Algoritmos Recursivos

Um objeto é denominado recursivo quando sua definição é parcialmente feita em termos dele mesmo. A recursividade (ou recursão) é encontrada principalmente na matemática, mas está presente em algumas situações do cotidiano. Por exemplo, quando um objeto é colocado entre dois espelhos planos paralelos e frente a frente surge uma imagem recursiva, porque a imagem do objeto refletida num espelho passa a ser o objeto a ser refletido no outro espelho e, assim, sucessivamente.

Em programação, a recursividade é um mecanismo útil e poderoso que permite a uma função chamar a si mesma direta ou indiretamente, ou seja, uma função é dita recursiva se ela contém pelo menos uma chamada explícita ou implícita a si própria.

A idéia básica de um algoritmo recursivo consiste em diminuir sucessivamente o problema em um problema menor ou mais simples, até que o tamanho ou a simplicidade do problema reduzido permita resolvê-lo de forma direta, sem recorrer a si mesmo. Quando isso ocorre, diz-se que o algoritmo atingiu uma condição de parada, a qual deve estar presente em pelo menos um local dentro algoritmo. Sem esta condição o algoritmo não pára de chamar a si mesmo, até estourar a capacidade da pilha, o que geralmente causa efeitos colaterais e até mesmo o término indesejável do programa.

Para todo algoritmo recursivo existe um outro correspondente iterativo (não recursivo), que executa a mesma tarefa. Implementar um algoritmo recursivo, partindo de uma definição recursiva do problema, em uma linguagem de programação de alto nível como Pascal e C é simples e quase imediato, pois o seu código é praticamente transcrito para a sintaxe da linguagem. Por essa razão, em geral, os algoritmos recursivos possuem código mais claro (legível) e mais compacto do que os correspondentes iterativos. Além disso, muitas vezes, é evidente a natureza recursiva do problema a ser resolvido, como é o caso de problemas envolvendo árvores — estruturas de dados naturalmente recursivas. Entretanto, também há desvantagens: i) algoritmos recursivos quase sempre consomem mais recursos (especialmente memória, devido uso intensivo da pilha) do computador, logo tendem a apresentar um desempenho inferior aos iterativos; e ii) algoritmos recursivos são mais difíceis de serem depurados, especialmente quando for alta a profundidade de recursão (número máximo de chamadas simultâneas).

Exemplo 1: Função Fatorial (!)

Esta função é um dos exemplos clássicos de recursividade e, por isso, de citação quase obrigatória. Eis sua definição recursiva:

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

Observe no Quadro 01 a facilidade em transformar a definição da função para Portugol e C.

Quadro 01 - Implementações recursivas, em Portugol e C, da função fatorial

função Fat(n : natural) : natural
início
  se n = 0 **então**
    retorne 1
  senão
    retorne n * Fat(n - 1)
fim
	unsigned int Fat(**unsigned int** n)
	{
	  if (n == 0)
	    return 1;
	  else
	    return n * Fat(n - 1);
	}

pela definição, valor de 4! é calculado como:

4!=4*3!=4*(3*2!)=4*(3*(2*1!))=4*(3*(2*(1***0!**)))=4*(3*(2*(1***1**)))=24

Note que função é chamada recursivamente com argumento decrescente até chegar ao caso trivial (0!), cujo valor é 1. Este caso trivial (condição de parada) encerra a seqüência de chamadas recursivas. A seqüência de chamadas é melhor ilustrada abaixo:

4! = 4 * 3!
    3! = 3 * 2!
        2! = 2 * 1!
            1! = 1 * 0!
	O contexto de cada chamada é empilhado

0! = 1	Condição de parada

1! = 1
        2! = 2
    3! = 6
4! = 24
	O contexto de cada chamada é desempilhado

Como $n! = n \times (n-1) \times (n-2) \times \cdots \times 3 \times 2 \times 1$, é muito simples implementar um algoritmo iterativo da função fatorial. No Quadro 02, são apresentados dois algoritmos iterativos que se equivalem.

