INF 112: Programação II

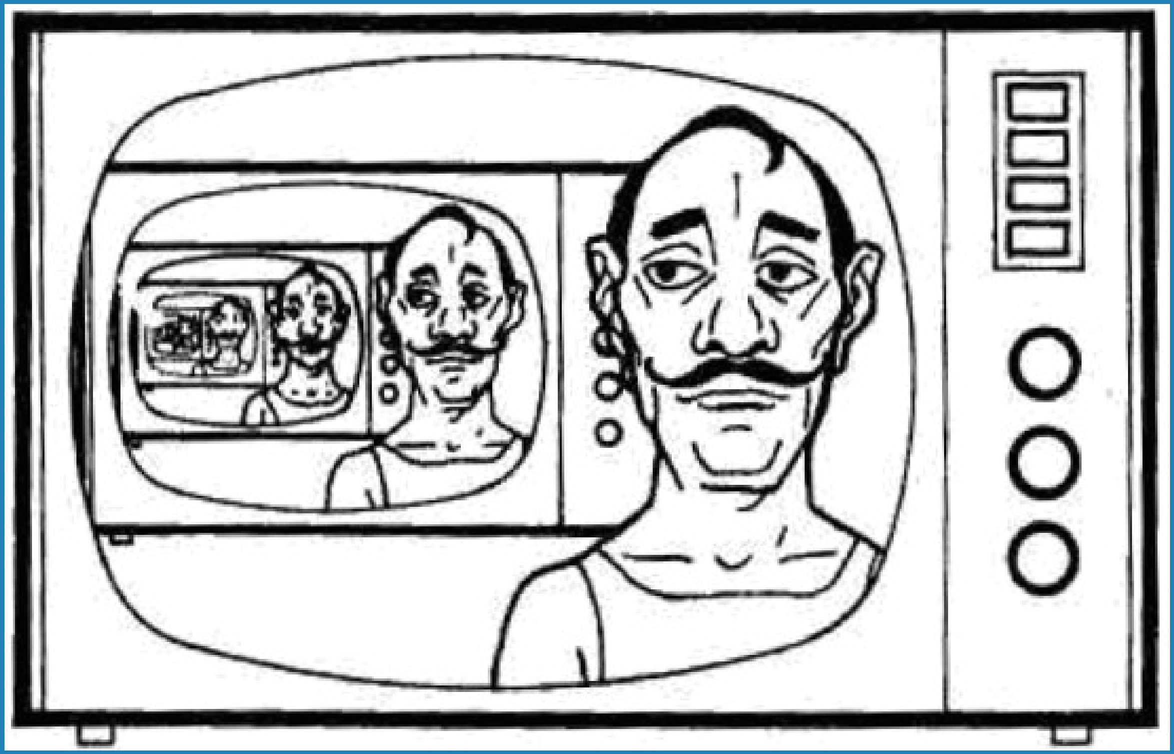
Aula 02

- **Recursividade**

Fábio R. Cerqueira, UFV, DPI, frcerqueira@gmail.com

Recursividade

- Um objeto é dito recursivo se ele consistir parcialmente ou for definido em termos de si próprio. *Recursividade* é encontrada não apenas em matemática ou computação, mas também no dia a dia.



Recursividade

➔ Recursividade é uma técnica particularmente poderosa em definições matemáticas. Um exemplo familiar é o cálculo de fatorial:

- A função fatorial $n!$ (para inteiros não negativos):
  - (a) $0! = 1$,
  - (b) $n > 0$: $n! = n \times (n-1)!$.

➔ Funções como esta, envolvendo argumentos inteiros, são chamadas de *relações de recorrência*. Note que a definição inclui a própria função. No entanto, existe a definição não recursiva da função para um *elemento inicial*. É isto que garante que a sequência gerada pela recorrência tenha fim.

➔ O poder da recursividade deve-se à possibilidade de se definir um conjunto infinito de objetos através de uma formulação finita. Veja que no caso da função fatorial, muitas vezes define-se $n!$, onde $n > 0$, como: $n \times (n-1) \times (n-2) \times \ldots \times 1$.

➔ Essa definição utilizando reticências é válida, mas é mais fraca se comparada à definição anterior, que é bem mais precisa.

