Algoritmos e Estruturas de Dados I – Linguagem C

Aula – Tópico 11
Recursividade

Recursão

- Na linguagem C, uma função pode chamar outra função.
  - A função main() pode chamar qualquer função, seja ela da biblioteca da linguagem (como a função printf()) ou definida pelo programador (função imprime()).
- Uma função também pode chamar a si própria
  - A qual chamamos de ***função recursiva***.

Problema 26

- Implemente uma função que classifique os elementos de um vetor em ordem crescente usando o algoritmo quicksort:

1. Seja m o elemento na posição central no vetor;
2. Seja i o índice do primeiro e j, o índice do último elemento do vetor;
3. Enquanto i for menor ou igual a j, faça com que:
   a) O valor de i cresça até encontrar um elemento maior que m;
   b) O valor de j diminua até encontrar um elemento menor que m;
   c) Haja a troca entre os elementos que ocupam as posições i e j.

Problema 26

- Ao final desses passos, a situação do vetor será a seguinte:

  à esquerda da posição central, existem somente elementos menores do que m;

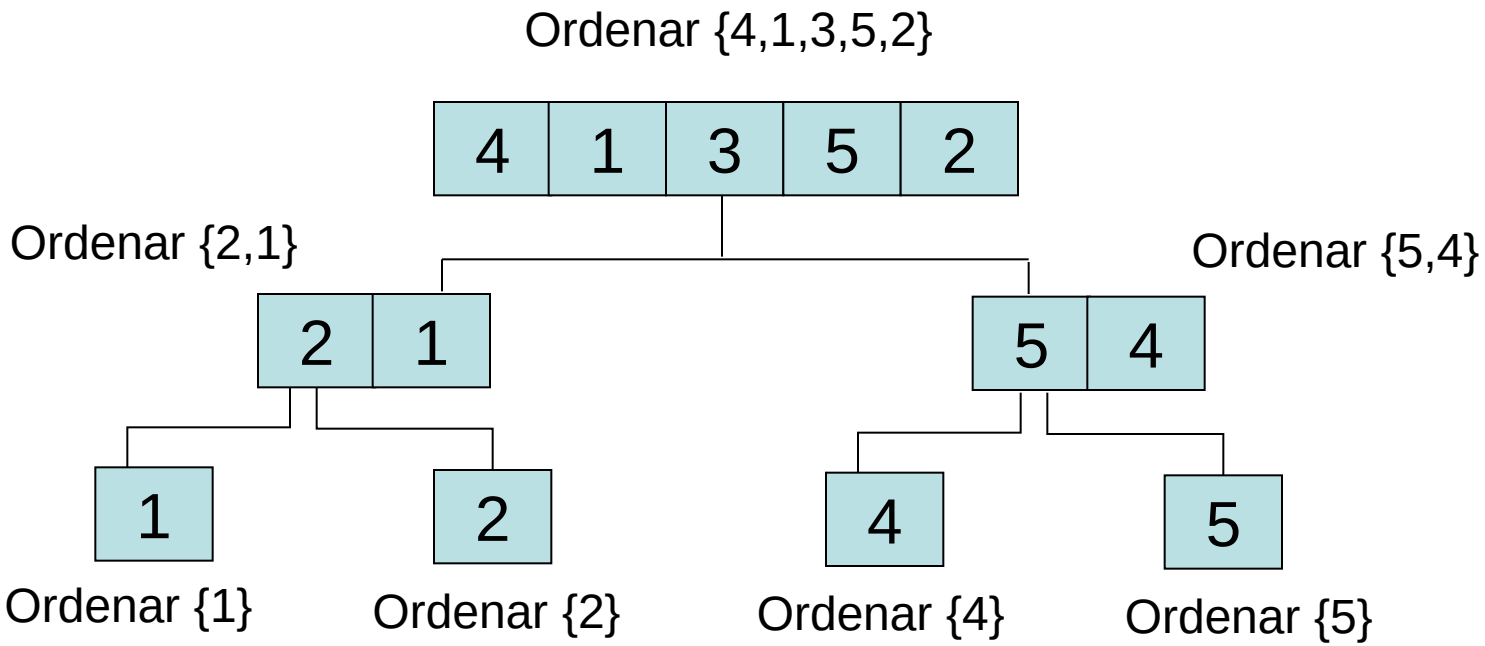
  à direita da posição central, existem somente elementos maiores do que m;

- Assim, o problema de ordenar o vetor se reduz ao problema de ordenar cada uma dessas “metades”.

- Os mesmos passos serão aplicadas repetidas vezes à cada nova “metade”, até que cada metade contenha um único elemento (caso trivial).

Problema 26

- Considere o exemplo abaixo:



- Como a natureza dos problemas é sempre a mesma (“ordenar um vetor”), o mesmo método pode ser usado para cada subproblema.

