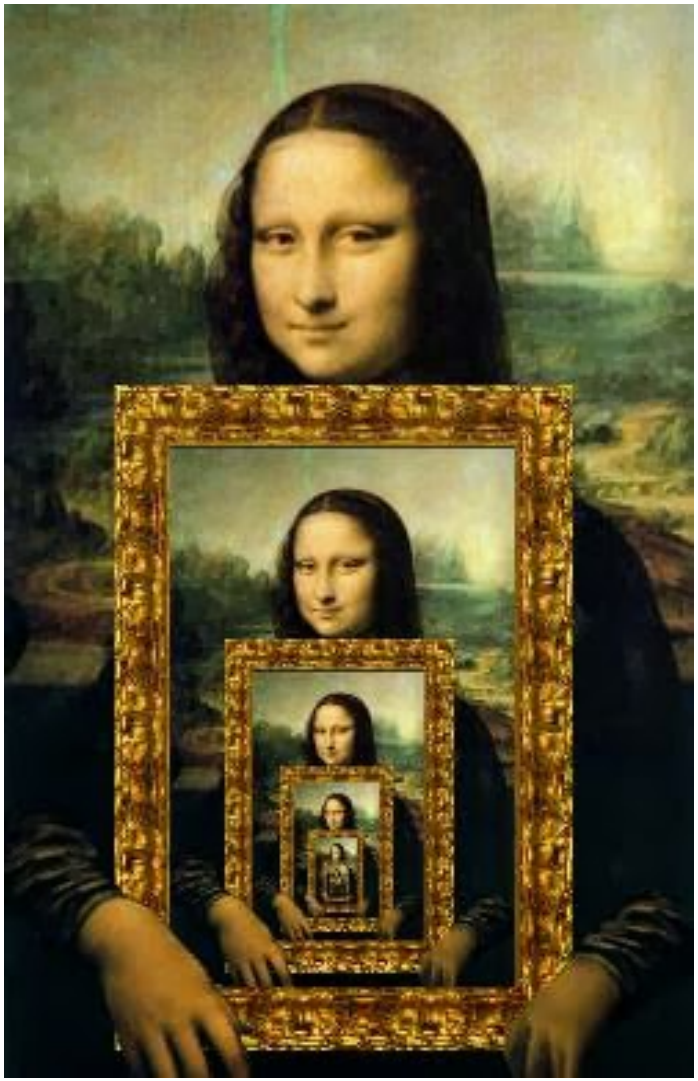
Algoritmos e Programação II

Apontadores - Parte I

Prof. Viviane Bonadia dos Santos

Definições Recursivas



Definições Recursivas

- Recursividade é um mecanismo de programação no qual a definição de uma função é dada em termos de si mesma. **A função que está sendo definida aparece como parte da definição**.

- São sinônimos: **recursividade**, **recursão**, **recorrência**.

Definições Recursivas

- Recursividade é um mecanismo de programação no qual a definição de uma função é dada em termos de si mesma. **A função que está sendo definida aparece como parte da definição**.
- Uma função é dita **recursiva** quando dentro dela há uma chamada para ela mesma.

- São sinônimos: **recursividade**, **recursão**, **recorrência**.

Definições Recursivas

- Recursividade é um mecanismo de programação no qual a definição de uma função é dada em termos de si mesma. **A função que está sendo definida aparece como parte da definição**.
- Uma função é dita **recursiva** quando dentro dela há uma chamada para ela mesma.
- Recursividade é um mecanismo básico para repetições nas linguagens funcionais.

- São sinônimos: **recursividade**, **recursão**, **recorrência**.

Definições Recursivas

- Recursividade é um mecanismo de programação no qual a definição de uma função é dada em termos de si mesma. **A função que está sendo definida aparece como parte da definição**.
- Uma função é dita **recursiva** quando dentro dela há uma chamada para ela mesma.
- Recursividade é um mecanismo básico para repetições nas linguagens funcionais.
- Podemos ter recursão direta e indireta.
- São sinônimos: **recursividade**, **recursão**, **recorrência**.