➔ Exploremos mais a função fatorial. Pela nossa relação de recorrência, 5! seria 5 x 4!. Então antes de avaliar 5!, devemos primeiro avaliar 4!. Usando a definição mais uma vez, vemos que 4! = 4 x 3!. Então devemos avaliar 3!...

Recursividade

(a) $0! = 1$,

(b) $n > 0$: $n! = n \times (n-1)!$.

... Repetindo este processo teremos:

1	$5! = 5 \times 4!$
2	$4! = 4 \times 3!$
3	$3! = 3 \times 2!$
4	$2! = 2 \times 1!$
5	$1! = 1 \times 0!$
6	$0! = 1$

➔ Cada caso é reduzido para o caso mais simples até que cheguemos no 0!, que é definido diretamente como 1 (linha 6). Podemos então ir voltando da linha 6 para a linha 1, retornando o valor computado em uma linha, de forma a avaliar o resultado da linha anterior. Isto produzirá:

6	$0! = 1$
5	$1! = 1 \times 0! = 1 \times 1 = 1$
4	$2! = 2 \times 1! = 2 \times 1 = 2$
3	$3! = 3 \times 2! = 3 \times 2 = 6$
2	$4! = 4 \times 3! = 4 \times 6 = 24$
1	$5! = 5 \times 4! = 5 \times 24 = 120$

- No caso da computação existem os procedimentos recursivos que são aqueles que fazem chamadas a eles mesmos. Como no caso da definição de funções matemáticas recursivas, um algoritmo recursivo deve possuir algo que o impeça de se chamar indefinidamente. Caso contrário, nunca parará até que se esgote os recursos da máquina. A parte que garante o fim da recursividade é muitas vezes chamada de caso *base* ou *trivial*.

- O uso de recursividade possibilita a construção de programas elegantes uma vez que parte do controle do fluxo do programa é incorporado na própria sequência de chamadas. Na verdade, podemos eliminar a necessidade de estruturas de iteração (*for*, *while*, etc.) com o uso da recursividade. Além disto, a recursividade se encaixa naturalmente em certos problemas que usam estruturas recursivas (listas, árvores, etc.) ou que são definidos de forma recursiva, como a função fatorial.

- No entanto, existem desvantagens no uso de recursividade. Os programas recursivos, em geral, são mais lentos do que seus equivalentes *iterativos*. Isto ocorre porque a chamada de uma função é uma operação lenta (necessidade de empilhar os argumentos, endereço de retorno, variáveis locais, etc.).

- Além disto, os programas recursivos podem ser menos manuteníveis, uma vez que é difícil acompanhar o fluxo de execução pelos diversos níveis de chamadas recursivas. Por isto, alguns programadores usam a técnica de expressar primeiramente seu programas de forma recursiva para depois transformá-los para a forma iterativa para melhorar a eficiência. Todo programa recursivo possui equivalentes iterativo e vice-versa.

Recursividade - fatorial

➔ Vejamos duas funções C++ distintas para calcular o fatorial de um número. A primeira é uma versão iterativa e a segunda uma versão recursiva.

```
int fatorial(int n) {
  int prod = 1, x;
  for (x = n; x > 0; x--)
    prod = prod * x;
  return prod;
}
```

