Recursividade

Recursão

- É uma técnica de programação na qual um método chama a si mesmo.
- Uma função é dita recursiva quando dentro dela é feita uma ou mais chamadas a ela mesma.
- A idéia é dividir um problema original um subproblemas menores de mesma natureza (divisão) e depois combinar as soluções obtidas para gerar a solução do problema original de tamanho maior (conquista).
- Os subproblemas são resolvidos recursivamente do mesmo modo em função de instâncias menores, até se tornarem problemas triviais que são resolvidos de forma direta, interrompendo a recursão.

Recursividade

- Considere por exemplo a operação de multiplicação, em termos da operação de adição:
- Multiplicar m por n (onde n não é negativo) é somar m, n vezes:

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

- Uma solução para os problemas que realizam operações repetidamente pode ser implementação usando comandos de repetição (também chamados de comandos iterativos ou comandos de iteração).

IMPLEMENTAÇÃO ITERATIVA DA MULTIPLICAÇÃO

```
public static int mult (int m, int n) {
     int r=0;
     for (int i=1; i<=n; i++) r += m;
     return r;
}

public static void main (String args[]){

     int resultado = Recursao.mult (3,5);
     System.out.println (resultado);

}
```

MULTIPLICAÇÃO RECURSIVA

- Podemos também implementar a multiplicação de um número m por n somando m com a multiplicação de m por n-1.
  - m x n = m+m x (n-1)
  - 2x4 = 2 + 2x(3)
- Chamamos novamente a operação de multiplicação, mas agora para resolver um sub-problema que é parte do anterior.

Um método que chama a si mesmo é chamado de **método recursivo**.

MULTIPLICAÇÃO RECURSIVA

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

$$m \times n = 0 \text{ se } n == 0$$

$$m \times n = m + (m \times (n - 1)) \text{ se } n > 0$$

- Que pode ser implementado em Java da seguinte maneira:

```
public static int multr (int m, int n) {
    if (n==0) return 0;
    else return (m + multr(m, n-1));
}
```

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

FUNÇÕES RECURSIVAS

- **Exemplo: Calcular o fatorial de um número.**
  - **Solução não recursiva**

```c
#include <stdio.h>

float fatorial(int n){
  float fat = 1.0;
  while(n>1){
    fat *= n;
    n--;
  }
  return fat;
}
int main(){
  float fat;
  fat = fatorial(6);
  printf("fatorial: %f\n",fat);
  return 0;
}
```

FUNÇÕES RECURSIVAS

- **Exemplo: Calcular o fatorial de um número.**
  - **Solução recursiva: n! = n.(n-1)!**

```c
#include <stdio.h>

float fatorial(int n){
  if(n==0)       //Caso trivial
    return 1.0; //Solução direta

  return n*fatorial(n-1); //Chamada recursiva
}
int main(){
  float fat;
  fat = fatorial(6);
  printf("fatorial: %f\n",fat);
  return 0;
}
```

FUNÇÕES RECURSIVAS

- **Exemplo: Calcular x elevado a n positivo.**
  - **Solução não recursiva**

```
#include <stdio.h>


float potencia(float x, int n){
  float pot=1.0;
  while(n>0){
    pot *= x;
    n--;
  }
  return pot;
}
```

FUNÇÕES RECURSIVAS

- **Exemplo: Calcular x elevado a n positivo.**
  - **Solução recursiva: x^n = x . x^(n-1)**

```
#include <stdio.h>


float potencia(float x, int n){
  if(n==0)          //Caso trivial
    return 1.0;     //Solução direta
  else
    return x*potencia(x, n-1); //Chamada recursiva
}
```

FUNÇÕES RECURSIVAS

- **Exemplo: Calcular x elevado a n positivo.**
  - **Solução recursiva: x^n = x^(n/2). x^(n/2) = (x^(n/2))^2**

```
#include <stdio.h>

//Função recursiva mais eficiente
float potencia(float x, int n){
  float pot;

  if(n==0) return 1.0; //Caso trivial
  if(n%2==0){ //Se n é par...
    pot = potencia(x, n/2);
    return pot*pot;
  }
  else{ //Se n é ímpar...
    pot = potencia(x, n/2);
    return pot*pot*x;
  }
}
```

FUNÇÕES RECURSIVAS

- **Exemplo: Encontrar maior elemento de um vetor.**
  - **Solução recursiva**

```c
#include <stdio.h>
int maiorinteiro(int v[], int n){
  int m;
  if(n==1) return v[0]; //Caso trivial
  else{
    m = maiorinteiro(v, n-1);
    if(m>v[n-1]) return m;
    else         return v[n-1];
  }
}
int main(){
  int max,v[5]={8,1,9,4,2};
  max = maiorinteiro(v, 5);
  printf("Max: %d\n",max);
  return 0;
}
```

FUNÇÕES RECURSIVAS

- **Exemplo: Imprimir elementos de um vetor.**
  - **Solução não recursiva**

```c
#include <stdio.h>

void printvetor(int v[], int n){
  int i;
  for(i=0; i<n; i++)
    printf("%d ",v[i]);
}
```

  - **Solução recursiva**

```c
#include <stdio.h>

void printvetor(int v[], int n){
  if(n>1)
    printvetor(v, n-1);
  printf("%d ",v[n-1]);
}
```

FUNÇÕES RECURSIVAS

- **Ordenação de vetores:**
  - **Ordenação por seleção *(Selection Sort)***
    - Percorre o vetor selecionando o maior elemento.
    - Troca com o da última posição, de forma que o maior passa a ocupar sua posição definitiva.
    - Repete o processo para os elementos ainda fora de posição.
  - **Ordenação por inserção *(Insertion Sort)***
    - Ordenamos parte do vetor.
    - Pega próximo elemento.
    - Insere na posição correta da parte já ordenada.
  - **Ordenação por permutação *(Bubble Sort)***
    - O vetor é percorrido a partir do início e trocas são feitas sempre que um elemento for maior que o próximo.
    - O maior passa a ocupar sua posição definitiva.
    - Repete o processo para os demais elementos fora de posição.

FUNÇÕES RECURSIVAS

- **Exemplo: Ordenar vetor por seleção.**
  - **Solução recursiva**

```
#include <stdio.h>
void selectionsort(int v[], int n){
  int i,im,tmp;
  if(n>1){
    im = 0; //im = índice do maior valor
    for(i=1; i<n; i++)
      if(v[i]>v[im]) //Seleciona o maior valor
        im = i;
    if(im!=n-1){ //Efetua troca
      tmp = v[n-1];
      v[n-1] = v[im]; //Move maior para o final
      v[im] = tmp;
    }
    selectionsort(v, n-1);
  }
}
```

