



LÓGICA DE PROGRAMAÇÃO E ALGORITMOS

AULA 3

CONVERSA INICIAL

O objetivo desta aula é darmos continuidade aos nossos estudos de programação com linguagem Python.

Ao longo desta aula, você irá aprender a criar códigos que tomam decisões, ou seja, dependendo de uma condição existente nele, as instruções executadas poderão ser distintas.

Ademais, iremos investigar todos os tipos de condicionais existentes: a condicional simples, a composta, a aninhada e a de múltipla escolha. Essas condicionais são existentes em todas as principais linguagens de programação modernas e podem ser adaptadas para fora do Python seguindo o mesmo raciocínio lógico aqui apresentado.

Assim como no conteúdo anterior, todos os exemplos apresentados neste material poderão ser praticados concomitantemente em um Jupyter Notebook, como o Google Colab, o que não requer a instalação de nenhum software de interpretação para a linguagem Python em sua máquina.

Ao longo do material, você encontrará alguns exercícios resolvidos. Esses exercícios estão colocados em linguagem Python com seus respectivos fluxogramas.

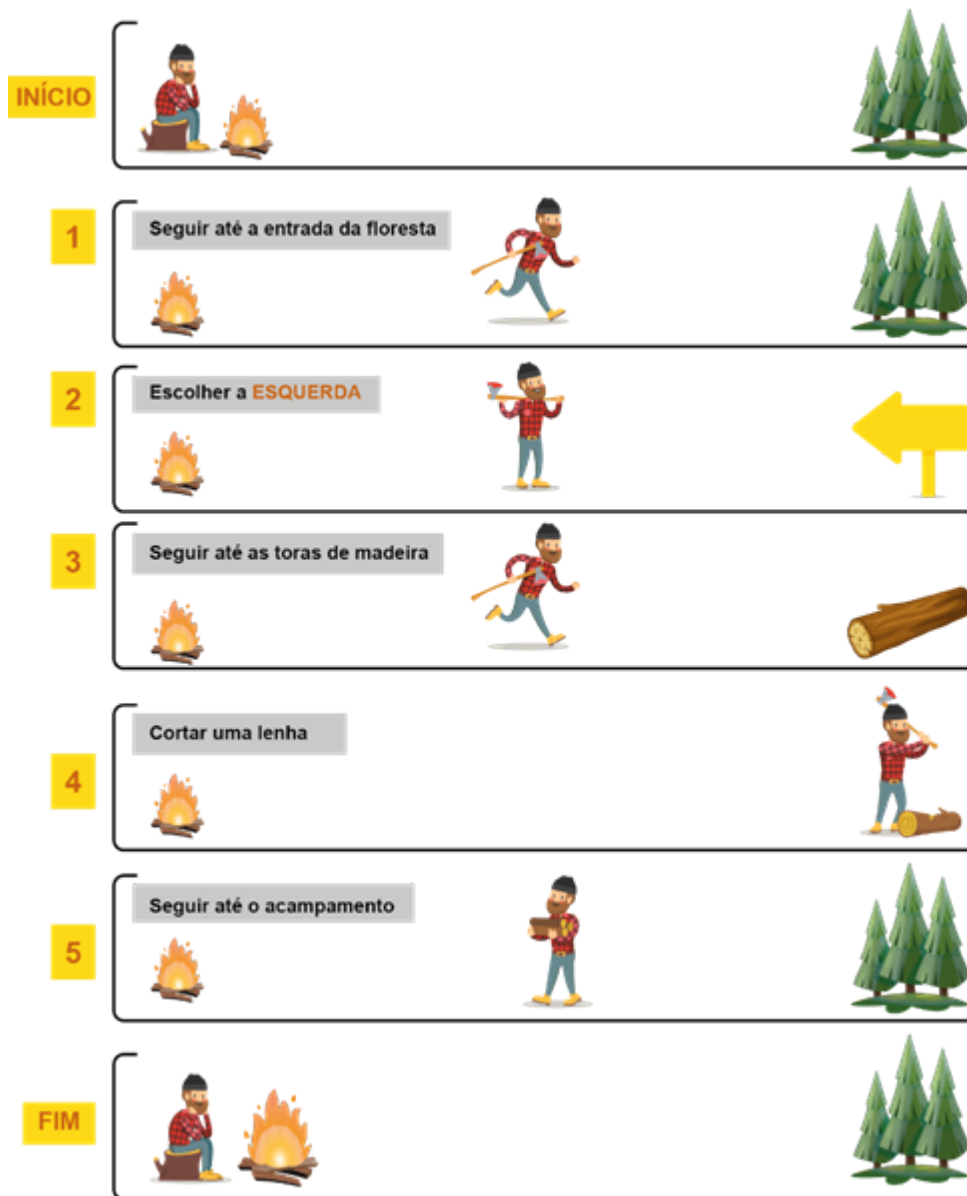
TEMA 1 – ESTRUTURA CONDICIONAL

Todos os algoritmos que você desenvolveu anteriormente são chamados de **algoritmos sequenciais**. Ser sequencial significa que todas as instruções são executadas uma após a outra, sem que pulemos qualquer instrução (Não esqueça que o fluxo de execução das instruções, apresentado em conteúdos anteriores, é válido e deve ser respeitado). Vamos relembrar, a seguir, um algoritmo sequencial com o exemplo lúdico.

Um lenhador está junto de sua fogueira e precisa ir até a floresta cortar mais lenha antes que seu fogo se esmaça. O lenhador tem dois caminhos a seguir e que levam até as toras de madeiras a serem cortadas.

O lenhador optou pelo caminho da esquerda. Nesse caminho, ele precisou seguir até a floresta, cortar a lenha e voltar tranquilamente para seu acampamento. A Figura 1 ilustra seus passos.

Figura 1 – Exemplo lúdico de um algoritmo. Caminho da esquerda



Créditos: Macrovector; ivector; Incomible; FARBAI/Shutterstock.

Podemos escrever o algoritmo do lenhador da Figura 1 usando uma descrição narrativa. Ficaria da seguinte forma:

Início

1. Seguir até a entrada da floresta.

*2. Tomar o caminho da **ESQUERDA**.*

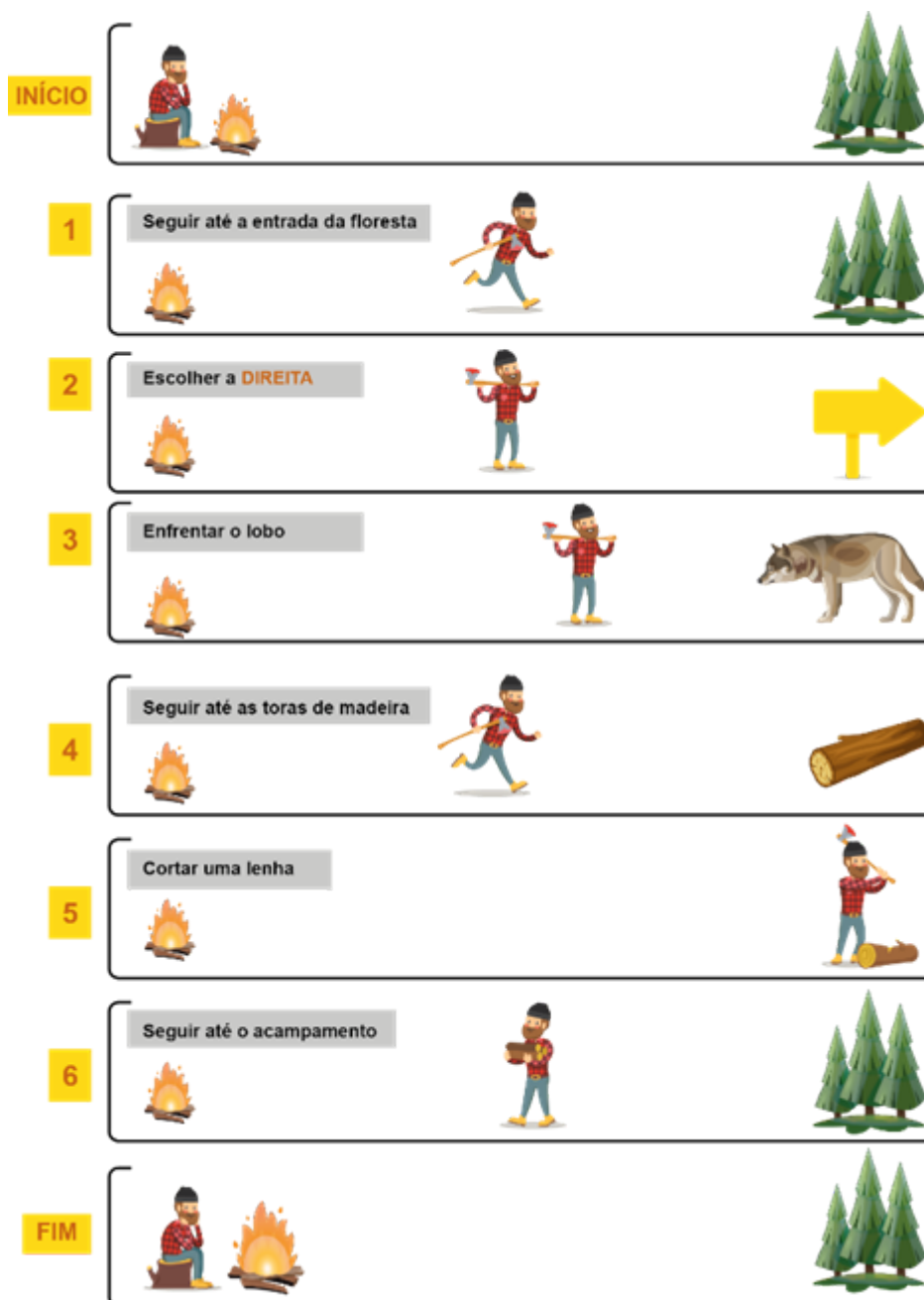
3. Seguir até as toras de madeira.