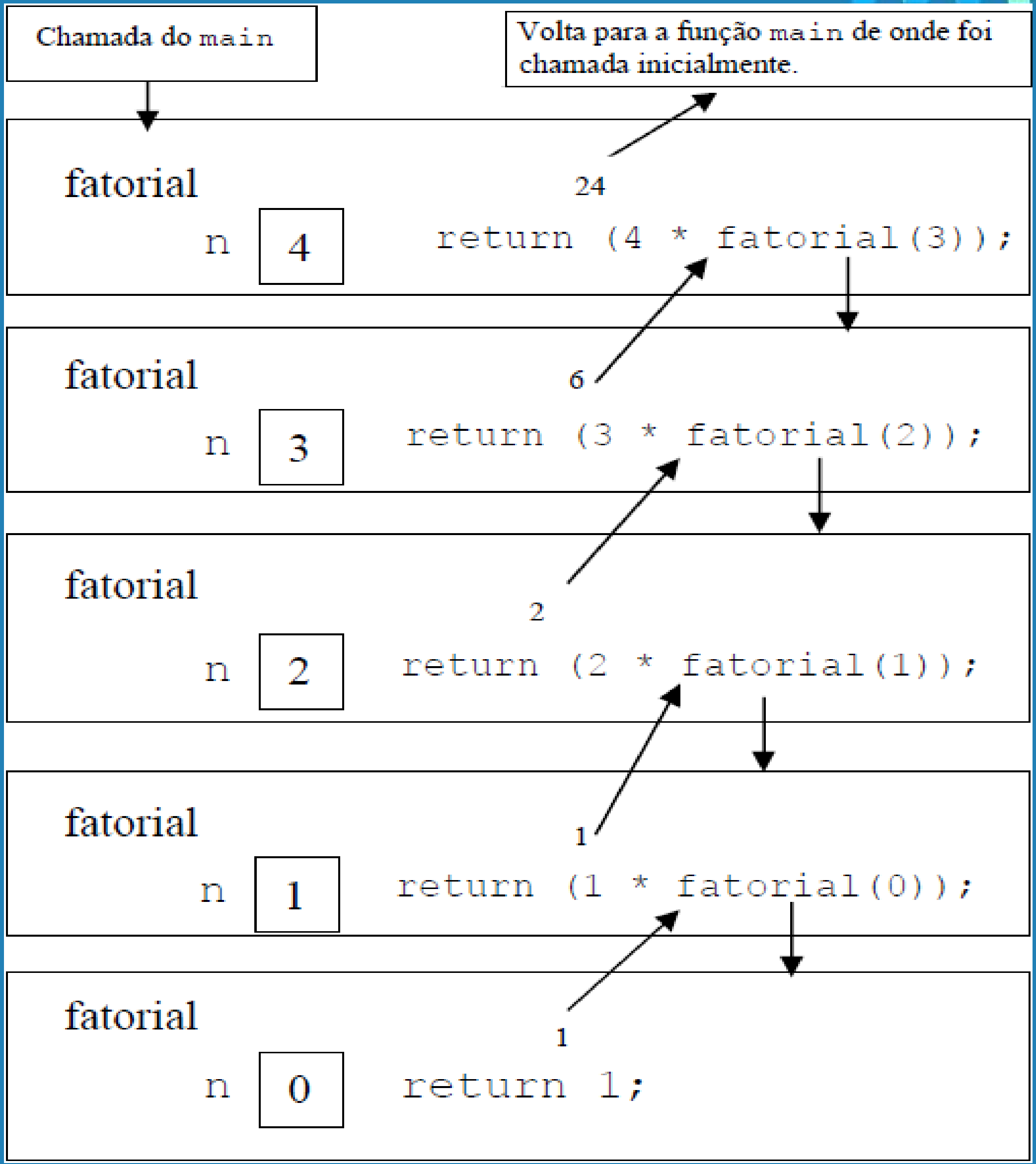
```
int fatorial(int n) {
  if (n == 0) // caso base
    return 1;
  return ( n * fatorial(n – 1) );
}
```

➔ Note que a função C++ seguiu a definição da função matemática fatorial, ficando com um formato bem próximo da especificação. Mas é importante não confundir a função matemática com o programa. O programa é uma implementação da função, tendo características próprias, como desempenho e restrições de entrada. Além disto, o programa não pode computar o fatorial para qualquer número passado como argumento, uma vez que está limitado ao maior número que o tipo `int` é capaz de representar.

Recursividade - fatorial

- Vejamos um esquema ilustrativo do que ocorre na memória quando a versão recursiva é chamada passando-se 4. Suponha que `fatorial` seja chamada da função `main`:

```
main() {
  ...
  fatorial(4);
 ...
}
```



- Exercícios:

1) Construa uma função recursiva que eleve *x* à potência *y*: x^y, *y* inteiro não negativo.

2) Construa uma função recursiva que receba um arranjo de números e os imprima na tela.