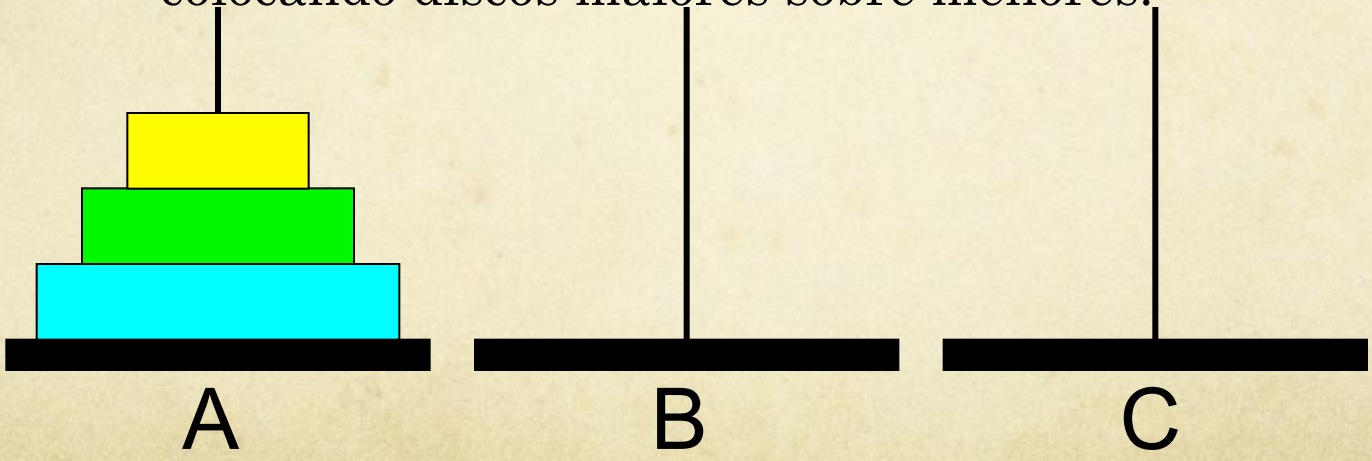
FUNÇÕES RECURSIVAS

- **Exemplo: Ordenar vetor por inserção.**
  - **Solução recursiva**

```
#include <stdio.h>
void insertionsort(int v[], int n){
  int i,tmp;
  //No caso trivial não faz nada
  if(n>1){
    insertionsort(v, n-1);
    //Insere elemento que falta na posição correta
    i = n-1;
    while((i>0) && (v[i-1]>v[i])){
      tmp = v[i-1];
      v[i-1] = v[i];
      v[i] = tmp;
      i--;
    }
  }
}
```

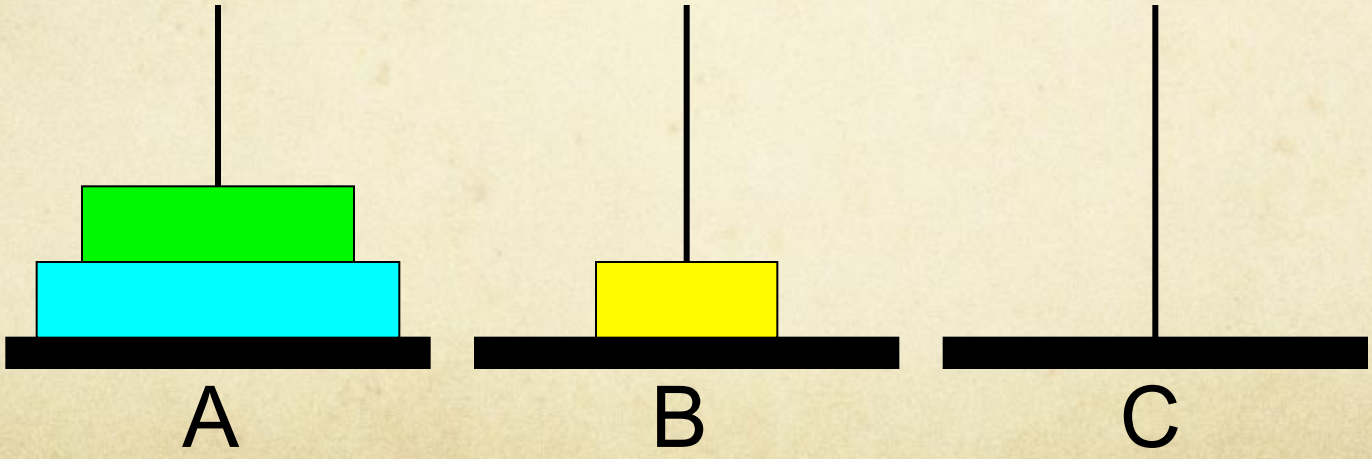
FUNÇÕES RECURSIVAS

- **Exemplo: Torre de Hanoi**
  - São dados um conjunto de N discos com diferentes tamanhos e três bases A, B e C.
  - O problema consiste em imprimir os passos necessários para transferir os discos da base A para a base B, usando a base C como auxiliar, nunca colocando discos maiores sobre menores.



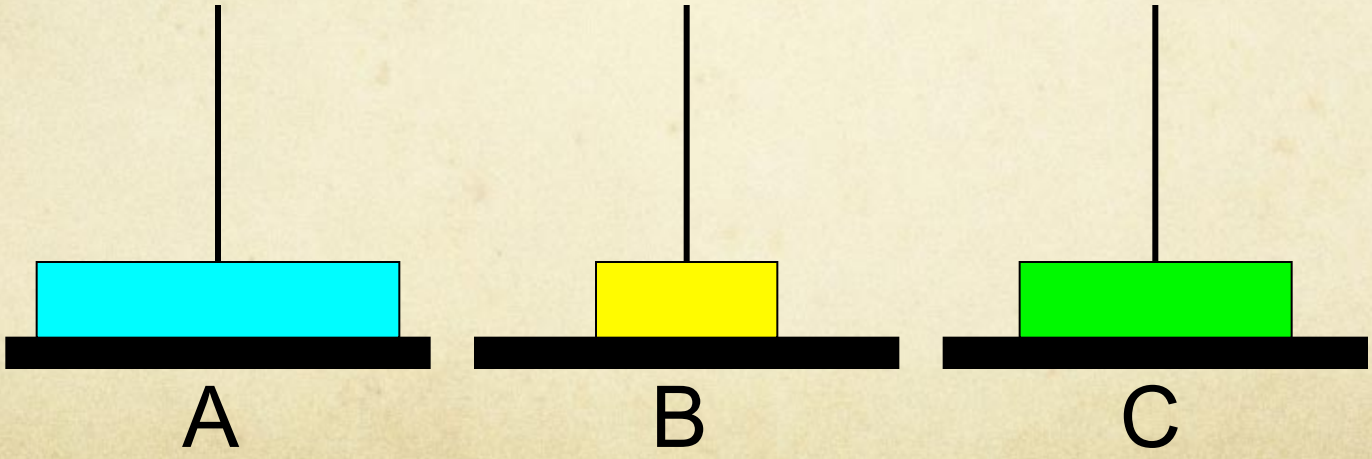
FUNÇÕES RECURSIVAS

- **Exemplo: Torre de Hanoi**
  - **<u>1° passo:</u>** Mover de A para B.



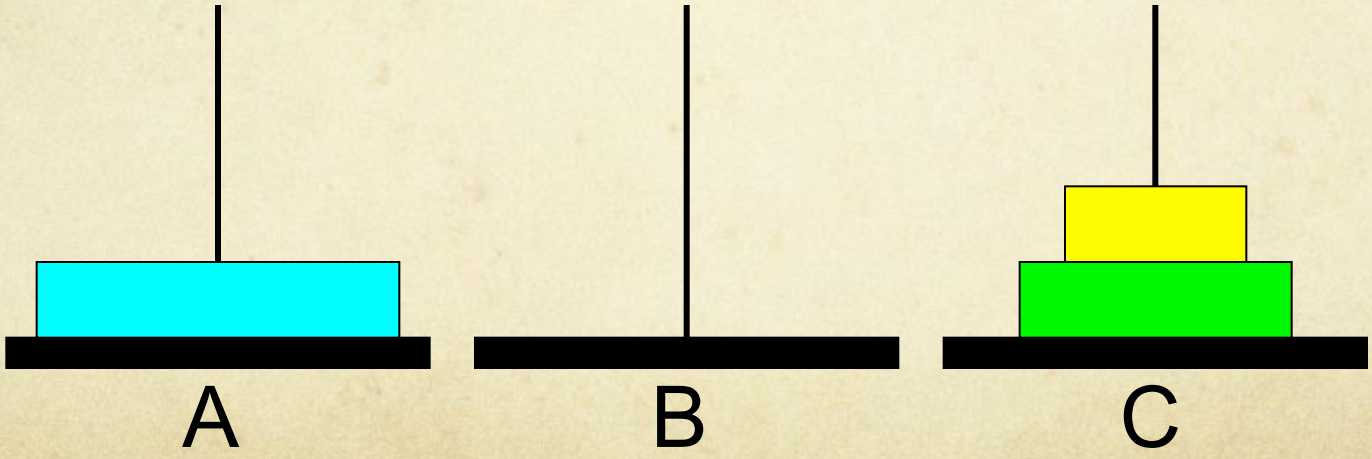
FUNÇÕES RECURSIVAS

- **Exemplo: Torre de Hanoi**
  - **<u>2° passo:</u>** Mover de A para C.



