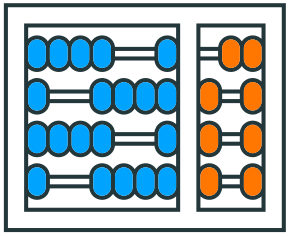


Instituto de Computação

UNIVERSIDADE ESTADUAL DE CAMPINAS

UNICAMP

MC102 – Aula 12
Recursão

Algoritmos e Programação de Computadores

Zanoni Dias

2020

Instituto de Computação

Roteiro

Recursão

Gerenciamento de Memória

Recursão $\times$ Iteração

Números de Fibonacci

Exemplos Simples

Exemplos com Listas e Strings

Torre de Hanói

Exercícios

Recursão

Recursão

Recursão

- Suponha que desejamos criar um algoritmo para resolver um determinado problema.
- Usando o método de recursão/indução, a solução de um problema pode ser expressa da seguinte forma:
  - Primeiramente, definimos a solução para casos base.
  - Em seguida, definimos como resolver o problema para um caso geral, utilizando soluções para instâncias menores do problema.

Indução

- Indução é uma técnica de demonstração matemática em que algum parâmetro da proposição a ser demonstrada envolve números naturais.
- Seja T uma proposição que desejamos provar como verdadeira para todos valores naturais n.
- Ao invés de provar diretamente que T é válido para todos os valores de n, basta provar os itens 1 e 3, listados a seguir:
  1) *Caso Base: Provar que T é válido para* $n = 1$.
  2) *Hipótese de Indução: Assumimos que T é válido para* $n - 1$.
  3) *Passo de Indução: Assumindo que T é válido para* $n - 1$, *devemos provar que T é válido para n.*

Indução

- Por que a indução funciona? Por que as duas condições são suficientes?
  - Mostramos que T é válido para um caso simples como $n = 1$.
  - Com o passo da indução, mostramos que T é válido para $n = 2$.
  - Como T é válido para $n = 2$, pelo passo de indução, T também é válido para $n = 3$ e assim por diante.

Exemplo

Teorema

A soma $S(n)$ dos primeiros n números naturais é $S(n) = n(n+1)/2$.

Prova

- *Base: Para $n = 1$, temos que $S(1) = n(n+1)/2 = 1(1+1)/2 = 1$.*
- *Hipótese de Indução: Vamos assumir que a fórmula é válida para $n-1$, ou seja, $S(n-1) = (n-1)n/2$.*
- *Passo: Devemos mostrar que é válido para n. Por definição, $S(n) = S(n-1) + n$. Por hipótese, $S(n-1) = (n-1)n/2$, logo:*

$$\begin{aligned} S(n) &= S(n-1) + n \\ &= (n-1)n/2 + n \\ &= (n^2 - n)/2 + n \\ &= (n^2 + n)/2 \\ &= n(n+1)/2 \end{aligned}$$

Recursão

- Definições recursivas de funções operam como o princípio matemático da indução visto anteriormente, ou seja, a solução é inicialmente definida para o(s) caso(s) base e estendida para o caso geral.

Fatorial

- Problema: calcular o fatorial de um número inteiro não negativo (n).
- Qual é o caso base?
  - $0! = 1$
- Qual é o passo indutivo?
  - $n! = n \times (n - 1)!$
- Este problema é trivial pois a própria definição de fatorial é recursiva.

Fatorial

- Portanto, a solução do problema pode ser expressa da seguinte forma:
  - Se $n = 0$ então $0! = 1$.
  - Se $n \geq 1$ então $n! = n \times (n-1)!$.
- Note como aplicamos o princípio da indução:
  - Sabemos a solução para um caso base ($n = 0$).
  - Definimos a solução do problema geral em termos do mesmo problema, mas para um caso mais simples.

Fatorial

```
1 def fatorial(n):
2   if n == 0:
3     return 1
4   else:
5     aux = fatorial(n - 1)
6     return n * aux
```

Fatorial

- Para solucionar o problema, fizemos uma chamada para a própria função, por isso esta função é chamada de recursiva.
- Recursividade geralmente permite uma descrição mais clara e concisa dos algoritmos, especialmente quando o problema é recursivo por natureza.

Gerenciamento de Memória

Gerenciamento de Memória

- Devemos entender como é feito o controle sobre as variáveis locais em chamadas recursivas.
- A memória de um sistema computacional é dividida em três partes:
  - *Espaço Estático*: contém o código do programa.
  - *Heap:* para alocação dinâmica de memória.
  - *Pilha:* para execução de funções.

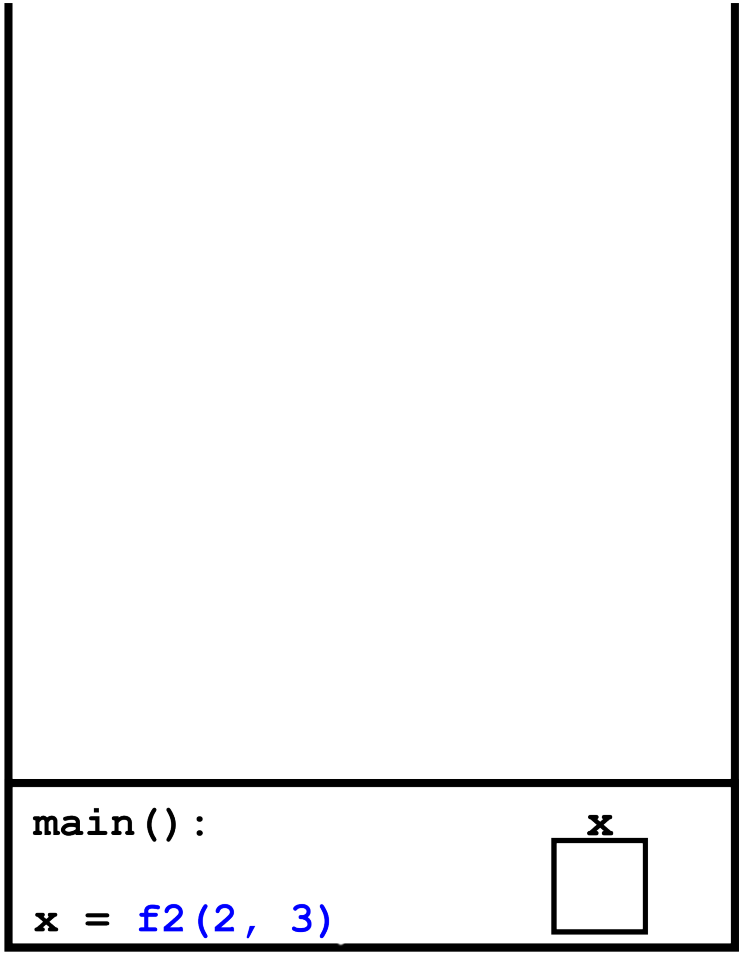
Gerenciamento de Memória

- Toda vez que uma função é invocada, suas variáveis locais são armazenadas no topo da pilha.
- Quando uma função termina a sua execução, suas variáveis locais são removidas da pilha.

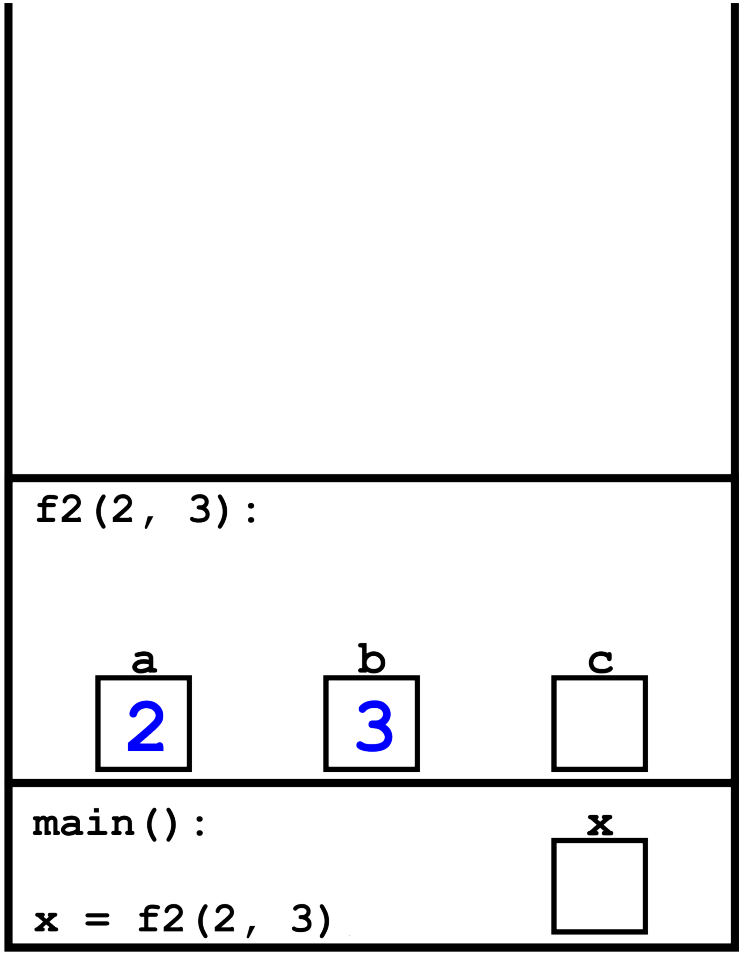
Gerenciamento de Memória

```
1  def f1(a, b):
2    c = a - b
3    return (a + b + c)
4
5  def f2(a, b):
6    c = f1(b, a)
7    return (b + c - a)
8
9  def main():
10   x = f2(2, 3)
11   return 0
12
13 main()
```

