

Algoritmos

Lucas Araújo Campos Szuster

25 de dezembro de 2025

Sumário

1. Definições básicas

2. Algoritmos condicionais

- 2.1 O que são algoritmos condicionais
- 2.2 O que é uma condição?
- 2.3 Encadeamento de condições

3. Algoritmos com estruturas de repetição

- 3.1 Conceitos básicos
- 3.2 Repetição controlada por condição
- 3.3 Repetição controlada por contagem
- 3.4 Considerações importantes sobre estruturas de repetição
- 3.5 Repetições alinhadas

O que é um Algoritmo?

Um algoritmo é um conjunto de instruções organizadas de maneira lógica e finita, criado para resolver um problema ou executar uma tarefa específica de forma eficiente e consistente. É fundamental destacar que a ordem dos passos é extremamente importante: embora existam algoritmos que possam chegar ao mesmo resultado independentemente da sequência das instruções, eles são a exceção, e não a regra.

Algoritmos podem ser comparados a uma receita de cozinha: assim como uma receita especifica os ingredientes, a sequência de preparo e o tempo de cozimento, um algoritmo indica passo a passo o que deve ser feito, em que ordem e com quais dados*, garantindo que o resultado final seja previsível e correto sempre que seguido.

Importância dos algoritmos

Os algoritmos são fundamentais na matemática, computação e engenharia, permitindo resolver problemas de forma estruturada e eficiente.

Eles podem lidar com tarefas simples, como somar números, até processos complexos, como:

- rotas de transporte;
- processamento de grandes volumes de dados;
- inteligência artificial;
- simulações científicas.

E *praticamente* quaisquer outros problemas imagináveis!

Principais características de um algoritmo

Todo algoritmo possui certas características que são essenciais para o seu funcionamento correto. Estas são:

Principais características de um algoritmo

Todo algoritmo possui certas características que são essenciais para o seu funcionamento correto. Estas são:

1. **Finitude:** o processo deve necessariamente encerrar após um número finito de passos, ainda que esse término ocorra apenas em um futuro distante.

Principais características de um algoritmo

Todo algoritmo possui certas características que são essenciais para o seu funcionamento correto. Estas são:

1. **Finitude:** o processo deve necessariamente encerrar após um número finito de passos, ainda que esse término ocorra apenas em um futuro distante.
2. **Precisão:** cada passo deve ser descrito de forma clara, inequívoca e rigorosamente definida.

Principais características de um algoritmo

Todo algoritmo possui certas características que são essenciais para o seu funcionamento correto. Estas são:

1. **Finitude**: o processo deve necessariamente encerrar após um número finito de passos, ainda que esse término ocorra apenas em um futuro distante.
2. **Precisão**: cada passo deve ser descrito de forma clara, inequívoca e rigorosamente definida.
3. **Efetividade**: deve sempre resolver o que tem para solucionar, antecipando falhas.

Principais características de um algoritmo

Todo algoritmo possui certas características que são essenciais para o seu funcionamento correto. Estas são:

1. **Finitude**: o processo deve necessariamente encerrar após um número finito de passos, ainda que esse término ocorra apenas em um futuro distante.
2. **Precisão**: cada passo deve ser descrito de forma clara, inequívoca e rigorosamente definida.
3. **Efetividade**: deve sempre resolver o que tem para solucionar, antecipando falhas.

“Extra”

Algoritmos também são ordenados de uma maneira específica, isto é, alterar a ordem dos seus passos altera o algoritmo e possivelmente seu resultado.

Entradas e saídas

Todo algoritmo trabalha com **entradas** e **saídas**. As **entradas** correspondem aos dados ou informações que o algoritmo recebe para processar e analisar. Com base nessas entradas, o algoritmo realiza um **processamento** e produz as **saídas**, que representam os resultados gerados.

As **entradas** e **saídas** são fundamentais, pois a mesma lógica pode gerar resultados diferentes dependendo das informações fornecidas. Vale destacar que alguns algoritmos podem ter **entradas** nulas (nenhum dado fornecido) ou **saídas** nulas (nenhum resultado gerado), dependendo do contexto ou da tarefa que realizam.

Exemplo de algoritmo

Exemplo de algoritmo

Aqui segue o exemplo clássico de algoritmo, uma receita de bolo:

1. Misture os ingredientes secos;
2. Adicione ovos e leite;
3. Bata até obter uma massa homogênea;
4. Coloque em uma forma untada;
5. Leve ao forno por 40 minutos.

