Sumário

1 Introdução 13

2 Sobre problemas e soluções 17
   2.1 Contando o número de presentes em um evento 17
   2.2 Trocando os quatro pneus 20
   2.3 Conclusão 21

3 Sobre algoritmos e programas 23
   3.1 O que é um algoritmo? 23
   3.2 O que é um programa? 26
   3.3 Exercícios 28

4 O modelo do computador 29
   4.1 Histórico 29
   4.2 Princípios do modelo 30
      4.2.1 Endereços versus conteúdos 30
      4.2.2 O repertório de instruções 31
      4.2.3 O ciclo de execução de instruções 33
      4.2.4 Exemplo de execução de um programa 33
   4.3 Humanos versus computadores 34
      4.3.1 Abstração dos endereços 35
      4.3.2 Abstração dos códigos das instruções 36
      4.3.3 Abstração do repertório de instruções 37
      4.3.4 Abstração dos endereços de memória (variáveis) 38
   4.4 Abstração das instruções (linguagem) 39
   4.5 Conclusão 41
   4.6 Exercícios 43

5 Conceitos elementares 45
   5.1 Algoritmos e linguagens de programação 45
   5.2 Fluxo de execução de um programa 46
   5.3 Comandos de entrada e saída 47
   5.4 O ciclo edição–compilação–testes 48
   5.5 Atribuições e expressões aritméticas 50
      5.5.1 Exemplo 52
<table>
<thead>
<tr>
<th>5.6</th>
<th>Variáveis e expressões booleanas</th>
<th>53</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.7</td>
<td>Repetição de comandos</td>
<td>54</td>
</tr>
<tr>
<td>5.7.1</td>
<td>Exemplo</td>
<td>57</td>
</tr>
<tr>
<td>5.7.2</td>
<td>Critério de parada</td>
<td>57</td>
</tr>
<tr>
<td>5.7.3</td>
<td>Critérios alternativos de parada da repetição</td>
<td>58</td>
</tr>
<tr>
<td>5.8</td>
<td>Desviando fluxo condicionalmente</td>
<td>59</td>
</tr>
<tr>
<td>5.8.1</td>
<td>Fluxo alternativo no desvio condicional</td>
<td>59</td>
</tr>
<tr>
<td>5.9</td>
<td>Resumo</td>
<td>60</td>
</tr>
<tr>
<td>5.10</td>
<td>Exercícios</td>
<td>63</td>
</tr>
<tr>
<td>5.10.1</td>
<td>Expressões aritméticas</td>
<td>63</td>
</tr>
<tr>
<td>5.10.2</td>
<td>Expressões booleanas</td>
<td>65</td>
</tr>
<tr>
<td>5.10.3</td>
<td>Expressões aritméticas e booleanas</td>
<td>65</td>
</tr>
<tr>
<td>5.10.4</td>
<td>Acompanhamento de programas</td>
<td>66</td>
</tr>
<tr>
<td>5.10.5</td>
<td>Programas com um desvio condicional</td>
<td>66</td>
</tr>
<tr>
<td>5.10.6</td>
<td>Programas com um laço</td>
<td>68</td>
</tr>
<tr>
<td>5.11</td>
<td>Exercícios complementares</td>
<td>70</td>
</tr>
<tr>
<td>5.11.1</td>
<td>Programas com cálculos simples</td>
<td>70</td>
</tr>
<tr>
<td>5.11.2</td>
<td>Programas com cálculos e desvios condicionais</td>
<td>78</td>
</tr>
<tr>
<td>6</td>
<td>Técnicas elementares</td>
<td>87</td>
</tr>
<tr>
<td>6.1</td>
<td>Lógica de programação</td>
<td>87</td>
</tr>
<tr>
<td>6.2</td>
<td>O teste de mesa</td>
<td>91</td>
</tr>
<tr>
<td>6.3</td>
<td>Técnica do acumulador</td>
<td>92</td>
</tr>
<tr>
<td>6.3.1</td>
<td>Exemplo</td>
<td>93</td>
</tr>
<tr>
<td>6.4</td>
<td>Árvores de decisão</td>
<td>93</td>
</tr>
<tr>
<td>6.5</td>
<td>Definir a priori e depois corrigir</td>
<td>95</td>
</tr>
<tr>
<td>6.6</td>
<td>Lembrar de mais de uma informação</td>
<td>96</td>
</tr>
<tr>
<td>6.7</td>
<td>Processar parte dos dados de entrada</td>
<td>97</td>
</tr>
<tr>
<td>6.8</td>
<td>Processar parte dos dados de um modo e outra parte de outro</td>
<td>98</td>
</tr>
<tr>
<td>6.9</td>
<td>Múltiplos acumuladores</td>
<td>99</td>
</tr>
<tr>
<td>6.10</td>
<td>Exercícios</td>
<td>102</td>
</tr>
<tr>
<td>7</td>
<td>Aplicações das técnicas elementares</td>
<td>107</td>
</tr>
<tr>
<td>7.1</td>
<td>Convertendo para binário</td>
<td>107</td>
</tr>
<tr>
<td>7.2</td>
<td>Cálculo do MDC</td>
<td>109</td>
</tr>
<tr>
<td>7.3</td>
<td>Tabuada</td>
<td>110</td>
</tr>
<tr>
<td>7.4</td>
<td>Fatorial</td>
<td>111</td>
</tr>
<tr>
<td>7.5</td>
<td>Números de Fibonacci revisitado</td>
<td>113</td>
</tr>
<tr>
<td>7.5.1</td>
<td>Alterando o critério de parada</td>
<td>113</td>
</tr>
<tr>
<td>7.5.2</td>
<td>O número áureo</td>
<td>114</td>
</tr>
<tr>
<td>7.6</td>
<td>Inverter um número de três dígitos</td>
<td>116</td>
</tr>
<tr>
<td>7.7</td>
<td>Palíndromos</td>
<td>118</td>
</tr>
<tr>
<td>7.8</td>
<td>Maior segmento crescente</td>
<td>120</td>
</tr>
<tr>
<td>7.9</td>
<td>Séries</td>
<td>121</td>
</tr>
<tr>
<td>7.9.1</td>
<td>Número neperiano</td>
<td>121</td>
</tr>
<tr>
<td>SUMÁRIO</td>
<td></td>
<td></td>
</tr>
<tr>
<td>---------</td>
<td>-------</td>
<td></td>
</tr>
<tr>
<td>7.9.2</td>
<td>Cálculo do seno .................................................. 123</td>
<td></td>
</tr>
<tr>
<td>7.10</td>
<td>Números primos .................................................. 126</td>
<td></td>
</tr>
<tr>
<td>7.11</td>
<td>Refinamentos sucessivos ........................................ 129</td>
<td></td>
</tr>
<tr>
<td>7.11.1</td>
<td>Primos entre si ................................................ 129</td>
<td></td>
</tr>
<tr>
<td>7.11.2</td>
<td>Amigos quadráticos .............................................. 131</td>
<td></td>
</tr>
<tr>
<td>7.11.3</td>
<td>Cálculo do MDC pela definição .................................. 135</td>
<td></td>
</tr>
<tr>
<td>7.12</td>
<td>Exercícios ....................................................... 138</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Funções e procedimentos ........................................ 145</td>
<td></td>
</tr>
<tr>
<td>8.1</td>
<td>Motivação ........................................................ 145</td>
<td></td>
</tr>
<tr>
<td>8.1.1</td>
<td>Modularidade .................................................... 146</td>
<td></td>
</tr>
<tr>
<td>8.1.2</td>
<td>Reaproveitamento de código .................................... 146</td>
<td></td>
</tr>
<tr>
<td>8.1.3</td>
<td>Legibilidade .................................................... 146</td>
<td></td>
</tr>
<tr>
<td>8.1.4</td>
<td>Comentário adicional ........................................... 147</td>
<td></td>
</tr>
<tr>
<td>8.2</td>
<td>Noções fundamentais ............................................. 147</td>
<td></td>
</tr>
<tr>
<td>8.2.1</td>
<td>Exemplo básico .................................................. 147</td>
<td></td>
</tr>
<tr>
<td>8.2.2</td>
<td>O programa principal ........................................... 147</td>
<td></td>
</tr>
<tr>
<td>8.2.3</td>
<td>Variáveis globais ............................................... 148</td>
<td></td>
</tr>
<tr>
<td>8.2.4</td>
<td>Funções .......................................................... 148</td>
<td></td>
</tr>
<tr>
<td>8.2.5</td>
<td>Parâmetros por valor ........................................... 150</td>
<td></td>
</tr>
<tr>
<td>8.2.6</td>
<td>Parâmetros por referência ...................................... 151</td>
<td></td>
</tr>
<tr>
<td>8.2.7</td>
<td>Procedimentos ................................................... 152</td>
<td></td>
</tr>
<tr>
<td>8.2.8</td>
<td>Variáveis locais ................................................. 153</td>
<td></td>
</tr>
<tr>
<td>8.3</td>
<td>Alguns exemplos .................................................. 154</td>
<td></td>
</tr>
<tr>
<td>8.3.1</td>
<td>Calculando dígito verificador .................................. 154</td>
<td></td>
</tr>
<tr>
<td>8.3.2</td>
<td>Calculando raízes de equações do segundo grau ............... 156</td>
<td></td>
</tr>
<tr>
<td>8.3.3</td>
<td>Cálculo do MDC pela definição .................................. 159</td>
<td></td>
</tr>
<tr>
<td>8.4</td>
<td>Exercícios ....................................................... 162</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>Estruturas de dados ............................................. 163</td>
<td></td>
</tr>
<tr>
<td>9.1</td>
<td>Vetores .......................................................... 163</td>
<td></td>
</tr>
<tr>
<td>9.1.1</td>
<td>Primeiros problemas com vetores ................................ 166</td>
<td></td>
</tr>
<tr>
<td>9.1.2</td>
<td>Soma e produto escalar de vetores ................................ 173</td>
<td></td>
</tr>
<tr>
<td>9.1.3</td>
<td>Busca em vetores ................................................ 176</td>
<td></td>
</tr>
<tr>
<td>9.1.4</td>
<td>Ordenação em vetores ........................................... 186</td>
<td></td>
</tr>
<tr>
<td>9.1.5</td>
<td>Outros algoritmos com vetores ................................ 190</td>
<td></td>
</tr>
<tr>
<td>9.1.6</td>
<td>Exercícios ....................................................... 202</td>
<td></td>
</tr>
<tr>
<td>9.2</td>
<td>Matrizes .......................................................... 218</td>
<td></td>
</tr>
<tr>
<td>9.2.1</td>
<td>Matrizes em Pascal .............................................. 218</td>
<td></td>
</tr>
<tr>
<td>9.2.2</td>
<td>Exemplos elementares ........................................... 219</td>
<td></td>
</tr>
<tr>
<td>9.2.3</td>
<td>Procurando elementos em matrizes ................................ 225</td>
<td></td>
</tr>
<tr>
<td>9.2.4</td>
<td>Inserindo uma coluna em uma matriz ................................ 226</td>
<td></td>
</tr>
<tr>
<td>9.2.5</td>
<td>Aplicações de matrizes em imagens ................................ 229</td>
<td></td>
</tr>
<tr>
<td>9.2.6</td>
<td>Exercícios ....................................................... 238</td>
<td></td>
</tr>
<tr>
<td>9.3</td>
<td>Registros ........................................................ 253</td>
<td></td>
</tr>
</tbody>
</table>
### SUMÁRIO

| 9.3.1  | Introdução aos registros | 253 |
| 9.3.2  | Registros com vetores     | 254 |
| 9.3.3  | Vetores de registros      | 255 |
| 9.3.4  | Exercícios               | 259 |

| 10     | Tipos abstratos de dados | 267 |
| 10.1   | Tipo Abstrato de Dados Conjunto | 267 |
| 10.1.1 | Usando o TAD Conjunto para resolver problemas | 270 |
| 10.1.2 | Implementações do TAD conjunto | 274 |
| 10.2   | Tipo Abstrato de Dados Pilha | 287 |
| 10.2.1 | Usando o TAD Pilha para resolver um problema | 289 |
| 10.2.2 | Implementações do TAD pilha | 291 |
| 10.3   | Exercícios               | 295 |

| 11     | Aplicações em jogos      | 297 |
| 11.1   | Campo minado            | 297 |
| 11.2   | Exercícios              | 309 |
Lista de Figuras

3.1 Algoritmo para fazer bolo de chocolate. ........................................... 25
3.2 Algoritmo para fazer claras em neve. ............................................. 26

4.1 Uma fotografia da memória. .......................................................... 31
4.2 O repertório de instruções do computador da BCC. ......................... 32
4.3 Separando as instruções. ................................................................. 35
4.4 Abstração dos endereços. ............................................................... 36
4.5 Programa reescrito com Mnemônicos. ............................................. 37
4.6 Notação para as instruções. ............................................................. 38
4.7 Programa escrito sob nova notação. ............................................... 38
4.8 Dando nomes para os endereços. ................................................... 39
4.9 Programa reescrito com nomes para variáveis. ................................. 40
4.10 Duas outras versões do programa. ............................................... 40
4.11 Versão do programa escrito em Pascal. ......................................... 41

5.1 Estrutura de um programa em Pascal. .............................................. 46
5.2 Entrada e saída em Pascal. ............................................................ 47
5.3 Programa que implementa o método de Bhaskara. ............................ 50
5.4 Primeira solução. ............................................................................. 52
5.5 Mesma solução, agora com interface amigável. ................................. 53
5.6 Primeira solução para contar de 1 a 5. ........................................... 55
5.7 Primeira solução modificada para números de 1 a 20. ....................... 55
5.8 Sexta solução. .................................................................................. 56
5.9 Tabela com os quadrados dos números de 1 a 30. ............................ 57
5.10 Duas formas para somar pares de números. .................................. 59
5.11 Imprime se for positivo. ................................................................. 60
5.12 Imprime se for positivo, segunda versão. ...................................... 60
5.13 Exemplo do uso do desvio incondicional. ...................................... 62

