INE5603 Introdução à POO

Prof. A. G. Silva

25 de setembro de 2017

Recursividade

Baseado em materiais da

Unisinos, Cesar Tacla (UTFPR), Renato Ferreira, David Menotti (UFPR)

Recursão

- Repetição pode ser obtida de duas maneiras:
  - Laços (for, while, etc)
  - Chamada recursiva de métodos (recursão)
- Recursão é uma técnica de programação na qual um método chama a si mesmo.

Exemplo da multiplicação

- Considere, por exemplo, a definição da multiplicação, em termos da operação mais simples de adição:
  - Informalmente, multiplicar *m* por *n* (onde *n* não é negativo) é somar *m*, *n* vezes:

$$m \times n = \underbrace{m + \ldots + m}_{n \text{ vezes}}$$

- Solução de problema que realiza operações repetidamente pode ser implementada usando comando de repetição (também chamado de comando iterativo ou comando de iteração).

Implementação iterativa da multiplicação

```
public static int mult (int m, int n) {
    int r=0;
    for (int i=1; i<=n; i++) r += m;
    return r;
}

public static void main (String args[]){

    int resultado = Recursao.mult (3,5);
    System.out.println (resultado);
    
}
```

Multiplicação recursiva

- Também é possível implementar a multiplicação de um número *m* por *n* somando *m* com a multiplicação de *m* por *n*-1.
  - $m * n = m + (m * (n\text{-}1))$
  - $2*4 = 2 + (2*3)$
- A operação de multiplicação está sendo chamada novamente, mas agora para resolver um sub-problema que é parte do anterior.

Multiplicação recursiva

- A multiplicação de um número inteiro por outro inteiro maior ou igual a zero pode ser definida recursivamente por indução matemática como a seguir:

$$\begin{cases} m \times n = 0 & \text{se } n == 0 \\ m \times n = m + (m \times (n - 1)) & \text{se } n > 0 \end{cases}$$

Recursão é o equivalente, em programação, à **indução matemática** que é uma maneira de definir algo em termos de si mesmo.

- Que pode ser implementado em Java da seguinte maneira:

```
public static int multr (int m, int n) {
    if (n==0) return 0;
    else return (m + multr(m, n-1));
}
```

Fatorial recursivo

- Definição não recursiva (tradicional):

$$\begin{cases} N! = 1, & \text{para } N \leq 1 \\ N! = 1 \times 2 \times 3 \times .... \times N, & \text{para } N > 0 \end{cases}$$

- Definição recursiva:

$$\begin{cases} N! = 1, & \text{para } N \leq 1 \\ N! = N \times (N - 1)!, & \text{para N} > 0 \end{cases}$$

Fatorial recursivo

- Definição não recursiva (tradicional):

$$\begin{cases} N! = 1, & \text{para } N \leq 1 \\ N! = 1 \times 2 \times 3 \times \ldots \times N, & \text{para } N > 0 \end{cases}$$

- implementação iterativa:

```
public static int fatorial (int numero) {
    int resultado = 1;
    for(int i=numero;i>0;i--){
        resultado = resultado * i;
    }
    return resultado;
}
```

Fatorial recursivo

Definição recursiva:

$$\begin{cases} N! = 1, & \text{para } N \leq 1 \\ N! = N \times (N-1)!, & \text{para } N > 0 \end{cases}$$

Caso base

Caso recursivo

```
public static int fatorialr(int n){
    if(n<=1){
        return 1;
    }else{
        return n*fatorialr(n-1);
    }

}
```

Um método que chama a si mesmo é chamado de **método recursivo**.

Características dos programas recursivos

a) Chama *a si mesmo* para resolver *parte do problema*;

b) Existe pelo menos um *caso base* que é resolvido sem chamar a si mesmo, definindo um critério de parada.