Observe que alterações nos ingredientes (as **entradas**) ou na ordem dos passos realizados (processamento) gera um resultado (**saídas**) diferentes.

Exemplo de algoritmo — análise

No algoritmo apresentado anteriormente, é possível identificar claramente os principais elementos que caracterizam um algoritmo:

- O algoritmo, como um todo, é **finito**, pois o processo termina quando o bolo fica pronto;
- Cada passo é **preciso** e bem definido, não deixando margem para ambiguidades;
- O algoritmo é **efetivo**, já que todas as instruções podem ser executadas na prática;
- Alguma alteração na ordem de realização dos passos altera o algoritmo;
- As **entradas** correspondem aos ingredientes utilizados;
- A **saída** é o resultado final do processo: o bolo pronto.

Mais exemplos de algoritmos

Aqui estão alguns exemplos de algoritmos presentes no nosso dia a dia. Em todos eles, há uma sequência de passos bem definidos que, quando seguidos corretamente, levam a um objetivo específico:

- Higiene do sono (estabelecer horário para dormir, desligar dispositivos eletrônicos, escovar os dentes, deitar);
- Preparar uma refeição (separar ingredientes, seguir uma receita passo a passo, cozinhar e servir);
- Fazer compras (elaborar uma lista, ir ao mercado, selecionar produtos, pagar e retornar);
- Lavar roupas (separar por cor, colocar na máquina, escolher o programa e iniciar a lavagem);
- Instruções de montagem, como as referentes à montagem de um móvel ou equipamento.

Algoritmos na computação

Na computação, praticamente todas as tarefas são realizadas por meio de algoritmos. Desde ações simples, como exibir uma mensagem na tela, até sistemas complexos, como motores de busca e redes sociais, tudo depende de sequências bem definidas de instruções.

O “pulo do gato” é perceber que muitos algoritmos não são isolados: eles são formados pela combinação de outros algoritmos menores, organizados de forma hierárquica.

Esse conceito será aprofundado mais adiante. Por enquanto, basta compreender algoritmo como uma sequência ordenada, precisa e finita de passos para resolver um problema.

Eficiência e complexidade

Dois algoritmos podem resolver o mesmo problema, mas com custos diferentes. Na computação, a eficiência de algoritmos é estudada pela **complexidade computacional**, que mede principalmente as seguintes dimensões:

- tempo de execução, e maneira com que diferentes entradas o afetam;
- utilização de memória (principalmente memória adicional);
- recursos necessários.

Algoritmos condicionais

Conceitos básicos

Algoritmos condicionais são aqueles capazes de tomar decisões durante a sua execução, alterando o fluxo de ações com base em condições previamente estabelecidas.

Em outras palavras, o algoritmo avalia uma situação por meio de uma **condição**, e, a partir desse resultado, decide quais ações devem ser executadas em cada caso.

Esse mecanismo permite que o algoritmo não siga sempre o mesmo caminho, tornando-o mais flexível e adequado para lidar com diferentes cenários ou entradas.

O que é uma condição?

Uma condição é uma expressão lógica que pode ser:

- Verdadeira (True);
- Falsa (False).

Observação importante:

Toda condição, para fins de análise e no contexto da computação, é *booleana*, ou seja, só pode assumir dois valores possíveis: verdadeiro ou falso. Não existe um valor intermediário ou ambíguo. Isso significa que, se você sabe que uma condição não é verdadeira, é suficiente para concluir que ela é falsa. Da mesma forma, se uma condição é falsa, podemos afirmar com certeza que não é verdadeira.

Exemplo de condições

A seguir, alguns exemplos de condições que podem ser utilizadas em algoritmos com estruturas condicionais:

- **Verificação de idade em uma loja de bebidas:** um algoritmo deve checar se o comprador é maior de idade. Para isso, verifica-se se a idade do comprador é igual ou superior a 18 anos, condição que sempre resultará em **verdadeiro** ou **falso**.
- **Admissão em um hospital:** o algoritmo deve admitir apenas pessoas que estejam doentes. Para isso, avalia-se se a pessoa apresenta sinais ou diagnóstico de doença, condição que será **verdadeira** ou **falsa**.
- **Acionamento de alarme de segurança:** o algoritmo verifica se uma porta ou janela foi aberta sem autorização. Para isso, avalia-se se a porta/janela está aberta enquanto o alarme está ativado, condição que será **verdadeira** ou **falsa**.