4. Cortar uma lenha.

5. Retornar ao seu acampamento.

Todos os passos nesse algoritmo são realizados, um após o outro, sem exceção. Agora, vamos analisar o que aconteceria caso nosso lenhador tomasse o caminho da direita, ilustrado na Figura 2.

Figura 2 – Exemplo lúdico de um algoritmo. Caminho da direita



Créditos: Macrovector; ivector; Incomible; FARBAI/Shutterstock, Hennadii H/Shutterstock.

Neste caminho, nosso lenhador acabou se deparando com algo que não havia acontecido para o lado esquerdo. Um lobo apareceu para atacá-lo! Deste modo, é necessário que nosso personagem disperse o lobo antes de conseguir cortar lenha, gerando um passo a mais em nosso algoritmo.

Início

1.Seguir até a entrada da floresta.

2.Tomar o caminho da DIREITA.

3.Enfrentar o lobo.

4.Seguir até as toras de madeira.

5.Cortar uma lenha.

6.Retornar ao seu acampamento.

Fim

Note que, dependendo do caminho tomado, a execução das ações do lenhador poderá ser diferente. Para a esquerda, ele não encontra nenhuma ameaça. Para a direita, existe o lobo. Dependendo de uma condição existente, neste caso, a decisão de qual caminho tomar, as ações resultantes poderão ser diferentes. Vejamos a Tabela 1. Nela, comparamos o que acontece em cada um dos caminhos tomados.

Tabela 1 – Comparativo de ações de ambos caminhos a serem tomados

| Ações | Esquerda | Direta |
|--------------|--|--|
| 1 | <i>Seguir até a entrada da floresta.</i> | <i>Seguir até a entrada da floresta.</i> |
| 2 | <i>Tomar o caminho da ESQUERDA.</i> | <i>Tomar o caminho da DIREITA.</i> |
| 3 | - | <i>Enfrentar o lobo.</i> |
| 4 | <i>Seguir até as toras de madeira.</i> | <i>Seguir até as toras de madeira.</i> |
| 5 | <i>Cortar uma lenha.</i> | <i>Cortar uma lenha.</i> |
| 6 | <i>Retornar ao seu acampamento.</i> | <i>Retornar ao seu acampamento.</i> |

Percebeste que a ação/decisão da Linha 2 na tabela é o que fará tomarmos ações distintas em seguida?

Neste caso, podemos reorganizar nossas ações de uma maneira com que a Ação 2 defina o que acontece em seguida. Para tal, utilizamos uma estrutura de decisão, que em pseudocódigo, nós chamamos de *estrutura condicional se*. Ou seja, se o caminho for o da esquerda, faça as instruções correspondentes à esquerda, senão (o caminho será obrigatoriamente da direita), faça essas outras instruções.

Em nosso exemplo, podemos reorganizar as ações conforme o pseudocódigo a seguir. Observe que existe uma decisão a ser tomada. Se o caminho for o da esquerda, execute todas as linhas dentro do Item 1. Senão, execute todas as instruções dentro do Item 2.

Início

1. se (caminho = esquerda)

- a. Seguir até a entrada da floresta
- b. Seguir até as toras de madeira
- c. Cortar uma lenha
- d. Retornar ao seu acampamento

2. senão

- a. Seguir até a entrada da floresta
- b. Enfrentar o lobo
- c. Seguir até as toras de madeira
- d. Cortar uma lenha
- e. Retornar ao seu acampamento

3. fim-se

Fim

Você conseguiu notar que, dentre todas as ações tomadas, muitas delas se repetem em ambos os caminhos tomados? Ademais, a única ação diferente é a de enfrentar o lobo.

Em programação, não é uma boa prática repetir a mesma instrução diversas vezes. Sempre que for possível evitar isso, faça. Neste caso, podemos otimizar nosso algoritmo para que todas as instruções iguais sejam colocadas fora da condicional. Puxamos uma ação para antes da condição, e as outras três para após a condicional. Veja como podemos reescrever o pseudocódigo, portanto.

Início

1. Seguir até a entrada da floresta
2. se (caminho = esquerda)
3. senão
 - a. Enfrentar o lobo
4. fimse
5. Seguir até as toras de madeira
6. Cortar uma lenha
7. Retornar ao seu acampamento

Fim

Como o lobo só aparece no caminho da direita, note que o algoritmo reescrito não executa nenhuma instrução quando o caminho for da esquerda. Só executa algo caso caia no *senão*. Neste caso, podemos melhorar a legibilidade do nosso programa caso alteremos a condição de verificação. Podemos, assim, inclusive remover o *senão*. Observe como ficaria a seguir.

Início

1. Seguir até a entrada da floresta
2. se (caminho = direita)
 - a. Enfrentar o lobo
3. fimse
4. Seguir até as toras de madeira
5. Cortar uma lenha
6. Retornar ao seu acampamento

Fim

Agora sim temos o algoritmo mais otimizado possível para resolver esse problema. Se o caminho for o da direita (Linha 2), executa-se a Linha 2a. Caso contrário, essa linha é ignorada pelo programa e seguimos direto para a Linha 4.

TEMA 2 – CONDICIONAL SIMPLES E COMPOSTA

Aprendemos e compreendemos o que são estruturas condicionais e como são usadas em um exemplo fora da computação. Agora, vamos formalizar os tipos de condicionais existentes e aprender como aplicá-los em linguagem Python.

Todas as condicionais apresentadas ao longo deste documento existem de alguma maneira em toda e qualquer linguagem de programação, portanto, todos os conceitos podem ser facilmente adaptáveis para outras linguagens, alterando somente a sua sintaxe.

2.1 CONDICIONAL SIMPLES

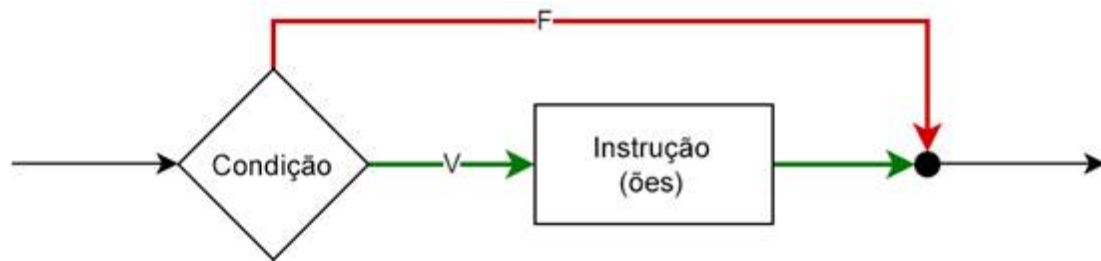
A estrutura condicional simples existe para quando precisamos decidir se um conjunto de instruções precisa ser executado ou não.

Caso um teste lógico realizado resulte em um valor verdadeiro, isso significa que as instruções dentro da condição deverão ser executadas. Caso contrário, ou seja, caso o teste lógico resulte em falso, o fluxo de execução ignora as linhas de código dentro da estrutura.

Na Figura 3, temos o fluxograma de decisão de uma estrutura condicional simples. Lembrando que o padrão adotado para construção de fluxogramas está apresentado no livro da Sangra Puga, Lógica de Programação e Estrutura de Dados, p. 14-15.

A forma geométrica do losango representa uma decisão. Com base nela, dois caminhos distintos podem ser tomados. Um caminho representa o resultado verdadeiro (V), e o outro o falso (F). Observe, na Figura 3, que temos um bloco de instruções que será executado para o cenário verdadeiro. Para o falso, a condição pula por completo o bloco de instruções e segue o fluxo do algoritmo.

Figura 3 – Fluxograma de uma estrutura de seleção simples



E como escrevemos a condicional simples em linguagem Python? A Figura 4 ilustra isso. Note que, em Python, a estrutura condicional é representada pela palavra *if*, que significa *se* em inglês, sendo, portanto, bastante intuitivo de se compreender seu significado.

Após o *if*, abrimos parênteses (é opcional) e colocamos a condição lógica que deve ser satisfeita para que o bloco de instruções seja executado. Após a condição devemos, obrigatoriamente, colocar dois pontos, nunca se esqueça dele!

Figura 4 – Construção da estrutura condicional em Python



Vamos comparar o código em Python com o pseudocódigo. A seguir, você encontra, do lado esquerdo, como fazemos a condição em pseudocódigo. O equivalente em Python está na direita.

| | |
|--|--|
| se (condição) # Instrução (ões) fim-se | if (condição): # Instrução (ões) |
|--|--|

O símbolo de # representa comentários em Python (vimos isso anteriormente). Veja que todas as instruções devem estar indentadas para serem reconhecidas como pertencentes ao bloco condicional. Ademais, a linguagem Python não tem um delimitador de final de estrutura, no qual no pseudocódigo é representado por *fim-se*. Muitas outras linguagens, como Java e C/C++, contêm um

delimitador. O Python trabalha com a indentação para identificar o que pertence ao seu bloco, portanto não requer um finalizador para ele.

