Recursividade

Objetivos do módulo

- Discutir o conceito de recursividade
- Mostrar exemplos de situações onde recursividade é importante
- Discutir a diferença entre recursividade e iteração

O que é recursividade

- Tipicamente em Java os programas são estruturados como objetos que enviam mensagens uns aos outros através de seus métodos de uma forma disciplinada, hierárquica.

```
public static void main(String[] args) {
    A a = new A();
    C c = new C()
    B b = new B(c);

    a.metodoA(b);
}

...
public void metodoA(B b) {
    …
    b.metodoB(getC());
    …
}

...
public void metodoB(C c) {
    …
    c.metodoC();
    ...
}
```

metodoA → metodoB → metodoC
metodoC → metodoB → metodoA (pilha de execução)

- Quando um método de um objeto faz uma chamada a ele mesmo, dizemos que este método é recursivo.
- Um método recursivo é, na verdade, capaz de resolver o caso mais simples do problema que se busca solucionar, ou seja, o caso base. Em outras palavras, quando o

método é chamado para o caso base, ele retorna um resultado. Quando ele é chamado para resolver um caso mais complexo ele divide o problema em duas partes:
    - Um pedaço que ele sabe resolver (caso base);
    - Um pedaço que ele não sabe resolver por completo, mas sabe resolver uma parte, que na verdade lembra o problema original, só que em uma versão mais simples. Assim, neste momento o método chama ele mesmo para resolver a versão mais simples. Cada vez que o método chama ele mesmo, dizemos que ele deu um passo de recursão.
        - A sequencia de chamadas ao método para problemas cada vez menores dele mesmo irá convergir para o caso base, para o qual o método tem uma solução e retornará um resultado, não sendo mais necessários passos de recursão.
- Vejamos um exemplo: o que é um diretório em um computador? Um diretório pode estar vazio, conter arquivos, ou conter outros diretórios (que por sua vez, enquanto diretório pode estar vazio, conter arquivos ou outros diretórios, que por sua vez...). Como faríamos para listar todos os arquivos de um diretório, incluindo os arquivos dos seus subdiretórios?
    - Um método que recebe um diretório e lê os arquivos deste diretório
    - Chamadas recursivas a este método precisariam ser realizadas sempre que um subdiretório for encontrado.
    - O caminhamento em árvores é tipicamente uma atividade recursiva.

```
listaArquivos(Arquivo D) {
   for( Arquivo a : D ) {
      if( a.isDiretorio() ) {
         return listaArquivos(a); //chamada recursiva, para resolver um problema menor
      }
      else {
         System.out.println(a.getNomeCompleto()); //caso base
      }
}
```

