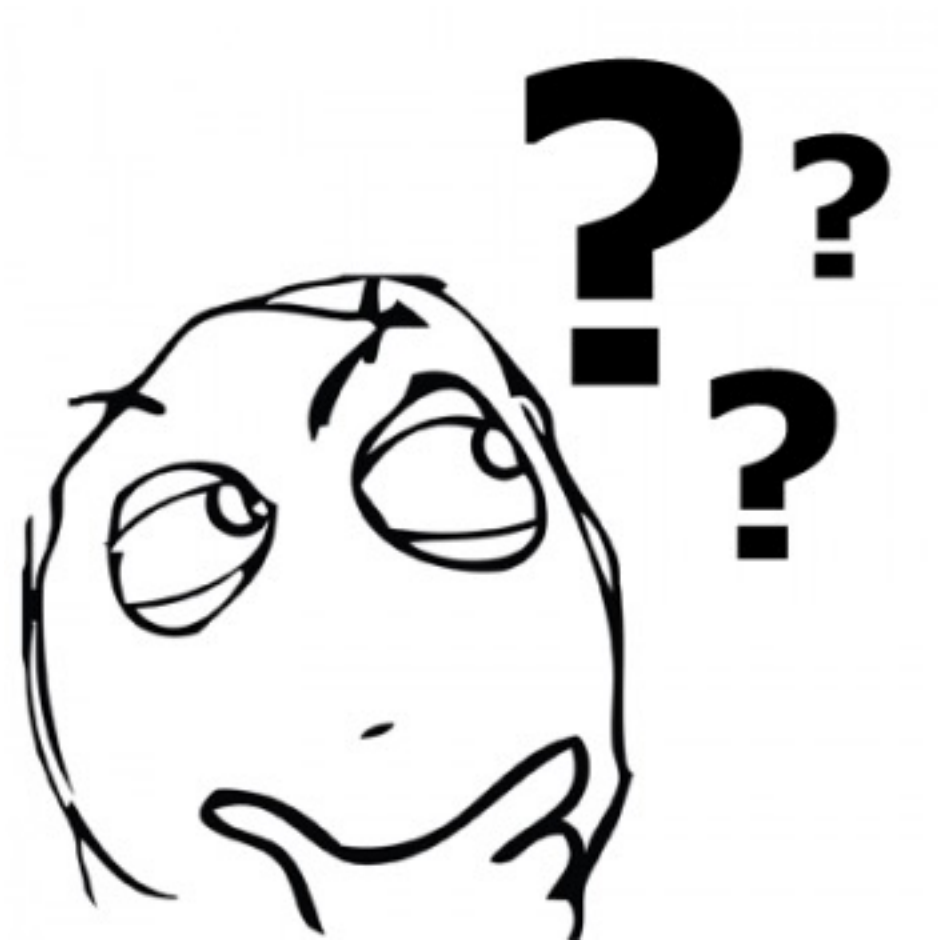
INSTITUTO FEDERAL
DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA
São Paulo

Estrutura de Dados

Aula 3

GUSTAVO FORTUNATO PUGA

Recursividade



Recursividade

- Recursividade é uma **técnica de programação** na qual uma **função (ou método)** pode **chamar a si mes**ma.

- Ela é bastante usada para **simplificar a resolução de problemas que demandariam muitos passos**.

Recursividade

https://www.youtube.com/watch?v=NKymAD4pJZI

Recursividade

O fatorial é um número **natural inteiro positivo**, o qual é representado por n!

O cálculo do fatorial de um número é obtido pela multiplicação desse número por todos os seus antecessores até chegar ao número 1.

Assim: 3! = 3 * 2 * 1 = 6

4! = 4 * 3 * 2 * 1 = 24

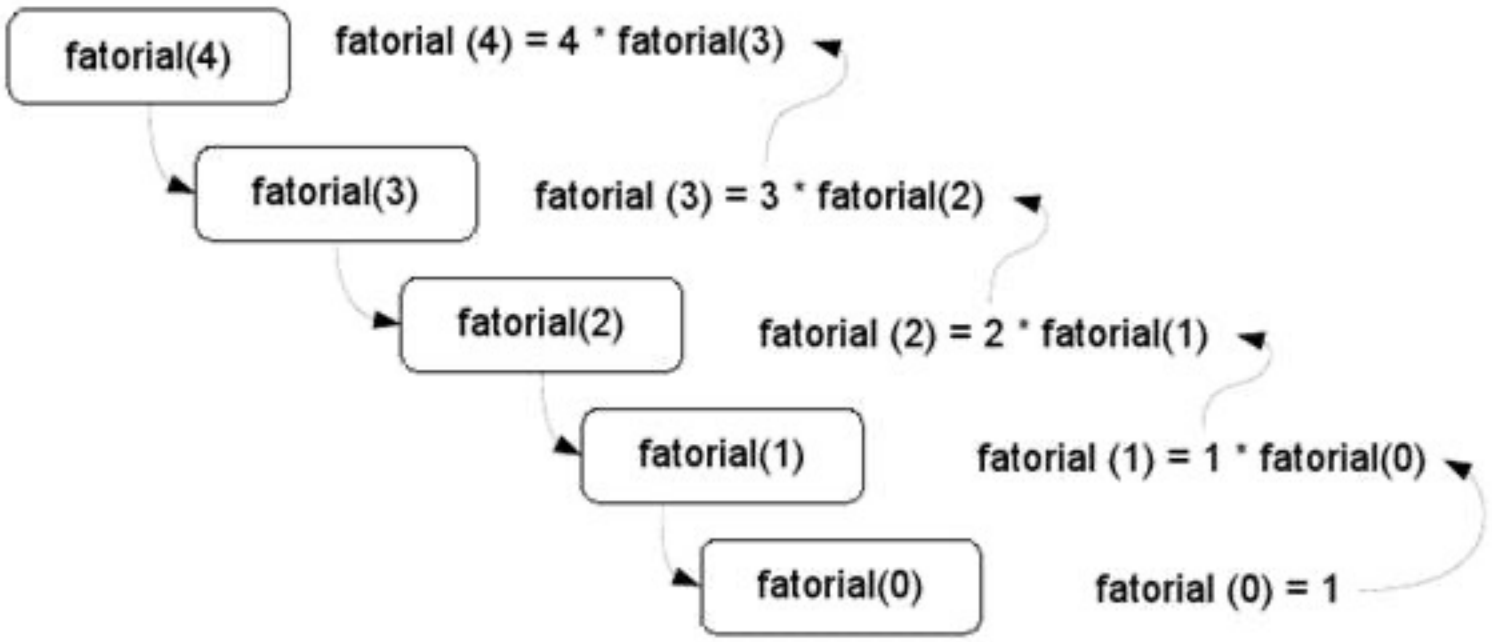
5! = 5 * 4 * 3 * 2 * 1 = 120

Note que, por definição:

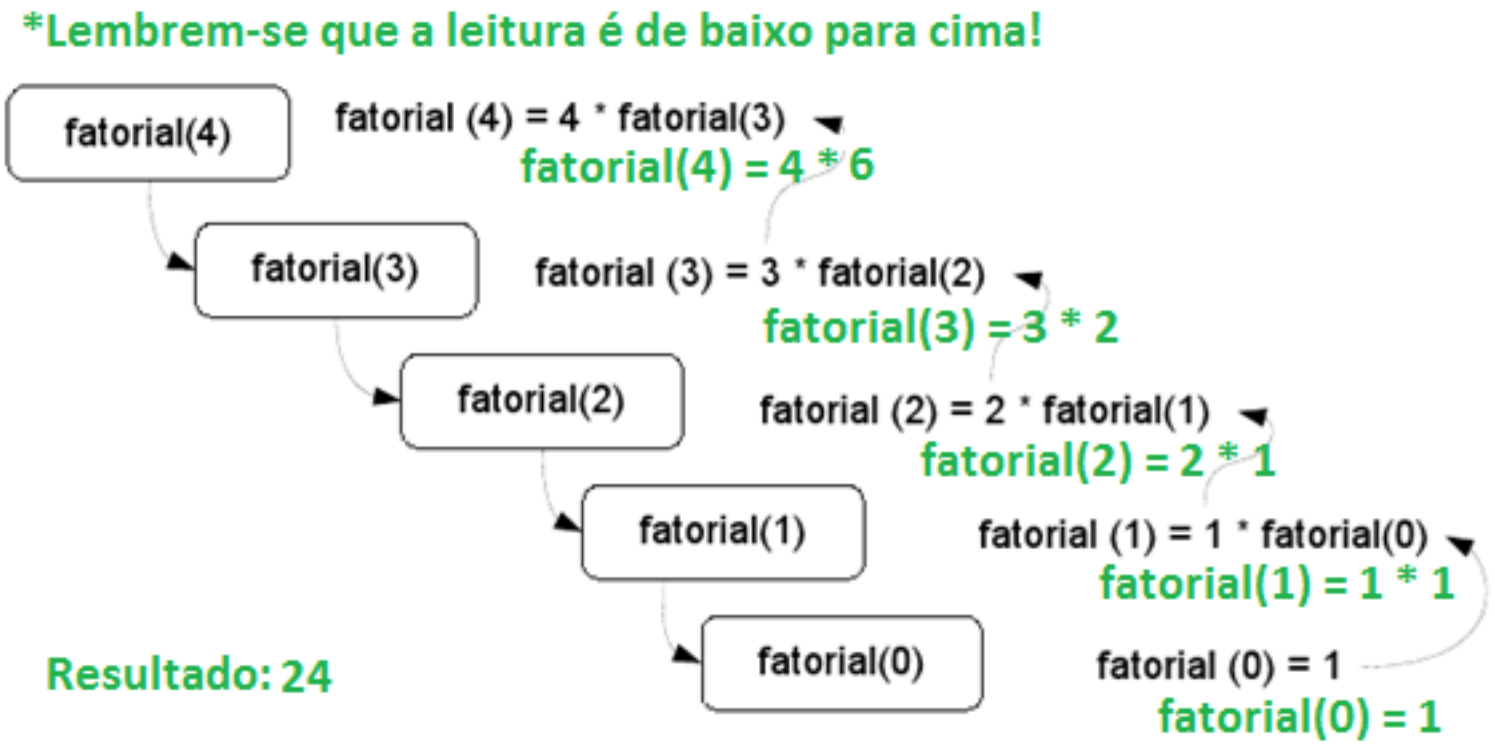
1! = 1

0! = 1

Recursividade



Recursividade



Recursividade

Todo algoritmo recursivo tem duas partes:

- <u>Caso base</u>: normalmente uma instância pequena, de solução trivial, que para a recursão.
  - Por exemplo: 1! = 1

- <u>Caso recursivo</u>: instância que seria difícil resolver diretamente, mas que resolvemos chamando uma ou mais instâncias menores do problema recursivamente, e tratando a resposta.
  - Por exemplo: 5! = 5 * 4!

Algoritmo

```
int  fatorialRecursivo (int n) {

   if (n == 0 || n == 1){
      /* CASO BASE */
      return 1;
   }

   else {
      /* CASO RECURSIVO */
      return ( n * fatorial_recursivo (n-1) );
   }

}
```

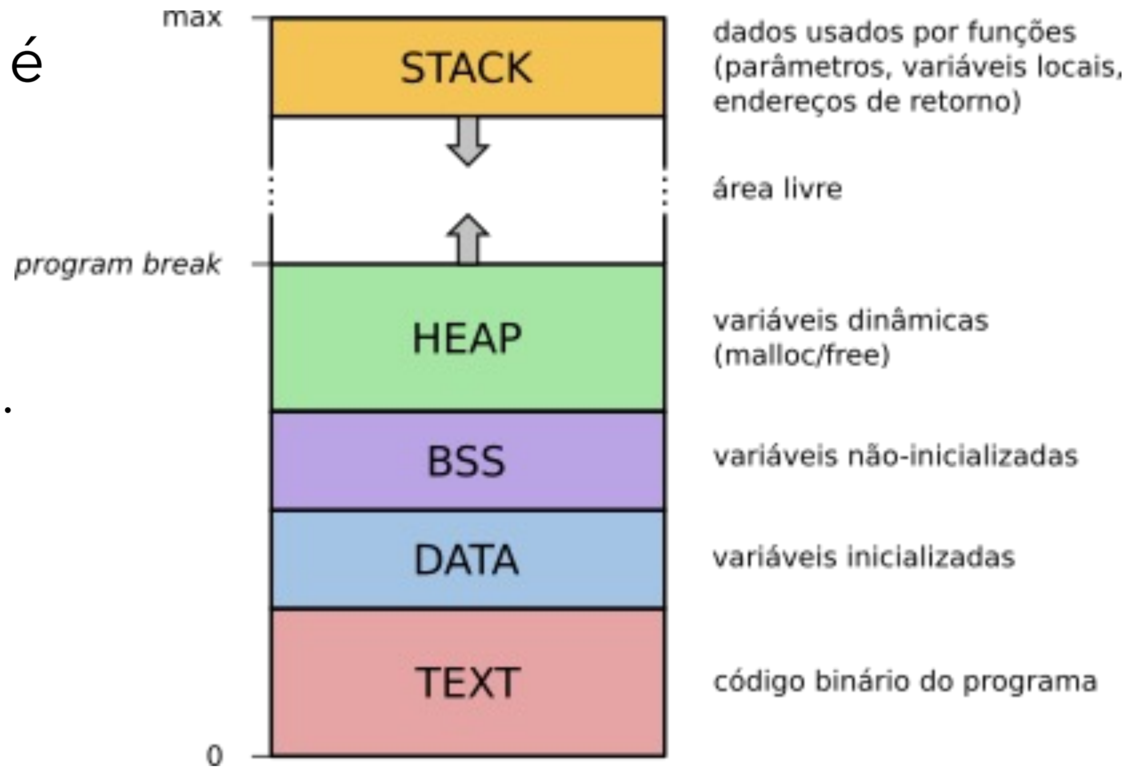
Implementação

```java
public class Fatorial{
    //Método recursivo para cálculo de fatorial
    public int fatorialRecursivo(int num){
        //se n é igual a 0, retorna 1
        if (num == 0)
            return 1;
        //Caso contrário, o método recursivo é chamado:
        return num*fatorialRecursivo(num-1);
    }
}
```

