INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA

DO SUL DE MINAS GERAIS CAMPUS MACHADO

CURSO DE GRADUAÇÃO EM SISTEMAS DE INFORMAÇÃO

GUSTAVO MARTINS DE LIMA

INDUÇÃO E RECURSÃO: DEFINIÇÕES, PROPRIEDADES E APLICAÇÕES COMPUTACIONAIS

MACHADO - MG

2025

GUSTAVO MARTINS DE LIMA

INDUÇÃO E RECURSÃO: DEFINIÇÕES, PROPRIEDADES E APLICAÇÕES COMPUTACIONAIS

Trabalho apresentado ao Curso de Sistemas de Informação do IFSULDEMINAS – Campus Machado, como requisito parcial para obtenção de nota na disciplina de Matemática Discreta.

Professor: Dr. Peterson Pereira de Oliveira.

MACHADO - MG

2025

Dedico este trabalho ao Prof. M. Gabriel da Silva Macedo, cujas aulas de física não apenas explicaram o movimento dos corpos, mas moveram minha curiosidade em direção à ciência.

RESUMO

O presente trabalho explora a indução matemática e a recorrência como pilares fundamentais da Ciência da Computação. O objetivo central é demonstrar a conexão inerente entre estes conceitos matemáticos e a lógica de programação, além de destacar a sua importância como um método de criação e prova para algoritmos recursivos. Os conceitos são inicialmente apresentados com sua respectiva demonstração formal e analogias didáticas, para depois serem analisados sob o contexto computacional com exemplos de aplicação na linguagem de programação Python 3. Por fim, conclui-se que o domínio do rigor matemático é essencial para profissionais de Sistemas de Informação, devido à robustez adicionada às soluções computacionais.

Palavras-Chave: Matemática Discreta. Indução Matemática. Recorrência. Algoritmos Recursivos. Python. Sistemas de Informação.

ABSTRACT

This work explores mathematical induction and recurrence as fundamental pillars of Computer Science. The central objective is to demonstrate the inherent connection between these mathematical concepts and programming logic, in addition to highlighting their importance as a method of creation and proof for recursive algorithms. The concepts are initially presented with their respective formal demonstration and didactic analogies, to then be analyzed under the computational context with application examples in the Python 3 programming language. Finally, it is concluded that the mastery of mathematical rigor is essential for Information Systems professionals, due to the robustness added to computational solutions.

Keywords: Discrete Mathematics. Mathematical Induction. Recurrence. Recursive Algorithms. Python. Information Systems.

LISTA DE FIGURAS

SUMÁRIO

1. INTRODUÇÃO

O princípio da Indução Matemática é fundamental para a lógica computacional, pois, além de ser a base do entendimento de algoritmos recursivos, é essencial para a verificação formal de sua corretude. Para o curso de Sistemas de Informação, dominar este conceito confere a capacidade de resolver problemas complexos, cuja solução seria menos intuitiva ou mais trabalhosa com a lógica de loops convencionais. Neste contexto, o presente trabalho explora as definições formais e demonstra aplicações por meio da Matemática Discreta e de sua respectiva implementação na linguagem de programação Python.