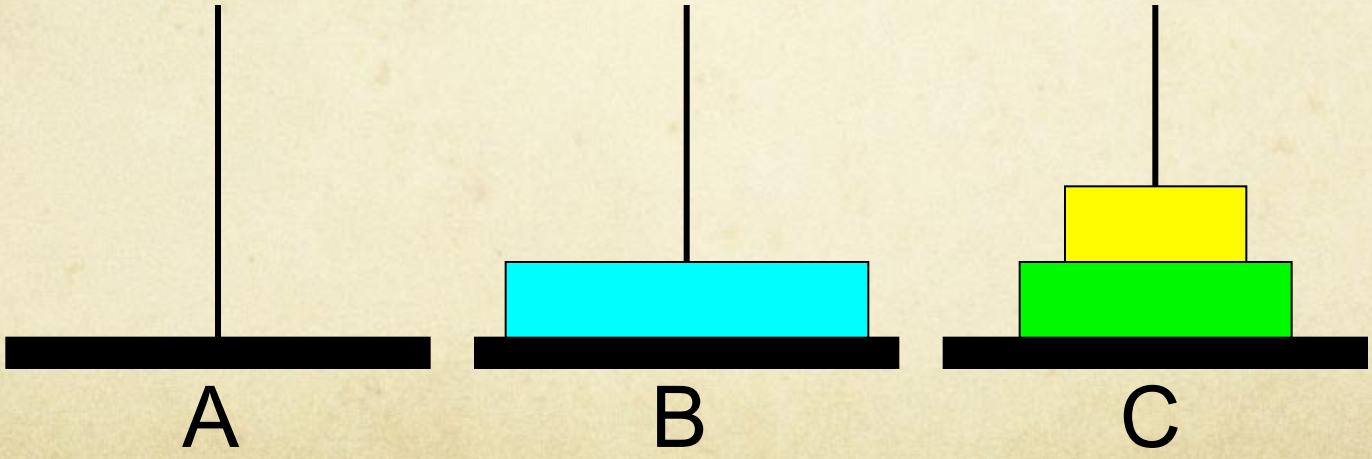
FUNÇÕES RECURSIVAS

- **Exemplo: Torre de Hanoi**
  - **3° passo:** Mover de B para C.



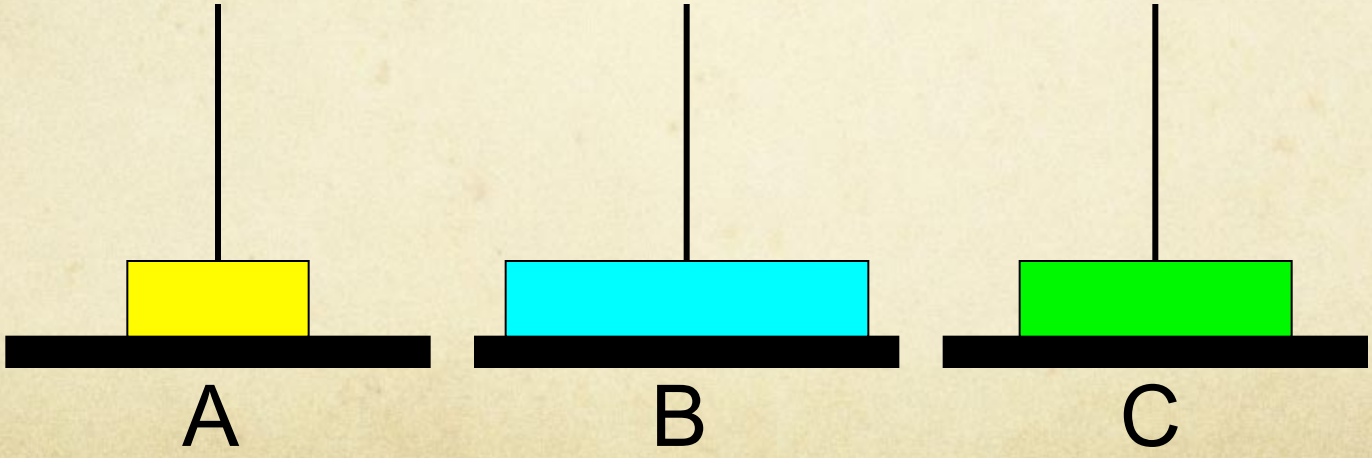
FUNÇÕES RECURSIVAS

- **Exemplo: Torre de Hanoi**
  - **4° passo:** Mover de A para B.



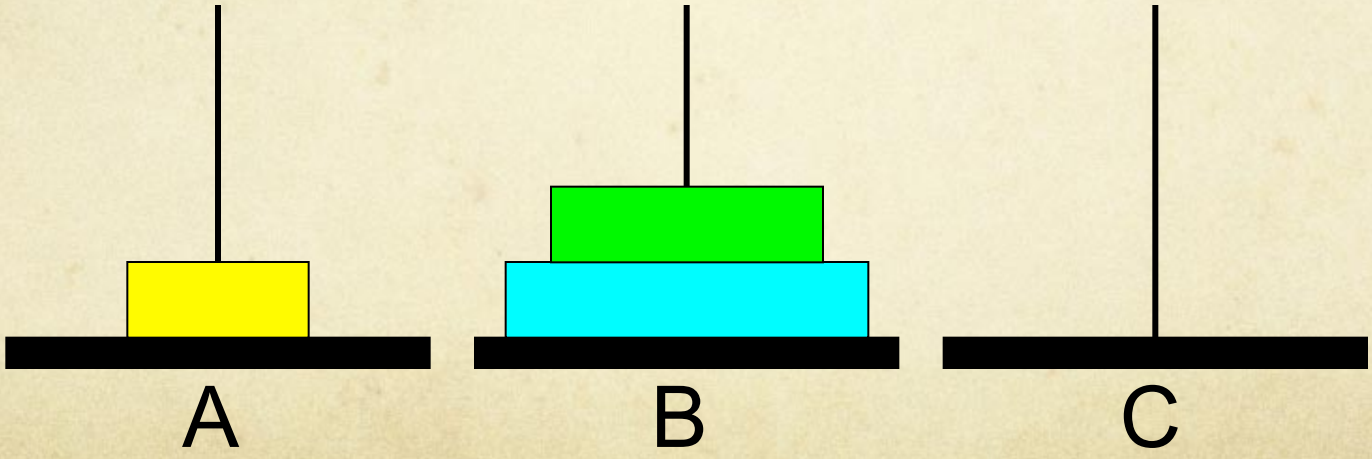
FUNÇÕES RECURSIVAS

- **Exemplo: Torre de Hanoi**
  - **5° passo:** Mover de C para A.



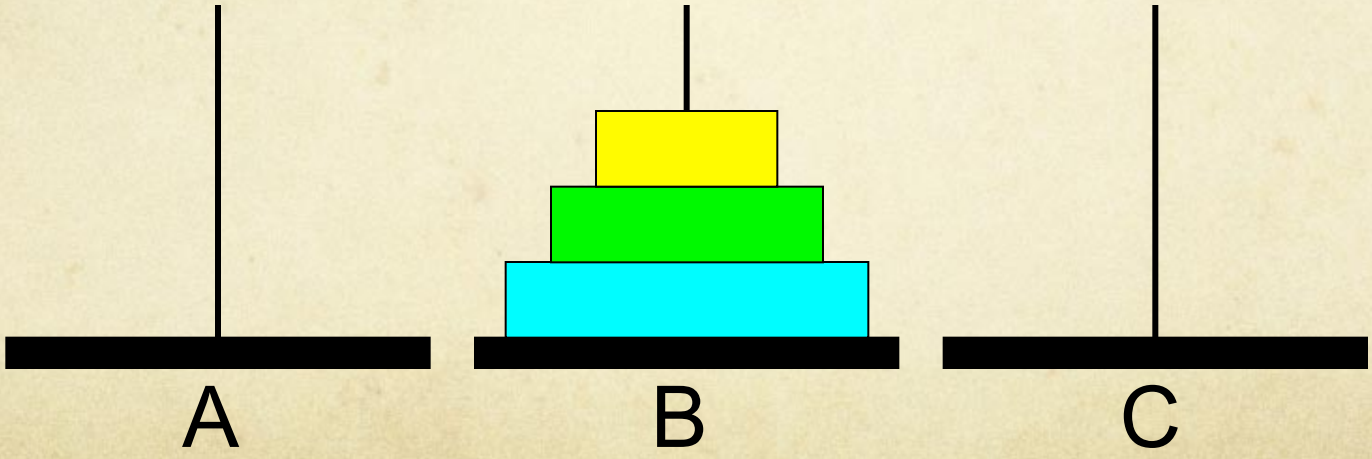
FUNÇÕES RECURSIVAS

- **Exemplo: Torre de Hanoi**
  - **<u>6° passo:</u>** Mover de C para B.