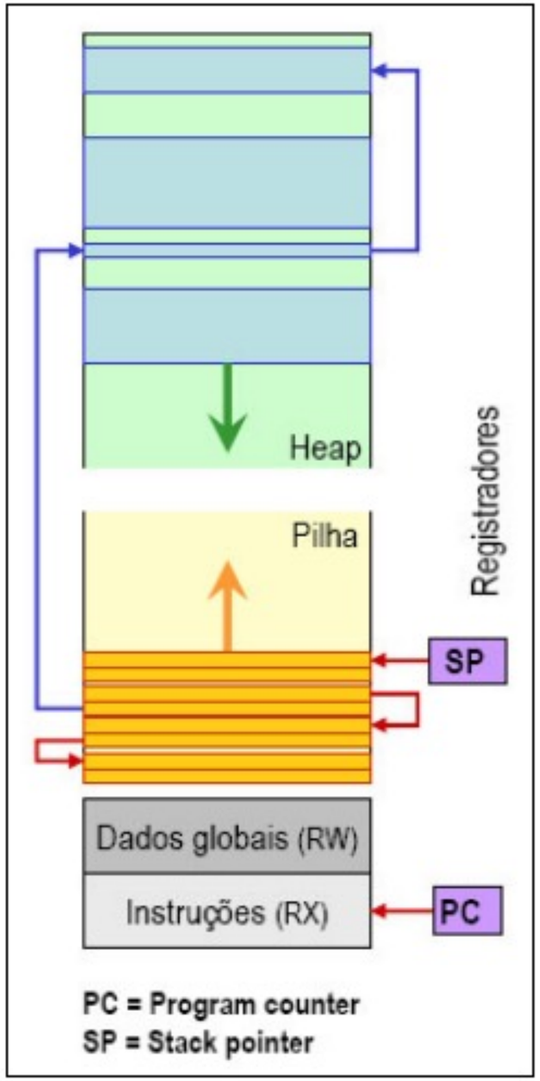
Gerenciamento de memória

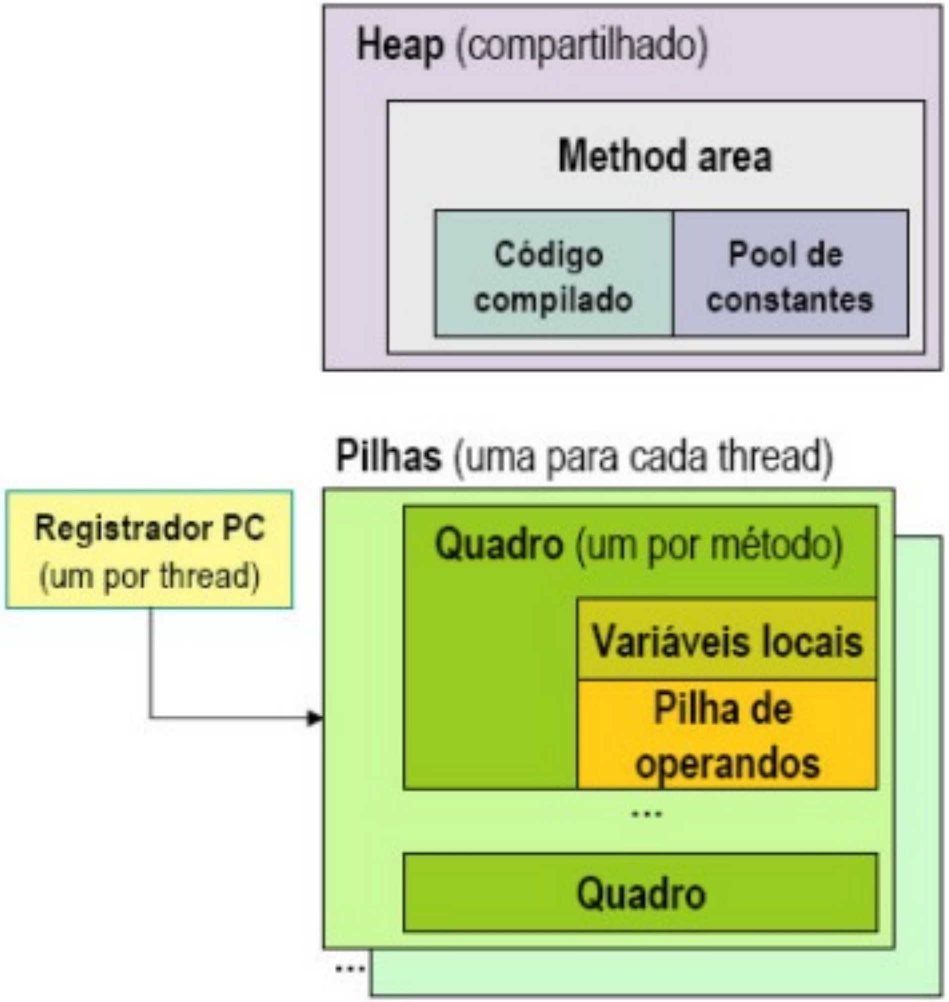
A memória de um sistema computacional é dividida em três partes:

- Espaço Estático: contém o código do programa.
- Heap: para alocação dinâmica de memória.
- Pilha: para execução de funções.

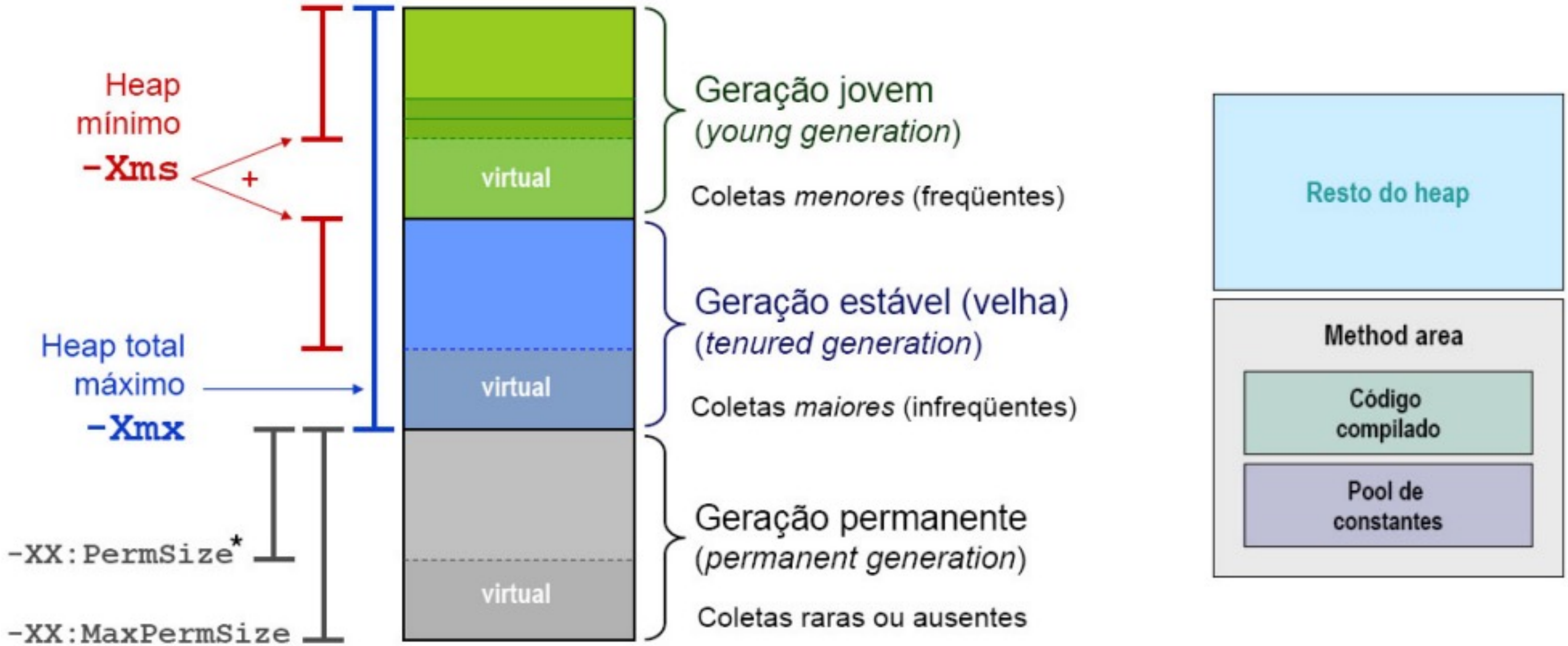


Gerenciamento de memória





Gerenciamento de memória

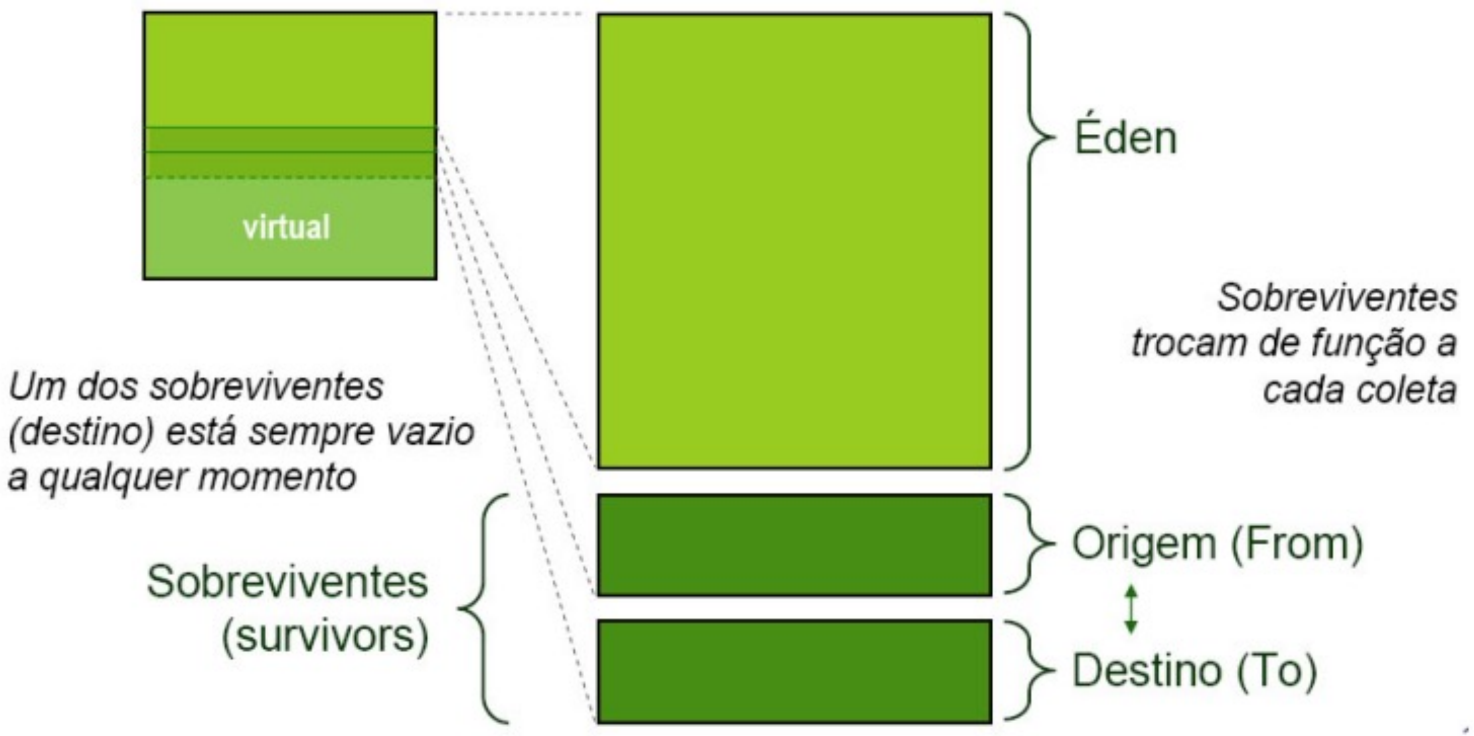


Gerenciamento de memória

A parte maior é chamada de Éden.

