Lógica e Técnicas de Programação

Prof. Simone Cristina Aléssio

UNIASSELVI
2017

Elaboração:
Prof. Simone Cristina Aléssio

Revisão, Diagramação e Produção:
Centro Universitário Leonardo da Vinci – UNIASSELVI

Ficha catalográfica elaborada na fonte pela Biblioteca Dante Alighieri
UNIASSELVI – Indaial.

001.642
A372l Aléssio; Simone Cristina
Lógica e técnicas de Programação / Simone Cristina Aléssio: UNIASSELVI, 2017.
175 p. : il.

ISBN 978-85-515-0051-4

1. Programação de Computadores.
I. Centro Universitário Leonardo da Vinci.

Impresso por:

Apresentação

Olá, caro acadêmico! Parabéns! Você está iniciando mais uma etapa de conhecimento. A equipe instrucional elaborou todo o material necessário ao suporte para o seu aprendizado. Neste formato, a disciplina Lógica e Técnicas de Programação foi elaborada pensando numa leitura rápida e dinâmica, abordando o centro de cada conteúdo, explanado em aulas bem objetivas. Como já é do seu conhecimento, estudar a distância é uma tarefa que envolve disciplina na resolução dos exercícios, contando com todo amparo da equipe que irá apoiá-lo no processo de ensino-aprendizagem. Para que isso ocorra de forma efetiva, faz-se necessário separar um tempo para estudar o material e fazer as leituras complementares indicadas no caderno de estudos. Esperamos que você utilize todos os recursos do ambiente, disponíveis para dar andamento aos estudos e avançar pelos módulos.

Um grande abraço!

Prof. Simone Cristina Aléssio

Você já me conhece das outras disciplinas? Não? É calouro? Enfim, tanto para você que está chegando agora à UNIASSELVI quanto para você que já é veterano, há novidades em nosso material.

Na Educação a Distância, o livro impresso, entregue a todos os acadêmicos desde 2005, é o material base da disciplina. A partir de 2017, nossos livros estão de visual novo, com um formato mais prático, que cabe na bolsa e facilita a leitura.

O conteúdo continua na íntegra, mas a estrutura interna foi aperfeiçoada com nova diagramação no texto, aproveitando ao máximo o espaço da página, o que também contribui para diminuir a extração de árvores para produção de folhas de papel, por exemplo.

Assim, a UNIASSELVI, preocupando-se com o impacto de nossas ações sobre o ambiente, apresenta também este livro no formato digital. Assim, você, acadêmico, tem a possibilidade de estudá-lo com versatilidade nas telas do celular, *tablet* ou computador.

Eu mesmo, UNI, ganhei um novo *layout*, você me verá frequentemente e surgirei para apresentar dicas de vídeos e outras fontes de conhecimento que complementam o assunto em questão.

Todos esses ajustes foram pensados a partir de relatos que recebemos nas pesquisas institucionais sobre os materiais impressos, para que você, nossa maior prioridade, possa continuar seus estudos com um material de qualidade.

Aproveito o momento para convidá-lo para um bate-papo sobre o Exame Nacional de Desempenho de Estudantes – ENADE.

Bons estudos!

Olá acadêmico! Para melhorar a qualidade dos materiais ofertados a você e dinamizar ainda mais os seus estudos, a Uniasselvi disponibiliza materiais que possuem o código *QR Code*, que é um código que permite que você acesse um conteúdo interativo relacionado ao tema que você está estudando. Para utilizar essa ferramenta, acesse as lojas de aplicativos e baixe um leitor de *QR Code*. Depois, é só aproveitar mais essa facilidade para aprimorar seus estudos!

BATE PAPO SOBRE O ENADE!

Olá, acadêmico!

Você já ouviu falar sobre o **ENADE**?

Se ainda não ouviu falar nada sobre o ENADE, agora você receberá algumas informações sobre o tema.

Ouviu falar? Ótimo, este informativo reforçará o que você já sabe e poderá lhe trazer novidades.

Vamos lá!

Qual é o significado da expressão ENADE?

EXAME NACIONAL DE DESEMPENHO DOS ESTUDANTES

Em algum momento de sua vida acadêmica você precisará fazer a prova ENADE.

Que prova é essa?

É **obrigatória**, organizada pelo INEP – Instituto Nacional de Estudos e Pesquisas Educacionais Anísio Teixeira.

Quem determina que esta prova é obrigatória... O **MEC – Ministério da Educação**.

O objetivo do MEC com esta prova é o de avaliar seu desempenho acadêmico assim como a qualidade do seu curso.

Fique atento! Quem não participa da prova fica impedido de se formar e não pode retirar o diploma de conclusão do curso até regularizar sua situação junto ao MEC.

Não se preocupe porque a partir de hoje nós estaremos auxiliando você nesta caminhada.

Você receberá outros informativos como este, complementando as orientações e esclarecendo suas dúvidas.

Você tem uma trilha de aprendizagem do ENADE, receberá e-mails, SMS, seu tutor e os profissionais do polo também estarão orientados.

Participará de webconferências entre outras tantas atividades para que esteja preparado para #mandar bem na prova ENADE.

Nós aqui no NEAD e também a equipe no polo estamos com você para vencermos este desafio.

Conte sempre com a gente, para juntos mandarmos bem no ENADE!

UNIASSELVI

SUMÁRIO

UNIDADE 1 - LÓGICA E ALGORITMOS: CONCEITOS BÁSICOS 1

TÓPICO 1 - LÓGICA E ALGORITMOS 3
1 INTRODUÇÃO 3
2 LÓGICA 4
2.1 TIPOS DE LÓGICA 5
2.1.1 Lógica aristotélica 5
2.1.2 Lógica de programação 5
2.1.3 Lógica de argumentação 5
2.1.4 Lógica matemática 6
2.1.5 Lógica proposicional 6
2.2 LÓGICA PROPOSICIONAL 7
2.3 APLICABILIDADE DA LÓGICA NO DESENVOLVIMENTO DE PROGRAMAS 8
3 CONCEITO DE ALGORITMOS 8
4 CARACTERÍSTICAS DE UM ALGORITMO 11
5 FASES DE UM ALGORITMO 11
6 MÉTODO PARA A CONSTRUÇÃO DE ALGORITMOS 12
7 REGRAS BÁSICAS PARA A CONSTRUÇÃO DOS ALGORITMOS 12
8 FORMAS DE REPRESENTAÇÃO DE ALGORITMOS 12
8.1 DESCRIÇÃO NARRATIVA 13
8.2 FLUXOGRAMA CONVENCIONAL 14
8.3 DIAGRAMA DE CHAPIN 17
8.4 PSEUDOCÓDIGO 17
8.4.1 Representação de um algoritmo na forma de pseudocódigo 18
9 TIPOS DE DADOS 21
RESUMO DO TÓPICO 1 24
AUTOATIVIDADE 26

TÓPICO 2 - VARIÁVEIS, CONSTANTES, OPERADORES E EXPRESSÕES 27
1 INTRODUÇÃO 27
2 VARIÁVEIS, CONSTANTES 27
3 CONSTANTES 29
4 EXPRESSÕES E OPERADORES 30
4.1 OPERADORES 30
4.1.1 Operadores de atribuição 30
4.1.2 Operadores aritméticos 31
4.1.3 Operadores relacionais 31
4.1.4 Operadores lógicos 32
4.1.5 Operadores literais 33
5 EXPRESSÕES 34
5.1 EXPRESSÕES ARITMÉTICAS 34
5.1.1 Expressões lógicas 34
5.1.2 Expressões literais 34
5.1.3 Avaliação de expressões 34

6 INSTRUÇÕES PRIMITIVAS 36
6.1 COMANDOS DE ATRIBUIÇÃO 36
6.2 COMANDOS DE SAÍDA DE DADOS 37
6.3 COMANDOS DE ENTRADA DE DADOS 37
RESUMO DO TÓPICO 2 39
AUTOATIVIDADE 41

TÓPICO 3 - ESTRUTURAS DE CONTROLE DO FLUXO DE EXECUÇÃO 43
1 INTRODUÇÃO 43
2 ESTRUTURA CONDICIONAL 43
2.1 ESTRUTURA DE CONDIÇÃO SIMPLES: SE-ENTÃO 43
2.2 ESTRUTURA DE CONDIÇÃO COMPOSTA: SE-ENTÃO-SENÃO 44
2.2.1 Expressões lógicas compostas em estruturas de condição 46
2.2.2 Estruturas de condições encadeadas 47
2.3 ESTRUTURA DE CONDIÇÃO CASO SEJA 47
3 ESTRUTURAS DE REPETIÇÃO 49
3.1 TESTE NO INÍCIO: ENQUANTO-FAÇA 49
3.2 TESTE NO FIM: FAÇA-ENQUANTO 51
3.3 REPETIÇÃO COM CONTROLE: FAÇA-PARA 52
4 CONTADORES E ACUMULADORES 53
4.1 CONTADORES 53
4.2 ACUMULADORES 54
LEITURA COMPLEMENTAR 55
RESUMO DE TÓPICO 3 60
AUTOATIVIDADE 62

UNIDADE 2 - TABELA VERDADE, MODULARIZAÇÃO, CONSISTÊNCIA DE DADOS, VETORES E MATRIZES 63

TÓPICO 1 - TABELA VERDADE 65
1 INTRODUÇÃO 65
2 LÓGICA PROPOSICIONAL 65
2.1 CONCEITOS ASSOCIADOS 65
2.2 TIPOS DE CONECTIVOS LÓGICOS 67
2.2.1 Conjunção – conectivo e 67
2.2.2 Disjunção – conectivo ou 69
2.2.3 Disjunção exclusiva – conectivo ou... ou... 70
2.2.4 Se então – conectivo condicional 71
2.2.5 Se e somente se – conectivo bicondicional 72
2.2.6 Partícula não – negação 73
2.2.7 A negação de uma proposição composta 73
2.2.8 Negação de uma proposição disjuntiva: ~(p ou q) 76
2.2.9 Negação de uma proposição condicional: ~(p → q) 78
3 OUTROS ASPECTOS IMPORTANTES DA TABELA VERDADE 79
3.1 TAUTOLOGIA 82
3.2 CONTRADIÇÃO 83
3.3 CONTINGÊNCIA 83
RESUMO DO TÓPICO 1 84
AUTOATIVIDADE 86

TÓPICO 2 - CONSISTÊNCIA DOS DADOS DE ENTRADA 87
1 INTRODUÇÃO 87

2 CONSISTÊNCIA E MODULARIZAÇÃO 87
3 MODULARIZAÇÃO 87
3.1 COMPONENTES DE UM MÓDULO 89
3.2 EXEMPLO DE ALGORITMO COM MÓDULOS 90
4 FUNÇÕES E PROCEDIMENTOS EM PORTUGOL 91
4.1 PASSAGEM DE PARÂMETROS 92
4.2 VARIÁVEIS GLOBAIS 96
4.3 VARIÁVEIS LOCAIS 96
5 FUNÇÕES 96
6 PROCEDIMENTOS 98
RESUMO DO TÓPICO 2 100
AUTOATIVIDADE 102

TÓPICO 3 - VETORES E MATRIZES 103
1 INTRODUÇÃO 103
2 CONCEITOS DE VETORES E MATRIZES 103
LEITURA COMPLEMENTAR 110
RESUMO DO TÓPICO 3 113
AUTOATIVIDADE 114

UNIDADE 3 - ARQUIVOS, REGISTROS, COMPLEXIDADE DE ALGORITMOS E PRÁTICA DE CONSTRUÇÃO 115

TÓPICO 1 - ARQUIVOS E REGISTROS 117
1 INTRODUÇÃO 117
2 REGISTROS 117
3 ARQUIVOS 120
RESUMO DO TÓPICO 1 126
AUTOATIVIDADE 127

TÓPICO 2 - COMPLEXIDADE DE ALGORITMOS 129
1 INTRODUÇÃO 129
2 ANÁLISE DA COMPLEXIDADE DE ALGORITMOS 129
2.1 NOTAÇÃO O – ANÁLISE DE ALGORITMOS 131
2.2 ANÁLISE DE COMPLEXIDADE DA BUSCA LINEAR 134
2.3 PIOR CASO 135
2.4 CASO MÉDIO 135
2.5 MELHOR CASO 136
RESUMO DO TÓPICO 2 138
AUTOATIVIDADE 139

TÓPICO 3 - PRÁTICA DE ALGORITMOS 141
1 INTRODUÇÃO 141
2 CONSTRUÇÃO DE ALGORITMOS 141
2.1 CONTROLE CONDICIONAL 141
2.2 CASE 152
2.3 PARA FACA 154
2.4 ENQUANTO FACA 159
2.5 CONTADORES E ACUMULADORES 160
LEITURA COMPLEMENTAR 168
RESUMO DO TÓPICO 3 172
AUTOATIVIDADE 173
REFERÊNCIAS 175

UNIDADE 1

LÓGICA E ALGORITMOS: CONCEITOS BÁSICOS

OBJETIVOS DE APRENDIZAGEM

A partir desta unidade, você será capaz de:

- perceber a importância fundamental que a lógica ocupa no seu dia a dia;
- conhecer os princípios da lógica e como representá-la na forma de algoritmos;
- conhecer as características e aplicabilidades dos algoritmos;
- conhecer as principais formas de escrita de algoritmos, seus tipos de dados, variáveis, constantes, operadores lógicos e matemáticos e suas estruturas de controle.

PLANO DE ESTUDOS

Esta unidade está dividida em três tópicos. Ao final de cada um deles você encontrará atividades que auxiliarão no seu aprendizado.

TÓPICO 1 – LÓGICA E ALGORITMOS

TÓPICO 2 – VARIÁVEIS, CONSTANTES, OPERADORES E EXPRESSÕES

TÓPICO 3 – ESTRUTURAS DE CONTROLE DO FLUXO DE EXECUÇÃO

TÓPICO 1

LÓGICA E ALGORITMOS

1 INTRODUÇÃO

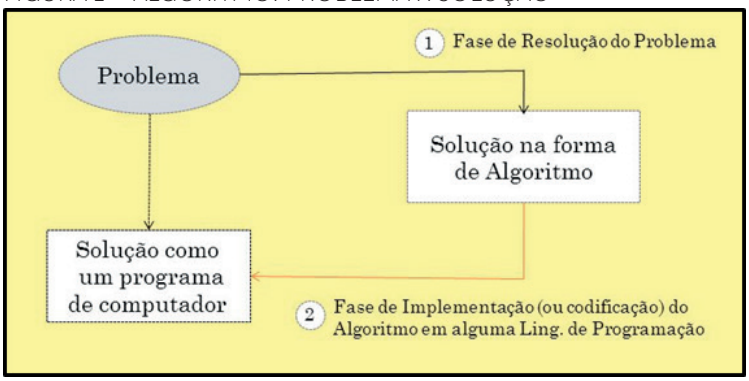
Historicamente, o algoritmo surgiu cerca de 300 a.C., com o algoritmo de Euclides. Este é o algoritmo mais antigo já estudado, que é utilizado até hoje. Auxilia na busca pelo máximo divisor comum entre dois números inteiros diferentes de zero, amplamente difundido na matemática até hoje.

Todavia, o conceito só foi documentado e reconhecido em 1936, através do estudioso Alan Turing, um dos fundadores da área computacional, que criou a máquina de Turing para capturar o significado de um algoritmo.

Hoje, podemos entender que o algoritmo não é a solução de um problema, mas o meio para se chegar à forma mais adequada para a solução. No decorrer das décadas, várias foram as maneiras propostas para se representar os algoritmos através de formas gráficas, como os fluxogramas, e através das próprias linguagens de programação.

Algoritmo exige prática, não é possível estudar ou copiar algoritmos. Somente aprendemos algoritmos no processo de construí-los e testá-los. Algoritmo não é um termo exclusivo e usado somente no setor computacional para codificar programas de computador. Pode ser usado para descrever os passos que devem ser seguidos na execução de processos, tarefas ou solução de qualquer tipo de problema (MANZANO, 2000).

FIGURA 1 - ALGORITMO: PROBLEMA X SOLUÇÃO



FONTE: Disponível em: <http://images.slideplayer.com.br/4/1540468/slides/slide_2.jpg>. Acesso em: 1º out. 2016.

Os problemas que são resolvidos tecnologicamente, através de programas de computador, somente terão suas soluções implementadas e executadas através dos algoritmos.

2 LÓGICA

Para desenvolver adequadamente um algoritmo, é necessário usar a lógica, que consiste em organizar o pensamento para resolver problemas usando a mesma sequência adotada pelo computador, ou seja, usando a mesma lógica.

Do que trata a lógica? Se quisermos buscar uma resposta simples à pergunta tão complexa, devemos dizer que a lógica, ao longo de toda sua história, justificou-se como um estudo sobre os argumentos. Em especial, a lógica justifica-se como um estudo sobre os critérios pelos quais se podem distinguir os "bons" dos "maus" argumentos (FORBELLONE, 1993).

FIGURA 2 - LÓGICA

FONTE: Disponível em: <http://www.fatosdesconhecidos.com.br/wp-content/uploads/2015/12/Racioc%C3%ADnio-l%C3%B3gico-para-conc.jpg>. Acesso em: 2 out. 2016.

A lógica pode ser entendida como sendo o estudo das leis do raciocínio e da forma como ela será aplicada no intuito de demonstrar a verdade (VENÂNCIO, 1997).

2.1 TIPOS DE LÓGICA

São vários os vários os tipos ou classificações para a lógica. A seguir estudaremos os principais.

2.1.1 Lógica aristotélica

De acordo com Aristóteles, a lógica tem como objeto de estudo o **pensamento**, assim como as leis e regras que o controlam, para que esse pensamento seja correto. Para o filósofo grego, os elementos constituintes da lógica são o **conceito**, **juízo** e **raciocínio**. As leis da lógica correspondem às ligações e relações que existem entre esses elementos.

Alguns sucessores de Aristóteles foram responsáveis pelos fundamentos da lógica medieval, que perdurou até o século XIII. Pensadores medievais como Galeno, Porfírio e Alexandre de Afrodísia classificavam a lógica como a ciência de julgar corretamente, que possibilita alcançar raciocínios corretos e formalmente válidos.

2.1.2 Lógica de programação

A lógica de programação é a linguagem usada para criar um programa de computador. A lógica de programação é essencial para desenvolver programas e sistemas informáticos, pois ela define o encadeamento lógico para esse desenvolvimento. Os passos para esse desenvolvimento são conhecidos como algoritmo, que consiste em uma sequência lógica de instruções para que a função seja executada.

2.1.3 Lógica de argumentação

A lógica de argumentação permite verificar a validade ou se um enunciado é verdadeiro ou não. Não é feito com conceitos relativos nem subjetivos. São

proposições tangíveis cuja validade pode ser verificada. Neste caso, a lógica tem como objetivo avaliar a forma das proposições e não o conteúdo. Os silogismos (compostos por duas premissas e uma conclusão), são um exemplo de lógica de argumentação, por exemplo: *O Fubá é um cachorro. Todos os cachorros são mamíferos. Logo, o fubá é um mamífero.*

2.1.4 Lógica matemática

A lógica matemática (ou lógica formal) estuda a lógica segundo a sua estrutura ou forma. A lógica matemática consiste em um **sistema dedutivo** de enunciados que tem como objetivo criar um grupo de leis e regras para determinar a validade dos raciocínios. Assim, um raciocínio é considerado válido se for possível alcançar uma conclusão verdadeira a partir de premissas verdadeiras. A lógica matemática também é usada para edificar raciocínios válidos mediante outros raciocínios. Os raciocínios podem ser **dedutivos** (a conclusão é obtida obrigatoriamente a partir da verdade das premissas) e **indutivos** (probabilísticos).

2.1.5 Lógica proposicional

A lógica proposicional é uma área da lógica que examina os raciocínios de acordo com as relações entre orações (proposições), as unidades mínimas do discurso, que podem ser verdadeiras ou falsas.

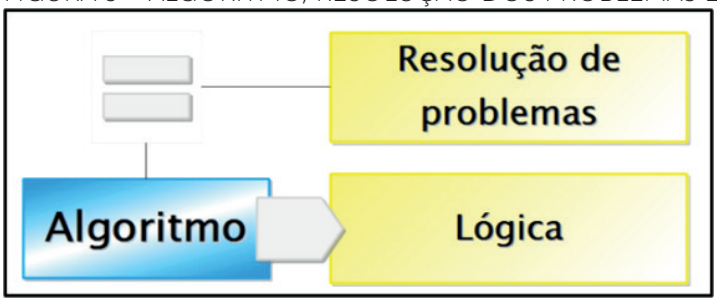
FONTE: Disponível em: <https://www.significados.com.br/logica/>. Acesso em: 3 out. 2016.

E o que seria um programa de computador? É a codificação do algoritmo em linguagem formal, ou seja, uma linguagem compreendida pelo computador e que garante que os passos do algoritmo sejam executados da forma como foram definidos.

É importante estar claro que:

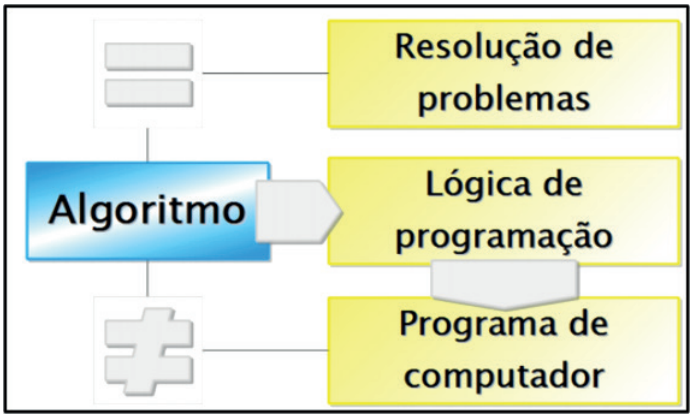
- A capacidade de resolução de problemas, independentemente de sua complexidade, está diretamente relacionada à capacidade de elaboração de algoritmos.
- É a lógica que nos permite construir corretamente os algoritmos.
- No caso de construção de um programa de computador, a lógica de programação consiste em aprender a pensar na mesma sequência de execução desse tipo de programa.
- Um algoritmo não representa, necessariamente, um programa de computador, e sim os passos necessários para realizar uma tarefa ou solucionar um problema.

FIGURA 3 – ALGORITMO, RESOLUÇÃO DOS PROBLEMAS E LÓGICA



FONTE: Disponível em: <http://docplayer.com.br/docs-images/24/3980980/images/33-0.png>. Acesso em: 3 out. 2016.

FIGURA 4 – ALGORITMO, RESOLUÇÃO DOS PROBLEMAS, LÓGICA E PROGRAMA



FONTE: Disponível em: <http://docplayer.com.br/docs-images/24/3980980/images/33-0.png>. Acesso em: 3 out. 2016.

Logo, podemos entender que a sequência para a solução de um problema consiste em:

- Entender e delimitar o problema.
- Esboçar os passos da solução na forma de algoritmos, usando a lógica.
- Escrever o algoritmo em uma linguagem que seja entendida pelos computadores.

2.2 LÓGICA PROPOSICIONAL

A lógica proposicional pode ser entendida por um sistema formal que utiliza proposições que são formadas por expressões ligadas por conectivos lógicos para análise do seu valor lógico. Neste sentido, a lógica proposicional serve para descobrir se o valor lógico de uma proposição é verdadeiro ou falso.

Na Unidade 2, detalharemos este tema e também a construção da tabela verdade.

2.3 APLICABILIDADE DA LÓGICA NO DESENVOLVIMENTO DE PROGRAMAS

Muitos programadores, principalmente os mais antigos profissionais desta área, preferem preparar um programa iniciando com um diagrama de blocos para demonstrar sua linha de raciocínio lógico. Esse diagrama, também denominado por alguns de fluxograma, estabelece a sequência de operações a se efetuar em um programa (MANZANO, 2000).

Essa técnica permite uma posterior codificação em qualquer linguagem de programação de computadores, pois na elaboração do diagrama de blocos não se atinge um detalhamento de instruções ou comandos específicos, os quais caracterizam uma linguagem. A técnica mais importante no projeto da lógica de programas é chamada programação estruturada, a qual consiste em uma metodologia de projeto, objetivando:

- Agilizar a codificação da escrita de programas.
- Facilitar a depuração da sua leitura.
- Permitir a verificação de possíveis falhas apresentadas pelos programas.
- Facilitar as alterações e atualizações dos programas.

E deve ser composta por quatro passos fundamentais:

- Escrever as instruções em sequências ligadas entre si apenas por estruturas sequenciais, repetitivas ou de selecionamento.
- Escrever instruções em grupos pequenos e combiná-las.
- Distribuir módulos do programa entre os diferentes programadores que trabalharão sob a supervisão de um programador sênior, ou chefe de programação.
- Revisar o trabalho executado em reuniões regulares e previamente programadas em que compareçam programadores de um mesmo nível (MANZANO, 2000).

3 CONCEITO DE ALGORITMOS

Atualmente, é difícil encontrarmos alguém que nunca tenha usado computador ou se beneficiado das agilidades possibilitadas pelo mesmo. O uso pode variar: jogos, edição de textos, redes sociais, *e-mails*, operações bancárias e

outras atividades mais complexas. É praticamente impossível imaginar o controle de informações sem o uso do computador.

Muitas pessoas, principalmente as leigas em assuntos tecnológicos, certamente se questionam como ocorre o funcionamento dos computadores, pois as tarefas antes executadas manualmente agora são controladas por uma máquina. O que as pessoas não sabem é que o computador segue as instruções que lhe são passadas através dos algoritmos.

Essas tarefas podem ser entendidas como atividades executadas com frequência e de modo repetido. Por este motivo, torna-se necessário especificar com exatidão e clareza o processo a ser executado e a sequência mais adequada de execução das tarefas.

Algoritmo é uma sequência de ações finitas encadeadas e lógicas que descrevem como um determinado problema deve ser resolvido.

Um algoritmo é formalmente uma sequência finita de passos que levam à execução de uma tarefa. Podemos pensar algoritmo como uma receita, uma sequência de instruções que executam uma meta específica. Estas tarefas não podem ser redundantes nem subjetivas na sua definição, devem ser claras e precisas.

Como exemplo de algoritmos podemos citar os algoritmos das operações básicas (adição, multiplicação, divisão e subtração) de números reais decimais. Outros exemplos seriam os manuais de aparelhos eletrônicos, como um aparelho de som, que explicam passo a passo como, por exemplo, gravar um evento.

Até mesmo as coisas mais simples do nosso dia a dia, podem ser descritas por sequências lógicas. Por exemplo: apesar do nome pouco usual, algoritmos são comuns em nosso cotidiano, como, por exemplo, em uma receita de bolo. Nela está descrita uma série de ingredientes necessários e uma sequência de diversos passos (ações) que devem ser fielmente cumpridos para que se consiga fazer o alimento desejado, conforme se esperava, antes do início das atividades (objetivo bem definido).

Porém, observa-se que uma ordem isolada não permite realizar o processo completo, e para isso é necessário um conjunto de instruções colocadas em ordem sequencial lógica. No exemplo do parágrafo anterior para fazermos um bolo, não podemos começar por colocar os ingredientes no forno. É necessário todo um processo passo a passo para se chegar a este fim.

FONTE: Disponível em: <http://www.ebah.com.br/content/ABAAAfNcgAB/apostila-algoritimos-fluxogramas>. Acesso em: 3 out. 2016.

Com base nisso podemos entender que o ALGORITMO é a sequência dos passos que devem ser seguidos de forma ordenada para garantir a repetibilidade do processo. De forma mais clara, o algoritmo pode ser entendido como qualquer procedimento que recebe um ou mais valores de entrada e nos retorna uma saída.

Um processo intermediário sempre estará associado ao valor de entrada e saída da informação: é o que chamamos de manipulação da informação. No exemplo acima, a manipulação ocorre no cálculo matemático que envolve as três notas na produção do resultado final (MANZANO, 2000).

um algoritmo sempre produz um resultado. E o mais importante em sua estrutura é a sua corretude, ou seja, se ele RESOLVE o problema que motivou a sua CRIAÇÃO.

Na Ciência da computação teórica, a corretude de um algoritmo pode ser afirmada quando se diz que o algoritmo é correto com respeito à determinada especificação. O termo corretudo se refere ao comportamento de entrada-saída do algoritmo (isto é, para cada entrada ele produz uma saída).

Considerações para a criação de um bom ALGORITMO:

- Elaborar um número adequado e finito de passos – nem mais, nem menos do que o necessário para a correta execução.
- Definir com exatidão cada passo – onde começa e onde termina.
- Elaborar de forma consistente as entradas do algoritmo.
- Analisar adequadamente a saída, chegando a sua veracidade.
- Criar a condição de fim, de forma a não permitir que o algoritmo entre em *loop* (execução infinita).

Os passos descritos acima criam o caminho e os procedimentos corretos que devem ser seguidos na resolução de problemas, ou seja, descreve COMO FAZER ALGO através de uma sequência lógica e com um fim predefinido.

Embora você não perceba, utiliza algoritmos de forma intuitiva e automática diariamente quando executamos tarefas comuns. Como estas atividades são simples e dispensam ficar pensando nas instruções necessárias para fazê-las, o algoritmo presente nelas acaba passando despercebido.

4 CARACTERÍSTICAS DE UM ALGORITMO

Todo algoritmo precisa possuir as seguintes características:

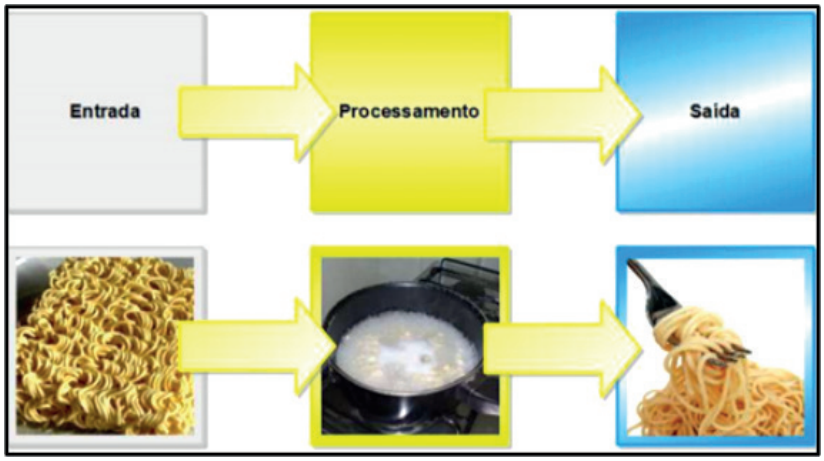
- Entrada: são as informações que alimentam a construção, geralmente usados como parâmetros ou filtros na busca das informações em uma base de dados. Um algoritmo pode não conter valores de entrada. Assim, como poderá apresentar um ou mais valores de tipos de dados distintos como entrada para a lógica construída.
- Saída: todo algoritmo deve produzir um resultado.
- Clareza ou definição: cada passo/instrução/etapa de um algoritmo deve ser claro e não gerar duplo entendimento.
- Efetividade: cada passo/instrução/etapa de um algoritmo deve ser executável.
- Finitude: o algoritmo deve ter uma condição para sair de sua execução. Isso evitará que entre em *loop*. O *loop* traduz a incapacidade do algoritmo de interromper a sua execução.

5 FASES DE UM ALGORITMO

A construção de um algoritmo apresenta três etapas distintas:

- Entrada: são os dados que serão processados pelo algoritmo.
- Processamento: representa os procedimentos necessários de manipulação das informações no intuito de produzir o resultado esperado.
- Saída: é o resultado esperado; são os dados produzidos na etapa de processamento.

FIGURA 5 – FASES DO ALGORITMO



FONTE: Disponível em: <http://docplayer.com.br/docs-images/24/3980980/images/33-0.png>. Acesso em: 3 out. 2016.

6 MÉTODO PARA A CONSTRUÇÃO DE ALGORITMOS

Várias são as práticas adotadas para a construção de algoritmos. De forma simplificada, poderíamos considerar que se deve, prioritariamente:

- Entender o problema a ser resolvido. Um exemplo de problema: somar dois números. Neste caso, imagine o que você precisa para executar a solução.
- Identificar e definir as entradas do algoritmo. No caso do problema proposto, você teria como entrada dois números.
- Descrever os passos para resolver o problema. Basicamente descrever o processo de soma dos dois números.
- Definir os dados de saída. Na situação proposta, o resultado da soma dos dois números usados como entrada do problema.
- Construir o algoritmo para representar a sequência de execução dos passos.
- Transcrever o algoritmo para uma linguagem interpretada por computador.
- Testar a lógica, bem como os passos de execução.

7 REGRAS BÁSICAS PARA A CONSTRUÇÃO DOS ALGORITMOS

Algumas regras precisam ser seguidas para que possamos desenvolver nosso algoritmo:

- Usar somente um verbo por passo/instrução/etapa.
- Escrever de uma forma simples para que possa ser entendido facilmente, inclusive por pessoas que não trabalham na área.
- Abusar da simplicidade e objetividade em relação aos termos e frases.
- Evitar termos ou palavras que permitam duplo entendimento.

8 FORMAS DE REPRESENTAÇÃO DE ALGORITMOS

Várias são as formas de representação dos algoritmos. Algumas técnicas tratam apenas da representação lógica, ignorando detalhes de programação que dependem de uma tecnologia específica (linguagem de programação). Em contrapartida, existem técnicas que de tão detalhistas acabam dificultando a compreensão da solução proposta.

As formas de representação mais conhecidas para a representação de algoritmos são:

- Descrição narrativa.
- Fluxograma convencional.
- Diagrama de Chapin.
- Pseudocódigo, também conhecido como linguagem estruturada ou Portugol.

8.1 DESCRIÇÃO NARRATIVA

Esta forma é conhecida como linguagem natural. É usada sempre quando se deseja que o receptor da mensagem entenda o que será feito, mesmo não tendo domínio acerca da elaboração de algoritmos. Uma das limitações é a imprecisão do entendimento, visto que fica a cargo de quem recebe interpretar o que foi proposto, por exemplo, duas pessoas podem interpretar a mesma mensagem de forma diferente. Outra limitação remete ao fato de que às vezes é necessário escrever muito para transmitir algo simples.

Podemos citar como exemplo o cálculo da média aritmética das notas de um aluno:

- Obter as notas da primeira e da segunda prova.
- Calcular a média aritmética entre as duas.
- Se a média for maior ou igual a 7, o aluno foi aprovado, senão ele foi reprovado.