2. INDUÇÃO MATEMÁTICA

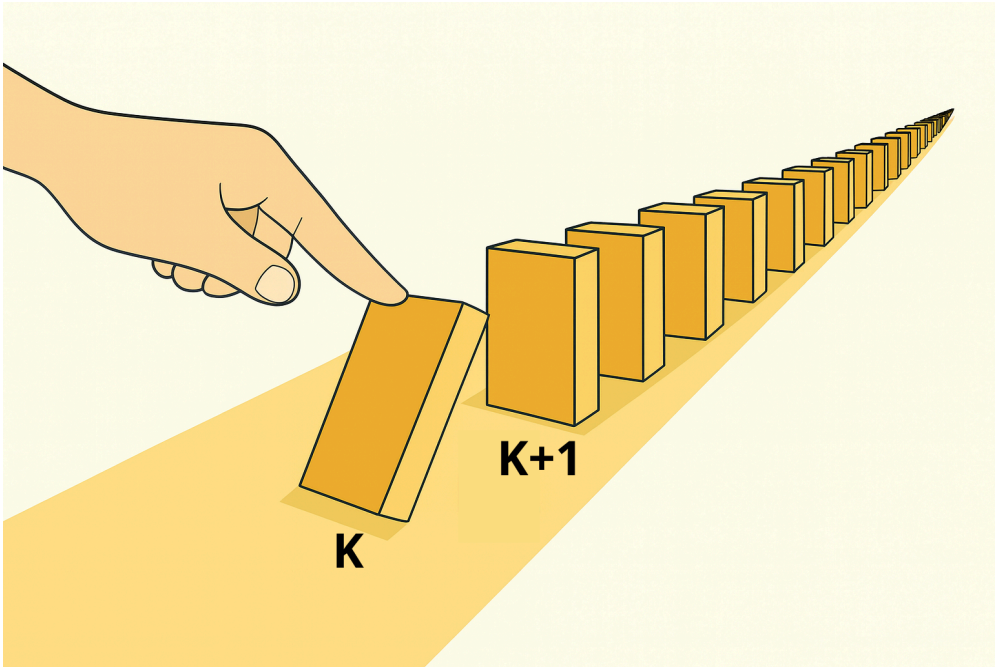
A indução matemática estabelece que para garantir que uma **propriedade P** é válida para todos os números naturais $\mathbb{N}$, dois requisitos devem ser satisfeitos: primeiramente, essa propriedade **P** deve ser válida para o número 1 (**caso base**); em segundo lugar, deve-se demonstrar que se um número n qualquer possui essa propriedade P, então obrigatoriamente **n+1** também possui (**passo indutivo**).

Se esses requisitos forem cumpridos, podemos afirmar que a propriedade **P** é válida para todo número $\mathbb{N}$. Portanto, o princípio da indução funciona como um mecanismo de prova em cascata, que, a partir de um caso base, induz a propriedade a seu termo sucessor.

2.1 O Dominó Infinito

Embora possa parecer estranho utilizar uma hipótese temporária como parte da prova, podemos entender de forma intuitiva ao analisar o exemplo do dominó infinito. Para visualizar o conceito, observe a analogia dos dominós na Figura 1. Derrubar a primeira peça equivale ao caso base.

Figura 1 - Representação do Efeito Dominó



Fonte: Elaborada pelo autor via Microsoft Copilot (2025).

Imagine que a superfície da imagem é plana e regular e que os dominós estão equidistantes. Assuma que um dominó qualquer na **posição K** cai, dessa forma assumimos que a propriedade **P (cair)** é verdadeira para o dominó **K**. Intuitivamente, se o dominó **K** caiu, o seu sucessor **K+1** obrigatoriamente também cairá (**P(K)** $\Rightarrow$ **P(K+1)**). Portanto, como o primeiro dominó caiu (**Passo Base**) e garantimos a transmissão da queda para o sucessor (**Passo Indutivo/Cascata**), conclui-se que todos os dominós da fileira cairão.

2.2 Aplicação Computacional

Em geral, o objetivo dos programadores é projetar soluções para problemas via linguagens de programação. No entanto, existe uma enorme diferença entre **testar** e **garantir** o funcionamento de um código. Para testar um algoritmo que abrange os números naturais, por exemplo, é necessário apenas escolher uma quantidade finita de casos teste, para assim afirmar que a solução satisfaz o problema, é como dizer que se não falhou para estes, então não falhará para os demais. Todavia esse método não prova **absolutamente nada**, para garantir que resolvemos o problema, precisamos provar que a solução é válida para **todas** as entradas possíveis. Assim, a Indução Matemática surge no contexto computacional como o método rigoroso para verificar a corretude de algoritmos. Para exemplificar esta aplicação, analisaremos o algoritmo de cálculo de Fatorial em Python a seguir:

