Funções Recursivas

Programação de Computadores

O que veremos hoje

- Funções recursivas
  - Definição
    - Caso de base
    - Caso geral
  - Implementação em Ruby
  - Exemplos
- Problemas de implementação
  - Estouro da pilha
- Como usar recursão sem Estouro da Pilha
  - Memorização
  - TCO: Recursão em Cauda
- Conversão de função recursiva em iterativa
- Conversão de função iterativa em recursiva
- Exercícios em sala

Recursão

- Recurso que permite definir algo em função dele mesmo
- **Função recursiva**
  - função definida usando a própria função
- Este conceito vocês já viram na Matemática

```
fatorial(n) =
          ⎧ 1                    , se n<2
          ⎩ n * fatorial(n-1), se n>=2
```

Exemplo

Função Fatorial

```
def fat(n)
  if n < 2 then
    1
  else
    n * fat(n-1)
  end
end
```

Caso base

Condição de parada
da recursão

Caso Geral

Exemplo

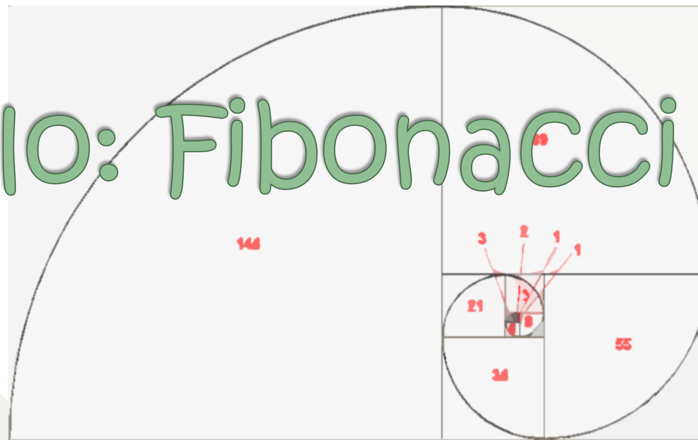
Função Fatorial

```
def fat(n)
  if n < 2 then
    1
  else
    n * fat(n-1)
  end
end
```

- Calculando o fatorial

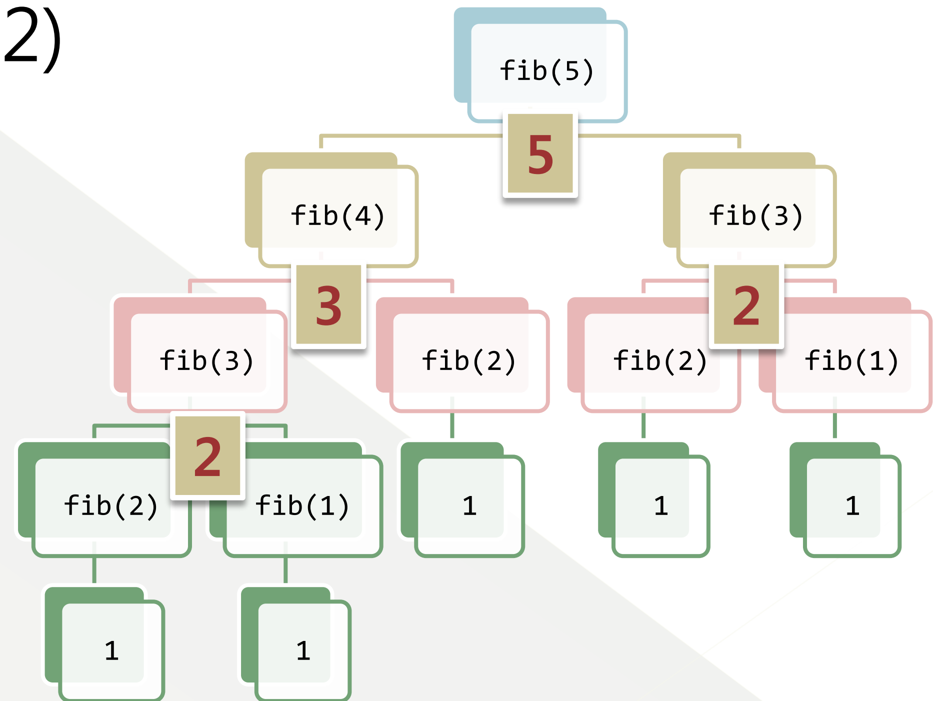
```
fat(4)
  = 4 * fat(3)
  = 4 * (3 * fat(2))
  = 4 * (3 * (2 * fat(1)))
  = 4 * (3 * (2 * 1))
  = 4 * (3 * 2)
  = 4 * 6
  = 24
```