Quando elaboramos um algoritmo devemos especificar ações claras e precisas, que a partir de um estado inicial, após um período de tempo finito, produzem um estado final previsível e bem definido. Isto significa que o algoritmo fixa um padrão de comportamento a ser seguido, uma norma de execução a ser trilhada, com vistas a alcançar, como resultado final, a solução de um problema, garantindo que sempre que executado, sob as mesmas condições, produza o mesmo resultado.

A importância de se construir um algoritmo: conseguimos visualizar e testar ainda no papel, a solução criada com lógica de programação sem nos preocupar com detalhes computacionais, e uma vez concebida uma solução algorítmica para um problema, esta pode ser traduzida facilmente para qualquer linguagem de programação e agregada das funcionalidades disponíveis nos diversos ambientes, ou seja, a codificação.

FONTE: Disponível em: <http://www.ebah.com.br/content/ABAAAAYK0AC/algoritmos>. Acesso em: 3 out. 2016.

8.2 FLUXOGRAMA CONVENCIONAL

Fluxogramas fazem uso de símbolos universais para ajudar no entendimento do algoritmo. São representações gráficas, em que as formas geométricas propõem ações específicas. Esta forma de representação preocupa-se com detalhes de nível físico da implementação do algoritmo, em que figuras geométricas diferentes representam a entrada e a saída de informações de dispositivos distintos.

Sabemos que uma figura fala por mil palavras. No processo de aprendizagem fixamos com mais facilidade imagens do que conceitos escritos.

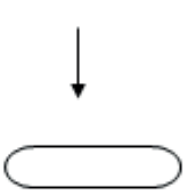
O **diagrama de blocos ou fluxograma** é uma forma padronizada eficaz para **representar** os passos lógicos de um determinado processamento. Com o diagrama podemos definir uma sequência de símbolos, com significado bem definido. Portanto, sua principal função é a de facilitar a visualização dos passos de um processamento.

O fluxograma é uma ferramenta usada e desenvolvida pelos profissionais de análise de sistemas, bem como, por alguns profissionais de Organização, Sistemas e Métodos. Tem como finalidade descrever o fluxo, seja manual ou mecânico, especificando os suportes usados para os dados e informações. Usa símbolos convencionais, permitindo poucas variações. Representado por alguns desenhos geométricos básicos, os quais indicarão os símbolos de entrada de dados, do processamento de dados e da saída de dados, acompanhados dos procedimentos requeridos pelo analista de sistemas e a serem realizados pelo programador por meio do desenvolvimento do raciocínio lógico, o qual deverá solucionar o problema do programa a ser processado pelo computador. É uma ferramenta de uso em diversas áreas do conhecimento humano por traduzir em formato gráfico algum procedimento ou norma.

FONTE: Disponível em: <https://www.emersonbarros.com.br/wp.../algoritmos-e-fluxogramas.doc>. Acesso em: 3 out. 2016.

A figura a seguir mostra as principais formas geométricas usadas em fluxogramas.

FIGURA 6 – FORMAS GEOMÉTRICAS



FLUXO DE DADOS

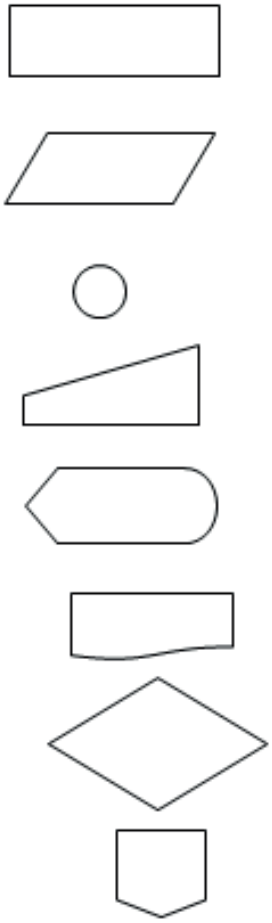
Indica o sentido do fluxo de dados

Conecta os demais símbolos

TERMINAL

Indica o INICIO ou FIM de um processamento

Exemplo: Início do algoritmo



PROCESSAMENTO

Processamento em geral

Exemplo: Cálculo de dois números

ENTRADA/SAIDA (Genérica)

Operação de entrada e saída de dados

Exemplo: Leitura e Gravação de Arquivos

DESVIO (conector)

Permite o desvio para um ponto qualquer do programa

ENTRADA MANUAL

Indica entrada de dados via Teclado

Exemplo: Digite a nota da prova 1

EXIBIR

Exibe informações

Exemplo: Exiba informações do cálculo

SAIDA

Representa operação de saída dos dados

Exemplo: Imprima o relatório

DECISÃO

Permite elaborar processos de decisão

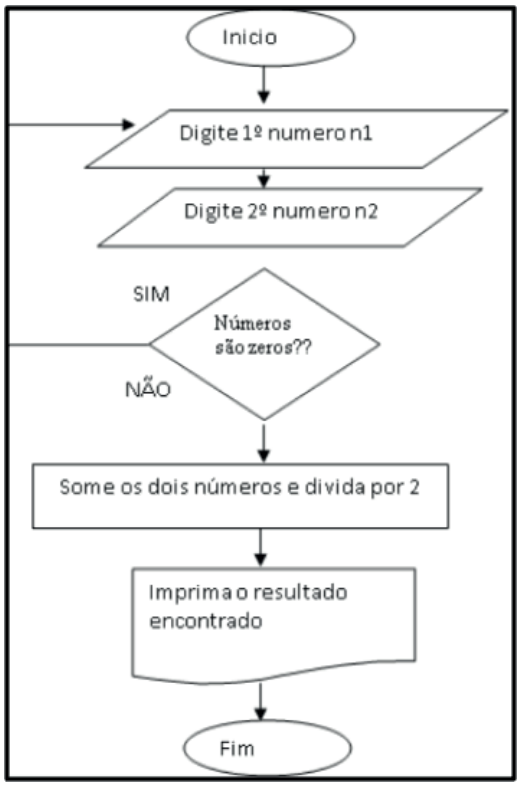
CONECTOR DE PAGINA

Permite informar de qual página vem o fluxograma

FONTE: Adaptado de Manzano (2000)

Quando você optar pelo uso de fluxograma para a representação inicial da solução, sempre coloque texto explicativo dentro das figuras. Sem o texto, não haverá entendimento da figura e do seu fluxo. **Exemplo**: Fluxograma de um programa para ler dois números aleatórios diferentes de zero, calcular a média dos mesmos e mostrar o resultado encontrado.

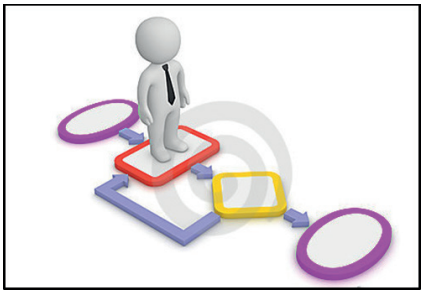
FIGURA 7 – EXEMPLO DE FLUXOGRAMA



FONTE: A autora

Os fluxogramas sempre apresentam um símbolo inicial, que indica o início do algoritmo. São vários os símbolos finais que indicam o encerramento do algoritmo. Há somente um caminho a ser seguido, tendo como ponto de partida o símbolo inicial, que indica que a sequência de execução é única. Isso significa que apesar de existirem vários caminhos apontados para a mesma figura de um diagrama, haverá somente um único caminho saindo dela. Os símbolos finais demonstram o encerramento da lógica. Logo, não há caminhos ou fluxos saindo deles (MANZANO, 2000).

FIGURA 8 – CAMINHOS DO ALGORITMO



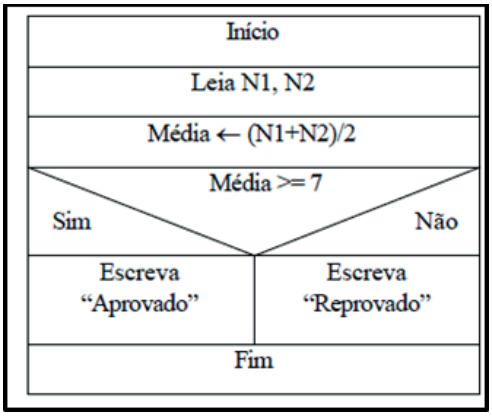
FONTE: Disponível em: <https://pt.dreamstime.com/foto-de-stock-royalty-free-algoritmo-com-o-homem-3d-image23460065>. Acesso em: 4 out. 2016.

8.3 DIAGRAMA DE CHAPIN

Foi criado com a intenção de substituir os diagramas tradicionais. O criador foi Ned Chapin. O objetivo era apresentar uma visão mais hierárquica e estruturada da lógica do sistema. A vantagem do uso consiste no fato de que é mais fácil representar as estruturas que tem um ponto de entrada e um ponto de saída e são compostas pelas estruturas básicas de controle de sequência, seleção e repartição. A recursividade é mais facilmente representada neste tipo de diagrama.

A figura a seguir apresenta um exemplo do tipo de diagrama de Chapin para o algoritmo de cálculo da média de um aluno.

FIGURA 9 – EXEMPLO DE DIAGRAMA DE CHAPIN



FONTE: A autora

8.4 PSEUDOCÓDIGO

O pseudocódigo ou portugol é uma representação muito parecida com a forma de escrita dos programas para a versão computacional. Esta técnica de algoritmização é baseada em uma PDL – *Program Design Language* (Linguagem de Projeto de Programação). Aqui, vamos apresentá-la em português. A forma original de escrita é conhecida como inglês estruturado, muito parecida com a notação da linguagem PASCAL. A PDL (neste caso, o pseudocódigo) é usada como referência genérica para uma linguagem de projeto de programação, tendo como finalidade mostrar uma notação para elaboração de algoritmos, os quais serão utilizados na definição, criação e desenvolvimento de uma linguagem computacional (Clipper, C, Fortran, Pascal, Delphi, Visual-Objects) e sua documentação. Adiante é apresentado um exemplo deste tipo de algoritmo.

Os algoritmos são independentes das linguagens de programação. Ao contrário de uma linguagem de programação, não existe um formalismo rígido de como deve ser escrito o algoritmo.

O algoritmo deve ser fácil de se interpretar e fácil de codificar, ou seja, ele deve ser o intermediário entre a linguagem falada e a linguagem de programação (MANZANO, 2000).

8.4.1 Representação de um algoritmo na forma de pseudocódigo

O pseudocódigo tem uma escrita simplificada. Não pode ser considerado um código real. Porém, assemelha-se à linguagem de programação. É um recurso didático eficiente, amplamente utilizado em disciplinas de lógica, a fim de desenvolver e habituar os alunos com a forma de escrita da lógica. Não há no pseudocódigo a preocupação com as questões técnicas apresentadas pelas linguagens de programação.

Qualquer pessoa pode desenvolver um pseudocódigo, pois não há regras ou padrões definidos para esta construção. A figura a seguir demonstra dois exemplos de algoritmos na forma de pseudocódigo:

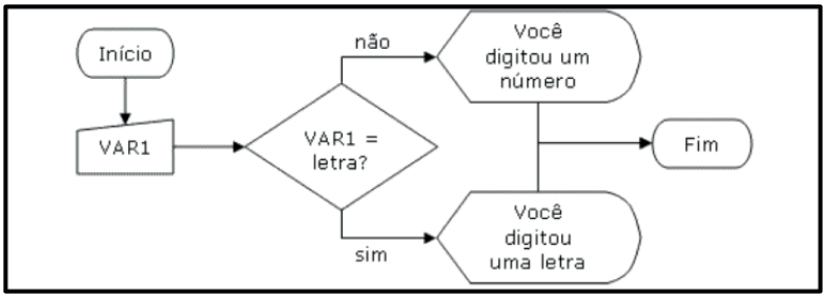
FIGURA 10 – EXEMPLO DE ALGORITMO EM PSEUDOCÓDIGO

```
pseudocódigo1
-------------------------------------------------------------------------------
INICIO
    entrada de dado : grava em VAR1
    verificar var1 : letra ?
        verdade : imprimir dado -> "Você digitou uma letra"
        falso : imprimir dado -> "Você digitou um número"
FIM
-------------------------------------------------------------------------------
pseudocódigo2
-------------------------------------------------------------------------------
INICIO: procedimento
    VARIÁVEIS var1
    var1 <- entrada de dados:TECLADO
    se (var1 É letra) então
        imprimir dado:MONITOR -> "Você digitou uma letra"
    caso contrário
        imprimir dado:MONITOR -> "Você digitou um número"
FIM: procedimento
-------------------------------------------------------------------------------
```

FONTE: Disponível em: <http://www.tiexpert.net/programacao/algoritmo/fluxogramas-e-pseudocodigo.php>. Acesso em: 14 out. 2016.

Agora, a situação do exemplo acima em fluxograma:

FIGURA 11 – EXEMPLO DE PSEUDOCÓDIGO REPRESENTADO POR FLUXOGRAMA



FONTE: Disponível em: <http://www.tiexpert.net/programacao/algoritmo/fluxogramas-e-pseudocodigo.php>. Acesso em: 14 out. 2016.

Na Figura 10, algoritmo é o termo que indica o início da definição de um algoritmo em forma de pseudocódigo.

Nome do algoritmo: cada algoritmo deverá ter um nome específico, preferencialmente indicando o objetivo para o qual foi criado. Exemplo: Algoritmo Calcula_Media_aluno.

Variáveis: todo algoritmo deve apresentar uma seção para a declaração do nome e tipo de dados das variáveis que terão a função de armazenar valores dentro da lógica.

Subalgoritmos ou código: é a parte onde o código é escrito e estruturado.

Início e Fim: são, respectivamente, as palavras que delimitam o início e o término das instruções que a lógica vai seguir

Sobre o algoritmo em portugol

A maioria esmagadora das linguagens de programação de computadores é em língua inglesa. Para facilitar o aprendizado de lógica de programação foram criadas algumas pseudolinguagens.

O portugol é uma pseudolinguagem de programação, uma simbiose de português, algol e pascal, criada originalmente em inglês, com a proposta de ser independente da linguagem nativa (ou seja, existe em japonês, javanês, russo...).

Basicamente, é uma notação para algoritmos, a ser utilizada na definição, criação, desenvolvimento e documentação dos programas.

FONTE: Disponível em: <https://www.emersonbarros.com.br/wp.../algoritmos-e-fluxogramas.doc>. Acesso em: 3 out. 2016.

Vamos usar como exemplo o problema proposto anteriormente para somar dois números:

$$Soma = \frac{N1 + N2}{2}$$

Para montar o algoritmo proposto, são necessários os seguintes questionamentos:

a) Quais são os dados de entrada?	**R.**: Os dados de entrada são N1, N2.
b) Qual será o processamento a ser utilizado?	**R.**: Somar os dois números.
c) Quais serão os dados de saída?	**R.**: O resultado é a soma dos dois números e sua divisão por 2.

INICIO do algoritmo
- Ler numero 1
- Ler numero 2
- Somar os dois numeros
- Exibir o resultado

FIM do algoritmo

Todo algoritmo deve sempre ser testado. Existe um recurso eficaz para o teste do algoritmo sem o uso do computador: **TESTE DE MESA**. Este teste consiste, basicamente, em seguir os passos propostos pelo algoritmo, no intuito de aferir a lógica proposta. Para isso é necessário criar uma tabela com as variáveis e seus resultados.

Exemplo de algoritmo em portugol:

```
        Programa média
Var
  Nome: caractere;
  N1, N2: real;
  Soma, média: real;
Início
        Imprima "Informe os dois números: "
  Leia n1;
        Leia n2;
        Soma <= n1 + n2 ;
  Imprima Soma;
        Fim
```

A diferença entre pseudocódigo, portugol e as linguagens de computador consistem no fato de que as linguagens de computador são compiladas e testadas no computador.

Lembre-se de que o fluxograma e o pseudocódigo são as duas técnicas importantes para a documentação da solução de um problema computacional.

9 TIPOS DE DADOS

Todos os algoritmos, depois de transcritos para linguagens de programação, ou já criados nelas, têm a função de manipular informações retornadas das bases de dados.

As informações computacionais podem ser divididas em duas categorias principais:

- As **instruções**, que comandam o funcionamento da máquina e determinam a maneira como devem ser tratados os dados.
- Os **dados** propriamente ditos, que correspondem à porção das informações a serem processadas pelo computador.

Os dados, geralmente, são armazenados em variáveis auxiliares que são utilizadas para manipular as informações. Por isso, o tipo de dado é uma característica importante na definição das variáveis. O tipo de dado é utilizado para mostrar ao compilador do programa quais conversões serão necessárias para carregar em memória os valores utilizados na execução dos programas. O tipo de dado também auxilia os programadores a detectar erros de escrita no código, bem como atribuições de valores indevidos às variáveis, exemplo, armazenar um nome em uma variável numérica.

As linguagens de programação apresentam variações na definição dos tipos de dados. Dependendo da linguagem de programação, o tipo de um dado é verificado diferentemente, de acordo com a análise léxica, sintática e semântica do compilador ou interpretador da linguagem.

Os tipos de dados podem ser compreendidos e classificados em três grupos distintos, que veremos na sequência.

1. Tipo de dado estático e dinâmico

A verificação do tipo de dado de uma informação é feita de forma estática, quando o código está sendo compilado. Em C, C++ e Java, por exemplo, os tipos de dados são estáticos.

A verificação na forma dinâmica ocorre em tempo de execução. Em Lisp, PHP e Python os dados são dinâmicos.

2. Tipo de dado forte e fraco

Nas linguagens de programação categorizadas como fortes (Java, Pascal, SQL), o tipo de dado de um valor deve, obrigatoriamente, ter o mesmo tipo de dado da variável. Do contrário, será gerado erro na hora da compilação.

Exemplo de (Sintaxe genérica)

```
Declarar Variáveis
        TEXTO nome
        INTEIRO idade

Atribuições
nome = "Ana"
idade = 37
```

Exemplo de linguagem com tipo de dado fraco: PHP

3. Tipo de dado primitivo e composto

Os **tipos primitivos** (nativos ou básicos) são fornecidos pelas linguagens de programação. Por este motivo, dependendo da linguagem de implementação utilizada na codificação, os tipos primitivos podem ou não possuírem os mesmos tipos de dados das informações guardadas na memória.

Em computação existem apenas quatro tipos de dados primitivos, algumas linguagens subdividem esses tipos de dados em outros de acordo com a capacidade de memória necessária para a variável, mas de modo geral, os tipos de dados primitivos são:

- **INTEIRO**: Representa valores numéricos negativo ou positivo sem casa decimal, ou seja, valores inteiros.
- **REAL**: Representa valores numéricos negativo ou positivo com casa decimal, ou seja, valores reais. Também são chamados de ponto flutuante.
- **LÓGICO**: Representa valores booleanos, assumindo apenas dois estados, VERDADEIRO ou FALSO. Pode ser representado apenas um *bit* (que aceita apenas 1 ou 0).

- **TEXTO**: Representa uma sequência de um ou mais caracteres. Colocamos os valores do tipo TEXTO entre " " (aspas duplas) ou ' ' (aspas simples), dependendo da linguagem.

Algumas linguagens de programação dividem esses tipos primitivos de acordo com o espaço necessário para os valores daquela variável. Na linguagem Java, por exemplo, o tipo de dados inteiro é dividido em quatro tipos primitivos: **byte**, **short**, **int** e **long**. A capacidade de armazenamento de cada um deles é diferente.

- **byte**: é capaz de armazenar valores entre -128 até 127.
- **short**: é capaz de armazenar valores entre – 32768 até 32767.
- **int**: é capaz de armazenar valores entre –2147483648 até 2147483647.
- **long**: é capaz de armazenar valores entre –9223372036854775808 até 9223372036854775807.

Essa divisão é uma particularidade da linguagem de programação que está sendo utilizada. O objetivo é otimizar a utilização da memória. Em algumas linguagens de programação não é necessário especificar o tipo de dados da variável, os quais são identificados dinamicamente. Porém, é necessário informar o tipo de dado de cada variável em algoritmos.

FONTE: Disponível em: <http://www.dicasdeprogramacao.com.br/tipos-de-dados-primitivos/>. Acesso em: 17 out. 2016.

Existem tipos de dados primitivos mais elaborados, como por exemplo: tuplas, listas ligadas, números complexos, números racionais e tabela *hash*. Estes tipos de dados são encontrados em linguagens funcionais (tudo é baseado em funções).

Os **tipos de dados compostos** podem ser construídos tendo como base os dados primitivos ou também outros tipos de dados compostos. Esse processo é conhecido como **COMPOSIÇÃO**. São usados para descrever a estrutura das linhas de registro. No caso da linguagem C, as cadeias de caracteres são tipos de dados compostos. No caso do JavaScript esta característica é nativa da própria linguagem.

RESUMO DO TÓPICO 1

Nesta unidade, você aprendeu que:

- O algoritmo surgiu no ano 300 a.C.
- Algoritmo exige prática: não é possível estudar ou copiar algoritmos para o aprendizado. Somente aprendemos algoritmos no processo de construí-los e testá-los.
- Os problemas que são resolvidos tecnologicamente através de programas de computador, somente terão suas soluções implementadas e executadas através dos algoritmos.
- A lógica pode ser entendida como sendo o estudo das leis do raciocínio e da forma como ela será aplicada no intuito de demonstrar a verdade.
- Os tipos de lógica são:
  - **Lógica aristotélica**: tem como objeto de estudo o **pensamento**, assim como as leis e regras que o controlam, para que esse pensamento seja correto.
  - **Lógica de programação**: é a linguagem usada para criar um programa de computador.
  - **Lógica de argumentação**: a lógica de argumentação permite verificar a validade ou se um enunciado é verdadeiro ou não.
  - **Lógica matemática**: a lógica matemática (ou lógica formal) estuda a lógica segundo a sua estrutura ou forma.
  - **Lógica proposicional**: a lógica proposicional é uma área da lógica que examina os raciocínios de acordo com as relações entre orações (proposições), as unidades mínimas do discurso, que podem ser verdadeiras ou falsas.
- **Algoritmo** é uma sequência de ações finitas encadeadas e lógicas que descrevem como um determinado problema deve ser resolvido.
- A construção de um algoritmo apresenta três etapas distintas:
  - Entrada: são os dados que serão processados pelo algoritmo.
  - Processamento: Representa os procedimentos necessários de manipulação das informações no intuito de produzir o resultado esperado.
  - Saída: É o resultado esperado, são os dados produzidos na etapa de processamento.
- **Descrição narrativa**: é a forma da linguagem natural. É usada sempre que se deseja que o receptor da mensagem entenda o que será feito, mesmo não tendo domínio acerca da elaboração de algoritmos.

- **Fluxogramas**: fazem uso de símbolos universais para ajudar no entendimento do algoritmo. São representações gráficas em que as formas geométricas propõem ações específicas.

- **Diagrama de Chapin**: foi criado com a intenção de substituir os diagramas tradicionais. O criador foi Ned Chapin. O objetivo era apresentar uma visão mais hierárquica e estruturada da lógica do sistema.

- O **pseudocódigo** ou portugol é uma representação muito parecida com a forma de escrita dos programas para a versão computacional.

- Todos os algoritmos, depois de transcritos para linguagens de programação, ou já criados nelas, têm a função de manipular informações retornadas das bases de dados.

- As informações computacionais podem ser divididas em duas categorias principais:

  - As **instruções**, que comandam o funcionamento da máquina e determinam a maneira como devem ser tratados os dados.
  - Os **dados** propriamente ditos, que correspondem à porção das informações a serem processadas pelo computador.

- Os tipos de dados podem ser compreendidos e classificados em três grupos distintos:

  - Tipo de dado estático e dinâmico.
  - Tipo de dado forte e fraco.
  - Tipo de dado primitivo e composto.

AUTOATIVIDADE

1 Diferencie um algoritmo de um programa.

2 Crie algoritmos simplificados para executar cada uma das tarefas a seguir:

a) Tomar um banho
b) Fazer um bolo

3 Pense em um problema existente na sua rotina diária (particular ou no trabalho) e monte um algoritmo de acordo com os passos que você normalmente utiliza para resolvê-lo.

TÓPICO 2

VARIÁVEIS, CONSTANTES, OPERADORES E EXPRESSÕES

1 INTRODUÇÃO

Todas as tarefas realizadas pelos computadores através de comandos codificados utilizam variáveis para armazenamento e manipulação de dados. São exemplos de informações armazenadas em variáveis: nome, idade, CPF, sexo, valor do salário, data de nascimento etc. As constantes têm um valor atribuído quando são declaradas. Por exemplo: o valor de Pi (π).

Neste tópico, você vai conhecer os diferentes tipos de variáveis usadas nos algoritmos bem como a declaração e uso das constantes.

2 VARIÁVEIS, CONSTANTES

Por regra, cada variável armazena apenas um tipo de dado. Variáveis de texto armazena caracteres, por exemplo, o nome da pessoa. Variáveis numéricas armazenam valores: idade, salário. Já na informação do sexo, armazena-se apenas um caractere. A informação de sexo, poderia ser numérica desde que se criasse um domínio na base de dados, indicando, por exemplo, que o número 1 indica feminino e 2, masculino. Para entender, compare o armazenamento em variáveis com objetos guardados em gavetas. Cada gaveta poderá guardar apenas um tipo de objeto: ou calças, ou camisas, ou jaquetas. O mesmo acontece com as variáveis, cada variável poderá armazenar apenas um tipo de dado: numérico, texto ou data, por exemplo.

Também podemos compreender a variável como algo incerto ou inconstante. Sua utilidade se baseia no fato de que o volume de informações a armazenar é muito grande e diversificado, justificando, dessa forma, a diversidade dos dados processados.

É importante lembrar que as informações apresentam tipos de dados diferentes e são armazenadas na memória através de variáveis criadas no código de programação com o objetivo de auxiliar a manipulação das mesmas, bem como guardá-las.

As variáveis, geralmente, são compostas por dois atributos distintos:

- nome
- o tipo de dados que armazenará

Dentro de um mesmo programa não poderão ser criadas duas ou mais variáveis com o mesmo nome.

Cada linguagem de programação estabelece suas próprias regras de formação de nomes de variáveis, e tem definições distintas para os mesmos tipos de dados.

Dicas importantes para a criação de variáveis:

- Usar nomes simples, que especificam sua funcionalidade, e ainda compostos e antecedidos por um prefixo. Ex.: AUX_SALARIO ou WW_SALARIO. Prefixos são importantes para identificar o termo como representativo de uma variável.
- Não usar palavras reservadas em linguagens de programação para criar variáveis. O mesmo se aplica para nomes de tabelas e colunas na base de dados.
- Não iniciar o nome das variáveis com símbolos ou caracteres especiais.
- O nome da variável deve indicar a função que a mesma irá exercer. Deve lembrar ou indicar a informação que será armazenada em sua estrutura.

Uma vez definidos, os atributos nome e tipo de dado de uma variável, estes não podem ser alterados e assim permanecem durante toda a sua existência, desde que o programa que utiliza não seja modificado. Por outro lado, o atributo informação está constantemente sujeito a mudanças de acordo com o fluxo de execução do programa.

Todas as variáveis utilizadas em algoritmos devem ser definidas antes de serem utilizadas. Isso se faz necessário para permitir que o compilador reserve um espaço na memória para elas. Nos algoritmos, todas as variáveis utilizadas serão definidas no início do mesmo, como por exemplo:

```
Algoritmo calcula_salario;

Aux_salario  numérico(10,5)
Aux_nom_func  texto (100);

 Inicio
 ....
            ....
 .....
 Fim
```

Outro exemplo

```
Algoritmo exemplo_2;

nome: caracter[30]
idade: inteiro
salário: real
tem_filhos: lógico

 Inicio
 ....
 ....
 ...
 Fim;
```

3 CONSTANTES

Constantes têm valores fixos ou estáveis. São declaradas na mesma seção das variáveis. Sua utilidade consiste no fato de que ela será declarada uma única vez com valor fixo. Caso seja necessário alterar o valor, esta mudança será feita na sua declaração, não sendo necessário alterar o código de programação que referencia a mesma, isto é, o valor de uma constante permanece o mesmo durante toda a execução do programa.

Exemplo de declaração de constantes:

CONST <nome_da_constante> = <valor>
Exemplo de definição de constantes:
CONST pi = 3.14159

4 EXPRESSÕES E OPERADORES

Uma expressão em um programa é a parte da instrução que produz um valor, normalmente através do uso de operandos (valores) e operadores (indicam a operação a ser realizada).

4.1 OPERADORES

Os operadores são meios pelos quais incrementamos, decrementamos, comparamos e avaliamos dados dentro do computador.

Operadores são elementos fundamentais que atuam sobre operandos e produzem um determinado resultado. Por exemplo, a expressão 3 + 2 relaciona dois operandos (os números 3 e 2) por meio do operador (+) que representa a operação de adição.

De acordo com o número de operandos sobre os quais os operadores atuam, os últimos podem ser classificados em:

- Binários: quando atuam sobre dois operandos. Esta operação é chamada diádica. Ex.: os operadores das operações aritméticas básicas (soma, subtração, multiplicação e divisão).
- Unários: quando atuam sobre um único operando. Esta operação é chamada monádica. Ex.: o sinal de menos (-) na frente de um número, cuja função é inverter seu sinal.

Outra classificação dos operadores é feita considerando o tipo de dado de seus operandos e do valor resultante de sua avaliação. Segundo esta classificação, os operandos dividem-se em aritméticos, lógicos e literais. Esta divisão está diretamente relacionada ao tipo de expressão onde aparecem os operadores.

Um caso especial é o dos operadores relacionais, que permite comparar pares de operandos de tipos de dados iguais, resultando sempre num valor lógico.

4.1.1 Operadores de atribuição

Um operador de atribuição serve para atribuir um valor a uma variável. Em Algoritmo usamos o operador de atribuição:

:=

A sintaxe de um comando de atribuição é:

NomedaVariável := expressão

A expressão localizada no lado direito do sinal de igual é avaliada e armazenado o valor resultante na variável à esquerda. O nome da variável aparece sempre sozinho, no lado esquerdo do sinal de igual deste comando.

4.1.2 Operadores aritméticos

Os operadores aritméticos relacionam as operações aritméticas básicas, conforme a tabela a seguir:

Operador	Tipo	Operação	Prioridade
+	Binário	Adição	4
-	Binário	Subtração	4
*	Binário	Multiplicação	3
/	Binário	Divisão	3
MOD	Binário	Resto da Divisão	3
DIV	Binário	Divisão Inteira	3
**	Binário	Exponenciação	2
+	Unário	Manutenção do Sinal	1
-	Unário	Inversão do Sinal	1

A prioridade entre operadores define a ordem em que devem ser avaliados dentro de uma mesma expressão.

4.1.3 Operadores relacionais

Os operadores relacionais são operadores binários que devolvem os valores lógicos “verdadeiro e falso”.

Operador	Comparação
>	maior que
<	menor que
>=	maior ou igual
<=	menor ou igual
=	igual
<>	diferente

Estes valores são somente usados quando se deseja efetuar comparações. Comparações só podem ser feitas entre objetos de mesma natureza, isto é, variáveis do mesmo tipo de dado. O resultado de uma comparação é sempre um valor lógico, por exemplo, digamos que a variável inteira escolha contenha o valor 7. A primeira das expressões a seguir fornece um valor falso, e a segunda um valor verdadeiro:

escolha <= 5
escolha > 5

O quadro a seguir apresenta mais alguns exemplos de operadores relacionais:

Expressão	Resultado Lógico
1 = 1	Verdadeiro
1 > 5	Falso
1 < 5	Verdadeiro
“teste” <> “teste”	Falso

Com valores string, os operadores relacionais comparam os valores ASCII dos caracteres correspondentes em cada string. Uma string é dita "menor que" outra se os caracteres correspondentes tiverem os números de códigos ASCII menores. Por exemplo, todas as expressões a seguir são verdadeiras:

“algoritmo” > “ALGORITMO”
“ABC” < “EFG”
“Pascal” < “Pascal compiler”

Observe que as letras minúsculas têm códigos ASCII maiores do que os das letras maiúsculas. Observe também que o comprimento da string se torna o fator determinante na comparação de duas strings, quando os caracteres existentes na string menor são os mesmos que os caracteres correspondentes na string maior. Neste caso, a string maior é dita “maior que” a menor.

4.1.4 Operadores lógicos

Os operadores lógicos ou booleanos são usados para combinar expressões relacionais. Também devolvem como resultado valores lógicos verdadeiro ou falso.

Operador	Tipo	Operação	Prioridade
OU	Binário	Disjunção	3
E	Binário	Conjunção	2
NÃO	Unário	Negação	1

Uma expressão relacional ou lógica retornará falso para o valor lógico falso, resultante de uma expressão, e verdadeiro para o valor lógico verdade.

Fornecendo dois valores ou expressões lógicas, representadas por expressão1 e expressão2, podemos descrever as quatro operações lógicas a seguir:

expressão1 E expressão2 é verdadeiro somente se ambas, expressão1 e expressão2, forem verdadeiras. Se uma for falsa, ou se ambas forem falsas, a operação E também será falsa.

expressão1 OU expressão2 é verdadeiro se tanto a expressão1 como a expressão2 forem verdadeiras, isto é, se uma delas for verdadeira.

As operações OU só resultam em valores falsos se ambas, expressão1 e expressão2, forem falsas.

NÃO expressão1 avalia verdadeiro se expressão1 for falsa; de modo contrário, a expressão NÃO resultará em falso, se expressão1 for verdadeira.

A tabela a seguir apresenta mais alguns exemplos de operadores lógicos:

Expressão	Resultado Lógico
(1 = 1) E (1 = 2)	Falso
(1 = 1) E (2 = 2)	Verdadeiro
(1 = 1) OU (1 = 2)	Verdadeiro
(1 = 3) OU (1 = 2)	Falso
NÃO (1 = 1)	Falso

4.1.5 Operadores literais

Os operadores que atuam sobre caracteres variam muito de uma linguagem para outra. O operador mais comum e mais usado é o operador que faz a concatenação de strings: toma-se duas strings e acrescenta-se (concatena-se) a segunda ao final da primeira.

O operador que faz esta operação é: +

Por exemplo, a concatenação das strings “ALGO” e “RITMO” é representada por:

“ALGO” + ”RITMO”

O resultado de sua avaliação é: “ALGORITMO”

5 EXPRESSÕES

O conceito de expressão em termos computacionais está intimamente ligado ao conceito de expressão ou fórmula matemática, em que um conjunto de variáveis e constantes numéricas relacionam-se por meio de operadores aritméticos compondo uma fórmula que, uma vez avaliada, resulta num valor.