Estamos falando bastante sobre como construir o código em Python, mas até o momento não praticamos nada e também não vimos como criar a tão comentada condição da estrutura. Vejamos um exemplo prático e executável em nossa ferramenta online. Lembrando que você pode inserir esse código em qualquer software de interpretação Python que desejar.

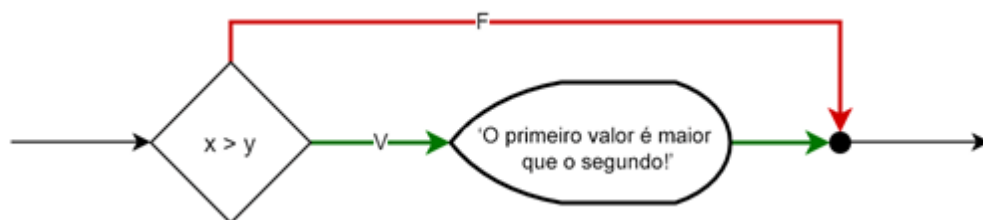
O exercício a seguir lê dois valores numéricos inteiros e compara se o primeiro é maior do que o segundo, utilizando uma condicional simples. Caso seja (resultado verdadeiro), ele imprime na tela a mensagem informando que o primeiro valor digitado é maior do que o segundo.

```
1  #Lê dois valores inteiros e compara ambos
2  x = int(input('Digite um valor inteiro: '))
3  y = int(input('Digite um segundo valor inteiro: '))
4  if (x > y):
5      print('O primeiro valor é maior que o segundo!')
```

☞ Digite um valor inteiro: 5
Digite um segundo valor inteiro: 2
O primeiro valor é maior que o segundo!

No exemplo, a condição imposta era de que x fosse maior do que y . Vejamos o fluxograma para esse mesmo exercício na Figura 5.

Figura 5 – Fluxograma de uma estrutura de seleção simples. Exemplo do maior e menor



Expandindo este exercício, podemos colocar duas condicionais simples. Uma realizando o teste para verificar se o primeiro valor é maior do que o segundo, e outra para verificar se o segundo é maior do que o primeiro. Veja como ficaria abaixo. Realize testes para diferentes valores de x e y .

```
1  #Lê dois valores inteiros e compara ambos
2  x = int(input('Digite um valor inteiro: '))
3  y = int(input('Digite um segundo valor inteiro: '))
4  if (x > y):
5      print('O primeiro valor é maior que o segundo!')
6  if (x < y):
7      print('O segundo valor é maior que o primeiro!')
```

➤ Digite um valor inteiro: 5
Digite um segundo valor inteiro: 6
O segundo valor é maior que o primeiro!

Sua vez de praticar

O que acontece caso ambos os valores (x e y) sejam iguais no exercício acima? O que irá aparecer na tela para o usuário? Explique.

2.2 INDENTAÇÃO

Gostaria de pausar aqui um pouco nosso estudo sobre condicionais para frisar a importância da indentação do código.

Indentar o código significa deixá-lo visualmente organizado e tabulado. Observe com atenção os exemplos que acabamos de resolver. Notou que a instrução que será, ou não, dentro do teste condicional, está com um recuo para a direita? Isso é proposital. Todo o código inserido em uma estrutura condicional deve ser deslocado para a direita a fim de garantir a compreensão de que aquelas instruções fazem parte da respectiva estrutura.

Na maioria das linguagens de programação, a indentação é um aspecto que não tem impacto no funcionamento do programa. Apesar disso, indentar um código é considerado uma boa prática de programação, assim como realizar comentários no seu código também é, prática esta que deve ser adotada por todo e qualquer bom programador. Isso porque indentar o código o deixa mais legível não só para o próprio desenvolver, mas também para outras pessoas que porventura venham a se deparar com seu algoritmo.

Em linguagem Python, a indentação assume um papel ainda mais importante. Isso porque a linguagem Python é uma das poucas que utiliza a própria indentação para delimitar seus blocos, como foi o caso da nossa condicional. Sendo assim, nunca deixe de indentar seus códigos em Python, caso contrário, seu algoritmo poderá nem mesmo executar!

2.3 CONDICIONAL COMPOSTA

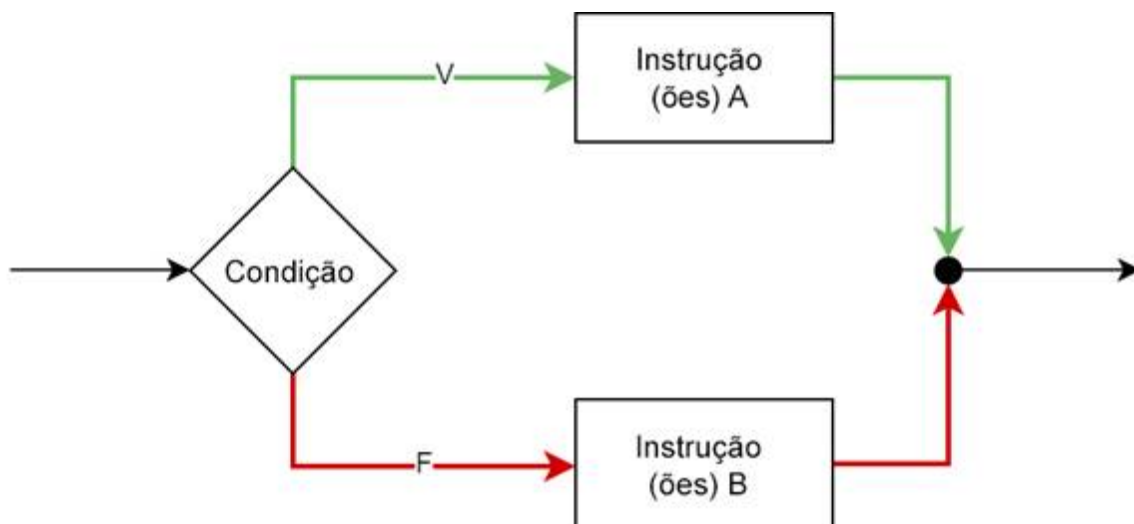
A estrutura condicional composta existe para quando precisamos decidir se um conjunto de instruções A precisa ser executado, ou se será outro conjunto de instruções B a ser executado.

Na Figura 6, temos o fluxograma da estrutura condicional composta. Se a condição a ser testada resultar em um valor lógico verdadeiro, observe na representação que executamos um conjunto A de instruções. Senão, o fluxo de execução segue por outro caminho, e outras instruções B são executadas.

Note que nunca acontecerá de ambos os conjuntos de instruções serem executadas em um mesmo fluxo de execução do algoritmo. Será sempre um ou outro. Nunca ambos.

Não obstante, o conjunto A e o conjunto B podem conter instruções completamente distintas entre si, podendo inclusive apresentar uma quantidade de instruções bastante diferentes. Não existem restrições quanto a isso.

Figura 6 – Fluxograma de uma estrutura de seleção composta



A escrita da condicional composta se dá da mesma maneira que a simples, em Python. A única diferença é que agora iremos acrescentar a palavra *else* (*senão*, em inglês), que irá representar o conjunto B de instruções a ser executado. Observe a comparação abaixo com o pseudocódigo.

| | |
|--|---|
| se (condição) # Instrução (ões) A senão | if (condição): # Instrução (ões) A else: |
|--|---|

| # Instrução (ões) B | # Instrução (ões) B |
|---------------------|---------------------|
| fim-se | |

O exemplo que vamos resolver agora descobre se um número inteiro digitado é par ou é ímpar. Para resolvermos esse exercício, devemos pensar em qual teste lógico deve ser feito para descobrirmos se um número é par ou ímpar.

Se pegarmos qualquer valor numérico e inteiro e dividirmos por 2, que é par, e essa divisão resultar em resto zero, sabemos que o número digitado é divisível por 2, e, portanto, é par[1]. Caso o resto desta divisão não dê zero, isso significa que o número não é divisível por 2, e, portanto, é ímpar.

Na linguagem Python, podemos obter o resto de uma divisão diretamente utilizando o símbolo de percentual. Deste modo, podemos escrever a expressão lógica `x % 2 == 0`, ou seja, obtemos o resto da divisão por 2 e, em seguida, pela comparação lógica de igualdade verificamos se é igual a zero.

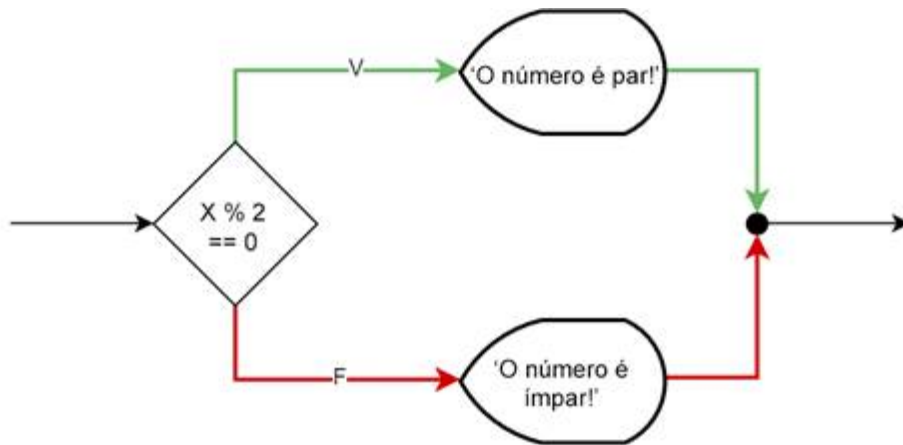
Ademais, a ordem de precedência dos operadores também deve ser respeitada aqui. Primeiro, o cálculo aritmético é realizado, e por último, o teste lógico. Ficamos com o algoritmo a seguir.

```
1  # par ou ímpar
2  x = int(input('Digite um valor inteiro: '))
3  if (x % 2 == 0):
4      print('O número é par!')
5  else:
6      print('O número é ímpar!')
```

☞ Digite um valor inteiro: 101
O número é ímpar!

