Estrutura de Dados I

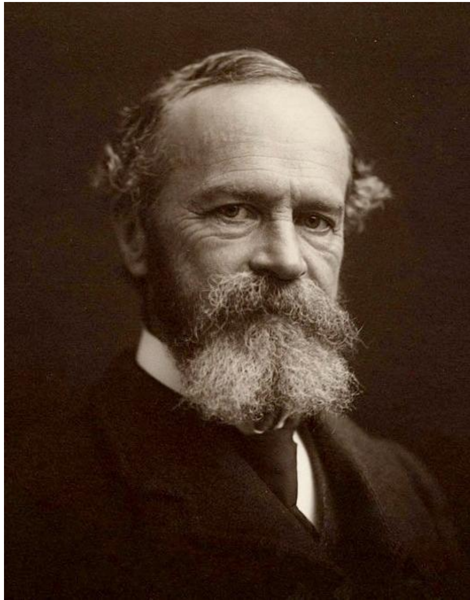
Capítulo 4: Introdução à Recursão

2025/1

Prof. Abrantes Araújo Silva Filho

E muitas vezes, nossa fé prévia em um determinado resultado é a única coisa que faz com que o resultado se torne realidade.

William James, *The Will To Believe*, 1897



Recursão

- É uma estratégia poderosa de resolução de problemas complexos

- A característica fundamental da recursão é a de que **problemas grandes e complexos são solucionados reduzindo-os à versões menores de problemas com a mesma forma**.

- Se os problemas forem reduzidos à versões menores de problemas com outras formas, não temos recursão! O que torna a recursão especial é que os subproblemas em uma solução recursiva têm a mesma forma do problema original, mas são menores.

Recursão

- Não é um conceito fácil

- Não têm muitos exemplos na vida real para fazermos uma analogia simples na programação.

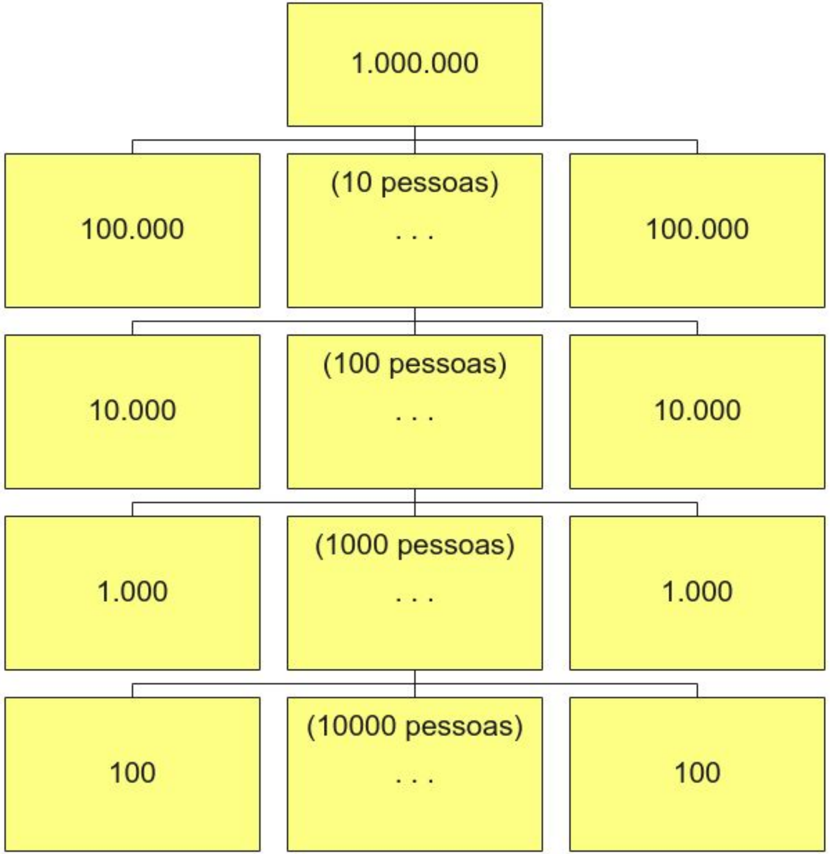
- Precisa de muito tempo e prática.

Recursão: exemplo na vida real

- Você está coordenando uma campanha de arrecadação de fundos para uma instituição de caridade que precisa de R$ 1.000.000,00. A contribuição média que as pessoas costumam fazer para a instituição é de R$ 100,00. Como você vai coletar R$ 1.000.000,00?

- Se 1 pessoa estiver disposta a pagar R$ 1.000.000,00 então o problema está resolvido. Mas dificilmente uma única pessoa estaria disposta a resolver esse problema tão grande. Você então vai adotar a seguinte estratégia:
  - Buscar 10 pessoas que arrecadarão R$ 100.000,00 cada um. Essas pessoas serão os coordenadores regionais.
  - Cada coordenador regional buscará 10 coordenadores locais, que arrecadarão R$ 10.000,00 cada um
  - O processo de delegação continua até que um certo número de pessoas seja responsável por arrecadar R$ 100,00 (que é contribuição média e fácil de obter).

Recursão: exemplo na vida real



1 Coordenador geral

10 Coordenadores regionais

100 Coordenadores locais

1.000 Coordenadores locais

10.000 Coletores

Recursão: exemplo na vida real

- Note que cada delegação do problema é simplesmente o problema original com a mesma forma, em uma escala menor. As delegações criaram **subproblemas menores com a mesma forma do original**:
    - coletar 1.000.000
    - coletar 100.000
    - coletar 10.000
    - coletar 1.000
    - coletar 100

- Em pseudocódigo teríamos:

Recursão: exemplo na vida real

```
void coletar_contribuicoes(int valor)
{
    if (valor <= 100)
    {
        // Coletar a contribuição
    }
    else
    {
        // Encontrar mais 10 pessoas
        // Fazer com que cada pessoa colete (valor/10) reais
        // Combinar o dinheiro arrecadado pelas pessoas
    }
}
```

Problema menor com a mesma forma do original.

Recursão: exemplo na vida real

```
void coletar_contribuicoes(int valor)
{
    if (valor <= 100)
    {
        // Coletar a contribuição
    }
    else
    {
        // Encontrar mais 10 pessoas
        coletar_contribuicoes(valor/10);
        // Combinar o dinheiro arrecadado pelas pessoas
    }
}
```

Como a forma é a mesma, podemos resolver chamando a mesma função

Recursão: entendimento inicial

- Em termos simples, costumamos dizer que uma das características mais marcantes da recursão é ter um **subprograma que chama a si mesmo**.

- Mas atenção: o que realmente define uma solução recursiva é aquela que **quebra o problema original em subproblemas menores com a mesma forma**!

Recursão: entendimento inicial

- Em geral um subprograma recursivo tem uma forma semelhante à:

```
/* tipo */ subprograma_recursivo (/* parâmetros */)
{
    if (/* teste para o caso simples */)
    {
        // Obter uma solução simples SEM USAR recursão
    }
    else
    {
        // Quebrar o problema em subproblemas menoras com a mesma forma
        // Resolver cada subproblema chamando o próprio subprograma_recursivo
        // Combinar as soluções dos subproblemas na solução unificada do todo
    }
}
```

Recursão: entendimento inicial

- Para soluções recursivas, você deve seguir o **PARADIGMA RECURSIVO**:
    - **Identifique os CASOS SIMPLES para os quais a resposta é obtida sem usar recursão;**
    - **Identifique a DECOMPOSIÇÃO RECURSIVA, que permitirá quebrar o problema complexo em subproblemas menores da mesma forma.**

Exemplo: fatorial

- **n**! corresponde ao produto dos inteiros entre 1 e **n**.
- Por definição, 0! = 1

