FIAP ON

ALGORITMOS:
APRENDA A PROGRAMAR

Tomada de DECISÃO

JORGE SURIAN

3

LISTA DE FIGURAS

LISTA DE QUADROS

LISTA DE TABELAS

SUMÁRIO

3 TOMADA DE DECISÃO

Ao ler o capítulo anterior, podemos concluir que nossos algoritmos “fluem” bem, mas não conseguem “lidar” com problemas. Ora, a vida real é repleta deles, portanto, nossos algoritmos em algum momento precisarão lidar com isso.

Por exemplo, embora toda adição de números inteiros apresente uma solução (x = a + b, onde “a” e “b” são números conhecidos), isso não ocorre nas equações de primeiro ou de segundo grau. Nas equações de primeiro grau (ax + b = 0), notamos que o fator (a variável “a”) que multiplica nossa incógnita (a variável “x”) jamais poderá ser zero, pois não temos solução para divisões por zero.

O mesmo inconveniente ocorre na resolução de uma equação de segundo grau, pois como resolver algebricamente uma situação em que a raiz quadrada presente na fórmula de Bhaskara se torne negativa?

Vamos analisar a fórmula de Bhaskara (ou ao menos a ele atribuída, vale a pena pesquisa a respeito):

$$\Delta = b^2 - 4.a.c$$

$$x = \frac{-b \pm \sqrt{\Delta}}{2a}$$

Figura 3.1 – Fórmula de Bhaskara
Fonte: Elaborado pelo autor (2015)

Observe que se testarmos a fórmula com vários valores arbitrários, se “insistirmos” o suficiente no cálculo da raiz de Δ teremos a ocorrência de um erro! Isso ocorre, pois, obrigatoriamente, Δ tem que ser maior ou igual a zero. Caso contrário, simplesmente não temos solução para a fórmula de Bhaskara.

Se o algoritmo for criado sem levar isso em consideração, irá parar pela execução de um erro, no caso, uma divisão por zero. Neste caso, ao invés de calcularmos e apresentarmos uma solução, deveríamos indicar que não há solução,

evitando assim a ocorrência do erro! Basta uma tomada de decisão bem pensada, para resolver essa inconveniente situação.

3.1 Como tomar decisões?

No problema descrito anteriormente, se pudermos identificar previamente que o número que terá sua raiz calculada é negativo, poderemos evitar essa situação.

Em nossa vida, tomamos decisões o tempo todo. Até dizemos, quando existem muitas hipóteses a serem analisadas “e se isso?” Ou “e se aquilo?”. Vamos nos deter nessa pequena, mas fundamental palavra quando se pensa em programação: se.

3.1.1 A INSTRUÇÃO SE

Vamos continuar a pensar na questão de uma raiz hipoteticamente negativa. Em situações como essa, precisaremos verificar o valor armazenado na variável Δ antes de efetuarmos alguma operação, como o envio de uma resposta ou a realização de um cálculo.

Dependendo do valor de uma ou mais variáveis, o algoritmo pode tomar rumos distintos e, muitas vezes, tornar-se errado.

Por exemplo, em uma equação de 2º grau, quando Δ é negativo não podemos extrair a raiz quadrada. Temos aqui então um interessantíssimo ponto de decisão, pois dependendo do valor de Δ, podemos tomar ao menos duas decisões distintas, a saber:

$\Delta < 0$: exibiremos a mensagem que não existem raízes reais para a equação.

$\Delta >= 0$: calcularemos as raízes de acordo com a fórmula e exibiremos seu resultado.

Para isso, temos a estrutura de decisão se/então[/senão] que como mostra a figura a seguir, nos permite abordar o problema em questão e solucioná-lo de uma forma bastante elegante:

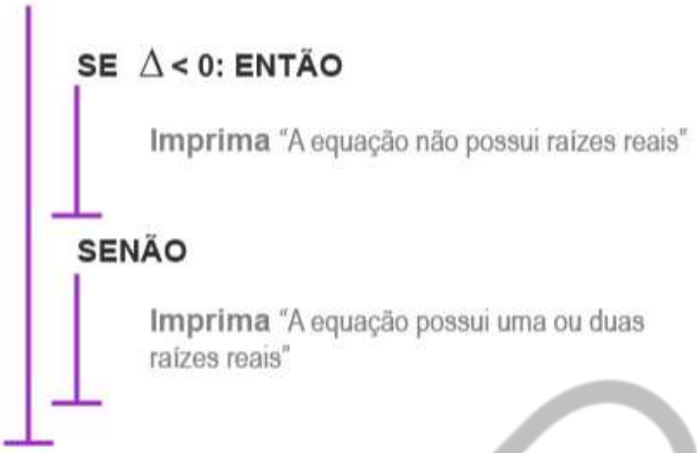


Figura 3.2 – Exemplo de estrutura SE evitando um erro no Bhaskara
Fonte: Elaborado pelo autor (2015)

Várias são as observações:

Note que da análise da expressão $\Delta < 0$ é que se "decide pelo caminho a seguir"!

Quando a expressão analisada for verdadeira, segue-se pela linha do **então**.

Por outro lado, quando a situação analisada é falsa, adota-se a linha do **senão.**

Observe, ainda, as barras de endentação presentes. A mais externa representa aquela que norteia o problema em questão.

Já as barras de endentação mais internas indicam alternativas. A existente entre as palavras **então** e **senão** representam as instruções que serão executadas quando o valor do Δ for negativo, ou seja, menor que zero.

De maneira análoga, quando a barra de endentação existente depois do **senão,** conterá o conjunto de instruções a serem executadas, caso a condição testada não seja verdadeira, ou seja, caso o valor do Δ seja positivo ou zero.

As linguagens de programação, cada uma a seu modo, procuram representar essas situações. Em nosso material iremos, sempre que necessário, nos valer de exemplos nas mais diversas linguagens de programação, e sempre que possível comparando a solução que de cada ao problema que se deseja resolver. Vamos então ver como ficaria esse teste em Java:

```
if (delta < 0) {
    System.out.println(“A equação não possui raízes reais”);
}
else {
    System.out.println(“A equação possui 2 raízes reais”);
}
```

Figura 3.3 – Estrutura SE evitando erro de Bhaskara em Java
Fonte: Elaborado pelo autor (2015)

Observamos facilmente que as barras foram substituídas por pares de instrução, cercados por “chaves”, através dos símbolos { }. Essa mesma simbologia é usada na linguagem C, por exemplo.

Já se a instrução em questão estivesse escrita em Pascal, teríamos uma solução ligeiramente distinta:

```
if delta &lt; 0 then
Write(‘A equação não possui raízes reais’);
else
Write(‘A equação possui 2 raízes reais’);
```

Figura 3.4 – Estrutura SE evitando erro de Bhaskara em Pascal
Fonte: Elaborado pelo autor (2015)

Observa-se que em Pascal, o bloco em questão “desapareceu”, sendo substituído por um terminador (o ponto e vírgula no final do último comando Write).

Mas, como ficaria em Pascal, se além de apresentarmos essa mensagem também tivéssemos outra instrução, que apenas atribuísse o valor 1 (o número um) a uma variável chamada “controle”, quando o problema tiver solução ou 0 (o número zero) ao “controle” quando o problema não tiver solução?

```
if delta < 0 then begin
    Write('A equação não possui raízes reais');
    Controle := 0;
    end
else begin
    Write('A equação possui 2 raízes reais');
    Controle := 1;
    end;
```

Figura 3.5 – Estrutura SE em Pascal com mais de uma instrução por bloco
Fonte: Elaborado pelo autor (2015)

É fácil perceber que simplesmente as chaves do Java foram substituídas pelas palavras “Begin-end” em Pascal, que representam o começo e o final do “bloco de instruções”. Observe ainda que as linguagens de programação têm sintaxes bem próprias e particulares, como são os idiomas humanos.

Mesmo a posição em que as instruções devem ser escritas, varia de linguagem para linguagem. A posição das instruções para “imprimir” um valor em Java deve ser escrita abaixo da expressão analisada ou da palavra “else”, deslocando-se o código um pouco à direita, para dar uma ideia de reentrância.

Já em Pascal, sugere-se fortemente que essa distância seja de três letras. Assim, o “W” de Write deveria estar abaixo do “e” de else. Os clássicos livros de Pascal ou de Turbo Pascal, como os escritos por Steve Wood, são categóricos a esse respeito.

Em COBOL, o espaço existente é obrigatoriamente estabelecido. Ou seja, se algo é esperado em determinada coluna e estiver colocado numa coluna adiante ou anterior, será tratado como erro. Daí a existência de vários editores de programa, que são muito específicos em relação à linha e à coluna em que se está escrevendo, justamente para permitir ao programador o controle exato da localização de um bloco de instruções.

Não é tão simples notar, mas antes da palavra “else” (senão) em Pascal, nunca se usa o terminador (ponto e vírgula), pois a instrução “se” só termina, se tiver “senão” ao final deste. Esse tipo de situação jamais ocorrerá em Java, pois em Java o símbolo **}** (fecha a chave) impede essa ocorrência.

Em Linguagem C, contudo, o problema não é resolvido tão facilmente. Tampouco em BASIC. Vale a pena pesquisar a respeito.

Como se vê, ao aprender lógica de programação você está aprendendo a pensar de forma a poder se expressar em qualquer que seja a linguagem de programação, mas antes de sair “programando” numa linguagem, obrigatoriamente terá de compreendê-la e dominar sua sintaxe, que pode ser algo muito mais complexo do que aparentemente parece.

3.1.2 Sintaxe da instrução SE

Toda e qualquer instrução tem uma sintaxe, ou seja, uma forma exata como deverá ser escrita. Vamos, por assim dizer, “definir” como se descreve as sintaxes, usando para isso a instrução “Se”.

```
se <condicao> entao
     <instrucao>
[ senao
     <instrucao> ]
fim-se
```

Vamos a uma série de observações:

a) Os símbolos [ ] (colchetes) indicam **opcionalidade.** Trata-se de uma parte do código que não é obrigatória, ficando a critério do programador usá-la ou não.

b) As expressões existentes entre os símbolos < > (que são chamadas popularmente de *tags*, devido ao seu grande uso em linguagens de hipertexto, como é o caso do HTML e suas derivadas) contêm expressões ou comandos a serem testados ou executados.

c) As palavras “reservadas” estão em negrito e não poderão jamais ser usadas como variáveis, quer seja em nossa codificação (Português Estruturado) ou nas linguagens de programação. Assim, “se” jamais poderá ser uma “variável” em nossos programas, como “if” jamais poderá ser uma variável

nas linguagens Java e Pascal, como pudemos notar nos exemplos escritos naquelas duas importantes linguagens de programação.