Estrutura básica de decisão

A seguir está a estrutura básica de como um programa realiza uma decisão:

```
se condição então
    faça algo
senão
    faça outra coisa
fim se
```

Observação importante

Primeiro, uma condição é avaliada; se ela for verdadeira, uma ação específica é executada. Caso a condição seja falsa (bloco **senão**), uma ação alternativa é realizada.

Exercício 1

Imagine o seguinte cenário: você trabalha em uma fábrica de brinquedos que produz dois tipos de produtos: carrinhos e bonecas. Seu trabalho é simples, mas essencial: uma grande esteira transporta os brinquedos já misturados, e sua função é separá-los em caixotes específicos, de acordo com o tipo de brinquedo.

Como poderíamos modelar essa tarefa usando uma condição simples? Ou seja, como o programa deve decidir, para cada brinquedo, em qual caixote ele deve ser colocado com base apenas em uma condição?

Soluções [parte 1]

Para resolver este problema, seria necessário verificar o tipo de brinquedo e adicioná-lo ao caixote correspondente. No entanto, a expressão “**checlar o tipo de brinquedo**” por si só não constitui uma condição válida, pois não retorna **verdadeiro** ou **falso**, mas sim categorias como “carrinho” ou “boneca”.

Para contornar isso, podemos utilizar uma comparação explícita: verificar se o tipo do brinquedo é igual a “carrinho” ou igual a “boneca”. Dessa forma, a condição sempre produzirá um resultado **verdadeiro** ou **falso**, permitindo que o algoritmo tome decisões corretas e diretas sobre em qual caixote o brinquedo deve ser colocado.

Soluções [parte 2]

Com isso, é possível perceber que existem diferentes maneiras de resolver este problema: uma abordagem compara o tipo de brinquedo com “boneca” e a outra compara com “carrinho”. Ambas levam ao mesmo resultado final, ou seja, o brinquedo é corretamente colocado no caixote correspondente.

Observação importante

Nenhuma das duas soluções é superior à outra. A complexidade de ambos os algoritmos é equivalente, e a escolha de qual abordagem utilizar pode depender apenas de uma “preferência pessoal” ou de critérios de legibilidade e organização do código. Em situações mais complexas, fatores como facilidade de manutenção ou clareza podem tornar uma abordagem mais vantajosa, mesmo que o desempenho seja idêntico.

Soluções [parte 3]

```
se (tipo do brinquedo) = carrinho então
    adicionar brinquedo ao caixote de carrinhos
senão
    adicionar brinquedo ao caixote de bonecas
fim se
```

```
se (tipo do brinquedo) = boneca então
    adicionar brinquedo ao caixote de bonecas
senão
    adicionar brinquedo ao caixote de carrinhos
fim se
```

Soluções [parte 4]

Há um aspecto relevante nas soluções apresentadas que merece destaque, pois foi deliberadamente simplificado com o objetivo de tornar o problema mais didático.

O tipo do brinquedo não foi *lido*, isto é, não foi obtido de nenhuma forma pelo algoritmo. Em uma abordagem mais adequada, os algoritmos que resolvem este problema deveriam incluir a leitura do tipo do brinquedo, uma vez que esse dado é essencial para a tomada de decisão com base na condição estabelecida.

Exercício 2

Imagine que você está em um quarto de hotel, que contém apenas uma cama e um ventilador, o qual está inicialmente desligado. Você se deita na cama com a intenção de descansar, mas a temperatura do quarto influencia sua decisão. Se o ambiente estiver muito quente, você decide ligar o ventilador para se refrescar; caso contrário, se a temperatura estiver agradável, você não faz nada, mantendo o ventilador desligado.

Escreva a condição que determina o que você faz com base na temperatura do ambiente.

Solução [parte 1]

Aqui está uma solução para este problema:

```
se (temperatura muito quente) então
    ligar o ventilador
senão
    não fazer nada
fim se
```

Entretanto, há algo estranho nesta solução. Parece muito desnecessário especificar que nada deve ser feito caso a temperatura esteja muito quente. Sera que não há uma maneira de melhorar isto?