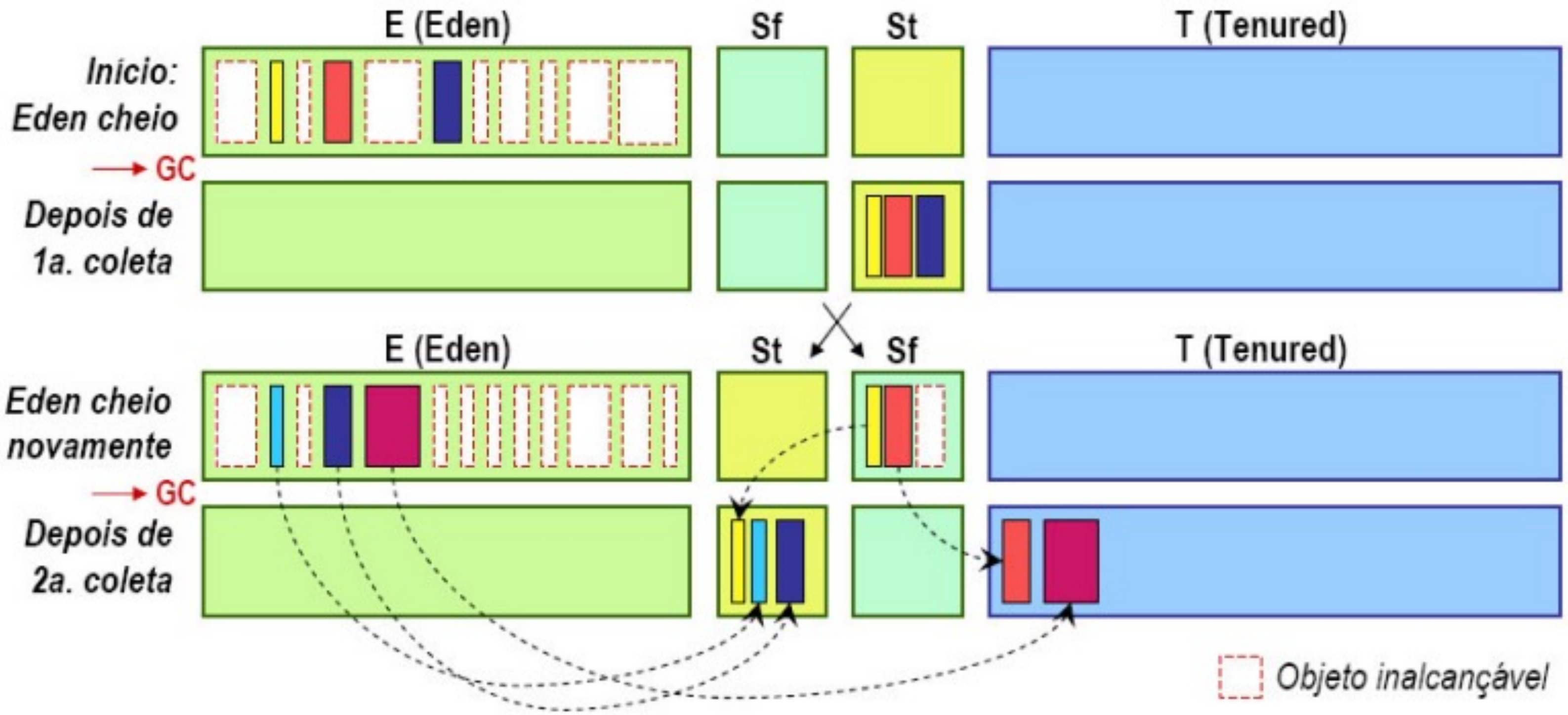
É onde novos objetos são criados.

No algoritmo de cópia, o Éden é sempre origem e nunca muda de papel. Sobreviventes de uma coleta esvaziam o Éden e são copiados para as áreas menores, chamadas de espaços sobreviventes.



Anatomia da geração jovem na máquina virtual HotSpot

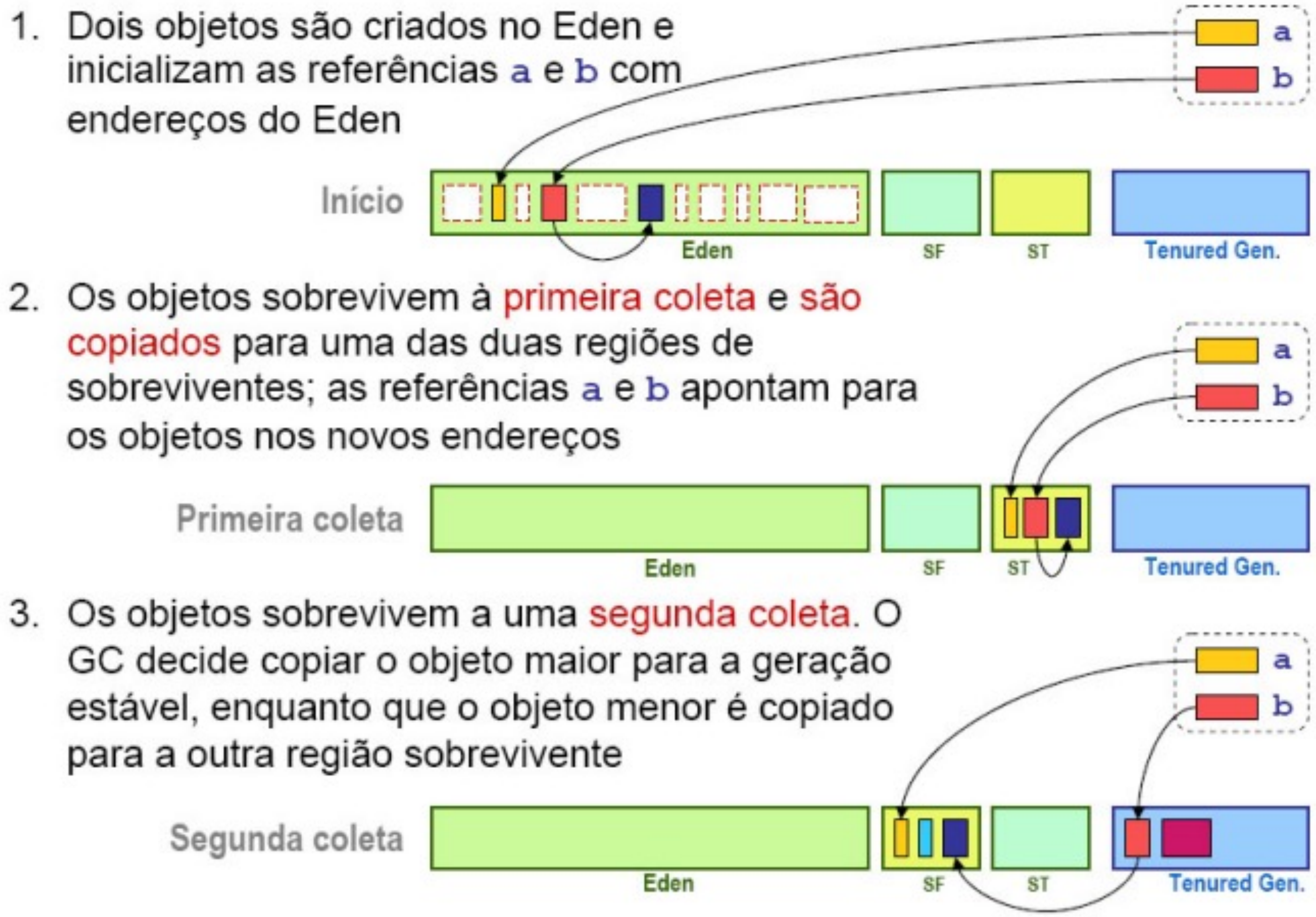
Gerenciamento de memória



Quando o Éden enche, coletor de lixo copia objetos alcançáveis do Éden (E) para sobrevivente To (St) – sempre esvazia o Éden; do sobrevivente From (Sf) para St – sempre esvazia o Sf; de Sf para a geração estável (T) (dependente de algoritmo); do Éden ou Sf para T (se não cabe em St)

Gerenciamento de memória

Como Java não realiza aritmética de ponteiros, não existe risco algum das mudanças de endereço causarem algum defeito em um programa.



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.

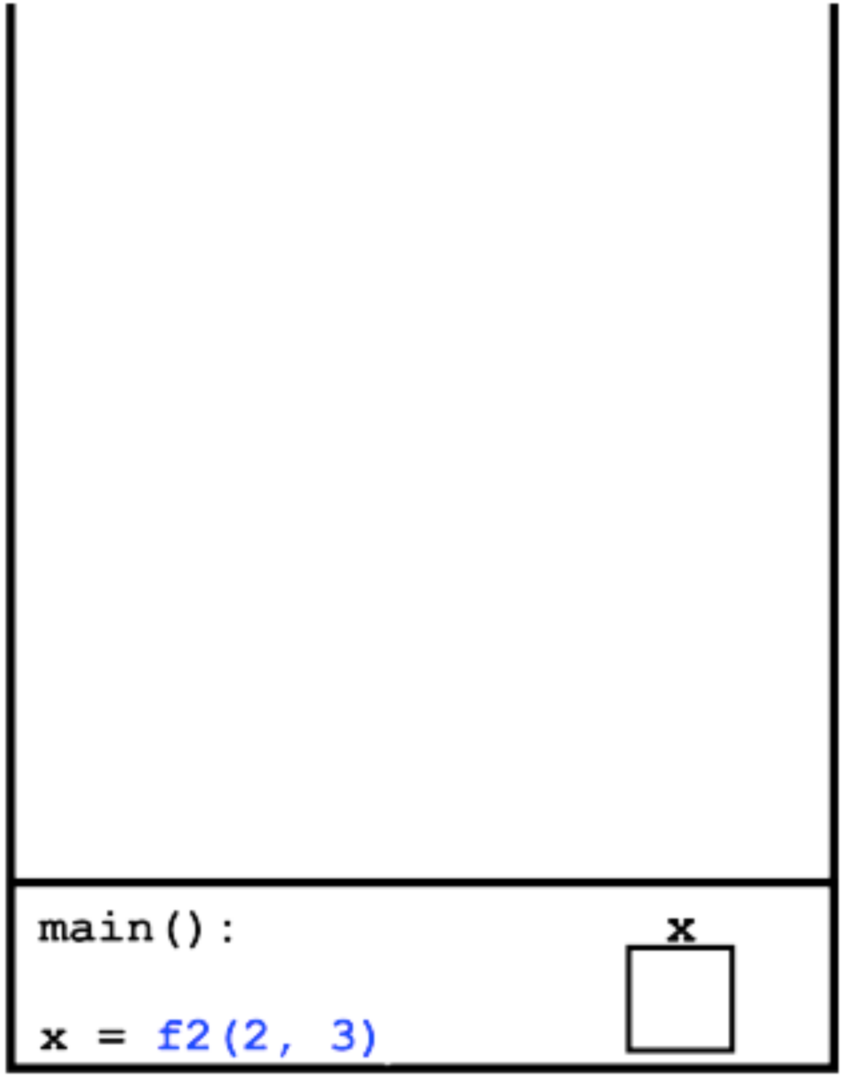
Gerenciamento de memória

```python
def f1(a, b):
  c = a - b
  return (a + b + c)

def f2(a, b):
  c = f1(b, a)
  return (b + c - a)

def main():
  x = f2(2, 3)
  return 0

main()
```