Exemplo: Fibonacci



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

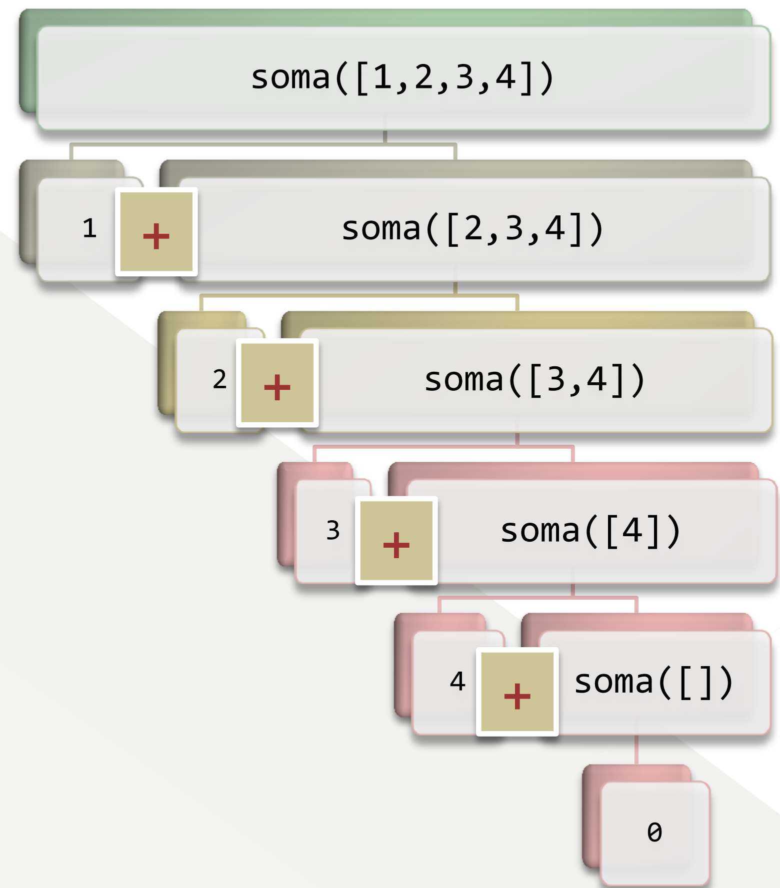
```
def fib(n)
  if (n<=2) then
    1
  else
    fib(n-1) + fib(n-2)
  end
end
```



Exemplo com Arrays

- Soma dos elementos de um array

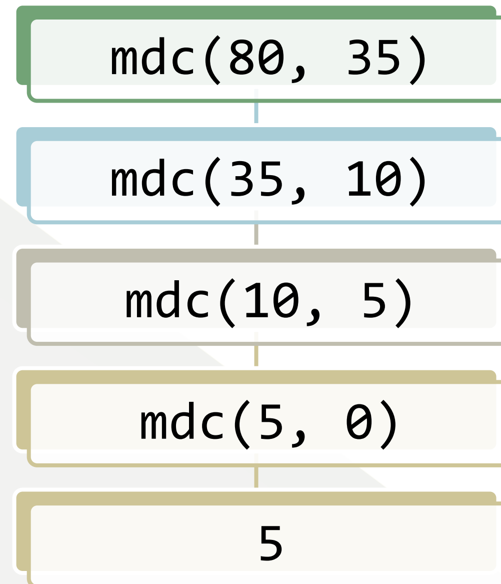
```
def soma (a)
  if (a.size > 0) then
    primeiro = a.first
    resto = a.drop(1)
    primeiro + soma(resto)
  else
    0
  end
end
x = [1,2,3,4]
puts soma(x)
```



