Recursão

Estrutura de Dados
Antonio Oseas

Definição de recursão

- Recursão
  - É uma técnica de programação na qual um método chama a si mesmo.
  - Repetição pode ser obtida de 2 maneiras:
    - Laços (for, while, do while)
    - É também um processo repetitivo caracterizado pelas chamadas a si mesma.

Regras para projetar uma função recursiva

- Determinar o caso base;
- Determinar o caso geral;
- Combinar o caso base e o caso geral na função.

Atenção: Cada chamada da função deve reduzir o tamanho do problema e movê-lo em direção do *caso base*. O *caso base* deve terminar sem chamar a função recursiva; isto é, executar um return.

Recursividade
Exemplo 1

- Função para mostrar os números de n até 1:

```c
void func (int n){
    if (n > 0){
        printf("%d ",n);
        func(n-1);
    }
}
```

- Desenvolva uma função que mostre os números de 1 até n:

Recursividade

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

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

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

Multiplicação Revursiva

- Podemos também implementar a multiplicação de um número m por n somando m com a multiplicação de m por n-1.
  - m * n = m + (m * (n-1))
  - 2*4 = 2 + (2*3)
- Estou chamando de novo a operação de multiplicação mas agora para resolver um sub-problema que é parte do anterior.

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

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

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

Treinando… Exemplo 2

- Implementar uma função recursiva que multiplique M x N usando apenas adição.

Treinando…
Exemplo 2 – possível solução

- Implementar uma função recursiva que multiplique M x N usando apenas adição.

```
int func1(int n, int m){
    if (n == 0)
        return 0;
    return m + func1(n-1, m);
}
```

Fatorial não recursivo

- Definição **não** recursiva (tradicional):

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

- Definição recursiva:

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

Fatorial recursivo

- Definição não recursiva (tradicional):

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

- implementação iterativa:

```
int fatNaoRecursivo(int n){
    int result = 1;
    for (int i = 1; i <= n; i++) {
        result*=i;
    }
    return result;
}
```

Fatorial recursivo
Exemplo 3

Definição recursivo:

$$
\begin{cases} N! = 1, \text{ para } N <= 1; \\ N! = N \times (N - 1)!, \text{ para } N > 1. \end{cases}
$$