Análise do programa

```
void quicksort(int v[], int primeiro, int ultimo)
{
  int i,j;
  int m,aux;

  i = primeiro;
  j = ultimo;
  m = v[(i+j)/2];
  do
  {
    while (v[i] < m) i++;
    while (v[j] > m) j--;
    if (i <= j)
    {
      aux = v[i];
      v[i] = v[j];
      v[j] = aux;
      i++;
      j--;
    }
  }
  while (i <= j);
  if (primeiro < j)
    quicksort(v,primeiro,j);
  if (ultimo > i)
    quicksort(v,i,ultimo);
}
```

Veja que interessante: a função quicksort chama a si mesma!!!

Recursividade

- A função quicksort implementada é um exemplo de função recursiva: função que chama a si mesma.
- A recursividade é uma forma interessante de resolver problemas por meio da divisão dos problemas em problemas menores de mesma natureza.
- Se a natureza dos subproblemas é a mesma do problema, o mesmo método usado para reduzir o problema pode ser usado para reduzir os subproblemas e assim por diante.
- Quando devemos parar? Quando alcançarmos um caso trivial que conhecemos a solução.

Recursividade

- Assim, um processo recursivo para a solução de um problema consiste em duas partes:

1. O caso trivial, cuja solução é conhecida;
2. Um método geral que reduz o problema a um ou mais problemas menores (subproblemas) de mesma natureza.

- Muitas funções podem ser definidas recursivamente. Para isso, é preciso identificar as duas partes acima.

- Exemplo: fatorial de um número e o n-ésimo termo da seqüência de Fibonacci.

Recursividade

- A recursão também é chamada de definição circular. Ela ocorre quando algo é definido em termos de si mesmo.
- Um exemplo clássico de função que usa recursão é o cálculo do fatorial de um número:
  - 3! = 3 * 2!
  - 4! = 4 * 3!
  - n! = n * (n - 1)!

Recursividade

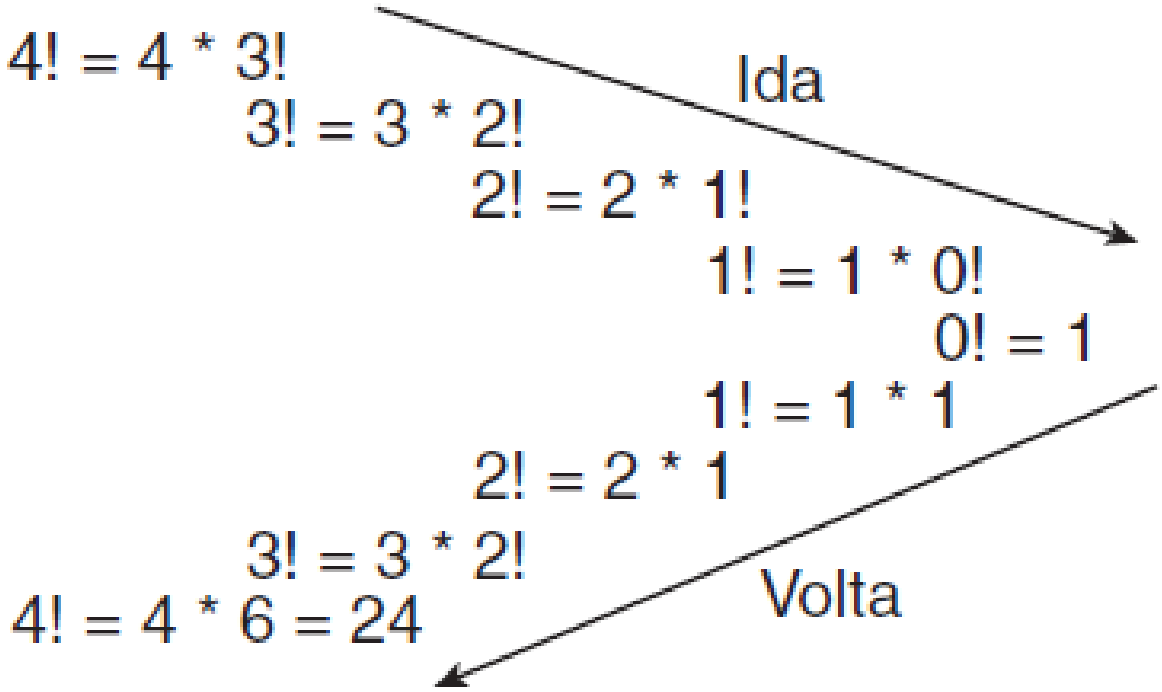
0! = 1

1! = 1 * 0!

2! = 2 * 1!

3! = 3 * 2!

4! = 4 * 3!



n! = n * (n - 1)! : fórmula geral

0! = 1 : caso-base

Recursividade

Com Recursão

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

Sem Recursão

```
int fatorial (int n){
if (n == 0)
    return 1;
else {
    int i;
    int f=1;
    for (i=2; i <= n;i++)
    f = f * i;
    return f;
    }
}
```

Recursividade

- Em geral, formulações recursivas de algoritmos são freqüentemente consideradas "mais enxutas" ou "mais elegantes" do que formulações iterativas.
- Porém, algoritmos recursivos tendem a necessitar de mais espaço do que algoritmos iterativos.