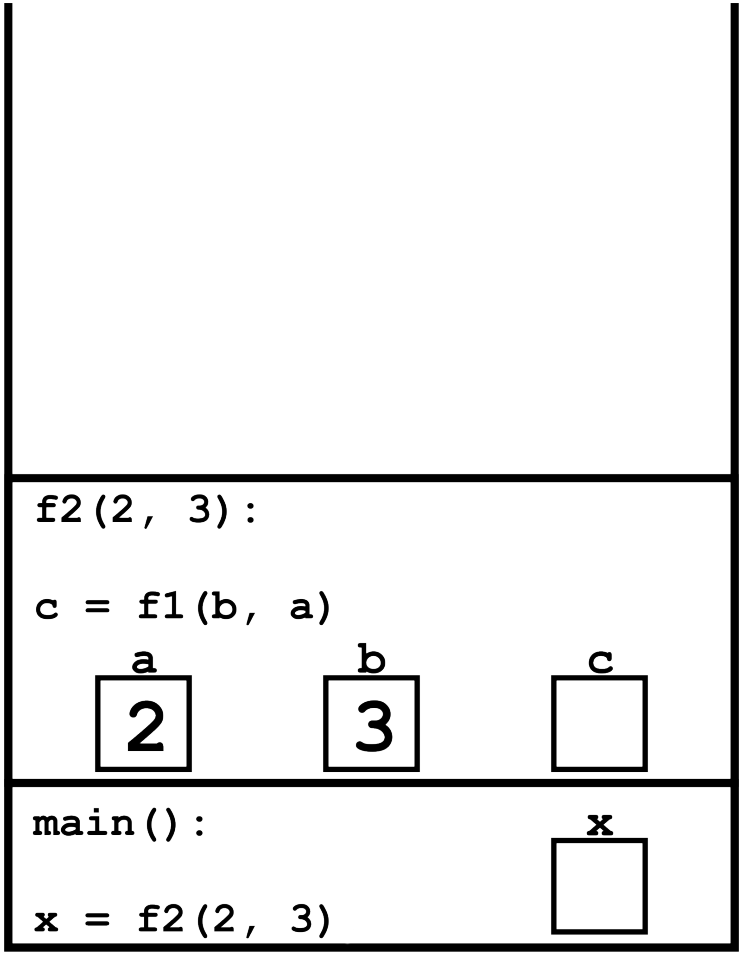
Gerenciamento de Memória



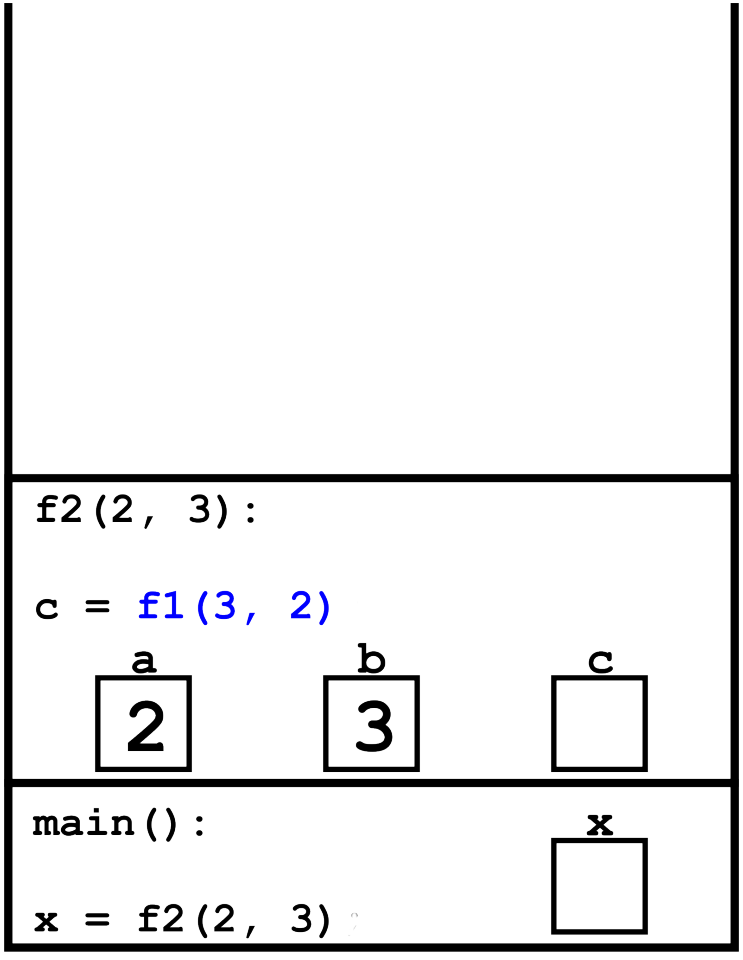
Gerenciamento de Memória



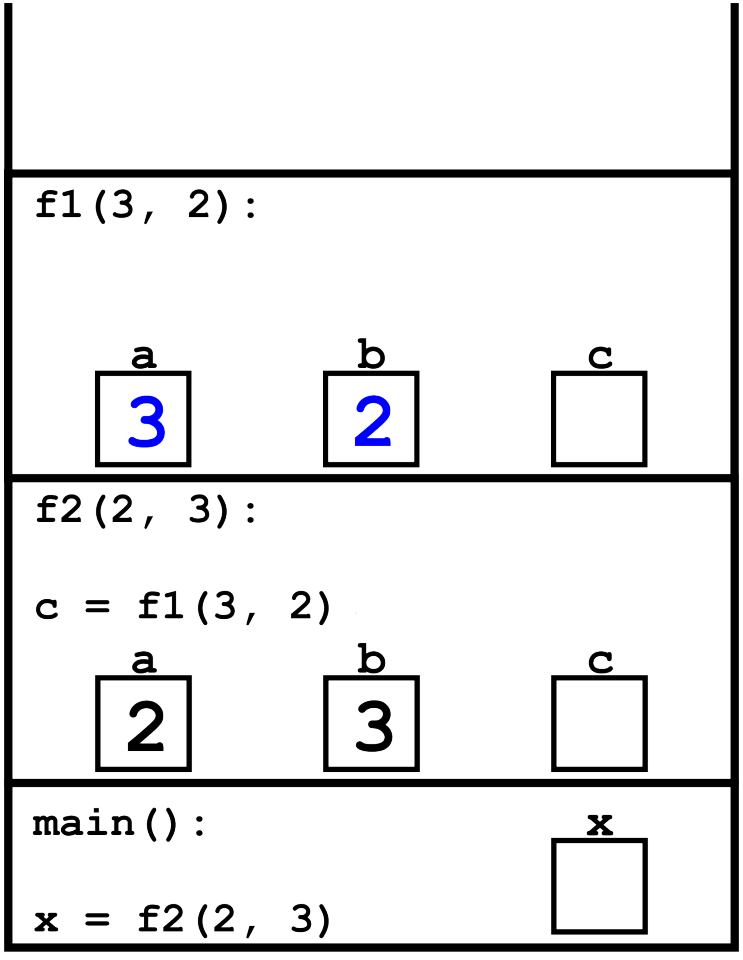
Gerenciamento de Memória



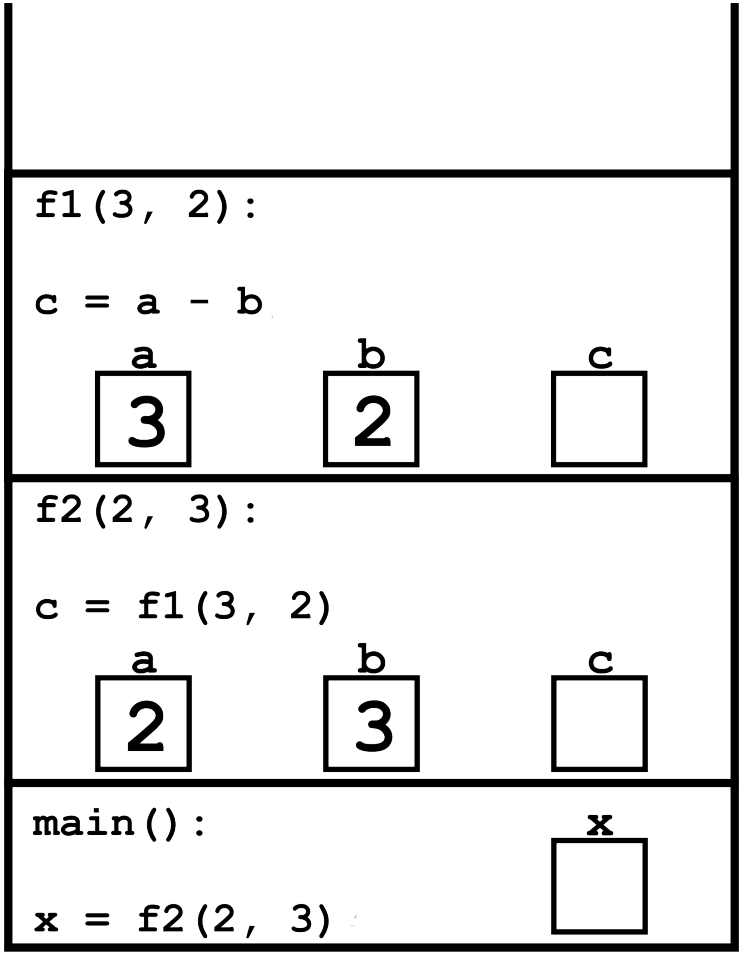
Gerenciamento de Memória



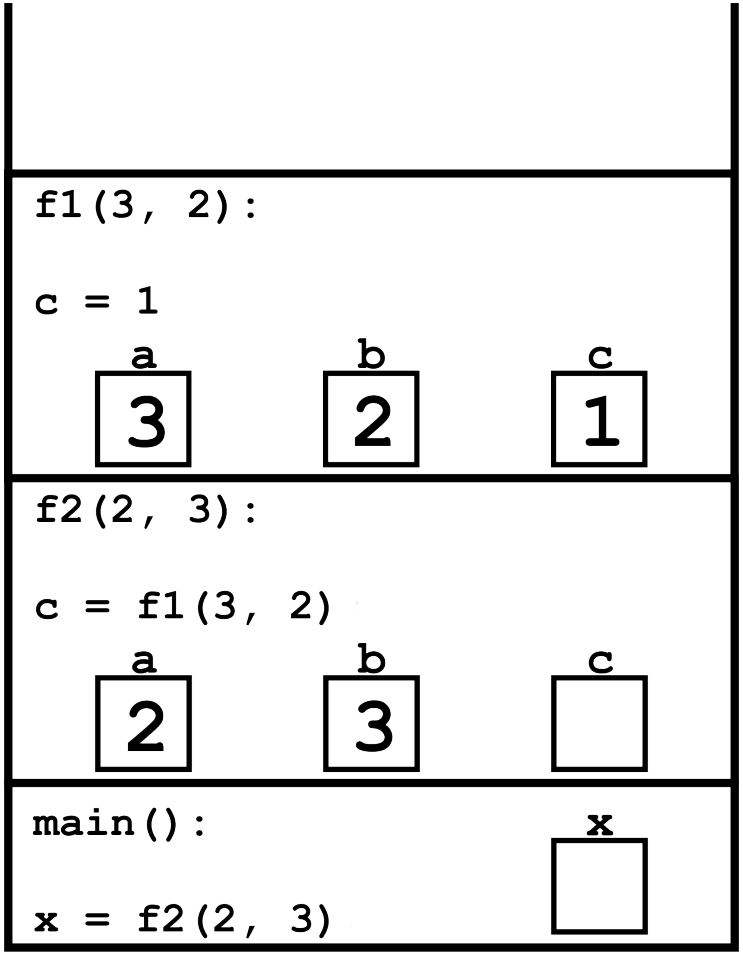
Gerenciamento de Memória



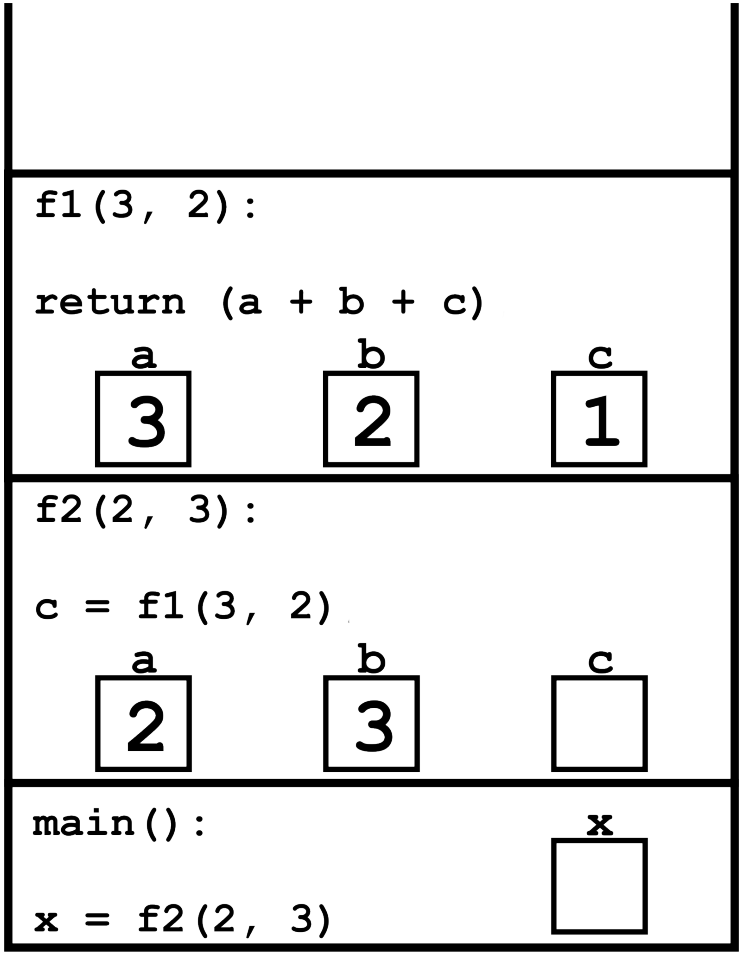
Gerenciamento de Memória



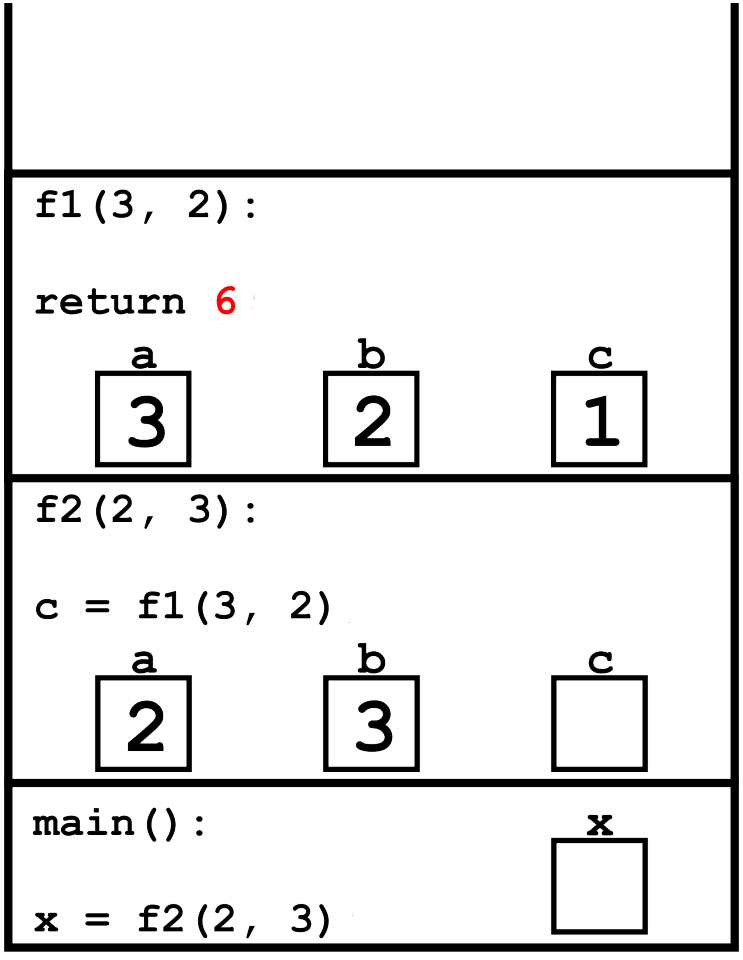
Gerenciamento de Memória



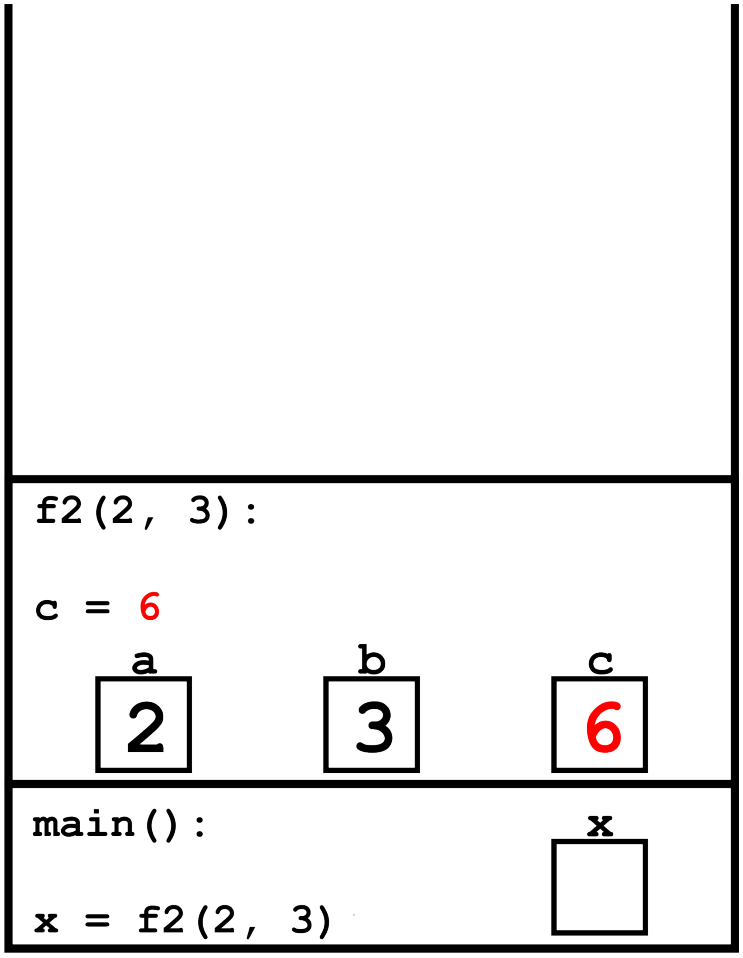
Gerenciamento de Memória



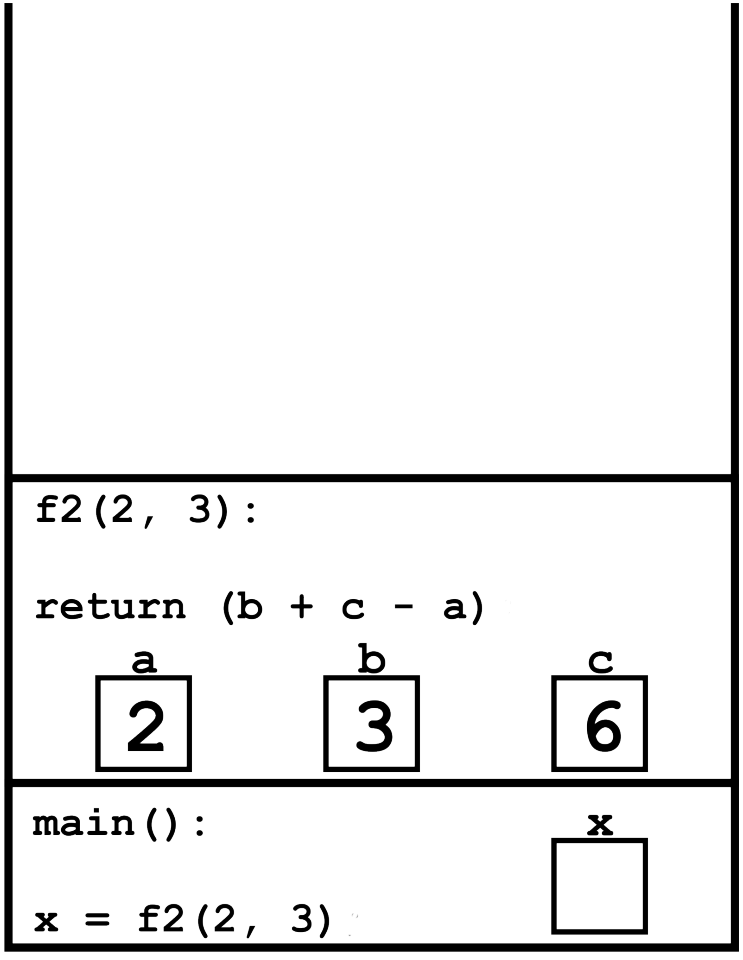
Gerenciamento de Memória



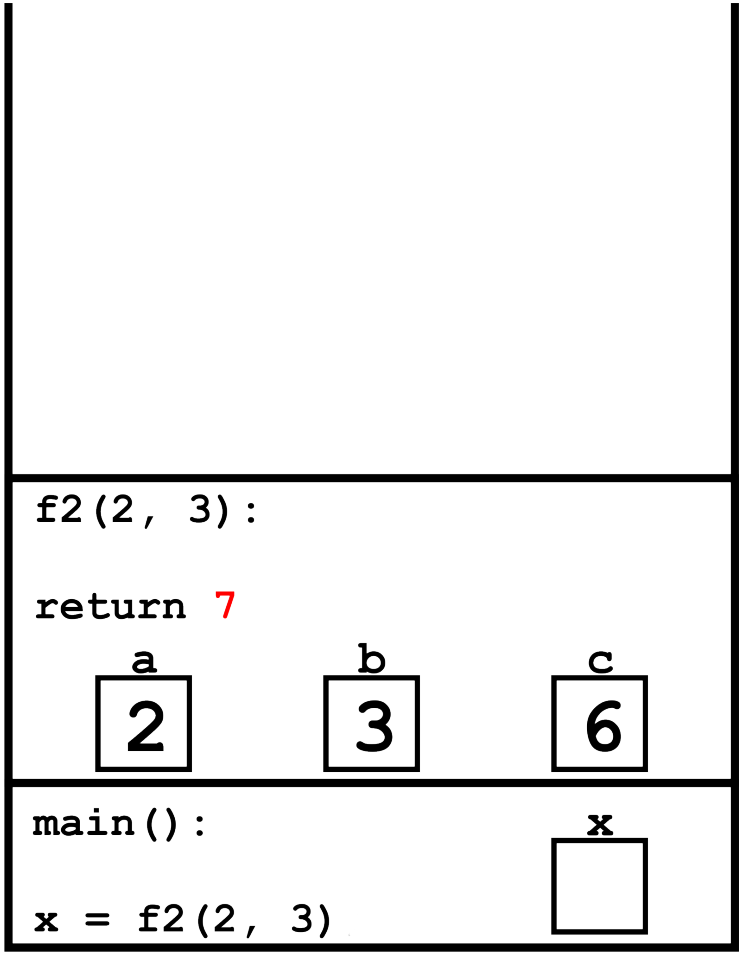
Gerenciamento de Memória



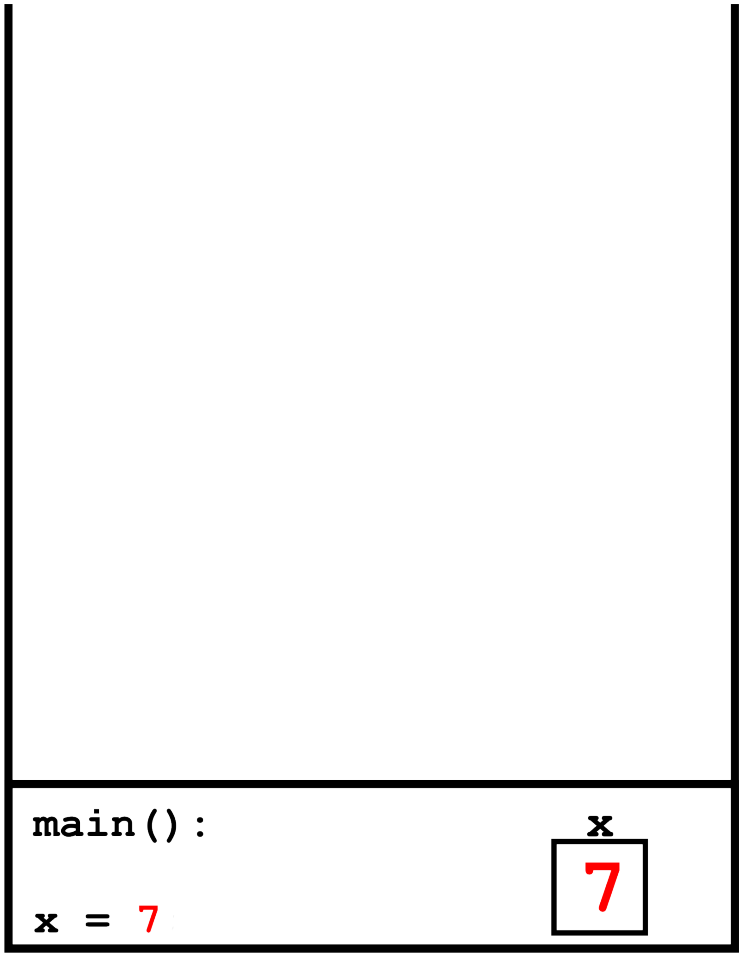
Gerenciamento de Memória



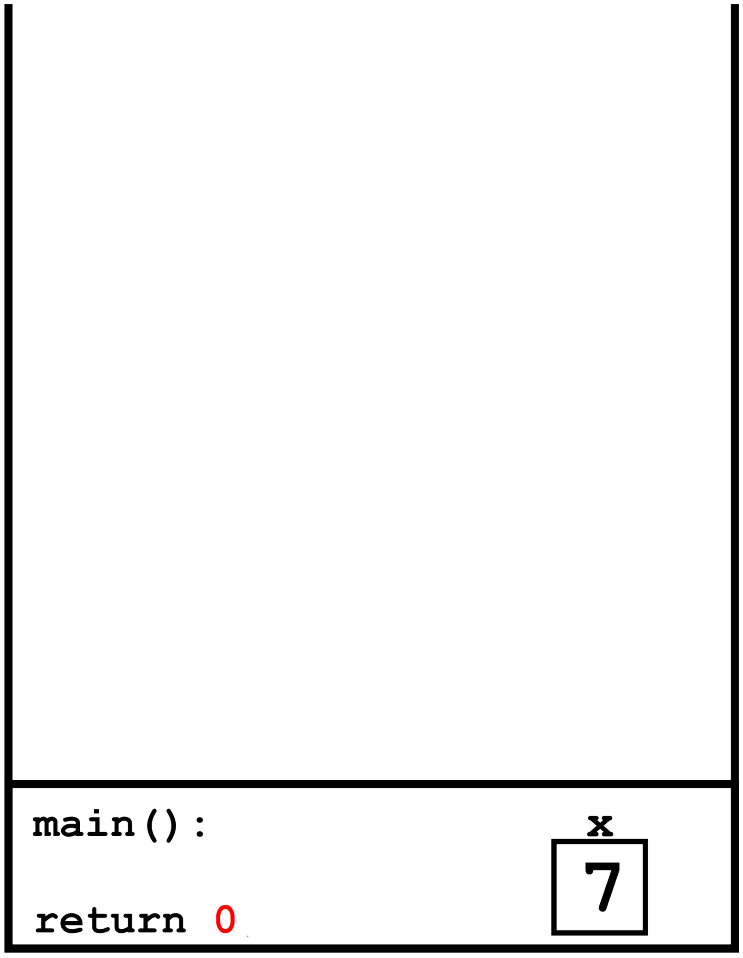
Gerenciamento de Memória



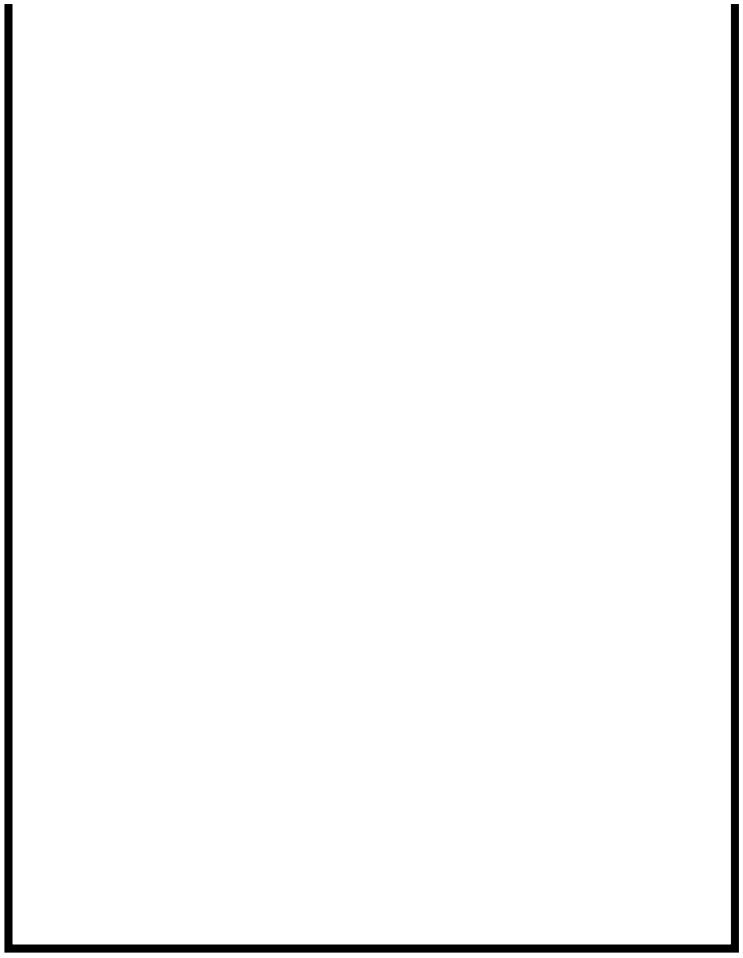
Gerenciamento de Memória



Gerenciamento de Memória



Gerenciamento de Memória



Gerenciamento de Memória

- No caso de chamadas recursivas é como se cada chamada correspondesse à chamada de uma função distinta.
- As execuções das chamadas de funções recursivas são feitas na pilha, assim como qualquer função.
- O último conjunto de variáveis alocadas na pilha, que está no topo, corresponde às variáveis da última chamada da função.
- Quando termina a execução de uma chamada da função, as variáveis locais desta chamada são removidas da pilha.

Gerenciamento de Memória

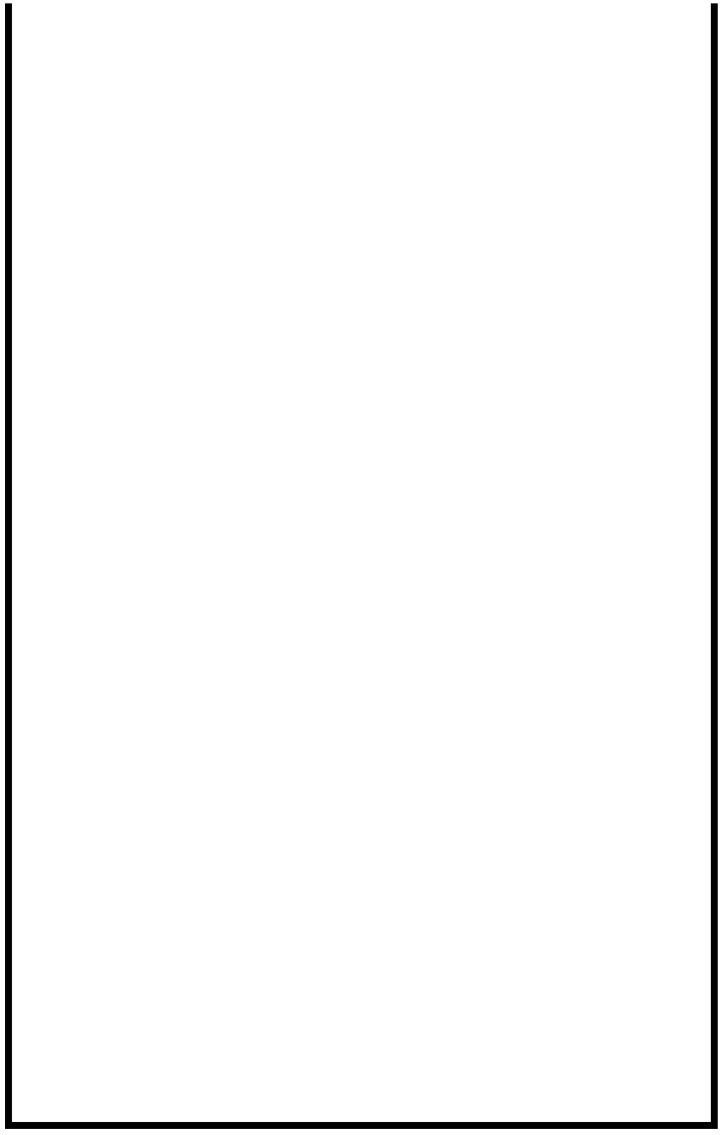
- Considere novamente a solução recursiva para se calcular o fatorial e assuma que seja feita a chamada `fatorial(4)`.

```
1 def fatorial(n):
2   if n == 0:
3     return 1
4   else:
5     aux = fatorial(n - 1)
6     return n * aux
```

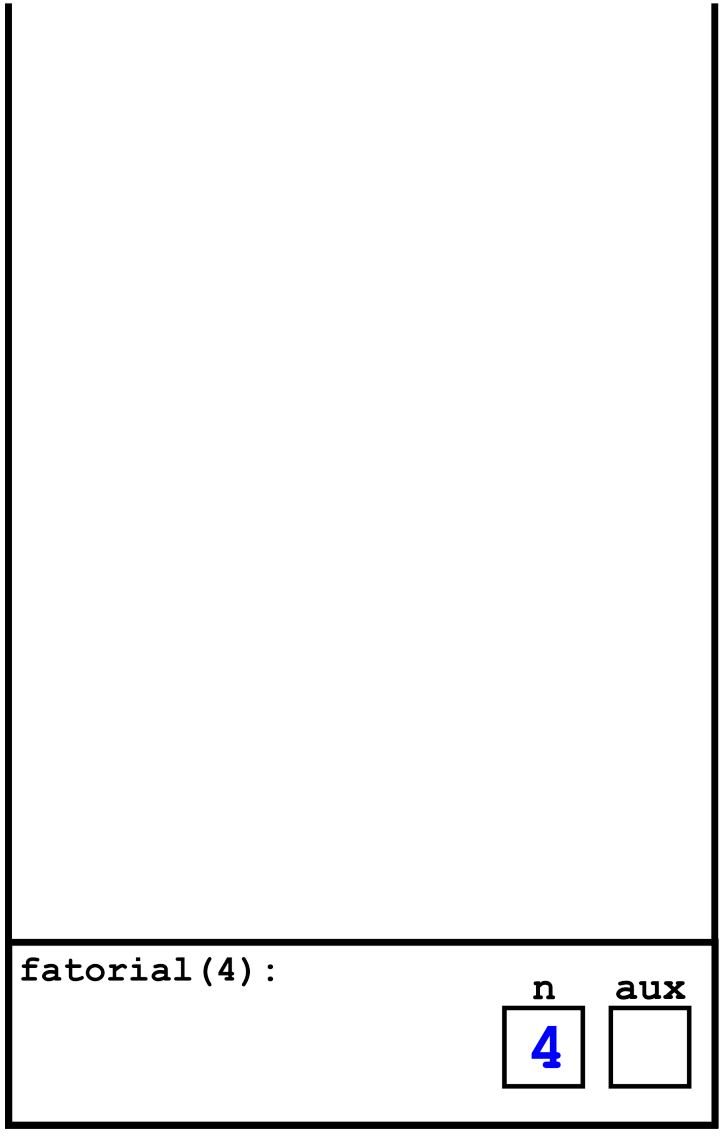
Gerenciamento de memória

- Cada chamada da função `fatorial` cria novas variáveis locais de mesmo nome (`n` e `aux`).
- Portanto, múltiplas variáveis (`n` e `aux`) podem existir em um dado momento.
- Em um dado instante, o nome `n` (ou `aux`) refere-se à variável local ao corpo da função que está sendo executada naquele instante.
- A seguir, veremos o estado da pilha de execução para a chamada `fatorial(4)`.