Solução [parte 2]

Aqui está uma solução melhorada:

```
se (temperatura muito quente) então
    ligar o ventilador
fim se
```

Observe que, quando não é necessário realizar nenhuma ação no caso contrário, não é preciso adicionar uma cláusula "**senão**". Dessa forma, a ação especificada na condição será executada apenas se a condição for verdadeira. Se a condição for falsa, nenhuma ação será tomada, mantendo o comportamento esperado de forma simples e direta.

Solução [parte 3]

Há dois aspectos relevantes na solução apresentada que merecem destaque, pois foram deliberadamente simplificados com o objetivo de tornar o problema mais didático:

- Foi adotada uma condição propositalmente genérica, expressa como (*temperatura muito quente*), a fim de facilitar a compreensão da lógica do algoritmo. Em uma situação real, entretanto, essa condição deveria ser definida de forma objetiva, por meio de uma comparação direta, como *temperatura > (valor específico)*;
- Da mesma forma, a temperatura não foi efetivamente *lida* ou obtida como um valor numérico. Em um cenário prático, esse dado deveria ser medido ou informado por algum meio apropriado antes de ser utilizado nas decisões do algoritmo.

Encadeamento de condições

Em muitas situações, uma única condição não é suficiente para realizar um processamento completo e satisfatório. Nesses casos, é necessário utilizar condições encadeadas, ou seja, múltiplas condições avaliadas em sequência, de forma que cada uma determine um resultado específico dependendo do contexto. Esse tipo de estrutura permite que o programa tome decisões mais complexas e trate diferentes cenários de maneira organizada.

Por exemplo, considere a classificação de notas de alunos: dependendo do valor da nota, o aluno pode ser considerado Excelente, Bom, Regular ou Reprovado. Cada faixa de nota corresponde a uma condição diferente, e elas são avaliadas em sequência até que a condição correta seja satisfeita, determinando o resultado apropriado.

Exemplo de encadeamento de condições

Leia nota

se nota $\geq 90\%$ **então**

Escreva conceito *A*

senão **se** nota $\geq 80\%$ **então**

Escreva conceito *B*

senão **se** nota $\geq 70\%$ **então**

Escreva conceito *C*

senão **se** nota $\geq 60\%$ **então**

Escreva conceito *D*

senão

Escreva conceito *F*

fim se

Exercício 3

Crie um pequeno algoritmo condicional capaz de classificar uma pessoa com base em seu peso. Aqui estão as classificações:

- Peso menor que 50kg, pessoa magra;
- Peso maior ou igual a 50kg e menor que 100kg, pessoa normal;
- Peso maior ou igual à 100kg, pessoa gorda.

Solução [parte 1]

Para solucionar este problema, precisamos primeiro “ler” o dado *peso*, ou seja, recebê-lo como entrada do nosso algoritmo. Em seguida, realizamos comparações utilizando esse dado, com o objetivo de categorizar uma pessoa com base no seu peso.

Para isso, podemos encadear diferentes condições de forma lógica, de modo que cada comparação vá eliminando possibilidades ou categorias, até que seja possível determinar a categoria final correta. Esse processo permite que o algoritmo tome decisões passo a passo, avaliando cada condição de maneira ordenada e precisa, garantindo que o resultado final seja consistente e confiável.

Solução [parte 2]

Aqui segue uma solução que segue exatamente a ordem de checagens proposta pelo exercício:

```
Leia peso  
se peso < 50kg então  
    Escreva pessoa magra  
senão se peso < 100kg então  
        Escreva pessoa normal  
senão  
    Escreva pessoa gorda  
fim se
```

Solução [parte 3]

Entretanto, assim como um dos problemas anteriores existe outra solução possível, alterando a ordem de checagens, e, neste caso, os operadores utilizados para cada checagem. Aqui está uma solução com outra ordem de checagens:

```
Leia peso  
se peso  $\geq$  100kg então  
    Escreva pessoa gorda  
senão se peso  $\geq$  50kg então  
        Escreva pessoa normal  
senão  
    Escreva pessoa magra  
fim se
```

Algoritmos com estruturas de repetição

O que é uma Estrutura de Repetição?

Uma **estrutura de repetição** é um mecanismo que permite executar um conjunto de instruções diversas vezes de forma **controlada**. Em outras palavras, ela automatiza tarefas repetitivas sem a necessidade de escrever o mesmo código várias vezes.