Dividir para conquistar

- Chave de muitos algoritmos:
  - Se o problema é pequeno:
    1. resolva-o diretamente.
  - Senão:
    1. reduza-o a um subproblema menor do mesmo problema;
    2. aplique o método ao subproblema;
    3. volte ao problema original.

Componentes da recursão

- Toda recursão é composta por:
  - **Um caso base:** uma instância do problema que pode ser solucionada facilmente. Por exemplo, é trivial fazer a soma de uma lista com um único elemento.
  - **Uma ou mais chamadas recursivas:** onde o objeto define-se em termos de si próprio, tentando convergir para o caso base. A soma de uma lista de `n` elementos pode ser definida a partir da soma da lista de `n-1` elementos.

Exemplo de função recursiva

Um exemplo clássico de função recursiva é o cálculo do fatorial de um número:

$4! = 4 * 3 * 2 * 1;$

$3! = 3 * 2 * 1;$

$2! = 2 * 1;$

$1! = 1;$

$0! = 1;$

Exemplo de função recursiva

Um exemplo clássico de função recursiva é o cálculo do fatorial de um número:

$4! = 4 * 3!;$

$3! = 3 * 2 * 1;$

$2! = 2 * 1;$

$1! = 1;$

$0! = 1;$

Exemplo de função recursiva

Um exemplo clássico de função recursiva é o cálculo do fatorial de um número:

$4! = 4 * 3!;$

$3! = 3 * 2!;$

$2! = 2 * 1;$

$1! = 1;$

$0! = 1;$

Exemplo de função recursiva

Um exemplo clássico de função recursiva é o cálculo do fatorial de um número:

$4! = 4 * 3!;$

$3! = 3 * 2!;$

$2! = 2 * 1!;$

$1! = 1;$

$0! = 1;$

Exemplo de função recursiva

Um exemplo clássico de função recursiva é o cálculo do fatorial de um número:
4! = 4 * 3!;
3! = 3 * 2!;
2! = 2 * 1!;
1! = 1;

0! = 1;
Generalizando....
N! = N * (N-1)!

Codificando uma função recursiva

```c
#include <stdio.h>

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

void main(){
    int fat;
    fat = fatorial(4);
    printf("Fatorial de 4 = %d", fat);
}
```

Pilhas de execução

- Como o computador executa um código recursivo?

Pilhas de execução

- Como o computador executa um código recursivo?
  - Todo programa C consiste em uma ou mais funções (o `main` é a primeira função a ser executada). Para administrar as chamadas de funções, o computador usa uma **pilha de execução**:
    - Quando uma função é chamada, o computador cria um novo "espaço de trabalho"para a função e aloca esse espaço em uma pilha (por cima do espaço de trabalho de quem invocou a função);
    - Quando a execução da função termina ela é removida da pilha;
    - A função que está no topo da pilha é a função que será executada.