Sendo assim, a cada execução desse programa, ou a linha de Código 4 ou a Linha 6 será executada. Nunca ambas. O fluxograma desses exercícios está colocado na Figura 7.

Figura 7 – Fluxograma de uma estrutura de seleção composta. Exemplo do par ou ímpar



2.4 COMPARANDO O DESEMPENHO

Acabamos de resolver o algoritmo do par ou ímpar utilizando uma condicional composta. E funcionou perfeitamente! Porém, e se resolvêssemos escrever o mesmo problema, mas agora empregando somente condicionais simples, é possível? A resposta é sim. Vejamos o resultado a seguir.

```
1 # par ou ímpar
2 x = int(input('Digite um valor inteiro: '))
3 if (x % 2 == 0):
4     print('O número é par!')
5 if (x % 2 == 1):
6     print('O número é ímpar!')
```

➤ Digite um valor inteiro: 101
O número é ímpar!

Na Linha 5, acabamos por substituir o *else* da condição composta por outro teste simples. Nesse novo teste, agora estamos testando se o resto da divisão por 2 resulta no valor 1, caso resulte, será ímpar. Sendo assim, se substituirmos a condicional composta por duas condicionais simples, o algoritmo também funcionará corretamente.

Você pode estar se perguntando agora, ambas maneiras estão corretas? Posso utilizar qualquer uma? Bom, de fato, ambas funcionam, porém, a solução que acabamos de ver não é recomendável para esse exercício.

E existe um motivo muito importante para que não se utilize duas condicionais simples neste caso: o desempenho do algoritmo, ou seja, a velocidade com que ele executa.

Para compreender melhor isso, precisamos entender o que o computador faz para executar uma instrução como a da linha, ou Linha 5. Neste caso, a CPU realizará dois cálculos: primeiro, a divisão, e em seguida, o teste de igualdade lógico. Portanto, são duas operações a cada teste condicional realizado.

Caso estivermos com nosso algoritmo realizando o teste lógico em uma estrutura composta, o teste será feito uma única vez e, dependendo do seu resultado (V ou F), irá saltar para uma determinada parte do código e executar. A estrutura do *else* não tem nenhum teste para ser feito, portanto não gera custo computacional dentro da CPU.

Agora, caso nosso algoritmo for construído com duas condicionais simples, faremos o mesmo teste (divisão e igualdade) duas vezes. O dobro!

Imagine uma situação em que o número digitado é par. No algoritmo com condicional composta, a condição dará verdadeira e a Linha 4 aparece na tela, encerrando o programa.

Já no algoritmo com condicionais simples, independentemente de o nosso número já ter sido identificado como par, por exemplo, o teste para ímpar ainda será desenvolvido (e dará falso), o que irá gerar o dobro de custo de execução dentro da CPU. E isso deve ser evitado a todo o custo.

Em resumo, sempre que for possível substituir 2 testes simples por um composto, o faça! Certo, mas como identificar se é possível fazer essa substituição? Não existe uma receita pronta para isso. Porém, a dica aqui é: sempre verifique se os seus 2 testes simples servem para o mesmo propósito dentro do seu programa.

No nosso exercício, ambos testes serviam para verificar se o número é par ou ímpar, portanto têm o mesmo objetivo e podem ser trocados por um único teste composto.

Expandindo o exercício. Se quiséssemos, além de verificar se o número é par ou ímpar, encontrar se o número é múltiplo de 7, poderíamos, agora, criar duas condicionais compostas e separadas. Isso porque o teste para o múltiplo de 7 distingue por completo da verificação de um número ser, ou não, par.

A seguir, temos o algoritmo com duas condicionais compostas. Para a verificação do múltiplo de 7, podemos novamente fazer o teste com o resto da divisão, pois um múltiplo de 7 deve, obrigatoriamente, ser divisível por 7 (resto zero).

```
1  # par ou ímpar e múltiplo de 7
2  x = int(input('Digite um valor inteiro: '))
3  if (x % 2 == 0):
4      print('O número é par!')
5  else:
6      print('O número é ímpar!')
7  if (x % 7 == 0):
8      print('Múltiplo de 7!')
9  else:
10     print('Não é múltiplo de 7!')
```

☞ Digite um valor inteiro: 21
O número é ímpar!
Múltiplo de 7!

2.5 EXERCÍCIOS

Vamos praticar alguns exercícios de condicional simples e composta.

Exercício 2.5.1: desenvolva um algoritmo que solicite o seu ano de nascimento e o ano atual. Calcule a sua idade e apresente na tela.

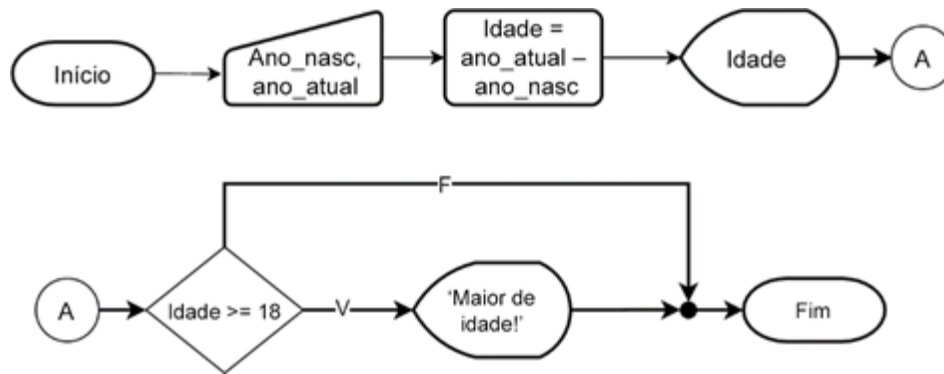
Para fins de simplificação, despreze o dia e o mês do ano. Após o cálculo, verifique se a idade é maior ou igual a 18 anos e apresente na tela uma mensagem informando que já é possível tirar a carteira de motorista caso seja de maior.

Python

```
1  # Exercício 2.5.1
2  ano_nasc = int(input('Qual seu ano de nascimento? '))
3  ano_atual = int(input('Em que ano estamos? '))
4  idade = ano_atual - ano_nasc
5  print('Você tem {} anos de idade.'.format(idade))
6  if (idade >= 18):
7      print('Você é de maior. Já pode tirar a carteira de motorista!')
```

☞ Qual seu ano de nascimento? 2001
Em que ano estamos? 2020
Você tem 19 anos de idade.
Você é de maior. Já pode tirar a carteira de motorista!

Fluxograma



Exercício 2.5.2 (adaptado de Puga, p. 44): uma empresa concedeu um bônus de 20% para todos seus funcionários com mais de 5 anos de empresa. Todos os outros que não se enquadram nessa categoria receberam uma bonificação de 10%, somente. Escreva um algoritmo que leia o salário do funcionário e seu tempo de empresa, e apresente a bonificação de cada funcionário na tela.

Python

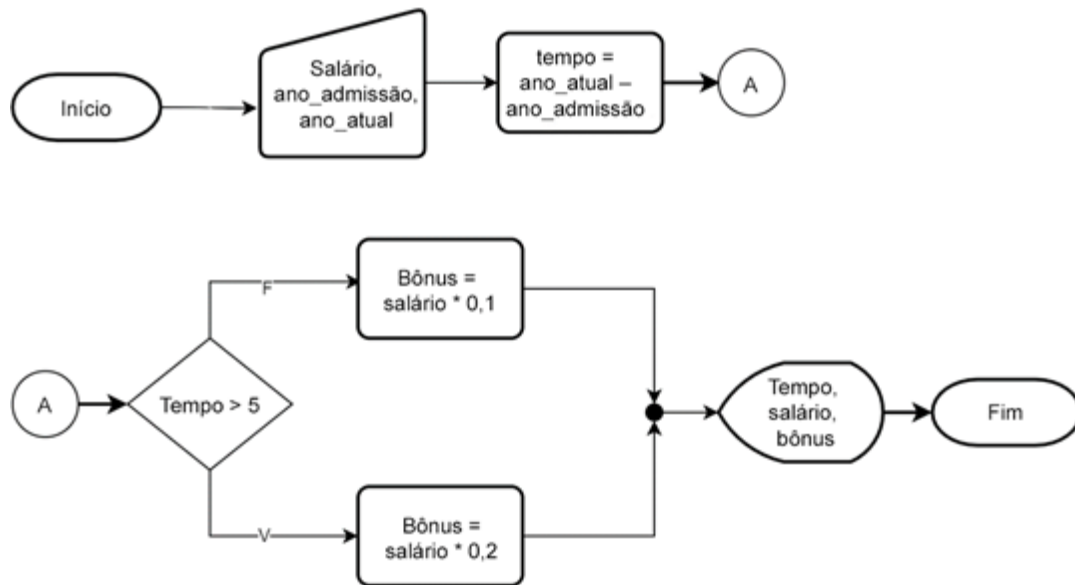
```

1  # Exercício 2.5.2
2  salario = float(input('Qual seu salário?'))
3  ano_admissao = int(input('Qual seu ano de admissão na empresa?'))
4  ano_atual = int(input('Em que ano estamos? '))
5  tempo = ano_atual - ano_admissao
6  if (tempo > 5):
7      bonus = salario * 0.2
8  else:
9      bonus = salario * 0.1
10 print('Você tem {} anos dentro desta empresa.'.format(tempo))
11 print('Seu salário é de {}'.format(salario))
12 print('E sua bonificação é de {}'.format(bonus))
13 print('Salário final {}'.format(bonus + salario))

```

Qual seu salário?1000
Qual seu ano de admissão na empresa?2018
Em que ano estamos? 2020
Você tem 2 anos dentro desta empresa.
Seu salário é de 1000.0.
E sua bonificação é de 100.0.
Salário final 1100.0.

