# Aula 03 – Técnicas de Desenvolvimento de Algoritmos: Indução e Recursão

Norton Trevisan Roman
norton@usp.br

1 de outubro de 2018

# Indução e Algoritmos

- Vimos que podemos provar que um determinado problema, definido no conjunto dos números inteiros positivos, possui solução.

# Indução e Algoritmos

- Vimos que podemos provar que um determinado problema, definido no conjunto dos números inteiros positivos, possui solução.

## O que é mesmo um algoritmo?

# Indução e Algoritmos

- Vimos que podemos provar que um determinado problema, definido no conjunto dos números inteiros positivos, possui solução.

**O que é mesmo um algoritmo?**

Uma sequência de passos a serem efetuados para que se obtenha uma solução para um determinado problema

# Indução e Algoritmos

- Vimos que podemos provar que um determinado problema, definido no conjunto dos números inteiros positivos, possui solução.

## O que é mesmo um algoritmo?

Uma sequência de passos a serem efetuados para que se obtenha uma solução para um determinado problema

- Então, por extensão, podemos provar que um determinado algoritmo, cuja entrada sejam números inteiros positivos, estará correto para todo inteiro positivo

# Indução e Algoritmos

- De fato, podemos usar a prova por indução em qualquer afirmação que contenha uma variável que possa assumir valores inteiros arbitrários e não-negativos

# Indução e Algoritmos

- De fato, podemos usar a prova por indução em qualquer afirmação que contenha uma variável que possa assumir valores inteiros arbitrários e não-negativos

E o  ?

# Indução e Algoritmos

- De fato, podemos usar a prova por indução em qualquer afirmação que contenha uma variável que possa assumir valores inteiros arbitrários e não-negativos

E o  ?

- Precisamos já ter o algoritmo para provar sua corretude

# Indução e Algoritmos

- De fato, podemos usar a prova por indução em qualquer afirmação que contenha uma variável que possa assumir valores inteiros arbitrários e não-negativos

E o  ?

- Precisamos já ter o algoritmo para provar sua corretude
- Então como isso me ajuda a criar o algoritmo?

# Indução e Algoritmos

- Ocorre que a prova matemática via indução finita é, ela própria, um algoritmo recursivo

# Indução e Algoritmos

- Ocorre que a prova matemática via indução finita é, ela própria, um algoritmo recursivo
- Algoritmo recursivo?

# Indução e Algoritmos

- Ocorre que a prova matemática via indução finita é, ela própria, um algoritmo recursivo
- Algoritmo recursivo?
- Vejamos o que diz o dicionário Fakess para a língua portuguesa:

# Indução e Algoritmos

- Ocorre que a prova matemática via indução finita é, ela própria, um algoritmo recursivo
- Algoritmo recursivo?
- Vejamos o que diz o dicionário Fakess para a língua portuguesa:

**Definição**

**Recursão**: Se não entendeu, vide Recursão.

# Recursão

## Definição

Um método é chamado de recursivo quando chama a sim mesmo, direta ou indiretamente.

# Recursão

## Definição

Um método é chamado de recursivo quando chama a sim mesmo, direta ou indiretamente.

## E qual a vantagem disso?

# Recursão

## Definição

Um método é chamado de recursivo quando chama a sim mesmo, direta ou indiretamente.

## E qual a vantagem disso?

- Em geral, a recursividade permite uma descrição mais clara e concisa do algoritmo

# Recursão

## Definição

Um método é chamado de recursivo quando chama a sim mesmo, direta ou indiretamente.

## E qual a vantagem disso?

- Em geral, a recursividade permite uma descrição mais clara e concisa do algoritmo
- Métodos recursivos baseiam-se no princípio da Indução Finita
  - São escolhas naturais para algoritmos definidos assim

# Indução e Recursão

- E como isso nos ajuda a construir um algoritmo com indução finita?

# Indução e Recursão

- E como isso nos ajuda a construir um algoritmo com indução finita?
- Tomemos um exemplo: o cálculo de um fatorial

# Indução e Recursão

- E como isso nos ajuda a construir um algoritmo com indução finita?
- Tomemos um exemplo: o cálculo de um fatorial
  - Sabemos que $\mathit{fatorial}(0) = 1$

# Indução e Recursão

- E como isso nos ajuda a construir um algoritmo com indução finita?
- Tomemos um exemplo: o cálculo de um fatorial
  - Sabemos que $fatorial(0) = 1$
  - Vamos assumir que temos magicamente um método que calcule o fatorial para $n$: `fatorial(n)`

# Indução e Recursão

- E como isso nos ajuda a construir um algoritmo com indução finita?
- Tomemos um exemplo: o cálculo de um fatorial
  - Sabemos que $fatorial(0) = 1$
  - Vamos assumir que temos magicamente um método que calcule o fatorial para $n$: `fatorial(n)`
  - Conseguimos então calcular o fatorial de $n + 1$, a partir de `fatorial(n)`?

# Indução e Recursão

- E como isso nos ajuda a construir um algoritmo com indução finita?
- Tomemos um exemplo: o cálculo de um fatorial
  - Sabemos que $fatorial(0) = 1$
  - Vamos assumir que temos magicamente um método que calcule o fatorial para $n$: `fatorial(n)`
  - Conseguimos então calcular o fatorial de $n+1$, a partir de `fatorial(n)`?

    Sim, basta fazer `(n+1) * fatorial(n)`

# Indução e Recursão

- E como isso nos ajuda a construir um algoritmo com indução finita?
- Tomemos um exemplo: o cálculo de um fatorial
  - Sabemos que $\mathit{fatorial}(0) = 1$
  - Vamos assumir que temos magicamente um método que calcule o fatorial para $n$: `fatorial(n)`
  - Conseguimos então calcular o fatorial de $n+1$, a partir de `fatorial(n)`?