Quadro 02 - Implementações iterativas, em Portugol e C, da função fatorial

função Fat_NR(n : natural) : natural
declare
  i, x : natural
início
  x ← 1
  para i ← 2 **até** n **faça**
    x ← x * i
  retorne x
fim
	unsigned int Fat_NR(**unsigned int** n)
	{
	   unsigned int i, x;
	   x = 1;
	   for (i = 2; i <= n; i++)
	     x = x * i;
	   return x;
	}

Exemplo 2: Seqüência de Fibonacci

A seqüência [0, 1, 1, 2, 3, 5, 8, 13, 21, ...] é conhecida como seqüência ou série de Fibonacci e tem aplicações teóricas e práticas, na medida em que alguns padrões na natureza parecem segui-la. Pode ser obtida através da definição recursiva:

$$Fib(n) = \begin{cases} 0 & \text{se } n = 0 \\ 1 & \text{se } n = 1 \\ Fib(n-1) + Fib(n-2) & \text{se } n > 1 \end{cases},$$

a qual pode ser implementada como:

```
função Fib(n : natural) : natural
início
  se n =0 ou n = 1 então
    retorne n
  senão
    retorne Fib(n-2) + Fib(n-1)
fim
```

Note que, para $n > 1$, cada chamada causa 2 novas chamadas de Fib, isto é, o número total de chamadas cresce exponencialmente. Observe no diagrama de execução abaixo que para Fib(5), são feitas 14 chamadas da função. Além disso, Fib(0) e Fib(2) são chamadas 3 vezes; Fib(1) é chamada 5 vezes. Para Fib(25) são feitas 242784 chamadas recursivas!

```
Fib(5)
 ├─Fib(3)
 │  ├─Fib(1)
 │  └─Fib(2)
 │     ├─Fib(0)
 │     └─Fib(1)
 └─Fib(4)
    ├─Fib(2)
    │  ├─Fib(0)
    │  └─Fib(1)
    └─Fib(3)
       ├─Fib(1)
       └─Fib(2)
          ├─Fib(0)
          └─Fib(1)
```

Figura 01 - Diagrama de execução de Fib(5).

No caso da seqüência de Fibonacci é relativamente simples implementar um algoritmo iterativo com complexidade O(n), que tire proveito dos valores já calculados. Eis uma possibilidade:

```
função Fib_NR(n : natural) : natural
declare
  i, F1, F2, F : natural
início
  se n = 0 ou n = 1 então
    retorne 1
  senão
  início
     F1 ← 0
     F2 ← 1
     para i ← 1 até n - 1 faça
     início
       F ← F1 + F2
       F1 ← F2
       F2 ← F
     fim
     retorne F
  fim
fim
```

Exemplo 3: Problema da Torre de Hanói

O problema ou quebra-cabeça conhecido como torre de Hanói foi publicado em 1883 pelo matemático francês Edouard Lucas, também conhecido por seus estudos com a série Fibonacci. Consiste em transferir, com o menor número de movimentos, a torre composta por N discos do pino **A** (origem) para o pino **C** (destino), utilizando o pino **B** como auxiliar. Somente um disco pode ser movimentado de cada vez e um disco não pode ser colocado sobre outro disco de menor diâmetro.

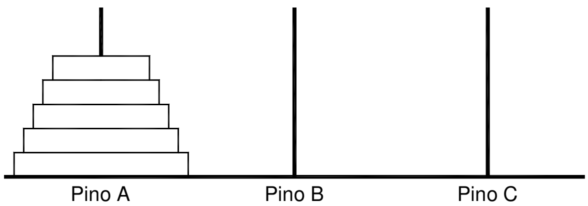

Figura 02 - Torre de Hanói.

Solução: Transferir a torre com N-1 discos de **A** para **B**, mover o maior disco de **A** para **C** e transferir a torre com N-1 de **B** para **C**. Embora não seja possível transferir a torre com N-1 de uma só vez, o problema torna-se mais simples: mover um disco e transferir duas torres com N-2 discos. Assim, cada transferência de torre implica em mover um disco e transferir de duas torres com um disco a menos e isso deve ser feito até que torre consista de um único disco.

