

# Estrutura de Dados I

## Capítulo 5: Procedimentos Recursivos

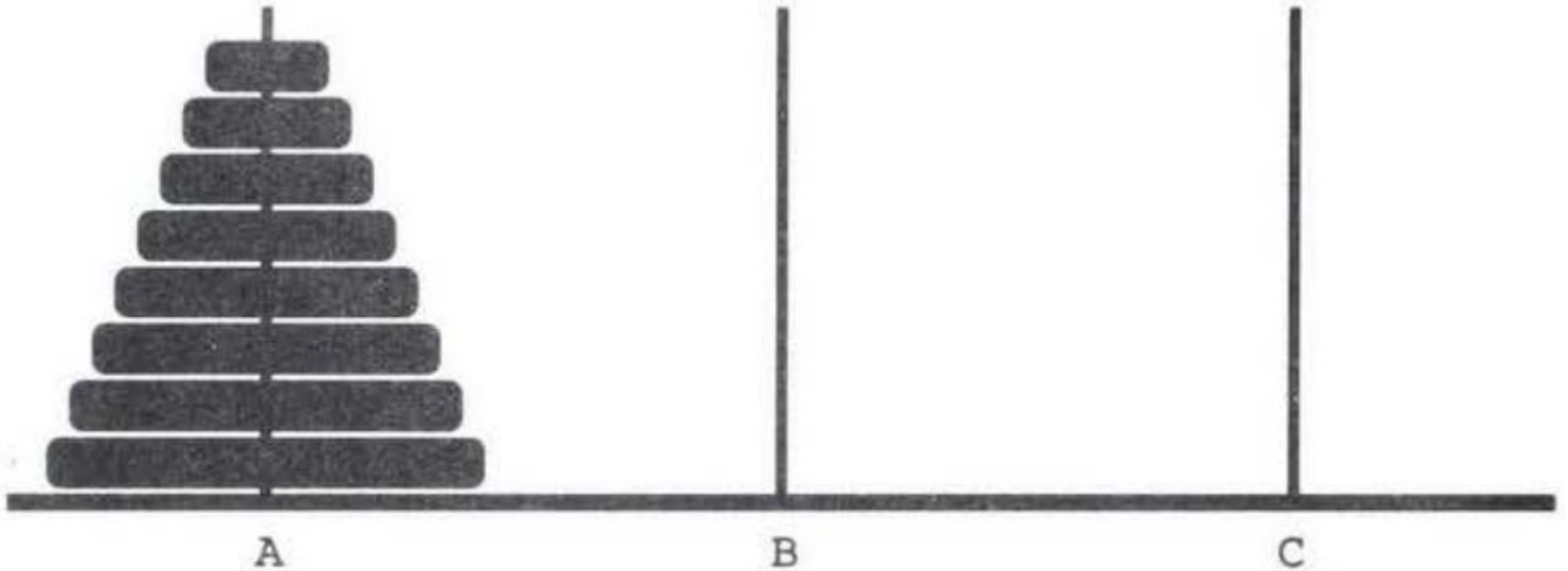
# Recursividade em problemas complexos

- Os exemplos que estudamos até agora eram exemplos simples, que tinham uma tradução praticamente direta entre uma definição matemática e a implementação recursiva.
- Quando os problemas são mais complexos a situação muda:
  - **É difícil encontrar a decomposição recursiva!**
- Você precisará de:
  - **Esperteza**
  - **Confiança**
  - **Salto de fé**

Você deve ser capaz de chegar a um ponto no qual você obteve um problema menor, da mesma forma que o original, e alguns casos simples. Nesse momento você deve parar e declarar o problema resolvido, sem tentar monitorar as chamadas internas.

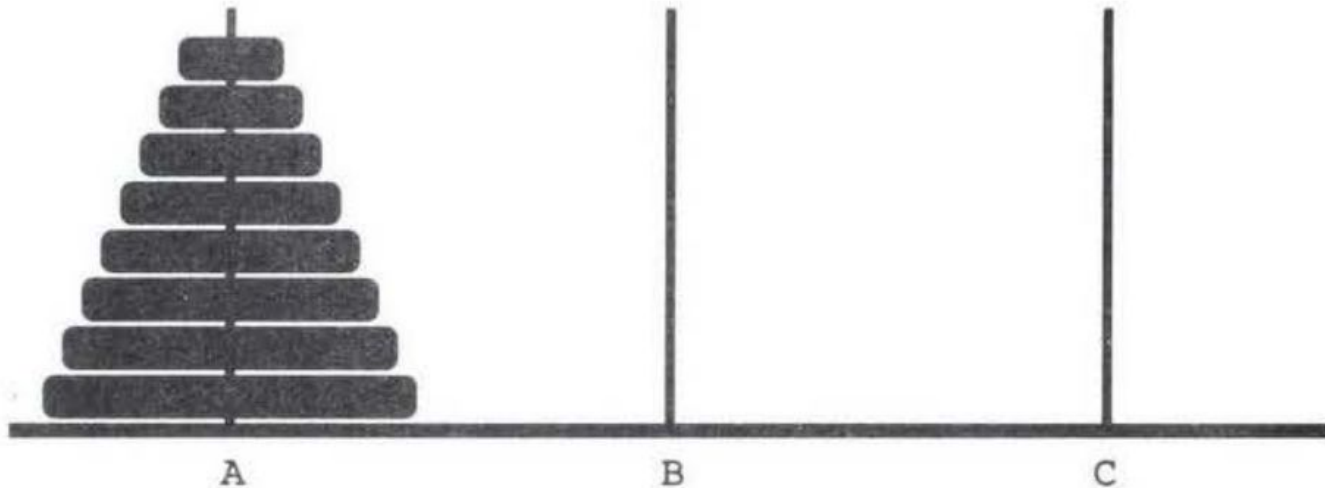
# A Torre de Hanoi

- Edouard Lucas, matemático Francês, na década de 1880.



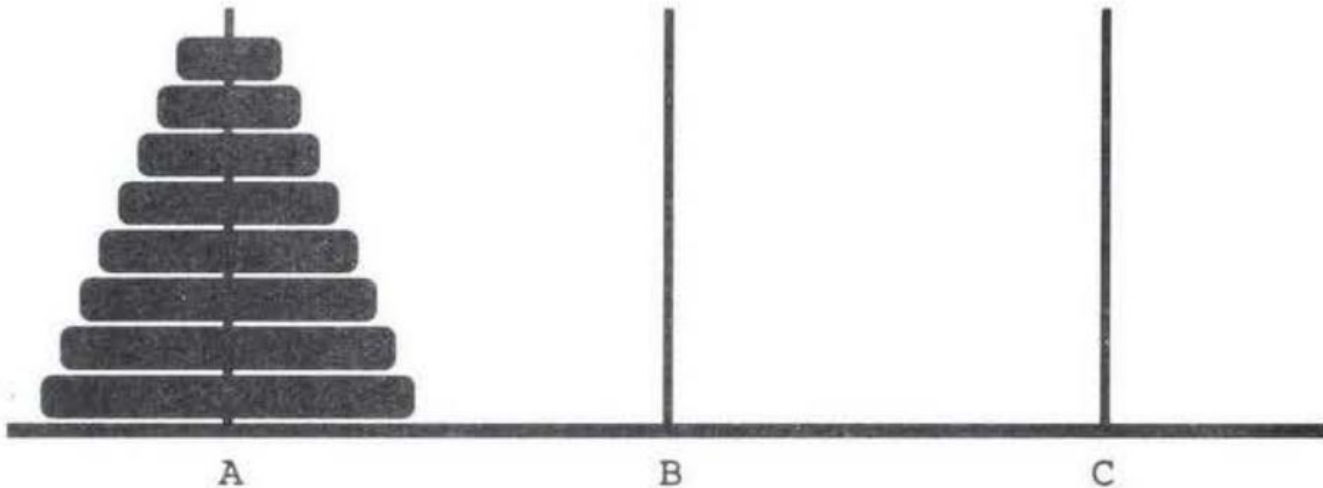
# A Torre de Hanoi

- Você deve mover todos os discos da torre A para a torre B, utilizando a torre C como armazenamento temporário, desde que:
  - Você só pode mover 1 disco de cada vez; e
  - Não é permitido colocar um disco maior sobre um disco menor.



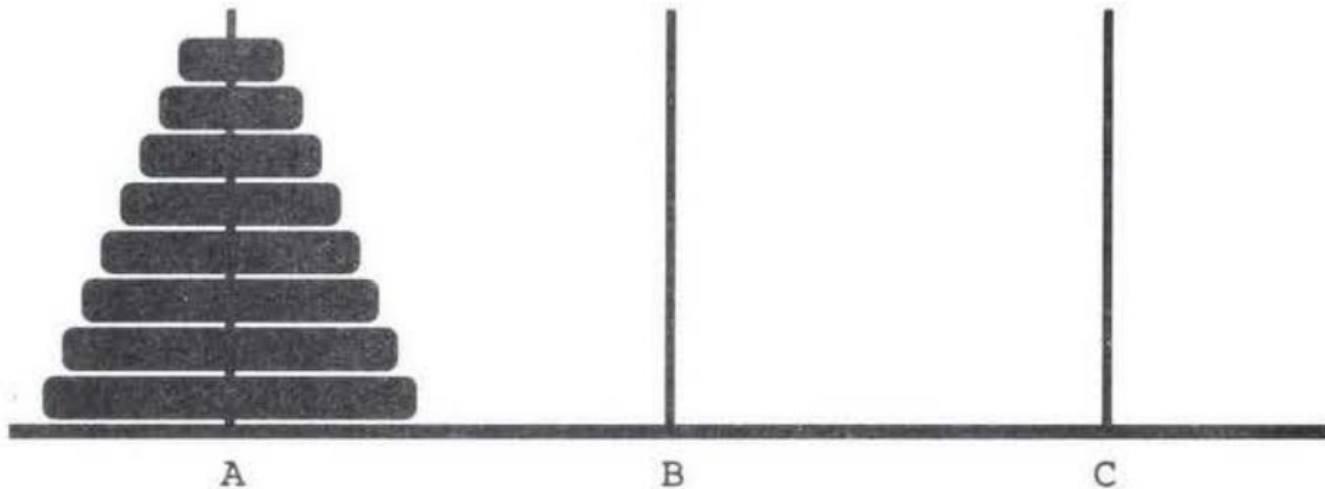
# A Torre de Hanoi

- Qual(is) o(s) caso(s) simples?
- Qual a decomposição recursiva?



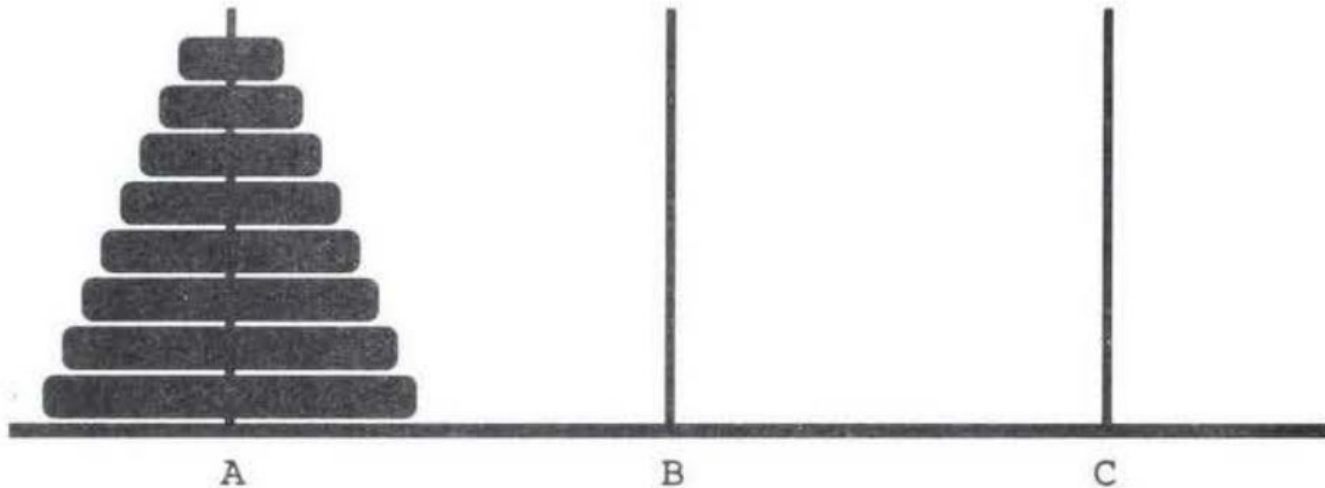
# A Torre de Hanoi

- Qual(is) o(s) caso(s) simples?
  - **Uma torre com apenas 1 disco**
- Qual a decomposição recursiva?
  - **Mover sub-torres menores de uma torre para outra**



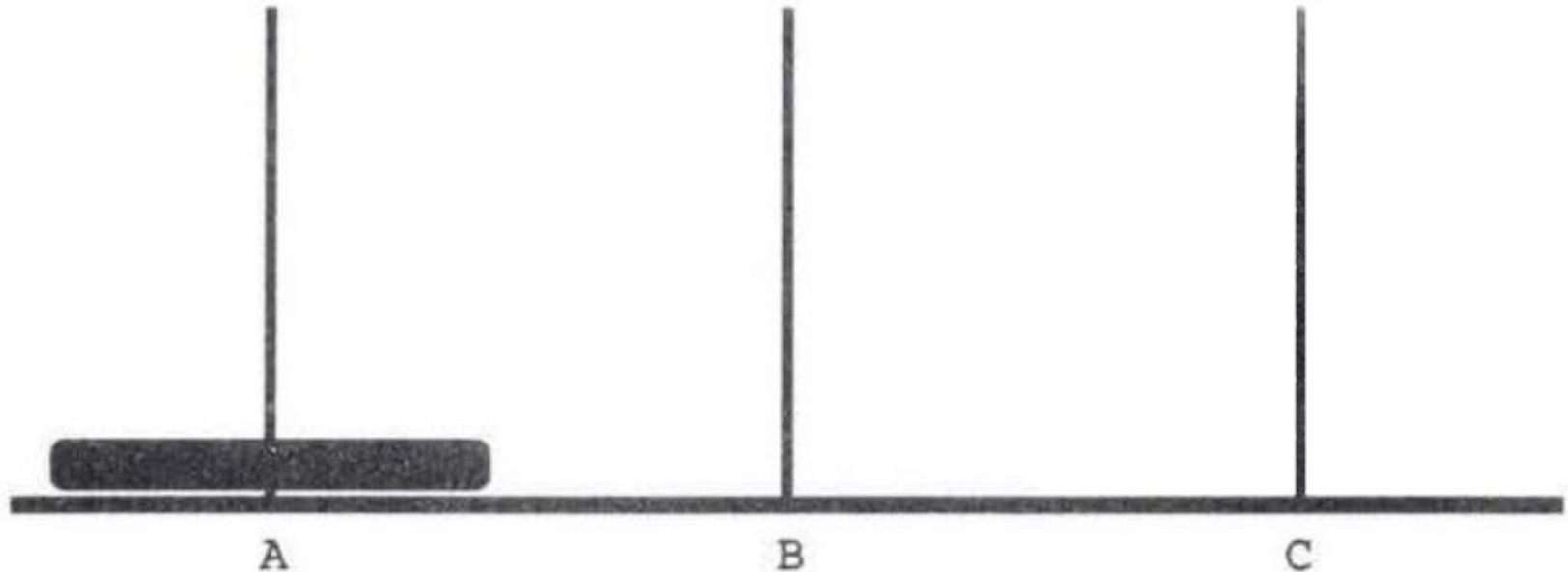
# A Torre de Hanoi

```
void hanoi (int n, char origem, char destino, char temp);
```