    Sim, basta fazer `(n+1) * fatorial(n)`

Base

# Indução e Recursão

- E como isso nos ajuda a construir um algoritmo com indução finita?
- Tomemos um exemplo: o cálculo de um fatorial
  - Sabemos que $fatorial(0) = 1$
  - Vamos assumir que temos magicamente um método que calcule o fatorial para $n$: `fatorial(n)`
  - Conseguimos então calcular o fatorial de $n + 1$, a partir de `fatorial(n)`?

    Sim, basta fazer `(n+1) * fatorial(n)`

Hipótese de Indução

# Indução e Recursão

- E como isso nos ajuda a construir um algoritmo com indução finita?
- Tomemos um exemplo: o cálculo de um fatorial
  - Sabemos que $fatorial(0) = 1$
  - Vamos assumir que temos magicamente um método que calcule o fatorial para $n$: `fatorial(n)`
  - Conseguimos então calcular o fatorial de $n + 1$, a partir de `fatorial(n)`?

    Sim, basta fazer `(n+1) * fatorial(n)` ← Passo

# Indução e Recursão

## Então...

```
long fatorial(long n) {
    if (n==0) return(1);
    return(n * fatorial(n-1));
}
```

# Indução e Recursão

## Então...

```
long fatorial(long n) {
    if (n==0) return(1);
    return(n * fatorial(n-1));
}
```

Base
($fatorial(0) = 1$)

# Indução e Recursão

## Então...

```
long fatorial(long n) {
    if (n==0) return(1);
    return(n * fatorial(n-1));
}
```

Hipótese de Indução

# Indução e Recursão

## Então...

```
long fatorial(long n) {
    if (n==0) return(1);
    return(n * fatorial(n-1));
}
```

Passo

# Indução e Recursão

## Então...

```
long fatorial(long n) {
    if (n==0) return(1);
    return(n * fatorial(n-1));
}
```

Hipótese de Indução

- Note que aqui usamos $n - 1$ em lugar de $n$ na hipótese de indução. O resultado é o mesmo.

# Implementação de Recursão

## E como o compilador implementa a recursão?

# Implementação de Recursão

## E como o compilador implementa a recursão?

- Por meio da pilha de execução

# Implementação de Recursão

## E como o compilador implementa a recursão?

- Por meio da pilha de execução
- A cada chamada de método, são empilhados
  - Seus atributos
  - Suas variáveis locais
  - Seu endereço de retorno (a quem chamou o método)

# Implementação de Recursão

## E como o compilador implementa a recursão?

- Por meio da pilha de execução
- A cada chamada de método, são empilhados
  - Seus atributos
  - Suas variáveis locais
  - Seu endereço de retorno (a quem chamou o método)
- Quando o método termina, esses dados são desempilhados

# Implementação de Recursão

O que acontece ao fazermos `fatorial(3)`?

```
long fatorial(long n) {
  if (n==0) return(1);
  return(n *
       fatorial(n-1));
}
```

# Implementação de Recursão

O que acontece ao fazermos `fatorial(3)`?

```
long fatorial(long n) {
  if (n==0) return(1);
  return(n *
       fatorial(n-1));
}
```

| | |
|---|---|
| n: 3 | retorna: ? |

Ao chamarmos `fatorial(3)`, o método é empilhado

# Implementação de Recursão

O que acontece ao fazermos `fatorial(3)`?

```
long fatorial(long n) {
  if (n==0) return(1);
  return(n *
       fatorial(n-1));
}
```

| n: 3 | retorna: ? |
|---|---|

Não é o caso base

# Implementação de Recursão

O que acontece ao fazermos `fatorial(3)`?

```
long fatorial(long n) {
  if (n==0) return(1);
  return(n *
      fatorial(n-1));
}
```

| | |
|---|---|
| n: 2 | retorna: ? |
| n: 3 | retorna: ? |

Então nova chamada é feita e empilhada

# Implementação de Recursão

O que acontece ao fazermos `fatorial(3)`?

```
long fatorial(long n) {
  if (n==0) return(1);
  return(n *
       fatorial(n-1));
}
```

| | |
|---|---|
| n: 2 | retorna: ? |
| n: 3 | retorna: ? |

Não é o caso base

# Implementação de Recursão

O que acontece ao fazermos `fatorial(3)`?

```
long fatorial(long n) {
  if (n==0) return(1);
  return(n *
       fatorial(n-1));
}
```

| | |
|---|---|
| n: 1 | retorna: ? |
| n: 2 | retorna: ? |
| n: 3 | retorna: ? |

Então nova chamada é feita e empilhada

# Implementação de Recursão

O que acontece ao fazermos `fatorial(3)`?

```
long fatorial(long n) {
  if (n==0) return(1);
  return(n *
        fatorial(n-1));
}
```

| | |
|---|---|
| n: 1 | retorna: ? |
| n: 2 | retorna: ? |
| n: 3 | retorna: ? |

Não é o caso base

# Implementação de Recursão

O que acontece ao fazermos `fatorial(3)`?

```
long fatorial(long n) {
  if (n==0) return(1);
  return(n *
       fatorial(n-1));
}
```

| | |
|---|---|
| n: 0 | retorna: ? |
| n: 1 | retorna: ? |
| n: 2 | retorna: ? |
| n: 3 | retorna: ? |

Então nova chamada é feita e empilhada

# Implementação de Recursão

O que acontece ao fazermos `fatorial(3)`?

```
long fatorial(long n) {
  if (n==0) return(1);
  return(n *
       fatorial(n-1));
}
```

| | |
|---|---|
| n: 0 | retorna: 1 |
| n: 1 | retorna: ? |
| n: 2 | retorna: ? |
| n: 3 | retorna: ? |

É o caso base!