Fluxograma



TEMA 3 – EXPRESSÕES LÓGICAS E ÁLGEBRA BOOLEANA

No tópico anterior, aprendemos a construir algoritmos com estruturas condicionais simples e compostas. Você deve ter percebido que a condição utilizada dentro da estrutura *if* é fundamental para que o algoritmo funcione como deveria.

Deste modo, vamos dedicar um tópico inteiro nesta aula para estudarmos mais a fundo as expressões lógicas que podemos criar, bem como estudaremos a álgebra *booleana*, essencial para a construção de expressões lógicas mais complexas.

3.1 OPERADORES LÓGICOS/BOOLOEANOS

Aprendemos em conteúdos anteriores os principais operadores aritméticos e também os operadores relacionais. Todos esses operadores podem ser combinados em expressões lógicas que veremos em breve. Porém, nos restou ainda aprender um terceiro tipo de operador, que será investigado agora, o operador lógico (ou operador *booleano*).

Esses operadores servem para agrupar operações e expressões lógicas e têm comportamentos distintos. O Python, por padrão, suporta três operadores lógicos, embora existam outros na álgebra de *Boole*: o operador de conjunção (*and*), de disjunção (*or*) e o de negação (*not*). Em linguagem Python, escrevemos esses operadores exatamente como se falamos em inglês, portanto:

Tabela 2 – Lista de operadores lógicos em Python e pseudocódigo

| Python | Pseudocódigo | Operação |
|--------|--------------|----------|
|--------|--------------|----------|

| | | |
|------------|-----|-----------|
| <i>not</i> | não | negação |
| <i>and</i> | e | conjunção |
| <i>or</i> | ou | disjunção |

Cada operador contém um conjunto de regras e são expressos pelo que chamamos de tabela verdade do operador. **A tabela verdade demonstra o resultado de uma operação com um ou dois valores lógicos.** O operador de negação é um operador unitário, já os de conjunção e disjunção são binários, ou seja, requerem dois operadores.

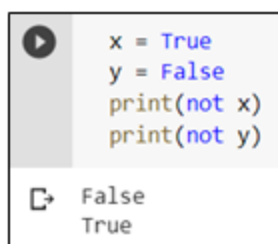
Operador not: Vamos iniciar pelo mais simples deles. O operador de negação *not* serve para negar um resultado lógico ou o resultado de uma expressão *booleana*. Na prática, isso significa que o resultado final de uma expressão será invertido. A tabela verdade do *not* é apresentada na Tabela 3, sendo que a coluna *V* representa um valor *booleano*.

Como em Python um valor *booleano* verdadeiro é representado pelo seu termo em inglês *True*, e o falso é *False*, vamos adotar essa nomenclatura a partir daqui.

Tabela 3 – Tabela verdade do operador not (não)

| <i>V</i> | <i>not V</i> |
|----------|--------------|
| True | False |
| False | True |

Façamos um teste bastante simples no Python. Crie duas variáveis quaisquer e atribua a elas ambos valores lógicos existentes (Atenção! Lembre-se que a linguagem Python compreende os valores lógicos escritos sempre com a primeira letra da palavra em maiúscula. Portanto, *true* e *false* não irão funcionar). Em seguida, faça um *print* da variável negada. Veja que sempre o resultado na tela será o inverso:



```

x = True
y = False
print(not x)
print(not y)

```

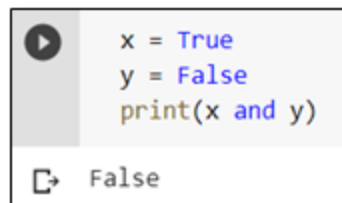
False
True

Operador and: O operador binário and utiliza dois valores, que chamaremos de *valores de entrada*, e irá gerar um valor como resultado de saída. Esse operador, além disso, irá prover um resultado verdadeiro se, e somente se, ambas entradas forem verdadeiras. A tabela verdade do and é apresentada na Tabela 4, em que as colunas V_1 e V_2 representam um valor booleano. Todas as combinações binárias possíveis estão apresentadas na tabela.

Tabela 4 – Tabela verdade do operador and (e)

| V_1 | V_2 | $V_1 \text{ and } V_2$ |
|-------|-------|------------------------|
| False | False | False |
| False | True | False |
| True | False | False |
| True | True | True |

Realize os testes abaixo para todas as combinações possíveis da tabela verdade e veja se os resultados coincidem com ela.



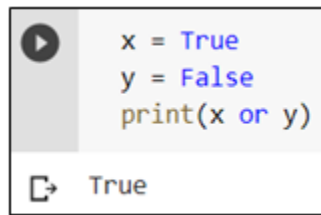
Operador or: O operador binário or utiliza dois valores, que chamaremos de *valores de entrada*, e irá gerar um valor como resultado de saída. Esse operador irá prover um resultado verdadeiro caso ao menos uma das entradas forem verdadeiras. A tabela verdade do or é apresentada a seguir, em que as colunas V_1 e V_2 representam um valor *booleano*. Todas as combinações binárias possíveis estão apresentadas na Tabela 5.

Tabela 5 – Tabela verdade do operador or (ou)

| V_1 | V_2 | $V_1 \text{ or } V_2$ |
|-------|-------|-----------------------|
| False | False | False |
| False | True | True |
| True | False | True |
| True | True | True |

| | | |
|------|------|------|
| True | True | True |
|------|------|------|

Realize os testes abaixo para todas as combinações possíveis da tabela verdade e veja se os resultados coincidem com ela.



```
x = True
y = False
print(x or y)
```

True

3.2 EXPRESSÕES LÓGICAS/BOOLOEANAS

Agora que vimos os operadores lógicos, vamos então aprender a combiná-los em expressões lógicas. Podemos fazer isso, inclusive, inserindo operadores relacionais e aritméticos também.

Porém, primeiramente, precisamos deixar bem claro a ordem de precedência na execução de cada operação pela linguagem de programação. No Python, assim como todas as linguagens de programação usuais, existe uma ordem com que cada operação é realizada pelo programa.

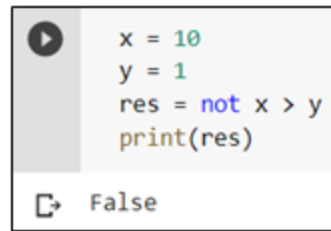
De modo geral, para operações aritméticas, a ordem de execução segue o que já aprendemos na matemática básica da escola. Por exemplo, divisão e multiplicação sempre executa antes da adição e subtração. Mas aqui temos também os operadores relacionais e os lógicos. Portanto, a ordem com que cada um é executado está colocada a seguir:

- 1) Parênteses;
- 2) Operadores aritméticos de potenciação ou raiz;
- 3) Operadores aritméticos de multiplicação, divisão e módulo;
- 4) Operadores aritméticos de adição e subtração;
- 5) Operadores relacionais;
- 6) Operadores lógicos *not*;
- 7) Operadores lógicos *and*;
- 8) Operadores lógicos *or*;

9) Atribuições.

Todas as operações de mesmo grau de importância são executadas sempre da esquerda para a direita na seguinte pelo programa. Por exemplo, se tivermos duas adições numa mesma expressão, primeiro a mais à esquerda ocorre antes.

Vejamos uma série de exemplos para colocar em prática tudo isso. No primeiro exemplo, temos duas variáveis inteiras com um teste relacional, e após uma negação.

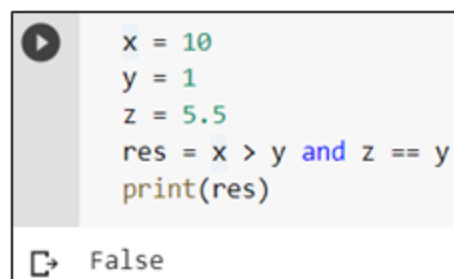


```
x = 10
y = 1
res = not x > y
print(res)
```

False

Seguindo a precedência apresentada, primeiro é realizado o operador relacional, fazendo o teste $10 > 1$, que irá resultar em *True*. Após, ele é negado, invertendo e ficando *False*.

No próximo exemplo, temos 3 variáveis numéricas. Na expressão lógica, temos duas operações relacionais, que acontecerão primeiro. E por fim, temos o operador *and*.

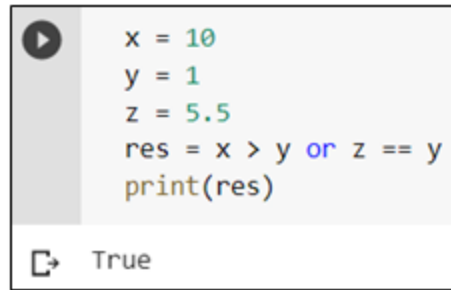


```
x = 10
y = 1
z = 5.5
res = x > y and z == y
print(res)
```

False

Temos duas operações de mesmo nível de importância na mesma expressão (operadores relacionais). Iniciamos pelo resultado de $x > y$, que será *True*. Já $z == y$ será *False*. Em seguida, teremos a operação de conjunção *True and False*, que irá resultar em *False*, pois, lembrando que uma *and* só fornece uma saída *True* caso ambas entradas também sejam *True*.

