1. Recursividade

A recursividade é uma técnica de programação através da qual uma função chama-se a si própria, uma ou várias vezes. As funções recursivas representam um mecanismo alternativo aos ciclos para programar a repetição de acções. Existe alguma correspondência entre uma iteração num ciclo e uma chamada duma função recursiva. Da mesma forma que num ciclo temos a preocupação de o controlar correctamente, nomeadamente para garantir que este eventualmente termina (senão, "entra em ciclo infinito"), também temos uma preocupação semelhante com as funções recursivas, para não entrarem em “recursão infinita”.

O clássico exemplo da recursividade é a função factorial, que é definida da seguinte maneira:

N! = N * (N-1)! (para N >= 0)

O factorial pode ser implementado com a seguinte função recursiva:

```
static int factorial(int numero) {
   if(numero == 0) {
      return 1;
   } else {
      return numero * factorial(numero - 1);
   }
}
```

Caso base e passo indutivo

Para se conceber e entender uma função recursiva, é fundamental conhecer os componentes que encontramos em praticamente todos os algoritmos recursivos.

O raciocínio que devemos ter é partir um problema em subproblemas mais simples, de modo que cada subproblema tem uma resolução simples. Qualquer algoritmo recursivo possui dois componentes fundamentais: o **caso base** e o **passo indutivo**.

O caso base é a parte que permite controlar o número de vezes que a função executa de modo a que certa altura a função não volte a chamar-se a si própria. Consiste num teste (ou vários, mas **sempre** pelo menos um) que é feito ao argumento da função recursiva, e cujo resultado correspondente é conhecido à partida. No exemplo do factorial, o caso base é o factorial de zero, que por definição é um.

Por vezes a função recursiva não tem argumentos, mas nesse caso existe algo (tem sempre de existir) que é usado de forma semelhante. Por exemplo, um valor lido do teclado, ou dum ficheiro.

O outro componente fundamental, o salto indutivo, consiste em fornecer o resultado pretendido para a função em termos da solução dum "problema menor" (embora da mesma natureza). No exemplo do factorial é o valor de (N-1)! Considera-se que calcular (N-1)! representa um "problema menor" do que calcular N! O salto indutivo consiste em partir do princípio que já possuímos a solução de (N-1)! e construir a solução de N! a partir da de (N-1)! A própria definição do factorial sugere como isso é feito: N * (N-1)!

Um aspecto importante é o facto do "problema menor" encontrar-se "mais próximo" do caso base do que o "problema maior". Sucessivas chamadas da função factorial em que lhe passamos a expressão N-1 vão "aproximando" o valor do argumento do caso base (0, no caso do factorial). Isto é fundamental para que a função não entre em recursão infinita.

Recursividade vs. ciclos iterativos

A recursividade representa um modelo de programação bastante diferente da programação imperativa, que se baseia no conceito de estado sobre o qual se vai iterando. Nas funções recursivas não existe o conceito de estado. É por isso que algumas pessoas que aprenderam a programar com ciclos – uma construção tipicamente imperativa – sentem alguma dificuldade

inicial em apreender o raciocínio subjacente na recursividade. É de facto uma maneira diferente de raciocinar. Quando vêm a versão recursiva do factorial, muitos estudantes tentam "entrar" na chamada chamada recursiva (para N-1), tentando simular mentalmente o funcionamento da função para a chamada seguinte. Não devem fazer isso: basta partir do princípio que a função funciona bem e devolve o resultado correcto.

A concepção duma solução recursiva para determinada tarefa tem o seguinte raciocínio:

- identificamos os casos bases do problema. Temde haver sempre pelo menos um. Dos casos base derivam-se as partes da função que NÃO voltam a chamar a função.
- dividimos a tarefa numa "versão maior" e numa "versão menor".
- partimos do princípio que já possuímos a solução para a "versão menor", que corresponde à(s) chamada(s) recursiva(s).
- pensamos como obter a solução para a "versão maior" a partir da "versão menor". Isso é o salto indutivo.