5.1 EXPRESSÕES ARITMÉTICAS

Expressões aritméticas são aquelas cujo resultado da avaliação é do tipo numérico, seja ele inteiro ou real. Somente o uso de operadores aritméticos, variáveis numéricas e parênteses é permitido em expressões deste tipo.

5.1.1 Expressões lógicas

Expressões lógicas são aquelas cujo resultado da avaliação é um valor lógico verdadeiro ou falso. Nestas expressões são usados os operadores relacionais e os operadores lógicos, podendo ainda serem combinados com expressões aritméticas.

Quando forem combinadas duas ou mais expressões que utilizem operadores relacionais e lógicos, os mesmos devem utilizar os parênteses para indicar a ordem de precedência.

5.1.2 Expressões literais

Expressões literais são aquelas cujo resultado da avaliação é um valor literal (caractere). Neste tipo de expressões só é usado o operador de literais (+).

5.1.3 Avaliação de expressões

Expressões que apresentam apenas um operador podem ser avaliadas diretamente. No entanto, à medida que as elas vão se tornando mais complexas com o aparecimento de mais de um operando na mesma expressão é necessária a avaliação passo a passo, tomando um operador por vez. A sequência destes passos é definida de acordo com o formato geral da expressão, considerando-se a prioridade (precedência) de avaliação de seus operadores e a existência ou não de parênteses.

As seguintes regras são essenciais para a correta avaliação de expressões:

1. Deve-se observar a prioridade dos operadores, conforme mostrado nas tabelas de operadores:

Operadores de maior prioridade devem ser avaliados primeiro. Se houver empate com relação à precedência, então a avaliação se faz da esquerda para a direita.

2. Os parênteses usados em expressões têm o poder de "roubar" prioridade dos demais operadores, forçando a avaliação da subexpressão em seu interior.

3. Entre os quatro grupos de operadores existentes, a saber, aritmético, lógico, literal e relacional, há uma certa prioridade de avaliação: os aritméticos e literais devem ser avaliados primeiro; a seguir, são avaliadas as subexpressões com operadores relacionais e, por último os operadores lógicos são avaliados.

Exercícios

1. Dadas as variáveis e operações:
v1 := 32
v2 := 5 + v1
v1 := v2 * 2

Como fazer para segurar e mostrar o valor inicial da variável v1 no final das operações?

2. Como fazer para passar o valor de uma variável para outra e vice-versa?

3. Quais as operações necessárias para intercambiar os valores de três variáveis, a, b e c, de modo que a fique com o valor de b; b fique com o valor de c e c fique com o valor de a?

4. Se X possui o valor 15 e foram executadas as seguintes instruções:
X := X + 3
X := X - 6
X := X / 2
X := 3 * X

Qual será o valor armazenado em X?

FONTE: Disponível em: <https://www.emersonbarros.com.br/wp.../algoritmos-e-fluxogramas.doc>. Acesso em: 7 out. 2016.

6 INSTRUÇÕES PRIMITIVAS

As instruções primitivas compõem os comandos básicos na funcionalidade dos computadores, como, por exemplo, a entrada e saída dos dados, fazendo a comunicação entre os usuários e a máquina.

Conceitos importantes para a fixação:

- Dispositivos de entrada: usado pelo usuário ou pela memória do computador para transferir informações. Exemplo: teclado, *mouse*, leitor de código de barras etc.
- Dispositivo de saída: forma pela qual o computador transfere informações ao usuário: monitor de vídeo, impressoras etc.
- Sintaxe é a forma como os comandos devem ser escritos, evitando erros de compilação dos programas.
- Semântica é o significado, ou seja, o conjunto de ações que serão exercidas pelo computador durante a execução do referido comando.

ESTUDOS FUTUROS: Nas próximas unidades, o código sempre será apresentado com sintaxe e semântica: forma como são escritos e a ação que deverão executar.

6.1 COMANDOS DE ATRIBUIÇÃO

É o comando que indica o recebimento de valor para a variável:

Sintaxe:
Variável := valor; (variável recebe valor);
Aux_preco := 154;

Cuidados em relação à atribuição de valores:

Não atribuir valores maiores do que a variável pode suportar. Neste caso, o compilador apresentará um erro especificando esta situação.
Dar um nome adequado à variável.

A seguir, temos um exemplo de um algoritmo utilizando o comando de atribuição:

```
Algoritmo exemplo_comando_de_atribuição
Var preço_unit, preço_tot : real
quant : inteiro
Início
  preco_unit := 5.0
  quant := 10
 preço_tot := preço_unit * quant
  Fim.
```

6.2 COMANDOS DE SAÍDA DE DADOS

Os comandos de saída exibem o resultado do algoritmo ou programa para o usuário.

A seguir, temos um exemplo de um algoritmo utilizando o comando de saída de dados:

```
Algoritmo exemplo_comando_de_saída_de_dados
Var preço_unit, preço_tot : real
 quant : inteiro
Início
 preco_unit := 5.0
 quant := 10
 preço_tot := preço_unit * quant
  Escreva preço_tot
   Fim.
```

O comando de saída é representado pela palavra “Escreva”, a qual exibirá o valor da variável preco_total.

6.3 COMANDOS DE ENTRADA DE DADOS

Os comandos de entrada são os comandos de leitura que transferem as informações para a memória para que possam ser usadas na lógica. Comandos de entrada são representados pelo comando LEIA. Há duas sintaxes possíveis para esta instrução:

· LEIA <variável>
 Ex.: LEIA X
· LEIA <lista_de_variáveis>
Ex.: LEIA nome, endereco, cidade

A seguir, temos um exemplo de um algoritmo utilizando o comando de entrada de dados:

```
Algoritmo exemplo_comando_de_entrada_de_dados
Var preço_unit, preço_tot : real
quant : inteiro
Início
  Leia preco_unit, quant
 preço_tot := preço_unit * quant
 Escreva preço_tot
  Fim.
```

É importante lembrar que:

- Toda vez que um programa estiver esperando que o usuário forneça a ele um determinado dado (operação de leitura), ele deve antes enviar uma mensagem dizendo ao usuário o que ele deve digitar, por meio de uma instrução de saída de dados.

- Antes de enviar qualquer resultado ao usuário, um programa deve escrever uma mensagem explicando o significado do mesmo.

Essas regras, sempre que aplicadas, geram uma interface mais amigável ao usuário.

A seguir, temos um exemplo do algoritmo anterior, utilizando as regras de construção de uma interface amigável!

```
Algoritmo exemplo_interface_amigavel
Var preço_unit, preço_tot : real
 quant : inteiro
  Início
   Escreva "Digite o preço unitário:"
   Leia preco_unit
  Escreva "Digite a quantidade:"
  Leia quant
  preço_tot := preço_unit * quant
  Escreva "Preço total: ", preço_tot
 Fim.
```

RESUMO DO TÓPICO 2

Neste tópico, você aprendeu que:

- Todas as tarefas realizadas pelos computadores através de comandos codificados utilizam variáveis para armazenamento e manipulação de dados.

- Cada linguagem de programação estabelece suas próprias regras de formação de nomes de variáveis, e tem definições distintas para os mesmos tipos de dados.

- Constantes têm valores fixos ou estáveis. São declaradas na mesma seção das variáveis. Sua utilidade consiste no fato de que a mesma será declarada uma única vez com valor fixo. Caso seja necessário alterar o valor, esta mudança será feita na sua declaração, não sendo necessário alterar o código de programação que referencia a mesma.

- Uma expressão em um programa é a parte da instrução que produz um valor, normalmente através do uso de operandos (valores) e operadores (indicam a operação a ser realizada).

- Os operadores são meios pelos quais incrementamos, decrementamos, comparamos e avaliamos dados dentro do computador. Temos três tipos de operadores:

  - Operadores são elementos fundamentais que atuam sobre operandos e produzem um determinado resultado. Um operador de atribuição serve para atribuir um valor a uma variável.
  - Os **operadores relacionais** são operadores binários que devolvem os valores lógicos "verdadeiro e falso".
  - Os **operadores lógicos** ou booleanos são usados para combinar expressões relacionais. Também devolvem como resultado valores lógicos verdadeiro ou falso.

- Os operadores que atuam sobre caracteres variam muito de uma linguagem para outra. O operador mais comum e mais usado é o operador que faz a concatenação de strings: toma-se duas strings e acrescenta-se (concatena-se) a segunda ao final da primeira.

- O conceito de expressão em termos computacionais está intimamente ligado ao conceito de expressão ou fórmula matemática, onde um conjunto de variáveis e constantes numéricas relaciona-se por meio de operadores aritméticos compondo uma fórmula que, uma vez avaliada, resulta num valor.

- **Expressões aritméticas:** são aquelas cujo resultado da avaliação é do tipo numérico, seja inteiro ou real. Somente o uso de operadores aritméticos, variáveis numéricas e parênteses é permitido em expressões deste tipo

- **Expressões lógicas:** são aquelas cujo resultado da avaliação é um valor lógico verdadeiro ou falso. Nestas expressões são usados os operadores relacionais e os operadores lógicos, podendo ainda serem combinados com expressões aritméticas. Quando forem combinadas duas ou mais expressões que utilizem operadores relacionais e lógicos, os mesmos devem utilizar os parênteses para indicar a ordem de precedência.

- **Expressões literais:** são aquelas cujo resultado da avaliação é um valor literal (caractere). Neste tipo de expressão só é usado o operador de literais (+).

- As instruções primitivas compõem os comandos básicos na funcionalidade dos computadores, como por exemplo, a entrada e saída dos dados, fazendo a comunicação entre os usuários e a máquina.

- Dispositivos de entrada: usado pelo usuário ou pela memória do computador para transferir informações. Exemplo: teclado, *mouse*, leitor de código de barras etc.

- Dispositivo de saída: forma pela qual o computador transfere informações ao usuário: monitor de vídeo, impressoras etc.

- Sintaxe é a forma como os comandos devem ser escritos, evitando erros de compilação dos programas.

- Semântica é o significado, ou seja, o conjunto de ações que serão exercidas pelo computador durante a execução do referido comando.

AUTOATIVIDADE

Em todas as questões a seguir explique a construção.

1 Construa um algoritmo que use, pelo menos, um operador relacional.

2 Construa um algoritmo que use constante.

3 Construa um algoritmo que use, pelo menos, duas variáveis de tipos de dados distintos.

TÓPICO 3

ESTRUTURAS DE CONTROLE DO FLUXO DE EXECUÇÃO

1 INTRODUÇÃO

Num processo geral de execução de um algoritmo implementado em uma linguagem de programação, a execução começa na primeira linha e vai avançando sequencialmente, executando o código linha após linha até chegar ao final. Entretanto, frequentemente surge a necessidade de colocar instruções dentro de um programa que só serão executadas caso alguma condição específica aconteça. Para esta finalidade a maioria das linguagens possui estruturas de condição para realizar esta tarefa. Neste tópico, são apresentados o seu funcionamento e suas peculiaridades.

2 ESTRUTURA CONDICIONAL

Considere que precisamos desenvolver um algoritmo que classifique uma determinada pessoa entre maior de idade ou menor de idade. Para esse problema sabemos que precisamos avaliar a idade da pessoa, e que se essa idade for maior (ou igual) a 18 anos, a pessoa é considerada maior de idade. Neste caso, para um intervalo de valores da idade, o algoritmo executa um conjunto de ações e para outro intervalo, executa um outro conjunto de ações. Neste tipo de situação, onde um determinado valor é avaliado para a partir do resultado dessa avaliação executar alguma ação, utilizamos as estruturas de condição.

FONTE: Disponível em: <https://www.emersonbarros.com.br/wp.../algoritmos-e-fluxogramas.doc>. Acesso em: 3 out. 2016.

2.1 ESTRUTURA DE CONDIÇÃO SIMPLES: SE-ENTÃO

Esta é a estrutura de condição mais simples e também a mais utilizada. Sua representação ocorre da seguinte forma:

```
Se <expressão logica> entao
Bloco de comandos;
fim-se
```

A expressão lógica sempre vai retornar um valor verdadeiro ou falso. Em caso de retorno falso, o bloco de comando em execução será ignorado e a sequência de execução segue a estrutura seguinte da condição. Exemplos de condições:

. 15 > 61, retorno FALSO

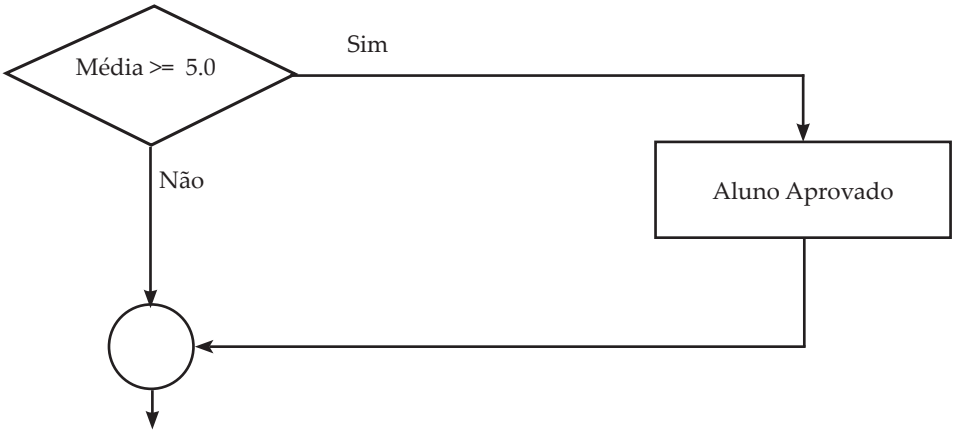
. 100 = 100, retorno VERDADEIRO

O bloco de comandos é uma sequência lógica que será executada somente se o retorno da condição imposta for verdadeiro.

Imagine a situação de aprovação de um aluno, considerando a média de aprovação = 7. Esta é uma situação condicional para a aprovação do mesmo e pode ser representada pela estrutura de condição simples, conforme demonstrado a seguir:

SE Media **>=** 5.0 **ENTÃO** "aluno Aprovado"

FIGURA 12 – REPRESENTAÇÃO DA CONDIÇÃO



FONTE: A autora

A estrutura de condição simples representa uma única regra: ou ela é verdadeira, ou é falsa, ou seja, o bloco de comandos pode ser executado ou não, de acordo com a condição de validação da regra.

2.2 ESTRUTURA DE CONDIÇÃO COMPOSTA: SE-ENTÃO-SENÃO

A estrutura de condição composta prevê o encadeamento de regras. A estrutura de condição se-então-**senão** oferece a possibilidade de executarmos uma determinada ação ou comando se o resultado da expressão lógica for verdadeiro e

de executarmos uma ação diferente se o resultado da expressão lógica for falso. A sintaxe se apresenta da seguinte forma:

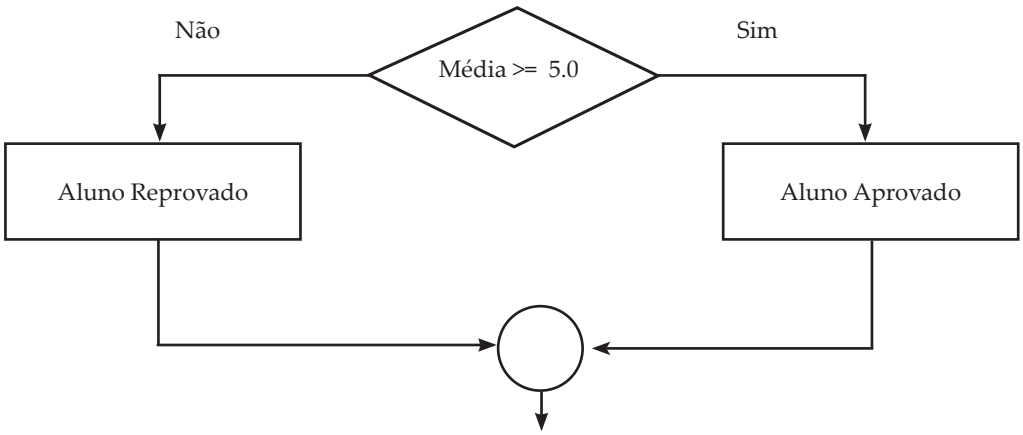
se <expressão-lógica> então:
<bloco de comandos verdade>
<verdade>
senão:
<bloco de comandos falsidade>
fim-se;

Considerando o exemplo da média do aluno, a solução seria construída conforme exposto a seguir.

Em algoritmo ficaria assim:

SE Média >= 5.0 **ENTÃO**
"aluno Aprovado"
SENÃO
"aluno Reprovado"
FimSe

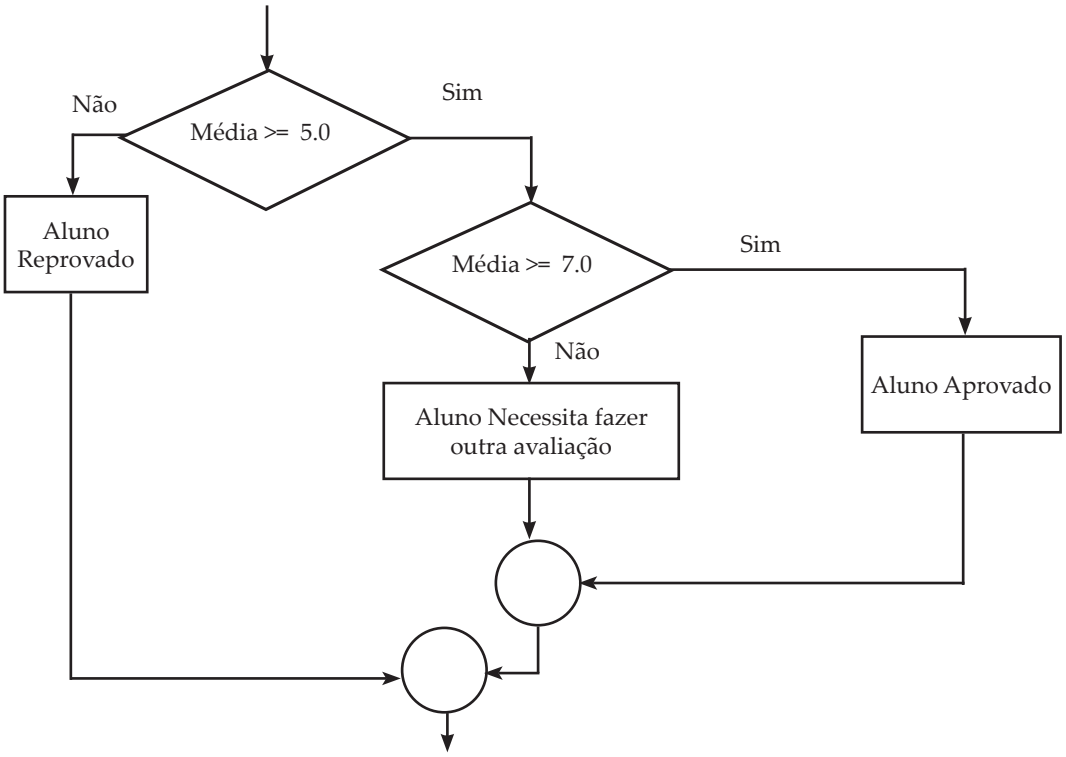
FIGURA 13 – REPRESENTAÇÃO SIMPLIFICADA



FONTE: A autora

No exemplo acima está sendo executada uma condição que, se for verdadeira, executa o comando "APROVADO". Se a mesma não for verdadeira, há um tratamento para a REPROVAÇÃO. Outras condições/regras podem ser adicionadas conforme a necessidade da situação.

FIGURA 14 – REPRESENTAÇÃO DA ESTRUTURA DE CONDIÇÃO COMPOSTA



FONTE: A autora

2.2.1 Expressões lógicas compostas em estruturas de condição

É possível compor expressões lógicas usando operadores relacionais como <; >; <>; = e operadores lógicos como ^ (conjunção), _ (disjunção) e : (negação). Nesse sentido, a expressão lógica que será avaliada na estrutura de condição se-entao também pode ser formada por uma expressão lógica composta.

Imaginando a situação de aprovação do aluno, temos que considerar nota e frequência. Sendo necessário nota >= 7 e frequência < 75%. Ambas devem ser atendidas para que o aluno seja aprovado. Esta situação representa uma conjunção lógica representada pelo operador e (^). Abaixo o exemplo:

```
Se nota >= 7 e (^) frequencia > 0,75 entao
Escreva ('O aluno está aprovado);
Senão
Escreva ('O aluno está reprovado);
Fim se;
```

2.2.2 Estruturas de condições encadeadas

Estas estruturas apresentam um encadeamento de condições que forma praticamente uma árvore ramificada, que cada caminho representa um conjunto de ações. Neste caso, várias estruturas se-então-senão são embutidas umas dentro das outras, conceitualmente entendidas como **ninhos**.

Sintaxe

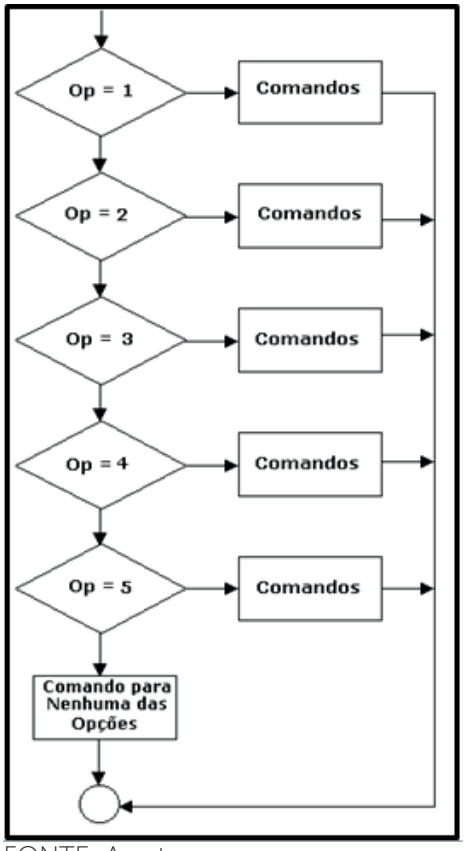
```
Se cond1 então
  Se condicao_a entao
 Resultado1;
  Senão
 Resultado2;
  Fim-se;
Senão
 Se cond2 entao
 Se condicao_c entao
  Resultado3;
 Senão
  Resultado4;
  Fim-se;
Fim-se;
```

A regra básica de execução de uma estrutura de decisão encadeada consiste no fato de que a estrutura de condição mais interna só será checada se a condição da estrutura mais externa apresente retorno VERDADEIRO.

2.3 ESTRUTURA DE CONDIÇÃO CASO SEJA

A estrutura condicional **<caso seja>** também permite trabalhar com regras condicionadas a um resultado ou valor. Nessa estrutura o valor de uma determinada variável é avaliado e caso esse valor coincida com determinado valor preestabelecido, um determinado comando é executado. A estrutura de condição **caso** é executada conforme demonstra a figura a seguir.

FIGURA 15 – ESTRUTURA CASO-SEJA



FONTE: A autora

Exemplo:

```
Caso variável seja A:
  EXECUTE COMANDO A;
Caso variável seja B:
  EXECUTE COMANDO B;
Caso variável seja C:
  EXECUTE COMANDO C;
```

A estrutura de condição caso oferece a opção padrão para ser executada nas situações em que nenhuma das condições anteriores seja atendida. No exemplo proposto a seguir, a variável n do tipo inteiro é testada, e caso tenha valor 1 é escrito na tela "um", caso tenha valor 2 é escrito na tela "dois" e caso não tenha nenhum desses valores será escrito na tela "outro valor".

```
Exemplo de Algoritmo
Algoritmo caso seja
  N numérico;
  Escreva ('Digite o valore de N');
  Leia (N);
Caso N seja
```

```
1:
 Escreva (“Você escolheu 1”)
2:
 Escreva (“Você escolheu 2”)
                Padrão:
                 Escreva(“Outro valor”)
```

3 ESTRUTURAS DE REPETIÇÃO

Uma das grandes vantagens em solucionar problemas computacionalmente se resume ao fato de que eles têm uma grande capacidade de repetir o processamento de soluções, assim como o conjunto de operações envolvidas nas mesmas para grandes quantidades de dados.

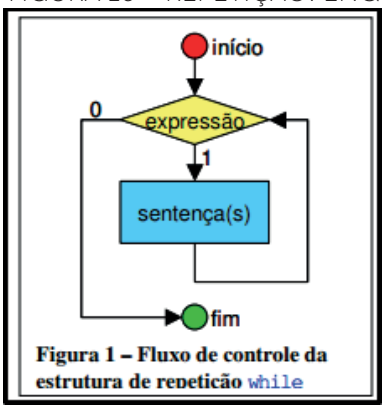
Nesse sentido, o conjunto de procedimentos é repetido até que o objetivo seja alcançado. Por causa disso, torna-se obrigatória a utilização de uma **CONDIÇÃO DE CONTROLE**, construída através de expressões lógicas que são testadas em cada ciclo afim de determinar se a sua execução prossegue ou não.

As estruturas de repetição são basicamente três: enquanto-faça, faça-enquanto e para-faça. A diferença básica é que enquanto-faça primeiro testa a condição para depois realizar o bloco de comando, e o faça-enquanto primeiro executa o bloco para depois realizar o teste. A estrutura para-faça usa um mecanismo de controle para determinar quando o laço de repetição do comando deverá ser terminado.

3.1 TESTE NO INÍCIO: ENQUANTO-FAÇA

No algoritmo 15 é apresentado o formato básico da estrutura de repetição **enquanto-faça**. Antes de entrar na estrutura de repetição, uma expressão lógica é avaliada, e caso o resultado for verdadeiro, os comandos que estão dentro da estrutura serão executados. Após a execução dos comandos, a expressão lógica é novamente avaliada. Caso o resultado da expressão lógica for falso, o algoritmo sai da estrutura de repetição e segue para a próxima linha.

FIGURA 16 – REPETIÇÃO: ENQUANTO-FAÇA



FONTE: Disponível em: <http://olimpiada.ic.unicamp.br/extras/cursoC/Cap06-RepeticaoControle-texto.pdf>. Acesso em: 8 nov. 2016.

FIGURA 17 – ALGORITMO ENQUANTO-FAÇA

```
Escreva os números na tela de 0 a 10:


algoritmo "contador"
var
   contador:inteiro
inicio
// Seção de Comandos

   enquanto contador <=10 faca
               escreval(contador)
               contador <- contador + 1
   fimenquanto
fimalgoritmo
```

FONTE: Disponível em: <http://www.ouropreto.ifmg.edu.br/lp/slides-aulas/08-estrutura-de-repeticao-enquanto-faca>. Acesso em: 8 nov. 2016.

Exemplo para o desenvolvimento de um algoritmo que lê duas notas de um aluno, calcula sua média e indica se foi aprovado ou reprovado. O algoritmo deverá ser executado até que o usuário diga que não temos mais alunos para avaliar.

FIGURA 18 – ALGORITMO NOTAS

```
algoritmo "notas"
var
   nota1, nota2, media: real
   aluno : caractere
   resposta : caractere
Inicio
      resposta <- "S"
      enquanto (resposta = "S") faca
         escreval("Digite o nome do aluno")
         leia(aluno)
         escreval("Digite a nota 1 ")
         leia(nota1)
         escreval("Digite a nota 2 ")
         leia(nota2)
         media <-(nota1 + nota2)/2
         se (media >=60) entao
            escreval("Aluno aprovado")
         senao
              escreval("Aluno reprovado")
         fimse
         escreval("Deseja a informar os dados de outro aluno? S ou N")
         leia(resposta)
      fimenquanto
fimalgoritmo
```

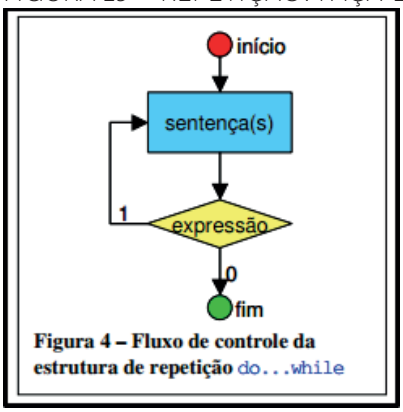
FONTE: Disponível em: <http://www.ouropreto.ifmg.edu.br/lp/slides-aulas/08-estrutura-de-repeticao-enquanto-faca>. Acesso em: 8 nov. 2016.

De maneira geral, o mecanismo que altera o valor da expressão lógica que controla o laço está embutido dentro do bloco de comandos ou depende de alguma variável externa que será fornecida em tempo de execução. A estrutura enquanto-faça é usada, principalmente, quando não se sabe com antecedência a quantidade de repetições que precisam ser realizadas.

3.2 TESTE NO FIM: FAÇA-ENQUANTO

A estrutura **faça-enquanto** difere da estrutura **enquanto-faça** somente por executar o bloco de comando antes de testar se a condição é verdadeira, ou seja, o teste da condição é realizado apenas ao final da estrutura. Assim, utilizando o **faça-enquanto** o bloco de comandos será sempre executado pelo menos uma vez, mesmo que a expressão de controle seja falsa.

FIGURA 19 – REPETIÇÃO: FAÇA-ENQUANTO

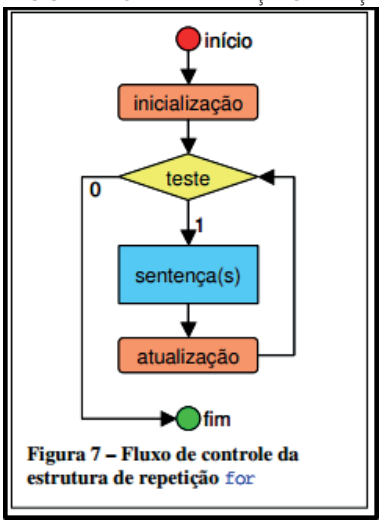


FONTE: Disponível em: <http://olimpiada.ic.unicamp.br/extras/cursoC/Cap06-RepeticaoControle-texto.pdf>. Acesso em: 8 nov. 2016.

3.3 REPETIÇÃO COM CONTROLE: FAÇA-PARA

Esta estrutura define antecipadamente quantas vezes o bloco da programação será executado.

FIGURA 20 – REPETIÇÃO: FAÇA-PARA



FONTE: Disponível em: <http://olimpiada.ic.unicamp.br/extras/cursoC/Cap06-RepeticaoControle-texto.pdf>. Acesso em: 8 nov. 2016.

Exemplo de sintaxe:

```
Algoritmo Estrutura de repetição
Para <variavel inicio> até <fim incremento> faça
<bloco de comandos>
Fim para;
```

Exemplo de algoritmo

```
Algoritmo repetição
I  number;
Para i de 1 até 100 faça
 Escreva ('Seja otimista e vença na vida');
Fim para;
```

O resultado será a exibição de 100 linhas com a frase: Seja otimista e vença na vida

4 CONTADORES E ACUMULADORES

Existem situações de controle específicas dentro dos algoritmos. Podemos percebê-las sempre que for necessário realizar a contagem de ocorrências ou somatórios, dentro de uma repetição do código. Os contadores são as variáveis que irão receber o acúmulo da contagem das informações e os acumuladores realizam o acúmulo do somatório de determinado valor.

4.1 CONTADORES

Os **contadores** são normalmente inicializados com valor 0 (zero) e incrementados em 1 (um) a cada vez que uma nova ocorrência (ou situação) é processada.

Imagine uma situação de uma zona eleitoral. Nesta zona existem 323 votantes, e desejamos saber quantas pessoas são **do sexo feminino e tem mais de 35 anos**. Nesta situação específica, precisaremos de variáveis para armazenar a quantidade de pessoas com estas características. Neste caso, usaremos um contador que incrementará as quantidades sempre que a situação for verdadeira.

Exemplo:

```
Algoritmos contadores
nome  texto;
idade  texto;
sexo  texto;
```

```
aux_tot numerico := 0;
 Inicio
Para i de 1 até 323 faça
 Escreva (‘digite o nome da pessoa’);
 Escreva (‘digite o sexo da pessoa’);
 Escreva (‘digite a idade da pessoa’);
 Leia (nome);
 Leia(sexo);
 Leia(idade);
 Se sexo = ‘F’ e idade > 35 entao
   aux_tot := aux_tot +1;
 fim-se;
 fim para;
  escreva (‘O total de pessoas do sexo feminino com idade superior a 35 anos é: ‘||aux_tot);
fim;
```

4.2 ACUMULADORES

Como comentado anteriormente, os **acumuladores** são utilizados em situações em que é necessário acumular uma soma. No caso dos **somatórios**, o acumulador é normalmente inicializado com o valor 0 e incrementado no valor de um outro termo qualquer, dependendo do problema em questão.

Imagine uma empresa que tem 18 funcionários e precisa saber o valor de sua folha de pagamento. Nesta situação, precisamos de um algoritmo que leia o nome e o salário de cada funcionário e acumule o valor total lido.

Exemplo:

```
Algoritmo Acumulador
Nome  texto;
Salario numerico;
Soma_salario numerico := 0;
 Inicio
   Para i de 1 até 18 faça
  Escreva (‘digite o nome da pessoa’);
  Escreva (‘digite o salario da pessoa’);
  Leia (nome);
  Leia(salario);
  Soma_salario := soma_salario + salario;
  Fim para;
   Escreva (‘ A soma de todos os salarios é:’||soma_salario);
 Fim;
```

LEITURA COMPLEMENTAR

HISTÓRIA DA LÓGICA DE PROGRAMAÇÃO

Felipe Falcão

A lógica da programação é, sem dúvida, a primeira coisa que deve ser estudada se você quiser se tornar um bom programador, seja qual for a linguagem. Por que ela é tão importante? Imagine que agora você é um soldado que recebe ordens e as coloca em prática. O seu capitão lhe diz: "Soldado, sua missão é encontrar a arma principal do inimigo e sabotá-la, como também fazer o reconhecimento do local. Assim que você se infiltrar, vai procurar colher o máximo possível de informações. No entanto, antes das 14h deve sair do local. Se observar alguma movimentação suspeita, saia imediatamente. E lembre-se, sua missão principal é encontrar a arma principal do inimigo e sabotá-la". Nessa missão, o soldado precisa sempre se lembrar das ordens do capitão, e para que elas sejam bem-sucedidas é necessário aplicar todas elas.

Vamos analisar essa missão:

Objetivo principal: Encontrar e sabotar a arma principal do inimigo.

Objetivo secundário: Colher o máximo de informações possíveis.

Condições: O soldado só vai terminar a missão com êxito se: Não houver movimentação suspeita que faça com que ele saia do local antes do horário, e caso isso não ocorra, ele deve sair de lá sem falta às 14 horas.

