"Você pode aprender muito mais com um jogo que você perde do que com um jogo que você ganha" (José Raúl Capablanca).

# Removendo Recursão de Cauda

Paulo Ricardo Lisboa de Almeida

DSBD

UFPR
UNIVERSIDADE FEDERAL DO PARANÁ

# O custo de uma recursão

Toda recursão tem custos computacionais envolvidos

Quais são os custos (o que precisamos salvar na memória)?

| max-heapify | | | | |
|---|---|---|---|---|
| i | n | l | r | maior |
| 2 | 10 | 4 | 5 | 4 |

| max-heapify | | | | |
|---|---|---|---|---|
| i | n | l | r | maior |
| 4 | 10 | 8 | 9 | 9 |

| max-heapify | | | | |
|---|---|---|---|---|
| i | n | l | r | maior |
| 9 | 10 | 18 | 19 | 9 |

```
função max-heapify(h,i,n)
l ← esquerda(i)
r ← direita(i)
se l ≤ n e h[l] > h[i]
      maior ← l
senão
      maior ← i
se r ≤ n e h[r] > h[maior]
      maior ← r
se maior ≠ i
      trocar(h,i,maior)
      max-heapify(h,maior,n)
```

# O custo de uma recursão

Toda recursão tem custos computacionais envolvidos

**Cada chamada** recursiva é uma chamada de função

- Cada chamada tem suas próprias
  - Variáveis locais
  - “Linha do programa” sendo executada atualmente
  - Endereço de retorno (para onde retornar o resultado)
  - ...

# O custo de uma recursão

Toda recursão tem custos computacionais envolvidos

**Cada chamada** recursiva é uma chamada de função

Cada chamada tem suas próprias

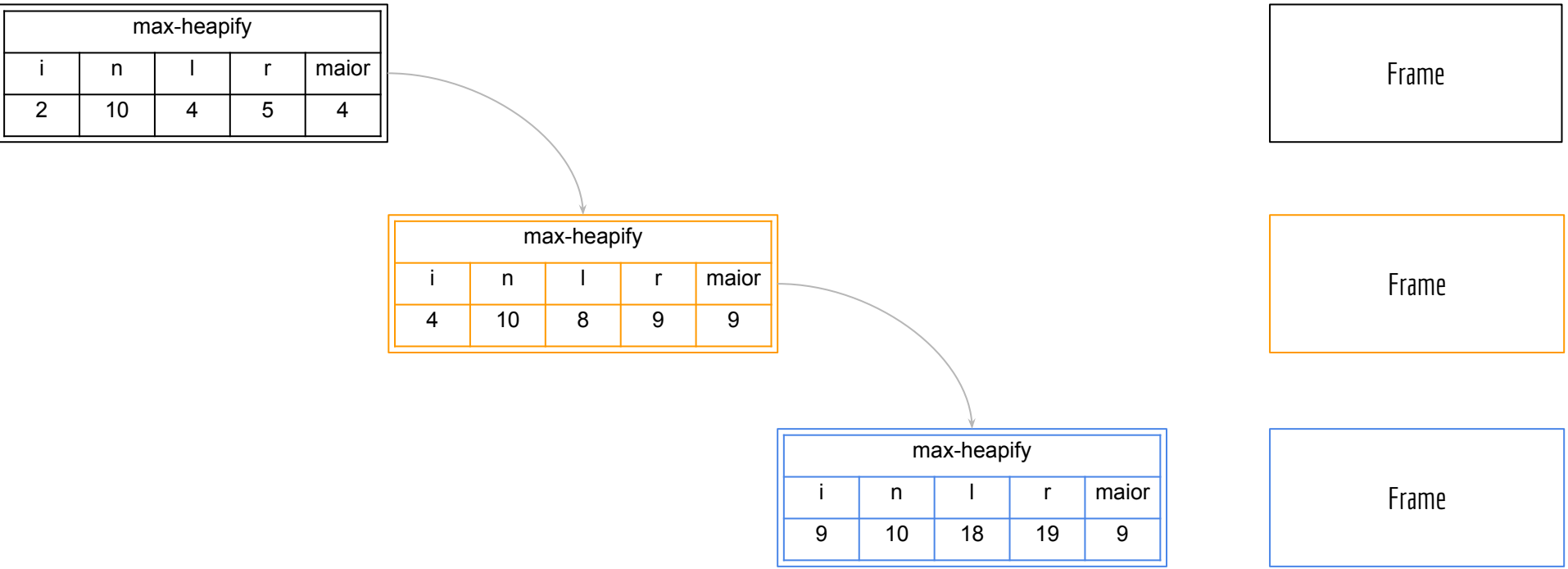
- Variáveis locais
- “Linha do programa” sendo executada atualmente
- Endereço de retorno (para onde retornar o resultado)
- ...