Recursão linear

- A recursão linear é a forma mais simples de recursão.
- O método faz apenas uma chamada recursiva (uma chamada a si mesmo).
- Esse tipo de recursão é útil quando se analisam os problemas de algoritmo em termos:
  - Do primeiro ou último elemento
  - Mais um conjunto restante que tem a mesma estrutura que o conjunto original

Recursão linear (exemplo)

- Exemplo: Soma de n inteiros em um array A

$$\begin{cases} \textbf{se } n{=}1 \ linearSum{=}A[0] \\ \textbf{senão } linearSum = A[\text{n-1}] + linearSum(A, n\text{-}1) \end{cases}$$

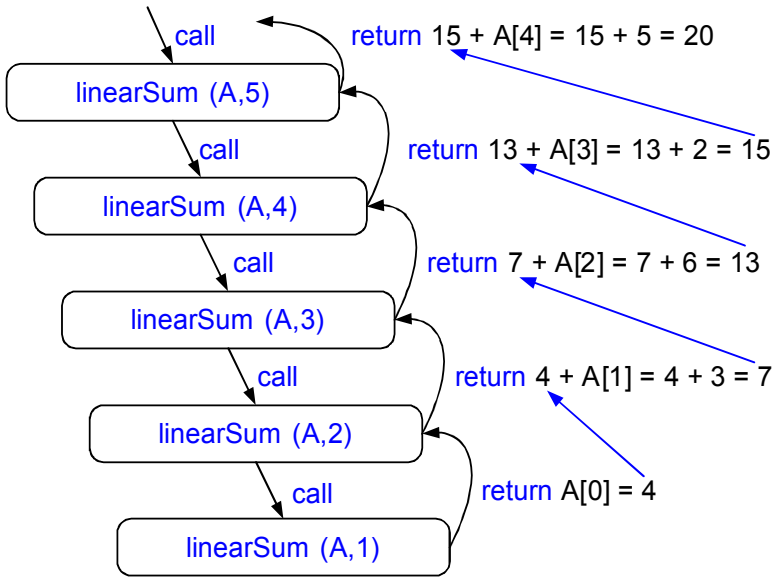
Algoritmo *linearSum*(*A*, *n*):
Entrada:
um array A e um inteiro *n* que contém o tamanho de A
Saída:
A soma dos primeiros *n* elementos de A

se *n* = 1 então
returna A[0]
caso contrário
returna *linearSum*(A, *n* - 1) + *A*[*n* -1]

Recursão linear (exemplo)

```
public static int linearSum (int A[], int n) {
    if (n==1) return A[0];
    else return linearSum(A, n-1) + A[n-1];
}
```

Trace:



Invertendo o conteúdo de um *array*

Algoritmo reverseArray(*A*, *i*, *j*):

Input: Um array *A* e dois índices inteiros *i* e *j*

Output: O inverso dos elementos em *A* iniciando no índice *i* e finalizando em *j*

se *i* < *j* então

swap *A*[*i*] e *A*[*j*] //troca entre si

reverseArray(*A*, *i* + 1, *j* - 1)

retorna

```
public static void reverseArray (int A[], int i, int j) {
     if (i <  j) {
        swap (A, i, j);
        reverseArray(A, i + 1,  j - 1);
     }
     else return;
}
```

```
private static void swap (int A[], int i, int j) {
     int temp = A[i];
     A[i] = A[j];
     A[j] = temp;
}
```

Definindo argumentos

- Ao criar métodos recursivos, é importante definir os métodos de maneira a facilitar a recursão.
- Isso algumas vezes requer que definamos alguns parâmetros adicionais a serem passados pelo método recursivo.
  - Por exemplo, nós definimos o método recursivo como *reverseArray*(*A*, *i*, *j*), não como *reverseArray*(*A*).
- Uma maneira mais elegante é:
  - criar um método público com menos argumentos e não recursivo (para as outras classes chamarem)
  - e um método privado recursivo com os argumentos necessários.

```
public static void reverseArray (int A[]){
    reverseArray (A, 0, A.length-1);
}

private static void reverseArray (int A[], int i, int j) {
     if (i <  j) {
        swap (A, i, j);
        reverseArray(A, i + 1,  j - 1);
     }
     else return;
}
```

método público não recursivo

método privado recursivo

Recursão binária

- Recursão binária ocorre sempre que houver duas chamadas recursivas para cada caso não que não é base.
- Essas chamadas podem, por exemplo, ser usadas para resolver duas metades do mesmo problema.

