritmos recursivos da série de *Fibonacci* e *Hanói*

A função recursiva para a série *Fibonacci* e Torres de *Hanói* tem uma semelhança na forma de dividir para conquistar. Cada uma consiste, essencialmente, de duas chamadas recursivas para elas próprias, menores do que o original. Por que, então, o programa *Hanói* é eficiente, enquanto o programa *Fibonacci* é bastante ineficiente? A resposta surge quando consideramos o tamanho da saída. No programa *Fibonacci*, estamos calculando somente um número e desejamos completar esse cálculo em poucos passos. Isso é feito pela função interativa, e não pela recursiva. O programa *Hanói* e, por outro lado, o tamanho da saída é o número de instruções a serem exibidas, que aumenta exponencialmente com o número de discos. Daí que qualquer implementação para as Torres de *Hanói*, necessariamente, irá exigir um tempo que aumentará exponencialmente com o número de discos.

Diretrizes e conclusões gerais sobre a recursividade

A construção da árvore da recursividade é uma boa recomendação para ajudar na decisão entre usar um algoritmo recursivo e um não recursivo. Se ela tiver uma forma simples, a versão interativa poderá ser melhor. Se ela envolver tarefas de duplicação, outras estruturas poderão ser mais apropriadas, como, por exemplo, as pilhas, fazendo desaparecer a necessidade de recursividade. Entretanto, se a árvore de recursividade parece ser consistente, com poucas tarefas de duplicação, então a recursividade poderá ser o método para a solução mais natural.

A recursividade é considerada uma abordagem *top-down* para resolver um problema. Ela divide o problema em partes ou seleciona um passo-chave, adiando o uso dos demais.

A interação está mais para uma abordagem *bottom-up*. Nessa abordagem, começa-se com o que é conhecido e, desse ponto em diante, a solução é construída passo a passo.

De forma geral, a recursividade pode ser substituída pela interação e pelas pilhas. Por outro lado, é verdade também que qualquer programa interativo que manipula uma pilha pode ser substituído por um programa recursivo sem nenhuma pilha.

Todas essas considerações fazem com que um programador cuidadoso pergunte sempre se a recursividade pode ser evitada, mas que também pergunte quando um programa usa pilhas e se a introdução da recursividade pode produzir um programa que represente um algoritmo mais natural e, portanto, de melhor entendimento, agregando melhorias na abordagem e nos resultados do problema.

EXERCÍCIOS DE APROFUNDAMENTO

1. Considere o seguinte algoritmo:

```
algoritmo exemplo1 (x : inteiro)
se (x<5) então
|           exemplo1 ← (3*x)
|     senão
|           exemplo1 ← (2 * exemplo1( x - 5 ) + 7)
fim se
```

O que é retornado nas seguintes chamadas?

Exemplo1(4) ?
Exemplo1(10) ?
Exemplo1(12) ?

2. Considere o seguinte algoritmo:

```
algoritmo exemplo2 ( x:inteiro, y : inteiro)
se ( x < y) então
|           exemplo2 ← -3
|     senão
|           exemplo2 ← exemplo2 (( x - y, y + 3 ) + y )
fim se
```

O que é retornado nas seguintes chamadas?

Exemplo2(2, 7) ?
Exemplo2(5, 3) ?
Exemplo(15, 3) ?

3. Considere o seguinte algoritmo:

```
algoritmo exemplo3 (x:inteiro, y:inteiro)
se (x> y) então
|                exemplo3 ← -1
|     senão
|       se (x = y)  então
|       |        exemplo3 ← 1
|       |   senão
|       |        exemplo3 ← (x * exemplo3(x + 1, y) )
|       fim_se
fim_se
```

O que é retornado nas seguintes chamadas?

Exemplo3(10, 4)
Exemplo3(4, 3)
Exemplo3(4, 7)
Exemplo3(0, 0)

4. Elabore um programa recursivo em C que calcule o MDC de dois inteiros usando o algoritmo de Euclides. Teste seu programa com os seguintes exemplos: MDC (4,28); MDC (22,4) e MDC (22,5). O algoritmo de Euclides pode ser descrito da seguinte forma: dividimos o número maior pelo menor e pegamos o resto. Na continuação do processo, pegamos o divisor e o dividimos pelo resto até que este seja zero. O quociente da última divisão será o próprio MDC. Por exemplo, suponha que sejam os números 928 e 100. Dividindo-os, obteremos um resto de 28. Em seguida, dividimos 100 por 28 e obtemos um resto de 16, depois 28 por 16 e obtemos um resto de 12, depois 16 por 12 e obtemos um resto de quatro e, finalmente, 12 por 4 e obtemos um resto de zero. Então, o MDC é 4, que é o quociente da última divisão. A seguir, encontramos uma definição matemática do algoritmo de Euclides, que pode ser útil para o desenvolvimento do algoritmo recursivo.

$$MDC\ (x,y) = \begin{cases} MDC\ (y,x) & \text{se } x < y \\ y & \text{se } x = 0 \\ MDC(y,\ x \bmod y) & \text{se } x > y \end{cases}$$

5. Codifique um programa recursivo para verificar se um caractere específico está ou não em uma *string*.

6. Codifique um programa recursivo para contar todas as ocorrências de um caractere específico em uma *string*.

7. Codifique um programa recursivo que remova todas as ocorrências de um caractere específico de uma *string*.

8. Escreva um algoritmo que transforme um inteiro decimal em um número binário.

9. Um número perfeito é um número que é igual à soma de seus fatores. Por exemplo, 6 é um número perfeito porque 6 = 1 + 2 + 3. Escreva um algoritmo recursivo para calcular todos os menores números perfeitos para um determinado número.