Funcionamento da pilha de execução

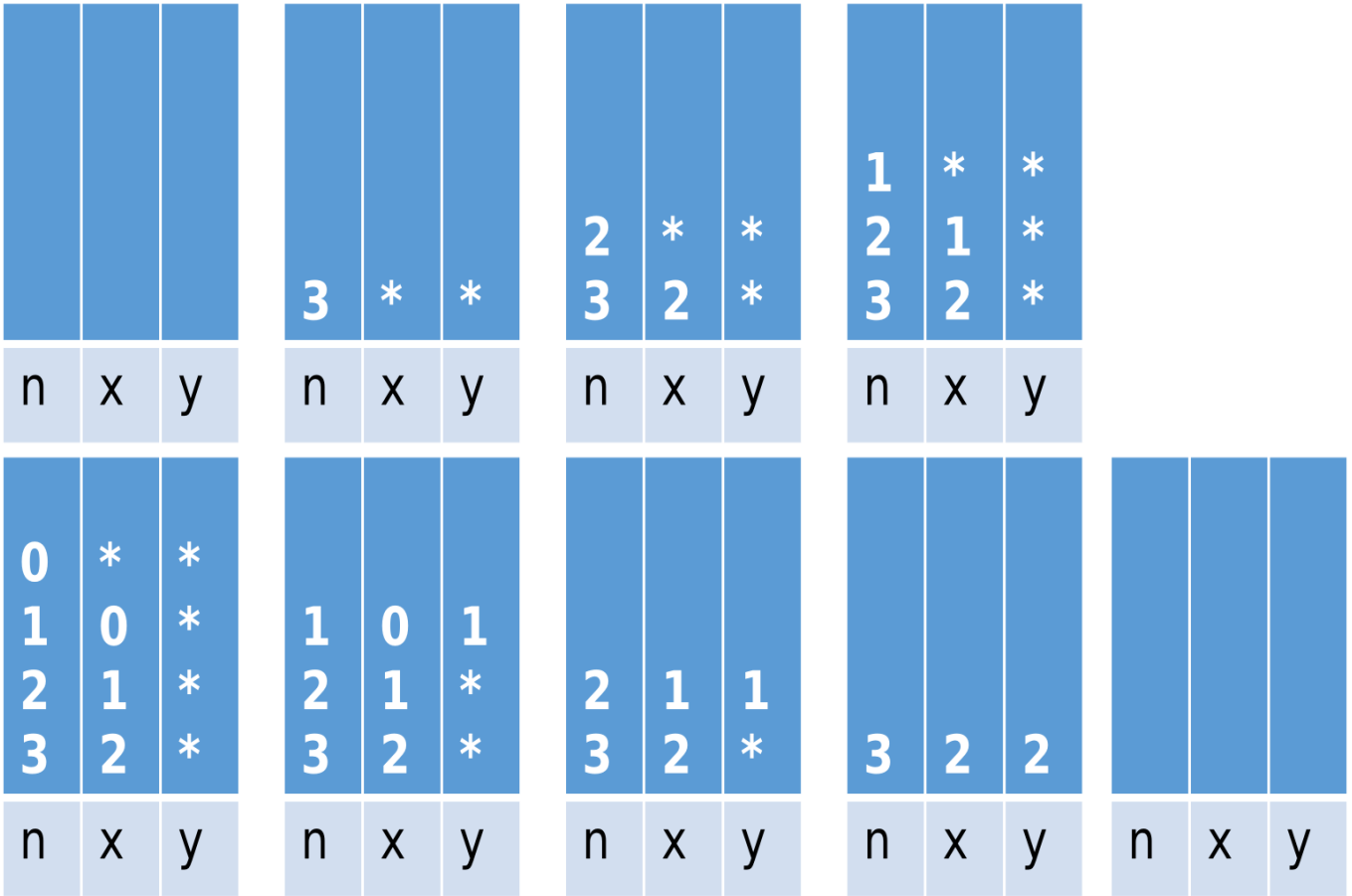
```c
#include <stdio.h>

void A() {
 printf("Sou a funcao A.\n");
}

void B () {
 printf("Sou a funcao B.Vou chamar A!");
 A();
 printf("A funcao A ja terminou, tchau!\n");
}

void main () {
 printf("Ola mundo! Vou chamar a funcao B!\n");
 B();
 printf("Adeus mundo!");
}
```

Funcionamento da pilha de execução

```
1  #include <stdio.h>
2
3  int fatorial(n){
4   if(n == 0)
5    return 1;
6   else
7    return n * fatorial(n-1);
8  }
9
10 void main(){
11  int fat;
12  fat = fatorial(4);
13 }
```

Funcionamento da pilha de execução

Exemplo ilustrativo da pilha de execução para o cálculo de 3!



Recursão na matemática

- Como definir recursivamente a soma abaixo?

$$\sum_{k=m}^{n} k = m + (m+1) + ... + (n-1) + n$$

Recursão na matemática

- Uma possível definição recursiva é:

$$\sum_{k=m}^{n} k = \begin{cases} m & \text{se } n = m \\ m + \sum_{k=m+1}^{n} k & \text{se } n > m \end{cases}$$

Recursão na matemática

```c
#include <stdio.h>

int soma(int m, int n){
  if(m == n)
    return n;
  else
    return m + soma(m+1, n);
}


void main() {
  printf("%d\n", soma(2, 6));

}
```

Busca em um vetor ordenado

- Busca: localizar um elemento dentro de um vetor em que os elementos estão em uma determinada ordem.
- Atividade MUITO conhecida em computação: necessidade de métodos eficientes.
- Método mais simples: busca sequencial ou linear.

