Aula 10:

Recursão

Prof. Edvard

edvard@unifei.edu.br

Universidade Federal de Itajubá

Pilha de Chamada

- Primeiramente, vamos pensar bem no conceito de **pilha** e elaborar nossa definição.
- Podemos imaginar inicialmente uma pilha de documentos.
  - Quando um documento é colocado sobre a pilha, ele é normalmente colocado no topo (empilhar).
  - Quando um documento é retirado na pilha, ele também é, normalmente, removido do topo (desempilhar).
- Portanto, pilhas são estruturas que não retribuem o tempo de espera. Muitas vezes, documentos que estão na sua base podem demorar muito tempo até serem processados.

Pilha de Chamada

- Pilhas possuem um mecanismo chamado, em computação, de **LIFO: *Last In, First Out***, ou seja, "Último a entrar, primeiro a sair".
- Essa estrutura é importantíssima em várias aplicações computacionais e é uma das estruturas de dados básicas e clássicas.
  - Existem muitos casos onde sua utilização é aconselhada. Principalmente, quando precisamos **conhecer um caminho de volta passo a passo**, utilizando os mesmos passos do caminho de ida.
  - **Ex.** Se empilharmos o caminho que seguimos de nossas casas até a UNIFEI, basta que o desempilhemos na mesma ordem e o façamos ao contrário para chegar de volta em nossas casas.

Pilha de Chamada

- **Caminho de Vinda**
  1. Saia de casa;
  2. Ande 100 metros;
  3. Vire para a esquerda;
  4. Ande 50m;
  5. Vire à esquerda;
  6. Ande 1500 m na Av. BPS;
  7. Você chegou a seu destino.

- **Caminho de Volta**

  7. Saia da UNIFEI;

  6. Ande 1500 m na Av. BPS;

  5. Vire para a direita;

  4. Ande 50m;

  3. Vire à direita;

  2. Ande 100 metros;

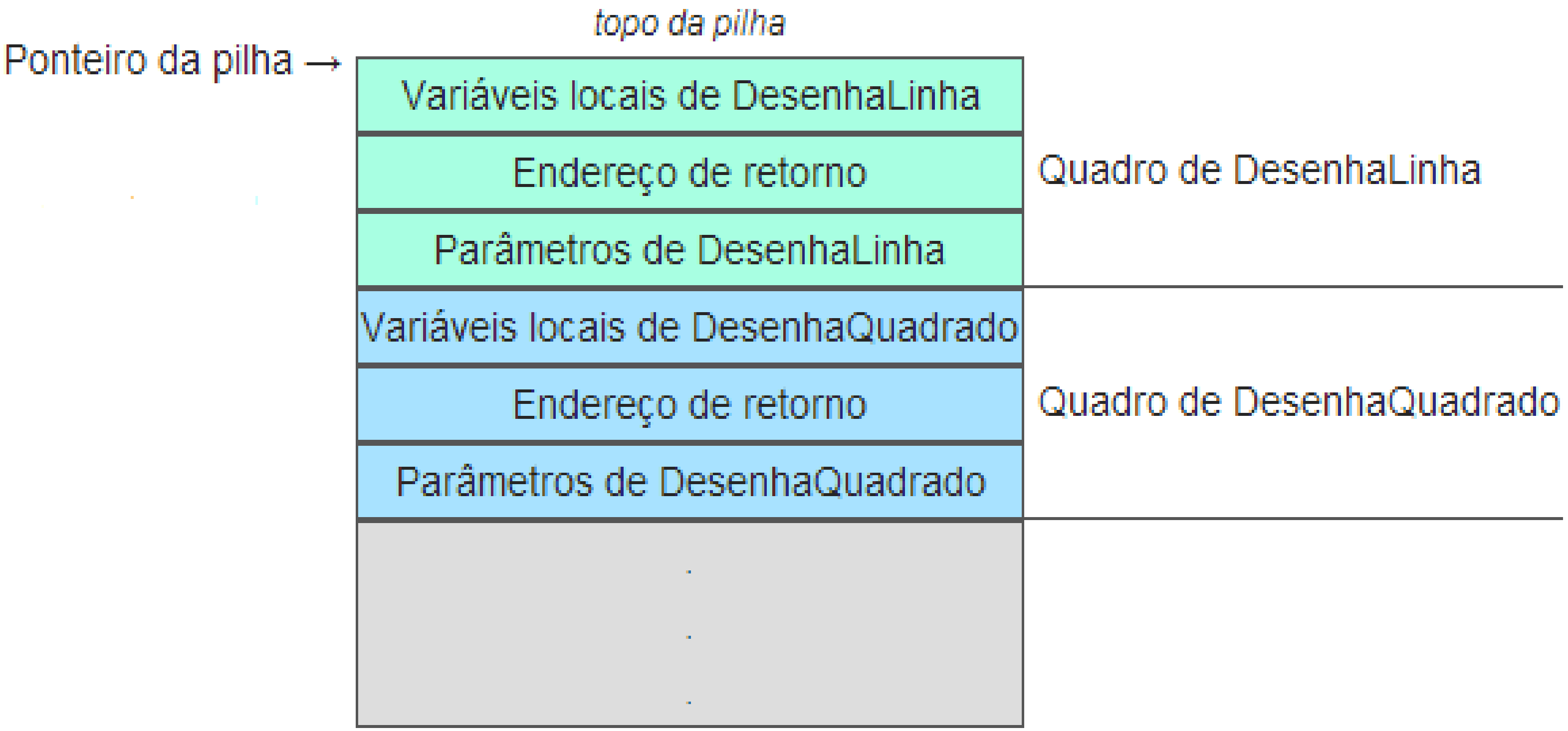
  1. Você chegou a seu destino

Pilha de Chamada

- Nossos programas em C utilizam, também, uma estrutura de pilha para guardar informações sobre chamadas de função: A **Pilha de Chamada** (*Call Stack*)
  - Cada vez que chamamos uma função, suas informações são guardadas em uma pilha, de maneira que, se outra função for chamada, e outra, e outra, elas saibam exatamente para quem retornar (em que endereço).
  - **Que informações são guardadas na pilha de chamada?**
    - **O endereço da instrução de retorno, na função chamadora;**
    - **Variáveis locais da função chamada;**
    - **Parâmetros da função chamada;**
    - **Etc.**