Principais características:

- Cada repetição é chamada de **iteração**.
- A repetição continua até que uma **condição de parada** seja satisfeita.
- Pode depender de uma **condição lógica** ou de um **número pré-determinado de repetições**.

Para que servem?

Dentre os diversos usos para estruturas de repetição em algoritmos, destacam-se:

- Automatizar tarefas repetitivas;
- Controlar processos até um objetivo;
- Reduzir redundância;
- Aumentar eficiência e escalabilidade.

Exemplo motivacional [parte 1]

Vamos retomar o exercício do slide 17, no qual utilizamos uma estrutura condicional para modelar o processo de decisão que determina se um determinado brinquedo, produzido numa fábrica, deve ser adicionado ao caixote de bonecas ou ao caixote de carrinhos.

O resultado final permitia tomar essa decisão apenas uma vez; no entanto, isso não seria muito útil num cenário real. Numa fábrica de verdade, não é produzida apenas uma unidade de um dos dois tipos de brinquedos, mas sim centenas ou milhares. Ou seja, essa verificação precisa ser realizada repetidas vezes.

Suponha agora que você é um trabalhador e que, no seu contrato, está estipulado que deve separar exatamente 150 brinquedos por dia. Para cumprir essa tarefa, é necessário repetir o algoritmo que desenvolvemos inicialmente 150 vezes.

Exemplo motivacional [parte 2]

Com o ferramentário que possuímos até o momento, para repetir o algoritmo 150 vezes, a única alternativa seria copiá-lo e colá-lo manualmente essa quantidade de vezes. No entanto, essa abordagem é extremamente ineficiente e apresenta sérias limitações:

- **Pouco adaptável:** suponha que agora seja necessário processar 200 brinquedos diariamente. Para ajustar o pseudocódigo, seria necessário copiar e colar o código mais 50 vezes, tornando a manutenção trabalhosa;
- **Consume espaço desnecessário:** copiar e colar centenas de linhas gera um pseudocódigo muito longo, tornando-o visualmente cansativo e difícil de ler.

A solução adequada é utilizar uma estrutura de repetição, que permite controlar de forma simples e eficiente o número de vezes que o algoritmo deve ser executado.

Exemplo motivacional [parte 3]

A solução ideal, portanto, seria algo deste tipo:

faça 150 vezes:

se (tipo do brinquedo) = carrinho **então**

 adicionar brinquedo ao caixote de carrinhos

senão

 adicionar brinquedo ao caixote de bonecas

fim se

Entretanto, esta **não** é a forma adequada de representar estruturas de repetição, já que precisamos de um controle mais preciso sobre a execução das instruções.

Funcionamento básico

O funcionamento básico de uma estrutura de repetição pode ser resumido em duas etapas principais, que permitem que um conjunto de instruções seja executado de forma controlada e eficiente.

- **Verificação da condição:** Antes de cada iteração, a condição é avaliada para determinar se o bloco de instruções deve ser executado. Se a condição for verdadeira, o ciclo continua e passa para a execução das instruções; se for falsa, a repetição é encerrada, garantindo que o programa siga adiante sem realizar iterações desnecessárias.
- **Execução do bloco de instruções:** Caso a condição seja verdadeira, o bloco de instruções é executado, realizando todas as tarefas definidas dentro do ciclo. Após a execução, o processo retorna para a verificação da condição, repetindo o ciclo enquanto a condição permanecer verdadeira.

Os dois tipos de estruturas de repetição

Com base no processo previamente descrito, é possível identificar dois tipos principais de estruturas de repetição:

- **Repetição controlada por condição:** estrutura em que o critério de parada não é previamente conhecido, sendo determinado apenas durante a execução do programa.
- **Repetição controlada por contagem:** estrutura em que o critério de parada é definido por um número fixo de repetições*, tornando-o mais simples e previsível do que uma condição genérica.

Repetição controlada por condição

Conhecida como *while* (enquanto), a estrutura de repetição controlada por condição caracteriza-se pelo fato de que o número de repetições a serem realizadas não é previamente conhecido, pois depende da avaliação de uma condição lógica cujo resultado pode variar de forma imprevisível durante a execução do programa.