# A Torre de Hanoi: o caso simples

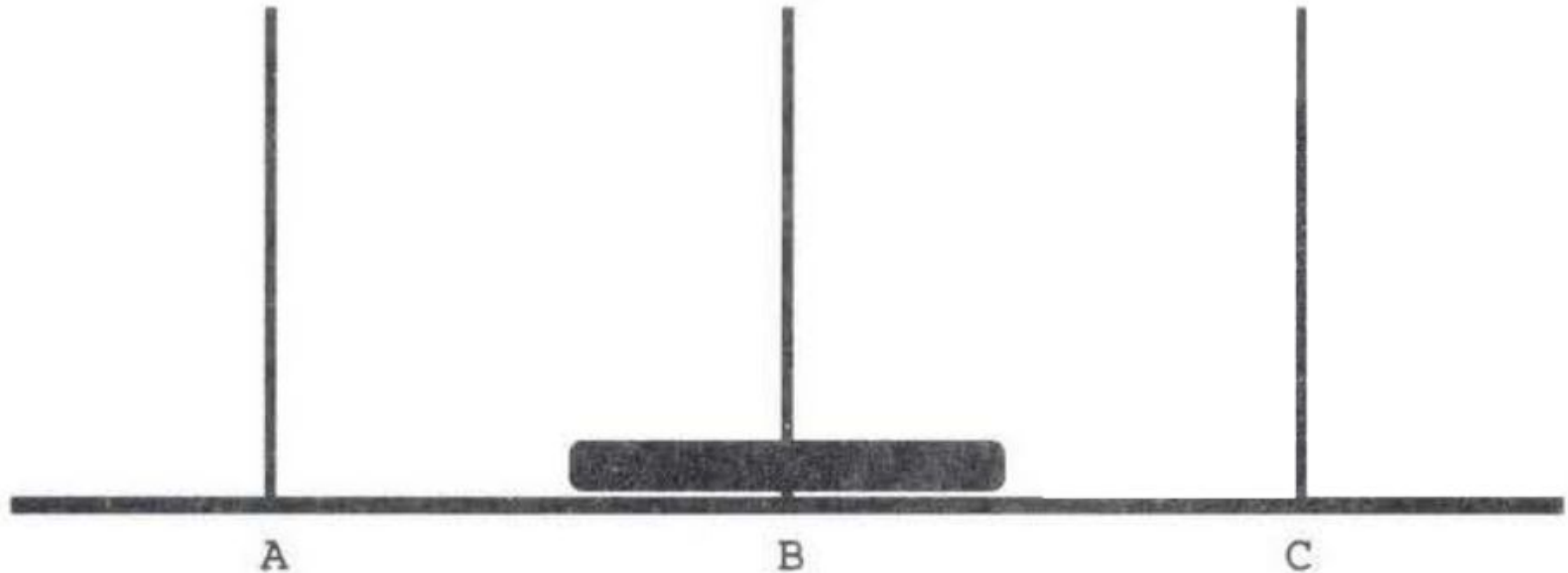
```
void hanoi (int n, char origem, char destino, char temp);
```





# A Torre de Hanoi: o caso simples

```
void hanoi (int n, char origem, char destino, char temp);
```

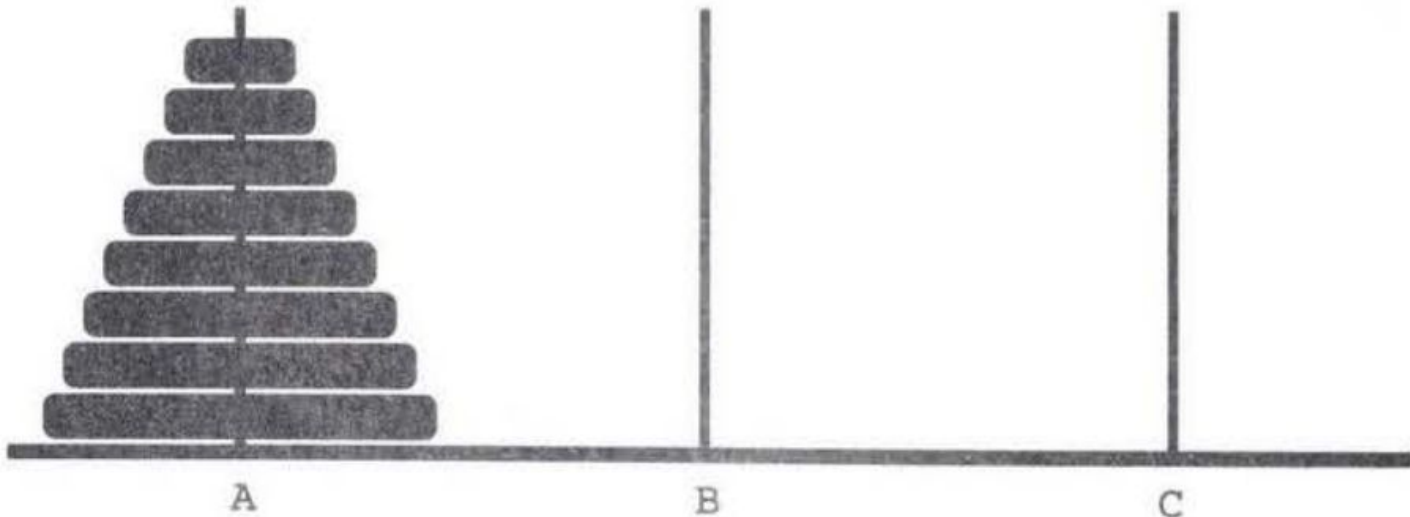


## A Torre de Hanoi: o caso simples

```
void hanoi (int n, char origem, char destino, char temp)
{
    if (n == 1)
        // Mover o único disco da ORIGEM para o DESTINO;
}
```

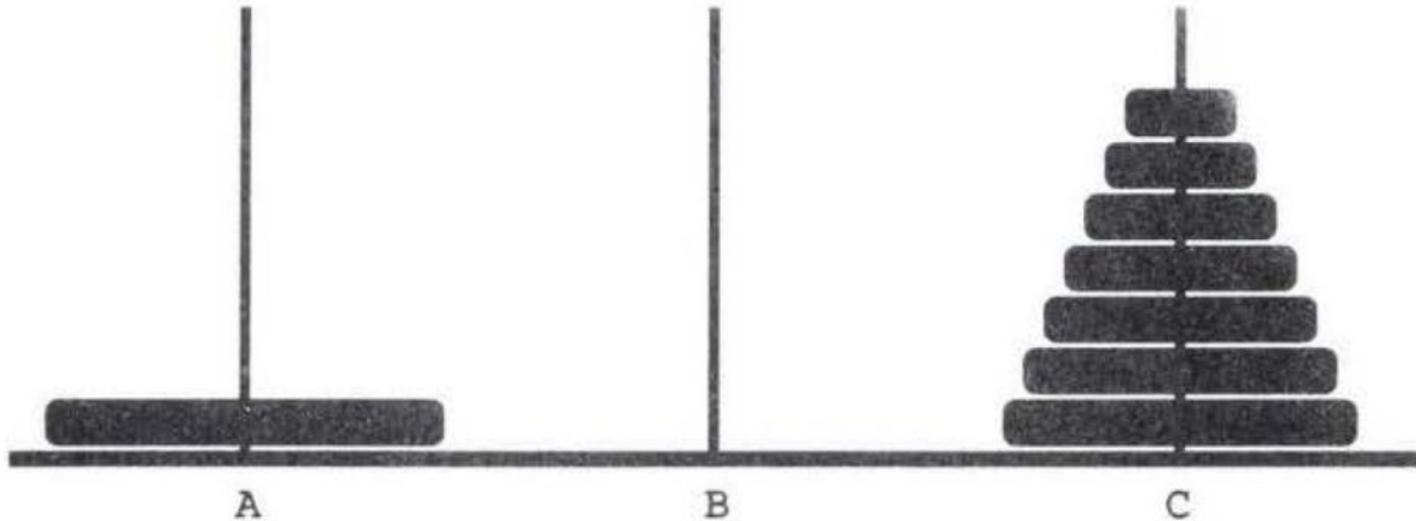
# A Torre de Hanoi: a decomposição recursiva

- **Como a solução para um problema menor ajudaria na solução do problema maior?** Se estamos inicialmente com 8 discos, como a solução para uma torre de 7 discos ajudaria no problema?



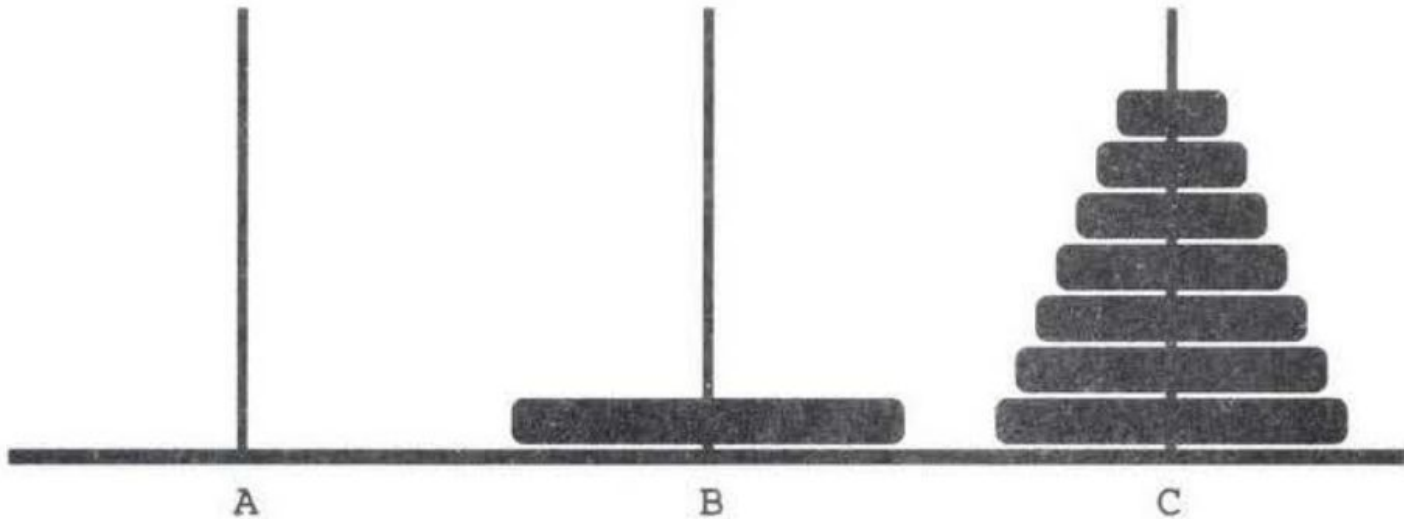
# A Torre de Hanoi: a decomposição recursiva

- 1º passo: mova a pilha de 7 discos de A (origem) para C (temporário):



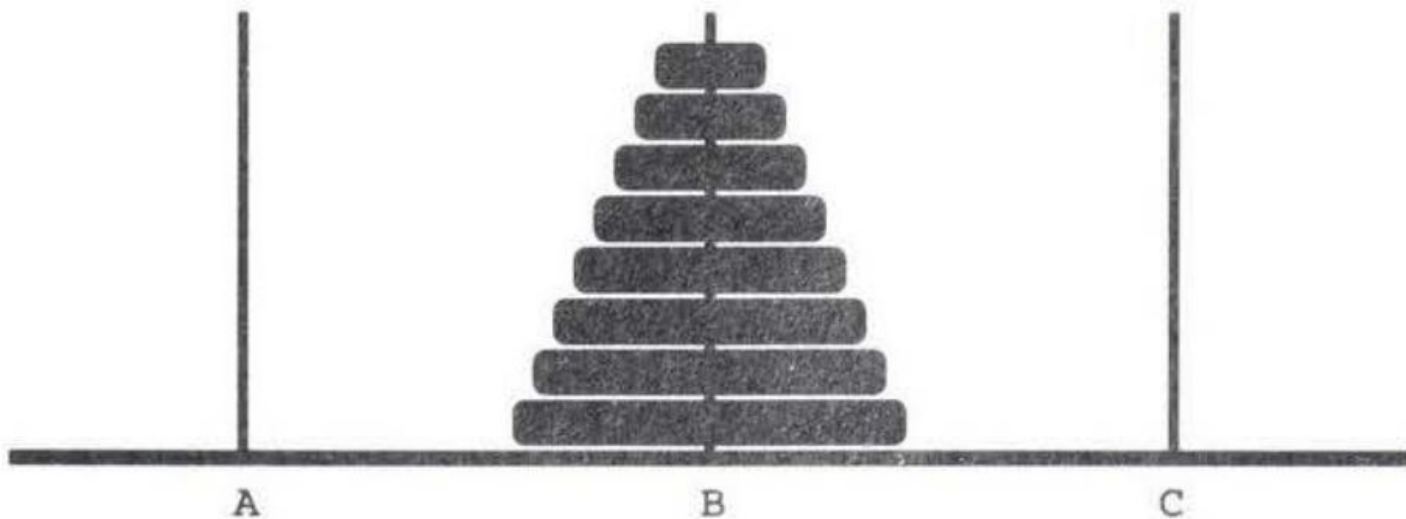
# A Torre de Hanoi: a decomposição recursiva

- 1º passo: mova a pilha de 7 discos de A (origem) para C (temporário);
- 2º passo: mova o único disco de A (origem) para B (destino):



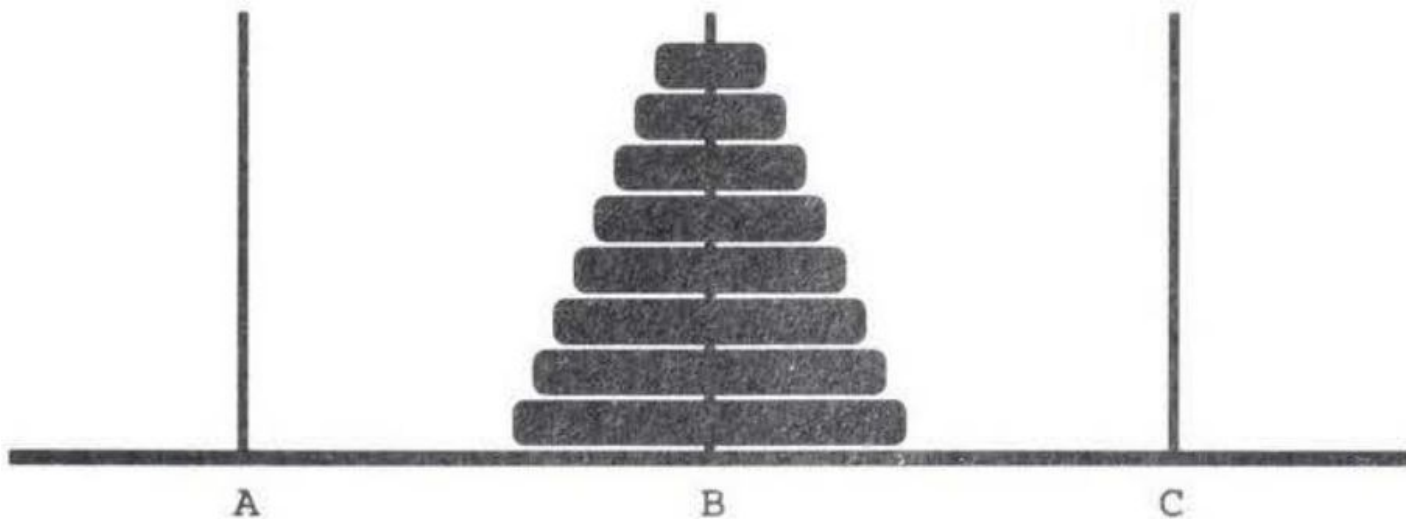
# A Torre de Hanoi: a decomposição recursiva