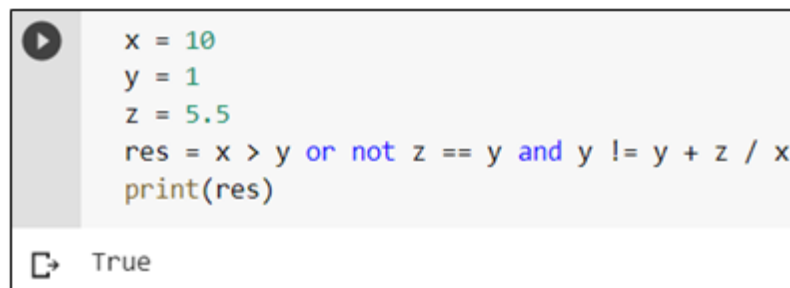
Temos uma expressão bastante semelhante a seguir. Porém, agora, trocamos uma *and* por uma *or*. Assim, teremos a disjunção *True or False*, que irá resultar em *True*, pois a *or* fornece uma saída *True* caso ao menos uma das entradas também sejam *True*.



```
x = 10
y = 1
z = 5.5
res = x > y or z == y
print(res)
```

True

No quarto exemplo, temos uma expressão bastante longa e aparentemente complexa. Porém, se seguirmos a precedência dos operadores, veremos como não é complexo compreendermos seu resultado.



```
x = 10
y = 1
z = 5.5
res = x > y or not z == y and y != y + z / x
print(res)
```

True

A seguir, é apresentada a sequência de passos que levam até o resultado *True* apresentado como saída:

- 1) Operação aritmética de divisão: $5.5 / 10 = 0.55$;
- 2) Operação aritmética de adição $y + 0.55$: $1 + 0.55 = 1.55$;
- 3) Operações lógicas relacionais. Como temos três delas, ficam:
 - a. $x > y \rightarrow 10 > 1 = True$;
 - b. $z == y \rightarrow 5.5 == 1 = False$;
 - c. $y != 1.55 \rightarrow 1 != 1.55 = True$;
- 4) Operação lógica *not* $False = True$;
- 5) Operação lógica *and*: $True \text{ and } True = True$;
- 6) Operação lógica *or*: $True \text{ or } True = True$

3.3 EXERCÍCIOS DE FIXAÇÃO

Suponha que A, B, C, X, Y e Z são variáveis numéricas, e V1 e V2 variáveis *booleanas*, e Nome e Rua são *strings*, com os respectivos valores:

$A = 1, B = 2, C = 3, X = 20, Y = 10, Z = -1;$

$V1 = \text{True}, V2 = \text{False};$

$\text{Nome} = \text{'Pedro'}, \text{Rua} = \text{'Pedrinho'}.$

Encontre o resultado das expressões lógicas abaixo:

a) $A + C / B$

b) $C / B / A$

c) $-X ** B$

d) $(-X) ** B$

e) $V1 \text{ or } V2$

f) $V1 \text{ and not } V2$

g) $V2 \text{ and not } V1$

h) $\text{not Nome} == \text{Rua}$

i) $V1 \text{ and not } V2 \text{ or } V2 \text{ and not True}$

j) $X > Y \text{ and } C \leq B$

k) $C - 3 * A < X + 2 * Z$

3.4 EXERCÍCIOS

Agora vamos juntar nossos conhecimentos resolvendo alguns exercícios de condicionais simples e compostas em que o uso de expressões lógicas na condição de decisão é fundamental.

Exercício 3.3.1 (adaptado de Menezes, p. 60): Um aluno, para passar de ano, precisa estar aprovado em todas as matérias que ele está cursando.

Assuma que a média para aprovação é a partir de 7, e que o aluno cursa 3 matérias, somente. Escreva um algoritmo que leia a nota final do aluno em cada matéria, e informe na tela se ele passou de ano ou não.

Python

```
# Exercício 3.3.1
m1 = float(input('Digite a nota da 1ª matéria:'))
m2 = float(input('Digite a nota da 2ª matéria:'))
m3 = float(input('Digite a nota da 3ª matéria:'))
if m1 >= 7 and m2 >= 7 and m3 >= 7:
    print('O aluno está aprovado de ano!');
else:
    print('O aluno não passou de ano!');
```

➤ Digite a nota da 1ª matéria:8.5
Digite a nota da 2ª matéria:6.2
Digite a nota da 3ª matéria:7.7
O aluno não passou de ano!

Exercício 3.3.2: Escreva um algoritmo que lê um valor inteiro qualquer. Após, verifique se este valor está contido dentro dos seguintes intervalos: $-100 < x < -1$ e $1 < x < 100$. Imprima na tela uma mensagem caso ele esteja em um dos intervalos.

Python

```
# Exercício 3.3.2
x = int(input('Digite um valor qualquer inteiro:'))
if x >= 1 and x <= 100 or x >= -100 and x <= -1:
    print('Um dos critérios foi atendido!');
```

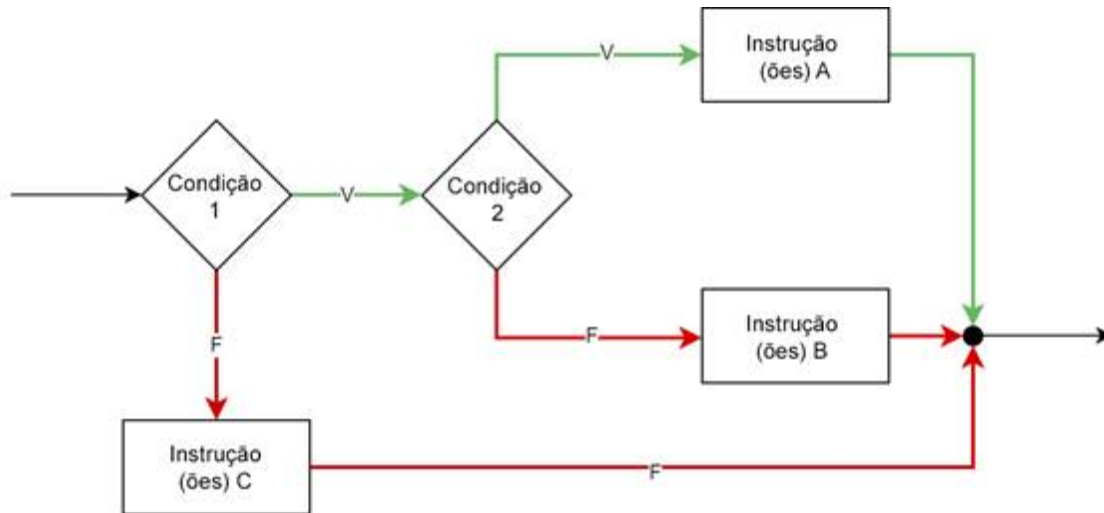
➤ Digite um valor qualquer inteiro:-1
Um dos critérios foi atendido!

TEMA 4 – CONDICIONAIS ANINHADAS

Muitos dos programas computacionais envolvem uma complexidade maior do que somente empregar uma única estrutura condicional simples, ou mesmo uma composta. É bastante comum que venhamos a utilizar duas ou mais condicionais aninhadas em um mesmo algoritmo, ou seja, uma condicional dentro da outra.

Na Figura 6, vemos duas estruturas condicionais aninhadas representadas em um fluxograma. Entenda que podemos aninhar quantas condicionais quisermos, e não somente duas. Para fins didáticos, focaremos na representação de duas, somente.

Figura 6 – Fluxograma de uma estrutura de seleção com dois níveis de aninhamento



No fluxograma, ambas condicionais são compostas. Perceba que, caso a Condição 1 resulte em verdadeiro, caímos diretamente na segunda estrutura de decisão. Caso a segunda também resulte em verdadeiro, executamos um conjunto de instruções A. Caso a segunda condição resulte em falso, executamos B. Agora, caso na primeira condição tivéssemos obtido um resultado falso imediatamente, nem iríamos para o segundo teste e executaríamos as instruções C.

Agora, vamos resolver um exemplo para que você compreenda melhor como aninhar estruturas de decisão. Vamos retomar o Exercício 2.5.2 em que uma empresa concedeu um bônus de 20% para todos seus funcionários com mais de 5 anos de empresa e todos os outros receberam uma bonificação de 10%.

Vamos inserir mais uma camada de restrição neste exercício. Agora, além dos critérios acima, funcionários com mais de 10 anos de empresa têm direito a uma bonificação de 30%.

