Solução de Problemas usando Pensamento Computacional

Programação Imperativa
DCOMP-UFS

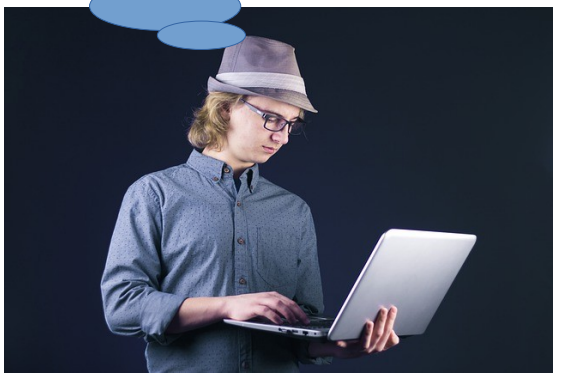
- Matemáticos, engenheiros, …, e cientistas da computação resolvem problemas
  - No entanto, raciocinam diferentemente

- Características próprias dos problemas em computação
  - Solução é expressa através de um algoritmo
  - Natureza dos produtos desenvolvidos
    - Em outras áreas se produz entidades concretas (prédios, pontes, aeronaves …)
    - Em computação produzimos ***software*** (entidades **abstratas**)

Expressar sua solução

Formular problema

Pensamento Computacional



Algoritmo/
Programa

Elementos do PC

- Decomposição
- Reconhecimento de padrões
- Abstração
  - Identificar similaridades
  - Esconder detalhes não relevantes
- Projeto de algoritmos

Abstração

- Processo crítico do pensamento computacional
- Objetivo: reduzir complexidade
- Processo de criar (definir) conceitos gerais ou ideias
  - Encontrar características gerais
  - Esconder detalhes não importantes
- Podemos abstrair
  - Dados
  - Expressões
  - Comandos
  - ...

```
if a > b:
    m = a
else:
    m = b
```

Podemos capturar este código dentro de uma abstração mediante uma definição de função

```
def maior(p, q):
    if p > q:
        return p
    else:
        return q
…
m = maior(a, b)
```

```
if 19 <= hora and  hora < 24:
   ...
   if tarifa1 > tarifa2:
      tarifa = tarifa1
   else:
      tarifa = tarifa2
   valorAPagar = tarifa * consumo + taxa
   ...
```

```
if 19 <= hora and  hora < 24:
   ...
   valorAPagar =
     maior(tarifa1, tarifa2)*consumo + taxa
   ...
```

```
consumo = consumoDia + consumoNoite
if consumoDia > consumoNoite
   consumoMaior = consumoDia
else:
   consumoMaior = consumoNoite
acrescimo = 0.01 * consumoMaior
if 19 <= hora and  hora < 24:
  if tarifa1 > tarifa2:
     tarifa = tarifa1
  else:
     tarifa = tarifa2
  valorAPagar = tarifa * consumo + taxas + acrescimo
```

```
consumo = consumoDia + consumoNoite
acrescimo = 0.01 * maior(consumoDia, consumoNoite)
if 19 <= hora and  hora < 24:
  valorAPagar = maior(tarifa1, tarifa2) * consumo +
                taxas + acrescimo
```

- A função maior é uma abstração de expressão
- Funções também nós permitem definir abstrações de comandos

```
def imprimeQuadrado():
    print(' *** ')
    print(' *** ')
    print(' *** ')

def imprimeTriangulo():
    print('  *  ')
    print(' *** ')
    print('*****')

def imprimeSetaAcima():
    imprimeTriangulo()
    imprimeQuadrado()
```

Outras abstrações

- Funções permitem abstrair
  - Expressões
  - Comandos
- Abstração de dados
  - Tipos abstratos de dados
  - Classes e objetos
- Interfaces de acesso
  - Interface de programação para acesso a uma aplicação (API)
- ...

Projetando Soluções

Entendendo o problema

- Entenda o problema
- **Projete** um solução
  - Decomponha em subproblemas
    - Resolva o problema passo a passo
  - Componha a solução
    - Use padrões que você conhece
    - Descubra padrões – use exemplos
- Implemente
- Verifique a solução
- Analise a solução – solidifique o conhecimento
  - Há outras formas de resolver?
  - Podemos melhorar?