Recursividade

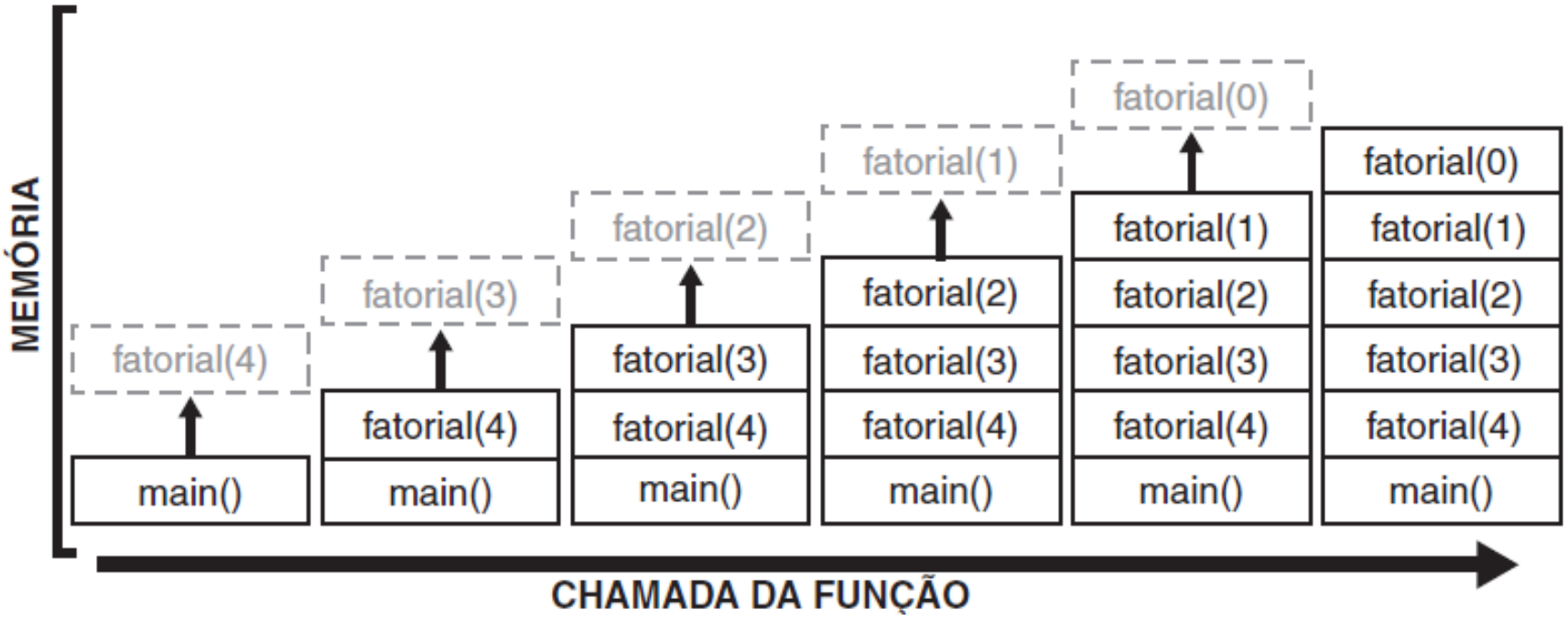
- Todo cuidado é pouco ao se fazer funções recursivas.
  - Critério de parada: determina quando a função deverá parar de chamar a si mesma.
  - O parâmetro da chamada recursiva deve ser sempre modificado, de forma que a recursão chegue a um término.

Recursividade

```
int fatorial (int n){
if (n == 0)//critério de parada
   return 1;
else
   return n*fatorial(n-1); /*parâmetro de
   fatorial sempre muda*/
}
```