Algoritmo 1 - Fatorial Recursivo

```
def fatorial(n):

   if n == 0:
       return 1

   else:
       return n * fatorial(n-1)
```

A função fatorial(n) recebe um número inteiro não negativo n qualquer como parâmetro e utiliza **recursividade** (falaremos a respeito na seção 3) para encontrar o caso base **n = 0**. A lógica do algoritmo divide-se em duas vias: a **condição de parada** (equivalente ao Passo Base) e a **chamada recursiva** (equivalente ao Passo Indutivo) .

Queremos provar que para um número qualquer n, a função fatorial(n) resulta exatamente em n!. Note que se n for zero, a condicional if n == 0 retorna 1, o que

matematicamente está correto pois 0! = 1, dessa forma encontramos o caso base n. Agora vamos assumir que um número arbitrário K >= 0 funciona na função. Dessa forma, **fatorial(K+1)** também está presente na função, pois fatorial(K+1) entra no bloco else que retorna K + 1 * fatorial (K + 1 - 1) , pela nossa hipótese, o retorno é **(K+1) multiplicando K!**, que é exatamente a definição de **(K+1)!** .

3. RECORRÊNCIA E RECURSÃO

Certos problemas têm uma representação formal muito complexa para definições explícitas, dificultando a sua representação. Para contornar isso, podemos utilizar a **recorrência** para definir objetos em termos de si próprio. Para fazer essa representação é preciso aplicar dois conceitos sobre indução matemática que aprendemos na **seção 2**: primeiro definimos o valor de um caso base, geralmente **n = 1 ou n = 0**; depois executamos um passo a partir do anterior **n+1**. Assim ficamos com a representação em si, que pode ser comprovada via indução matemática.

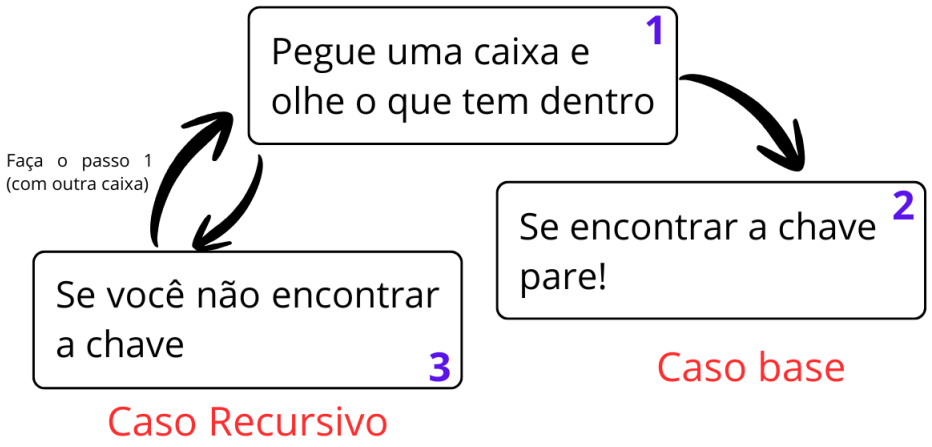
Exemplo, vamos definir a soma dos n primeiros números naturais $\mathbf{S_n = a_1 + a_2 + ... + a_n}$**:** Primeiro, definimos $\mathbf{S_1 = a_1}$, pois a soma de um único termo, por definição, é o próprio termo; depois estabelecemos a fórmula com um termo subsequente de um n arbitrário $\mathbf{S_n + 1 = S_n + a_n + 1}$, ou seja, a soma de um termo subsequente de $\mathbf{S_n}$ é a somatória dos termos até **n** somado a seu sucessor. Para validar as demonstrações por recorrência, basta utilizar a técnica de indução matemática que aprendemos na seção 2.