Quando vamos programar, devemos ser como o comandante. Primeiro, devemos saber qual o objetivo principal e daí programar as suas condições para que esse objetivo seja atingido. Veja o caso dessa tirinha:

*Claro que o marido levou o que a mulher falou ao pé da letra. Sabemos que no dia a dia ninguém leva tudo ao pé da letra. No entanto, essa tirinha não representa o cotidiano. Ela representa o modo de pensar do programar. Notou? A condição, por assim dizer, não foi bem "programada" pela esposa, levando o marido a trazer nove ovos, ao invés de seis ovos e nove batatas, que era o que a esposa gostaria que ele trouxesse. Ela deveria ter dito assim: Me traga seis ovos. Se tiver batatas, me traga nove batatas.

A lógica da programação é essencial para um bom código. Mas como tudo isso começou? Vamos, a partir de agora, viajar pelo tempo e entender um pouco sobre a história da lógica da programação.

Como tudo começou

A lógica da programação é um assunto muito grande e complexo. Assim, vamos dar apenas uma introdução à história da lógica de programação. A lógica de programação é quando se pretende realizar alguma função ou um esquema lógico por meio de parâmetros e metas. Existe uma associação direta da Lógica de Programação com o Raciocínio Matemático, onde o importante é a interpretação de um problema e a utilização correta de uma fórmula. De fato, não existem "fórmulas" em informática, o que existe é nosso modo de pensar em como resolver e extrair o máximo de informações de um problema, de maneira eficaz e eficiente sobre um ângulo de visão.

Essa solução precisa ser exteriorizada e expressa numa linguagem conhecida. A lógica da programação entra nesse ponto, para desenvolvermos soluções e algoritmos para apresentar essa solução ao mundo. A primeira pessoa a pensar em usar lógica matemática para a programação foi John McCarthy, ele propôs usar programas para manipular com sentenças instrumentais comuns apropriadas à linguagem formal, ou seja, o programa básico formará conclusões imediatas a partir de uma lista de premissas. Essas conclusões serão tanto sentenças declarativas quanto imperativas. Quando uma sentença imperativa é deduzida, o programa toma uma ação correspondente.

A primeira linguagem de alto nível do mundo foi a Plankalkül, criada pelo cientista alemão Konrad Zuse, entre os anos 1942-1946 no desenvolvimento dos primeiros computadores. Ela é considerada de alto nível porque em termos simples, ela é mais "humana" e está longe do código de máquina, se aproximando mais da linguagem humana. Vamos entender melhor o que é uma linguagem de alto nível e a diferença entra ela e a máquina.

Quando falamos de linguagem mais humana, é justamente pelo fato de a entendermos melhor, de ser humanamente mais fácil de se compreender. Você facilmente entenderia, com um pouco de estudo, esse código, por exemplo:

```
program HelloWorld;
begin
  writeln('HelloWorld');
end.
```

Uma pessoa que não entende nada de programação, mas sabe inglês, entenderia algumas palavras da linguagem e poderia facilmente interpretar alguma coisa desse código em PASCAL. "Program" é programa em inglês, "begin" é começar, "write" é escrever e end é fim. Mesmo sem saber programar, as pessoas que sabem um pouco de inglês chegariam a mais ou menos a essa conclusão: "É um programa e vai começar a escrever alguma coisa e depois terminar". E é justamente isso! Agora vamos ver um código de máquina. Primeiramente um código de máquina não pode ser aberto em um editor de texto normal. Caso você abra aparece apenas um texto sem significado, devido aos caracteres de controle, como esse, por exemplo:

```
MZÀ□$Pÿv□èŠÿ]Ë3ÀP¸□F□
ë□ƒF□□¸□< uè2Àëä□Àt□Bª
```

Para se ver um código de máquina é necessário um editor de hexadecimal. Assim vemos:

```
0E3D:0000 CD 20 FF 9F 00 9A F0 FE-1D F0 4F 03 F0 07 8A 03 . ……..O…..
0E3D:0010 F0 07 17 03 F0 07 DF 07-01 01 01 00 02 FF FF FF …….
0E3D:0020 FF FF FF FF FF FF FF FF-FF FF FF FF BD 0D 4C 01 ……..L.
0E3D:0030 D0 0C 14 00 18 00 3D 0E-FF FF FF FF 00 00 00 00 ……=……
0E3D:0040 05 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 …….
```

Consegue compreender esse código? E agora, compreende a diferença entre linguagem de alto nível e a linguagem de máquina? Voltando ao assunto, a linguagem Plankalkül foi só publicada em 1972 e seu compilador em 1998. Depois surgiram inúmeras outras, como o Prolog.

A linguagem Prolog foi desenvolvida em 1972 por Alain Colmerauer e foi apresentada como baseada em lógica matemática. Ela é uma simplificação do Planner, a qual permitia a invocação orientada a padrões de planos procedimentais de asserções e de objetivos. A primeira implementação do Prolog foi a Marseille Prolog, desenvolvida em 1972. O uso do Prolog como uma linguagem de programação prática teve seu ápice com o desenvolvimento de um compilador por David Warren em Edinburgo, em 1977.

A partir do Planner, foram desenvolvidas as linguagens de programação QA-4, Popler, Conniver e QLISP. As linguagens de programação Mercury, Visual Prolog, Oz e Frill, foram desenvolvidas a partir do Prolog. Outra linguagem bastante usada é o Pascal.

O professor Niklaus Wirth, vendo a necessidade de implementar as ideias de maneira mais fácil, junto com seus colegas da Universidade Técnica de Zurique (Suíça) desenvolveram, no início dos anos 70, a linguagem PASCAL – uma derivação da linguagem ALGOL 60, porém, de implementação mais simples e com uma estrutura de dados mais poderosa. O nome Pascal foi uma homenagem a Blaise Pascal, famoso matemático, que criou a calculadora baseada em discos

de madeira, que foi a predecessora da calculadora de mesa e serviu de inspiração para diversos computadores. Essa linguagem é bastante utilizada para ensinar programação aos novatos, pelo fato de ser bem fácil de ser entendida.

Hoje existem muitas linguagens fáceis de serem entendidas e estudadas, pois a melhor maneira de aprender a lógica da programação é por meio da prática! Assim, incentivamos a todos os interessados a continuar acompanhando as aulas que estamos colocando aqui no Guia do PC.

A seguir você pode ver uma lista com as principais linguagens de programação:

- PLANKALKÜL – 1945 – Criada por Konrad Zuse; foi a primeira linguagem de programação de alto nível do mundo.
- FORTRAN – 1954 – Criada pela IBM 704; usada para computadores com pouca capacidade de processamento; não tinha metodologia e ferramentas para programação.
- FORTRAN I – 1957 – Primeira versão implementada do FORTRAN; Nomes podiam ter até 6 caracteres; Programas maiores que 400 linhas raramente compilavam corretamente, principalmente devido à baixa confiabilidade do 704.
- FORTRAN II – 1958 – 50% do código escrito para o IBM 704 era FORTRAN.
- FORTRAN IV – 1960- 62 – Declaração de tipos explícita; Comando de seleção lógica; Nomes de subprogramas podiam ser parâmetros; Padrão ANSI em 1966.
- FORTRAN 77 – 1978 – Manuseio de character string; Comando de controle de *loop* lógico; Comando IF- THEN- ELSE.
- FORTRAN 90 – 1990 – Funções built – in para operações com arrays; Arrays dinâmicos; Ponteiros; Tipo registro; Recursão, dentre outros.
- LISP – 1959 – Somente dois tipos de dados: átomos e listas; Pioneira da programação funcional
- ALGOL 58 – 1958 – Nomes podiam ter qualquer tamanho; Comandos de composição de blocos (begin …end); F com cláusula ELSE- IF.
- ALGOL 60 – 1960 – Modificou o ALGOL 58 durante um encontro de 6 dias em Paris.
- ALGOL 68 – 1968.
- SIMULA 67 – 1967 – Baseada no SIMULA I, criada por Kristen Nygaard e OleJohan Dahal, entre 1962 e 1964 na Noruega. Projetada inicialmente para simulação de sistemas. Baseada no ALGOL 60 e SIMULA I.
- COBOL – 1960 – Deve parecer com inglês simples; deve ser fácil de usar, mesmo que signifique menos recursos.
- PL/ I – 1965 – Projetado pela IBM e SHARE; incluía o que era considerado o melhor do: ALGOL 60.
- Pascal – 1971 – Simplicidade e tamanho reduzido eram objetivos de projeto. Projetada para o ensino de programação estruturada. Ainda é a uma das LPs mais usadas para o ensino de programação estruturada nas universidades.
- BASIC – 1964 – Criada por Kemeny e Kurtz em Dartmouth.
- Modula 2 – 1975.
- Modula 3 – Final dos anos 80.
- Delphi – 1995 – Pascal mais características de suporte a POO; Mais elegante e seguro que C+ +; Baseada em Object Pascal, projetada anos antes.

- C – 1972 – Projetada para a programação de sistemas nos Lab.
- Prolog – 1972.
- Ada – 1983.
- Ada 95 – 1988 – Facilidades para interface gráfica.
- Smalltalk – 1972- 1980 – Primeira implementação completa de uma linguagem orientada por objetos (abstração de dados, herança e vinculação de tipos dinâmica). Pioneira na interface gráfica baseada em janelas.
- C++ – 1985 – Desenvolvida nos Lab. Bell por Stroustrup. Baseada no C e SIMULA 67
- Java – 1995 – Desenvolvida pela Sun no começo dos anos 90. Para POO, baseada em C+ +, mas significativamente simplificada.

Outras linguagens:

- ML (1973).
- Clipper (1984).
- SQL.
- Perl (1987).
- PYTHON (1991).
- PHP (1995).
- ASP.
- JavaScript (1995).
- HTML.
- XML.
- TeX/ LaTeX (1985) /TeX/ LaTeX (1985).

FONTE: Disponível em: <http://www.guiadopc.com.br/noticias/34848/historia-logica-programacao.html>. Acesso em: 2 ago. 2013.

RESUMO DO TÓPICO 3

Neste tópico, você aprendeu que:

- Algoritmos executam verticalmente: da primeira à última linha de programação, tendo as estruturas de condição como componentes da lógica de construção controles específicos.

- **Estrutura de condição simples**: se-então: é a estrutura de condição mais simples e também a mais utilizada. Sua representação ocorre da seguinte forma:

Se <expressão logica> então
Bloco de comandos;
Fim-se;

- **Estrutura de condição composta**: prevê o encadeamento de regras. A estrutura de condição se-então oferece a possibilidade de executarmos uma determinada ação ou comando se o resultado da expressão lógica for verdadeiro e de executarmos uma ação diferente se o resultado da expressão lógica for falso. A sintaxe se apresenta da seguinte forma:

Se <exp1> então
Bloco de comandos1;
senão:
<bloco de comandos falsidade>
fim-se;

- **Estruturas de condições encadeadas**: estas estruturas apresentam um encadeamento de condições que forma praticamente uma árvore ramificada, onde cada caminho representa um conjunto de ações. Neste caso, várias estruturas se-então-senão são embutidas umas dentro das outras, conceitualmente entendidas como ninhos.

Se cond1 então
Se condicao_a entao
Resultado1;
Senão
Resultado2;
Fim-se;
Senão
Se cond2 entao
Se condicao_c entao
Resultado3;
Senão

Resultado4;
Fim-se;
Fim-se;

- **Estrutura de Condição caso seja**: A estrutura condicional <caso seja> também permite trabalhar com regras condicionadas a um resultado ou valor. Nessa estrutura, o valor de uma determinada variável é avaliado, e caso esse valor coincida com determinado valor preestabelecido, um determinado comando é executado.

  Exemplo:

```
Caso variavel seja A:
  EXECUTE COMANDO A;
Caso variavel seja B:
  EXECUTE COMANDO B;
Caso variavel seja C:
  EXECUTE COMANDO C;
```

- **Estruturas de repetição**: Uma das grandes vantagens em solucionar problemas computacionalmente se resume ao fato de que eles têm uma grande capacidade de repetir o processamento de soluções, bem como o conjunto de operações envolvidas, para grandes quantidades de dados.

- **Teste no início**: enquanto-faça: antes de entrar na estrutura de repetição, uma expressão lógica é avaliada, e caso o resultado da mesma for verdadeiro, os comandos que estão dentro da estrutura serão executados. Após a execução dos comandos, a expressão lógica é novamente avaliada.

- **Teste no fim**: faça-enquanto: a estrutura faça-enquanto difere da estrutura enquanto-faça somente por executar o bloco de comando antes de testar se a condição é verdadeira, ou seja, o teste da condição é realizado apenas ao final da estrutura. Assim, utilizando o faça-enquanto, o bloco de comandos será sempre executado pelo menos uma vez, mesmo que a expressão de controle seja falsa.

- **Repetição com controle**: faça-para: esta estrutura define antecipadamente quantas vezes o bloco da programação será executado.

- Os **contadores** são normalmente inicializados com valor 0 (zero) e incrementados em 1 (um) a cada vez que uma nova ocorrência (ou situação) é observada.

- Os **acumuladores** são utilizados em situações em que é necessário acumular uma soma. No caso dos **somatórios**, o acumulador é normalmente inicializado com o valor 0 e incrementado no valor de um outro termo qualquer, dependendo do problema em questão.

AUTOATIVIDADE

1 Crie um algoritmo que use estruturas de repetição. Descreva a situação em forma de texto e fluxograma.

2 Qual é a diferença de acumuladores e contadores. Exemplifique o uso de cada um.

3 Crie um algoritmo que necessite de estrutura de condição encadeada para a solução.

UNIDADE 2

TABELA VERDADE, MODULARIZAÇÃO, CONSISTÊNCIA DE DADOS, VETORES E MATRIZES

OBJETIVOS DE APRENDIZAGEM

A partir desta unidade você será capaz de:

- entender a lógica matemática, ou lógica simbólica, que trata do estudo das proposições;
- entender a construção da tabela verdade;
- avaliar a importância da modularização, ou seja, a construção de algoritmos através dos recursos conhecidos como procedures e funções;
- reconhecer a adequada entrada de dados em algoritmos;
- construir algoritmos usando vetores e matrizes.

PLANO DE ESTUDOS

Esta unidade está dividida em três tópicos. Ao final de cada um deles você encontrará atividades que auxiliarão no seu aprendizado.

TÓPICO 1 – TABELA VERDADE

TÓPICO 2 – CONSISTÊNCIA DOS DADOS DE ENTRADA

TÓPICO 3 – VETORES E MATRIZES

TÓPICO 1

TABELA VERDADE

1 INTRODUÇÃO

Neste tópico, você vai aprender a construir a tabela verdade, através da lógica proposicional, que pode ser considerada como todo conjunto de palavras ou símbolos que exprimem uma ideia de sentido completo.

2 LÓGICA PROPOSICIONAL

A tabela verdade consiste em um tipo de tabela matemática usada em lógica para determinar a validade de uma fórmula. O trabalho precursor remete ao ano de 1922, com autoria de Gottlob Frege e Charles Peirce. As tabelas verdade incorporaram a apresentação atual através do trabalho desenvolvido por Emil Post e Ludwig Wittgenstein. A tabela verdade trabalha com as seguintes operações do cálculo proposicional:

- Negação (~)
- Conjunção (^)
- Disjunção (v)
- Condição < se....então>
- Bicondicional (se e somente se) que é a equivalência
- Disjunção exclusiva (XOR)
- Adaga de Quine (NOR)

De acordo com Forbellone e Eberspächer (2005, p. 23), "tabela verdade é o conjunto de todas as possibilidades combinatórias entre os valores de diversas variáveis lógicas, as quais se encontram em apenas duas situações (V ou F) e em um conjunto de operadores lógicos".

2.1 CONCEITOS ASSOCIADOS

O conceito mais básico da lógica está diretamente ligado ao conceito de proposição, que significa submeter à apreciação e requer juízo de avaliação. A proposição é uma sentença que necessita ser declarada através de termos ou sentenças bem objetivas, o que a torna declarativa. O conteúdo das sentenças ou termos sempre será verdadeiro ou falso.

Imagine a seguinte afirmação:

O feijão é preto e o arroz é branco.

Verifique que estamos diante de uma proposição com VALOR LÓGICO VERDADEIRO.

O valor lógico de uma proposição refere-se aos dois possíveis juízos que podemos fazer acerca da afirmação: **verdadeiro (V)** ou **falso (F)**.

Existem outros tipos de sentenças:

- **Sentenças exclamativas**: *"Como você está feliz!"*
- **Sentenças interrogativas**: *"Quantos anos você tem?"*
- **Sentenças imperativas**: *"Você precisa se preparar melhor para a prova"*.

Somente para as sentenças declarativas podemos atribuir os valores de juízo: verdadeiro ou falso (MANZANO, 2000).

Normalmente, as **proposições** são representadas por letras minúsculas (t, v, d, s, r etc.).

Outros exemplos de **proposições**:

p: Simone é professora.
q: 10 > 12508.

A forma de apresentação das sentenças acima, baseadas no raciocínio lógico, é representada da seguinte forma em estrutura de sentença:

VL(**p**) = **V**, indicando que o valor de **p** é **VERDADEIRO**
VL(**q**) = **F**, indicando que o valor de **q** é **FALSO**

Nenhuma proposição pode assumir os dois valores de verdadeiro ou falso ao mesmo tempo. Ou a proposição é verdadeira, ou é falsa. Existem três princípios básicos a que as proposições devem atender:

- ***Princípio da Identidade****: a proposição é verdadeira ou falsa.*
- ***Não contradição****: a proposição não poderá ser verdadeira ou falsa ao mesmo tempo.*
- **Terceiro excluído**: não existem outros valores de juízo: somente verdadeiro e falso.

Outra característica importante das preposições é que elas podem ser simples ou compostas:

Todas as pessoas envelhecem – SIMPLES.
Simone é Professora e Pedro é aluno – COMPOSTA.

Existem ainda os conectivos lógicos ligando às sentenças:

Se *chover amanhã,* ***então*** *não irei à escola.*
Vou ganhar um celular ***se e somente se*** *eu gabaritar na prova.*

2.2 TIPOS DE CONECTIVOS LÓGICOS

Os **conectivos lógicos** são expressões cuja finalidade é ligar duas ou mais proposições. Eles estão presentes nas proposições compostas (MANZANO, 2000).

Duas checagens são importantes para determinar se as proposições compostas são verdadeiras ou falsas:

- O valor das proposições que compõem as sentenças.
- O tipo de conectivo que liga as proposições de uma mesma sentença.

2.2.1 Conjunção – conectivo e

Este conectivo representa a conjunção. Também pode ser representado pelo símbolo (^). Portanto, a proposição – "Simone é professora **e** Pedro é aluno" – pode ser representada da seguinte forma:

: **p∧q.** onde: p = Simone é Professora e q = Pedro é aluno.

E como definimos o valor lógico de uma conjunção? É bem simples: uma conjunção somente será verdadeira se todas as suas proposições componentes forem verdadeiras.

Em relação à proposição exemplo, ela só será verdadeira se Simone for realmente professora e Pedro realmente for aluno.

E como identificaremos se a conjunção é falsa? Identificando se uma de suas proposições componentes é falsa: Simone não é professora / Pedro não é aluno.

Essas situações podem ser atribuídas a uma tabela para facilitar o entendimento. Essa tabela é a Tabela Verdade.

Vamos iniciar as análises e a construção da tabela.

Proposição da Conjunção: 'Simone é professora **e** Pedro é aluno'.
p = Simone é professora e **q** = Pedro é aluno.

Se ambas forem verdadeiras, a conjunção formada por elas (Simone é professora **e** Pedro é aluno) será também verdadeira. Teremos:

Simone é professora	Pedro é aluno	Simone é professora **e** Pedro é aluno
p	q	**p ∧ q**
V	V	V

Se for verdade apenas que *Simone é professora*, mas falso que *Pedro é aluno*:

Simone é professora	Pedro é aluno	Simone é professora **e** Pedro é aluno
p	**q**	**p ∧ q**
V	**F**	**F**

Se for verdadeiro que *Pedro é aluno*, e falso que *Simone é professora*:

Simone é professora	Pedro é aluno	Simone é professora **e** Pedro é aluno
p	**q**	**p ∧ q**
F	V	**F**

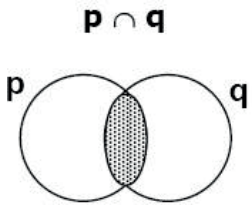
Se as duas proposições forem falsas:

Simone é professora	Pedro é aluno	Simone é professora **e** Pedro é aluno
p	**q**	**p ∧ q**
F	**F**	**F**

Como resultado, temos a tabela verdade, a qual representa todas as situações de combinações de resultados possíveis para a situação proposta:

p	q	**p ∧ q**
V	V	V
V	F	F
F	V	F
F	**F**	F

Se as proposições **p** e **q** forem representadas como conjuntos, por meio de um diagrama, a conjunção "**p e q"** corresponderá à **interseção** do conjunto **p** com o conjunto **q**. Teremos:



2.2.2 Disjunção – conectivo ou

Nesta situação as proposições são unidas pelo conectivo **ou**.

Para entendimento, usaremos o mesmo exemplo da conjunção, apenas trocando o conectivo lógico **e** pelo **ou: "**Simone é professora **ou** Pedro é aluno".

Em relação à disjunção, uma proposição será falsa, se todas as suas componentes forem falsas. E será verdadeira, se uma das proposições for verdadeira, ou se todas elas forem. Resumindo: para ser verdadeira, pelo menos uma das sentenças deve ser verdadeira.

Simone é professora	Pedro é aluno	Simone é professora **ou** Pedro é aluno
p	**q**	**P ∨ Q**
V	V	V

Se for verdade apenas que *Simone é professora*, mas falso que *Pedro é aluno*:

Simone é professora	Pedro é aluno	Simone é professora **ou** Pedro é aluno
p	**q**	**P ∨ Q**
V	**F**	V

Se for verdadeiro que *pedro é aluno*, e falso que *Simone é professora*:

Simone é professora	Pedro é aluno	Simone é professora **ou** Pedro é aluno
p	**q**	**P ∨ Q**
F	V	V

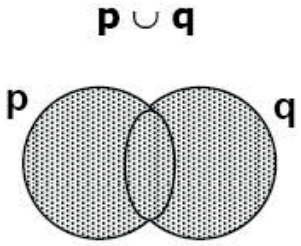
Se as duas proposições forem falsas:

Simone é professora	Pedro é aluno	Simone é professora **ou** Pedro é aluno
p	**q**	**P ∨ Q**
F	**F**	**F**

Como resultado, temos a tabela verdade, a qual representa todas as situações de combinações de resultados possíveis para a situação proposta da disjunção:

p	q	p V q
V	V	V
V	F	V
F	V	V
F	F	F

Se as proposições **p** e **q** forem representadas como conjuntos por meio de um diagrama, a disjunção "**p ou q"** corresponderá à **união** do conjunto **p** com o conjunto **q**,



2.2.3 Disjunção exclusiva – conectivo ou... ou...

Há um terceiro tipo de proposição composta, bem parecido com a disjunção que acabamos de ver, mas com uma pequena diferença. Imagine a sentença a seguir:

Ou irei à praia ou irei ao cinema.

No exemplo da proposição disjuntiva, **ou** eu poderia ir a um lugar ou a outro, poderia ir até aos dois, entretanto, na *disjunção exclusiva* SÓ posso ir ou a um ou a outro, nunca aos dois.

O símbolo que designa a *disjunção exclusiva* é o "**V**". Assim ficará a tabela verdade para a situação proposta:

p	q	p V q
V	V	F
V	F	V
F	V	V
F	F	F

As linhas intermediárias merecem destaque. Conforme afirmado acima, somente poderemos ir a um lugar apenas, não sendo possível ir aos dois lugares ao mesmo tempo.

2.2.4 Se então – conectivo condicional

Estamos agora falando de proposições como as que se seguem:

Se Simone é professora, então Pedro é ortopedista.
Se chover amanhã, então não irei à escola.

Muita gente tem dificuldade em entender o funcionamento desse tipo de proposição. Convém, para facilitar nosso entendimento, que trabalhemos com a seguinte sentença:

***Se** nasci em Santa Catarina, **então** sou catarinense.*

Em relação à sentença proposta, observe que ela será falsa se:

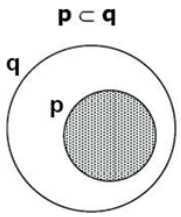
- a **primeira proposição for falsa**: se sou catarinense, obrigatoriamente devo ter nascido em Santa Catarina;
- se nasci em Santa Catarina (**a primeira verdadeira**) e sou paulista (**a segunda é falsa**), logo a preposição toda é falsa.

A primeira parte da condição é suficiente para obtenção de um resultado necessário.

Na proposição "*Se **p**, então **q**"*, a proposição **p** é denominada de antecedente, enquanto a proposição **q** é dita consequente. Teremos como tabela verdade:

p	q	p → q
V	V	V
V	F	F
F	V	V
F	F	V

Se as proposições **p** e **q** forem representadas como conjuntos, por meio de um diagrama, a proposição condicional "**Se p então q"** corresponderá à **inclusão** do conjunto **p** no conjunto **q** (p está contido em q):



2.2.5 Se e somente se – conectivo bicondicional

A estrutura dita *bicondicional* apresenta o conectivo *"se e somente se"*, separando as duas sentenças simples. Trata-se de uma proposição de fácil entendimento. Se alguém disser:

*"Simone fica feliz **se e somente se** os alunos forem aprovados".*

É o mesmo que fazer a conjunção entre as duas proposições condicionais: Simone fica feliz **somente se** os alunos forem aprovados e os alunos são aprovados somente **se** Simone fica feliz.

Outra forma de entendimento: "Se Simone fica feliz, então os alunos são aprovados **e** se os alunos forem aprovados, então Simone fica feliz".

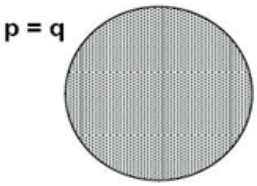
Ambas as colocações apresentam o mesmo sentido para a proposição: *"Simone fica feliz **se e somente se** os alunos forem aprovados".*

A bicondicional liga duas condições. Somente em duas situações será verdadeira: quando a proposição antecedente e a consequente forem verdadeiras, ou quando ambas forem falsas. Nas demais situações será sempre falsa.

Sabendo que a frase *"**p** se e somente se **q**"* é representada por **"p ↔ q"**, então nossa tabela verdade será a seguinte:

p	q	p ↔ q
V	V	V
V	F	F
F	V	F
F	F	V

Se as proposições **p** e **q** forem representadas como conjuntos, por meio de um diagrama, a proposição bicondicional "**p se e somente se q"** corresponderá à **igualdade** dos conjuntos **p** e **q**.



Observação: uma proposição bicondicional "**p se e somente se q"** equivale à proposição composta: "**se p então q e se q então p**", ou seja, "**p ↔ q** " é a mesma coisa que "**(p → q) e (q → p)**"

2.2.6 Partícula não – negação

Estabelece a forma de negação de uma proposição. No caso de proposições uniformes e afirmativas, basta incluir a palavra não (negativa na sentença).

O queijo é barato. **Negativa**: *O queijo não é barato.*

No caso de sentenças cuja afirmativa incluir a palavra não, a negativa ocorre de forma contrária: excluindo-se este termo da proposição.

O queijo não é bom. **Negativa**: *O queijo é bom.*

Geralmente, o símbolo que representa a negação é um sinal de til (~), inserido ao lado esquerdo da proposição. O símbolo (¬) também pode ser utilizado para representar a negação.

A tabela a seguir representa a tabela verdade da negação de uma determinada proposição.

p	~p
V	F
F	V

2.2.7 A negação de uma proposição composta

Imagine uma sentença composta com uma proposição conjuntiva (p e q). Para negar uma proposição conjuntiva composta, podemos adotar algumas regras básicas:

1. Negar a primeira parte (~p);
2. Negar a segunda parte (~q);
3. Trocar e por ou.

Vamos entender melhor:

Simone é professora e Pedro é aluno.

Fazendo a negação teremos:

1. **Nega-se a primeira parte (~p) = Simone não é professora**.
2. **Nega-se a segunda parte (~q) = Pedro não é aluno**.
3. **Troca-se E por OU, e o resultado final será o seguinte**:

SIMONE NÃO É PROFESSORA OU PEDRO NÃO É ALUNO

Logo, em linguagem lógica, podemos representar a regra da seguinte forma:

$$\mathbf{\sim(p \wedge q) = \sim p \ V \sim q}$$

Como chegar neste resultado?

Os passos estão representados pelas etapas propostas a seguir para a construção da tabela verdade da negação de (p ^ q).

Como chegamos a essa conclusão? Ora, por meio da comparação entre as tabelas verdade das duas proposições acima. Vejamos como foi isso. Primeiramente, trabalhemos a tabela verdade do **~(p ∧ q)**.

1) Primeira etapa: representar as sentenças e suas possíveis combinações:

p	q
V	V
V	F
F	V
F	F

2) Incluir a coluna da conjunção apresentando o seu respectivo resultado. Essa é a tabela verdade da proposição. Em caso de dúvida, volte para o início do tópico e revise a regra. Lembre que a conjunção é o **e**.

p	q	p ∧ q
V	V	V
V	F	F
F	V	F
F	F	F

3) Incluir a coluna de negação da tabela verdade, em que o verdadeiro vira falso, e o falso se transforma em verdadeiro:

p	q	p ∧ q	~(p ∧ q)
V	V	V	F
V	F	F	V
F	V	F	V
F	F	F	V

Observe a última coluna. Ela será usada no resultado final da tabela (valor lógico da tabela verdade).

4) Outra forma de encontrar o valor lógico da tabela verdade, para a expressão ~(p ∧ q), é fazer a negativa das sentenças de forma separada. Neste caso teremos:

p	q	~p	~q
V	V	F	F
V	F	F	V
F	V	V	F
F	F	V	V

5) Neste ponto é necessário fazer a negação final. Para ser verdadeira, basta que uma das sentenças componente seja verdadeira.

p	q	~p	~q	~p V ~q
V	V	F	F	F
V	F	F	V	V
F	V	V	F	V
F	F	V	V	V

Compare a última coluna desta tabela com a última coluna do item 2 (em destaque) desta estrutura **(~p ∨ ~q)** com aquela que estava *guardada* da estrutura do item 3. Ambas são a negação do resultado do item 2.

~(p ∧ q)	~p V ~q
F	F
V	V
V	V
V	V

Guarde a regra: Para negar p e q, negaremos p, negaremos q, e trocaremos e por ou. Ou seja, para dizer se uma proposição é, do ponto de vista lógico, equivalente à outra, basta fazer uma comparação entre suas tabelas verdade.

2.2.8 Negação de uma proposição disjuntiva: ~(p ou q)

Os passos para negar uma proposição no formato de disjunção (**p ou q**) segue a mesma lógica da conjunção:

1. Negar a primeira parte (~p);
2. Negar a segunda parte (~q);
3. Trocar OU por E.

Exemplo: Não é verdade que Simone é professora ou Pedro é aluno.

Na prática:

1. Nega-se a primeira parte (~p) = Simone não é professora.
2. Nega-se a segunda parte (~q) = Pedro não é aluno.
3. Troca-se OU por E, e o resultado final será o seguinte.

SIMONE NÃO É PROFESSORA E PEDRO NÃO É ALUNO.

Na linguagem lógica:

$$\sim(p \vee q) = \sim p \wedge \sim q$$

Comprovando o resultado através da tabela verdade:

1) ~(p V q):

p	q
V	V
V	F
F	V
F	F

2) Construir a coluna da disjunção (p ou q):

p	q	p V q
V	V	V
V	F	V
F	V	V
F	F	F

3) Fazer a negação da coluna disjunção:

p	q	p V q	~(p V q)
V	V	V	F
V	F	V	F
F	V	V	F
F	F	F	V

4) Guardar o resultado para comparar com o resultado final **~p** ∧**~q**:

p	q
V	V
V	F
F	V
F	F

5) Negar **p** e **q**:

p	q	~p	~q
V	V	F	F
V	F	F	V
F	V	V	F
F	F	V	V

6) Fazer a conjunção **~p** e **~q**:

p	q	~p	~q	~p ∧ ~q
V	V	F	F	F
V	F	F	V	F
F	V	V	F	F
F	F	V	V	V

7) Comparar o resultado do item 9 com o item 4. São exatamente iguais. **(~p ∧ ~q) = ~(p ∨ q)**:

~(p V q)	~p ∧ ~q
F	F
F	F
F	F
V	V

2.2.9 Negação de uma proposição condicional: ~(p → q)

Para negar uma proposição condicional, devemos acompanhar a seguinte regra:

1º) A primeira parte não sofre alterações.
2º) A segunda parte recebe a negação.

Exemplo: *"Se chover amanhã, então não irei para a escola".*

1º) Mantemos a primeira parte: *"Chove" E*
2º) Negamos a segunda parte: *"eu não vou para a escola".*

Resultado final: *"Chove e eu não vou para a escola".*

Na linguagem lógica: **~(p → q) = p ∧ ~q**

Resumo das tabelas abordadas até o momento e sua estrutura lógica

TABELA 1 – RESUMO DAS SITUAÇÕES DE MONTAGEM PARA A TABELA VERDADE

Estrutura Lógica	É verdadeiro quando	É falsa quando
p ∧ q	as duas proposições são verdadeiras	uma ou as duas proposições sejam falsas
p ∨ q	uma ou as duas proposições sejam verdadeiras	as duas proposições sejam falsas
p → q	**p** não seja verdadeira ao mesmo tempo que **q** não seja falsa	**p** é verdadeira e **q** é falsa
p↔q	**p** e **q** tiverem valores lógicos iguais	**p** e **q** tiverem valores lógicos diferentes
~p	**p** é falsa	**p** é verdadeira

Negando as proposições compostas

Negativa de (p e q)	**~p ou ~q**
Negativa de (p ou q)	**~p e ~q**
Negativa de (p → q)	**p e ~q**
Negativa de (p↔q)	**[(p e ~q) ou (q e ~p)]**

FONTE: A autora

3 OUTROS ASPECTOS IMPORTANTES DA TABELA VERDADE

Até o momento montamos a tabela verdade com duas proposições, mas em situações reais podemos ter um número maior de sentenças que podem ser representadas pela sentença a seguir:

Nº linhas da Tabela Verdade = $2^{nº \text{ de proposições}}$

Já vimos que para duas proposições a tabela verdade terá, obrigatoriamente, 4 linhas. Logo, se acrescentarmos uma terceira proposição (**p, q e r)**, a tabela verdade terá $\mathbf{2^3 = 8}$. E assim, por diante.