# Implementação de Recursão

O que acontece ao fazermos `fatorial(3)`?

```
long fatorial(long n) {
  if (n==0) return(1);
  return(n *
       fatorial(n-1));
}
```

| | |
|---|---|
| n: 1 | retorna: ? |
| n: 2 | retorna: ? |
| n: 3 | retorna: ? |

Desempilha, retornando 1

# Implementação de Recursão

O que acontece ao fazermos `fatorial(3)`?

```
long fatorial(long n) {
  if (n==0) return(1);
  return(n *
        fatorial(n-1));
}
```

| | |
|---|---|
| n: 1 | retorna: 1 |
| n: 2 | retorna: ? |
| n: 3 | retorna: ? |

Faz 1 ∗ 1

# Implementação de Recursão

O que acontece ao fazermos `fatorial(3)`?

```
long fatorial(long n) {
  if (n==0) return(1);
  return(n *
       fatorial(n-1));
}
```

| | |
|---|---|
| n: 2 | retorna: ? |
| n: 3 | retorna: ? |

Desempilha, retornando 1

# Implementação de Recursão

O que acontece ao fazermos `fatorial(3)`?

```
long fatorial(long n) {
  if (n==0) return(1);
  return(n *
       fatorial(n-1));
}
```

| | |
|---|---|
| n: 2 | retorna: 2 |
| n: 3 | retorna: ? |

Faz 2 ∗ 1

# Implementação de Recursão

O que acontece ao fazermos `fatorial(3)`?

```
long fatorial(long n) {
  if (n==0) return(1);
  return(n *
        fatorial(n-1));
}
```

| | |
|---|---|
| n: 3 | retorna: ? |

Desempilha, retornando 2

# Implementação de Recursão

O que acontece ao fazermos `fatorial(3)`?

```
long fatorial(long n) {
  if (n==0) return(1);
  return(n *
       fatorial(n-1));
}
```

| | |
|---|---|
| n: 3 | retorna: 6 |

Faz 3 ∗ 2

# Implementação de Recursão

O que acontece ao fazermos `fatorial(3)`?

```
long fatorial(long n) {
  if (n==0) return(1);
  return(n *
        fatorial(n-1));
}
```

Desempilha, retornando 6

# Implementação de Recursão

Note que, a cada chamada recursiva, criamos cópias distintas do método na pilha

```
long fatorial(long n) {
  if (n==0) return(1);
  return(n *
       fatorial(n-1));
}
```

| | |
|---|---|
| n: 0 | retorna: ? |
| n: 1 | retorna: ? |
| n: 2 | retorna: ? |
| n: 3 | retorna: ? |

Recursão pode ser bastante custosa!

# Implementação de Recursão

- E se esquecermos do caso base?

```
long fatorial(long n) {
  if (n==0) return(1);
  return(n *
          fatorial(n-1));
}
```

# Implementação de Recursão

- E se esquecermos do caso base?

```
long fatorial(long n) {

  return(n *
         fatorial(n-1));
}
```

# Implementação de Recursão

- E se esquecermos do caso base?
- Serão feitas chamadas recursivas, até o limite do segmento de memória alocado à pilha

```
long fatorial(long n) {

  return(n *
         fatorial(n-1));
}
```

# Implementação de Recursão

- E se esquecermos do caso base?
- Serão feitas chamadas recursivas, até o limite do segmento de memória alocado à pilha

```
long fatorial(long n) {
  return(n *
         fatorial(n-1));
}
```

- E teremos então um estouro de pilha
  - Stack overflow

# Implementação de Recursão

- E qual a diferença entre uma chamada recursiva e uma comum?

# Implementação de Recursão

- E qual a diferença entre uma chamada recursiva e uma comum? Nenhuma!

# Implementação de Recursão

- E qual a diferença entre uma chamada recursiva e uma comum? Nenhuma!
- Considere o código ao lado

```
void m1() {
  c1;
  m2();
  c2;
}

void m2() {
  c3;
  m3();
  c4;
}

void m3() {
  c5;
}
```

# Implementação de Recursão

- E qual a diferença entre uma chamada recursiva e uma comum? Nenhuma!
- Considere o código ao lado
  - Ao chamarmos `m1()`, este é empilhado

| |
|---|
| m1 |

```
void m1() {
  c1;
  m2();
  c2;
}

void m2() {
  c3;
  m3();
  c4;
}

void m3() {
  c5;
}
```

# Implementação de Recursão

- E qual a diferença entre uma chamada recursiva e uma comum? Nenhuma!
- Considere o código ao lado
  - Ao chamarmos `m1()`, este é empilhado

| m1 |
|---|

```
void m1() {
  c1;
  m2();
  c2;
}

void m2() {
  c3;
  m3();
  c4;
}

void m3() {
  c5;
}
```

# Implementação de Recursão

- E qual a diferença entre uma chamada recursiva e uma comum? Nenhuma!
- Considere o código ao lado
  - O mesmo ocorre com `m2()`

```
void m1() {
   c1;
   m2();
   c2;
}

void m2() {
   c3;
   m3();
   c4;
}

void m3() {
   c5;
}
```

| |
|---|
| m2 |
| m1 |

# Implementação de Recursão

- E qual a diferença entre uma chamada recursiva e uma comum? Nenhuma!
- Considere o código ao lado
  - O mesmo ocorre com `m2()`

```
void m1() {
  c1;
  m2();
  c2;
}

void m2() {
  c3;
  m3();
  c4;
}

void m3() {
  c5;
}
```

| |
|---|
| m2 |
| m1 |

# Implementação de Recursão

- E qual a diferença entre uma chamada recursiva e uma comum? Nenhuma!
- Considere o código ao lado
  - E `m3()`

| |
|---|
| m3 |
| m2 |
| m1 |

```
void m1() {
  c1;
  m2();
  c2;
}

void m2() {
  c3;
  m3();
  c4;
}

void m3() {
  c5;
}
```