Por onde começo?

- Eu entendo o problema?
  - Descrições incompletas, ambíguas e/ou inconsistentes

"Dados três números, devolva o valor do meio"

  - Média aritmética?
  - Se listados em ordem, aquele que fica no meio?
    - Para 2, 8 e 5 o resultado é 5
    - Para 2, 8 e 2?

- Ainda para problemas simples podem existir detalhes a serem pensados
- Não há resposta correta – programador e usuário devem definir
- Exemplos são facilitadores
- Quais são os casos limites ou especiais?
- Importante descobrir falhas na especificação o mais cedo possível

Outro exemplos

- Dados três números, calcule a soma daqueles que são pares
  - Se somente um for par?
  - Se nenhum for par?

- Quantos dias há entre duas datas
  - Incluímos início e fim?
  - Incluímos só ínício (só fim)?
  - Não incluímos nem início nem fim?

- Para os problemas do The Huxley
  - Quais são os dados de entrada?
  - Quais são as saídas?
  - Quais são os formatos da entrada e da saída?

Projetando com exemplos

- Implementar - Codificar
  - Escrever um programa
- Projetar
  - Planejamento prévio à implementação
  - Arquitetar uma solução
  - Tudo o que fazemos antes de implementar

- Primeiro projetar para depois implementar
- A menos que o problema seja muito simples (tenhamos a solução na cabeça)
  - Exemplo: calcular o maior entre a e b

```
if a > b:
    m = a
else:
    m = b
```

```
def maior(p, q):
    if p > q:
        return p
    else:
        return q
…
m = maior(a, b)
```

Projetando uma solução

Exercício pedagógico: Calcular o maior de três números

```
def maior3(m, n, p):
    if m > n: # descartamos n
        if m > p:
            return m
        else:
            return p
    else:     # descartamos m
        if n > p:
            return n
        else:
            return p
```

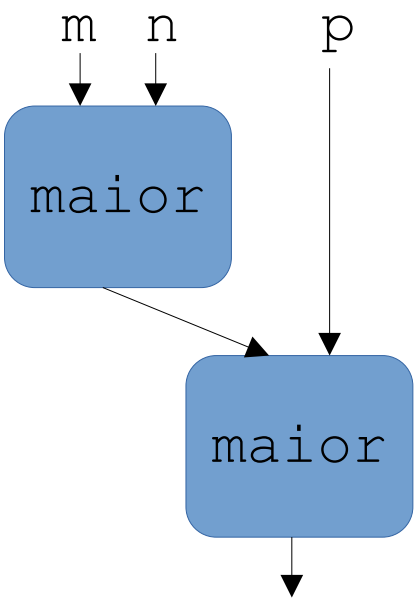
- Lições do exemplo anterior:
  - Está perdido na busca de uma solução?
    - Comece trabalhando com exemplos
  - Exemplos (instâncias do problema) são importantes facilitadores

Composição

Reuse recursos

- Que recursos são disponíveis?
  - pré-definidos, bibliotecas, outros problemas resolvidos, …
  - Posso usá-los “dentro da minha solução”?

```
def maior3(m, n, p):
    if m > n:
        return maior(m, p)
    else:
        return maior(n, p)
```




```
def maior3(m, n, p):
    mai = maior(m, n)
    return maior(mai, p)
```

```
def maior3(m, n, p):
    return maior(maior(m,n), p)
```

Reuse modelos de soluções

- Identificação de padrões
  - Já resolvi algum problema similar?
  - Posso usá-lo como modelo?