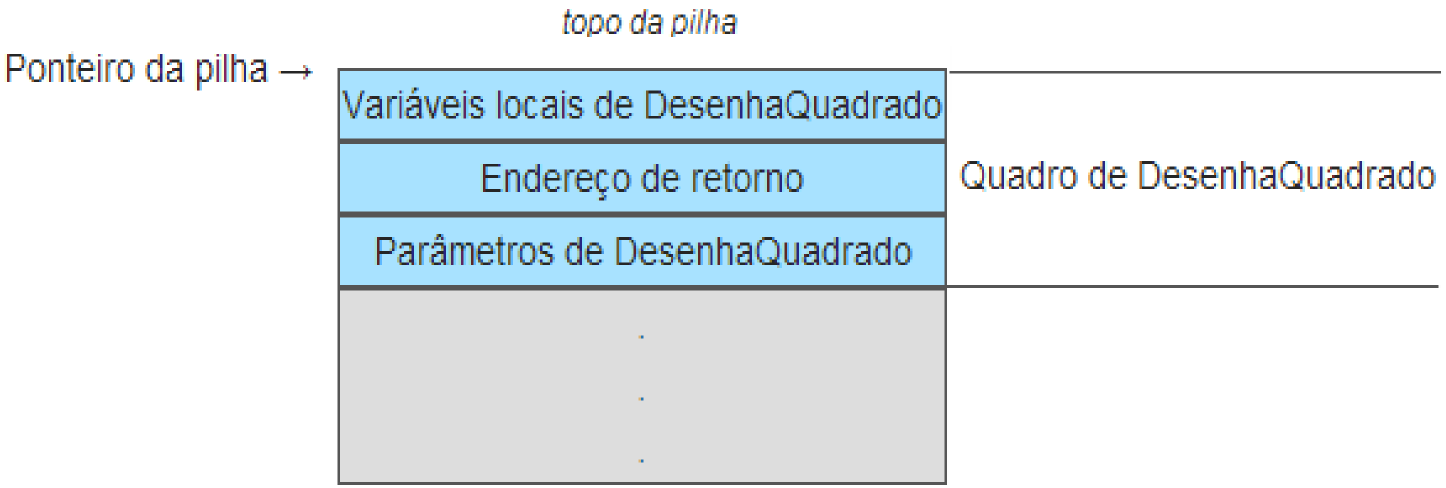
Pilha de Chamada

- Imagine o seguinte exemplo:
  - Imagine que uma função **DesenhaLinha** está em execução, e foi chamada por **DesenhaQuadrado**. O topo da pilha de chamada pode ser estruturado da seguinte maneira:



Pilha de Chamada

- Cada quadro possui um tamanho específico, determinado pelo número e tamanho de parâmetros e variáveis locais.
  - O tamanho é fixo para diferentes chamadas de uma mesma função.
- O quadro no topo da pilha se refere à função em execução. Ao **terminar a execução**, o respectivo quadro é **desempilhado** e o topo da pilha passa a ser o quadro da função chamadora:



Pilha de Chamada

- Naturalmente, como a quantidade de memória em um computador é finita, apenas uma quantidade limitada de espaço é reservada para armazenamento da pilha de chamadas.
  - Se houver um número muito grande de chamadas de funções dentro de chamadas de funções (por exemplo, em funções recursivas), o **espaço torna-se insuficiente** e gera um erro conhecido como **Estouro da Pilha** (*Stack Overflow*, em inglês).
  - Fique atento!

Curiosidade

- Tamanho padrão da Pilha de Chamada em alguns compiladores e sistemas operacionais:
  - glibc i386, x86_64 **7.4 MB**
  - Tru64 5.1 **5.2 MB**
  - Cygwin **1.8 MB**
  - Solaris 7..10 **1 MB**
  - MacOS X 10.5 **460 KB**
  - AIX 5 **98 KB**
  - OpenBSD 4.0 **64 KB**
  - HP-UX 11 **16 KB**

Funções Recursivas

- **Mas o que são as tais funções recursivas?**
  - Em C, como já estamos acostumados a ver, uma função pode, livremente, chamar outra função que já esteja declarada.
  - **Mas uma função pode chamar a ela mesma?**
  - SIM! Funções que chamam a si mesmas durante sua definição são chamadas de funções recursivas.
    - Em alguns casos (bem de vez em quando), a recursão também pode ser chamada de “**definição circular**”. Ela ocorre sempre que algo é definido em termos de si mesmo.

Funções Recursivas

- Funções com **recursão** normalmente utilizam o paradigma de programação chamado de **divisão e conquista**.
  - A divisão é **realizada várias vezes**, até que encontremos um subproblema cuja **resposta é simples o suficiente**.
- O trabalho real é feito em **três lugares**:
  - Na **partição dos problemas** em subproblemas;
  - Na **base final da recursão**, quando os problemas são tão pequenos que podem ser resolvidos diretamente, de maneira trivial;
  - Na **combinação das respostas** parciais, formando a resposta total.
- Tudo isso, coordenado e reunido pela **estrutura recursiva da função**.