3.1.3 Exemplos

Exemplo 1: Tomada de decisão simples (sem senão).

Problema: Imprimir “x” se este for igual a 1.

Resolução:

```
algoritmo se01;
variáveis
    x : inteiro;
fim-variáveis
início
    x := leia();
    se x = 1 então
        imprima(x);
    fim-se
fim
```

Figura 3.6 – Exemplo de estrutura SE
Fonte: Elaborado pelo autor (2015)

Exemplo 2: Tomada de decisão com senão.

Problema: Imprimir “x+1” se este for diferente de 2 ou imprima o dobro de “x”, quando esse for 2.

Resolução:

```
algoritmo se02;
variáveis
    x : inteiro;
fim-variáveis
início
    x := leia();
    se x <> 2 então
        imprima(x + 1);
    senão
        imprima(x * 2);
    fim-se
fim
```

Figura 3.7 – Exemplo de estrutura SE com senão
Fonte: Elaborado pelo autor (2015)

Exemplo 3: Solução parcial para a questão da equação do segundo grau.

Problema: dada uma expressão do tipo "$ax^2 + bx + c = 0$", indique se é possível calcular o valor de x.

```
algoritmo delta;
variáveis
    x1, x2, delta, raiz : real;
    a, b, c : inteiro;
fim-variáveis
início
    x1 := 0;
    x2 := 0;
    delta :=  0;
    raiz := 0;
    a := leia();
    b := leia();
    c := leia();
    se a = 0 então
        imprima("Não existe raiz");
    fim-se
    delta := b*b - 4*a*c;
    se delta < 0 então
        imprima("A equação não tem raízes reais");
    senão
        imprima("A equação tem raízes reais");
    fim-se
fim
```

Figura 3.8 – Exemplo de estrutura SE usada para validar Bhaskara
Fonte: Elaborado pelo autor (2015)

3.1.4 EXERCÍCIOS

a) Dados dois números, por hipótese distintos entre si, imprima o maior deles.

b) Dados dois números, imprima o maior deles, se houver.

3.1.5 Tabela verdade

Vamos considerar termos duas variáveis "a" e "b", mas que em vez de valores, tenham conteúdos *booleanos,* ou seja, assumem apenas situação de verdadeiro ou

falso. Podemos fazer uma analogia com um trem, que tem a frente um desvio. Uma baliza (ou *flag*, se usarmos seu sinônimo em inglês) vai definir o destino desse hipotético trem.

Supondo que nosso trem esteja partindo de Paris, mas num entroncamento é decidido se ele irá para Roma ou para Berlim. Se esse entroncamento for uma variável, “a” por exemplo, podemos definir que sempre que “a” for Verdade, então o trem irá para Roma. Em caso contrário, irá para Berlim.

Todavia, podemos inserir novas variáveis. Por exemplo, uma variável “b” que quando o trem não for para Roma, permita decidir se seu destino será Berlim ou Praga. E assim sucessivamente.

Além disso, podemos incluir as palavras “E”, para indicar que uma condição somente estará satisfeita se duas premissas forem simultaneamente verdadeiras ou “OU” que indica que uma condição estará satisfeita, quando ao menos uma das condições for verdadeira.

Voltando ao nosso hipotético trem, poderíamos dizer que ele irá para Roma se “a” E “b” forem simultaneamente verdadeiros. Quando um dos dois for falso, poderíamos estabelecer que se “a” for falso E “b” verdadeiro, então vamos para Berlim. Já se “a” for falso E “b” verdadeiro, então vamos para Praga. Teríamos agora a possibilidade de incluir um novo destino a nosso hipotético trem: Bratislava. Esse seria o destino, na hipótese de “a” E “b” serem ambas falsas, simultaneamente.

O mesmo raciocínio se aplica a palavra “OU”. Todavia, seu uso é bem distinto. Vamos imaginar que um vendedor seja recompensado com um bônus, quando tiver feito mais de 50 visitas no mês ou quando tiver vendido além da meta. Se considerarmos “a” uma variável booleana que tenha como conteúdo “Verdade” se o vendedor fez mais de 50 visitas ou “Falso” quando tiver feito 50 ou menos visitas e “b” de forma análoga, mas relativa a meta comercial, teríamos:

- Se “a” verdade e “b” falso (visitou mais de 50, mas não atingiu a meta) então bônus
- Se “a” falso e “b” verdade (visitou 50 ou menos, mas atingiu a meta) então bônus
- Se “a” e “b” verdadeiros (visitou mais de 50 e atingiu a meta) então bônus

- Se “a” e “b” falsos (nem visitou mais de 50 e também não atingiu a meta) então sem bônus

Ou seja, somente numa condição nosso vendedor não receberá seu bônus.

Para facilitar a compreensão costumamos pensar nessas situações de maneira mais geral, sem trens ou vendedores...

Usamos, então, uma tabela genérica, que chamamos de Tabela Verdade, conforme a que é apresentada adiante:

a	b	a E b	a OU b
Verdade	Verdade	Verdade	Verdade
Verdade	Falso	Falso	Verdade
Falso	Verdade	Falso	Verdade
Falso	Falso	Falso	Falso

Figura 3.9 – Exemplo de tabela verdade
Fonte: Elaborado pelo autor (2017)

3.2 Alguns tipos de utilização da instrução SE

A instrução SE pode ser usada de forma simples, como fizemos nos algoritmos anteriores, mas também pode analisar expressões mais complexas, como veremos no nosso próximo exemplo.

Testaremos duas condições e, sendo qualquer delas for verdadeira, apresentaremos uma mensagem e, caso nenhuma delas for verdadeira, apresentaremos outra mensagem.

Exemplo 4: Dados dois números, apresente a mensagem “ao menos um é par”, se ao menos um deles for par.

Resoluções

```
algoritmo se_um_par;
variáveis
    a, b : inteiro;
fim-variáveis
início
    a := leia();
    b := leia();
    se (a % 2 = 0) ou (b % 2 = 0) então
        imprima("Ao menos um Par");
    senão
        imprima("Nenhum dos dois é Par");
    fim-se
fim
```

Figura 3.10 – Exemplo de estrutura SE com duas condições a serem verificadas
Fonte: Elaborado pelo autor (2015)

3.2.1 SE em expressão complexa (“teste OU”)

```
algoritmo se_um_par;
variáveis
    a, b : inteiro;
fim-variáveis
início
    a := leia();
    b := leia();
    se (a % 2 = 0) ou (b % 2 = 0) então
        imprima("Ao menos um Par");
    senão
        imprima("Nenhum dos dois é Par");
    fim-se
fim
```

Figura 3.11 – Exemplo de estrutura SE em expressão complexa com OU
Fonte: Elaborado pelo autor (2015)

Na solução do problema acima, nota-se que a instrução “Se (a % 2 = 0) ou (b % 2 = 0) então” executa a condição de então sempre que ao menos um dos dois números indicados for par.

Mas seria essa a única maneira de solucionar a questão? Seguramente não.

3.2.2 Aninhando instruções SE

E se em vez de analisarmos as duas condições numa única instrução, o fizéssemos separadamente?

Como descobrir se um número é par ou ímpar? Bem, basta dividi-lo por dois: Se a divisão for exata, ou seja, não deixar resto, o número é par, caso contrário, ele é ímpar. Relembrando, o símbolo “%” (porcentual) é utilizado para realizar divisões e retornar o resto da mesma.

Observaríamos se “a” é par e aí imprimiríamos a mensagem. Em não sendo, testaríamos “b” e trataríamos a questão da mesma maneira. Estaríamos, então, “aninhando” as tomadas de decisão, como demonstra a solução alternativa apresentada a seguir:

```
algoritmo se_um_par_ninho;
variáveis
    a, b : inteiro;
fim-variáveis
início
    a := leia();
    b := leia();
    se (a % 2 = 0) então
     imprima("Ao menos um Par");
     senão
        se (b % 2 = 0) então
            imprima("Ao menos um Par");
         senão
             imprima("Nenhum dos dois é Par");
        fim-se
    fim-se
fim
```

Figura 3.12 – Exemplo de estrutura SE aninhada, alternativa à expressão complexa OU
Fonte: Elaborado pelo autor (2015)

Dessa vez, primeiramente testamos a variável “a” isoladamente (“se (a % 2 = 0) então”) e apenas no caso dessa ser ímpar é que passamos a testar “b” (“se (b % 2 = 0) então)”. Caso ambos os testes falhem, isto é, ambos os números forem ímpares, teremos a mensagem sobre a inexistência de pares entre os dois números digitados.

Surpreendente ou não, é exatamente assim que uma expressão SE com duas condições ligadas com OU se comporta: Se a primeira expressão for verdadeira, o

processamento segue adiante e o tratamento é realizado, sem sequer verificar a segunda condição – essa só é realmente verificada se a primeira condição for falsa.

3.2.3 SE encadeados

Exemplo 5: Dados dois números, apresente a mensagem “O primeiro é par”, “O segundo é par” ou “Nenhum é par”.

Resolução

```
algoritmo se_par;
variáveis
    a, b, c : inteiro;
fim-variáveis
início
    a := leia();
    b := leia();
    c := 0;
    se (a % 2 = 0) então
        imprima("O primeiro é Par");
         c := 1;
    fim-se
    se (b % 2 = 0) então
        imprima("O segundo é Par");
        c := 1;
    fim-se
    se (c = 0) então
        imprima("Nenhum dos dois é Par");
    fim-se
fim
```

Figura 3.13 – Exemplo de estruturas SEindependentes
Fonte: Elaborado pelo autor (2015)

Nosso algoritmo agora precisa resolver um problema que admite três saídas distintas e testes que permitam identificar qual número é par. Podemos ter, inclusive, como resposta a indicação de que o “primeiro é par” e o “segundo é par”. Notamos que a variável “c” funciona como uma **baliza,** ou seja, ela indica se o fluxo do programa passou num certo trecho. Assim, toda vez que alguma mensagem for apresentada, “c” passa a valer 1 (um). Como “c” inicialmente vale 0 (zero), ora, se ao fim da execução “c” continuar valendo zero, simplesmente saberemos que não houve nenhuma mensagem apresentada até aquele momento, isto é, “a” e “b” são ímpares.