Reusando um modelo

```
def maior(m, n):
    if m > n:
        return m
    else:
        return n
```

```
def maior3(m, n, p):
    if m é o maior dos três:
        return m
    elif n é o maior dos três:
        return n
    else:
        return p
```

```
def maior3(m, n, p):
    if m >= n and m >= p:
        return m
    elif n >= m and n >= p:
        return n
    else:
        return p
```

Pseudo-código permite abstrair detalhes

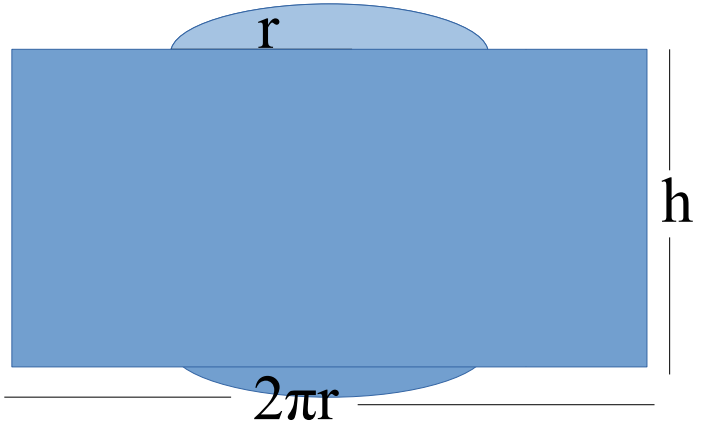
Refinando as abstrações

Decomposição

Decomponha o problema

- Decomponha em subproblemas mais simples (dividir para conquistar)
  - Útil para resolver problemas complexos
  - Idealmente, cada subproblema resolve algum aspecto separado do problema
- Componha os subproblemas para formar a solução
  - Abstraia os subproblemas
    - Não se preocupe com detalhes
    - **Suponha que tem uma função** que precisa
    - Permite desenvolver a solução passo a passo

Problema: calcular a área do cilindro tendo como dados o raio da base e a altura do cilindro



Podemos decompor em subproblemas

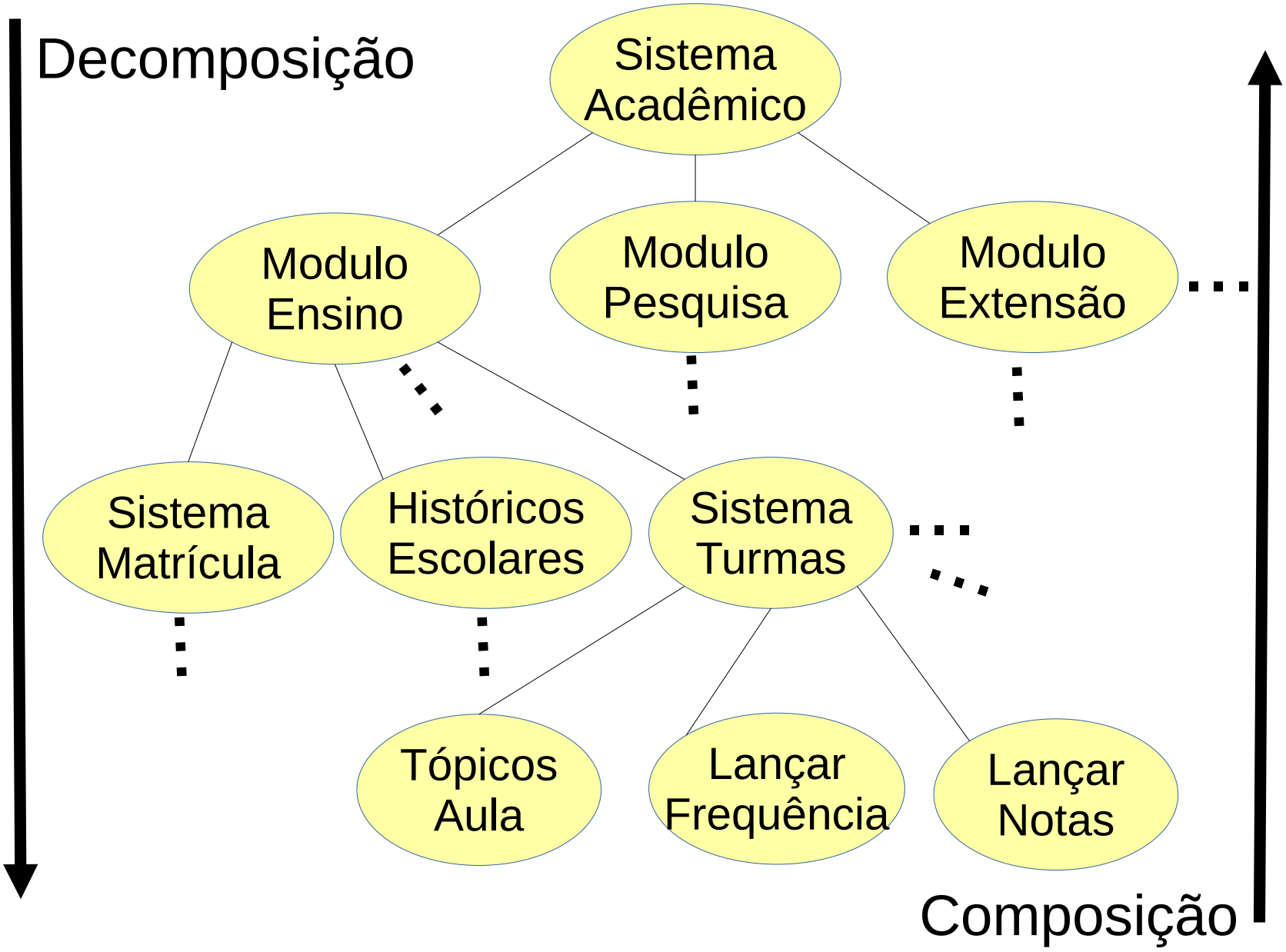
- Calcular a área da base
- Calcular a área do contorno lateral

```
def areaLateralC(r, h):
    return 2 * pi * r * h

def areaCirculo(r):
  return pi * r**2

def areaCilindro(r, h):
  return 2*areaCirculo(r) +
         areaContornoL(r, h)
```