Chamamos isso de **Frame** (Quadro) da função. O frame da função existe enquanto a função não terminar de executar.

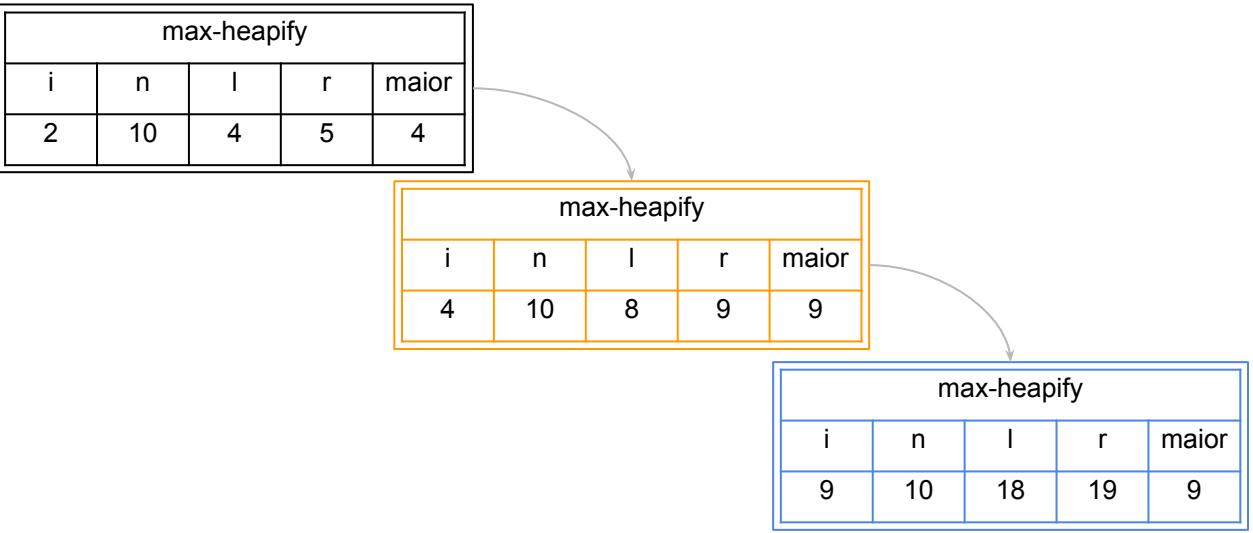
# Voltando ao exemplo

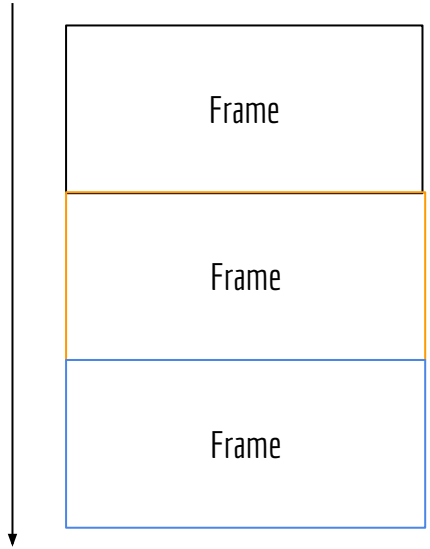
Memória



# Voltando ao exemplo

Ee olharmos “de ponta cabeça”, cada novo frame fica acima do anterior. Além disso, ou uma função termina de executar (e seu frame é eliminado), ou ela chama outra função, e o frame da função chamada fica acima do seu.
Que estrutura de dados é usada?



