O que Significa GNU?
GNU is Not Unix (mas o que significa o GNU?)
GNU is Not Unix (mas o que significa o GNU?)
...

Recursão

Paulo Ricardo Lisboa de Almeida

Chamadas de função

Em programação aprendemos que uma função pode chamar outra função

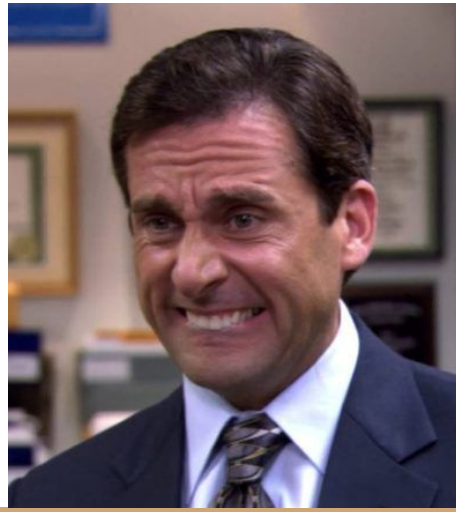
```c
int minhaFuncao(int val){
      return val+1;
}

int main(){
      int resultado = minhaFuncao(1);
      printf("%d\n", resultado);

      return 0;
}
```

função *main* chamando a função *minhaFuncao*

função *main* chamando a função *printf*

Chamadas de função

Mas o que acontece quando uma **função chama ela mesma**?



Chamadas de função

Mas o que acontece quando uma **função chama ela mesma**?

Esse é o princípio da **recursão**

Para que isso funcione, precisamos **garantir duas propriedades**

Recursão

Recursão: resolver um problema resolvendo instâncias menores do mesmo problema.

Propriedade 1: A chamada recursiva deve ser uma instância menor do mesmo problema

Recursão

Recursão: resolver um problema resolvendo instâncias menores do mesmo problema.

Exemplo: calcular 4!

4! = 4.3.2.1 = 24

Recursão

Recursão: resolver um problema resolvendo instâncias menores do mesmo problema.

Exemplo: calcular 4!

4! = 4.3.2.1 = 24

3!

Logo 4! = 4.3!

Para resolver 4!, podemos resolver um problema menor (3!), e multiplicar por 4

Exemplo

Como aplicar essa ideia de maneira geral, para $n!$?

Exemplo

Como aplicar essa ideia de maneira geral, para *n!*?

$n! = n(n-1)!$

Função recursiva para fatorial

```
função fatorial (n)
entrada: inteiro n ≥ 0
saída: n!
retorne n*fatorial(n-1)
```

Função recursiva para fatorial

```
função fatorial (n)
entrada: inteiro n ≥ 0
saída: n!
retorne n*fatorial(n-1)
```

Propriedade 1: A chamada recursiva deve ser uma instância menor do mesmo problema

Essa propriedade é satisfeita neste algoritmo?

Função recursiva para fatorial

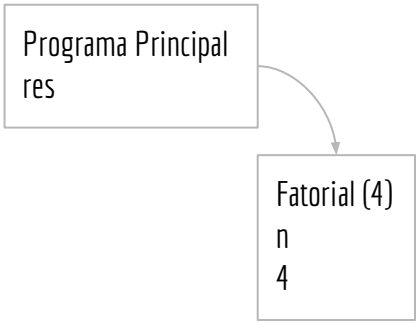
```
função fatorial (n)
entrada: inteiro n ≥ 0
saída: n!
retorne n*fatorial(n-1)
```

Propriedade 1: A chamada recursiva deve ser uma instância menor do mesmo problema

Essa propriedade é satisfeita neste algoritmo?

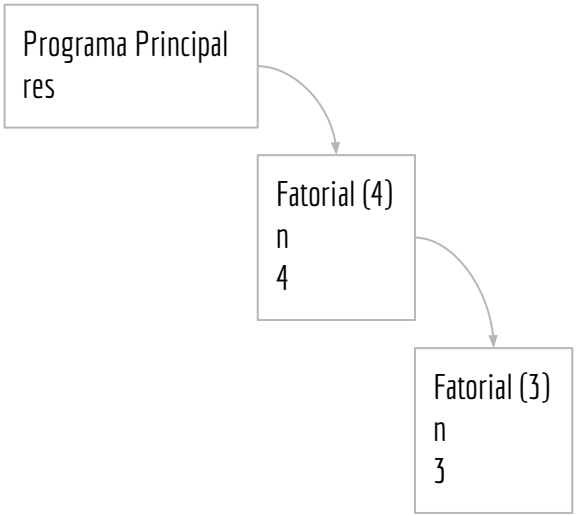
Sim!

Teste de mesa



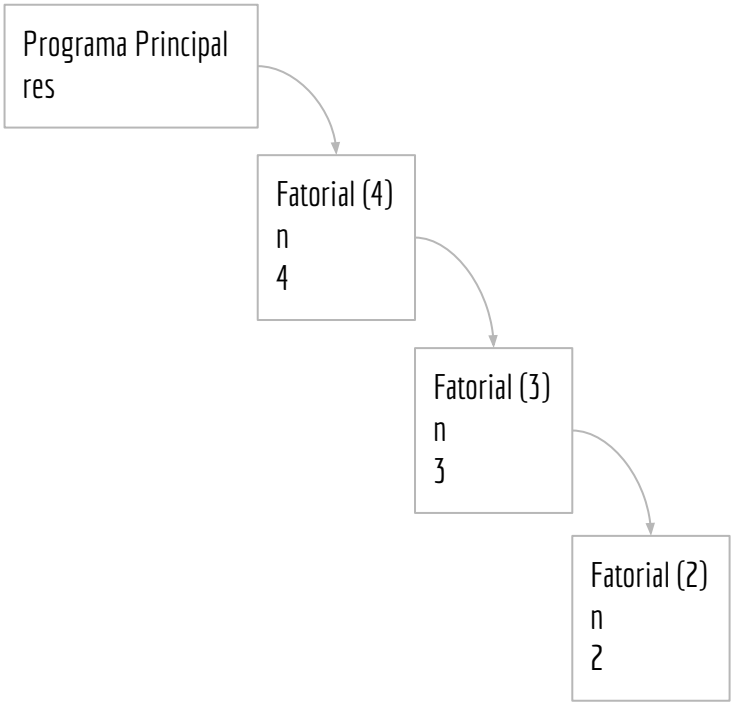

```
programa principal
res ← fatorial(4)
imprima(res)

função fatorial (n)
entrada: inteiro n ≥ 0
saída: n!
retorne n*fatorial(n-1)
```

Teste de mesa



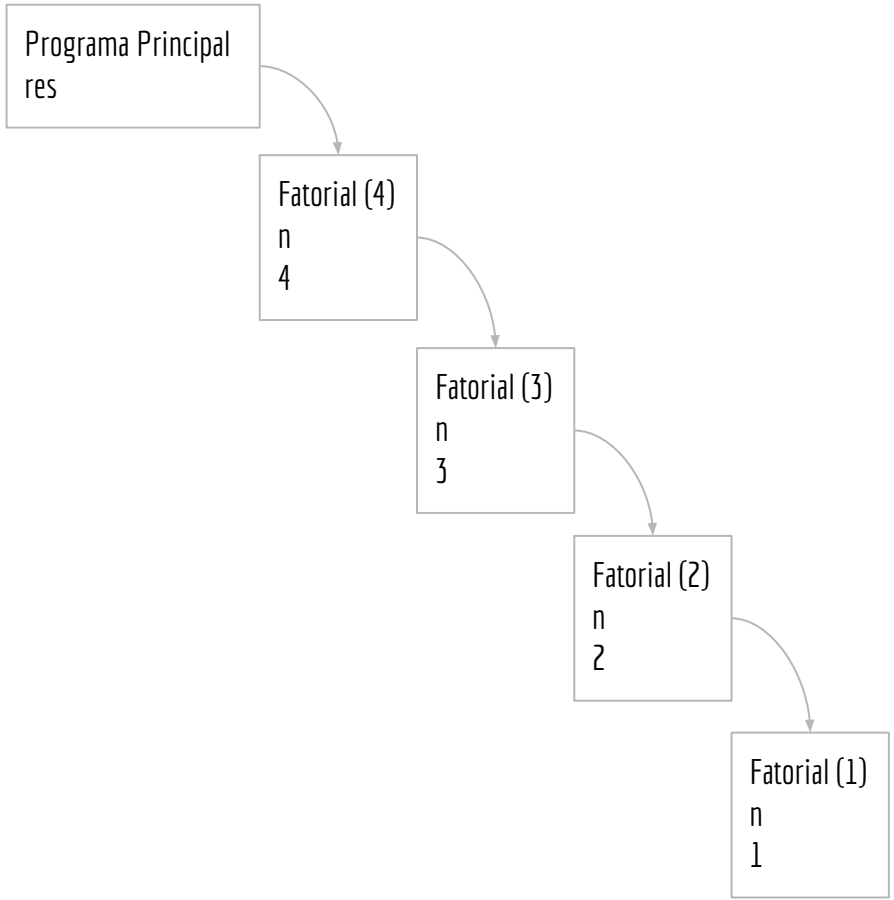

```
programa principal
res ← fatorial(4)
imprima(res)

função fatorial (n)
entrada: inteiro n ≥ 0
saída: n!
retorne n*fatorial(n-1)
```