Divisão e Conquista

- Como **exemplo**, vejamos como podemos criar uma função recursiva para o cálculo do fatorial de um número n.
  - O fatorial de n segue a seguinte fórmula:

$$n! = 1 * 2 * 3 * 4 * 5 * \cdots * (n-2) * (n-1) * n$$

- No entanto, se analisarmos bem, o que $1 * 2 * 3 * 4 * 5 * \cdots * (n-2) * (n-1)$ representa?

- Portanto, podemos representar o fatorial de n por:

$$n! = (n-1)! * n$$

Ao Trabalho

Escreva uma **função recursiva** que calcule o **fatorial de n**. Não é permitido utilizar nenhuma estrutura de repetição (sem iteração)

```c
#include <stdio.h>

// Prototipo
int fatorial(int);

int main()
{
    int num;

    printf("Entre com um numero: ");
    scanf("%d", &num);

    printf("%d! = %d", num, fatorial(num));

    return 0;
}

// Calcula fatorial recursivamente
int fatorial(int n)
{
    if(n == 0)
        return 1;

    return fatorial(n-1)*n;
}
```

Recursão

Função recursiva que calcula o **fatorial de n**.

Repare que a declaração do protótipo e a chamada da função não possuem nenhuma diferença em relação à versão iterativa.

A função recursiva deve sempre ter um **caso base**, que finalize a sequência de chamadas.
Caso não houvesse o teste (n == 0), ela entraria em um loop infinito que só terminaria quando o limite da pilha de chamada fosse estourado (stack overflow)

Funções Recursivas

- Note que o fatorial de n, **n!**, é definido em termos do fatorial de n-1.
- Trata-se, portanto, de uma **definição circular** do problema, onde precisamos saber o fatorial de um número para calcular o de outro.
  - Esse processo continua até que se chegue a um caso mais simples (**caso base**), que é a base do cálculo do fatorial: 0! = 1.
  - Nesse momento, cessam as chamadas, e as funções começam a retornar os valores para as chamadoras. Veja a ilustração:

Funções Recursivas

- Veja que temos sempre um **caminho de ida da recursão**, onde são realizadas chamadas recursivas até chegar ao caso base.
- Quando a recursão para, o programa faz o caminho inverso, ou o **caminho de volta**, que consiste em retornar o valor calculado para a função chamadora. Veja o exemplo, para cálculo de 4!.

4! = 4***3!**

3! = 3***2!**

Ida

2! = 2***1!**

1! = 1***0!**

0! = 1

Caso Base

1! = 1*1

2! = 2*1

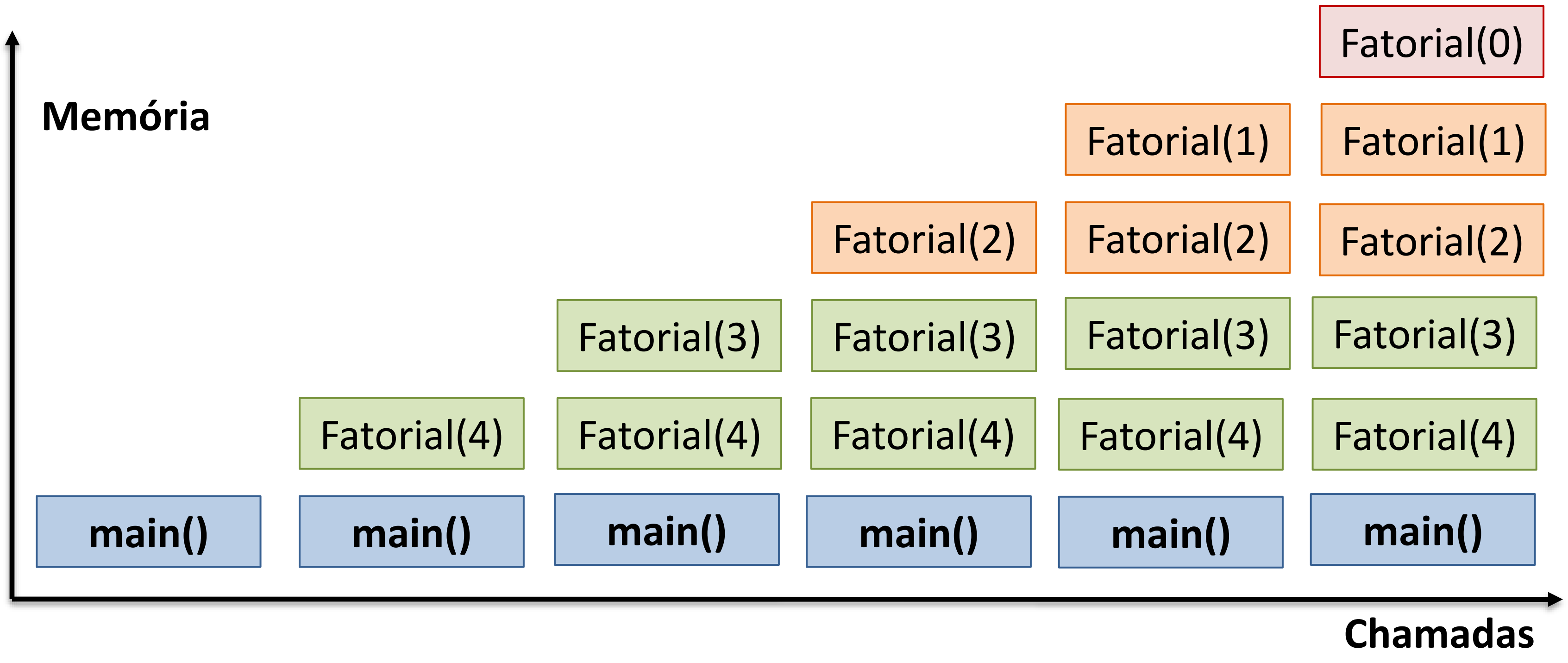
3! = 3*2

Volta

4! = 4*6 = **24**

Funções Recursivas

- Sempre que fazemos uma chamada de função, ela é empilhada na **pilha de chamada**. Portanto, para um caso simples como o exemplificado anteriormente, a pilha ficaria da seguinte maneira:



Funções Recursivas

- Um número muito grande de chamadas recursivas gera um alto consumo de memória.
  - É preciso saber identificar precisamente o caso base, para que o caminho de ida da recursão e seu caminho de volta, sejam realizados corretamente, sem estouro de pilha ou consumo exagerado de memória.
- Em geral, implementações recursivas de funções são consideradas mais **elegantes**, **intuitivas** e **enxutas**. Vamos comparar as implementações iterativa e recursiva do fatorial:

Funções Recursivas

Iterativo

```
// Calcula fatorial iterativamente
int fatorial(int n)
{
    int i, fatorial = 1;

    for(i = 1; i <= n; i++)
    {
        fatorial *= i;
    }

    return fatorial;
}
```

Recursivo

```
// Calcula fatorial recursivamente
int fatorial(int n)
{
    if(n == 0)
        return 1;

    return fatorial(n-1)*n;
}
```

Qualquer problema que possa ser resolvido recursivamente pode, também, ser resolvido com iteração. A técnica iterativa normalmente é deixada de lado quando a sua forma recursiva representa o problema mais naturalmente e resulta em uma solução mais fácil de entender e depurar.

Ao Trabalho!

- Faça uma função em que, dado um número informado pelo usuário, retorna o somatório de todos os números partindo de 1 até o valor informado.

$$\sum_{1} n$$

```c
#include<stdio.h>

int soma_num(int num)
{
   int resultado;
   if (num == 1)
    {
      return (1);
    }
   else
     {
        resultado = num + soma_num(num - 1);
      }
   return (resultado);
}

int main()
{
   int num_N;
   int somatorio;

   printf("\n Digite o numero N : ");
   scanf("%d", &num_N); /*o número digitado vai ser guardado na memória*/
   somatorio = soma_num(num_N); /*A variável somatório está chamando a função soma_num*/
   printf("\n O somatorio dos numeros de 1 a %d = %d \n", num_N, somatorio);

 return 0;
}
```

Ao Trabalho!

- Outro caso clássico de solução recursiva é o da sequência de Fibonacci.
  - Escreva uma função recursiva que calcule o enésimo número da sequência de Fibonacci:
  - **1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, …**
  - Onde: Fibonacci(1) = 1
    Fibonacci(2) = 1
    Fibonacci(3) = 2
    Fibonacci(4) = 3
    Fibonacci(5) = 5
    Fibonacci(6) = 8...

  ou seja, **Fibonacci(n) = Fibonacci(n-1) + Fibonacci(n-2)**

```c
#include <stdio.h>

// prototipo
int fibonacci(int);

// Função principal
int main()
{
    int num, fibo;
    printf("Entre com um numero inteiro: ");
    scanf("%d", &num);

    // chamada da funcao
    fibo = fibonacci(num);
    printf("Enesimo numero: %d. \n", fibo);
    return 0;
}

// Calcula Fibonacci Recursivamente
int fibonacci(int n)
{
    if(n == 1 || n == 2)
        return 1; // caso base

    return fibonacci(n-1) + fibonacci(n-2);
}
```

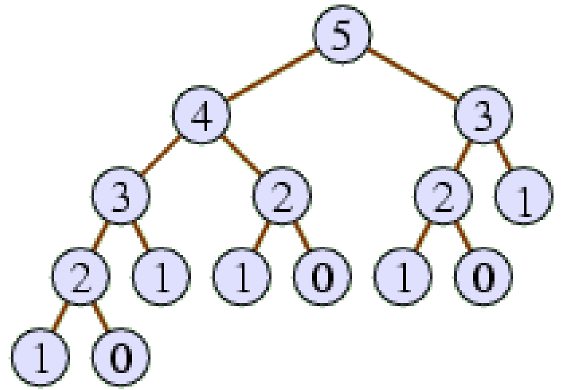
Como podemos observar, a definição de uma função recursiva para Fibonacci é muito intuitiva e elegante.

Infelizmente, no entanto, cada chamada da função realiza outras duas chamadas recursivas.

Funções Recursivas

- O algoritmo recursivo, apesar de resolver corretamente o problema, gasta uma grande quantidade de memória para armazenamento de dados na pilha de chamada.

Repare que cada chamada da função dispara outras duas chamadas, fazendo com que a quantidade de chamadas para a função aumente **exponencialmente** (**2^n**) com o tamanho da entrada n.



- Todas as tarefas de alocação e liberação de memória, cópia de informações para a pilha de chamada, envolvem um tempo computacional, de maneira que a versão recursiva é **mais elegante e intuitiva** mas, também, **menos eficiente** (gasta mais tempo).

Recursivo vs. Iterativo

Iterativo

```
// Calcula Fibonacci Iterativamente
int fibonacci(int numero)
{
    if(numero == 0 || numero == 1)
        return 1;

    int i, f, a1 = 1, a2 = 1;

    for(i = 2; i < numero; i++)
    {
        f = a1 + a2;
        a1 = a2;
        a2 = f;
    }

    return f;
}
```

Recursivo

```
// Calcula Fibonacci Recursivamente
int fibonacci(int n)
{
    if(n == 1 || n == 2)
        return 1; // caso base

    return fibonacci(n-1) + fibonacci(n-2);
}
```

Repare como a implementação recursiva representa muito mais claramente o conteúdo da série de Fibonacci.