- Implementação iterativa:

$$n! = n \times (n-1) \times (n-2) \times \ldots \times 2 \times 1$$

Exemplo: fatorial (iterativo)

```c
/**
 * Função: fat
 * Uso: n = fat(x)
 * ---------------
 * Esta função retorna o fatorial de um número inteiro x. Caso x < 0 retorna -1
 * para indicar que estamos tentando calcular o fatorial de um número negativo.
 * O tipo de retorno é int.
 */

int fat (int n)
{
    int res = 1;

    if (n < 0)
        return -1;
    else if (n == 0)
        return 1;
    else
        for (int i = 1; i <= n; i++)
            res *= i;

    return res;
}
```

$$n! = n \times (n-1)!$$

Exemplo: fatorial (recursivo)

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

```c
/**
 * Função: fat
 * Uso: n = fat(x)
 * ----------------
 * Esta função retorna o fatorial de um número inteiro x. Caso x < 0 retorna -1
 * para indicar que estamos tentando calcular o fatorial de um número negativo.
 * O tipo de retorno é int.
 */

int fat (int n)
{
    if (n < 0)
        return -1;
    else if (n == 0)
        return 1;
    else
        return n * fat(n - 1);
}
```

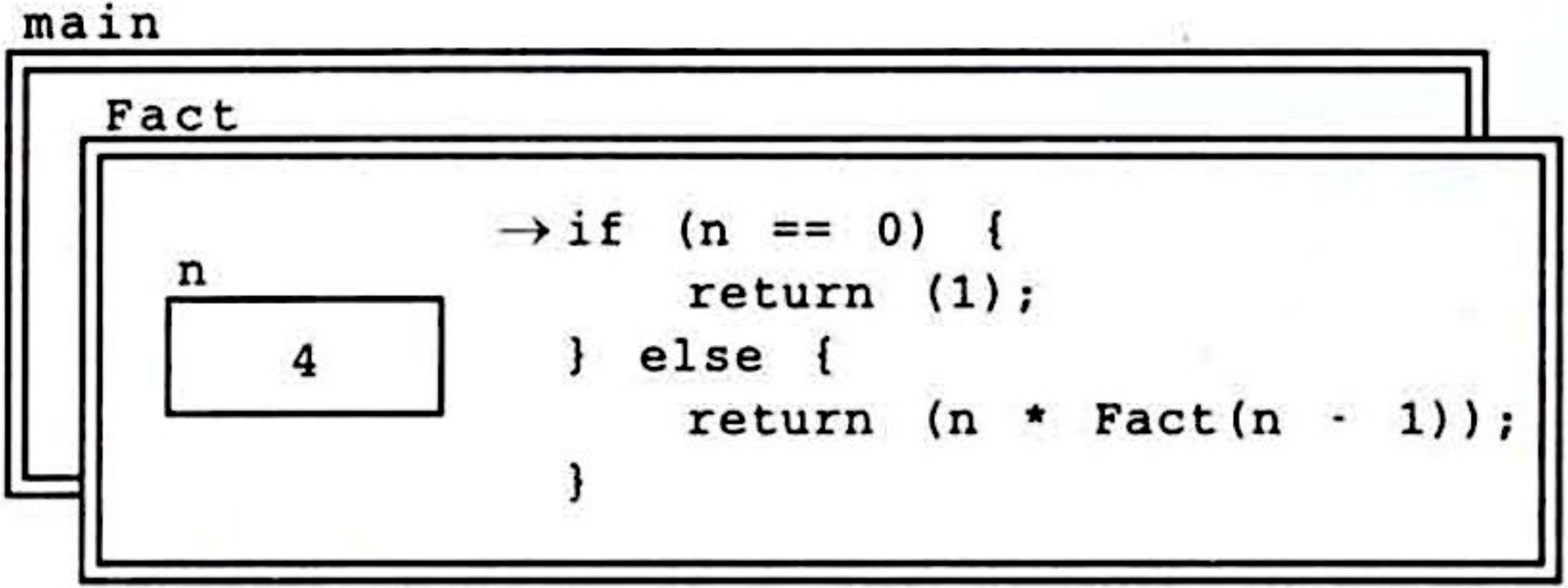
casos simples

decomposição recursiva

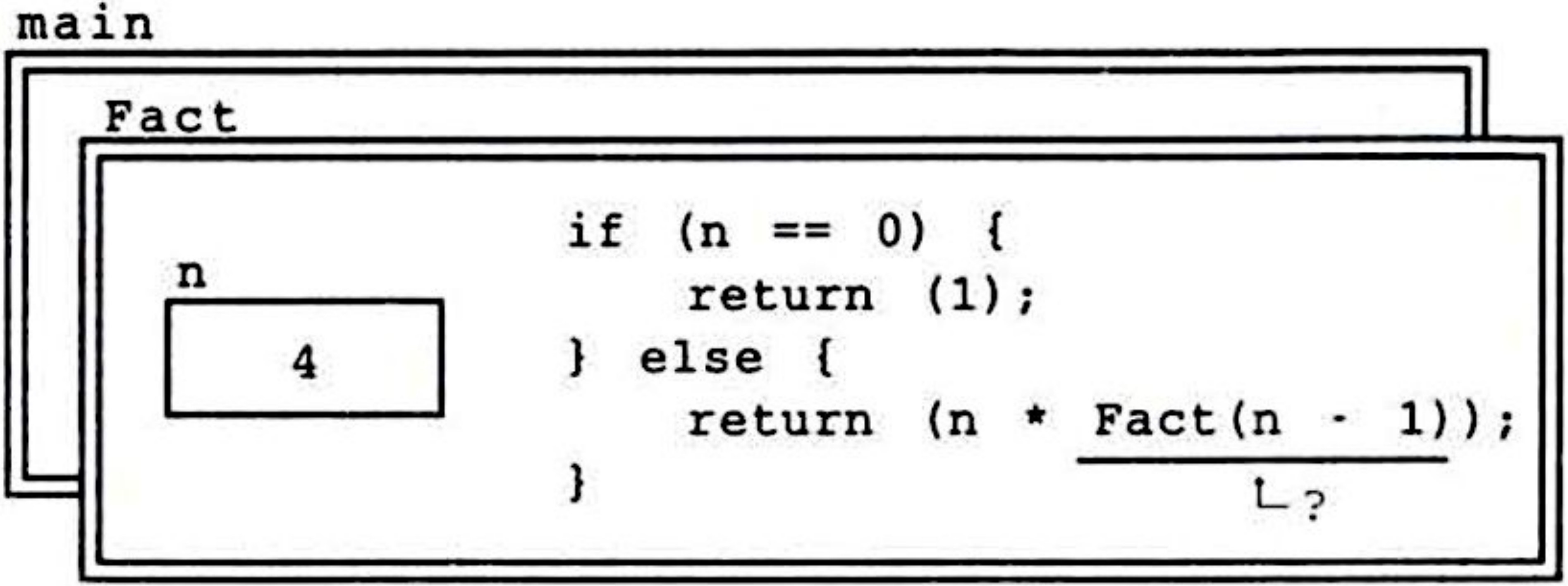
Exemplo: fatorial (recursivo)

- Onde os cálculos realmente ocorrem no fatorial recursivo?
- Os passos estão escondidos. Parece mágica.
- O segredo está no stack:

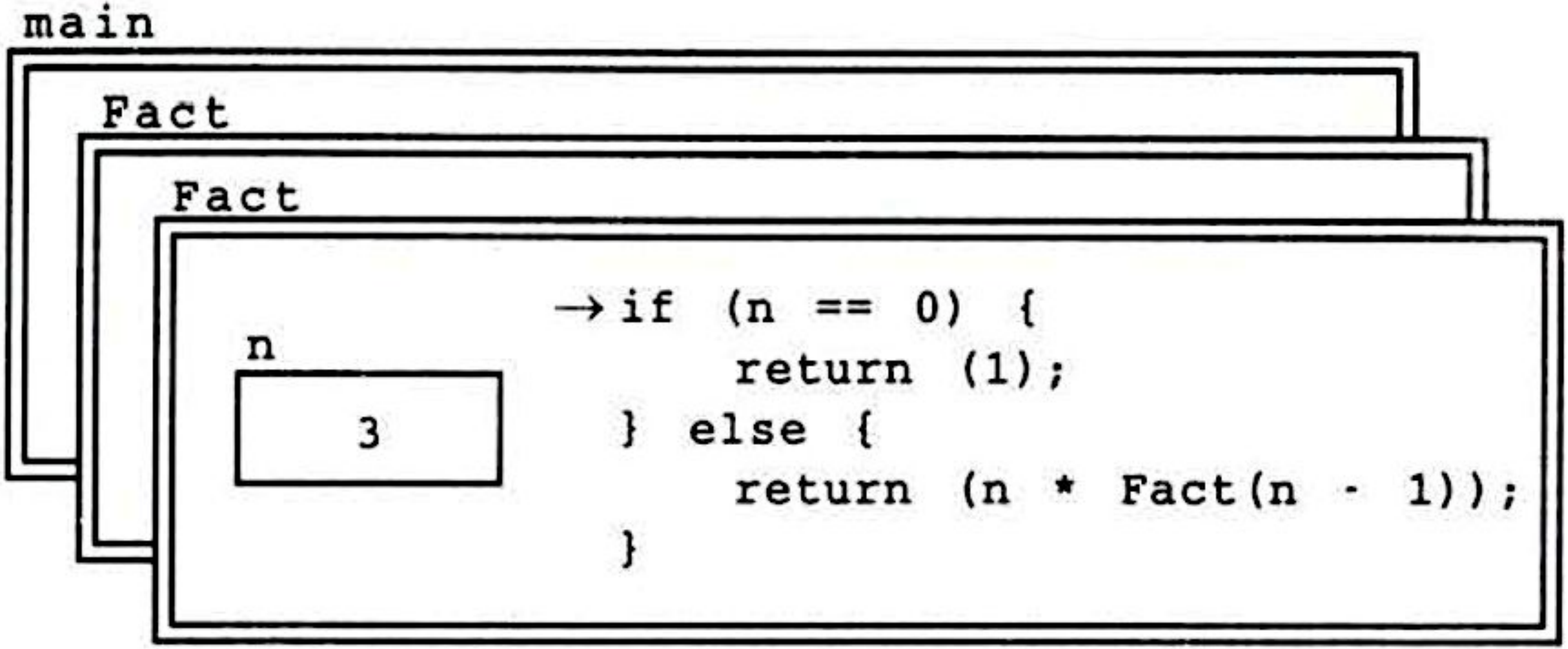
Exemplo: fatorial (recursivo)



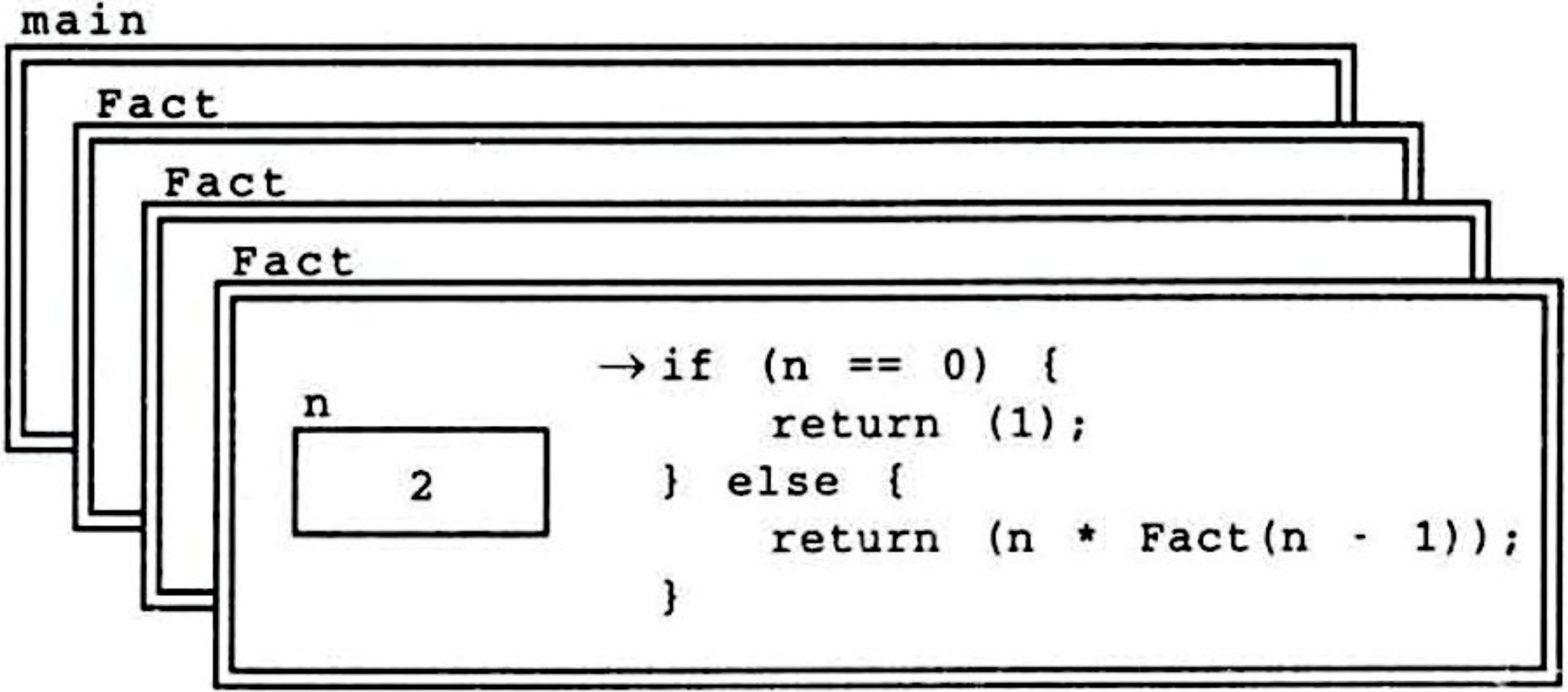
Exemplo: fatorial (recursivo)



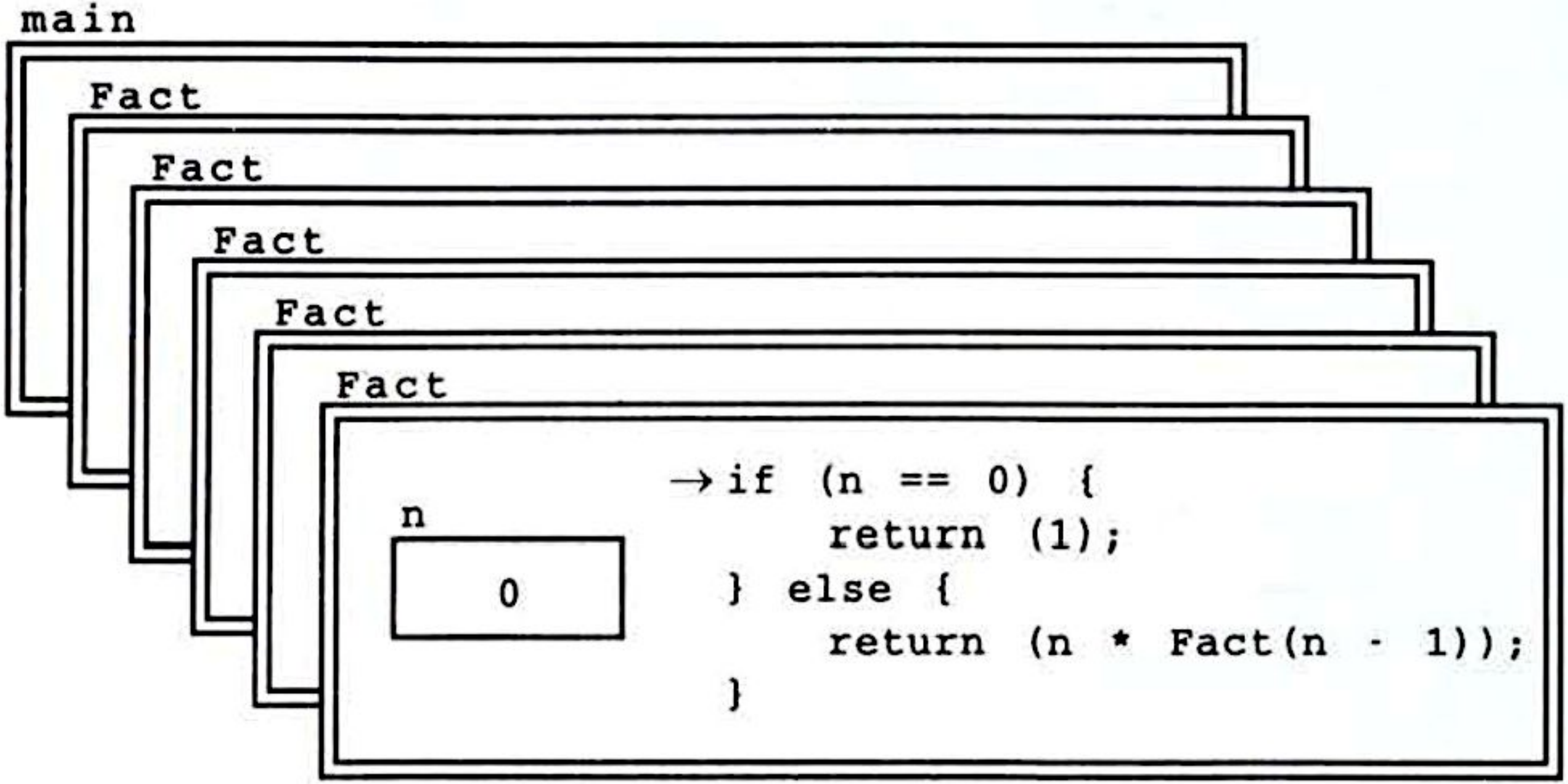
Exemplo: fatorial (recursivo)