Outro exemplo: fatoriais

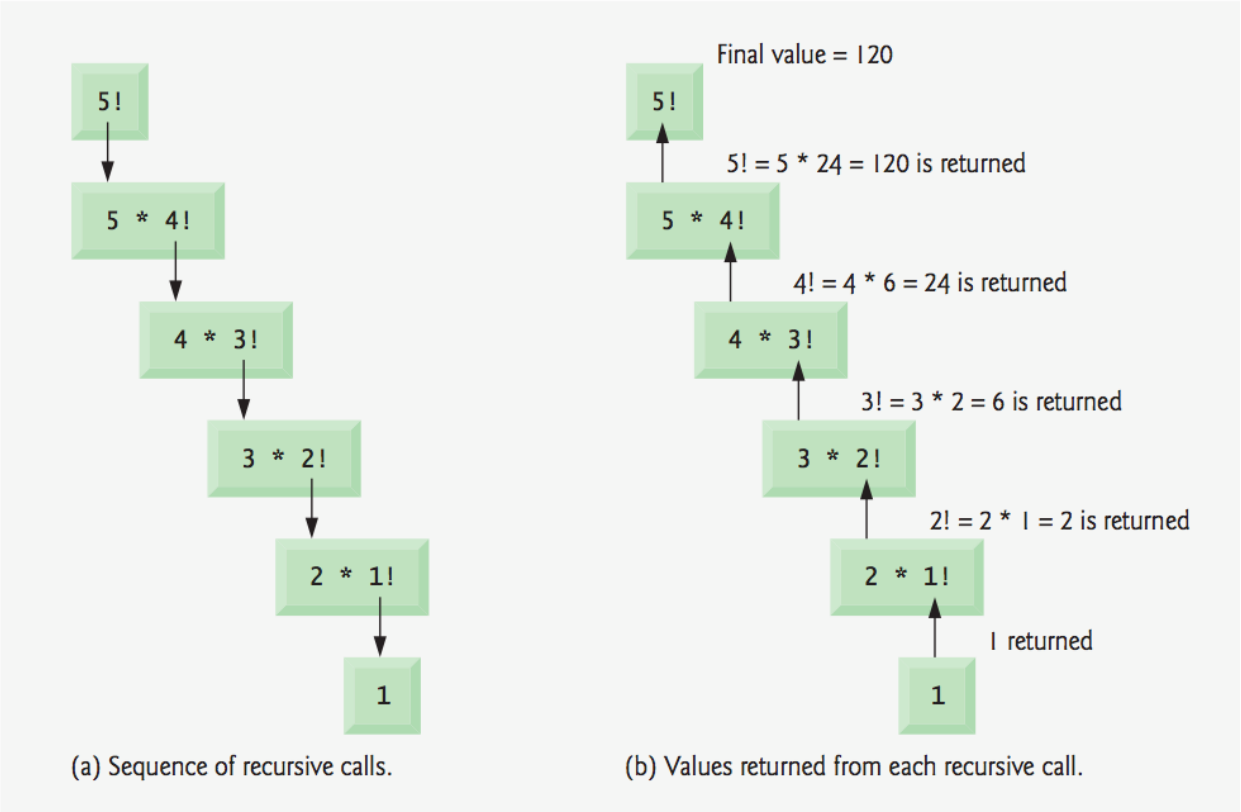
- Relembrando: n! = n . (n-1) . (n-2) . … . 1; 1! = 0! =1
- 5! = 5 . 4 . 3 . 2 . 1 = 120
- Podemos calcular isso de forma iterativa:

```
public void fatorial( int numero ) {
   int fatorial = 1;
   for ( int i = numero; i >= 1; i-- )
       fatorial *= i;
}
```

- Mas também podemos pensar no fatorial como um modelo recursivo: n! = n . (n-1)!
    - 5! = 5 . 4!
- Neste caso, teríamos uma solução diferente:

```
public void fatorial( int numero ) {
   if( numero <= 1 )  //testa se é caso base
      return 1;
   else  //passo de recursão
      return numero * fatorial( numero - 1 );
}
```

- Se chamamos fatorial(5) veja abaixo a sequencia de chamadas recursivas e o retorno de cada uma delas (*imagem retirada de Deitel, Java: How to Program*):



- Erro de recursão infinita: o programador escreve o passo de recursão incorreto, de forma que as chamadas recursivas nunca convergem para o caso base.
- Vamos fazer juntos o exemplo da série de Fibonacci?
  - Vamos analisar a execução de Fibonacci(3); (o i-ésimo termo da série)
  - Veja a explosão de chamadas recursivas.

Escolhendo entre iteração e recursão

- Problemas que podem ser resolvidos recursivamente também podem ser resolvidos por iterações
  - Se a solução é iterativa vamos usar comandos de repetição, como while, for, etc.
  - Se a solução for recursiva vamos usar comandos de seleção, como if, switch, etc.

- Ambos envolvem repetição, mas na solução recursiva ela aparece em chamadas repetidas do método. Ambas terminam, mas a condição de parada é diferente:
    - Para a solução iterativa a condição de parada é a condição de parada do próprio comando de repetição, quando ela falha em ser verdade, a iteração para;
    - Para a solução recursiva, a condição de parada ocorre quando o caso base é alcançado.
- A recursão tem muitas negativas, causando repetidas invocações do mecanismo e consequentemente seu overhead de execução.
    - Maior tempo de processamento e/ou consumo de memória
    - Por exemplo, na execução de Fibonacci(3) teríamos 5 chamadas do mecanismo para alcançar o resultado esperado; já para Fibonacci(6) teríamos 25 chamadas (custo de execução cresce exponencialmente, enquanto na abordagem iterativa o custo é linear)
- Então, por que escolher a recursão?
    - uma abordagem recursiva é preferida quando for uma forma mais natural de resolver o problema, gerando um código mais fácil de entender e depurar
    - Para alguns problemas uma abordagem iterativa não é clara. Isso é bem evidente em algoritmos complexos que necessitam do uso explícito de pilha. Ex. caminhamento em árvore e ordenação baseada em divisão e conquista (Quicksort)
- Porém, o importante é usar recursão com cautela!