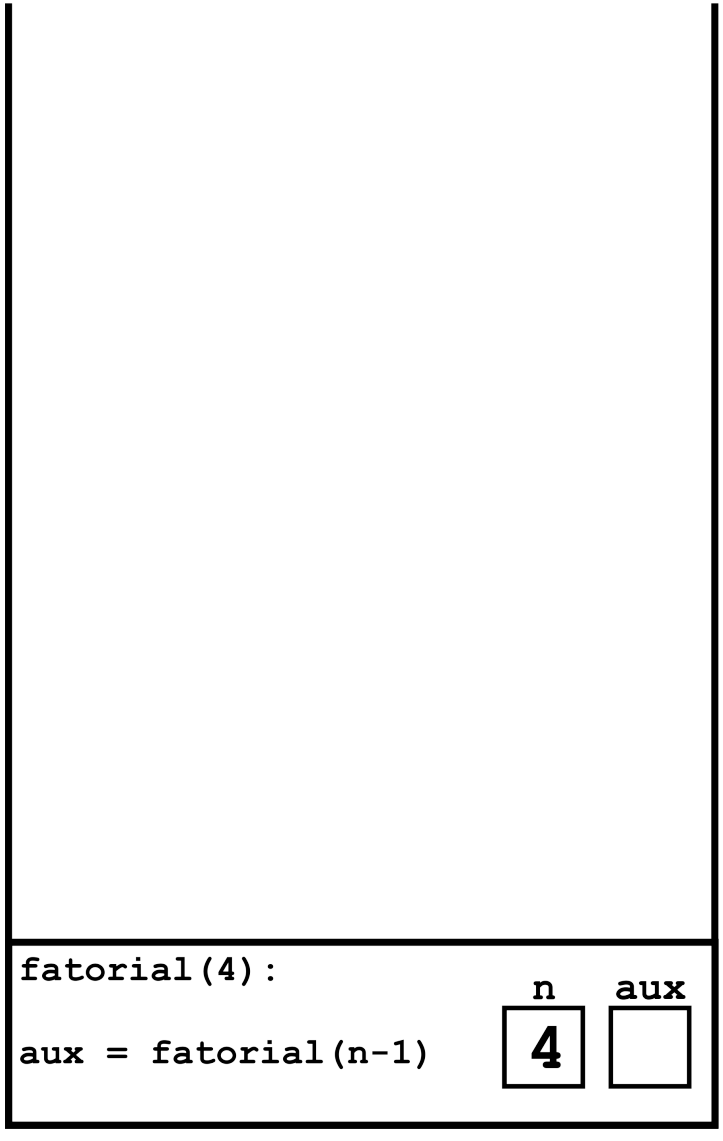
Gerenciamento de Memória



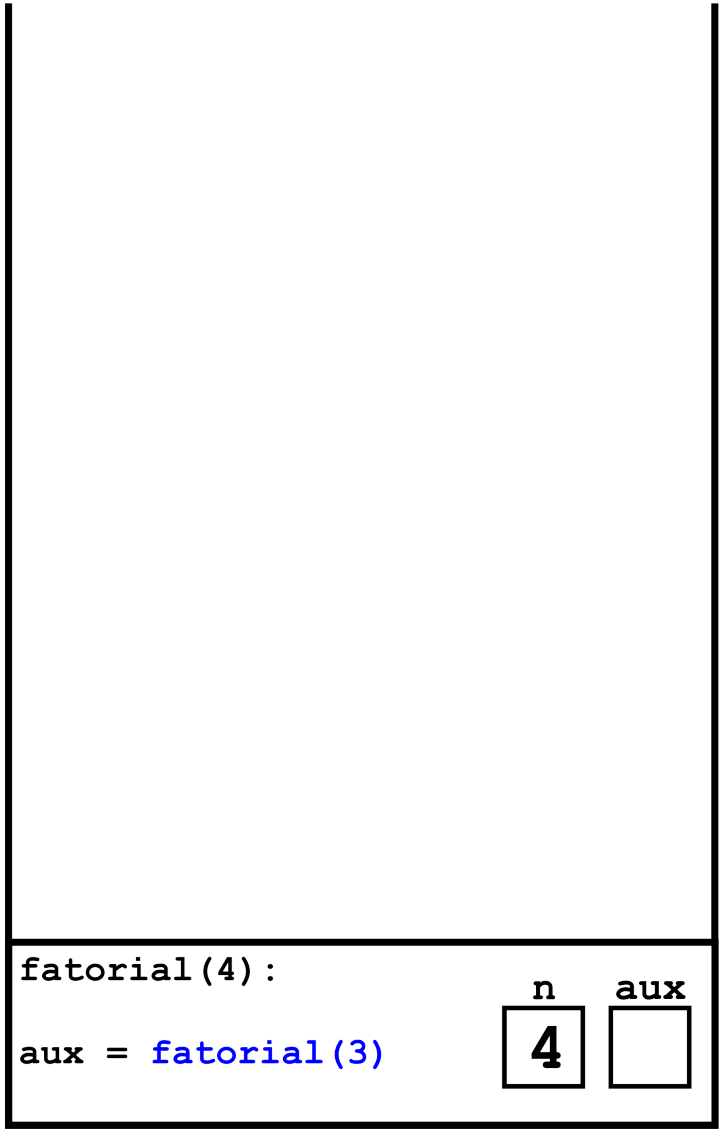
Gerenciamento de Memória



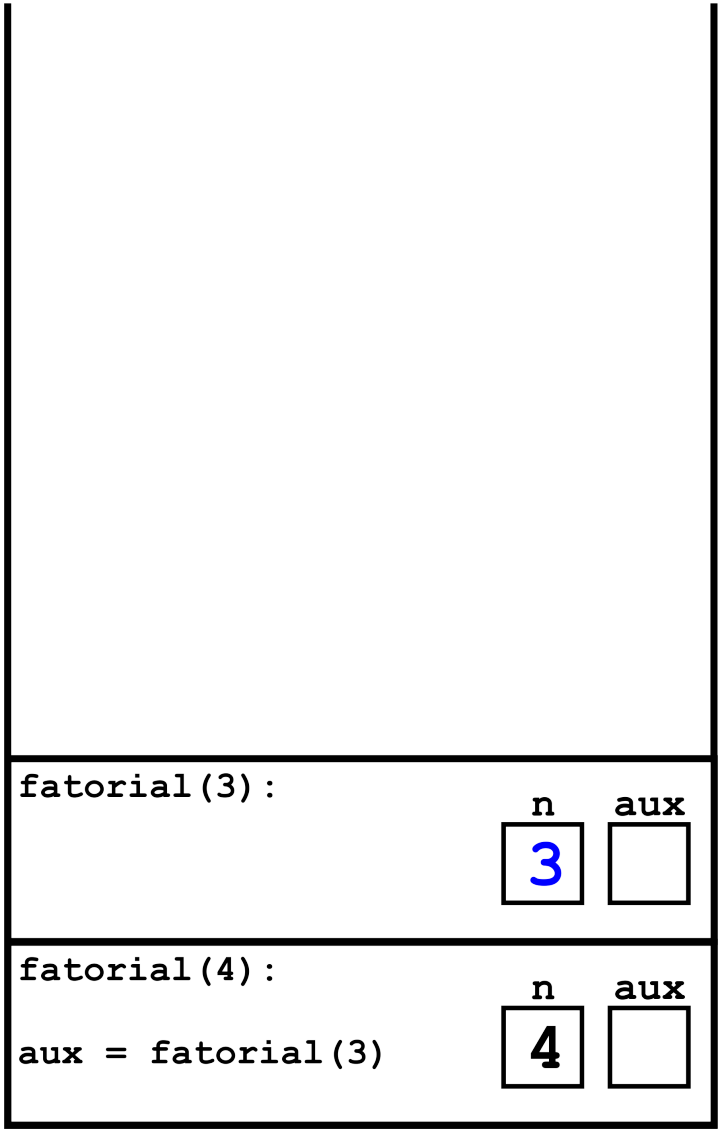
Gerenciamento de Memória



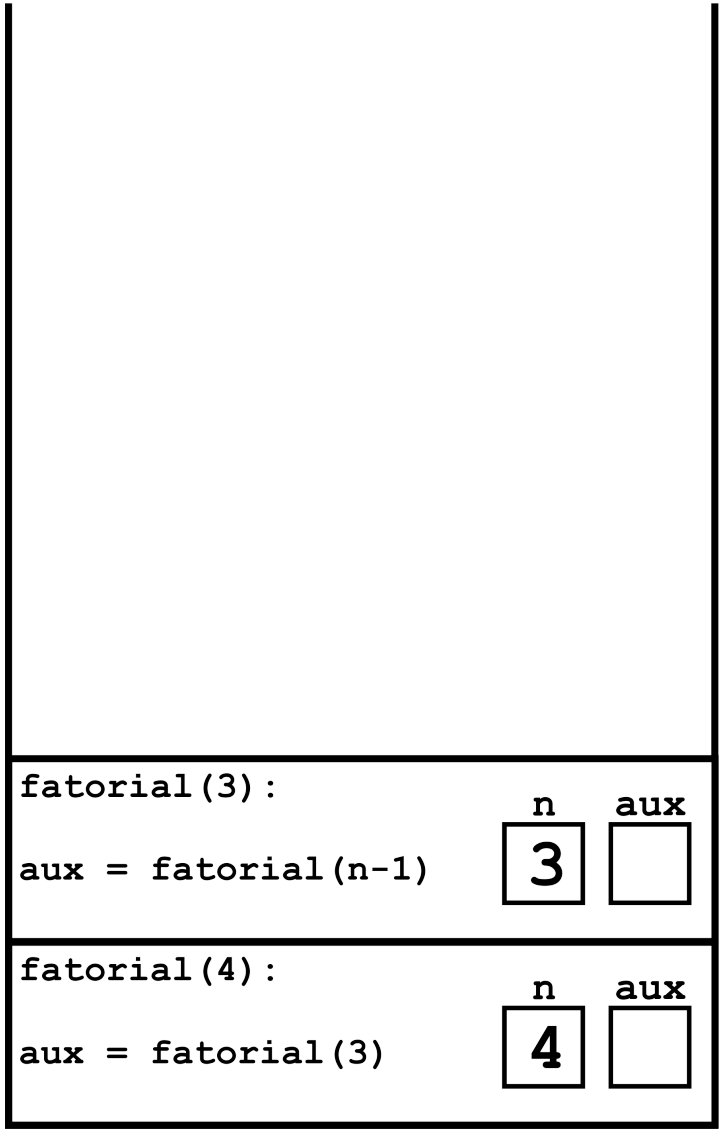
Gerenciamento de Memória



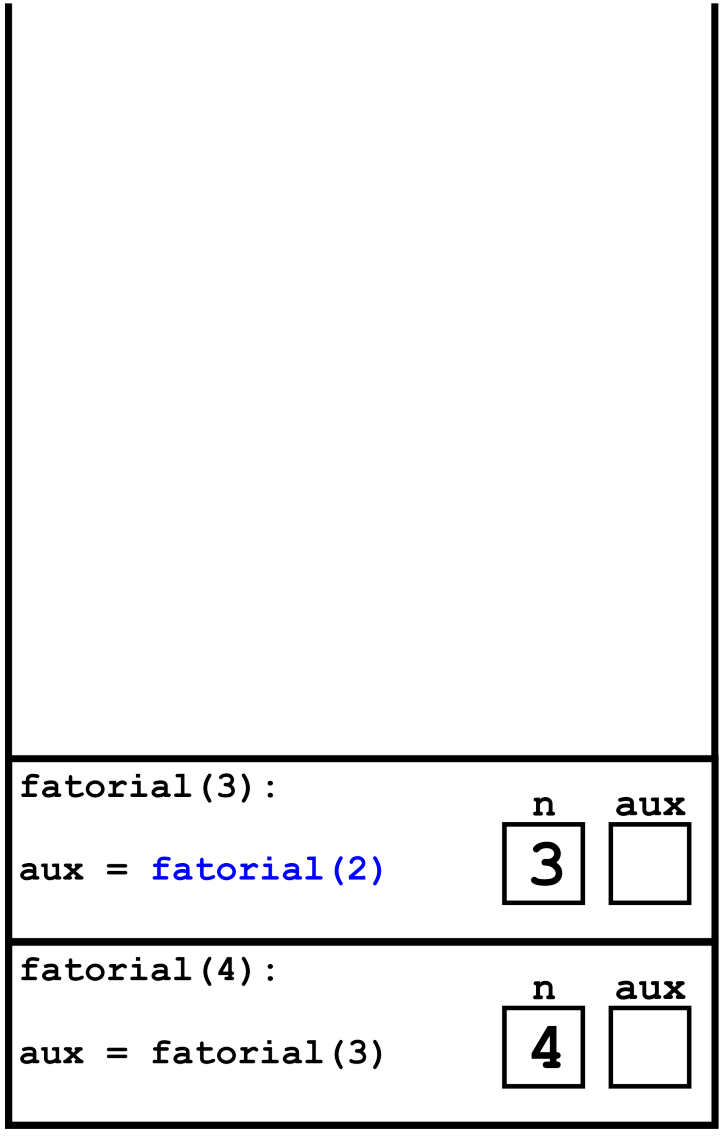
Gerenciamento de Memória



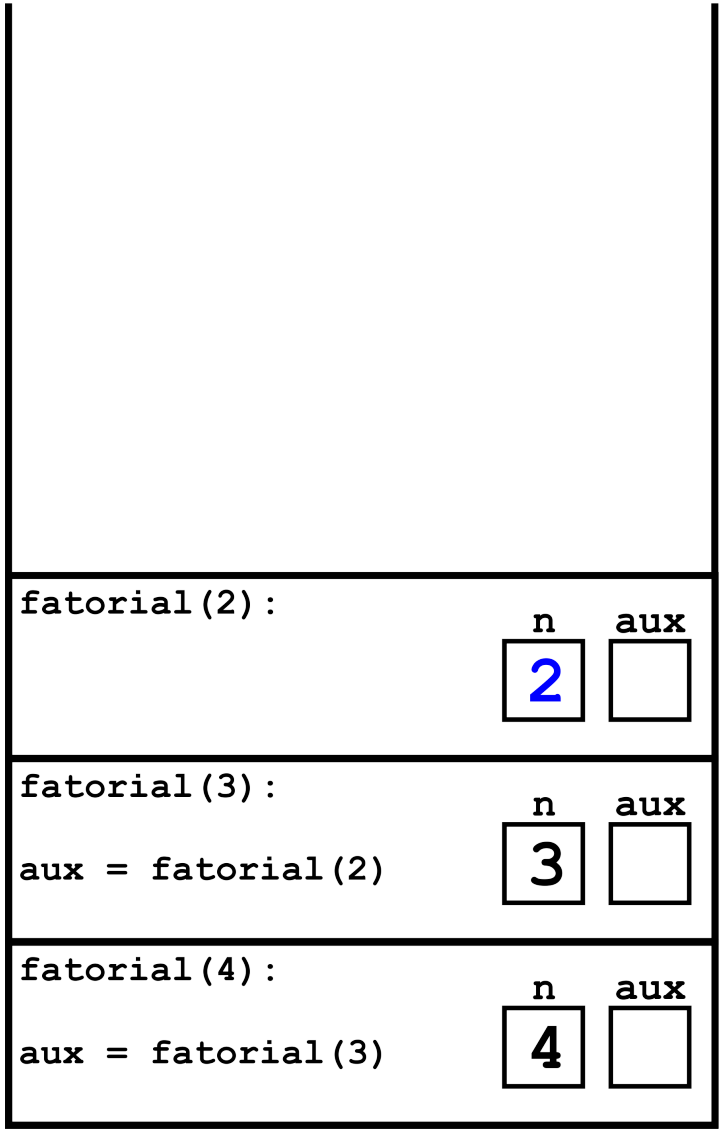
Gerenciamento de Memória



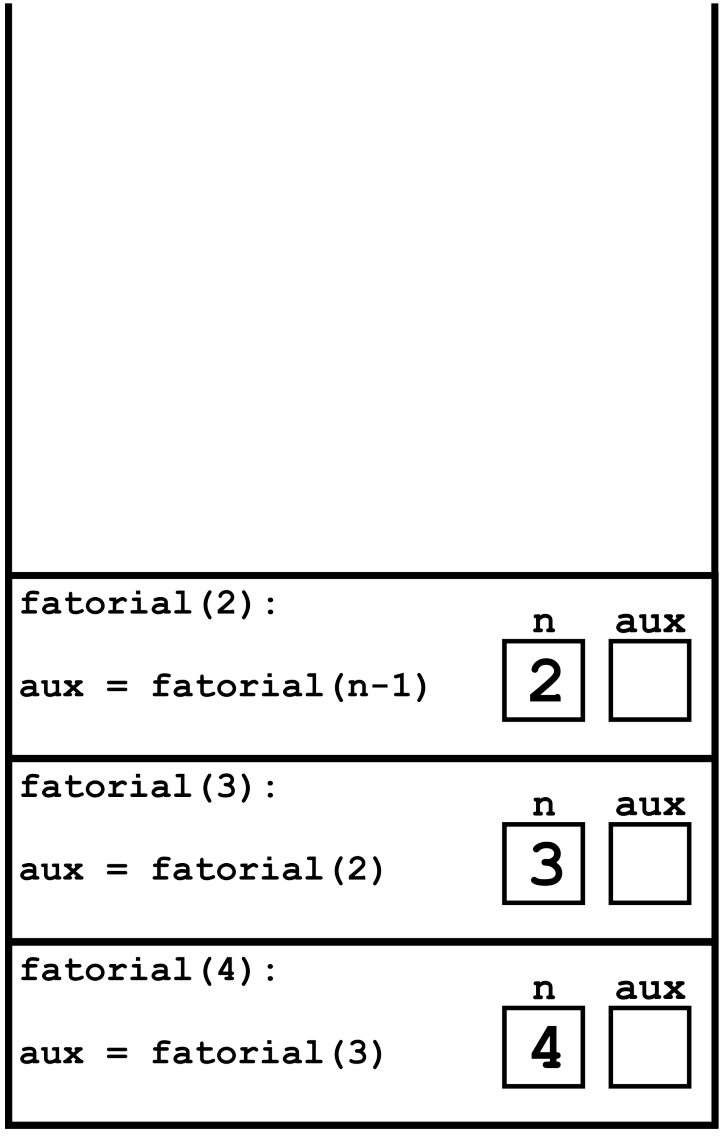
Gerenciamento de Memória



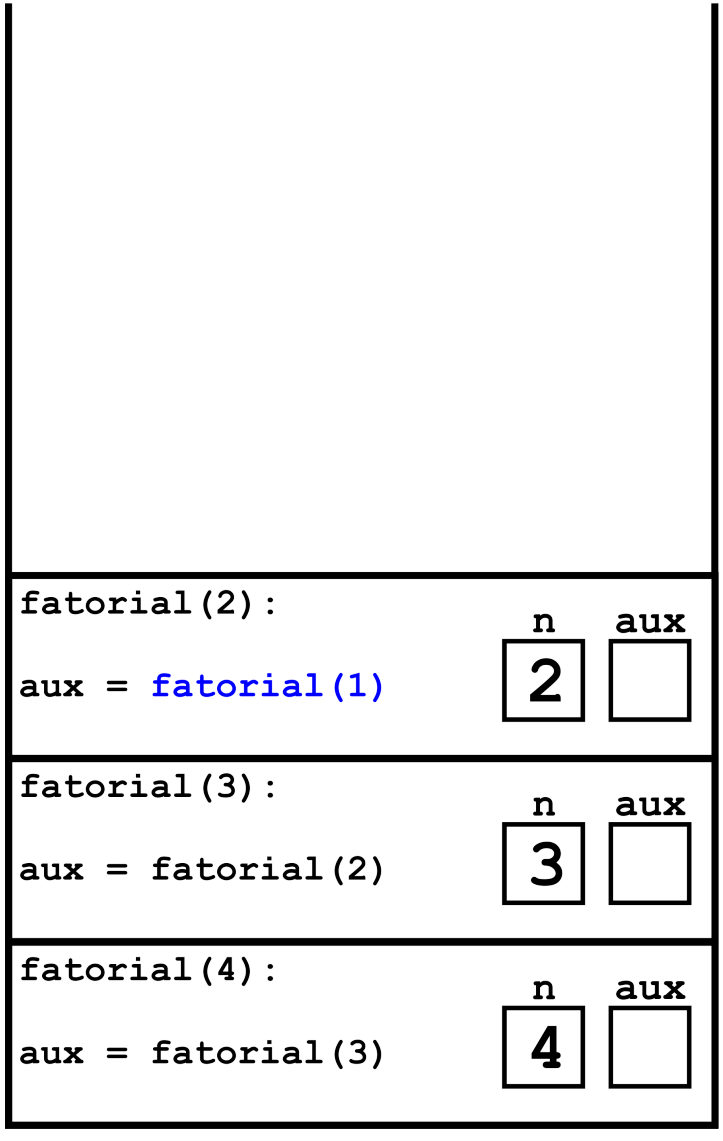
Gerenciamento de Memória



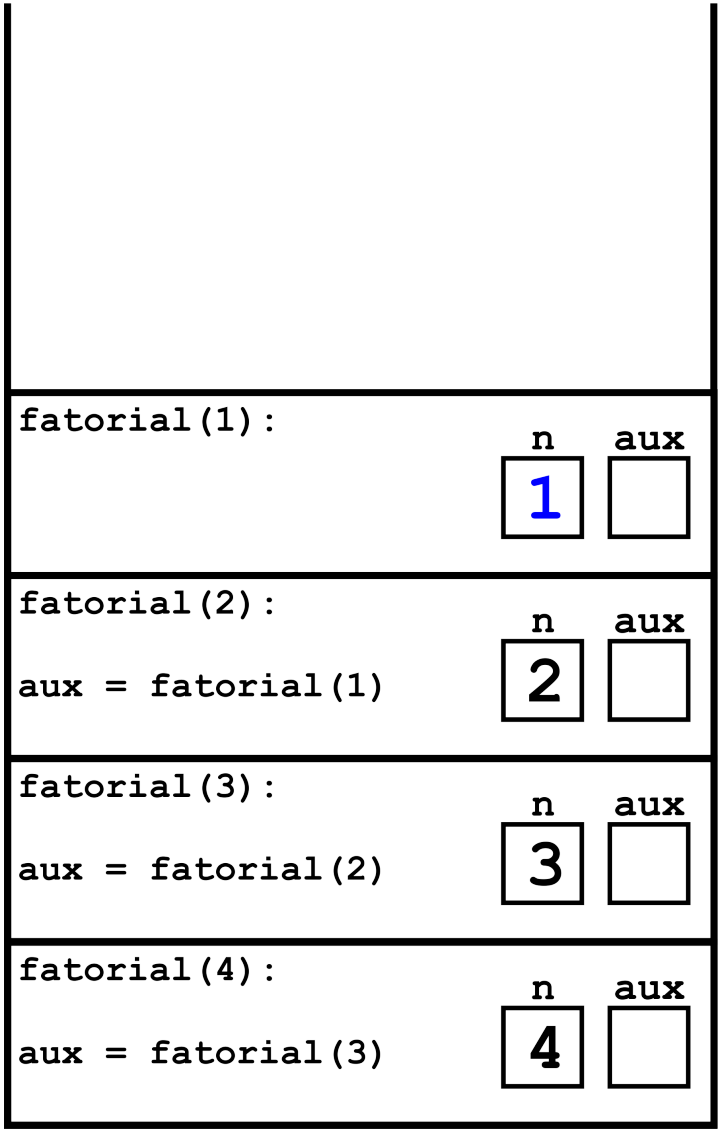
Gerenciamento de Memória



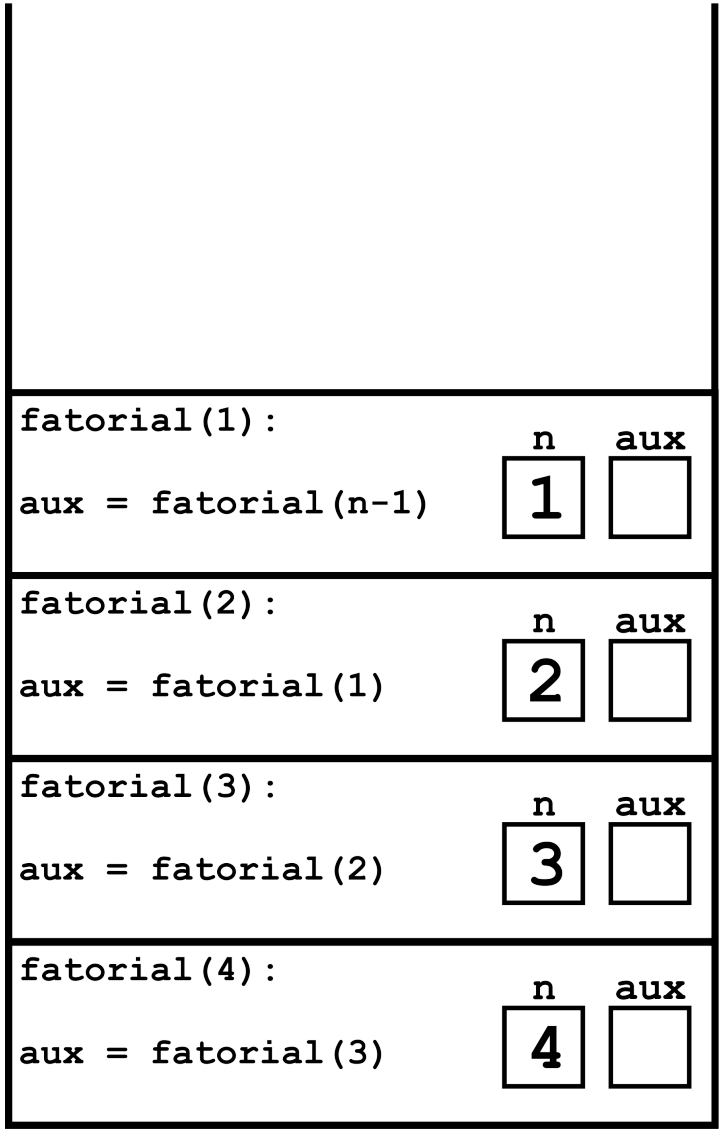
Gerenciamento de Memória



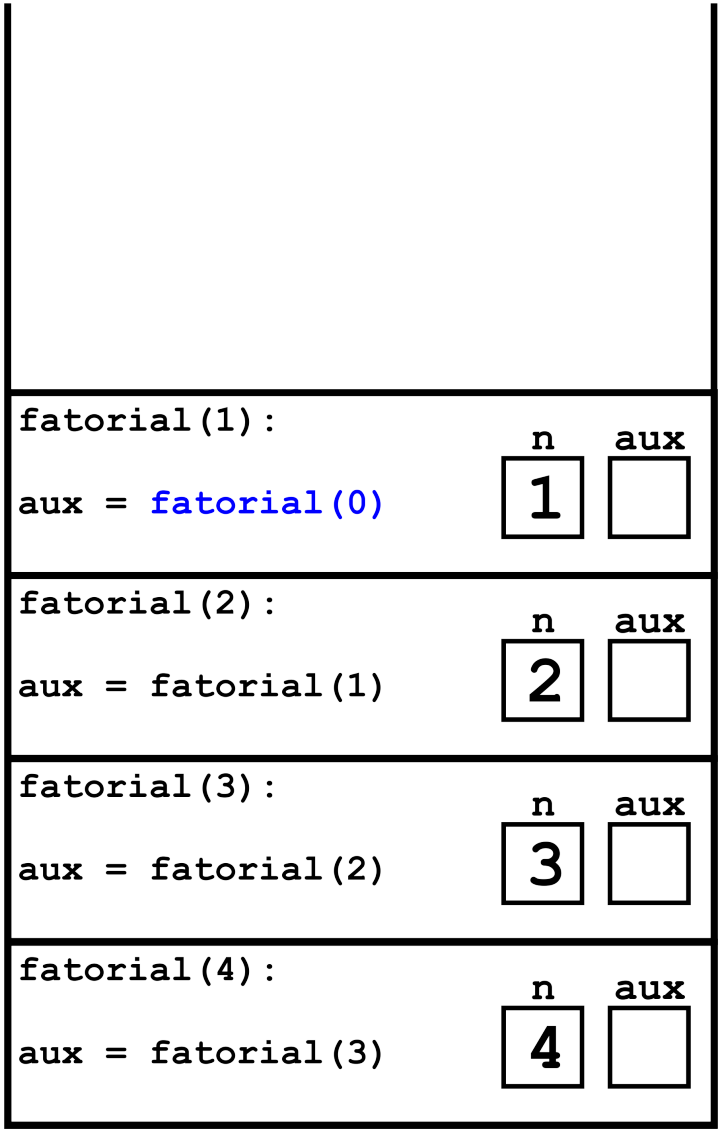
Gerenciamento de Memória



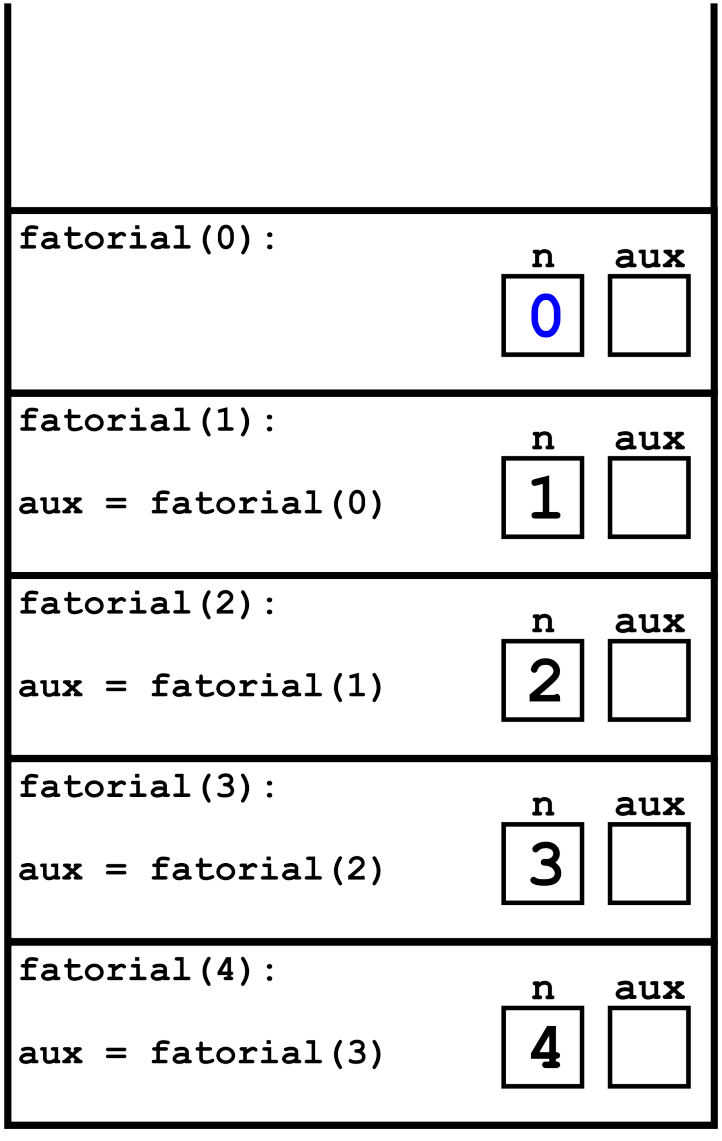
Gerenciamento de Memória



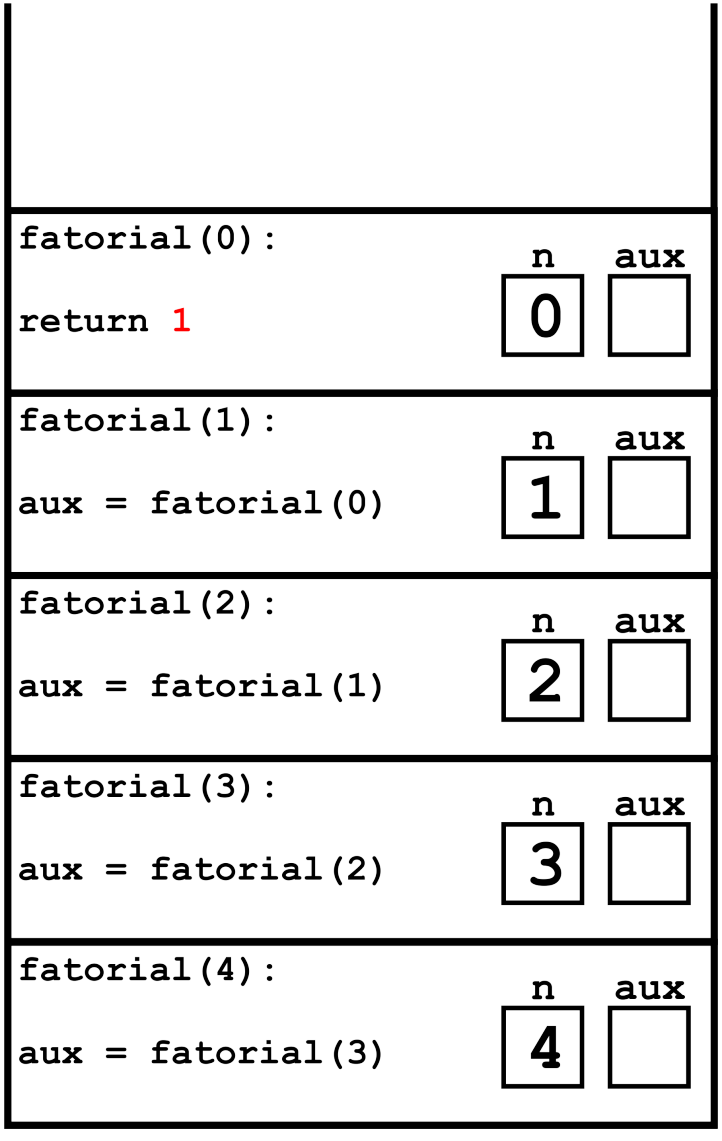
Gerenciamento de Memória



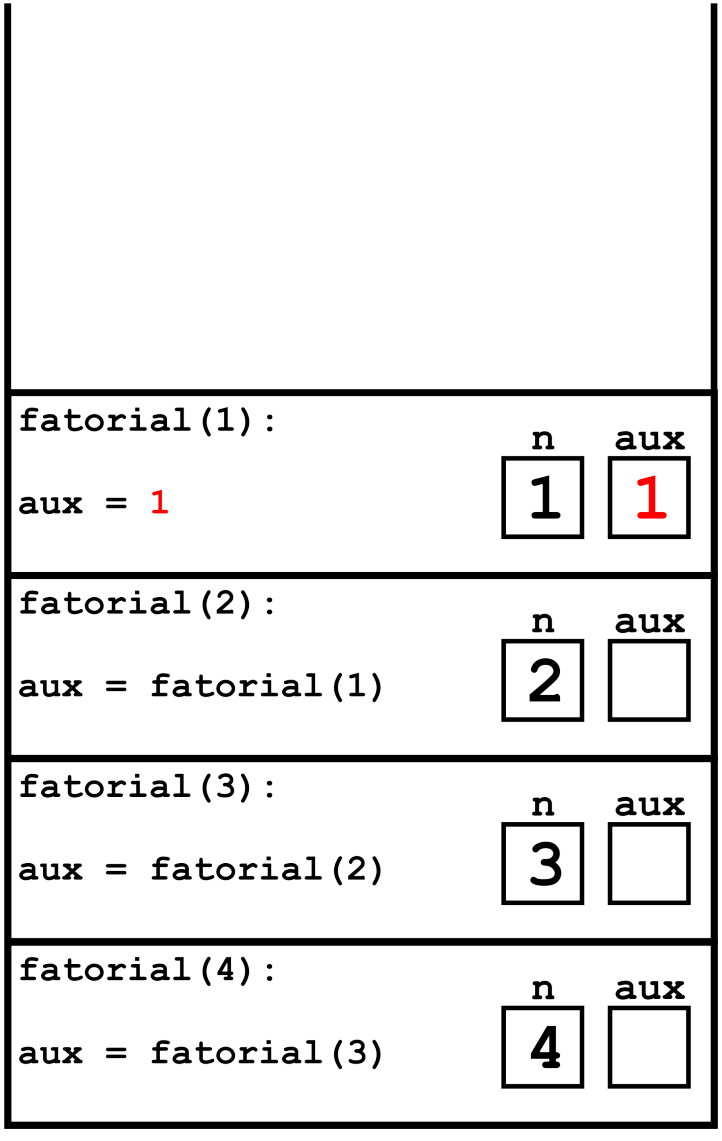
Gerenciamento de Memória



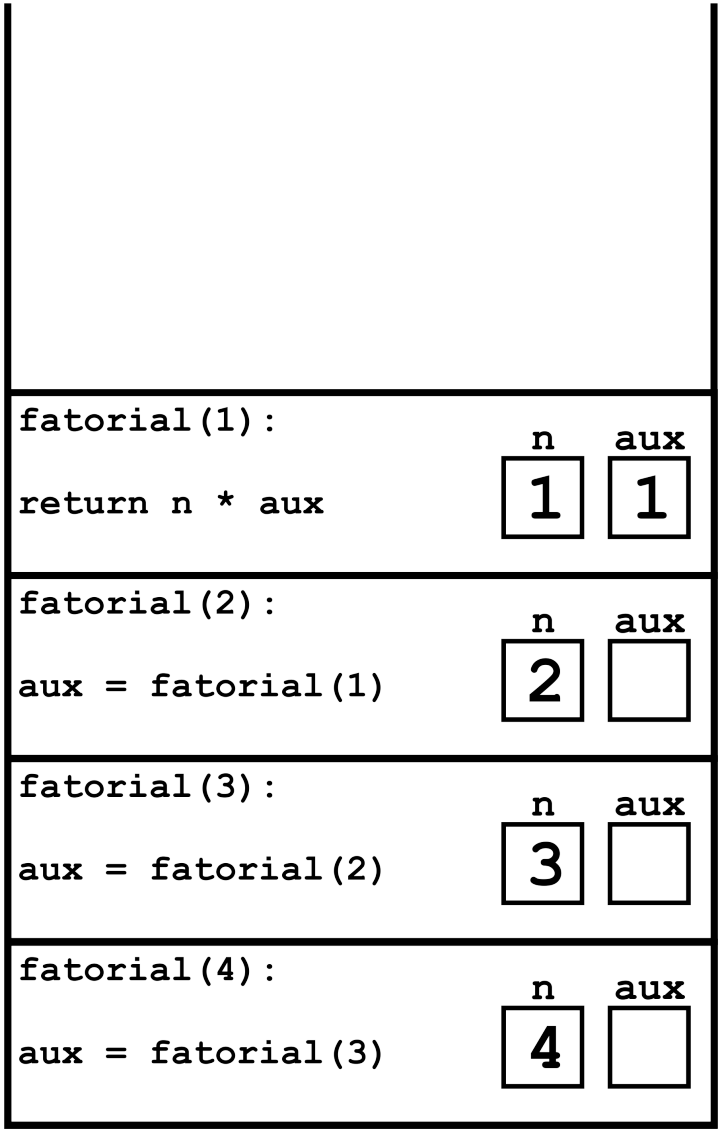
Gerenciamento de Memória



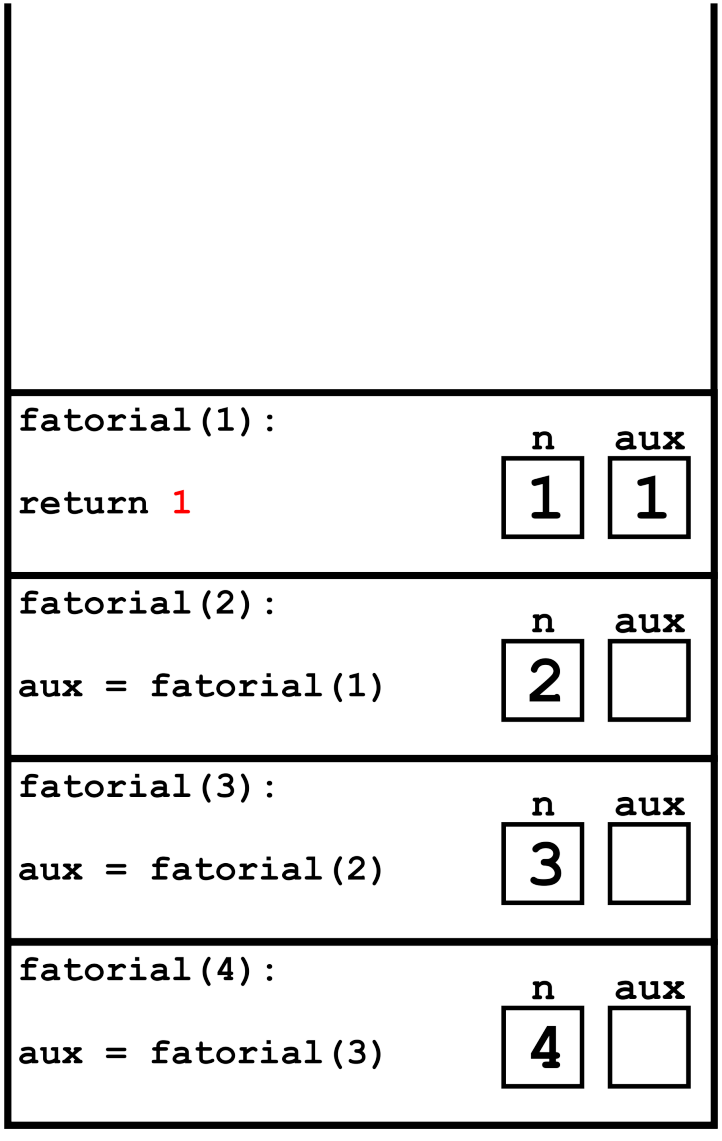
Gerenciamento de Memória



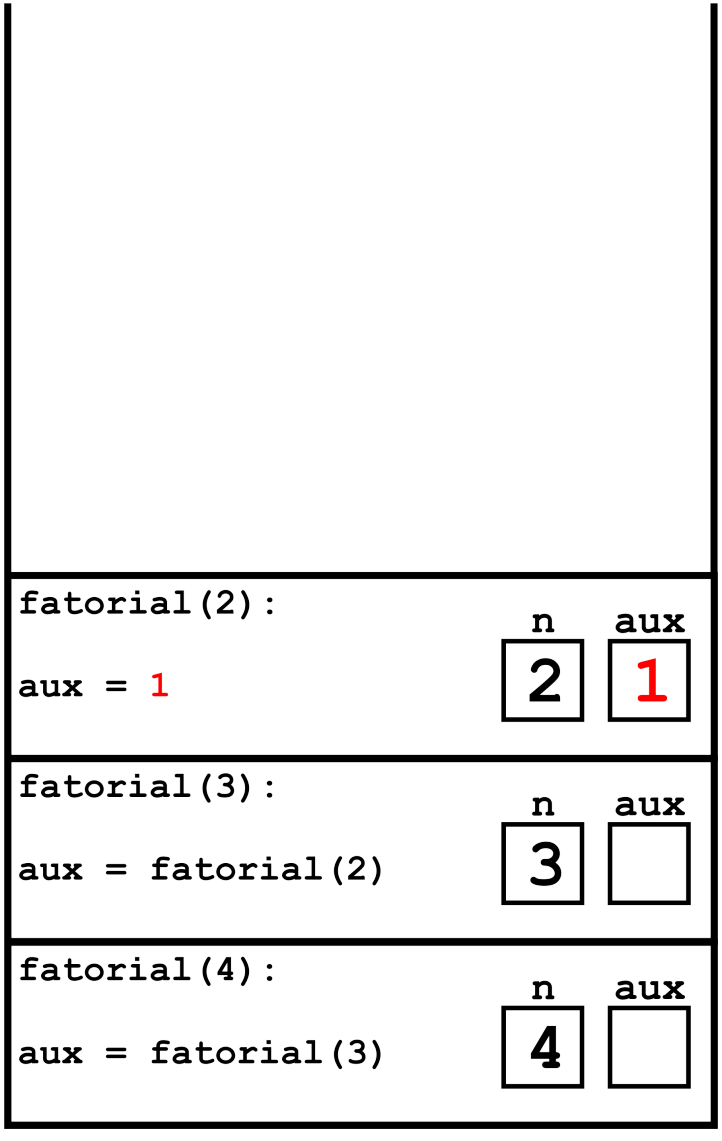
Gerenciamento de Memória



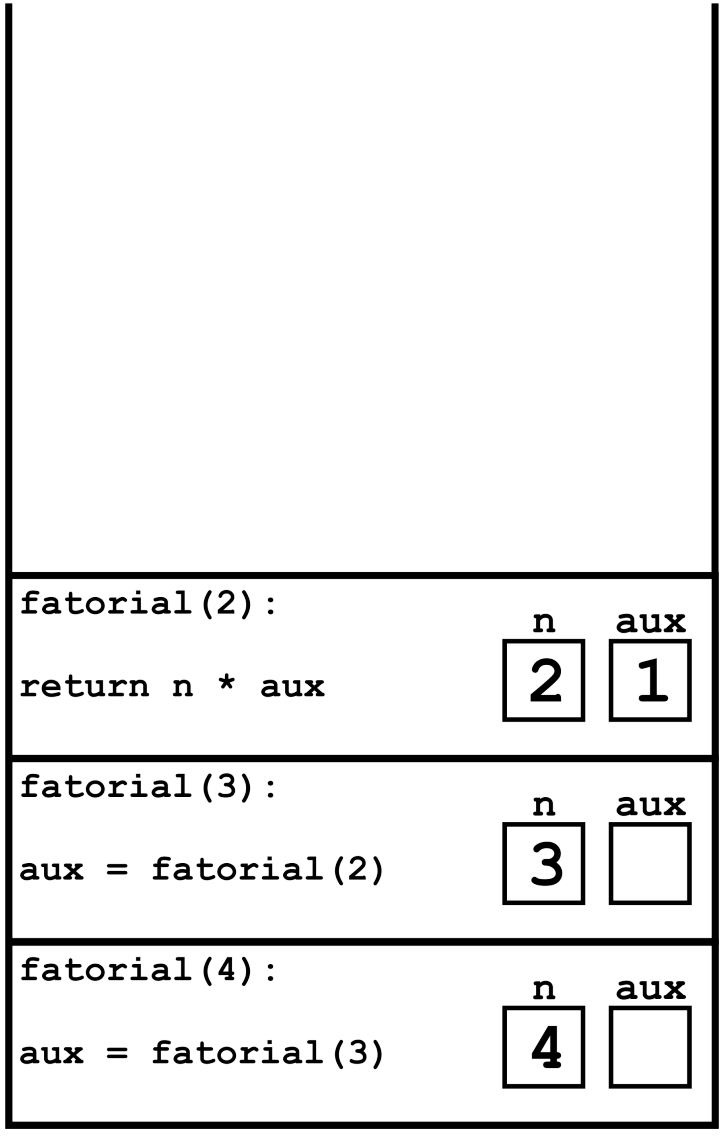
Gerenciamento de Memória



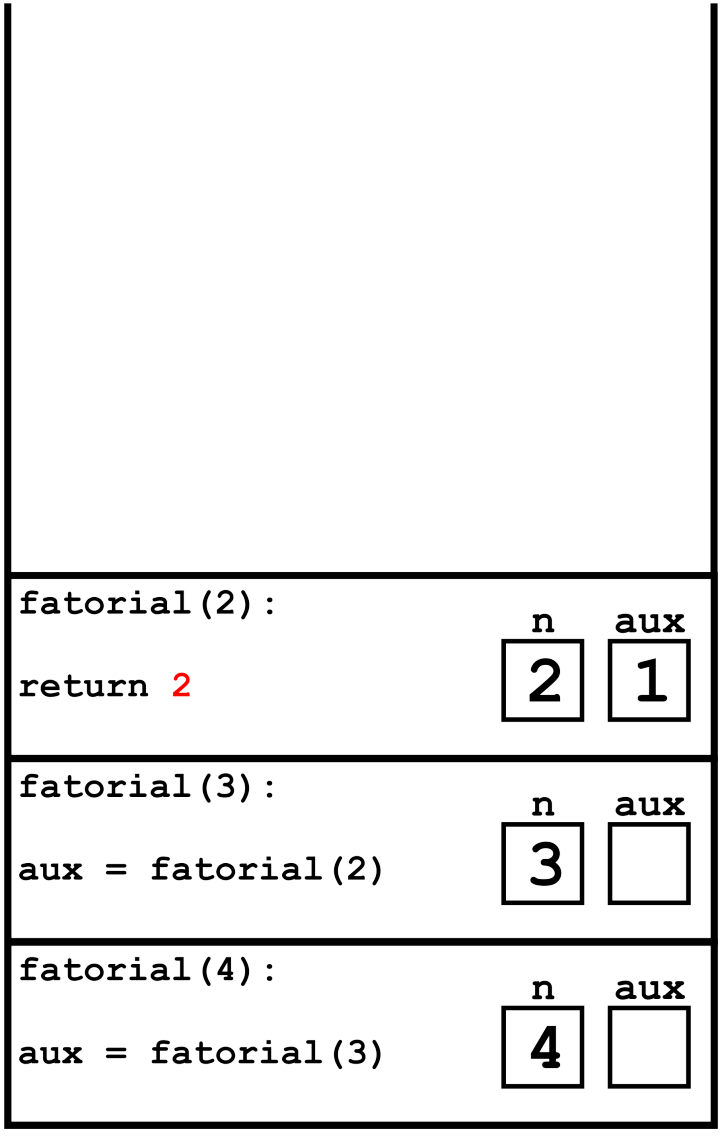
Gerenciamento de Memória



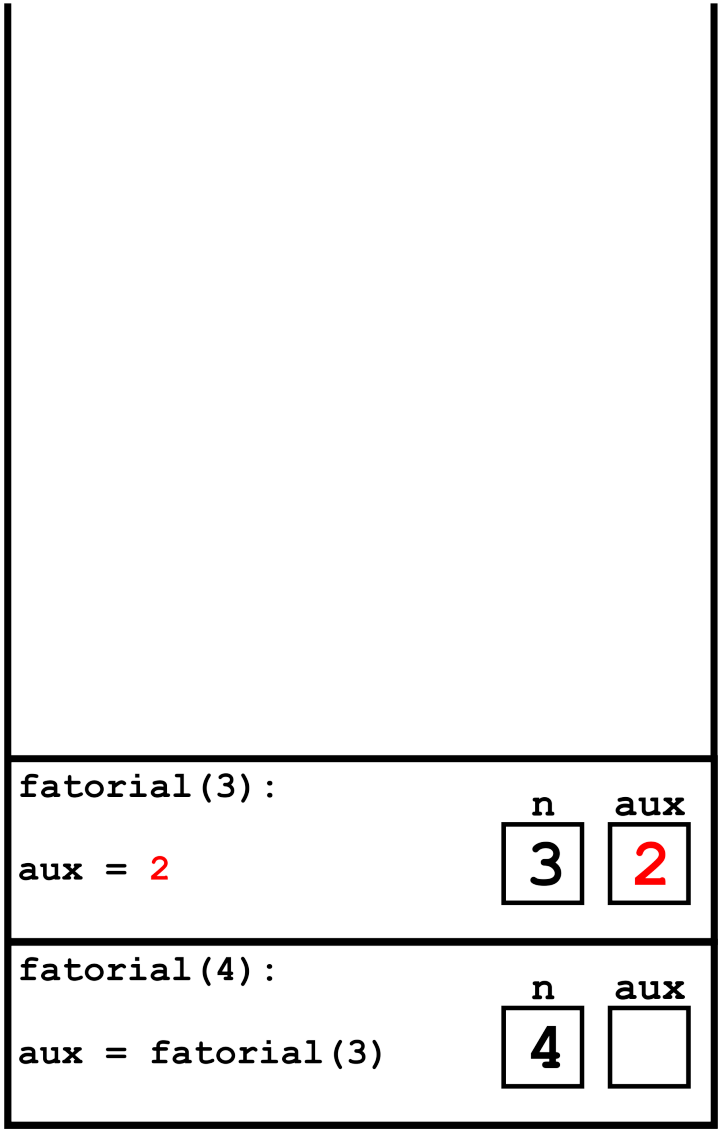
Gerenciamento de Memória



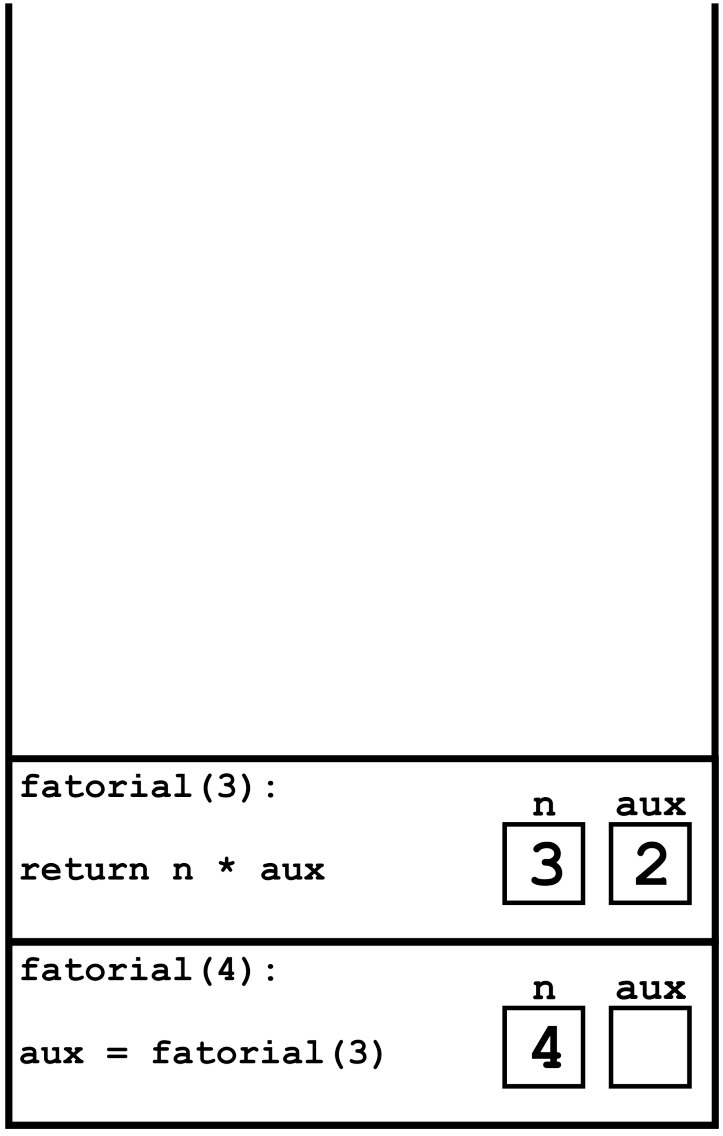
Gerenciamento de Memória



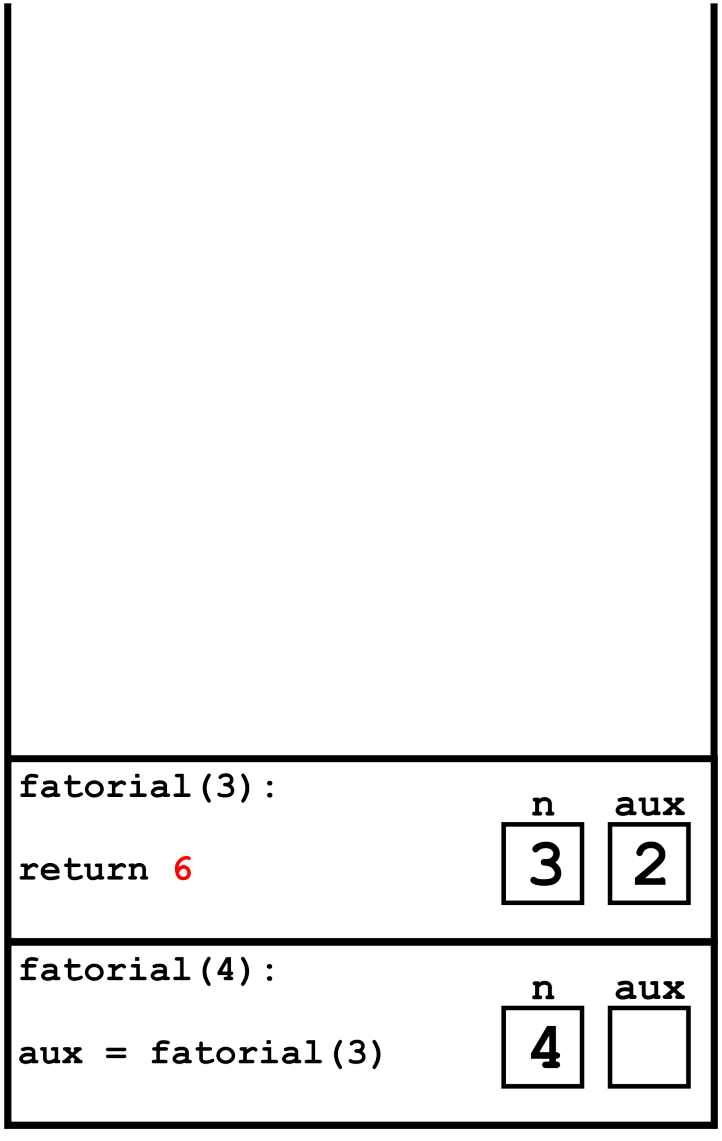
Gerenciamento de Memória



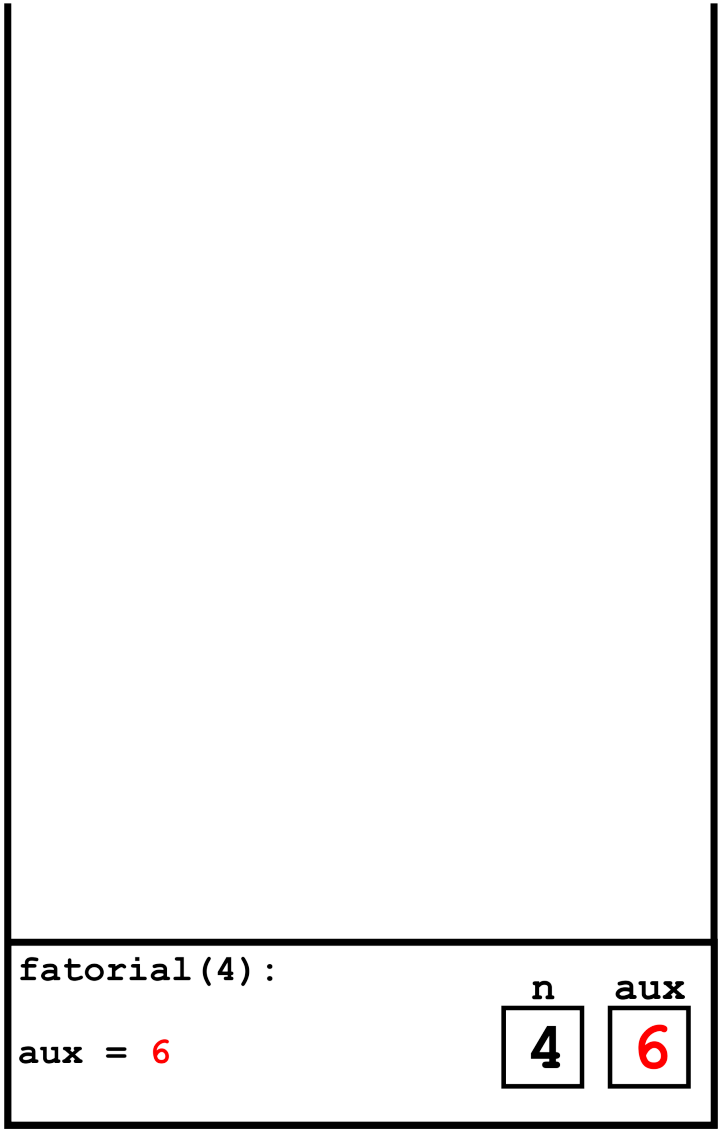
Gerenciamento de Memória



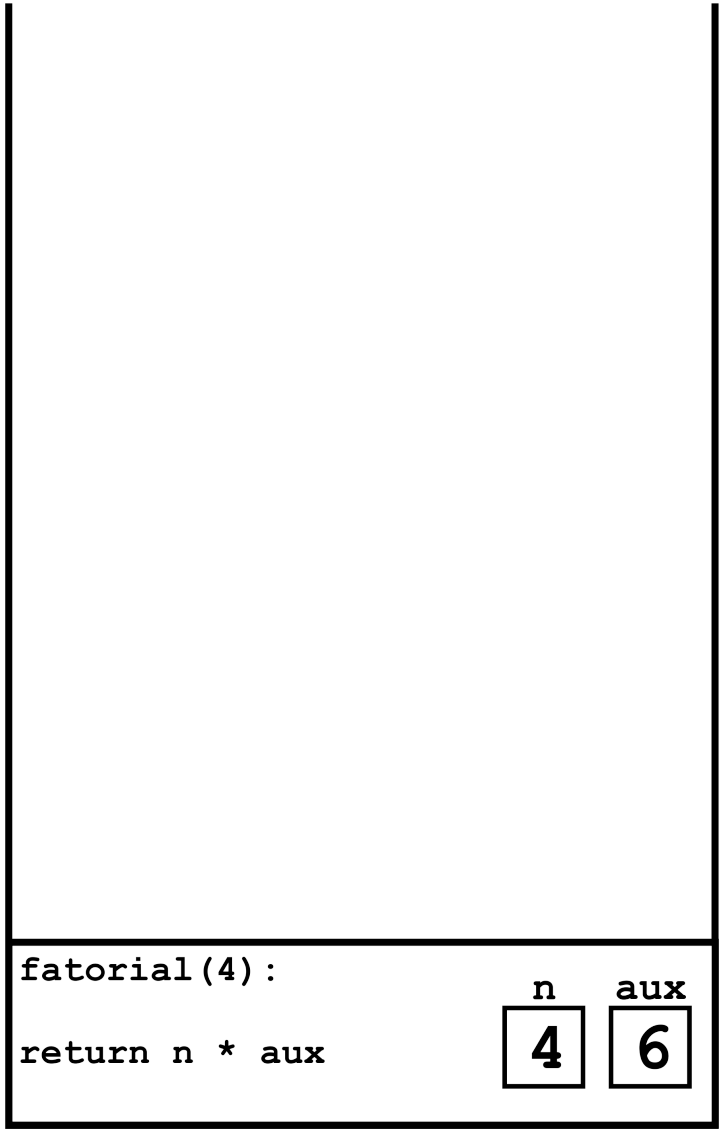
Gerenciamento de Memória



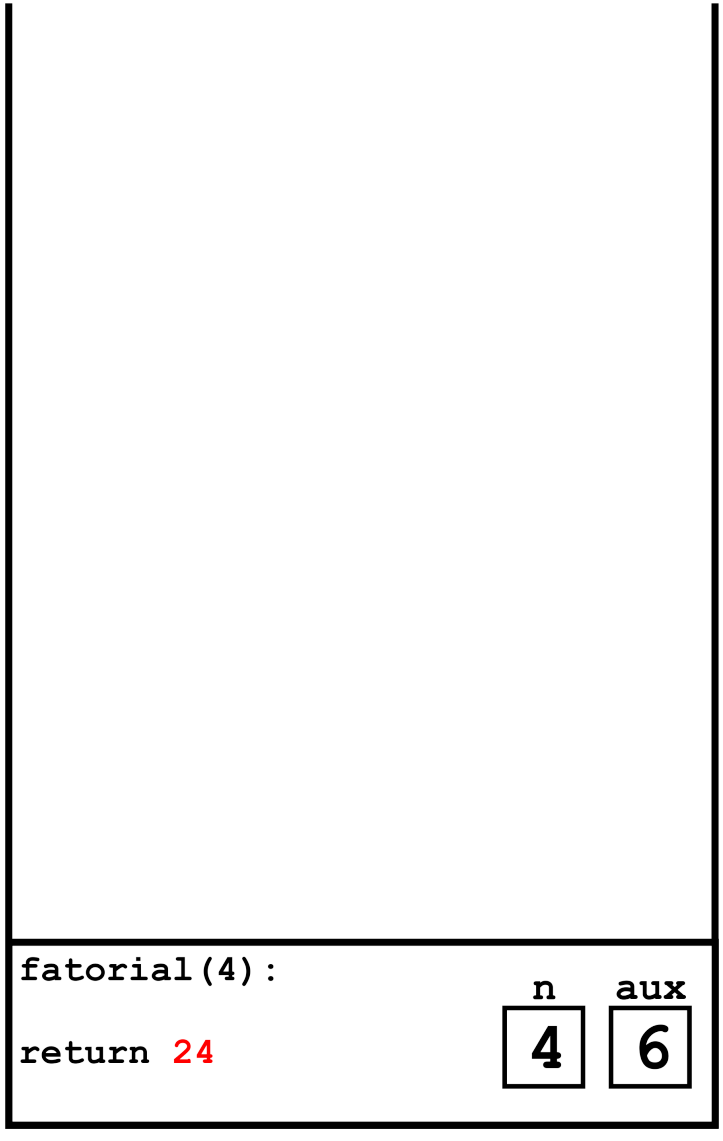
Gerenciamento de Memória



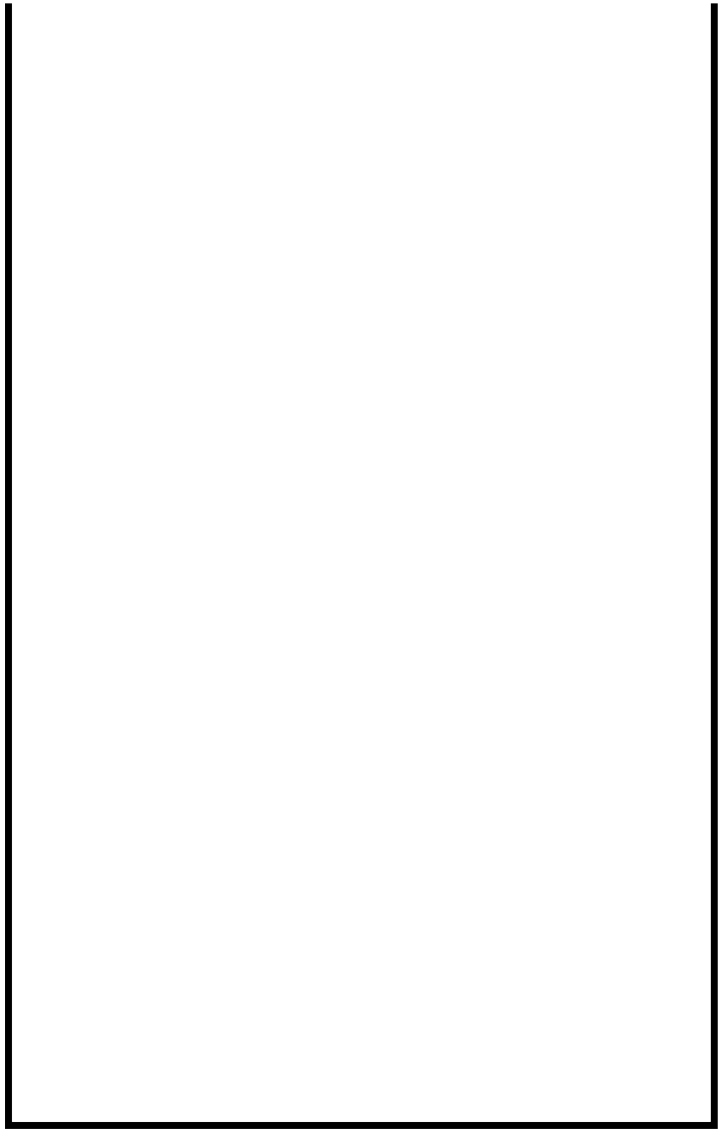
Gerenciamento de Memória



Gerenciamento de Memória



Gerenciamento de Memória



Gerenciamento de Memória

- Note que a variável `aux` é desnecessária. Além disso, o `else` também é desnecessário.

```
1 def fatorial(n):
2   if n == 0:
3     return 1
4   else:
5     return n * fatorial(n - 1)
6
```

Gerenciamento de Memória

- Note que a variável `aux` é desnecessária. Além disso, o `else` também é desnecessário.

```
1 def fatorial(n):
2   if n == 0:
3     return 1
4
5   return n * fatorial(n - 1)
6
```

Recursão $\times$ Iteração

Recursão × Iteração

- Soluções recursivas são geralmente mais concisas do que as iterativas.
- Soluções iterativas em geral consomem menos memória do que as soluções recursivas.
- A cópia dos parâmetros a cada chamada recursiva gera um custo adicional para as soluções recursivas.

Recursão × Iteração

- No caso da função `fatorial`, a solução iterativa é mais eficiente:

```
1 def fatorial(n):
2   fat = 1
3   for i in range(1, n + 1):
4     fat = fat * i
5   return fat
```

Soma dos Elementos de uma Lista

- Seja `l` uma lista de inteiros de tamanho `n`.
- Queremos saber a soma de todos os seus elementos.
- Como podemos descrever este problema de forma recursiva?
- Vamos denotar por `soma(l, n)` a soma dos `n` primeiros elementos de uma lista `l`. Com isso, temos:
  - Se `n = 0`, `soma(l, 0) = 0`.
  - Se `n > 0`, `soma(l, n) = soma(l, n - 1) + l[n - 1]`.

Soma dos Elementos de uma Lista

```
1 def soma(l, n):
2   if n == 0:
3     return 0
4
5   return soma(l, n - 1) + l[n - 1]
```

Soma dos Elementos de uma Lista

- O método recursivo sempre termina, pois:
  - Existe um caso base bem definido.