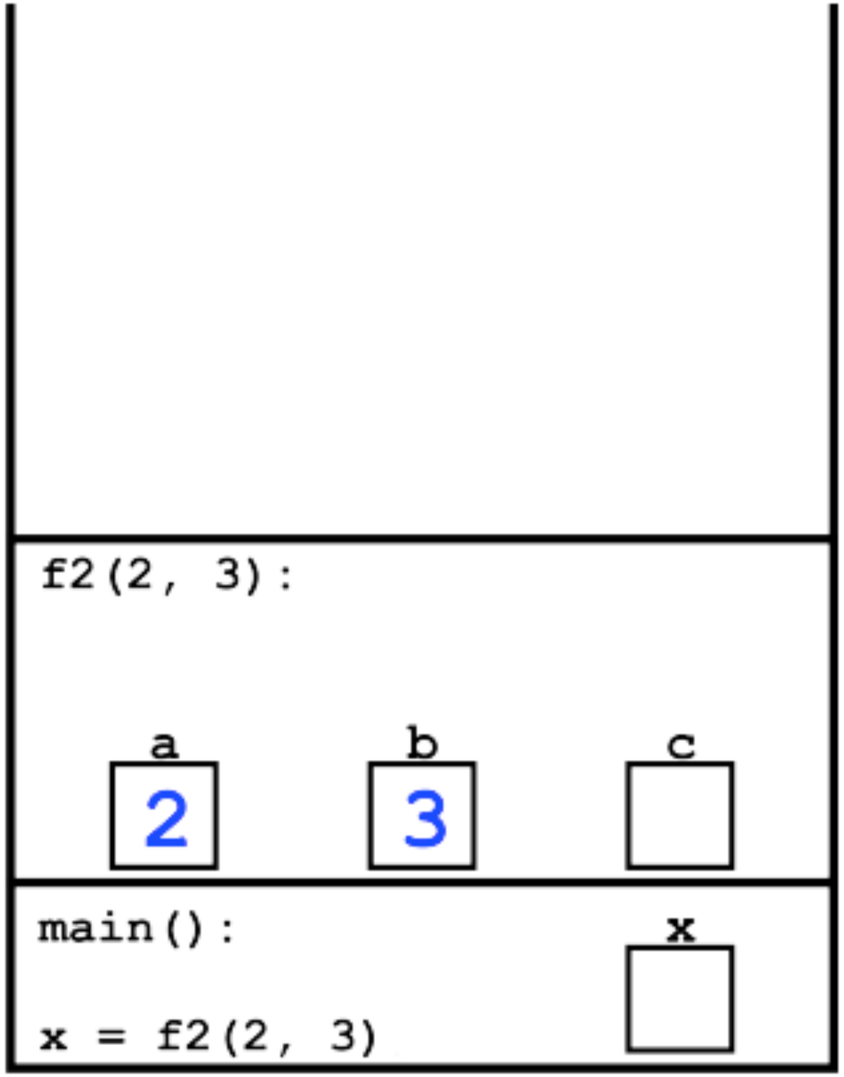
Gerenciamento de memória

```
def f1(a, b):
  c = a - b
  return (a + b + c)

def f2(a, b):
  c = f1(b, a)
  return (b + c - a)

def main():
  x = f2(2, 3)
  return 0

main()
```



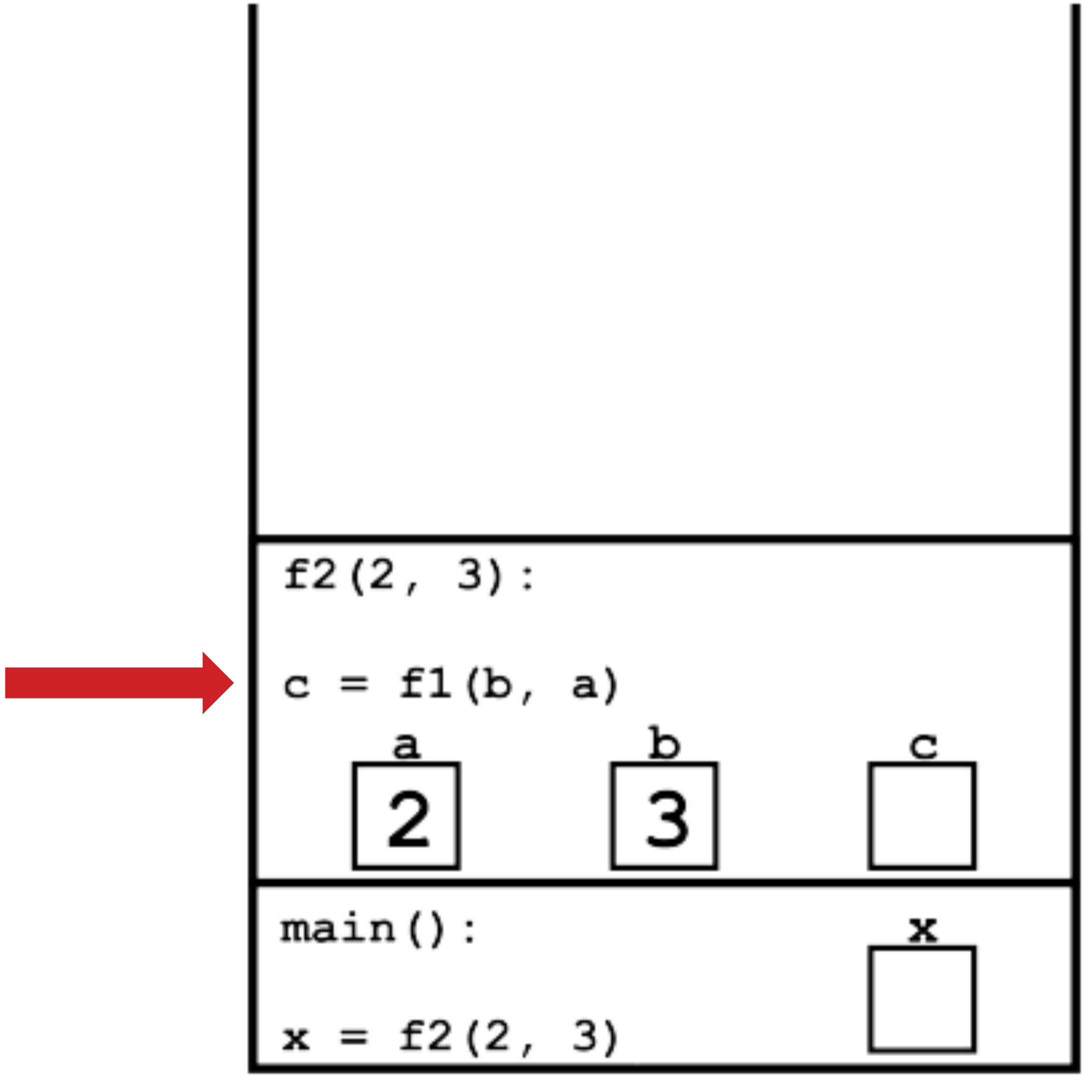
Gerenciamento de memória

```python
def f1(a, b):
  c = a - b
  return (a + b + c)

def f2(a, b):
  c = f1(b, a)
  return (b + c - a)

def main():
  x = f2(2, 3)
  return 0

main()
```



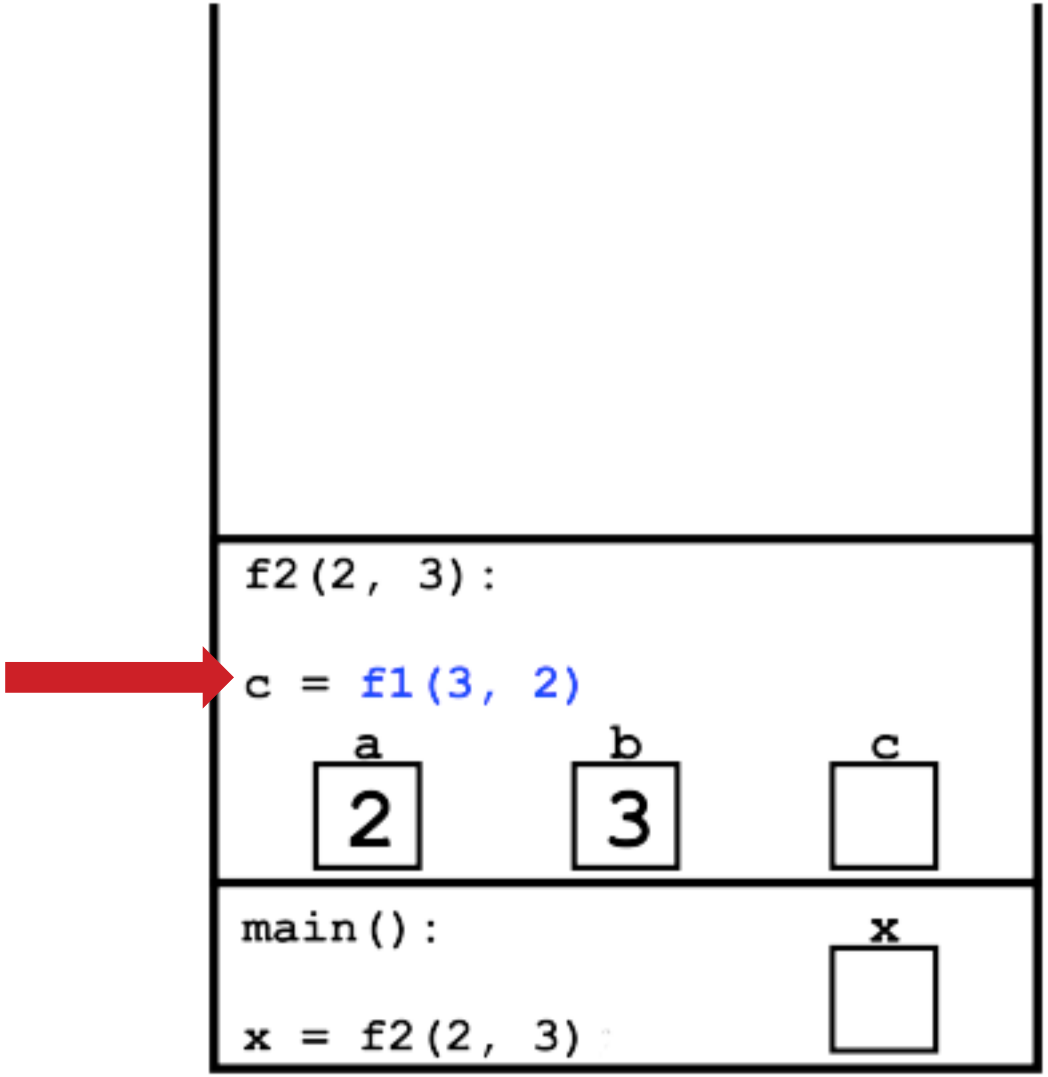
Gerenciamento de memória

```
def f1(a, b):
  c = a - b
  return (a + b + c)

def f2(a, b):
  c = f1(b, a)
  return (b + c - a)

def main():
  x = f2(2, 3)
  return 0

main()
```



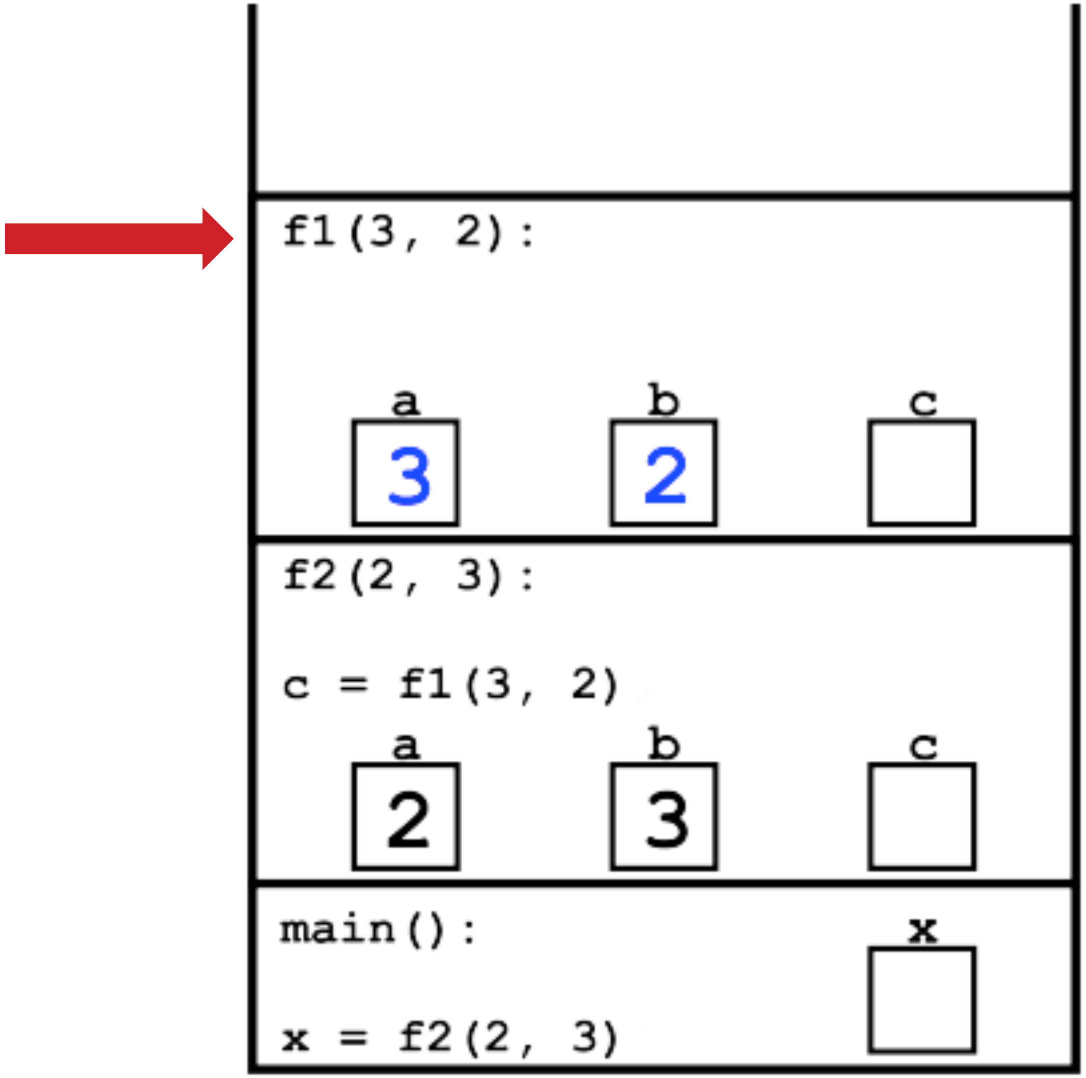
Gerenciamento de memória

```
def f1(a, b):
  c = a - b
  return (a + b + c)

def f2(a, b):
  c = f1(b, a)
  return (b + c - a)

def main():
  x = f2(2, 3)
  return 0

main()
```