6.1 Contando números. ........................................................................... 90
6.2 Contando números. .......................................................................... 91
6.3 Técnica do acumulador. .................................................................. 93
6.4 Solução para o problema de somar N números. ................................ 94
6.5 Imprimir o menor dentre 3 números lidos. ...................................... 94
6.6 Encontrando o menor elemento. ..................................................... 96
6.7 Gerando números da sequência de Fibonacci. ................................. 97
<table>
<thead>
<tr>
<th>Number</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.8</td>
<td>Lendo e imprimindo 30 números.</td>
<td>98</td>
</tr>
<tr>
<td>6.9</td>
<td>Lendo e imprimindo os positivos.</td>
<td>98</td>
</tr>
<tr>
<td>6.10</td>
<td>Lendo e imprimindo os positivos e os quadrados dos outros.</td>
<td>99</td>
</tr>
<tr>
<td>6.11</td>
<td>Um programa para construir um histograma.</td>
<td>100</td>
</tr>
<tr>
<td>6.12</td>
<td>O mesmo programa usando <em>case</em>.</td>
<td>101</td>
</tr>
<tr>
<td>7.1</td>
<td>Convertendo para binário, versão 1.</td>
<td>108</td>
</tr>
<tr>
<td>7.2</td>
<td>Convertendo para binário, versão 2.</td>
<td>108</td>
</tr>
<tr>
<td>7.3</td>
<td>Algoritmo de Euclides para cálculo do MDC.</td>
<td>109</td>
</tr>
<tr>
<td>7.4</td>
<td>Tabuadas do 1 ao 10.</td>
<td>110</td>
</tr>
<tr>
<td>7.5</td>
<td>Saída do programa da tabuada.</td>
<td>110</td>
</tr>
<tr>
<td>7.6</td>
<td>Obtendo o fatorial de <em>n</em>.</td>
<td>111</td>
</tr>
<tr>
<td>7.7</td>
<td>Obtendo vários fatoriais.</td>
<td>112</td>
</tr>
<tr>
<td>7.8</td>
<td>Otimizando o cálculo de vários fatoriais.</td>
<td>113</td>
</tr>
<tr>
<td>7.9</td>
<td>Imprimindo o primeiro número de Fibonacci maior do que 1000.</td>
<td>114</td>
</tr>
<tr>
<td>7.10</td>
<td>Verificando a convergência do número áureo.</td>
<td>115</td>
</tr>
<tr>
<td>7.11</td>
<td>Primeira solução para inverter um número de 3 dígitos.</td>
<td>116</td>
</tr>
<tr>
<td>7.12</td>
<td>Mesmo algoritmo, agora para 4 dígitos.</td>
<td>116</td>
</tr>
<tr>
<td>7.13</td>
<td>Solução com uso de acumuladores.</td>
<td>117</td>
</tr>
<tr>
<td>7.14</td>
<td>Imprimindo todos os palíndromos de 1 a 1000.</td>
<td>118</td>
</tr>
<tr>
<td>7.15</td>
<td>Gerando todos os palíndromos de 1 a 1000.</td>
<td>119</td>
</tr>
<tr>
<td>7.16</td>
<td>Maior segmento crescente.</td>
<td>120</td>
</tr>
<tr>
<td>7.17</td>
<td>Cálculo do número neperiano.</td>
<td>121</td>
</tr>
<tr>
<td>7.18</td>
<td>Cálculo do número neperiano.</td>
<td>122</td>
</tr>
<tr>
<td>7.19</td>
<td>Série com troca de sinais.</td>
<td>123</td>
</tr>
<tr>
<td>7.20</td>
<td>Série com troca de sinais e fatoriais no denominador corrigidos.</td>
<td>124</td>
</tr>
<tr>
<td>7.21</td>
<td>Cálculo do seno de <em>x</em>.</td>
<td>125</td>
</tr>
<tr>
<td>7.22</td>
<td>Verifica se <em>n</em> é primo contando os divisores.</td>
<td>126</td>
</tr>
<tr>
<td>7.23</td>
<td>Testa se <em>n</em> é primo parando no primeiro divisor.</td>
<td>127</td>
</tr>
<tr>
<td>7.24</td>
<td>Testa se <em>n</em> é primo parando na raiz de <em>n</em>.</td>
<td>128</td>
</tr>
<tr>
<td>7.25</td>
<td>Pseudocódigo para o problema dos primos entre si.</td>
<td>129</td>
</tr>
<tr>
<td>7.26</td>
<td>Gerando todos os primos entre si.</td>
<td>130</td>
</tr>
<tr>
<td>7.27</td>
<td>Pseudocódigo para o problema dos amigos quadráticos.</td>
<td>131</td>
</tr>
<tr>
<td>7.28</td>
<td>Pseudo-código para decidir sobre amigos quadráticos.</td>
<td>131</td>
</tr>
<tr>
<td>7.29</td>
<td>Separando os dígitos de um dado número <em>n</em>.</td>
<td>132</td>
</tr>
<tr>
<td>7.30</td>
<td>Somando os dígitos de um dado número <em>n</em>.</td>
<td>132</td>
</tr>
<tr>
<td>7.31</td>
<td>Verificando se números <em>n</em> e <em>m</em> são amigos quadráticos.</td>
<td>133</td>
</tr>
<tr>
<td>7.32</td>
<td>Tornando amigos quadráticos mais eficiente.</td>
<td>134</td>
</tr>
<tr>
<td>7.33</td>
<td>Pseudo-código para o cálculo do MDC pela definição.</td>
<td>135</td>
</tr>
<tr>
<td>7.34</td>
<td>Calcula MDC entre <em>a</em> e <em>b</em> pela definição (caso primo=2).</td>
<td>136</td>
</tr>
<tr>
<td>7.35</td>
<td>Calcula MDC entre <em>a</em> e <em>b</em> pela definição (caso primo é ímpar).</td>
<td>137</td>
</tr>
<tr>
<td>8.1</td>
<td>Programa que imprime os números da entrada que são pares.</td>
<td>147</td>
</tr>
<tr>
<td>8.2</td>
<td>Programa que imprime os números da entrada que são pares.</td>
<td>149</td>
</tr>
<tr>
<td>8.3</td>
<td>Programa que imprime os números da entrada que são pares.</td>
<td>151</td>
</tr>
</tbody>
</table>
8.4 Versão com parâmetros por referência. ........................................... 152
8.5 Versão com parâmetros por referência. ........................................... 153
8.6 Versão com uma variável local. ...................................................... 153
8.7 Calculando dígito verificador. .......................................................... 155
8.8 Calculando raízes de equação do segundo grau. ............................... 157
8.9 Calculando raízes de equação do segundo grau. ............................... 158
8.10 Função que calcula as raízes de equação do segundo grau. ............... 158
8.11 Versão final do programa. .............................................................. 159
8.12 Calcula MDC entre $a$ e $b$ pela definição usando funções. ............. 160
8.13 Calcula quantas vezes um número divide outro. ............................ 161
8.14 Calcula a potência de um número elevado a outro. .......................... 161

9.1 Lendo elementos e colocando no vetor. ........................................... 167
9.2 Lendo elementos e colocando no vetor, usando for. ........................ 167
9.3 Lendo elementos e colocando no vetor, usando repeat. .................... 167
9.4 Lendo e imprimindo usando vetores. .............................................. 168
9.5 Lendo e imprimindo sem usar vetores. ........................................... 168
9.6 Lendo e imprimindo: outra versão. ............................................... 169
9.7 Lendo e imprimindo, agora com procedimentos. ............................... 170
9.8 Procedimento que imprime os elementos do vetor ao contrário. ........ 171
9.9 Lendo e imprimindo ao contrário, versão final. ............................... 171
9.10 Imprimindo os elementos do vetor que são pares. .......................... 173
9.11 Encontrando o menor de $N$ números lidos, sem vetor. .................. 174
9.12 Encontrando o menor de $N$ números lidos, com vetor. .................. 174
9.13 Somando dois vetores. ................................................................. 175
9.14 Produto escalar de dois vetores. ................................................... 176
9.15 Busca em vetores, primeira versão. .............................................. 177
9.16 Busca em vetores, segunda versão. .............................................. 177
9.17 Busca em vetores com sentinela. .................................................. 178
9.18 Busca em vetores ordenados. ........................................................ 179
9.19 Tabela resumindo número de comparações para algoritmos de busca. 180
9.20 Busca binária. .............................................................................. 181
9.21 Removendo de um vetor ordenado. .............................................. 181
9.22 Inserindo em um vetor ordenado. ............................................... 183
9.23 Fundindo dois vetores ordenados. ................................................ 185
9.24 Método de ordenação por seleção. .............................................. 187
9.25 Método de ordenação por inserção. .............................................. 190
9.26 Verifica se um vetor define uma permutação. ............................... 191
9.27 Verifica linearmente se um vetor define uma permutação. ............... 192
9.28 Gerando uma permutação, versão 1. ............................................. 193
9.29 Gerando uma permutação, versão 2. ............................................. 193
9.30 Gerando uma permutação, versão 3. ............................................. 194
9.31 Gerando uma permutação, versão 4. ............................................. 195
9.32 Calcula a ordem de uma permutação. ......................................... 197
9.33 Calcula o valor de $P(x)$ para um dado $x \in \mathbb{R}$. ..................... 198
9.34 Calcula o polinómio derivada de $P(x)$. ........................................ 199
9.35 Calcula o valor de $P(x)$ para um dado $x \in \mathbb{R}$. ................. 199
9.36 Calcula a soma de $P(x)$ com $Q(x)$. ........................................ 200
9.37 Calcula o produto de $P(x)$ com $Q(x)$. ....................................... 201
9.38 Lendo uma matriz ................................................................. 219
9.39 Procedimento para ler uma matriz ............................................. 220
9.40 Procedimento para imprimir uma matriz ...................................... 220
9.41 Procedimento para imprimir a transposta de uma matriz ............... 221
9.42 Procedimento para imprimir uma única linha da matriz ................ 221
9.43 Procedimento para imprimir uma única coluna da matriz ................ 222
9.44 Procedimento para imprimir os elementos pares matriz ................. 222
9.45 Procedimento para imprimir os elementos cujos índices são pares. ... 223
9.46 Encontrando o menor elemento de uma matriz ............................ 223
9.47 Somando duas matrizes .......................................................... 224
9.48 Produto escalar de uma linha da matriz por uma coluna da outra ... 225
9.49 Multiplicação de duas matrizes .................................................. 225
9.50 Busca em uma matriz .............................................................. 226
9.51 Verifica se uma matriz tem elementos repetidos ........................... 227
9.52 Insere um vetor como última coluna de uma matriz ..................... 228
9.53 Insere um vetor como K-ésima coluna de uma matriz ................... 228
9.54 Exemplo de imagem no formato PGM ........................................ 230
9.55 Leitura de uma imagem PGM .................................................... 231
9.56 Impressão de uma imagem PGM ............................................... 231
9.57 Clareando uma imagem .......................................................... 232
9.58 Procedure para clarear uma imagem PGM .................................. 232
9.59 Reduzindo uma imagem .......................................................... 233
9.60 Visualizando uma matriz $4 \times 4$ em quatro blocos $2 \times 2$ .......... 233
9.61 Procedure para fazer zoom em uma imagem PGM ....................... 234
9.62 Função que calcula média dos quatro vizinhos de um pixel .......... 234
9.63 Cálculo do valor do maior pixel ................................................. 235
9.64 Detecção de bordas em uma imagem .......................................... 235
9.65 Procedure para detectar bordas de uma imagem PGM .................. 237
9.66 Imprimindo registros ............................................................... 254
9.67 Lendo vetores implementados em registros ................................. 255
9.68 Lendo os clientes do banco ...................................................... 256
9.69 Imprime o telefone do cliente que tem um certo CPF ................. 256
9.70 Ordena pelo CPF ................................................................. 257
9.71 Troca os elementos do banco de dados ..................................... 258
10.1 Programa que usa o TAD conjunto para o problema da megassena 271
10.2 Lendo os números apostados ou sorteados .................................. 271
10.3 Encontrando uma celebridade usando conjuntos .......................... 273
10.4 Confirmando se um presente é uma celebridade ........................... 273
10.5 Inicializar um conjunto ............................................................ 275
10.6 Testa se um conjunto é vazio ................................................... 275
11.15 Revela todos os quadrados para impressão final. . . . . . . . . . . . . . . . . . 307
11.16 Revela todos os vizinhos sem minas ao redor. . . . . . . . . . . . . . . . . . 308
Capítulo 1

Introdução

Neste texto materializamos o conteúdo ministrado nos últimos anos na disciplina Algoritmos e Estruturas de Dados I, ofertada para os cursos de Ciência da Computação e Informática Biomédica da Universidade Federal do Paraná.

Esta disciplina é ministrada no primeiro semestre (para os calouros) e é a primeira das quatro que cobrem o conteúdo básico de algoritmos sem o qual um curso de Computação não faz sentido. As disciplinas subsequentes são:

- Algoritmos e Estruturas de Dados II;
- Algoritmos e Estruturas de Dados III; e
- Algoritmos e Teoria dos Grafos.

A orientação dos colegiados dos cursos é que esta disciplina deve ter um conteúdo forte em conceitos de algoritmos no qual a implementação final em uma linguagem de programação é vista apenas como um mecanismo facilitador ao aprendizado dos conceitos teóricos.

Os currículos destes cursos contêm duas disciplinas nas quais a arte da programação é explorada ao máximo. Elas são ministradas respectivamente no segundo e terceiros semestres.

- Programação I; e
- Programação II.

Por este motivo, embora adotemos a linguagem Pascal na redação dos diversos programas feitos neste texto, é um livro que se preocupa com algoritmos. Ele contém o mínimo necessário sobre Pascal, somente o suficiente para que os estudantes possam praticar a programação dos algoritmos estudados. Os estudantes são encorajados a buscarem apoio na literatura e nos guias de referência disponíveis para o compilador escolhido, incluindo um guia de referência básico que foi escrito pelos monitores da disciplina no ano de 2009.

O texto está dividido em duas partes bem definidas, a primeira entre os capítulos ?? e ??, contém os princípios básicos da construção de algoritmos elementares, incluindo a parte de subprogramas, passagem de parâmetros e variáveis locais e globais.
A segunda parte, a partir do capítulo 9, contém princípios de estruturas de dados básicas, onde se introduz a noção de vetores uni e multidimensionais. Nestas estruturas, praticamos a elaboração de algoritmos modulares e procuramos mostrar como construir programas pela abordagem de refinamentos sucessivos (top down) na qual um programa principal contém a solução em alto nível para o problema e as funções e procedimentos são detalhadas na medida em que são necessárias.

Alguns algoritmos importantes são estudados em suas versões básicas. Noções de complexidade de algoritmos são mencionadas, ainda que de modo informal, pois isto é conteúdo de períodos mais avançados. Contudo, é importante ao aprendiz ter noção clara da diferença de custo entre diferentes algoritmos que resolvem o mesmo problema. Também apresentamos em caráter introdutório o conceito de tipos abstratos de dados.

Finalmente oferecemos um desafio aos alunos. O objetivo é de mostrar uma aplicação interessante para os conceitos que eles já dominam. Normalmente trabalhamos em sala de aula o desenvolvimento de um programa que tem sido a construção de um jogo simples que pode ser implementado em uma estrutura de matriz usando conceitos de tipos abstratos de dados. A ideia é que eles possam fazer um programa mais extenso para treinarem a construção de programas modulares e desenvolvimento top down.

O estudante não deve iniciar uma parte sem antes ter compreendido o conteúdo da anterior. Também não deve iniciar um novo capítulo sem ter compreendido os anteriores.

A leitura deste texto não livra o estudante de buscar literatura complementar, sempre bem-vinda. Em particular, uma ótima história da computação pode ser encontrada em [Tre83]. Alguns excelentes textos introdutórios em algoritmos estão em [Car82], [SB98], [MF05] e [Wir78]. Para mais detalhes de programação em Pascal o leitor pode consultar [Feo99] e também os guias de referência da linguagem [Pre]. Finalmente, embora talvez de difícil compreensão para um iniciante, recomendamos pelo menos folhar o material em [Km68].

Os autores agradecem imensamente aos seguintes professores que ministraram a disciplina de Algoritmos e Estruturas e Dados I e que de alguma maneira contribuíram com ideias e sugestões para este texto: Alexandre Ibrahim Direne, André Luis Vignatti, Carlos Alberto Maziero, Carme Satie Hara, Letícia Mara Peres, Lucas Ferrari de Oliveira e Luis Carlos Erpen de Bona. Também agradecemos imensamente aos professores Renato José do Carmo e David Menotti Gomes pela autorização do uso de parte de seus materiais, no caso, respectivamente, a base para a redação do capítulo 4 e pelos exercícios complementares ao longo de diversos capítulos.


A propósito, acreditamos que é fundamental para um estudante de Ciência da Computação fazer muitos exercícios, aprender programar exige programar! Dizemos sempre que é mais ou menos como andar de bicicleta: ninguém aprende vendo vídeos.
na Internet, tem que andar de bicicleta para aprender a andar de bicicleta.
Capítulo 2
Sobre problemas e soluções

Vamos iniciar nosso estudo com uma breve discussão sobre problemas e soluções. O objetivo é deixar claro desde o início que:

• não existe, em geral, uma única solução para o mesmo problema;
• algumas soluções são melhores do que outras, sob algum critério;
• problemas e instâncias de problemas são conceitos diferentes;
• nosso foco é em resolver problemas.

Apresentaremos dois problemas reais e para cada um deles seguirá uma discussão sobre a existência de diversas soluções para eles. Daremos ênfase nas diferenças entre as soluções e também sobre até que ponto deve-se ficar satisfeito com a primeira solução obtida ou se ela pode ser generalizada para problemas similares.

2.1 Contando o número de presentes em um evento

No primeiro dia letivo do primeiro semestre de 2009, um dos autores deste material colocou o seguinte problema aos novos alunos: queríamos saber quantos estudantes estavam presentes na sala de aula naquele momento. A sala tinha capacidade aproximada de 100 lugares e a naquele momento estava razoavelmente cheia.

Os estudantes discutiram várias possibilidades. Apresentamos todas elas a seguir.

Primeira solução

A primeira solução parecia tão óbvia que levou algum tempo até algum aluno verbalizar: o professor conta os alunos um por um, tomando o cuidado de não contar alguém duas vezes e também de não esquecer de contar alguém.

Quais são as vantagens deste método? Trata-se de uma solução simples, fácil de executar e produz o resultado correto. É uma solução perfeita para salas de aula com poucos alunos, digamos, 20 ou 30. Outro aspecto considerado foi o fato de que este
método não exige nenhum conhecimento prévio de quem vai executar a operação, a
não ser saber contar. Também não exige nenhum equipamento adicional.

Quais as desvantagens? Se o número de alunos na sala for grande, o tempo ne-
cessário para o término da operação pode ser insatisfatório. Para piorar, quanto
maior o número, maior a chance de aparecerem erros na contagem. Foi discutida a
adequação desta solução para se contar os presentes em um comício ou manifestação
popular numa praça pública. Concluiu-se pela inviabilidade do método nestes casos.

Neste momento ficou claro a diferença entre um problema e uma instância do
problema. No nosso caso, o problema é contar os elementos de um conjunto. A
instância do problema que estamos resolvendo é contar as pessoas presentes naquela
sala de aula, naquele semestre. Outras instâncias são contar os presentes em um
comício específico. Outra é Contar os presentes em um estádio de futebol em um
certo dia de jogo.

Executamos a contagem em aproximadamente 1 minuto. Dois alunos também
fizeram a contagem e, após conferência, obtivemos o resultado correto, que serviu
para análise das outras soluções.

Segunda solução

Pensando no problema de se contar na ordem de 100 alunos, um estudante sugeriu
que se fizesse a contagem das carteiras vazias e em seguida uma subtração com relação
ao número total de carteiras na sala.

A solução também é muito simples e funciona perfeitamente bem, mas exige um
conhecimento prévio: deve-se saber antecipadamente o total de carteiras na sala.

Esta maneira de contar é cada vez melhor quanto maior for o número de presentes,
pois o número de carteiras vazias é menor do que o das ocupadas. Por outro lado, se
a sala estiver com pouca gente, o método anterior é mais eficiente.