3.3 Exemplo clássico

Exemplo 6: Dados três números, por hipótese inteiros, positivos e distintos entre si, apresente o maior deles.

Antes de prosseguirmos, cabe analisar cuidadosamente nosso enunciado. Na medida em que existe uma hipótese a ser considerada, não precisaremos testar se, de fato, os números informados são maiores que zero e diferentes entre si. Isso o enunciado já nos garante!

Vamos ficar atentos ao que é solicitado pelo enunciado, pois qualquer que seja a hipótese indicada, testar sua veracidade sempre significará má compreensão do que é pedido.

O exemplo a seguir será resolvido de várias maneiras, de forma a possibilitar algumas das diversas maneiras de combinar a instrução SE. Teremos nada menos que seis “estratégias” distintas, cada uma delas, levando a soluções totalmente distintas também, mas corretas, do problema em questão.

3.3.1 Maior de três usando ninho de instruções

Nossa primeira estratégia será aninhando nossas instruções, ou seja, testaremos primeiramente a possibilidade de “a” ser maior que “b”. Sendo verdadeiro, testamos “a” contra “c”: se, de fato, “a” for também maior que “c”, então seguramente “a” é o maior dos três números.

Repare que, se “a” era maior que “b”, mas não que “c”, então “c” é o maior dos três!

Na hipótese de “a” não ser maior que “b”, resta apenas testar se “b” é maior que “c”. Analogamente ao que ocorreu no teste anterior, se “b” também for maior que “c”, então “b” será o maior dos três, caso contrário, o maior será “c”.

Resolução

```
algoritmo maior_de_3_ninho;
variáveis
   a, b, c : inteiro;
fim-variáveis
início
    imprima("1o Número: ");
    a := leia();
    imprima("2o Número: ");
    b := leia();
    imprima("3o Número: ");
    c := leia();
    se a > b então
        se a > c então
              imprima("Maior: ",a);
        senão
              imprima("Maior: ",c);
        fim-se
    senão
        se b > c então
                imprima("Maior: ",b);
        senão
              imprima("Maior: ",c);
        fim-se
    fim-se
fim
```

Figura 3.14 – Exemplo de maior número utilizando condicionais aninhadas
Fonte: Elaborado pelo autor (2015)

Notemos que faremos apenas dois testes, qualquer que seja o número maior, mesmo que tenhamos três estruturas de tomada de decisão. Isso ocorre porque o bloco SE (ou bloco “verdadeiro”) e o SENÃO (ou bloco “falso”) nunca serão executados juntos, irão sempre se alternar. Ao colocar uma estrutura de tomada de decisão em cada bloco, aninhando-as, se a condição do primeiro teste for verdadeira, nunca passaremos pelo SE que compara “b” com “c” e vice-versa. Trata-se de um algoritmo muito interessante, que usa apenas **três variáveis e sempre faz no máximo duas comparações.**

3.3.2 Maior de três usando teste de possibilidades

Nossa segunda estratégia será a mais óbvia de todas e, quase que em consequência disso, a pior das estratégias possíveis. Consiste em testes, um a um, de todas as seis possibilidades existentes, uma a uma. Testaremos as seguintes possibilidades:

1) a > b > c
2) a > c > b
3) b > a > c
4) b > c > a
5) c > a > b
6) c > b > a

Em tese, testaremos as possibilidades indicadas pela combinatória.

Resolução

```
algoritmo maior_de_3_permutacao;
variáveis
    a, b, c : inteiro;
fim-variáveis
início
    imprima("1o Número: ");
    a := leia();
    imprima("2o Número: ");
    b := leia();
    imprima("3o Número: ");
    c := leia();
    se (a > b) e (b > c) então
        imprima(a);
    fim-se
    se (a > c) e (c > b) então
        imprima(a);
    fim-se
    se (b > a) e (a > c) então
        imprima(b);
    fim-se
    se (b > c) e (c > a) então
        imprima(b);
    fim-se
    se (c > a) e (a > b) então
        imprima(c);
    fim-se
    se (c > b) e (b > a) então
        imprima(c);
    fim-se
fim
```

Figura 3.15 – Exemplo de maior número usando teste de probabilidades
Fonte: Elaborado pelo autor (2015)

Trata-se de um algoritmo muito ruim, que usa apenas **três variáveis** e sempre faz três comparações por expressão**.** Como conta com seis expressões comparativas, acaba executando, em qualquer situação, **dezoito comparações.**

Por não estarem aninhadas, mesmo que “tenhamos sorte” e o maior número seja descoberto logo na primeira estrutura condicional, ainda assim, as outras cinco condicionais serão testadas. Péssimo, não é?

Devemos observar que quando comparamos “a > b” e “b > c” realizamos duas comparações. Todavia, no final, ainda verificamos se a primeira e a segunda comparações são válidas simultaneamente, donde teremos sempre três comparações por instrução.

3.3.3 Maior de três usando cláusula “E”

Nossa terceira estratégia será um claro aperfeiçoamento da estratégia anterior, pois reduziremos de seis para três testes, bastaria “armarmos” melhor nossa estratégia para resolver a questão de maneira muito mais performática e inteligente. Vamos a ela, então:

1) a>b e a>c
2) b>a e b>c
3) c>a e c>b

Resolução

```
algoritmo maior_de_3_E;
variáveis
    a, b, c : inteiro;
fim-variáveis
início
    imprima("1o Número: ");
    a := leia();
    imprima("2o Número: ");
    b := leia();
    imprima("3o Número: ");
    c := leia();
    se (a > b) e (a > c) então
        imprima(a);
    fim-se
    se (b > a) e (b > c) então
        imprima(b);
    fim-se
    se (c > a) e (c > b) então
        imprima(c);
    fim-se
fim
```

Figura 3.16 – Exemplo de maior número usando operador lógico E, melhorado
Fonte: Elaborado pelo autor (2015)

Nossa terceira estratégia, embora seja muito melhor que a segunda, ainda é bastante inferior a primeira.

Vale notar que como nas demais, também se utiliza de **três variáveis**. Conta com três expressões comparativas, cada uma delas com três comparações, portanto

executa **sempre nove comparações, mesmo que o maior número seja descoberto precocemente**

3.3.4 Maior de três usando cláusula “E e SENÃO”

Nossa quarta estratégia será um novo aperfeiçoamento, agora da terceira estratégia. Ora, poderíamos não mais comparar “a” se soubéssemos que esse não é o maior, certo? Teríamos uma sequência de testes bem mais eficaz, portanto:

- a>b e a>c, portanto “a” é o maior.

Mas, se “a” não for o maior, basta testar.

- b>c, identificando, conforme o caso, quem será o maior.

Resolução

```
algoritmo maior_de_3_E_senao;
variáveis
   a, b, c : inteiro;
fim-variáveis
início
    imprima("1o Número: ");
    a := leia();
    imprima("2o Número: ");
    b := leia();
    imprima("3o Número: ");
    c := leia();
    se (a > b) e (a > c) então
           imprima(a);
    senão
        se (b > c) então
              imprima(b);
        senão
              imprima(c);
        fim-se
    fim-se
fim
```

Figura 3.17 – Exemplo de maior número usando operador lógico E e SENÃO
Fonte: Elaborado pelo autor (2015)

Desta feita, temos, na pior das situações, apenas **quatro comparações**. Estas ocorrem quando “a” não é o maior, situação em que apenas três comparações seriam executadas. Novamente temos apenas **três variáveis** sendo usadas.

3.3.5 Maior de três usando uma variável auxiliar

Nossa quinta estratégia terá uma abordagem distinta, armazenaremos numa nova variável, que chamaremos sintomaticamente de “maior” o maior dos três números. Vamos a ela, então...

Resolução

```
algoritmo maior_de_3_Var_Aux;
variáveis
   a, b, c, maior : inteiro;
fim-variáveis
início
    imprima("1o Número: ");
    a := leia();
    imprima("2o Número: ");
    b := leia();
    imprima("3o Número: ");
    c := leia();
    se (a > b)  então
        maior := a;
    senão
           maior := b;
    fim-se
    se (c > maior) então
           maior := c;
    fim-se
    imprima(maior);
fim
```

Figura 3.18 – Exemplo de maior número usando uma variável extra
Fonte: Elaborado pelo autor (2015)

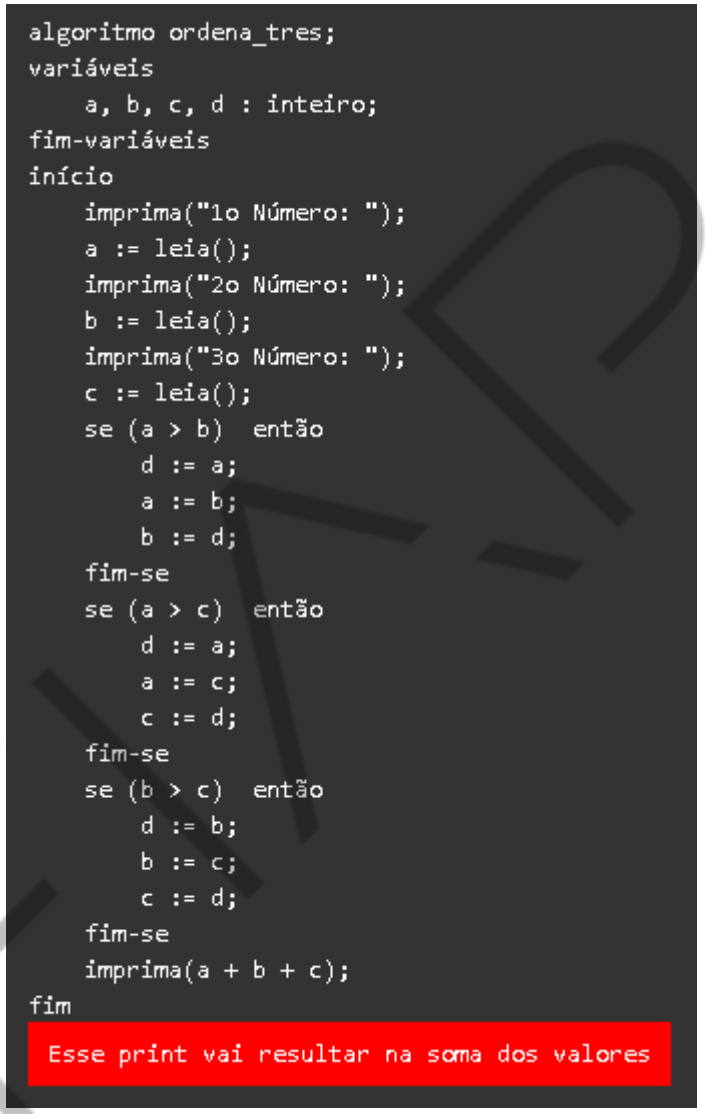
Agora temos apenas **duas comparações**, como ocorreu na primeira resolução, todavia temos **quatro variáveis** sendo utilizadas, o que faz com que esse algoritmo seja apenas ligeiramente inferior ao primeiro visto. Tal diferença, todavia, não é significativa, pois a economia de uma simples variável é algo de insignificante relevância.