  - A cada chamada recursiva, usamos um valor menor de **n**.

Soma dos Elementos de uma Lista

- Neste caso, a solução iterativa também é melhor que a recursiva:

```
1 def soma(l, n):
2   soma = 0
3
4   for i in range(n):
5     soma = soma + l[i]
6
7   return soma
```

Soma dos Elementos de uma Lista

- Neste caso, a solução iterativa também é melhor que a recursiva:

```
1 def soma(l, n):
2   soma = 0
3
4   for i in l:
5     soma = soma + i
6
7   return soma
```

Números de Fibonacci

Números de Fibonacci

- A série de Fibonacci é a seguinte:
  - $1, 1, 2, 3, 5, 8, 13, 21, 34, 55, \ldots$.
- Suponha que queremos determinar qual é o `n`-ésimo número da série de Fibonacci, que denotaremos por `Fibonacci(n)`.
- Como descrever o `n`-ésimo número de Fibonacci de forma recursiva?

Números de Fibonacci

- Temos os seguintes casos base:
  - Se `n = 1` ou `n = 2`, então `Fibonacci(n) = 1`.
- Caso contrário, podemos computar `Fibonacci(n)` como:
  - `Fibonacci(n) = Fibonacci(n-1) + Fibonacci(n-2)`

Números de Fibonacci (Versão Recursiva)

```
1 def fibonacci(n):
2   if n == 1 or n == 2:
3     return 1
4
5   return fibonacci(n - 1) + fibonacci(n - 2)
```

Números de Fibonacci (Versão Iterativa)

```
1  def fibonacci(n):
2    fib = [1, 1]
3
4    for i in range(2, n):
5      fib.append(fib[i - 1] + fib[i - 2])
6
7    return fib[n - 1]
```

Números de Fibonacci (Versão Iterativa)

```
1 def fibonacci(n):
2   anterior = atual = 1
3
4   for i in range(2, n):
5     (anterior, atual) = (atual, atual + anterior)
6
7   return atual
```

Números de Fibonacci

- Para `n > 2`, quantas somas são necessárias para calcular `Fibonacci(n)` usando uma estratégia iterativa?
  - São necessárias `n - 2` somas.
- Para `n > 2`, quantas somas são realizadas para calcular `Fibonacci(n)` usando a função recursiva vista anteriormente?
  - `Fibonacci(3)`: 1 soma.
  - `Fibonacci(4)`: 2 somas.
  - `Fibonacci(5)`: 4 somas.
  - `Fibonacci(6)`: 7 somas.
  - `Fibonacci(7)`: 12 somas.
  - `Fibonacci(8)`: 20 somas.
  - `Fibonacci(9)`: 33 somas.
  - `Fibonacci(10)`: 54 somas.
  - `Fibonacci(20)`: 6.764 somas.
  - `Fibonacci(50)`: 12.586.269.024 somas.
  - São realizadas `Fibonacci(n) - 1` somas.

Números de Fibonacci

- Por exemplo, para `n = 20`, os números de vezes que a função recursiva calcula `Fibonacci(k)`, para $1 \leq k \leq 20$, são os seguintes:

`Fibonacci(1)`: 2584	`Fibonacci(11)`: 55
`Fibonacci(2)`: 4181	`Fibonacci(12)`: 34
`Fibonacci(3)`: 2584	`Fibonacci(13)`: 21
`Fibonacci(4)`: 1597	`Fibonacci(14)`: 13
`Fibonacci(5)`: 987	`Fibonacci(15)`: 8
`Fibonacci(6)`: 610	`Fibonacci(16)`: 5
`Fibonacci(7)`: 377	`Fibonacci(17)`: 3
`Fibonacci(8)`: 233	`Fibonacci(18)`: 2
`Fibonacci(9)`: 144	`Fibonacci(19)`: 1
`Fibonacci(10)`: 89	`Fibonacci(20)`: 1

Números de Fibonacci

- A versão iterativa calcula uma única vez todos os valores de `Fibonacci(k)`, para $1 \leq$ `k` $\leq$ `n`.
- Para calcular o valor de `Fibonacci(n)`, a função recursiva calcula o valor de `Fibonacci(k)`, para $2 \leq k \leq n$, `Fibonacci(n - k + 1)` vezes.

Resumindo

- Recursão é uma técnica para se criar algoritmos em que:
  - Devemos descrever soluções para casos base.
  - Assumindo a existência de soluções para casos mais simples, mostramos como obter uma solução para um caso mais complexo.
- Algoritmos recursivos geralmente são mais claros e concisos.
- Devemos avaliar a clareza de código $\times$ eficiência do algoritmo.

Exemplos Simples

Recursão

