RECURSIVIDADE

Existe quando algo é definido em termos de si próprio.

Exemplos:

a. Conjunto dos números naturais
   1 é nro. natural;
   o sucessor de um nro. natural é um nro. natural.
b. Árvores Binárias (AB)
   uma AB é um conjunto vazio de nós ou um conjunto de um nó raiz e uma sub-árvore binária à esquerda e uma sub-árvore binária à direita.
c. Fatorial
   0! = 1
   n! = n * (n-1)! ; n>0
d. Sequência de Fibonacci
   Fib(0) = 0;
   Fib(1) = 1;
   Fib(n) = Fib(n-1) + Fib(n-2); n>1

<u>Consequência:</u> Definição finita de um conjunto infinito de objetos.

Portanto, um programa recursivo finito pode descrever um número infinito de computações, mesmo sem possuir repetições explícitas.
Programa recursivo P º C [S_i, P] onde S_i são comandos independentes de P.
Requisito sobre a Linguagem de Programação: existência de procedimentos.

Um procedimento pode ser:

a. <u>diretamente recursivo</u>: P ativa P;
b. <u>indiretamente recursivo:</u> P ativa Q ativa R....ativa P

Importante: a cada vez que um procedimento é ativado, um novo conjunto de parâmetros (passados por valor) e variáveis locais é criado, ou seja, é alocada memória para eles.

<u>Término de Procedimentos Recursivos</u>

Recursividade possibilita loops infinitos.
Requisito para evitar isso: sujeitar a chamada recursiva a uma condição B que, em algum instante, não será satisfeita:
P º if B then C [S_i, P] ou P º C [S_i, if B then P]
Em geral, um dos parâmetros de P representa o "tamanho" do problema para aquela chamada, e a parada se dá num limite inferior desse parâmetro: P(n) º if n > 0 then C [S_i, P(n-1)]
ou P(n) º C [S_i, if n > 0 then P(n-1)]

<u>Restrições Práticas:</u>

a. Nível máximo finito (número finito de chamadas)
b. Nível máximo pequeno

Quando não usar Recursividade

A natureza recursiva do problema ou da E.D. não garante que um algoritmo recursivo seja a melhor solução. Todo algoritmo recursivo pode ser transformado num algoritmo não recursivo que, apesar de mais complexo e menos claro, muitas vezes é mais eficiente em relação à espaço e tempo.
Programas cujos esquemas sejam do tipo:
P º if B then (S_i, P) ou P º (S_i, if B then P)

ou seja, quando a chamada recursiva é o último comando do procedimento, são mais bem expressos como
P º (while B do S) e P º (repeat S until not B) respectivamente.

Considere o exemplo da Sequência de Fibonacci.
Com base em sua definição recursiva, somos levados a construir o seguinte procedimento recursivo:

```
function Fib(n:posint) : posint; /* posint = o..maxint */
begin
case n of
0: Fib := 0;
1: Fib := 1;
else Fib := Fib(n-1) + Fib(n-2)
end
end;
```

Note que, para n > 1, cada chamada causa 2 novas chamadas de Fib, i.é, <u>o número total de chamadas cresce exponencialmente.</u>
Verifique que, para Fib(5), são feitas 14 chamadas da função. Além disso, Fib(i); i=0, 1, 2 e 3 são chamadas mais de uma vez. Fib(0) é chamada (e calculada) 3 vezes; Fib(1), 5 vezes; Fib(2), 3 vezes; Fib(3), 2 vezes.
Sem dúvida, esse programa é inviável!
No entanto, com o simples uso de 2 variáveis auxiliares, construímos um esquema iterativo que calcula o n-ésimo número de Fibonacci sem recomputar valores já calculados:

```
begin /* para n > 0 */
i := 1; fib := 1; y := 0;
while i < n do
begin
i := i+1;
fib := fib + y;
y := fib - y
end
end /* fib = Fib(n)*/
```

Conclusão: Evitar recursão quando existe uma solução <u>óbvia</u> via iteração.