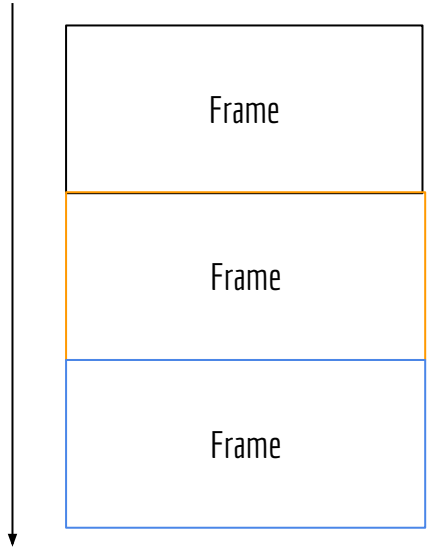
# Voltando ao exemplo

| max-heapify | | | | |
|---|---|---|---|---|
| i | n | l | r | maior |
| 2 | 10 | 4 | 5 | 4 |

| max-heapify | | | | |
|---|---|---|---|---|
| i | n | l | r | maior |
| 4 | 10 | 8 | 9 | 9 |

| max-heapify | | | | |
|---|---|---|---|---|
| i | n | l | r | maior |
| 9 | 10 | 18 | 19 | 9 |

Isso é uma pilha (de Frames).



# Recursão

Para usar recursão, precisamos então armazenar as informações de cada função sendo executada na memória em uma pilha de Frames

Mas até agora **não vimos essa pilha!**

A pilha é **gerenciada automaticamente pelo compilador.**

Mas você precisa saber que ela existe, e que existe um custo atrelado a ela.

Aprenda como o compilador gerencia essa pilha em disciplinas como:

- Arquitetura de computadores.
- Compiladores.
- Sistemas Operacionais.

# Removendo a recursão

Podemos aplicar técnicas para remover a recursão

Em alguns casos, é possível transformar a recursão em um algoritmo iterativo.

Em outros, podemos implementar o conceito de pilha e *Frame* manualmente.

Fazemos o trabalho do compilador

Pode (veja bem, **pode**) tornar o programa mais eficiente (em tempo e memória)

Dependendo do hardware, do programa, do programador, ...

# Recursão de Cauda

Uma **recursão de cauda** é caracterizada pela chamada recursiva apenas no final da função

Após a chamada recursiva, nenhuma outra instrução é executada

**Não vamos precisar de pilha!**

Reduzimos o custo de memória e processamento.

**Exemplos:**

```
função f1(a,b)
…
f1(c,d)
```

```
função f2(a,b)
se a > b
      f2(c,d)
senão
      f2(e,f)
```

# Recursão de Cauda

Para remover a recursão de cauda, faça o seguinte:

```
função f1(a,b)
se caso_base
    retorne resposta
senão
    operações_da_função
    retorne f1(x,y)
```

```
função f1Iterativa(a,b)
enquanto não caso_base
    operações_da_função
    a ← x
    b ← y

retorne resposta
```

# Exemplo - Fatorial

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



A chamada a fatorial no final implica ainda em uma multiplicação.
Mas podemos encaixar no framework com alguns ajustes.

# Exemplo - Fatorial

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

```
função fatorialIterativo (n)
entrada: inteiro n ≥ 0
saída: n!
fat ← n
enquanto n > 1
    n ← n-1
    fat ← fat*n
retorne fat
```

# Exemplo - Fatorial

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

Enquanto não caso base.

```
função fatorialIterativo (n)
entrada: inteiro n ≥ 0
saída: n!
fat ← n
enquanto n > 1
    n ← n-1
    fat ← fat*n
retorne fat
```

# Exemplo - Fatorial

Atribuir novo valor na variável
que era usada na chamada.

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

```
função fatorialIterativo (n)
entrada: inteiro n ≥ 0
saída: n!
fat ← n
enquanto n > 1
    n ← n-1
    fat ← fat*n
retorne fat
```

# Exemplo - Fatorial

Ajuste necessário por conta do n*fat...