- Dada a seguinte função e chamada recursiva:

```
1 def rec(n):
2   print(n)
3   if n > 0:
4     rec(n-1)
5
6 rec(5)
```

- O que a função imprime como resposta?

```
1 5
2 4
3 3
4 2
5 1
6 0
```

Recursão

- Mudando a posição do comando de impressão na função.

```
def rec(n):
  if n > 0:
    rec(n-1)
  print(n)

rec(5)
```

- O que a função imprime como resposta agora?

```
0
1
2
3
4
5
```

Soma

- Soma de dois números inteiros não negativos, *x* e *y*, usando apenas incrementos e decrementos unitários.

```
1 def soma(x, y):
2   if y == 0:
3     return x
4
5   return soma(x + 1, y - 1)
```

Soma

- Soma de dois números inteiros não negativos, *x* e *y*, usando apenas incrementos e decrementos unitários.

```
1 def soma(x, y):
2   if y == 0:
3     return x
4
5   return soma(x, y - 1) + 1
```

Multiplicação

- Multiplicação de dois números inteiros positivos, *x* e *y*, usando apenas somas e subtrações.

```
1 def mult(x, y):
2   if y == 1:
3     return x
4
5   return mult(x, y - 1) + x
```

Soma de Valores Pares

- Soma de todos os inteiros positivos pares menores ou iguais a um valor inteiro *n*.

```
1 def soma_par(n):
2   if n <= 0:
3     return 0
4   else:
5     if n % 2 == 0:
6       return soma_par(n - 2) + n
7     else:
8       return soma_par(n - 1)
```

Soma de Valores Pares

- Soma de todos os inteiros positivos pares menores ou iguais a um valor inteiro *n*.

```
1 def soma_par(n):
2   if n <= 0:
3     return 0
4
5   if n % 2 == 0:
6     return soma_par(n - 2) + n
7   else:
8     return soma_par(n - 1)
```

Soma de Valores Pares

- Soma de todos os inteiros positivos pares menores ou iguais a um valor inteiro *n*.

```
1 def soma_par(n):
2   if n <= 1:
3     return 0
4
5   if n % 2 == 0:
6     return soma_par(n - 2) + n
7   else:
8     return soma_par(n - 1)
```

Produtório

- Cálculo do produtório $\prod_{i=m}^{n} i = m \times (m+1) \times (m+2) \times \cdots \times n$, tal que m e n são inteiros tais que $m \leq n$.

```
1 def produtorio(m, n):
2   if m == n:
3     return m
4   else:
5     return produtorio(m, n - 1) * n
```

Produtório

- Cálculo do produtório $\prod_{i=m}^{n} i = m \times (m+1) \times (m+2) \times \cdots \times n$, tal que m e n são inteiros tais que $m \leq n$.

```
1 def produtorio(m, n):
2   if m == n:
3     return m
4   else:
5     return m * produtorio(m + 1, n)
```

Potência

- Cálculo do valor de k^n, onde n é um número inteiro não negativo.

$$k^n = \begin{cases} 1, & \text{se } n = 0 \\ k \times k^{n-1}, & \text{caso contrário} \end{cases}$$

```
1 def potencia(k, n):
2   if n == 0:
3     return 1
4   else:
5     return k * potencia(k, n - 1)
```

Potência

- Cálculo do valor de k^n, onde n é um número inteiro não negativo.

$$k^n = \begin{cases} 1, & \text{se } n = 0 \\ k \times k^{n-1}, & \text{caso contrário} \end{cases}$$

```
1 def potencia(k, n):
2   pot = 1
3   for i in range(n):
4     pot = pot * k
5   return pot
```

Potência

- Podemos redefinir k^n (sendo n um inteiro não negativo) da seguinte forma:

$$k^n = \begin{cases} 1, & \text{se } n = 0 \\ k^{n/2} \times k^{n/2}, & \text{se } n \text{ for positivo e par} \\ k \times k^{\lfloor n/2 \rfloor} \times k^{\lfloor n/2 \rfloor}, & \text{se } n \text{ for positivo e ímpar} \end{cases}$$

- Note que definimos a solução do caso mais complexo em termos de casos mais simples. Usando esta definição, podemos implementar uma função iterativa ou recursiva, ambas mais eficientes que as versões anteriores.