Os alunos observaram também que a solução não se aplica para os casos de con-
tagem de presentes a um comício numa praça pública, pois não há carteiras na rua.

Terceira solução

Para resolver o problema do comício, outro estudante sugeriu que se fizesse uma
estimativa baseada na metragem total da praça, multiplicada pelo número estimado
de pessoas por metro quadrado.

Solução elegante, na prática é o que a organização do comício e a polícia usam.
Mas deve-se saber de antemão a metragem da praça e estimar a taxa de pessoas por
metro quadrado. O método é tão bom quanto melhor for a estimativa. Também é
melhor se a população estiver uniformemente distribuída.

Concluiu-se que é um bom método, mas que não é preciso. Isto é, a chance
do número estimado ser exatamente o número de presentes é baixa. Os métodos
anteriores são exatos, isto é, nos dão o número correto de presentes. Este método
também serve razoavelmente bem para o número de alunos na sala de aula. De fato,
nesta aula, o professor conseguiu o número com aproximação 80% correta (tinha feito
2.1. CONTANDO O NÚMERO DE PRESENTES EM UM EVENTO

isso antes da contagem da primeira solução). A questão que restou é se o erro de 20% é aceitável ou não. Isto depende do motivo pelo qual se quer contar os presentes.

Quarta solução

Para resolver o problema da precisão, outro estudante sugeriu o uso de roletas.

Efetivamente é esta a solução para contar torcedores no estádio ou presentes em um show de rock. Mas também foi considerado que a solução exige uma ou mais catracas, uma barreira para ninguém entrar sem passar pela roleta e etc, para se garantir a exatidão do resultado. No caso do comício não seria viável. No caso da sala de aula foi constatado que não havia roletas e portanto o método não se aplicava.

Quinta solução

Mais uma vez outro estudante apresentou uma boa alternativa: contar o número de filas de carteiras e, dado que todas tenham o mesmo número de estudantes, então bastaria uma simples multiplicação para a determinação do número correto.

De fato esta solução funciona perfeitamente bem em lugares como por exemplo o exército. As filas são rapidamente arrumadas com, digamos, 10 soldados em cada fila, sabendo-se o número de filas basta multiplicar por 10, eventualmente tendo-se que contar o número de pessoas em uma fila que não tenha completado 10.

Infelizmente as carteiras estavam bagunçadas na nossa sala e este cálculo não pode ser feito. Também ficaria estranho o professor colocar todos os alunos em filas. Foi também observado que o método fornece a solução exata para o problema.

Sexta solução

Nova sugestão de outro aluno: cada estudante no início de cada fila conta o número de alunos da sua fila, tomando o cuidado de contar a si próprio também. Depois soma-se todas as contagens de todos os primeiros de fila.

Solução muito boa. Na verdade é a versão em paralelo da primeira solução. Distribuindo-se a tarefa, cada primeiro de fila tem entre 10 e 15 alunos para contar em sua fila. Se a soma foi correta o número obtido ao final do processo é exato. No caso daquela aula os estudantes realizaram a operação em poucos segundos, mais algum tempo para as somas (isto demorou mais...). Mas o resultado foi exato.

A solução não exige muito conhecimento prévio, não exige equipamento adicional e é razoavelmente escalável, isto é, funciona para salas de tamanhos diferentes.

Sétima solução

Para finalizar, o professor apresentou a solução seguinte: todos os estudantes se levantam e se atribuem o número 1. Em seguida os alunos se organizam em pares. Em cada par, primeiro é somado o número de cada um dos dois, um deles guarda este número e permanece de pé, o outro deve se sentar. Os que ficaram em pé repetem
este processo até que só exista um único aluno em pé. Ele tem o número exato de estudantes na sala.

Como se divide a sala em pares, após a primeira rodada metade da sala deve ter o número 2 e a outra metade está sentada, considerando que a sala tem o número de alunos par. Se for impar um deles terá ainda o número 1. Após a segunda rodada um quarto dos alunos deverá ter o número 4 e três quartos estarão sentados, eventualmente um deles terá um número ímpar. É fácil perceber que o resultado sai em tempo proporcional ao logaritmo do número total de alunos, o que é bem rápido. De fato, para mil pessoas o processo termina em 10 passos e para um milhão de pessoas termina em 20 passos.

Parece um bom algoritmo, ele dá resultado exato, não exige conhecimento prévio, é escalável, isto é, funciona muito bem para um grande número de pessoas, mas exige organização dos presentes.

Infelizmente aquela turma não se organizou direito e o resultado veio com um erro de 40%. Mas após duas rodadas de treinamento, na terceira conseguimos obter o resultado correto.

2.2 Trocando os quatro pneus

Todo mundo sabe trocar pneus, embora não goste. O processo que um cidadão comum executa é muito simples: levanta o carro com o macaco, tira todos os quatro parafusos da roda com o pneu furado, tira a roda do eixo, coloca a roda com o pneu novo no eixo, em seguida aperta os quatro parafusos. Finalmente, baixa o carro e está pronto. O problema aqui é o de trocar pneus.

Nos anos 1980, Nelson Piquet, que se tornaria tricampeão mundial de fórmula 1, imaginou que poderia ser campeão do mundo se pudesse usar um composto de pneu mais mole e com isto ganhar preciosos segundos com relação aos seus concorrentes. O problema é que estes compostos mais moles se deterioram rapidamente exigindo a troca dos quatro pneus no meio da corrida. Piquet, após alguns cálculos, concluiu que se levasse menos de 8 segundos para trocar os quatro pneus, valeria a pena aplicar este método.

Obviamente a solução caseira não serve. O método descrito acima custa em geral 20 minutos por pneu, com um pouco de prática 10 minutos. Com muita prática 2 ou 3 minutos. Para trocar os quatro pneus, 8 a 12 minutos.

Daí a instância do problema: Como trocar os pneus em menos de 8 segundos?

Um dos grandes custos de tempo é ter que trocar o macaco para cada roda: usamos um macaco hidráulico, destes de loja de pneus, e levantamos o carro todo de uma só vez.

Mas, para cada roda, temos 4 parafusos, isto é, 16 no total, ou melhor, 32, pois tem que tirar e depois recolocar: usa-se uma aparafusadeira elétrica para amenizar o problema, mas ainda não é suficiente.

Se a roda tiver um único parafuso a economia de tempo é maior ainda. Mas ainda estamos na casa dos minutos, e o tempo total deve ser menor que 8 segundos. Desistimos do campeonato?
2.3 Conclusão

Mesmo para um problema simples existem diversas soluções. A escolha da melhor depende de vários fatores. Por exemplo, se a resposta deve ser exata ou não ou se os conhecimentos prévios necessários estão disponíveis, e assim por diante.

É importante notar que somente após uma série de considerações é possível escolher a melhor técnica e somente em seguida executar a tarefa.

Para algumas soluções a noção de paralelismo pode ajudar. Hoje em dia os computadores vêm com vários núcleos de processamento e sempre existe a chance de se tentar quebrar um problema em vários outros menores e deixar que vários processadores resolvam seus pedaços de solução e depois tentar juntar os resultados com mais alguma operação simples.

No caso da fórmula 1 isto funcionou, mas em geral não é verdade. Infelizmente existe o problema da dependência de dados. Por exemplo, o mecânico que vai colocar a roda nova só pode trabalhar depois que o outro tirou a roda velha. Em problemas com alto grau de dependência, paralelizar é complicado.

É importante perceber a diferença entre problema e instância de problema. Por exemplo, somar 2 mais 2 é uma instância do problema de somar dois números. Multiplicar uma matriz por outra é um problema que tem como instância, por exemplo, multiplicar uma dada matriz 3 por 3. Em geral, na ciência da computação, estamos interessados em resolver problemas, ou melhor dizendo, em estudar problemas, e não somente resolver instâncias dele.

\[\text{Não se estudarão algoritmos paralelos nesta disciplina.}\]
Capítulo 3

Sobre algoritmos e programas

Após o estudo do problema, análise das diversas possibilidades de solução e a escolha da melhor delas, cabe agora a tarefa de escrever um programa que implemente esta solução. Antes, contudo, é preciso saber a diferença entre um algoritmo em um programa. Isto será discutido neste capítulo.

3.1 O que é um algoritmo?

Um algoritmo é uma sequência extremamente precisa de instruções que, quando lida e executada por uma outra pessoa, produz o resultado esperado, isto é, a solução de um problema. Esta sequência de instruções é nada mais nada menos que um registro escrito da sequência de passos necessários que devem ser executados para manipular informações, ou dados, para se chegar na resposta do problema.

Isto serve por dois motivos: o primeiro é que através do registro se garante que não haverá necessidade de se redescobrir a solução quando muito tempo tiver passado e todos tiverem esquecido do problema; o outro motivo é que, as vezes, queremos que outra pessoa execute a solução, mas através de instruções precisas, de maneira que não haja erros durante o processo. Queremos um *algoritmo* para a solução do problema.

Uma receita de bolo de chocolate é um bom exemplo de um algoritmo (a lista de ingredientes e as quantidades foram omitidas, bem como a receita da cobertura):

Bata em uma batedeira a manteiga e o açúcar. Junte as gemas uma a uma até obter um creme homogêneo. Adicione o leite aos poucos. Desligue a batedeira e adicione a farinha de trigo, o chocolate em pó, o fermento e reserve. Bata as claras em neve e junte-as à massa de chocolate misturando delicadamente. Unte uma forma retangular pequena com manteiga e farinha, coloque a massa nela e leve para assar em forno médio preaquecido por aproximadamente 30 minutos. Desenforme o bolo ainda quente e reserve.

Este é um bom exemplo de algoritmo pois podemos extrair características bastante interessantes do texto. Em primeiro lugar, a pessoa que escreveu a receita não é
CAPÍTULO 3. SOBRE ALGORITMOS E PROGRAMAS

necessariamente a mesma pessoa que vai fazer o bolo. Logo, podemos estabelecer, sem prejuízo, que foi escrita por um mas será executada por outro.

Outras características interessantes que estão implícitas são as seguintes:

• as frases são instruções no modo imperativo: *bata isso, unte aquilo*. São ordens, não sugestões. Quem segue uma receita *obedece* quem a escreveu;

• as instruções estão na forma sequencial: apenas uma pessoa executa. Não existem ações simultâneas.

• existe uma ordem para se executar as instruções: *primeiro* bata a manteiga e o açúcar; *depois* junte as gemas, uma a uma, até acabar os ovos; *em seguida* adicione o leite.

• algumas instruções não são executadas imediatamente, é preciso entrar em um modo de repetição de um conjunto de outras instruções: enquanto houver ovos não usados, junte mais uma gema. Só pare quando tiver usado todos os ovos.

• algumas outras instruções não foram mencionadas, mas são obviamente necessárias que ocorram: é preciso separar as gemas das claras *antes* de começar a tarefa de se fazer o bolo, assim como é preciso *ainda antes* quebrar os ovos.

• algumas instruções, ou conjunto de instruções, podem ter a ordem invertida: pode-se fazer primeiro a massa e depois a cobertura, ou vice-versa. Mas nunca se pode colocar no forno a assadeira antes de se chegar ao término do preparo da massa.

Mesmo levando estas coisas em consideração, qualquer ser humano bem treinado em cozinha conseguiria fazer um bolo de chocolate razoável com as instruções acima, pois todas as receitas seguem o mesmo padrão. As convenções que estão implícitas no algoritmo são conhecidas de qualquer cozinheiro, pois seguem um formato padrão.

O formato padrão para algoritmos que vamos considerar é o seguinte:

• as instruções serão escritas uma em cada linha;

• as instruções serão executadas uma a uma, da primeira até a última linha, nesta ordem, a menos que o próprio algoritmo tenha instruções que alterem este comportamento;

• em cada linha, uma instrução faz somente uma coisa;

• tudo o que está implícito deverá ser explicitado.

A figura 3.1 ilustra a receita de bolo de chocolate escrita dentro deste formato padrão.
Algoritmo para fazer um bolo de chocolate.
início
    Providencie todos os ingredientes da receita.
    Providencie uma forma pequena.
    Ligue o forno em temperatura média.
    Coloque a manteiga na batedeira.
    Coloque o açúcar na batedeira.
    Ligue a batedeira.
    Enquanto um creme homogêneo não for obtido, junte mais uma gema.
    Adicione aos poucos o leite.
    Desligue a batedeira.
    Adicione a farinha de trigo.
    Adicione o chocolate em pó.
    Adicione o fermento.
    Reserve a massa obtida em um lugar temporário.
    Execute o algoritmo para obter as claras em neve.
    Junte as claras em neve à massa de chocolate que estava reservada.
    Misture esta massa delicadamente.
    Execute o algoritmo para untar a forma com manteiga e farinha.
    Coloque a massa na forma.
    Coloque a forma no forno.
    Espere 30 minutos.
    Tire a forma do forno.
    Desenforme o bolo ainda quente.
    Separe o bolo em um lugar temporário.
    Faça a cobertura segundo o algoritmo de fazer cobertura.
    Coloque a cobertura no bolo.
fim.

Figura 3.1: Algoritmo para fazer bolo de chocolate.

Infelizmente, nem todos conseguem fazer o bolo, pois existem instruções que somente os iniciados decifram:

- “adicione aos poucos”;
- “misturando delicadamente”;
- “quando o creme fica homogêneo?”…

No caso do computador a situação é pior ainda, pois trata-se de um circuito eletrônico, de uma máquina. Por este motivo, as instruções devem ser precisas e organizadas.

Um algoritmo feito para um computador executar deve tornar explícito todas as informações implícitas. Também deve evitar o uso de frases ambíguas ou imprecisas
e deve ser o mais detalhado possível. Também não pode ter frases de significado desconhecido.
Na próxima seção vamos desenvolver melhor este tema.

3.2 O que é um programa?

Um programa é a codificação em alguma linguagem formal que garanta que os passos do algoritmo sejam executados da maneira como se espera por quem executa as instruções.

Vamos imaginar, a título de ilustração, que é a primeira vez que a pessoa entra na cozinha em toda a sua vida e resolve fazer um bolo de chocolate seguindo o algoritmo 3.1

O algoritmo 3.1 foi escrito por um cozinheiro para ser executado por um outro cozinheiro, o que não é o caso, pois a pessoa é inexperiente em cozinha e não sabe o que significa “bater as claras em neve”. Significa que o novato vai ficar sem o bolo.

O novato precisaria de algo mais detalhado, isto é, de instruções meticulosas de como se obtém claras em neve. Poderia ser algo como ilustrado na figura 3.2.

Algoritmo para fazer claras em neve

início
Repita os quatro seguintes passos:
   Pegue um ovo.
   Quebre o ovo.
   Separe a clara da gema.
   Coloque somente a clara em um prato fundo.
Até que todos os ovos tenham sido utilizados, então pare.
Pegue um garfo.
Mergulhe a ponta do garfo no prato.
Repita os seguintes passos:
   Bata a clara com o garfo por um tempo.
   Levante o garfo.
Observe se a espuma produzida fica presa no garfo
Até que a espuma fique presa no garfo, então pare.
Neste ponto suas claras em neve estão prontas.
fim.

Figura 3.2: Algoritmo para fazer claras em neve.

Já temos algo mais detalhado, mas nosso inexperiente cozinheiro pode ainda ter problemas: como se separa a clara da gema? Este tipo de situação parece não ter fim. Qual é o limite do processo de detalhamento da solução?

O problema é que o cozinheiro que escreveu a receita original não sabia o nível de instrução de quem ia efetivamente fazer o bolo. Para isto, é preciso que se estabeleça o nível mínimo de conhecimento para quem vai executar, assim quem escreve sabe até onde deve ir o nível de detalhamento de sua receita.
Um programa, neste sentido, é um algoritmo escrito de forma tão detalhada quanto for necessário para quem executa as instruções. O algoritmo pode ser mais genérico, o programa não.

Como estamos pensando em deixar que o computador execute um algoritmo, precisamos escrever um programa em uma linguagem na qual o computador possa entender as instruções para posteriormente poder executá-las com sucesso.

Qual é, afinal, o conjunto de instruções que o computador conhece? Para responder a esta pergunta precisamos conhecer melhor como funciona um computador, para, em seguida, continuarmos no estudo de algoritmos.
3.3 Exercícios

1. Escreva algoritmos como os que foram escritos neste capítulo para cada uma das soluções do problema discutido na seção 2.1.

2. Escreva um algoritmo para o problema da troca de um único pneu de um carro.

3. Escreva um algoritmo para o problema de trocar um pneu de uma bicicleta.
Capítulo 4

O modelo do computador

Esta seção tem dois objetivos, o primeiro é mostrar como é o funcionamento dos computadores modernos, isto é, no nível de máquina. A segunda é que o aluno perceba, desde o início do seu aprendizado, as limitações a que está sujeito quando programa, e quais são todas as instruções que o computador conhece.

Ao final da leitura, o estudante deve compreender que, por mais sofisticada que seja a linguagem de programação utilizada, a computação de verdade ocorre como será mostrado aqui.

