# Tutorial 2

Recursão

Ciências ULisboa

Unidade Curricular de
Laboratório de Programação

2020/2021

# O que é a recursão

Numa frase ilustrativa será:

***Definição de Recursão:***

***Se já entendeu a definição, pare.***
***Senão releia outra vez a "Definição de Recursão"!***

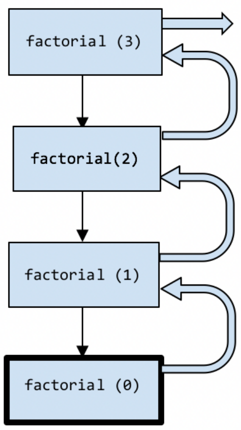
Em qualquer linguagem de programação e em particular em Java, um método pode invocar-se a ele próprio. Nesse caso, diz-se que o método é *recursivo*. Este conceito não deverá ser novo pois já foi abordado nas aulas de AED.

# Exemplo

A título de exemplo, analisamos uma forma recursiva de calcular o fatorial de um número inteiro positivo. Recorde que `0! = 1` e que `n! = n x (n-1) x … x 1= n x (n-1)!` para `n > 0`.

```
/**
 * The factorial of a given number
 * @param n The number
 * @return the factorial of n (n * (n-1) * ... * 1)
 */
public static long factorial(int n) {
      long result;
      if (n == 0){
            result = 1;
      } else {
            result =   n * factorial(n-1);
      }
      return result;
 }
```

Repare que, para um dado `n > 0`, a função só termina quando a invocação feita sobre `n-1` termina, devolvendo um resultado. Por isso, a execução de `factorial(3)` só terminará depois de a invocação `factorial(2)` terminar e devolver um resultado, que será multiplicado por 3, concluindo o cálculo e terminando a execução da invocação de `factorial(3)`.




A invocação de `factorial(2)`, por sua vez, só termina depois de `factorial(1)` terminar, a qual, por sua vez, só termina depois de `factorial(0)` terminar. Esta última invocação já não provoca mais nenhuma invocação recursiva e retorna logo o valor 1. Este valor é depois multiplicado por 1 para obter o resultado de `factorial(1)`. De seguida este novo valor 1 é multiplicado por 2 para obter o resultado de `factorial(2)`. Finalmente, este valor 2 é multiplicado por 3, terminando a execução de `factorial(3)` com o retorno de 6.

Podemos estudar um método recursivo através da sua *árvore de recursão*, a qual representa todas as invocações recursivas do método em questão. Ao lado a árvore de recursão para a invocação `factorial(3)` (o rebordo do caso base a *bold*).

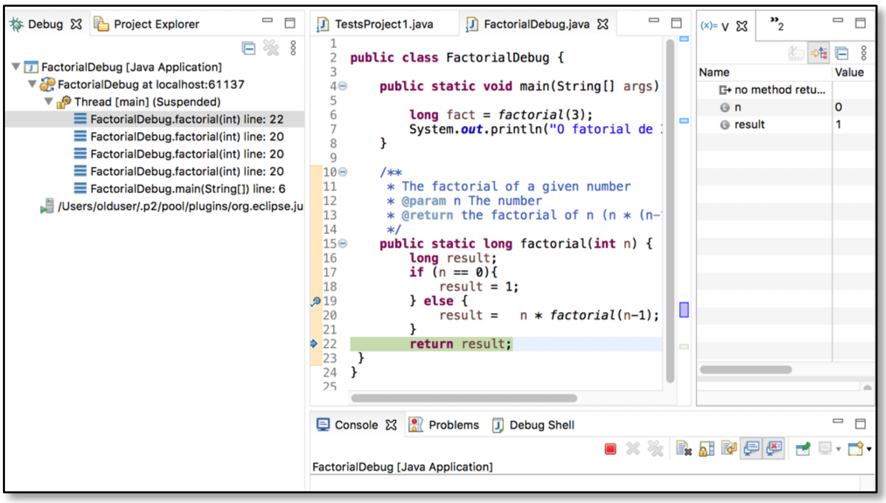
O *número de elementos* da árvore influencia o tempo de execução (cada elemento da árvore corresponde a uma invocação). A *altura máxima da árvore* determina os requisitos de memória (é igual ao número máximo de invocações à espera de resultado).

Podemos usar a perspetiva de Debug do Eclipse para ver a sequência de invocações resultante da execução de uma função recursiva.

Defina no eclipse uma classe `FactorialDebug`, onde inclui a declaração da função recursiva `factorial` dada acima e, no método `main`, a invocação `factorial(3)` seguida da impressão do resultado.

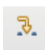
Experimente agora inserir um *break point* na linha `return result` da função.

De seguida execute o programa em modo *debug*, usando **Step Into**  (ou **F5**) para execução passo a passo.




Na subjanela *Debug* verá o traço da execução até atingir (pela 1ª vez) esse *break point*. Esse traço mostra as 4 invocações da função fatorial e pára, na 4ª chamada, imediatamente antes do *break point*, ou seja, antes de retornar da chamada `factorial(0)`. Pode ver na subjanela *Variables* os valores do parâmetro `n` e da variável `result`.