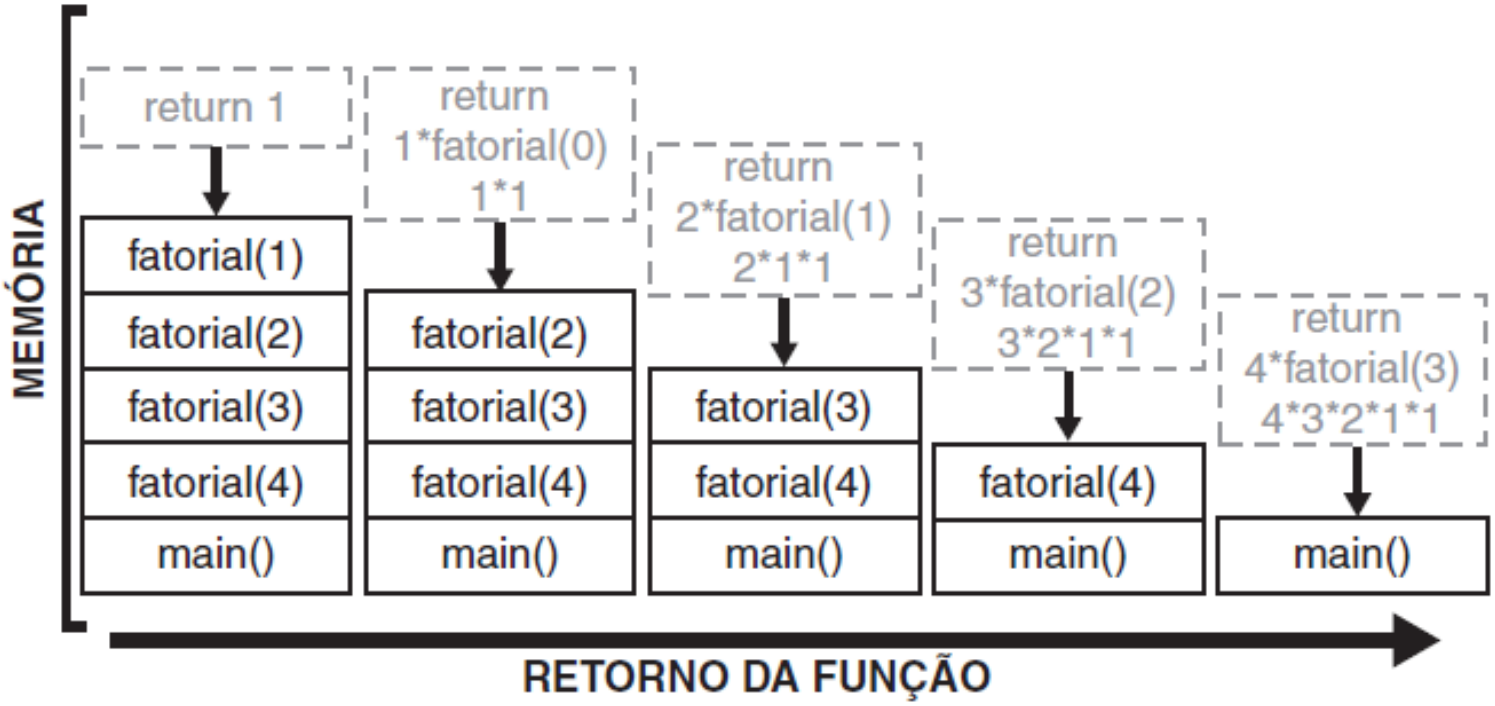
Recursividade

- O que acontece na chamada da função fatorial com um valor como n = 4?
  - int x = fatorial (4);



Recursividade

- Uma vez que chegamos ao caso-base, é hora de fazer o caminho de volta da recursão.