- 1º passo: mova a pilha de 7 discos de A (origem) para C (temporário);
- 2º passo: mova o único disco de A (origem) para B (destino); e
- 3º passo: mova a pilha de 7 discos de C (temporário) para B (destino):



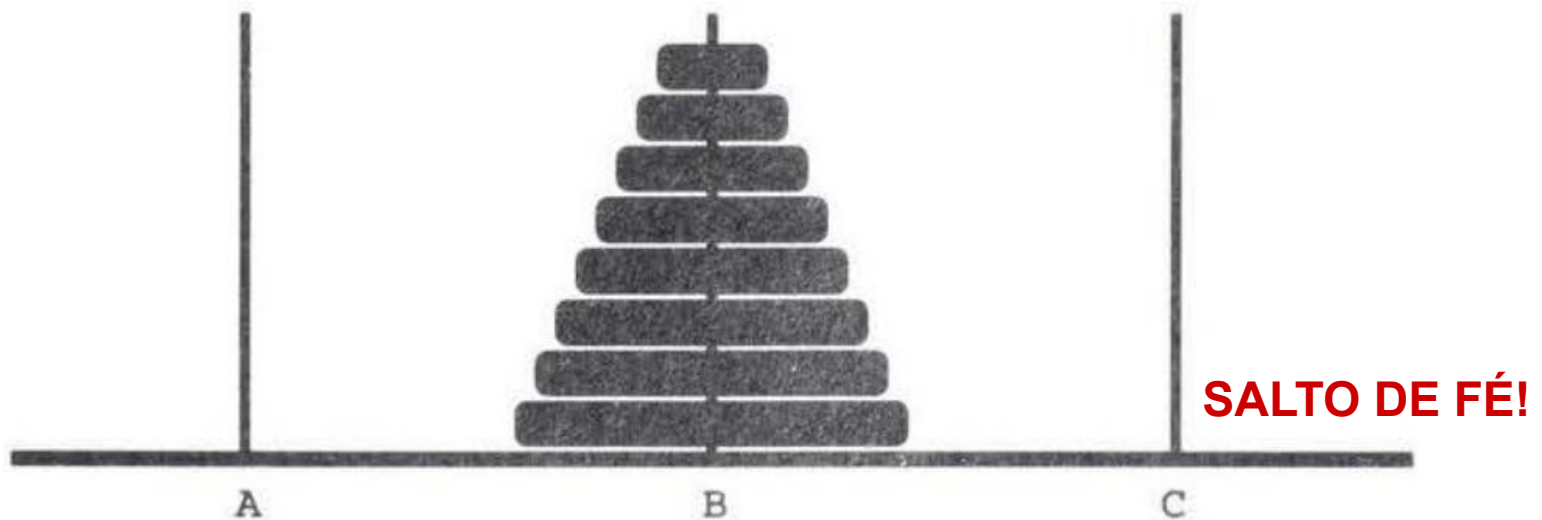
# A Torre de Hanoi: a decomposição recursiva

- É isso, acabou! Você reduziu o problema de mover uma torre de tamanho 8 para o problema de mover uma torre de tamanho 7. Mais ainda: esse procedimento generaliza para torres de tamanho  $N$ !



# A Torre de Hanoi: a decomposição recursiva

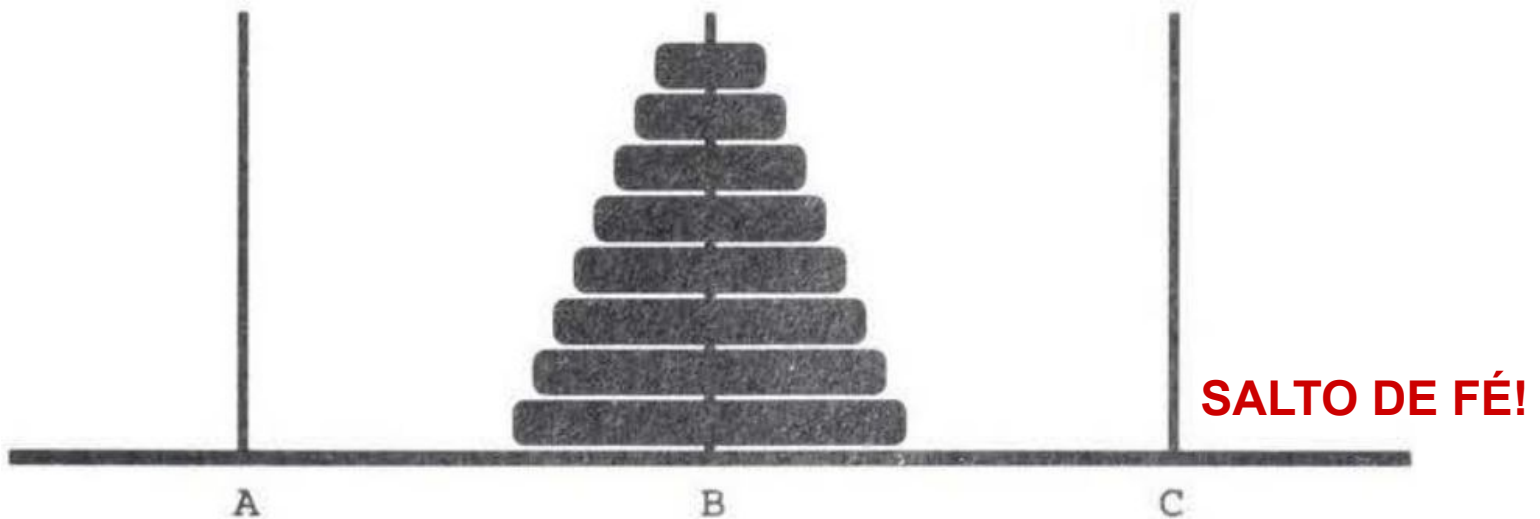
- 1º passo: mova a pilha de  $N-1$  discos de A (origem) para C (temporário);
- 2º passo: mova o único disco de A (origem) para B (destino); 3
- 3º passo: mova a pilha de  $N-1$  discos de C (temporário) para B (destino):





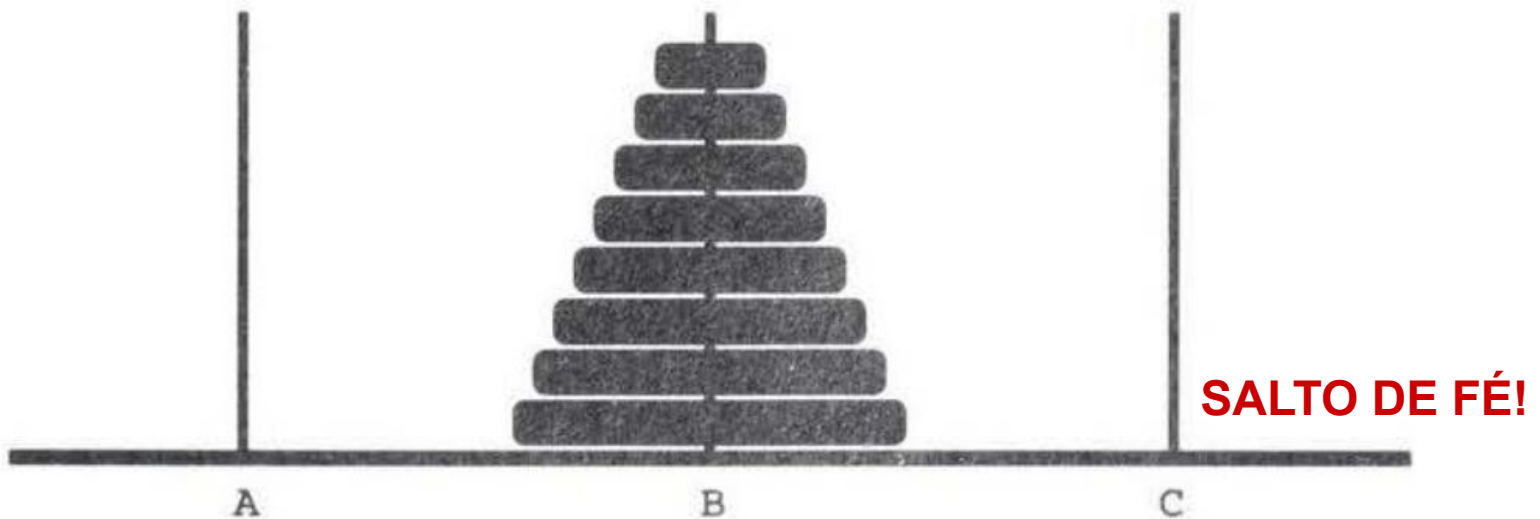
# A Torre de Hanoi: a decomposição recursiva

- OK, e como eu movo uma pilha de  $N-1$  discos?
- Ué, movendo uma pilha de  $N-2$  discos!



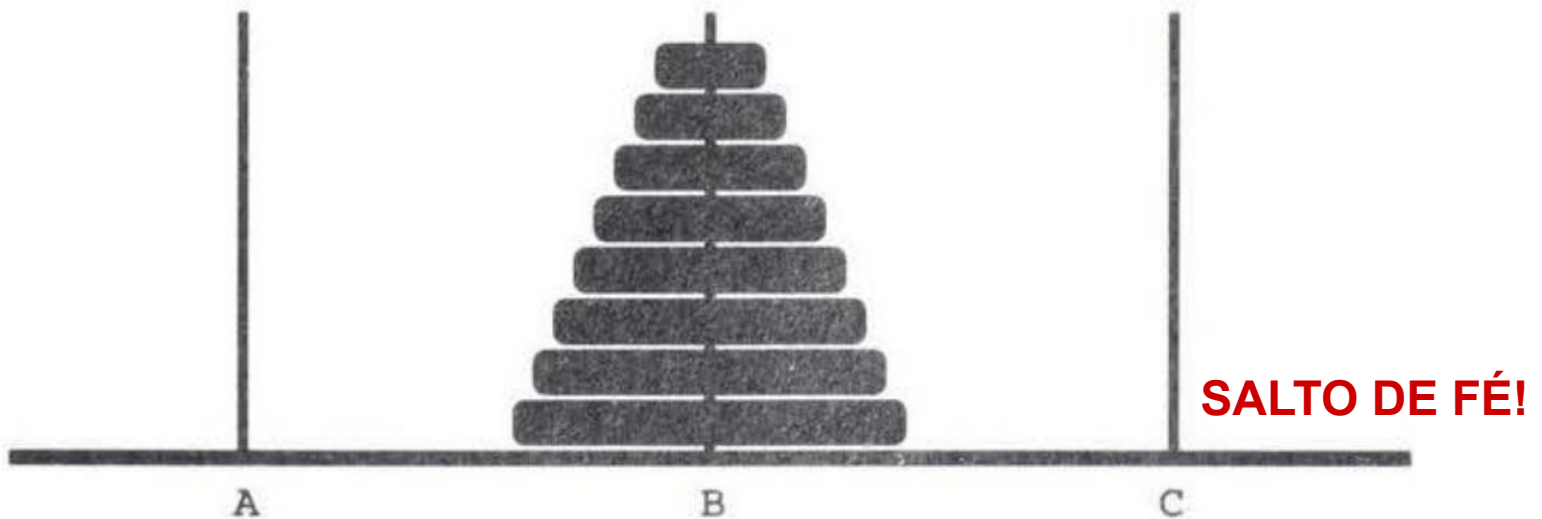
# A Torre de Hanoi: a decomposição recursiva

- OK, e como eu movo uma pilha de  $N-2$  discos?
- Ué, movendo uma pilha de  $N-3$  discos! (... e assim por diante ...)



# A Torre de Hanoi: o salto de fé recursivo

- **NÃO FAÇA PERGUNTAS, ACREDITE NO SALTO DE FÉ RECURSIVO!**
  - Você identificou o caso simples!
  - Você identificou a decomposição recursiva!



# A Torre de Hanoi: o salto de fé recursivo

```
void hanoi (int n, char origem, char destino, char temp)
{
    if (n == 1)
        // Mover o único disco da ORIGEM para o DESTINO;
    else
    {
        // Mover a torre com N-1 discos da ORIGEM para o TEMP;
        // Mover o único disco da ORIGEM para o DESTINO;
        // Mover a torre com N-1 discos do TEMP para o DESTINO.
    }
}
```

# A Torre de Hanoi: o salto de fé é válido?

- Existe um pequeno detalhe aqui: a decomposição recursiva realmente quebra o problema em menor com a **mesma forma do original**?
  - No problema original a torre B e a torre C estavam vazias;
  - Quando você moveu os  $N-1$  discos, deixou 1 disco na torre A.
- Essa pequenina diferença não invalidaria o salto de fé?
- Aqui você tem que pensar no subproblema recursivo de acordo com as regras do jogo: se a decomposição recursiva não quebrar as regras do jogo, ela é válida sim.
  - Mover 1 disco de cada vez: OK
  - Não colocar um disco maior sobre um menor: OK (convença-se de que isso é verdade)

# A Torre de Hanoi: implementando o código

- Nosso procedimento já é capaz de mover os N-1 discos. Precisamos somente de alguma coisa para mover o único disco:

```
void hanoi (int n, char origem, char destino, char temp)
{
    if (n == 1)
        // Mover o único disco da ORIGEM para o DESTINO;
    else
    {
        hanoi(. . .);
        // Mover o único disco da ORIGEM para o DESTINO;
        hanoi(. . .);
    }
}
```

# A Torre de Hanoi: implementando o código

```
void mover_disco (char origem, char destino)
{
    printf("    De %c para %c\n", origem, destino);
}
```

```
void hanoi (int n, char origem, char destino, char temp)
{
    if (n == 1)
        mover_disco(origem, destino);
    else
    {
        hanoi(n - 1, origem, temp, destino);
        mover_disco(origem, destino);
        hanoi(n - 1, temp, destino, origem);
    }
}
```



# A Torre de Hanoi: implementando o código

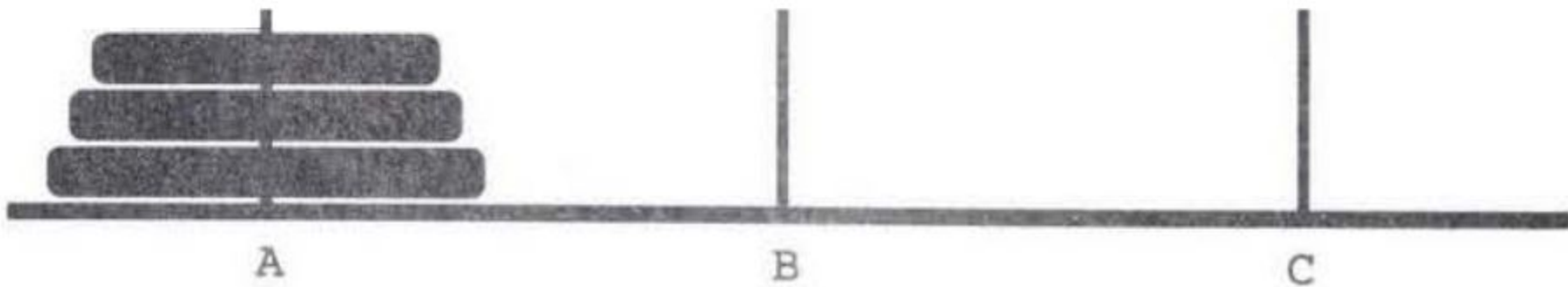
```
void mover_disco (char origem, char destino)
{
    printf("    De %c para %c\n", origem, destino);
}
```

```
void hanoi (int n, char origem, char destino, char temp)
{
    if (n == 1)
        mover_disco(origem, destino);
    else
    {
        hanoi(n - 1, origem, temp, destino); ←
        mover_disco(origem, destino);
        hanoi(n - 1, temp, destino, origem); ←
    }
}
```

**Por que a ordem está trocada?**



A Torre de Hanoi: rastreando o processo (péssima idéia!)