Note que a última invocação de um método é sempre a que aparece no topo desse traço. Se clicar no símbolo ≡ que precede uma das outras invocações, poderá ver na subjanela *Variables* o valor das variáveis locais a essa invocação (neste caso, apenas o valor do parâmetro `n`).

Se continuar a clicar em  verá o resultado de cada invocação a ser calculado usando o retorno da invocação terminada imediatamente antes.

# Quando se usa Recursão?

A recursão é útil na resolução de problemas que podem ser decompostos em subproblemas mais simples (do mesmo tipo que o anterior) e onde a solução final se obtém por composição das soluções desses subproblemas como se fez no exemplo anterior.

Para aplicar soluções recursivas é necessário que se verifiquem três condições:

1. Existir um conjunto de um ou mais casos triviais que não precisam de nenhuma chamada recursiva suplementar – denominados **caso base** ou **base da recursão**,
2. Ser possível decompor o problema em subproblemas mais simples que permitem construir a solução final – denomina-se esta decomposição por **passo da recursão** e
3. A decomposição sucessiva dos subproblemas levará inevitavelmente ao caso base.

Assim, a estrutura de uma solução recursiva de um problema `P` é:

```
problema(P):
      se o caso base B responde a P
            devolver Resposta ao caso Base
      senão
            decompor P em P1, ..., Pn
            R1 = problema(P1)
            …
            Rn = problema(Pn)
            R = resposta construída com R1, ..., Rn
            devolver R
```

Esta técnica de decomposição de um problema em subproblemas relacionados também se designa por ***dividir para conquistar*** (do inglês, *divide and conquer*).

É crucial que os subproblemas recebam a informação necessária para a resolução e devolvam informação suficiente para a construção da resposta global.

Se a decomposição dos subproblemas nunca chegar ao caso base, a **recursão não tem forma de parar**. É uma recursão infinita e o programa terminará eventualmente por falta de recursos de memória.

O exemplo seguinte ilustra um caso de recursão infinita, dado que não há caso base que permita o fim natural da computação.

```java
public static long factorial(int n) {
        return n * factorial(n-1);
}
```

Experimente usar o debugger sobre esta nova versão da função `factorial` e observe o traço para a execução de `factorial(3)`.

# Recursão não-linear

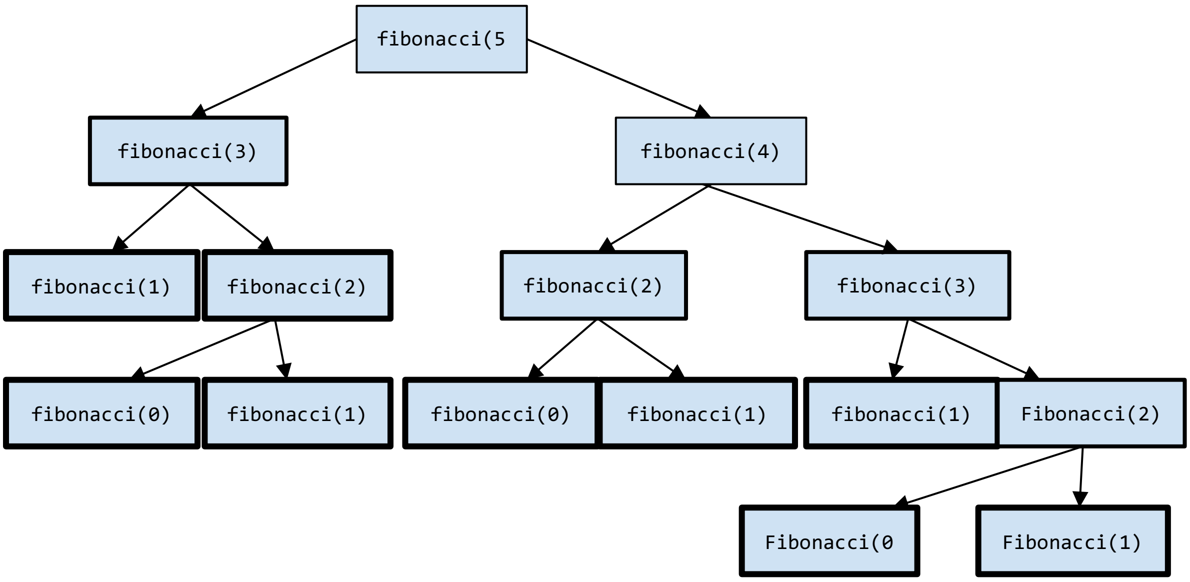
Consideremos agora um outro exemplo de um método que permite calcular o n-ésimo termo da sequência de Fibonacci (em que se assume que os dois primeiros elementos da sequência são `fib(0) = 1` e `fib(1) = 1`).

```
public static long fibonacci (int n) {
      long resultado;
      if ( n < 2 ){
            resultado = 1;
      }
      else{
            resultado = fibonacci(n - 1) + fibonacci(n - 2);
      }
      return resultado;
 }
```

Ao contrário do método para calcular o fatorial, em que existe apenas uma invocação do método a si próprio – recursão **linear** – o método `fibonacci` invoca-se a si próprio duas vezes. Quando temos um método que faz duas ou mais invocações recursivas estamos perante **recursão não-linear**.