# Implementação de Recursão

- E qual a diferença entre uma chamada recursiva e uma comum? Nenhuma!
- Considere o código ao lado
  - E m3()

| |
|---|
| m3 |
| m2 |
| m1 |

```
void m1() {
  c1;
  m2();
  c2;
}

void m2() {
  c3;
  m3();
  c4;
}

void m3() {
  c5;
}
```

# Implementação de Recursão

- E qual a diferença entre uma chamada recursiva e uma comum? Nenhuma!
- Considere o código ao lado
  - Ao terminar, `m3()` é desempilhado

| |
|---|
| m2 |
| m1 |

```
void m1() {
  c1;
  m2();
  c2;
}

void m2() {
  c3;
  m3();
  c4;
}

void m3() {
  c5;
}
```

# Implementação de Recursão

- E qual a diferença entre uma chamada recursiva e uma comum? Nenhuma!
- Considere o código ao lado
  - Ao terminar, `m3()` é desempilhado

| |
|---|
| m2 |
| m1 |

```
void m1() {
  c1;
  m2();
  c2;
}

void m2() {
  c3;
  m3();
  c4;
}

void m3() {
  c5;
}
```

# Implementação de Recursão

- E qual a diferença entre uma chamada recursiva e uma comum? Nenhuma!
- Considere o código ao lado
  - E `m2()`

```
void m1() {
  c1;
  m2();
  c2;
}

void m2() {
  c3;
  m3();
  c4;
}

void m3() {
  c5;
}
```

| |
|---|
| m1 |

# Implementação de Recursão

- E qual a diferença entre uma chamada recursiva e uma comum? Nenhuma!
- Considere o código ao lado
  - E `m2()`

```
void m1() {
  c1;
  m2();
  c2;
}

void m2() {
  c3;
  m3();
  c4;
}

void m3() {
  c5;
}
```

| m1 |
|---|

# Implementação de Recursão

- E qual a diferença entre uma chamada recursiva e uma comum? Nenhuma!
- Considere o código ao lado
  - E finalmente `m1()`

```
void m1() {
  c1;
  m2();
  c2;
}

void m2() {
  c3;
  m3();
  c4;
}

void m3() {
  c5;
}
```

# Implementação de Recursão

- E se o método for recursivo?

```
void m1() {
  if (condição) {
    c1;
    m1();
    c2;
  }
}
```

# Implementação de Recursão

- E se o método for recursivo?
- Ocorre a mesma coisa...

```
void m1() {
  if (condição) {
    c1;
    m1();
    c2;
  }
}
```

# Implementação de Recursão

- E se o método for recursivo?
- Ocorre a mesma coisa...
  - Ao chamarmos `m1()`, este é empilhado

```
void m1() {
  if (condição) {
    c1;
    m1();
    c2;
  }
}
```

m1

# Implementação de Recursão

- E se o método for recursivo?
- Ocorre a mesma coisa...
  - Ao chamarmos `m1()`, este é empilhado

```
void m1() {
  if (condição) {
    c1;
    m1();
    c2;
  }
}
```

| m1 |
|---|

# Implementação de Recursão

- E se o método for recursivo?
- Ocorre a mesma coisa...
  - Ao chamarmos `m1()`, este é empilhado

```
void m1() {
  if (condição) {
    c1;
    m1();
    c2;
  }
}
```

m1

# Implementação de Recursão

- E se o método for recursivo?
- Ocorre a mesma coisa...
  - Mais uma vez...

```
void m1() {
  if (condição) {
    c1;
    m1();
    c2;
  }
}
```

# Implementação de Recursão

- E se o método for recursivo?
- Ocorre a mesma coisa...
  - Mais uma vez...

```
void m1() {
  if (condição) {
    c1;
    m1();
    c2;
  }
}
```

| |
|---|
| m1 |
| m1 |

# Implementação de Recursão

- E se o método for recursivo?
- Ocorre a mesma coisa...
  - Mais uma vez...

```
void m1() {
  if (condição) {
    c1;
    m1();
    c2;
  }
}
```

| |
|---|
| m1 |
| m1 |

# Implementação de Recursão

- E se o método for recursivo?
- Ocorre a mesma coisa...
  - E mais uma...

```
void m1() {
  if (condição) {
    c1;
    m1();
    c2;
  }
}
```

| |
|---|
| m1 |
| m1 |
| m1 |

# Implementação de Recursão

- E se o método for recursivo?
- Ocorre a mesma coisa...
  - Até que a condição ser falsa

```
void m1() {
  if (condição) {
    c1;
    m1();
    c2;
  }
}
```

| |
|---|
| m1 |
| m1 |
| m1 |

# Implementação de Recursão

- E se o método for recursivo?
- Ocorre a mesma coisa...
  - E um retorne, sendo desempilhado

```
void m1() {
  if (condição) {
    c1;
    m1();
    c2;
  }
}
```

| |
|---|
| m1 |
| m1 |

# Implementação de Recursão

- E se o método for recursivo?
- Ocorre a mesma coisa...
  - E um retorne, sendo desempilhado

```
void m1() {
  if (condição) {
    c1;
    m1();
    c2;
  }
}
```

# Implementação de Recursão

- E se o método for recursivo?
- Ocorre a mesma coisa...
  - E outro

```
void m1() {
  if (condição) {
    c1;
    m1();
    c2;
  }
}
```

m1

# Implementação de Recursão

- E se o método for recursivo?
- Ocorre a mesma coisa...
  - E outro

```
void m1() {
  if (condição) {
    c1;
    m1();
    c2;
  }
}
```

m1

# Implementação de Recursão

- E se o método for recursivo?
- Ocorre a mesma coisa...
  - Até a chamada original

```
void m1() {
  if (condição) {
    c1;
    m1();
    c2;
  }
}
```

# Implementação de Recursão

- E se o método for recursivo?
- Ocorre a mesma coisa...
  - Até a chamada original

```
void m1() {
  if (condição) {
    c1;
    m1();
    c2;
  }
}
```

- São todas chamadas distintas. A única diferença é que na recursão, elas possuem o mesmo nome

## Sequências

- Uma sequência S é uma lista de objetos que são enumerados <u>segundo alguma ordem</u>
  - Existe um primeiro objeto, um segundo etc

# Recursão e Sequências

## Sequências

- Uma sequência S é uma lista de objetos que são enumerados segundo alguma ordem
  - Existe um primeiro objeto, um segundo etc
- Uma sequência é definida recursivamente explicitando-se seu primeiro valor (ou primeiros valores) e, a partir daí, definindo-se outros valores em termos desses iniciais.

