Algoritmos e estruturas de Dados

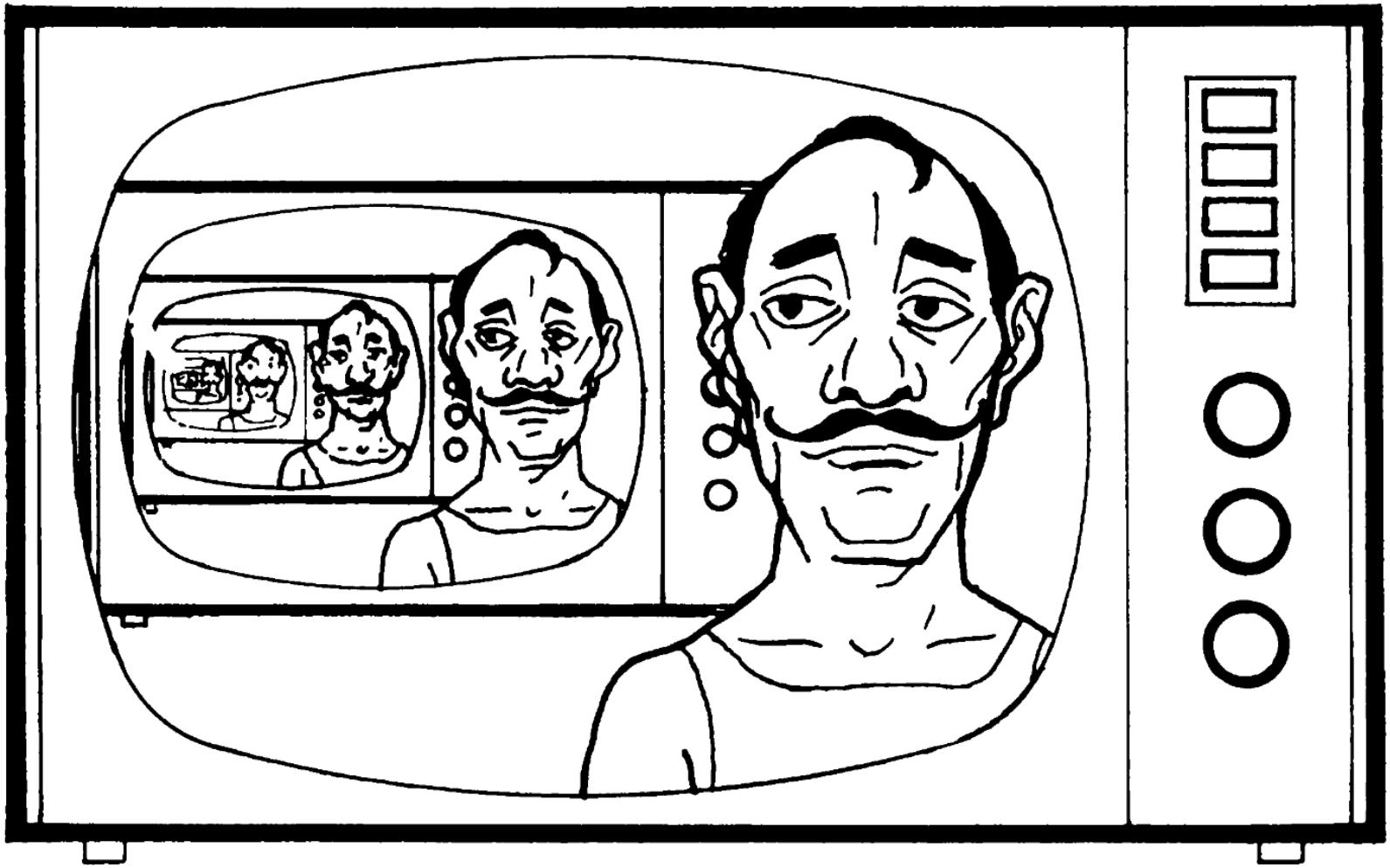
Recursividade

Prof. Maiquel de Brito

Recursividade

- *Para entender recursividade é preciso primeiro entender a recursividade*

Recursividade

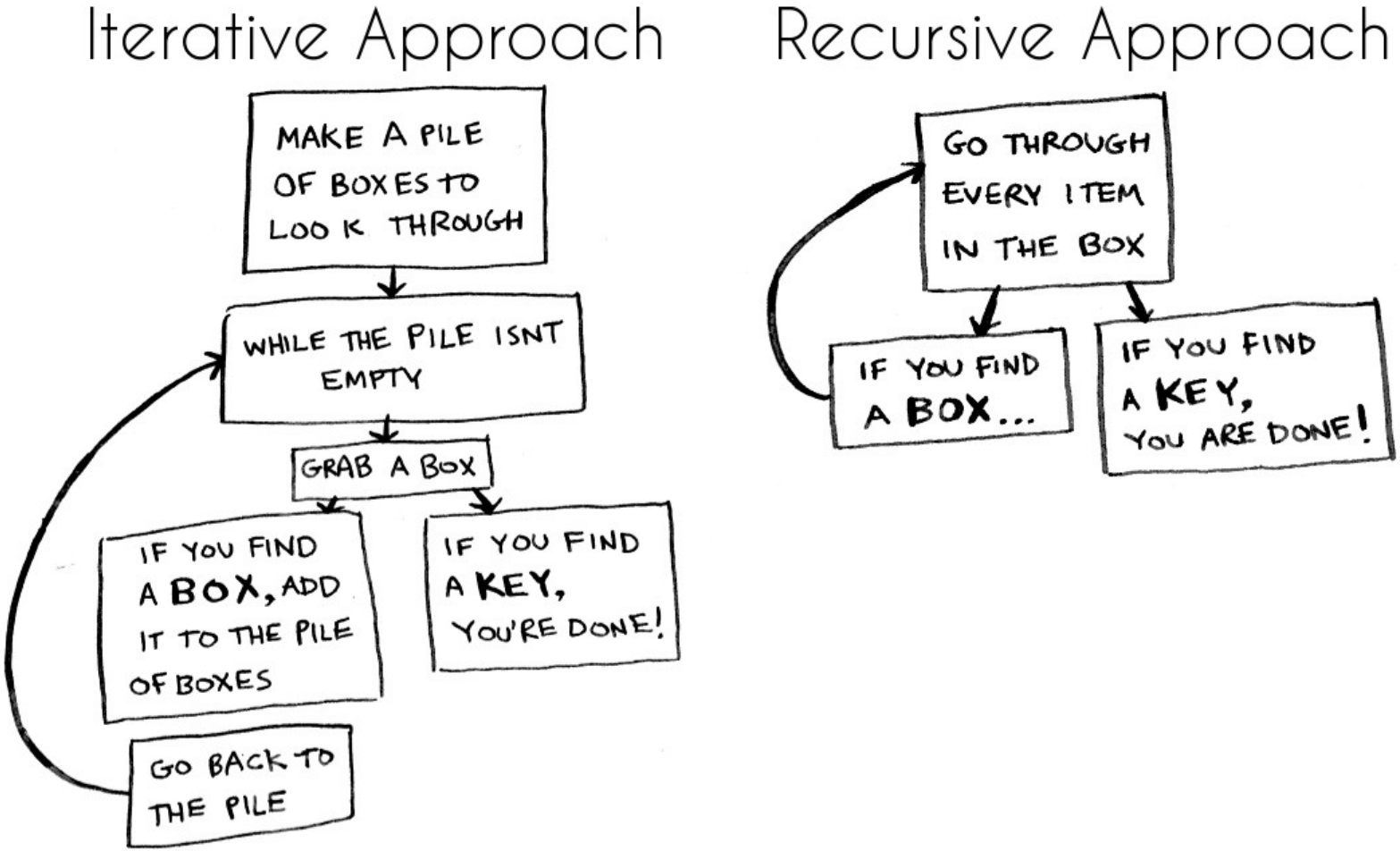


Fonte: WIRTH, Niklaus. Algorithms + Data Structures = Programs

Recursividade

Fonte: www.itcuties.com/java/recursion-and-iteration/

Recursividade



Recursividade

- **Definição**
  - É um princípio poderoso que permite que um problema seja definido em termos de instâncias menores e menores do próprio problema.

- **Computação**
  - Na computação resolvemos problemas recursivos usando funções recursivas que são funções que invocam (chamam) a si próprias.

Recursividade

- Exemplo: cálculo do fatorial.
  - Fatorial de n (n!): produto de todos os números de n até 1
  - Ex. 4! = 4 * 3 * 2 * 1.
  - Solução iterativa: n! = (n) * (n - 1) * (n - 2) ... (1)
  - Solução iterativa – implementação em C

```
int fatorial(int n){
  int i, result = 1;
  for (i = n; i > 1; i--){
    result = result * i;
  }
  return result;
}
```

Recursividade

- Exemplo: cálculo do fatorial.

  Outra forma de pensarmos o problema do fatorial é definirmos: **n! = n * (n – 1)!**

  - Desta forma, a solução para (n -1)! é a mesma usada para n!, apenas um pouco menor. Generalizando, se assumirmos que: (n -1)! = (n – 1) * (n – 2)! e assim por diante até n = 1, encontraremos uma solução decompondo o problema em partes menores. Esta é uma solução recursiva para o problema, denotada por:

Solução recursiva em C

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

$$F(n)=\begin{cases}1 & \text{se } n=0, n=1\\ n.F(n-1) & \text{se } n>1\end{cases}$$

Recursividade

Uma função recursiva sempre terá uma *condição de parada*, que é a condição em que não há nova chamada à mesma função.