Recursivo vs. Iterativo

- São iguais em:
  - Ambos envolvem **repetição**;
  - Envolvem **testes de término**;
  - Ambas podem ocorrer **infinitamente**;
- Pontos negativos:
  - Cada chamada realiza uma cópia da função, consumindo uma quantidade de memória considerável e um tempo precioso de processamento (overhead)
- Pontos positivos:
  - Intuitiva, fácil de programar e depurar

Desempenho vs. Boa Engenharia

- Criar programas de uma maneira elegante e hierárquica promove a **boa engenharia de software**.
- Um programa rigorosamente dividido em funções cria, potencialmente, grande quantidade de chamadas de funções, que consomem tempo de execução nos processadores.
- **Dilema:** Programas monolíticos podem ter um desempenho superior, mas são mais difíceis de desenvolver, testar, depurar e manter.
  - Qual a minha prioridade? Desempenho ou clareza?

Funções Recursivas

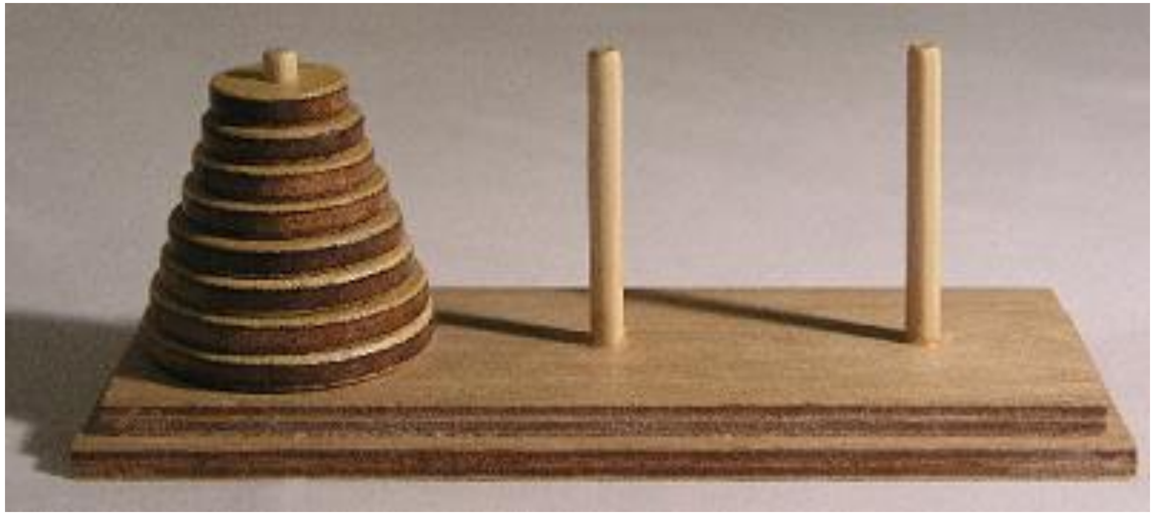
- Muitos outros problemas possuem fórmulas simples que se traduzem facilmente em código recursivo:
  - **Busca Binária** (*binary search*): Método de busca de um elemento chave em vetor com elementos ordenados – O(log n)
  - **Ordenação Rápida** (*quick sort*): Baseia-se na divisão de um vetor para ordenar seus elementos. O(n logn) no melhor caso.
  - **Problemas matemáticos:** Como, por exemplo, máximo divisor comum, e a Torre de Hanói, que veremos a seguir.

Torre de Hanói

- **Pano de Fundo Histórico:**
  - Brahma é considerado, pelos hindus, a representação da força criadora ativa no universo. A visão de universo pelos hindus é cíclica. Depois que um universo é destruído por Shiva, Vixnu se encontra dormindo e flutuando no oceano primordial. Quando o próximo universo está para ser criado, Brahma aparece montado numa flor de lótus brotada do umbigo de Vixnu e recria todo o universo.
  - Depois que Brahma cria o universo, ele permanece em existência por um dia de Brahma, que vem a ser aproximadamente 4 320 000 000 anos em termos de calendário hindu. Quando Brahma vai dormir, após o fim do dia, o mundo e tudo que nele existe é consumido pelo fogo. Quando ele acorda de novo, ele recria toda a criação, e assim sucessivamente, até que se completem 100 anos de Brahma.

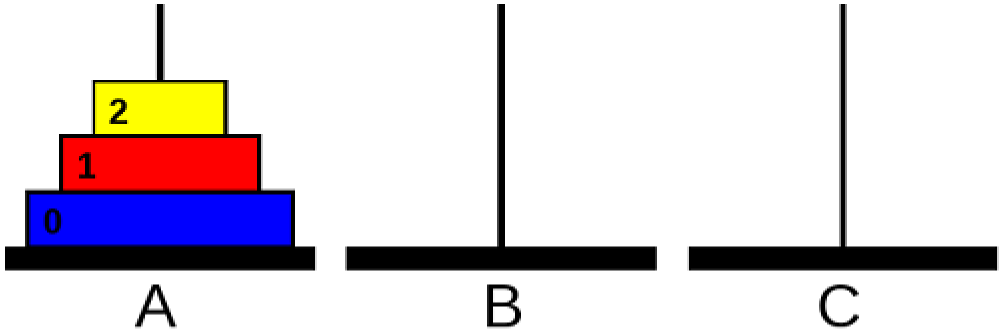
Torre de Hanói

- **Pano de Fundo Histórico (curiosidade):**
  - Diz a lenda que um monge muito preocupado com o fim do Universo perguntou a seu mestre quando isso iria ocorrer.
  - O mestre, então, vendo a aflição do discípulo, pediu a ele que olhasse para os três postes do monastério e observasse os 64 discos de tamanhos diferentes empilhados no primeiro poste.