4.1 Histórico

Um computador (hardware) é um conjunto de circuitos eletrônicos que manipulam sinais elétricos e que são capazes de transformar sinais de entrada em sinais de saída. Os sinais elétricos podem ser representados, basicamente, pelos números zero e um. Existem várias maneiras de se fazer isto, mas não entraremos em detalhes neste texto.

O importante a destacar é que uma computação é uma manipulação de dados residentes em memória através de alterações de sinais elétricos realizadas por circuitos integrados implementados normalmente em placas de silício.

Quando os computadores foram criados, na década de 1940, a programação deles era feita de maneira muito precária. Era necessário configurar uma situação dos circuitos para manipular os sinais elétricos da maneira desejada para cada programa particular. Para se executar outro programa era necessário alterar os circuitos, assim se reprogramando o computador para manipular os dados de outra maneira.

Um computador era algo raro naqueles tempos, e devia rodar vários programas diferentes, o que resultava em imenso trabalho.

A memória do computador, naqueles tempos, era exclusividade dos dados que seriam manipulados. O programa era feito nos circuitos eletrônicos.

John von Neumann publicou um modelo bastante simples que sintetizava a ideia de muitos pesquisadores da época, incluindo Alan Turing, no qual tanto o programa

---

1O texto que segue foi adaptado de outro escrito pelo prof. Renato Carmo para a disciplina CI-208 - Programação de Computadores ministrada para diversos cursos na UFPR.
CAPÍTULO 4. O MODELO DO COMPUTADOR

quanto os dados poderiam ficar simultaneamente em memória, desde que a parte que
ficaria programada nos circuitos pudesse interpretar o que era dado e o que era o
programa e realizar os cálculos, isto é, manipular os dados.

Isto foi possível pela implementação em hardware de um limitado conjunto de
instruções que são usados pelo programa que está em memória. Isto revolucionou a
arte da programação. Os computadores modernos ainda funcionam assim.

Nesta seção pretende-se mostrar através de um exemplo os princípios deste modelo.

4.2 Princípios do modelo

Conforme explicado, a ideia era que os dados e o programa poderiam ser carregados
em memória ao mesmo tempo. Um elemento adicional denominado ciclo de execução
de instruções controla a execução do programa.

A ideia é implementar em hardware um pequeno conjunto de instruções que não
mudam e programar o computador para realizar operações complexas a partir da
execução de várias instruções básicas da máquina.

Cada fabricante define o seu conjunto de instruções básicas, mas o importante a
observar é que, uma vez implementadas, este conjunto define tudo o que o computador
sabe fazer. É isto que queremos saber.

Neste capítulo vamos usar como exemplo um computador fabricado pela Big Com-
puter Company (BCC).

4.2.1 Endereços versus conteúdos

O computador da BCC implementa o modelo conhecido como Von Neumann, logo,
sua memória contém os dados e o programa.

A memória do computador em um dado instante do tempo é uma configuração de
sinais elétricos que podem ser vistos pelo ser humano como uma sequência absurda
de zeros e uns (chamados de bits)²

O ser humano costuma não gostar muito desta forma de visualização, então convencionou algumas maneiras de enxergar números inteiros que representam os bits. Não vamos apresentar neste texto as diversas maneiras de conversão de números, o leitor interessado pode estudar sobre representação binária na literatura.

Vamos imaginar que a memória do computador é uma tabela contendo índices
(endereços) com conteúdos (dados). A título de exemplo, vamos considerar uma
“fotografia” da memória de um computador da BCC em um certo momento, fotografia
esta apresentada na figura 4.1.

Para melhor entendimento, é importante que o leitor tenha em mente a diferença
entre endereço e conteúdo do endereço. Para facilitar a compreensão, vamos adotar

²Quem assistiu ao filme Matrix pode imaginar a complicação.
 uma notação. Seja \( p \) um endereço. Denotamos por \([p]\) o conteúdo do endereço \( p \).

Vejamos alguns exemplos com base na figura 4.1:

\[
\begin{align*}
[0] &= 1 \\
[[0]] &= [1] = 54 \\
[[[0]]] &= [[1]] = [54] = 235 \\
[0] + 1 &= 1 + 1 = 2 \\
[0 + 1] &= [1] = 54 \\
[[0] + 1] &= [1 + 1] = [2] = 2 \\
[[0] + [1]] &= [1 + 54] = [55] = 35
\end{align*}
\]

### 4.2.2 O repertório de instruções

Conforme mencionado, este modelo pressupõe que o computador que está em uso possui um conjunto limitado de instruções programado em hardware.

Cada equipamento tem o seu repertório de instruções. O repertório do computador da BCC foi definido após longas discussões da equipe técnica da empresa e tem um conjunto extremamente limitado de instruções, na verdade somente nove.

Esta definição levou em conta a capacidade financeira da empresa que não tinha muita verba para gravar em circuitos integrados um conjunto mais rico de instruções.
As nove instruções do computador da BCC são apresentadas na figura 4.2.

<table>
<thead>
<tr>
<th>Código</th>
<th>Mnemônico</th>
<th>Descrição</th>
<th>Notação</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>load</td>
<td>escreva em ([p + 1]) o valor do número em ([p + 2]) e some 3 em p</td>
<td>([p + 1] \leftarrow [p + 2]).</td>
</tr>
<tr>
<td>2</td>
<td>add</td>
<td>escreva em ([p + 1]) o valor da soma dos números em ([p + 2]) e ([p + 3]) e some 4 em p</td>
<td>([p + 1] \leftarrow ([p + 2]) + ([p + 3]).</td>
</tr>
<tr>
<td>3</td>
<td>sub</td>
<td>escreva em ([p + 1]) o valor da subtração do número em ([p + 2]) pelo número em ([p + 3]) e some 4 em p</td>
<td>([p + 1] \leftarrow ([p + 2]) - ([p + 3]).</td>
</tr>
<tr>
<td>4</td>
<td>mult</td>
<td>escreva em ([p + 1]) o valor do produto dos números em ([p + 2]) e ([p + 3]) e some 4 em p</td>
<td>([p + 1] \leftarrow ([p + 2]) \times ([p + 3]).</td>
</tr>
<tr>
<td>5</td>
<td>div</td>
<td>escreva em ([p + 1]) o valor da divisão do número em ([p + 2]) pelo número em ([p + 3]) e some 4 em p</td>
<td>([p + 1] \leftarrow \frac{[p + 2]}{[p + 3]}).</td>
</tr>
<tr>
<td>6</td>
<td>sqrt</td>
<td>escreva em ([p + 1]) o valor da raiz quadrada de ([p + 2]) e some 3 em p</td>
<td>([p + 1] \leftarrow \sqrt{[p + 2]}).</td>
</tr>
<tr>
<td>7</td>
<td>read</td>
<td>leia um número do teclado, escreva-o em ([p + 1]) e some 2 em p</td>
<td>([p + 1] \leftarrow ____.</td>
</tr>
<tr>
<td>8</td>
<td>write</td>
<td>escreva ([p + 1]) na tala e some 2 em p</td>
<td>[\square \leftarrow [p + 1]].</td>
</tr>
<tr>
<td>9</td>
<td>stop</td>
<td>pare</td>
<td>●</td>
</tr>
</tbody>
</table>

Figura 4.2: O repertório de instruções do computador da BCC.

---

3 Os concorrentes comerciais famosos da BCC implementam algumas poucas centenas de instruções, e ainda nenhuma delas é a de bater claras em neve.
4.2.3 O ciclo de execução de instruções

O ciclo de execução de instruções define o comportamento do computador. Funciona assim (no computador da BCC):

1. comece com \( p = 0 \);

2. interprete \([p]\) de acordo com a tabela de instruções e pare somente quando a instrução for uma ordem de parar (instrução 9, \textit{stop}).

Devemos lembrar que este comportamento também está implementado nos circuitos eletrônicos do computador da BCC.

4.2.4 Exemplo de execução de um programa

A grande surpresa por trás deste modelo é que, mesmo que o leitor ainda não comprenda, o que existe na verdade “disfarçado” na fotografia da memória da figura [4.1] é um programa que pode ser executado pelo computador, desde que todo o processo siga as instruções descritas na seção anterior.

Vamos tentar acompanhar passo a passo como é o funcionamento deste esquema. Para isto, o leitor talvez queira ir marcando, a lápis, as alterações que serão feitas a seguir em uma cópia da “fotografia da memória” acima. É recomendado neste momento se ter uma versão impressa daquela página.

Notem que, no momento, não é necessário sabermos qual o programa implementado, afinal de contas, o computador jamais saberá... Ele executa cegamente as instruções. Nós saberemos logo à frente, mas, agora, para entendermos como é o funcionamento deste modelo, vamos nos imaginar fazendo o papel do computador.

1. O programa começa com \( p = 0 \)

2. Em seguida, é preciso interpretar \([p]\), isto é \([0] = 1\). A instrução de código “1” é \textit{load}, cujo comportamento é, segundo a tabela de instruções “escreva em \([2]\) o valor do número em \([3]\) e some 3 em \(p\)”. Ora, \([1] = 54\) e \([2] = 2\). Logo, o valor 2 é colocado como sendo o conteúdo da posição 54. Havia nesta posição o valor 235. Após a execução da instrução, existe um 2 neste lugar. O valor 235 não existe mais. Ao final foi somado 3 no valor de \(p\), isto é, agora \(p = 3\).

3. Como \( p = 3 \) devemos interpretar \([3] = 1\). Logo, a instrução é novamente \textit{load}.


4. Como \( p = 6 \) devemos interpretar \([6] = 7\). Logo, a instrução para ser executada agora é \textit{read}, isto é, esperar o usuário digitar algo no teclado e carregar este valor em \([p + 1] = [7] = 46\). Supondo que o usuário digitou o valor 5, este agora substitui o antigo valor, que era 33. Ao final, o valor de \(p\) foi atualizado de 6 para 8.
5. Como \( p = 8 \) devemos interpretar \([8] = 4\). Logo, a instrução a ser executada é “mult”. Isto faz com que o computador faça a multiplicação do valor em \([10] = 46\) pelo mesmo valor em \([11] = 46\). O valor em \([46]\) é 5 (aquele número que o usuário tinha digitado no teclado). O resultado da multiplicação, \(5 \times 5 = 25\), é carregado na posição de memória \([9] = 47\). O valor ali que era 2 agora passa a ser 25. Ao final, ao valor de \( p \) foi somado 4, logo neste momento \( p = 12\).

É importante salientar que este é um processo repetitivo que só terminará quando a instrução “stop” for a da vez. O leitor é encorajado a acompanhar a execução passo a passo até o final para entender como é exatamente o comportamento dos computadores quando executam programas. Isto é, fica como exercício ao leitor! Destacamos que os circuitos implementados cuidam da alteração do estado elétrico dos circuitos da memória.

4.3 Humanos versus computadores

Nós seres humanos não nos sentimos muito à vontade com este tipo de trabalho repetitivo. Temos a tendência a identificar “meta-regras” e executar a operação com base em um comportamento de mais alto nível. Em suma, nós aprendemos algo neste processo, coisa que o computador só faz em filmes de ficção científica.

A primeira coisa que nos perguntamos é: por qual motivo ora se soma um valor em \( p \), ora outro? Isto é, quando executamos a operação “load”, o valor somado em \( p \) foi 3. Depois, quando executada a operação “read” o valor somado foi 2. Em seguida, para a instrução “mult” o valor somado foi 4.

O estudante atento, notadamente aquele que foi até o final na execução do ciclo de operações deixado como exercício, talvez tenha percebido uma sutileza por trás deste modelo.

De fato, quando se executa a instrução \([p]\), o conteúdo de \([p+1]\) sempre é o endereço de destino dos dados que são resultado da operação em execução. Os endereços subsequentes apontam para os operandos da operação que está programada para acontecer.

Assim:

- Se for uma multiplicação, subtração ou soma, precisamos de dois operandos e do endereço destino, por isto se soma 4 em \( p \);
- Se for um “load” precisamos de um único operando, assim como para a raiz quadrada, por isto se soma 3 em \( p \);
- Se for uma leitura do teclado ou uma escrita na tela do computador, então um único argumento é necessário, por isto se soma 2 em \( p \).

Uma outra observação importante é que, por questões de hardware, o computador precisa entender a memória como esta espécie de “tripa”. O ser humano, ao contrário,
uma vez que já identificou pequenos blocos relacionados às instruções, pode tentar entender esta mesma memória em outro formato, isto é, separando cada um destes pequenos blocos em uma única linha.

Observamos que somente os números de 1 a 9 podem ser interpretados como códigos de alguma instrução, pois são os únicos códigos da tabela de instruções da BCC.

A separação dos pequenos blocos resulta em uma visualização em que os dados são separados das instruções numa mesma linha, cada linha agora representa toda a instrução com os dados. Isto pode ser visto na figura 4.3. Importante notar que é a mesma informação da figura 4.1, só que em outro formato visual.

<table>
<thead>
<tr>
<th>Endereço</th>
<th>Instrução</th>
<th>Operando</th>
<th>Operando</th>
<th>Operando</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>54</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>50</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>46</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>47</td>
<td>46</td>
<td>46</td>
</tr>
<tr>
<td>12</td>
<td>7</td>
<td>48</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>4</td>
<td>49</td>
<td>50</td>
<td>48</td>
</tr>
<tr>
<td>18</td>
<td>3</td>
<td>51</td>
<td>47</td>
<td>49</td>
</tr>
<tr>
<td>22</td>
<td>6</td>
<td>52</td>
<td>51</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>3</td>
<td>53</td>
<td>46</td>
<td>52</td>
</tr>
<tr>
<td>29</td>
<td>5</td>
<td>55</td>
<td>53</td>
<td>54</td>
</tr>
<tr>
<td>33</td>
<td>8</td>
<td>55</td>
<td></td>
<td></td>
</tr>
<tr>
<td>35</td>
<td>2</td>
<td>56</td>
<td>46</td>
<td>52</td>
</tr>
<tr>
<td>39</td>
<td>5</td>
<td>57</td>
<td>56</td>
<td>54</td>
</tr>
<tr>
<td>43</td>
<td>8</td>
<td>57</td>
<td></td>
<td></td>
</tr>
<tr>
<td>45</td>
<td>9</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figura 4.3: Separando as instruções.

4.3.1 Abstração dos endereços

Continuando nossa exploração de aspectos percebidos pelo ser humano a partir da maneira como o computador trabalha, agora é possível percebermos mais duas coisas importantes:

1. O computador não mudou seu modo de operar, ele continua executando as instruções na memória conforme foi apresentado na seção 4.2.4.

2. A visualização na forma apresentada na figura 4.3 é somente uma maneira mais simples para o ser humano perceber o que o computador está fazendo.

Esta segunda observação é muito importante. Ela nos permite aumentar o grau de “facilidade visual” ou, dizendo de outra maneira, o grau de notação sobre o modelo
Von Neumann, o que vai nos permitir a compreensão do processo de uma maneira cada vez mais “humana”.

De fato, a partir da figura 4.3, podemos perceber que o endereço em que ocorre a instrução é irrelevante para nós, humanos. Podemos perfeitamente compreender o processo de computação se eliminarmos a coluna do “Endereço” na figura 4.3.

A figura 4.4 ilustra como ficaria o programa em memória visto de outra maneira, agora não apenas em formato de blocos mas também sem a coluna dos endereços. Vamos perceber que, apesar de muito parecido com a figura anterior, o grau de “poluição visual” é bem menor.

<table>
<thead>
<tr>
<th>Instrução</th>
<th>Operando</th>
<th>Operando</th>
<th>Operando</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>54</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>50</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>46</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>47</td>
<td>46</td>
<td>46</td>
</tr>
<tr>
<td>7</td>
<td>48</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>49</td>
<td>50</td>
<td>48</td>
</tr>
<tr>
<td>3</td>
<td>51</td>
<td>47</td>
<td>49</td>
</tr>
<tr>
<td>6</td>
<td>52</td>
<td>51</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>53</td>
<td>46</td>
<td>52</td>
</tr>
<tr>
<td>5</td>
<td>55</td>
<td>53</td>
<td>54</td>
</tr>
<tr>
<td>8</td>
<td>55</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>56</td>
<td>46</td>
<td>52</td>
</tr>
<tr>
<td>5</td>
<td>57</td>
<td>56</td>
<td>54</td>
</tr>
<tr>
<td>8</td>
<td>57</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figura 4.4: Abstração dos endereços.

Vale a pena reforçar: o computador não mudou, ele continua operando sobre os circuitos eletrônicos, o que estamos fazendo é uma tentativa, um pouco mais humana, de enxergar isto.

4.3.2 Abstração dos códigos das instruções

Continuando neste processo de “fazer a coisa do modo mais confortável para o ser humano”, afirmamos que é possível aumentar ainda mais o grau de notação.

Para a etapa seguinte, vamos observar que, embora os computadores manipulem números (em particular, números binários) de maneira muito eficiente e rápida, o mesmo não ocorre para os humanos, que têm a tendência a preferirem *nomes*.

De fato, basta observar que nós usamos no nosso dia a dia nomes tais como Marcos, José ou Maria e não números tais como o RG do Marcos, o CPF do José ou o PIS da Maria.