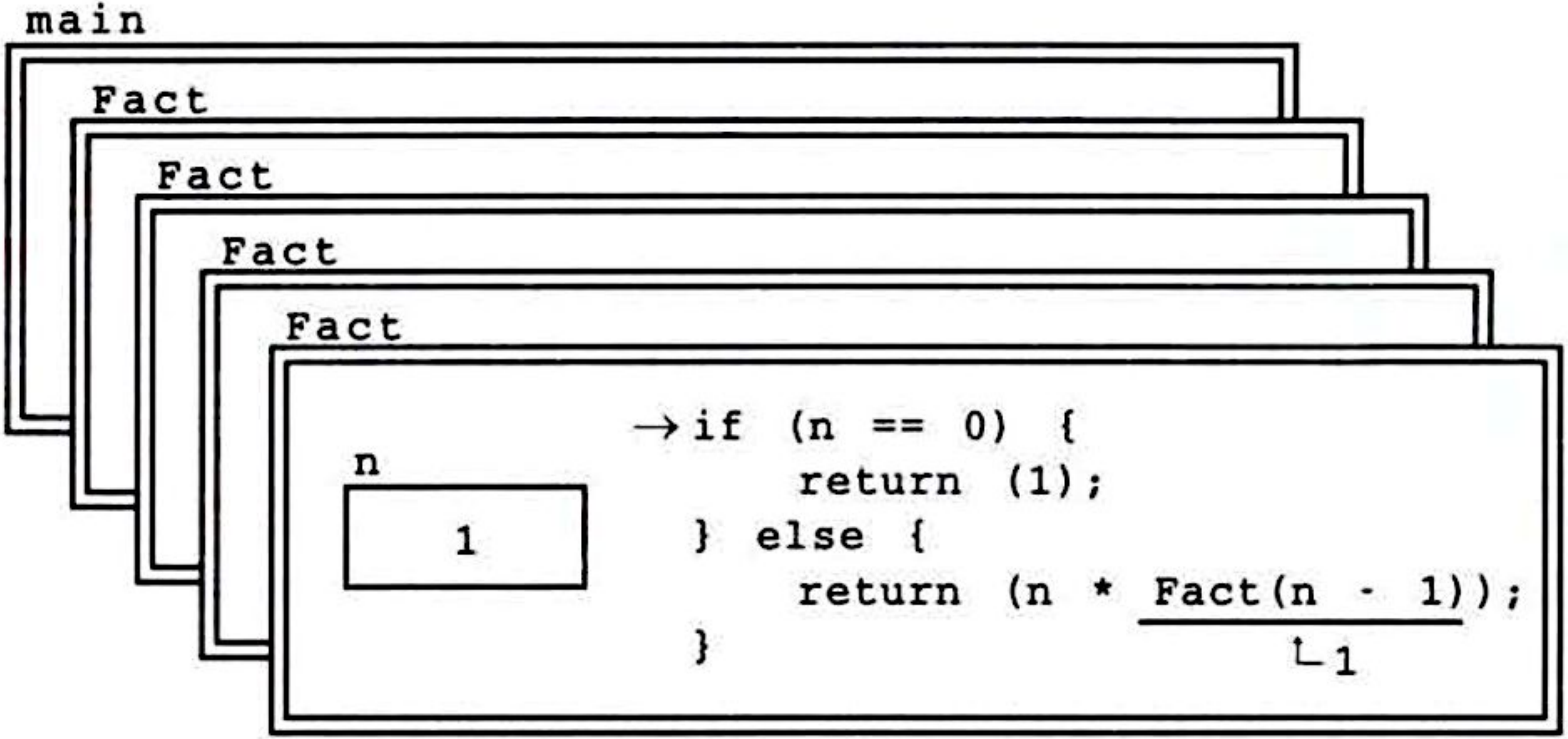
Exemplo: fatorial (recursivo)



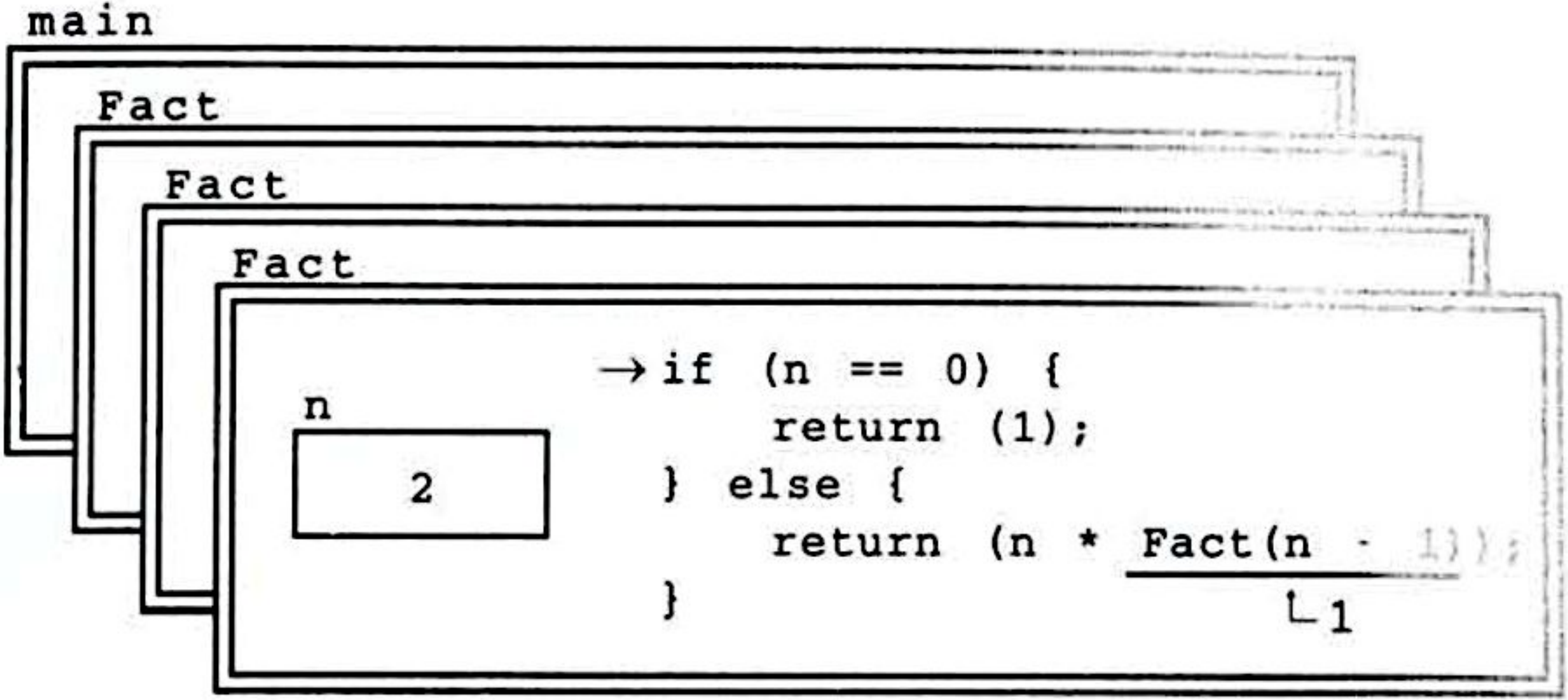
Exemplo: fatorial (recursivo)