Exemplos da necessidade de recursão

Exemplo simples de divisão e conquista: Régua

fonte: http://www2.dcc.ufmg.br/livros/algoritmos/

- Na divisão e conquista tem-se duas (ou mais) chamadas recursivas por passo de recursão
    - Não é feito recomputação excessiva como ocorre no exemplo da série de fibonacci
    - Cada chamada resolve metade do problema
    - Muito usado na prática
- No problema da régua deseja-se desenhar marcas em uma régua

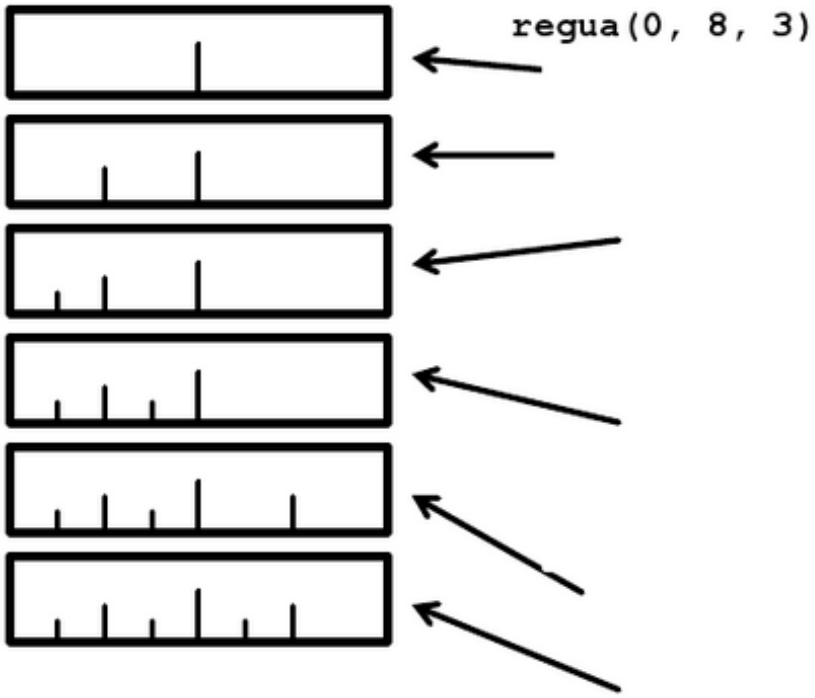
    - As marcas ocorrem em uma determinada posição e altura
    - Para cada centímetro na régua há uma marca no ponto do meio, marcas menores em intervalos de ¼, ainda menores em intervalor de ⅛ e assim por diante.

- Usando a idéia da divisão e conquista: 1) fazer marcas em um intervalo, primeiro uma grande marca no meio que divide o intervalo em duas metades iguais.2) fazer a marca mais curta em cada metade, usando o mesmo procedimento.

- O algoritmo:

```
void regua(int esq, dir, alt){
    if(alt <= 0) return;
    int m = (esq + dir) / 2;
    marca(m, alt);
    regua(esq, m, alt - 1);
    regua(m, dir, alt - 1);
}
```