Potência

```
1 def potencia(k, n):
2   if n == 0:
3     return 1
4   else:
5     aux = potencia(k, n // 2)
6     if n % 2 == 0:
7       return aux * aux
8     else:
9       return k * aux * aux
```

Potência

- Na nova versão do algoritmo, a cada chamada recursiva, o valor de n é dividido por 2. Ou seja, a cada chamada recursiva, o valor de n é reduzido, pelo menos, para a metade.
- Usando divisões inteiras, faremos no máximo $\lfloor \log_2 n \rfloor + 2$ chamadas recursivas.
- Por outro lado, a função iterativa original executa o laço n vezes.

Soma dos Dígitos de um Inteiro

- Soma dos dígitos de um número inteiro não negativo.

```
1 def soma_digitos(n):
2   if n == 0:
3     return 0
4   else:
5     return soma_digitos(n // 10) + (n % 10)
```

Soma dos Dígitos de um Inteiro

- Soma dos dígitos de um número inteiro não negativo.

```
1 def soma_digitos(n):
2   if n < 10:
3     return n
4   else:
5     return soma_digitos(n // 10) + (n % 10)
```

Máximo Divisor Comum

- O algoritmo de Euclides para o cálculo do Máximo Divisor Comum entre dois números inteiros não negativos x e y pode ser resumido na seguinte fórmula:

$$mdc(x, y) = \begin{cases} x, & \text{se } y = 0 \\ mdc(y, x \,\%\, y), & \text{se } y > 0 \end{cases}$$

Máximo Divisor Comum (Versão Iterativa)

```
1  def mdc(x, y):
2    while y > 0:
3      (x, y) = (y, x % y)
4
5    return x
```

Máximo Divisor Comum (Versão Recursiva)

```
1  def mdc(x, y):
2    if y == 0:
3      return x
4    else:
5      return mdc(y, x % y)
```

Exemplos com Listas e Strings

Maior Elemento de uma Lista

- Encontrar o maior elemento de uma `lista`.

```
def max_lista(lista):
  if len(n) == 1:
    return lista[0]
  else:
    aux = max_lista(lista[:-1])
    if aux > lista[-1]:
      return aux
    else:
      return lista[-1]
```

Maior Elemento de uma Lista

- Encontrar o maior elemento de uma `lista` com `n` números.

```
1 def max_lista(lista, n):
2   if n == 1:
3     return lista[0]
4   else:
5     aux = max_lista(lista, n - 1)
6     if aux > lista[n - 1]:
7       return aux
8     else:
9       return lista[n - 1]
```

Busca de um Caractere em uma String

- Buscar um caractere `c` em uma string `s` a partir de uma posição `index` e retornar a posição da primeira ocorrência deste caractere (caso encontre) ou -1 (caso contrário).

```
1 def strchr(s, c, index):
2   if index >= len(s):
3     return -1
4   if s[index] == c:
5     return index
6   else:
7     return strchr(s, c, index + 1)
```

Palíndromo

- Verificar se uma string é um palíndromo.

```
1 def palindromo(s):
2   if len(s) <= 1:
3     return True
4
5   if s[0] != s[-1]:
6     return False
7
8   return palindromo(s[1:-1])
```

Inverter uma String

- Inverter uma string dada.

```
1 def inverte(s):
2   if len(s) <= 1:
3     return s
4   else:
5     return s[-1] + inverte(s[:-1])
```

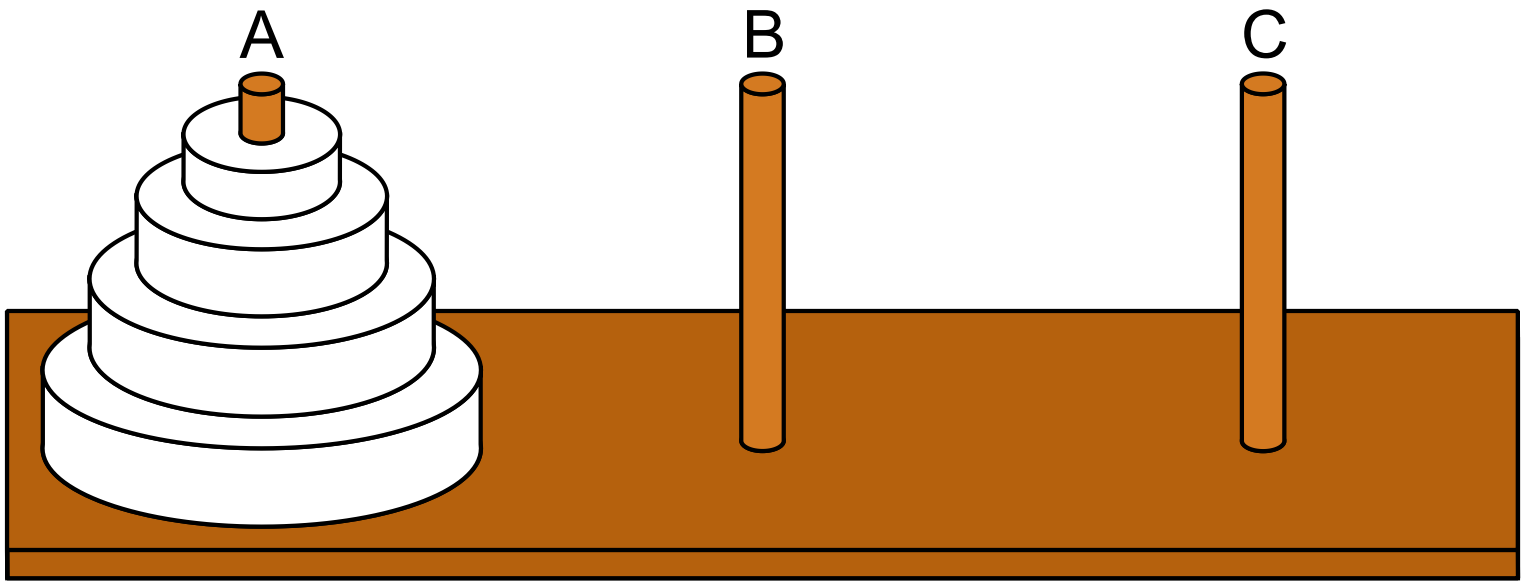
Inverter uma String

- Inverter uma string dada.

```
1 def inverte(s):
2   if len(s) <= 1:
3     return s
4   else:
5     return s[-1] + inverte(s[1:-1]) + s[0]
```

Torre de Hanói

Torre de Hanói



Torre de Hanói

- Considere n discos de diâmetros diferentes colocados em um pino A.
- O problema da Torre de Hanói consiste em transferir os n discos do pino A (inicial) para o pino C (final), usando um pino B como auxiliar.
- Entretanto, deve-se respeitar algumas regras:
  - Apenas o disco do topo de um pino pode ser movido.
  - Nunca um disco de diâmetro maior pode ficar sobre um disco de diâmetro menor.

Torre de Hanói

- O problema foi descrito pela primeira vez no ocidente em 1883 pelo matemático francês Édouard Lucas, baseado numa lenda hindu, onde Brahma havia ordenado que os monges do templo de Kashi Vishwanath movessem uma pilha de 64 discos de ouro, segundo as regras previamentes descritas.
- Quando todos os discos tivessem sido movidos, o mundo acabaria.

Torre de Hanói

- Vamos usar indução para obter um algoritmo para este problema.

Teorema

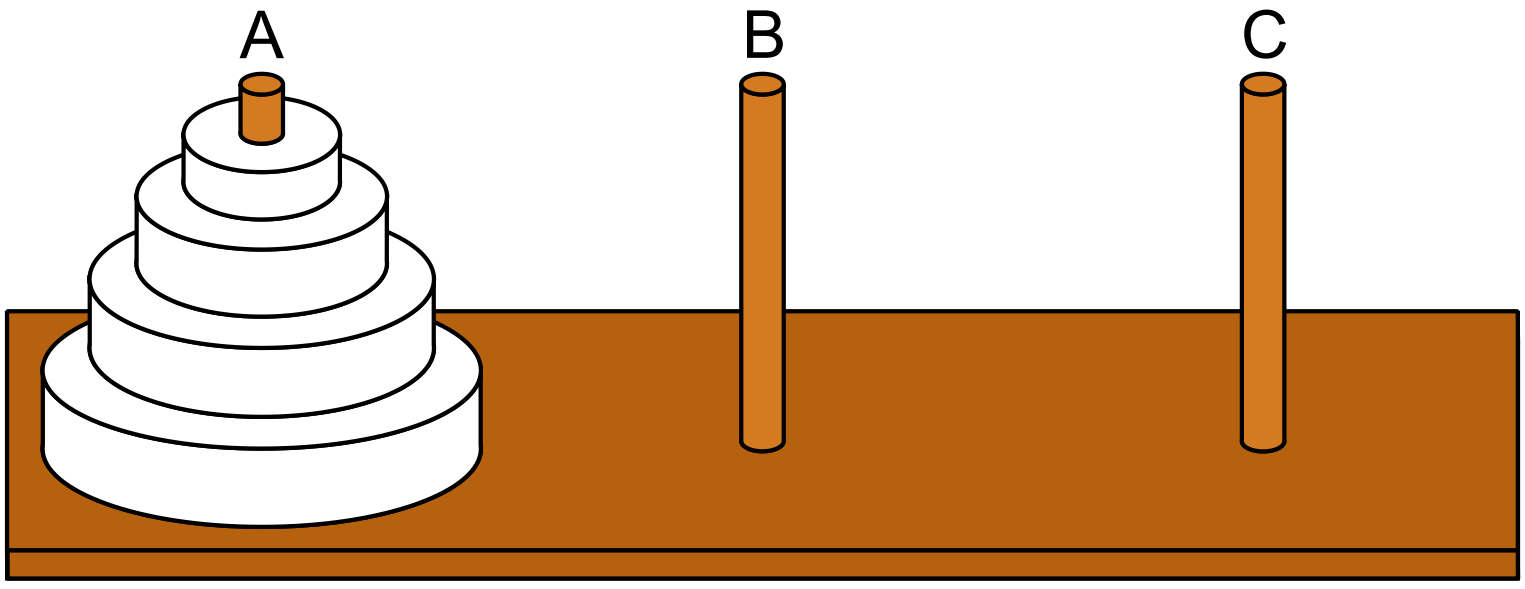
É possível resolver o problema da Torre de Hanói com n discos.

Torre de Hanói

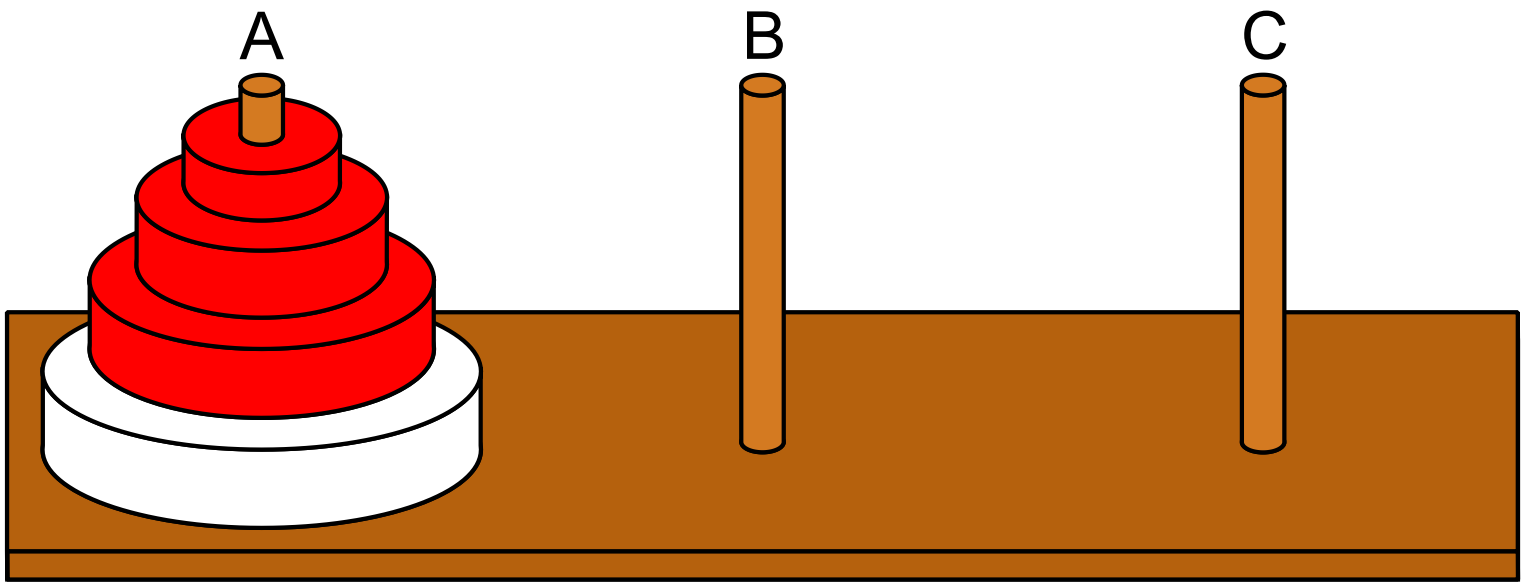
Prova

- Base da Indução: $n = 1$. Neste caso, temos apenas um disco. Basta mover este disco do pino A para o pino C.
- Hipótese de Indução: Sabemos como resolver o problema quando há $n - 1$ discos.
- Passo de Indução: Devemos resolver o problema para n discos assumindo que sabemos resolver o problema com $n - 1$ discos.
- Por hipótese de indução, sabemos mover os $n - 1$ primeiros discos do pino A para o pino B usando o pino C como auxiliar.
- Depois de movermos estes $n - 1$ discos, movemos o maior disco (que continua no pino A) para o pino C.
- Novamente, pela hipótese de indução, sabemos mover os $n - 1$ discos do pino B para o pino C usando o pino A como auxiliar.
- Com isso, temos uma solução para o caso em que há n discos.

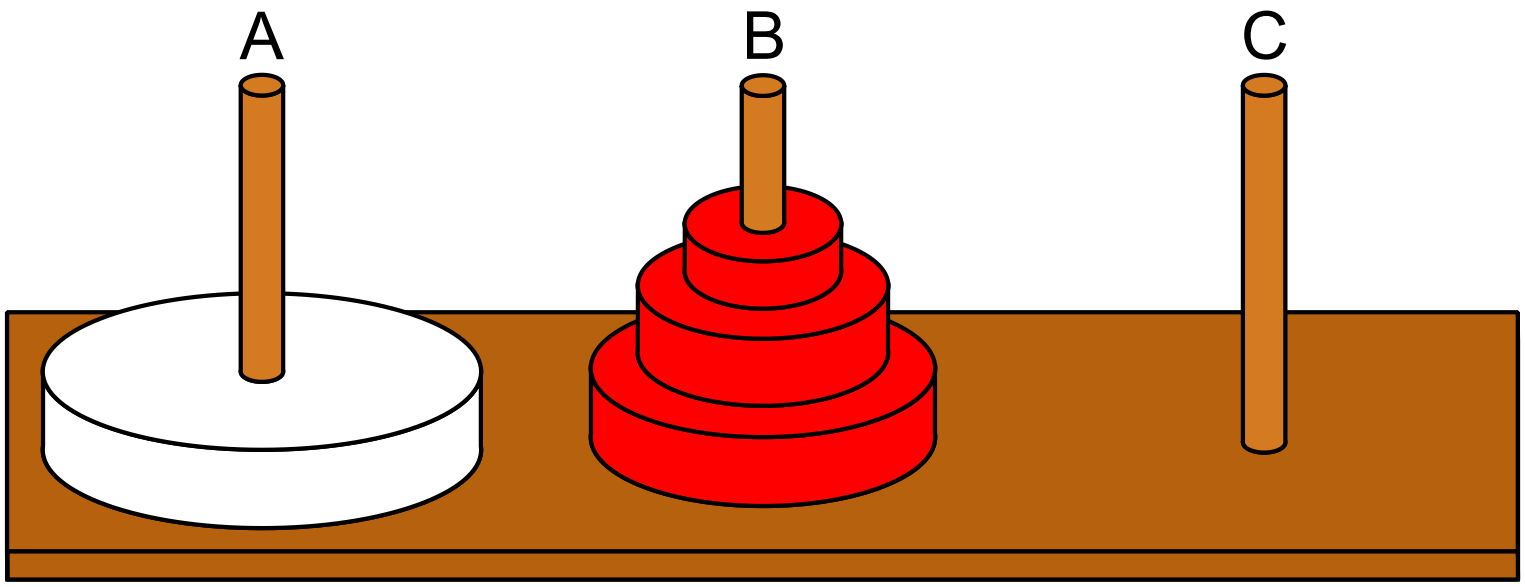
Torre de Hanói



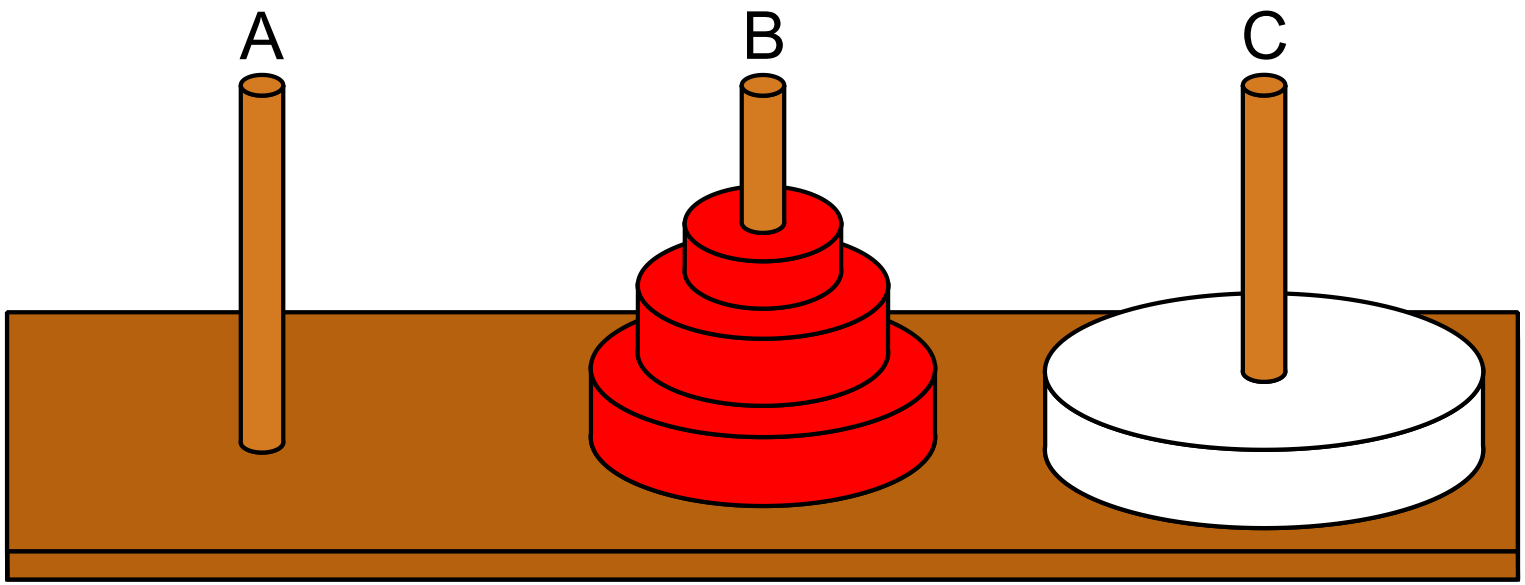
Torre de Hanói



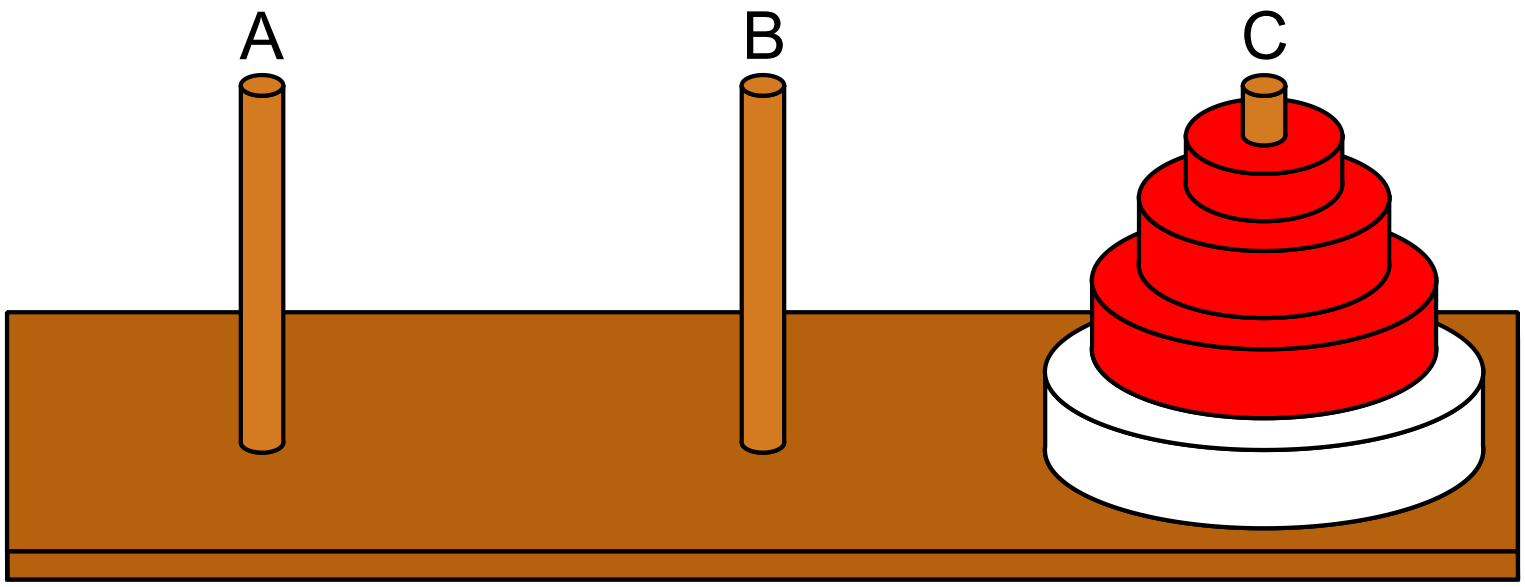
Torre de Hanói



Torre de Hanói



Torre de Hanói



Torre de Hanói

- Como solucionar o problema de forma recursiva:
  - Se $n = 1$ então mova o único disco de A para C.
  - Caso contrário ($n > 1$), desloque de forma recursiva os $n - 1$ primeiros discos de A para B, usando C como auxiliar.
  - Mova o último disco de A para C.
  - Mova, de forma recursiva, os $n - 1$ discos de B para C, usando A como auxiliar.