Tabelas verdade para p e q:

Com duas proposições componentes, a estrutura inicial da tabela verdade será sempre a mesma, conforme já estudamos no início do tópico:

p	q
V	V
V	F
F	V
F	F

As próximas colunas da tabela verdade dependerão do conectivo que liga as duas sentenças.

Você já aprendeu a construir tabelas verdade com **conjunção, disjunção, disjunção exclusiva, condicional e bicondicional**. Veremos agora a construção com três proposições **(p, q, r)**.

Para iniciar a construção da tabela verdade com três proposições, o primeiro passo é definir o número de linhas que a tabela vai ter. Este cálculo será fornecido pela fórmula **2** **Nº de proposições.**

Então temos (**2^3 = 8**) = tabela verdade com 8 linhas. O início será p mesmo da situação com duas proposições, porém, como são 8 linhas, teremos 4 valores **V** e 4 valores **F,** mantendo sempre a estrutura inicial.

p	q	r
V	V	V
V	V	F
V	F	V
V	F	F
F	V	V
F	V	F
F	F	V
F	F	F

Imagine uma situação onde se deve construir a tabela verdade para a sentença lógica proposta: **P(p, q, r) = (p $\wedge$ ~q) $\rightarrow$ (q v ~r)**.

Traduzindo: *Se **p** e não **q**, então **q** ou não **r**.*

p	q	r
V	V	V
V	V	F
V	F	V
V	F	F
F	V	V
F	V	F
F	F	V
F	F	F

Percebemos que existe uma ordem de precedência. Então, seguiremos os passos a seguir para a montagem da tabela.

1º passo: Negação de q

p	q	r	~q
V	V	V	F
V	V	F	F
V	F	V	V
V	F	F	V
F	V	V	F
F	V	F	F
F	F	V	V
F	F	F	V

2º passo: Conjunção do primeiro parêntese

p	q	r	~q	p ∧ ~q
V	V	V	F	F
V	V	F	F	F
V	F	V	V	V
V	F	F	V	V
F	V	V	F	F
F	V	F	F	F
F	F	V	V	F
F	F	F	V	F

3º passo: Negação de r

p	q	r	~q	p ∧ ~q	~r
V	V	V	F	F	F
V	V	F	F	F	V
V	F	V	V	V	F
V	F	F	V	V	V
F	V	V	F	F	F
F	V	F	F	F	V
F	F	V	V	F	F
F	F	F	V	F	V

4º passo: Disjunção do segundo parêntese

p	q	r	~q	p ∧ ~q	~r	q V ~r
V	V	V	F	F	F	V
V	V	F	F	F	V	V
V	F	V	V	V	F	F
V	F	F	V	V	V	V
F	V	V	F	F	F	V
F	V	F	F	F	V	V
F	F	V	V	F	F	F
F	F	F	V	F	V	V

5º passo: fazer a condicional

RECORDANDO: a condicional só será falsa se tivermos **VERDADEIRO** na primeira parte e **FALSO** na segunda. E, assim, construímos a tabela verdade para três proposições.

p	q	r	~q	p ∧ ~q	~r	q V ~r	(p ∧ ~q) → (q V ~r)
V	V	V	F	F	F	V	V
V	V	F	F	F	V	V	V
V	F	V	V	V	F	F	F
V	F	F	V	V	V	V	V
F	V	V	F	F	F	V	V
F	V	F	F	F	V	V	V
F	F	V	V	F	F	F	V
F	F	F	V	F	V	V	V

3.1 TAUTOLOGIA

Só existe tautologia se o resultado sempre for verdadeiro, independentemente dos valores que compõem as sentenças (MANZANO, 2000).

Para identificar uma tautologia, devemos seguir a regra:

- Construir a tabela verdade da situação.
- Se todos os valores da última coluna forem todos verdadeiros, estaremos diante de uma tautologia.

Exemplo: A proposição **(p ∧ q) → (p V q)** é uma tautologia, pois é sempre verdadeira, independentemente dos valores lógicos de **p** e de **q**, como se pode observar na tabela verdade.

p	q	p ∧ q	p v q	(p ∧ q) → (p V q)
V	V	V	V	V
V	F	F	V	V
F	V	F	V	V
F	F	F	F	V

3.2 CONTRADIÇÃO

Uma proposição composta formada por duas ou mais proposições **p**, **q**, **r**, ... será dita uma **contradição** se ela for **sempre falsa**, independentemente dos valores lógicos das proposições **p**, **q, r** ... que a compõem, ou seja, **construindo a *tabela verdade* de uma proposição composta, se todos os resultados da última coluna forem *FALSOS*, então estaremos diante de uma *contradição***.

Exemplo: A proposição " **p ↔ ~p** " é uma contradição, pois sempre é falsa independentemente do valor lógico de **p**, como é possível observar na tabela verdade a seguir:

p	~p	p ↔ ~p
V	F	F
F	V	F

3.3 CONTINGÊNCIA

Uma proposição composta será dita uma ***contingência*** sempre que não for uma *tautologia* e uma *contradição*. Somente isso! Você pegará a proposição composta e construirá a sua *tabela verdade*. Se você verificar que aquela proposição nem é uma *tautologia* (só resultados **V**), e nem é uma *contradição* (só resultados **F**), então, pela via de exceção, será dita uma *contingência*!

Exemplo: A proposição "**p ↔ (p ∧ q)**" é uma contingência. Por que essa proposição é uma contingência? Porque não é uma tautologia e nem uma contradição. Só por isso! Vejamos sua tabela verdade:

p	q	p ∧ q	p ↔ (p ∧ q)
V	V	V	V
V	F	F	F
F	V	F	V
F	F	F	V

RESUMO DO TÓPICO 1

Neste tópico, você aprendeu que:

- A tabela verdade consiste em um tipo de tabela matemática usada em lógica para determinar a validade de uma fórmula.

- A tabela verdade trabalha com as seguintes operações do cálculo proposicional:

  - Negação (~)
  - Conjunção (^)
  - Disjunção (v)
  - Condição < se....então>
  - Bicondicional (se e somente se) que é a equivalência
  - Disjunção exclusiva (XOR)
  - Adaga de Quine (NOR)

- O conceito mais básico da lógica está diretamente ligado ao conceito de proposição, que significa submeter à apreciação e requer juízo de avaliação. A proposição é uma sentença que necessita ser declarada através de termos ou sentenças bem objetivas, o que a torna declarativa. O conteúdo das sentenças ou termos sempre será verdadeiro ou falso.

- Somente para as sentenças declarativas podemos atribuir os valores de juízo: verdadeiro ou falso.

- Normalmente, as proposições são representadas por letras minúsculas (t, v, d, s, r etc.).

- Nenhuma proposição pode assumir os dois valores de verdadeiro ou falso ao mesmo tempo. Ou a proposição é verdadeira, ou é falsa. Existem três princípios básicos que as proposições devem atender:

  - Princípio da Identidade: a proposição é verdadeira ou falsa.
  - Não contradição: a proposição não poderá ser verdadeira ou falsa ao mesmo tempo.
  - Terceiro excluído: não existem outros valores de juízo: somente verdadeiro e falso.

- Os conectivos lógicos são expressões cuja finalidade é ligar duas ou mais proposições. Eles estão presentes nas proposições compostas. Duas checagens são importantes para determinar se as proposições compostas são verdadeiras ou falsas.

- O valor das proposições que compõem as sentenças.
- O tipo de conectivo que liga as proposições de uma mesma sentença.

- O conectivo e representa a conjunção. Também pode ser representado pelo símbolo (^).

- O conectivo ou representa a disjunção.

- O conectivo ou ...ou... representa disjunção exclusiva.

- O conectivo se... então representa uma condição.

- O conectivo bicondicional é representado por se e somente se.

- A partícula não representa a negação.

AUTOATIVIDADE

1 Explique o que é tautologia e contingência.

2 Cite e explique os três princípios básicos a que as proposições devem atender.

3 Assinale a alternativa que exibe a quantidade de linhas que uma proposição composta com quatro proposições simples pode possuir em uma tabela verdade.

a) ( ) 16 linhas.
b) ( ) 32 linhas.
c) ( ) 64 linhas.
d) ( ) 128 linhas.

4 Construa a tabela da verdade para a seguinte proposição: $E = (p \vee (\neg p \vee q)) \wedge \neg(q \wedge \neg r)$

TÓPICO 2

CONSISTÊNCIA DOS DADOS DE ENTRADA

1 INTRODUÇÃO

Neste tópico, você será apresentado ao conteúdo de Consistência de Dados e Modularização. A consistência remete à importância da validação dos dados de entrada dos algoritmos e a Modularização, à construção de algoritmos através de procedimentos e funções cujo intuito é organizar melhor o código de programação.

2 CONSISTÊNCIA E MODULARIZAÇÃO

Consistir os dados significa validá-los, ou seja, verificar se os valores digitados como entrada são válidos ou não. Para isso, trabalhamos com as "críticas" ou mensagens de alerta, que serão exibidas ao usuário, referentes aos valores digitados que não atendam às condições estabelecidas. Por exemplo, quando desejamos calcular a nota de um aluno por média aritmética, solicitamos a entrada de 3 números válidos entre 0 e 10. Logo, o intervalo de valores permitidos varia de 0 a 10, não sendo aceitos números negativos e nem maiores do que 10.

No caso de digitação errada ou proposital por parte do usuário, o algoritmo exibe uma mensagem de crítica, alertando-o para que digite de forma correta os valores solicitados.

3 MODULARIZAÇÃO

Existem algoritmos de todas as complexidades, sendo que esta característica está diretamente relacionada com a finalidade que o algoritmo apresenta, com o tipo de negócio ao qual está associado, bem como os requisitos e regras de negócios para a elaboração do mesmo.

Algoritmos que apresentam muitas regras em sua construção tendem a ficar com código extenso, dificultando a sua interpretação e futura manutenção por trechos de código que não ficam claros ou que são repetidos dentro da lógica

de construção. Uma solução eficaz para o problema descrito é a modularização, ou seja, um algoritmo maior é quebrado em módulos, ou subalgoritmos. Esta técnica facilita a compreensão, além de organizar melhor o código, permitindo a sua reutilização.

Na prática, dividimos um algoritmo maior em diversas partes menores – tantas quantas forem necessárias. O módulo principal se diferencia dos outros por ser o início da execução. Este módulo é que faz a chamada e contempla em sua estrutura a chamada dos módulos menores. Importante ter em mente que a chamada de um módulo significa a execução das regras contidas nele. Terminada a execução do módulo, esta volta para o ponto do módulo, podendo seguir com a programação estabelecida no módulo principal, chamar outro módulo ou até mesmo o módulo que acabou de ser executado, caso o mesmo se encontre dentro de uma condição de repetição.

Um módulo nada mais é do que um grupo de comandos que constitui um trecho de algoritmo com uma função bem definida o mais independente possível das demais partes do algoritmo. Cada módulo, durante a execução do algoritmo, realiza uma tarefa específica da solução do problema e, para tal, pode contar com o auxílio de outros módulos do algoritmo. Desta forma, a execução de um algoritmo contendo vários módulos pode ser vista como um processo cooperativo.

A construção de algoritmos compostos por módulos, ou seja, a construção de algoritmos através de modularização possui uma série de vantagens:

- Torna o algoritmo mais fácil de escrever. O desenvolvedor pode focalizar em pequenas partes de um problema complicado e escrever a solução para estas partes, uma de cada vez, ao invés de tentar resolver o problema como um todo de uma só vez.
- Torna o algoritmo mais fácil de ler. O fato do algoritmo estar dividido em módulos permite que alguém, que não seja o seu autor, possa entender o algoritmo mais rapidamente por tentar entender os seus módulos separadamente, pois como cada módulo é menor e mais simples do que o algoritmo monolítico correspondente, entender cada um deles separadamente é menos complicado do que tentar entender o algoritmo como um todo de uma só vez.
- Eleva o nível de abstração. É possível entender o que um algoritmo faz por saber apenas o que os seus módulos fazem, sem que haja a necessidade de entender os detalhes internos aos módulos. Além disso, se os detalhes nos interessam, sabemos exatamente onde examiná-los.
- Economia de tempo, espaço e esforço.

O módulo principal solicita a execução dos vários módulos em uma dada ordem, os quais, antes de iniciar a execução, recebem dados do módulo principal e, ao final da execução, devolvem o resultado de suas computações.

3.1 COMPONENTES DE UM MÓDULO

Os módulos que estudaremos daqui em diante possuem dois componentes: corpo e interface. O corpo de um módulo é o grupo de comandos que compõe o trecho de algoritmo correspondente ao módulo. Já a interface de um módulo pode ser vista como a descrição dos dados de entrada e de saída do módulo. O conhecimento da interface de um módulo é tudo o que é necessário para a utilização correta do módulo em um algoritmo.

A interface de um módulo é definida em termos de parâmetros. Um parâmetro é um tipo especial de variável utilizado pelos módulos para receber ou comunicar valores de dados a outras partes do algoritmo. Existem três categorias de parâmetros:

- parâmetros de entrada, que permitem que valores sejam passados para um módulo a partir do algoritmo que solicitou sua execução;
- parâmetros de saída, que permitem que valores sejam passados do módulo para o algoritmo que solicitou sua execução;
- parâmetros de entrada e saída, que permitem tanto a passagem de valores para o módulo quanto a passagem de valores do módulo para o algoritmo que solicitou sua execução.

Quando criamos um módulo, especificamos o número e os tipos das variáveis correspondentes aos parâmetros que ele necessita, isto determina a interface do módulo. Qualquer algoritmo que use um módulo deve utilizar os parâmetros que são especificados na interface do módulo para se comunicar com ele.

FONTE: Disponível em: <https://www.dimap.ufrn.br/~richard/pubs/dim0320/readings/aula27.pdf>. Acesso em: 28 out. 2016.

Exemplo

```
início
        {declarações  globais}
        módulo principal;
        {declarações  locais}
        início
        {corpo do algoritmo principal}
fim;

    {definições de outros módulos ou subalgoritmos}
fim.
```

Declarações globais: são variáveis, tipos e/ou constantes declaradas no início do algoritmo que podem ser utilizadas por qualquer módulo, inclusive pelo módulo principal.
Módulo principal: é por onde se inicia a execução do algoritmo.
Declarações locais: são variáveis, tipos e/ou constantes que só podem ser utilizadas dentro do módulo no qual foram declaradas.
Corpo do algoritmo: conjunto de ações ou comandos.
Definição dos módulos ou subalgoritmos: compreende um cabeçalho chamado módulo seguido de um nome, declarações locais e o corpo do subalgoritmo.

Exemplo de Declaração de um módulo:

```
módulo identificador;
                    {declarações locais}
                        início
                            {corpo do subalgoritmo}
                        fim;
```

3.2 EXEMPLO DE ALGORITMO COM MÓDULOS

Considere um aluno com três notas. Com base nisso, calcularemos a média deste aluno e exibiremos sua situação de acordo com a média obtida.

```
Algoritmo media
                Nome     texto;
                Nota1    numerico;
              Nota2      numerico;
              Nota3      numerico;
                Situacao  texto;
Início
   Início

        Leia(nome);
        Leia(nota1);
        Leia(nota2);
        Leia(nota3);

       --Chamada do módulo de cálculo
 calculo;
   Fim;
```

```
Módulo calculo;
  Início
        Media := nota1+nota2+nota3 / 3;
        Se média >= 7
              Situação := "Aprovado";
         Senão
          Situação := "Reprovado";
         Fim-se
        Escreva(media,situacao);
   Fim;
Fim.
```

Em linguagens de programação, os módulos ou subalgoritmos, são chamados de procedimentos ou funções.

4 FUNÇÕES E PROCEDIMENTOS EM PORTUGOL

São duas as formas de definição de módulos em Portugol e nas linguagens de programação.

- Função: uma função é um módulo que produz um único valor de saída. Ela pode ser vista como uma expressão que é avaliada para um único valor, sua saída, assim como uma função em Matemática.
- Procedimento: um procedimento é um tipo de módulo usado para várias tarefas, não produzindo valores de saída. As diferenças entre as definições de função e procedimento permitem determinar se um módulo para uma dada tarefa deve ser implementado como uma função ou procedimento. Nas definições acima, estamos considerando que a produção de um valor de saída por uma função é diferente da utilização de parâmetros de saída ou de entrada e saída. Veremos mais adiante que os parâmetros de saída e de entrada e saída modificam o valor de uma variável do algoritmo que a chamou, diferentemente do valor de saída produzido por uma função que será um valor a ser usado em uma atribuição ou envolvido em alguma expressão.

Como exemplo da diferenciação do uso de funções ou procedimentos, considere, por exemplo, que um módulo para determinar o menor de dois números é necessário. Neste caso, o módulo deve ser implementado como uma função, pois ele vai produzir um valor de saída. Por outro lado, se um módulo, para determinar o maior e o menor valor de uma sequência de números, é requerido, ele deverá produzir seus resultados via os parâmetros de saída e será implementado como um procedimento, pois ele vai produzir dois valores de saída. A fim de escrevermos uma função ou procedimento, precisamos construir as seguintes partes: interface e corpo.

FONTE: Disponível em: <https://www.dimap.ufrn.br/~richard/pubs/dim0320/readings/aula27.pdf>. Acesso em: 28 out. 2016.

A forma mais simples de resolver problemas através de algoritmos consiste em elaborar construções com apenas quatro tipos de estruturas de controle de fluxo: comandos de entrada e saída, comandos de atribuição e repetição, e, comandos condicionais. São usadas ainda as expressões lógicas e aritméticas. Além disso, precisamos nos preocupar com as limitações impostas pelos compiladores de cada ferramenta no momento de elaborar a lógica a ser executada.

Várias são as situações que contribuem para o entendimento e adequada manutenção dos programas. Por isso, a construção deve sempre considerar formas de facilitar o trabalho dos analistas e programadores. Isso é possível elaborando-se o código em partes bem definidas e que contenham partes importantes e coerentes do problema em si. Ter conhecimento e domínio da construção de procedimentos e funções é o melhor caminho para a construção de códigos de programação com qualidade, facilitando a manutenção e reutilização, sempre que necessário.

Procedimentos e funções são subprogramas, ou seja, são pedações de um programa menor, dentro de um programa maior. Este particionamento permite a construção de programas mais enxutos, porém, altamente elaborados, pois trabalham de forma modular, com reaproveitamento e maior legibilidade do código.

4.1 PASSAGEM DE PARÂMETROS

Quando construímos algoritmos de forma modularizada é muito comum precisar passar informações de um módulo para outro. Essas informações são denominadas **parâmetros**. Os parâmetros são responsáveis por estabelecer a comunicação entre os módulos. Um parâmetro que sai de um módulo é um valor de entrada para o módulo que está sendo chamado, sendo que a lógica do módulo é que vai definir a forma de manipulação da informação: se será somente para leitura ou se o valor será alterado. Caso esta informação seja alterada no novo módulo, isto não significa que é alterada no módulo de origem, onde permanece inalterada. Um parâmetro também pode ser chamado de argumento (MANZANO, 2000). Existem dois tipos distintos de parâmetros:

Passagem de parâmetros por valor: Consiste em copiar o valor das variáveis locais usadas na lógica e passá-las para outro módulo sem alterar informações originais.

Passagem de parâmetros por referência: Neste caso, é feita a cópia do endereço da memória onde a variável está armazenada. Nesse mecanismo, outra variável ocupando outro espaço diferente na memória não armazena o dado em si, e sim o endereço onde ele se localiza na memória. Assim, todas as modificações efetuadas nos dados do parâmetro estarão sendo feitas no conteúdo original da variável.

Podemos identificar os dois tipos de situação em um mesmo algoritmo.

EXEMPLO 2: Dados dois valores positivos, calcular e exibir o resultado das operações aritméticas efetuadas utilizando passagem por parâmetros.

```
início

    módulo principal;
  real: x, y, res;   {variáveis locais}
    início
      leia(x);
      leia(y);
      se (x>0) e (y>0) então
            início
              calc(x,y,'+', res);
   escreva(res);
              calc(x,y,'-', res);
        escreva(res);
              calc(x,y,'*', res);
              escreva(res);
              calc(x,y,'/', res);
              escreva(res);
            fim;
  fim;
    módulo calc(real:a,b;caracter:oper; ref real:re);
    início
      escolha oper
            caso '+' : re ← a + b;
            caso '-' : re ← a - b;
            caso '*' : re ← a * b;
            caso '/' : re ← a / b;
      fim;
    fim;
fim.
```

Ainda podemos ter um módulo que tenha a função de calcular um resultado, como exemplo as funções matemáticas. Este tipo de módulo tem o objetivo de retornar uma única informação ao módulo chamador, como se fosse uma resposta. Esta resposta será feita através do comando **retorne()**:

```
Declaração: retorne(expressão);
```

onde:
expressão: pode ser uma informação numérica, alfanumérica ou uma expressão aritmética.

EXEMPLO 3: O mesmo do exemplo 1 modificado.

```
início
    módulo principal;
```

```
        início
          entrada;
        fim;

        módulo entrada;
        real:re;
        caracter:op;
        início
                repita
                        leia(re);
                        escreva(conv(re));
                        leia(op);
                até op='N';
        fim;

        módulo conv(real:r);
        início
                retorne(r/1.20);
        fim;
fim.
```

EXEMPLO 4: O mesmo do exemplo 2 modificado usando retorne. Observe que sempre irá retornar um único resultado de cada vez.

```
início
        módulo principal;
       real: x, y;    {variáveis locais}
        início
          leia(x);
          leia(y);
          se (x>0) e (y>0) então
                início
                  escreva(calc(x,y,'+'));
                  escreva(calc(x,y,'-'));
                  escreva(calc(x,y,'*'));
                  escreva(calc(x,y,'/'));
                fim;
      fim;
        módulo calc(real:a,b;caracter:oper);
        início
          escolha oper
                caso '+' : retorne(a + b);
                caso '-' : retorne(a – b);
                caso '*' : retorne(a * b);
                caso '/' : retorne (a / b);
          fim;
        fim;
fim.
```

EXEMPLO 5: Dados três números quaisquer, desenvolver um algoritmo que receba via parâmetro estes números, calcule o quadrado de cada um e retorne o resultado para o módulo chamador (passagem de parâmetro por referência).

```
Início
  Módulo principal;
        Real: n1, n2, n3, q1, q2, q3;
        Início
              Leia(n1,n2,n3);
              Quadrado(n1,n2,n3,q1,q2,q3);
              Escreva( 'O quadrado de n1 =',q1);
         Escreva( 'O quadrado de n2 =',q2);
         Escreva( 'O quadrado de n3 =',q3);
        Fim;

Módulo quadrado (real:n1,n2,n3; ref real:q1,q2,q3);
        início
              q1← n1 **2;
              q2← n2 **2;
              q3← n3 **2;
        Fim;
Fim.
```

EXEMPLO6: O mesmo exercício usando retorne.

```
Início
  Módulo principal;
        Real: n1,n2,n3;
        Início
              Leia(n1,n2,n3);
              Escreva( 'O quadrado de n1 =',quadrado(n1));
          Escreva( 'O quadrado de n2 =',quadrado(n2));
          Escreva( 'O quadrado de n3 =',quadrado(n3));
        Fim;

Módulo quadrado (real:n);
        início
              retorne(n **2);
        fim;
Fim.
```

Antes de se definir pela modularização é importante pensar nas seguintes questões:

- Saber exatamente quando usar procedure e quando usar uma função.
- Saber a diferença entre variáveis locais e globais e quando usá-las.
- Saber quando qual tipo de passagem de parâmetros deverá ser usado: valor ou referência.

Imagine um algoritmo que você precisa ler uma sequência de valores (números inteiros terminados com zero) exibindo como resultado somente os números pares.

Exemplo:

```
Algoritmo Pares;;

        Num   inteiro;
Inicio
    Leia (num);
      Enquanto num <> o faça
          Se num mod2 = 0 entao
                        Escreva ('Num');
                         Leia (Num)
                   Fim-se;
              Fim enquanto;
Fim;
```

Note que no exemplo acima o código inteiro está no programa principal, entre o comando "inicio" e o comando "Fim".

4.2 VARIÁVEIS GLOBAIS

São as variáveis declaradas logo após o cabeçalho do programa (seu título) e antes do comando de início (Inicio) da execução do programa principal. As variáveis são endereços de memória (local) onde os dados ficam armazenados temporariamente até serem usados. As variáveis globais podem ser utilizadas dentro do programa onde foram criadas, mas não fora dele.

No exemplo anterior, temos uma única variável. Seu nome é Num e seu tipo é inteiro.

4.3 VARIÁVEIS LOCAIS

As variáveis locais são declaradas dentro dos subprogramas e só podem ser usadas dentro dos mesmos. Elas não podem ser usadas pelo programa principal.

5 FUNÇÕES

No exemplo acima (do algoritmo que verifica se um número é par), como poderíamos usar uma Função?

Note que existe um cálculo no programa principal, representado pelo código **Num mod2 = 0**.

Este cálculo é uma expressão booleana (retorna verdadeiro ou falso) que calcula a divisão de um número por 2 e verifica se o resultado é zero. Se for zero, significa que o número é par. Caso contrário, não. O exemplo proposto é simples, apresentando poucas linhas de código em sua construção, mas imagine um cálculo mais complexo, com muitas linhas de código e muitas checagens até se chegar ao resultado final. Esta parte mais extensa e bem específica (do cálculo) poderia ser transformada em uma função.

Basicamente, o programa principal passaria como parâmetro para a função um número lido e se encarregaria de validar as checagens, devolvendo o resultado. Esta forma de escrita cria um vínculo entre o programa principal e o subprograma.

Toda função apresenta as seguintes características:

- Toda função tem um nome.
- Toda função pode ou não receber parâmetros ou argumentos de entrada.
- Toda função retorna, obrigatoriamente, um valor de um único tipo de dado (data, texto ou número).

Vejamos o exemplo:

```
Algoritmo função 01;
program imprime pares;

            Num   inteiro:

 ( funcao que verifica se o número recebido como parâmetro é par )

       Funcao  verifica par  (Num inteiro)  retorna boolean
          Epar   boolean;

inicio
se Num mod 2 = 0 then
Epar := verdadeiro;
senao
Epar := falso;
                  Fim se;
         Fim;

Inicio  (programa principal)

 Leia (Num) ;
 Enquanto Num <> 0 faça
```

```
    Se Epar ( Num) então –(aqui ocorre a passagem de parâmetro por valor)
Escreva (Num);
Leia (Num);
                    Fim-se;
        Fim;
```

Perceba que transferir parte do código para um sobprograma não altera e nem interfere na execução do programa principal. É importante notar que o código fica mais legível e organizado.

6 PROCEDIMENTOS

Os procedimentos se diferem das funções no sentido de que podem não retornar nenhum valor, ou mais de um inclusive com tipos de dados distintos. Na prática, um procedimento poderia retornar um valor texto, um valor numérico e uma data, por exemplo.

A sintaxe de criação da procedure se resume em informar seu nome e a lista de parâmetros de entrada.

O procedimento também precisa ter um nome definido. E caso existam, deve ocorrer também a passagem de parâmetros como entrada para a lógica que será desenvolvida.

Exemplo de procedimento:

- Criar um procedimento que receba dois valores inteiros por parâmetro e realize a troca desses valores.

```
algoritmo "TrocarValores"
var
   x, y: inteiro

   procedimento troca(var a, b: inteiro)
   var
      auxiliar: inteiro
   inicio
      auxiliar <- a
      a <- b
      b <- auxiliar
   fimprocedimento

inicio
   x <- 5
   y <- 8
   escreval("Os valores de x e y ANTES da troca são: ", x, y)
   troca(x, y)
   escreval("Os valores de x e y DEPOIS da troca são: ", x, y)
fimalgoritmo
```

FONTE: Disponível em: <http://docente.ifrn.edu.br/albalopes/semestres-anteriores/2012.1/disciplinas/algoritmos-e-p.o.o-integrado/Aula13Procedimentos.pdf>. Acesso em: 27 out. 2016.

Na Unidade 3 praticaremos a construção de algoritmos, bem como a construção de procedimentos e funções. No momento é importante que você perceba a diferença entre as duas construções e conheça a sintaxe de construção.

RESUMO DO TÓPICO 2

Neste tópico, você aprendeu que:

- Consistir os dados significa validá-los, ou seja, verificar se os valores digitados como entrada são válidos ou não.

- Existem algoritmos de todas as complexidades, sendo que esta característica está diretamente relacionada com a finalidade, com o tipo de negócio ao qual está associado, bem como os requisitos e regras de negócios para a elaboração do mesmo.

- Algoritmos que apresentam muitas regras em sua construção tendem a ficar com código extenso, dificultando a sua interpretação e futura manutenção por trechos de código que não ficam claros ou que são repetidos dentro da lógica de construção. Uma solução eficaz para o problema descrito é a modularização, ou seja, um algoritmo maior é quebrado em módulos, ou subalgoritmos. Esta técnica facilita a compreensão, além de organizar melhor o código, permitindo a sua reutilização.

- Um módulo nada mais é do que um grupo de comandos que constitui um trecho de algoritmo com uma função bem definida e o mais independente possível das demais partes do algoritmo.

- Cada módulo, durante a execução do algoritmo, realiza uma tarefa específica da solução do problema e, para tal, pode contar com o auxílio de outros módulos do algoritmo. São duas as formas de definição de módulos em Portugol e nas linguagens de programação:

  - **Função:** uma função é um módulo que produz um único valor de saída. Ela pode ser vista como uma expressão que é avaliada para um único valor, sua saída, assim como uma função em matemática.
  - **Procedimento:** um procedimento é um tipo de módulo usado para várias tarefas. Pode apresentar ou não valores de saídas. Tem uma ou várias entradas. A principal diferença em relação às funções, é que a função sempre apresentará um retorno de tipo de dado específico. Uma função retorna um único valor. Os procedimentos podem retornar valores de tipos de dados distintos.
    Nas definições acima, estamos considerando que a produção de um valor de saída por uma função é diferente da utilização de parâmetros de saída ou de entrada e saída. Veremos mais adiante que os parâmetros de saída e de entrada e saída modificam o valor de uma variável do algoritmo que a chamou, diferentemente do valor de saída produzido por uma função que será um valor a ser usado em uma atribuição ou envolvido em alguma expressão.

- **Passagem de parâmetros por valor**: Consiste em copiar o valor das variáveis locais usadas na lógica e passá-las para outro módulo sem alterar informações originais.

- **Passagem de parâmetros por referência**: Neste caso, é feita a cópia do endereço da memória onde a variável está armazenada. Nesse mecanismo, outra variável ocupando outro espaço diferente na memória não armazena o dado em si, e sim o endereço onde ele se localiza na memória. Assim, todas as modificações efetuadas nos dados do parâmetro estarão sendo feitas no conteúdo original da variável.

- **Variáveis globais:** São as variáveis declaradas logo após o cabeçalho do programa (seu título) e antes do comando de início (Inicio) da execução do programa principal. As variáveis são endereços de memória (local) onde os dados ficam armazenados temporariamente até serem usados.

- **Variáveis locais**: As variáveis locais são declaradas dentro dos subprogramas e só podem ser usadas dentro dos mesmos. Elas não podem ser usadas pelo programa principal.

AUTOATIVIDADE

1 Diferencie procedimentos e funções.

2 Faça uma função que recebe um número inteiro por parâmetro e retorna verdadeiro se ele for par e falso se for ímpar.

3 Faça um programa que através de uma função, calcule o valor de um número elevado ao quadrado.

TÓPICO 3

VETORES E MATRIZES

1 INTRODUÇÃO

Nesta unidade, você, além de aprender a estruturar algoritmos em forma de vetores e matrizes, também estará apto a identificar as situações de uso destas estruturas.

2 CONCEITOS DE VETORES E MATRIZES

Vetores e **matrizes** são estruturas de dados muito simples que podem nos ajudar muito quando temos muitas variáveis do mesmo tipo em um algoritmo. Imagine o seguinte problema: Você precisa criar um algoritmo que lê o nome e as 4 notas de 50 alunos, tendo que calcular a média de cada aluno e informar quais foram aprovados e quais foram reprovados. Conseguiu imaginar quantas variáveis você vai precisar? Muitas né?! Vamos fazer uma conta rápida: 50 variáveis para armazenar os nomes dos alunos, (4 * 50 =) 200 variáveis para armazenar as 4 notas de cada aluno e por fim, 50 variáveis para armazenar as médias de cada aluno. 300 variáveis no total, sem contar a quantidade de linhas de código que você vai precisar para ler todos os dados do usuário, calcular as médias e apresentar os resultados. Mas temos uma boa notícia para você. Nós não precisamos criar 300 variáveis! Podemos utilizar **Vetores** e **Matrizes** (também conhecidos como **ARRAYs**)!

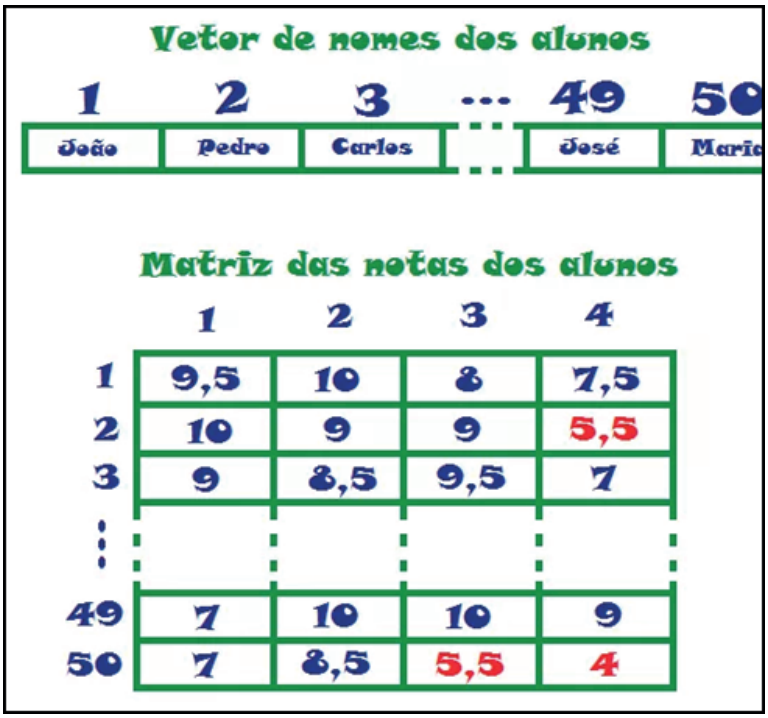
Vetor (**array** unidimensional) é uma variável que armazena várias variáveis do mesmo tipo. No problema apresentado anteriormente, nós podemos utilizar um vetor de 50 posições para armazenar os nomes dos 50 alunos.