Abstraindo os subproblemas a solução é

2 * areaCirculo + areaContornoLateral

Raciocínios *Top-down* e *Bottom-up*



Verificar a solução

A solução é correta?

- Testes
- Provas

- Não é correta?
  - Depuração
  - ...

Testes

- Caixa Preta vs Caixa Branca
- Como escolho os casos de teste?
  - Escolha casos “comuns”
  - Preste atenção a casos limites
  - Caixa Branca:
    - Abranja todos os caminhos do código

```
def maior3(m, n, p):
    if m > n and m > p:
        return m
    elif n > m and n > p:
        return n
    else:
        return p
```

- Para exercícios do The Huxley, a tarefa de testes é feita automaticamente
  - Mesmo assim, para descobrir erros, podem ser úteis testes manuais de funções.
- No trabalho prático precisão realizar testes

  Existem ferramentas automáticas e/ou semiautomáticas

Analisar a solução

Reflita a solução

- A solução pode ser melhorada?
- Há outras maneiras?

- Assimile conhecimento
  - Descobri algum padrão novo?

Podemos melhorar?

```
def maior3(m, n, p):
    if m >= n and m >= p:
        return m
    elif n >= m and n >= p:
        return n
    else:
        return p
```

Se a primeira condição é falsa, podemos descartar `m` como possível solução

```
def maior3(m, n, p):
    if m >= n and m >= p:
        return m
    elif n >= p:
        return n
    else:
        return p
```

```
def maior3(m, n, p):
    if m >= n and m >= p:
        return m
    else:
        return maior(n,p)
```

Há outras maneiras de resolver?

```
def maior3(m, n, p):
    mai = m
    if n > mai:
        mai = n
    if p > mai:
        mai = p
    return mai
```

Oba! … um novo padrão que talvez possa reusar em outros problemas

Resumo

- Entenda o problema
- Use estratégias
  - Que recursos tenho?
  - Já resolvi algo similar?
    - Posso reusar?
    - Posso usar como modelo?
  - Analise como faria com alguns exemplos
    - Há algum padrão?
- O problema é complexo?
  - Decomponha em subproblemas
- Valide a solução
- Analise a solução para solidificar o aprendido