# Recursão e Sequências

## Sequências

- Uma sequência S é uma lista de objetos que são enumerados segundo alguma ordem
  - Existe um primeiro objeto, um segundo etc
- Uma sequência é definida recursivamente explicitando-se seu primeiro valor (ou primeiros valores) e, a partir daí, definindo-se outros valores em termos desses iniciais.
- Conjuntos, por outro lado, são coleções nas quais nenhuma regra de ordenação é imposta

## Sequências

- Uma sequência S é uma lista de objetos que são enumerados segundo alguma ordem
  - Existe um primeiro objeto, um segundo etc
- Uma sequência é definida recursivamente explicitando-se seu primeiro valor (ou primeiros valores) e, a partir daí, definindo-se outros valores em termos desses iniciais.
- Conjuntos, por outro lado, são coleções nas quais nenhuma regra de ordenação é imposta
  - Alguns conjuntos podem ser definidos recursivamente

# Recursão × Iteração

- Sequências têm tanto algoritmos iterativos quanto recursivos
  - Então algoritmos recursivos, **que tratam de sequências**, têm sua versão iterativa
- Ex:

```
long fatorial(long n) {
  if (n==0) return(1);
  return(n *
       fatorial(n-1));
}
```

```
long fatorial(long n) {
  long res = 1;
  for (int i=2; i<=n; i++)
    res *= i;
  return(res);
}
```

# Recursão × Iteração

- Sequências têm tanto algoritmos iterativos quanto recursivos
  - Então algoritmos recursivos, **que tratam de sequências**, têm sua versão iterativa
- Ex:

```
long fatorial(long n) {
  if (n==0) return(1);
  return(n *
       fatorial(n-1));
}
```

Recursivo

```
long fatorial(long n) {
  long res = 1;
  for (int i=2; i<=n; i++)
    res *= i;
  return(res);
}
```

Iterativo

# Recursão × Iteração

## Iterativo

```
int buscaBin(int[] arr, int el){
  int fim = arr.length-1;
  int ini = 0;
  while (ini <= fim) {
    int meio = (fim + ini)/2;
    if (arr[meio] < el)
      ini = meio + 1;
    else
      if (arr[meio] > el)
        fim = meio - 1;
      else
        return(meio);
  }
  return(-1);
}
```

# Recursão × Iteração

## Iterativo

```
int buscaBin(int[] arr, int el){
  int fim = arr.length-1;
  int ini = 0;
  while (ini <= fim) {
    int meio = (fim + ini)/2;
    if (arr[meio] < el)
      ini = meio + 1;
    else
      if (arr[meio] > el)
        fim = meio - 1;
      else
        return(meio);
  }
  return(-1);
}
```

## Recursivo

```
int buscaBin(int[] arr, int ini,
                        int fim, int el){
  if (ini > fim) return(-1);

  int meio = (fim + ini)/2;

  if (arr[meio] < el)
    return(buscaBin(arr,meio+1,
                        fim,el));
  if (arr[meio] > el)
    return(buscaBin(arr,ini,
                        meio-1,el));
  return(meio);
}
```

# Recursão × Iteração

- Soluções recursivas são geralmente mais concisas que as iterativas, gerando programas mais simples.

# Recursão × Iteração

- Soluções recursivas são geralmente mais concisas que as iterativas, gerando programas mais simples.
- Soluções recursivas advém diretamente da prova por indução

# Recursão × Iteração

- Soluções recursivas são geralmente mais concisas que as iterativas, gerando programas mais simples.
- Soluções recursivas advém diretamente da prova por indução
- Soluções iterativas em geral usam uma quantia de memória limitada, enquanto as recursivas não

# Recursão $\times$ Iteração

- Soluções recursivas são geralmente mais concisas que as iterativas, gerando programas mais simples.
- Soluções recursivas advém diretamente da prova por indução
- Soluções iterativas em geral usam uma quantia de memória limitada, enquanto as recursivas não
- A cópia dos parâmetros a cada chamada recursiva é um custo adicional para as soluções recursivas

## Exemplo: Cálculo de Polinômios

- Dado um arranjo $(a_n, a_{n-1}, \ldots, a_1, a_0)$ de coeficientes, e um valor $x$, queremos calcular o polinômio
  $P_n(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0$

# Indução e Algoritmos

## Exemplo: Cálculo de Polinômios

- Dado um arranjo $(a_n, a_{n-1}, \ldots, a_1, a_0)$ de coeficientes, e um valor $x$, queremos calcular o polinômio
  $P_n(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0$
- Vamos projetar um algoritmo para resolver esse problema a partir da demonstração indutiva da seguinte afirmação:

  *Dados $x$ e $(a_n, a_{n-1}, \ldots, a_1, a_0)$, posso calcular $P_n(x)$*

## Exemplo: Cálculo de Polinômios

- Demonstração:
  - **Base**: $P_0(x) = a_0$

# Indução e Algoritmos

## Exemplo: Cálculo de Polinômios

- Demonstração:
  - **Base**: $P_0(x) = a_0$
  - **Hipótese**: Dados $x$ e $(a_n, a_{n-1}, \ldots, a_1, a_0)$, conseguimos calcular $P_{n-1}(x)$