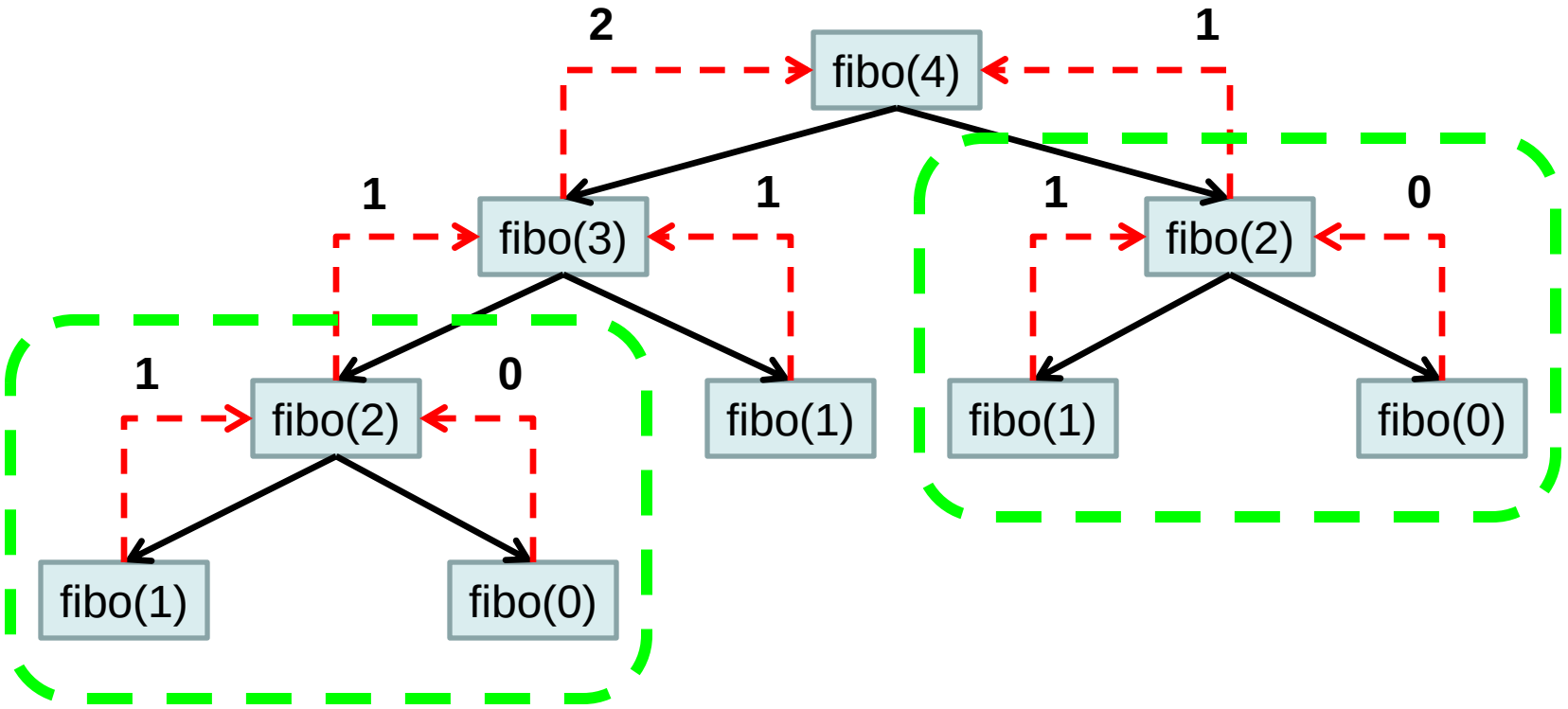
Fibonacci

- Essa seqüência é um clássico da recursão
  - 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...
- Sua solução recursiva é muito elegante ...

```
int fibo(int n){
if (n == 0 || n == 1)
   return n;
else
   return fibo(n-1) + fibo(n-2);
}
```

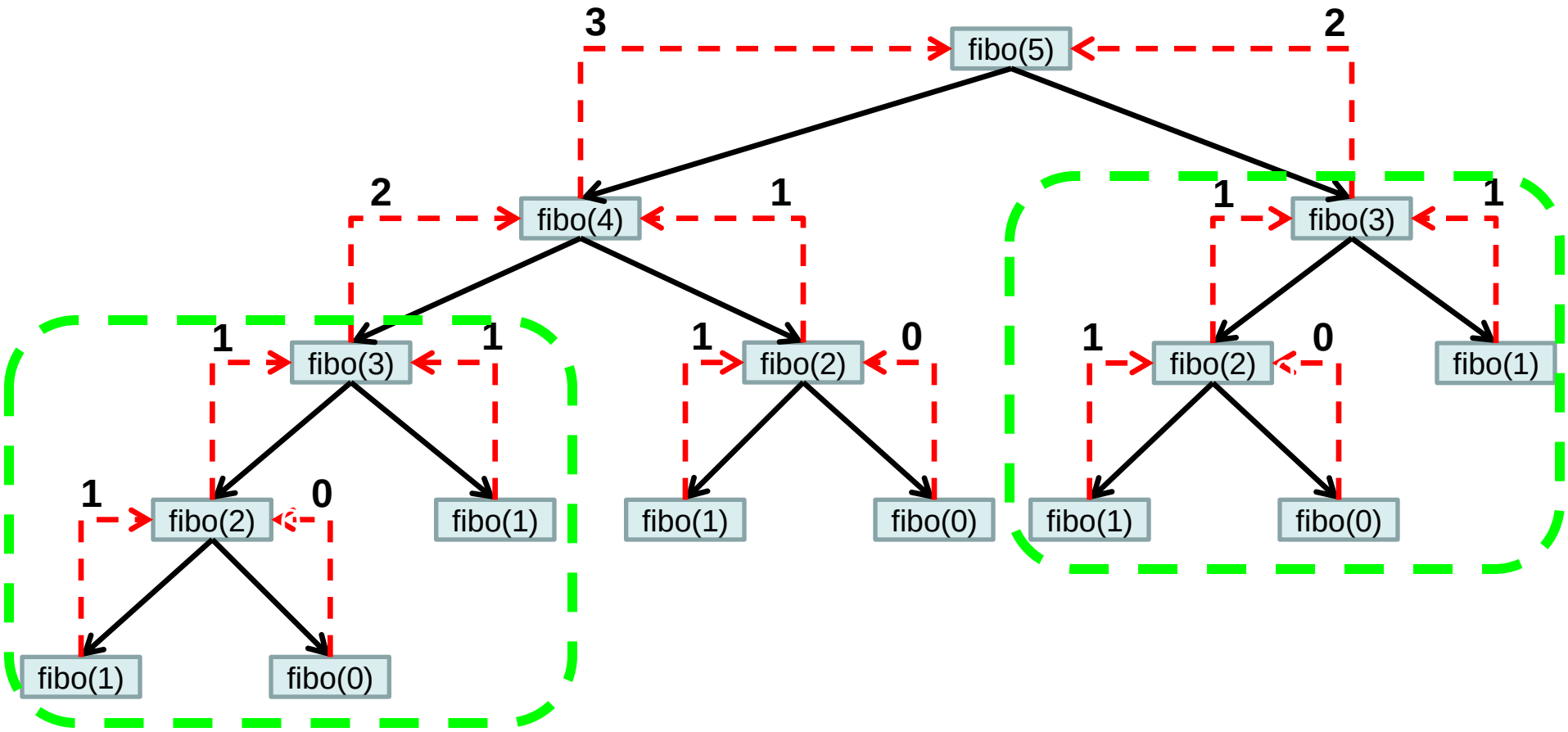
Fibonacci

- ... mas como se verifica na imagem, elegância não significa eficiência



Fibonacci

- Aumentando para **fibo(5)**



Função fatorial

- A função fatorial de um inteiro não negativo pode ser definida como:

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

- Esta definição estabelece um processo recursivo para calcular o fatorial de um inteiro n.