Se não houver condição de parada, a função fica eternamente fazendo chamadas recursivas

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

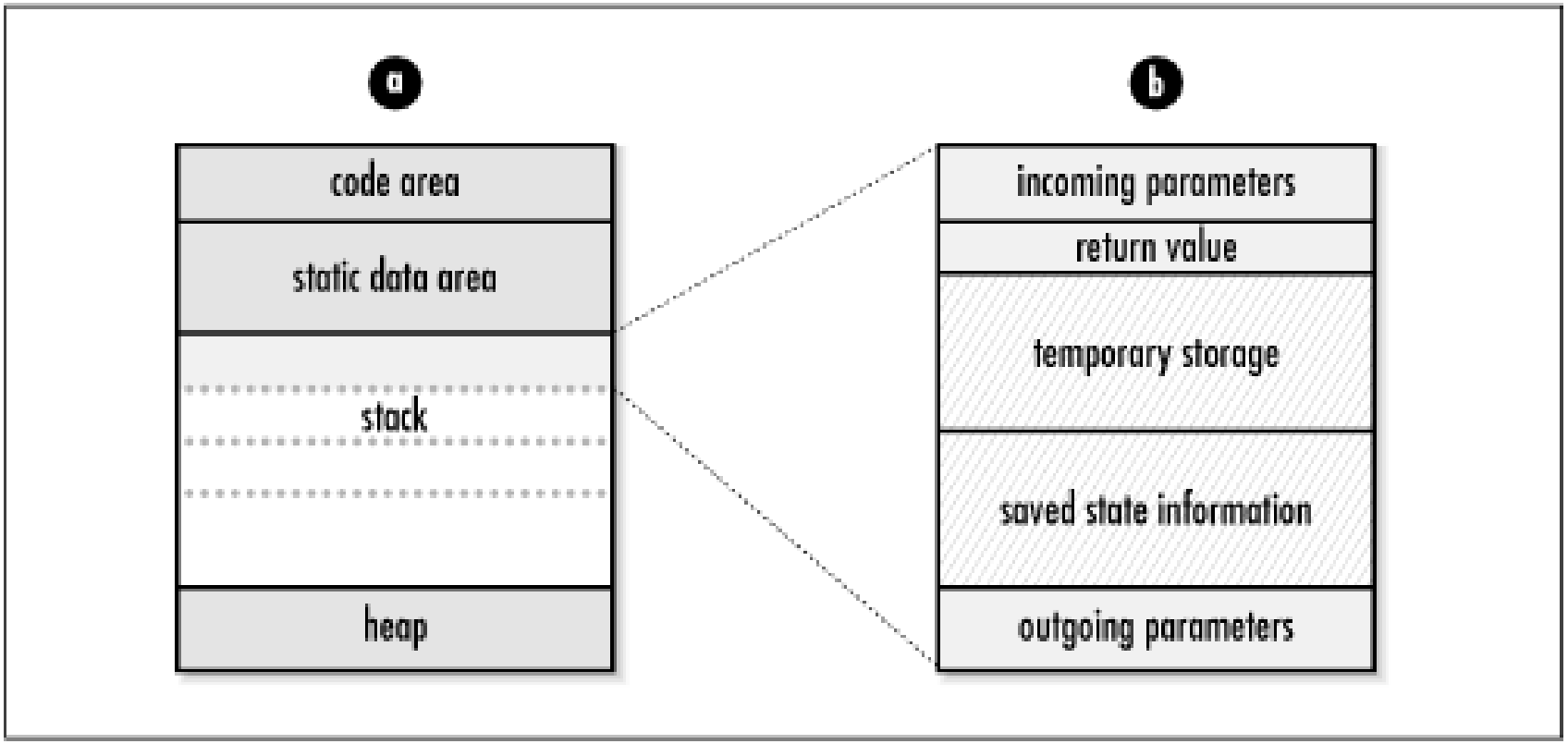
$$F(n) = \begin{cases} 1 & \text{se } n=0, n=1 \\ n \,.\, F(n-1) & \text{se } n>1 \end{cases}$$

Recursividade: fatorial(4) = ?