Matriz (**array** multidimensional) é um **vetor** de **vetores**. No nosso problema, imagine uma matriz para armazenar as 4 notas de cada um dos 50 alunos, ou seja, um vetor de 50 posições, e em cada posição do vetor há outro vetor com 4 posições. Isso é uma matriz.

Cada item do vetor (ou matriz) é acessado por um número chamado de **índice**. Vamos representar os vetores e matrizes graficamente para facilitar o entendimento do conceito.

FONTE: Disponível em: <http://www.dicasdeprogramacao.com.br/o-que-sao-vetores-e-matrizes-arrays/>. Acesso em: 28 out. 2016.

FIGURA 21 - VETORES E MATRIZES



FONTE: Disponível em: <http://www.dicasdeprogramacao.com.br/o-que-sao-vetores-e-matrizes-arrays/>. Acesso em: 28 out. 2016.

Perceba, na figura acima, que cada posição do vetor é identificada por um número (chamado de índice). No caso da matriz são dois números (um na vertical e um na horizontal).

Vamos entender melhor desenvolvendo um algoritmo na prática. O exemplo descrito calcula a média de cinco alunos. Os dados são armazenados dentro de vetores para a exibição do resultado final. Importante perceber que nesta situação não é possível a utilização de variáveis normais, as quais estudamos nos primeiros algoritmos, devido a sua incapacidade de armazenar mais de um valor.

FIGURA 22 - ALGORITMO MÉDIA DOS ALUNOS

```
algoritmo "MediaDe5Alunos"

var

   nomes: vetor [1..5] texto
   notas: vetor [1..5,1..4] numerico
   medias: vetor [1..5] de real
   contadorLoop1, contadorLoop2: inteiro

inicio

     //Leitura dos nomes e as notas de cada aluno
     PARA contadorLoop1 DE 1 ATE 5 FACA
          ESCREVA("Digite o nome do aluno(a) número ", contadorLoop1, " de 5: ")
          LEIA(nomes[contadorLoop1])
          PARA contadorLoop2 DE 1 ATE 4 FACA
               ESCREVA("Digite a nota ", contadorLoop2, " do aluno(a) ", nomes[contadorLoop1], ": ")
               LEIA(notas[contadorLoop1, contadorLoop2])
          FIMPARA
          //CÁLCULO DAS MÉDIAS
          medias[contadorLoop1] := (notas[contadorLoop1, 1] + notas[contadorLoop1, 2]
                                                            + notas[contadorLoop1, 3]
                                                            + notas[contadorLoop1, 4]) / 4
     FIMPARA

     //APRESENTAÇÃO DOS RESULTADOS
     PARA contadorLoop1 DE 1 ATE 5 FACA
          SE medias[contadorLoop1] >= 6 ENTAO
             ESCREVAL("O aluno(a) ", nomes[contadorLoop1], " foi aprovado com as notas (", notas[contadorLoop1, 1], ",
                                    ", notas[contadorLoop1, 2], ",
                                    ", notas[contadorLoop1, 3], ",
                                    ", notas[contadorLoop1, 4], ") e média: ", medias[contadorLoop1])
          SENAO
             ESCREVAL("O aluno(a) ", nomes[contadorLoop1], " foi reprovado com as notas (", notas[contadorLoop1, 1], ",
                                    ", notas[contadorLoop1, 2], ",
                                    ", notas[contadorLoop1, 3], ",
                                    ", notas[contadorLoop1, 4], ") e média: ", medias[contadorLoop1])
          FIMSE
     FIMPARA

fimalgoritmo
```

FONTE: Disponível em: <http://www.dicasdeprogramacao.com.br/o-que-sao-vetores-e-matrizes-arrays/>. Acesso em: 28 out. 2016.

Podemos dizer que as matrizes e vetores são estruturas de dados que se organizam a partir de dados primitivos que já existem. Estas estruturas podem armazenar um conjunto de dados e são definidas como variáveis compostas. Estas variáveis compostas são classificadas de duas formas distintas. Variáveis homogêneas e heterogêneas.

Variáveis compostas homogêneas são variáveis que armazenam vários elementos do mesmo tipo primitivo. Exemplo: um conjunto de números inteiros e positivos.

As variáveis compostas heterogêneas são estruturas de dados (armazenam um conjunto de dados). Esses dados podem ser armazenados em dois tipos de variáveis: as variáveis unidimensionais (conhecidas como vetores) e as variáveis multidimensionais (conhecidas como matrizes).

Variáveis compostas unidimensionais – VETORES

Precisam de um índice para individualizar um elemento do conjunto.

Declaração: tipo: identificador[n];

onde**: identificador** – é o nome de uma variável;
n – é o tamanho do vetor

Exemplo: Média de 40 alunos

Estrutura que se forma para guardar a nota e a posição:

8.0	7.0	5.5	9.5	6.4	9.9	1.0	.....	4.8

Para acessar cada elemento da variável composta unidimensional (vetor) é necessário especificar o nome da variável seguido do número do elemento que se deseja acessar (deverá estar entre colchetes). Este número tem o nome de índice. O índice pode variar de 0 até n -1. Por exemplo, se um vetor possui 40 valores, os índices válidos são entre 0 até 39.

```
Leia (media[0]);
media [2] ← 8.5;
escreva(media[3]);
```

Exemplo 1: Dados 5 valores numéricos, elaborar um algoritmo que calcule e exiba a média dos valores dados e quantos desses valores estão acima da média.

Resolução sem utilizar vetor:
Algoritmo sem vetor

```
 inteiro:n1,n2,n3,n4,n5,cont;
 real:media; -- não é um número inteiro  -- pode apresentar casas decimais
 cont← 0;

início
    leia (n1);
    leia (n2);
    leia (n3);
    leia (n4);
    leia (n5);
      media :=  (n1+n2+n3+n4+n5)/5;
      escreva(media);

      Se n1 > media então
          Cont := cont +1;
      Senão se n2 > media
          Cont := cont +1;
      Senão se n3 > media
          Cont := cont +1;
      Senão se n4 > media
```

```
            Cont := cont +1;

        Senão se n5 > media
            Cont := cont +1;
        Fim-se;
        Escreva (cont);
    Fim;
```

Perceba a diferença agora, usando vetores (variável composta).

Resolução utilizando variável composta:

```
Início
   N[ 5 ]   numerico;
   Media  real    := 0;   -- já inicializamos em zero
   Cont    inteiro := 0; -- já inicializamos em zero

Para i de 0 até 4 faça
   início
     leia(n[i]);
     media := media + n[i];
   fim;

   media := media/5;
   escreva(media);
   Para i de 0 até 4 faça
        se (n[i]>media) então
            cont := cont +1;
        fim-se;
   fim para;
    escreva(cont);
fim.
```

Em relação ao exercício anterior, note que a lógica muda bastante. O controle condicional foi substituído por um *loop* para faça com um controle condicional reduzido. O código ficou mais enxuto e legível.

Variáveis compostas multidimensionais - MATRIZ

São variáveis que utilizam mais de um índice para acessar cada um de seus elementos. Podemos ter duas ou mais dimensões.

A declaração de uma matriz ocorre da seguinte forma:

Identificador[n][m] e tipo

onde: **identificador -** é o nome de uma variável;
n - é o número de linhas da matriz;
m - é o número de colunas da matriz;

Exemplo aluno [3][5];

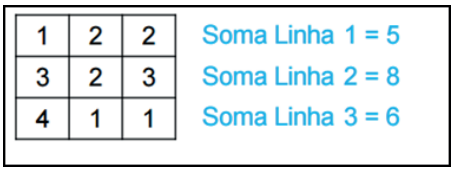
A estrutura de uma matriz pode ser entendida pelas dimensões de uma tabela conforme o exemplo proposto:

	0	1	2	3	4
0					
1					
2					

O acesso aos elementos da matriz, para leitura e escrita, são expressos por índices, conforme segue exemplo:

```
Leia (aluno[1][1]);
Aux_teste :=          Aluno [1][2];
Escreva (aluno [1][3]);
 ........
```

Exemplo 1: Criar um algoritmo que leia uma matriz 3x3. Em seguida, exiba a soma dos elementos de cada uma das linhas.



FONTE: Disponível em: <http://www.ouropreto.ifmg.edu.br/lp/slides-aulas/10-vetores-e-matrizes>. Acesso em: 30 out. 2016.

```
algoritmo "exemplo01"
var
   numeros: vetor[1..3, 1..3] de inteiro
   i, j: inteiro
   soma: inteiro
inicio
      para i de 1 ate 3 faca
            para j de 1 ate 3 faca
                  escreva("Digite um valor para a posição [", i, ",", j, "]: "
                  leia(numeros[i,j])
            fimpara
      fimpara

      para i de 1 ate 3 faca
            soma <- 0
            para j de 1 ate 3 faca
                  soma <- soma + numeros[i, j]
            fimpara
            escreval ("Soma Linha ", i, ": ", soma)
      fimpara
fimalgoritmo
```

FONTE: Disponível em: <http://www.ouropreto.ifmg.edu.br/lp/slides-aulas/10-vetores-e-matrizes>. Acesso em: 30 out. 2016.

Exemplo 2: Escreva um algoritmo que leia uma matriz 4x3. Em seguida, receba um novo valor do usuário e verifique se este valor se encontra na matriz. Caso o valor se encontre na matriz, escreva a mensagem "O valor se encontra na matriz". Caso contrário, escreva a mensagem "O valor NÃO se encontra na matriz.

```
algoritmo "exemplo03"
var
   numeros: vetor[1..4, 1..3] de inteiro
   i, j, buscar: inteiro
   achou: logico
inicio
      para i de 1 ate 4 faca
            para j de 1 ate 3 faca
                  escreva("Digite um valor para a posição [", i, ",", j, "]: ")
                  leia(numeros[i,j])
            fimpara
      fimpara

      escreva("Digite um valor para ser buscado na matriz: ")
      leia(buscar)

      achou <- falso
      para i de 1 ate 4 faca
            para j de 1 ate 3 faca
                  se (numeros[i,j] = buscar) entao
                     achou <- verdadeiro
                  fimse
            fimpara
      fimpara
      se achou=verdadeiro entao
         escreva("O número se encontra na matriz.")
      senao
         escreva("O número NÃO se encontra na matriz.")
      fimse
fimalgoritmo
```

FONTE: Disponível em: <http://www.ouropreto.ifmg.edu.br/lp/slides-aulas/10-vetores-e-matrizes>. Acesso em: 30 out. 2016.

LEITURA COMPLEMENTAR

A SILENCIOSA DITADURA DO ALGORITMO

Em sociedades digitalizadas, decisões cruciais sobre a vida são tomadas por máquinas e códigos. Isso multiplica a desigualdade e ameaça à democracia

Por Pepe Escobar | Tradução: Inês Castilho

Vivemos todos na Era do Algoritmo. Aqui está uma história que não apenas resume a era, mas mostra como a obsessão pelo algoritmo pode dar terrivelmente errado.

Tudo começou no início de setembro, quando o *Facebook* censurou a foto ícone de Kim Phuch, a "menina do Napalm", símbolo reconhecido em todo o mundo da Guerra do Vietnã. A foto figurava em *post* no *Facebook* do escritor norueguês Tom Egeland, que pretendia iniciar um debate sobre "sete fotos que mudaram a história da guerra".

Não só o seu *post* foi apagado, como Egeland foi suspenso do *Facebook*. O Aftenposten, principal jornal diário da Noruega, propriedade do grupo de mídia escandinavo Schibsted, transmitiu devidamente a notícia, lado a lado com a foto. O *Facebook* pediu então que o jornal apagasse a foto – ou a tornasse irreconhecível em sua edição *on-line*. Antes mesmo de o jornal responder, artigo e foto já haviam sido censurados na página do Aftenposten do *Facebook.*

A primeira ministra norueguesa, Erna Solberg, protestou contra tudo isso em sua página do Facebook. Também foi censurada. O Aftenposten então sapecou a história inteira em sua primeira página, ao lado de carta aberta a Mark Zuckerberg, assinada pelo diretor do jornal, Espen Egil Hansen, acusando o *Facebook* de abuso do poder.

Passaram-se 24 longas horas até que o colosso de Palo Alto recuasse e "desbloqueasse" a publicação.

Uma opinião embrulhada em código

O *Facebook* empenhou-se ao máximo para controlar os danos depois do episódio. Isso não alterou o fato de que o imbróglio "menina da Napalm" é um clássico drama do algoritmo, como ocorre na aplicação de inteligência artificial para avaliar conteúdo.

Como outros gigantes da Economia de Dados, o Facebook deslocaliza a filtragem de dados para um exército de moderadores em empresas localizadas do Oriente Médio ao Sul da Ásia. Isso foi confirmado por Monika Bickert, do *Facebook.*

Esses moderadores têm um papel no controle daquilo que deve ser eliminado da rede social, a partir de sinalizações dos usuários, mas a informação é então comparada a um algoritmo, que tem a decisão final.

Não é necessário ter PhD para perceber que esses moderadores não têm, necessariamente, vasta competência cultural, ou capacidade de analisar contextos. Isso para não mencionar que os algoritmos são incapazes de "entender" contexto cultural e certamente não são programados para interpretar ironia, sarcasmo ou metáforas culturais.

Os algoritmos são literais. Em poucas palavras, são uma opinião embrulhada em código. E, no entanto, estejamos atingindo um estágio em que a máquina decide o que é notícia. O *Facebook*, por exemplo, conta agora apenas com o algoritmo para definir quais histórias coloca em destaque.

Pode haver um lado positivo nessa tendência – como o Facebook, o Google e o YouTube usarem sistemas para bloquear rapidamente vídeos do ISIS e propaganda jihadista semelhante. Logo estará em operação eGLYPH – um sistema que censura vídeos que violam supostos direitos autorais por meio "*hashing*", ou codificação para busca rápida. Uma única marca será atribuída a vídeos e áudios considerados "extremistas", possibilitando assim sua remoção automática em qualquer nova versão e bloqueando novos uploads.

E isso nos traz para um território ainda mais turvo; o próprio conceito de "extremista". E os efeitos, sobre todos nós, de sistemas de censura baseados em lógica algorítmica.

Como as "Armas de Destruição Matemática" controlam nossa vida

É neste cenário que um livro como Weapons of Math Destruction [ou "Armas de Destruição Matemática"] de Cathy O'Neil (*Crown Publishing*), torna-se tão essencial quanto o ar que respiramos.

O'Neil lida com a coisa real; é PHD em Matemática em Harvard, ex-professora do Barnard College, ex-analista quantitativa num fundo de hedge antes de reconverter-se a pesquisadora e blogueira no mathbabe.org.

Modelos matemáticos são o motor de nossa economia digital. Isso leva O'Neil a formular seus dois insights decisivos – que podem surpreender legiões de pessoas que veem as máquinas como simplesmente "neutras".

1) "Aplicações baseadas em matemática e que empoderam a Economia de Dados são baseadas em escolhas feitas por seres humanos falíveis".
2) "Esses modelos matemáticos são opacos, e seu trabalho é invisível para todos, exceto os cardeais em suas áreas: matemáticos e cientistas computacionais. Seus vereditos são imunes a disputas ou apelos, mesmo quando errados ou nocivos. E tendem a punir pobres e oprimidos, enquanto tornam os ricos mais ricos em nossa sociedade".

Daí o conceito de Armas de Destruição Matemática (WMDs), de O'Neil; ou de o quanto modelos matemáticos destrutivos estão acelerando um terremoto social.

O'Neil detalha extensivamente como modelos matemáticos destrutivos microgerenciam vastas faixas da economia real, da publicidade ao sistema prisional, sem falar do sistema financeiro (e dos efeitos posteriores à interminável crise de 2008).

Esses modelos matemáticos são essencialmente opacos; não responsáveis; e miram acima de toda "otimização" das massas (consumidoras).

A regra de ouro é – o que mais seria? – seguir o dinheiro. Como diz O'Neil, para "as pessoas que executam os WMDs", o "*feedback* é a grana"; "os sistemas são construídos para devorar mais e mais dados, e afinar suas análises de modo a despejar nele mais e mais dinheiro".

As vítimas – como nos ataques de drone na administração Obama – são mero "dano colateral".

Paralelos entre o cassino financeiro e os Big Data são inevitáveis – e é útil o fato de que O'Neil tenha trabalhado nos dois setores.

O Vale do Silício segue o dinheiro. Vemos nele os mesmos bancos de talentos das universidades de elite norte-americanas (MIT, Stanford, Princeton), a mesma obsessão por fazer o necessário para juntar mais e mais dinheiro para a empresa empregadora.

As Armas de Destruição Matemática favorecem a eficiência. "Justiça" não passa de um conceito. Computadores não entendem conceitos. Programadores não sabem codificar um conceito – como vimos na história da "menina do Napalm". E também não sabem como ajustar algoritmos para refletir equidade.

O que temos é o conceito de "amizade" sendo medido por likes e conexões no *Facebook*. O'Neil soma tudo; "Se você pensa no WMD como indústria, injustiça é o que está sendo expelido pela fumaça da chaminé. É uma emissão tóxica".

Mande um fluxo de caixa, já

No fim, é a Deusa do Mercado que regula tudo – premiando eficiência, crescimento e fluxo de caixa sem fim.

Mesmo antes do fiasco da "menina do Napalm", O'Neil já apontara o fato crucial de que o *Facebook* determina, na realidade, e segundo seus próprios interesses, o que todos veem – e aprendem – na rede social. Nada menos que dois terços dos norte-americanos adultos têm perfil no *Facebook*. Quase a metade, afirma relatório do Centro de Pesquisa Pew, conta com o *Facebook* para parte, ao menos, das notícias que leem.

A maioria dos norte-americanos – para não falar da maioria dos 1,7 bilhão de usuários do *Facebook* espalhados pelo mundo – ignora que o *Facebook* canaliza o *feed* de notícias. As pessoas de fato acreditam que o sistema compartilha instantaneamente, com sua comunidade de amigos, qualquer coisa que é postada.

O que nos traz, mais uma vez, à questão-chave no *front* das notícias. Ao ajustar seus algoritmos para modelar as notícias que as pessoas veem, o *Facebook* tem agora tudo o que é necessário para jogar com todo o sistema político. Como observa O'Neil, "*Facebook, Google, Apple, Microsoft, Amazon* têm todos uma vasta quantidade de informação sobre grande parte da humanidade – e os meios para nos dirigir para onde queiram".

Estrategicamente, seus algoritmos não têm preço, é claro; segredo comercial supremo, não transparente. "Eles fazem seus negócios nas sombras".

Em sua recente e propagandeada viagem a Roma, Mark Zuckerberg disse que o *Facebook* é "uma empresa *high-tech*, não uma empresa jornalística". Não é bem isso. O aspecto mais intrigante do fiasco da "menina do Napalm" pode ser o fato de que Shibsted, o grupo de mídia escandinavo, está planejando investir um dinheiro enorme na criação de um novo fórum social para derrotar – quem? – o *Facebook*. Prepare-se para uma guerra novinha em folha no fronte do WMD.

FONTE: Disponível em: <http://www.cartacapital.com.br/blogs/outras-palavras/a-silenciosa-ditadura-do-algoritmo>. Acesso em: 20 out. 2016.

RESUMO DO TÓPICO 3

Neste tópico, você aprendeu que:

- **Vetores** e **matrizes** são estruturas de dados muito simples que podem nos ajudar quando temos muitas variáveis do mesmo tipo em um algoritmo.

- **Vetor** (**array** unidimensional) é uma variável que armazena várias variáveis do mesmo tipo. No problema apresentado anteriormente, nós podemos utilizar um vetor de 50 posições para armazenar os nomes dos 50 alunos.

- **Matriz** (**array** multidimensional) é um **vetor** de **vetores**. No nosso problema, imagine uma matriz para armazenar as quatro notas de cada um dos 50 alunos, ou seja, um vetor de 50 posições, e em cada posição do vetor, há outro vetor com quatro posições. Isso é uma matriz. Cada item do vetor (ou matriz) é acessado por um número chamado de **índice**.

AUTOATIVIDADE

1 Diferencie matrizes e vetores. Exemplifique a utilidade de cada estrutura.

2 Criar um algoritmo que leia uma matrizes 3x3. Em seguida, exiba a soma dos elementos de cada uma das linhas. Ex.:

1	2	2	Soma Linha 1 = 5
3	2	3	Soma Linha 2 = 8
4	1	1	Soma Linha 3 = 6

3 Escreva um algoritmo que leia um vetor com 10 posições de números inteiros. Em seguida, receba um novo valor do usuário e verifique se este valor se encontra no vetor.

UNIDADE 3

ARQUIVOS, REGISTROS, COMPLEXIDADE DE ALGORITMOS E PRÁTICA DE CONSTRUÇÃO

OBJETIVOS DE APRENDIZAGEM

A partir desta unidade você será capaz de:

- conhecer e entender o uso de arquivos e registros;
- conhecer as principais maneiras de determinar a complexidade de um algoritmo;
- praticar e aprimorar a sua experiência na construção de algoritmos.

PLANO DE ESTUDOS

Esta unidade está dividida em três tópicos. Ao final de cada um deles, você encontrará atividades que o auxiliarão no seu aprendizado.

TÓPICO 1 – ARQUIVOS E REGISTROS

TÓPICO 2 – COMPLEXIDADE DE ALGORITMOS

TÓPICO 3 – PRÁTICA DE ALGORITMOS

TÓPICO 1

ARQUIVOS E REGISTROS

1 INTRODUÇÃO

Nos tópicos anteriores, você aprendeu a declarar distintos tipos de variáveis. As variáveis de tipos de dados comuns e primitivos, como números, datas e textos, são armazenadas em variáveis em espaços de memória. Nestas situações, não há preocupação em ordenar as informações, sendo que isso pode afetar o desempenho dos algoritmos.

2 REGISTROS

Para melhorar o acesso aos dados e em consequência a performance de execução, é possível criar estruturas que armazenam variáveis de tipos de dados diferentes. Estas estruturas são chamadas de registros.

A sintaxe para a criação de registros é:
nome do registro: **registro**
campos que compõem o registro
fim_registro;
Para acessar qualquer elemento do **registro**, usamos a sintaxe:

Nome do Registro.Variável interna

Para você entender melhor, vamos usar como exemplo um algoritmo que armazene informações: aluno e nota. Como exemplo, vamos criar o algoritmo aluno e nota, utilizando **o registro** ALUNO que possua as variáveis NOME e NOTA, e recebendo um nome e nota do usuário, para depois exibir os dados do **registro** na tela:

```
Algoritmo AlunoNota
Início
ALUNO: registro;
NOME: texto(60);
NOTA: real;
fim_registro
escreva "Digite o nome do aluno";
leia ALUNO.NOME;
escreva "Digite a nota do aluno";
leia ALUNO.NOTA
escreva "O aluno" ALUNO.NOME "teve nota" ALUNO.NOTA;
Fim
```

Perceba que ao criar o registro ALUNO, as duas informações: aluno e nota, sempre serão armazenadas juntas na memória, facilitando o acesso à informação e também à criação de novos tipos de variáveis dentro do mesmo registro.

Em algumas situações, as variáveis que conhecemos não são suficientes para resolver problemas específicos nos algoritmos. Podemos então, criar um tipo de registro, colocando apenas a palavra **TIPO** antes do nome do registro, conforme exemplo a seguir:

```
Tipo nome_do_registro: registro
{campos que compõe o registro}
fim_registro;
```

Mantendo-se a situação do aluno e nota usada acima, imagine que você precise armazenar as informações de uma turma inteira, com cerca de 100 alunos. O registro resolve este problema, uma vez que será criado um registro básico denominado ALUNO e que será usado como tipo de variável. Na sequência, usaremos o conhecimento adquirido em vetores para criar um vetor do tipo ALUNO, que chamaremos de LISTAALUNO com tamanho de 100. Veja a situação resolvida no algoritmo a seguir que calcula a média das notas digitadas apresentando-a sempre que um novo nome e nota forem inseridos:

```
Algoritmo Exemplo Registro_Vetor
Início
tipo ALUNO: registro;
    NOME: texto(60);
    NOTA: real;
fim_registro
LISTAALUNOS: vetor[100]: ALUNO;
INDICE: inteiro;
SOMA: real;
para INDICE=1 até INDICE<=100 passo 1 faça
        escreva "Digite o nome do aluno";
leia LISTAALUNOS. NOME [INDICE];
escreva "Leia a nota do aluno";
leia LISTAALUNOS. NOTA [INDICE];
SOMA= SOMA+LISTAALUNOS. NOTA [INDICE];
escreva "A média atual é de " SOMA/INDICE;
fim para;
Fim;
```

Imagine uma ficha de cadastro de clientes. Em qualquer organização, as informações básicas de armazenamento seriam:

Código do Cliente:

Nome do Cliente:

Endereço:

Fone:

Estas informações podem ser usadas nos programas, e nestes, são referenciadas como registros. Por isso, quando você trabalhar com registros, o mais importante é definir como ele será constituído. Ou seja: quais informações são realmente importantes para compor o registro que vai armazenar os dados, para em seguida, declarar as variáveis do tipo registro. Veja o exemplo:

SINTAXE DA VARIÁVEL COMPOSTA HETEROGÊNEA – REGISTRO

Um registro pode ser declarado utilizando a seguinte sintaxe:

```
declare id_registro registro
(campo1 <tipobásico>; campo2 <tipobásico>;
...
campon <tipobásico>);

nome_da_variavel_registro id_registro;
```

onde:

id_registro é o nome associado ao tipo registro construído;
campo1, campo2, ...,campon são os nomes dos campos que irão compor o registro ;
<tipobásico> representa qualquer um dos tipos básicos de dados já conhecidos;
nome_da_variavel_registro é o nome da variável do tipo registro que será usada no algoritmo.

Exemplo 1: a ficha de cadastro de cliente vista anteriormente pode ser declarada, como mostrado a seguir:

```
declare
FICHACAD registro
        (CODIGO numérico;
         NOME literal;
 ENDERECO literal;
 SALDO numérico);

 CLIENTE FICHA;
```

Os dados do registro podem ser acessados pela variável cliente, através da sintaxe abaixo:

nomevar.nomedocampo

Exemplos:

Atribuindo valor à uma posição do REGISTRO:

leia CLIENTE.CODIGO; **leia** CLIENTE.NOME;
leia CLIENTE.ENDERECO;
...
CLIENTE.FONE 9210 5477;

Atribuindo valores no registro;

CLIENTE {128, “Simone da Luz”, “XV de Novembro, SP”, 9354 6987};

Mostrando valores do registro:

escreva CLIENTE.FONE;

3 ARQUIVOS

Os algoritmos desenvolvidos até agora foram armazenados em variáveis de memória consideradas voláteis. Isso significa que ao encerrar o aplicativo ou desligar a máquina, os dados são perdidos. Estes dados ficam armazenados em memória apenas enquanto o algoritmo é executado.

Para um armazenamento em tempo maior, a opção é o uso de arquivos, que se caracteriza por ser uma estrutura de dados que pode ser fisicamente alocada em outro meio de armazenamento não volátil (memória secundária), podendo ser lido ou gravado por um programa.

A diferença entre **registros** e **arquivos** consiste no fato de que o registro é a parte lógica da estrutura de dados. O arquivo é a parte física, composta por um ou mais registros. Os registros, por sua vez, são formados pelas unidades de dados chamadas de **campos**.

FIGURA 23 – ARQUIVOS

Fonte: Disponível em: <https://pt.dreamstime.com/fotos-de-stock-arquivos-3d-image17105793>. Acesso em: 7 dez. 2016.

IMPORTANTE

Um arquivo é formado por uma coleção de registros, e cada registro é composto por campos, e cada campo possui características específicas.

Para que você entenda melhor a aplicabilidade dos arquivos, imagine o sistema de uma biblioteca.

O cadastro de livros poderia ser exemplificado pela tabela a seguir:

Código do livro:
Título:
Autor:
Editora:

Note que a ficha pode armazenar informações de um único livro. O recurso para armazenar informações de vários livros ao mesmo tempo é o registro.

declare LIV **registro**
(CODIGO, ANO **numérico**;
TITULO, AUTOR, EDITORA **texto**);

LIVRO **LIV**;

Depois de definida a estrutura do registro, podemos criar o arquivo que vai armazenar o registro:

SINTAXE DE ARQUIVOS
declare
nome_arquivo **arquivo de** id_registro;

onde,

nome_arquivo nome do arquivo referenciado que será manipulado pelo algoritmo;
id_registro é o tipo de registro que foi anteriormente definido

Exemplo 2:

declare
BIBLIOTECA **arquivo de** LIV;

Verifique agora o exemplo completo com registro e arquivo!

declare LIV **registro**
(CODIGO, ANO **numérico**;
TITULO, AUTOR, EDITORA **texto**);

LIVRO **LIV**;

BIBLIOTECA **arquivo de** LIV;

Agora que você sabe definir e criar as duas estruturas, é necessário aprender os comandos para abrir o arquivo, fechar o arquivo, ler um registro, alterar campos de um registro, gravar um registro e apagar um registro.

É importante você lembrar que os arquivos permitem:

- consultar dados do arquivo
- incluir dados no arquivo
- modificar dados no arquivo
- excluir dados do arquivo

MANIPULAÇÃO DE ARQUIVOS

Verifique a seguir os comandos de manipulação dos arquivos.

ABRIR ARQUIVOS

Para se executar qualquer operação em um arquivo é necessário que ele esteja aberto para manipulação. A sintaxe para o comando de abertura do arquivo é a seguinte:

nome_arquivo = **abra** (“caminho do arquivo”, modo_de_abertura);

onde:
nome_arquivo é o nome do arquivo no algoritmo (declarado em **declare**);
caminho do arquivo é o ***path*** onde o arquivo está fisicamente;
modo_de_abertura representa a forma como o arquivo será aberto.

Modos de abertura dos arquivos:

r – abre um arquivo de texto onde poderão ser realizas apenas leituras

w – cria um arquivo de texto onde poderão ser realizadas apenas operações de escrita

r+ - abre um arquivo de texto onde poderão ser realizadas operações de leitura e de escrita

w+ - cria um arquivo de texto onde poderão ser realizadas operações de escrita e de leitura e outros.

FECHAR ARQUIVOS

Todo arquivo aberto e manipulado deve ser fechado para que as informações permaneçam armazenadas. A sintaxe do comando para fechamento do arquivo é a seguinte:

feche (nome_arquivo);
onde:
nome_arquivo é o nome do arquivo declarado no algoritmo (mesmo nome usado na abertura;

feche (BIBLIOTECA);

Quando um arquivo é fechado ele fica indisponível. Para acessar e manipular as informações, o mesmo deve ser aberto novamente.

LER REGISTROS DO ARQUIVO

A leitura de dados no arquivo requer que seja lido um registro por vez. A sintaxe do comando para leitura é a seguinte:

leia (nome_arquivo, nome_registro);

onde:
nome_arquivo é o nome do arquivo no algoritmo
nome_registro representa o nome da variável do tipo registro definido.

Após a execução deste comando, as informações lidas no registro do arquivo, serão armazenadas na variável nome_registro.

Note que a variável do tipo registro deve obrigatoriamente apresentar a mesma estrutura que foi definida para o registro do arquivo.

leia (BIBLIOTECA, FICHA);

GRAVAR REGISTROS NO ARQUIVO

A gravação do registro no arquivo usa o conteúdo de uma variável do tipo registro que formou o arquivo. A sintaxe é a seguinte:

grave (nome_arquivo, nome_registro);

onde:
nome_arquivo é o nome do arquivo no algoritmo;
nome_registro representa o nome da variável do tipo registro definido.

grave (BIBLIOTECA, FICHA);

PONTEIROS

Para ler ou procurar por um registro específico em um arquivo usamos os ponteiros, através do comando **procure**.

O primeiro registro do arquivo é sempre o de número zero. Sintaxe:

procure (nome_arquivo, posiçao);

onde:
nome_arquivo é o nome do arquivo no algoritmo;
posição é a posição em que o ponteiro ficará posicionado, no início, fim ou registro atual.

Exemplos:

procure (BIBLIOTECA, **INICIO**); // o ponteiro ficará posicionado no início do arquivo

procure (BIBLIOTECA, **FIM**); // o ponteiro ficará posicionado no final do arquivo

procure (BIBLIOTECA, **ATUAL**); // o ponteiro ficará posicionado no início do registro
// corrente

FINALIZAR UM ARQUIVO

Uma função chamada FDA (Fim De Arquivo), pode ser usada para saber se o arquivo chegou ao fim ou não, ou seja, se o último registro foi alcançado. Se a função retornar zero, o arquivo ainda não foi finalizado. A sintaxe para usarmos esta função é:

FDA (nome_arquivo);

Onde:
nome_arquivo representa o nome do arquivo no algoritmo.

Exemplo de uso do comando FDA que permite verificar o final do arquivo.
BIBLIOTECA = **abra** ("C:\biblioteca.dat", r+);

enquanto não FDA (BIBLIOTECA) **faça inicio**

comando1;
comando2;
fim;

REMOVER ARQUIVOS

Este comando apaga os arquivos. Ou seja, os arquivos são eliminados fisicamente. Sintaxe:

remove (nome_arquivo);

onde:

nome_arquivo indica o nome do arquivo que será removido

remove (BIBLIOTECA);

RESUMO DO TÓPICO 1

Neste tópico, você aprendeu:

- A diferença entre registros e arquivos.
- A manipular arquivos.
- Usar de forma conjunta registros e arquivos.

AUTOATIVIDADE

1 Explique a diferença entre registros e arquivos.

2 Quando os arquivos devem ser usados?

3 Quais etapas devem ser seguidas ao se optar por trabalhar com arquivos?

4 Imagine uma situação onde você tenha que controlar dados de funcionários como se fossem fichas cadastrais, conforme a figura a seguir:

NOME DO FUNCIONÁRIO	ENDEREÇO		
JOÃO DA SILVA	RUA DA SAUDADE, 100 CASA 1		
CPF	ESTADO CIVIL	DATA NASC	ESCOLARIDADE
000.001.002-	CASADO	01/01/1960	SUPERIOR
CARGO		SALÁRIO	DATA DE ADMISSÃO
GERENTE DE CONTAS		1000,00	10/05/1997

NOME DO FUNCIONÁRIO	ENDEREÇO		
MARIA SANTOS	AVENIDA DESPERTAR, 1000 CASA 101-A		
CPF	ESTADO CIVIL	DATA NASC	ESCOLARIDADE
100.201.202-	SOLTEIRA	01/01/1980	2º GRAU COMPLETO
CARGO		SALÁRIO	DATA DE ADMISSÃO
DIGITADORA		650,00	01/08/1999

Crie a estrutura acima em forma de registro.