Sequência de Fibonacci

- A sequência de Fibonacci consiste nos inteiros:
  - 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...
- Cada elemento nessa sequência é a soma dos dois elementos anteriores. Por exemplo:
  - 0+1=1; 1+1=2; 1+2=3; 2+3=5 ...
  - Assume-se que os 2 primeiros elementos são 0 e 1.
- Se partirmos que fib(0)=0 e fib(1)=1 e assim por diante, podemos definir um número da sequência Fibonnaci da seguinte maneira:

$$\begin{cases} \text{fib}(n) = n & \text{se } n=0 \text{ OU } n=1 \\ \text{fib}(n) = \text{fib}(n\text{-}2) + \text{fib}(n\text{-}1) & \text{se } n \geq 2 \end{cases}$$

Seqüência de Fibonacci (implementação)

$$\begin{cases} \text{fib}(n) = n & \text{se } n=0 \text{ OU } n=1 \\ \text{fib}(n) = \text{fib}(n\text{-}2) + \text{fib}(n\text{-}1) & \text{se } n \geq 2 \end{cases}$$

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

Atenção: esta não é a solução mais indicada para Fibonacci devido à "explosão" de recálculos efetuados

Múltiplas chamadas recursivas

- Recursão múltipla ocorre quando um método faz mais de duas chamadas recursivas.
- Uma aplicação possível: enumerar várias configurações para resolver um quebra-cabeças

Chamada a um método

```
public class Ponto {
      private int x, y;

      public Ponto(int x, int y)  {  setaCoordenadas(x, y);  }

      public void setaCoordenadas(int x, int y) {
        this.x = x;
        this.y = y;
      }

      public String exibe() {..
    }
```

parâmetros formais

```
int x=2, y=3;
Ponto p = new Ponto ();
p.setaCoordenadas (x, y);
p.exibe();
```

parâmetros reais

Tipos dos parâmetros reais devem ser compatíveis com tipos dos parâmetros formais

Chamada a um método

```
public class Ponto {
        private int x, y;

        public Ponto(int x, int y)  {  setaCoordenadas(x, y);  }

        public void setaCoordenadas(int x, int y) {
          this.x = x;
          this.y = y;
        }

        public String exibe() {..
    }
```

parâmetros formais

```
int x=2, y=3;
Ponto p = new Ponto ();
p.setaCoordenadas (x, y);
p.exibe();
```

parâmetros reais

Parâmetros formais são variáveis locais do método. Outras variáveis locais podem ser declaradas (ex: r em mult).

Chamada a um método

```
public class Ponto {
        private int x, y;

        public Ponto(int x, int y)  {  setaCoordenadas(x, y);  }

        public void setaCoordenadas(int x, int y) {
          this.x = x;
          this.y = y;
        }

        public String exibe() {..
    }
```

parâmetros formais

```
int x=2, y=3;
Ponto p = new Ponto ();
p.setaCoordenadas (x, y);
p.exibe();
```

parâmetros reais

Quando execução de uma chamada termina, execução retorna ao ponto da chamada.

Chamada de método

- Quando um método é chamado:
  - é necessário inicializar os parâmetros formais com os valores passados como argumentos;
  - sistema precisa saber onde reiniciar a execução do programa;
  - ...
- Informações de cada método (variáveis e endereço de retorno) devem ser guardadas até o método acabar a sua execução.
- Mas como o programa diferencia a variável *n* da primeira chamada da variável *n* da segunda chamada do método *fatorialr* ?