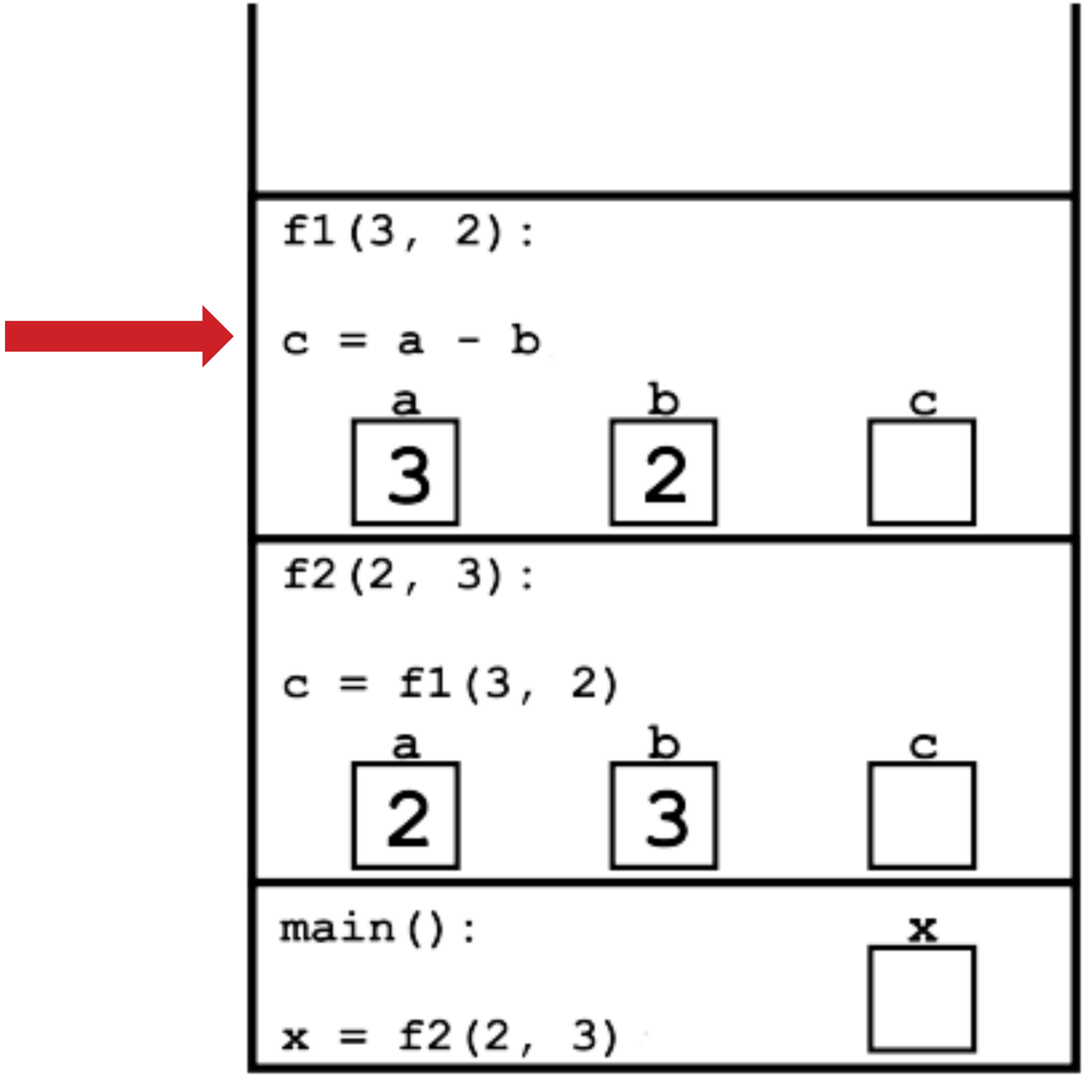
Gerenciamento de memória

```python
def f1(a, b):
  c = a - b
  return (a + b + c)

def f2(a, b):
  c = f1(b, a)
  return (b + c - a)

def main():
  x = f2(2, 3)
  return 0

main()
```



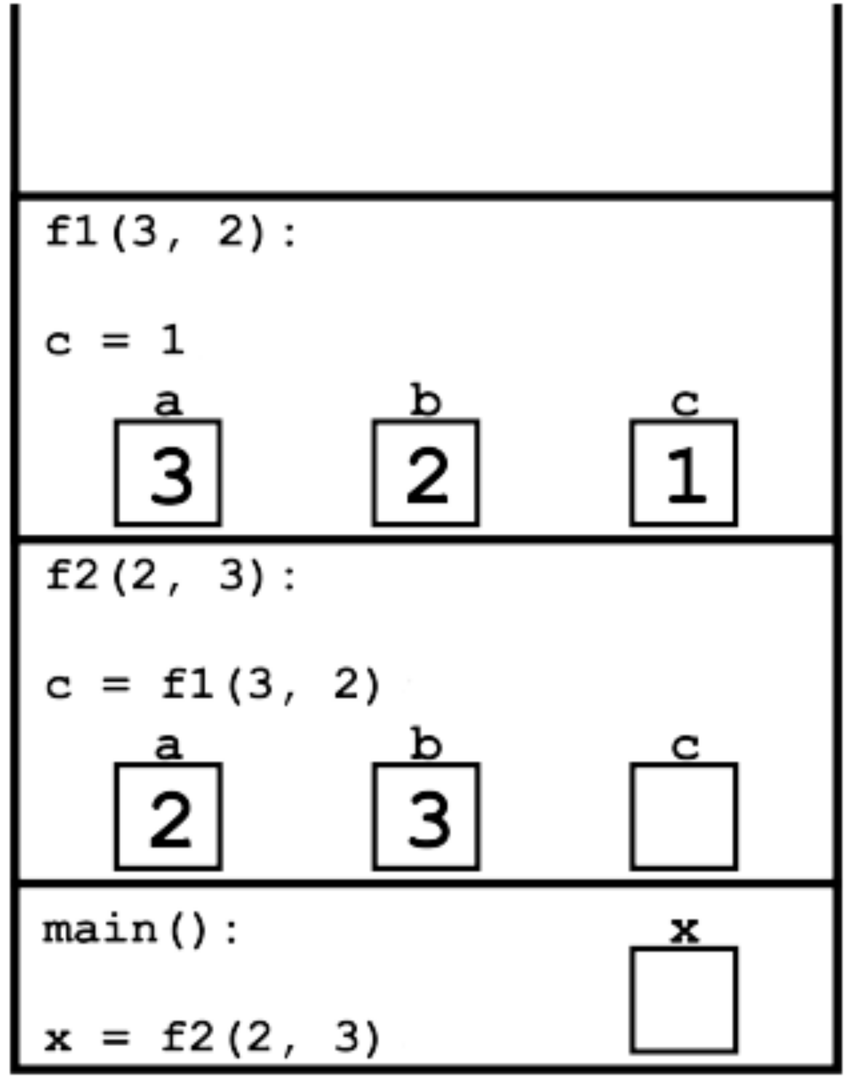
Gerenciamento de memória

```
def f1(a, b):
  c = a - b
  return (a + b + c)

def f2(a, b):
  c = f1(b, a)
  return (b + c - a)

def main():
  x = f2(2, 3)
  return 0

main()
```



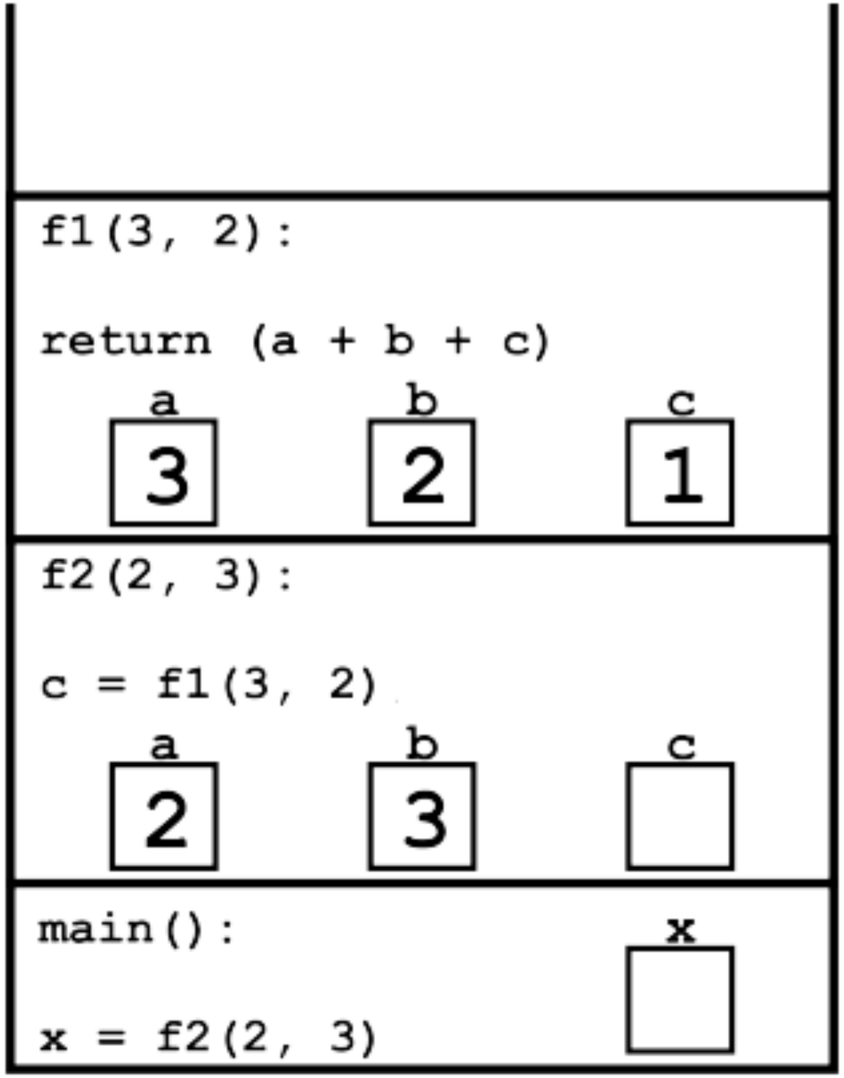
Gerenciamento de memória

```
def f1(a, b):
  c = a - b
  return (a + b + c)

def f2(a, b):
  c = f1(b, a)
  return (b + c - a)

def main():
  x = f2(2, 3)
  return 0

main()
```