# A Torre de Hanoi: rastreando o processo (péssima idéia!)

```
hanoi(3, 'A', 'B', 'C');
```

hanoi



```
void hanoi (int n, char origem, char destino, char temp)
{
   if (n == 1)
      mover_disco(origem, destino);
  else
  {
      hanoi(n - 1, origem, temp, destino);
      mover_disco(origem, destino);
      hanoi(n - 1, temp, destino, origem);
  }
}
```


# A Torre de Hanoi: rastreando o processo (péssima idéia!)

```
hanoi(3, 'A', 'B', 'C');
```

hanoi

n	origem	destino	temp
3	A	B	C

```
void hanoi (int n, char origem, char destino, char temp)
{
    if (n == 1)
        mover_disco(origem, destino);
    else
    {
        hanoi(n - 1, origem, temp, destino);
        mover_disco(origem, destino);
        hanoi(n - 1, temp, destino, origem);
    }
}
```



# A Torre de Hanoi: rastreando o processo (péssima idéia!)

```
hanoi(2, 'A', 'C', 'B');
```

hanoi

hanoi

n	origem	destino	temp
2	A	C	B

```
void hanoi (int n, char origem, char destino, char temp)
{
  → if (n == 1)
      mover_disco(origem, destino);
  else
  {
    hanoi(n - 1, origem, temp, destino);
    mover_disco(origem, destino);
    hanoi(n - 1, temp, destino, origem);
  }
}
```

# A Torre de Hanoi: rastreando o processo (péssima idéia!)


```
hanoi(2, 'A', 'C', 'B');
```

hanoi

hanoi

n	origem	destino	temp
2	A	C	B

```
void hanoi (int n, char origem, char destino, char temp)
{
    if (n == 1)
        mover_disco(origem, destino);
    else
    {
        hanoi(n - 1, origem, temp, destino);
        mover_disco(origem, destino);
        hanoi(n - 1, temp, destino, origem);
    }
}
```



# A Torre de Hanoi: rastreando o processo (péssima idéia!)

```
hanoi(1, 'A', 'B', 'C');
```

hanoi

hanoi

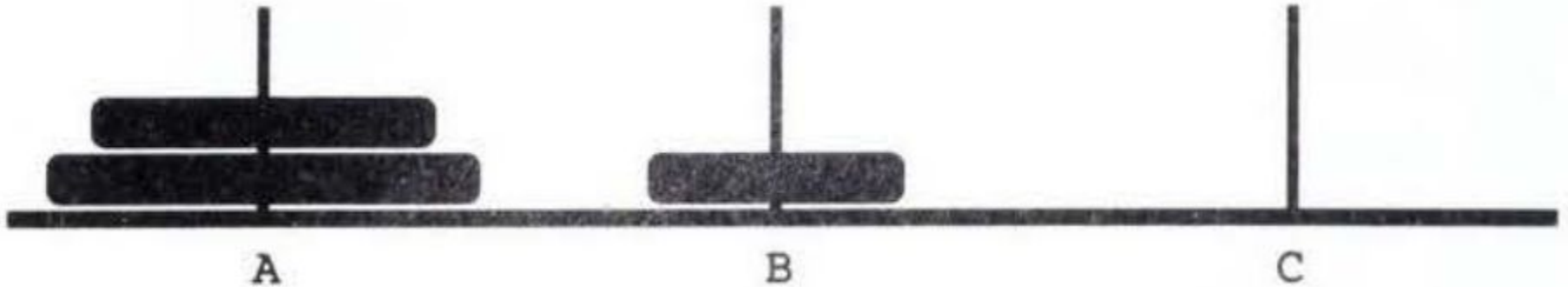
hanoi

n	origem	destino	temp
1	A	B	C

```
void hanoi (int n, char origem, char destino, char temp)
{
  → if (n == 1)
      mover_disco(origem, destino);
  else
  {
    hanoi(n - 1, origem, temp, destino);
    mover_disco(origem, destino);
    hanoi(n - 1, temp, destino, origem);
  }
}
```

A Torre de Hanoi: rastreando o processo (péssima idéia!)

```
hanoi(1, 'A', 'B', 'C');
```



# A Torre de Hanoi: rastreando o processo (péssima idéia!)

```
hanoi(1, 'A', 'B', 'C');
```

hanoi

hanoi

hanoi

n	origem	destino	temp
1	A	B	C

```
void hanoi (int n, char origem, char destino, char temp)
{
    if (n == 1)
        mover_disco(origem, destino);
    else
    {
        hanoi(n - 1, origem, temp, destino);
        mover_disco(origem, destino);
        hanoi(n - 1, temp, destino, origem);
    }
}
```

Terminou!



# A Torre de Hanoi: rastreando o processo (péssima idéia!)


```
hanoi(2, 'A', 'C', 'B');
```

hanoi

hanoi

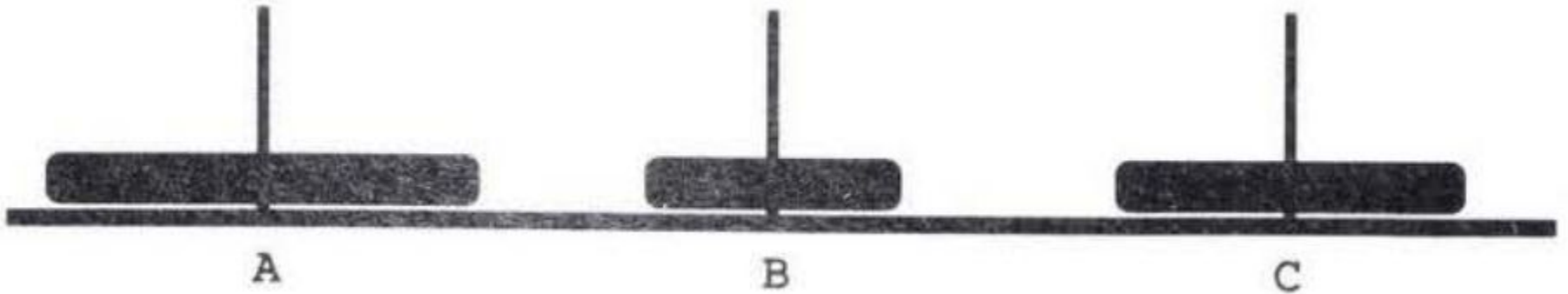
n	origem	destino	temp
2	A	C	B

```
void hanoi (int n, char origem, char destino, char temp)
{
    if (n == 1)
        mover_disco(origem, destino);
    else
    {
        hanoi(n - 1, origem, temp, destino);
        mover_disco(origem, destino);
        hanoi(n - 1, temp, destino, origem);
    }
}
```



# A Torre de Hanoi: rastreando o processo (má idéia!)

```
hanoi(2, 'A', 'C', 'B');
```



# A Torre de Hanoi: rastreando o processo (péssima idéia!)


```
hanoi(2, 'A', 'C', 'B');
```

hanoi

hanoi

n	origem	destino	temp
2	A	C	B

```
void hanoi (int n, char origem, char destino, char temp)
{
    if (n == 1)
        mover_disco(origem, destino);
    else
    {
        hanoi(n - 1, origem, temp, destino);
        mover_disco(origem, destino);
        hanoi(n - 1, temp, destino, origem);
    }
}
```



# A Torre de Hanoi: rastreando o processo (péssima idéia!)


```
hanoi(1, 'B', 'C', 'A');
```

hanoi

hanoi

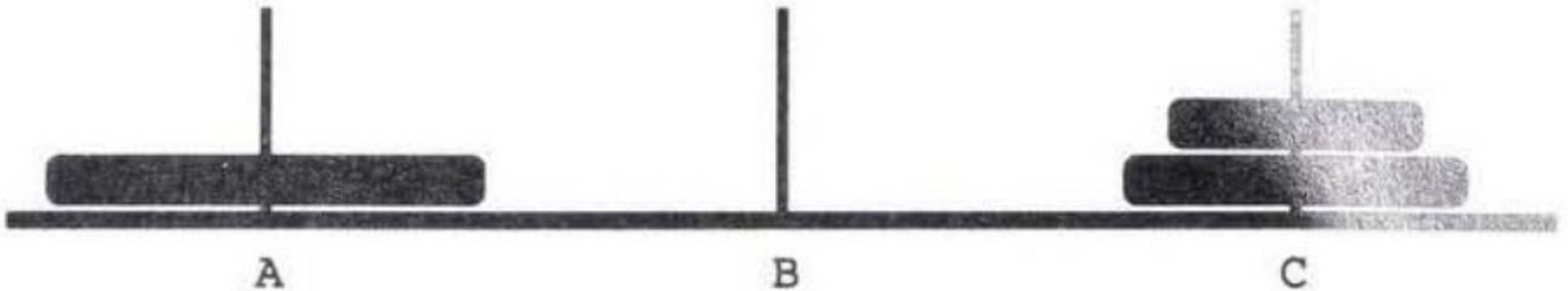
hanoi

n	origem	destino	temp
1	B	C	A

```
void hanoi (int n, char origem, char destino, char temp)
{
   if (n == 1)
    mover_disco(origem, destino);
  else
  {
    hanoi(n - 1, origem, temp, destino);
    mover_disco(origem, destino);
    hanoi(n - 1, temp, destino, origem);
  }
}
```

# A Torre de Hanoi: rastreando o processo (má idéia!)

```
hanoi(1, 'B', 'C', 'A');
```



# A Torre de Hanoi: rastreando o processo (péssima idéia!)

```
hanoi(1, 'B', 'C', 'A');
```

hanoi

hanoi

hanoi

n	origem	destino	temp
1	B	C	A

```
void hanoi (int n, char origem, char destino, char temp)
{
    if (n == 1)
        mover_disco(origem, destino);
    else
    {
        hanoi(n - 1, origem, temp, destino);
        mover_disco(origem, destino);
        hanoi(n - 1, temp, destino, origem);
    }
}
```

Terminou!

# A Torre de Hanoi: rastreando o processo (péssima idéia!)

```
hanoi(1, 'B', 'C', 'A');
```

hanoi

hanoi

n	origem	destino	temp
2	A	C	B

```
void hanoi (int n, char origem, char destino, char temp)
{
    if (n == 1)
        mover_disco(origem, destino);
    else
    {
        hanoi(n - 1, origem, temp, destino);
        mover_disco(origem, destino);
        hanoi(n - 1, temp, destino, origem);
    }
}
```

Terminou!


# A Torre de Hanoi: rastreando o processo (péssima idéia!)

```
hanoi(3, 'A', 'B', 'C');
```

hanoi

n	origem	destino	temp
3	A	B	C

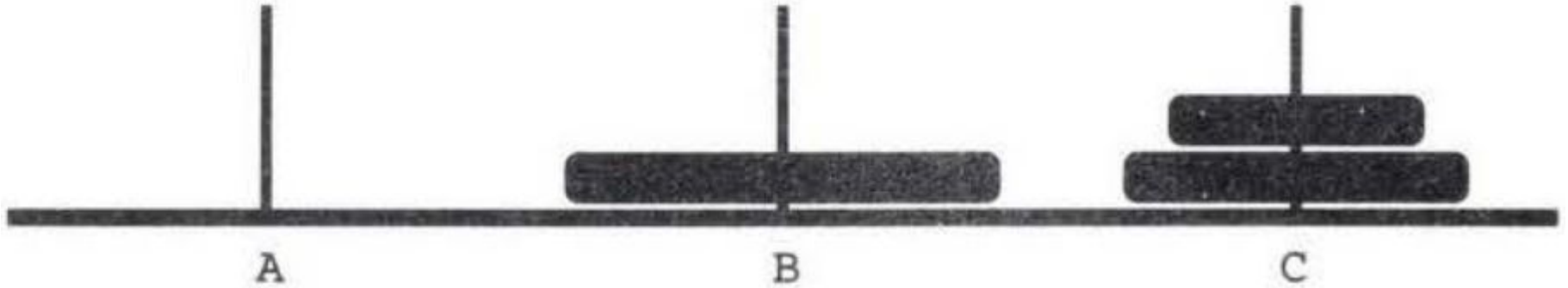
```
void hanoi (int n, char origem, char destino, char temp)
{
    if (n == 1)
        mover_disco(origem, destino);
    else
    {
        hanoi(n - 1, origem, temp, destino);
        mover_disco(origem, destino);
        hanoi(n - 1, temp, destino, origem);
    }
}
```





# A Torre de Hanoi: rastreando o processo (má idéia!)

```
hanoi(3, 'A', 'B', 'C');
```




# A Torre de Hanoi: rastreando o processo (péssima idéia!)

```
hanoi(3, 'A', 'B', 'C');
```

hanoi

n	origem	destino	temp
3	A	B	C

```
void hanoi (int n, char origem, char destino, char temp)
{
    if (n == 1)
        mover_disco(origem, destino);
    else
    {
        hanoi(n - 1, origem, temp, destino);
        mover_disco(origem, destino);
        hanoi(n - 1, temp, destino, origem);
    }
}
```




# A Torre de Hanoi: rastreando o processo (péssima idéia!)

```
hanoi(2, 'C', 'B', 'A');
```

hanoi

hanoi

n	origem	destino	temp
2	C	B	A

```
void hanoi (int n, char origem, char destino, char temp)
{
   if (n == 1)
    mover_disco(origem, destino);
  else
  {
    hanoi(n - 1, origem, temp, destino);
    mover_disco(origem, destino);
    hanoi(n - 1, temp, destino, origem);
  }
}
```