TÓPICO 2

COMPLEXIDADE DE ALGORITMOS

1 INTRODUÇÃO

Como selecionar um algoritmo quando existem vários que solucionam o problema? Uma resposta pode ser, escolher um algoritmo de fácil entendimento, codificação e depuração ou então uma outra resposta pode ser, um algoritmo que faz uso eficiente dos recursos do computador. Qual a melhor solução? Como escolher? Vários critérios podem ser utilizados para escolher o algoritmo, mas vai depender das pretensões de utilização do algoritmo.

2 ANÁLISE DA COMPLEXIDADE DE ALGORITMOS

Pode-se selecionar o algoritmo somente para um experimento, ou será um programa de grande utilização, ou será utilizado poucas vezes e será descartado, ou ainda, terá aplicações futuras que podem demandar alterações no código. Para cada uma das respostas anteriores, pode-se pensar em uma solução diferente. Calcular o tempo de execução e o espaço exigido por um algoritmo para uma determinada entrada de dados é um estudo da complexidade de algoritmos. Para o cálculo de complexidade, pode-se medir o número de passos de execução em um modelo matemático denominado máquina de Turing, ou medir o número de segundos gastos em um computador específico.

Ao invés de calcular os tempos de execução em máquinas específicas, a maioria das análises conta apenas o número de operações "elementares" executadas. A medida de complexidade é o crescimento assintótico dessa contagem de operações. Por exemplo, em um algoritmo para achar o elemento máximo entre "n" objetos, a operação elementar seria a comparação das grandezas dos objetos, e a complexidade seria Análise de Algoritmos do Prof. Walteno Martins Parreira Jr., p. 3, uma função em "n" do número de comparações efetuadas pelo algoritmo, para valores grandes de "n". A complexidade assintótica de um algoritmo é que determina o tamanho de problemas que pode ser solucionado pelo algoritmo.

Se o algoritmo processa entradas de tamanho "n" no tempo c*n2, para alguma constante c, então dizemos que a complexidade de tempo do algoritmo é O(n2), onde se lê: " de ordem n2 ". Suponha que temos cinco algoritmos A1,... A5 com as seguintes complexidades de tempo:

Algoritmo	Complexidade de tempo
A_1	n
A_2	$n \log_2 n$
A_3	n^2
A_4	n^3
A_5	2^n

A finalidade de se fazer a análise de complexidade de um algoritmo é obter estimativas de tempos de execução de programas que implementam esse algoritmo.

A complexidade do algoritmo dá ideia do esforço computacional do programa, que é uma medida do número de operações executadas pelo programa. Uma das preocupações com a eficiência é com problemas que envolvem um considerável número de elementos. Se existir uma tabela com apenas dez elementos, mesmo o algoritmo considerado menos eficiente resolve o problema, no entanto, à medida que o número de elementos aumenta, o esforço necessário começa a fazer diferença de algoritmo para algoritmo.

O que se deseja na verdade é uma avaliação do desempenho do algoritmo independentemente da sua implementação, em função somente do número de instruções executadas para entradas determinadas. São consideradas somente a instruções preponderantes, isto é, as operações básicas para a execução do algoritmo. O número de vezes que essas operações são executadas é denominado Complexidade do Algoritmo.

Em geral, a complexidade de um algoritmo depende da entrada e esta é caracterizada pelo seu tamanho, por seus valores e também pela configuração dos dados. De forma intuitiva, sabemos que a complexidade depende da quantidade de dados que são processados e isso se traduz pelo tamanho da entrada: o número de operações executadas para localizar o último registro de uma lista com 1000 registros deve ser maior que o de uma lista com apenas 10 registros.

Muitas vezes, o valor da entrada determina o esforço, por exemplo, na execução de uma busca em uma lista linear não ordenada, o número de comparações executadas varia muito conforme o valor procurado ocorrer no primeiro registro, no final ou no meio da lista. Por exemplo, nos algoritmos que executam operações sobre listas lineares, a complexidade é expressa em função do tamanho da lista. Se n indica o número de registros, temos que a complexidade será uma função de n. Por outro lado, como os valores ou a configuração dos

dados de entrada são fatores que também interferem no processo, não é possível obter uma única função que descreva todos os casos possíveis.

Para cada possibilidade de entrada há uma função de complexidade do algoritmo. Reduzimos o estudo para alguns casos especiais: a) Pior Caso, caracterizado por entradas que resultam em maior crescimento do número de operações, conforme aumenta o valor de n; b) Melhor Caso, caracterizado por entradas que resultam em menor crescimento do número de operações, conforme aumenta o valor de n; c) Caso Médio, que retrata o comportamento médio do algoritmo, quando se consideram todas as entradas possíveis e as respectivas probabilidades de ocorrência (esperança matemática). Somente o estudo da complexidade de algoritmos permite a comparação de dois algoritmos equivalentes, isto é, desenvolvidos para resolver o mesmo problema.

2.1 NOTAÇÃO O – ANÁLISE DE ALGORITMOS

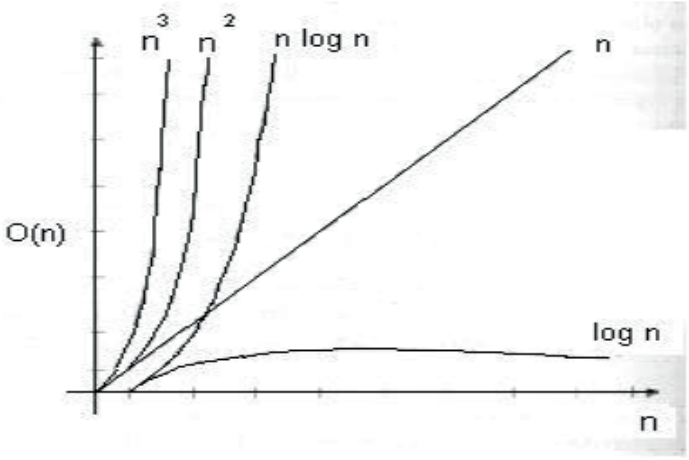
A notação O (leia-se ó grande, ou big oh) é utilizada para expressar comparativamente o crescimento assintótico representa a velocidade com que uma função tende ao infinito. No estudo de complexidade de algoritmos é mais interessante saber como se comporta essa função à medida que aumentarmos o valor de n, do que conhecer valores da função correspondentes a particulares valores de n.

Por exemplo, é mais importante saber que o número de operações executadas num algoritmo dobra se dobrarmos o valor de n, do que saber que para n igual a 100 são executadas 300 operações. Ao dizermos que uma função de complexidade f(n) é da ordem de n2 , queremos dizer que as duas funções, f(n) e n2 tendem ao infinito com a mesma velocidade, ou que têm o mesmo comportamento assintótico. Indicamos por f(n) = O(n2) em matemática essa informação é expressa por um limite:

$$\lim_{n \to \infty} \frac{f(n)}{n^2} = c \ (c > 0)$$

Se, por exemplo, outro algoritmo para o mesmo problema tem função de complexidade f1(n) = O(n), podemos comparar f(n) e f1(n) e, em consequência, comparar a eficiência dos programas que os implementam. Em um deles, o tempo de execução é linear e no outro, o tempo é quadrático.

Função	Significado (tamanho da entrada = n)
1	tempo constante – o número de operações é o mesmo para qualquer tamanho da entrada
n	tempo linear – se n dobra, o número de operações também dobra
n^2	tempo quadrático – se n dobra, o número de operações quadruplica
log n	tempo logarítmico – se n dobra, o número de operações aumenta de uma constante
nlog n	tempo n log n - se n dobra, o número de operações ultrapassa o dobro do tempo da entrada de tamanho n
2^n	tempo exponencial - se n dobra, o número de operações é elevado ao quadrado



Os resultados expressos em notação O devem ser interpretados com cuidados, pois indicam apenas que o tempo de execução do programa é proporcional a um determinado valor ou que nunca supera determinado valor; na verdade o tempo de execução pode ser inferior ao valor indicado e pode ser que o pior caso nunca ocorra.

Convenções para as expressões de O

Existem algumas convenções quanto à expressão de O:

- É prática comum escrever a expressão de O sem os termos menos significantes. Assim, em vez de O(n2 + nlog n + n), escrevemos simplesmente O(n2).

- É comum desconsiderar os coeficientes constantes. Em lugar de O(3n2), escrevemos simplesmente O(n2). Como caso especial desta regra, se a função é constante, por exemplo O(1024), escrevemos simplesmente O(1). Algumas expressões de O são tão frequentes que receberam denominações próprias:

Expressão	Nome
O(1)	Constante
O(log n)	Logarítmica
$O(\log^2 n)$	Log quadrado
O(n)	Linear
O(nlog n)	n log n
$O(n^2)$	Quadrática
$O(n^3)$	Cúbica
$O(2^n)$	Exponencial

Exemplo de análise da notação O

Para demonstrar os conceitos de eficiência, apresentaremos dois algoritmos que fazem a soma e a multiplicação de duas matrizes. Uma possibilidade para a soma dos elementos das duas matrizes m1 e m2 é colocar o resultado em m3, que pode ser feito da seguinte forma:

```
1   Linha ← 1
2   enquanto (linha ≤ tamanho_da_matriz ) faça
3       col ← 1
4       enquanto (col ≤ tamanho_da_matriz ) faça
5           m3 [ linha, col ] ← m1 [ linha, col ] + m2 [ linha, col ]
6           col ← col + 1
7       fim-enquanto
8       linha ← linha + 1
9   fim-enquanto
```

Analisando o trecho apresentado, vemos que o "enquanto" externo (linha 2) é dependente do tamanho da matriz. Para cada execução dessa estrutura de repetição, tem-se que executar o "enquanto" mais interno (linha 4), pode-se observar que ela depende do tamanho da matriz.

Pode-se observar na linha 2 a estrutura de repetição enquanto que percorre a matriz de 1 até N e para cada incremento da linha 2, a linha 4 percorre a matriz de 1 a N (portanto N execuções), onde N é o tamanho da matriz, logo N x N = N2.

Essa é uma situação clássica da repetição quadrática. A eficiência do algoritmo é O(n2). O segundo exemplo, supondo a multiplicação de duas matrizes m1 e m2, cujo resultado será colocado na matriz m3, podemos considerar o seguinte algoritmo:

```
1  coluna ← 1
2  enquanto (coluna ≤ tamanho_da_matriz ) faça
3      col ← 1
4      enquanto (col ≤ tamanho_da_matriz ) faça
5              m3 [ linha, col ] ← 0
6              k ← 1
7              enquanto (k ≤ tamanho_da_matriz ) faça
8                      m3 [ linha, col ] ← m3 [ linha, col ] + m1 [ linha, k ] * m2 [ k, col ]
9                      k ← k + 1
10             fim-enquanto
11             col ← col + 1
12     fim-enquanto
13     coluna ← coluna + 1
14     linha ← linha + 1
15 fim-enquanto
```

Fazendo uma análise deste algoritmo, nota-se a presença de três estruturas de repetição (linhas 2, 4 e 7) aninhadas. Como cada uma delas será iniciada e finalizada no primeiro elemento, observa-se a presença de uma repetição cúbica. A eficiência do algoritmo é O(n3).

Pode-se mostrar outro exemplo para ilustrar os passos a serem executados, para isto, calcular a notação O da seguinte função: f(n) = 4n2 + 2n + 3 primeiramente assumir que os coeficientes são iguais a 1, logo f(n) = n2 + n +1 em seguida são removidos os fatores de menor importância: f(n) = n2 finalmente, a notação será: O(f)n)) = O(n2).

2.2 ANÁLISE DE COMPLEXIDADE DA BUSCA LINEAR

Nos algoritmos de operações em listas o parâmetro utilizado na análise de complexidade é o tamanho da lista, indicado por n. A operação preponderante em operações de busca é a comparação de valores dos registros.

Vamos considerar dois casos neste estudo:

- W(n) = número de comparações no pior caso;
- A(n) = número de comparações no caso médio.
- A(n) = número de comparações no melhor caso.

O algoritmo que vamos analisar é o algoritmo de busca da primeira ocorrência de um valor x na lista A. Supomos que A tenha n registros. A sequência de instruções abaixo é o algoritmo “BuscaPrimeiraOcorrencia”:

```
1  j ← 1
2  enquanto ( A[ j ] ≠ x ) e (j < n) faça
3           j ← j + 1
4  fim-enquanto
5  se A[ j ] ≠ x
6    então sinal ← false
7    senão
8           sinal ← true
9           local ← j
10 fim-senão
```

A função de complexidade deve indicar, portanto, o número de vezes que é testada a condição (A[j] ≠ x).

2.3 PIOR CASO

A condição A[j] ≠ x ocorre como condição de controle do loop e, mais uma vez após o mesmo, na instrução de seleção. A outra condição que controla o loop, (j < n), limita o número de repetições do loop a n - 1. Isto é, se a primeira condição tiver sempre o valor false, o loop será executado n-1 vezes, porque j é inicializado com valor 1.

Entretanto, a condição que controla o loop é avaliada n vezes, que são as n -1 vezes que o loop é executado mais uma vez quando j atinge o valor n e torna a condição (j atinge o valor n e torna a condição)

$W(n) = n + 1$

(n: loop; $+1$: comando de seleção)

por outro lado, temos que $W(n) = O(n)$, pois $\lim_{n \to \infty} \frac{W(n)}{n} = 1$

2.4 CASO MÉDIO

Se uma lista contém n registros, as possíveis entradas numa operação de busca são correspondentes a um valor x procurado em que: x ocorre no primeiro registro, ou x ocorre no segundo registro, ou x ocorre no terceiro registro, e assim por diante, ou x ocorre no último registro, ou ainda, x não pertence à lista. São, portanto, n+1 casos diferentes de entradas possíveis.

A cada uma dessas entradas devemos associar um número de probabilidade de que a entrada ocorra. Vamos supor que todas as entradas sejam igualmente prováveis, portanto com probabilidade igual a 1/(n+1).

Assim, A(n) = p1xC1 + p2xC2 + ... + pn+1xCn+1

Pode-se concluir que: A(n) = O(log n)

2.5 MELHOR CASO

Ocorre quando a condição A[j] = x é satisfeita na primeira ou na segunda execução do algoritmo. Fazendo assim com que o loop seja executado 1 ou 2 vezes somente. Logo O(1)

Um exemplo numérico

Supondo uma lista com três elementos (10, 20 e 30) e chave de busca igual a 90. Montando os passos da execução do algoritmo:

Passo do algoritmo	Chave (x)	n	Valor de j	A[j]	Comparação: A[j] = x	Valor de sinal	Quantidade de comparações
1	90	3	1	10	falso		1
2	90	3	2	20	falso		2
3	90	3	3	30	falso		3
4	90	3	3	30	falso	falso	4

Pode-se observar que para o pior caso, onde foi percorrida toda a lista, foram realizadas quatro (4) comparações, o que significa (n+1) comparações, pois n = 3. Se a lista for maior, vai crescer linearmente o número de comparações.

Supondo uma lista com três elementos (10, 20 e 30) e chave de busca igual a 10. Montando os passos da execução do algoritmo:

Passo do algoritmo	Chave (x)	n	Valor de j	A[j]	Comparação: A[j] = x	Valor de sinal	Quantidade de comparações
1	10	3	1	10	verdade		1
2	10	3	2	10	verdade	verdade	2

Pode-se observar que para o melhor caso, onde a chave procurada está na primeira posição da lista, foram realizadas duas (2) comparações, uma no laço e outra no teste para confirmar se foi encontrado ou não a chave procurada, o que significa duas (2) comparações. Se a lista for maior, este valor ficará constante.

FONTE: Disponível em: <http://www.dainf.ct.utfpr.edu.br/~pbueno/Arquivos/Algoritmos.pdf>. Acesso em: 20 nov. 2016.

RESUMO DO TÓPICO 2

Neste tópico, você aprendeu que:

- Um problema pode ser resolvido por vários algoritmos e que escolher a solução mais adequada nem sempre é uma tarefa fácil.

- A finalidade de se fazer a análise de complexidade de um algoritmo é obter estimativas de tempos de execução de programas que implementam esse algoritmo.

- A complexidade do algoritmo dá ideia do esforço computacional do programa, que é uma medida do número de operações executadas pelo programa.

- Somente o estudo da complexidade de algoritmos permite a comparação de dois algoritmos equivalentes, isto é, desenvolvidos para resolver o mesmo problema.

AUTOATIVIDADE

1 Explique quando se usa o método da Análise de Complexidade da Busca Linear.

2 Cite as principais formas de análise de complexidade de algoritmos.

3 Disserte acerca das observações a serem feitas ao se determinar a complexidade de um algoritmo.

4 Explique do que depende a complexidade de um algoritmo.

TÓPICO 3

PRÁTICA DE ALGORITMOS

1 INTRODUÇÃO

Caro acadêmico! Após conhecer e incrementar seus conhecimentos em relação às teorias que sustentam a construção de algoritmos, você está apto a praticá-las.

Este tópico foi elaborado no intuito de estimular a sua percepção em relação à adequada solução para os problemas, fazendo uso de algoritmos.

Todos os problemas apresentados contemplam o enunciado seguido de uma forma de solução. Procure resolvê-los antes de consultar a solução proposta.

2 CONSTRUÇÃO DE ALGORITMOS

2.1 CONTROLE CONDICIONAL

Os comandos de condição servem para alterar o fluxo da execução do código-fonte de um programa. Neste caso, são impostas condições que serão satisfeitas por uma escolha. Normalmente, em linguagens de programação, utiliza-se o comando "if" no lugar do "senão" do pseudocódigo, e "else" no lugar do "senão".

1) Elaborar um algoritmo em pseudocódigo que efetue a leitura de um número inteiro e apresentar uma mensagem informando se o número é par ou ímpar.

```
     algoritmo "Par ou Ímpar"
     var
n: inteiro
     inicio
escreval("Insira um número inteiro: ")
leia(n)
se(n mod 2 = 0) entao
          escreval("O número: ",n," é par")
senao
```

```
            escreval("O número: ",n," é impar")
    fimse
        fimalgoritmo
```

2) Elaborar um algoritmo em pseudocódigo que efetue a leitura de um valor que esteja entre a faixa de 1 a 9. Após a leitura do valor fornecido pelo usuário, o programa deverá indicar uma de duas mensagens: "O valor está na faixa permitida", caso o usuário forneça o valor nesta faixa, ou a mensagem "O valor está fora da faixa permitida", caso o usuário forneça valores menores que 1 ou maiores que 9.

```
    algoritmo "Faixa Permitida"
    var
n :numerico
    inicio
escreval("Digite um valor: ")
leia(n)
se(n >= 1) e (n <= 9) entao
        escreval("O valor está na faixa permitida")
senao
        escreval("O valor não está na faixa permitida")
fimse
    fimalgoritmo
```

3) Elaborar um algoritmo em pseudocódigo que efetue a leitura do nome e do sexo de uma pessoa, apresentando como saída uma das seguintes mensagens: "Ilmo. Sr.", para o sexo informado como masculino, ou a mensagem "Ilma Sra.", para o sexo informado como feminino. Apresente na sequência da mensagem impressa o nome da pessoa.

```
    algoritmo "Sexo da pessoa"
    var
nome, sexo :texto;
    inicio
escreval("Digite o seu nome: ")
leia(nome)
escreval("Digite o seu sexo: ")
        leia(sexo)
se(sexo = "Masculino") entao
        escreval("Ilmo Sr. ",nome)
senao
se(sexo = "Feminino") entao
        escreval("Ilmo Sra. ",nome)
senao
        escreval("Digite um sexo válido")
fimse
fimse
fimalgoritmo
```

4) Elaborar um algoritmo em pseudocódigo que leia um número. Se positivo armazene-o em uma variável chamada "A", se for negativo, em uma variável chamada "B". No final mostrar o resultado das duas variáveis.

```
    algoritmo "Armazenamento"
    var
n,a,b :numerico
    inicio
escreval("Digite um número: ")
leia(n)
se(n >= 0) entao
        a:=  n
    escreval("O número :",a," é variável de A")
senao
                b := n
        escreval("O número :",b," é variável de B")
fimse
    fimalgoritmo
```

5) Tendo como dados de entrada a altura e o sexo de uma pessoa, construa um algoritmo em pseudocódigo que calcule peso ideal, utilizando as seguintes fórmulas:

Para homens: (72.7*h) – 58; Para mulheres: (62.1*h) – 44.7; Onde h equivale a altura da pessoa

```
algoritmo "Peso Ideal"
var
        a, p : numerico
        s : texto
inicio
escreval("Digite o seu sexo F ou M: ")
leia(s)
escreval("Digite a sua altura: ")
leia(a)
se(s = "F") entao
                p := (62.1 * a) - 44.7
escreval("Seu peso ideal é: ",p)
senao
se(s = "M") entao
                p :=  (72.7 * a) - 58
        escreval("Seu peso ideal é: ",p)
senao
        escreval("Digite um sexo válido")
fimse
fimse
fimalgoritmo
```

6) Fazer um algoritmo em pseudocódigo para ler quatro valores referentes a quatro notas escolares de um aluno e imprimir uma mensagem dizendo que o aluno foi aprovado, se o valor da média escolar for maior ou igual a 5. Se o aluno não foi aprovado, indicar uma mensagem informando esta condição. Apresentar junto com uma das mensagens o valor da média do aluno para qualquer condição.

```
algoritmo "Situação com média"
var
n1, n2, n3, n4, media :real
inicio
escreval("Digite a 1ª nota :")
leia(n1)
escreval("Digite a 2ª nota :")
leia(n2)
escreval("Digite a 3ª nota :")
leia(n3)
escreval("Digite a 4ª nota :")
leia(n4)
media<- (n1 + n2 + n3 + n4) / 4
        se(media >= 5) entao
                escreval("O aluno foi aprovado com média: ",media)
        senao
                escreval("O aluno não foi aprovado com média:
        ",media)
        fimse
fimalgoritmo
```

7) Fazer um algoritmo em pseudocódigo ler quatro valores referentes a quatro notas escolares de um aluno e imprimir uma mensagem dizendo que o aluno foi aprovado, se o valor da média escolar for maior ou igual a 7.0. Se o valor da média for menor que 7.0, solicitar a nota de exame, somar com o valor da média e obter nova média. Se a nova média for maior ou igual a 5, apresentar uma mensagem dizendo que o aluno foi aprovado em exame. Se o aluno não foi aprovado, indicar uma mensagem informando esta condição. Apresentar junto com as mensagens o valor da média do aluno, para qualquer condição.

```
algoritmo "Situação do Aluno"
var
n1, n2, n3, n4, media, nrecup, mrecup :real
inicio
escreval ("Digite a primeira nota: ")
leia (n1)
escreval ("Digite a segunda nota: ")
leia (n2)
escreval ("Digite a terceira nota: ")
leia (n3)
escreval ("Digite a quarta nota: ")
leia (n4)
```

```
media<-(n1+n2+n3+n4)/4
        se(media >=7)entao
                escreval (''O aluno está aprovado com média: '',media)
        senao
                escreval (''O aluno está de recuperação com média:
        '',media)
                escreval (''Digite a nota de recuperação: '')
        leia(nrecup) mrecup<-(media + nrecup)/2
                se(mrecup>=5)entao
                        escreval(''O aluno foi aprovado na recuperação
                com media: '',mrecup)
                senao
                        escreval(''O aluno não foi aprovado na
                recuperação com média: '',mrecup)
        fimse
fimse
fimalgoritmo
```

8) Fazer um algoritmo em pseudocódigo para ler o ano de nascimento de uma pessoa, calcular e mostrar sua idade e, também, verificar e mostrar se ela já tem idade para votar (16 anos ou mais) e para conseguir a Carteira de Habilitação (18 anos ou mais).

```
algoritmo ''Maior Idade''
var
ano, idade: inteiro
inicio
escreval(''digite seu ano de nascimento: '')
leia(ano)
idade :=  2016 - ano
escreval(''Sua idade é: '',idade)
        se(idade >= 18) entao
                escreval(''Ja tem idade para votar'')
                escreval(''Já tem idade para ter habilitação'')
        senao
                se(idade >= 16 ) entao
                        escreval(''Já tem idade npara votar'')
                senao
                        escreval(''Não pode votar e nem ter
                habilitação'')
                fimse
        fimse
fimalgoritmo
```

9) Escrever um algoritmo para ler três valores inteiros e escrever na tela o maior e o menor deles. Considere que todos os valores são diferentes.

```
algoritmo "Maior e Menor Valor"
var
v1, v2, v3: inteiro
inicio
escreval("Digite o primeiro valor: ")
leia(v1)
escreval("Digite o segundo valor: ")
leia(v2)
escreval("Digite o terceiro valor: ")
leia(v3)

se(v1 > v2) e (v1 > v3) e (v2 > v3) entao
        escreval("O maior valor é: ",v1," e o menor é: ",v3)
senao
        se(v1 > v2) e (v1 > v3) e (v3 > v2) entao
                escreval("O maior valor é: ",v1," e o menor é: ",v2)
senao
        se(v2 > v1) e (v2 > v3) e (v1 > v3) entao
                        escreval("O maior valor é: ",v2," e o menor
                é: ",v3)
senao
        se(v2 > v1) e (v2 > v3) e (v3 > v1) entao
                escreval("O maior valor é: ",v2," e o menor é: ",v1)
senao
        se(v3 > v1) e (v3 > v2) e (v1 > v2) entao
                escreval("O maior valor é: ",v3," e o menor é: ",v2)
senao
        se(v3 > v1) e (v3 > v2) e (v2 > v1) entao
                escreval("O maior valor é: ",v3," e o menor é: ",v1)
            fimse
fimse
fimse
fimse
fimse
fimse
fimalgoritmo
```

10) Escreva um algoritmo que leia três valores para os lados de um triângulo. O algoritmo deve verificar se o triângulo é equilátero (todos lados iguais), isósceles (dois lados iguais) ou scaleno (todos lados diferentes).

```
algoritmo "Tipo de triangulo"
var
l1, l2, l3: numérico
inicio
escreval("Digite o lado 1 do triângulo: ")
leia(l1)
        escreval("Digite o lado 1 do triângulo: ")
```

```
        leia(l2)
escreval("Digite o lado 1 do triângulo: ")
leia(l3)
        se(l1 = l2) e (l2 = l3) entao
                escreval("O triângulo é equilátaro")
        senao
        se(l1 <> l2) e (l1 <> l3) e (l2 <> l3) entao
                escreval("O triângulo é escaleno")
        senao
        se(l1 = l2) ou (l1 = l3) ou (l2 = l3) entao
                escreval("O triângulo é isósceles")
fimse
fimse
fimse
fimalgoritmo
```

11) Escrever um algoritmo que leia valores inteiros em duas variáveis distintas.

- Se o resto da divisão da primeira pela segunda for 1 mostre a soma dessas variáveis mais o resto da divisão;
- Se for 2 escreva se o primeiro e o segundo valor são pares ou ímpares;
- Se for igual a 3 multiplique a soma dos valores lidos pelo primeiro;
- Se for igual a 4 divida a soma dos números lidos pelo segundo, se este for diferente de zero.
- Em qualquer outra situação mostre o quadrado dos números lidos.

```
algoritmo "Dois Valores"
var
v1, v2, soma, mult: inteiro
divisao, q1, q2: real
inicio
escreval("Digite o primeiro número: ")
leia(v1)
escreval("Digite o segundo número: ")
leia(v2)
se(v1 mod v2 = 1) entao
        soma<- v1 + v2 + 1
        escreval("A soma de v1 e v2 mais o resto é: ",soma)
senao
        se(v1 mod v2 = 2) entao
                se(v1 mod 2 = 0) e (v2 mod 2 = 0) entao
                        escreval("v1 e v2 são par")
                senao
                        se(v1 mod 2 <> 0) e (v2 mod 2 <> 0) entao
                                escreval("v1 e v2 são impar")
                        fimse
                fimse
        senao
```

```
                se(v1 mod v2 = 3) entao
        mult :=  (v1 + v2) * v1
        escreval("A soma de v1 e v2 multiplicado por v1 é: ",mult)
        senao
                se(v2 = 0) entao
                escreval("Não haver divisão por zero")
        senao
                se(v1 mod v2 = 4) entao
                        divisão:=  (v1 + v2) / v2
                escreval("A soma de v1 e v2 dividido por v2 é:
        ",divisao)
        senao
                q1:=  v1^2
                q2:=  v2^2
                escreval("O quadrado do número 1 é: ",q1)
                escreval("O quadrado do número 2 é: ",q2)
        fimse
fimse
fimse
fimse
fimse
fimalgoritmo
```

12) Zezinho comprou um microcomputador para controlar o rendimento diário de seu trabalho como pescador. Toda vez que ele traz um peso de peixes maior que o estabelecido pelo regulamento de pesca do estado de Santa Catarina (50 quilos) deve pagar uma multa de R$ 4,00 por quilo excedente. Zezinho precisa que você faça um algoritmo que leia a o peso de peixes e verifique se há excesso. Se houver, o excesso e o valor da multa que Zezinho deverá pagar. Caso contrário mostrar uma mensagem que ele não deve pagar nada.

```
algoritmo "Controle de peso"
var
multa, peso, pesopermitido, excesso : numerico

inicio
escreval("Digite o peso permtido: ")
leia(pesopermitido)
escreval("Digite o peso da pesca")
leia(peso)
se(pesopermitido< peso) entao
        excesso :=  peso - pesopermitido
                multa :=  excesso * 4
        escreval("Houve excesso de: ",excesso," Kg, com multa de:
",multa)
senao
```

```
        escreval("O pescador não vai pagar nada")
fimse
fimalgoritmo
```

13) Faça um algoritmo que receba o valor do salário de uma pessoa e o valor de um financiamento pretendido. Caso o financiamento seja menor ou igual a 5 vezes o salário da pessoa, o algoritmo deverá escrever "Financiamento Concedido"; senão, ele deverá escrever "Financiamento Negado". Independente de conceder ou não o financiamento, o algoritmo escreverá depois a frase "Obrigado por nos consultar."

```
algoritmo "Financiamento"
var
sala, financ: real
inicio
        escreval("Digite o valor do salário: ")
leia(sala)
escreval("Digite o valor do financiamento pretendido: ")
leia(financ)
se(financ<= 5 * sala) entao
        escreval("Financiamento concedido, obrigado por nos
consultar")
senao
        escreval("Financiamento negado, obrigado por nos consultar")
fimse
fimalgoritmo
```

14) A Secretaria de Meio Ambiente que controla o índice de poluição mantém 3 grupos de indústrias que são altamente poluentes do meio ambiente. O índice de poluição aceitável varia de 0 (zero) até 0,25. Se o índice sobe para 0,3 as indústrias do 1º grupo são intimadas a suspenderem suas atividades, se o índice crescer para 0,4 as indústrias do 1º e 2º grupo são intimadas a suspenderem suas atividades, se o índice atingir 0,5 todos os grupos devem ser notificados a paralisarem suas atividades. Faça um algoritmo que leia o índice de poluição medido e emita a notificação adequada aos diferentes grupos de empresas.

```
algoritmo "Controle Ambiental"
var
indice: numerico
inicio
escreval("Digite o indice de poluição: ")
leia(indice)

se(indice>= 0.5) entao
        escreval("Todos os grupos devem suspender suas atividades")
senao
        se(indice>= 0.4) entao
                escreval("1º e 2º grupos devem suspender suas
```

```
        atividades")
        senao
                se(indice>= 0.3) entao
                        escreval("Apenas o 1º grupo deve suspender
                suas atividades")
        senao
                escreval("Todas os grupos podem manter as
        atividades")
        fimse
fimse
fimse
fimalgoritmo
```

15) Faça um programa que leia 4 valores X, A, B e C onde X é um número inteiro e positivo e A, B ,e C são quaisquer valores reais. O programa deve escrever os valores lidos e:

- se X = 1, escrever os três valores A, B e C em ordem crescente;
- se X = 2, escrever os três valores A, B e C em ordem decrescente;
- se X = 3, escrever os três valores A, B e C de forma que o maior valor fique entre os outros dois;
- se X não for um dos três valores acima, dar uma mensagem indicando isso.

```
algoritmo "Manipulando valores"
var
        x: inteiro
        a, b, c, menor, meio, maior: real
inicio
escreval("Digite o valor de A: ")
leia(a)
        escreval("Digite o valor de b: ")
        leia(b)
        escreval("Digite o valor de C: ")
        leia©
se (a > b) e (b > c) entao
                maior := a
                meio := b
                menor := c
senao
        se(a > c) e (c > b)  entao
                maior := a
                        meio := c
                        menor := b
        senao
        se(b > a) e (a > c) entao
                        maior := b
                        meio := a
```

```
                        menor := c
                senao
                        se(b > c) e (c > a) entao
                                maior := b
                                meio := c
                                menor := a
                        senao
                                se(c > a) e (a > b) entao
                                        maior := c
                                        meio := a
                                        menor := b
                                senao
                                        se(c > b) e (b > a) entao
                                                maior := c
                                                meio := b
                                                menor := a
                                        fimse
                                fimse
                        fimse
                fimse
                fimse
                escreval("Digite o valor de X: ")
                leia(x)
                escolha x
                        caso 1
                        escreval(menor, meio, maior)
                        caso 2
                        escreval(maior, meio, menor)
                        caso 3
                        escreval(menor, maior, meio)
                        outrocaso
                escreval("Não é nem 1, 2 ou 3")
                fimescolha
        fimalgoritmo
```

16) Elabore um algoritmo que informe se um dado ano é ou não bissexto. Obs.: um ano é bissexto se ele for divisível por 400 ou se ele for divisível por 4 e não por 100.

```
algoritmo "Ano bissexto"
var
ano: inteiro

inicio
escreval("Digite um ano: ")
leia(ano)
se((ano mod 4 = 0) e (ano mod 100 <> 0)) ou (ano mod 400 = 0) entao
```

```
        escreval("O ano: ",ano," é bissexto")
senao
        escreval("O ano: ",ano," não é bissexto")
fimse
fimalgoritmo
```

17) Faça um algoritmo que determine o maior entre N números. A condição de parada é a entrada de um valor 0, ou seja, o algoritmo deve processar o maior até que a entrada seja igual a 0(ZERO).

```
algoritmo "Maior número"
var
n, maior: numerico
inicio
maior :=  0
repita
escreval("Digite um número positivo maior que zero: ")
leia(n)
se(n > maior) entao
        maior:= n
fimse
        ate n = 0
escreval("O maior número é: ",maior)
fimalgoritmo
```

2.2 CASE

O comando "case" normalmente é utilizado para substituir a utilização de vários comandos "if" encadeados, ou "se" no pseudocódigo. Neste comando, o conteúdo de uma variável é comparado com um valor constante, e caso esta comparação seja verdadeira, um determinado comando é executado.