- Caso trivial: n=0. Neste caso: n!=1.

- Método geral: n x (n-1)!.

Função fatorial

- Assim, usando-se este processo recursivo, o cálculo de 4!, por exemplo, é feito como a seguir:

```
4! = 4 * 3!
   = 4 * (3 * 2!)
   = 4 * (3 * (2 * 1!))
   = 4 * (3 * (2 * (1 * 0!)))
   = 4 * (3 * (2 * (1 * 1)))
   = 4 * (3 * (2 * 1))
   = 4 * (3 * 2)
   = 4 * 6
   = 24
```

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

Mas como uma função recursiva como esta é de fato implementada no computador?

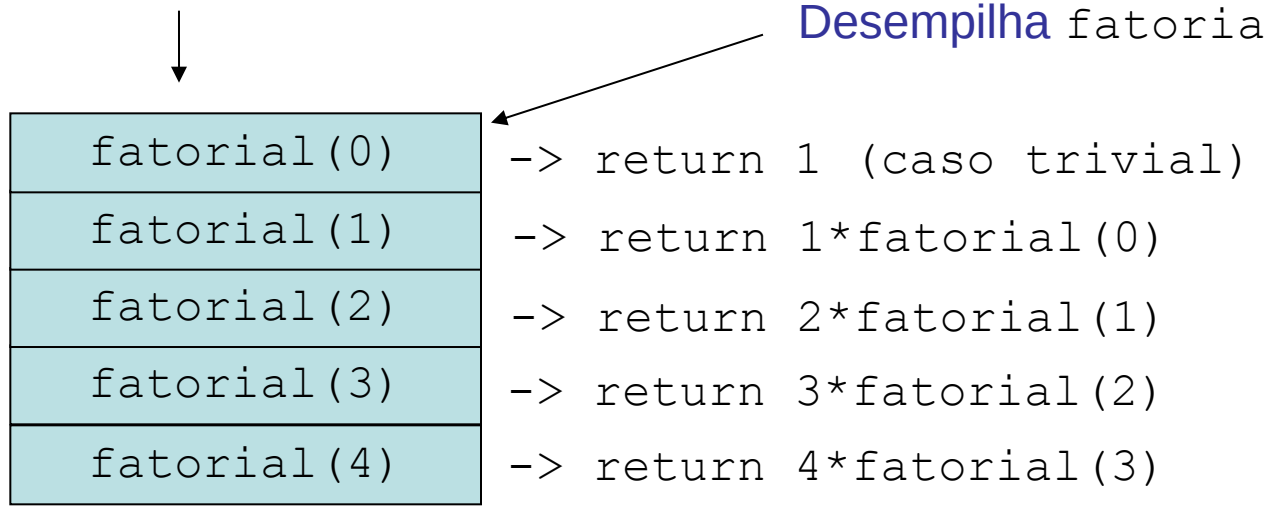
Usando-se o mecanismo conhecido como Pilha de Execução!

Função fatorial

- Considere, novamente, o exemplo para 4!:

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

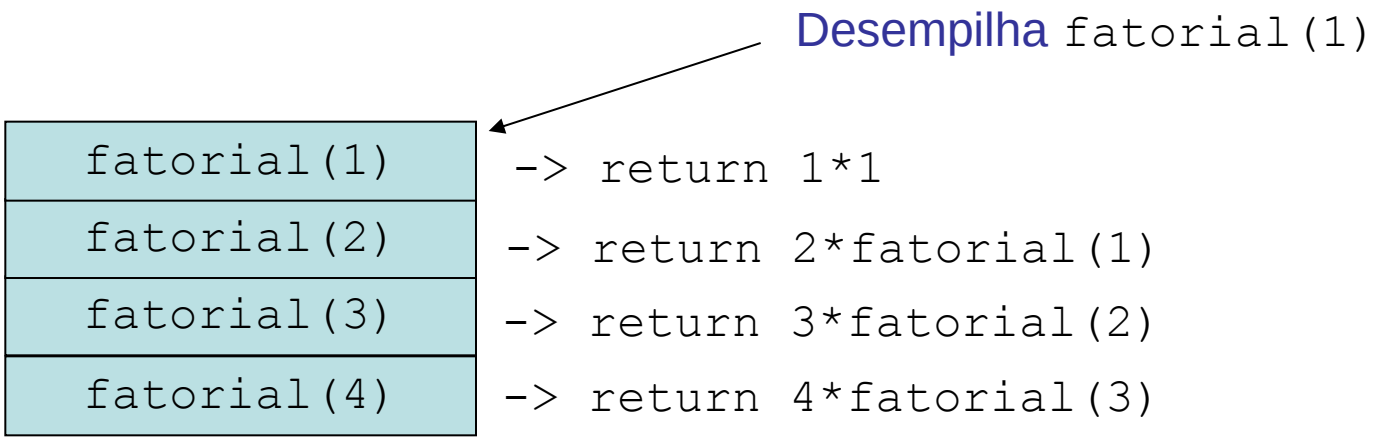
Pilha de Execução



Função fatorial

- Considere, novamente, o exemplo para 4!:

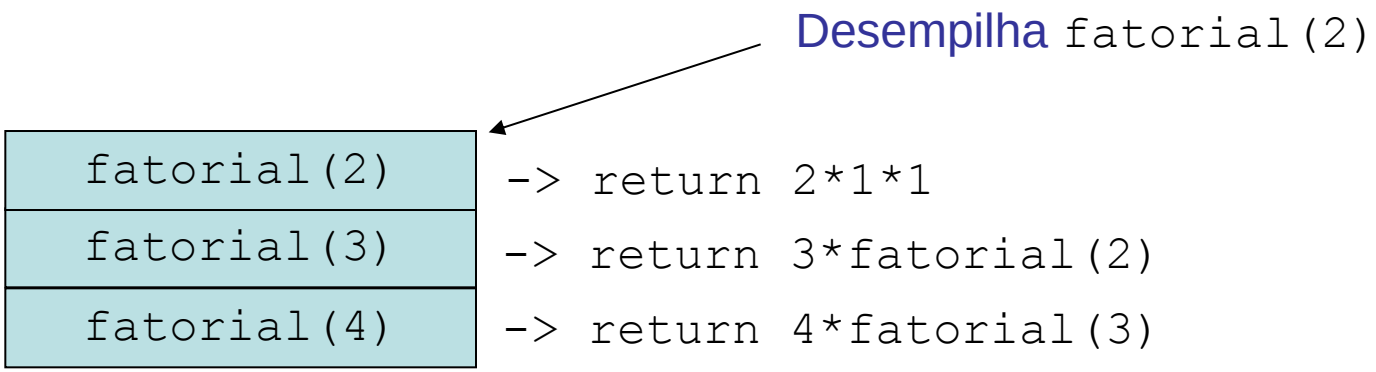
```
int fatorial(int n)
{
  if (n == 0)
    return 1;
  else
    return n * fatorial(n-1);
}
```



Função fatorial

- Considere, novamente, o exemplo para 4!:

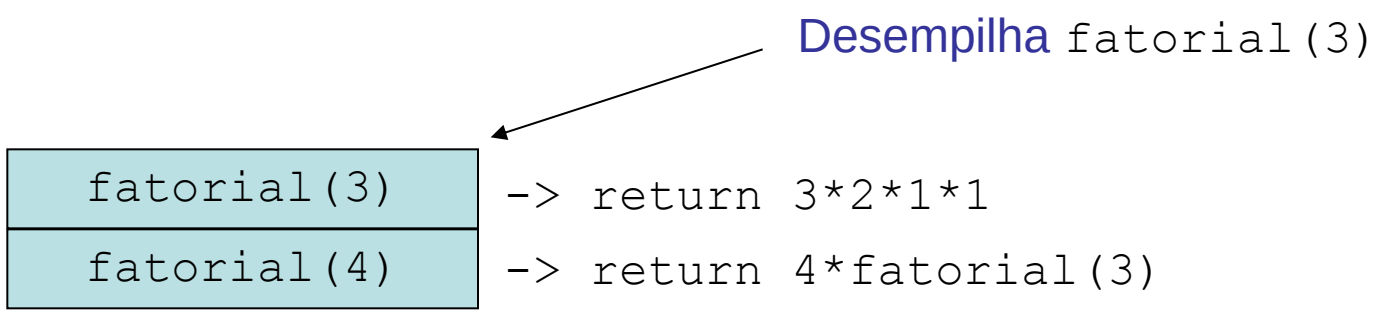
```
int fatorial(int n)
{
  if (n == 0)
    return 1;
  else
    return n * fatorial(n-1);
}
```



Função fatorial

- Considere, novamente, o exemplo para 4!:

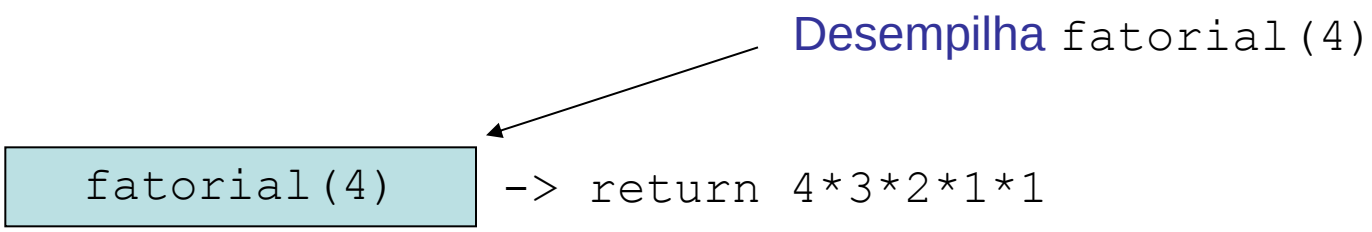
```
int fatorial(int n)
{
  if (n == 0)
    return 1;
  else
    return n * fatorial(n-1);
}
```



Função fatorial

- Considere, novamente, o exemplo para 4!:

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



Função fatorial

- Considere, novamente, o exemplo para 4!:

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

Resultado = 24

Função Fibonacci

- A função Fibonacci retorna o n-ésimo número da seqüência: 1, 1, 2, 3, 5, 8, 13, ...