Forma geral:

```
enquanto condição faça
    algo
fim enquanto
```

Exemplo de algoritmo com *while*

Imagine que você é um jogador de futebol treinando cobranças de pênalti. Nesse treino, você deve continuar batendo pênaltis até conseguir marcar um gol contra um excelente goleiro. Essa situação exemplifica um *loop* do tipo *while*:

```
enquanto não converter o pênalti faça
    outra tentativa de conversão
fim enquanto
```

Esta situação é extremamente típica para utilização de uma estrutura de repetição controlada por condição, uma vez que a quantidade de vezes que o pênalti deve ser treinado é altamente variável, impossível de determinar previamente e depende de uma condição específica.

Exercício 4

A situação descrita anteriormente não representa de forma fiel a realidade, pois um treino de cobranças de pênalti normalmente não é encerrado após apenas um acerto. Em um contexto mais realista, o jogador continuaria treinando até atingir um determinado objetivo. Dessa forma, ajuste o cenário previamente apresentado e elabore um algoritmo que represente o comportamento de um jogador que interrompe seu treinamento somente após realizar com sucesso 10 cobranças de pênalti.

Solução [parte 1]

Para resolver este problema, deve-se utilizar um contador responsável por registrar a quantidade de cobranças convertidas com sucesso. Inicialmente, o contador deve ser iniciado com valor zero. A cada cobrança convertida corretamente, esse contador deve ser incrementado em uma unidade. Esse processo deve ser repetido continuamente até que o contador atinja o total de 10 cobranças convertidas com êxito, momento em que a execução do algoritmo pode ser encerrada.

solução [parte 2]

cobranças corretas \leftarrow 0

enquanto cobranças corretas $<$ 10 **faça**

 outra tentativa de conversão

 se cobrança convertida **então**

 cobranças corretas \leftarrow cobranças corretas + 1

fim se

fim enquanto

Repetição controlada por contagem [parte 1]

Conhecida como *for* (para), a estrutura de repetição controlada por contagem é especialmente útil quando o número de repetições é conhecido ou, ao menos, pode ser determinado de forma previsível antes do início da execução do laço.

Nesse tipo de estrutura, a repetição é guiada por uma variável de controle, que normalmente assume valores dentro de um intervalo bem definido. Por esse motivo, os *loops* do tipo *for* são amplamente utilizados para percorrer conjuntos de dados, como listas.

Por exemplo, ao acessar os elementos de uma lista, mesmo que a quantidade de itens varie a cada execução do programa, a lógica da repetição permanece a mesma: o laço será executado exatamente tantas vezes quanto o número de elementos contidos na lista.

Repetição controlada por contagem [parte 2]

A estrutura geral de um *loop* do tipo *for* é:

```
para i ← (valor inicial) até (valor final) faça
    alguma instrução (ou instruções)
fim para
```

Nesta estrutura, estamos comparando o número *i* com um valor final, até que que eles sejam iguais, e repetindo a estrutura enquanto eles não forem.

Observação importante

Nesta forma mais simples, é assumido que *i* está sendo incrementado com o valor 1 a cada iteração (passo 1), mas isto pode variar.

Exemplo de algoritmo com *for* [parte 1]

Imagine uma lista (L) com n elementos, como a lista abaixo:

$$L = [X_1, X_2, X_3, X_4, X_5, X_6, X_7, \dots, X_n]$$

Imagine que possamos denotar um elemento genérico de L por X_i .

Podemos acessar todos os elementos da lista com o seguinte algoritmo:

```
para  $i \leftarrow 1$  até  $n$  faça
    alguma ação com  $X_i$ 
fim para
```

Observação importante

Neste exemplo, a lista está sendo indexada com 1, ou seja, o primeiro elemento é X_1 . Entretanto, na realidade, listas em código são geralmente indexadas a partir do 0.

Exemplo de algoritmo com *for* [parte 2]

Uma outra implementação de um algoritmo que realiza exatamente o mesmo procedimento (de acessar todos os elementos em uma lista e fazer algo com cada um deles) é o seguinte:

```
para  $X_i$  em  $L$  faça
    alguma ação com  $X_i$ ;
fim para
```

Esta estrutura, chamada de *foreach*, é mais simples, porém também mais limitada, pois funciona apenas para acessar elementos de listas de uma forma específica — embora extremamente comum.

Exercício 5

Imagine que você deseja criar um algoritmo capaz de calcular a soma de todos os números inteiros de 1 até um número n qualquer. Escreva o pseudocódigo que represente este procedimento.