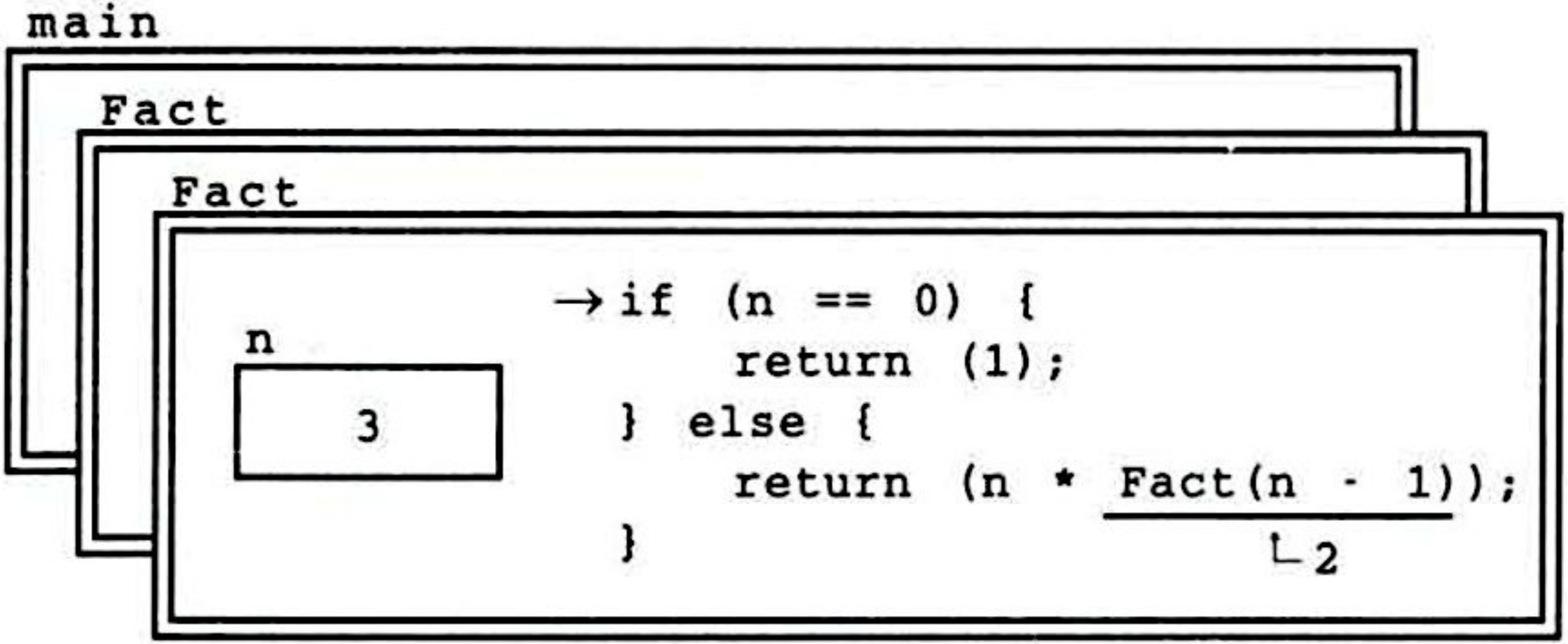
Exemplo: fatorial (recursivo)



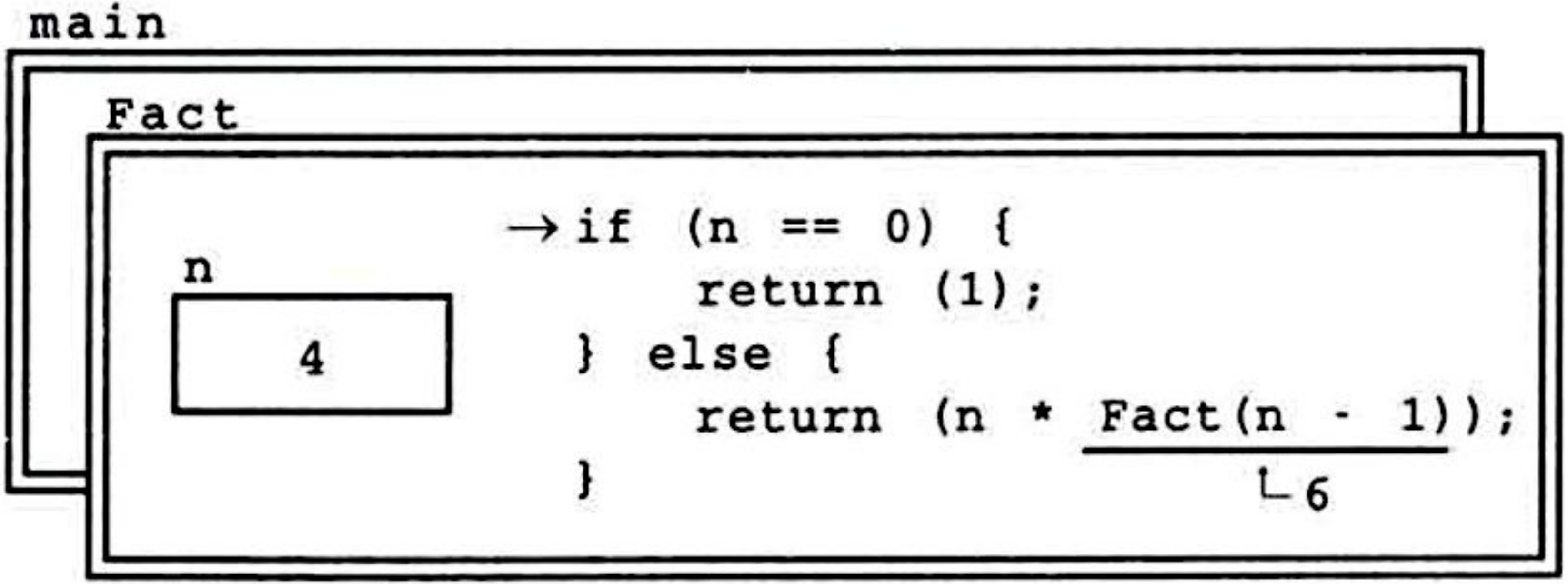
Exemplo: fatorial (recursivo)



Exemplo: fatorial (recursivo)



Exemplo: fatorial (recursivo)



O **SALTO DE FÉ** recursivo

- O entendimento de como a função fatorial funciona através de chamadas no stack teve como objetivo principal mostrar que o computador trata as chamadas de funções recursivas como qualquer outra função comum.