3.3.6 Maior de três usando o princípio da ordenação

Nossa sexta e última estratégia contempla um princípio muito interessante, que é a ordenação de três números. De fato, antes de identificar o maior dos três números,

vamos ordená-los de forma crescente, fazendo com que a variável “c” sempre contenha o maior entre três números distintos.

Para tanto, será necessária uma quarta variável (“d”) que servirá como variável auxiliar ou de transporte, já que as variáveis trocarão seus valores.



```
algoritmo ordena_tres;
variáveis
    a, b, c, d : inteiro;
fim-variáveis
início
    imprima("1o Número: ");
    a := leia();
    imprima("2o Número: ");
    b := leia();
    imprima("3o Número: ");
    c := leia();
    se (a > b)  então
        d := a;
        a := b;
        b := d;
    fim-se
    se (a > c)  então
        d := a;
        a := c;
        c := d;
    fim-se
    se (b > c)  então
        d := b;
        b := c;
        c := d;
    fim-se
    imprima(a + b + c);
fim
```

Esse print vai resultar na soma dos valores

Figura 3.19 – Exemplo de maior e ordenação
Fonte: Elaborado pelo autor (2015)

Notamos que a estratégia tornar a inequação “a < b < c” verdadeira. Essa estratégia pode ser simplificada para contemplar que o maior fique em “c”, embora não determinemos se o menor estará em “a” ou “b”, que nos leva ao seguinte algoritmo:

Resolução

```
algoritmo maior_de_3_ordena;
variáveis
   a, b, c, d : inteiro;
fim-variáveis
início
    imprima("1o Número: ");
    a := leia();
    imprima("2o Número: ");
    b := leia();
    imprima("3o Número: ");
    c := leia();
    se (a > b)  então
        d := a;
           a := b;
           b := d;
    fim-se
    se (b > c)  então
           d := b;
           b := c;
           c := d;
    fim-se
    imprima(c);
fim
```

Figura 3.20 – Exemplo de maior e ordenação melhorado
Fonte: Elaborado pelo autor (2015)

3.4 Simulação ou teste de mesa

Todo algoritmo pode ser vertido para uma linguagem de programação e nela ser executado, de forma a obtermos os resultados desejados através de seu processamento. Todavia, não precisamos de linguagens para saber se nosso raciocínio está correto. Podemos simular nossos algoritmos.

Na verdade, a simulação não passa do acompanhamento dos valores obtidos com o processamento de qualquer algoritmo. Como resolvemos o problema anterior de seis formas distintas, temos então seis simulações diferentes também.

Vamos agora simular uma dessas versões, com a qual poderemos comparar o desempenho de cada uma das soluções.

3.4.1 Simulação: maior de três usando ninho de instruções

Na simulação temos a indicação das variáveis usadas no algoritmo, que são iniciadas com valores aleatórios, como seriam se algum usuário as tivesse informado.

```
algoritmo maior_de_3_ninho;
variáveis
   a, b, c : inteiro;
fim-variáveis
início
    imprima("1o Número: ");
    a := leia();
    imprima("2o Número: ");
    b := leia();
    imprima("3o Número: ");
    c := leia();
    se a > b então
        se a > c então
              imprima("Maior: ",a);
        senão
              imprima("Maior: ",c);
        fim-se
    senão
        se b > c então
                imprima("Maior: ",b);
        senão
              imprima("Maior: ",c);
        fim-se
    fim-se
fim
```

Figura 3.21 – Exemplo de maior número entre três valores
Fonte: Elaborado pelo autor (2015)

Além das variáveis, também estão indicadas as expressões analisadas e seus resultados. Não é usual representarmos essas situações nas simulações, mas estas podem comportar todo e qualquer detalhe que possa ser útil à compreensão do algoritmo.

Exercício: simular, com os mesmos números, cada uma das demais soluções apresentadas para o problema maior de 3.

3.5 Exemplos introdutórios

Nesta seção, há vários exemplos apresentando técnicas, estratégias e outros aspectos importantes a serem aprendidos, antes da finalização desse módulo com uma bateria de exercícios.

- Elabore algoritmo que dados três números inteiros, informe quantos deles são ímpares:

Esse problema admite duas estratégias distintas para sua solução. A primeira delas, bastante simples, chega a uma solução comum: contá-los conforme são identificados. A segunda, aproveitando-se das características dos números inteiros, chega a uma solução muito mais elegante e sofisticada. Vejamos o porquê disso.

Solução 1: Estratégia “contar” números ímpares a medida em que estes forem identificados.

```
algoritmo soma_impar_simples;
variáveis
    a,b,c,s : inteiro;
fim-variáveis
início
      s := 0;
      a := leia();
    se a % 2 = 1 então
        s := s + 1;
    fim-se
    se b % 2 = 1 então
         s := s + 1;
    fim-se
    se c % 2 = 1 então
        s := s + 1;
    fim-se
    imprima(s);
fim
```

Figura 3.22 – Exemplo de contabilização de números ímpares
Fonte: Elaborado pelo autor (2015)

São feitas três comparações antes de realizarmos cada uma das somas. Mas pensando bem, será que isso é realmente necessário?

Solução 2: Estratégia aproveitando o fato de que o resto da divisão de qualquer número ímpar por dois sempre resulta em um.

```
algoritmo soma_impar_direito;
variáveis
    a,b,c,s : inteiro;
fim-variáveis
início
    s := 0;
    a := leia();
    s := a % 2 + b % 2 + c % 2;
    imprima(s);
fim
```

Figura 3.23 – Exemplo de contabilização de números ímpares, extremamente melhorado
Fonte: Elaborado pelo autor (2015)

Podemos observar nessa segunda solução que não fazemos nenhuma comparação. Ora, como o resto da divisão de um número por dois resulta em um, basta somar todos os restos dessas divisões para termos o total de ímpares informados... simples e elegante.

Conclusão: Embora os resultados finais sejam idênticos e igualmente corretos, vê-se que uma estratégia mais elaborada costuma resultar num algoritmo de leitura mais complexa e de execução mais simples e veloz. Essa é a essência da programação: Não basta que funcione, deve funcionar bem. E funcionar bem, em programação, significa ter sempre uma visão de desempenho do algoritmo.

Há algumas décadas atrás, essa era a diferença de um sistema computacional que executava nos antigos mainframes em minutos ou em horas. Embora o poder de processamento tenha sido multiplicado em centenas de vezes de lá para cá e, em um primeiro momento pareça bobagem, lembre-se que algoritmos como este serão utilizados em servidor www com milhares de usuários simultâneos ou smartphones que possuem performances mais modestas e com dezenas de programas ao mesmo tempo. Esta pode ser a diferença entre um bom programa ou não.

- Claramente, economizar três comparações pode ser um ótimo negócio, principalmente em programas em que determinados trechos serão executados centenas de milhares de vezes. Elabore algoritmo que de posse de dois números inteiros e positivos escreva, conforme o caso: “verde”, se ambos forem positivos; “vermelho”, se ambos forem negativos ou “amarelo” nos demais casos

Temos duas maneiras distintas de solucionar esse problema, e dependendo da estratégia escolhida, similar ao ocorrido anteriormente, teremos códigos de qualidades diferentes.

Solução 1: Vamos abordar o problema pela solução mais óbvia, ou seja, separar a situação que gera “verde”, depois a que gera “vermelho” e finalmente a que gera “amarelo”.

```
algoritmo semaforo_01;
variáveis
    a, b : inteiro;
fim-variáveis
início
    a := leia();
    b := leia();
    se a % 2 = 0 e b % 2 = 0 então
        imprima("Verde");
    fim-se
    se a % 2 = 1 e b % 2 = 1 então
        imprima("Vermelho");
    fim-se
    se (a % 2 = 1 e b % 2 = 0) ou (a % 2 = 0 e b % 2 = 1) então
        imprima("Amarelo");
    fim-se
fim
```

Figura 3.24 – Exemplo de condicionais gerando um semáforo
Fonte: Elaborado pelo autor (2015)

Notamos que esse algoritmo simplesmente despreza descobertas anteriores. Imagine que ambos os números sejam pares. Ao passar pela condicional, conclui-se que ele é verde. Problema resolvido, certo? Não do jeito que o algoritmo foi construído: as verificações se ele é vermelho e amarelo serão realizadas. Há uma falha no raciocínio lógico, são três situações mutuamente excludentes. Ora, se a cor não é nem “verde” e nem “vermelha”, só pode ser “amarela”!

Como aspecto notável nesse algoritmo, somente o uso simultâneo na comparação das expressões **“e”** e **“ou”**.

