Recursividade

1. Definição

Recursividade é propriedade que permite a uma função chamar a si mesma. Isso é muito útil em certos tipos de algoritmos que podem ser simplificados quando são divididos em outros subproblemas menores do mesmo tipo.

Para explicar melhor o funcionamento da recursividade, usaremos alguns exemplos de implementação.

2. Fatorial

Um exemplo comum usando recursão é a função para calcular o fatorial de um natural N, representado por N!.

Matematicamente, o fatorial de um número natural N é o produto de todos os números naturais de 1 até N. Por definição, o fatorial de 0 é igual a 1.

Assim, temos a sequência:

0! = 1;
1! = 1;
2! = 1*2 = 2;
3! = 1*2*3 = 6;
4! = 1*2*3*4 = 24;
5! = 1*2*3*4*5 = 120;
N! = 1*2*3* … *(N-1) * (N).

Um algoritmo iterativo para o cálculo do fatorial pode ser implementado em C como abaixo:

```
int fat(int n)
{
   int i, f;

   f = 1;
   for (i=1; i<=n; i++)
   {
      f = f * i;
   }
   return (f);
}
```

Mas podemos implementar uma função recursiva para o cálculo do fatorial.

Em geral, uma definição recursiva é definida por casos: um número limitado de casos base e um caso recursivo. Os casos base são geralmente situações triviais e não envolvem recursão.

Para o fatorial, o caso base é o valor de 0!, que é 1. No caso recursivo, dado um N > 0, o valor de N! é calculado multiplicando por N o valor de (N-1)!, e assim por diante, de tal forma que N! tem como valor N * (N-1) * (N-2) * ... * (N-N)!, onde (N-N)! representa obviamente o caso base. E a função recursiva pode ser implementada em C do modo abaixo:

```
int fat(int n)
{
   if (n == 0)
      return (1);
   else
      return (n*fat(n-1));
}
```

Note que no último comando, a função **fat** chama ela mesma, para calcular o valor do fatorial de `n-1`. Essa é a chamada recursiva.

Observe que na versão recursiva da função não há um comando específico para o laço. **A repetição está implícita na chamada recursiva**. Assim como todo laço precisa de uma condição de parada, para evitar o laço infinito, em função recursiva, **a condição de parada em geral é um if que não chama a recursividade**. Nesse caso, se `n` for igual a `0`, a função retorna pelo valor do caso base, sem chamar novamente ela mesma.

Outro aspecto importante a observar. **Cada chamada recursiva precisa passar o parâmetro modificado para a função**. Caso seja executada uma chamada recursiva sem alterar o parâmetro, a próxima chamada vai executar exatamente a mesma situação, e isso leva a um looping infinito. Nesse caso, a chamada recursiva está passando o parâmetro `n-1`, e não `n`.

3. Sequência de Fibonacci

A sequência de Fibonacci é uma sequência infinita de números naturais, que aproxima alguns fenômenos da natureza, incluindo crescimento populacional de microorganismos.

Os dois primeiros números da sequência são 1 e 1. A partir do terceiro elemento, cada um deles é a soma dos dois anteriores. O terceiro elemento é a soma de 1+1=2. O quarto é , 1+2=3. O quinto é 2+3=5, e assim por diante, formando a sequência:

1, 1, 2, 3, 5, 8, 13, 21, …

Podemos implementar a uma função que retorna o n-ésimo termo da sequência de Fibonacci usando um algoritmo iterativo ou recursivo. Uma possível implementação iterativa é:

```
int fib(int n)
{
    int i = 0;
    int j = 1;
    int k;
    int aux;
    for(k = 1; k < n; k++){
        aux = i+j;
        i = j;
        j = aux;
    }
    return j;
}
```

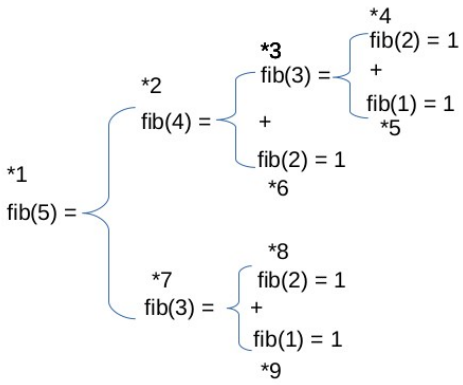
Nesse caso as variáveis i e j se referem aos termos anteriores, aux é uma variável auxiliar, para atualizar os valores de i e j para a próxima iteração, e a variável k é usada para indicar a posição do termo na sequência.