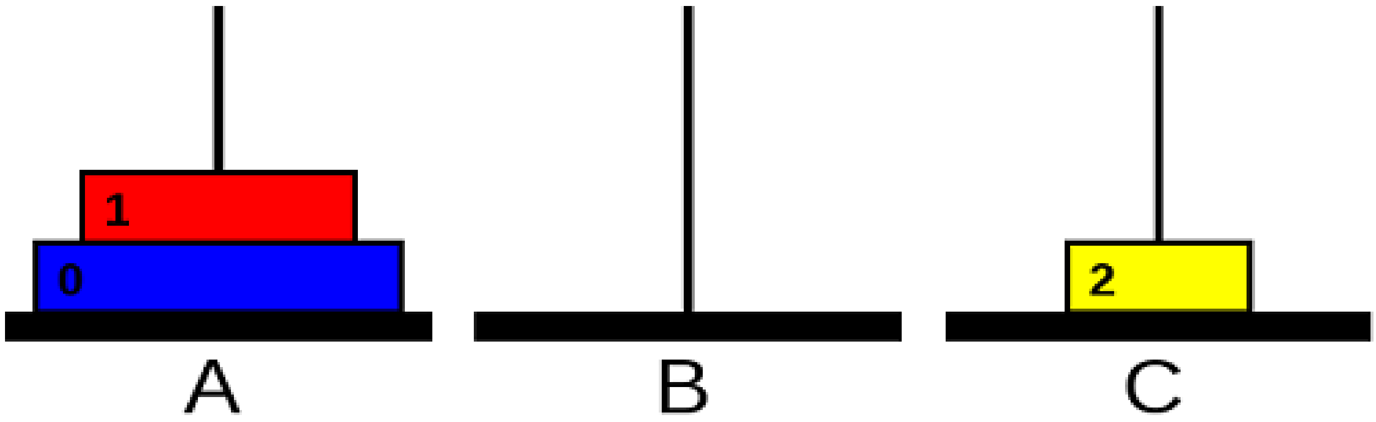
Torre de Hanói

- **Definição do Problema**
  - O mestre disse que, para saber o tempo que levaria para o universo acabar, bastava calcular a quantidade de movimentos necessários para mover todos os discos do primeiro para o terceiro poste, seguindo as regras básicas:
    - Só é possível movimentar um disco de cada vez;
    - Um disco maior nunca pode ficar sobre um menor.
  - Quais seriam os movimentos necessários para mover os discos da haste A para a haste C, utilizando a haste B auxiliar?