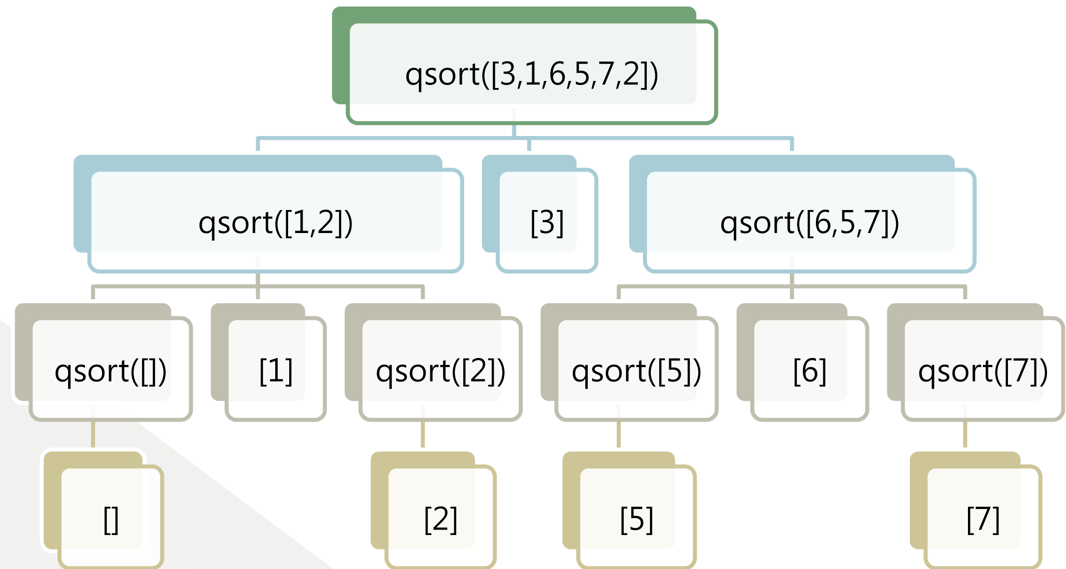
Exemplo MDC

- Calcule o MDC de dois números

```
def mdc(a, b)
  if (b == 0) then
    a
  else
    mdc(b % a, a)
  end
end
puts mdc(80, 35)
```



Exemplo QuickSort




```
def qsort(a)
  if (a.size <= 1) then
    a
  else
    pivot = a.first
    r = a.drop(1)
    menor = r.select{|n| n <= pivot}
    maior = r.select{|n| n >  pivot}
    qsort(menor) + [pivot] + qsort(maior)
  end
end
puts qsort([3,1,6,5,7,2])
```

Avaliação sobre o uso de Funções Recursivas

- As funções recursivas, geralmente, tornam o programa mais legível
  - As funções são definidas como na Matemática
  - Evita a reatribuição de valores a variáveis

- As funções recursivas podem substituir trechos de código que envolvem repetição
  - While
  - For

Convertendo while em funções recursivas

```
a = 80
b = 35
while (a > 0)
  a, b = b % a, a
end
puts b
```

```
def mdc(a, b)
  if (a > 0) then
    mdc(b % a, b)
  else
    b
  end
end

puts mdc(80,35)
```

Convertendo for em funções recursivas

```
def fat(n)
  f = 1
  for i in 2 .. n
    f = f * i
  end
  return f
end
```

```
def fat(n)
  if n < 2 then
    1
  else
    n * fat(n-1)
  end
end
```