3.1 O Presente da Vovó

Para que possamos solidificar o entendimento do conceito de recorrência, vamos propor um cenário hipotético. Imagine que sua avó tem um presente para lhe entregar, porém esse presente está trancado em um baú. Sua avó sabe que a chave está dentro de uma caixa, mas tem um problema, ela guarda todas as caixas menores que possui em uma caixa maior. Dessa forma, sua tarefa é encontrar a caixa menor que possui a chave dentre as diversas outras caixas menores armazenadas na caixa maior.

Para resolver isso vamos utilizar a recorrência, isso é, vamos resolver o nosso **grande problema** a partir dele mesmo. Observe a imagem a seguir sobre a resolução do problema:

Figura 2 - Resolução do problema da caixa com recorrência



Fonte: Elaborada pelo autor (2025).

Dessa forma, utilizamos a nossa resolução como parte dela mesma. A caixa com a chave é o **caso base**, e para a acharmos: **1** olhamos uma caixa; **2** se a chave está dentro paramos (caso base) ; **3** se a chave não está dentro repetimos o passo 1 **na caixa que acabamos de encontrar** (passo recursivo/indutivo). Na programação, chamamos essa resolução de **algoritmo recursivo**.

3.2 Recursão e a Pilha de Chamada:

A partir do conceito de recorrência, surge a **recursão** na programação. Embora sejam parecidos por definição, não se deve confundir esses conceitos, a **recursividade** é uma técnica de resolução de problemas algorítmicos que utiliza a recorrência como base, isso é, utiliza a ideia de definir um objeto a partir dele mesmo. Sendo mais específico, recursão é quando uma **função chama a si mesma** para resolver seu próprio problema dividindo a resolução em **camadas ou instâncias menores**.

Para que possamos entender o conceito de recursão, é indispensável saber onde que as camadas menores do nosso problema ficam armazenadas, pois é preciso que haja um espaço disponível na memória. Dessa forma, precisamos compreender a estrutura de dados chamada **pilha (stack),** ela funciona como uma lista ou array, mas de forma extremamente

simplificada. Digamos que a pilha é um array em que você tem acesso somente a seu último item, sendo que as únicas operações disponíveis são: **push** (adiciona um item no topo da pilha) e **pop** (remove e lê o item do topo da pilha).

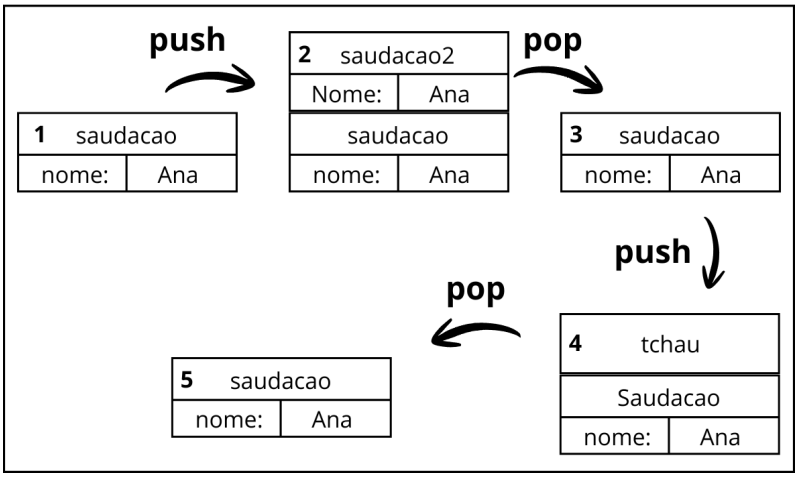
Os computadores utilizam uma pilha interna denominada como **pilha de chamada,** vamos ver isso na prática com o algoritmo a seguir:

Algoritmo 2 - Exemplo de Pilha de Chamada

```
def saudacao(nome):
    print("Olá " + nome + "!")
    saudacao2(nome)
    tchau()

def saudacao2(nome):
    print("Como vai " + nome + "?")

def tchau():
    print("Ok, tchau!")

saudacao("Ana")
```