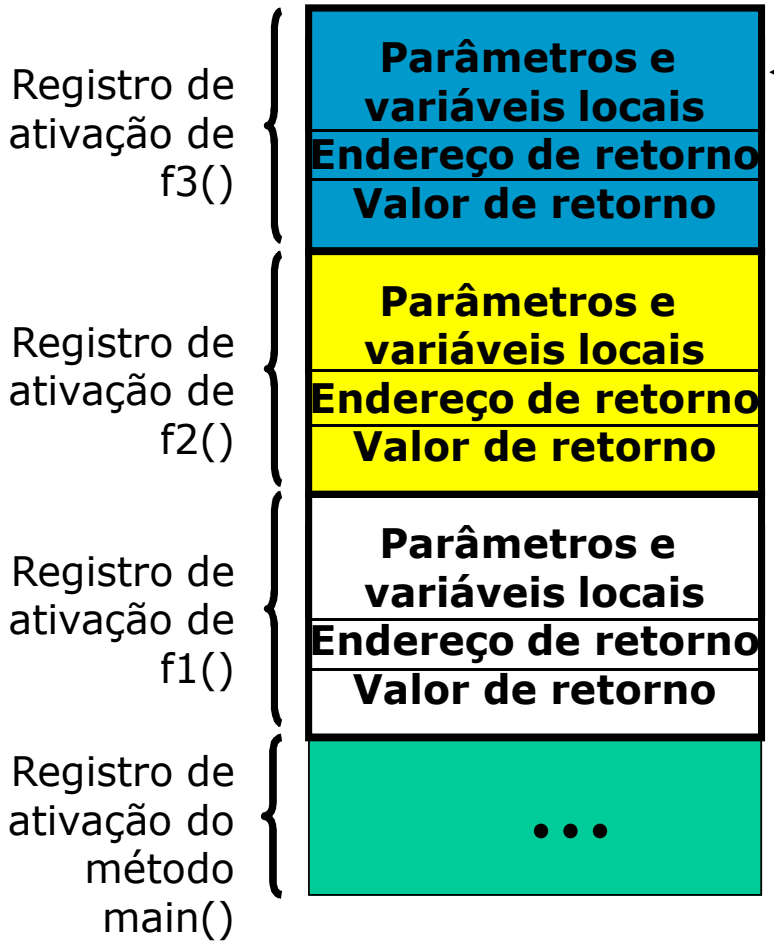
```
public static int fatorialr(int n){
    if(n<=1){
        return 1;
    }else{
        return n*fatorialr(n-1);
    }

}
```

Registro de ativação

- Registro de ativação:
  - área de memória que guarda o estado de uma função, ou seja:
    - variáveis locais
    - valores dos parâmetros;
    - endereço de retorno (instrução após a chamada do método corrente);
    - valor de retorno.
- Registro de ativação são criados em uma pilha em tempo de execução;
- Existe um registro de ativação (um nó na pilha) para cada chamada ao método;
- Quando um método é chamado é criado um registro de ativação para este e este é empilhado na pilha;
- Quando o método finaliza sua execução o registro de ativação desse método é desalocado.

Registro de ativação



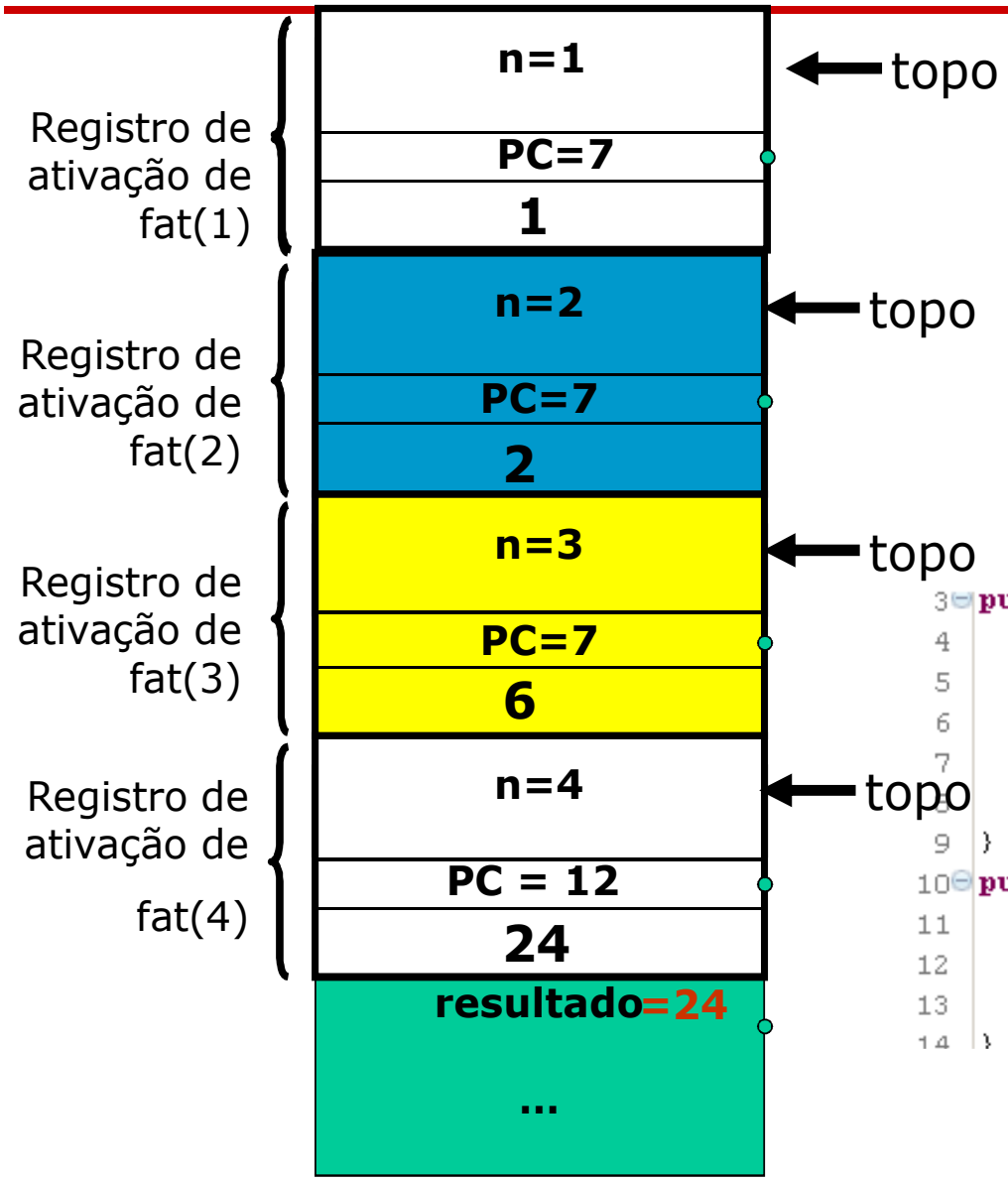

```java
public static void f3() {
    int x=3;
}

public static void f2() {
    int x=2;
    f3();
}

public static void f1() {
    int x=1;
    f2();
}

public static void main (String args[]){
    f1();
}
```