Vamos então observar como fica nosso algoritmo, se observarmos que se a cor não for nem verde, nem vermelha, só poderá ser amarela.

```
algoritmo semaforo_02;
variáveis
    a, b : inteiro;
fim-variáveis
início
    a := leia();
    b := leia();
    se a % 2 = 0 e b % 2 = 0 então
        imprima("Verde");
    senão
        se a % 2 = 1 e b % 2 = 1 então
            imprima("Vermelho");
        senão
            imprima("Amarelo");
        fim-se
    fim-se
fim
```

Figura 3.25 – Exemplo de condicionais gerando um semáforo melhorado
Fonte: Elaborado pelo autor (2015)

Perceba que as verificações foram encadeadas; é o mesmo sendo analisado e como foi dito, são situações excludentes. Não é possível imprimir “verde”, “vermelho” e “amarelo” simultaneamente. Construído desta forma, se ambos os números forem pares, conclui-se que a cor é verde e a segunda verificação será descartada, afinal, ela está em um bloco “senão”. Muito mais eficaz.

3.5.1 Exercícios

a) Dados dois números inteiros e distintos entre si, imprima “Ambos Pares”, “Ambos Ímpares” ou “Demais Casos”, conforme a situação

Algoritmo "Pares"

// Algoritmo desenvolvido em VisualG 3.0

// Observação: foram duas variáveis lógicas

// que recebem o resultado do resto da divisão (MOD)

// dos números digitados

```
// A variável teste1 terá o valor VERDADEIRO se ambos os números forem pares
// A variável teste2 terá o valor VERDADEIRO se ambos os números fore ímpares

Var

numero_1: real;
numero_2: real;
teste1 : logico;
teste2 : logico;

Inicio

escreva("Digite o primeiro numero: ")

leia(numero_1)

escreva("Digite o segundo numero: ")

leia(numero_2)

teste1 := numero_1 MOD 2 = 0 e numero_2 MOD 2 = 0
teste2 := numero_1 MOD 2 = 1 e numero_2 MOD 2 = 1
```

```
SE teste1 ENTÃO
  escreva ("Ambos Pares")
SENAO
  SE teste2 ENTÃO
    escreva ("Ambos Impares")
  SENAO
    escreva ("Demais Casos")
  FIMSE
FIMSE
Fimalgoritmo
```

b) Dados dois números, imprima sempre a subtração do maior pelo menor, admitindo que ambos são distintos entre si

```
Algoritmo "Subtracao_distintos"

// Algoritmo desenvolvido em VisualG 3.0

Var

numero_1: real;
numero_2: real;
calculo: real;

Inicio

escreva("Digite o primeiro numero: ")
```

```
leia(numero_1)

escreva("Digite o segundo numero: ")

leia(numero_2)

SE numero_1 > numero_2 ENTÃO
  calculo := numero_1 - numero_2
  escreva ("O primeiro numero é maior. Diferença: ", calculo)
SENAO
  SE numero_2 > numero_1 ENTÃO
    calculo := numero_2 - numero_1
    escreva ("O segundo numero é maior. Diferença: ", calculo)
  FIMSE
FIMSE
Fimalgoritmo
```

c) Dados dois números, imprima sempre a subtração do maior pelo menor, todavia se forem iguais, apresente a mensagem “são iguais”

```
Algoritmo "Subtracao_idênticos"

// Algoritmo desenvolvido em VisualG 3.0
```

```
Var

numero_1: real;
numero_2: real;
calculo: real;

Inicio

escreva("Digite o primeiro numero: ")

leia(numero_1)

escreva("Digite o segundo numero: ")

leia(numero_2)

SE numero_1 > numero_2 ENTÃO
  calculo := numero_1 - numero_2
  escreva ("O primeiro numero é maior. Diferença: ", calculo)
SENAO
  SE numero_2 > numero_1 ENTÃO
    calculo := numero_2 - numero_1
    escreva ("O segundo numero é maior. Diferença: ", calculo)
  SENÃO
    escreva ("Os numeros são iguais")
  FIMSE
```

```
FIMSE



Fimalgoritmo
```

d) Retome o problema Semáforo. Como ficaria a solução se usássemos uma estratégia que identificasse, no caso de um número ser ímpar e outro par, se o primeiro é ímpar e o segundo par e vice-versa. A solução é interessante? Justifique.

Não. Isso aumentaria o número comandos que o programa precisaria executar e isso reduziria sua eficiência.

3.6 Múltiplas possibilidades

Existem situações em que uma variável sozinha pode indicar múltiplas possibilidades de execução de um algoritmo. Nessas situações, essa variável funciona como um seletor de opções, semelhante a escolher um canal de TV ou uma estação de Rádio.

Figura 3.26 – Botão seletor moderno
Fonte: Banco de imagens Shutterstock (2016)

- Elabore um algoritmo que receba uma variável qualquer inteira. Se esta tiver o valor 1 (um), imprima “vermelho”, se tiver o valor 2 (dois) imprima “verde” e em outras situações, imprima “amarelo”.

Solução: Diferentemente das situações vistas até aqui, em que as tomadas de decisão ocorrem em função de condições que não dependem de uma única variável, teremos agora uma situação em que uma única variável permite controlar uma série de situações.

```
algoritmo semaforo_03;
variáveis
    a : inteiro;
fim-variáveis
início
    a := leia();
    se a = 1 então
        imprima("Vermelho");
    senão
        se a = 2 então
              imprima("Verde");
           senão
              imprima("Amarelo");
        fim-se
    fim-se
fim
```

Figura 3.27 – Exemplo de variável de seleção
Fonte: Elaborado pelo autor (2015)

Notamos que fazermos uma série de comparações, entretanto parece ser possível “resolver” a questão simplesmente analisando, se possível, um conjunto de possibilidades simultaneamente e, uma vez definida cada situação, definiríamos um fluxo para cada uma dessas possibilidades. Vamos ver como fazer isso, então, usando a instrução “caso”.

3.6.1 Sintaxe da instrução caso

```
escolha <variável>:

            caso <valor 1> :
```

```
                    <instrução 1>;
                    fim-caso
          [     caso <valor 2>:
                    <instrução 2>;
                    fim-caso
                caso <valor n>:
                    <instrução n>;
                    fim-caso
                outrocaso:
                    <instrução>;
                    fim-caso ]
          fim-escolha
```

Notamos pela sintaxe anterior, que poderemos tratar as várias situações e, opcionalmente, poderemos ainda resolver situações que não foram previamente separadas. Retomemos então nosso programa semáforo.

```
algoritmo semaforo_04;
variáveis
    a : inteiro;
fim-variáveis
início
    a := leia();
    escolha a:
      caso 1:
          imprima("Vermelho");
      fim-caso
      caso 2:
          imprima("Verde");
      fim-caso
      outrocaso:
          imprima("Amarelo");
      fim-caso
    fim-escolha
fim
```

Figura 3.28 – Exemplo de variável de seleção utilizando a instrução “caso”
Fonte: Elaborado pelo autor (2015)

É fácil observar que a estrutura lógica equivale e pode ser substituída por uma série de decisões do tipo “se então senão se...”, mas a complexidade ao menos de leitura causada pelos inúmeros aninhamentos, aumenta exponencialmente quando usamos essa estratégia.

3.7 Exemplos com variáveis seletoras

Nessa seção, serão apresentados vários exemplos, apresentando técnicas, estratégias e outros aspectos importantes a serem aprendidos, antes da finalização desse módulo com uma bateria de exercícios.

- Elabore um algoritmo que leia o valor de dois números inteiros e a operação aritmética desejada; calcule e exiba em tela a resposta adequada. Utilize os símbolos do quadro a seguir para ler qual a operação aritmética escolhida.

Símbolo	Operação Aritmética
Soma (+)	Adição
Subtração (-)	Subtração
Multiplicação (*)	Multiplicação
Divisão (/)	Divisão
Resto da divisão (%)	Divisão retornando o resto

Quadro 3.1 – Símbolos para operações aritméticas
Fonte: FIAP (2016)

Embora nenhum tratamento especial seja necessário nas adições, multiplicações e subtrações, o mesmo não ocorre nas divisões, nas quais precisamos garantir que o denominador jamais seja igual a zero, sob risco de gerarmos um erro matemático que abortará o programa (“Não dividirás por zero!”)

```
algoritmo calculadora;
variáveis
n1, n2 : inteiro;
resultado : real;
simbolo : literal;
fim-variáveis
início
    n1 := leia();
    n2 := leia();
    imprima("Escolha o tipo de operação");
    imprima("+ para Soma");
    imprima("- para Subtração");
    imprima("* para Multiplicação");
    imprima("/ para Divisão");
    simbolo := leia();
    escolha simbolo:
        caso "+":
            resultado := n1 + n2;
            imprima("A soma resulta em: ", resultado);
        fim-caso
        caso "-":
            resultado := n1 - n2;
            imprima("A subtração resulta em: ", resultado);
        fim-caso
        caso "*":
            resultado := n1 * n2;
            imprima("A multiplicação resulta em: ", resultado);
        fim-caso
        caso "/":
            se n2 = 0 então
                imprima("Denominador igual a zero. Não existe divisão.");
             senão
                 resultado := n1 / n2;
                imprima("A divisão resulta em: ", resultado);
            fim-se
        fim-caso
        outrocaso:
            imprima("Opção Inexistente");
        fim-caso
    fim-escolha
fim
```

Figura 3.29 – Exemplo de calculadora utilizando a instrução “caso”
Fonte: Elaborado pelo autor (2015)

- Dado um número, imprima:

incluir, se o número for 1;

alterar, se o número for 2;

excluir, se o número for 3;

consultar, se o número for 4; opção inexistente, nos demais casos.

O exemplo, embora muito simples, retrata uma situação bastante usual em que um conjunto de opções é apresentado numa espécie de menu, que resulta no seguinte código:

```
algoritmo seletor;
variáveis
    seletor : inteiro;
fim-variáveis
início
    imprima("Escolha:");
    imprima("\n1 para Incluir");
    imprima("\n2 para Alterar");
    imprima("\n3 para Excluir");
    imprima("\n4 para Consultar");
    seletor := leia();
    escolha seletor:
        caso 1:
            imprima("Incluir");
        fim-caso
        caso 2:
            imprima("Alterar");
        fim-caso
        caso 3:
            imprima("Excluir");
        fim-caso
        caso 4:
            imprima("Consultar");
        fim-caso
    fim-escolha
fim
```

Figura 3.30 – Exemplo de seletor de operações utilizando a instrução “caso”
Fonte: Elaborado pelo autor (2015)

3.8 Funções e procedimentos

Conceitua-se uma função ou um procedimento como um bloco de programa com início e fim, que é chamada por outra função ou por um programa principal. Trata-se de um conceito importante, pois permite que determinadas rotinas sejam reaproveitadas sem que exista a necessidade de criar programas idênticos.