## Exemplo: Cálculo de Polinômios

- Demonstração:
  - **Base**: $P_0(x) = a_0$
  - **Hipótese**: Dados $x$ e $(a_n, a_{n-1}, \ldots, a_1, a_0)$, conseguimos calcular $P_{n-1}(x)$
  - **Passo**: É direto que $P_n(x) = a_n x^n + P_{n-1}(x)$

# Indução e Algoritmos

## Exemplo: Cálculo de Polinômios

- Demonstração:
  - **Base**: $P_0(x) = a_0$
  - **Hipótese**: Dados $x$ e $(a_n, a_{n-1}, \ldots, a_1, a_0)$, conseguimos calcular $P_{n-1}(x)$
  - **Passo**: É direto que $P_n(x) = a_n x^n + P_{n-1}(x)$
- Algoritmo derivado:

```
P(A, x, n)
  se n = 0 retorne A[0]
  senão
    retorne A[n] * x^n + P(A, x, n-1)
```

# Indução e Algoritmos

## Exemplo: Cálculo de Polinômios

- Demonstração:
  - **Base**: $P_0(x) = a_0$
  - **Hipótese**: Dados $x$ e $(a_n, a_{n-1}, \ldots, a_1, a_0)$, conseguimos calcular $P_{n-1}(x)$
  - **Passo**: É direto que $P_n(x) = a_n x^n + P_{n-1}(x)$
- Algoritmo derivado:

```
P(A, x, n)
  se n = 0 retorne A[0]
  senão
    retorne A[n] * x^n + P(A, x, n-1)
```

Base

# Indução e Algoritmos

## Exemplo: Cálculo de Polinômios

- Demonstração:
  - **Base**: $P_0(x) = a_0$
  - **Hipótese**: Dados $x$ e $(a_n, a_{n-1}, \ldots, a_1, a_0)$, conseguimos calcular $P_{n-1}(x)$
  - **Passo**: É direto que $P_n(x) = a_n x^n + P_{n-1}(x)$
- Algoritmo derivado:

```
P(A, x, n)
  se n = 0 retorne A[0]
  senão
    retorne A[n] * x^n + P(A, x, n-1)
```

Hipótese de Indução

# Indução e Algoritmos

## Exemplo: Cálculo de Polinômios

- Demonstração:
  - **Base**: $P_0(x) = a_0$
  - **Hipótese**: Dados $x$ e $(a_n, a_{n-1}, \ldots, a_1, a_0)$, conseguimos calcular $P_{n-1}(x)$
  - **Passo**: É direto que $P_n(x) = a_n x^n + P_{n-1}(x)$
- Algoritmo derivado:

```
P(A, x, n)
  se n = 0 retorne A[0]
  senão
    retorne A[n] * x^n + P(A, x, n-1)
```

Passo

# Indução e Algoritmos

## Exemplo: Cálculo de Polinômios

- Algoritmo:

```
P(A, x, n)
  se n = 0 retorne A[0]
  senão
    retorne A[n] * x^n + P(A, x, n-1)
```

- E em java:

```
double pol(double[] A, double x, int n) {
  if (n==0) return(A[0]);
  return(A[n] * Math.pow(x,n) + pol(A,x,n-1));
}
```

## Exemplo: Busca Binária

- Vimos o algoritmo "já pronto". Mas será que conseguimos implementá-lo a partir da prova?

## Exemplo: Busca Binária

- Vimos o algoritmo "já pronto". Mas será que conseguimos implementá-lo a partir da prova?
- Considere a afirmação:

"Dado um arranjo $r$, ordenado de forma crescente, de tamanho $n \geq 1$, consigo dizer se um determinado valor $x$ está ou não está no arranjo"

# Indução e Algoritmos

## Exemplo: Busca Binária

- **Base**: $n = 1$.

# Indução e Algoritmos

## Exemplo: Busca Binária

- **Base**: $n = 1$. Se $r_1 = x$, então $x \in r$, senão $x \notin r$

## Exemplo: Busca Binária

- **Base**: $n = 1$. Se $r_1 = x$, então $x \in r$, senão $x \notin r$
- **Hipótese (H.I.)**:

# Indução e Algoritmos

## Exemplo: Busca Binária

- **Base**: $n = 1$. Se $r_1 = x$, então $x \in r$, senão $x \notin r$
- **Hipótese (H.I.)**:
  - Dado $j \geq 1$ e um arranjo $r$ de $j$ elementos, em ordem crescente, sei dizer se $x \in r$, $\forall i = |r|$, $1 \leq i \leq j$

## Exemplo: Busca Binária

- **Base**: $n = 1$. Se $r_1 = x$, então $x \in r$, senão $x \notin r$
- **Hipótese (H.I.)**:
  - Dado $j \geq 1$ e um arranjo $r$ de $j$ elementos, em ordem crescente, sei dizer se $x \in r$, $\forall i = |r|, 1 \leq i \leq j$ (ind. forte)

# Indução e Algoritmos

## Exemplo: Busca Binária

- **Base**: $n = 1$. Se $r_1 = x$, então $x \in r$, senão $x \notin r$
- **Hipótese (H.I.)**:
  - Dado $j \geq 1$ e um arranjo $r$ de $j$ elementos, em ordem crescente, sei dizer se $x \in r$, $\forall i = |r|$, $1 \leq i \leq j$ (ind. forte)
- **Passo**: Seja $s$ um arranjo de $j + 1$ elementos.