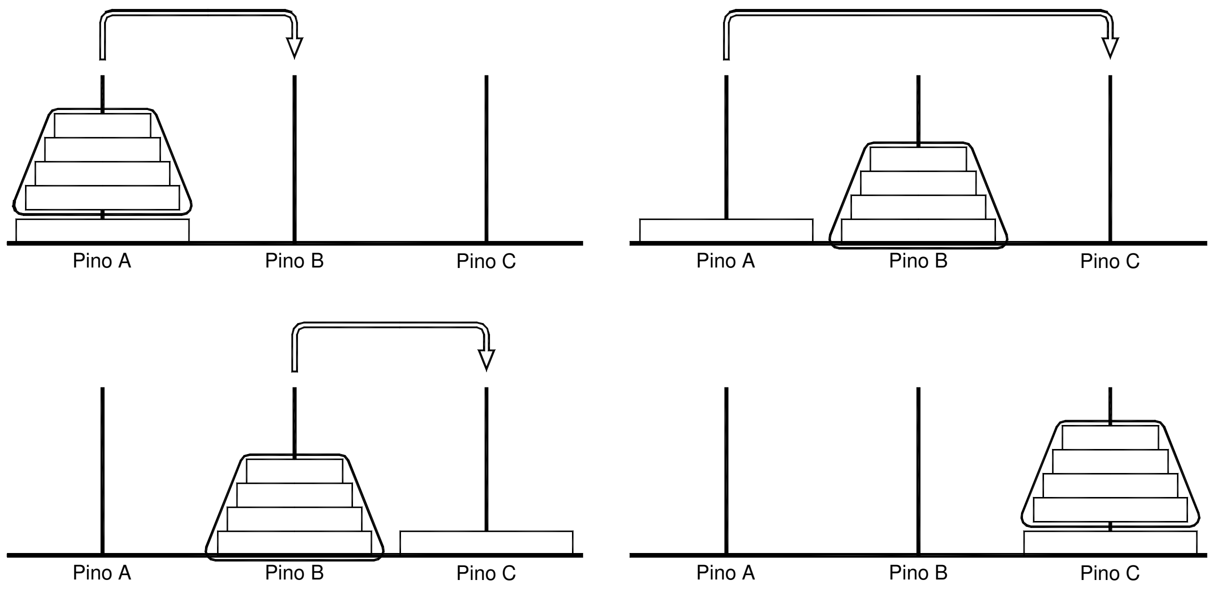

Figura 03 - Solução do problema da Torre de Hanói.

O algoritmo que soluciona o problema é surpreendentemente simples:

```
procedimento MoveTorre(N : natural; Orig, Dest, Aux : caracter)
início
  se N = 1 então
    MoveDisco(Orig, Dest)  senão
  início
    MoveTorre(N – 1, Orig, Aux, Dest)
    MoveDisco(Orig, Dest)
    MoveTorre(N – 1, Aux, Dest, Orig)
  fim
fim


procedimento MoveDisco(Orig, Dest : caracter)
início
  Escreva("Movimento: ", Orig, " -> ", Dest)
fim
```

Uma chamada a MoveTorre(3, ‘A’, ‘C’, ‘B’) produziria seguinte saída:

```
Movimento: A -> C
Movimento: A -> B
Movimento: C -> B
Movimento: A -> C
Movimento: B -> A
Movimento: B -> C
Movimento: A -> C
```