Teste de mesa



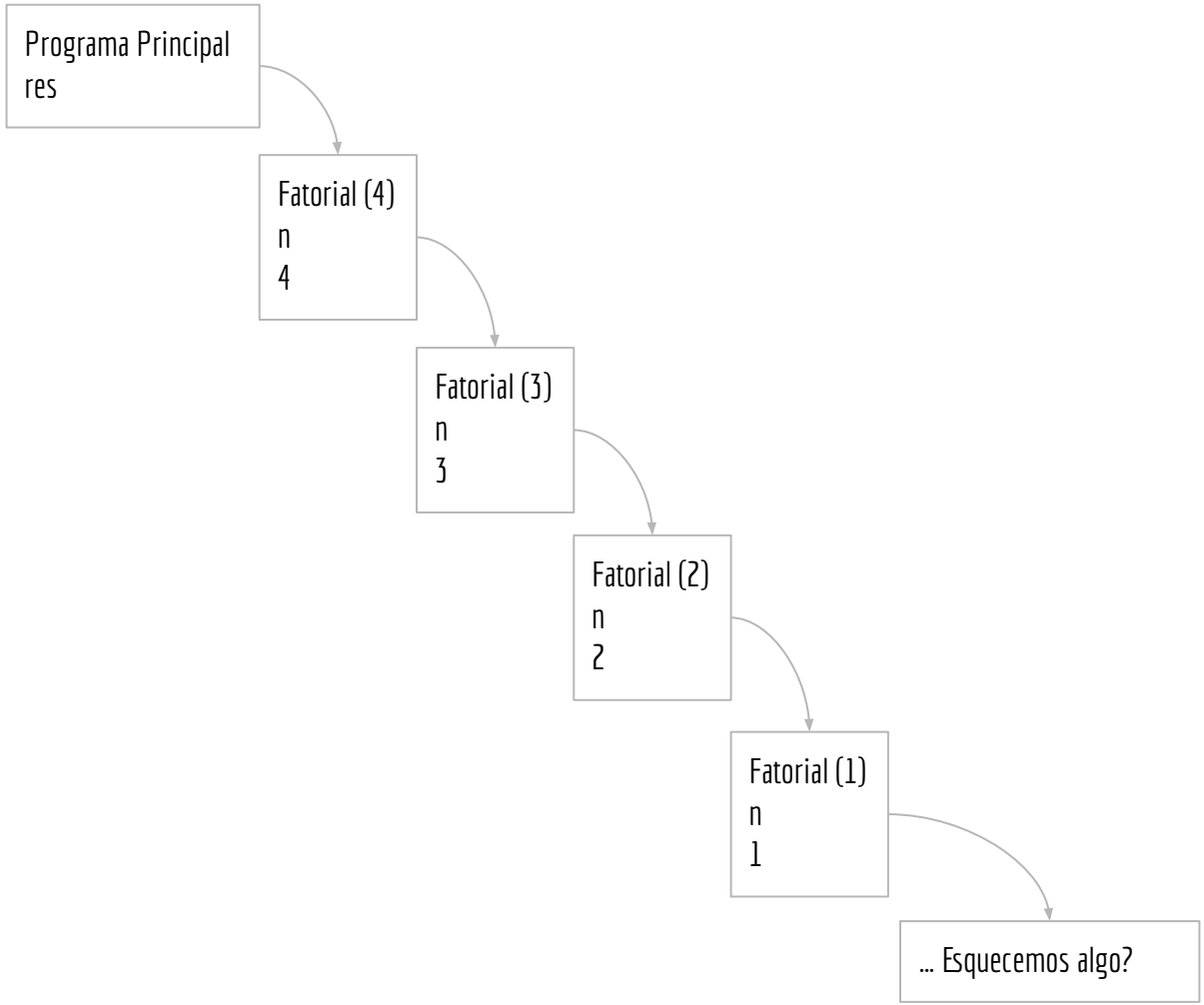

```
programa principal
res ← fatorial(4)
imprima(res)

função fatorial (n)
entrada: inteiro n ≥ 0
saída: n!
retorne n*fatorial(n-1)
```

Teste de mesa




```
programa principal
res ← fatorial(4)
imprima(res)

função fatorial (n)
entrada: inteiro n ≥ 0
saída: n!
retorne n*fatorial(n-1)
```

Teste de mesa




```
programa principal
res ← fatorial(4)
imprima(res)

função fatorial (n)
entrada: inteiro n ≥ 0
saída: n!
retorne n*fatorial(n-1)
```

Caso base

Em algum momento as chamadas recursivas devem terminar atingindo um (ou mais) **caso base**

Um caso base possui **solução trivial** (podemos computar diretamente sua solução)

Essa é a propriedade 2: Deve haver um ou mais casos-bases, nos quais computamos a solução diretamente sem recursão.

No problema fatorial, qual um caso-base que sabemos trivialmente a solução?

Caso base

Em algum momento as chamadas recursivas devem terminar atingindo um (ou mais) **caso base**

Um caso base possui **solução trivial** (podemos computar diretamente sua solução)

Essa é a propriedade 2: Deve haver um ou mais casos-bases, nos quais computamos a solução diretamente sem recursão.

No problema fatorial, qual um caso-base que sabemos trivialmente a solução?

0! = 1

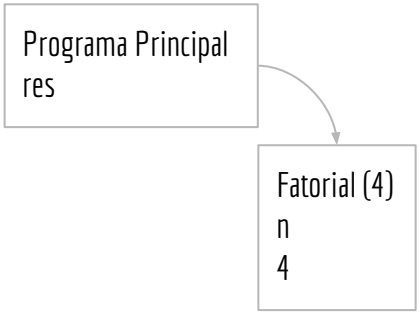
Como adicionar isso em nosso algoritmo?

```
função fatorial (n)
entrada: inteiro n ≥ 0
saída: n!
retorne n*fatorial(n-1)
```

Caso base

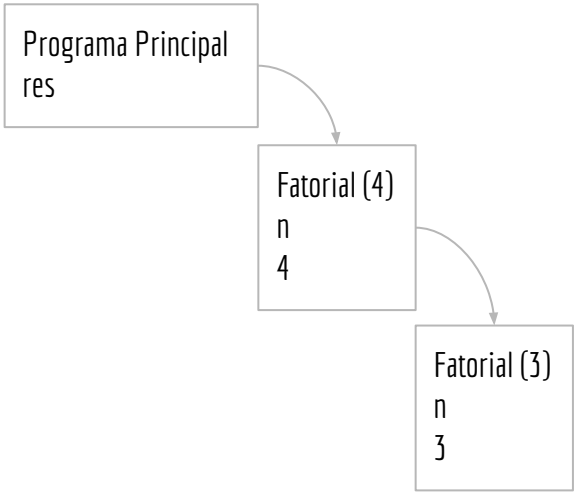
```
função fatorial (n)
entrada: inteiro n ≥ 0
saída: n!
se n = 0 retorne 1
senão retorne n*fatorial(n-1)
```