## Exemplo: Busca Binária

- **Base**: $n = 1$. Se $r_1 = x$, então $x \in r$, senão $x \notin r$
- **Hipótese (H.I.)**:
  - Dado $j \geq 1$ e um arranjo $r$ de $j$ elementos, em ordem crescente, sei dizer se $x \in r$, $\forall i = |r|$, $1 \leq i \leq j$ (ind. forte)
- **Passo**: Seja $s$ um arranjo de $j + 1$ elementos.
  - Se o elemento do meio de $s$, $s_m$ for $x$, então $x \in s$

# Indução e Algoritmos

## Exemplo: Busca Binária

- **Base**: $n = 1$. Se $r_1 = x$, então $x \in r$, senão $x \notin r$
- **Hipótese (H.I.)**:
  - Dado $j \geq 1$ e um arranjo $r$ de $j$ elementos, em ordem crescente, sei dizer se $x \in r$, $\forall i = |r|, 1 \leq i \leq j$ (ind. forte)
- **Passo**: Seja $s$ um arranjo de $j + 1$ elementos.
  - Se o elemento do meio de $s$, $s_m$ for $x$, então $x \in s$
  - Senão, se $s_m > x$, olho no sub-arranjo $s' = [s_1, s_2, \ldots, s_{m-1}]$. Pela H.I., sei dizer se $x \in s'$, então sei dizer se $x \in s$

# Indução e Algoritmos

## Exemplo: Busca Binária

- **Base**: $n = 1$. Se $r_1 = x$, então $x \in r$, senão $x \notin r$
- **Hipótese (H.I.)**:
  - Dado $j \geq 1$ e um arranjo $r$ de $j$ elementos, em ordem crescente, sei dizer se $x \in r$, $\forall i = |r|$, $1 \leq i \leq j$ (ind. forte)
- **Passo**: Seja $s$ um arranjo de $j + 1$ elementos.
  - Se o elemento do meio de $s$, $s_m$ for $x$, então $x \in s$
  - Senão, se $s_m > x$, olho no sub-arranjo $s' = [s_1, s_2, \ldots, s_{m-1}]$. Pela H.I., sei dizer se $x \in s'$, então sei dizer se $x \in s$
  - Senão ($s_m < x$), olho no sub-arranjo $s' = [s_{m+1}, s_{m+2}, \ldots, s_{j+1}]$. Pela H.I., sei dizer se $x \in s'$, então sei dizer se $x \in s$

# Indução e Algoritmos

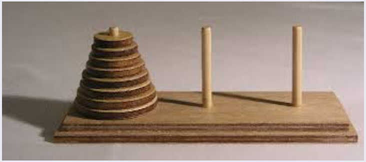
## Exemplo: Busca Binária

- E disso temos o algoritmo:

```
busca(r, x)
  se |r| = 1
    se r_1=x retorna "está"
    senão retorna "não está"
  senão
    m <- índice do meio do arranjo
    se r_m = x, retorna "está"
    senão, se r_m > x
      retorna busca([r_1, r_2, ..., r_{m-1}], x)
      senão
        retorna busca([r_{m+1}, r_{m+2}, ..., r_{|r|}], x)
```

## Exemplo: Torres de Hanói

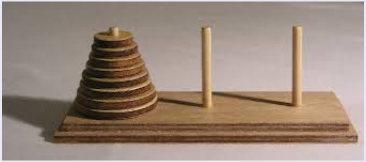
- Considere a imagem



# Indução e Algoritmos

## Exemplo: Torres de Hanói

- Considere a imagem



- O objetivo é mover os $n$ discos do primeiro pino a um outro, de acordo com as seguintes regras:
  - Apenas um disco pode ser movido por vez
  - Um disco maior não pode ser colocado sobre um menor
  - Pode-se usar o terceiro pino como auxiliar

## Exemplo: Torres de Hanói

- Como fazer?

# Indução e Algoritmos

## Exemplo: Torres de Hanói

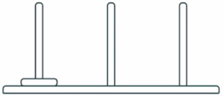
- Como fazer?
- Quero demonstrar que: Dados 3 pinos, e uma torre de $n$ discos em um desses pinos, consigo mover essa torre para um segundo pino, sujeito às condições:
  - Apenas um disco pode ser movido por vez
  - Um disco maior não pode ser colocado sobre um menor
  - Pode-se usar o terceiro pino como auxiliar

## Exemplo: Torres de Hanói

- **Base**: $n = 1$

## Exemplo: Torres de Hanói

- **Base**: $n = 1$
  - Mova o disco do pino de origem para o de destino

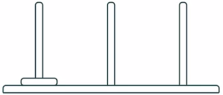


$\rightarrow$

# Indução e Algoritmos

## Exemplo: Torres de Hanói

- **Base**: $n = 1$
  - Mova o disco do pino de origem para o de destino



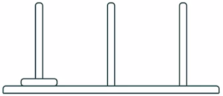
$\longrightarrow$

- **Hipótese**: Consigo mover uma torre de $j - 1$ discos, $(j - 1) \geq 1 \Rightarrow j > 1$

# Indução e Algoritmos

## Exemplo: Torres de Hanói

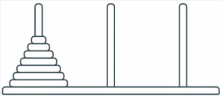
- **Base**: $n = 1$
  - Mova o disco do pino de origem para o de destino



$\longrightarrow$

- **Hipótese**: Consigo mover uma torre de $j - 1$ discos, $(j - 1) \geq 1 \Rightarrow j > 1$

- **Passo**: Seja uma torre de Hanói com $j$ discos:



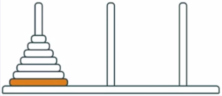
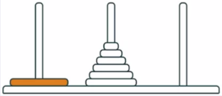
## Exemplo: Torres de Hanói

- **Passo** (cont.): Pela H.I., consigo mover os $j-1$ discos do topo para um pino auxiliar