- Quando você escreve ou tenta entender um programa recursivo, você deve:
  - **IGNORAR OS DETALHES DE COMO AS CHAMADAS RECURSIVAS FUNCIONAM**, não fique pensando nos detalhes internos de funcionamento;

  - **SE FOCAR APENAS EM UM ÚNICO NÍVEL DE OPERAÇÃO**:
    - Apenas **TENHA FÉ** de que **qualquer chamada recursiva automaticamente obterá a resposta correta**, desde que os argumentos para a chamada recursiva sejam mais simples do que o problema original e que tenham a mesma forma que o problema original; não fique pensando nas chamadas recursivas posteriores!

O **SALTO DE FÉ** recursivo

- Considere que você quer calcular:
  fat(n) onde n = 4.
- A implementação recursiva deve calcular
  n * fat(n - 1)
- Substituindo o valor de n, temos então que:
  4 * fat(3)
- **PARE AGORA**! Como **fat(3) é um subproblema menor que fat(4)**, e como **fat(3) tem a mesma forma que fat(4)**, **o salto de fé recursivo nos permite assumir que fat(3) simplesmente funciona e retornará o resultado correto**. Eu **não preciso me preocupar em como fat(3) faz isso**!

O **SALTO DE FÉ** recursivo

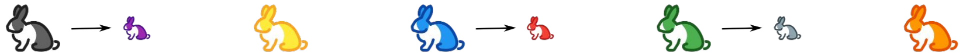
- Ao criar programas recursivos seu foco deve estar no macro ao invés dos detalhes. Você só precisa garantir que seguiu o paradigma recursivo:
    - identificou os casos simples
    - identificou a decomposição recursiva

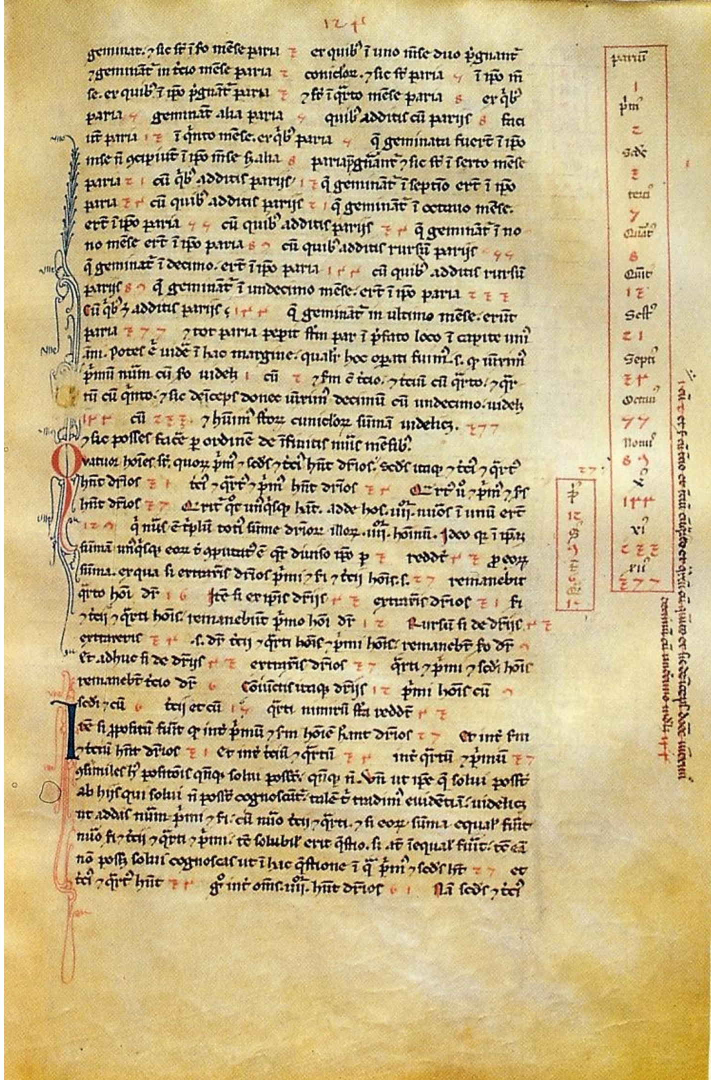
- **Se você seguiu o paradigma recursivo, dê o SALTO DE FÉ e confie que o computador vai fazer a coisa certa**.

Outro exemplo: Fibonacci

- Leonardo de Pisa (c. 1170 - c. 1250)
  - Matemático Italiano da República de Pisa
  - “O mais talentoso matemático ocidental mais talentoso da Idade Média”
  - Popularizou os algarismos Indo-Arábicos no ocidente através de seu livro, “Liber Abacci” (Livro de Cálculo), publicado em 1202
  - Foi também no “Liber Abacci” que a Seqüência de Fibonacci foi introduzida, como um exercício de biologia populacional.

Outro exemplo: Fibonacci

Tempo	Pares de coelhos bebês e adultos	Pares
t_0		0
t_1		1
t_2		1
t_3		2
t_4		3
t_5		5
t_6		8



Outro exemplo: Fibonacci

t	0	1	2	3	4	5	6	7	8	9	10	11	12
fib(t)	0	1	1	2	3	5	8	13	21	34	55	89	144

- Percebe-se então que:

$$t_n = t_{n-1} + t_{n-2}$$

- Uma expressão do tipo acima, onde **cada elemento de uma seqüência é definido em termos de elementos anteriores** é chamada de **relação de recorrência**.

- Note que a expressão acima não é suficiente para definir a Seqüência de Fibonacci. Por quê???

Outro exemplo: Fibonacci

$$t_n = \begin{cases} n & \text{se } n = 0 \text{ ou se } n = 1 \\ t_{n-1} + t_{n-2} & \text{caso contrário} \end{cases}$$

- A relação de recorrência acima é completa e define toda a seqüencia.
- A única coisa que você tem que fazer para implementar um algoritmo recursivo é:
    - Testar para casos simples
    - Decomposição recursiva
    - Salto de fé recursivo

Outro exemplo: Fibonacci

- Pense no paradigma recursivo:
  - Quais seriam os **casos simples**?
  - Qual é a **decomposição recursiva**?

```
/* tipo */ subprograma_recursivo (/* parâmetros */)
{
    if (/* teste para o caso simples */)
    {
        // Obter uma solução simples SEM USAR recursão
    }
    else
    {
        // Quebrar o problema em subproblemas menoras com a mesma forma
        // Resolver cada subproblema chamando o próprio subprograma_recursivo
        // Combinar as soluções dos subproblemas na solução unificada do todo
    }
}
```

Outro exemplo: Fibonacci

- Casos simples:
  - n = 0 e n = 1;
  - acréscimo de n < 0 para indicar erro:

```
if (n < 0)
    return -1;
else if (n < 2)
    return n;
```

Outro exemplo: Fibonacci

- Decomposição recursiva:
  - é o que transforma o problema em subproblemas menores da mesma forma, ou seja, é a soma de fib(n -1) + fib(n - 2), chamando a função de forma recursiva:

```
return fib_rec(n - 1) + fib_rec(n - 2);
```