Neste caso, temos agora 3 possíveis resultados de bonificação: 30%, 20% e 10%. No algoritmo em Python, podemos inserir uma primeira condicional composta que verifica se o funcionário tem mais do que 10 anos de empresa. Caso a condição não seja satisfeita, caímos no *else* e, dentro dele, temos um segundo teste condicional que verifica o tempo de 5 anos. Por fim, caso nem essa

condição seja satisfeita, caímos no segundo *else*, que corresponde à bonificação de 10%. O código em linguagem Python está colocado a seguir.

```
1 # Exercício 2.5.2
2 salario = float(input('Qual seu salário?'))
3 ano_admissao = int(input('Qual seu ano de admissão na empresa?'))
4 ano_atual = int(input('Em que ano estamos? '))
5 tempo = ano_atual - ano_admissao
6 if (tempo > 10):
7     bonus = salario * 0.3
8 else:
9     if (tempo > 5):
10        bonus = salario * 0.2
11    else:
12        bonus = salario * 0.1
13 print('Você tem {} anos dentro desta empresa.'.format(tempo))
14 print('Seu salário é de {}'.format(salario))
15 print('E sua bonificação é de {}'.format(bonus))
16 print('Salário final {}'.format(bonus + salario))
```

Qual seu salário?1000
Qual seu ano de admissão na empresa?1980
Em que ano estamos? 2020
Você tem 40 anos dentro desta empresa.
Seu salário é de 1000.0.
E sua bonificação é de 300.0.
Salário final 1300.0.

Percebeste a condicional dentro da outra? Então nunca esqueça da indentação! Observe que tudo colocado dentro do primeiro teste deve estar recuado para a direita. Portanto, toda a segunda condicional, inserida na primeira, deve estar recuada. Assim como tudo que estiver dentro da segunda condição deve estar recuado novamente.

Sua vez de praticar

Puxe na memória o que você estudou na seção 2.4 Comparando desempenho.

No algoritmo que acabemos de resolver, se em vez de duas condicionais aninhadas, usássemos elas de maneira separada, testando individualmente cada necessidade do exercício, o algoritmo teria um desempenho igual ou inferior ao resolvido acima? Por quê?

4.1 EXERCÍCIOS

Vamos praticar alguns exercícios de condicionais aninhadas.

Exercício 4.1.1: Escreva um algoritmo em Python em que o usuário escolhe se ele quer comprar maçãs, laranjas ou bananas. Deverá ser apresentado na tela um menu com a opção 1 para maçã, 2 para laranja e 3 para banana.

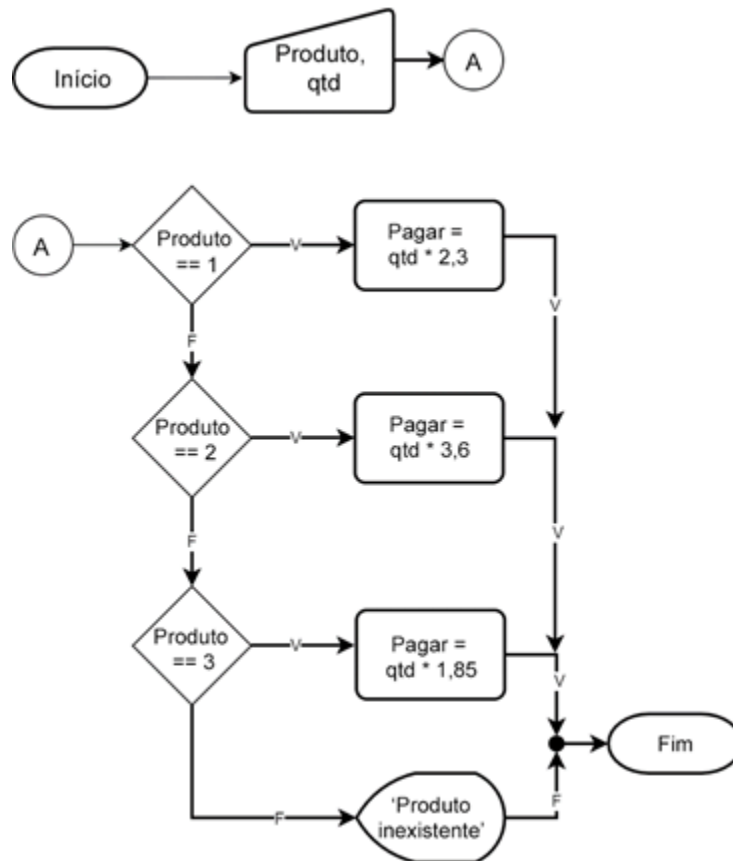
Após escolhida a fruta, deve-se digitar quantas unidades se quer comprar. O algoritmo deve calcular o preço total a pagar do produto escolhido e mostrá-lo na tela. Considere que uma maçã custa R\$ 2,30, uma laranja R\$ 3,60 e uma banana 1,85.

Python

```
1  # Exercício 4.1.1
2  print('Escolha o que deseja comprar:')
3  print('1 - Maça')
4  print('2 - Laranja')
5  print('3 - Banana')
6  produto = int(input('Qual sua escolha?'))
7  qtd = int(input('Quantas unidades?'))
8  if (produto == 1):
9      pagar = qtd * 2.3
10     print('Você comprou {} maçãs. Total à pagar: {}'.format(qtd, pagar))
11 else:
12     if (produto == 2):
13         pagar = qtd * 3.6
14         print('Você comprou {} laranjas. Total à pagar: {}'.format(qtd, pagar))
15     else:
16         if (produto == 3):
17             pagar = qtd * 1.85
18             print('Você comprou {} bananas. Total à pagar: {}'.format(qtd, pagar))
19         else:
20             print('Produto inexistente!')
```

Escolha o que deseja comprar:
1 - Maça
2 - Laranja
3 - Banana
Qual sua escolha?3
Quantas unidades?2
Você comprou 2 bananas. Total à pagar: 3.7

Fluxograma



Exercício 4.1.2 (adaptado de Puga, p. 48): Faça um algoritmo que receba três valores, representando os lados de um triângulo fornecidos pelo usuário. Verifique se os valores formam um triângulo e classifique como:

- a) Equilátero (três lados iguais);
- b) Isósceles (dois lados iguais);
- c) Escaleno (três lados diferentes).

Lembre-se de que, para formar um triângulo, nenhum dos lados pode ser igual a zero e um lado não pode ser maior do que a soma dos outros dois.

Python

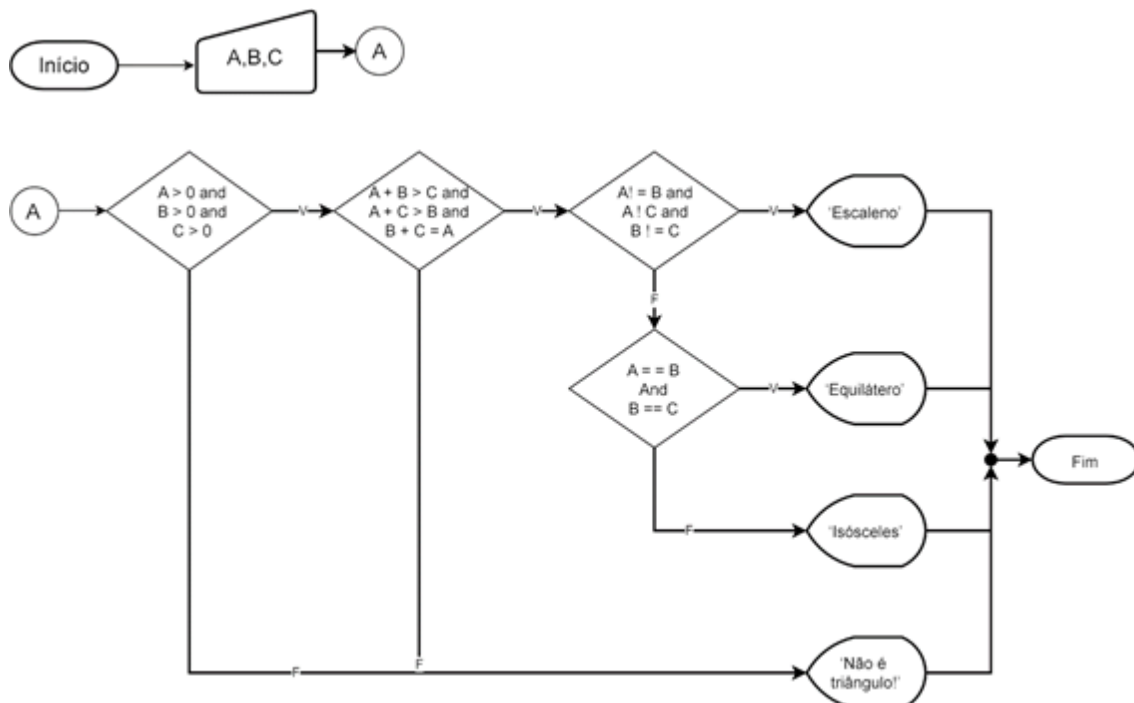
```

1 # Exercício 4.1.2
2 A = int(input('Digite o 1º lado do triângulo:'))
3 B = int(input('Digite o 2º lado do triângulo:'))
4 C = int(input('Digite o 3º lado do triângulo:'))
5 if A > 0 and B > 0 and C > 0:
6     if A + B > C and A + C > B and B + C > A:
7         #Se chegou até aqui, é porque o triângulo é válido
8         if A != B and A != C and B != C:
9             print('Triângulo escaleno!')
10        else:
11            if A == B and B == C:
12                print('Triângulo equilátero')
13            else:
14                print('Triângulo isósceles')
15        else:
16            print('Ao menos um dos valores indicados não servem para formar um triângulo.')
17    else:
18        print('Ao menos um dos valores indicados não servem para formar um triângulo.')

```

Digite o 1º lado do triângulo:4
 Digite o 2º lado do triângulo:5
 Digite o 3º lado do triângulo:6
 Triângulo escaleno!

Fluxograma



TEMA 5 – CONDICIONAIS DE MÚLTIPLA ESCOLHA (ELIF)

Durante os programas que desenvolveu utilizando condicionais aninhadas, deve ter percebido que, quanto mais aninhamentos você fazia, mais complexo de organizar o código ia ficando, pois era necessário ficar bastante atento aos recuos.