FUNÇÕES RECURSIVAS

- **Exemplo: Torre de Hanoi**
  - **7° passo:** Mover de A para B.



FUNÇÕES RECURSIVAS

- **Exemplo:** Torre de Hanoi

```c
#include <stdio.h>
void hanoi(int n, char orig, char dest, char aux){
  if(n==1)
    printf("1 -> %c\n",dest);
  else{
    hanoi(n-1, orig, aux, dest);
    printf("%d -> %c\n",n,dest);
    hanoi(n-1, aux, dest, orig);
  }
}
int main(){
  int n;
  printf("Número de discos: ");
  scanf("%d",&n);
  hanoi(n, 'A', 'B', 'C');
  return 0;
}
```

Fatorial recursivo

- Definição não recursiva (tradicional):

  N! = 1, para N = 0.

  N! = 1 x 2 x 3 x .... x N, para N>0

- Definição recursiva:

  N! = 1, para N = 0;

  N! = N x (N - 1)!, para N > 0.

Fatorial recursivo

- Definição não recursiva (tradicional):

  N! = 1, para N = 0.

  N! = 1 x 2 x 3 x .... x N, para N>0

- implementação iterativa:

```
public static int fatorial (int numero) {
    int resultado = 1;
    for(int i=numero;i>0;i--){
        resultado = resultado * i;
    }
    return resultado;
}
```

Fatorial recursivo

Definição recursiva:

N! = 1, para N <= 1;

N! = N x (N - 1)!, para N > 0.

```
public static int fatorialr(int n){
    if(n<=1){
        return 1;
    }else{
        return n*fatorialr(n-1);
    }

}
```

SEQÜÊNCIA DE FIBONACCI

- A seqüência de Fibonacci é a seqüência de inteiros:
  - 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...

- Cada elemento nessa seqüência é a soma dos dois elementos anteriores. Por exemplo:
  - 0+1=1; 1+1=2; 1+2=3; 2+3=5 ...

- Se partirmos que fib(0)=0 e fib(1)=1 e assim por diante, podemos definir um número da seqüência fibonnaci da seguinte maneira:

fib(n)=n se n==0 OU n==1

fib(n)=fib(n-2)+fib(n-1) se n>=2

SEQÜÊNCIA DE FIBONACCI (IMPLEMENTAÇÃO)

fib(n)=n se n==0 OU n==1

fib(n)=fib(n-2)+fib(n-1) se n>=2

```
public static int fibonacci (int n){
    if (n<=1) return n;
    else return fibonacci (n-2) + fibonacci(n-1);
}
```

CHAMADA DE MÉTODO

- Quando um método é chamado:
  - É necessário inicializar os parâmetros formais com os valores passados como argumento;
  - O sistema precisa saber onde reiniciar a execução do programa;

- Informações de cada método (variáveis e endereço de retorno) devem ser guardadas até o método acabar a sua execução.

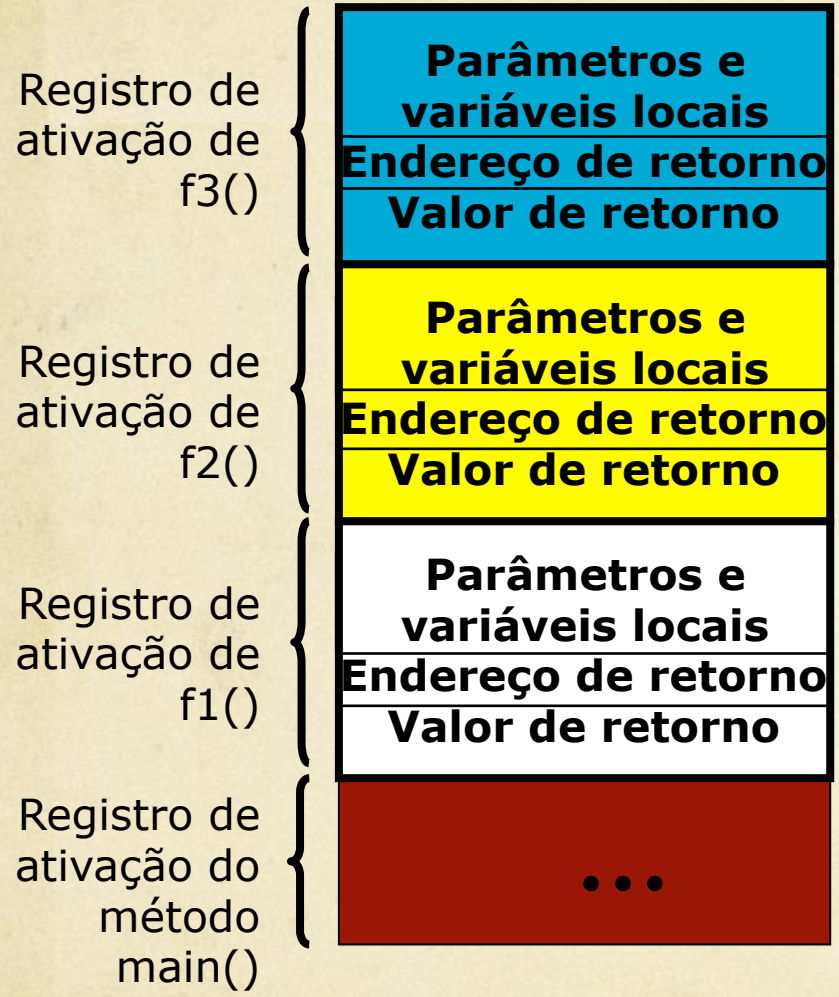
- Mas como o programa diferencia a variável n da primeira chamada da variável n da segunda chamada do método fatorialr?

```
public static int fatorialr(int n){
    if(n<=1){
        return 1;
    }else{
        return n*fatorialr(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 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.

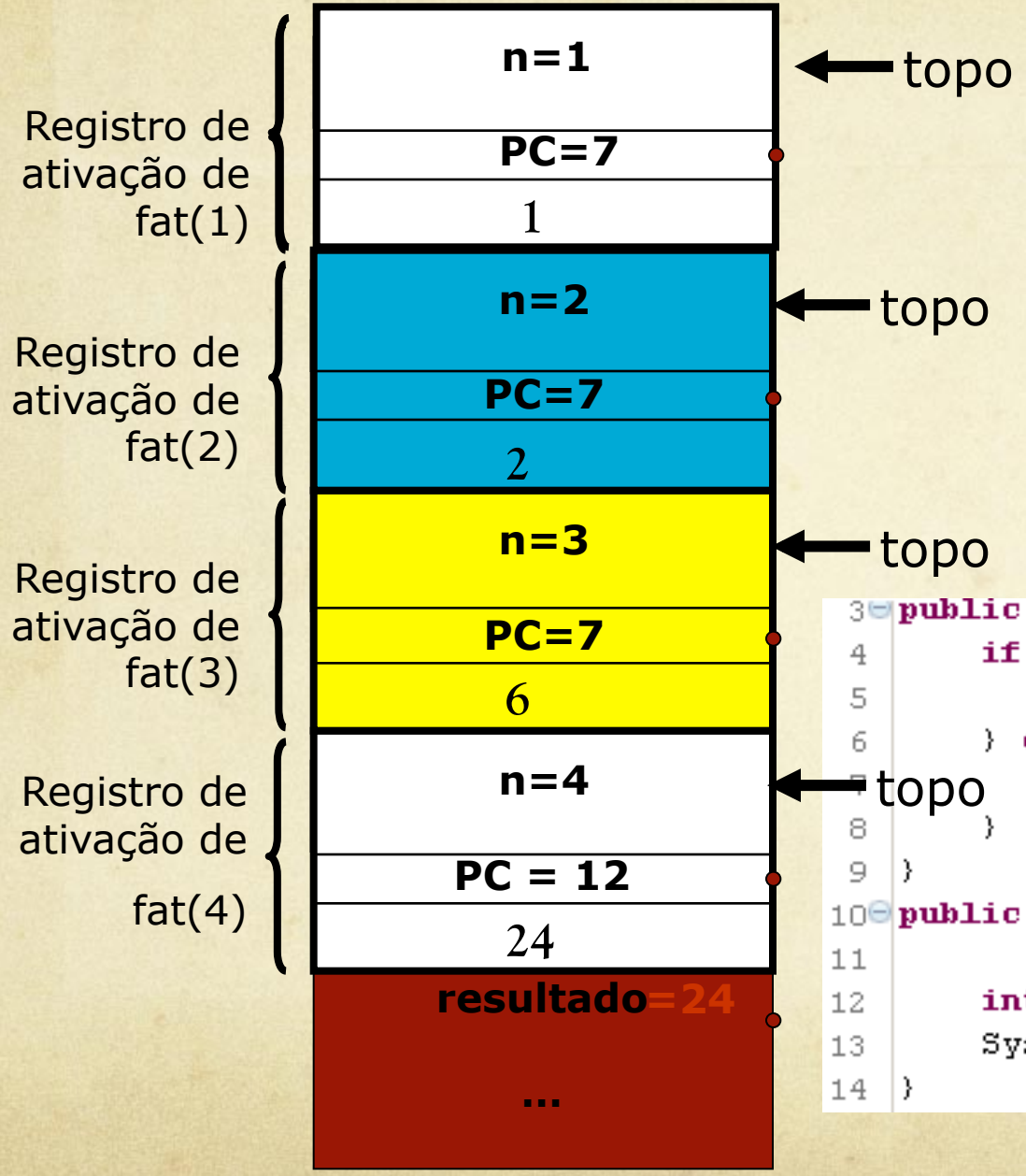
REGISTRO DE ATIVAÇÃO