# A Torre de Hanoi: rastreando o processo (péssima idéia!)


```
hanoi(2, 'C', 'B', 'A');
```

hanoi

hanoi

n	origem	destino	temp
2	C	B	A

```
void hanoi (int n, char origem, char destino, char temp)
{
    if (n == 1)
        mover_disco(origem, destino);
    else
    {
        hanoi(n - 1, origem, temp, destino);
        mover_disco(origem, destino);
        hanoi(n - 1, temp, destino, origem);
    }
}
```



# A Torre de Hanoi: rastreando o processo (péssima idéia!)

```
hanoi(1, 'C', 'A', 'B');
```

hanoi

hanoi

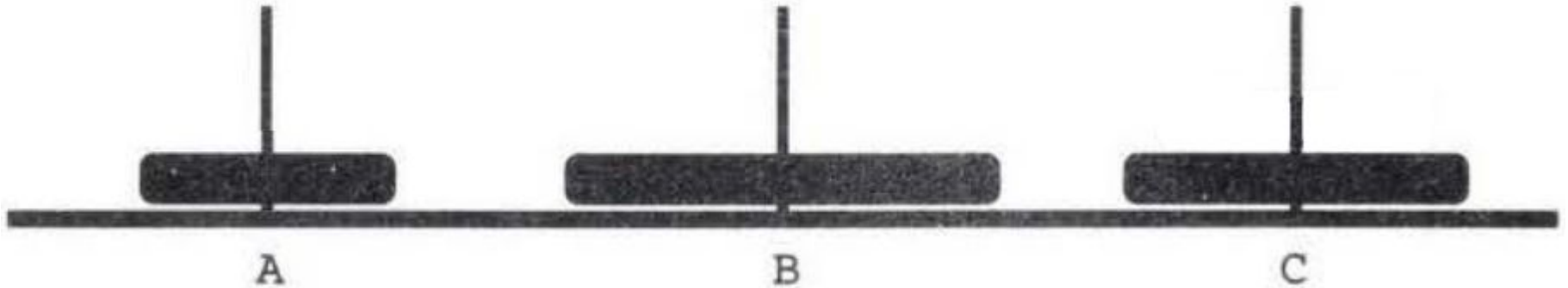
hanoi

n	origem	destino	temp
1	C	A	B

```
void hanoi (int n, char origem, char destino, char temp)
{
  → if (n == 1)
      mover_disco(origem, destino);
  else
  {
    hanoi(n - 1, origem, temp, destino);
    mover_disco(origem, destino);
    hanoi(n - 1, temp, destino, origem);
  }
}
```

# A Torre de Hanoi: rastreando o processo (má idéia!)

```
hanoi(1, 'C', 'A', 'B');
```



# A Torre de Hanoi: rastreando o processo (péssima idéia!)

```
hanoi(1, 'C', 'A', 'B');
```

hanoi

hanoi

hanoi

n	origem	destino	temp
1	C	A	B

```
void hanoi (int n, char origem, char destino, char temp)
{
    if (n == 1)
        mover_disco(origem, destino);
    else
    {
        hanoi(n - 1, origem, temp, destino);
        mover_disco(origem, destino);
        hanoi(n - 1, temp, destino, origem);
    }
}
```

Terminou!


# A Torre de Hanoi: rastreando o processo (péssima idéia!)

```
hanoi(2, 'C', 'B', 'A');
```

hanoi

hanoi

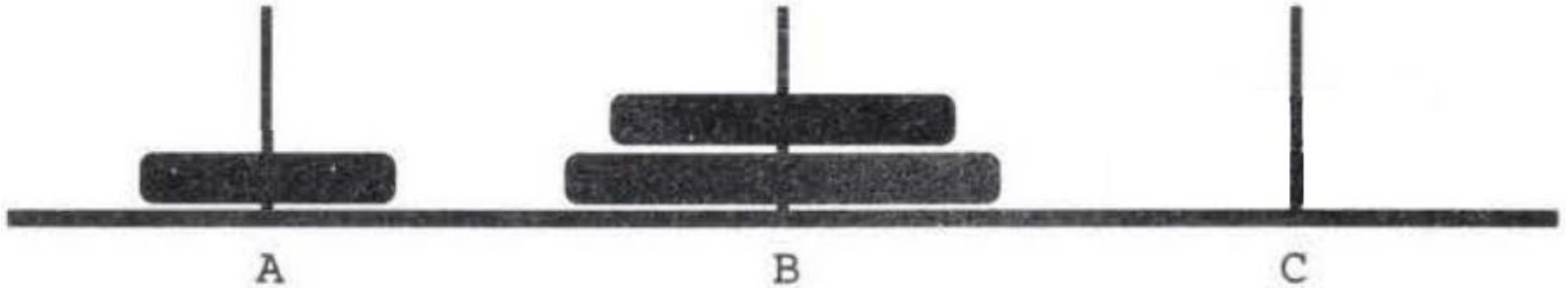
n	origem	destino	temp
2	C	B	A

```
void hanoi (int n, char origem, char destino, char temp)
{
    if (n == 1)
        mover_disco(origem, destino);
    else
    {
        hanoi(n - 1, origem, temp, destino);
         mover_disco(origem, destino);
        hanoi(n - 1, temp, destino, origem);
    }
}
```



# A Torre de Hanoi: rastreando o processo (má idéia!)

```
hanoi(2, 'C', 'B', 'A');
```



# A Torre de Hanoi: rastreando o processo (péssima idéia!)

```
hanoi(2, 'C', 'B', 'A');
```

hanoi

hanoi

n	origem	destino	temp
2	C	B	A

```
void hanoi (int n, char origem, char destino, char temp)
{
    if (n == 1)
        mover_disco(origem, destino);
    else
    {
        hanoi(n - 1, origem, temp, destino);
        mover_disco(origem, destino);
        hanoi(n - 1, temp, destino, origem);
    }
}
```

# A Torre de Hanoi: rastreando o processo (péssima idéia!)

```
hanoi(1, 'A', 'B', 'C');
```

hanoi

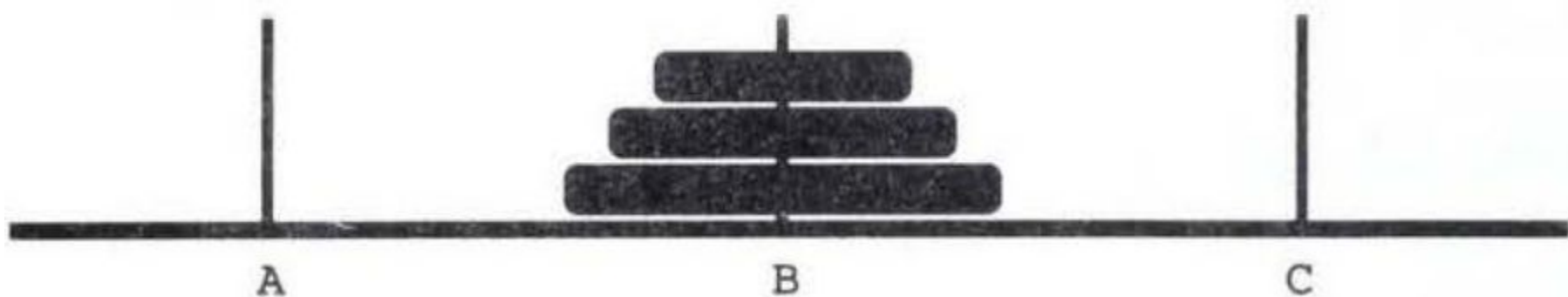
hanoi

hanoi

n	origem	destino	temp
1	A	B	C

```
void hanoi (int n, char origem, char destino, char temp)
{
  → if (n == 1)
      mover_disco(origem, destino);
  else
  {
    hanoi(n - 1, origem, temp, destino);
    mover_disco(origem, destino);
    hanoi(n - 1, temp, destino, origem);
  }
}
```

# A Torre de Hanoi: rastreando o processo (má idéia!)



# A Torre de Hanoi: rastreando o processo (péssima idéia!)

```
hanoi(1, 'A', 'B', 'C');
```

hanoi

hanoi

hanoi

n	origem	destino	temp
1	A	B	C

```
void hanoi (int n, char origem, char destino, char temp)
{
    if (n == 1)
        mover_disco(origem, destino);
    else
    {
        hanoi(n - 1, origem, temp, destino);
        mover_disco(origem, destino);
        hanoi(n - 1, temp, destino, origem);
    }
}
```

Terminou!

# A Torre de Hanoi: rastreando o processo (péssima idéia!)

hanoi

hanoi

n

origem

destino

temp

2

C

B

A

```
void hanoi (int n, char origem, char destino, char temp)
{
    if (n == 1)
        mover_disco(origem, destino);
    else
    {
        hanoi(n - 1, origem, temp, destino);
        mover_disco(origem, destino);
        hanoi(n - 1, temp, destino, origem);
    }
}
```

Terminou!

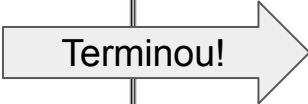
# A Torre de Hanoi: rastreando o processo (péssima idéia!)

hanoi

n	origem	destino	temp
3	A	B	C

```
void hanoi (int n, char origem, char destino, char temp)
{
    if (n == 1)
        mover_disco(origem, destino);
    else
    {
        hanoi(n - 1, origem, temp, destino);
        mover_disco(origem, destino);
        hanoi(n - 1, temp, destino, origem);
    }
}
```

Terminou!



# A Torre de Hanoi: rastreando o processo (péssima idéia!)

FIM DO PROCEDIMENTO



# A Torre de Hanoi: rastreando o processo (péssima idéia!)

- O rastreamento do stack em funções recursivas pode se tornar muito complexo ao ponto de deixar tudo mais confuso ao invés de ajudar a entender a recursividade.
- É ESSENCIAL que você aprenda a CONFIAR NO SALTO DE FÉ RECURSIVO e pense no algoritmo recursivo como algo holístico, como uma operação única.

# Permutações

- Em muitos problemas computacionais é importante ser capaz de gerar todas as permutações de um determinado conjunto. Por exemplo, podemos gerar todos os anagramas de uma determinada palavra.
- A geração de permutação é um problema que pode ser resolvido de forma recursiva. Considere que você quer criar um procedimento para gerar todas as permutações da string "ABC":

```
void permutar (char str[]);
```

- Quais os casos simples?
- Qual a decomposição recursiva?

```
permutar("ABC");
```

Só para ilustrar, é um erro!

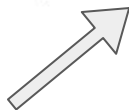
ABC  
ACB  
BAC  
BCA  
CAB  
CBA

# Permutações

- Considere o caso concreto de permutar a string “ABCDE”. Na definição da decomposição recursiva e dos casos simples, **aplique o salto de fé para gerar todas as permutações de qualquer string menor. Apenas assumam que a chamada recursiva funciona, aceite e viva com isso.**

```
permutar("ABC");
```

Só para ilustrar, é um erro!



```
ABC  
ACB  
BAC  
BCA  
CAB  
CBA
```

# Permutações

- Considere o caso concreto de permutar a string “ABCDE”. Na definição da decomposição recursiva e dos casos simples, **aplique o salto de fé para gerar todas as permutações de qualquer string menor. Apenas assuma que a chamada recursiva funciona, aceite e viva com isso.**

`permutar( "ABCDE" );` ← Só para ilustrar, é um erro!

```
// 'A' seguida de todas as permutações de "BCDE"  
// 'B' seguida de todas as permutações de "ACDE"  
// 'C' seguida de todas as permutações de "ABDE"  
// 'D' seguida de todas as permutações de "ABCE"  
// 'E' seguida de todas as permutações de "ABCD"
```

# Permutações

- De modo geral: para mostrar as permutações de tamanho  $N$ , temos que pegar cada um dos  $n$  caracteres, e mostrar esse caractere seguido por todas as permutações restantes de  $n-1$  caracteres.
- Dificuldade: **O SUBPROBLEMA RECURSIVO NÃO TEM A MESMA FORMA QUE O PROBLEMA ORIGINAL:**
  - Problema original: **mostrar todas as permutações de uma string**
  - Subproblema recursivo: mostrar um caractere de uma string seguido por todas as permutações das letras restantes, ou seja: **gerar todas as permutações de uma string deixando alguns caracteres no início da string fixos em suas posições.**
- **Se o subproblema recursivo não tem a mesma forma do problema original, NÃO podemos usar a recursão.** Será que tem algum jeito?

# Permutações

- A solução é usar um wrapper e modificar ligeiramente o problema a ser resolvido! Nós tornamos o procedimento `permutar` em um wrapper, que **chama um procedimento recursivo que resolve um problema ligeiramente diferente, só que mais geral:**

```
void permutar (char str[]);  
static void realizar_permutacao (char str[], int k);
```

- O procedimento `realizar_permutacao` gera **todas as permutações de uma string com as primeiras k letras fixas**. Quando  $k = 0$ , todas as letras são livres para permutar, o que nos dá o problema original. Quando  $k$  aumenta, o subproblema torna-se mais simples. Quando  $k = \text{tamanho de str}$ , nenhuma letra pode permutar de lugar e a string pode ser exibida como aparece.