Ora, já que é assim, qual o motivo de usarmos os números 1 – 9 para as instruções se podemos perfeitamente usar o mnemônico associado ao código?
4.3. HUMANOS VERSUS COMPUTADORES

Desta forma, vamos modificar ainda mais uma vez nossa visualização da memória, desta vez escrevendo os nomes dos mnemônicos no lugar dos números. Assim, na coluna **Instrução**, substituiremos o número 1 por `load`, 2 por `add`, 3 por `sub` e assim por diante, conforme a figura 4.2.

O programa da figura 4.4 pode ser visualizado novamente de outra forma, tal como apresentado na figura 4.5. Notem que o grau de compreensão do código, embora ainda não esteja em uma forma totalmente amigável, já é bastante melhor do que aquela primeira apresentação da “fotografia da memória”.

De fato, agora é possível compreender o significado das linhas, em que existe um destaque para a operação (o mnemônico), a segunda coluna é o endereço de destino da operação e as outras colunas são os operandos.

<table>
<thead>
<tr>
<th>Instrução</th>
<th>Operando</th>
<th>Operando</th>
<th>Operando</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>load</code></td>
<td>54</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td><code>load</code></td>
<td>50</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td><code>read</code></td>
<td>46</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>mult</code></td>
<td>47</td>
<td>46</td>
<td>46</td>
</tr>
<tr>
<td><code>read</code></td>
<td>48</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>mult</code></td>
<td>49</td>
<td>50</td>
<td>48</td>
</tr>
<tr>
<td><code>sub</code></td>
<td>51</td>
<td>47</td>
<td>49</td>
</tr>
<tr>
<td><code>sqrt</code></td>
<td>52</td>
<td>51</td>
<td></td>
</tr>
<tr>
<td><code>sub</code></td>
<td>53</td>
<td>46</td>
<td>52</td>
</tr>
<tr>
<td><code>div</code></td>
<td>55</td>
<td>53</td>
<td>54</td>
</tr>
<tr>
<td><code>write</code></td>
<td>55</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>add</code></td>
<td>56</td>
<td>46</td>
<td>52</td>
</tr>
<tr>
<td><code>div</code></td>
<td>57</td>
<td>56</td>
<td>54</td>
</tr>
<tr>
<td><code>write</code></td>
<td>57</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>stop</code></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figura 4.5: Programa reescrito com Mnemônicos.

O que falta ainda a ser melhorado? Nós humanos usamos desde a mais tenra idade outro tipo de notação para representarmos operações. Este é o próximo passo.

4.3.3 Abstração do repertório de instruções

Nesta etapa, observaremos que as instruções executadas pelo computador nada mais são do que manipulação dos dados em memória. Os cálculos são feitos sobre os dados e o resultado é colocado em alguma posição de memória.

Podemos melhorar o grau de abstração considerando a notação apresentada na figura 4.6. Isto é, vamos usar as tradicionais letras finais do alfabeto para ajudar na melhoria da facilidade visual. Assim poderemos reescrever o programa mais uma vez e ele ficará como apresentado na figura 4.7.
4.3.4 Abstração dos endereços de memória (variáveis)

Na verdade, assim como já observamos que o código da instrução não nos interessa, vamos perceber que o mesmo é verdade com relação aos endereços de memória. Então, convencionaremos que os números podem ser trocados por nomes. Fazemos isto na versão seguinte do mesmo programa.

Convém notar que o ser humano gosta de dar nomes apropriados para as coisas. Quem quiser ter uma ideia do estrago quando não damos bons nomes para as coisas, pode acompanhar dois diálogos humorísticos, um deles, o original, da dupla Abbot & Costello sobre um jogo de beisebol, a outra de um ex-presidente dos Estados Unidos falando sobre o então novo presidente da China. Ambos os vídeos estão disponíveis no YouTube (infelizmente em inglês):
4.4. ABSTRAÇÃO DAS INSTRUÇÕES (LINGUAGEM)

Assim, é importante que os nomes que usarmos tenham alguma relação com o significado que eles estão desempenhando no programa.

Então chega o momento de nós sabermos que o programa que está em memória recebe como entrada dois valores $b$ e $c$ e escreve como saída as raízes da equação $x^2 - bx + c = 0$. Os cálculos usam o método de Bhaskara. A figura 4.8 mostra a convenção para a substituição dos endereços por nomes.

<table>
<thead>
<tr>
<th>Endereço</th>
<th>Nome</th>
</tr>
</thead>
<tbody>
<tr>
<td>54</td>
<td>dois</td>
</tr>
<tr>
<td>50</td>
<td>quatro</td>
</tr>
<tr>
<td>46</td>
<td>B</td>
</tr>
<tr>
<td>47</td>
<td>quadradoB</td>
</tr>
<tr>
<td>48</td>
<td>C</td>
</tr>
<tr>
<td>49</td>
<td>quadruploC</td>
</tr>
<tr>
<td>51</td>
<td>discriminante</td>
</tr>
<tr>
<td>52</td>
<td>raizDiscriminante</td>
</tr>
<tr>
<td>53</td>
<td>dobroMenorRaiz</td>
</tr>
<tr>
<td>55</td>
<td>menorRaiz</td>
</tr>
<tr>
<td>56</td>
<td>dobroMaiorRaiz</td>
</tr>
<tr>
<td>57</td>
<td>maiorRaiz</td>
</tr>
</tbody>
</table>

Figura 4.8: Dando nomes para os endereços.

Agora, usaremos esta convenção de troca de endereços por nomes para podermos reescrever o programa ainda mais uma vez, obtendo a versão da figura 4.9.

Esta versão define o último grau de abstração simples. A partir deste ponto a notação deixa de ser só uma abreviatura das instruções e a tradução deixa de ser direta.

Apesar do fato de ser o último nível de notação simples, ainda é possível melhorarmos o grau de facilidade visual, mas desta vez passamos para a notação ou linguagem de “alto nível”, que vai exigir a introdução dos chamados compiladores.

4.4 Abstração das instruções (linguagem)

Apesar de todas as notações e convenções que foram feitas no programa, até se chegar na versão mostrada na figura 4.9, de certa maneira, o programa ainda está em um formato muito parecido com o do programa original.
Para que seja possível aumentar o nível de notação ainda mais é preciso contar com a ajuda de programas tradutores, ou como eles são mais conhecidos, os compiladores. Estes programas conseguem receber como entrada um texto escrito em um formato adequado e gerar como saída um programa no formato da máquina. Isto é possível somente se os programas forem escritos em um formato que respeite regras extremamente rígidas, pois senão a tarefa não seria possível.

As linguagens de alto nível são definidas a partir de uma gramática extremamente mais rígida que a do português, por exemplo. Estas gramáticas, conhecidas como gramáticas livre de contexto, têm como uma de suas principais características que elas não permitem escrever programas ambíguos. O português permite.

Sem entrarmos muito em detalhes desta gramática, a título de exemplo mostraremos versões em mais alto nível do programa da figura 4.9. Estas versões são apresentadas na figura 4.10.

<table>
<thead>
<tr>
<th>Variável</th>
<th>Expressão</th>
</tr>
</thead>
<tbody>
<tr>
<td>read B</td>
<td>read B</td>
</tr>
<tr>
<td>read C</td>
<td>read C</td>
</tr>
<tr>
<td>discriminante ← $B^2 - 4 \times C$</td>
<td>raizDiscriminante ← $\sqrt{discriminante}$</td>
</tr>
<tr>
<td>menorRaiz ← $\frac{B - \text{raizDiscriminante}}{2}$</td>
<td>menorRaiz ← $\frac{dobroMenorRaiz}{dois}$</td>
</tr>
<tr>
<td>maiorRaiz ← $\frac{B + \text{raizDiscriminante}}{2}$</td>
<td>maiorRaiz ← $\frac{dobroMaiorRaiz}{dois}$</td>
</tr>
<tr>
<td>write menorRaiz</td>
<td>write maiorRaiz</td>
</tr>
</tbody>
</table>

Figura 4.9: Programa reescrito com nomes para variáveis.

Figura 4.10: Duas outras versões do programa.
rigidez, infelizmente. A disciplina Construção de Compiladores, no sexto período do
curso, é exclusiva para o estudo profundo dos motivos desta dificuldade, tanto de se
verificar se o programa está correto do ponto de vista gramatical, quanto do ponto de
vista de se traduzir o programa para linguagem de máquina.

No momento, vamos nos limitar a apresentar na figura 4.11 uma versão do mesmo
programa escrito em Pascal. Após compilação, o resultado é um programa que pode
ser executado pelo computador.

Em suma, o compilador Pascal é um programa que, entre outras coisas, consegue
transformar o código de alto nível mostrado na figura 4.11 e gerar um código que o
computador possa executar tal como mostrado na primeira figura.

```pascal
program bhaskara;
var b, c, raizdiscriminante: real;
begin
  read (b);
  read (c);
  raizdiscriminante:= sqrt(b*b - 4*c);
  write ((b - raizdiscriminante)/2);
  write ((b + raizdiscriminante)/2);
end.
```

Figura 4.11: Versão do programa escrito em Pascal.

### 4.5 Conclusão

Nesta parte do texto procuramos mostrar que qualquer linguagem de programação
de alto nível (tal como Pascal, C ou JAVA) é meramente uma notação convencionada
visando facilitar a vida do ser humano que programar o computador. Esta notação
trata de como um texto se traduz em um programa executável em um determinado
sistema operacional (que usa um determinado conjunto reduzido de instruções).

Um programa que traduz um texto que emprega uma certa notação convencionada
em um programa executável é chamado de “compilador”.

Assim, a arte de se programar um computador em alto nível é, basicamente,
conhecer e dominar uma notação através da qual textos (ou programas fonte) são
traduzidos em programas executáveis.

Programar, independentemente da linguagem utilizada, significa concatenar as
instruções disponíveis dentro de um repertório a fim de transformar dados de entrada
em dados de saída para resolver um problema.

Nas linguagens de alto nível, as instruções complexas são traduzidas em uma
sequência de operações elementares do repertório básico da máquina. Por isto os-
programas fonte, quando compilados, geram executáveis que são dependentes do sistema
operacional e do hardware da máquina onde o programa executa.
A partir destas ideias, partindo do princípio que se tem um algoritmo que resolve um problema, o que é preciso saber para se programar um computador?

- Ter à disposição um editor de textos, para codificar o algoritmo na forma de programa fonte;

- Ter à disposição um compilador para a linguagem escolhida (no nosso caso, o Free Pascal), para transformar automaticamente um programa fonte em um programa executável.

No restante deste curso, vamos nos preocupar com a arte de se construir algoritmos, que é a parte difícil, tendo em mente que o estudante deverá ser capaz de saber transformar este algoritmo em forma de programa fonte de maneira que este possa ser compilado e finalmente executado em um computador. Esta última parte é mais simples, embora calouros tenham uma certa dificuldade. Neste livro daremos mais ênfase na construção de algoritmos, mas evidentemente apresentaremos também a arte de se escrever programas em Pascal.
4.6 Exercícios

1. Para perceber como o ambiente do computador é limitado em função do reduzido número de instruções disponíveis em baixo nível, você pode tentar jogar este jogo (http://armorgames.com/play/6061/light-bot-20). Nele, existe um boneco que tem que cumprir um percurso com o objetivo de apagar todas as células azuis do terreno quadriculado usando poucos comandos e com pouca “memória” disponível. Você pode fazer o uso de duas funções que auxiliam nas tarefas repetitivas. Divirta-se!

2. Modifique a “fotografia da memória” apresentada para que o computador resolva a equação $ax^2 + bx + c = 0$ pelo método de Bhaskara. A diferença do que foi apresentado é o coeficiente $a$ do termo $x^2$ e o sinal de $b$.

3. Leia os seguintes textos da wikipedia:

   (a) http://pt.wikipedia.org/wiki/Arquitetura_de_von_Neumann, sobre a arquitetura de von Neumann;

   (b) http://pt.wikipedia.org/wiki/Von_Neumann, sobre a vida de von Neumann, em especial a parte sobre computação.

   (c) https://pt.wikipedia.org/wiki/Alan_Turing, sobre a vida de Alan Turing, em especial a parte sobre computação.
Capítulo 5

Conceitos elementares

Agora que sabemos os princípios de algoritmos e as limitações da máquina, é preciso introduzir conceitos elementares de linguagens de programação, sem os quais não é possível seguir adiante.

Neste capítulo apresentaremos os conceitos elementares presentes em qualquer linguagem de programação e nos próximos capítulos estes conceitos serão utilizados na construção de soluções para problemas cada vez mais sofisticados.

5.1 Algoritmos e linguagens de programação

Conforme vimos, os algoritmos devem ser escritos em um nível de detalhamento suficiente para que o compilador consiga fazer a tradução do código para linguagem de máquina.

O compilador precisa receber um texto formatado em uma linguagem simples, não ambígua e precisa. Para isto as linguagens de programação seguem uma gramática rígida, se comparada com a da língua portuguesa. Também segue um vocabulário limitado, constituído de alguns poucos elementos.

Lembramos que os compiladores são programas que traduzem uma linguagem de programação para a linguagem de máquina, parecida com a que foi apresentada no capítulo 4.

Embora as linguagens sejam classificadas como sendo de alto nível, é preciso esclarecer que a gramática delas permite escrever textos de uma maneira bastante limitada, mas até o momento é o melhor que pode ser feito, pois ainda não é possível escrever programas em linguagens naturais tais como português ou inglês. É um ponto intermediário entre a capacidade de redação do ser humano e a capacidade de compreensão do computador.

A base das linguagens de programação, em geral, é constituída por:

- a noção de fluxo de execução de um programa;
- os comandos da linguagem que manipulam os dados em memória e a interação com o usuário (attribution, entrada e saída de dados);
as expressões da linguagem que permitem a realização de cálculos aritméticos e lógicos;

• os comandos da linguagem que modificam o fluxo de execução de um programa.

Neste capítulo usaremos as regras do compilador Free Pascal e para isto o leitor deve ter em mãos algum guia de referência desta linguagem, por exemplo, o mini guia de referência que está disponível no site oficial da disciplina CI055 e outras obras da literatura já citadas na introdução, nos quais as explicações são detalhadas.

Adotamos a linguagem Pascal, pois acreditamos que esta é a melhor linguagem de programação para iniciantes. Pascal é uma linguagem que tem os requisitos que achamos importantes: é uma linguagem fortemente tipada, com sintaxe relativamente simples, possui vários compiladores para diversos sistemas operacionais e, finalmente, este compilador apresenta mensagens de erro relativamente simples de serem compreendidas.

5.2 Fluxo de execução de um programa

Um programa em Pascal contém obrigatoriamente um cabeçalho e um programa principal. A figura 5.1 ajuda a entender estes conceitos.

```
program exemplo1;

begin
    comando 1;
    comando 2;
    comando 3;
end.
```

Figura 5.1: Estrutura de um programa em Pascal.

O cabeçalho deste programa é a linha contendo a palavra reservada program seguido de um identificador que é o nome deste programa, no caso exemplo1, terminado por um símbolo de ponto-e-vírgula (;).

O programa principal é tudo o que está entre as palavras reservadas begin e end, terminada em um ponto final (.). Aqui mostramos três “comandos” fictícios, para fins desta apresentação. Queremos dizer que o programa vai executar três comandos quaisquer. Observe que todos os comandos em Pascal são terminados com um ponto-e-vírgula (;).

O importante é que o fluxo de execução do programa obedece a seguinte ordem: primeiro executa o comando 1, depois de terminado, executa o comando 2, depois de

terminado executa o comando 3. Quando este termina, o programa encontra o end. e encerra o programa.

Na medida em que os conceitos fundamentais forem sendo apresentados iremos detalhar melhor os elementos do cabeçalho.

5.3 Comandos de entrada e saída

Toda linguagem de programação possui comandos que permitem a um usuário fornecer dados ao computador pelo teclado e também receber retornos do computador através do monitor de vídeo.

Conforme vimos, os dados em memória são manipulados pelo uso de variáveis, que são nada mais nada menos do que endereços físicos de memória que podem armazenar conteúdos de interesse do programador. Já explicamos também que a localização exata destes endereços não nos interessa, uma vez que nós seres humanos preferimos abstrair esta informação e usar nomes (palavras, textos) que abstratamente nos permitem visualizar como os dados estão em memória.

Vamos fazer um programa bem simples para entender esta nova informação e também apresentar os comandos de entrada (read), ou de leitura dos dados pelo teclado e saída (write), ou de impressão de informações na tela. Este programa é apresentado na figura 5.2.

```pascal
program entrada_saida;
var a: integer;
begin
  read (a);
  write (a);
end.
```

Figura 5.2: Entrada e saída em Pascal.

Este programa tem somente dois comandos, o primeiro de leitura de uma informação do teclado (algum usuário tem que digitar algo e apertar ENTER). O segundo comando implica na impressão de um valor na tela do computador.

A leitura de dados implica no uso de uma variável, no caso do exemplo ela é a. A leitura implementa a instrução load visto no capítulo anterior. O comando utilizado foi read(a);. A execução deste comando implica que seja lá qual for a informação que o usuário digitar no teclado será armazenada na memória interna no exato endereço de memória associado ao nome da variável, no caso a.