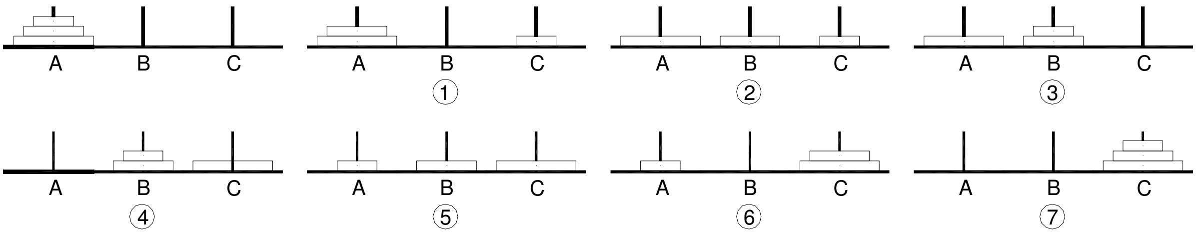

Figura 04 - Seqüência de movimentos para solucionar o problema da Torre de Hanói com 3 discos.

```
MT(3, A, C, B)
 ├─MT(2, A, B, C)
 │   ├─MT(1, A, C, B)
 │   │   └─MD(A, C)................ A -> C
 │   ├─MD(A, B).................... A -> B
 │   └─MT(1, C, B, A)
 │       └─MD(C, B)................ C -> B
 ├─MD(A, C)........................ A -> C
 └─MT(2, B, C, A)
     ├─MT(1, B, A, C)
     │   └─MD(B, A)................ B -> A
     ├─MD(B, C).................... B -> C
     └─MT(1, A, C, B)
         └─MD(A, C)................ A -> C
```

Figura 05 - Diagrama de execução da chamada MoveTorre(3, ‘A’, ‘C’, ‘B’)

Observe que o algoritmo MoveTorre faz duas chamadas a si mesmo, portanto o número de movimentos cresce exponencialmente com o número de discos. Mais precisamente, a solução do problema com N discos requer 2^N-1 movimentos. Assim, se uma pessoa se dispusesse a resolver o problema com 25 discos, com dedicação exclusiva de 8 horas por dia e realizando um movimento por segundo, esta “pobre criatura” gastaria 3354431 segundos, ou seja, mais de 3 anos para solucionar o problema. Esse tempo dobraria a cada disco acrescentado!

RESUMO

Conceitos

- Um algoritmo recursivo deve fazer pelo menos uma chamada a si mesmo, de forma direta (podemos ver o algoritmo sendo chamado dentro dele mesmo) ou indireta (o algoritmo chama um outro algoritmo, que por sua vez invoca uma chamada ao primeiro).
- Um algoritmo recursivo deve ter pelo menos uma condição de parada, para que não seja invocado indefinidamente. Esta condição de parada corresponde a instâncias suficientemente pequenas ou simples do problema original, que podem ser resolvidas diretamente.
- Para todo algoritmo recursivo existe pelo menos um algoritmo iterativo correspondente e vice-versa. Todavia, muitas vezes pode ser difícil encontrar essa correspondência.

Vantagens

Os algoritmos recursivos normalmente são mais compactos, mais legíveis e mais fáceis de serem compreendidos. Algoritmos para resolver problemas de natureza recursiva são fáceis de serem implementados em linguagens de programação de alto nível.

Desvantagens

Por usarem intensivamente a pilha, o que requer alocações e desalocações de memória, os algoritmos recursivos tendem a ser mais lentos que os equivalentes iterativos, porém pode valer a pena sacrificar a eficiência em benefício da clareza. Algoritmos recursivos são mais difíceis de serem depurados durante a fase de desenvolvimento.

Aplicações

- Nem sempre a natureza recursiva do problema garante que um algoritmo recursivo seja a melhor opção para resolvê-lo. O algoritmo recursivo para obter a seqüência de Fibonacci é um ótimo exemplo disso.
- Algoritmos recursivos são aplicados em diversas situações como em: i) problemas envolvendo manipulações de árvores; ii) analisadores léxicos recursivos de compiladores; e iii) problemas que envolvem tentativa e erro ("Backtracking").