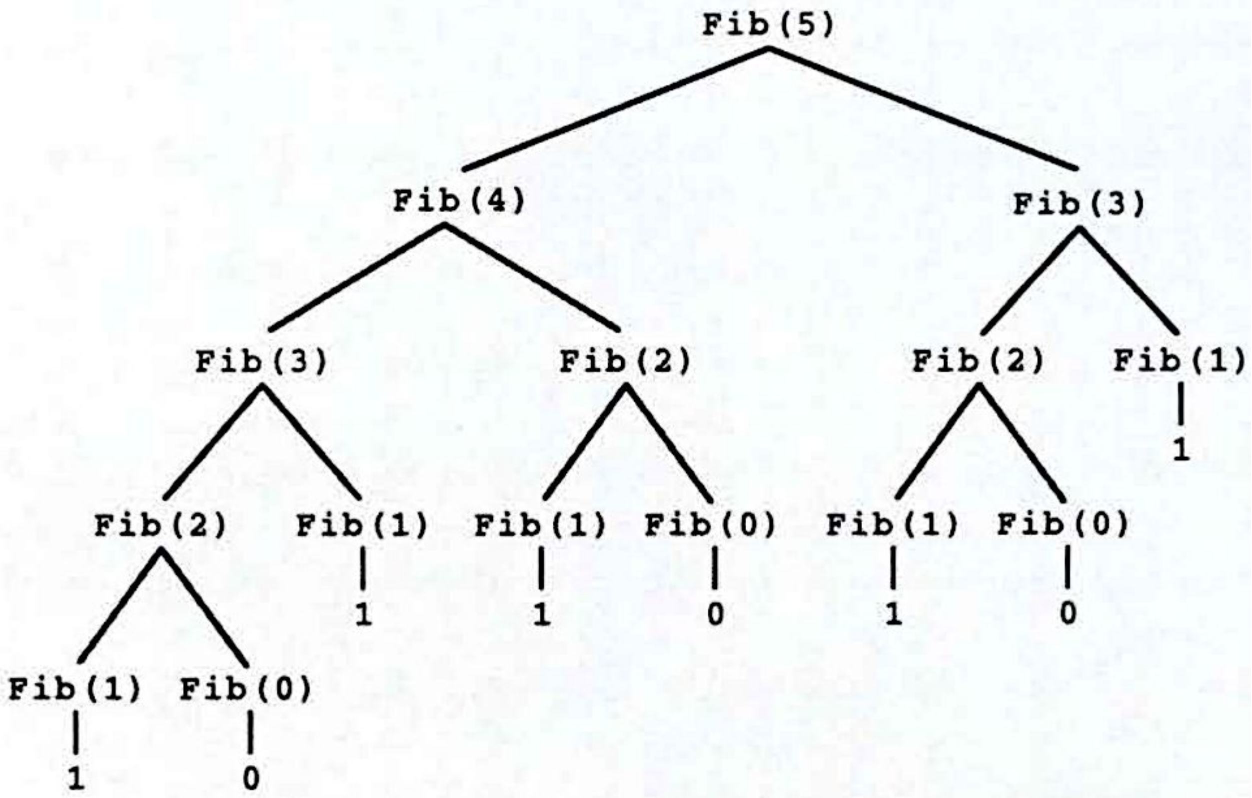
- Como saber se a decomposição recursiva está correta? Considere um nível do problema, por exemplo, calcular fib(5):
  - como fib(5) = fib(4) + fib(3); e
  - como fib(4) e fib(3) são subproblemas menores da mesma forma; então
  - podemos usar o **SALTO DE FÉ recursivo** e assumir que o programa obterá o valor correto de todos esses cálculos, sem ter que prestar atenção aos detalhes de cada nível.

Outro exemplo: Fibonacci

```c
/**
 * Função fib_rec
 * Uso: t = fib_rec(n);
 * --------------------
 * Esta função recebe um número inteiro "n" >= 0, e retorna o n-ésimo termo da
 * Seqüência de Fibonacci, utilizando uma implementação recursiva da relação
 * de recorrência:
 *
 *     fib_rec(n) = fib_rec(n - 1) + fib_rec(n - 2)
 *
 * onde: fib_rec(0) = 1
 *        fib_rec(1) = 1.
 *
 * Se o usuário informar um valor inválido (n < 0) a função retorna "-1" como
 * um valor sentinela informativo de input inválido.
 */

int fib_rec (int n)
{
    if (n < 0)
        return -1;
    else if (n < 2)
        return n;
    else
        return fib_rec(n - 1) + fib_rec(n - 2);
}
```

Fibonacci: ineficiência da solução recursiva



Fibonacci: ineficiência da solução recursiva

- A ineficiência da solução recursiva do Fibonacci não tem nada a ver com a recursão por si mesma mas, sim, com o modo que a recursão está sendo utilizada.

- Frequentemente **a chave para encontrar uma solução mais eficiente é encontrar uma abordagem mais geral**.

Seqüências: abordagem mais geral

- Note que a Seqüência de Fibonacci não é a única seqüência que pode ser definida em termos da relação de recorrência abaixo:

$$t_n = t_{n-1} + t_{n-2}$$

- Dependendo da escolha dos 2 primeiros termos, teremos diversas dessas seqüências, por exemplo:

```
 0 1  1  2  3  5  8  13  21  34  55  89  144 ...
 3 7 10 17 27 44 71 115 186 301 487 788 1275 ...
-1 2  1  3  4  7 11  18  29  47  76 123  199 ...
```

- Essas seqüências são chamadas de **seqüências aditivas**, pois utilizam a mesma relação de recorrência e só diferem pelos termos iniciais.

Seqüências: abordagem mais geral

- Podemos **transformar** o problema de obter o n-ésimo número da Seqüência de Fibonacci **no problema mais geral** de **encontrar o n-ésimo número de uma seqüência aditiva que começa com os termos iniciais t_0 e t_1**. Como criar essa função mais geral?

- Os **casos simples** são dados pelos termos t_0 e t_1, que são passados como argumentos, já que o usuário pode escolher os termos de início:

```
int seq_adit (int n, int t0, int t1);
```

o n-ésimo termo a ser retornado também é passado como argumento.

Seqüências: abordagem mais geral

- Agora considere que queremos achar t_6 da seqüência aditiva que começa com os termos 3 e 7: $t_6 = 71$

t	0	1	2	3	4	5	6	7	8	9	10	11	12
f(t)	3	7	10	17	27	44	71	115	186	301	487	788	1275

- Para calcular t_6, o **PULO DO GATO** é perceber que o **n-ésimo termo de qualquer seqüência aditiva é o (n-1)-ésimo termo da seqüência aditiva que começa um passo adiante**: essa é a **decomposição recursiva**!

t	0	1	2	3	4	5	6	7	8	9	10	11	12
f(t)	7	10	17	27	44	71	115	186	301	487	788	1275	2063

Seqüências: abordagem mais geral

- Perceba as mudanças de uma situação para outra:

t	0	1	2	3	4	5	6	7	8	9	10	11	12
f(t)	3	7	10	17	27	44	71	115	186	301	487	788	1275

t	0	1	2	3	4	5	6	7	8	9	10	11	12
f(t)	7	10	17	27	44	71	115	186	301	487	788	1275	2063

- O novo n saiu de 6 para 5, então n = n - 1;
  O novo t_0 passou a ser o t_1 original; e
  O novo t_1 é a soma dos valores originais de t_0 e t_1.

Seqüências: abordagem mais geral

```c
/**
 * Função: seq_adit
 * Uso: n = seq_adit(n, t0, t1);
 * ----------------------------
 * Esta função recebe 3 números inteiros: t0 e t1 são os dois primeiros números
 * de uma implementação recursiva da relação de recorrência (seqüência aditiva)
 *
 *     seq_adit(n) = seq_adit(n - 1) + seq_adit(n - 2)
 *
 * e n é um número inteiro tal que n >= 0. A função retorna o n-ésimo número
 * da seqüência aditiva gerada utilizando-se como ponto de partida t0 e t1,
 * utilizando como estratégio o fato de que: o n-ésimo termo de uma seqüência
 * aditiva que começa em t0 e t1, é igual ao (n-1)-ésimo termo de uma seqüência
 * aditiva que começa um passo adiante.
 */

int seq_adit (int n, int t0, int t1)
{
    if (n < 0)  return -1;
    if (n == 0) return t0;
    if (n == 1) return t1;
    return seq_adit(n - 1, t1, t0 + t1);
}
```

Onde está o SALTO DE FÉ recursivo?