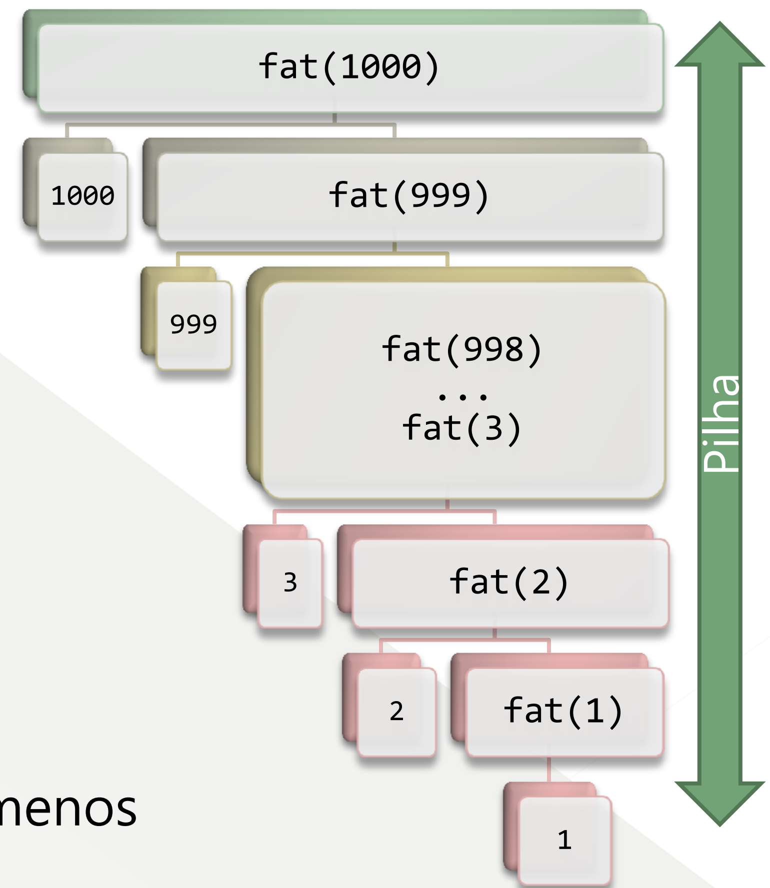
Problemas com Funções Recursivas

- A implementação de funções recursivas em Ruby é eficiente para muitas situações
- Mas ... Há problemas
  - Estouro de Pilha
  - Re-calculos desnecessários

Problemas com Funções Recursivas

- Estouro de Pilha
  - A partir de fat(8171) dá erro!

```
irb(main):048:0> fat 8171
SystemStackError: stack level too deep
        from
/Ruby193/lib/ruby/1.9.1/irb/workspace.rb:80
Maybe IRB bug!
irb(main):049:0>
```



- Re-calculos desnecessários
  - Para calcular fib(20)
    - É preciso calcular fib(2) pelo menos 5000 vezes!

É possível tornar eficientes as funções recursivas

- Técnicas mais usadas
  - Memorização
    - Guardar os valores já calculados
  - TCO (Otimização para recursão em cauda)
    - Permitir que Ruby otimize o código para chamar a função sem coloca-la na pilha
    - Internamente Ruby transforma a recursão em um laço.