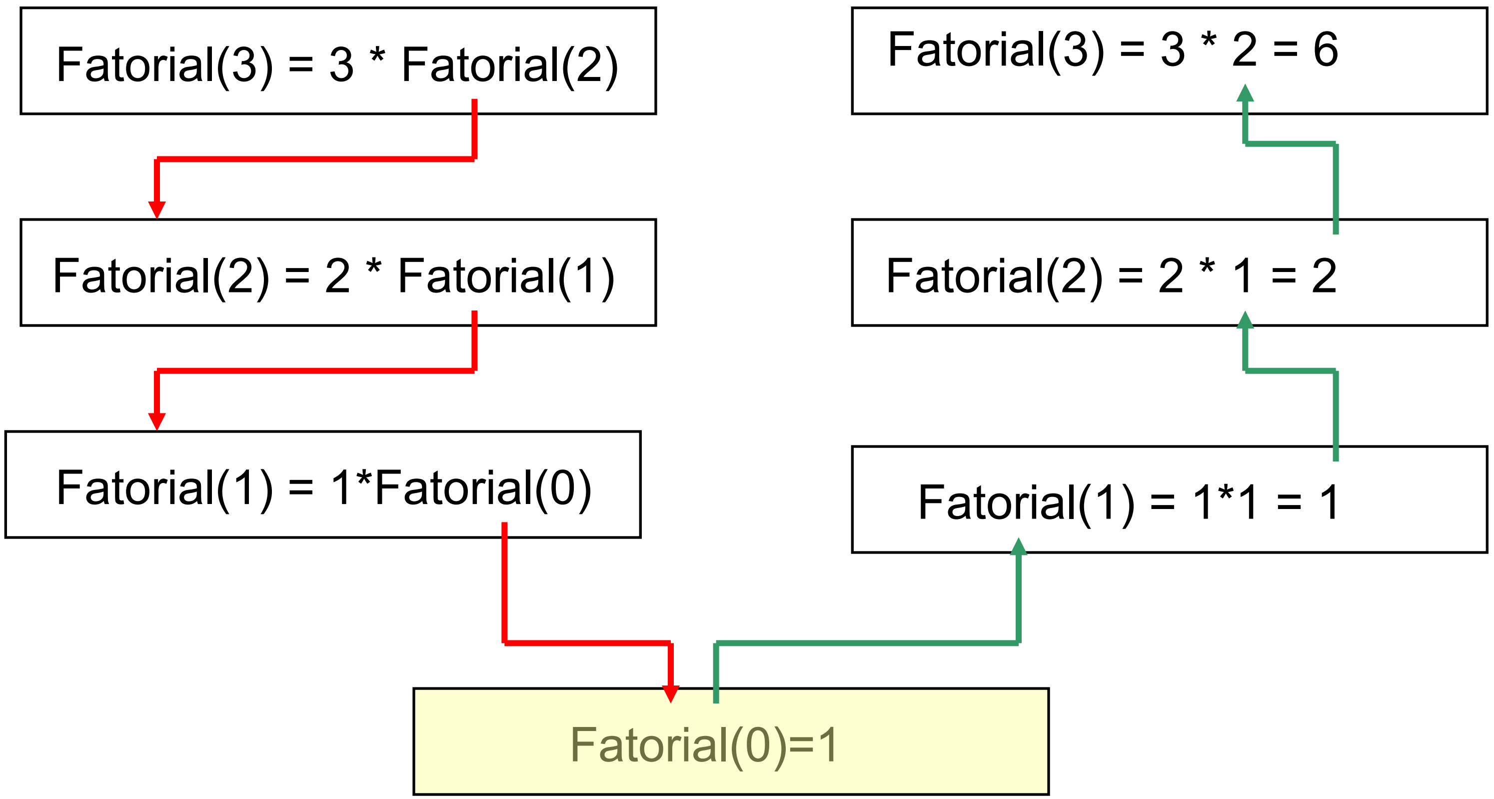
Caso base

Caso recursivo

Praticando...

Implementem a função recursiva para cálculo do fatorial.

Na prática...
Decomposição do Fatorial(3)



Em outras palavras...

Note que a solução recursiva para um problema envolve um caminho de dois sentidos: primeiro o problema é decomposto no sentido top/down e depois resolvido no sentido bottom/up.

Recapitulando...

Toda função recursiva possui dois elementos:

- Resolver parte do problema (caso base) ou
- Reduzir o tamanho do problema (caso geral).

No caso do nosso exemplo, Fatorial(1) é o caso base

Recursão Linear

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

Recursão Linear (Exemplo)

- Exemplo: Soma de *n* inteiros em um array A

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

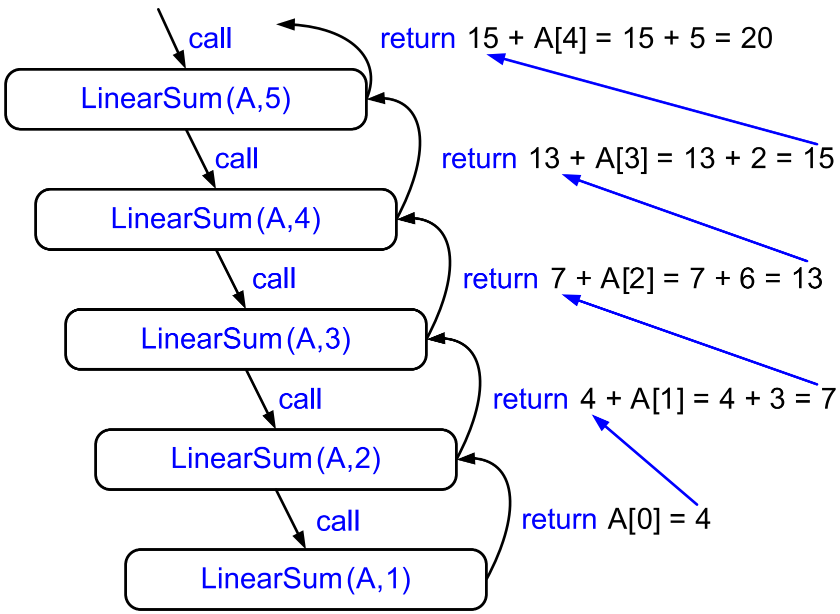
Algorithm linearSum(*A, n*):
Input:
um array A e um inteiro n que contém o tamanho de A
Output:
A soma dos primeiros n elementos de A
if *n* = 1 **then**
  return *A*[0]
else
  return linearSum(*A, n* - 1) + *A*[*n* -1]

Exercício

- Implementar a soma recursiva de um vetor de inteiros.