Seqüências: abordagem mais geral

- Como usar seq_adit para calcular o n-ésimo fatorial então?

```
n = seq_adit(6, 0, 1);
```

- Geralmente não usamos as funções genéricas como seq_adit diretamente: em programação recursiva costumamos criar as famosas **funções wrappers** (**invólucros**) que simplesmente retornam o resultado de outra função (podendo fazer alterações nos argumentos):

```
int fib (int n)
{
    if (n < 0) return -1;
    return seq_adit(n, 0, 1);
}
```

Seqüências: abordagem mais geral

- Na maioria das vezes **uma função wrapper é utilizada para fornecer argumentos adicionais para uma função auxiliar que resolve um problema mais geral**:

```c
/**
 * Função: fib
 * Uso: n = fib(n);
 * ----------------
 * Esta função é um wrapper (invólucro) que simplesmente retorna o resultado de
 * outra função mais geral (seq_adit) para o cálculo do n-ésimo termo da
 * Seqüência de Fibonacci.
 */

int fib (int n)
{
    if (n < 0) return -1;
    return seq_adit(n, 0, 1);
}
```

Seqüências: abordagem eficiente

```
fib(5)
    = seq_adit(5, 0, 1)
        = seq_adit(4, 1, 1)
            = seq_adit(3, 1, 2)
                = seq_adit(2, 2, 3)
                    = seq_adit(1, 3, 5)
                        = 5
```

```
int fib (int n)
{
    if (n < 0) return -1;
    return seq_adit(n, 0, 1);
}
```

Outro exemplo: palíndromos

- É fácil verificar se uma palavra ou uma string é um palíndromo iterando através de seus caracteres. Mas um palíndromo também pode ser definido recursivamente. O PULO DO GATO aqui é perceber que **qualquer palíndromo maior do que 1 caractere contém um palíndromo menor em seu interior**:

```
level -> eve

noon -> oo
```

Essa é a **decomposição recursiva**!

Outro exemplo: palíndromos

- E quais seriam os **casos simples**?

Qualquer string com 1 caractere e também a string vazia!

```
“level” -> “eve” -> “v”

“noon” -> “oo” -> “”
```

Outro exemplo: palíndromos (não tão eficiente)

```
bool e_palindromo (const string str)
{
    int len;

    len = StringLength(str);
    if (len <= 1)
        return TRUE;
    else
        return (IthChar(str, 0) == IthChar(str, len - 1)
                && e_palindromo(SubString(str, 1, len -2)));
}
```

Outro exemplo: palíndromos (eficiente)

```
bool palindromo_p (const string str)
{
    return checa_palindromo(str, StringLength(str));
}
```

```
static bool checa_palindromo (const string str, const int tam)
{
    if (tam <= 1)
        return TRUE;
    else
        return (str[0] == str[tam - 1]
                && checa_palindromo(str + 1, tam - 2));
}
```

Mais um exemplo: busca binária

- Você já está familiarizado com a busca binária iterativa. Mas também é possível implementar uma versão recursiva!

- Vamos implementar um algoritmo recursivo de busca binária para encontrarmos uma string em um array de strings ordenado lexicograficamente (ordem ASCIIbética).

Mais um exemplo: busca binária

```
/**
 * Função: busca_binaria
 * Uso: n = busca_binaria(chave, str[], inf, sup);
 * -----------------------------------------------
 * Esta função implementa um algoritmo de busca binária recursiva para encontrar
 * uma determinada string (chave) em um array de strings (str) que esteja
 * ordenado lexicograficamente (ordem "ASCIIbética"). Se a chave é encontrada,
 * a função retorna o índice da posição no array de strings (str) na qual a
 * chave está (se a chave aparece mais de uma vez no array, qualquer um dos
 * índices pode ser retornado). Se a chave não existe no array, a função retorna
 * o valor -1.
 */

static int busca_binaria (const string chave, const string str[],
                          int inf, int sup)
{
    int meio, comp;

    if (inf > sup) return -1;
    meio = (inf + sup) / 2;
    comp = StringCompare(chave, str[meio]);
    if (comp == 0) return meio;
    if (comp < 0)
        return busca_binaria(chave, str, inf, meio - 1);
    else
        return busca_binaria(chave, str, meio + 1, sup);
}
```

Quais os casos simples?
Qual é a decomposição recursiva utilizada?

Último exemplo: recursão mútua

- A característica fundamental da recursão é a de que **problemas grandes e complexos são solucionados reduzindo-os à versões menores de problemas com a mesma forma**.

- Geralmente isso é feito com um subprograma chamando a si mesmo, mas isso NÃO QUER DIZER que a chamada tem de ocorrer diretamente: a chamada recursiva pode estar em um nível inferior de aninhamento de chamadas de funções.
    - Se a função F chama a função G, e a função G chama a função F, essas chamadas são recursivas!
    - Como F e G chamam uma à outra, esse tipo de recursão é dito **recursão mútua**.

Último exemplo: recursão mútua

- Como podemos usar recursão para testar se um número natural n >= 0 é par ou ímpar?
  - Quais os casos simples?
  - Qual a decomposição recursiva?

- Algumas considerações:
  - Um número é par se seu antecessor é ímpar
  - Um número é ímpar se não for par
  - O número 0 é par por definição

Último exemplo: recursão mútua

```
/**
 * Predicado: e_par
 * Uso: if (e_par(n)) . . .
 * -------------------------
 * Este predicado retorna TRUE se o inteiro n for par. O número 0 é considerado
 * par por definição; qualquer outro número é par se o seu antecessor for ímpar.
 * O predicado só aceita argumentos unsigned.
 */

bool e_par(unsigned int n)
{
    if (n == 0)
        return TRUE;
    else
        return e_impar(n - 1);
}
```

Último exemplo: recursão mútua

```
/**
 * Predicado: e_impar
 * Uso: if (e_impar(n)) . . .
 * ---------------------------
 * Este predicado retorna TRUE se n é ímpar, onde um número é definido como
 * ímpar se ele não for par. O predicado só aceita argumentos unsigned.
 */

bool e_impar(unsigned int n)
{
    return (!e_par(n));

    // Uma outra implementação possível:
    #if 0
    if (n == 0)
        return FALSE;
    else
        return (e_par(n - 1));
    #endif
}
```

Pensar recursivamente: é DIFÍCIL

- Recursividade é um conceito difícil de entender, difícil de aprender e difícil de implementar.

- Você terá que PRATICAR MUITO e pensar de modo completamente diferente do que você está acostumado.

- Holismo x Reducionismo: na programação em geral deve haver um balanço entre essas duas visões, mas na recursividade o que importa é o holismo:
    - Reducionismo: é a crença de que o todo de um objeto pode ser compreendido através da compreensão separada de suas partes;
    - Holismo: é a crença de que o todo é maior do que a soma das partes.

Pensar recursivamente: é DIFÍCIL

- Para manter uma perspectiva holística:
  - Adote o **SALTO DE FÉ** recursivo:
    - Se você **identificou os casos simples**, identificou a **decomposição recursiva** e **implementou sua estratégia** corretamente, **ignore completamente os detalhes das chamadas recursivas, elas irão simplesmente funcionar**… não fique pensando sobre elas!
    - Para nosso azar é difícil identificar os casos simples e/ou a decomposição recursiva até que tenhamos experiência em programar recursivamente. O SALTO DE FÉ não vem fácil…
  - Quando alguma coisa der errado lembre-se de que O ERRO ESTÁ EM SUA IMPLEMENTAÇÃO, não está na recursividade por si mesmo. Se estiver com erro:
    - Olhe APENAS 1 ÚNICO NÍVEL da hierarquia de recursividade, não adianta tentar encontrar erros nas chamadas mais profundas… o erro está no primeiro nível.

Pensar recursivamente: lista para evitar erros comuns

- Sua implementação recursiva começou verificando os casos simples?

- Você resolveu todos os casos simples corretamente?

- A decomposição recursiva identificada está fazendo com que o problema seja menor e da mesma forma que o original?
    - Que métrica você está usando para “diminuir” o problema?

- A decomposição eventualmente alcança os casos simples? Tem certeza de que não deixou nenhum caso simples de fora?

Pensar recursivamente: lista para evitar erros comuns

- As chamadas recursivas são compostas por problemas que verdadeiramente têm uma forma idêntica ao do problema original?
    - Se as chamadas recursivas alterarem a natureza do problema, se violarem um dos pressupostos iniciais ou não tiverem a mesma forma do original, todo o processo está com falha.

- Quando você aplica o SALTO DE FÉ RECURSIVO, as soluções para os subproblemas fornecem uma solução completa e correta para o problema original?