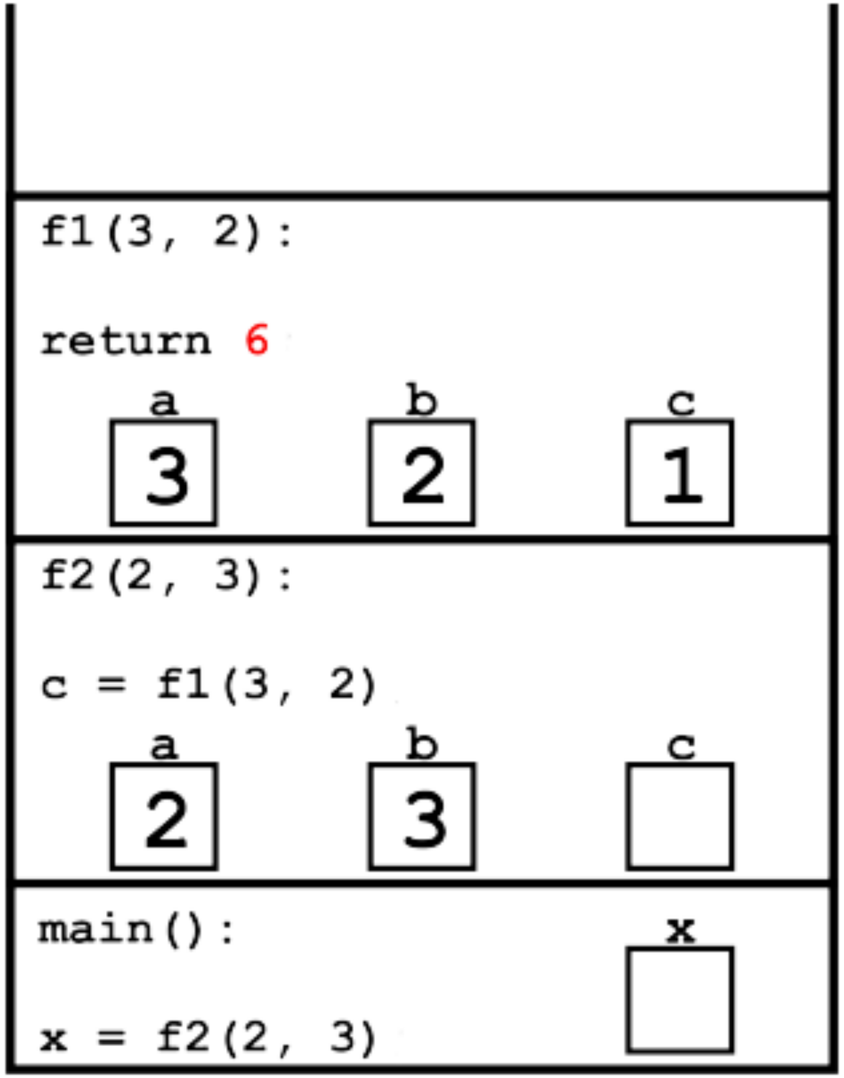
Gerenciamento de memória

```python
def f1(a, b):
    c = a - b
    return (a + b + c)

def f2(a, b):
    c = f1(b, a)
    return (b + c - a)

def main():
    x = f2(2, 3)
    return 0

main()
```



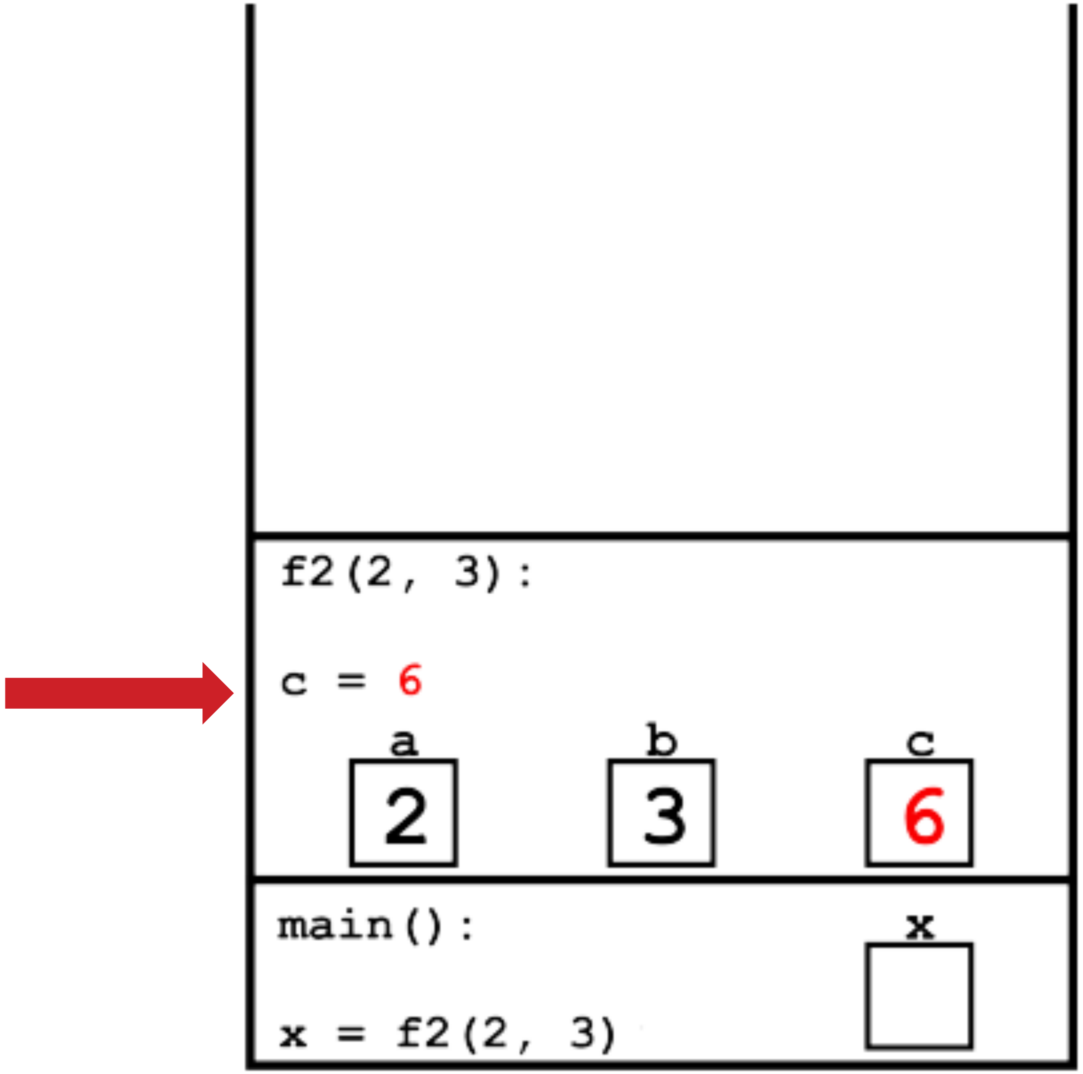
Gerenciamento de memória

```python
def f1(a, b):
  c = a - b
  return (a + b + c)

def f2(a, b):
  c = f1(b, a)
  return (b + c - a)

def main():
  x = f2(2, 3)
  return 0

main()
```



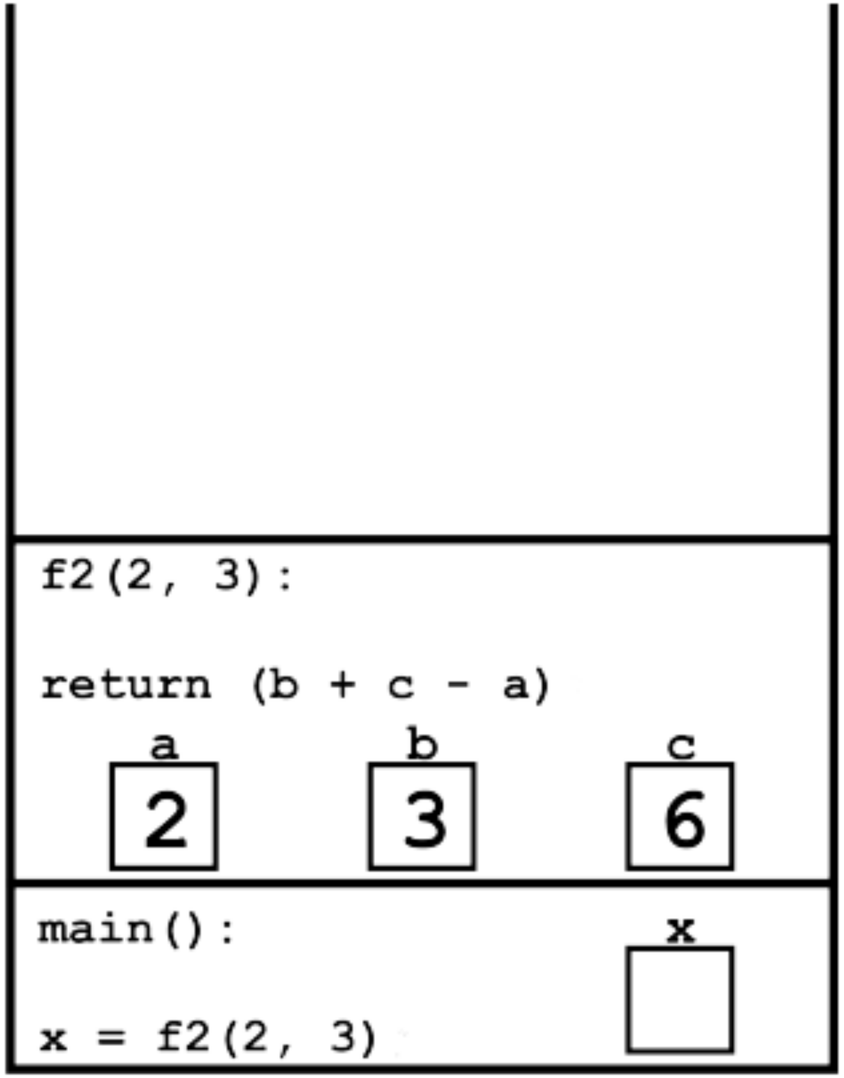
Gerenciamento de memória

```
def f1(a, b):
  c = a - b
  return (a + b + c)

def f2(a, b):
  c = f1(b, a)
  return (b + c - a)

def main():
  x = f2(2, 3)
  return 0

main()
```



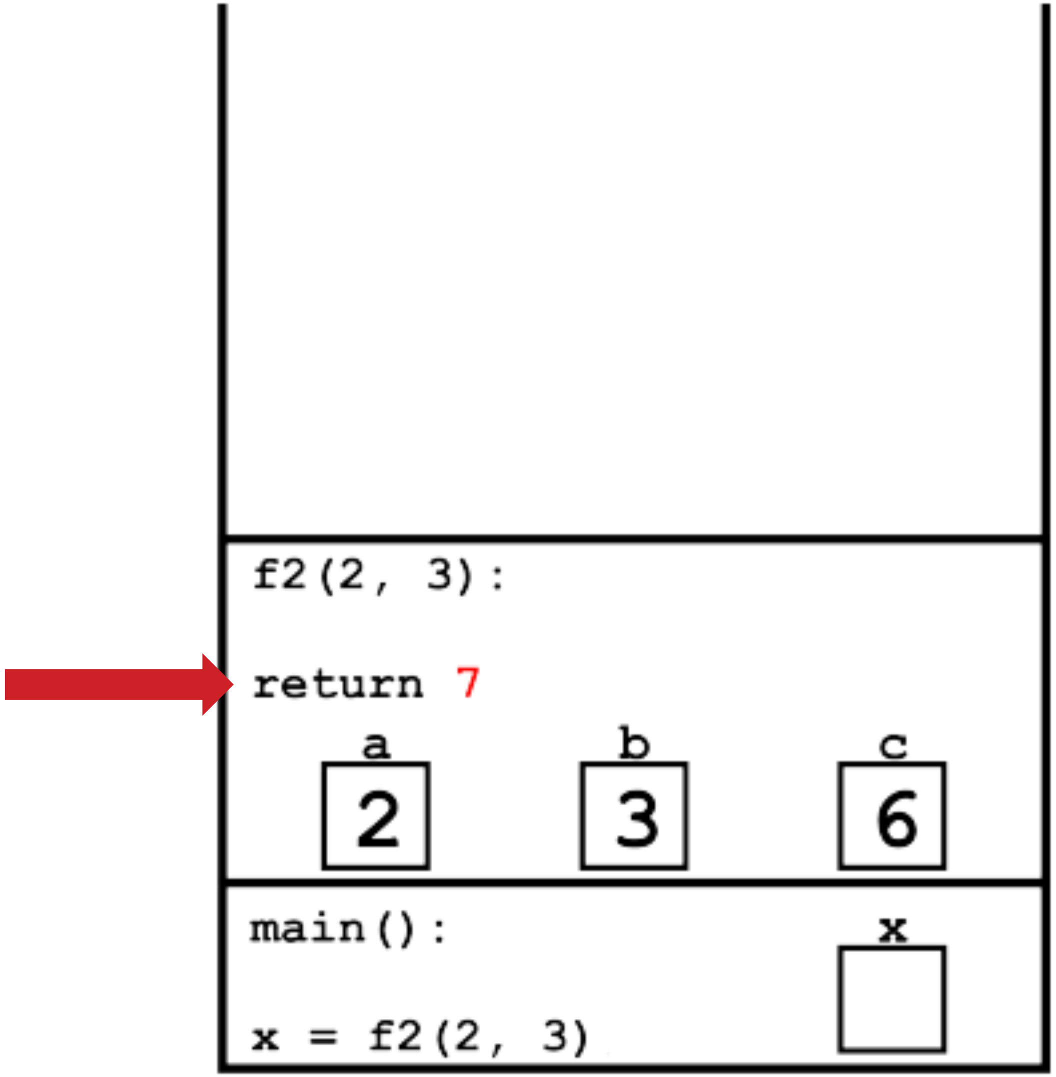
Gerenciamento de memória

```python
def f1(a, b):
  c = a - b
  return (a + b + c)

def f2(a, b):
  c = f1(b, a)
  return (b + c - a)

def main():
  x = f2(2, 3)
  return 0

main()
```