A impressão da informação na tela do computador é feita pelo comando write(a);. O que ocorre é que o computador acessa o endereço de memória associado ao nome da variável a e a imprime na tela.
Observe que o uso de uma variável exigiu, pelas regras desta linguagem, a declaração no cabeçalho: \texttt{var a: real;}. Toda variável em \textit{Pascal} deve ter um tipo de dados, que é a forma como o computador vai interpretar a informação, isto é, qual é a codificação em binário que deve ser usada.

Esta declaração faz com que o sistema operacional, antes de executar o programa principal, reserve espaço de memória para a variável \texttt{a}. Este nome é uma abstração para o endereço e seu conteúdo é acessado pelo uso dos nomes. O endereço físico propriamente dito não é de interesse humano e pode ser desprezado.

Existem vários tipos de variáveis, uma das mais simples é \texttt{real}, que usa a codificação em \textit{ponto flutuante} para os bits que estão em memória. A linguagem é fortemente tipada, o que significa que o computador está esperando um número real. Se o usuário digitar qualquer letra o programa aborta com erro.

Em resumo, o funcionamento deste programa é assim:

1. Antes de iniciar o programa principal, o compilador solicita ao \textit{sistema operacional} que reserve um espaço de memória para a variável \texttt{a}, que deve ser interpretada como um número real;

2. Inicia o programa pela primeira instrução (\texttt{read(a);}), que faz com que o computador espere o usuário digitar algo no teclado e apertar ENTER. Se o usuário digitar, por exemplo, o número 2, então o endereço de memória associado à \texttt{a} conterá o valor 2.

3. Executa a segunda instrução (\texttt{write(a);}), que faz com que o computador busque na memória o valor de \texttt{a}, que é 2, e imprima esta informação, isto é, 2, na tela.

4. O programa encontra o \texttt{end.} e termina o programa.

### 5.4 O ciclo edição–compilação–testes

A partir deste ponto o leitor deve ter uma versão do \textit{Free Pascal} para que possa fazer seus experimentos com os códigos aqui fornecidos, praticando o chamado ciclo da programação: edição de código, compilação, execução e testes de programas.

Também deve ter disponível em seu sistema um editor de textos \texttt{ASCII}\footnote{https://pt.wikipedia.org/wiki/ASCII} Isto exclui os pacotes tipo \textit{office}, que fazem negrito, itálico, etc, pois eles introduzem no texto caracteres não visíveis que o compilador não aceita. Exemplos de bons editores de texto: \texttt{vi, emacs, gedit, nano}.

Você pode copiar o programa da figura 5.2 em um destes editores de texto, compilar e executar o programa. Veja o mini guia de referencia para ver como se faz isso, pois depende do seu sistema operacional.

No \textit{linux} pode fazer assim, após o programa editado em um arquivo chamado \texttt{exemplo1.pas}:
$> fpc exemplo1.pas
Free Pascal Compiler version 3.0.4+dfsg-18ubuntu2 [2018/08/29] for x86_64
Copyright (c) 1993-2017 by Florian Klaempfl and others
Target OS: Linux for x86-64
Compiling exemplo1.pas
Linking exemplo1
/usr/bin/ld.bfd: aviso: link.res contém seções de saída; você se esqueceu de -T?
7 lines compiled, 0.4 sec
$> ./exemplo1
Agora você pode digitar o número 2 no teclado e apertar ENTER. Verá a saída assim:

2.00000000000000E+000

Isto porque o formato de saída usado pelo comando write usa notação em ponto flutuante (meio parecido com notação científica). Para imprimir com duas casas decimais, use assim:

```pascal
read (a);
write (a:0:2);
```

Pode também usar uma variante do write. O comando writeln muda de linha após a impressão, melhorando o efeito da impressão. Ficaria assim:

```pascal
read (a);
writeln (a:0:2);
```

2.00

Também é possível imprimir textos (entre aspas simples) misturados com valores de variáveis ou cálculos de expressões aritméticas, que serão apresentadas logo abaixo.

```pascal
read (a);
writeln ('O valor digitado no teclado foi ',a:0:2);
```

2.00

Este processo é um ciclo porque a compilação pode dar erro (erro de compilação: por exemplo você esqueceu um ponto-e-vírgula). O erro pode ser de lógica, quando você pensa que programou certo a sequência de comandos, mas na verdade errou. Então a saída não é a que você esperava.

Em qualquer dos casos é necessário voltar ao editor de textos, corrigir os erros, salvar, recompilar e testar novamente.
5.5 Atribuições e expressões aritméticas

O comando de atribuição em *Pascal* serve para armazenar uma informação em uma variável, isto é, em um endereço de memória. O símbolo usado pelo compilador nesta linguagem é `:=`, ou seja, um dois pontos seguido imediatamente por um igual. Por exemplo: `a := 2;`, faz com que a variável `a` tenha como seu conteúdo o número 2.

O comando de atribuição tem na verdade três componentes: o próprio símbolo `:=`, o lado esquerdo e o lado direito. O lado esquerdo é obrigatoriamente o nome de uma única variável. O lado direito é uma expressão (uma sequência de operações) que deve resultar em um valor do mesmo tipo da variável que está do lado esquerdo. Esta expressão pode ser de dois tipos: ou é uma *expressão aritmética* ou é uma *expressão booleana*. Esta última veremos na próxima sessão.

Tomemos como exemplo o programa mostrado no capítulo 4, que implementava uma solução para o problema de se calcular as raízes da equação do segundo graus: $x^2 - bx + c = 0$. A figura 5.3 é uma cópia daquela solução apresentada na figura 4.11.

![Figura 5.3: Programa que implementa o método de Bhaskara.](image)

Este código simples é rico em elementos das linguagens de programação. Ele contém quatro elementos importantes: os comandos de entrada e saída, o comando de atribuição e as expressões aritméticas. Os três primeiros já estamos mais familiarizados.

Lembrando que foram declaradas no cabeçalho três variáveis do tipo *real*:

```pascal
var b, c, raizdiscriminante: real;
```

Este programa têm:

1. Um primeiro comando de leitura, da variável `b`: `read(b);`.
2. Um segundo comando de leitura, da variável `c`: `read(c);`.
3. Um comando de atribuição para a variável `raizdiscriminante`). Este comando faz uso de uma *expressão aritmética*

   `raizdiscriminante := sqrt(b*b - 4*c);`
5.5. ATRIBUIÇÕES E EXPRESSÕES ARITMÉTICAS

4. Um primeiro comando de impressão, de uma outra expressão aritmética:
   \[ \text{write}((\text{b} - \sqrt{\text{discriminante}})/2); \]

5. Um segundo comando de impressão, de uma terceira expressão aritmética:
   \[ \text{write}((\text{b} + \sqrt{\text{discriminante}})/2);+ \]

Os comandos de leitura (\textit{read}) servem para o usuário que está rodando o programa informar ao computador qual é a equação do segundo grau que ele deseja conhecer as raízes, isto é feito pela digitação, no teclado, dos valores das variáveis \textit{b} e \textit{c}. Uma vez executados, os valores digitados no teclado estarão armazenados nos conteúdos destas duas variáveis, respectivamente. O compilador é rígido, está esperando valores reais, por isso digitar letras ou qualquer coisa que não seja número dará erro na execução. Você pode digitar 5 e 6 por exemplo (ou 5.0 e 6.0) e depois apertar \textit{ENTER}. A equação a ser resolvida será \( x^2 - 5x + 6 = 0 \).

O terceiro comando (:=) é uma \textit{atribuição}, que significa atribuir o resultado do cálculo que está no lado direito da notação := para a variável que está no lado esquerdo. Basicamente é uma conta que deve ser feita (calcular a raiz quadrada do cálculo de outra expressão, que é multiplicar \textit{b} por \textit{b} e subtrair o resultado pela operação de multiplicar 4 por \textit{c}). O resultado disso tudo é um valor numérico do tipo \textit{real}. Este último valor é atribuído na variável \textit{raizdiscriminante}.

Vamos detalhar o que ocorre, supondo que os valores de \textit{b} e \textit{c} sejam respectivamente 5 e 6. Maiores detalhes devem ser vistos no material complementar sobre a linguagem \textit{Pascal}.

Antes de fazer a operação de atribuição, o lado direito que segue o := deve ser definido. Para isso, a primeira operação é a multiplicação de \textit{b} com \textit{b}, denotado \textit{b} \(*\textit{b}\. Isto resulta no valor 25. Em seguida, é feita a multiplicação de 4 pelo valor de \textit{c} , denotado \textit{4} * \textit{c}. Isto resulta no valor 24. O terceiro passo é a subtração de 25 por 24, resultando no valor 1. Finalmente, \textit{sqrt} é uma função pré-definida em \textit{Pascal} que calcula o valor da raiz quadrada do valor da expressão entre parênteses. O resultado de fazer \textit{sqrt}(1) resulta no valor 1. Finalmente, este valor, 1, é atribuído na variável \textit{raizdiscriminante}.

Resumindo, o computador deve realizar o cálculo da expressão aritmética do lado direito do símbolo := e somente após armazenar o valor resultante na variável que aparece do lado esquerdo.

As duas últimas linhas contêm o comando \textit{write}, que serve para imprimir alguma coisa na tela do usuário, neste caso, o resultado dos cálculos das expressões aritméticas \( \frac{\textit{b} - \sqrt{\textit{discriminante}}}{2} \) e \( \frac{\textit{b} + \sqrt{\textit{discriminante}}}{2} \), nesta ordem. Observe a notação no programa, o modo de se representar estas operações faz uso de parênteses e dos símbolos + para adição, - para subtração, \(*\ para multiplicação e uma barra (/) para divisão real. O material complementar detalha melhor a forma das expressões aritméticas e a ordem em que as operações ocorrem.

Desta forma, expressões aritméticas servem para fazer cálculos que resultam em um valor. Elas seguem algumas restrições que estão fora do escopo deste livro, você deve procurar estudar na literatura indicada.

Mas podemos dizer que as expressões usam \textit{operadores} e estes operadores são executados segundo uma \textit{ordem de precedência}. Por exemplo, multiplicação (*) e
divisão (/) têm precedência sobre a adição (+). Assim, se a conta a ser feita for

\[ a = \frac{b + c}{2} \]

A expressão correta é: \((b + c)/2\). Exatamente porquê a divisão ocorre primeiro. Se não proteger com os parênteses, escrevendo errado assim: \(b + c/2\), a conta que estaria sendo feita é:

\[ a = b + \frac{c}{2} \]

Observação: quando usamos a função \texttt{sqrt} não previmos que seu \textit{argumento}, isto é: \(b*b - 4*c\) pode resultar em valor negativo. O cálculo da raiz quadrada de um valor negativo gera erro de execução (\textit{runtime error}). Nós vamos ver como controlar esta situação ao longo deste texto. Por enquanto vamos acreditar que o usuário sempre vai entrar números que não resultem nesse erro.

### 5.5.1 Exemplo

Vamos aqui considerar um outro problema bem simples.

**Problema:** Ler dois números do teclado e imprimir a soma deles na tela.

O programa apresentado na figura 5.4 está correto e captura a essência da solução! Os comandos de leitura carregam os números digitados no teclado na memória, em posições acessíveis a partir dos nomes \(a\) e \(b\). Em seguida, uma \textit{expressão aritmética} faz o cálculo da soma de ambos e o resultado é impresso na tela.

```plaintext
program soma2;
var a,b: integer;
begin
  read (a);
  read (b);
  write (a+b);
end.
```

Figura 5.4: Primeira solução.

Um pequeno problema é que, quando executado, o cursor fica piscando na tela e não deixa nenhuma mensagem sobre o que o usuário deve digitar.

O estudante pode querer modificar ligeiramente este código para produzir uma interface um pouco mais amigável para quem usa o programa. A versão minimamente modificada para este problema é apresentada na figura 5.5.

O programador iniciante deve ter em mente que não deve perder muito tempo com “firulas” na tela, pelo menos não neste curso. Em outras disciplinas, quando a arte da programação estiver dominada, o estudante aprenderá a integrar elegantemente uma
5.6. VARIÁVEIS E EXPRESSÕES BOOLEANAS

```pascal
program soma2;
var a,b: integer;
begin
    write ('entre com o valor de a: ');
    read (a);
    write ('entre com o valor de b: ');
    read (b);
    writeln (a,'+',b,'=', a+b);
end.
```

Figura 5.5: Mesma solução, agora com interface amigável.

interface amigável com o usuário do programa ao mesmo tempo mantendo o código legível. Neste exemplo usamos a outra versão do comando de impressão, o `writeln`, que além de imprimir na tela muda o cursor de linha.

Também aproveitamos para mostrar que o argumento para o comando `write` (ou o `writeln`) pode ser constituído de sequências de informações separadas por vírgula. Assim, misturamos mensagens de texto, entre aspas simples, com valores de variáveis. Os textos contêm também espaços em branco, que são impressos tal como o programador escreveu.

Assim a execução deste programa para a entrada 2, 3 é:

```
entre com o valor de a: 2
entre com o valor de b: 3
2+3= 5
```

Aproveitamos para mostrar também um segundo tipo de dados aceito pela linguagem Pascal, que é o tipo `integer`, usado para números inteiros que usam representação de complemento de dois de endereços de 2 bytes (8 bits). Isto permite representar números entre -32768 e 32767. Leia sobre representação binária para entender melhor este aspecto das linguagens de programação.

Observação: Importante testar bem este programa para vários valores diferentes para as entradas a e b. Em especial, teste para valores negativos, nulos, valores grandes ou mesmo valores reais (como por exemplo `a= 1.3` e `b= 1.5`). Anote suas observações sobre os erros que surgirem.

5.6  Variáveis e expressões booleanas

As expressões booleanas, também conhecidas como expressões lógicas, são utilizadas para resultar em um valor do tipo `boolean`, que passa a ser o terceiro tipo básico da linguagem Pascal.

O tipo `boolean` só pode conter dois valores: verdadeiro (`true`) e falso (`false`). O caso mais simples de uma expressão booleana é uma variável do tipo `boolean`. Por exemplo:
var OK: boolean;

Assim podemos fazer uma atribuição de um valor verdadeiro ou falso para esta variável: OK := true; ou OK := false.

As expressões booleanas mais complexas podem envolver o uso de expressões aritméticas juntamente com os operadores relacionais, que são: estritamente maior (>), maior ou igual (⟩=), estritamente menor (<), menor ou igual (⟨=), igual (=) e diferente (≟). Também é possível usar conectivos lógicos, que são: o ou lógico (OR), o e lógico (AND) e a negação lógica (NOT).

Alguns exemplos de expressões booleanas deste tipo são:

- \( a > 0 \): \( a \) é estritamente maior do que zero;
- \( a + b = x + y \): a soma de \( a \) com \( b \) é igual à soma de \( x \) com \( y \);
- \( \sqrt{b^2 + 4ac} \geq 0 \): a raiz quadrada do discriminante de uma equação do segundo grau é positiva ou nula, isto é, não é negativa.
- \( (a \geq 3) \land (a \leq 5) \): \( a \) está no intervalo \( 3 \leq a \leq 5 \). Observação: não pode escrever \( 3 \leq a \leq 5 \), o compilador rejeita.
- \( \neg ((a \geq 3) \land (a \leq 5)) \): resulta no valor contrário à da expressão anterior, isto é, \( a \) deve ser estritamente menor do que 3 OU estritamente maior do que 5.

Evidentemente o valor destas expressões depende do valor individual de cada variável ou constante utilizada, mas sempre resulta em verdadeiro ou falso.

Você deve consultar o mini guia da linguagem Pascal e também a literatura ali recomendada para entender como construir expressões booleanas válidas, pois elas também seguem algumas regras e ordens de precedência. Nas próximas sessões o entendimento de expressões booleanas é fundamental.

5.7 Repetição de comandos

Nesta seção usaremos como apoio um problema muito simples.

Problema: Imprimir todos os números entre 1 e 5.

Uma solução óbvia, porém ruim, é ilustrado na figura 5.6.

Após compilado e executado, os números 1, 2, 3, 4 e 5 aparecem na tela, atendendo ao enunciado. Mas, à luz das discussões do capítulo 2, é fácil ver que esta solução não é muito boa, pois não permite uma reimplementação simples de problemas similares. Por exemplo, se o enunciado fosse “Imprimir todos os números entre 1 e 10” no lugar de “todos entre 1 e 5”, teríamos que escrever um código como o ilustrado na figura 5.7.

Extrapolando o enunciado do problema, se fosse para imprimir números entre 1 e 30.000 ou entre 1 e 100.000.000, o programa ficaria com 30 mil ou 100 milhões de linhas de código extremamente repetitivo e de difícil e custosa edição.
5.7. REPETIÇÃO DE COMANDOS

program contar;
begin
  writeln (1);
  writeln (2);
  writeln (3);
  writeln (4);
  writeln (5);
end.

Figura 5.6: Primeira solução para contar de 1 a 5.

program contar;
begin
  write (1);
  write (2);
  write (3);
  write (4);
  write (5);
  write (6);
  write (7);
  write (8);
  write (9);
  write (10);
end.