O problema é que, se uma recursão não-linear invocar um método, com os mesmos argumentos, mais do que uma vez, a resolução do problema pode ser ineficiente.

Considere a árvore de recursão da invocação do método de `fibonacci` com valor inicial 5. As caixas com texto carregado indicam onde a computação foi repetida. A maioria das invocações foram repetições. Para números maiores, esta proporção aumenta de tal modo que o cálculo de números de Fibonacci rapidamente se torna inviável.



Para obtermos uma solução melhor temos duas alternativas. A primeira passa por arranjar uma solução iterativa (caso haja uma solução deste tipo mais simples!) como por exemplo:

```
public static long fibonacci (int n) {
        int a = 1, b = 1;
        for (int i = 2; i <= n; i++) {
                int temp = b;
                b += a;
                a = temp;
        }
        return b;
}
```

A segunda solução passa pela alteração da recursão de forma a obter uma recursão linear (tente perceber como este algoritmo encontra a solução):

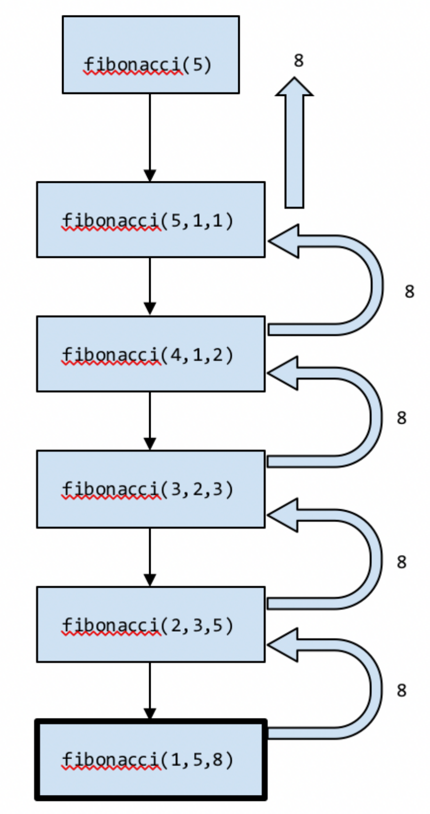
```
public static long fibonacci(int n) {
        return fibonacci(n, 1, 1);
}

private long fibonacci(int n, long a, long b) {
        long resultado;
        if (n < 2){
                resultado = b;
        }
        else{
                resultado = fibonacci(n-1, b, a+b);
        }
        return resultado;
}
```

Se analisar a árvore de recursão ao lado, nota de imediato as diferenças.

De uma recursão não-linear e pouco eficiente, passamos para uma recursão linear com o transporte de um valor intermediário (ou vários) entre as chamadas recursivas. Esta é uma técnica muito poderosa: "simula um *ciclo* através de uma recursão utilizando argumentos extra na invocação recursiva para transportar os valores necessários à execução (neste caso, os argumentos `a` e `b`).

O exemplo mostra o quão importante é a escolha da representação recursiva adequada ao problema, fazendo a diferença entre uma solução recursiva eficiente e uma extremamente ineficiente.



# Memorização de soluções

A *memorização* de soluções é uma outra técnica muito utilizada para tornar mais eficiente uma recursão não-linear.

Com esta técnica trocamos eficiência espacial (uso de pouca memória) por eficiência temporal, isto é, com um custo no uso de memória consegue-se um ganho no tempo de execução.

De notar ainda que esta técnica requer o uso de uma estrutura de dados que seja capaz de guardar as soluções obtidas para eventual uso no futuro.

Para o caso do problema da sucessão de Fibonacci, vejamos como podemos usar esta técnica para o resolver. A estrutura que se usa para as novas soluções encontradas durante a computação é um *array*.

```
private static final int UNKNOWN = -1;

public static long fibonacci(int n) {

     long[] sols = new long[n+1];
     for(int i = 0; i <= n; i++){
          // no inicio, nao se conhece qualquer solucao
          sols[i] = UNKNOWN;
     }

     // invocar a recursao passando o vector de solucoes
    return fibonacci(sols, n);
}

public static long fibonacci(long[] sols, int n) {

     long resultado;
     // jah foi calculado?
     if (sols[n] != UNKNOWN){
          resultado = sols [n];
     } else {
     // base da recursão
         if (n == 0 || n == 1) {
             sols[n] = 1;
             resultado = sols [n];
         } else{
             // passo da recursao
             sols[n] = fibonacci(sols, n-1) + fibonacci(sols, n-2);
             resultado = sols [n];
         }
     }
     return resultado;
}
```

Esta solução do Fibonacci, claro está, é mais complicada que as precedentes, mas serve para ilustrar o conceito.

Este tutorial é uma adaptação (e cópia) de material já existente de outros anos e todos os créditos da originalidade do trabalho são devidos aos seus autores, nomeadamente aos professores João Neto e André Souto.

Este documento apenas serve para suporte às aulas de LabP.