O programa chamador da função pode ou não passar parâmetros para a função, que por sua vez pode ou não devolver valores ao programa chamador.

Nos algoritmos desenvolvidos, tivemos uma situação em que precisamos trocar os valores de variáveis entre si. Embora seja um processo trivial, se tivéssemos uma "função" que permitisse escrevermos algo como: Se a > b então Troque (a,b) evitaríamos eventuais erros de digitação, por exemplo.

Além disso, se tivermos rotinas escritas dessa forma, também possibilitamos aos programadores reaproveitarem esses códigos, isto é, podem-se ter rotinas complexas sendo usada por desenvolvedores que, em tese, nem teriam condições de construir tais estruturas.

Vamos começar a estudar como construir funções e procedimentos.

3.8.1 Função para soma de dois números

Uma função muito simples é a que recebe dois números e retorna sua soma. Primeiramente, vamos fazer esse "programa" da forma mais trivial, sem usarmos uma função:

```
algoritmo soma;
variáveis
    a, b, s : inteiro;
fim-variáveis
início
    imprima("Digite o número 1:");
    a := leia();
    imprima("Digite o número 2:");
    b := leia();
    s := a + b;
    imprima("A soma dos dois números é: ", s);
fim
```

Figura 3.31 – Exemplo de programa que soma dois números fornecidos pelo usuário
Fonte: Elaborado pelo autor (2015)

Na verdade, apenas retomamos o "velho" conhecido programa soma de dois números, realizado no capítulo anterior. Vamos agora refazê-lo, somente que usando uma função.

```
algoritmo funcao_soma;
variáveis
    a, b, r : inteiro;
fim-variáveis
início
    a := leia();
    b := leia();
    r := Soma(a,b);
    imprima(r);
fim
função Soma (n1 : inteiro, n2 : inteiro) : inteiro
início
    retorne n1 + n2;
fim
```

Figura 3.32 – Exemplo de programa que soma dois números fornecidos pelo usuário com uma função
Fonte: Elaborado pelo autor (2015)

Pudemos notar que a soma foi realizada numa rotina à parte que, de posse dos valores digitados, fez o cálculo e o retornou o valor e seu processamento ao programa chamador. Portanto, a função recebe dois parâmetros de entrada de informação, os utiliza no cálculo e retorna um valor inteiro como saída, ao programa que o chamou.

3.8.2 Funções internas a algoritmos principais

Certas linguagens de programação têm funções internas a algoritmos principais.

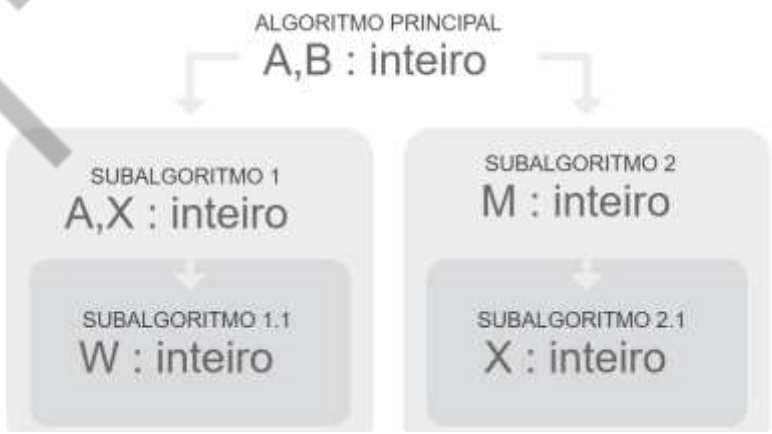


Figura 3.33 – Algoritmo principal
Fonte: Elaborado pelo autor, adaptado por FIAP (2015)

Observando a figura anterior, notamos que “A” e “B” são variáveis globais, ou seja, são vistas por qualquer subalgoritmo (função/procedimento).

Agora, no caso do subalgoritmo 1, que possui as variáveis “X” e “A”, note-se que “A”, quando for referida pelo subalgoritmo 1.1 será diferente de “A” do algoritmo principal. Torna-se duas variáveis batizadas da mesma forma e, por isso, podem assumir valores diferentes. “X” não existe para o programa principal, embora exista para os subalgoritmos 1 e 1.1.

Naturalmente “X” do subalgoritmo 2.1 será distinta de X do subalgoritmo 1.

Esse estilo de programação é usado em linguagens como Pascal, por exemplo.

3.8.3 Funções independentes de algoritmos principais

Outra concepção de implementação de funções é a que considera a situação em que cada função trabalha paralelamente a outras, como na figura 3, a seguir.

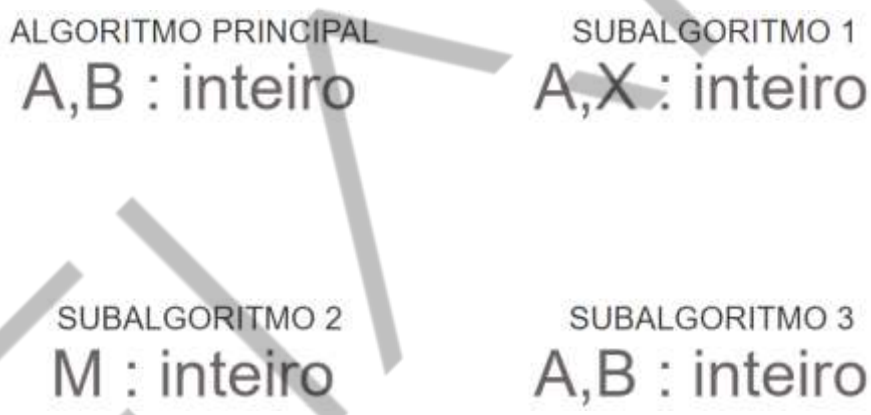


Figura 3.34 – Funções independentes
Fonte: Elaborado pelo autor, adaptado por FIAP (2015)

Em todos esses Subalgoritmos e algoritmos, A, B, M ou X são vistos apenas internamente a cada um desses algoritmos, ou seja, o fato de um algoritmo ser “principal” não torna nenhuma de suas variáveis globais ou visualizadas por algoritmos mais internos.

3.8.4 Passagem de parâmetros entre funções

Quando concluída a execução de uma função, esta coloca o valor de retorno num registrador, devolve o controle ao programa ou função chamadora.

Deve ser observado que parâmetros podem ser passados por **valor** ou por **referência**.

3.8.5 Passagem de parâmetros entre funções por valor

Considere-se a caixa do desenho, **passagem por valor** é equivalente a abrir a caixa, copiar o conteúdo (valor da variável) e só dar este valor ao procedimento. Obviamente, o valor original ficará na caixa e não mudará.

Figura 3.35 – Passagem de valor
Fonte: FIAP (2015)

No exemplo a seguir, embora seja solicitada uma troca, esta ocorre apenas nas variáveis “n1” e “n2”, mas não entre “a” e “b”, as variáveis que foram usadas para passagem dos parâmetros.

```
Algoritmo "TentandoComValores";

Var

a,b: Inteiro


procedimento trocarComValores(x,y: Inteiro)

Var

 z: Inteiro

inicio

z<-x

x<-y

y<-z

fimprocedimento
```

```
Inicio

leia(a)

leia(b)

trocarComValores(a,b)

Escreval(a)

Escreval(b)


Fimalgoritmo
```

Código-Fonte 3.1 – Exemplo de função com passagem de valor
Fonte: FIAP (2019)

Explicação: Nesse caso, quando passamos os valores a e b para **de**ntro do procedimento através da linha *trocarComValores(a,b)*, o conteúdo **de**ssas variáveis está sendo copiado para **de**ntro dos argumentos x e y. Ou seja, as alterações que fizemos com x e y impactam apenas as cópias, e não os valores originais. É por isso que quando exibimos a e b nas linhas *Escreval(a)* e *Escreval(b)* não houve alteração nenhuma.

3.8.6 Passagem de parâmetros entre funções por referência

Já na **passagem por referência** é dar a caixa (variável) ao procedimento e o procedimento pode usar e mudar o valor na caixa. No fim, o procedimento devolve a caixa com o novo valor.

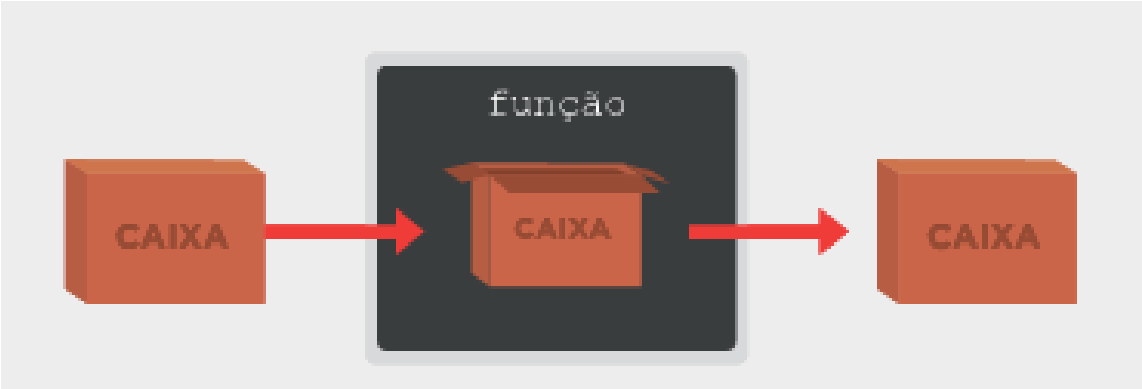


Figura 3.36 – Passagem por referência
Fonte: FIAP (2015)

```
Algoritmo "TentandoComReferencia";
Var
a,b: Inteiro

procedimento trocarComReferencia(var x,y: Inteiro)
Var
 z: Inteiro
inicio
z<-x
x<-y
y<-z
fimprocedimento

Inicio
leia(a)
leia(b)
trocarComReferencia(a,b)
Escreval(a)
Escreval(b)

Fimalgoritmo
```

Código-Fonte 3.2 – Exemplo de função com passagem por referência
Fonte: FIAP (2015)

Nesse caso, quando passamos os valores a e b para dentro do procedimento através da linha *trocarComReferencia(a,b)*, o ENDEREÇO dessas variáveis está sendo passado para dentro dos argumentos x e y. Ou seja, as alterações que fizemos com x e y vão ser direcionados para os endereços de memória dos valores originais. É por isso que quando exibimos a e b nas linhas *Escreval(a)* e *Escreval(b)* verificamos alterações: os ENDEREÇOS DE MEMÓRIA onde estavam guardados os valores originais foram acessados diretamente pelo procedimento.