As função recursivas costumam ser simples. Tão simples que muitos estudantes não acreditam que possa ser assim tão simples. Se o estudante estiver a escrever uma função recursiva com muitas linhas de código, isso é provavelmente sinal de que não está a concebê-la bem. Tipicamente está a tentar meter mais do que uma "chamada" na função. Um exemplo, aplicado ao factorial, é tentar incluir na função o cálculo de 1!, para além de 0!

Exemplos de algoritmos recursivos

Exemplo 1: cálculo de números fibonacci

Tarefa: calcular o número da sequência fibonacci com base na sua posição na sequência.

Casos base: os dois primeiros números da sequência são conhecidos à partida. Assim, Fib(0) = 0 e Fib(1) = 1.

Salto indutivo: dado uma posição (índice) n na sequência, o número é a soma de Fib(n-1) com Fib(n-2).

Dos casos base e salto indutivo apresentados acima, deduz-se o seguinte método Java:

```
static int fibonacci(int posicao) {
   if(posicao == 0) {
      return 0;
   }
   if(posicao == 1) {
      return 1;
   }
   return fibonacci(posicao-1) + fibonacci(posicao-2);
}

public static void main(String[] args) {
   for(int i=0; i<45; i++) {
      System.out.println(i + ": " + fibonacci(i));
   }
   System.out.println("Terminado.");
}
```

Notem que o tempo execução de Fib(n) é quase o dobro do tempo execução de Fib(n-1). Ao correr o programa fica óbvio, mesmo em máquinas actuais o enorme peso computacional que acarretam o cálculo dos últimos números, que podem levar vários minutos a executar (foi por isso que foi incluído o println final, já que se poderia pensar que o processo entrou em *crash*).

Exemplo 2: somatório de números lidos do teclado

Tarefa: calcular a soma todos os valores numéricos introduzidos pelo utilizador até ser introduzido um zero.

Caso base: nós sabemos à partida que quando o utilizador introduz o zero o valor que a função deve devolver é zero. A introdução do zero representa o caso base.

Salto indutivo: se após a leitura dum valor partirmos do princípio que a soma de todos os restantes (os que ainda não foram lidos) é correctamente calculada pela chamada seguinte à função, temos as versões "maior" e "menor" do problema:

- a versão maior é a soma de todos os valores, incluindo o que acabou de ser lido.
- a versão menor é a soma de todos os valores, excepto o que acabou de ser lido.

Se a chamada recursiva representar a solução para a versão menor, construímos a solução para a versão maior somando o valor que acabou de ser lido com a solução da versão menor.

O raciocínio apresentado permite deduzir a seguinte função Kenya:

```
static int somaNumerosDoTeclado(Scanner scanner) {
   System.out.print("Proximo numero: ");
   int numero = scanner.nextInt();
   if (numero == 0) {
      return 0;
   } else {
      return numero + somaNumerosDoTeclado(scanner);
   }
}
public static void main(String[] args) {
   System.out.println(
      "Pf introduza uma sequencia de numeros, terminada com 0.");
   System.out.println("Somatorio dos numeros lidos: " +
      somaNumerosDoTeclado());
}
```

Conceitos de cabeça + cauda

No processamento recursivo de vectores recorre-se ao conceito de **cabeça-cauda**. Qualquer sequência de valores (mesmo uma sequência de valores lidos do teclado) pode ser encarada de acordo com a seguinte definição recursiva:

- Uma sequência é constituída por uma cabeça seguida duma cauda.
- A cauda também é uma sequência.
- A sequência vazia (i.e., com zero elementos) também é uma sequência.