Teste de mesa



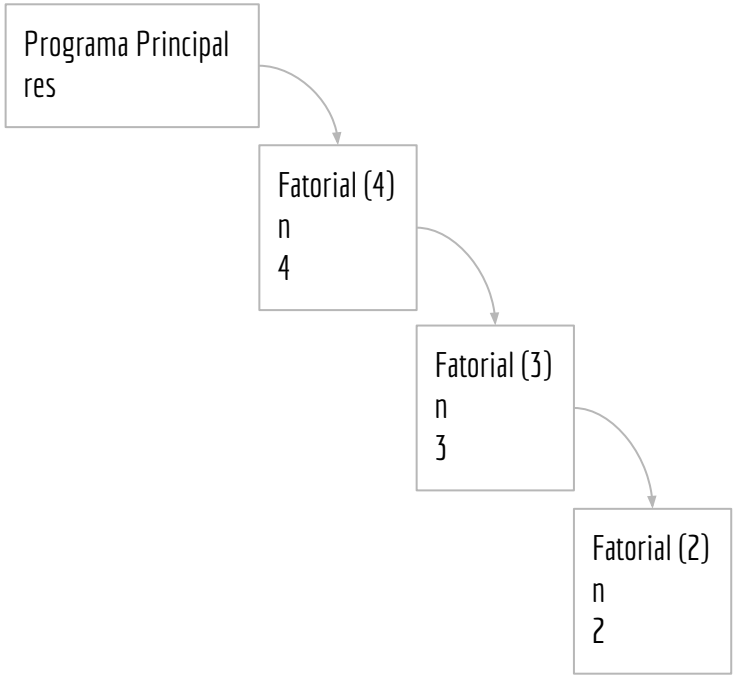

```
programa principal
res ← fatorial(4)
imprima(res)

função fatorial (n)
entrada: inteiro n ≥ 0
saída: n!
se n = 0 retorne 1
senão retorne n*fatorial(n-1)
```

Teste de mesa



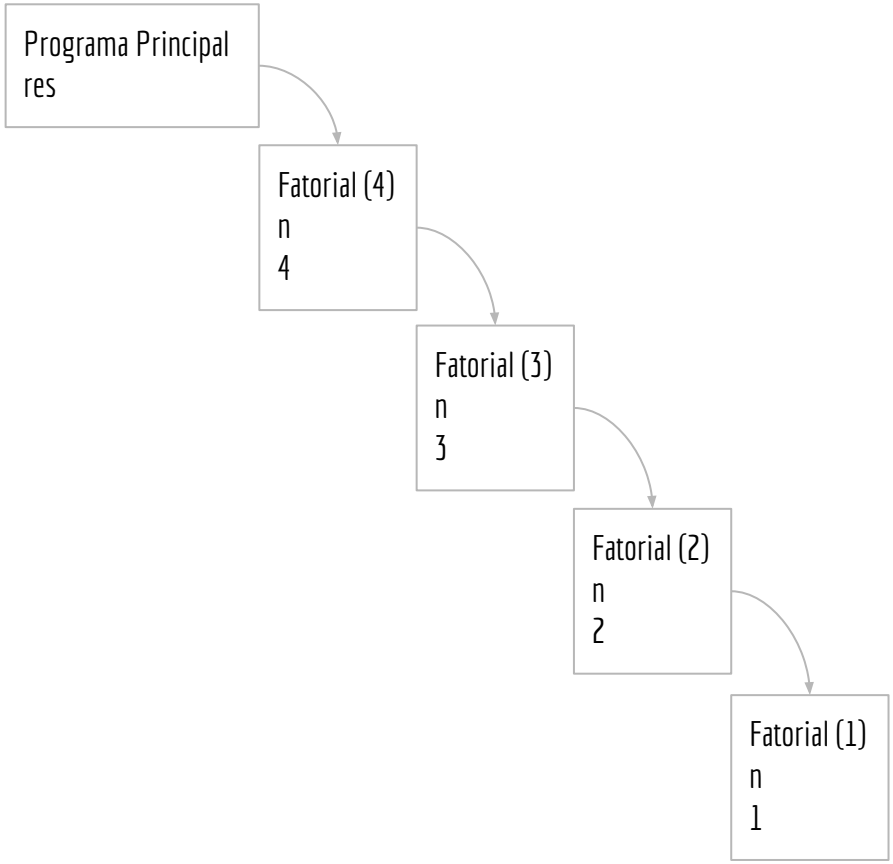

```
programa principal
res ← fatorial(4)
imprima(res)

função fatorial (n)
entrada: inteiro n ≥ 0
saída: n!
se n = 0 retorne 1
senão retorne n*fatorial(n-1)
```

Teste de mesa



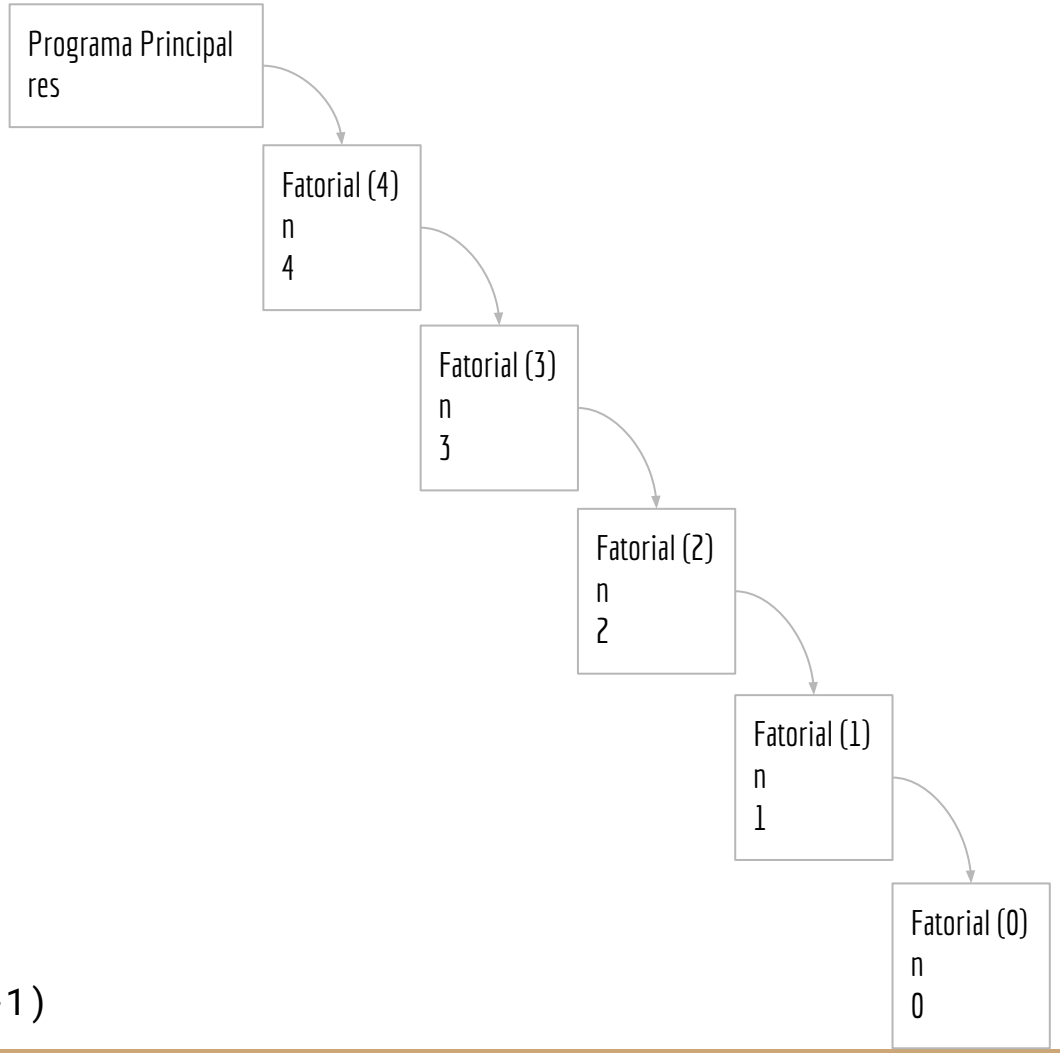

```
programa principal
res ← fatorial(4)
imprima(res)

função fatorial (n)
entrada: inteiro n ≥ 0
saída: n!
se n = 0 retorne 1
senão retorne n*fatorial(n-1)
```