```
public static void f3() {
    int x=3;
}

public static void f2() {
    int x=2;
    f3();
}

public static void f1() {
    int x=1;
    f2();
}

public static void main (String args[]){
    f1();
}
```

REGISTRO DE ATIVAÇÃO: EXEMPLO

fatorial de 4



fat (4) → main

$4*fat(3)$ =24

$3*fat(2)$ =6

$2*fat(1)$ =2

1

```
3  public static int fatorialr(int n) {
4      if (n <= 1) {
5          return 1;
6      } else {
7          return n * fatorialr(n - 1);
8      }
9  }
10 public static void main(String args[]) {
11
12     int resultado = Recursao.fatorialr(4);
13     System.out.println(resultado);
14 }
```

REGISTRO DE ATIVAÇÃO: EXEMPLO

- 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.

Dicas para desenvolver algoritmos recursivos

- Montar, inicialmente, uma definição (especificação) recursiva do problema, como segue:
  1. Definir pelo menos um caso básico;
  2. Quebrar o problema em subproblemas, definindo o(s) caso (s) recursivo(s);
  3. Fazer o teste de finitude, isto é, certificar-se de que as sucessivas chamadas recursivas levam obrigatoriamente, e numa quantidade finita de vezes, ao(s) caso(s) básico(s).
- Depois, é só traduzir essa especificação para a linguagem de programação.

Vantagens e Desvantagens

- **Vantagens da recursão**
  - Redução do tamanho do código fonte
  - Maior clareza do algoritmo para problemas de definição *naturalmente recursiva*
- **Desvantagens da recursão**
  - Baixo desempenho na execução devido ao tempo para gerenciamento das chamadas
  - Dificuldade de depuração dos subprogramas recursivos, principalmente se a recursão for muito profunda

Tradução X Interpretação

- Uma linguagem de programação pode ser convertida, ou traduzida, em código de máquina por compilação ou interpretação, que juntas podem ser chamadas de **tradução**.

- Se o texto do programa é traduzido à medida que vai sendo executado, como em JavaScript, Python ou Perl, num processo de tradução de trechos seguidos de sua execução imediata, então diz-se que o programa foi interpretado e que o mecanismo utilizado para a tradução é um interpretador. Programas interpretados são geralmente mais lentos do que os compilados, mas são também geralmente mais flexíveis, já que podem interagir com o ambiente mais facilmente.

Tradução X Interpretação

- Se o método utilizado traduz todo o texto do programa (também chamado de código), para só depois executar (ou rodar, como se diz no jargão da computação) o programa, então diz-se que o programa foi compilado e que o mecanismo utilizado para a tradução é um compilador (que por sua vez nada mais é do que um programa). A versão compilada do programa tipicamente é armazenada, de forma que o programa pode ser executado um número indefinido de vezes sem que seja necessária nova compilação, o que compensa o tempo gasto na compilação. Isso acontece com linguagens como Pascal e C.

Tradução

- A tradução é tipicamente feita em várias fases, sendo as mais comuns:
  - Análise léxica
  - Análise sintática ou Parsing
  - Análise Semântica
  - Geração de código e a Otimização.
  - Em compiladores também é comum a Geração de código intermediário.

Análise léxica

- É o processo de analisar a entrada de linhas de caracteres (tal como o código-fonte de um programa de computador) e produzir uma seqüência de símbolos chamado "símbolos léxicos" (lexical tokens), ou somente "símbolos" (tokens), que podem ser manipulados mais facilmente por um parser (leitor de saída).

- A **Análise Léxica** é a forma de verificar determinado alfabeto. Quando analisamos uma palavra, podemos definir através da análise léxica se existe ou não algum caractere que não faz parte do nosso alfabeto, ou um alfabeto inventado por nós. O analisador léxico é a primeira etapa de um compilador, logo após virá a análise sintática.

Análise sintática

- Também conhecida pelo termo em inglês *parsing* é o processo de analisar uma seqüência de entrada (lida de um arquivo de computador ou do teclado, por exemplo) para determinar sua estrutura gramatical segundo uma determinada gramática formal. Essa análise faz parte de um compilador, junto com a análise léxica e análise semântica.

- A análise sintática transforma um texto na entrada em uma estrutura de dados, em geral uma árvore, o que é conveniente para processamento posterior e captura a hierarquia implícita desta entrada. Através da análise léxica é obtido um grupo de tokens, para que o analisador sintático use um conjunto de regras para construir uma árvore sintática da estrutura.

- Em termos práticos, pode também ser usada para decompor *um texto* em unidades estruturais para serem organizadas dentro de um bloco, por exemplo.

Análise semântica

- É um sinônimo de Análise sintática e é a terceira fase da compilação onde se verifica os erros semânticos, (por exemplo, uma multiplicação entre tipos de dados diferentes) no código fonte e coleta as informações necessárias para a próxima fase da compilação que é a geração de código objeto.

Gerador de Código

- Dentro do diversificado leque de categorias de ferramentas que prestam apoio às atividades da Engenharia de Software (CASE), uma específica vem ganhando cada vez mais destaque e, sobre ela, tem-se aplicado muito investimento nos últimos tempos: as **Ferramentas de Geração de Código**, ou simplesmente **Geradores de Código**.

- Dessa forma, Gerador de Código é aquela ferramenta que possui a capacidade de gerar código a partir de um determinado modelo de software. Inclusive, de acordo com alguns pontos de vista e a partir das características específicas do tipo de Gerador de Código, ele passa a ser conversor de códigos de linguagens distintas. Isso acontece, por exemplo, com o compilador, que transforma um código escrito através de uma linguagem de programação para código de máquina ou código objeto.

Otimização

- Em matemática, o termo **otimização**, ou **programação matemática**, refere-se ao estudo de problemas em que se busca minimizar ou maximizar uma função através da escolha sistemática dos valores de variáveis reais ou inteiras dentro de um conjunto viável.

- A Otimização de código é a estratégia de examinar o código intermediário, produzido durante a fase de geração de código com objetivo de produzir, através de algumas técnicas, um código que execute com bastante eficiência. O nome otimizador deve sempre ser encarado com cuidado, pois não se pode criar um programa que leia um programa **P** e gere um programa **P'** equivalente sendo melhor possível segundo o critério adotado. Várias técnicas e várias tarefas se reúnem sob o nome de Otimização. Estas técnicas consistem em detectar padrões dentro do código produzido e substituí-los por códigos mais eficientes

Geração de código intermediário

- Ocorre a transformação da árvore sintática em uma representação intermediária do código fonte. Um tipo popular de linguagem intermediária é conhecido como código de três endereços. Neste tipo de código uma sentença típica tem a forma **X := A op B**, onde **X**,**A** e **B** são operandos e **op** uma operação qualquer. Uma forma prática de representar sentenças de três endereços é através do uso de quádruplas (**operador, argumento-1, argumento-2 e resultado**). Este esquema de representação de código intermediário é preferido por diversos compiladores, principalmente aqueles que executam extensivas otimizações de código, uma vez que o código intermediário pode ser rearranjado de uma maneira conveniente com facilidade.

Técnica de Programação Linear

- Em matemática, problemas de Programação Linear são problemas de otimização nos quais a função objetivo e as restrições são todas lineares.
  Programação Linear é uma importante área da otimização por várias razões. Muitos problemas práticos em pesquisa operacional podem ser expressos como problemas de programação linear. Certos casos especiais de programação linear, tais como problemas de *network flow* e problemas de *multicommodity flow* são considerados importantes o suficiente para que se tenha gerado muita pesquisa em algoritmos especializados para suas soluções. Vários algoritmos para outros tipos de problemas de otimização funcionam resolvendo problemas de PL como sub-problemas. Historicamente, idéias da programação linear inspiraram muitos dos conceitos centrais de teoria da otimização, tais como dualidade, decomposição, e a importância da convexidade e suas generalizações.