Figura 5.7: Primeira solução modificada para números de 1 a 20.

A simplicidade do raciocínio inicial resultou em uma tarefa tão exaustiva que o computador deixou de ajudar, ele passou a atrapalhar!

Felizmente, há algoritmos melhores!

O computador é, conforme vimos, uma máquina com pouquíssimos recursos, mas que, se bem explorados, nos permite escrever programas fantásticos. O raciocínio deve ser então desenvolvido de tal maneira que o trabalho exaustivo fique com o computador e não com o programador.

Para resolver este problema de modo elegante é de uma estrutura que permita que um determinado trecho de código seja repetido. Mas não só isso, é preciso entender a lógica do raciocínio que está por trás deste comando.

A estrutura é um comando de repetição. Ele funciona com base em uma expressão booleana. Os comandos desejados serão repetidos enquanto uma determinada condição for verdadeira, forçando assim uma mudança natural do fluxo de execução que aprendemos até aqui, isto é, executar os comandos de de cima para baixo.

A linguagem Pascal oferece três maneiras de se fazer isto. Veremos somente uma delas por enquanto. No decorrer do curso voltaremos às outras formas. A figura 5.8 ilustra o uso deste conceito.

O comando de repetição (while/do) executa os comandos aninhados no bloco entre o begin e o end; enquanto a expressão booleana $i \leq 5$ for verdadeira. No primeiro
CAPÍTULO 5. CONCEITOS ELEMENTARES

program contar;
var i: integer;
begin
  i:= 1;
  while i <= 5 do
    begin
      writeln(i);
      i:= i + 1;
    end;
end.

Figura 5.8: Sexta solução.

momento em que for falsa, o fluxo é alterado para depois do end; , ou seja, para o fim do programa.

Esta expressão vai resultar falso quando $i$ for estritamente maior do que 5. O símbolo $\leq$ significa menor ou igual, logo, $i \leq 5$ significa testar se o valor de $i$ é ou não é menor ou igual a 5. Se for, o resultado da expressão booleana é true (verdadeiro), senão, é false (falso). Enquanto a expressão for avaliada como verdadeiro, os comandos do laço são executados. Na primeira avaliação da expressão de resultar em falso, pula o laço e termina o programa.

Neste exemplo, a variável $i$ foi inicializada com o valor 1. Em seguida, os comandos de impressão e incremento são executados enquanto o valor de $i$ for menor ou igual a 5. Desta forma, o número 1 é impresso, $i$ passa a valer 2, que é menor ou igual a 5, então 2 é impresso e $i$ passa a valer 3, que por sua vez ainda é menor ou igual a 5. Então 3 é impresso na tela e $i$ passa a valer 4. Então 4 é impresso na tela e $i$ passa a valer 5, que ainda é menor ou igual a 5. Por isso o valor 5 é impresso na tela e $i$ passa a valer 6. Finalmente, o teste é executado e o programa determina que 6 não é menor ou igual a 5. Neste momento o programa termina.

Observem que a cada vez que o comando $i:= i + 1$; é executado, o programa encontra o end; (e não o end.) e altera o fluxo de execução para a reavaliação da expressão booleana $i \leq 5$.

Observação: Este programa pode ser facilmente modificado para imprimir valores que vão de 1 até o maior valor integer, que é 32767. Basta trocar o 5 pelo novo valor, recompilar e executar novamente.

Observação 2: Se quisermos imprimir valores maiores, basta trocar o tipo integer por outro que admita uma faixa maior de valores. Consultando o mini guia de referência da linguagem Pascal, podemos utilizar o tipo longint, que usa representação de 4 bytes.
5.7. REPETIÇÃO DE COMANDOS

5.7.1 Exemplo

Ainda no contexto de problemas simples, este outro problema nos permite combinar as técnicas usadas nos problemas anteriores e resolver algo ligeiramente mais complicado.

Problema: Imprimir uma tabela com os valores de \( x \) e \( x^2 \) para todos os valores inteiros de \( x \) tais que \( 1 \leq x \leq 30 \).

O programa que ilustra a solução para este problema é apresentado na figura 5.9 e imprime todos os valores inteiros de 1 a 30, juntamente com seus respectivos quadrados.

```plaintext
program quadrados;
var i: integer;
begin
  i:= 1;
  while i <= 30 do
  begin
    writeln (i, ', ', i*i);
    i:= i + 1;
  end;
end.
```

Figura 5.9: Tabela com os quadrados dos números de 1 a 30.

O programa inicia com o valor de \( i \) igual a 1 e para cada valor entre 1 e 30 imprime na tela o próprio valor, seguido de um espaço em branco e finalmente o quadrado do número, calculado pela expressão aritmética \( i \times i \), que significa \( i^2 \).

5.7.2 Critério de parada

Vimos que um comando de repetição altera o fluxo natural do programa de maneira a executar um certo número de vezes alguns comandos, aqueles que estão entre o `begin` e o `end;`.

É preciso tomar cuidado para que esta repetição não seja infinita, pois como foi dito, a repetição deve ocorrer um certo número de vezes, mas de maneira controlada.

A repetição sempre é controlada pela variável de controle, que no caso do exemplo anterior é \( i \). Observem dois fatos importantes:

- A variável de controle ocorre do lado esquerdo de um comando que está dentro do laço!

Significa que ela, a variável de controle, sofre alterações com o passar das diversas repetições e com isso existe a chance de em algum tempo futuro, contando da primeira entrada no laço, da condição do laço \( i \leq 5 \) vir um dia a ser falsa, o que provocará o término deste laço.
O teste \(i \leq 5\) do comando `while/do` é feito *antes* dos comandos internos do laço (dizemos que estes são os comandos que estão no *escopo* do `while`).

Isto significa que, dependendo do valor inicial da variável de controle, os comandos no escopo do `while` podem nunca ocorrerem. De fato, veja o que ocorre que na linha 5 do programa o comando `i:=1` for substituído por `i:=6` (ou qualquer valor maior): o teste do laço resultará `falso` logo de cara, então os comandos do laço não são executados e o fluxo do programa passa diretamente para o que vier após o `end;`.

### 5.7.3 Critérios alternativos de parada da repetição

**Problema:** Ler vários pares de números e imprimir a soma de cada par.

Este é uma generalização do problema da seção 5.5.1. Naquele caso um único par de número era lido do teclado e a soma deles impressa. Agora temos que fazer a mesma coisa para vários pares de números dados como entrada.

A solução para um único par já estando feita, basta que o programa repita o mesmo trecho de código várias vezes, usando o comando `while/do`.


É necessário estabelecer uma condição de parada e isto deve estar claro no enunciado. Existem duas formas de se resolver isto:

- ou o enunciado estabelece a quantidade exata de números a serem lidos;
- ou o enunciado estabelece uma condição de parada alternativa.

No primeiro caso o enunciado deveria ser algo assim: “ler 30 pares de números do teclado e imprimir, para cada par, a soma deles”.

No segundo caso poderia ser algo mais ou menos assim: “ler pares de números do teclado e imprimir, para cada par, a soma deles. O algoritmo deve parar a execução quando os dois números lidos forem iguais a zero”.

A figura 5.10 ilustra as duas formas. A da esquerda apresenta a solução usando-se um contador, a da direita apresenta a solução com critério de parada alternativo.

O que difere de um para o outro é a estrutura de controle do laço, isto é, do bloco de comandos que se repete. No quadro da esquerda, o algoritmo precisa contar até 30, por isto existe uma variável `cont` que faz esta contagem. No quadro da direita, o laço é controlado por uma expressão booleana um pouco mais sofisticada que a do exemplo anterior, pois faz uso do operador `or`. A expressão em questão se torna falsa quando ambos os números lidos forem nulos. Se um dois dois for não nulo, o laço é executado.

O comando de impressão ficou aparentemente invertido na versão da direita pela simples razão de que o teste depende de uma primeira leitura das variáveis `a` e `b *antes do teste*, senão o teste não pode ser feito. Mas, ressaltamos, o núcleo do programa é
5.8 Desviando fluxo condicionalmente

Problema: Ler um único número do teclado e imprimi-lo se ele for positivo.

Parece simples para o ser humano saber se um número é positivo ou não, mas para o computador o que está em memória é basicamente uma sequência de bits. Logo, para ele decidir se um número é positivo deve usar uma expressão booleana. Mas, como executar o comando de impressão somente em um caso (do número ser positivo) e ignorar a impressão caso não seja? Este problema permitirá introduzir a noção de desvio condicional.

Um desvio condicional produz exatamente o efeito desejado, faz um teste e dependendo do resultado executa ou não o comando subsequente, ou melhor dizendo, no seu escopo.

A figura 5.11 ilustra a solução deste problema. No caso, o comando writeln só é executado se a expressão booleana for verdadeira. Caso o número lido seja nulo ou negativo o fluxo de execução do programa “pula” para o comando subsequente, que no caso é o fim do programa.

Observamos que, diferentemente do comando while, o if não implica em repetição de comandos. O if apenas decide se o próximo comando será executado ou não.

5.8.1 Fluxo alternativo no desvio condicional

O comando de desvio condicional admite uma segunda forma, que estabelece um fluxo alternativo ao programa dependendo do teste.
Programa 5.11: Imprime se for positivo.

Considere o seguinte problema.

Problema: Ler um único número do teclado e imprimi-lo se ele for positivo. Se não for imprimir a mensagem “número inválido”.

A solução deste problema requer uma variante do comando if, conforme ilustrado na figura 5.12. Apenas um dos dois comandos de impressão será executado. Ou imprime o número se ele for positivo, ou imprime a mensagem se ele for zero ou negativo. Nunca ambos.

Figura 5.12: Imprime se for positivo, segunda versão.

Observação: Em Pascal não é permitido colocar um ponto-e-vírgula (;) antes do else.

5.9 Resumo

Neste capítulo vimos todas as estruturas elementares para qualquer algoritmo (ou programa) presentes em qualquer linguagem de programação. São eles:

Comandos

- Entrada e saída (read e write, respectivamente);
- Atribuição (:=);
- Repetição (while/do);
5.9. RESUMO

- Desvio condicional (if/then, ou if/then/else);

Expressões

- Aritméticas;
- Booleanas.

Qualquer algoritmo é escrito como uma combinação destes comandos manipulando dados em memória (variáveis) da maneira como o algoritmo estabelece. Como sabemos, cada problema tem várias soluções possíveis, mas uma vez fixado uma, pode-se escrever o programa na forma como o computador entenda, usando-se somente as noções apresentadas neste capítulo.

A menos do uso de estruturas de dados sofisticadas, as quais veremos a partir do capítulo 9, agora já é possível escrever qualquer algoritmo, e consequentemente, qualquer programa!

O que muda de uma linguagem de programação para outra é basicamente a grafa dos comandos, as vezes com alguma ligeira modificação na sintaxe e na semântica do comando.

Por exemplo, na linguagem C, o comando write é grafado printf, na linguagem BASIC é grafado print. Cada linguagem tem um comportamento diferente, não é apenas o nome que muda. Por exemplo, se imprime e muda de linha ou não, em qual formato escreve, etc.

Mas, estes são os elementos básicos para se programar. No próximo capítulo veremos como usar estes comandos de maneira inteligente para resolver problemas mais complexos. O leitor pode pular o apêndice da próxima seção sem prejuízo da compreensão do restante do texto.

Importante: Antes de continuar a leitura deste texto você deve fazer os exercícios deste capítulo. Como já anteriormente mencionado, não se aprende a programar sem programar. Sem de fato saber resolver estes exercícios, você não acompanhará os conceitos que virão.

Apêndice: Desviando fluxo incondicionalmente

Existe mais um comando de controle de fluxo, presente em qualquer linguagem de programação: o comando de desvio incondicional. Segundo o modelo de Von Neumann estudado, isto nada mais é do que alterar o controlador de instruções para um endereço arbitrário (na verdade controlado) quando pensamos no modelo de baixo nível.

A figura 5.13 ilustra o uso do desvio incondicional para resolver o problema de se imprimir todos os números de 1 a 30.

Neste exemplo, a variável i é inicializada com o valor 1 e em seguida é executado o comando que imprime 1 na tela. Em seguida a variável i é incrementada e, após verificar que 1 é menor ou igual a 30, o fluxo do programa executa o comando de desvio incondicional goto, que faz com que o fluxo de execução do programa vá para a linha indicada pelo rótulo 10, isto é, imprime o valor da variável i, incrementa o i e
CAPÍTULO 5. CONCEITOS ELEMENTARES

program contar;
label: 10;
var i: integer;
begin
    i:= 1;

    10:write (i);
    i:= i + 1;

    if i <= 30 then
        goto 10;
end.

Figura 5.13: Exemplo do uso do desvio incondicional.

assim por diante. Quando $i$ valer 31 o goto não é executado, o que faz com que o end final seja atingido e o programa termina.

Em outras palavras, é uma outra maneira (mais estranha) de se implementar o mesmo programa da figura 5.8. A observação interessante é que, no nível de máquina, o que o compilador faz é gerar a partir do programa da figura 5.8, um código de máquina implementado usando-se uma noção de desvio incondicional, implementada no repertório reduzido de instruções, mas isto o programador não é obrigado a saber agora.

O que ele deve saber é que o uso de comandos de desvio incondicional não é recomendado pois na medida em que os programas vão ficando com muitas linhas de código, digamos algumas centenas de linhas ou mais, o que ocorre é que o programador tende a se perder e passa a ter muita dificuldade em continuar o desenvolvimento do programa quando o uso do goto é feito de qualquer maneira e, em especial, de forma exagerada. Isto tende a tornar o código ilegível e de difícil manutenção.

Como vimos, as linguagens de alto nível permitem mecanismos mais elegantes para repetir trechos de código sem necessidade do uso do goto, usando-se o while. Isto é baseado no princípio da programação estruturada, que nasceu com a linguagem ALGOL-60. Esta linguagem, apesar de não ser mais usada, deu origem à maior parte das linguagens modernas, entre elas o próprio Pascal.

Este comando está em um apêndice pois nós nunca mais o usaremos.
5.10 Exercícios

Baixe o mini guia da linguagem *Pascal*, disponível em:


também disponível online em:

http://wiki.inf.ufpr.br/marcos/doku.php?id=pascal

Você vai precisar estudá-lo para poder resolver boa parte dos exercícios desta seção, pois como já informado, este material complementa partes dependentes da versão da linguagem *Pascal*, versão *Free Pascal*, que naturalmente evolui com o tempo.

Os exemplos de entrada e saída que acompanham os enunciados devem ser vistos como um caso de entrada, você deve testar com diferentes entradas para se certificar que o seu programa funciona para diferentes entradas.

### 5.10.1 Expressões aritméticas

1. Considere o seguinte programa incompleto em *Pascal*:

```pascal
program tipos;
var
    A: <tipo>;
    B: <tipo>;
    C: <tipo>;
    D: <tipo>;
    E: <tipo>;
begin
    A := 1 + 2 * 3;
    B := 1 + 2 * 3 / 7;
    C := 1 + 2 * 3 div 7;
    D := 3 div 3 * 4.0;
    E := A + B * C - D
end.
```

Você deve completar este programa indicando, para cada variável de *A* até *E*, qual é o tipo correto desta variável. Algumas delas podem ser tanto inteiros como reais, enquanto que algumas só podem ser de um tipo específico. Para resolver este exercício você precisa estudar sobre os operadores inteiros e reais e também sobre a ordem de precedência de operadores que aparecem em uma expressão aritmética. Sua solução estará correta se seu programa compilar.

2. Faça um programa em *Pascal* que leia 6 valores reais para as variáveis *A, B, C, D, E, F* e imprima o valor de *X* após o cálculo da seguinte expressão aritmética:

\[
X = \frac{A + B}{C - D} E + E
\]
Seu programa deve assumir que nunca haverá divisões por zero para as variáveis dadas como entrada. Note que neste programa a variável $X$ deve ser do tipo real, enquanto que as outras variáveis podem ser tanto da família ordinal ($integer$, $longint$, etc) como também podem ser do tipo real.

<table>
<thead>
<tr>
<th>Exemplo Entrada</th>
<th>Saída esperada</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2 3 4 5 6</td>
<td>-1.8750000000000000E+000</td>
</tr>
<tr>
<td>1 -1 1 -1 1 -1</td>
<td>0.0000000000000000E+000</td>
</tr>
<tr>
<td>3 5 8 1 1 2</td>
<td>1.008403361345378E+000</td>
</tr>
</tbody>
</table>

3. Faça um programa em Pascal as seguintes expressões aritméticas usando o mínimo possível de parênteses. Para resolver este exercício você precisa estudar sobre precedência de operadores em uma expressão aritmética. Dica: para elevar um número ao quadrado multiplique este número por ele mesmo ($x^2 = x \times x$).

(a)

\[
\frac{W^2}{Ax^2 + Bx + C}
\]

(b)

\[
\frac{\frac{P_1 + P_2}{Y - Z} R}{\frac{W}{AB} + R}
\]

Observe que os compiladores não suportam o uso de subescritos, que são utilizados na notação matemática. Então no lugar de $P_1$ e $P_2$, você pode dar os nomes para as variáveis de $p1$ e $p2$ respectivamente.