Teste de mesa



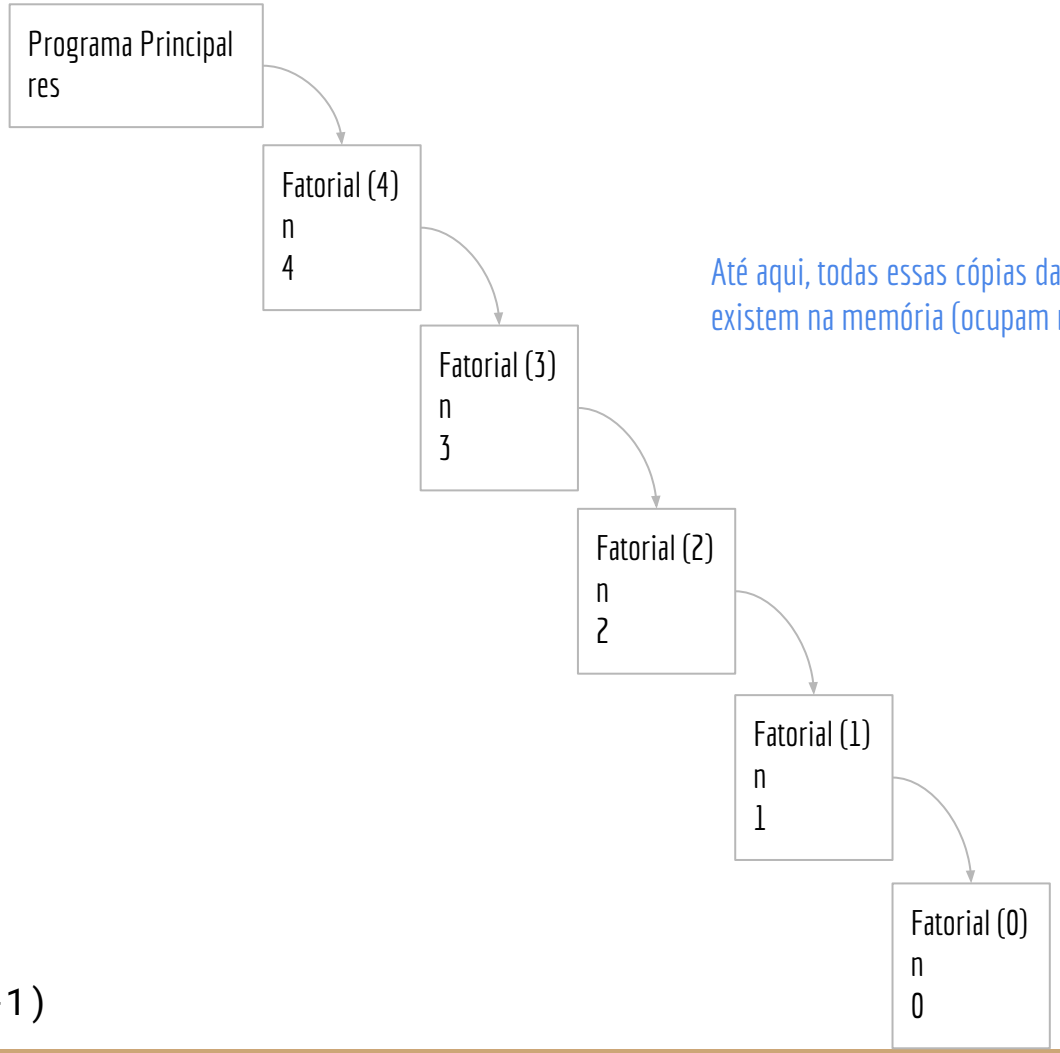

```
programa principal
res ← fatorial(4)
imprima(res)

função fatorial (n)
entrada: inteiro n ≥ 0
saída: n!
se n = 0 retorne 1
senão retorne n*fatorial(n-1)
```

Teste de mesa




```
programa principal
res ← fatorial(4)
imprima(res)

função fatorial (n)
entrada: inteiro n ≥ 0
saída: n!
se n = 0 retorne 1
senão retorne n*fatorial(n-1)
```

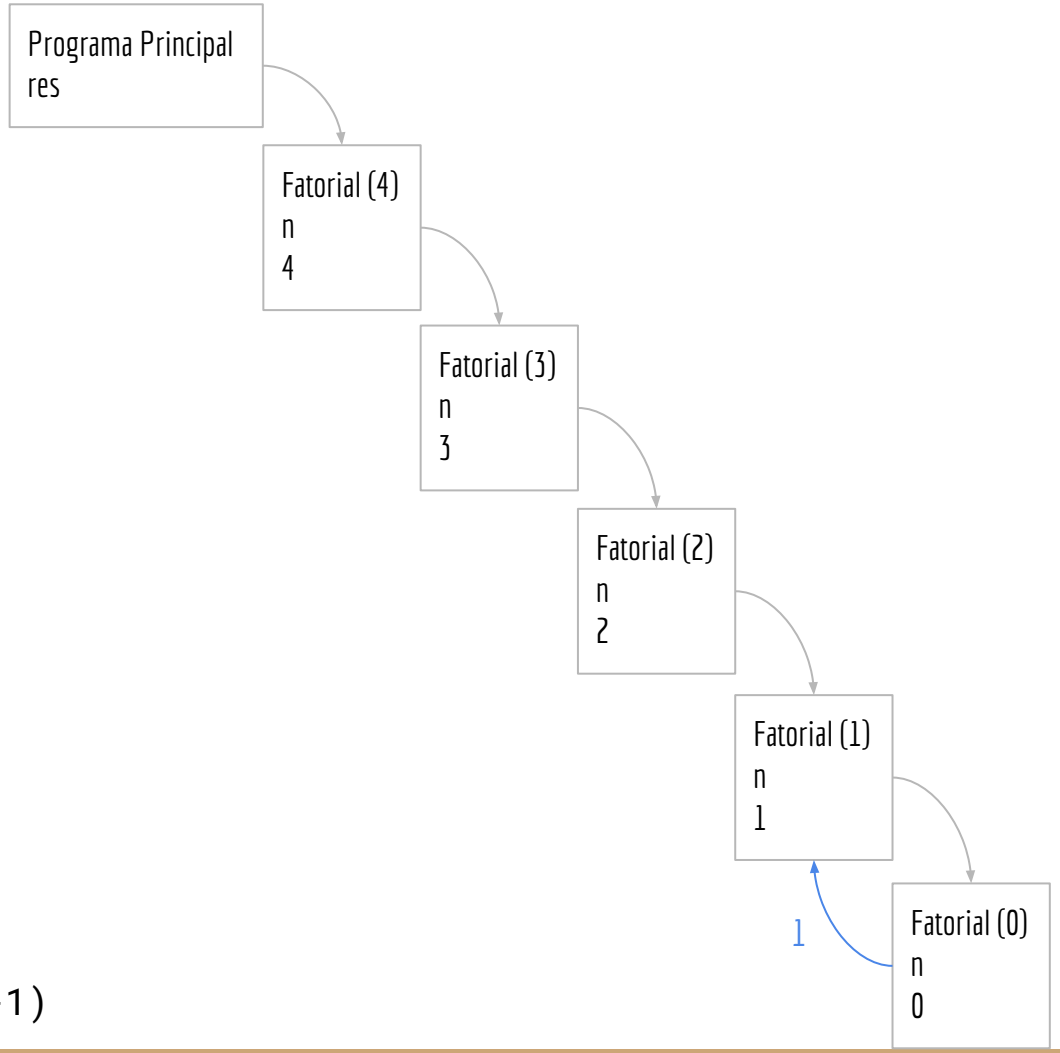
Teste de mesa



Até aqui, todas essas cópias da função existem na memória (ocupam memória)

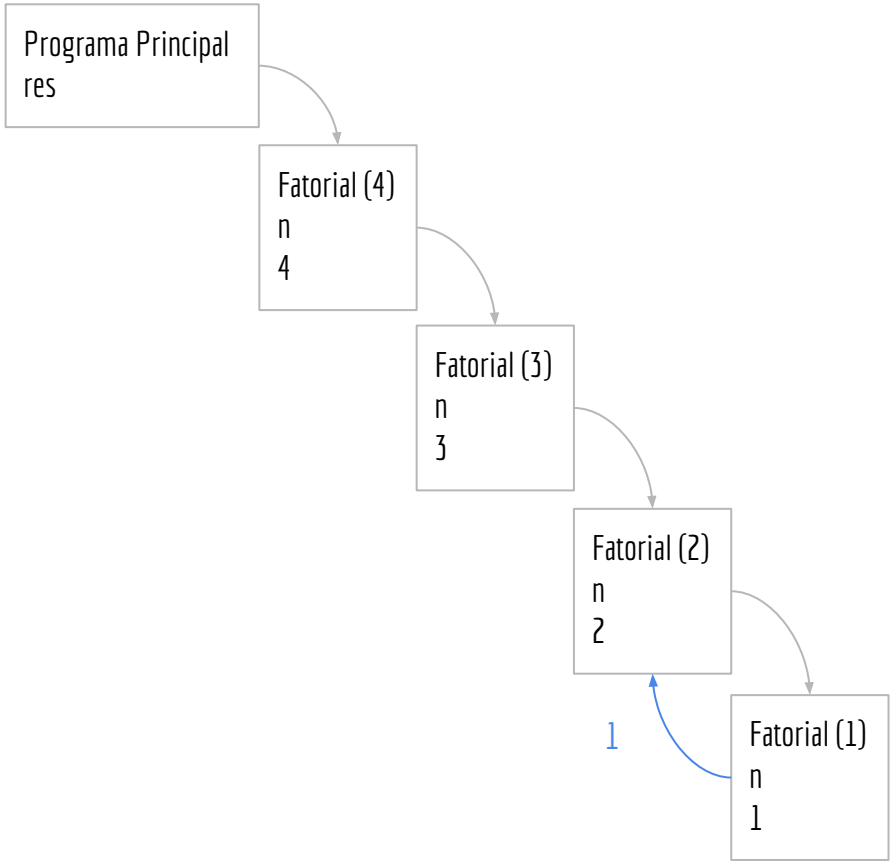
```
programa principal
res ← fatorial(4)
imprima(res)

função fatorial (n)
entrada: inteiro n ≥ 0
saída: n!
se n = 0 retorne 1
senão retorne n*fatorial(n-1)
```