# Indução e Algoritmos

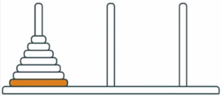
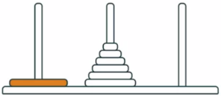
## Exemplo: Torres de Hanói

- **Passo** (cont.): Pela H.I., consigo mover os $j-1$ discos do topo para um pino auxiliar

 $\longrightarrow$ 

## Exemplo: Torres de Hanói

- **Passo** (cont.): Pela H.I., consigo mover os $j-1$ discos do topo para um pino auxiliar

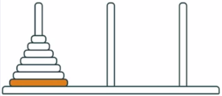
 $\longrightarrow$ 

Movo então o disco da base para o pino de destino

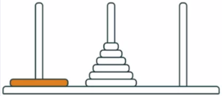
# Indução e Algoritmos

## Exemplo: Torres de Hanói

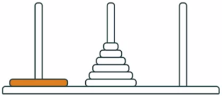
- **Passo** (cont.): Pela H.I., consigo mover os $j-1$ discos do topo para um pino auxiliar



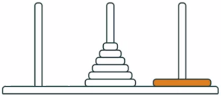
$\longrightarrow$



Movo então o disco da base para o pino de destino



$\longrightarrow$



# Indução e Algoritmos

## Exemplo: Torres de Hanói

- **Passo** (cont.): E, novamente pela H.I., consigo mover os $j - 1$ discos do pino auxiliar para o de destino

# Indução e Algoritmos

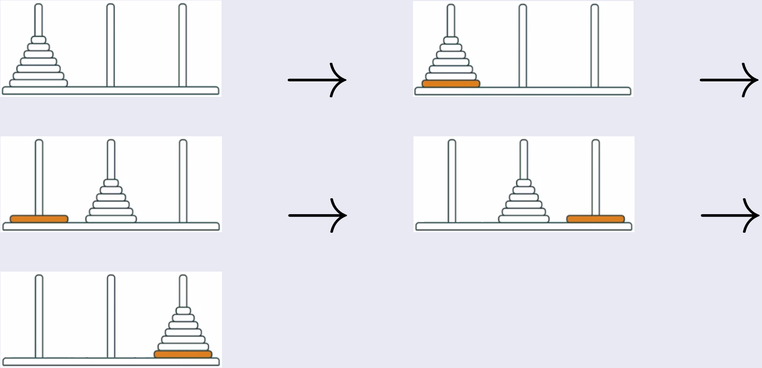
## Exemplo: Torres de Hanói

- **Passo** (cont.): E, novamente pela H.I., consigo mover os $j-1$ discos do pino auxiliar para o de destino

# Indução e Algoritmos

## Exemplo: Torres de Hanói

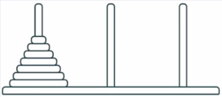
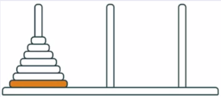
- Ou seja...

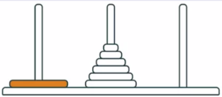
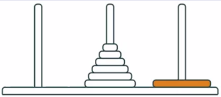


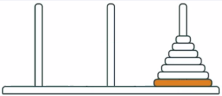
# Indução e Algoritmos

## Exemplo: Torres de Hanói

- Ou seja...

 $\rightarrow$  $\rightarrow$

 $\rightarrow$  $\rightarrow$



Nada intuitivo!

# Indução e Algoritmos

## Exemplo: Torres de Hanói

- E...

```
void hanoi(char ori, char dst, char aux, int n) {
  if(n == 1) {
    System.out.print("Move de " + ori);
    System.out.println(" para " + dst);
  }
  else {
    hanoi(ori, aux, dst, n-1);
    hanoi(ori, dst, aux, 1);
    hanoi(aux, dst, ori, n-1);
  }
}
```

# Indução e Algoritmos

## Torres de Hanói: resultado

- `hanoi('A','C','B',3):`

  Move de A para C

  Move de A para B

  Move de C para B

  Move de A para C

  Move de B para A

  Move de B para C

  Move de A para C

# Em Suma

- Algoritmos tratam sempre de se demonstrar um teorema:

# Em Suma

- Algoritmos tratam sempre de se demonstrar um teorema:
  - Meu algoritmo resolve um determinado problema

# Em Suma

- Algoritmos tratam sempre de se demonstrar um teorema:
  - Meu algoritmo resolve um determinado problema
- A demonstração por indução é uma técnica bastante interessante, pois:

# Em Suma

- Algoritmos tratam sempre de se demonstrar um teorema:
  - Meu algoritmo resolve um determinado problema
- A demonstração por indução é uma técnica bastante interessante, pois:
  - Possui a característica de ser construtiva, evidenciando os passos necessários para se obter o resultado do teorema

# Em Suma

- Algoritmos tratam sempre de se demonstrar um teorema:
  - Meu algoritmo resolve um determinado problema
- A demonstração por indução é uma técnica bastante interessante, pois:
  - Possui a característica de ser construtiva, evidenciando os passos necessários para se obter o resultado do teorema
  - Útil para formular procedimentos recursivos e indutivos cujas provas de corretude são as provas por indução que lhes deram origem.

# Referências


- Ziviani, Nivio. Projeto de Algoritmos: com implementações em Java e C++. Cengage. 2007.
- Manber, Udi. Introduction to Algorithms: A Creative Approach. Addison-Wesley. 1989.
- Gersting, Judith L. Fundamentos Matemáticos para a Ciência da Computação. 3a ed. LTC. 1993.
- Manber, Udi. Using Induction to Design Algorithms. Communications of the ACM, 31(11). 1988.

# Referências


- https://www.coursera.org/lecture/what-is-a-proof/hanoi-towers-V5KXZ
- https://algorithms.tutorialhorizon.com/towers-of-hanoi/
- http://larc.unt.edu/ian/TowersOfHanoi/index64.html