  - Conversão manual da função em um while

Memorização

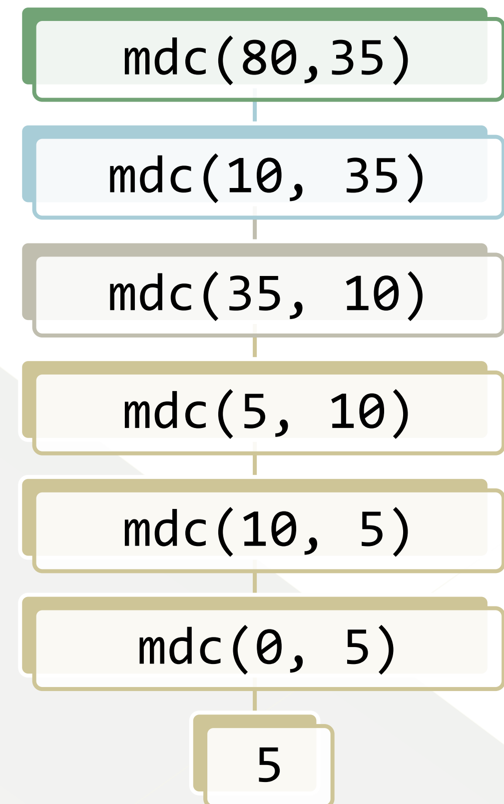
- Guarda em uma variável global os valores já calculados

```
$fibm = [0,1,1]
def fib(n)
  # Calcula fib(n) se ele ainda
  # não foi calculado
  $fibm[n] = $fibm[n] || (fib(n-1) + fib(n-2))
  return $fibm[n]
end
```

TCO: Recursão em Cauda

- Este tipo de recursão Ruby consegue otimizar sem precisar chamar a função novamente

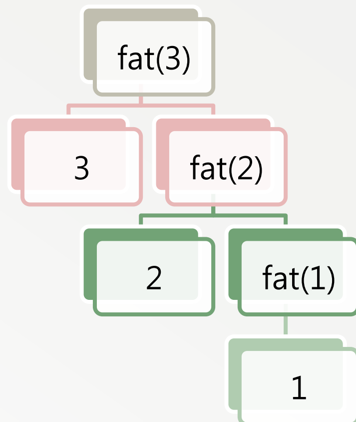
```
def mdc(a, b)
  if (a == 0) then
    b
  elsif (a > b) then
    mdc(a % b, b)
  else
    mdc(b, a)
  end
end
```



Transformando uma função para Recursão em Cauda

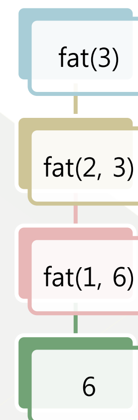
- Sem recursão em cauda

```
def fat(n)
  if n < 2 then
    1
  else
    n * fat(n-1)
  end
end
```



- Com recursão em cauda

```
def fat(n, total = 1)
  if n < 2 then
    total
  else
    fat(n-1, total*n)
  end
end
```



TCO

- A Otimização da Recursão em Cauda normalmente está desabilitada em Ruby
  - É necessário habilitar

```
RubyVM::InstructionSequence.compile_option = {
  :tailcall_optimization => true,
  :trace_instruction => false
}
RubyVM::InstructionSequence.new(<<-EOF).eval
def fib(n, f1=1, f2=0)
  if n<2 then
    f1
  else
    fib(n-1, f1+f2, f1)
  end
end
EOF
puts fib(10000)
```

Em último caso: Tire a Recursão

- Faça a função funcionar corretamente usando recursão
- Depois transforme a recursão em um while

```
def mdc(a, b)
  if (b > 0) then
    mdc(b % a, a)
  else
    a
  end
end
```

```
def mdc(a,b)
  while (b > 0)
    a, b = b % a, a
  end
  a
end
```

Entendeu tudo?

- Sim?
  - OK, pode ir para casa.
    - Condição de parada
- Não?
  - Então volte para

O que veremos hoje

- Funções recursivas
  - Definição
    - Caso de base
    - Caso geral
  - Implementação em Ruby
  - Exemplos
- Problemas de implementação
  - Estouro da pilha
- Como usar recursão sem Estouro da Pilha
  - Memorização
  - TCO: Recursão em Cauda
- Conversão de função recursiva em iterativa
- Conversão de função iterativa em recursiva
- Exercícios em sala