Registro de ativação: exemplo



fatorial de 4

fat (4) → main

4*fat(3) =24

3*fat(2) =6

2*fat(1) =2

1

```
3  public static int fatorialr(int n) {
4      if (n <= 1) {
5          return 1;
6      } else {
7          return n * fatorialr(n - 1);
8      }
9  }
10 public static void main(String args[]) {
11
12     int resultado = Recursao.fatorialr(4);
13     System.out.println(resultado);
14 }
```

Registro de ativação

- A cada término de FAT, o controle retorna para a expressão onde foi feita a chamada na execução anterior, e o último conjunto de variáveis que foi alocado é liberado nesse momento. Esse mecanismo utiliza uma `pilha`.
- A cada nova chamada do método FAT, um novo conjunto de variáveis (no caso, apenas *n*) é alocado.

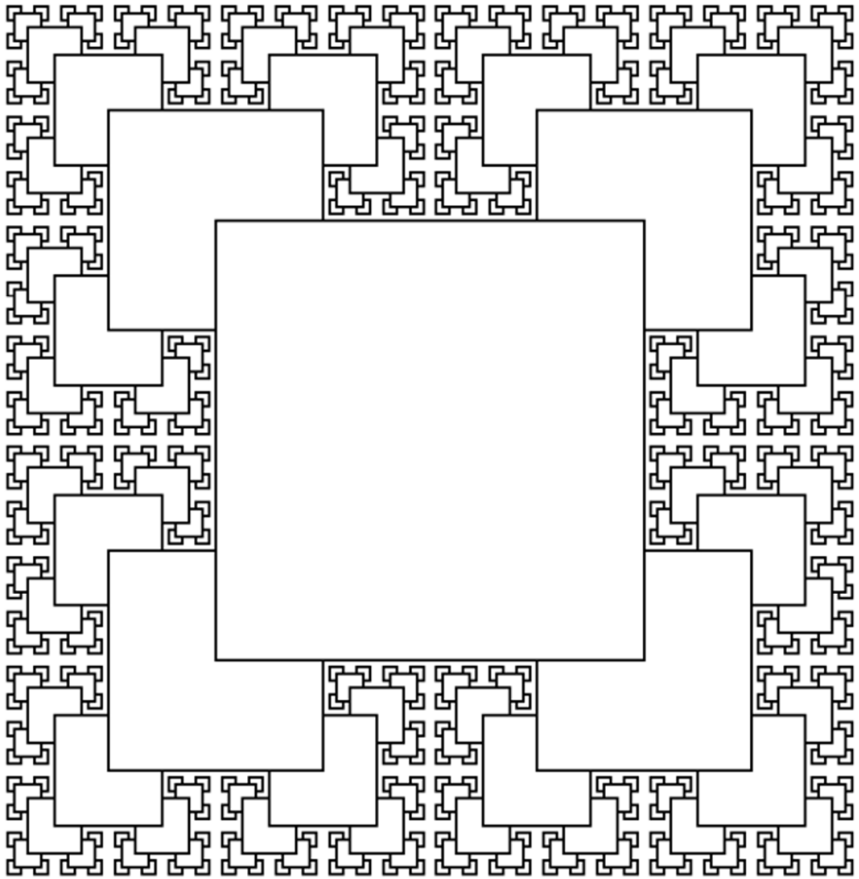
Dicas para desenvolver algoritmos recursivos