Técnica de Programação Modular

- Programação modular é um paradigma de programação no qual o desenvolvimento das rotinas de programação é feito através de módulos, que são interligados entre si através de uma interface comum.
  Foi apresentado originalmente pela Information & Systems Institute, Inc. no National Symposium on Modular Programming em 1968, com a liderança de Larry Constantine.

Técnica de Programação Estruturada

- Programação estruturada é uma forma de programação de computadores que preconiza que todos os programas possíveis podem ser reduzidos a apenas três estruturas: sequência, decisão e iteração.
  Tendo, na prática, sido transformada na Programação modular, a Programação estruturada orienta os programadores para a criação de estruturas simples em seus programas, usando as sub-rotinas e as funções. Foi a forma dominante na criação de software entre a programação linear e a programação orientada por objetos.
  Apesar de ter sido sucedida pela programação orientada por objetos, pode-se dizer que a programação estruturada ainda é marcantemente influente, uma vez que grande parte das pessoas ainda aprendem programação através dela. Porém, a orientação a objetos superou o uso das linguagens estruturadas no mercado

Técnica de Programação Orientada a Objeto

- Programação Orientada a Objetos (POO) é um paradigma de análise, projeto e programação de sistemas de software baseado na composição e interação entre diversas unidades de software chamadas de objetos.

- Em alguns contextos, prefere-se usar modelagem orientada ao objeto, em vez de programação. De fato, o paradigma "orientação a objeto" tem bases conceituais e origem no campo de estudo da cognição, que influenciou a área de inteligência artificial e da lingüística no campo da abstração de conceitos do mundo real. Na qualidade de método de modelagem, é tida como a melhor estratégia, e mais natural, para se eliminar o "gap semântico", dificuldade recorrente no processo de modelar o mundo real, no domínio do problema, em um conjunto de componentes de software que seja o mais fiel na sua representação deste domínio. Facilitaria a comunicação do profissional modelador e do usuário da área alvo, na medida em que a correlação da simbologia e conceitos abstratos do mundo real e da ferramenta de modelagem (conceitos, terminologia, símbolos, grafismo e estratégias) fosse a mais óbvia, natural e exata possível.

- A análise e projeto orientados a objetos tem como meta identificar o melhor conjunto de objetos para descrever um sistema de software. O funcionamento deste sistema se dá através do relacionamento e troca de mensagens entre estes objetos.
  Na programação orientada a objetos, implementa-se um conjunto de classes que definem os objetos presentes no sistema de software. Cada classe determina o comportamento (definido nos métodos) e estados possíveis (atributos) de seus objetos, assim como o relacionamento com outros objetos.

Classificação de Linguagens

- As linguagens de programação podem ser classificadas e sub-classificadas de várias formas.
  - Classificação da ACM
  - Quanto ao paradigma
  - Quanto a estrutura de tipos
  - Quanto ao grau de abstração
  - Quanto à geração

Classificação da *ACM*

- A ACM mantém um sistema de classificação com os seguintes sub-itens:
  - Linguagens aplicativas, ou de aplicação
  - Linguagens concorrentes, distribuídas e paralelas
  - Linguagens de fluxo de dados
  - Linguagens de projeto
  - Linguagens extensíveis
  - Linguagens de montagem e de macro
  - Linguagens de microprogramação
  - Linguagens não determinísticas
  - Linguagens não procedurais
  - Linguagens orientadas a objeto
  - Linguagens de aplicação especializada
  - Linguagens de altíssimo nível

Quanto ao Paradigma

- Diferentes linguagens de programação podem ser agrupadas segundo o paradigma que seguem para abordar a sua sintaxe:
- Linguagem funcional
- Linguagem natural
- Programação lógica
- Programação imperativa
  - Programação estruturada
  - Linguagem orientada a objetos

Quanto a estrutura de tipos

- **Fracamente tipada**, como Smalltalk, onde o tipo da variável muda dinamicamente conforme a situação.
- **Fortemente tipada**, como Java, Ruby, onde o tipo da variável, uma vez atribuído, se mantém o mesmo até ser descartada da memória.
- **Dinamicamente tipada**, como Perl, Python ou Ruby, onde o tipo da variável é definido em tempo de execução.
- **Estaticamente tipada**, como Java e C, onde o tipo da variável é definido em tempo de compilação.

Quanto ao grau de abstração

- **Linguagem de programação de baixo nível**, cujos símbolos são uma representação direta do código de máquina que será gerado, onde cada comando da linguagem equivale a um "opcode" do processador, como Assembly.

- **Linguagem de programação de médio nível**, que possui símbolos que podem ser convertidos diretamente para código de máquina (goto, expressões matemáticas, atribuição de variáveis), mas também símbolos complexos que são convertidos por um compilador. Exemplo: C, C++.

- **Linguagem de programação de alto nível**, composta de símbolos mais complexos, inteligível pelo ser humano e não-executável diretamente pela máquina, no nível da especificação de algoritmos, como Pascal, Fortran, ALGOL e SQL.

Quanto à geração

- **Primeira geração**, as linguagens de baixo nível
  - Assembly
- **Segunda geração**, as primeiras linguagens
  - Fortran, ALGOL,...
- **Terceira geração**, as procedurais e estruturadas
  - Pascal, C.
- **Quarta geração**, linguagens que geram programas em outras linguagens
  - Java, C++, linguagens de consulta SQL.
- **Quinta geração**, linguagens lógicas
  - Prolog.

	Modelo de execução	Influências	Paradigma principal	Modelo de tipo de dados	Introdução
C	Compilação	Algol, BCPL	Estruturada, Procedimental, Orientada por fluxo	Estático, fraco	Início de 1970
C++	Compilação	C, Simula, Algol 68	Principalmente orientada a objectos, múltiplos paradigmas	Estático, fraco	1979
Objective-C	Compilação	C, Smalltalk	Principalmente orientada a objectos, Reflectiva, Passagem de mensagens	Dinâmico e estático, fraco	1986
Python	Interpretação	ABC, Perl	Orientada a objectos	Dinâmico, forte	1990
Ruby	Interpretação	Smalltalk, Perl	Orientada a objectos	Dinâmico, forte	1995
Mathematica	Interpretação	LISP	Múltiplos paradigmas	Dinâmico, forte	1986
C#	Interpretação e Compilação	C++	Orientada a objectos, múltiplos paradigmas	Estático, forte	2002
Java	Interpretação e Compilação	C++	Orientada a objectos	Estático, forte	1996
Perl	Interpretação	C, Shell, awk, sed, Lisp	Funcional, Orientada a objectos e Procedural	Dinâmico	1987
Boo	Interpretação	Python	Orientada a objectos	Estático	2003
PHP	Interpretação	C e Perl	Orientada a objectos	Dinâmico	1995
Harbour	Interpretação e Compilação	Clipper e xBase	Estruturada, Procedimental, Orientada a objectos	Dinâmico, forte	1999

Principais linguagens de programação	
Linguagens históricas (2GL, 3GL)	ALGOL • APL • Assembly • AWK • B • BASIC • BCPL • COBOL • CPL • Forth • Fortran • Lisp • Logo • Simula • Smalltalk
Linguagens acadêmicas	Gödel • Haskell • Icon • Lisp • Logo • Lua • Pascal • Prolog • Scala • Scheme • Scratch • Simula • Smalltalk • Tcl
Linguagens proprietárias	ABAP • ActionScript • AWK • COBOL • Delphi • Eiffel • MATLAB • PL/SQL • PowerShell • Scratch • Visual Basic
Linguagens não-proprietárias	Ada • Assembly • C • C++ • C# • Icon • Lisp • Logo • Object Pascal • Objective-C • Pascal • Scheme • Simula • Smalltalk
Linguagens livres	Boo • D • Erlang • Go • Groovy • Harbour • Haskell • Java • JavaScript • Lua • Perl • PHP • Python • Ruby • Scala • Tcl
Linguagens esotéricas	Befunge • brainfuck • FALSE • INTERCAL • LOLCODE • Malbolge • PATH • Pbrain • SNUSP • Unlambda • Whitespace

O que o código abaixo faz?