Primeiro, o computador empilha um novo registro (stack frame) na memória para chamar a função saudacao() e passa um nome como argumento. No escopo da função saudacao(), o parâmetro nome é utilizado no print() e é passado como argumento para a função saudacao2(), que para ser chamada, aloca um novo registro na memória. Depois que saudacao2() é concluída, o espaço reservado a ela é desempilhado. Agora a função saudacao() chama a função tchau(), que, da mesma forma, tem um registro empilhado na memória que depois de utilizado é desalocado. Assim a função saudacao() termina. Observe a imagem a seguir para visualizar a explicação:

Figura 3 - Ilustração da pilha de chamada no algoritmo



Fonte: Elaborada pelo autor (2025).

3.3 Algoritmos Recursivos na Pilha de Chamada

O algoritmo que utilizamos para entender a pilha de chamada não era recursivo, isto é, não chamava a si mesmo para resolver o problema principal. Quando estamos lidando com algoritmos recursivos é necessário agirmos com cautela em sua concepção, pois caso a função não alcance o caso base, o algoritmo resulta em um erro chamado **stack overflow** (estouro de pilha). Vamos analisar o algoritmo a seguir:

Algoritmo 3 - Stack Overflow

```
def regressiva(i):
    print(i)
    regressiva(i-1)

regressiva(10)
```

A execução da função regressiva() está condenada a uma falha técnica. Quando passamos 10 como argumento, a função imprime 10 e depois chama a si mesma com i atual - 1“para sempre”. Basicamente estamos pedindo para que o computador faça a operação **push** indefinidamente, mas a memória RAM não é infinita.

Agora vamos corrigir o algoritmo, estabelecendo um caso base para que o passo indutivo de nossa função possa alcançar:

Algoritmo 4 - Correção do Stack Overflow

```
def regressiva(i):

    print(i)

    if i <= 0: #na programação é comum que o caso base seja 0
        return
    else:
        regressiva(i-1) #passo indutivo

regressiva(5)
```

Como estabelecemos um **caso base** (i=0), nossa função tem um limite. Observe que o **passo indutivo** (dentro do bloco else) só é chamado se o argumento de entrada respeita o caso base.

3.4 Loops vs Recursão

É pertinente observar que os algoritmos recursivos apresentados no trabalho possam facilmente ser concebidos utilizando laços de repetição (loops) ao invés de funções recursivas. De fato, muitas vezes loops são até mesmo mais performáticos para determinados problemas. Devido a isso cito uma frase de Leigh Caldwell, que li no livro ***Entendendo Algoritmos***:

"Os loops podem melhorar o desempenho do seu programa. A recursão melhora o desempenho de seu programador. Escolha o que for mais importante para a sua situação" (CALDWELL *apud* BHARGAVA, 2017, p. 59).

A recursão é simplesmente uma técnica para deixar uma resposta mais clara, mesmo que superficialmente pareça mais complexa do que os loops convencionais. Utilize o que for mais adequado para o contexto de aplicação. Porém, tenha em mente que existem algoritmos recursivos muito importantes na programação, devido a isso, entender o conceito de recursão se torna uma necessidade.

4. EXERCÍCIOS E FERRAMENTAS

Utilizando tudo que foi abordado até agora, vamos explorar mais aplicações dos conceitos em exercícios e ferramentas populares na matemática.

4.1 Os Coelhos de Fibonacci

Este problema foi proposto e resolvido pelo matemático italiano Leonardo de Pisa em 1202. O problema consiste em determinar a quantidade total de casais coelhos após um ano, seguindo as seguintes regras: a cada mês, um casal de coelhos produz outro casal e um casal começa a procriar dois meses após o seu nascimento.