- Montar, inicialmente, uma definição (especificação) recursiva do problema, como segue:
  1. Definir pelo menos um caso base;
  2. Quebrar o problema em subproblemas, definindo o(s) caso(s) recursivo(s);
  3. Fazer o teste de finitude, isto é, certificar-se de que as sucessivas chamadas recursivas levam obrigatoriamente, e numa quantidade finita de vezes, ao(s) caso(s) básico(s).
- Depois, é só traduzir essa especificação para a linguagem de programação.

Vantagens e Desvantagens

- Vantagens da recursão
  - Redução do tamanho do código fonte
  - Maior clareza do algoritmo para problemas de definição *naturalmente recursiva*
- Desvantagens da recursão
  - Memória e tempo de execução extra para gerenciamento das chamadas
  - Depuração inicialmente mais complicada de subprogramas recursivos

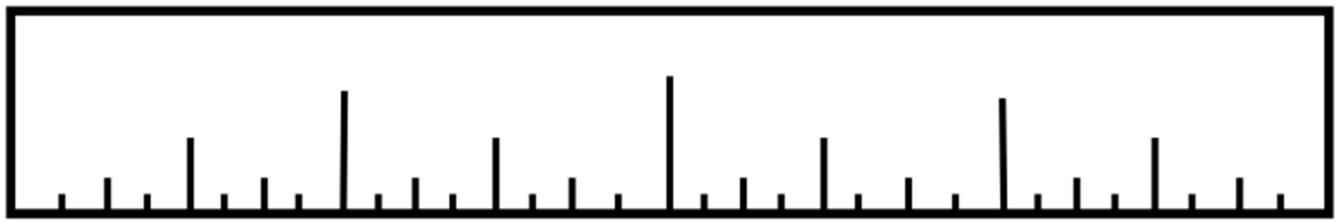
Outro exemplo de recursividade: estrela



```
void estrela(int x,int y, int r)
{
  if ( r > 0 )
  {
    estrela(x-r, y+r, r div 2);
    estrela(x+r, y+r, r div 2);
    estrela(x-r, y-r, r div 2);
    estrela(x+r, y-r, r div 2);
    box(x, y, r);
  }
}
```