```
#include <stdio.h>

#include <stdlib.h>

int main(int argc, char *argv[]) {

  int i, n;

  teste:

  printf("\nDigite um numero");

  scanf("%d", &n);

  if(n<10)

    goto teste;

  else if(n==10)

    for(i=0;i<n;i++)

       printf("%d - nada\n",(i+1));

  else goto fim;

  fim;
```

O que o código abaixo faz?

```
#include<stdio.h>

#include<stdlib.h>

main(int argc, char *argv[]){

    int i,n;

    printf("\nDigite um numero");

    scanf("%d",&n);

    i=n-1;

    loop:

    n += i;

    i--;

    if(i<=0)

    goto fim;
```

Fórmula Matamática

$$F(n) = \frac{1}{\sqrt{5}} \left\{ \left(\frac{1+\sqrt{5}}{2} \right)^n - \left(\frac{1-\sqrt{5}}{2} \right)^n \right\} = \frac{\phi^n}{\sqrt{5}} - \frac{(1-\phi)^n}{\sqrt{5}}$$

```
public class NewClass {

    public static void main(String args[]) {

        int x = 10, y = x + 1;

        int sequencia[] = new int[y];

        sequencia[0] = 0;

        sequencia[1] = 1;

        System.out.print("\nSequencia:\n" + sequencia[0] + ", "

                + sequencia[1] + ", ");

        for (int i = 2;  i < sequencia.length; i++) {

            sequencia[i] = sequencia[i - 1] + sequencia[i - 2];

            if (i == (sequencia.length - 1)) {

                System.out.print(" " + sequencia[i]);

            } else {
```

```
public class Fibonacci {

    public static int calcular(int n) {

        if (n == 0 || n == 1)

                return n;

        else

                return calcular(n - 1) + calcular(n - 2);

    }

}

public class Main {

    public static void main(String[] args) {

        Fibonacci f = new Fibonacci();

        for (int x = 1; x < 10; x++) {
```

Recursão Eficiente

```
{
    fibCalc(20, 0, 1);
}
int fibCalc(int n, int a, int b) { System.out.println(a);
    return (n == 0) ? a : fibCalc(--n, a + b, a);
}
```

Algoritmo Recursivo em Scheme

```
(define (fib n)
    (cond ((= n 0) 0)
          ((= n 1) 1)
          (else (+ (fib (- n 1)) (fib (- n 2))))))
```

Algoritmo Iterativo em Scheme

```
(define (fib n)
    (define (iter i p r)
        (if (= i n)
                r
                (iter (+ i 1) r (+ p r))))
    (cond ((= n 0) 0)
        ((= n 1) 1)
        (else (iter 2 1 1))))
```

Algoritmo em ECMAScript / JavaScript

```
function fibonacci(i) {
    return i < 2 ? i : fibonacci(i - 1) + fibonacci(i - 2);
} /* Chamando ... */
    for(i = 1; i < 10; i++) {
        alert(fibonacci(i));
}
```

Algoritmo recursivo em Python

```
def fibo(n):
    if n < 2:
        return n
    else:
        return fibo(n-1) + fibo(n-2)
    for i in range(10):
        print fibo(i)
```

Algoritmo eficiente em Python

```
def fib(n):
    c, n1, n2 = 0, 0, 1
    while c < n:
        yield n1
        n1, n2 = n2, n1 + n2
        c += 1
# Calcular os 10 primeiros termos
    for x in fib(10):
        print x
```

Algoritmo em PHP

```
<?php

/*Algoritmo Recursivo da Sequência Fibonacci */

function fibonacci($i) {

    return $i < 2 ? $i : fibonacci($i-1) + fibonacci($i-2);

} /* Chamando ... */

for($i=1;$i<10;$i++) {

    echo fibonacci($i)."\r\n"; }

?>
```

Algoritmo em PHP Bem Mais Rápido

```
<?php

function fibonacci ($number=10, $returnLast = false) {

        $start = array();

        $start[1] = 1;

        $start[2] = 1;

        if ( $number == 1 ) {

                unset($start[2]);

        }

        for ( $i = 3; $i <= $number; $i ++ ) {

                array_push($start, $start[$i - 2] + $start[$i - 1]);

        }

        return $returnLast === true ? end($start) : $start;

} /* Para utilizar ... */

echo "<pre>";
```

Algoritmo em Perl

```
# !/usr/bin/perl

use strict;

sub fibo {

    return $_[0] < 2 ? $_[0] : fibo($_[0]-1) + fibo($_[0]-2);

}

for (1..10){

    print fibo($_)."\n";

}
```

Algoritmo em Perl Mais Eficiente

```
# !/usr/bin/perl

use strict;

my ($a,$b)=(1,2);

print "$a\n$b\n";

for(1..100) {

      ($a,$b) = ($b,$a+$b);

      print "$b\n“

}
```

Algoritmo em C

```
#include <stdio.h>

#include <stdlib.h>

 long int fibo(int);

 int main(int argc, char** argv) {

        setbuf(stdout, NULL);

        int n, fib;

        printf("Digite N: ");

        scanf("%i", &n);

        fflush(stdin);

        fib = fibo(n);

        printf("Fibonacci =%i\n", fib);

        printf("\n");

        system("PAUSE");/* Tomem cuidado que isso faz o windows parar */
```

Algoritmo em C Outro Método

```
#include <stdio.h>

#include <stdlib.h>

int main() {

    int n_anterior1=1,n_anterior2=0,n=1;

        for(int i = 1;i<=15;i++){

        printf("%d ",n);

        n = n_anterior1 + n_anterior2;

        n_anterior2=n_anterior1;

        n_anterior1=n;

    }

    printf("\n");

        system("PAUSE");

        return 0;
```

Algoritmo em Ruby

```
# Algoritmo Recursivo da Sequência Fibonacci

def fib(n)

  return n if n<2

  fib(n-1)+fib(n-2)

end

# Chamando...

for i in (1..10)

  puts fib(i)

end

# Ou com memorização

fib = Hash.new{|hash, n| hash[n] = (n < 2) ? n : hash[n - 1] + hash[n - 2];};

(1..100).each do |i|

  puts fib[i]
```

Algoritmo em Erlang

```
fib(0) -> 0 ;

fib(1) -> 1 ;

fib(N) when N > 0 -> fib(N-1) + fib(N-2).