A versão recursiva do código é

```
long fib(int n)
{
    if (n <= 2)
       return (1);
    else
       return (fib(n-1) + fib(n-2));
}
```

Observe novamente a existência da condição de parada. Se `n` é menor ou igual a 2 (para o primeiro ou segundo termo da série), o número é 1. Caso contrário (do terceiro termo em diante), o número é a soma dos dois anteriores (`fib(n-1)` e `fib(n-2)`). Nesse caso, são duas chamadas recursivas, uma para cada um dos dois números anteriores da série. Para esse algoritmo, na versão recursiva, cada chamada vai desencadear outras duas chamadas, o que torna o processamento mais lento do que a versão iterativa.

Exemplo, para resolver `fib(5)`, a função recursiva chama a si mesma outras 8 vezes, totalizando 9 chamadas da função fib, ilustradas na figura abaixo. O número indicado com o asterisco é a ordem em que elas são executadas pelo algoritmo acima.



O algoritmo recursivo fica mais elegante, mas nesse caso específico, mais pesado.

4. Impressão de Lista Encadeada

Outro algoritmo que podemos facilmente desenvolver usando recursividade é a impressão de uma lista encadeada. A versão iterativa, já vista anteriormente é:

```
// Imprime todos os nodos da lista l
void imprimeLista(Lista l)
{
     Lista p;       // Ponteiro auxiliar, para percorrer a lista

     p = l;         // Fazer p apontar para o início da lista
     while (p != NULL)   // Enquanto não chegar ao final da lista
     {
          printf("%d - ",p->dado); // Imprime o elemento
          p = p->prox;   // Avança o ponteiro para o próximo nodo
     }
     printf("\n");  // Avança para a próxima linha
```

```
}
```

Basicamente esse algoritmo define um ponteiro para o primeiro elemento da lista, e usa um laço (while) para percorrer a lista, imprimindo todos os elementos, enquanto não chegar ao final da lista.

Uma versão recursiva deste algoritmo pode ser pensada, com os seguintes passos:

Se a lista `l` não estiver vazia, imprime `l→dado` e chama `imprime_lista(l→prox)`

Em C, a função pode ser implementada da sequinte forma:

```
// Imprime todos os nodos da lista l
void imprimeLista(Lista l)
{
     if (l != NULL)
     {                              // Se a lista não está vazia
          printf("%d - ",l->dado); // Imprime o primeiro elemento
          imprimeLista(l->prox);   // Imprime o restante da lista
     }
}
```

Observe que o código fica novamente mais compacto e elegante. O laço de repetição está implícito na chamada recursiva, e a condição de parada é `if (l != NULL)`, ou seja, ele continua imprimindo o primeiro nodo e chamando novamente somente se a lista não está vazia.

Respeitamos assim a existência da condição de parada, e a regra de que as chamadas recursivas devem usar parâmetros diferente do que recebeu. Nesse caso, o ponteiro para o próximo elemento.

Embora a versão recursiva dos códigos trabalhados até agora seja mais elegante, não trouxe ganho de desempenho nem de funcionalidade.

Mas nesse caso, de listar o conteúdo de uma lista, o código recursivo só consegue imprimir a lista na mesma ordem dos ponteiros. Se for necessário imprimir na ordem inversa, o método iterativo precisará dispor de uma pilha, para armazenar os valores (ou os endereços dos nosos), para depois retirar da pilha e imprimir, dado que a retirada da pilha é sempre na ordem inversa da inserção.

Entretanto, alterar a versão recursiva para imprimir na ordem inversa é muito simples, bastando inverter a ordem dos comandos de impressão e de chamada recursiva.

```
// Imprime todos os nodos da lista l em ordem inversa
void imprimeListaInv(Lista l)
{
   if (l != NULL)
   {                              // Se a lista não está vazia
      imprimeListaInv(l->prox);   // Imprime o restante da lista
      printf("%d - ",l->dado);    // Imprime o primeiro elemento
   }
}
```