# Permutações

```
/**
 * Procedimento: permutar
 * Uso: permutar("string");
 * -----
 * Este procedimento é um wrapper que recebe uma string str (a rigor recebe um
 * ponteiro para um array de char) e chama o procedimento realizar_permutacao.
 * Como, inicialmente, todas as letras podem ser permutadas, este procedimento
 * utiliza o valor 0 (zero) para a quantidade de letras fixas no começo da
 * string.
 */

void permutar (char str[])
{
    realizar_permutacao(str, 0);
}
```

# Permutações

```
static void realizar_permutacao (char str[], int k)
{
    if ( /* k == tamanho da string */ )
        // mostrar a string
    else
    {
        // para cada caractere i entre k e o final da string:
        {
            // troque os caracteres i e k
            // use recursão para gerar permutações com k+1 caracteres fixos
            // restaure a string original trocando novamente os caracteres i e k
        }
    }
}
```



# Permutações

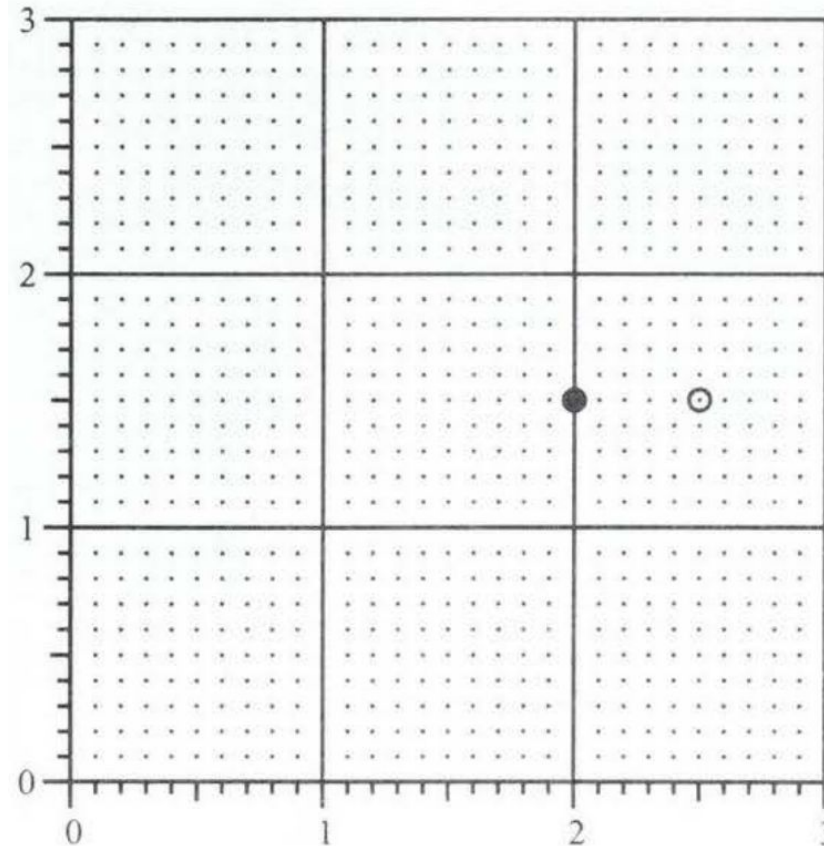
```
static void realizar_permutacao (char str[], int k)
{
    if (k == StringLength(str))
        printf("%s\n", str);
    else
    {
        for (int i = k; i < StringLength(str); i++)
        {
            trocar_letras(str, k, i);
            realizar_permutacao(str, k + 1);
            trocar_letras(str, k, i);
        }
    }
}
```

# Aplicações gráficas da recursividade: `graphics.h`

- Usaremos uma simples biblioteca gráfica, disponível com as bibliotecas que acompanham nosso livro de referência, para gerar desenhos no display utilizando linhas e arcos. A biblioteca é a `graphics.h`.
- Ao inicializar as funções gráficas, a JANELA GRÁFICA é exibida. Os demais subprogramas da biblioteca “desenham” nessa janela.
- A janela gráfica tem um sistema de coordenadas, sendo o ponto de origem -  $O(0,0)$  - localizado no canto inferior esquerdo da janela. O eixo x é horizontal, com sentido crescente para direita, e o eixo y é vertical, com sentido crescente para cima. Usando esses eixos é possível localizar um ponto através das coordenadas x e y:  $P(x, y)$ . Unidade de medida: polegadas.

# Aplicações gráficas da recursividade: `graphics.h`

- As coordenadas podem ser especificadas de 2 formas:
  - **Absoluta**: informa a posição com as coordenadas em relação à origem
  - **Relativa**: informa a posição com base em um deslocamento a partir do último ponto especificado.
- Exemplo:
  - $P1 = (2.0, 1.5)$
  - $P2 = (0.5, 0.0)$  a partir de  $P1$



# Aplicações gráficas da recursividade: `graphics.h`

`InitGraphics()`

This procedure creates the graphics window on the screen. The call to `InitGraphics` must precede any output operations of any kind and is usually the first statement in the function `main`.

`MovePen(x, y)`

This procedure picks up the pen and moves it—without drawing any lines—to the position  $(x, y)$ , which is specified in absolute coordinates.

`GetWindowWidth()`  
`GetWindowHeight()`

These functions return the width and height of the graphics window, respectively.

`GetCurrentX()`  
`GetCurrentY()`

These functions return the absolute coordinates of the current point.

**Não compatíveis com os programas de hoje em dia! Foi criada em 1994!  
Preste atenção às explicações do professor!!!!!!**

# Aplicações gráficas da recursividade: `graphics.h`

`DrawLine(dx, dy)`

This procedure draws a line extending from the current point by moving the pen  $dx$  inches in the  $x$  direction and  $dy$  inches in the  $y$  direction. The final position becomes the new current point.

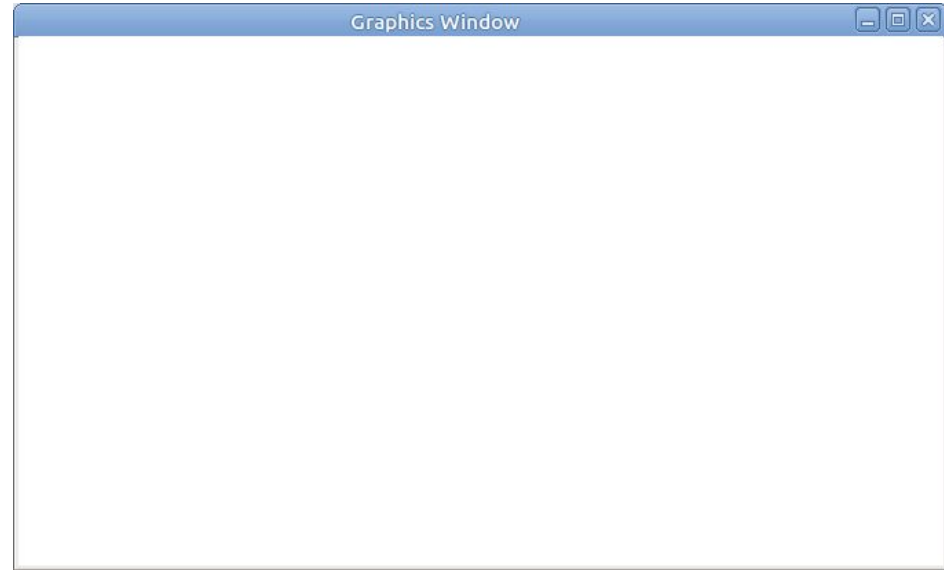
`DrawArc(r, start, sweep)`

This procedure draws a circular arc, which always begins at the current point. The arc itself has radius  $r$ , and starts at the angle specified by the parameter  $start$ , relative to the center of the circle. This angle is measured in degrees counterclockwise from the 3 o'clock position along the  $x$ -axis, as in traditional mathematics. For example, if  $start$  is 0, the arc begins at the 3 o'clock position; if  $start$  is 90, the arc begins at the 12 o'clock position; and so on. The fraction of the circle drawn is specified by the parameter  $sweep$ , which is also measured in degrees. If  $sweep$  is 360, `DrawArc` draws a complete circle; if  $sweep$  is 90, it draws a quarter of a circle. If the value of  $sweep$  is positive, the arc is drawn counterclockwise from the current point; if  $sweep$  is negative, the arc is drawn clockwise. The current point at the end of the `DrawArc` operation is the final position of the pen along the arc.

# Aplicações gráficas da recursividade: `graphics.h`

```
int main (void)
{
    // Inicializa a janela gráfica:
    → InitGraphics();

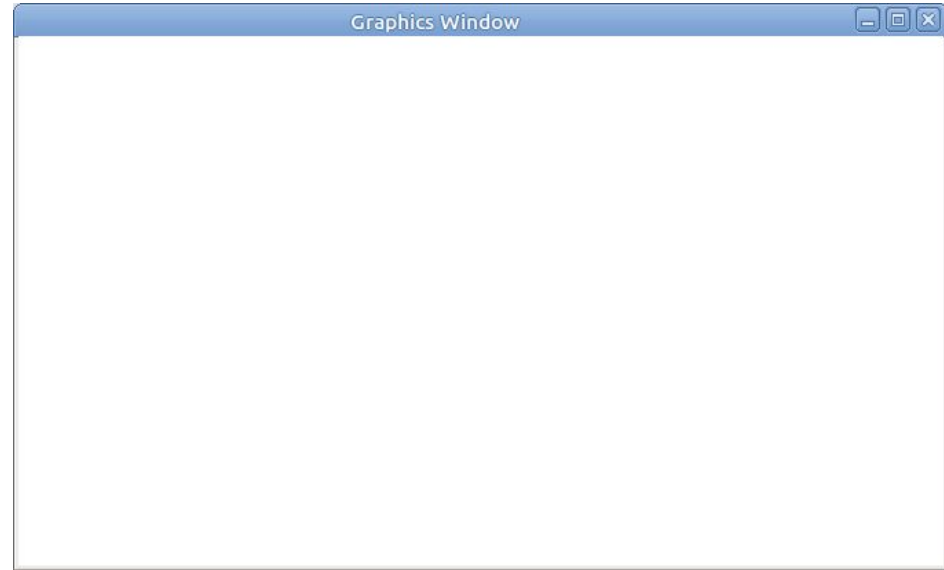
    // Desenha uma janela em arco:
    MovePen(2.0, 0.5);
    DrawLine(1.0, 0.0);
    DrawLine(0.0, 1.0);
    DrawArc(0.5, 0, 180);
    DrawLine(0.0, -1.0);
}
```



# Aplicações gráficas da recursividade: `graphics.h`

```
int main (void)
{
    // Inicializa a janela gráfica:
    InitGraphics();

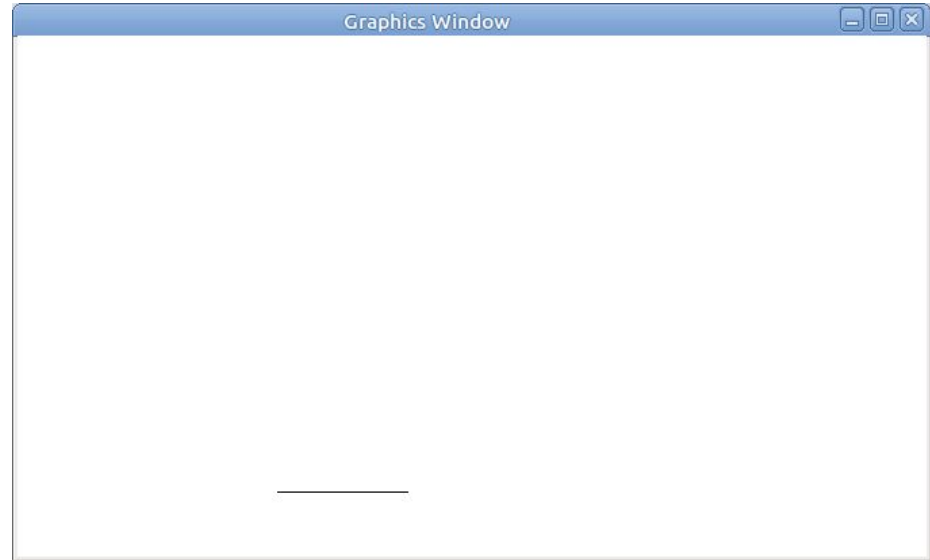
    // Desenha uma janela em arco:
    MovePen(2.0, 0.5);
    DrawLine(1.0, 0.0);
    DrawLine(0.0, 1.0);
    DrawArc(0.5, 0, 180);
    DrawLine(0.0, -1.0);
}
```



# Aplicações gráficas da recursividade: `graphics.h`

```
int main (void)
{
    // Inicializa a janela gráfica:
    InitGraphics();

    // Desenha uma janela em arco:
    MovePen(2.0, 0.5);
    DrawLine(1.0, 0.0);
    DrawLine(0.0, 1.0);
    DrawArc(0.5, 0, 180);
    DrawLine(0.0, -1.0);
}
```

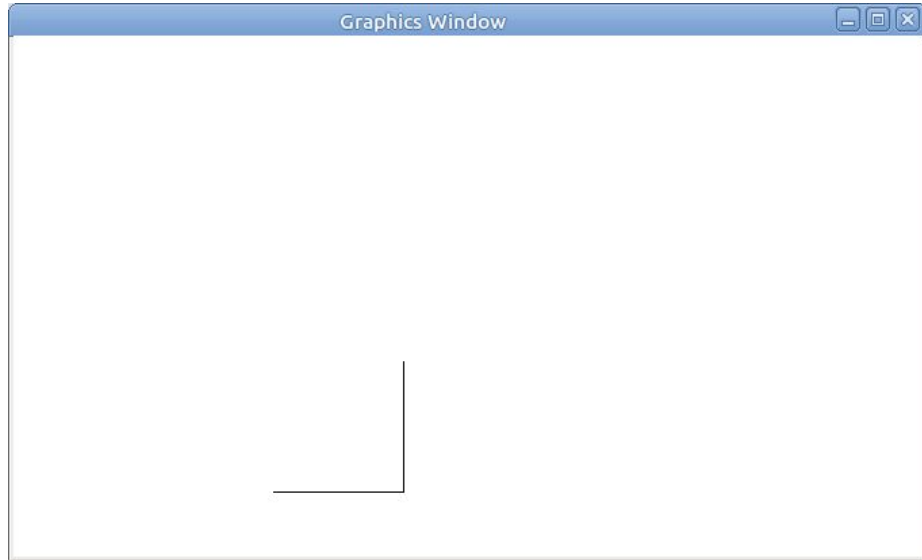




# Aplicações gráficas da recursividade: `graphics.h`

```
int main (void)
{
    // Inicializa a janela gráfica:
    InitGraphics();

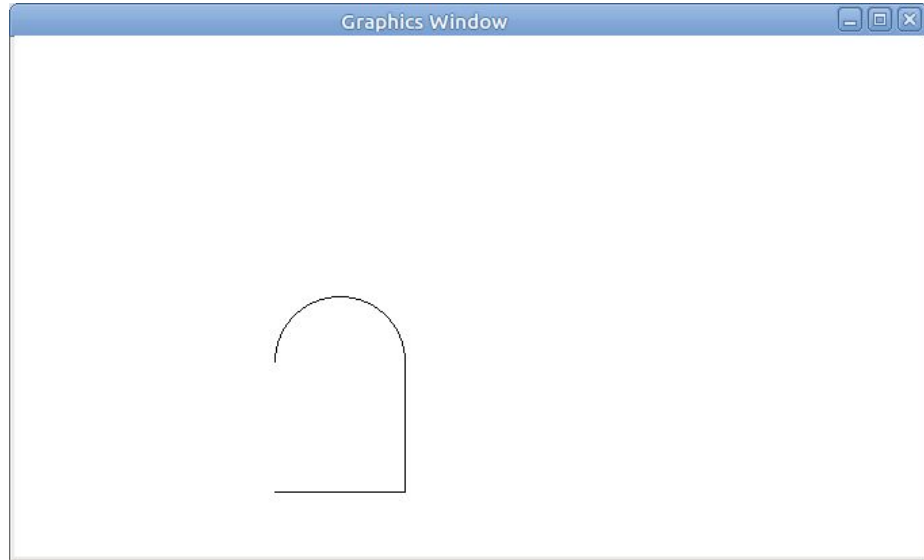
    // Desenha uma janela em arco:
    MovePen(2.0, 0.5);
    DrawLine(1.0, 0.0);
    DrawLine(0.0, 1.0);
    DrawArc(0.5, 0, 180);
    DrawLine(0.0, -1.0);
}
```