A sequência vazia representa o caso base. Desempenhar a tarefa para a cauda representa uma "versão menor" da tarefa para a lista toda. A maneira de resolver o problema para a lista toda a partir da cabeça a da solução obtida para a cauda (que, repito, devemos partir do princípio que é fornecida pela mera chamada da mesma função, passando-lhe a cauda como argumento) representa o passo indutivo.

Esta abordagem pode ser aplicada para implementar um grande conjunto de tarefas, nomeadamente os que envolvem vectores e sequências de valores lidos do teclado (ou de qualquer outra fonte). Por exemplo, o primeiro elemento dum vector é a cabeça; a parte restante do vector é a cauda. O salto indutivo consiste em considerar que já se possui a solução para a parte restante do vector, e descobrir a solução para o vector todo a partir da cabeça do vector e da solução para a cauda do vector.

Exemplo 3: calcular o menor elemento dum vector de inteiros.

A seguir apresenta-se uma função recursiva que recebe um vector e um índice e devolve o valor do menor elemento do vector:

```
static int lower(int[] vector, int i) {
   if(i < vector.length -1) {
      int menorRestante = lower(vector, i+1);
```

```
      if(vector[i] < menorRestante) {
        return vector[i];
      } else {
        return menorRestante;
      }
   } else {
      return vector[vector.length - 1];
   }
}
```

O mecanismo da recursividade obriga-nos a passar um argumento adicional, para além do próprio vector. É assim porque cada instância da função, em cada chamada, não possui informação de contexto, contrariamente ao que acontece com os ciclos. Isso é inconveniente, mas podemos "embrulhar" a função numa outra que nos livra de passarmos o índice do primeiro elemento (zero):

```
int menorElemento(int[] vector) {
   return lower(vectorDeTeste(), 0);
}
```

A necessidade de passar o segundo argumento parece uma desvantagem relativamente a uma função iterativa (i.e., que usa ciclos), mas talvez seja mais justo encarar a função recursiva como um mecanismo para obter repetições, alternativo **ao ciclo**. Nesse caso, a comparação que se deve fazer é entre a função recursiva e o ciclo, não com toda a função iterativa completa. Afinal, as funções iterativas também incluem frequentemente instruções antes e depois do ciclo.

Call stack e *activation records*

Existe uma pilha de chamadas associada ao funcionamento de funções e métodos, frequentemente chamada de *call stack*. Cada vez que uma função é chamada, é efectuado o *push* na *call stack* duma espécie de registo contendo dados associados à chamada da função. Esse registo é chamado *activation record* ou, por vezes, *stack frame*. O *activation record* contém coisas como os argumentos passados à função (ou método), as variáveis locais da função, o endereço do *activation record* anterior (de modo a que o sistema saiba que instruções devem ser executadas a seguir ao retorno da função). Duas chamadas da mesma função correspondem a *activation records* distintos, e é por isso que a recursividade funciona. Cada vez que uma função termina a sua execução, é efectuado um *pop* da *call stack*.

Porque a recursividade é ineficiente

Comparada com os processos iterativos, a recursividade é ineficiente quer no tempo de processamento, quer no consumo de memória.

A recursividade é ineficiente no tempo de execução porque faz corresponder, para cada iteração do processo iterativo, a chamada duma função. A chamada a uma função é computacionalmente mais pesada do que os saltos de endereço de memória, que são usados para implementar ciclos. A função recursiva para o problema dos números fibonacci permite ilustrar isso com muita clareza.

A recursividade é ineficiente no consumo de memória relativamente a processos iterativos porque as iterações geralmente trabalham sempre com as mesmas variáveis, não gastando por isso mais memória ao longo do processo. As funções recursivas, porém, gastam memória para um *activation record* por cada repetição. Esses *activation records* vão-se acumulando até que uma instância da função atinja um caso base. Só então é que começam a ser efectuados os *pops* da *call stack*, libertando memória. Num problema que envolva um número muito elevado de chamadas, há o risco da memória disponível (para o programa que está a correr) se esgotar, fazendo o processo abortar.