A partir dessas regras temos as seguintes informações: $\mathbf{F_1 = F_2 = 1}$ (onde **n** representa o mês e $\mathbf{F_n}$ a quantidade de casais.), ou seja, a quantidade de casais de coelhos no primeiro e segundo mês é igual a 1, pois embora eles produzam dois casais, eles ainda não podem procriar devido à restrição de dois meses. Além disso, podemos inferir que o número total de casais é a soma total de casais do mês anterior acrescida a soma total do mês anterior ao anterior, ou seja **Fn = Fn - 1 + Fn - 2**. Acabamos de utilizar a recorrência para descrever o problema.

4.1.1 As Células de Fibonacci

Para que possamos aplicar o problema no contexto computacional vamos mudar algumas variáveis mantendo a ideia inicial. Suponha que um programador foi contratado para trabalhar em um laboratório de biologia. A equipe de cientistas está estudando uma determinada cultura de células chamada **Bonacci**. Essas células operam exclusivamente em pares e possuem um ciclo reprodutivo peculiar: um par maduro **gera** um novo par diariamente (sem deixar de existir) e um par recém-formado precisa de **dois dias de maturação** antes de começar a gerar seus próprios descendentes (o tempo de maturação é o dia do nascimento e o próximo dia). O trabalho do programador é determinar o número total de pares $\mathbf{C_n}$ em uma quantidade de **n** dias.

Observe que, da mesma forma que o problema dos coelhos, $\mathbf{C_1 = C_2 = 1}$ (onde **n** representa o dia e $\mathbf{C_n}$ a quantidade de pares de células), encontramos o caso base. Além disso, também de mesmo modo, **Cn = Cn - 1 + Cn - 2**, logo encontramos o passo indutivo. Portanto, temos tudo que é necessário para criar um algoritmo recursivo:

Algoritmo 5 - As Células Bonacci

```
def bonacci(n):
    if n == 1 or n == 2: #caso base
        return 1
    else:
        return bonacci(n-1) + bonacci(n-2) #caso recursivo
```

Agora vamos utilizar a técnica de indução para comprovar o funcionamento do algoritmo. A função bonacci() recebe um número inteiro **n > 0** e utiliza recursão (chama a si mesma) para encontrar o caso base. Como o algoritmo é recursivo, a lógica de funcionamento se divide em duas partes: a condição de parada (**passo bas**e) e a chamada recursiva (**passo indutivo**)

Queremos provar que um número **n** dentro do limite estabelecido resulta exatamente no enésimo termo da sequência de fibonacci. Note que se n for 1 ou 2, a condicional **if n == 1 or n == 2** retorna **1**, o que está correto pois os dois primeiros números da sequência são 1, portanto o caso base está correto. Agora vamos assumir que um número arbitrário **K > 0** funciona na função. Desse modo, **bonacci(K+1)** também está presente na função, pois **bonacci(k+1)** entra no bloco **else** que retorna **bonacci(K + 1 - 1) + bonacci(K + 1 - 2)**, pela nossa hipótese, o retorno é **bonacci(K) + bonacci(K-1)**, que é exatamente a definição do termo de índice **(k+1)** na sequência de Fibonacci, respeitando a regra **Cn = Cn - 1 + Cn - 2**.

4.2 Somatórios

No início da seção 3 foi abordado brevemente o conceito de somatório, onde definimos por recorrência a soma dos n primeiros termos de uma sequência numérica. No entanto, não dedicamos o tempo ou os créditos necessários para explicar esse conceito de maneira mais aprofundada, tendo em vista a ligação intrínseca com a programação. Podemos afirmar que os somatórios são os laços de repetição (loops) da matemática.