Solução [parte 1]

Para solucionar este problema, precisamos entender o que ele significa.
Na prática, estamos descrevendo a seguinte equação:

$$\sum_{i=1}^n i = 1 + 2 + 3 + 4 + \cdots + n$$

Assim, para resolver este problema, é necessário armazenar o valor da soma a cada iteração e incrementá-lo de acordo.

Solução [parte 2]

Portanto, para resolver o problema, é necessário inicialmente realizar a leitura do valor de n . Esse passo pode ser incluído na solução, uma vez que n é um valor numérico e a leitura de números constitui um conceito simples (diferente de algumas outras leituras anteriores, como a leitura do tipo de um brinquedo).

Após obter o valor de n , vamos definir um número R_p , iniciando-o com valor zero, e, de forma sequencial, cada número inteiro da série é somado ao resultado acumulado até que se alcance o valor n . A cada iteração, o valor parcial da soma é armazenado em R_p .

Dessa forma, ao final da primeira iteração, tem-se $R_p = 1$; ao final da segunda, $R_p = 3$; ao final da terceira, $R_p = 6$; e assim sucessivamente, até a conclusão do processo.

Solução [parte 3]

Por fim, temos este resultado final:

Leia n

$R_p \leftarrow 0$

para $i \leftarrow 1$ até n **faça**

$R_p \leftarrow R_p + i$

fim para

Escreva R_p

Exercício 6

Sabendo que esta é a forma geral para uma estrutura do tipo *for* com passo personalizado:

```
para i  $\leftarrow$  (valor inicial) até (valor final) passo (passo) faça
    alguma instrução (ou instruções)
fim para
```

Escreva um algoritmo que leia um número inteiro *n* qualquer e realize a contagem regressiva até zero.

Solução [parte 1]

Para resolver este exercício, deve-se inicialmente realizar a leitura do número n , que servirá como valor inicial da contagem.

Em seguida, o contador é inicializado com esse valor e utiliza-se uma estrutura de repetição com passo negativo, de modo que, a cada iteração, o valor do contador seja decrementado em uma unidade.

Esse processo de subtração deve ser repetido continuamente até que o contador atinja o valor zero, momento em que a contagem regressiva é finalizada e o algoritmo é encerrado.

Solução [parte 2]

Aqui está a resposta:

```
Leia n  
para i  $\leftarrow$  n até 0 passo -1 faça  
    Escreva i  
fim para
```

Observação importante sobre estruturas de repetição

- Tudo o que pode ser realizado utilizando uma estrutura do tipo *for* também pode ser implementado com uma estrutura do tipo *while*. No entanto, saber escolher a estrutura mais adequada para cada situação é uma habilidade essencial, pois o uso correto torna o código mais simples, legível e evita complexidades desnecessárias.
- *i* é um nome padrão para o contador de uma estrutura de repetição do tipo *for*, mas não é obrigatório que o contador tenha este nome.

Como escolher a melhor estrutura?

Dicas para escolher a estrutura mais apropriada:

- Use *for* quando o número de repetições é conhecido ou previsível, como ao percorrer uma lista ou executar uma ação um número fixo de vezes.
- Use *while* quando a repetição depende de uma condição que pode variar durante a execução e cujo número de iterações não é conhecido antecipadamente.
- Pergunte-se: “Posso determinar de antemão quantas vezes este bloco deve ser executado?” Se sim, o *for* geralmente é mais claro; se não, o *while* é geralmente mais apropriado.

Para maior clareza, evite tentar forçar uma *while* onde um *for* é suficiente, e vice-versa. O objetivo é tornar o código intuitivo para quem lê.

Repetições alinhadas

Em muitas aplicações, é necessário que estruturas de repetição sejam combinadas de forma aninhada — ou seja, uma dentro da outra — para executar tarefas mais complexas. Esse conceito, conhecido como *loops* aninhados, permite percorrer múltiplas dimensões de dados ou realizar operações repetitivas de forma estruturada.

Na programação, o uso de *loops* aninhados é especialmente comum ao trabalhar com matrizes ou tabelas de valores, onde precisamos iterar sobre linhas e colunas simultaneamente. Como matrizes serão estudadas posteriormente, este assunto será abordado mais a frente no curso.

Fim