Busca em um vetor ordenado

Definição recursiva:

- Busca binária (sequências ordenadas em ordem crescente):
  - Se o vetor tiver apenas um elemento: caso trivial;
  - Caso contrário, compare o item sendo procurado ao item posicionado no meio do vetor. Se forem iguais a busca terminou;
  - Se o item do meio for maior que o item sendo procurando, o processo de busca deve ser repetido na primeira metade; caso contrário, o processo deve ser repetido na segunda metade.

A cada comparação, o número de elementos a pesquisar é reduzido pela metade!!

Busca em um vetor ordenado

```
1  // Recebe um vetor crescente v[e+1..d-1]
2  // e um inteiro x tal que v[e] < x <= v[d]
3  // e devolve um indice j em e+1..d tal que
4  // v[j-1] < x <= v[j].
5
6  int bb (int x, int e, int d, int v[]) {
7      if (e > d) return -1;
8      else {
9          int m = (e + d)/2;
10         if (v[m] == x) return m;
11         if (v[m] < x)
12             return bb (x, m+1, d, v);
13         else
14             return bb (x, e, m-1, v);
15     }
16 }
```

Torre de Hanói

Estratégia:

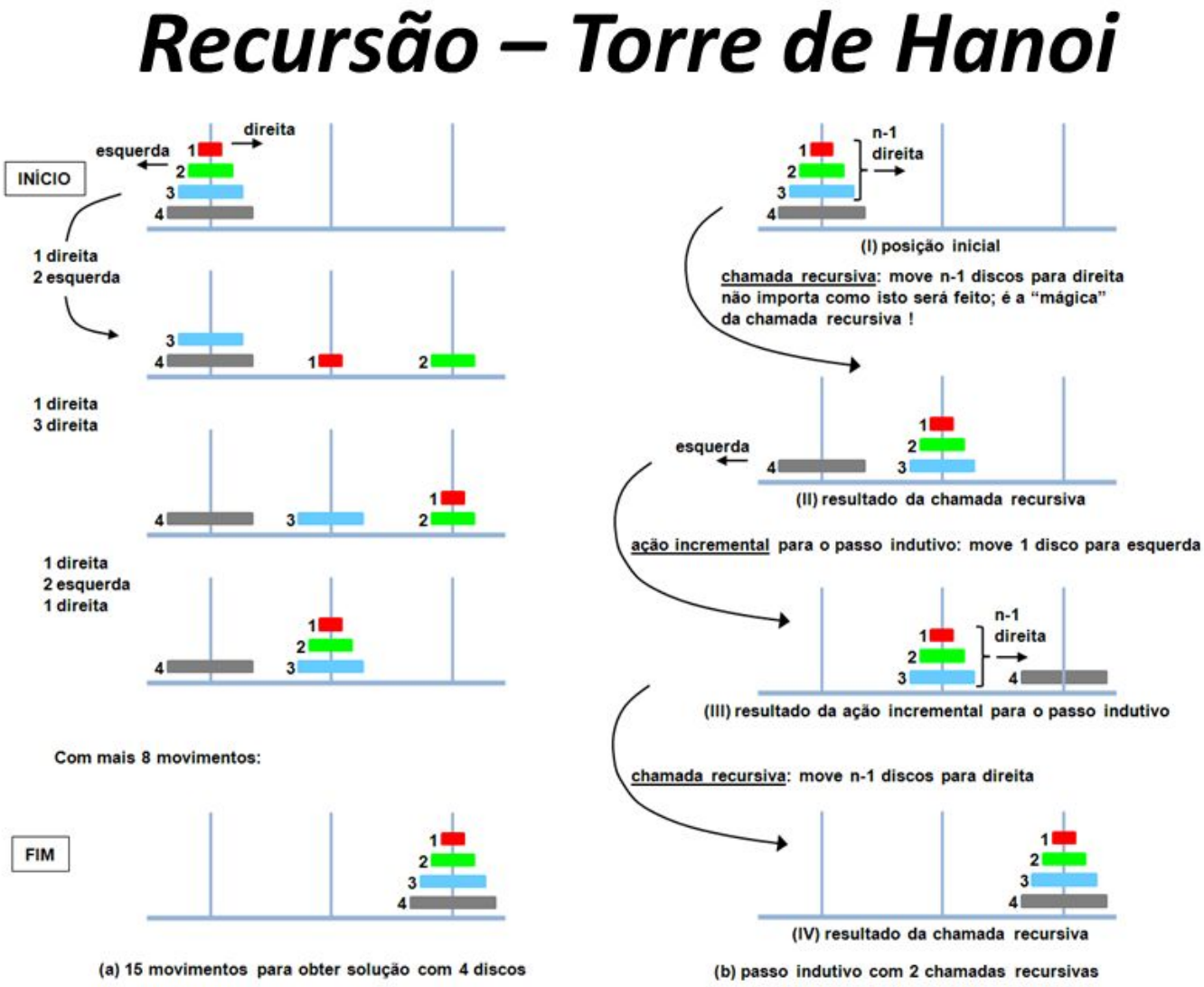
1. Mover n-1 discos para a torre auxiliar
2. Mover o disco maior para a torre destino
3. Mover n-1 discos para a torre destino