Outro exemplo de recursividade: régua

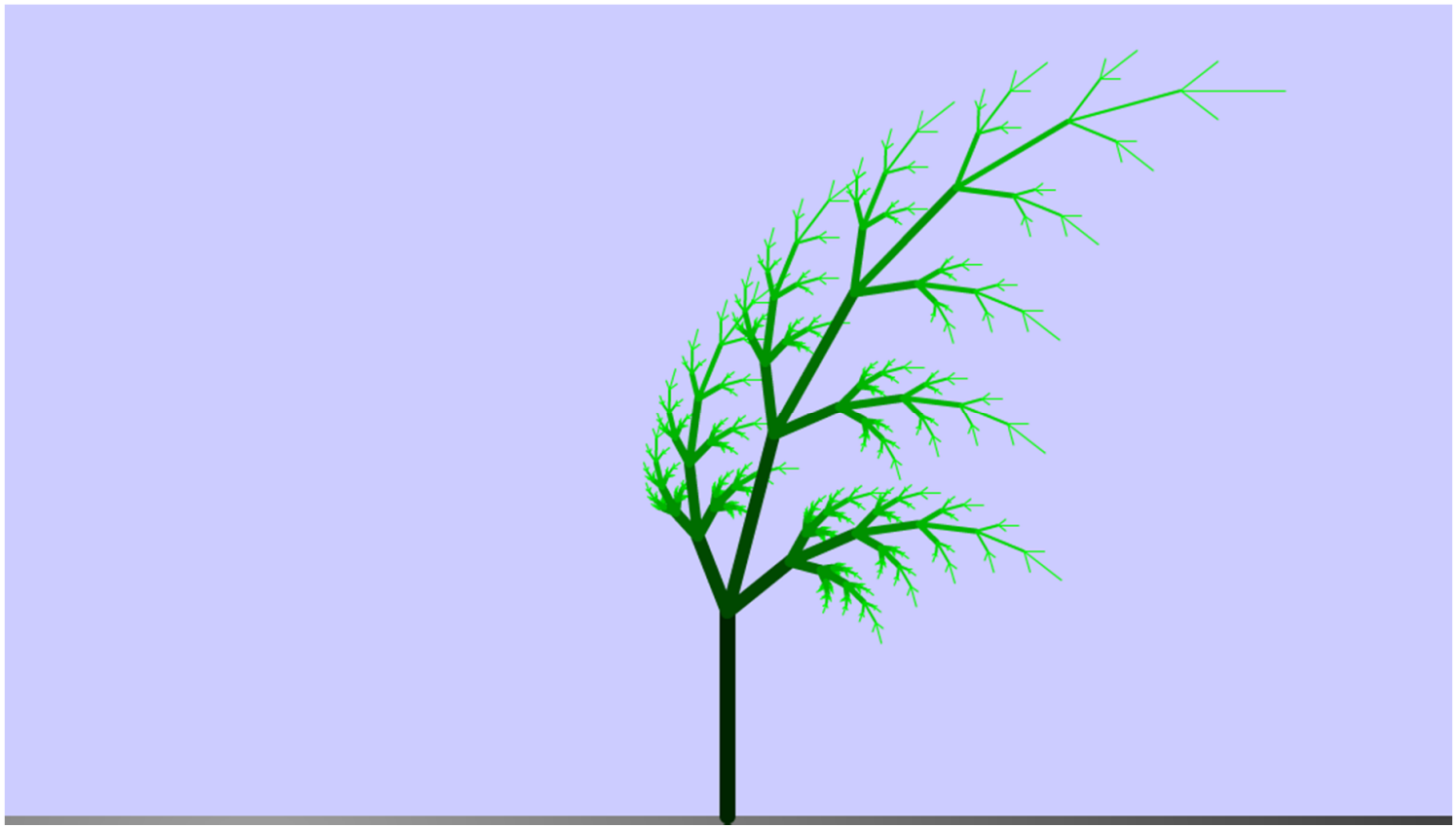
```
int regua(int l,int r,int h)
{
  int m;
  if ( h > 0 )
  {
    m = (l + r) / 2;
    marca(m, h);
    regua(l, m, h - 1);
    regua(m, r, h - 1);
  }
}
```



Execução da recursividade: régua

```
regua(0, 8, 3)
  marca(4, 3)
  regua(0, 4, 2)
    marca(2, 2)
    regua(0, 2, 1)
      marca(1, 1)
      regua(0, 1, 0)
      regua(1, 2, 0)
    regua(2, 4, 1)
      marca(3, 1)
      regua(2, 3, 0)
      regua(3, 4, 0)
regua(4, 8, 2)
  marca(6, 2)
  regua(4, 6, 1)
    marca(5, 1)
    regua(4, 5, 0)
    regua(5, 6, 0)
  regua(6, 8, 1)
    marca(7, 1)
    regua(6, 7, 0)
    regua(7, 8, 0)
```

Outro exemplo de recursividade: fractal de Fern



https://krazydad.com/bestiary/bestiary_fern.html

Conceito de recursividade

- Fundamental em Matemática e Ciência da Computação
  - Uma função recursiva é definida em termos dela mesma
  - Um programa recursivo é um programa que chama a si mesmo
- Exemplos
  - Função fatorial, sequências, fractais, jogos (ex.: Torres de Hanoi), crescimento de regiões (algoritmo de pintura), árvores (serão vistas na disciplina de "Estruturas de Dados")...
- Conceito poderoso
  - Define conjuntos infinitos com comandos finitos