Recursão Linear (Exemplo)

Trace:



Recursão Binária

- Recursão binária ocorre sempre que houver 2 chamadas recursivas para cada caso não básico.
- Estas chamadas podem, por exemplo, ser usadas para resolver duas metades do mesmo problema.

Números de Fibonacci

Números de Fibonacci são uma série na qual cada número é a soma dos dois números anteriores:

0, 1, 1, 2, 3, 5, ...

Para iniciar a série é preciso conhecer os dois primeiros números.

Generalização da série de Fibonacci

Dados (caso base):

$Fibonacci_0 = 0;$

$Fibonacci_1 = 1;$

Então (caso geral):

$Fibonacci_n = Fibonacci_{n-1} + Fibonacci_{n-2}$

Exemplo 4

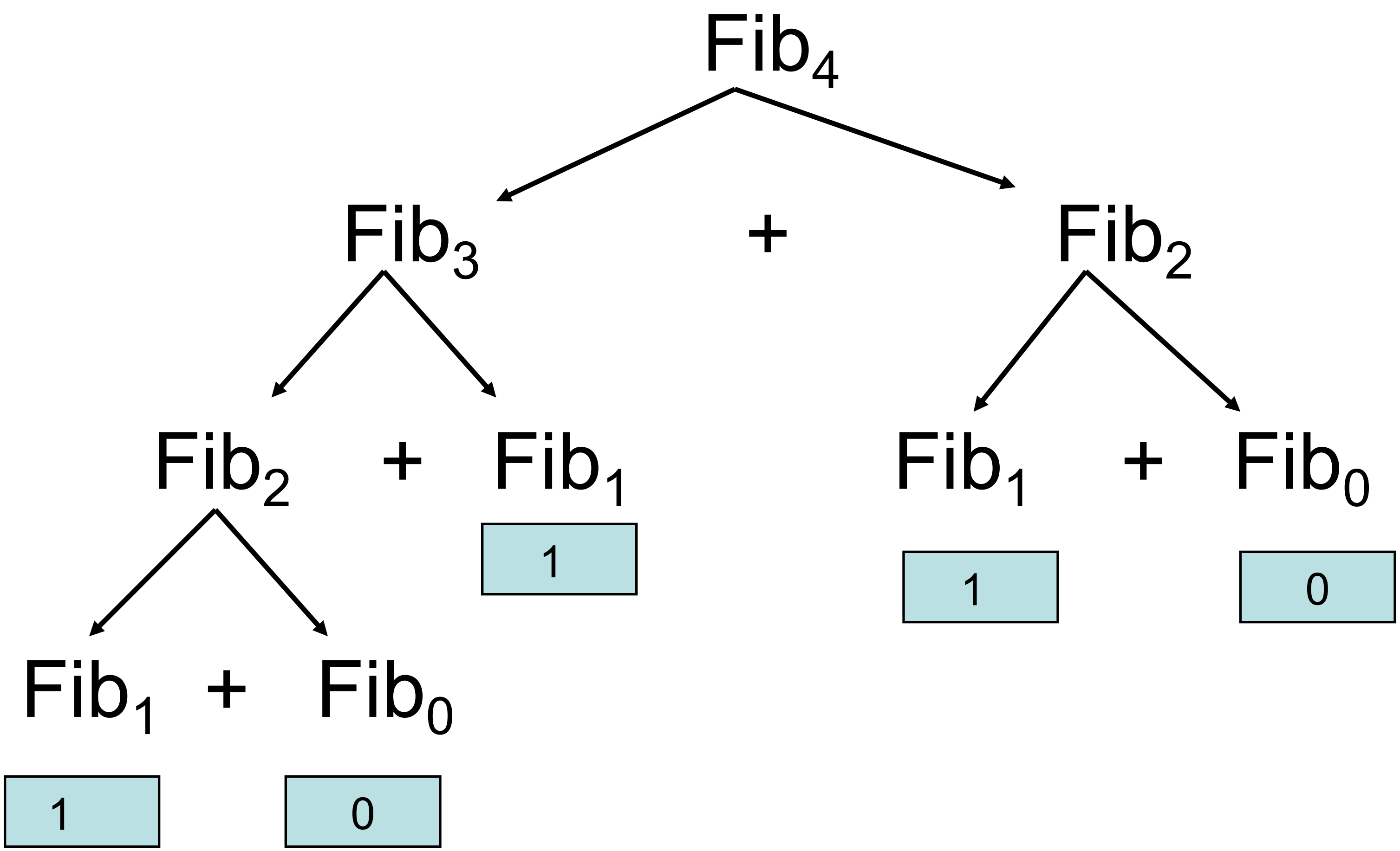
1) Escreva uma função que determine a serie de Fibbonacci com n termos.