10. Codifique um programa recursivo que faça a pesquisa binária.

11. Escreva um algoritmo recursivo que leia uma string de caracteres de um teclado e os imprima na ordem reversa.

12. Considere a função f(n) definida como é exibido abaixo, onde *n* é um inteiro positivo:

$$f(n) = \begin{cases} 0 & \text{se } n = 0; \\ f(\underline{1}\ n) & \text{se n é par, } n > 0; \\ \quad 2 & \\ 1 + f(n-1) & \text{se n é ímpar, } n > 0; \end{cases}$$

Calcule o valor de f(n) para os seguintes valores de *n*:

n = 1 n = 2 n = 3 n = 99 n = 100 n = 128

13. Considere a função f(n) definida como abaixo, onde *n* é um inteiro positivo:

$$f(n) = \begin{cases} n & \text{se } n <= 1; \\ n + f(\underline{1}\ n) & \text{se n é par, } n > 1; \\ \quad 2 & \\ f(\underline{1}(n+1)) + f(\underline{1}(n-1)) & \text{se n é ímpar, } n > 1; \\ \quad 2 \qquad\quad 2 & \end{cases}$$

Para cada um dos valores de *n*, desenhe a árvore de recursão e calcule o valor de f(n):

n = 1 n = 2 n = 3 n = 4 n = 5 n = 6

14. Compare os valores do tempo de execução para a função fatorial recursiva já apresentada no início deste capítulo, com uma função não recursiva obtida pela inicialização de uma variável local com 1 e usando um *'loop'* para calcular o produto n! = 1x2x...X_n. Para obter comparações significativas do tempo de CPU, você provavelmente precisará escrever um 'loop' no seu programa-guia, o qual repetirá o mesmo cálculo de um fatorial centenas de vezes. Haverá um *overflow* no tipo inteiro, se você calcular o fatorial para um número grande. Para prevenir isso, declare *n* e o valor da função como tipo *double*, em vez de *int*. Use arquivos de cabeçalho do tipo *'ctime.h'* ou *'time.h'* para calcular o tempo de CPU usado por programas C ou C++.

15. Confirme que o tempo de execução para o programa *Hanói* aumenta aproximadamente como uma constante multiplicada por *2n*, onde *n* é o número de discos movidos. Para fazer isso, baseando-se no algoritmo recursivo apresentado neste capítulo, faça *qtd* uma variável, coloque comentário nas linhas que escreve uma mensagem para o usuário e execute o programa para vários valores sucessivos de *qtd*, tais como 10,11, . . ., 15. Como o tempo de CPU altera de um valor para o próximo de *qtd*? Use arquivos de cabeçalho do tipo *'ctime.h'* ou *'time.h'* para calcular o tempo de CPU usado por programas C ou C++.

16. Escreva uma função recursiva para calcular a soma dos *n* primeiros números inteiros positivos.

17. Escreva uma função recursiva para calcular *x* elevado a *n* (*x* pertence aos reais e *n* aos naturais).

18. Escreva uma função recursiva para calcular as permutações de *n* elementos.

19. Escreva uma função recursiva para calcular o maior e o menor elemento de um conjunto de elementos.

20. Escreva uma função recursiva que conte quantos dígitos tem um inteiro maior ou igual a zero.

21. A função *exibeBin( )*, apresentada a seguir, exibe a representação binária de um número inteiro maior ou igual a zero.

```
void exibeBin(int n)
/* Exibe a representação binária de um número inteiro n)  */
/* Versão recursiva
    */
{
    if (n!=0){
            printf("%i", n); // n=0 ou n=1; fim
        else {
            exibeBin(n  2); // chamada recursiva, passa o quociente da
                            // divisão de n por 2
            printf("%i", n%2) // exibe o valor do resto
        }
    }
}
```


   a. Complete a função na ***parte em negrito*** que faz o cálculo do quociente da divisão por dois, de tal forma que a função passe a ter funcionalidade.

   b. Escreva uma versão interativa da função acima e compare com a versão recursiva.

   c. Escreva a sua própria versão recursiva desse problema e compare com a versão recursiva sugerida neste exercício. Quais as vantagens e as desvantagens, se existirem, de ambas as versões?

22. Dada a definição da função abaixo, avalie f(1,10), desenhando a árvore de recursão.

```
double f(double x, double y)
{
      if (x > = y)
      f = (x+y)/2;
      else
      f= f(f(x+2,y-1),f(x+1,y-2));
}
```