Os somatórios são denotados por $\sum_{i=1}^{n} i$ onde **i = 1** representa o início do loop, **n** o fim e **i** o número que será somado a cada iteração. Dessa forma, se, por exemplo, n = 10

$\sum_{i=1}^{10} i$ **= 1 + 2 +3 + 4 +5 + 6 + 7 + 8 + 9 + 10 = 55.** Para efetuar o cálculo dessa maneira, um computador poderia resolver da seguinte forma:

Algoritmo 6 - Somatória com Loop Simples

```
def somatorio(n):

    total = 0

    for i in range(n+1): # n + 1 serve para incluir o último
termo
        total += i

    print(total)

somatorio(10)
```

A lógica dos somatórios consiste em somas sucessivas de um **limite inferior** até um **limite superior.** Portanto, se os limites são respectivamente 1 e 1.000.000 serão necessárias **um milhão de somas sucessivas**. Por este motivo, as ferramentas que aprendemos até aqui são tão úteis. No exemplo citado, $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$ **.** Ou seja, podemos substituir **1.000.000 operações** de soma **por apenas três operações aritméticas** simples (uma adição, uma multiplicação e uma divisão), obtendo o mesmo resultado instantaneamente.

5. CONSIDERAÇÕES FINAIS

A realização deste trabalho permitiu compreender a magnitude da importância da indução matemática e recorrência no contexto computacional. A utilização dessas ferramentas transforma um palpite de sorte em resoluções incontestáveis e confiáveis. Além disso, são a base para entender técnicas de programação amplamente utilizadas, como a recursão.

Para um profissional de Sistemas de Informação, que preza pelo pleno funcionamento de seus sistemas, o domínio dos conceitos abordados não é apenas um diferencial, mas sim uma habilidade técnica obrigatória, principalmente tendo em vista a evolução da internet das coisas e sua escalabilidade. Cada vez mais, serão exigidos profissionais que conseguem lidar com essa complexidade gerada pela integração de vários sistemas.

Embora existam mais pontos a serem abordados sobre a utilização da matemática discreta no contexto computacional, como por exemplo a aplicação da indução em algoritmos de grafos, o presente trabalho serve como uma visão introdutória e linear. A aplicação dos conceitos abordados em estruturas complexas é contraintuitiva ao escopo aqui apresentado.

REFERÊNCIAS

ALVIM, Mário Sérgio. **Definições Recursivas e Indução Estrutural**. 2024. Slides da disciplina de Introdução à Lógica Computacional, Departamento de Ciência da Computação, Universidade Federal de Minas Gerais (UFMG), Belo Horizonte, 2024. **Disponível em:** https://homepages.dcc.ufmg.br/~msalvim/courses/ilc/Aula3.2_DefinicoesRecursivasInducaoEstruturalAlgoritmosRecursivos%5Bstill%5D.pdf. **Acesso em:** 25 nov. 2025.

BHARGAVA, Aditya Y. **Entendendo Algoritmos**: um guia ilustrado para programadores e outros curiosos. São Paulo: Novatec, 2017.

HEFEZ, Abramo. **Indução Matemática**. Rio de Janeiro: IMPA, 2009. (Apostilas do PIC da OBMEP, n. 4). **Disponível em**: http://www.obmep.org.br/docs/apostila4.pdf. **Acesso em:** 21 nov. 2025.

LIMA, Gustavo M. **Algoritmos-Inducao-Recursao**. Repositório de código fonte. GitHub, 2025. **Disponível em:** https://github.com/gustavo-gml/Algoritmos-Inducao-Recursao. **Acesso em:** 29 nov. 2025.

PROFESSOR OCTAVIO. **O que é indução matemática?** 16 maio 2018. 1 vídeo (4 min). **Disponível em:** https://youtu.be/nt6adjaEVO4?si=OZb5ZXkMnDtGlQX1. **Acesso em:** 25 nov. 2025.

SOUZA, Marcelo Maraschin de. **Indução e Recursão**. 2016. Notas de aula da disciplina de Matemática Discreta, Curso de Ciência da Computação, Instituto Federal de Santa Catarina (IFSC), [S. l.], 2016. **Disponível em:** https://docente.ifsc.edu.br/marcelo.maraschin/Material/Ci%C3%AAncia%20da%20Computa%C3%A7%C3%A3o/Matem%C3%A1tica%20Discreta%2020161/Recursividade.pdf. **Acesso em:** 25 nov. 2025.