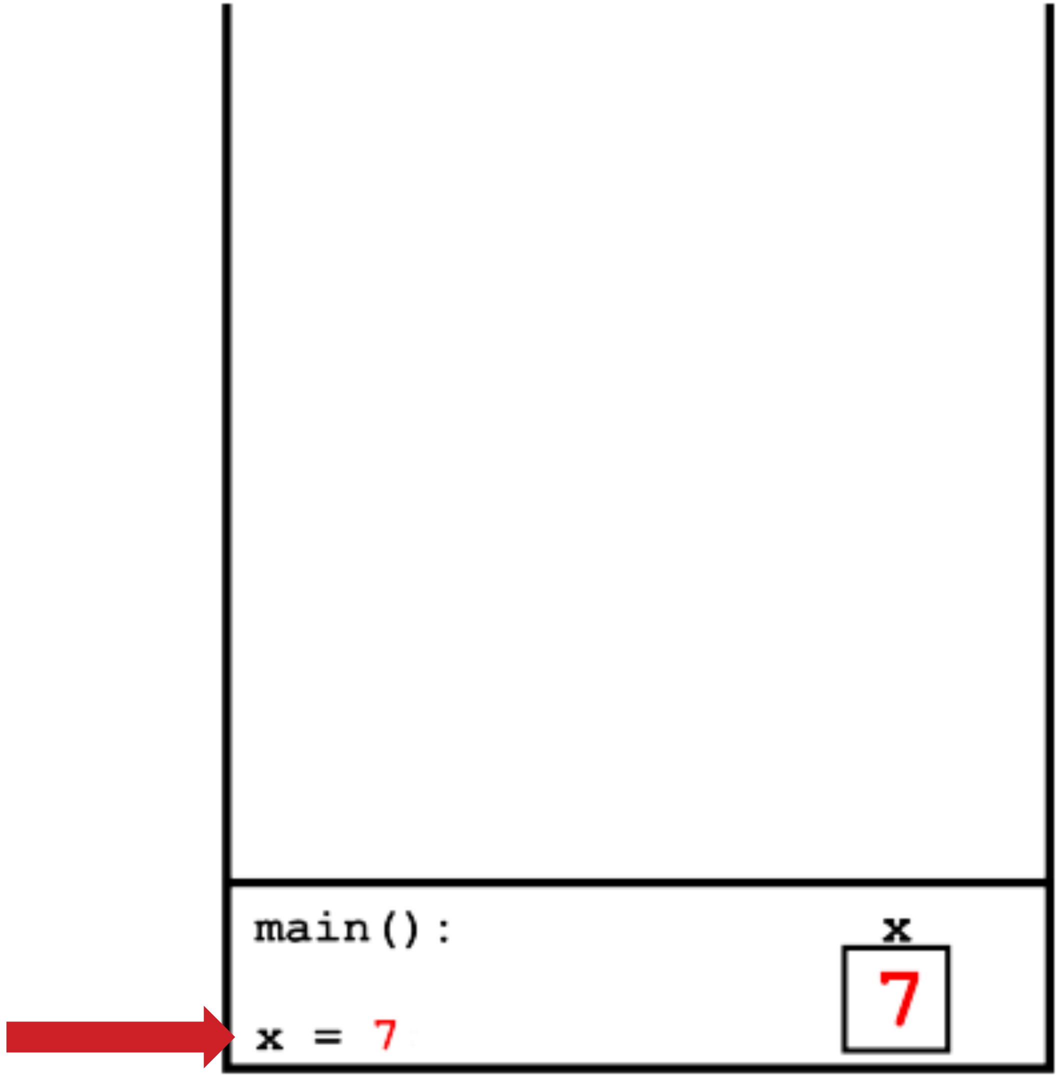
Gerenciamento de memória

```python
def f1(a, b):
  c = a - b
  return (a + b + c)

def f2(a, b):
  c = f1(b, a)
  return (b + c - a)

def main():
  x = f2(2, 3)
  return 0

main()
```



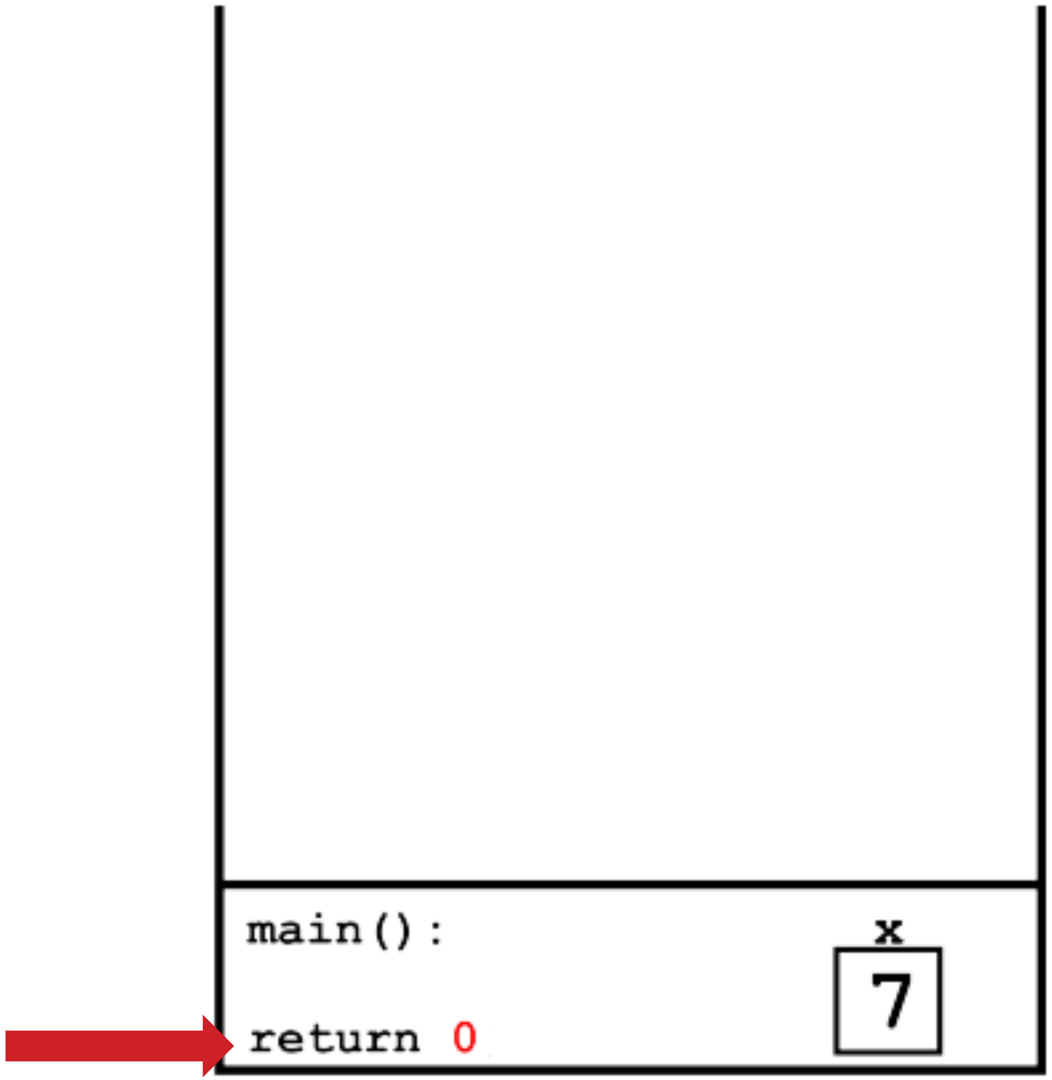
Gerenciamento de memória

```
def f1(a, b):
  c = a - b
  return (a + b + c)

def f2(a, b):
  c = f1(b, a)
  return (b + c - a)

def main():
  x = f2(2, 3)
  return 0

main()
```



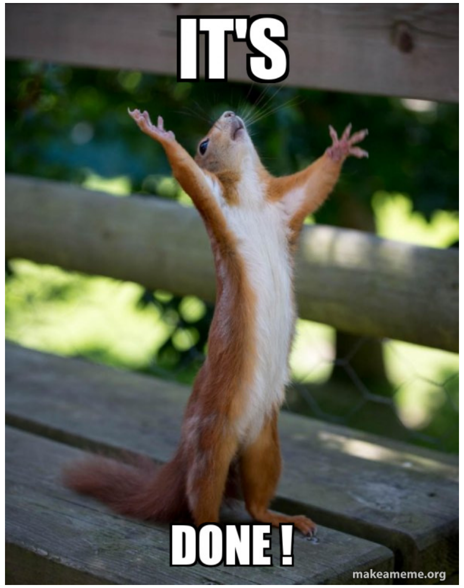
Gerenciamento de memória

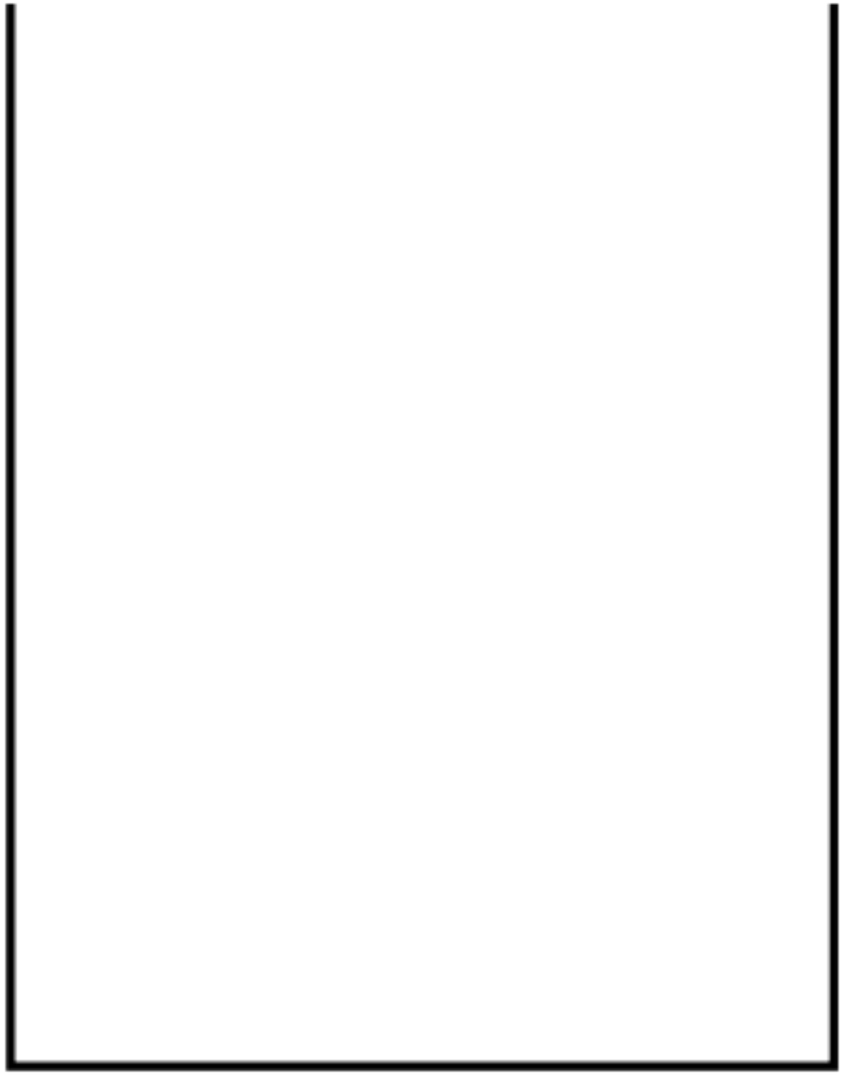
```python
def f1(a, b):
  c = a - b
  return (a + b + c)

def f2(a, b):
  c = f1(b, a)
  return (b + c - a)

def main():
  x = f2(2, 3)
  return 0

main()
```





Recursão e iteração

- Em geral, uma **função definida recursivamente pode ser também definida de uma forma iterativa** (através de estruturas de repetição).

- A definição recursiva é mais "declarativa" – explicita o que se pretende obter e não a forma como se obtém (ou seja, o algoritmo que é usado).

Recursão e iteração

Por outro lado, uma **definição iterativa**, embora **não permita uma compreensão tão imediata**, **é** geralmente **mais eficiente**, dado que a **implementação recursiva precisa registrar o estado atual** do processamento **para continuar** de onde parou após a conclusão de cada nova execução, e **isso consome tempo e memória**.