Teste de mesa



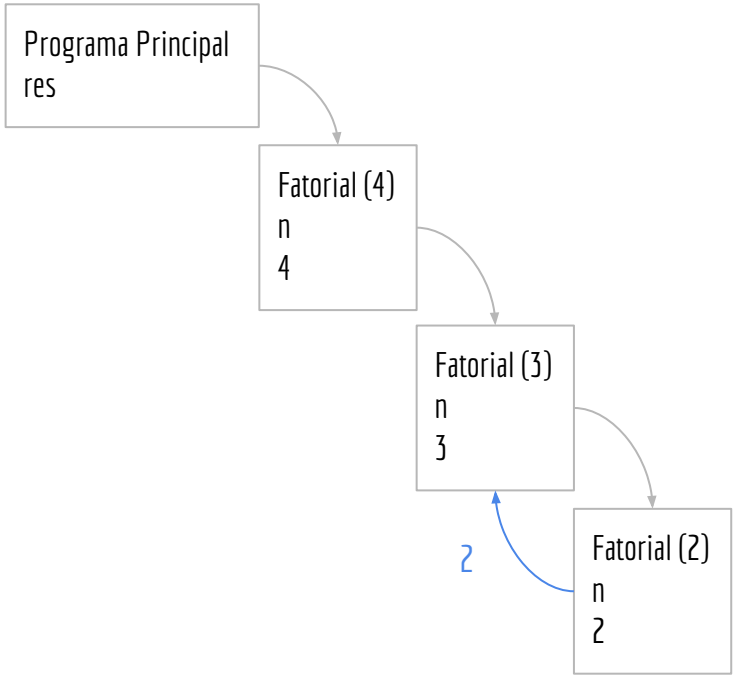

```
programa principal
res ← fatorial(4)
imprima(res)

função fatorial (n)
entrada: inteiro n ≥ 0
saída: n!
se n = 0 retorne 1
senão retorne n*fatorial(n-1)
```

Teste de mesa



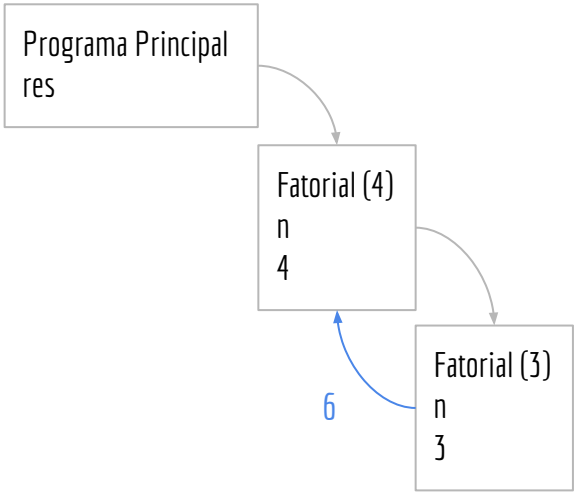

```
programa principal
res ← fatorial(4)
imprima(res)

função fatorial (n)
entrada: inteiro n ≥ 0
saída: n!
se n = 0 retorne 1
senão retorne n*fatorial(n-1)
```

Teste de mesa



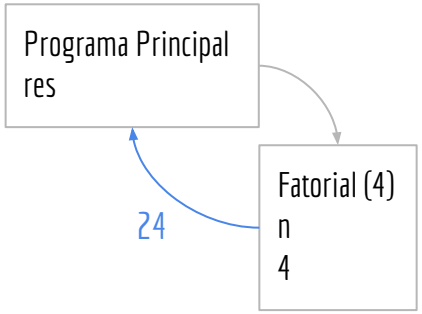

```
programa principal
res ← fatorial(4)
imprima(res)

função fatorial (n)
entrada: inteiro n ≥ 0
saída: n!
se n = 0 retorne 1
senão retorne n*fatorial(n-1)
```

Teste de mesa




```
programa principal
res ← fatorial(4)
imprima(res)

função fatorial (n)
entrada: inteiro n ≥ 0
saída: n!
se n = 0 retorne 1
senão retorne n*fatorial(n-1)
```

Teste de mesa




```
programa principal
res ← fatorial(4)
imprima(res)

função fatorial (n)
entrada: inteiro n ≥ 0
saída: n!
se n = 0 retorne 1
senão retorne n*fatorial(n-1)
```

Teste de mesa

Programa Principal
res
24

```
programa principal
res ← fatorial(4)
imprima(res)

função fatorial (n)
entrada: inteiro n ≥ 0
saída: n!
se n = 0 retorne 1
senão retorne n*fatorial(n-1)
```

Em C

```c
#include <stdio.h>

int fat(unsigned int n){
     if(n == 0)
          return 1;
     return n*fat(n-1);
}

int main(){
     unsigned int valFat, res;
     printf("Digite o valor para calcular: ");
     scanf("%u", &valFat);
     res =  fat(valFat);
     printf("%u\n", res);

     return 0;
}
```

programa principal
res ← fatorial(4)
imprima(res)

função fatorial (n)
entrada: inteiro n ≥ 0
saída: n!
se n = 0 retorne 1
senão retorne n*fatorial(n-1)

Valor mínimo

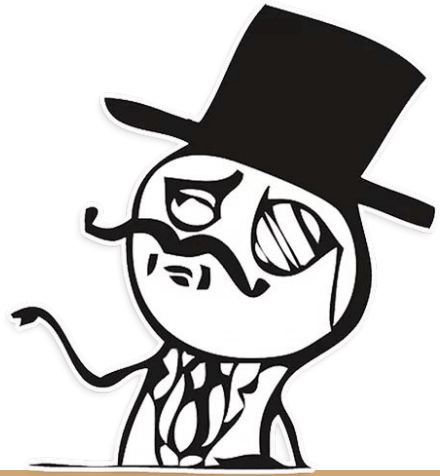
Crie um algoritmo que dado um vetor *v* não ordenado de tamanho *n*, e retorna o índice do vetor que contém o menor valor de *v*

Valor mínimo

Crie um algoritmo que dado um vetor *v* não ordenado de tamanho *n*, e retorna o índice do vetor que contém o menor valor de *v*

Vamos manter a classe e formular matematicamente o problema

Seja um vetor v indexado por $[a..b]$, com $a < b$, obtenha um índice $m \in [a..b]$ tal que $v[m] \leq v[j], \forall j \in [a..b]$.