Torre de Hanói

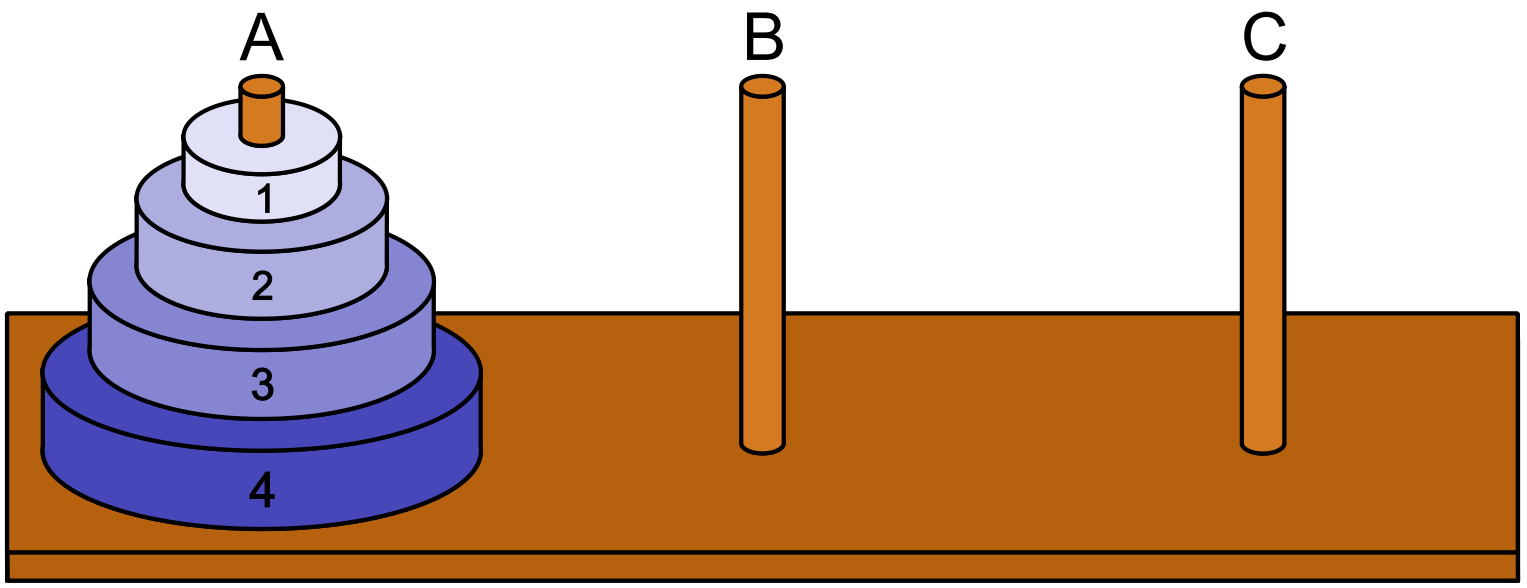
```
1 def hanoi(n, inicial, final, auxiliar):
2   s = "Mova o disco {} do pino {} para o pino {}"
3   if n == 1:
4     print(s.format(n, inicial, final))
5   else:
6     hanoi(n - 1, inicial, auxiliar, final)
7     print(s.format(n, inicial, final))
8     hanoi(n - 1, auxiliar, final, inicial)
```

Torre de Hanói

- Solução para `hanoi(4, "A", "C", "B")`:

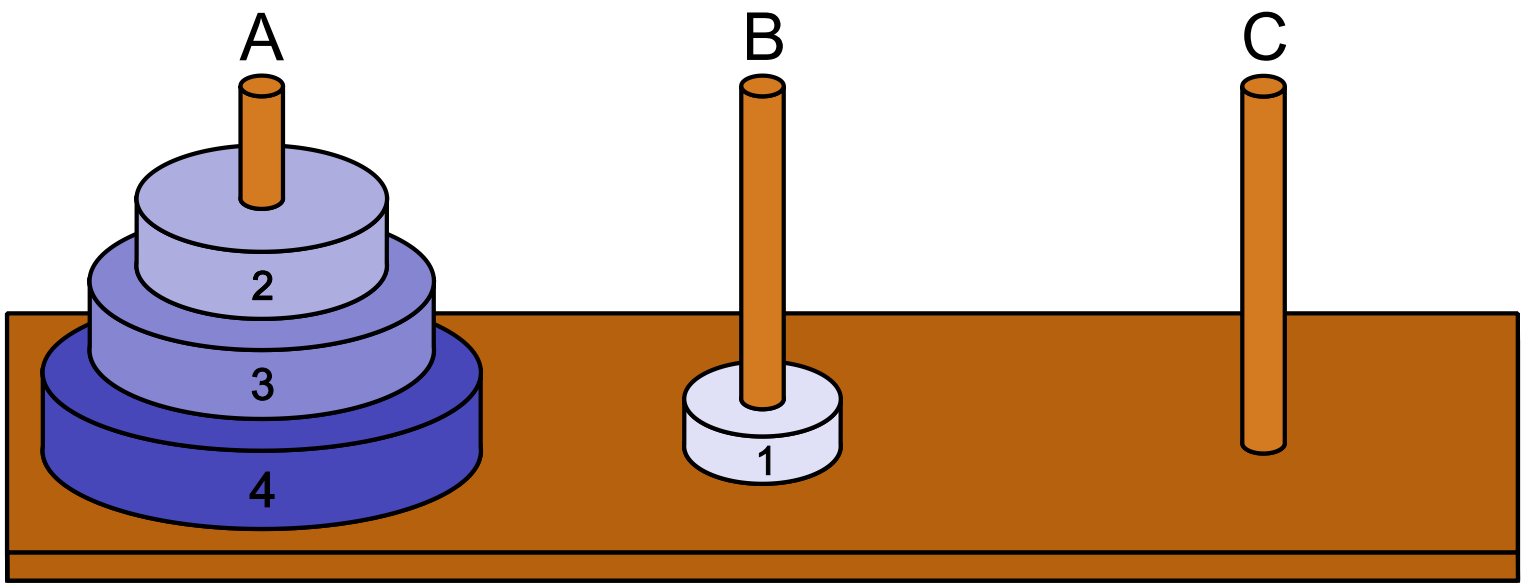
```
Mova o disco 1 do pino A para o pino B
Mova o disco 2 do pino A para o pino C
Mova o disco 1 do pino B para o pino C
Mova o disco 3 do pino A para o pino B
Mova o disco 1 do pino C para o pino A
Mova o disco 2 do pino C para o pino B
Mova o disco 1 do pino A para o pino B
Mova o disco 4 do pino A para o pino C
Mova o disco 1 do pino B para o pino C
Mova o disco 2 do pino B para o pino A
Mova o disco 1 do pino C para o pino A
Mova o disco 3 do pino B para o pino C
Mova o disco 1 do pino A para o pino B
Mova o disco 2 do pino A para o pino C
Mova o disco 1 do pino B para o pino C
```

Torre de Hanói



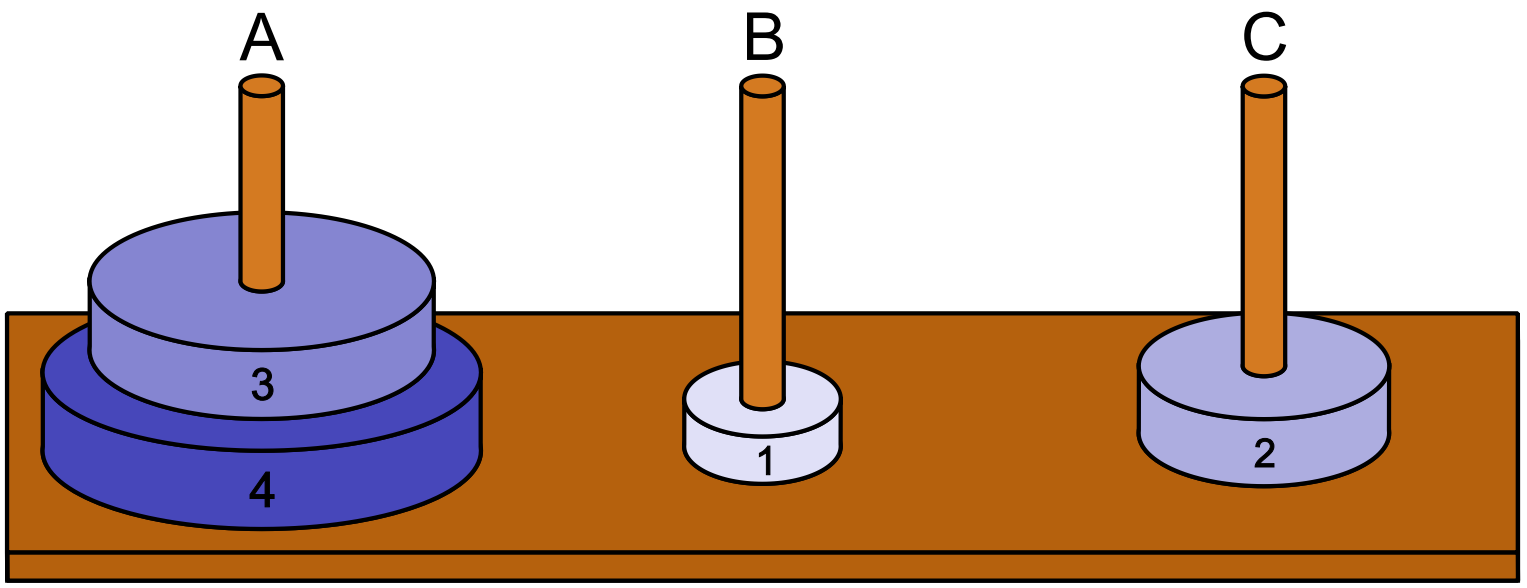

```
1 Configuração Inicial
```

Torre de Hanói



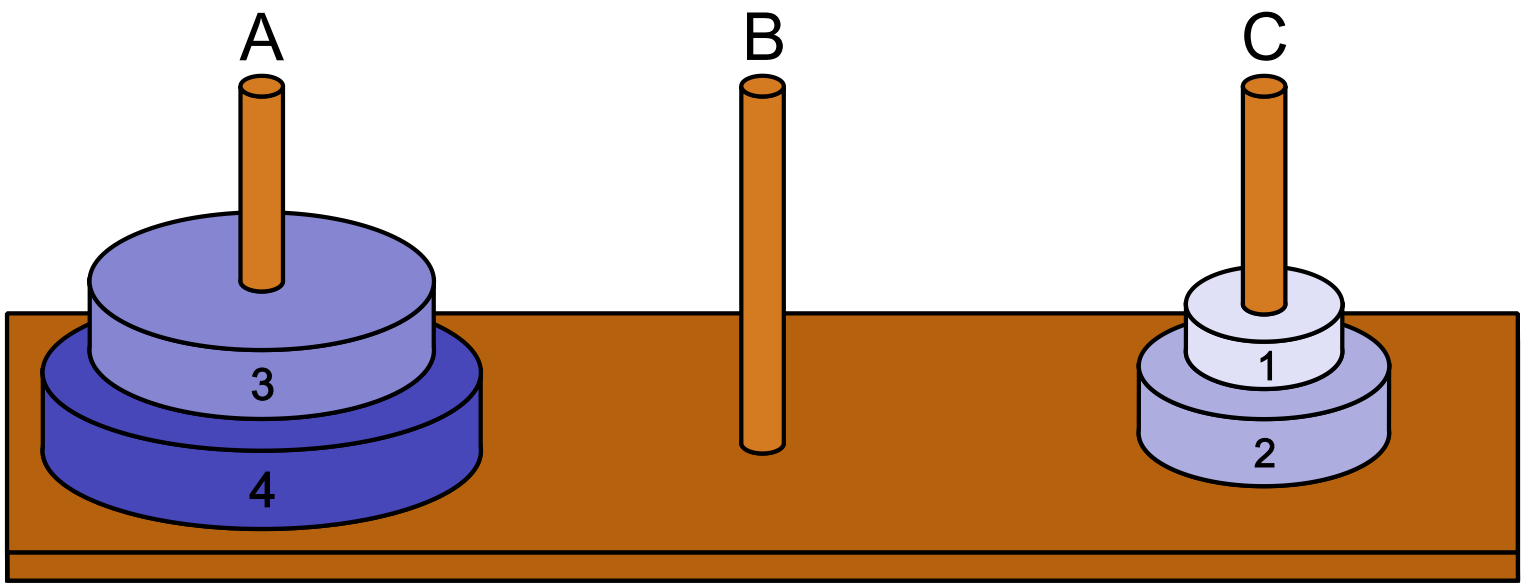

```
1  Mova o disco 1 do pino A para o pino B
```

Torre de Hanói



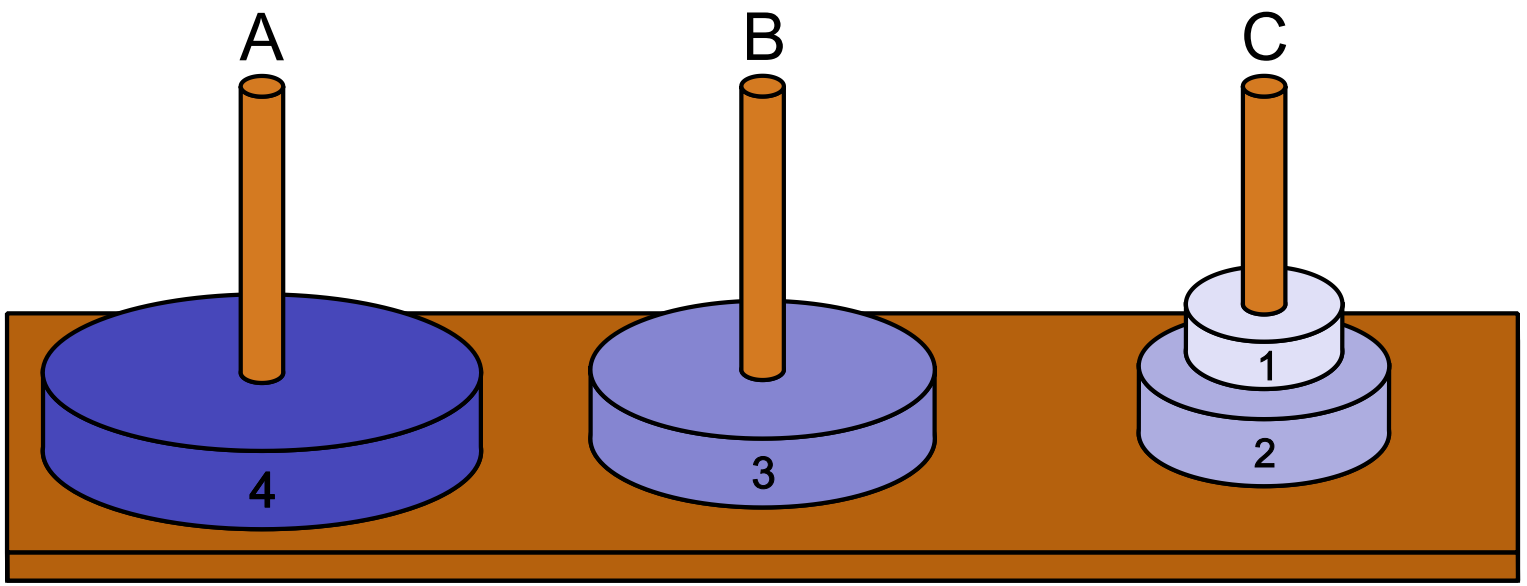

```
1  Mova o disco 2 do pino A para o pino C
```

Torre de Hanói



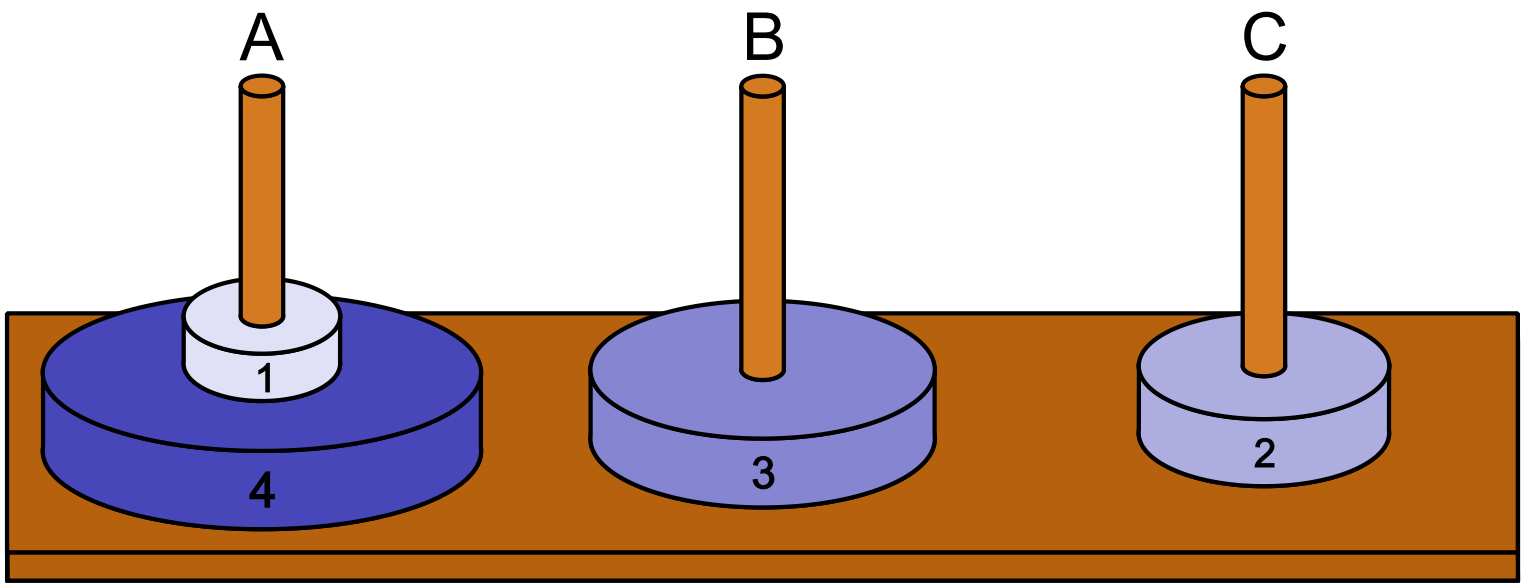

```
1  Mova o disco 1 do pino B para o pino C
```

Torre de Hanói



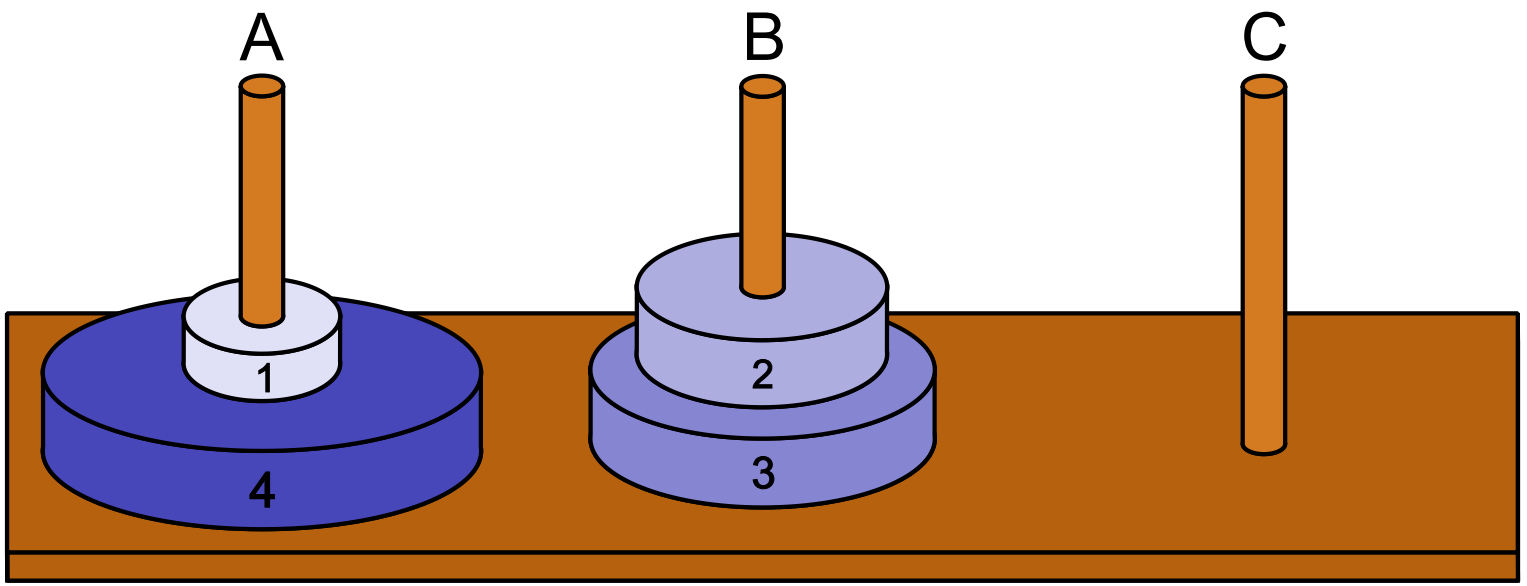

```
1  Mova o disco 3 do pino A para o pino B
```

Torre de Hanói



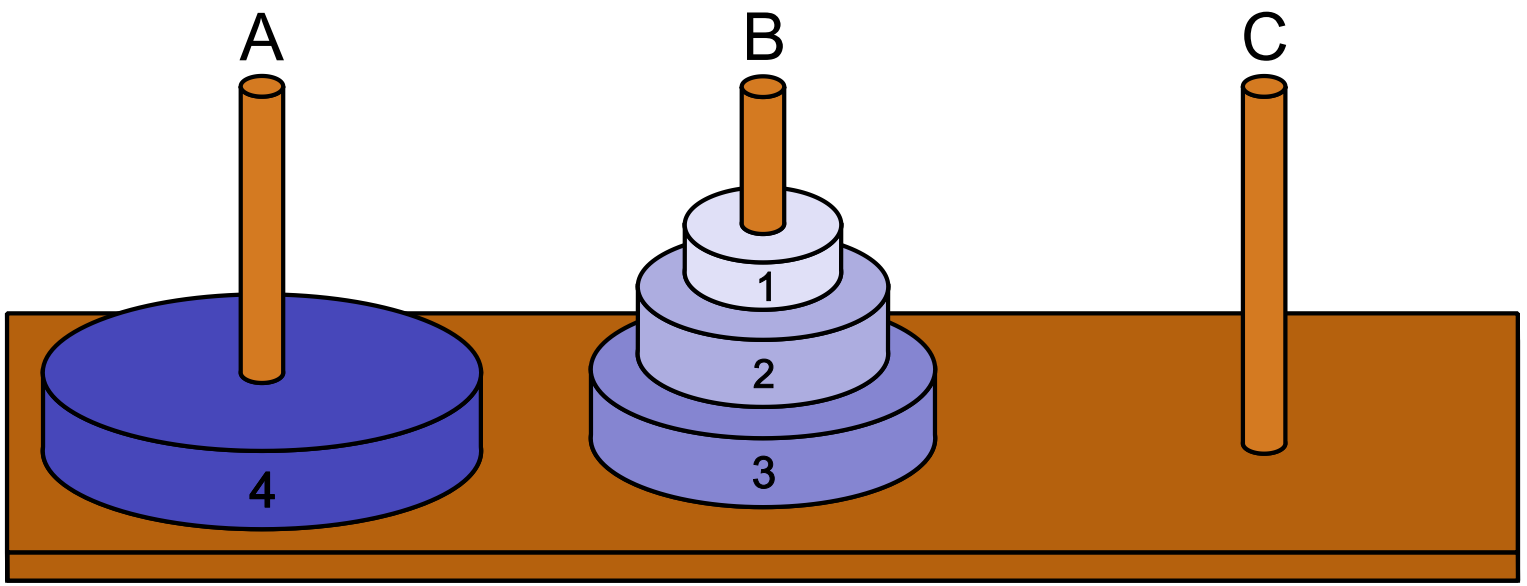

```
1  Mova o disco 1 do pino C para o pino A
```

Torre de Hanói



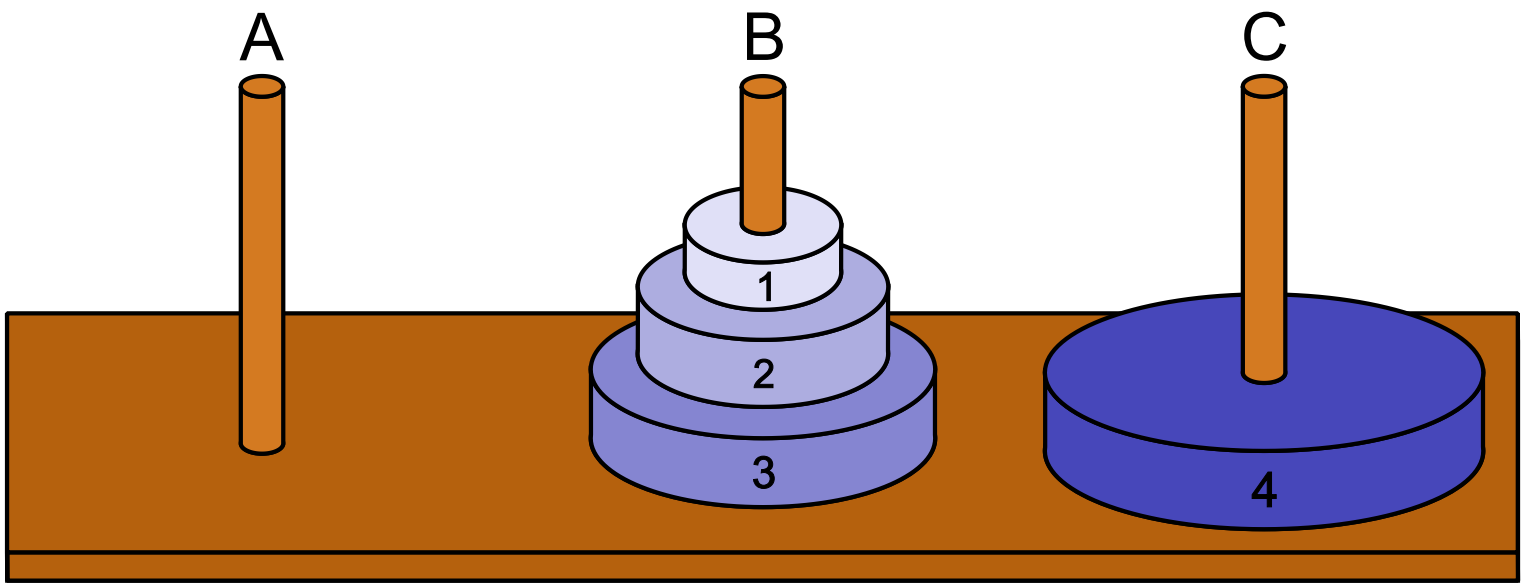

```
1  Mova o disco 2 do pino C para o pino B
```