Exercício

1) Escreva uma função que determine a serie de Fibbonacci com n termos.

```
int fib(int n)
{
    if (n == 0 || n == 1)  // caso base
        return n;
    return (fib(n-1) + fib(n-2)); // caso geral
}
```

Generalização de Fibonacci$_4$



Chamada de método

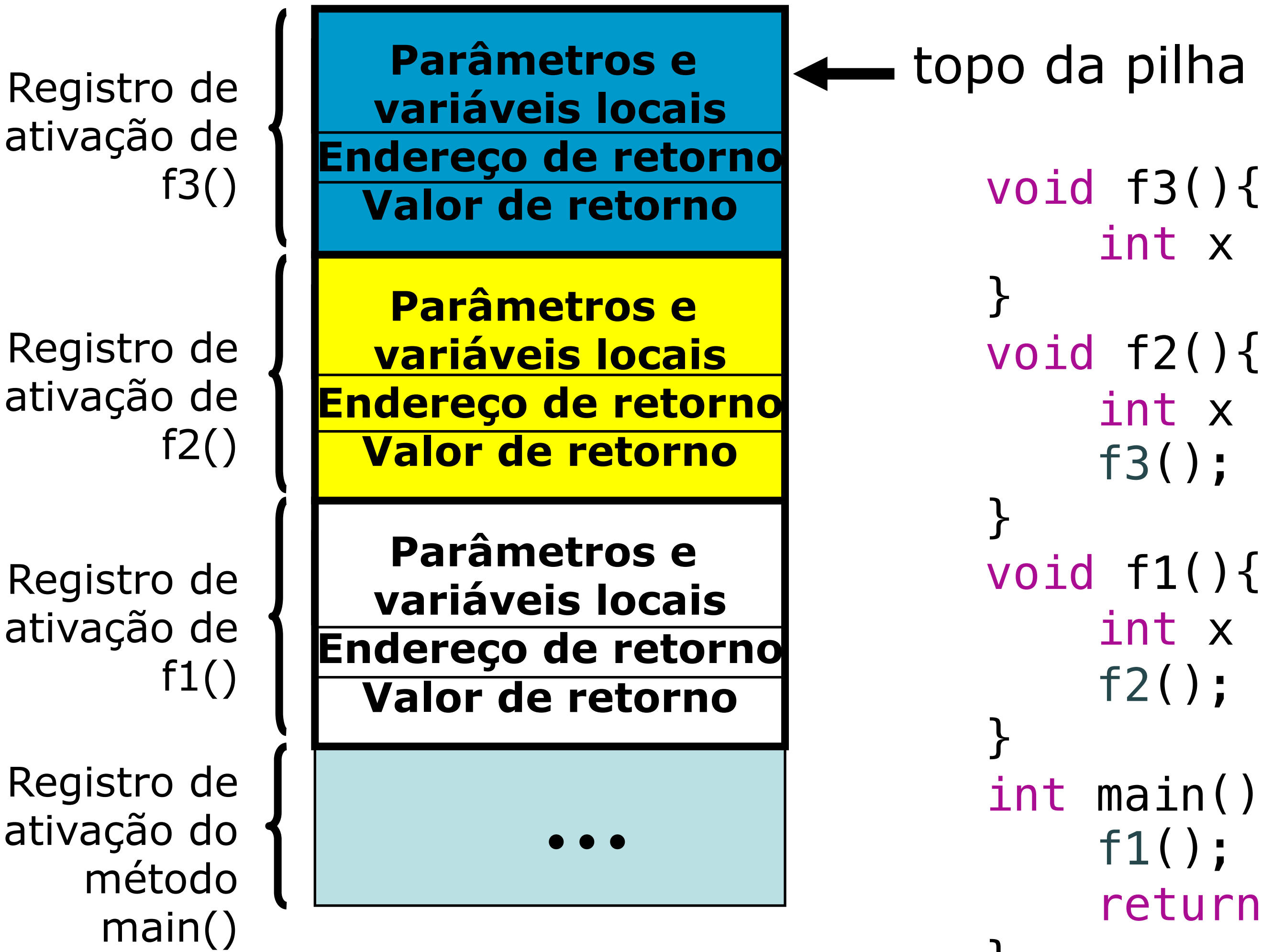
- Quando um método é chamado:
  - é necessário inicializar os parâmetros formais (parâmetros da função) com os valores passados como argumento;
  - sistema precisa saber onde reiniciar a execução do programa;
- Informações de cada função (variáveis e endereço de retorno) devem ser guardadas até o método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

```
int fatorial(int n){
    if (n <=1) {
        return 1;
    }
    return n*fatorial(n-1);
}
```