Valor mínimo

Crie uma versão **não recursiva** para esse algoritmo

Seja um vetor v indexado por $[a..b]$, com $a < b$, obtenha um índice $m \in [a..b]$ tal que $v[m] \leq v[j], \forall j \in [a..b]$.

```
função minimoVetor (v,a,b)
entrada: vetor v indexado por [a..b], com a ≤ b
saída: m ∈ [a..b], tal que v[m] ≤ v[j] ∀j ∈ [a..b]
…
```

Valor mínimo

Crie uma versão **não recursiva** para esse algoritmo

Seja um vetor v indexado por $[a..b]$, com $a < b$, obtenha um índice $m \in [a..b]$ tal que $v[m] \leq v[j], \forall j \in [a..b]$.

```
função minimoVetor (v,a,b)
entrada: vetor v indexado por [a..b], com a ≤ b
saída: m ∈ [a..b], tal que v[m] ≤ v[j] ∀j ∈ [a..b]
j ← a
m ← a
enquanto j < b
    j ← j + 1
    se v[j] < v[m]
       m ← j
retorne m
```

Em C

Em C use [] para passar um vetor como parâmetro

Um vetor *v* de tamanho *m* em C é indexado como v[0..m-1].

```
função minimoVetor (v,a,b)
entrada: vetor v indexado por [a..b], com a ≤ b
saída: m ∈ [a..b], tal que v[m] ≤ v[j] ∀j ∈ [a..b]
j ← a
m ← a
enquanto j < b
    j ← j + 1
    se v[j] < v[m]
       m ← j
retorne m
```

```c
#include <stdio.h>
#include <time.h>
#include <stdlib.h>

#define M 10

void preencherAleatorio(int vet[], int numPos){
    int i;
    for(i = 0; i < numPos; i++)
        vet[i] = rand()%100;
}
void imprimirVetor(int vet[], int numPos){
       //...
}
int minimoVetor(int vetor[], int tamanhoVetor){
       int j = 0;
       int minimo = 0;
       while( j < tamanhoVetor - 1){
              j = j + 1;
              if(vetor[j] < vetor[minimo])
                     minimo = j;
       }
       return minimo;
}
int main(){
       int vetor[10];
       int idxMenor;

       srand(time(NULL));

       preencherAleatorio(vetor, M);
       imprimirVetor(vetor, M);
       idxMenor = minimoVetor(vetor, M);
       printf("%d %d\n", idxMenor, vetor[idxMenor]);
       return 0;
}
```

Valor mínimo

Agora, crie uma versão **recursiva** para esse algoritmo

Seja um vetor v indexado por $[a..b]$, com $a < b$, obtenha um índice $m \in [a..b]$ tal que $v[m] \leq v[j], \forall j \in [a..b]$.

Como são os problemas menores que vamos resolver?

Qual o caso base?

Valor mínimo

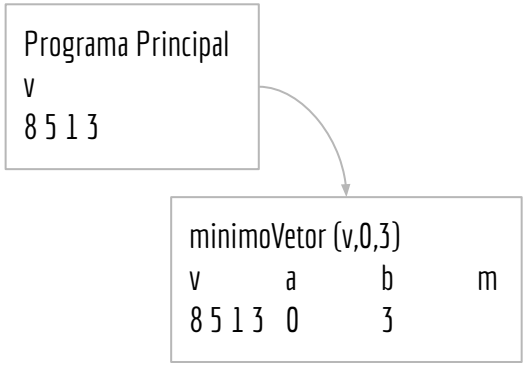
```
função minimoVetor (v,a,b)
entrada: vetor v indexado por [a..b], com a ≤ b
saída: m ∈ [a..b], tal que v[m] ≤ v[j] ∀j ∈ [a..b]
se a = b
    retorne a
m ← minimoVetor(v,a,b-1)
Se v[b] < v[m]
    m ← b
retorne m
```

Teste de mesa

Programa Principal
v
8 5 1 3

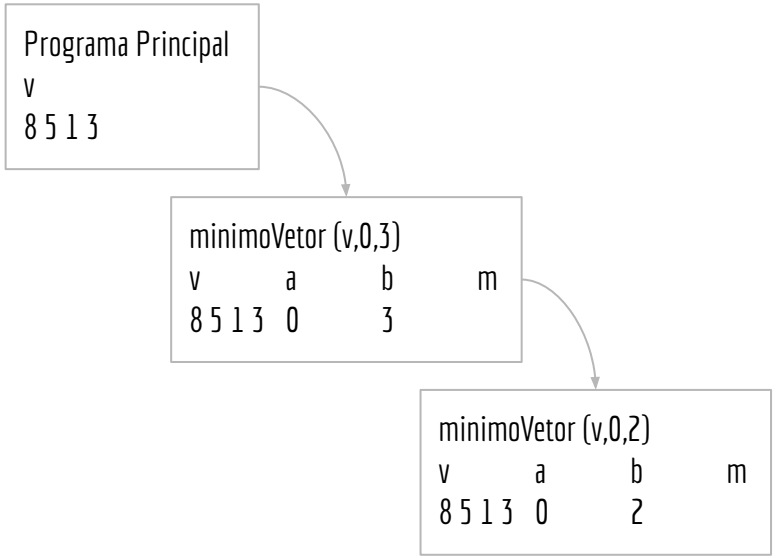
```
programa principal
v ← [8,5,1,3]
min ← minimoVetor(v,0,3)
imprima(v[min])

função minimoVetor (v,a,b)
entrada: vetor v indexado por [a..b], com a ≤ b
saída: m ∈ [a..b], tal que v[m] ≤ v[j] ∀j ∈ [a..b]
se a = b
    retorne a
m ← minimoVetor(v,a,b-1)
Se v[b] < v[m]
      m ← b
retorne m
```

Teste de mesa



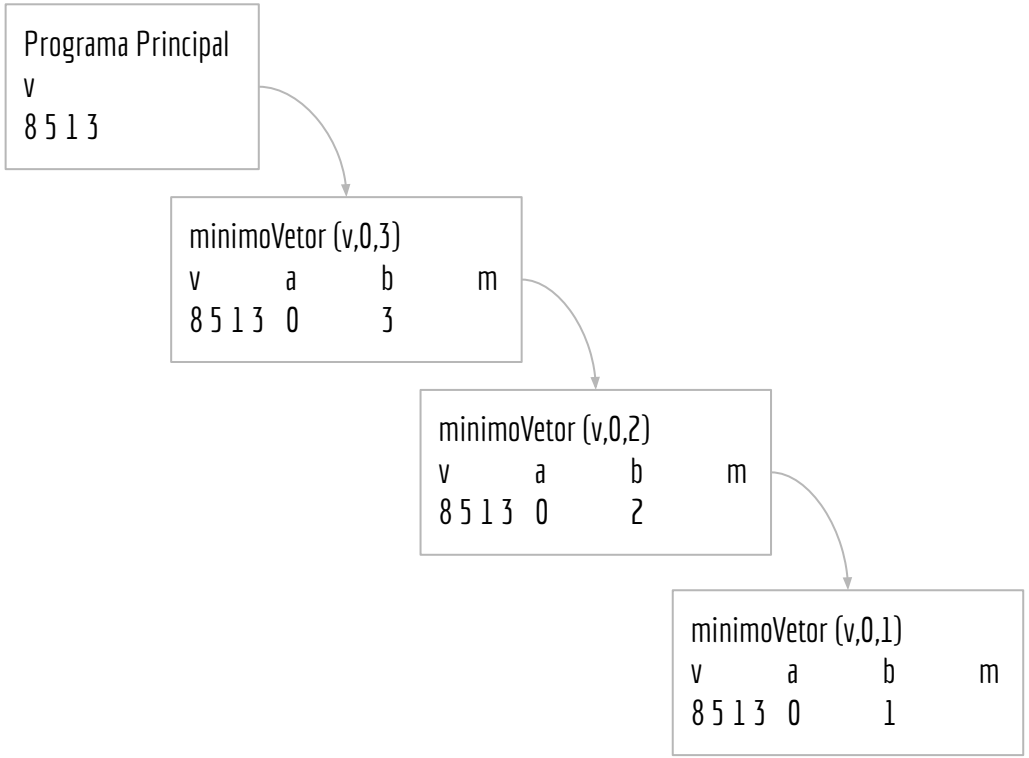

```
programa principal
v ← [8,5,1,3]
min ← minimoVetor(v,0,3)
imprima(v[min])

função minimoVetor (v,a,b)
entrada: vetor v indexado por [a..b], com a ≤ b
saída: m ∈ [a..b], tal que v[m] ≤ v[j] ∀j ∈ [a..b]
se a = b
    retorne a
m ← minimoVetor(v,a,b-1)
Se v[b] < v[m]
      m ← b
retorne m
```