Torre de Hanói

```
void move_hanoi (int n, int origem, int destino,
                 int auxiliar) {
  if (n == 0) return;
  if (n == 1) {
    move_disco(origem, destino);
    return;
  }
  move_hanoi(n-1, origem, auxiliar, destino);
  move_disco(origem, destino);
  move_hanoi(n-1, auxiliar, destino, origem);
}
```

`n` é o número de discos.

Torre de Hanói



Recursividade indireta e cadeias recursivas

"Funções podem ser recursivas (invocar a si próprias) indiretamente, fazendo isto através de outras funções: assim, "P"pode chamar "Q"que chama "R"e assim por diante, até que "P"seja novamente invocada."

```
1
2  p () {
3        ...
4        b();
5        ...
6  }
7
8  b () {
9        ...
10       p();
11       ...
12 }
```

Estouro de pilha de execução

"O que acontece se a função não tiver um caso base?"

Estouro de pilha de execução

"O que acontece se a função não tiver um caso base?"

"O sistema de execução não consegue implementar infinitas chamadas!!!"

Algumas considerações

- Três pontos devem ser lembrados quando queremos construir uma função recursiva:
  1. Definir o problema em termos recursivos (definir o problema usando ele próprio na definição);
  2. Encontrar a condição básica da função (condição de término);
  3. Toda vez que a função recursiva for chamada ela tem que estar mais próxima de satisfazer a condição básica.

Recursão vs. Iteração

- Quando não usar recursão?
  - Solução recursiva causa ineficiência quando comparada com a versão iterativa;
  - Quando for conhecida uma solução obvia que utilize a técnica de iteração;
  - Quando não é possível prever se o número de chamadas recursivas irá causar sobrecarga na pilha de execução.

Recursão vs. Iteração

- Quando usar recursão?
  - O problema é naturalmente recursivo e a versão recursiva do algoritmo não gera ineficiência se comparado com o algoritmo iterativo;
  - Quando o algoritmo de se torna compacto sem perda de eficiência;
  - Usar quando o algoritmo requer uso explícito de pilha: QuickSort, percurso em árvore...

Referências

- *Treinamento em linguagem C*. Victorine Viviane Mizrahi;
- *Introduction to Algorithms*. T. Cormen, C. Leiserson, R. Rivest, C. Stein. Hill. 2001.
- Notas de aula do professor Paulo Feofiloff (http://www.ime.usp.br/ pf/algoritmos/);