Registro de ativação

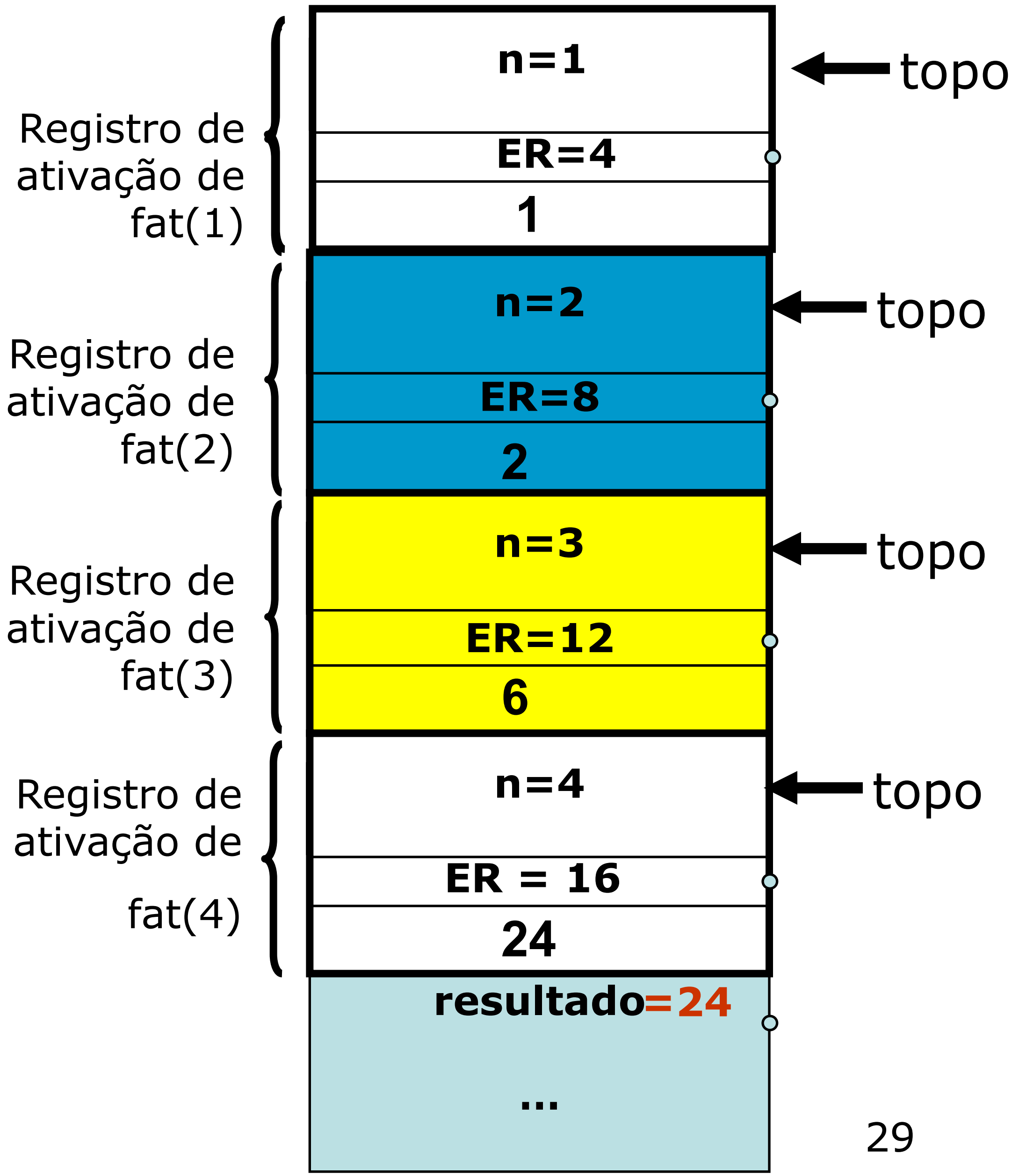
- Registro de ativação:
  - área de memória que guarda o estado de uma função, ou seja:
    - variáveis locais
    - valores dos parâmetros;
    - endereço de retorno (instrução após a chamada do método corrente);
    - valor de retorno.
- Registro de ativação são criados em uma pilha em tempo de execução;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