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

```
função fatorialIterativo (n)
entrada: inteiro n ≥ 0
saída: n!
fat ← n
enquanto n > 1
    n ← n-1
    fat ← fat*n
retorne fat
```

# Faça você mesmo

Faça a versão iterativa do algoritmo de busca em vetor não ordenado. Utilize o framework para transformar.

```
função f1(a,b)
se caso_base
     retorne resposta
senão
     operações_da_função
     retorne f1(x,y)
```

```
função f1Iterativa(a,b)
enquanto não caso_base
     operações_da_função
     a ← x
     b ← y

retorne resposta
```

```
função busca(x,v,a,b)
entrada: vetor v indexado por [a..b], com a ≤ b, e um valor x a ser buscado
saída: m ∈ [a..b], tal que v[m] = x, ou nao se x não existe no vetor
se a > b
     retorne nao
se v[b] = x
     retorne b
retorne busca(x,v,a,b-1)
```

# Faça você mesmo

Faça a versão iterativa do algoritmo de busca em vetor não ordenado. Utilize o framework para transformar.

```
função buscarIter(x,v,a,b)
entrada: vetor v indexado por [a..b], com a ≤ b, e um valor x a ser buscado
saída: m ∈ [a..b], tal que v[m] = x, ou nao se x não existe no vetor
enquanto b ≥ a
     se v[b] = x
          retorne b
     b = b - 1
retorne nao
```

```
função busca(x,v,a,b)
entrada: vetor v indexado por [a..b], com a ≤ b, e um valor x a ser buscado
saída: m ∈ [a..b], tal que v[m] = x, ou nao se x não existe no vetor
se a > b
     retorne nao
se v[b] = x
     retorne b
retorne busca(x,v,a,b-1)
```

# Faça você mesmo

Faça uma versão iterativa da busca binária.

```
função buscaBinaria (x,v,a,b)
entrada: vetor v ordenado, indexado por [a..b], com a ≤ b, e um valor x a ser buscado
saída: o menor m ∈ [a-1..b], tal que x < v[i] ∀i ∈ [m+1..b]
se a > b
     retorne a-1

m ← ⌊(a + b)/2⌋

se x < v[m]
     retorne buscaBinaria (x,v,a,m-1)
retorne buscaBinaria (x,v,m+1,b)
```

# Faça você mesmo

```
função buscaBinariaIterativa (x,v,a,b)
enquanto b ≥ a
      m ← ⌊(a + b)/2⌋

      se x < v[m]
            b ← m-1
      senão
            a ← m+1
retorne a-1
```

```
função buscaBinaria (x,v,a,b)
entrada: vetor v ordenado, indexado por [a..b], com a ≤ b, e um valor x a ser buscado
saída: o menor m ∈ [a-1..b], tal que x < v[i] ∀i ∈ [m+1..b]
se a > b
      retorne a-1

m ← ⌊(a + b)/2⌋

se x < v[m]
      retorne buscaBinaria (x,v,a,m-1)
retorne buscaBinaria (x,v,m+1,b)
```

# Dica de programação

Ao remover a recursão de cauda, geramos um algoritmo iterativo que **não requer uma estrutura de pilha**.

Economia de memória e processamento

Especialmente em algoritmos lineares ($f(n) = k.n, \forall\ k \in \mathbb{N}$) ou de custo superior

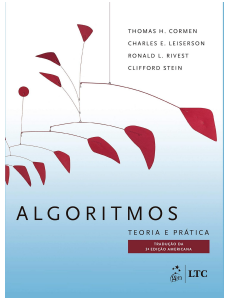
Então nesse caso, **prefira a versão iterativa**, exceto se você tiver alguma razão forte para usar o recursivo

# Exercícios

1. Encontre os demais algoritmos estudados na disciplina que possuem recursão de Cauda (e.g., Busca Ingênua e Selection Sort) e os transforme em algoritmos iterativos.

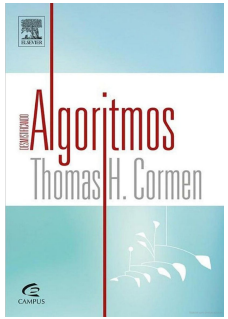
# Referências


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

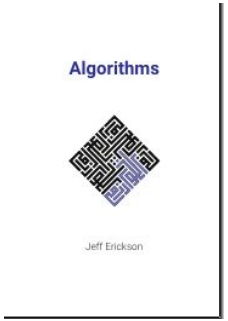



R. Sedgewick, K. Wayne. Algorithms Part I. 4a ed. 2014


T. Cormen. Desmistificando algoritmos. 2017.




J. Erickson. Algorithms. 2019.



# Licença