Recursividade – busca binária

➔ Outro exemplo interessante é o da *busca binária*. A busca binária consiste em um algoritmo eficiente para a realização de pesquisas em conjuntos de elementos *ordenados*.

➔ A ideia da busca binária é fazer algo semelhante ao que fazemos com uma lista telefônica. Ao invés de procurarmos um nome de forma sequencial, abrimos, por exemplo, no meio do catálogo e se notamos que o nome desejado vem antes (depois) dos nomes encontrados ao abrir, então eliminamos todos os nomes que vêm depois (antes) da parte que abrimos.

Recursividade – busca binária

➔ Abaixo mostramos uma versão iterativa do algoritmo para encontrar a posição de um número inteiro num arranjo *ordenado*:

```
int buscaBin(int * v, int x, int inicio, int fim) {
  int meio;
  while(inicio<=fim){
    meio=(inicio+fim)/2;
    if (x==v[meio])
      return meio; // encontrou elemento
    // eliminando metade das possibilidades
    if (x>v[meio])
      inicio=meio+1;
    else
      fim=meio-1;
  }
  return -1; // se não encontrar retorna -1
}
```

Recursividade – busca binária

- Agora a versão recursiva:

```
int buscaBinR(int * v, int x, int inicio, int fim) {
  if (inicio>fim) // 1o caso base
    return -1;
  int meio=(inicio+fim)/2;
  if (x==v[meio]) // 2o caso base
    return meio;
  if(x>v[meio])
    return buscaBinR(v, x, meio+1, fim);
  return buscaBinR(v, x, inicio,meio-1);
}
```

Recursividade – sequência de Fibonacci

➔ Outro exemplo clássico de algoritmo recursivo é a determinação do número de Fibonacci de ordem *n*. A sequência de Fibonacci é a seguinte:

```
nfib = 0 para n=0,
nfib = 1 para n=1,
nfib= nfib(n-2)+nfib(n-1) para n≥2.
```

O que resulta em:

n_0	n_1	n_2	n_3	n_4	n_5	n_6 ...
0	1	1	2	3	5	8 ...

➔ Vejamos primeiro uma versão iterativa.

Recursividade – sequência de Fibonacci

- Versão iterativa:

```
long double fibonacci(int n) {
  if (n==0) return 0;
  if (n==1) return 1;

  long double t1=0, t2=1, aux;
  int i=1;
  while(i<n){
    aux=t1;
    t1=t2;
    t2=aux+t2;
    i++;
  }
  return t2;
}
```