Existe uma maneira de deixar a implementação com múltiplos aninhamentos mais atrativa e legível para o desenvolvedor, a qual chamamos de *condicional de múltipla escolha*. Essa solução não será sempre implementável para substituir qualquer aninhamento, mas pode ajudar em algumas situações.

A condicional múltipla existe em as linguagens de programação, no pseudocódigo chamamos ela de *condicional escolha*. Ela tem como objetivo facilitar e organizar melhor um código com múltiplos *if*, quando todos os *if* precisam realizar testes com uma mesma variável. Esse tipo de condicional é ótima para criar menu de opções, por exemplo, ou gerenciar catálogo de produtos e cadastros.

Em linguagem Python, escrevemos a múltipla escolha utilizando uma cláusula chamada de *elif*. A palavra *elif* é uma junção da palavra *else* com a palavra *if*, e irá substituir um *else* e um *if* aninhados por ela. Veja o exemplo a seguir.

Note que lemos primeiro um nome e uma idade. Aí, caso o nome seja Vinicius, uma mensagem aparece na tela. Agora, caso você digite qualquer outra palavra ao invés de Vinicius, o programa irá cair em um dos dois no *elif* e irá executar ou a Linha 6 ou a Linha 8, dependendo da idade que você digitou.

```
1 nome = input('Qual seu nome?')
2 idade = int(input('Qual sua idade?'))
3 if nome == 'Vinicius':
4     print('Olá, Vinicius!')
5 elif idade < 18:
6     print('Você não é o Vinicius! E é menor de idade!')
7 elif idade > 100:
8     print('Diferente de você, o Vinicius não é imortal!')
```

Qual seu nome?João
Qual sua idade?160
Diferente de você, o Vinicius não é imortal!

Vejamos outro exemplo. Vamos praticar esta estrutura retomando o exercício da compra de frutas, o Exercício 4.1.1. Nele havíamos aninhado três estruturas condicionais compostas com o objetivo de, para cada uma delas, calcular um dos produtos. Vamos reescrever esse exercício, mas agora com a estrutura *elif*.

```
1 # Exercício 4.1.1 com elif
2 print('Escolha o que deseja comprar:')
3 print('1 - Maça')
4 print('2 - Laranja')
5 print('3 - Banana')
6 produto = int(input('Qual sua escolha?'))
7 qtd = int(input('Quantas unidades?'))
8 if (produto == 1):
9     pagar = qtd * 2.3
10    print('Você comprou {} maçãs. Total à pagar: {}'.format(qtd, pagar))
11 elif (produto == 2):
12     pagar = qtd * 3.6
13     print('Você comprou {} laranjas. Total à pagar: {}'.format(qtd, pagar))
14 elif (produto == 3):
15     pagar = qtd * 1.85
16     print('Você comprou {} bananas. Total à pagar: {}'.format(qtd, pagar))
17 else:
18     print('Produto inexistente!')
```

Escolha o que deseja comprar:
1 - Maça
2 - Laranja
3 - Banana
Qual sua escolha?3
Quantas unidades?2
Você comprou 2 bananas. Total à pagar: 3.7

Na nova solução para esse problema, observe que as palavras *else* e *if* fundiram-se em uma só (*elif*). Após a palavra *elif*, sempre existirá uma condição também.

Neste exercício, note que após todos os *elif* existe um *else*. Esse *else* é opcional, e existe para que caso nenhuma das opções da múltipla escolha seja satisfeita, o *else* acontecerá.

A implementação usando a múltipla escolha não tem impacto no desempenho do algoritmo em relação a solução aninhada. Ou seja, algoritmo não tende a executar mais rápido, ou mais lento, devido a ela.

O que difere o *elif* de estruturas aninhadas é que nosso código se torna mais simples de se compreender, pois trabalhamos com menos recuos. Imagine que você deseja adicionar mais frutas para vender no algoritmo acima. Utilizar o *elif* facilita esse processo também, pois basta que insira mais clausulas destas.

5.1 EXERCÍCIOS

Vamos praticar alguns exercícios de condicional de múltipla escolha. A condição de fluxogramas com *elif* se dá da mesma maneira que com aninhamentos. Portanto, não serão apresentados os fluxogramas dos exercícios a seguir.

Exercício 5.1.1 (adaptado de Menezes, p. 83): Escreva um algoritmo que leia dois valores numéricos e que pergunte ao usuário qual operação ele deseja realizar: adição (+), subtração (-),

multiplicação (*) ou divisão (/). Exiba na tela o resultado da operação desejada.

Python

```

1  # Exercício 5.1.1
2  print('CALCULADORA:')
3  print('+ Adição')
4  print('- Subtração')
5  print('* Multiplicação')
6  print('/ Divisão')
7  print('Pressione outra tecla para sair')
8  op = input('Qual operação deseja fazer?')
9  x = int(input('Digite o primeiro valor: '))
10 y = int(input('Digite o segundo valor: '))
11 if (op == '+'):
12     res = x + y
13     print('Resultado: {} + {} = {}'.format(x, y, res))
14 elif (op == '-'):
15     res = x - y
16     print('Resultado: {} - {} = {}'.format(x, y, res))
17 elif (op == '*'):
18     res = x * y
19     print('Resultado: {} * {} = {}'.format(x, y, res))
20 elif (op == '/'):
21     res = x / y
22     print('Resultado: {} / {} = {}'.format(x, y, res))
23 else:
24     print('Operação inválida!')

```

CALCULADORA:
 + Adição
 - Subtração
 * Multiplicação
 / Divisão
 Pressione outra tecla para sair
 Qual operação deseja fazer?
 Digite o primeiro valor: 4
 Digite o segundo valor: 3
 Resultado: 4 * 3 = 12

Exercício 5.1.2: Uma loja de departamentos está oferecendo diferentes formas de pagamento, conforme opções listadas a seguir. Faça um algoritmo que leia o valor total de uma compra e calcule o valor do pagamento final de acordo com a opção escolhida.

Se a escolha for por pagamento parcelado, calcule também o valor de cada parcela. Ao final, apresente o valor total da compra e o valor das parcelas:

- Pagamento à vista – conceder desconto de 5%;
- Pagamento em 3x – o valor não sofre alterações;
- Pagamento em 5x – acréscimo de 2%;
- Pagamento em 10x – acréscimo 8%.

Python

```
1 # Exercício 5.1.2
2 print('PAGAMENTO:')
3 print('1 - à vista')
4 print('2 - Parcelamento em 3x')
5 print('3 - Parcelamento em 5x')
6 print('4 - Parcelamento em 10x')
7 print('Pressione outra tecla para sair')
8 op = int(input('Qual forma de pagamento deseja fazer?'))
9 valor = float(input('Qual o preço do produto? '))
10 if (op == 1): #à vista desconto de 5%
11     valor_final = valor * 0.95
12     print('Produto comprado à vista. Total a pagar: {}'.format(valor_final))
13 elif (op == 2): #em 3x. sem alterações
14     valor_final = valor
15     parcela = valor_final / 3
16     print('Produto parcelado em 3x. Total a pagar: {}. Valor da parcela: {}'.format(valor_final, parcela))
17 elif (op == 3): #em 5x. acréscimo de 2%
18     valor_final = valor * 1.02
19     parcela = valor_final / 5
20     print('Produto parcelado em 5x. Total a pagar: {}. Valor da parcela: {}'.format(valor_final, parcela))
21 elif (op == 4): #em 10x. acréscimo de 8%
22     valor_final = valor * 1.08
23     parcela = valor_final / 10
24     print('Produto parcelado em 10x. Total a pagar: {}. Valor da parcela: {}'.format(valor_final, parcela))
25 else:
26     print('Operação inválida!')
```

PAGAMENTO:
1 - à vista
2 - Parcelamento em 3x
3 - Parcelamento em 5x
4 - Parcelamento em 10x
Pressione outra tecla para sair
Qual forma de pagamento deseja fazer?4
Qual o preço do produto? 1000
Produto parcelado em 10x. Total a pagar: 1080.0. Valor da parcela: 108.0

FINALIZANDO

Nesta aula, aprendemos todos os tipos de estruturas condicionais existentes na literatura e sua aplicação em linguagem Python. Aprendemos sobre estruturas condicionais simples, compostas, aninhadas e de múltipla escolha (elif).

Reforçamos que tudo o que vimos nesta aula continuará a ser utilizado de maneira extensiva ao longo dos próximos conteúdos, portanto pratique e resolva todos os exercícios do seu material.

REFERÊNCIAS

PUGA, S.; RISSETI, G. **Lógica de Programação e Estrutura de Dados**. 3. ed. São Paulo: Pearson, 2016.

PERKOVIC, L. **Introdução à Computação Usando Python - Um Foco no Desenvolvimento de Aplicações**. Rio de Janeiro: LTC, 2016.

FORBELLONE, A. L. V. et al. **Lógica de programação: a construção de algoritmos e estruturas de dados**. 3. ed. São Paulo: Pearson, 2005.

MENEZES, N. N. C. **Introdução à Programação Python:** algoritmos e lógica de programação para iniciantes. 3. ed. São Paulo: Novatec, 2019.

MATTHES, Eric. **Curso Intensivo de Python:** uma introdução prática baseada em projetos à programação. 1. ed. São Paulo: Novatec, 2015.

[1] Para fins de simplificação, vamos assumir que o número 0 também é par.