Recursividade

```
int fatorialNaoRecursivo (int n){

    int resultado, contador;
    resultado = 1;
    if ( n!=0 ){

        for(contador=n; contador >= 1; contador--){
            resultado = resultado * contador;
        }

    }
    return resultado;
}
```

Torres de Hanói

“Torres de Hanói” é um jogo matemático onde dispomos de **3 pinos**: “pino origem”, “pino de trabalho” e “pino destino”. O “pino origem” contém **n discos** empilhados por ordem crescente de tamanho (o maior disco fica embaixo). O objetivo do jogo é levar todos os discos do “pino origem” para o “pino destino”, utilizando o “pino de trabalho” para auxiliar a tarefa, e atendendo às seguintes restrições:

Torres de Hanói

1. Apenas um disco pode ser movido por vez (o disco que estiver no topo da pilha de um dos pinos).

2. Um disco de tamanho maior nunca pode ser colocado sobre um disco de tamanho menor.

Torres de Hanói

Para resolver um jogo onde precisamos mover n discos, considerando n > 1, podemos executar os seguintes passos:

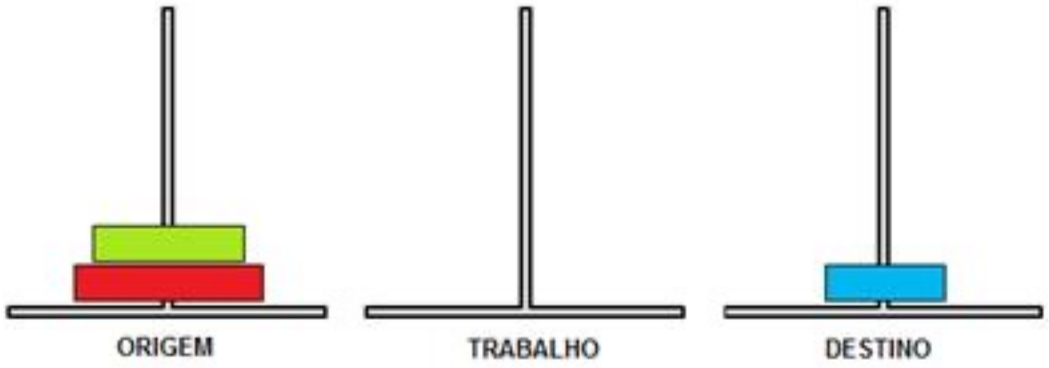
- Mover n-1 discos para o “pino de trabalho”.
- Mover o n-ésimo pino (o maior de todos) do “pino origem” para o “pino destino”.
- Após isto, devemos resolver o problema da “Torre de Hanói” para os n-1 discos dispostos no “pino de trabalho”, movendo-os para o “pino destino” utilizando o mesmo princípio.

Torres de Hanói

PASSO 1

Os movimentos 1, 2 e 3 mostram a transferência de n-1 discos do “pino origem” para o “pino de trabalho. Nesta caso, “pino destino” atua como auxiliar.

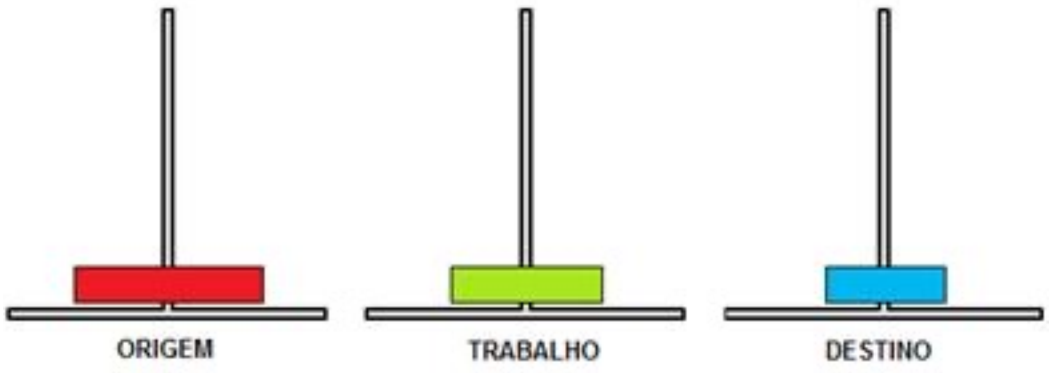
Movimento 1: Origem -> Destino



jogo com 3 discos (n = 3)

Torres de Hanói

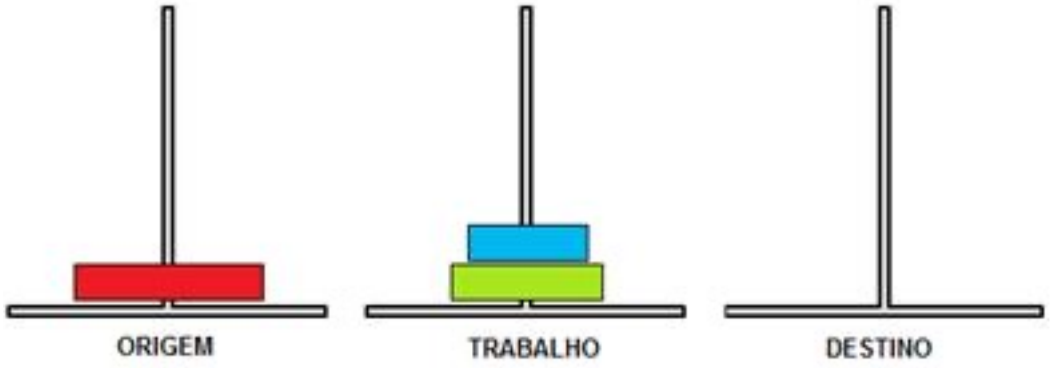
Movimento 2: Origem -> Trabalho



jogo com 3 discos (n = 3)

Torres de Hanói

Movimento 3: Destino -> Trabalho



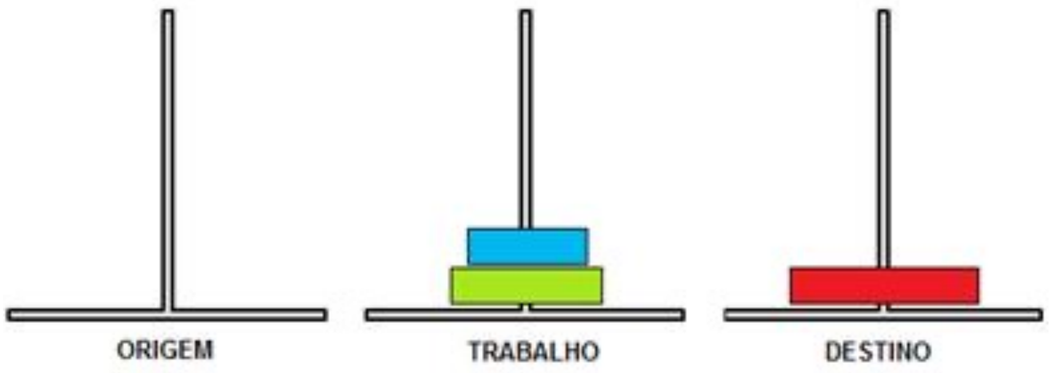
jogo com 3 discos (n = 3)

Torres de Hanói

Movimento 4: Origem -> Destino

PASSO 2

O movimento 4 mostra a transferência do maior disco do “pino origem” para o “pino destino”



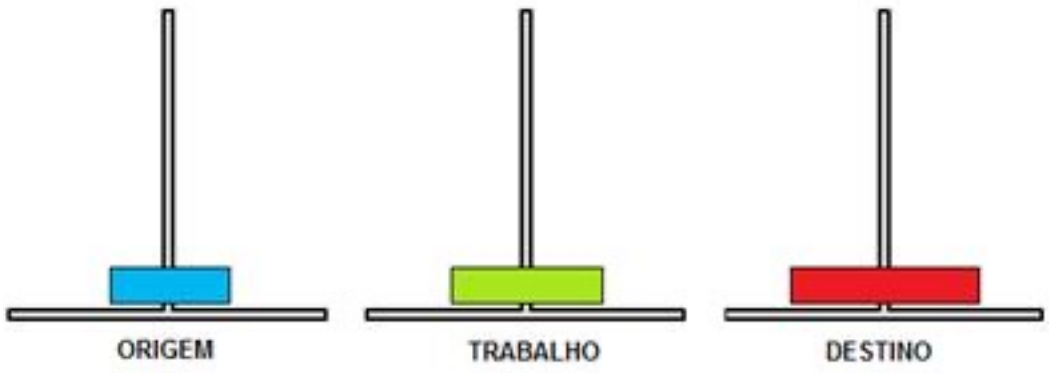
jogo com 3 discos (n = 3)

Torres de Hanói

PASSO 3

Por fim, os movimentos 5, 6 e 7 ilustram a transferência dos n-1 discos do “pino de trabalho” para o “pino destino”. Veja que, desta vez, o “pino de origem” é que atua como área de armazenamento auxiliar.

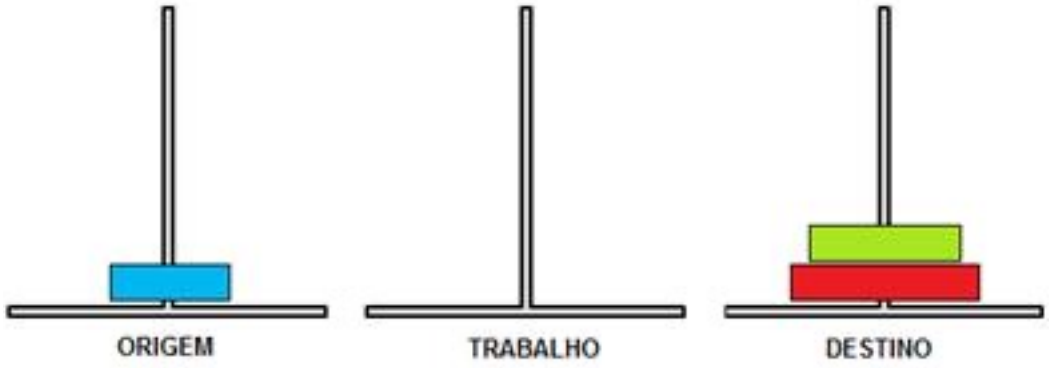
Movimento 5: Trabalho -> Origem



jogo com 3 discos (n = 3)

Torres de Hanói

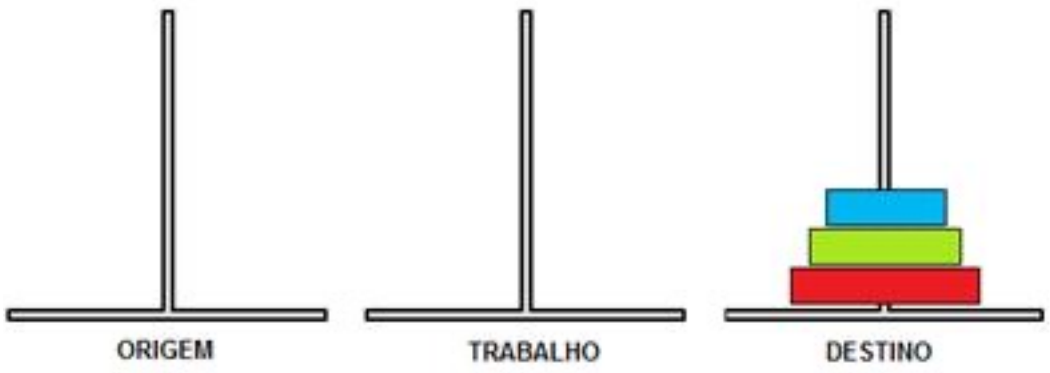
Movimento 6: Trabalho -> Destino



jogo com 3 discos (n = 3)

Torres de Hanói

Movimento 7: Origem -> Destino



jogo com 3 discos (n = 3)

Torres de Hanói

Faça a versão recursiva e iterativa

Uma solução recursiva para o problema:

- Se N=1N=1: mova um disco do pino A para o pino C.
- Se N>1N>1:
  - primeiro transfira N−1N−1 discos de A para B;
  - em seguida mova um disco do pino A para o pino C;
  - por último transfira N−1N−1 discos de B para C.

Para saber mais

https://www.youtube.com/watch?v=X56_FjmbmE4

Bibliografia

BARNES, David J.; KOLLING, Michael. **Programação orientada a objetos com Java**. 4. ed. São Paulo: Prentice Hall – Br, 2009.

Aditya Y. Bhargava. **Entendendo Algoritmos**. Novatec Editora. ISBN 9788575226629.

Cormen, T. H. (2012). Algoritmos: Teoria E prática. Campus. ISBN 8535236996.

DEITEL, Paul; DEITEL, Harvey. **Java: como programar**. 10. ed. São Paulo: Pearson, 2017. ISBN 9788543004792. Disponível em: https://ifsp.bv3.digitalpages.com.br/users/publications/9788543004792. Acesso em: 14 jun. 2019.

Helder da Rocha.. **Gerência de memória em Java.** Argonavis: 2005.

Sierra, Kathy. **Use A Cabeça Java**. S.l: Alta Books, 2009. ISBN: 8576081733



INSTITUTO FEDERAL
São Paulo
Câmpus
São Paulo

Obrigado!

Gustavo Fortunato Puga

gustavo.puga@ifsp.edu.br