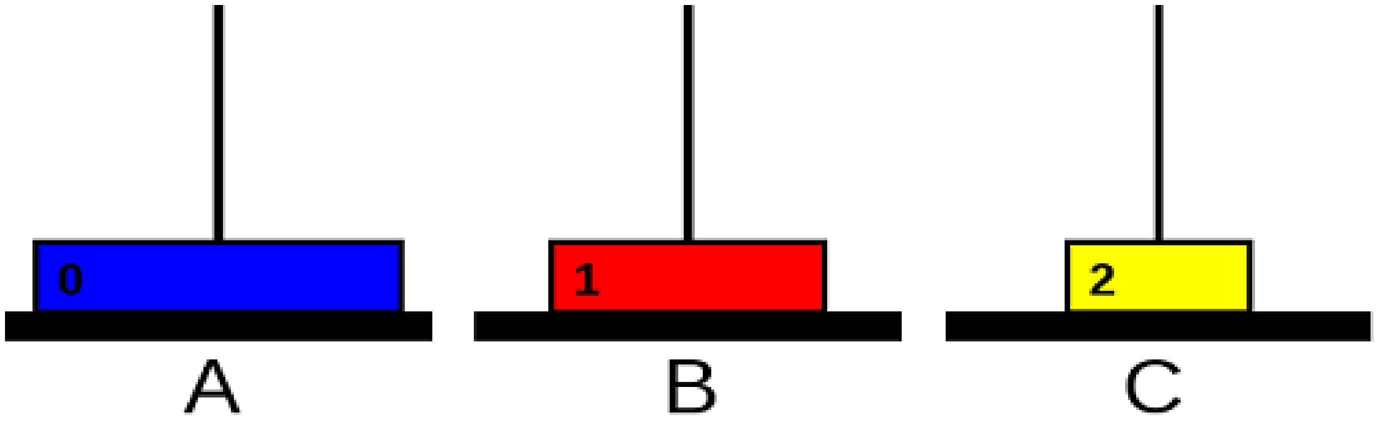


Torre de Hanói

1º **Passo**: mover disco 1 de A para B

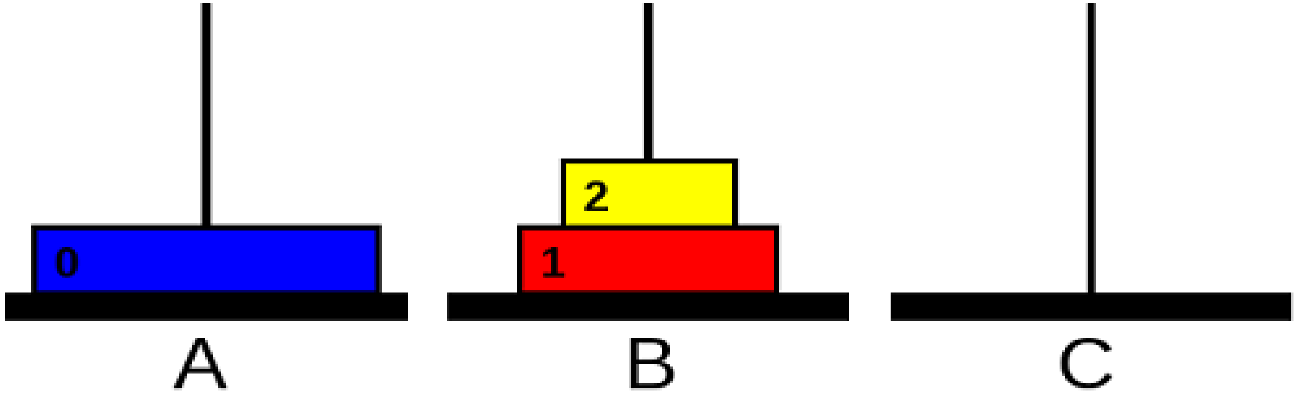


2º **Passo**: mover disco 2 de A para AUX

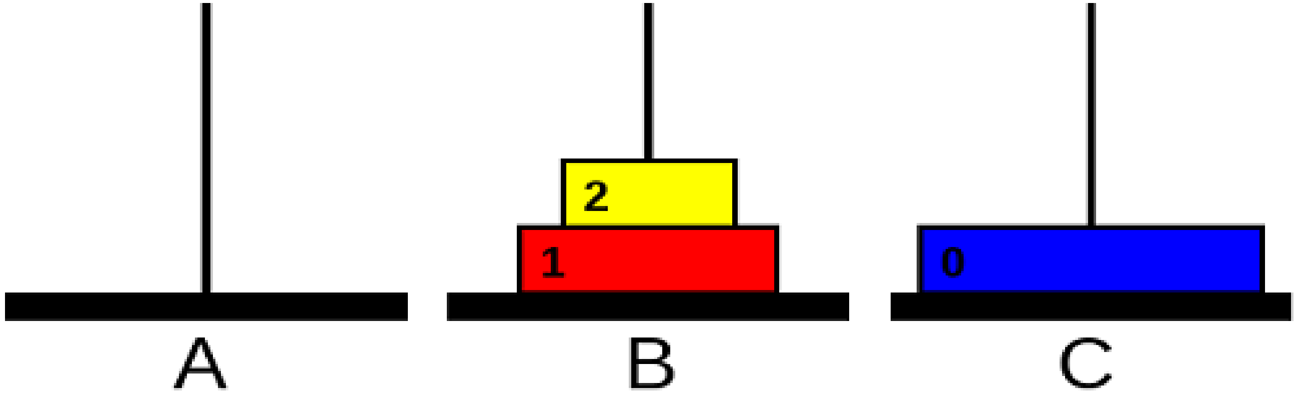


Torre de Hanói

3º **Passo**: mover disco 1 de B para AUX

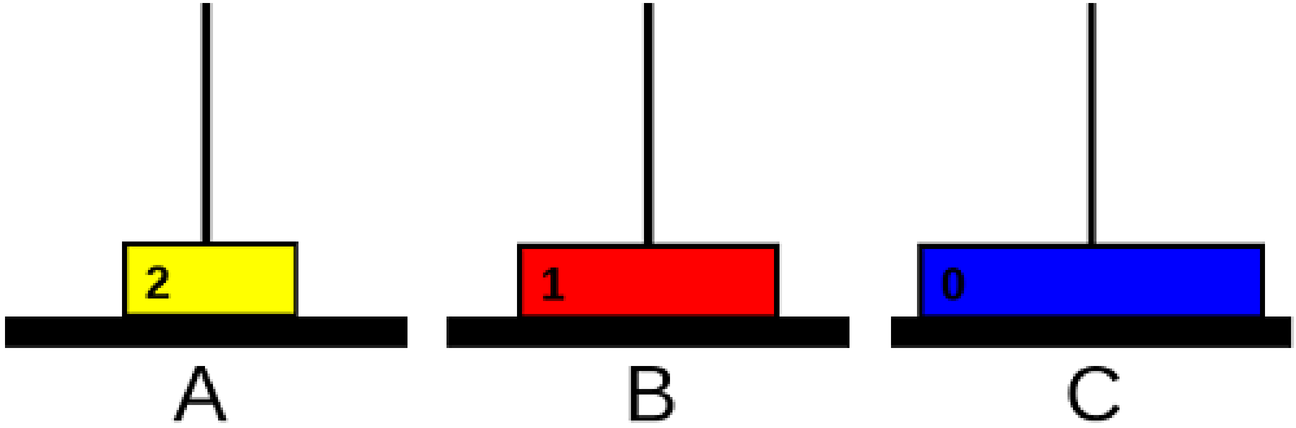


4º **Passo**: mover disco 3 de A para B

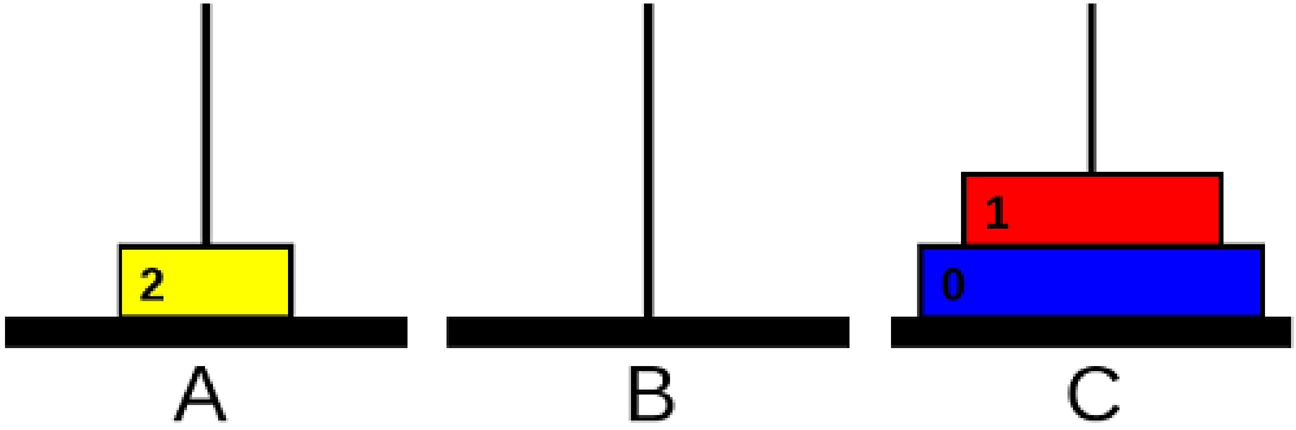


Torre de Hanói

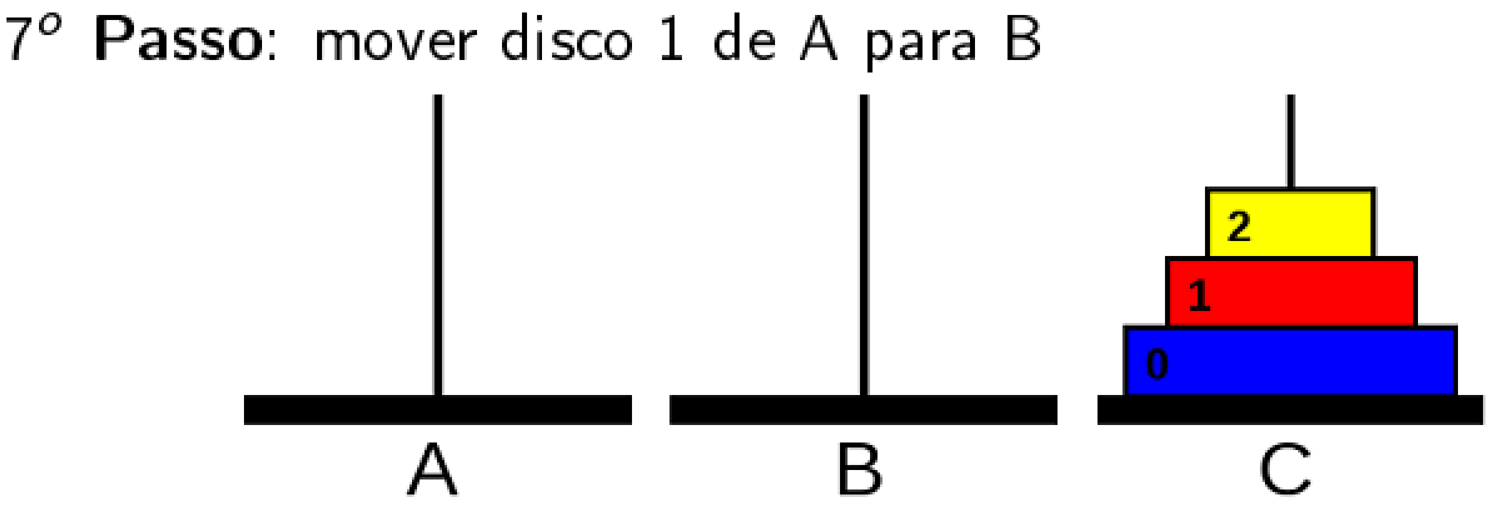
5^o **Passo**: mover disco 1 de AUX para A



6^o **Passo**: mover disco 2 de AUX para B



Torre de Hanói



Como ficaria, portanto, um algoritmo para definição da quantidade de movimentos necessários para mover uma determinada quantidade de discos da primeira para a última haste, utilizando uma haste auxiliar?

Torre de Hanói

- Podemos ver, através principalmente dos passos 3 e 4, que o problema se resume a três passos básicos:
  1. Mover **n-1** discos de A para B, utilizando C como auxiliar (recursão);
  2. Mover o disco grande de A para C;
  3. Mover **n-1** discos de B para C utilizando A como auxiliar (recursão).
- E quanto ao **caso base**?
  – Consiste no caso onde **somente há um disco**, que é movido diretamente de sua haste original para o destino.