Torre de Hanói



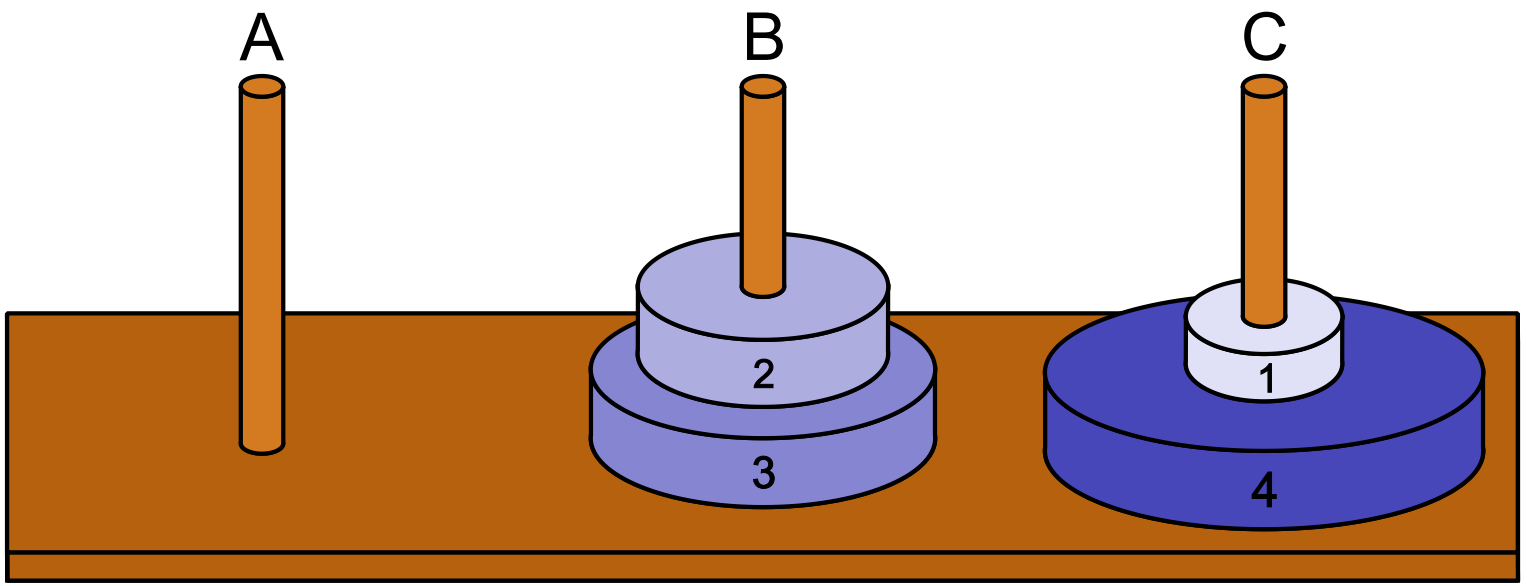

```
1  Mova o disco 1 do pino A para o pino B
```

Torre de Hanói



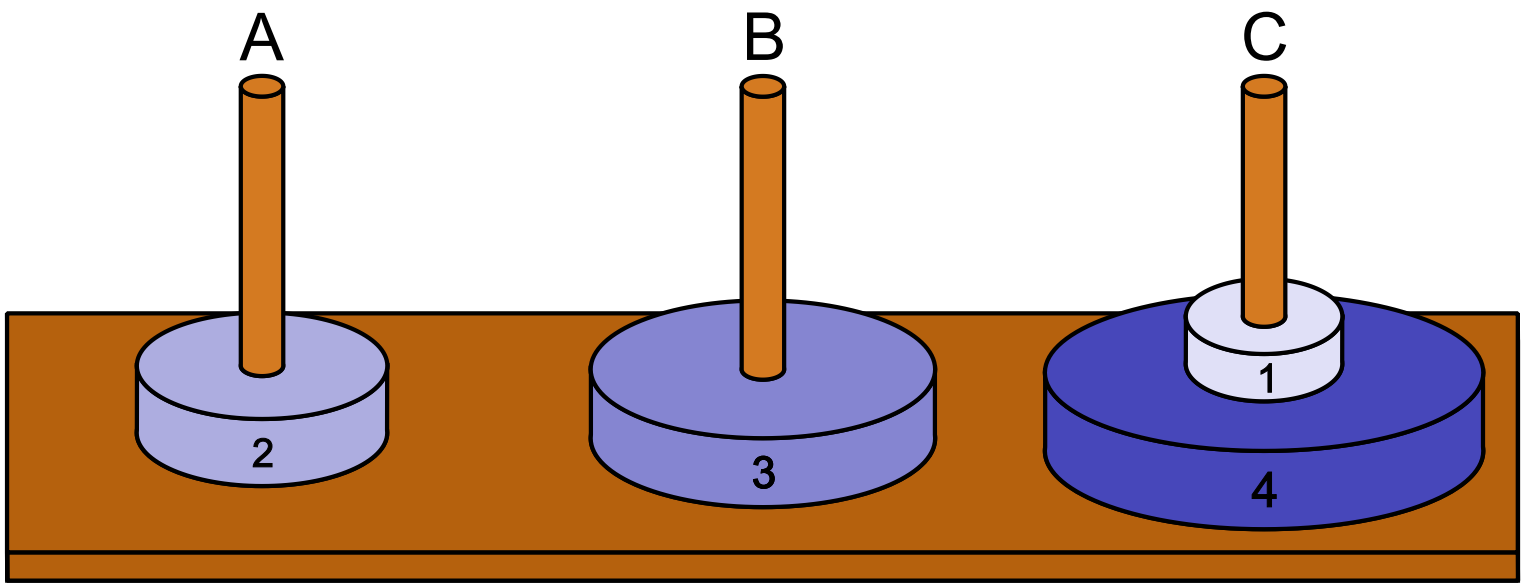

```
1  Mova o disco 4 do pino A para o pino C
```

Torre de Hanói



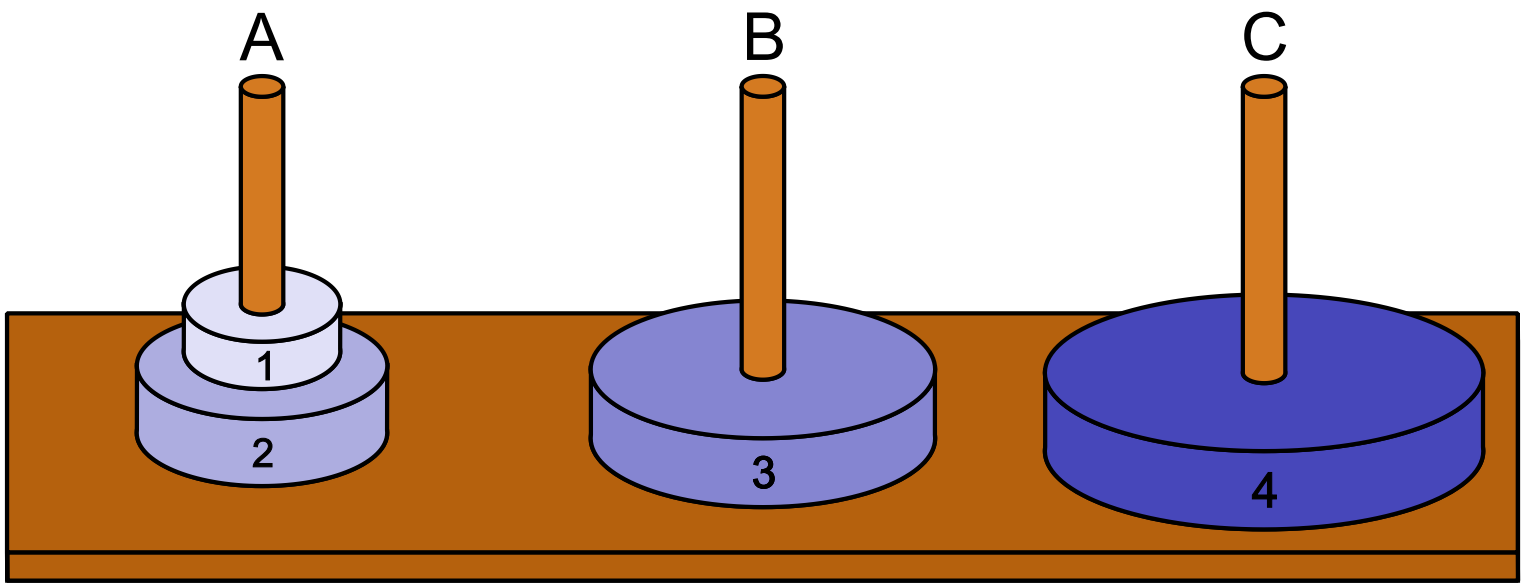

```
1  Mova o disco 1 do pino B para o pino C
```

Torre de Hanói



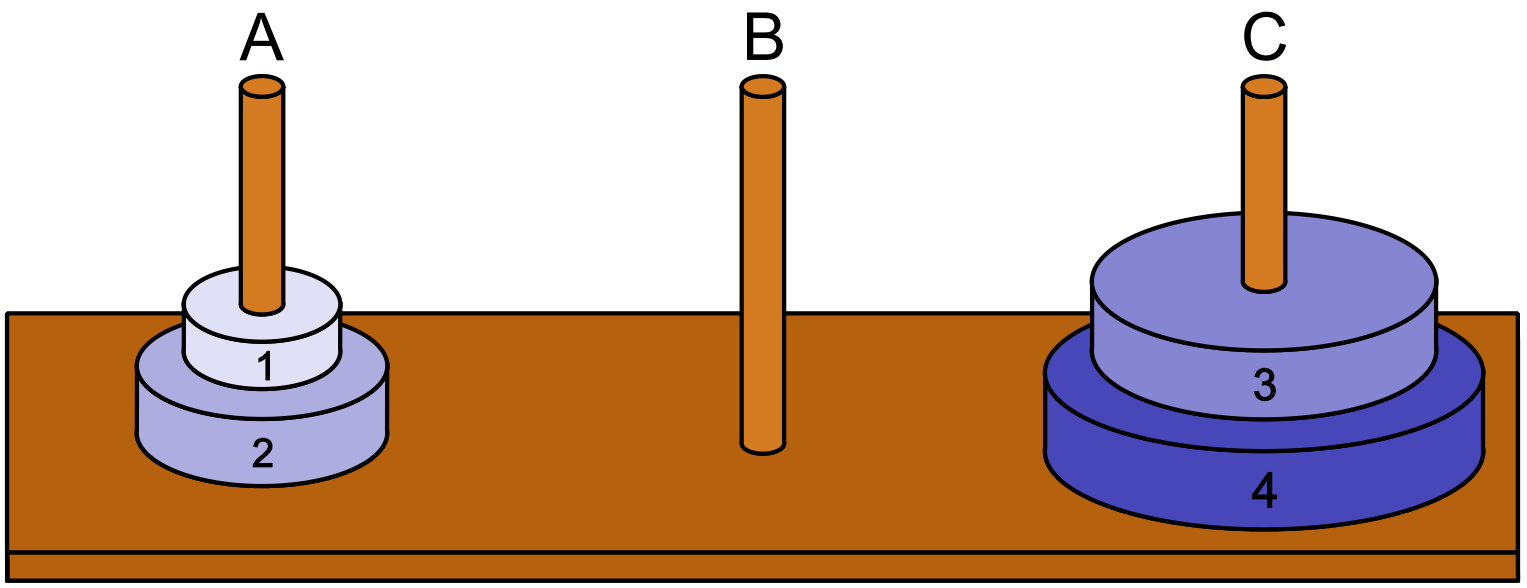

```
1  Mova o disco 2 do pino B para o pino A
```

Torre de Hanói



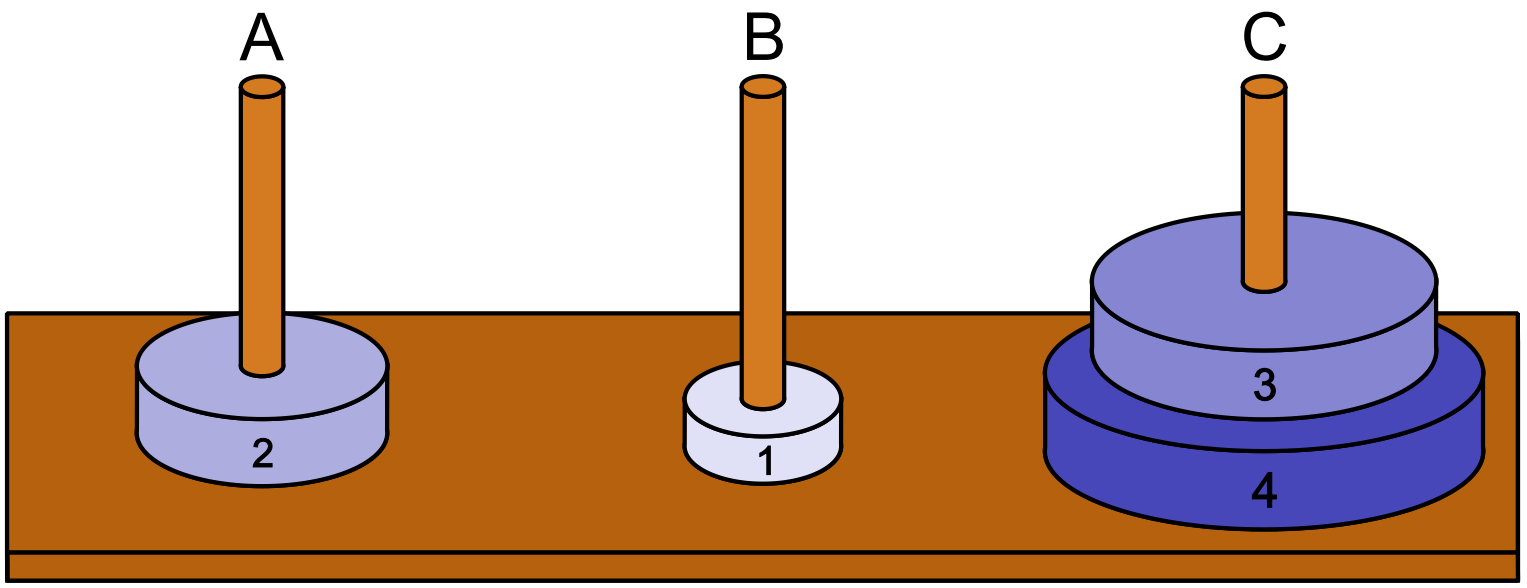

```
1  Mova o disco 1 do pino C para o pino A
```

Torre de Hanói



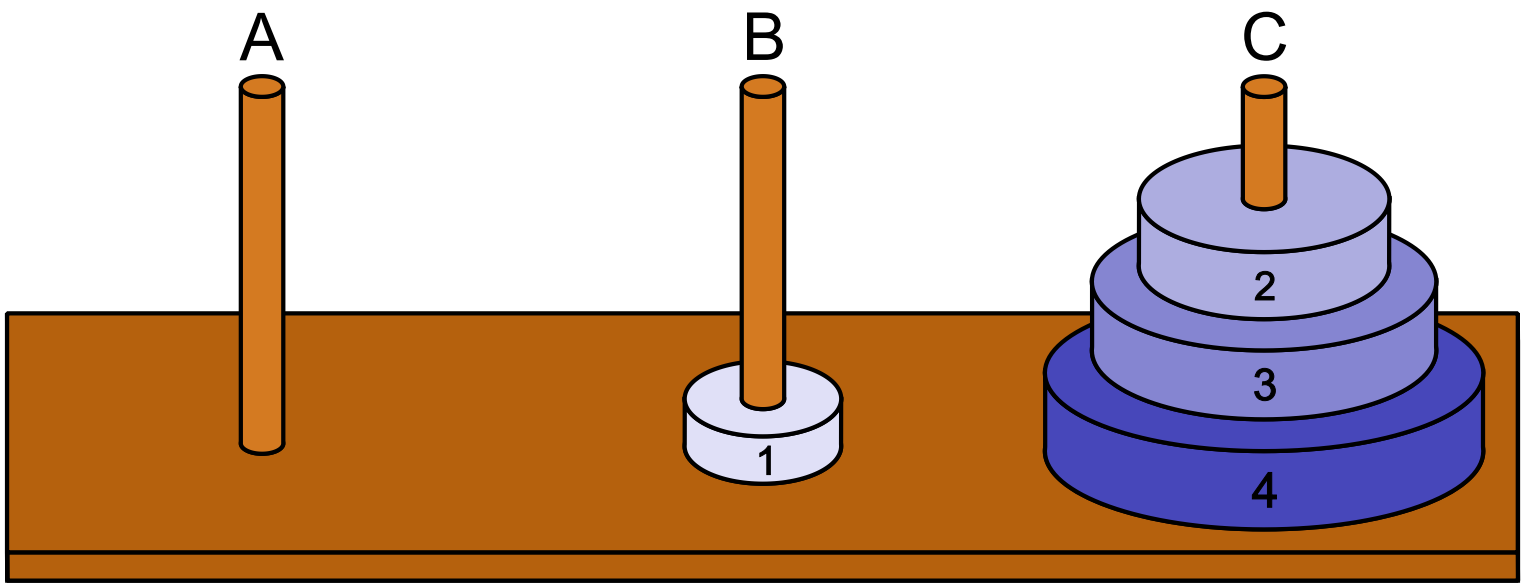

```
1  Mova o disco 3 do pino B para o pino C
```

Torre de Hanói



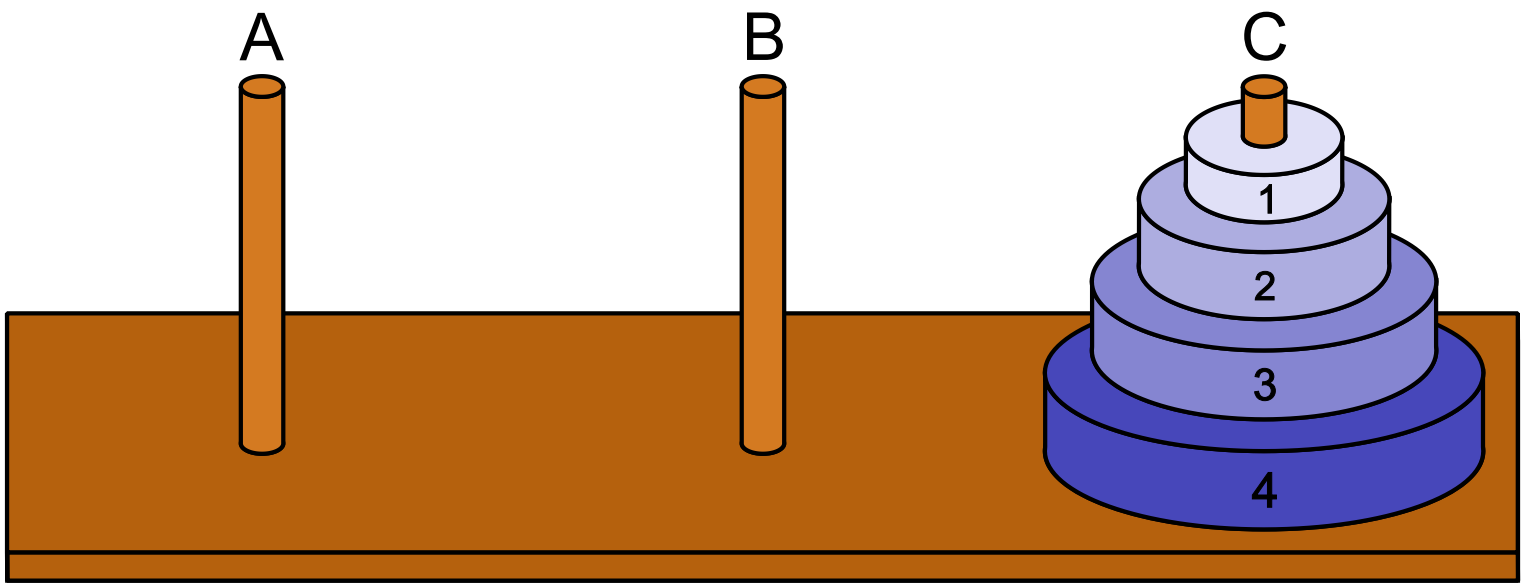

```
1  Mova o disco 1 do pino A para o pino B
```

Torre de Hanói




```
Mova o disco 2 do pino A para o pino C
```

Torre de Hanói




```
1  Mova o disco 1 do pino B para o pino C
```

Torre de Hanói

- Seja $T(n)$ o número de movimentos necessários para mover uma pilha de n discos.
- Claramente temos que:
  - $T(1) = 1$
  - $T(n) = 2T(n-1) + 1$
- O que nos permite deduzir que:
  - $T(2) = 2T(1) + 1 = 3$
  - $T(3) = 2T(2) + 1 = 7$
  - $T(4) = 2T(3) + 1 = 15$
  - $T(5) = 2T(4) + 1 = 31$
  - $\ldots$
  - $T(n) = 2^n - 1$
- No caso de 64 discos são necessários 18.446.744.073.709.551.615 movimentos ou, aproximadamente, 585 bilhões de anos, se cada movimento puder ser feito em um segundo.

Exercícios

Exercícios

1. Escreva uma função recursiva que, dado um número inteiro positivo n, imprima a representação binária de n.
2. Escreva uma função recursiva que, dada uma string `s` e uma caractere `c`, conte o número de ocorrências do caractere `c` na string `s`.
3. Escreva uma função recursiva que, dado uma lista `l` de n números inteiros ordenados ($n \geq 1$) e um inteiro `x`, retorne, usando uma busca binária, o índice de `l` na lista ou o valor -1, caso `x` não pertença à lista.
4. Escreva versões recursivas das funções `indiceMenor` e `selectionSort`.

   Importante: todas as funções acima devem ser implementadas sem qualquer comando de repetição (`for`, `while`, etc).

Exercício 1 - Resposta

```
1 def binario(n):
2   if n <= 1:
3     return str(n)
4   else:
5     return binario(n // 2) + str(n % 2)
```

Exercício 2 - Resposta

```
1 def conta(s, c):
2   if len(s) == 0:
3     return 0
4   if s[0] == c:
5     return 1 + conta(s[1:], c)
6   else:
7     return conta(s[1:], c)
```

Exercício 2 - Resposta

```
1 def conta(s, c):
2   if s == "":
3     return 0
4   if s[0] == c:
5     return 1 + conta(s[1:], c)
6   else:
7     return conta(s[1:], c)
```

Exercício 2 - Resposta

```
1 def conta(s, c):
2   if s == "":
3     return 0
4
5   return (s[0] == c) + conta(s[1:], c)
```

Exercício 3 - Resposta

```python
def busca_binaria(lista, inicio, fim, chave):
  if inicio > fim:
    return -1

  meio = (inicio + fim) // 2
  if lista[meio] == chave:
    return meio

  if lista[meio] > chave:
    return busca_binaria(lista, inicio, meio - 1, chave)
  else:
    return busca_binaria(lista, meio + 1, fim, chave)
```

Exercício 4 - Resposta

```
1 def indiceMenor(lista, inicio):
2   if inicio == len(lista) - 1:
3     return inicio
4
5   aux = indiceMenor(lista, inicio + 1)
6   if lista[inicio] < lista[aux]:
7     return inicio
8   else:
9     return aux
```

Exercício 4 - Resposta

```
1 def selectionSort(lista, i):
2   if i == len(lista) - 1:
3     return
4   else:
5     minimo = indiceMenor(lista, i)
6     (lista[i], lista[minimo]) = (lista[minimo], lista[i])
7     selectionSort(lista, i + 1)
```

Exercício 4 - Resposta

```
1 def selectionSort(lista, i):
2   if i < len(lista) - 1:
3     minimo = indiceMenor(lista, i)
4     (lista[i], lista[minimo]) = (lista[minimo], lista[i])
5     selectionSort(lista, i + 1)
```