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

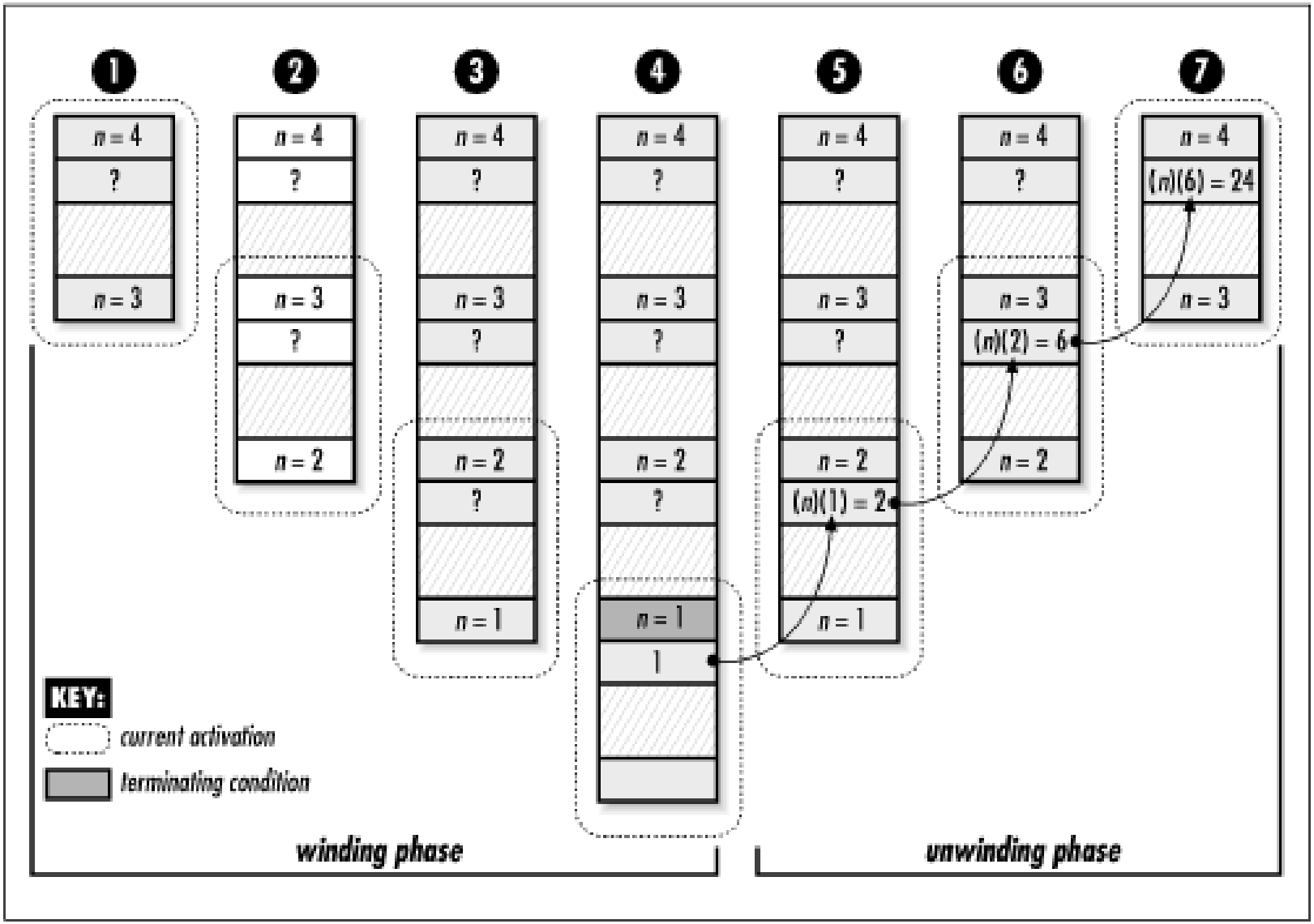
$F(4) = 4 \times F(3)$	winding phase
$F(3) = 3 \times F(2)$	.
$F(2) = 2 \times F(1)$	.
$F(1) = 1$	terminating condition
$F(2) = (2)(1)$	unwinding phase
$F(3) = (3)(2)$	.
$F(4) = (4)(6)$	.
24	recursion complete

Estrutura de uma chamada de função (memória)



code area: instruções a serem executadas (programa)
static data area: dados que persistentes (variáveis globais, *static*, etc)
heap: alocação dinâmica
stack: chamadas de funções

Memória: fatorial(4)



Quando não usar recursividade?

$$f(n)=\begin{cases}1 & \text{se } n\leqslant 1\\ f(n-1)+f(n-2) & \text{se } n>1\end{cases}$$

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

A uma única chamada à função *f(n)* acima pode fazer múltiplas chamadas à própria função para um mesmo *n.*

Estudo avançado (opcional)

Recursividade de cauda (*tail recursion*).

Referências

Figuras retiradas do livro:
Loudon, Kyle. Mastering Algorithms with C. O'Reilly Media, Inc. 2009.