# Aplicações gráficas da recursividade: `graphics.h`

```
int main (void)
{
    // Inicializa a janela gráfica:
    InitGraphics();

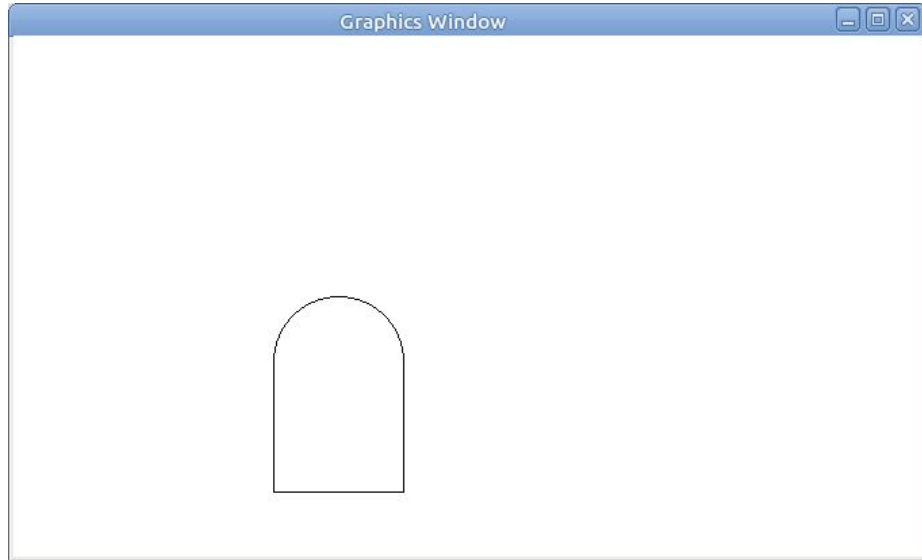
    // Desenha uma janela em arco:
    MovePen(2.0, 0.5);
    DrawLine(1.0, 0.0);
    DrawLine(0.0, 1.0);
    DrawArc(0.5, 0, 180);
    DrawLine(0.0, -1.0);
}
```



# Aplicações gráficas da recursividade: `graphics.h`

```
int main (void)
{
    // Inicializa a janela gráfica:
    InitGraphics();

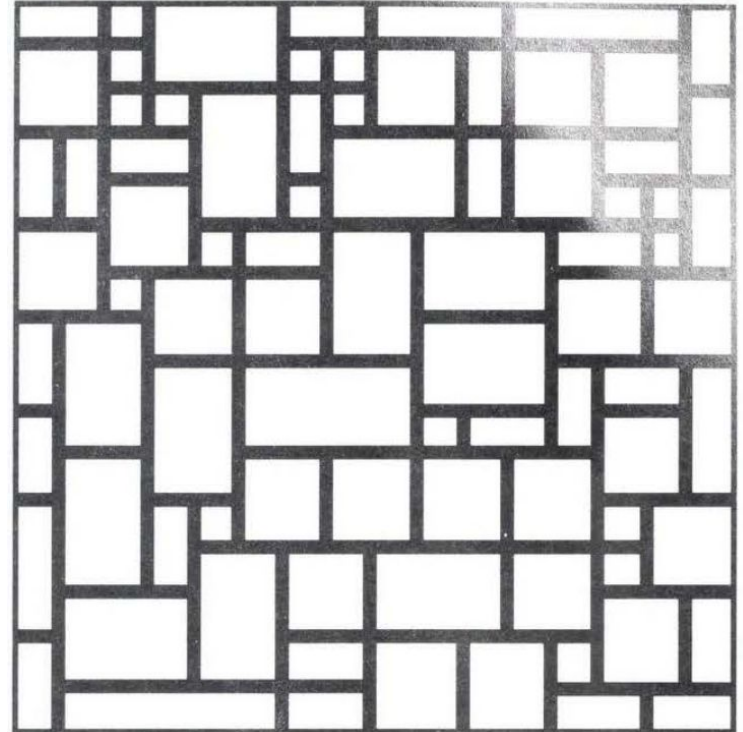
    // Desenha uma janela em arco:
    MovePen(2.0, 0.5);
    DrawLine(1.0, 0.0);
    DrawLine(0.0, 1.0);
    DrawArc(0.5, 0, 180);
    DrawLine(0.0, -1.0);
}
```



# Aplicações gráficas da recursividade: Mondrian

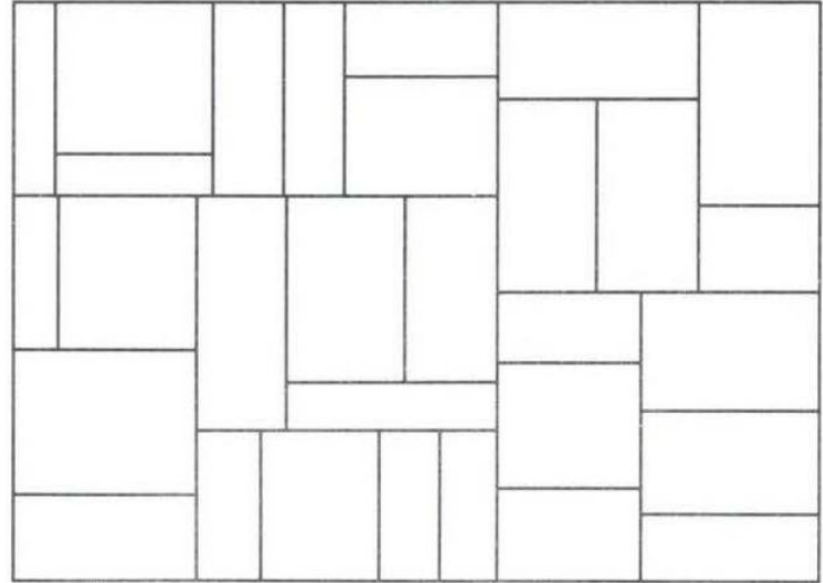
- Cubismo surgiu no início do séc. XX, principalmente por Pablo Picasso e Georges Braque, usavam formas geométricas primitivas.
- Piet Mondrian (1872-1944), influenciado pelo cubismo, produziu muitos quadros baseados apenas em linhas verticais e horizontais.
- Como reproduzir no computador?

Grid pattern from Piet Mondrian, "Composition with Grid 6," 1919



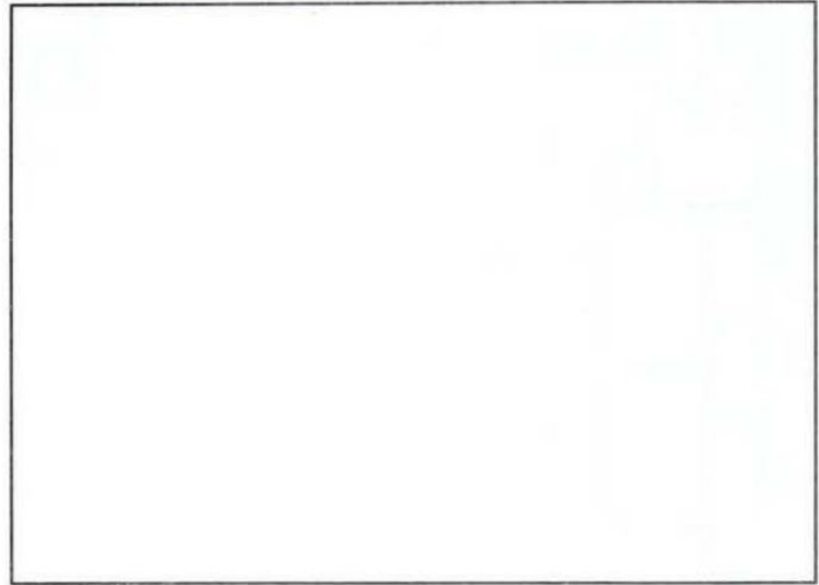
# Aplicações gráficas da recursividade: Mondrian

- Você quer gerar uma figura parecida com a figura ao lado.
- Pense no processo de desenhar esse padrão como um processo de decomposição recursiva!



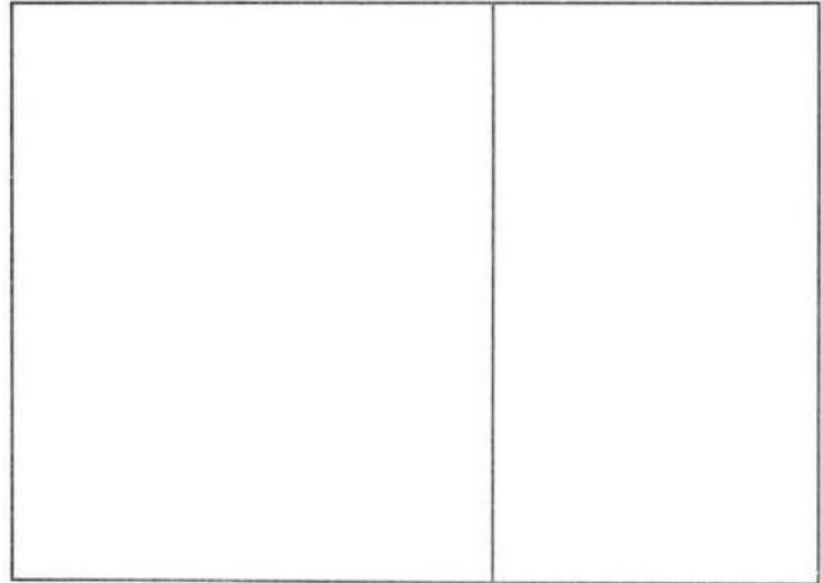
# Aplicações gráficas da recursividade: Mondrian

- Inicialmente temos apenas um retângulo vazio.
- Podemos dividir esse retângulo com uma linha horizontal ou vertical.



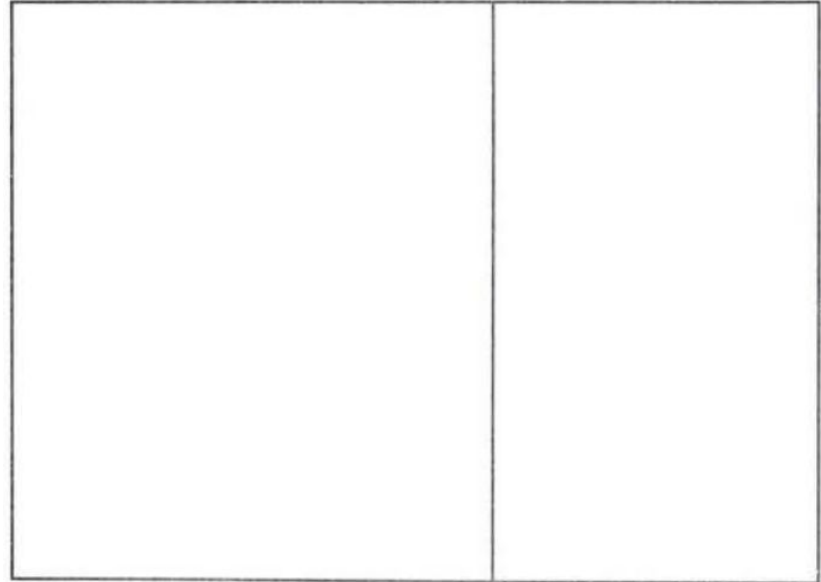
# Aplicações gráficas da recursividade: Mondrian

- Como o retângulo tem largura maior do que a altura, vamos dividir com uma linha vertical:
- Note que agora temos 2 subproblemas menores, os 2 retângulos vazios, da mesma forma que o problema original. Então podemos usar recursividade!
- Basta ir subdividindo esses retângulos, sempre com uma linha vertical à maior dimensão.



# Aplicações gráficas da recursividade: Mondrian

- E qual seria o caso simples? Quando a decomposição recursiva seria encerrada?
- Como é possível dividir um número infinitamente, temos que achar alguma coisa que sinalize que o processo de decomposição já está bom o suficiente. Pode ser:
  - a área de cada retângulo
  - o tamanho do menor lado





# Aplicações gráficas da recursividade: Mondrian

```
#include "genlib.h"
#include "graphics.h"
#include "random.h"
#include "simpio.h"

/* Constantes Simbólicas: */

#define AREA_MINIMA 0.50 // Área do menor retângulo que será dividido
#define LADO_MINIMO 0.15 // Tamanho mínimo de cada lado da divisória

/* Declarações de Subprogramas: */

static void dividir_tela (double x, double y,
                        double largura, double altura);

/* Função Main: */

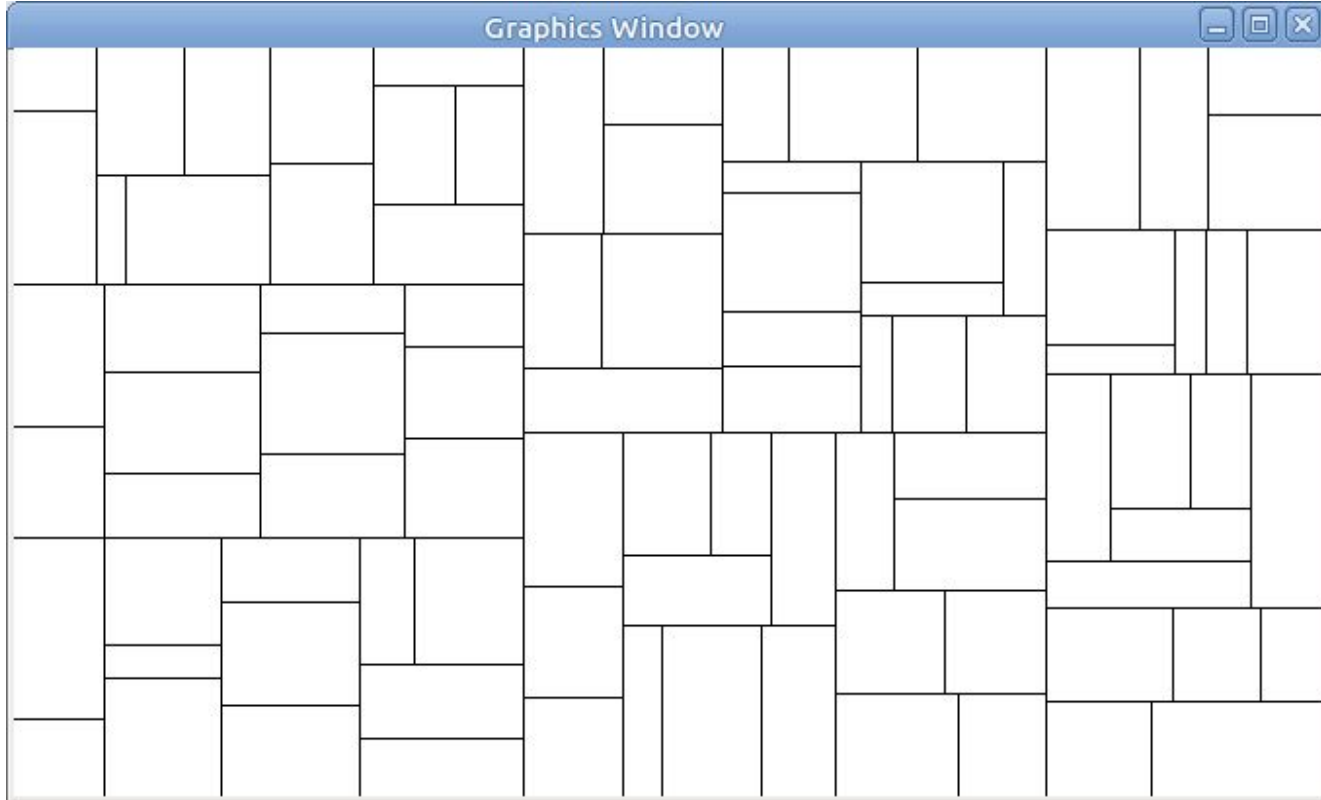
int main (void)
{
    InitGraphics();
    Randomize();
    dividir_tela(0, 0, GetWindowWidth(), GetWindowHeight());
}
```