- Os dois primeiros termos são iguais a 1 e cada um dos demais números é a soma dos dois números imediatamente anteriores.

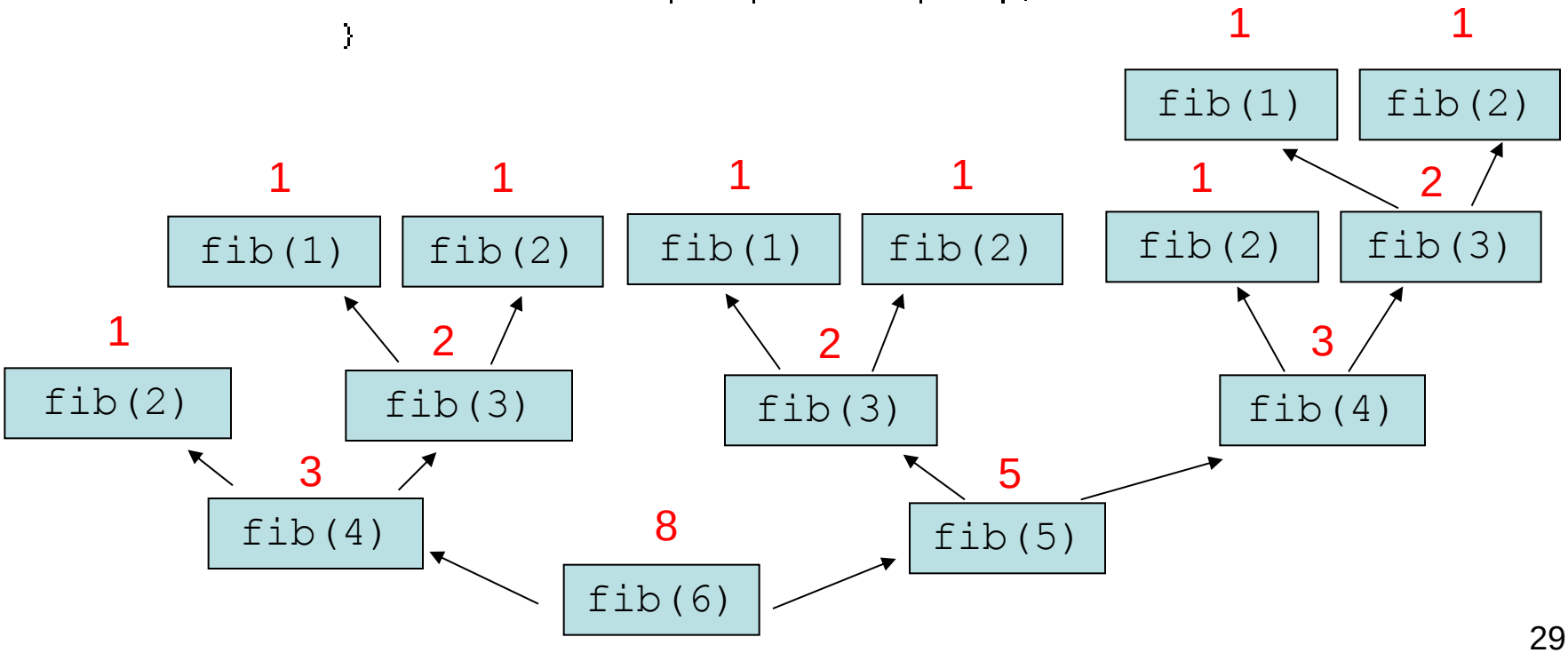
- Sendo assim, o n-ésimo número *fib(n)* é dado por:

$$fib(n) = \begin{cases} 1 & n = 1 \\ 1 & n = 2 \\ fib(n-2) + fib(n-1) & n > 2 \end{cases}$$

Função Fibonacci

- Veja uma implementação recursiva para esta função:

```
int fib(int n)
{
  if ((n == 1) || (n == 2))
    return 1;
  else
    return fib(n-2) + fib(n-1);
}
```



Função Fibonacci

- A função recursiva para cálculo do n-ésimo termo da seqüência é extremamente ineficiente, uma vez que recalcula o mesmo valor várias vezes

Observe agora uma versão iterativa da função fib:

```
int fib(int n)
{
  int i,a,b,c;
  if ((n == 1) || (n == 2))
    return 1;
  else
  {
    a = 0;
    b = 1;
    for (i = 3; i <= n; i++)
    {
      c = a + b;
      a = b;
      b = c;
    }
    return c;
  }
}
```

Função Fibonacci

- O livro dos autores Brassard e Bradley (*Fundamentals of Algorithmics*, 1996, pág. 73) apresenta um quadro comparativo de tempos de execução das versões iterativa e recursiva:

n	10	20	30	50	100
recursivo	8 ms	1 s	2 min	21 dias	10^9 anos
iterativo	0.17 ms	0.33 ms	0.50 ms	0.75 ms	1,50 ms

- Portanto: um algoritmo recursivo nem sempre é o melhor caminho para se resolver um problema.
- No entanto, a recursividade muitas vezes torna o algoritmo mais simples.