Registro de ativação




```
void f3(){
    int x = 3;
}
void f2(){
    int x = 2;
    f3();
}
void f1(){
    int x = 1;
    f2();
}
int main(){
    f1();
    return 0;
}
```

Registro de ativação: exemplo



fatorial de 4

fat (4) → main

4*fat(3) =24

3*fat(2) =6

2*fat(1) =2

1

```
int fat(int n){

    if (n < 1)
        return 1;
    return n * fat(n-1);
}

int main(){
    printf("%d",fat(4));

    return 0;
}
```

- A cada término de FAT, o controle retorna para a expressão onde foi feita a chamada na execução anterior, e o último conjunto de variáveis que foi alocado é liberado nesse momento. Esse mecanismo utiliza uma pilha.
- A cada nova chamada do método FAT, um novo conjunto de variáveis (n, FAT) é alocado.

Limitações da recursão

- Soluções recursivas podem envolver alto overhead devido à chamada de funções;
- A cada chamada da função recursiva, espaço de memória (na pilha) é alocada. Se o número de chamada recursiva é muito grande, pode não ter espaço suficiente na pilha para executar o programa (segment fault)

Limitações da recursão

- As funções para determinar o fatorial e a serie de Fibonacci são melhores implementadas por funções iterativas;

Isto significa que soluçoes iterativas são sempre melhores que as recursivas????

NÃO: ALGUNS ALGORITMOS SÃO MAIS FACEIS DE IMPLEMENTAR USANDO RECURSÃO E SÃO MAIS EFICIENTES

Exercícios:

1. Crie uma função recursiva que calcule a exponenciação de um valor b por um expoente p sem usar o operador de exponenciação.
2. Escreva uma função recursiva que escreva na tela todos os números inteiros positivos desde um valor K informado pelo usuário até 0.
3. Escreva um algoritmo recursivo que escreva na tela a soma de todos os números inteiros positivos de K até 0.
4. Escreva uma função recursiva que calcule a soma de todos os números compreendidos entre os valores A e B passados por parâmetro.
5. Escreva uma função recursiva que calcule os juros compostos de um valor. Para isso o programa deverá ler um valor inicial, o número de meses e a taxa de juros ao mês, e passar estes valores à função como parâmetros.
6. Escreva uma função que faça a procura sequencial de um valor passado por parâmetro num vetor também passado por parâmetro.