Exercícios

1) Considere a função $\boldsymbol{Comb(n, k)}$, que representa o número de grupos distintos com $\boldsymbol{k}$ pessoas que podem ser formados a partir de $\boldsymbol{n}$ pessoas. Por exemplo, $Comb(4, 3) = 4$, pois com 4 pessoas (A, B, C, D), é possível formar 4 diferentes grupos: ABC, ABD, ACD e BCD. Sabe-se que:

$$Comb(n,k) = \begin{cases} n & \text{se} \quad k = 1 \\ 1 & \text{se} \quad k = n \\ Comb(n-1,k-1) + Comb(n-1,k) & \text{se} \quad 1 < k < n \end{cases}$$

Implemente em portugol uma função recursiva para $Comb\ (n, k)$ e mostre o diagrama de execução para chamada $Comb\ (5, 3)$. Sabendo-se ainda $Comb\ (n, k) = n!\ /\ (k!\ (n\text{-}k)!)$, implemente uma função não recursiva de $Comb\ (n, k)$.

2) Implemente recursivamente uma função Max que retorne o maior valor armazenado em um vetor V, contendo n números inteiros.

3) Dada a implementação em portugol da função abaixo:

```
função F(N : natural) : natural
início
  se N < 4 então
    retorne 3 N
   senão
    retorne 2 * F(N - 4) + 5
fim
```

Quais são os valores de F(3) e de F(7)?

4) O cálculo da raiz quadrada de um número real x pode ser feito através do seguinte algoritmo:

$$RaizQ(x, x_0, \varepsilon) = \begin{cases} x_0 & se \mid x_0^2 - x \mid \leq \varepsilon \\ RaizQ(x, \dfrac{x_0^2 + x}{2\, x_0}, \varepsilon) & \text{caso contrário} \end{cases}$$

em que x_0 é uma aproximação inicial do valor $\sqrt{x}$ e ε é um erro admissível. Implemente o algoritmo em Portugol e mostre o diagrama de execução para a chamada $RaizQ(13, 3.2, 0.001)$.

5) Dada a definição da função de Ackerman:

$$A(m,n) = \begin{cases} n+1 & \text{se } m = 0 \\ A(n-1, 1) & \text{se } m > 0 \text{ e } n = 0 \\ A(m-1, A(m, n-1)) & \text{se } m > 0 \text{ e } n > 0 \end{cases}$$

válida para valores inteiros não negativos de m e n, implemente uma versão recursiva do algoritmo e faça o diagrama de execução de $A(1, 2)$.

6) A função $f(x, n) = x^n$, em que x é um número real e n um número inteiro, pode ser calculada eficientemente como nos exemplos abaixo:
$x^0 = 1$; $x^1 = x$; $x^2 = x^2$; $x^3 = x\,x^2$; $x^4 = (x^2)^2$ $x^5 = x\,(x^2)^2$; $x^6 = (x\,x^2)^2$; $x^{11} = x\,((x\,(x^2)^2)^2$; $x^{-2} = 1/x^2$ etc.
Elabore a definição recursiva de $f(x, n)$ e implemente um algoritmo recursivo para $f(x,n)$.

7) A recursividade pode ser utilizada para gerar todas as possíveis permutações de um conjunto de símbolos. Por exemplo, existem seis permutações no conjunto de símbolos A, B e C: ABC, ACB, BAC, BCA, CBA e CAB. O conjunto de permutações de N símbolos é gerado tomando-se cada símbolo por vez e prefixando-o a todas as permutações que resultam dos $(N - 1)$ símbolos restantes. Conseqüentemente, permutações num conjunto de símbolos podem ser especificadas em termos de permutações num conjunto menor de símbolos. Formule um algoritmo recursivo para este problema.

8) Considere uma partida de futebol entre duas equipes A x B, cujo placar final é m x n, em que m e n são os números de gols marcados por A e B, respectivamente. Implemente um algoritmo recursivo que imprima todas as possíveis sucessões de gols marcados. Por exemplo, para um resultado de 3 x 1 as possíveis sucessões de gols são "A A A B", "A A B A", "A B A A" e "B A A A".