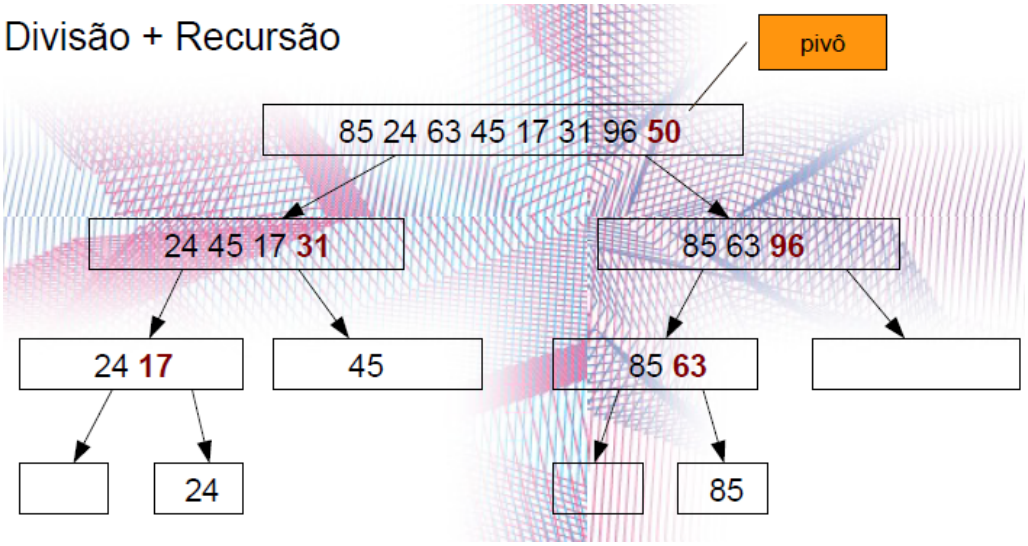
- Execução

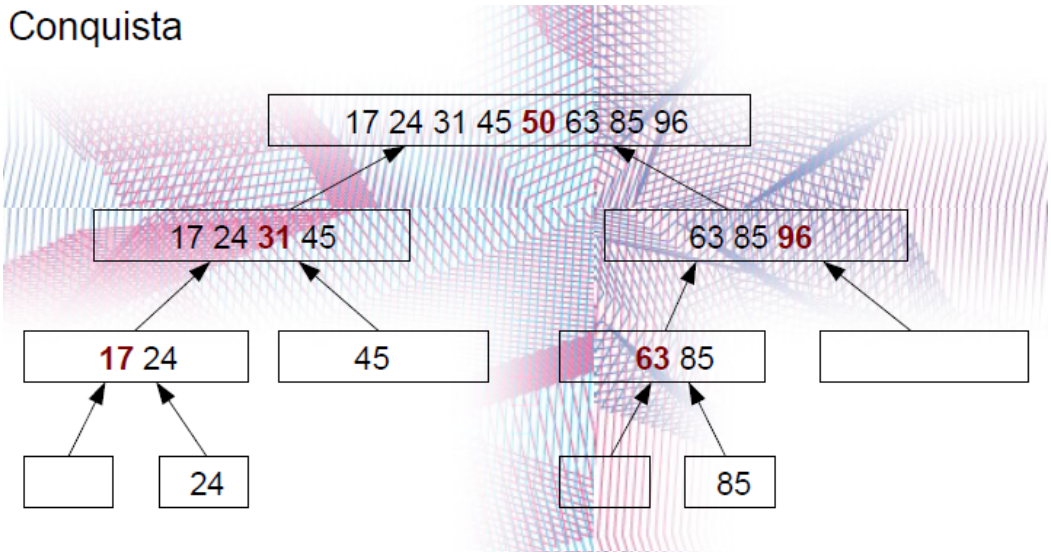


Outros exemplos: Quicksort

- O algoritmo de ordenação quicksort consiste de 3 fases:
  - Divisão: Se a sequência S tem zero ou um elemento, retorne-a imediatamente; já está ordenada. Em qualquer outro caso, (S tem pelo menos dois elementos) escolha um elemento x de S,chamado de pivô. Assim, serão formadas 3 sequências: L (com os elementos de S menores do que x), E (com elementos de S iguais a x) e G (com elementos de S maiores do que x)
  - Recursão: ordene as sequências L e G recursivamente.

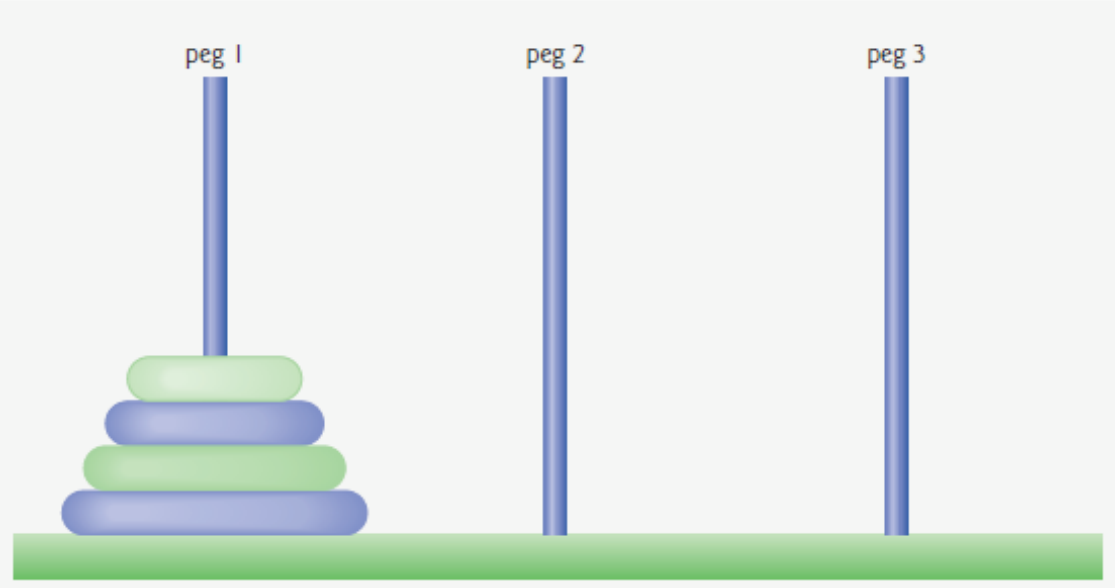
    - Conquista: recomponha S concatenando os elementos de L, em seguida E e, por fim, os elementos de G.
- Animação:
    - http://www.site.uottawa.ca/%7Estan/csi2514/applets/sort/sort.html
- Ilustração:





Outros exemplos: Torres de Hanói
fonte: Deitel, Java: How to Program

- Problema clássico da ciência da computação
    - diz a lenda... (http://pt.wikipedia.org/wiki/Torre_de_Han%C3%B3i)
    - o objetivo é mover uma pilha de disco de um pino para outro, usando um pino intermediário, de modo que um disco maior nunca seja colocado sobre um disco menor

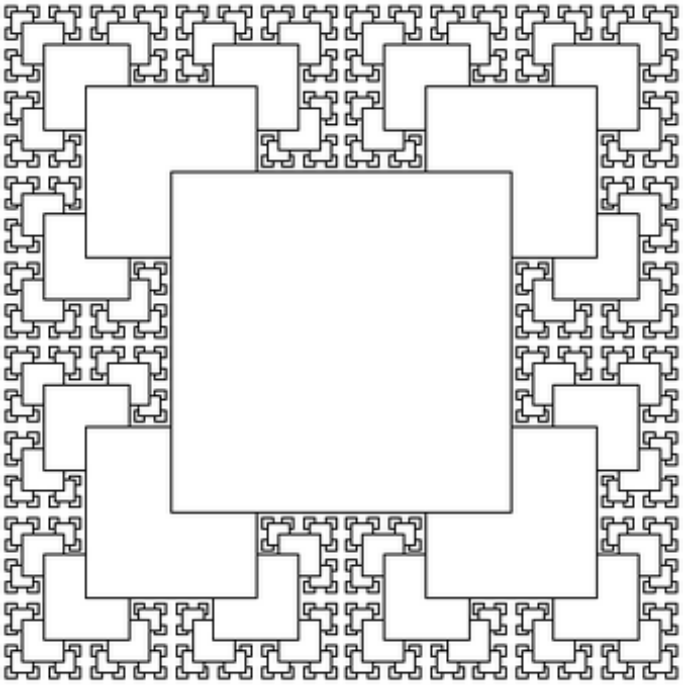


- Algoritmo segue a idéia de que mover n discos pode ser visualizado em termos de mover somente n-1 discos:
  - mova n-1 discos do pino 1 para o pino 2, utilizando o pino 3 como área de armazenamento intermediário
  - mova o último disco (o maior) do pino 1 para o pino 3
  - mova os n-1 discos do pino 2 para o pino 3, utilizando o pino 1 como área de armazenamento temporário
- Exemplo de implementação:

```
// recursively move disks between towers
public void solveTowers( int disks, int sourcePeg, int destinationPeg,
   int tempPeg )
{
   // base case -- only one disk to move
   if ( disks == 1 )
   {
      System.out.printf( "\n%d --> %d", sourcePeg, destinationPeg );
      return;
   } // end if

   // recursion step -- move (disk - 1) disks from sourcePeg
   // to tempPeg using destinationPeg
   solveTowers( disks - 1, sourcePeg, tempPeg, destinationPeg );

   // move last disk from sourcePeg to destinationPeg
   System.out.printf( "\n%d --> %d", sourcePeg, destinationPeg );

   // move ( disks - 1 ) disks from tempPeg to destinationPeg
   solveTowers( disks - 1, tempPeg, destinationPeg, sourcePeg );
} // end method solveTowers
```

Outros exemplos: Fractais

- Fractais são figuras geométricas que podem ser frequentemente geradas a partir de um padrão repetido recursiva e infinitamente.
    - A figura é modificada aplicando o padrão a cada segmento da figura original



    - Algoritmo:

```
void fractal(int x, y, r)
{
    if(r <= 0) { return; }
    fractal(x-r, y+r, r / 2);
    fractal (x+r, y+r, r / 2);
    fractal (x-r, y-r, r / 2);
    fractal (x+r, y-r, r / 2);
    quadrado(x, y, r);
}
```

x e y são as coordenadas do centro.
r o valor da metade do lado.