# Aplicações gráficas da recursividade: Mondrian

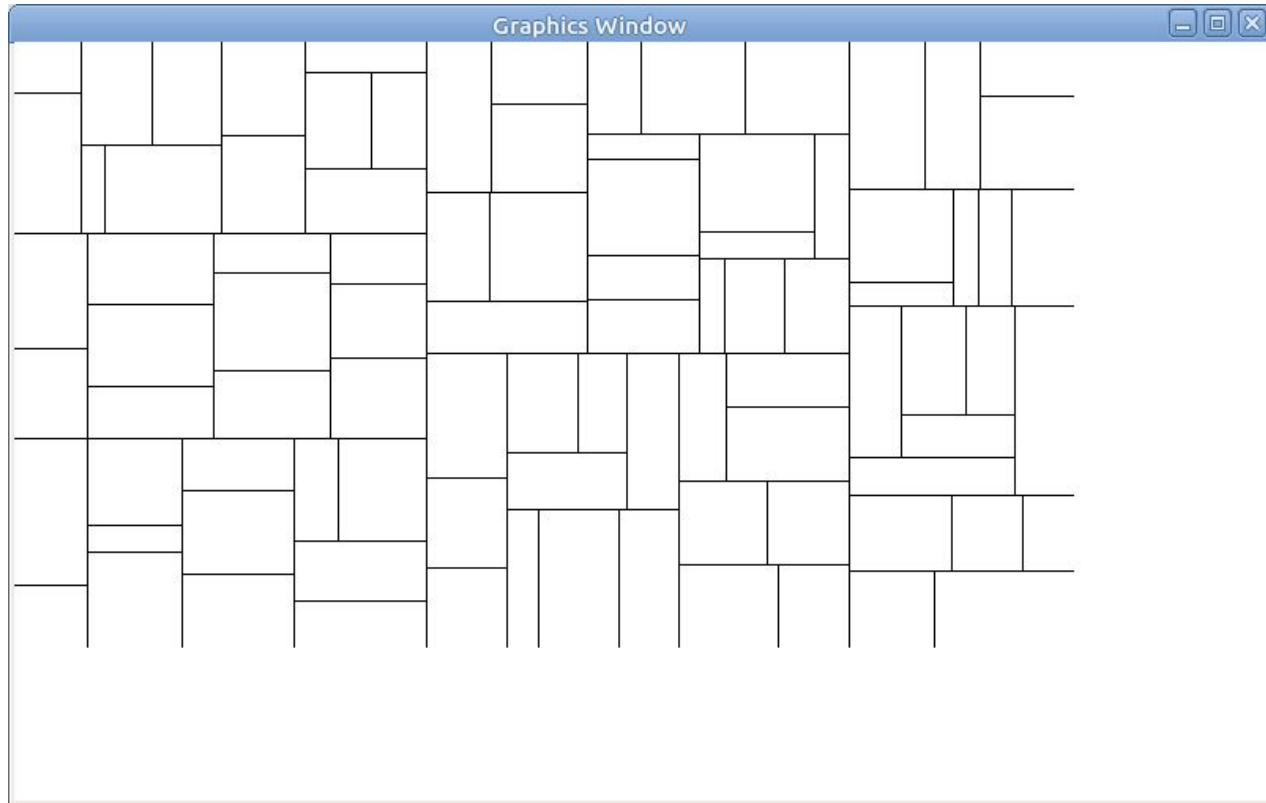
```
static void dividir_tela (double x, double y,
                          double largura, double altura)
{
    double divisoria, area;
    area = largura * altura;

    if (area < AREA_MINIMA)
        ;
    else
    {
        if (largura > altura)
        {
            divisoria = largura * RandomReal(LADO_MINIMO, 1 - LADO_MINIMO);
            MovePen(x + divisoria, y);
            DrawLine(0, altura);
            dividir_tela(x, y, divisoria, altura);
            dividir_tela(x + divisoria, y, largura - divisoria, altura);
        }
        else
        {
            divisoria = altura * RandomReal(LADO_MINIMO, 1 - LADO_MINIMO);
            MovePen(x, y + divisoria);
            DrawLine(largura, 0);
            dividir_tela(x, y, largura, divisoria);
            dividir_tela(x, y + divisoria, largura, altura - divisoria);
        }
    }
}
```

# Aplicações gráficas da recursividade: Mondrian



# Aplicações gráficas da recursividade: Mondrian

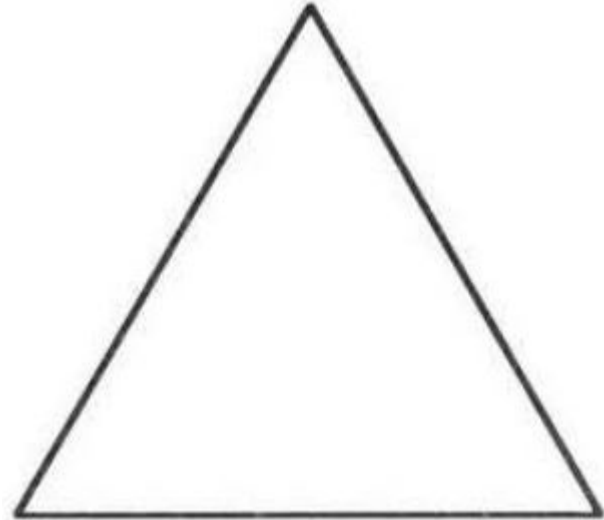


# Aplicações gráficas da recursividade: Fractais

- Fractais são estruturas geométricas nas quais um mesmo padrão é repetido em diferentes escalas.
- São conhecidos pelos matemáticos há muito tempo.
- No final da década de 1970, um pesquisador da IBM chamado **Benoit Mandelbrot** publicou um livro sobre fractais que reacendeu o interesse sobre o tema, principalmente pela possibilidade de gerar essas estruturas por computador.

# Aplicações gráficas da recursividade: Fractais

- Um os fractais mais simples e conhecidos é o Floco de Neve de Koch, criado por Helge von Koch.
- Iniciamos com um triângulo equilátero: esse é o chamado fractal de Koch de ordem 0:



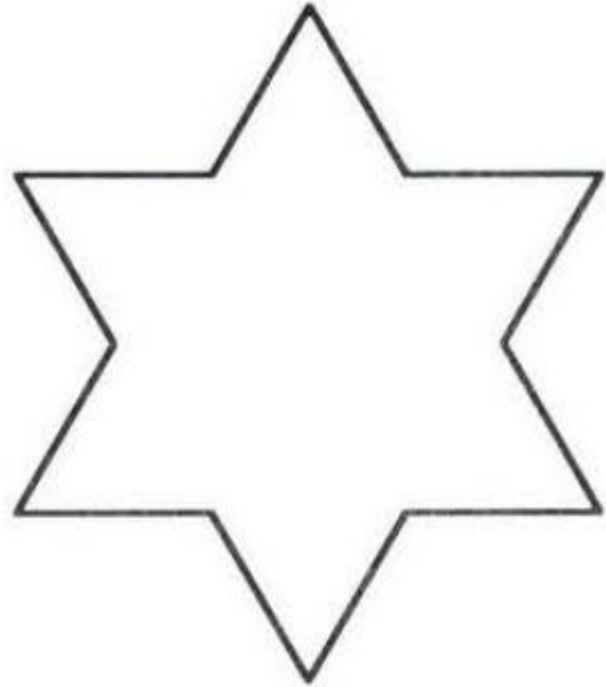
# Aplicações gráficas da recursividade: Fractais

- A partir do fractal de Koch de ordem 0, vamos revisando a figura para gerar fractais de ordem sucessivamente maiores.
- A revisão é feita da seguinte maneira: cada segmento de reta é substituído por um outro segmento que tem, no meio, os lados de um triângulo menor:



# Aplicações gráficas da recursividade: Fractais

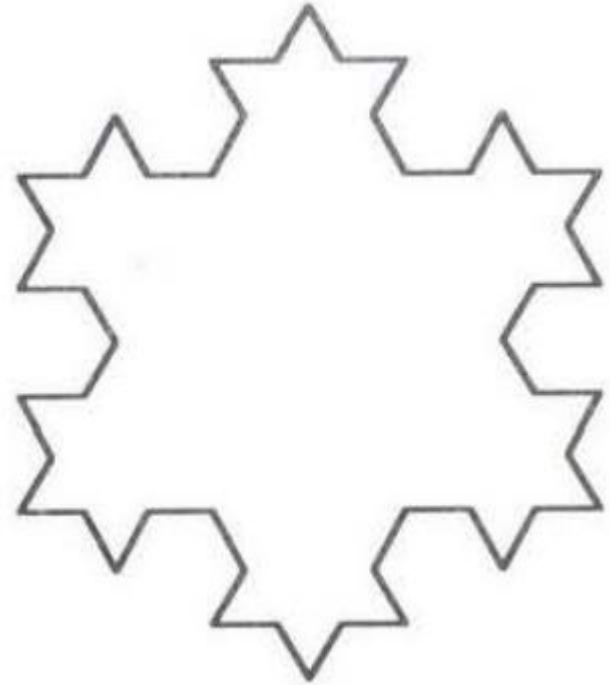
- Aplicando a transformação anterior a todos os lados do fractal de Koch de ordem 0, obtemos o fractal de Koch de ordem 1:





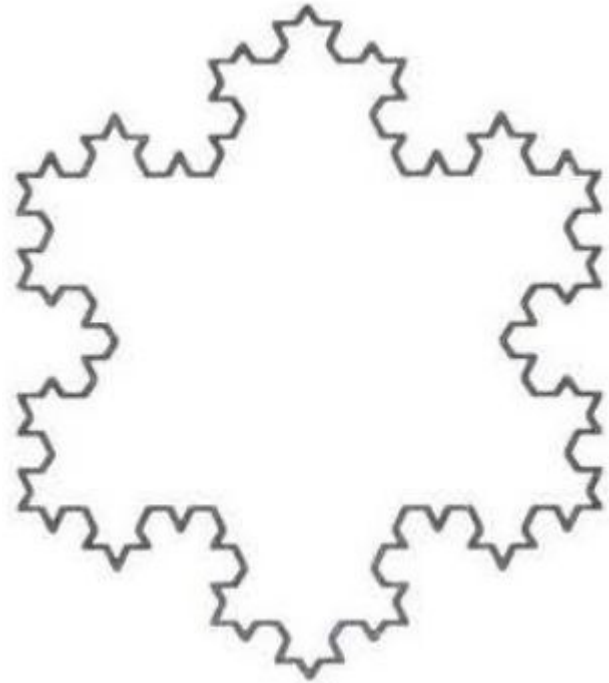
# Aplicações gráficas da recursividade: Fractais

- Continuando o processo, obtemos o fractal de Koch de ordem 2:



# Aplicações gráficas da recursividade: Fractais

- Continuando o processo, obtemos o fractal de Koch de ordem 3:



- E assim por diante...

# Aplicações gráficas da recursividade: Fractais

- Como usar recursão para criar fractais?
- Em aplicações gráficas, é mais fácil pensar nas retas como tendo um comprimento ( $r$ ) e uma direção theta ( $\theta$ ).
- Os parâmetros  $r$  e  $\theta$  são chamados de coordenadas polares da linha.



# Aplicações gráficas da recursividade: Fractais

- Vamos usar o procedimento `DrawLine` para criar o `DrawPolarLine` para nos ajudar no processo de criar fractais.

```
static void DrawPolarLine (double r, double theta)
{
    double radians;
    radians = theta / 180 * M_PI;
    DrawLine(r * cos(radians), r * sin(radians));
}
```

# Aplicações gráficas da recursividade: Fractais

- Somente com o `DrawPolarLine` já conseguimos criar o fractal de Koch de ordem 0, se fizermos algo como:

```
DrawPolarLine(2, 0);  
DrawPolarLine(2, 120);  
DrawPolarLine(2, 240);
```

- Mas como desenhar um fractal de ordem superior? Criando uma função wrapper genérica que recebe o tamanho, o ângulo e a ordem! Vamos chamar essa função de `DrawFractalLine`.

# Aplicações gráficas da recursividade: Fractais

- **DrawFractalLine** funciona da seguinte maneira:
  - Se o caso for simples (ordem 0), DrawFractalLine simplesmente desenha uma linha reta com o tamanho e a direção especificada;
  - Se a ordem for maior do que 0, a linha é quebrada em 4 pedaços, cada um sendo em si mesmo uma linha fractal de menor ordem.

```
static void DrawFractalLine(double tamanho, double theta, int ordem)
{
    if (ordem == 0)
        DrawPolarLine(tamanho, theta);
    else
    {
        DrawFractalLine(tamanho/3, theta, ordem - 1);
        DrawFractalLine(tamanho/3, theta - 60, ordem - 1);
        DrawFractalLine(tamanho/3, theta + 60, ordem - 1);
        DrawFractalLine(tamanho/3, theta, ordem - 1);
    }
}
```

# Aplicações gráficas da recursividade: Fractais

- Por fim temos que ter uma função para dar início ao desenho do fractal, com o tamanho e a ordem informadas pelo usuário:

```
static void fractal (double tamanho, int ordem)
{
    double x0, y0;
    x0 = GetWindowWidth() / 2.0 - tamanho / 2.0;
    y0 = GetWindowHeight() / 2.0 - sqrt(3.0) * tamanho / 6.0;
    MovePen(x0, y0);
    DrawFractalLine(tamanho, 0, ordem);
    DrawFractalLine(tamanho, 120, ordem);
    DrawFractalLine(tamanho, 240, ordem);
}
```

## Em resumo:

- O objetivo principal desta discussão é mostrar que existem problemas altamente sofisticados e complexos que podem ser resolvidos com facilidade com procedimentos recursivos, mas são muito difíceis de resolver iterativamente.
- Como você pôde perceber, os problemas discutidos aqui são DIFÍCEIS de entender; a solução recursiva também é de DIFÍCIL compreensão. O único jeito para aprender é praticar.



## Pontos para se lembrar dos capítulos 4 e 5:

- Sempre que você quiser aplicar recursividade para um problema, tenha certeza de que você consegue quebrar o problema em instâncias menores da mesma forma. Se você não conseguir achar o insight que permita a decomposição recursiva, você não pode usar recursividade.
- Se você achou uma estratégia de decomposição recursiva, verifique se ela não está quebrando nenhuma regra da própria recursividade ou do problema que você está trabalhando.
- Aceito o salto de fé recursivo.
- Se necessário, faça o desenho do stack, mas vença o ceticismo que está te forçando a olhar os detalhes do stack.

## Pontos para se lembrar dos capítulos 4 e 5:

- Subprogramas wrapper são muito úteis em programas recursivos:
  - Servem para chamar soluções mais gerais aos problemas
  - Servem para passar argumentos adicionais que o usuário não precisa se importar
- A biblioteca `graphics.h` é antiga e pode congelar seu computador! Leia as instruções que eu escrevi nos códigos fonte deste capítulo e use com cuidado.