%% Tail recursive

fibo3(N) ->

    {Fib, _} = fibo3(N, {1, 1}, {0, 1}),

    Fib.

fibo3(0, _, Pair) -> Pair;

fibo3(N, {Fib1, Fib2}, Pair) when N rem 2 == 0 ->

    SquareFib1 = Fib1*Fib1,

    fibo3(N div 2, {2*Fib1*Fib2 - SquareFib1, SquareFib1 + Fib2*Fib2}, Pair);

fibo3(N, {FibA1, FibA2}=Pair, {FibB1, FibB2}) ->

    fibo3(N-1, Pair, {FibA1*FibB2 + FibB1*(FibA2 - FibA1), FibA1*FibB1 + FibA2*FibB2}).
```

Algoritmo em Shell Script

```
fibonacci() {
        local a c
        local -F1 b
        a=0 ; b=1
        print $a
        repeat $1
        do
                print "${b%.*}"
                c=$a
                a=$b
                ((b = c + b))
        done
}
```

Algoritmo em bc (comando Unix)

```
define void fibonacci(valor)
{
   auto x, y, z, i;
   x = 0;
   y = 1;
   x;
   while (i++ < valor) {
      y;
      z = x;
      x = y;
      y = z + y;
   }
}
```

Algoritmo em Pascal

```
program fibonacci (input,output);

var

        i,n,ni,ni1,ni2:longint;

begin

        writeln ('NÚME ROS DE FIBONACCI');

        write('Quantos termos de Fibonacci você quer calcular? ');

        read(n);

        ni:=1;

        ni1:=1;

        ni2:=0;

        for i:=1 to n do

                begin
```

Algoritmo em MATLAB

```
a=0;

b=1;

c=a+b;

N=0;

while N?0

    N=input('Defina limite da sequência fibonacci: ');

end

while c?N

    disp(num2str(c))

    a=b;

    b=c;

    c=a+b;
```

Algoritmo em Prompt Script (BAT)

```
@echo off

setlocal ENABLEDELAYEDEXPANSION

set/an=-1

set/af0=0

set/af1=1

:loop

set/an+=1

set/am=%n%-1

set/al=%m%-1

set /a f%n%=!f%m%!+!f%l%!

echo F(%n%)=!f%n%!

pause&goto loop
```

Algoritmo em PROLOG

```
fib(1, 1).

fib(2, 1).

fib(X, Y):- X > 1, X1 is X - 1, X2 is X - 2, fib(X1, Z), fib(X2, W), Y is W +
    Z.
```

Algoritmo em Tcl

```
proc fib {n} {
    if {$n < "2"} {
        return $n
    }
    return [expr [fib [expr $n - 1]] + [fib [expr $n - 2]]]
}
# Chamando
fib 10
```

Português estruturado

```
algoritmo "série de fibonnacci"

 var fibo:inteiro

 n0,n1,n2:inteiro

 I:inteiro

 soma:inteiro

 aux: caracter

 inicio

 n0:=0

 enquanto n0<2 faca

 escreva("quantos num.? [ n > 1 ]:")

 leia(n0)

 se n0<2 entao

 escreval(" digite novamente [pressione enter]")

 leia(aux)

 limpatela

 fimse

 fimenquanto

 escreva("1° numero da seq.: ")

 leia(n1)

 escreva("2° numero da seq.: ")

 leia(n2)

 escreva(" ",n1," ",n2," ")

 soma:=n1+n2
```

Algoritmo em COBOL

```
*-----------------

 300-FIBONACCI SECTION.

*-----------------

    IF 999E-REG EQUAL 0

       MOVE PNULT TO FIB

    ELSE

       IF  999E-REG EQUAL 1

          MOVE ULT TO FIB

       ELSE

          MOVE 999E-REG TO GDA-POSICAO

          PERFORM 400-CALCULA UNTIL GDA-POSICAO EQUAL 1.

*-----------------------------------------------*

*

*----------------

 400-CALCULA SECTION.
```

Algoritmo em Visual Fox Pro

```
? Fibonacci(22)

FUNCTION Fibonacci (tnNumeros)

	LOCAL lnNumero, lnNumero1, lnNumero2, lnI, lcRetorno

	lnI = 1

	lcRetorno = ""

	lnNumero = 1

	lnNumero1 = 1

	lnNumero2 = 0

	FOR lnI = 1 TO tnNumeros STEP 1

		lcRetorno = lcRetorno + " " + TRANSFORM(lnNumero)

		lnNumero = lnNumero1 + lnNumero2

		lnNumero2 = lnNumero1

		lnNumero1 = lnNumero

	ENDFOR
```

Algoritmo em C#

```
public String Fibonacci(int numeros)
{
    String retorno = "";
    int numero = 1;
    int numero1 = 1;
    int numero2 = 0;
    for(int i=0; i < numeros; i++)
    {
        retorno = retorno + " " + numero.ToString();
        numero = numero1 + numero2;
        numero2 = numero1;
        numero1 = numero;
    }
```

Algoritmo em C# (modo com caixa de texto)

```
public void fibonacci(object s, EventArgs e){

    int x;

     try{

        x = int.Parse(xfb.Text);

        int r = 0, n1 = 1, n2 = 1, nn = 0;

        for (int i = 0; i < x; i++){

            r = n1;

            n1 = n2 + nn;

            nn = n2;

            n2 = n1;

        }

        rfb.Text = r.ToString();

        if (x > 46){
```

Algoritmo em C++

```
#include <iostream>

using namespace std;

int Fibonacci(int);

int main(){

    int quantidade;

    cout << "Deseja imprimir quantos numeros?";

    cin >> quantidade;

    for(int x = 1; x < quantidade; x++)

        cout << "O " << x << "# numero de Fibonacci é: " << Fibonacci(x) << "\n";

}

int Fibonacci(int Number){

    if(number == 0 || number == 1)

        return number;

    else
```

Algoritmo em Fortran 2003

```
PROGRAM FIBONACCI

IMPLICIT NONE

INTEGER, DIMENSION (1:1000) :: FIB

INTEGER :: I,N

WRITE(*,*) "SEQUENCIA DE FIBONACCI COM N ELEMENTOS. INFORME N"

READ (*,*) N

FIB(1) = 1

FIB(2) = 1

WRITE(*,*) !pular linha

WRITE(*,*) FIB(1)

WRITE(*,*) FIB(2)

DO I = 3,N

   FIB(I) = FIB(I-1) + FIB(I-2)

   WRITE(*,*) FIB(I)
```

Algoritmo em Haskell

```
fib n
    |n==0 = 0
    |n==1 = 1
    |otherwise = fib(n-1) + fib(n-2)
```

Algoritmo em Haskell

```
fibs n = fibGen 0 1 n

fibGen a b n = case n of

    0 -> a

    n -> fibGen b (a + b) (n - 1)
```

Tarefa

- Como tarefa, defina qual o paradigma que cada linguagem desses slides usa por padrão.
- Tente fazer ou buscar algoritmos em cada linguagem mostrada nos programas de Fibonacci para o problema fatorial e números primos.
- Que implementação você indicaria para cada problema? Qual o paradigma é melhor para cada problema? Por que?