Recursividade – sequência de Fibonacci

- A versão recursiva fica ainda mais simples:

```
long double fibonacciR(int n){
  if (n==0) return 0;
  If (n==1) return 1;

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

Recursividade – sequência de Fibonacci

- A solução recursiva é direta e mais simples que a solução iterativa. Contudo, apresenta uma armadilha grave que deve ser levada em conta. Observe que a função refere-se a si mesma 2 vezes: `fibonacciR(n-1)` e `fibonaccirR(n-2)` e, neste caso, faz muitos cálculos repetidos.

- No cálculo de `fibonacci(100)`, por exemplo, a função necessita calcular independentemente `fibonacci(99)` e `fibonacci(98)`. Para calcular `fibonacci(99)`, a função vai calcular `fibonacci(98)` independente de ele já ter sido calculado antes. Assim, uma série de cálculos repetidos serão feitos.

Recursividade – sequência de Fibonacci

- Pode-se mostrar que o número de vezes que a função `fibonacciR` é chamada no cálculo de `fibonacciR(n)` é:

$$\text{fibonacciR}(n-1) + \left(\sum_{i=1}^{n} \text{fibonacciR}(i) \right)$$

- Para n=50, por exemplo, a função `fibonacciR` será chamada 40.730.022.147 vezes.

- De um modo geral, podemos representar as chamadas recursivas da função `fibonacciR` com a seguinte árvore:

***F*(*n*)**

***F*(*n*-1)** + ***F*(*n*-2)**

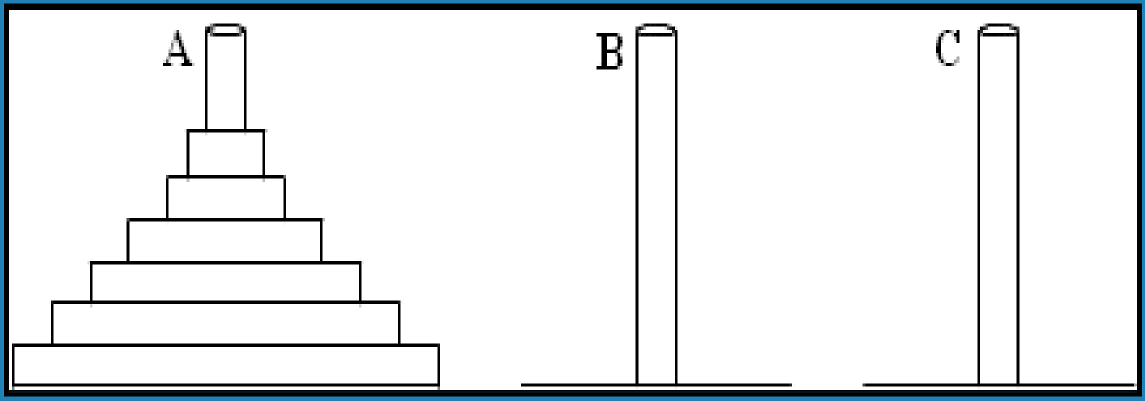
***F*(*n*-2)** + ***F*(*n*-3)** ***F*(*n*-3)** + ***F*(*n*-4)**

...

Recursividade – Torres de Hanoi

➔ Um outro problema muito conhecido com solução recursiva é o problema das *torres de Hanoi*.

➔ Este problema consiste de três pinos A, B e C, denominados de *origem*, *trabalho* e *destino,* e de *n* discos de diâmetros diferentes. Na configuração inicial do problema todos os discos encontram-se empilhados no pino A em ordem decrescente de tamanho. O objetivo é empilhar todos os discos no pino C, obedecendo as seguintes restrições: a) apenas um disco pode ser movido de cada vez e b) em nenhum momento um disco pode ser colocado sobre outro de tamanho menor. A figura abaixo mostra o problema para seis discos:



Recursividade – Torres de Hanoi

➔ Obviamente que queremos realizar a tarefa com o menor número de movimentos possível. Isto pode ser atingido com o seguinte algoritmo recursivo:

1. Mova *n*-1 discos do pino de origem para o pino de trabalho.
2. Mova o *n*-ésimo do pino de origem para o pino de destino.
3. Mova os *n*-1 discos do pino de trabalho para o pino de destino.

➔ Este algoritmo realiza exatamente 2^n-1 movimentos, que é provado ser o número mínimo.

➔ Como você deve ter notado, a ideia do algoritmo recursivo é tratar instâncias cada vez menores do problema até atingir o caso trivial, em que a solução é também trivial.

- A seguir uma implementação do algoritmo mostrado:

```
void hanoi(int n, char origem, char dest, char trab) {
  if (n>0) {
    hanoi(n-1,origem,trab,dest);
    cout << "Mova o disco do pino " << origem
         << "para " << dest << endl;
    hanoi(n-1,trab,dest,origem);
  }
}
```

- Faça o rastreio desta função para a chamada:

  `hanoi(4,'A','C','B').`

- Acesse: `http://www.mazeworks.com/hanoi/index.htm` para ver uma animação do problema das torres de Hanoi.

Exercícios

1) Mostre a sequência de chamadas recursivas do algoritmo `buscaBinR` (a exemplo do que fizemos para a função recursiva `fatorial`) no arranjo abaixo, para buscar o valor 23:

```
5   12   23   45   63   89   114   255   368
```

2) Mostre a sequência de chamadas recursivas do algoritmo `fibonacciR` (a exemplo do que fizemos para a função recursiva `fatorial`) para *n* = 5.

3) Mostre a sequência de chamadas recursivas do algoritmo `hanoi` (a exemplo do que fizemos para a função recursiva `fatorial`), considerando a chamada inicial: `hanoi(4,'A','C','B')`.

4) Mostre para os casos dos exercícios 2 e 3 a árvore de chamadas recursivas com os nodos numerados para indicar a ordem em que as chamadas são feitas.