Teste de mesa



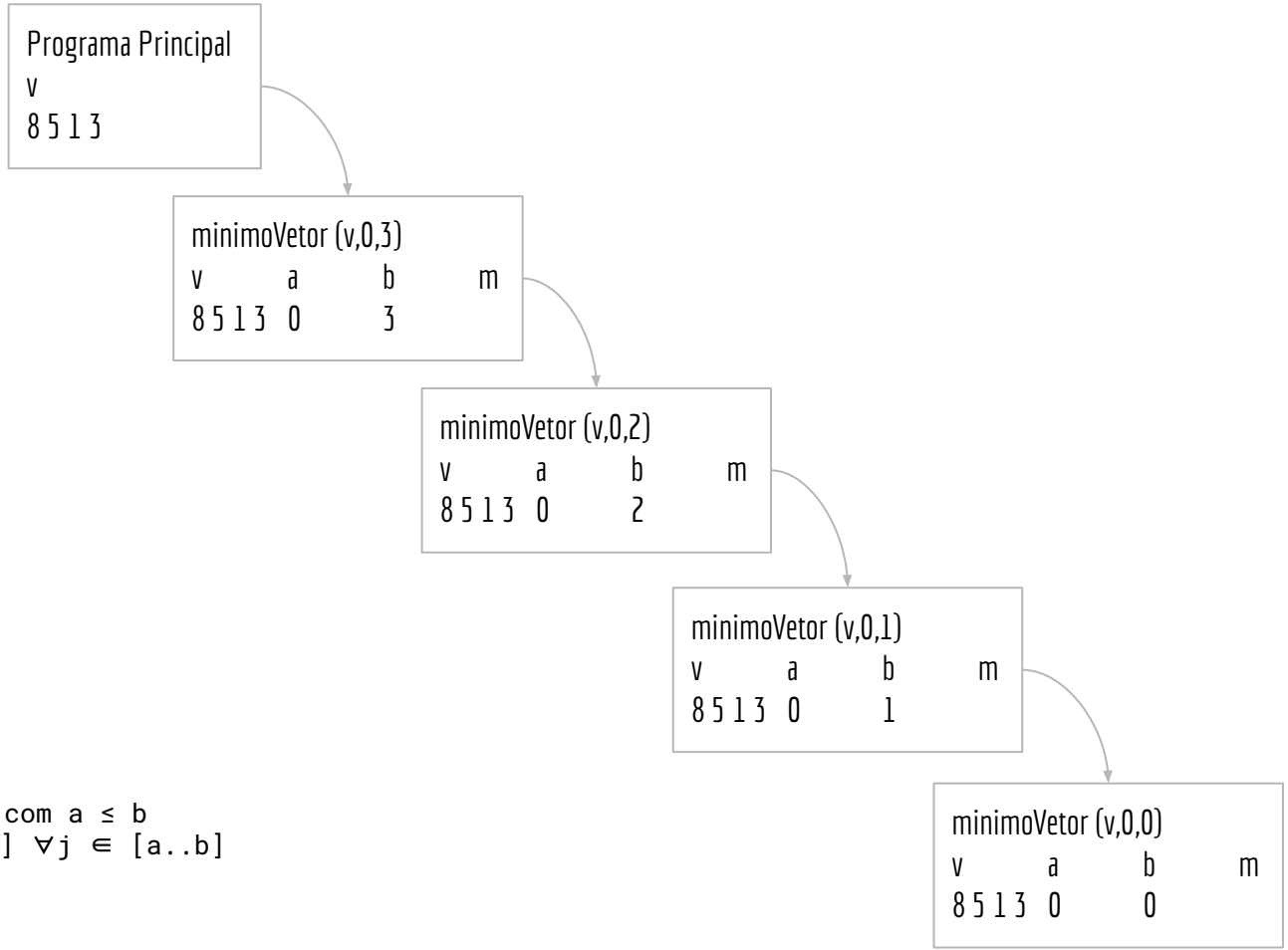

```
programa principal
v ← [8,5,1,3]
min ← minimoVetor(v,0,3)
imprima(v[min])

função minimoVetor (v,a,b)
entrada: vetor v indexado por [a..b], com a ≤ b
saída: m ∈ [a..b], tal que v[m] ≤ v[j] ∀j ∈ [a..b]
se a = b
    retorne a
m ← minimoVetor(v,a,b-1)
Se v[b] < v[m]
      m ← b
retorne m
```

Teste de mesa



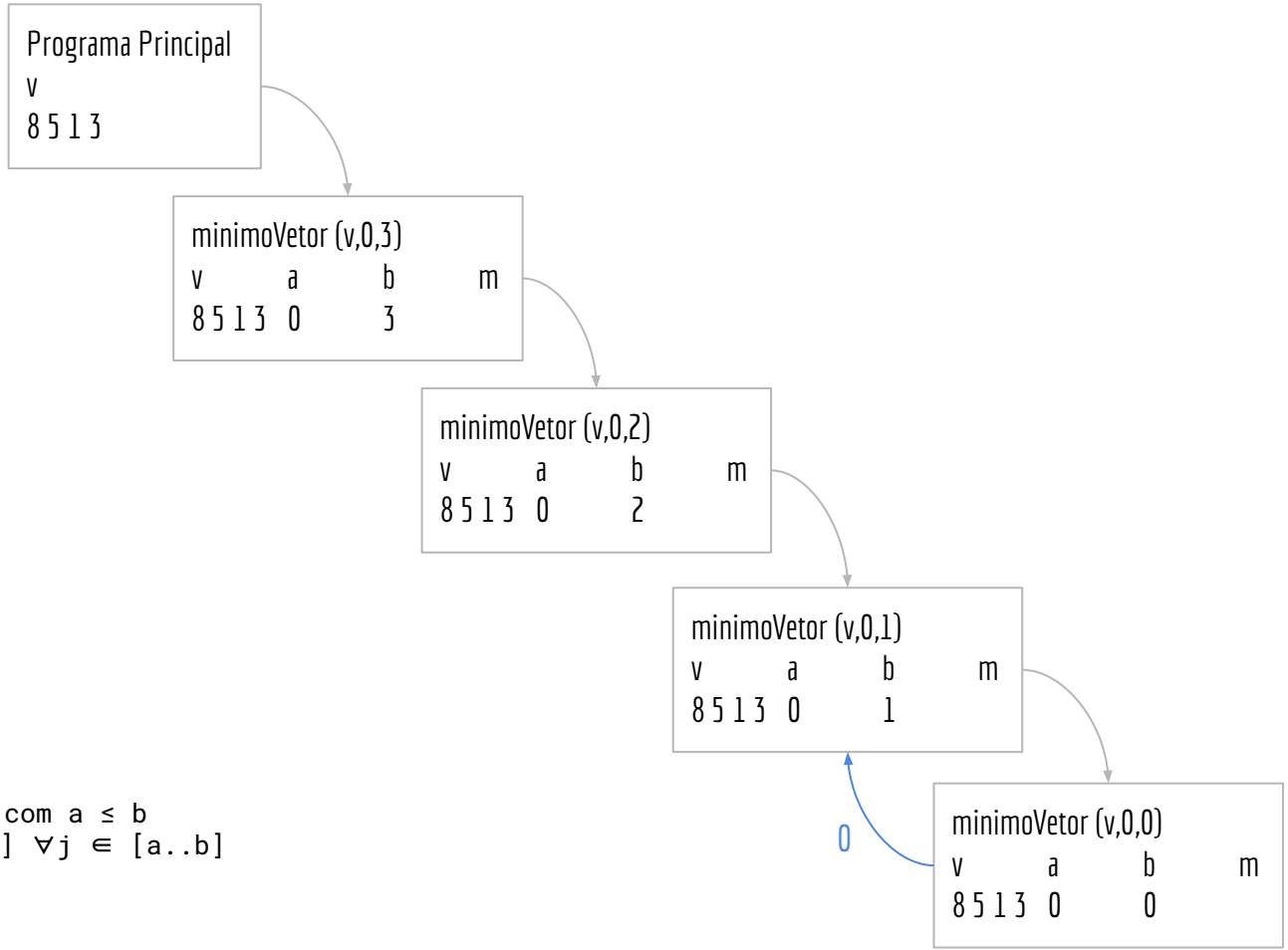

```
programa principal
v ← [8,5,1,3]
min ← minimoVetor(v,0,3)
imprima(v[min])

função minimoVetor (v,a,b)
entrada: vetor v indexado por [a..b], com a ≤ b
saída: m ∈ [a..b], tal que v[m] ≤ v[j] ∀j ∈ [a..b]
se a = b
    retorne a
m ← minimoVetor(v,a,b-1)
Se v[b] < v[m]
      m ← b
retorne m
```