Ao Trabalho!

```c
#include <stdio.h>

// Calcula quantidade de movimentos e Imprime cada um deles
void hanoi(int ndiscos, int orig, int dest, int aux, int* nmov)
{
    if(ndiscos == 1)
    {
        printf("Move disco %d de haste %d para haste %d\n", ndiscos, orig, dest);
        (*nmov)++;
    }
    else
    {
        hanoi(ndiscos-1, orig, aux, dest, nmov);
        printf("Move disco %d de haste %d para haste %d\n", ndiscos, orig, dest);
        (*nmov)++;
        hanoi(ndiscos-1, aux, dest, orig, nmov);
    }
}

int main()
{
    int num_discos, num_movimentos = 0;

    printf("Entre com a quantidade de discos: ");
    scanf("%d", &num_discos);

    // Passa variável num_movimentos por referência, para modificar seu valor
    hanoi(num_discos, 0, 2, 1, &num_movimentos);
    printf("\nNumero total de movimentacoes: %d\n\n", num_movimentos);

    return 0;
}
```

Ao Trabalho!

- **Curiosidade:**
  - O número de movimentos para conseguir mover todos os discos da haste origem para a haste destino segue uma regra simples e é igual a 2^n-1, sendo n o número de discos.
  - Assim:
    - **3 discos:** 7 movimentos
    - **7 discos:** 127 movimentos
    - **15 discos:** 32767 movimentos
    - **64 discos:** 18.446.744.073.709.551.615 movimentos