1) Fazer um algoritmo em pseudocódigo ler o código de um determinado produto e mostrar a sua classificação. Utilize a seguinte tabela como referência:

```
algoritmo "pesquisa por codigo"
var
codigo, outro :inteiro
cla:  texto
inicio
escreva("Digite o código da pesquisa: ")
leia(codigo)
escolhacodigo
caso 1
        escreval("Alimento não perecível")
caso 2, 3, 4
```

```
            escreval("Alimento perecível")
        caso 5, 6
            escreval("Vestuário")
        caso 7
            escreval("Higiene pessoal")
        caso 8, 9, 10, 11, 12, 13, 14, 15
            escreval("Limpeza e ultensíliosdomesticos")
        outrocaso
        leia(outro)
        escreva ("Inválido")
        fimescolha
        fimalgoritmo
```

2) Escrever um algoritmo para ler dois valores e uma das seguintes operações a serem executadas (codificadas da seguinte forma: 1 – Adição, 2 – Subtração, 3 – Multiplicação e 4 – Divisão). Calcular e escrever o resultado dessa operação sobre os dois valores lidos.

```
        algoritmo "Codificação"
        var
        v1, v2, cod, adicao, mult: inteiro
        divisao, subtracao: numerico
        inicio
        escreval("Digite o primeiro valor: ")
        leia(v1)
        escreval("Digite o segundo valor: ")
        leia(v2)
        escreval("-------------- x ----------- x ------------ x ------------------
---")
        repita
            escreval("Digite o código: ")
            leia(cod)
            escolhacod
            caso 1
                adicao:= v1 + v2
                        escreval("A soma é: ",adicao)
            caso 2
                subtracao:= v1 - v2
                escreval("A subtração é: ",subtracao)
            caso 3
                mult := v1 * v2
                escreval("A multiplicação é: ",mult)
            caso 4
                divisao:= v1 / v2
                escreval("A divisão é: ",divisao)
            outrocaso
```

```
                    escreval("Digite um código válido")
            fimescolha
            ate (cod>= 1) e (cod<= 4)
fimalgoritmo
```

3) Escreva um algoritmo que mostre as opções de um *menu* que simula uma calculadora.

```
Algoritmo Menu_Calculadora;

Var escolha:Numérico;

Início

Escreva('Escolha uma das opções');
Leia(escolha);
Caso escolha De
        1: Escreva('Você escolheu a soma!')
        2: Escreva('Você escolheu o produto!')
        3: Escreva('Você escolheu a subtração!')
        4: Escreva('Você escolheu a divisão!')
Fim;
```

2.3 PARA FACA

O comando "para faca" é um dos comandos de repetição utilizados no pseudocódigo e nas linguagens de programação, cujo funcionamento se depara em repetir um comando por um determinado número de vezes. Este comando, em linguagens de programação, normalmente é representado pelo comando "for".

1) Escreva um algoritmo (pseudocódigo) que coloque os números de 1 a 100 na tela na ordem inversa (começando em 100 e terminando em 1).

```
variáveis
inteiro: Numero
início
        para Numero ← 100 até 1 passo -1 repetir
        Escreva Numero
 fim para
Fim
```

2) Faça um algoritmo (pseudocódigo) para listar todos os múltiplos positivos do número 7 menores ou iguais a 100.

Solução 1

```
variáveis
inteiro: N
início
para N ← 7 até 100 passo 7 repetir
        Escreva N
fim para
Fim
```

Solução 2

```
variáveis
inteiro: N
início
para N ← 1 até 100 repetir
         se Mod( N, 7 ) = 0 então
                  Escreva N
         fim se
 fim para
Fim
```

3) Calcule a soma dos números pares de 50 a 100

```
Algoritmo Soma_Pares
Var
  soma,n : inteiro
Início
  soma<-0
  para n de 50 até 100 faca
        se(n mod 2) = 0 entao
                soma<-soma+n
        fimse
  fimpara
  escreva(soma)
Fimalgoritmo
```

4) Escrever um algoritmo para ler cinco valores inteiros, calcular a sua média, e escrever na tela os números que são superiores à média.

```
algoritmo "Média e comparação"
var
        notas: vetor [1..5] de inteiro
        i, maior_media, soma: inteiro
        media: real
inicio
        soma := 0;
escreval("Digite as 5 notas: ")
para i de 1 ate 5 faca
```

```
        leia(notas[i])
        soma :=  soma + notas[i]
        media :=  soma / 5
fimpara
escreval("A Soma das notas é: ",soma)
escreval("A média das notas é: ",media)
escreval("Os valores maiores que a média são: ")

para i de 1 ate 5 faca
        se(notas[i] > media) entao
                maior_media<- notas[i]
                escreval(maior_media)
        fimse
fimpara
fimalgoritmo
```

5) Escrever um algoritmo para ler a quantidade de horas/aula de dois professores e o valor por hora recebido por cada um. Mostrar na tela qual dos professores tem salário total maior.

```
algoritmo "Média e comparação"
var
h_a, valor: vetor [1..2] de real
prof: vetor [1..2] de literal
i: inteiro
salario, maior: real
inicio
maior :=  0
salario :=  0
para i de 1 ate 2 faca
        escreval("Digite o nome do professor ",i)
        leia(prof[i])
        escreval("Digite a quantidade de Hora/Aula do professor ",i)
        leia(h_a[i])
        escreval("Digite o valor da Hora/Aula do professor ",i)
        leia(valor[i])
        escreval("----------- x ------- x ------------- x ----------")
fimpara

para i de 1 ate 2 faca
        salario:=  h_a[i] * valor[i]
        escreval("O salário do professor ",i," é: ",salario)
        se( salario > maior) entao
                maior :=  salario
        fimse
fimpara
        escreval("O maior salário é o do professor que ganha: ",maior)
fimalgoritmo
```

6) Escreva um algoritmo que leia um número inteiro. Se o número lido for positivo, escreva uma mensagem indicando se ele é par ou ímpar. Se o número for negativo, escreva a seguinte mensagem "Este número não é positivo".

```
algoritmo "Positivo ou Negativo"
var
n: inteiro
inicio
escreval("Digite um número inteiro: ")
repita
        leia(n)
        se( n = 0 ) entao
                escreval("Digite um número diferente de zero")
        senao
                se( n > 0 ) e ( n mod 2 = 0 ) entao
                        escreval("O número ",n," é positivo e par")
        senao
                se( n > 0 ) e ( n mod 2 <> 0 ) entao
                        escreval("O número ",n," é positivo e impar")
                senao
                escreval("O número não é positivo")
        fimse
        fimse
        fimse
ate n > 0
fimalgoritmo
```

7) Faça um algoritmo que conte de 1 a 100 e a cada múltiplo de 10 emita uma mensagem: "Múltiplo de 10".

```
algoritmo "Multiplos de 10"
var
        n: vetor[1..100] de inteiro
        i: inteiro
inicio
escreval("Os multiplos de 10 de 1 a 100 são:")
para i de 1 ate 100 faca
        se(i mod 10 = 0) entao
                escreval(i)
        fimse
fimpara
fimalgoritmo
```

8) Elabore um algoritmo que gere e escreve os números ímpares entre números entre 100 e 200.

```
algoritmo "Numeros Impares de 100 a 200"
var
        n: vetor[100..200] de inteiro
        i, impar: inteiro
inicio
escreval("Os números impares de 100 a 200 são:")
para i de 100 ate 200 faca
        se(i mod 2 <> 0) entao
                escreval(i)
        fimse
fimpara
fimalgoritmo
```

9) Construa um algoritmo que leia 50 valores inteiros e positivos e:
· Encontre o maior valor.
· Encontre o menor valor.
· Calcule a média dos números lidos.

```
algoritmo "Maior menor e média"
var
        valor: vetor[1..50] de inteiro
        i, menor, maior, soma: inteiro
        media: real
inicio
        maior :=  0
        menor :=  10000
escreval("Digite 50 números inteiros e positivos: ")
para i de 1 ate 50 faca

        leia(i)
        se(i > maior) entao
                maior := i
        se(i < menor) entao
                menor :=  i
        fimse
        fimse
        soma  := soma + i
        media := soma / 50
fimpara
escreval("O maior valor é: ",maior)
escreval("O menor valor é: ",menor)
escreval("A média dos valores é: ",media)
fimalgoritmo
```

10) A conversão de graus Farenheit para graus Centígrados é obtida por: C ← (F-32)*5/9 Fazer um algoritmo que calcule e escreva uma tabela em centígrados em função de graus Farenheit, que variam de 50 a 150 de 2 em 2.

```
algoritmo "De Farenheit para Centígrados"
var
graus: vetor[50..150] de inteiro
c: real
i: inteiro
inicio
para i de 50 ate 150 passo 2 faca
                c := (i - 32) * 5 / 9
        escreval(i," F é igual a ",c," C")
fimpara
fimalgoritmo
```

2.4 ENQUANTO FACA

O comando "enquanto faca" é muito utilizado no pseudocódigo e nas linguagens de programação, sendo um comando de repetição. Esta estrutura repete a execução de um comando enquanto uma determinada condição esteja sendo satisfeita. Normalmente este pseudocódigo é representado por "while" em linguagens de programação.

1) Supondo que a população de um país A seja da ordem de 90.000.000 habitantes com uma taxa anual de crescimento de 3% e que a população de um país B seja aproximadamente de 200.000.000 de habitantes com uma taxa anual de crescimento de 1,5%. Fazer um algoritmo que calcule e escreva o número de anos necessários para que a população do país A ultrapasse ou iguale a população do país B, mantidas essas taxas de crescimento.

```
algoritmo "Crescimento anual"
var
        a, b: real
        ano: inteiro
inicio
a := 90000000
b := 200000000
ano := 0
enquanto a <= b faca
        a :=  a * 0.03
                b := b * 0.015
        ano :=  ano + 1
fimenquanto
escreval("A quantida de anos para A chegar até B é: ",ano)
fimalgoritmo
```

2) Refaça o exercício 29, lendo as taxas e as populações.

```
algoritmo "Crescimento anual"
var
        a, b, taxaa, taxab: real
        ano: inteiro
inicio
ano := 0
escreval("Digite a população do país A: ")
leia(a)
escreval("Digite a taxa do país A: ")
leia(taxaa)
escreval("---------- x xx----------")
escreval("Digite a população do país B: ")
leia(b)
escreval("Digite a taxa do país B: ")
leia(taxab)
enquanto a <= b faca
        a := a * (taxaa / 100)
        b := b * (taxab / 100)
        ano :=  ano + 1
fimenquanto
escreval("A quantida de anos para A chegar até B é: ",ano)
fimalgoritmo
```

3) Calcule a soma dos números pares de 50 a 100

```
Algoritmo Soma_Pares;
Var
soma, par :Numérico;
Início
soma:=0;
par:=50;
Enquanto par <= 100 Faça
        soma := soma+par;
        par := par+2;
Fim enquanto;
        Escreva(soma);
Fim.
```

2.5 CONTADORES E ACUMULADORES

Contador pode ser definido por uma variável utilizada para contar a quantidade de vezes que a execução do programa passou por uma determinada linha do código-fonte, ou seja, serve para contar a quantidade de vezes que um evento ocorre. Um exemplo de contador pode ser o código "contador := contador +

1". Acumulador pode ser definido por uma variável utilizada para realizar cálculos, a qual recebe valores que são somados a ela mesma. Além disso, acumuladores podem ser chamados de somadores. Um exemplo de acumulador pode ser a linha de código "valor := valor + 100".

1) Em um curso de programação, um programador cometeu 1 erro em seu primeiro programa, 2 erros no segundo, 4 erros no terceiro e assim sucessivamente. Ele está cometendo por programa duas vezes a quantidade de erros do programa anterior. Se o curso dura treze semanas e o programador realiza dois programas por semana, calcular o número de erros que ele espera cometer em seu programa final.

```
Algoritmo Erro_Programador;
Var
Erros, Programas, contador :Numérico;
Início
Erros := 1;
Programas := 2 * 13;
Para contador := 2 Até Programas Faça
        Erros := Erros * 2;
      Fim para;
       Escreva('Número de erros ==>',Erros);
Fim.
```

2)Faça um algoritmo que leia a nota de 20 alunos, calcule e mostre a média das notas.

```
Algoritmo média;
Var
soma, media, nota, contador :Numérico;

Início
soma := 0;
Para contador := 0 Até 20 Faça
        Escreva('Digite as 20 notas dos alunos 0➔ ');
        Leia(nota);
        soma := soma+nota;
Fim para;
media := soma+nota;
Escreva('A média é '||media ');
Fim.
```

3) Escreva um algoritmo que leia 100 números e informe quais são múltiplos de três.

```
Algoritmo múltiplo_de_três;
Var num, contador :Numérico;
```

```
Início
Para contador := 1 Até 100 Faça
        Escreva('Digite os 100 números ');
        Leia(num);
        Se (num MOD 3 = 0 ) Então
                Escreva(num);
              Fim se;
Fimpara;
Fim.
```

4) Fazer um algoritmo que:

- Leia um número indeterminado de linhas contendo cada uma a idade de um indivíduo.
- Calcule e escreva a idade média deste grupo de indivíduos.

```
algoritmo "Idade média"
var
idade, soma, cont: inteiro
media: numerico
inicio
cont := 1
repita
        escreval("Digite uma relação de idades: ")
        leia(idade)
        soma := soma + idade
        cont := cont + 1
ate idade = 0
media :=  soma / cont
escreval("A soma das idades é: ",soma)
escreval("A quantidade é: ",cont)
escreval("A média das idades é de: ",media)
fimalgoritmo
```

5) Tem-se um conjunto de dados contendo a altura e o sexo (masculino, feminino) de 50 pessoas. Fazer um algoritmo que calcule e escreva:

- A maior e a menor altura do grupo.
- A média de altura das mulheres.
- O número de homens.

```
algoritmo "Conjunto de dados"
var
        cont, soma, nhomens, nmulher: inteiro
        altura: real
        alturaM, alturaH, media, maior, menor: real
        sexo: texto
inicio
```

```
cont := 0
nhomens := 0
nmulher := 0
escreval("Digite a altura e o sexo de 5 pessoas:")
repita
        escreva("Altura: ")
        leia(altura)
        escreva("Sexo: ")
        leia(sexo)
        escreval("---------- x xx ----------")
        cont :=  cont + 1
        se(altura > maior ) entao
                maior := altura
                se(altura < menor) entao
                        menor :=  altura
                fimse
        fimse
        se(sexo = "m") entao
                nmulher := nmulher + 1
atecont = 5
fimalgoritmo
```

6) Um comerciante deseja fazer o levantamento do lucro das mercadorias que ele comercializa. Para isto, mandou digitar cada mercadoria com o nome, preço de compra preço de venda das mercadorias. Fazer um algoritmo que:

- Determine e escreva quantas mercadorias proporcionam:
- Lucro menor 10 %.
- Lucro entre 10% e 20%.
- Lucro maior que 20%.
- Determine e escreva o valor total de compra e de venda de todas as mercadorias, assim como o lucro total.

```
algoritmo "Levantamento de lucro"
var
        nome, x: literal
        precompra, prevenda, lucro, menorl, mediol, maiorl, totalc,
totalv, lucrog: numerico

inicio
repita
        escreva("Produto: ")
        leia(nome)
        escreva("Preço de compra: ")
        leia(precompra)
        escreva("Preço de venda: ")
        leia(prevenda)
        escreva("Cadastrar outro produto,(s/n)?: ")
```

```
        leia(x)
        escreval("---------- x xx ----------")
        totalc := totalc + precompra
        totalv :=  totalv + prevenda
        lucro := prevenda - precompra
        lucrog := lucrog + lucro
        se (lucro < precompra * 0.1)  entao
                menorl := menorl + 1
        senao
                se(lucro >= precompra * 0.1) e (lucro <= precompra *
        0.2) entao
                mediol := mediol + 1
        senao
                se(lucro >precompra * 0.2) entao
                        maiorl := maiorl + 1
                fimse
        fimse
            fimse
    ate x = "n"
escreval("Nº de produtos com 10% de lucro: ",menorl)
escreval("Nº de produtos entre 10% e 20% de lucro: ",mediol)
escreval("Nº de produtos com mais de 20% de lucro: ",maiorl)
escreval("O valor total de compra é: ",totalc)
escreval("O valor total de venda é: ",totalv)
escreval("O Lucro geral é: ",lucrog)
fimalgoritmo
```

7) Um determinado material radioativo perde metade de sua massa a cada 50 segundos. Dada a massa inicial, em gramas, fazer um programa que determine o tempo necessário para que essa massa se torne menor que 0,5 grama. Escreva a massa inicial, a massa final e o tempo calculado em horas, minutos e segundos.

```
algoritmo "Material Radioativo"
var
        massa, massai, massaf: real
        contador, hora, minuto, segundo: inteiro
inicio
contador := 0
massaf := 0
escreval("Digite a massa inicial: ")
leia(massa)
massai := massa
enquanto (massa >= 0.5) faca
        massa := massa / 2
        contador :=- contador + 50
fimenquanto
```

```
massaf := massa
segundo := contador mod 60
minuto := contador div 60
hora := minuto div 60
escreval("A massa inicial é: ",massai)
escreval("A massa final é: ",massaf)
escreval("O tempo gasto foi: ",hora," h ",minuto," m ",segundo," s")
fimalgoritmo
```

8) Uma certa empresa fez uma pesquisa para saber se as pessoas gostaram ou não de um novo produto lançado no mercado. Para isso, forneceu-se o sexo do entrevistado e a sua resposta (sim ou não). Sabendo-se que foram entrevistadas 2.000 pessoas, fazer um algoritmo que calcule e escreva:

- O número de pessoas que responderam sim.
- O número de pessoas que responderam não.
- A percentagem de pessoas do sexo feminino que responderam sim.
- A percentagem de pessoas do sexo masculino que responderam não.

```
algoritmo "Pesquisa"
var
        sexo, res: literal
        cont, conts, contn, nf, nm: inteiro
        pf, pm: numerico
inicio
enquanto cont < 2000 faca
        escreva("Qual o seu sexo(m/f)?: ")
        leia(sexo)
        escreva("O produto está aprovado(s/n)?: ")
        leia(res)
        escreval("---------- x xx ----------")
        se(res = "s") e (sexo = "f") entao
                nf := nf + 1
        senao
                se(res = "n") e (sexo = "m") entao
                        nm :=  nm + 1
                fimse
        fimse
        se(res = "s") entao
                conts := conts + 1
        senao
                contn := contn + 1
        fimse
                cont := cont + 1
fimenquanto
pf := nf / cont * 100
pm := nm / cont * 100
```

```
escreval("O numero de reposta igual a sim é: ",conts)
escreval("O numero de reposta igual a não é: ",contn)
escreval("A percentagem de sim feminino é: ",pf)
escreval("A percentagem de não masculino é: ",pm)
fimalgoritmo
```

9) Foi feita uma pesquisa de audiência de canal de TV em várias casas de uma certa cidade, num determinado dia. Para cada casa visitada, é fornecido o número do canal (4, 5, 7, 12) e o número de pessoas a que o estavam assistindo naquela casa. Se a televisão estivesse desligada, nada era anotado, ou seja, esta casa não entrava na pesquisa. Fazer um algoritmo que:

- Leia número de casas pesquisadas.
- Calcule a percentagem de audiência para cada emissora.
- Escreva o número de cada canal e a sua respectiva percentagem.

```
algoritmo "Pesquisa 2"
var
        ncasa, ncanal, p_a, cont, c4, c5, c7, c12: inteiro
        tv_l: texto
        pc4, pc5, pc7, pc12: numerico
inicio
tv_l :=  "n"
escreval("Televisão desligada (s/n)?: ")
leia(tv_l)
enquanto tv_l = "n" faca
        cont := cont + 1
        escreva("Qual canal estão assistindo? ")
        leia(ncanal)
        escreva("Quantas pessoas estão assistindo? ")
        leia(p_a)
        escreval("------------- x xx ------------")
        se(ncanal = 4) entao
                c4 := c4 + 1
        senao
                se(ncanal = 5) entao
                        c5 := c5 + 1
        senao
        se(ncanal = 7) entao
                c7 := c7 + 1
        senao
        se(ncanal = 12) entao
                c12:= c12 + 1
        fimse
        fimse
        fimse
fimse
          pc4 := c4 / cont * 100
```

```
            pc5 := c5 / cont * 100
            pc7 := c7 / cont * 100
            pc12 := c12 / cont * 100
        escreva("Encerra pesquisa(s/n)? ")
        leia(tv_l)
fimenquanto

escreval("O número de casas pesquisadas é: ",cont)
escreval("A audiência da emissora do canal 4 é: ",c4)
escreval("A audiência da emissora do canal 5 é: ",c5)
escreval("A audiência da emissora do canal 7 é: ",c7)
escreval("A audiência da emissora do canal 12 é: ",c12)
escreval("A percentagem do canal 4 é: ",pc4," %")
escreval("A percentagem do canal 5 é: ",pc5," %")
escreval("A percentagem do canal 7 é: ",pc7," %")
escreval("A percentagem do canal 12 é: ",pc12," %")

fimalgoritmo
```

10) Uma Universidade deseja fazer um levantamento a respeito de seu concurso vestibular. Para cada curso é fornecido o seguinte conjunto de valores: um código do curso;

- Número de vagas.
- Número de candidatos do sexo masculino.
- Número de candidatos do sexo feminino.

Fazer um programa que:

- calcule e escreva, para cada curso, o número de candidatos por vaga e a percentagem de candidatos do sexo feminino (escreva também o código correspondente do curso);
- determine o maior número de candidatos por vaga e escreva esse número juntamente com o código do curso correspondente (supor que não haja empate);
- calcule e escreva o total de candidatos.

```
algoritmo "semnome"
var
        nvagas, cm, cf, tc: inteiro
        pcf, cpv, mncv: real
        cadastrar: texto
        curso, cod_c: caracter
inicio
cadastrar :=  "s"
mncv :=  0
        tc := 0
enquanto cadastrar = "s" faca
        escreva("O código do curso: ")
```

```
                leia(cod_c)
                escreva("O nº de vagas do curso: ")
                leia(nvagas)
                escreva("O nº de candidatos masculino: ")
                leia(cm)
                escreva("O nº de candidatos feminino: ")
                leia(cf)
                tc := tc + (cm + cf)
                cpv := (cm + cf) / nvagas
                pcf := cf / (cm + cf) * 100

                se(mncv < cpv) entao
                        mncv := cpv
                        curso := cod_c
                fimse
                escreval("------------------- x xx--------------------")
                escreval("O numero de cadidatos por vaga é: ",cpv)
                        escreval("O curso ",cod_c," teve ",pcf," % de
                candidatas mulheres")
                escreva("Cadastrar outro curso(s/n)? ")
                leia(cadastrar)
        fimenquanto
        escreval("---------------------------------- x xx -----------------------
------------")
        escreval("O curso ",curso," teve o maior numero de cadidatos por
vaga que é: ",mncv)
        escreval("O total de candidatos é: ",tc)
        fimalgoritmo
```

LEITURA COMPLEMENTAR

PROGRAMADOS PARA MUDAR

Fórum Econômico Mundial prevê redefinição de 7 milhões de empregos até 2020. E a programação pode ser a chave para encarar essa mudança.

Um mundo dominado por máquinas, casas suspensas, carros voadores. Se é isso que vem a sua mente quando você pensa em automação, em sistemas informatizados e inteligência artificial, está na hora de parar de assistir a filmes de ficção científica. A tecnologia estará, sim, muito mais presente na nossa vida no futuro próximo, mas como nossa aliada. E quem estiver aberto para essa nova realidade terá muito a ganhar em uma sociedade que mistura o real e o virtual e encontra nas informações disponíveis na internet um vasto campo de pesquisa para a tomada de decisões. "A Internet das Coisas (do inglês, *Internet of Things*), juntamente com o *Big Data*, já é e continuará sendo a forma de sistema inteligente

mais utilizada no futuro próximo", afirma Mateus Raeder, coordenador do curso de Ciência da Computação da Unisinos. "Com essas **tecnologias**, a saúde, a educação, a política, o mercado de trabalho, e diversos outros setores serão amplamente beneficiados". O gerenciamento do patrimônio de uma empresa possibilitado por meio de sensores inteligentes interligados a um sistema computacional é um dos exemplos mais comuns de **Internet das Coisas**, segundo o estudioso.

Quando o assunto é **Big Data**, Raeder cita o sucesso da série *House of Cards*, da Netflix, como um bom exemplo do alcance da computação. Foram coletadas inúmeras informações sobre as preferências dos usuários: cliques em vídeos, popularidade de atores, tipos de personagem, tipo de trama que o público mais gosta. "Essas informações, quando reunidas e analisadas, mostraram o formato mais certeiro para o seriado ser um sucesso", comenta.

De acordo com o pesquisador, esse processo de crescimento tecnológico é irreversível e, por isso mesmo, precisamos nos preparar para ele. O caminho? A **programação**. "O conhecimento de programação ajudará todos os profissionais e certamente se destacarão no mercado aqueles que souberem utilizar ferramentas certas a seu favor", afirma. Para ele, recusar o aprendizado de novas tecnologias tornará o profissional defasado em relação aos demais, um problema e tanto no contexto em que algumas profissões serão extintas e muitas outras reformuladas.

A previsão do Fórum Econômico Mundial é de que a tecnologia vai redefinir o **trabalho** de 7 milhões de pessoas até 2020. É muita gente. "Os processos estão todos sendo automatizados e o profissional não pode ignorar essas mudanças. A própria profissão de programador é um exemplo do que estamos falando. Ela certamente não será extinta, mas passará a outro patamar, em que os clientes de um programador não vão solicitar o desenvolvimento de um produto pronto, mas, sim, vão querer programar seu próprio produto", prevê.

A linguagem do futuro

Se, nos dias de hoje, o real e o virtual já se confundem, essa fusão será ainda maior nas próximas décadas. E, obviamente, quem melhor transitar entre essas duas realidades será protagonista na sociedade do **futuro**. "Certamente, as pessoas estarão desenvolvendo seus próprios sites, criando seus próprios aplicativos e gerando seus próprios sistemas. Estar no mundo sem conhecer programação, no futuro, será como não saber ler e escrever atualmente", acredita Mateus. O professor prevê que, em algumas gerações, todos saberão programar, mesmo que de maneira inconsciente. E isso será aprendido também nas escolas do Brasil. "Tenho certeza de que dentro de alguns anos a **educação** básica brasileira (ensino infantil, ensino fundamental e ensino médio) contará com atividades específicas de programação. Isso se deve principalmente ao fato de que o futuro da sociedade depende da tecnologia, e todas as áreas do conhecimento (sem exceção) vão utilizá-la. Assim, é necessário que o conhecimento de programação seja disseminado na geração atual, para que no futuro os profissionais estejam preparados para sua utilização", afirma.

É importante ressaltar que o ensino de programação na educação básica não quer dizer que os alunos serão programadores quando crescerem e encararem o mercado de trabalho. O ensino de programação para crianças é uma prática que estimula o **desenvolvimento** do pensamento lógico, a capacidade de resolução de problemas, a criatividade e auxilia no aprendizado de outras disciplinas, como matemática. Essas habilidades são cruciais para qualquer profissional de sucesso.

E não existe idade para começar. Quanto antes, melhor. "O ensino de programação para crianças acontece comumente por meio de jogos lúdicos e softwares educacionais. Esses **sistemas** utilizam cores e formas geométricas para fazer as crianças aprenderem a programar sem se darem conta do que estão aprendendo. São as habilidades criativas e lógicas que estão sendo exercitadas e aprendidas. Ensinar programação para crianças é fazer com que elas percebam e apliquem o **raciocínio lógico** para resolver as situações que aparecem, por mais simples que sejam. Elas estão aprendendo a programar", explica Mateus.

Se não há discriminação para idade, tampouco há de gênero. Hoje, para expandir os espaços de atuação das mulheres no campo da tecnologia, muitas empresas – como Google e Microsoft – têm projetos de incentivo para programadoras. No futuro, essa realidade pode mudar, se o estímulo começar cedo também para as meninas. Encorajar o estudo da área de exatas para as pequenas desenvolvedoras pode ampliar suas opções de carreira e apresentar **profissões** que, antes, eram ocupadas majoritariamente por homens.

O futuro é agora

Neste contexto, a universidade aparece como um moderador importante, que tem entre suas funções a de mostrar o lado perigoso desta nova realidade e ressaltar a importância da ética no uso da tecnologia. "Ela precisa ensinar aos alunos que a tecnologia deve servir para o bem da **sociedade**. Além disso, a universidade deve mostrar as diferentes formas de utilização da evolução tecnológica. Deve dar subsídios para que os egressos sejam capazes de escolher que tipo de tecnologia será mais apropriado para cada caso, dentro do contexto específico de cada área", afirma o professor da Unisinos. "Outro papel fundamental é o de instigar os alunos a desenvolverem novas soluções tecnológicas, sempre pensando no bem da sociedade".

Para o professor, a hora de pensar nos problemas da sociedade e de começar a resolvê-los com a ajuda da tecnologia é agora. "Os temas emergentes da sociedade atual só vão se agravar se não pensarmos nisso hoje. Pontos cruciais como mobilidade urbana, respeito às diferenças, transformação das profissões. São assuntos que devem ser alvo da inovação tecnológica aliada ao pensamento e à **criatividade**. E criatividade e pensamento são qualidades nossas, humanas, não das máquinas", finaliza o especialista.

8 Passos rumo à programação

1- Entenda que é possível criar coisas incríveis com a programação. Programar é uma atividade extremamente divertida e desafiadora. Além disso, a programação é uma "arte multidisciplinar", pois permite construir programas de computador para qualquer área do conhecimento.
2- Tenha em mente que programar não é uma tarefa trivial. Requer muito mais do que apenas vontade. Exige dedicação, tempo, estudo e muita paciência.
3- Para começar a **programar**, você precisará de uma lógica bem afiada. Para isso, existem cursos e tutoriais (alguns até mesmo gratuitos) que ensinam alguns aspectos básicos de lógica de programação.
4- Em seguida, é interessante você já ter algo em mente para começar a **desenvolver**. Ter um problema para atacar. Por exemplo: se você tem uma empresa, pode pensar em desenvolver um sistema de cadastro de clientes e/ou de produtos do seu estoque. Se você é nutricionista, pode pensar em um website que cadastra seus clientes e realiza acompanhamentos e oferece dicas lá. Essas soluções prontas já existem e já foram programadas por alguém. Entretanto, desenvolver seu próprio sistema, além de proporcionar um grande aprendizado, permite que as soluções desenvolvidas sejam exclusivas.
5- Defina a linguagem em que você irá desenvolver o sistema. Se for um sistema que rodará na internet, ele deverá ser implementado em uma **linguagem** web. Se seu sistema for um aplicativo móvel, você utilizará linguagens específicas para o tipo de sistema do dispositivo alvo (Android, iOS etc.).
6- Pense em um **algoritmo** para resolver o seu problema-alvo. Um algoritmo nada mais é que uma sequência finita de passos que serão realizados para chegar à resolução de um problema. Tente colocar no papel algumas ações imprescindíveis para que seu problema seja resolvido.
7- Com essas informações em mente, procure tutoriais na internet, vídeos e leia livros específicos sobre a linguagem escolhida. Você conseguirá adquirir **conhecimentos** para desenvolver suas primeiras aplicações.
8- Entretanto, se você deseja ser um programador profissional, o mais indicado é realizar um curso técnico ou uma graduação, como Ciência da Computação. Em cursos desse tipo, você aprenderá não apenas aspectos técnicos, mas conceitos importantes sobre lógica de programação e linguagens de programação, além de conhecer todas as fases do processo de desenvolvimento de software. É este profissional que as empresas estão buscando: que tenha conhecimento técnico e uma base sólida de conhecimentos, para desenvolver soluções criativas e inovadoras no contexto em que estiverem inseridos.

FONTE: Disponível em: <http://revistagalileu.globo.com/Sociedade/noticia/2016/11/programados-para-mudar.html>. Acesso em: 7 dez. 2016.

RESUMO DO TÓPICO 3

Neste tópico, você aprendeu que:

- Você foi estimulado e desafiado a construir algoritmos de complexidades distintas.
- Percebeu que o processo de aprendizado dos conceitos iniciais das linguagens de programação é complexo e marcado pela presença de inúmeras dificuldades.
- Aprendeu a ler atentamente o enunciado do problema, compreendendo-o e destacando os pontos mais importantes para iniciar a construção da lógica.
- Aprendeu a identificar os dados de entrada, ou seja, quais dados serão fornecidos para o algoritmo.
- Aprendeu a definir os dados de saída, ou seja, quais dados serão gerados depois do processamento.
- Apendeu a definir o processamento, ou seja, quais cálculos serão efetuados e quais as restrições para esses cálculos. O processamento é responsável pela obtenção dos dados de saída com base nos dados de entrada.
- Apendeu a definir as variáveis necessárias para armazenar as entradas e efetuar o processamento e elaborar o algoritmo.
- Aprendeu a testar o algoritmo.

AUTOATIVIDADE

1 Criar um algoritmo que leia 10 números pelo teclado e exiba os números na ordem inversa da que os números foram digitados.

2 Escreva um algoritmo que leia um vetor com 10 posições de números inteiros. Em seguida, receba um novo valor do usuário e verifique se este valor se encontra no vetor.

3 Criar um algoritmo que leia uma matriz 3x3 e exiba a matriz preenchida

4 Escreva um algoritmo que permita a leitura das notas de uma turma de 20 alunos. Calcular a média da turma e contar quantos alunos obtiveram nota acima desta média calculada. Escrever a média da turma e o resultado da contagem. Esboce a solução através do diagrama de Chapin.

REFERÊNCIAS

FORBELLONE, A. L. V.; EBERSPÄCHER, H. F. **Lógica de programação**. 3. ed. São Paulo: Makron Books, 2005.

JÚNIOR, W. M. P. **Apostila de algoritmos**. Universidade do Estado de Minas Gerais, 2014. Disponível em: <http://www.dainf.ct.utfpr.edu.br/~pbueno/Arquivos/Algoritmos.pdf>. Acesso em: 20 nov. 2016.

MANZANO, José Augusto N. G. **Estudo dirigido**: algoritmos. São Paulo: Érica, 2000.