Teste de mesa



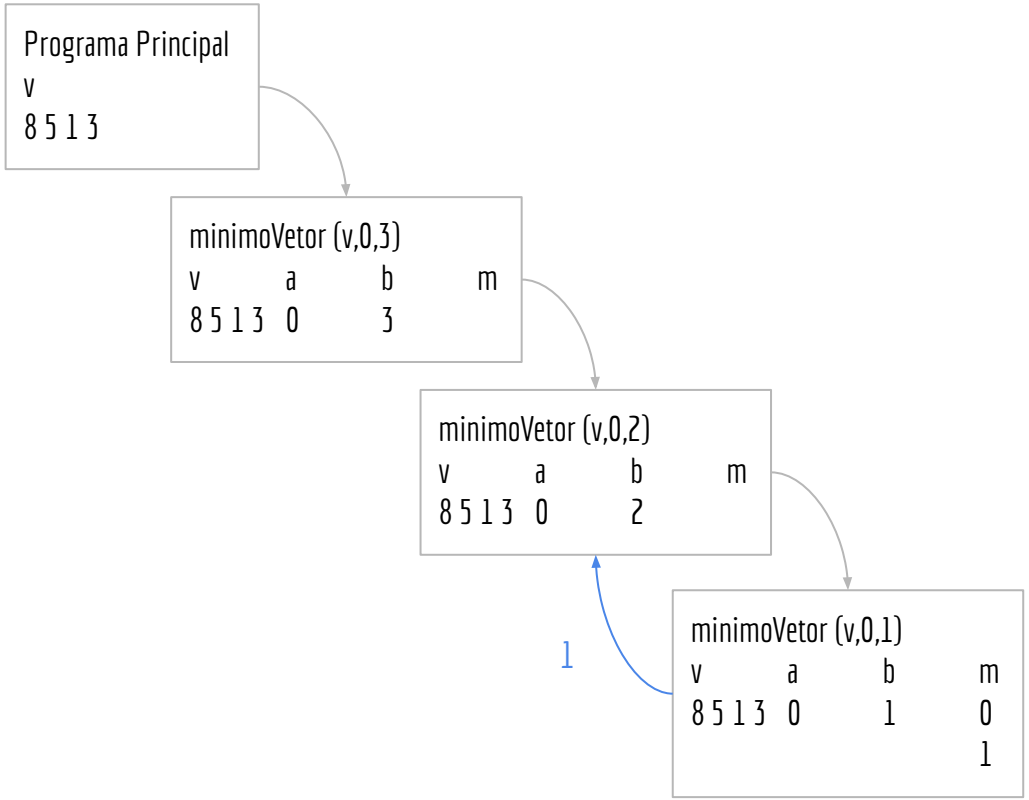

```
programa principal
v ← [8,5,1,3]
min ← minimoVetor(v,0,3)
imprima(v[min])

função minimoVetor (v,a,b)
entrada: vetor v indexado por [a..b], com a ≤ b
saída: m ∈ [a..b], tal que v[m] ≤ v[j] ∀j ∈ [a..b]
se a = b
    retorne a
m ← minimoVetor(v,a,b-1)
Se v[b] < v[m]
      m ← b
retorne m
```

Teste de mesa



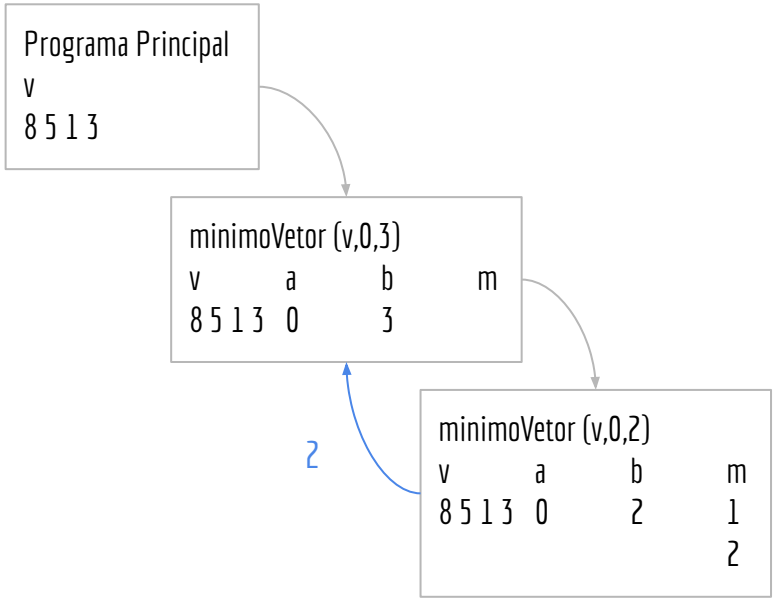

```
programa principal
v ← [8,5,1,3]
min ← minimoVetor(v,0,3)
imprima(v[min])

função minimoVetor (v,a,b)
entrada: vetor v indexado por [a..b], com a ≤ b
saída: m ∈ [a..b], tal que v[m] ≤ v[j] ∀j ∈ [a..b]
se a = b
    retorne a
m ← minimoVetor(v,a,b-1)
Se v[b] < v[m]
      m ← b
retorne m
```

Teste de mesa



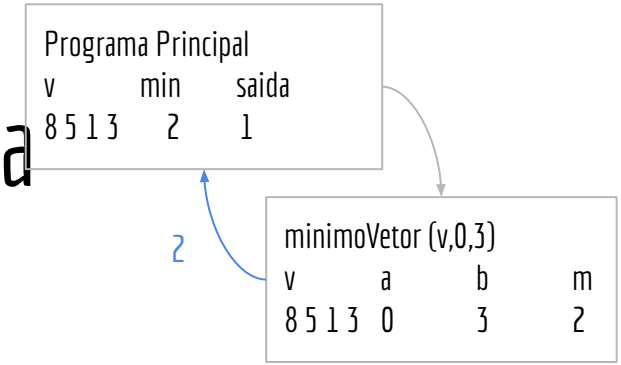

```
programa principal
v ← [8,5,1,3]
min ← minimoVetor(v,0,3)
imprima(v[min])

função minimoVetor (v,a,b)
entrada: vetor v indexado por [a..b], com a ≤ b
saída: m ∈ [a..b], tal que v[m] ≤ v[j] ∀j ∈ [a..b]
se a = b
    retorne a
m ← minimoVetor(v,a,b-1)
Se v[b] < v[m]
      m ← b
retorne m
```

Teste de mesa




```
programa principal
v ← [8,5,1,3]
min ← minimoVetor(v,0,3)
imprima(v[min])

função minimoVetor (v,a,b)
entrada: vetor v indexado por [a..b], com a ≤ b
saída: m ∈ [a..b], tal que v[m] ≤ v[j] ∀j ∈ [a..b]
se a = b
    retorne a
m ← minimoVetor(v,a,b-1)
Se v[b] < v[m]
      m ← b
retorne m
```

Teste de mesa




```
programa principal
v ← [8,5,1,3]
min ← minimoVetor(v,0,3)
imprima(v[min])

função minimoVetor (v,a,b)
entrada: vetor v indexado por [a..b], com a ≤ b
saída: m ∈ [a..b], tal que v[m] ≤ v[j] ∀j ∈ [a..b]
se a = b
    retorne a
m ← minimoVetor(v,a,b-1)
Se v[b] < v[m]
      m ← b
retorne m
```

Exercícios

1. Sabendo-se que $x^n = x*x^{n-1}$, crie o algoritmo recursivo, e também o implemente em C.

   função potência (x,n)
   entrada: $x \neq 0, x \in \mathbb{R}, n \in \mathbb{N}$
   saída: x^n

2. Crie um algoritmo recursivo para somar os elementos de um vetor. Também o implemente em C.

   função somaVetor (v,a,b)
   entrada: vetor v indexado por [a..b], com a ≤ b

   saída: $\sum_{i=a}^{b} v[i]$

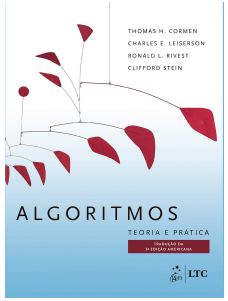
Exercícios

3. Sabendo-se que a função de Fibonacci $F : \mathbb{N} \mapsto \mathbb{N}$ pode ser definida pela seguinte fórmula, crie um algoritmo recursivo e o implemente em C.

$$F(n) = \begin{cases} n, & \text{se } n \leq 1, \\ F(n-1) + F(n-2), & \text{se } n > 1, \end{cases}$$

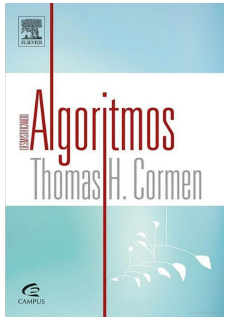
Referências

T. Cormen, C. Leiserson, R. Rivest, C. Stein. Algoritmos: Teoria e Prática. 3a ed. 2012

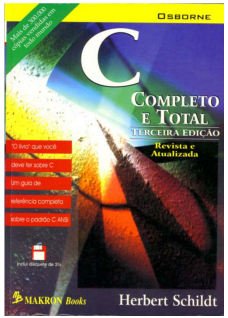


R. Sedgewick, K. Wayne. Algorithms Part I. 4a ed. 2014

T. Cormen. Desmistificando algoritmos. 2017.



H. Schildt. C completo e total. 1996



Licença