3.9 Exercícios de fixação do capítulo

Nessa seção temos uma bateria de exercícios que permitirão avaliar se o conhecimento adquirido permite você passar ao próximo conteúdo. É fundamental fazer todos os exercícios antes de se aventurar em novos conteúdos, mesmo porque, os conhecimentos aqui adquiridos serão necessários adiante.

a) Dados dois números, imprima “azul” se ambos forem positivos, “laranja” se ambos forem ímpares ou “roxo” nos demais casos.

```
Algoritmo "Exercicio_Fixação_A"

// Algoritmo desenvolvido em VisualG 3.0

Var

numero_1: real;
numero_2: real;
teste_1: logico;
teste_2: logico;

Inicio
```

```
escreva("Digite o primeiro numero: ")

leia(numero_1)

escreva("Digite o segundo numero: ")

leia(numero_2)

teste_1 := (numero_1 > 0) e (numero_2 > 0)
teste_2 := (numero_1 MOD 2 = 1) e (numero_2 MOD 2 = 1)



SE teste_1 ENTÃO
  escreva ("Azul ")
SENAO
  SE teste_2 ENTÃO
    escreva ("Laranja")
  SENÃO
    escreva ("Roxo")
  FIMSE
FIMSE



Fimalgoritmo
```

b) Dados três números, necessariamente distintos entre si, imprima a soma dos dois maiores.

```
algoritmo "Exercicio_Fixação_b"
Var
num1: real;
num2: real;
num3: real;
menor: real;
soma: real;

Inicio

menor := 99999999999999999999

soma := 0

escreva ("Digite o priemeiro numero: ")

leia (num1)

escreva ("Digite o segundo numero: ")

leia (num2)

escreva ("Digite o terceiro numero: ")
```

```
leia (num3)

SE num1 < menor ENTÃO
  menor := num1
FIMSE

SE num2 < menor ENTÃO
  menor := num2
FIMSE

SE num3 < menor ENTÃO
  menor := num3
FIMSE

SE num1 <> menor ENTÃO
  soma := soma + num1
FIMSE

SE num2 <> menor ENTÃO
  soma := soma + num2
FIMSE

SE num3 <> menor ENTÃO
  soma := soma + num3
FIMSE
```

```
escreval ("O menor numero é: ", menor)


escreval ("A soma dos 2 maiores numeros é: ", soma)


fimalgoritmo
```

c) Dados três números, por hipótese distintos entre si, imprima-os em ordem crescente.

```
algoritmo "Exercicio_Fixação_c"


var


a: inteiro;
b: inteiro;
c: inteiro;


inicio
escreva("Entre com três números positivos e naturais: ")
leia(a,b,c)


se (a>b)e (b>=c) ENTÃO
  escreva(c,b,a)
senao
  se (b>=a)e(b>c)e(a>=c)ENTÃO
    escreva(c,a,
  senao
    se(b>a)e(b>=c)e(a<=c)ENTÃO
```

```
        escreva(a,c,
      senao
        se (b>a)e(b<=c)ENTÃO
          escreva(a,b,c)
        senao
          se(b<a)e(a<c)ENTÃO
            escreva(b,a,c)
          senao
            se(b<c)e(c<=a)ENTÃO
              escreva(b,c,a)
            senao
            escreval(a,b,c)
            fimse
          fimse
        fimse
      fimse
    fimse
  fimse
fimalgoritmo
```

d) Certa companhia de seguros possui nove categorias de seguro, em função da idade e do tipo de trabalho do segurado, em função da tabela apresentada a seguir:

Tabela 3.1 – Idade x Risco

Idade	Baixo Risco	Médio Risco	Alto Risco
17 a 20	1	2	3
21 a 24	2	3	4
25 a 34	3	4	5
35 a 64	4	5	6
65 a 70	7	8	9

Fonte: FIAP (2015)

Elabore um algoritmo que solicite ao usuário a idade e o nível de risco (“B” para “baixo risco”, “M” para “médio risco” e “A” para “alto risco”) e a partir deles calcule a categoria do seguro, exibindo o número de 1 a 9 na tela.

```
algoritmo "Exercicio_Fixação_d"

var

   idade: inteiro;
   risco: caractere;

inicio

   escreva("Informe a idade do segurado: ")
   leia(idade)

   escreva("Informe a categoria de risco do segurado (B, M ou A): ")
   leia(risco)

   se (idade>16)e (idade<21) ENTÃO

      escolha (risco)
      caso "B"
```

```
            escreval ("Categoria de risco 1")
         caso "M"
            escreval ("Categoria de risco 2")
         caso "A"
            escreval ("Categoria de risco 3")
         fimescolha

      senao

         se (idade>20)e (idade<25) ENTÃO

            escolha (risco)
            caso "B"
               escreval ("Categoria de risco 2")
            caso "M"
               escreval ("Categoria de risco 3")
            caso "A"
               escreval ("Categoria de risco 4")
            fimescolha

         senao

            se (idade>24)e (idade<35) ENTÃO

               escolha (risco)
               caso "B"
                  escreval ("Categoria de risco 3")
               caso "M"
                  escreval ("Categoria de risco 4")
               caso "A"
                  escreval ("Categoria de risco 5")
               fimescolha
```

```
            senao

               se (idade>34)e (idade<65) ENTÃO

                  escolha (risco)
                  caso "B"
                     escreval ("Categoria de risco 4")
                  caso "M"
                     escreval ("Categoria de risco 5")
                  caso "A"
                     escreval ("Categoria de risco 6")
                  fimescolha
               senao

                  se (idade>65)e (idade<71) ENTÃO

                     escolha (risco)
                     caso "B"
                        escreval ("Categoria de risco 7")
                     caso "M"
                        escreval ("Categoria de risco 8")
                     caso "A"
                        escreval ("Categoria de risco 9")
                     fimescolha

                  fimse
               fimse
            fimse
         fimse
      fimse
   fimalgoritmo
```

e) Elabore um algoritmo que solicite ao usuário dois números inteiros e positivos, imprima “Pares” se ambos forem pares, “Ímpares” se ambos forem ímpares ou “Outro” nos demais casos.

```
algoritmo "Exercicio_Fixação_e"

var

   numero_1: inteiro;
   numero_2: inteiro;

inicio

   escreva("Informe o primeiro numero: ")
   leia(numero_1)

   escreva("Informe o segundo numero: ")
   leia(numero_2)

   se (numero_1 MOD 2 = 0) e (numero_2 MOD 2 = 0) ENTÃO
      escreval ("Pares")
   senao
      se (numero_1 MOD 2 = 1) e (numero_2 MOD 2 = 1) ENTÃO
         escreval ("Impares")
      senao
         escreval ("Outro")
      fimse
   fimse

fimalgoritmo
```

f) Elabore um algoritmo que solicite ao usuário três números inteiros e imprima a subtração do maior pelo menor. Assuma como verdade que os números informados são diferentes entre si.

```
algoritmo "Exercicio_Fixação_f"
Var
   num1: real;
   num2: real;
   num3: real;
   menor: real;
   maior: real;
   subtracao: real;

Inicio

   escreva ("Digite o priemeiro numero: ")

   leia (num1)

   escreva ("Digite o segundo numero: ")

   leia (num2)

   escreva ("Digite o terceiro numero: ")

   leia (num3)

   SE (num1 > num2) e (num1 > num3) e (num2 > num3) ENTÃO
      menor := num3
      maior := num1
   SENÃO
      SE (num1 > num2) e (num1 > num3) e (num3 > num2) ENTÃO
         menor := num2
         maior := num1
      SENÃO
         SE (num2 > num1) e (num2 > num3) e (num1 > num3) ENTÃO
            menor := num3
            maior := num2
         SENÃO
            SE (num2 > num1) e (num2 > num3) e (num3 > num1) ENTÃO
```

```
                    menor := num1
                    maior := num2
                SENÃO
                    SE (num3 > num1) e (num3 > num2) e (num1 > num2) ENTÃO
                        menor := num2
                        maior := num3
                    SENÃO
                        SE (num3 > num1) e (num3 > num2) e (num2 > num1) ENTÃO
                            menor := num1
                            maior := num3
                        FIMSE
                    FIMSE
                FIMSE
            FIMSE
        FIMSE
    FIMSE


    subtracao := maior - menor

    escreval ("O maior numero é: ", maior)

    escreval ("O menor numero é: ", menor)

    escreval ("A subtraçao do maior número pelo menor é: ", subtracao)

fimalgoritmo
```

g) Elabore um algoritmo que solicite ao usuário três números inteiros e imprima a subtração do maior pelo menor, caso estes sejam diferentes entre si. Sendo iguais, informe impossibilidade de cálculo.

```
algoritmo "Exercicio_Fixação_g"
```

```
Var
   num1: real;
   num2: real;
   num3: real;
   menor: real;
   maior: real;
   subtracao: real;
   iguais: logico;

Inicio

   escreva ("Digite o priemeiro numero: ")

   leia (num1)

   escreva ("Digite o segundo numero: ")

   leia (num2)

   escreva ("Digite o terceiro numero: ")

   leia (num3)

   SE (num1 > num2) e (num1 > num3) e (num2 > num3) ENTÃO
      menor := num3
      maior := num1
   SENÃO
      SE (num1 > num2) e (num1 > num3) e (num3 > num2) ENTÃO
         menor := num2
         maior := num1
      SENÃO
         SE (num2 > num1) e (num2 > num3) e (num1 > num3) ENTÃO
            menor := num3
            maior := num2
         SENÃO
            SE (num2 > num1) e (num2 > num3) e (num3 > num1) ENTÃO
```

```
                    menor := num1
                    maior := num2
                SENÃO
                    SE (num3 > num1) e (num3 > num2) e (num1 > num2) ENTÃO
                        menor := num2
                        maior := num3
                    SENÃO
                        SE (num3 > num1) e (num3 > num2) e (num2 > num1) ENTÃO
                            menor := num1
                            maior := num3
                        SENÃO
                            iguais := VERDADEIRO
                        FIMSE
                    FIMSE
                FIMSE
            FIMSE
        FIMSE
    FIMSE

    SE (iguais) ENTÃO
        escreval ("Dois ou mais números iguais. Não é possível fazer a operação")
    SENÃO
        subtracao := maior - menor
        escreval ("O maior numero é: ", maior)
        escreval ("O menor numero é: ", menor)
        escreval ("A subtraçao do maior número pelo menor é: ", subtracao)
    FIMSE

fimalgoritmo
```

h) Considere a tabela adiante:

Tabela 3.2 – Idade x Fator

Idade	Fator
17 a 20	1
21 a 24	2
25 a 34	3
35 a 64	4
65 a 70	7

Fonte: FIAP (2015)

O valor da mensalidade é calculado a partir do risco. Elabore um algoritmo que solicite do usuário a idade da pessoa, seu sexo (“M” para “Masculino”, “F” para “Feminino”) e o valor base de mensalidade, calculando e exibindo em tela o valor final da seguinte forma:

- Caso seja um homem: Valor Final = Valor Base * Fator.
- Caso seja uma mulher: Valor Final = Valor Base * Fator * 0,8.

```
algoritmo "Exercicio_Fixação_h"

var

   idade: inteiro;
   valor_base: real;
   valor_final: real;
   valor_final1: real;
   genero: caractere;

inicio

   escreva("Informe a idade do segurado: ")
   leia(idade)

   escreva ("Informe o valor base da mensalidade: ")
   leia (valor_base)

   escreva("Informe o genero do segurado (M ou F): ")
   leia(genero)

   se (idade>16)e (idade<21) ENTÃO

      escolha (genero)
```

```
            caso "F"
               valor_final := valor_base * 1 * (8/10)
            caso "M"
               valor_final := valor_base * 1
            fimescolha

         senao

            se (idade>20)e (idade<25) ENTÃO

               escolha (genero)
               caso "M"
                  valor_final := valor_base * 2
               caso "F"
                  valor_final := valor_base * 2 * (8/10)
               fimescolha

            senao

               se (idade>24)e (idade<35) ENTÃO

                  escolha (genero)
                  caso "M"
                     valor_final := valor_base * 3
                  caso "F"
                     valor_final := valor_base * 3 * (8/10)
                  fimescolha

               senao

                  se (idade>34)e (idade<65) ENTÃO

                     escolha (genero)
                     caso "M"
                        valor_final := valor_base * 4
                     caso "F"
                        valor_final := valor_base * 4 * (8/10)
                     fimescolha
                  senao

                     se (idade>65)e (idade<71) ENTÃO

                        escolha (genero)
                        caso "M"
                           valor_final := valor_base * 7
```

```
                    caso "F"
                       valor_final := valor_base * 7 * (8/10)
                    fimescolha

                 fimse
              fimse
           fimse
        fimse
     fimse

     escreva ("Valor final da mensalidade: ", valor_final)

fimalgoritmo
```

i) Elabore um algoritmo que solicite do usuário um número inteiro e imprima emtela:

- 0, caso o número seja positivo e par.
- 1, caso o número seja ímpar e negativo.
- 2, caso o número seja zero.
- 3, nos demais casos.

```
algoritmo "Exercicio_Fixação_i"

var

   numero: inteiro;

inicio

   escreva("Informe um número inteiro: ")
   leia(numero)

   se (numero MOD 2 = 0 )e (numero > 0) ENTÃO
      escreva ("0 - Número positivo e par")

   senao

      se (numero MOD 2 = -1)e (numero < 0) ENTÃO
         escreva ("1 - Número negativo e ímpar")

      senao
```

```
            se (numero = 0) ENTÃO

               escreva ("2 - Número igual a zero")

            senao

               escreva ("3 - Demais casos")

            fimse
         fimse
      fimse

   fimalgoritmo
```

j) Elabore um algoritmo que solicite do usuário três números inteiros e imprima em tela:

- 0, se o primeiro for maior que a soma dos demais.
- 1, se ao menos dois forem iguais.
- 2, se o segundo for par.
- 3, se o primeiro for ímpar.
- 4, nos demais casos

Repare que as situações não são excludentes. Sendo assim, a primeira verificação que se mostrar verdadeira deve ter seu número exibido.

```
algoritmo "Exercicio_Fixação_j"

var

   numero_1: inteiro;
   numero_2: inteiro;
   numero_3: inteiro;

inicio

   escreva("Informe o primeiro número inteiro: ")
   leia(numero_1)

   escreva("Informe o segundo número inteiro: ")
   leia(numero_2)

   escreva("Informe o terceiro número inteiro: ")
   leia(numero_3)

   se (numero_1 > numero_2 + numero_3) ENTÃO
      escreva ("0 - O primeiro número é maior que a soma dos demais números")
```

```
        senao

           se (numero_1 = numero_2) ou (numero_1 = numero_3) ou (numero_2 = numero_3)  ENTÃO
              escreva ("1 - Dois dos números informados são iguais")

           senao

              se (numero_2 MOD 2 = 0) ENTÃO
                 escreva ("2 - O segundo número é par")

              senao

                 se (numero_1 MOD 2 = 1) ENTÃO
                    escreva ("3 - O primeiro número é impar")

                 senao

                    escreva ("4 - Demais casos")

                 fimse
              fimse
           fimse
        fimse

     fimalgoritmo
```

k) Certo país usa para cartas simples selos de 3 e de 5 centavos. A taxa mínima existente (peso da carta) é de 8 centavos. Deseja-se determinar o menor número de selos de 3 e 5 centavos combinados que completem uma taxa dada. Elabore algoritmo que faça esse cálculo para as caixas.

Exemplos:

8 = 3 + 5	11 = 3 + 8	14 = 3 + 11
9 = 3 * 3	12 = 3 + 9	15 = 3 + 12
10 = 5 * 2	13 = 3 + 10	16 = 3 + 13

(Portanto, uma postagem de 16 centavos precisa usar 2 selos de 3 centavos (13 = 3 + 10 e 16 = 3 + 13) e 2 selos de 5 centavos (10 = 5 * 2). Repare que os valores acima são exemplos, utilize a estratégia que julgar melhor).

```
      algoritmo "Exercicio_Fixação_k"


    var


       taxa: inteiro;
```

```
      tres: inteiro;
      cinco: inteiro;

   inicio

      escreva("Informe a taxa cobrada pelo correio: ")
      leia(taxa)

      se (taxa MOD 5 = 0) ENTÃO
         cinco := INT(taxa / 5)
      SENÃO

         se (taxa MOD 5 = 1) ENTÃO
            cinco := INT((taxa - 6) / 5)
            tres := 2
         SENÃO

            se (taxa MOD 5 = 2) ENTÃO
               cinco := INT((taxa - 12) / 5)
               tres := 4
            SENÃO

               se (taxa MOD 5 = 3) ENTÃO
                  cinco := INT((taxa - 3) / 5)
                  tres := 1
               SENÃO

                  se (taxa MOD 5 = 4) ENTÃO
                     cinco := INT((taxa - 9) / 5)
                     tres := 3
                  fimse
               fimse
            fimse
```

```
            fimse
         fimse


         escreval ("Para enviar a carta é preciso: ")
         escreval (cinco, " selos de cinco centavos ")
         escreval (tres, " selos de tres centavos ")


      fimalgoritmo
```

REFERÊNCIAS

ENCYCLOPEDIA and history of programming languages. [s.d.]. Disponível em: <http://www.scriptol.org/>. Acesso em: 14 jan. 2011.

FEOFILOFF, Paulo. **Algoritmos em Linguagem C.** Rio de Janeiro: Campus, 2009.

FORBELLONE, André L.V.; EBERSPACHER, Henri F. **Construção de Algoritmos e Estruturas de Dados**. São Paulo: Pearson Prentice Hall, 2010.

FURGERI, Sérgio. **Java 2, Ensino Didático***.* São Paulo: Érica, 2002.

GANE, Chris e SARSON, Trish. **Análise Estruturada de Sistemas***.* São Paulo: LTC, 1983.

GONDO, Eduardo. **Apostila: Notas de Aula***.* São Paulo, 2008.

LATORE, Robert. **Aprenda em 24 horas Estrutura de Dados e Algoritmos***.* Rio de Janeiro: Campus, 1999.

MANZANO, José A.N.G. e OLIVEIRA, Jayr F. **Algoritmos:** Lógica para o Desenvolvimento de Programação. 23.ed. São Paulo: Érica, 2010.

PIVA JUNIOR, Dilermando et al. **Algoritmos e Programação de Computadores.** Rio de Janeiro: Campus, 2012.

PUGA, Sandra; RISSETTI, Gerson. **Lógica de Programação e Estrutura de Dados**. São Paulo: Pearson Prentice Hall, 2009.

ROCHA, Antonio Adrego. **Estrutura de Dados e Algoritmos em Java***.* Lisboa: FCA-Editora de Informática, 2011.

RODRIGUES, Rita. *Apostila: Notas de Aula.* 2008.

SALVETTI, Dirceu Douglas e BARBOSA, Lisbete Madsen. **Algoritmos***.* São Paulo: Makron Books, 1998.

SCHILDT, Herbert. **Linguagem C - Guia Prático***.* São Paulo: McGraw Hill, 1989.

WOOD, Steve. **Turbo Pascal – Guia do Usuário***.* São Paulo: McGraw Hill, 1987.

ZIVIANI, Nivio. **Projeto de Algoritmos com implementações em Pascal e C***.* São Paulo: Pioneira, 1999.