4. Faça um programa em Pascal que some duas horas. A entrada deve ser feita lendo-se dois inteiros por linha, em duas linhas, e a saída deve ser feita no formato especificado no exemplo abaixo:

<table>
<thead>
<tr>
<th>Exemplo Entrada</th>
<th>Saída esperada</th>
</tr>
</thead>
<tbody>
<tr>
<td>12 52</td>
<td>12:52 + 7:13 = 20:05</td>
</tr>
<tr>
<td>7 13</td>
<td>20:15 + 1:45 = 22:00</td>
</tr>
<tr>
<td>0 0</td>
<td>0:0 + 8:35 = 8:35</td>
</tr>
</tbody>
</table>

Você deve observar que o comando de impressão deve imprimir os espaços em branco e os símbolos “+” e “=” conforme o enunciado exige.

5. Faça um programa em Pascal que, dado um número inteiro que representa uma quantidade de segundos, determine o seu valor equivalente em graus, minutos e segundos. Se a quantidade de segundos for insuficiente para dar um valor em graus, o valor em graus deve ser 0 (zero). A mesma observação vale em relação aos minutos e segundos.
5.10. **EXERCÍCIOS**

<table>
<thead>
<tr>
<th>Exemplo Entrada</th>
<th>Saída esperada</th>
</tr>
</thead>
<tbody>
<tr>
<td>3600</td>
<td>1, 0, 0</td>
</tr>
<tr>
<td>3500</td>
<td>0, 58, 20</td>
</tr>
<tr>
<td>7220</td>
<td>2, 0, 20</td>
</tr>
</tbody>
</table>

6. Faça um programa em *Pascal* que troque o conteúdo de duas variáveis. Exemplo:

<table>
<thead>
<tr>
<th>Exemplo Entrada</th>
<th>Saída esperada</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 7</td>
<td>7 3</td>
</tr>
<tr>
<td>-5 15</td>
<td>15 -5</td>
</tr>
<tr>
<td>2 10</td>
<td>10 2</td>
</tr>
</tbody>
</table>

7. (*) Desafio: Faça um programa em *Pascal* que troque o conteúdo de duas variáveis *inteiros* sem utilizar variáveis auxiliares. Pense em fazer contas de adição e/ou subtração com os números.

5.10.2 **Expressões booleanas**

Para resolver estes exercícios você precisa estudar sobre sobre expressões lógicas (ou booleanas) e sobre a ordem de precedência dos operadores lógicos (NOT, AND, OR).

8. Indique qual o resultado das expressões abaixo, sendo:

\[ a = 6;\ b = 9.5;\ d = 14;\ p = 4;\ q = 5;\ r = 10;\ z = 6.0;\ \text{sim} = \text{TRUE}. \]

(a) `\text{sim AND (q \ge p)}`
(b) `(0 \le b) AND (z > a) OR (a = b)`
(c) `(0 \le b) AND ((z > a) OR (a = b))`
(d) `(0 \le b) OR ((z > a) AND (a = b))`

5.10.3 **Expressões aritméticas e booleanas**

Para resolver estes exercícios você precisa estudar sobre os operadores inteiros e reais e também sobre a ordem de precedência de operadores que aparecem em uma expressão aritmética. Adicionalmente, você precisa estudar sobre expressões lógicas (ou booleanas) e sobre a ordem de precedência dos operadores relacionais (=, <>, ≤, ≥, >, <) e lógicos (NOT, AND, OR).

9. Indique qual o resultado das expressões abaixo, sendo:

\[ a = 6;\ b = 9.5;\ d = 14;\ p = 4;\ q = 5;\ r = 10;\ z = 6.0;\ \text{sim} = \text{TRUE}. \]
(a) \( \text{NOT sim AND (} z \div b + 1 = r \) \)
(b) \( (x + y > z) \text{ AND sim OR (} d \ge b \) \)
(c) \( (x + y <> z) \text{ AND (} sim \text{ OR (} d \ge b \) \)

10. Indique qual o resultado das expressões abaixo, sendo:

\[ a = 5; b = 3; d = 7; p = 4; q = 5; r = 2; x = 8; y = 4; z = 6; \text{sim} = \text{TRUE}. \]

(a) \( (z \div a + b \times a) - d \div 2 \)
(b) \( p \div r \text{ mod q} - q \div 2 \)
(c) \( (z \div y + 1 = x) \text{ AND sim OR (} y >= x \) \)

5.10.4 Acompanhamento de programas

11. Dado o programa em Pascal abaixo, mostre o acompanhamento de sua execução para três valores de entrada (valores pequenos, por exemplo para \( x = 0 \), \( x = 10 \), e \( x = -1 \)). Em seguida, descreva o que o programa faz.

```pascal
program questao1;
var
m, x, y: integer;
begin
read(x);
y := 0;
m := 1;
while x > 0 do
begin
y := y + (x mod 2) \times m;
x := x \div 2;
m := m \times 10;
end;
writeln(y)
end.
```

5.10.5 Programas com um desvio condicional

12. Faça um programa em Pascal que leia um número \( n \) do teclado e decida se ele é positivo ou negativo. Seu programa deve imprimir a mensagem “par” ou “impar” conforme o caso. Exemplo:

<table>
<thead>
<tr>
<th>Exemplo Entrada</th>
<th>Saída esperada</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>impar</td>
</tr>
<tr>
<td>4</td>
<td>par</td>
</tr>
<tr>
<td>15</td>
<td>impar</td>
</tr>
</tbody>
</table>
13. Faça um programa em *Pascal* que leia dois números \( n, m \) do teclado e decida se ele o primeiro é maior do que o segundo. Seu programa deve imprimir a mensagem “primeiro eh maior” ou “segundo eh maior ou igual” conforme o caso. Exemplo:

<table>
<thead>
<tr>
<th>Exemplo Entrada</th>
<th>Saída esperada</th>
</tr>
</thead>
<tbody>
<tr>
<td>5 2</td>
<td>primeiro eh maior</td>
</tr>
<tr>
<td>2 5</td>
<td>segundo eh maior ou igual</td>
</tr>
<tr>
<td>5 5</td>
<td>segundo eh maior ou igual</td>
</tr>
</tbody>
</table>

14. Faça um programa em *Pascal* que leia três números \( x, y, z \) do teclado e decida se \( x \leq y < z \). Seu programa deve imprimir a mensagem “esta no intervalo” ou “nao esta no intervalo” conforme o caso. Exemplo:

<table>
<thead>
<tr>
<th>Exemplo Entrada</th>
<th>Saída esperada</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 5 8</td>
<td>esta no intervalo</td>
</tr>
<tr>
<td>3 8 8</td>
<td>nao esta no intervalo</td>
</tr>
<tr>
<td>4 12 5</td>
<td>nao esta no intervalo</td>
</tr>
</tbody>
</table>

15. Faça um programa em *Pascal* que leia três números \( x, y, z \) do teclado e decida se \( x > y \) ou se \( y < z \). Seu programa deve imprimir a mensagem “sim” em caso afirmativo e “nao” caso contrário. Exemplo:

<table>
<thead>
<tr>
<th>Exemplo Entrada</th>
<th>Saída esperada</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 5 8</td>
<td>sim</td>
</tr>
<tr>
<td>3 8 8</td>
<td>nao</td>
</tr>
<tr>
<td>4 12 5</td>
<td>nao</td>
</tr>
</tbody>
</table>

16. Faça um programa em *Pascal* que leia 6 valores reais para as variáveis \( A, B, C, D, E, F \) e imprima o valor de \( X \) após o cálculo da seguinte expressão aritmética:

\[
X = \frac{A + B}{\frac{C - D}{E}} + E
\]

Seu programa deve imprimir a mensagem “divisão por zero” caso o denominador seja zero. Caso isso não ocorra seu programa irá abortar neste caso, o que não é correto.

Exemplos de entrada e saída:

<table>
<thead>
<tr>
<th>Exemplo Entrada</th>
<th>Saída esperada</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2 3 4 5 6</td>
<td>-1.8750000000000000E+000</td>
</tr>
<tr>
<td>0 0 0 0 0 0</td>
<td>divisao por zero</td>
</tr>
<tr>
<td>1 1 2 2 1 3</td>
<td>divisao por zero</td>
</tr>
</tbody>
</table>
5.10.6 Programas com um laço

17. Faça um programa em *Pascal* que leia uma massa de dados onde cada linha da entrada contém um número par. Para cada número lido, calcule o seu sucessor par, imprimindo-os dois a dois em listagem de saída. A última linha de dados contém o número zero, o qual não deve ser processado e serve apenas para indicar o final da leitura dos dados. Exemplo:

<table>
<thead>
<tr>
<th>Exemplo Entrada</th>
<th>Saída esperada</th>
</tr>
</thead>
<tbody>
<tr>
<td>12 6 26 86 0</td>
<td>12 14</td>
</tr>
<tr>
<td></td>
<td>6 8</td>
</tr>
<tr>
<td></td>
<td>26 28</td>
</tr>
<tr>
<td></td>
<td>86 88</td>
</tr>
<tr>
<td>-2 -5 -1 0</td>
<td>-2 0</td>
</tr>
<tr>
<td></td>
<td>-5 -3</td>
</tr>
<tr>
<td></td>
<td>-1 1</td>
</tr>
<tr>
<td>1 2 3 4 5 0</td>
<td>1 3</td>
</tr>
<tr>
<td></td>
<td>2 4</td>
</tr>
<tr>
<td></td>
<td>3 5</td>
</tr>
<tr>
<td></td>
<td>4 6</td>
</tr>
<tr>
<td></td>
<td>5 7</td>
</tr>
</tbody>
</table>

18. Faça um programa em *Pascal* que leia uma massa de dados contendo a definição de várias equações do segundo grau da forma $Ax^2 + Bx + C = 0$. Cada linha de dados contém a definição de uma equação por meio dos valores de $A$, $B$ e $C$ do conjunto dos números reais. A última linha informada ao sistema contém 3 (três) valores zero (exemplo 0.0 0.0 0.0). Após a leitura de cada linha o programa deve tentar calcular as duas raízes da equação. A listagem de saída, em cada linha, deverá conter os valores das duas raízes reais. Considere que o usuário entrará somente com valores $A$, $B$ e $C$ tais que a equação garantidamente tenha duas raízes reais.

<table>
<thead>
<tr>
<th>Exemplo Entrada</th>
<th>Saída esperada</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.00 -1.00 -6.00</td>
<td>-3.00 2.00</td>
</tr>
<tr>
<td>1.00 0.00 -1.00</td>
<td>-1.00 1.00</td>
</tr>
<tr>
<td>1.00 0.00 0.00</td>
<td>0.00 0.00</td>
</tr>
<tr>
<td>0.00 0.00 0.00</td>
<td></td>
</tr>
</tbody>
</table>

19. Faça um programa em *Pascal* que leia dois números inteiros $N$ e $M$ como entrada e retorne como saída $N \mod M$ (o resto da divisão inteira de $N$ por $M$) usando para isto apenas operações de subtração. O seu programa deve considerar que o usuário entra com $N$ sempre maior do que $M$.

<table>
<thead>
<tr>
<th>Exemplo Entrada</th>
<th>Saída esperada</th>
</tr>
</thead>
<tbody>
<tr>
<td>30 7</td>
<td>2</td>
</tr>
<tr>
<td>3 2</td>
<td>1</td>
</tr>
<tr>
<td>12 3</td>
<td>0</td>
</tr>
</tbody>
</table>
20. Faça um programa em *Pascal* que leia um número $n > 0$ do teclado e imprima a tabuada de $n$ de 1 até 10.

<table>
<thead>
<tr>
<th>Exemplo</th>
<th>Entrada</th>
<th>Saída esperada</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td></td>
<td>5 x 1 = 5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5 x 2 = 10</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5 x 3 = 15</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5 x 4 = 20</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5 x 5 = 25</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5 x 6 = 30</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5 x 7 = 35</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5 x 8 = 40</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5 x 9 = 45</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5 x 10 = 50</td>
</tr>
</tbody>
</table>
5.11 Exercícios complementares

Estes exercícios complementam os anteriores e podem ser feitos por aqueles que querem um reforço nos conceitos deste capítulo que é fundamental para compreensão do restante desta disciplina. Muitos dos problemas aqui propostos são similares, os estudantes podem resolver os problemas até se sentirem confiantes que compreenderam os conceitos básicos de entrada e saída e expressões aritméticas e booleanas.

A maior parte dos problemas pode ser resolvida com base em conceitos básicos de Matemática e Física, mas sempre apresentamos as fórmulas necessárias, pois o que está sendo solicitado são implementações de conceitos fundamentais do ensino básico, que deveriam ser de conhecimento dos alunos.

5.11.1 Programas com cálculos simples

1. Faça um programa em *Pascal* que leia um número inteiro e imprima o seu sucessor e seu antecessor, na mesma linha.

<table>
<thead>
<tr>
<th>Exemplo de entrada</th>
<th>Saída esperada</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2 0</td>
</tr>
<tr>
<td>100</td>
<td>101 99</td>
</tr>
<tr>
<td>-3</td>
<td>-2 -4</td>
</tr>
</tbody>
</table>

2. Faça um programa em *Pascal* que leia dois números inteiros e imprima o resultado da soma destes dois valores. Antes do resultado, deve ser impressa a seguinte mensagem “SOMA=”.

<table>
<thead>
<tr>
<th>Exemplo de entrada</th>
<th>Saída esperada</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2</td>
<td>SOMA= 3</td>
</tr>
<tr>
<td>100 -50</td>
<td>SOMA= 50</td>
</tr>
<tr>
<td>-5 -40</td>
<td>SOMA= -45</td>
</tr>
</tbody>
</table>

3. Faça um programa em *Pascal* que leia dois números reais, um será o valor de um produto e outro o valor de desconto que esse produto está recebendo. Imprima quantos reais o produto custa na promoção.

<table>
<thead>
<tr>
<th>Exemplo de entrada</th>
<th>Saída esperada</th>
</tr>
</thead>
<tbody>
<tr>
<td>Valor original</td>
<td>Desconto</td>
</tr>
<tr>
<td>500.00</td>
<td>50.00</td>
</tr>
<tr>
<td>10500.00</td>
<td>500.00</td>
</tr>
<tr>
<td>90.00</td>
<td>0.80</td>
</tr>
</tbody>
</table>

\[3\] Os enunciados aqui apresentados foram compilados pelo professor David Menotti Gomes e gentilmente cedidos para que constem neste material com o objetivo de servir de prática para os alunos interessados em complementar seu domínio na programação em *Pascal*. Os autores modificaram minimamente o texto para fins de padronização com o restante dos enunciados deste livro. O professor David informou que os exemplos de execução foram adicionados pelo professor Marcelo da Silva, da Universidade Federal de Ouro Preto.
4. Faça um programa em *Pascal* que leia dois números reais e imprima a média
aritmética entre esses dois valores.

<table>
<thead>
<tr>
<th>Exemplo de entrada</th>
<th>Saída esperada</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.2 2.3</td>
<td>1.75</td>
</tr>
<tr>
<td>750 1500</td>
<td>1125.00</td>
</tr>
<tr>
<td>8900 12300</td>
<td>10600.00</td>
</tr>
</tbody>
</table>

5. Faça um programa em *Pascal* que leia um número real e imprima a terceira parte
deste número.

<table>
<thead>
<tr>
<th>Exemplo de entrada</th>
<th>Saída esperada</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>1.00</td>
</tr>
<tr>
<td>10</td>
<td>3.33</td>
</tr>
<tr>
<td>90</td>
<td>30.00</td>
</tr>
</tbody>
</table>

6. Uma P.A. (progressão aritmética) é determinada pela sua razão (*r*) e pelo pri-
meiro termo (*a*<sub>1</sub>). Faça um programa em *Pascal* que seja capaz de determinar o
enésimo (*n*) termo (*a*<sub>n</sub>) de uma P.A., dado a razão (*r*) e o primeiro termo (*a*<sub>1</sub>).
Seu programa deve ler *n*, *r*, *a*<sub>1</sub> do teclado e imprimir *a*<sub>n</sub>.

\[ a_n = a_1 + (n - 1) \times r. \]

<table>
<thead>
<tr>
<th>Exemplo de entrada</th>
<th>Saída esperada</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>n</em></td>
<td><em>r</em></td>
</tr>
<tr>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>100</td>
<td>10</td>
</tr>
<tr>
<td>5</td>
<td>-2</td>
</tr>
</tbody>
</table>

7. Dada a razão (*r*) de uma P.A. (progressão aritmética) e um termo qualquer, *k*
(*a*<sub>*k*</sub>). Faça um programa em *Pascal* que calcule o enésimo termo *n* (*a*<sub>*n*</sub>). Seu
programa deve ler *k*, *a*<sub>*k*</sub>, *r*, *n* do teclado e imprimir *a*<sub>*n*</sub>.

\[ a_n = a_k + (n - r) \times r \]

<table>
<thead>
<tr>
<th>Exemplo de entrada</th>
<th>Saída esperada</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>k</em></td>
<td><em>a</em>&lt;sub&gt;<em>k</em>&lt;/sub&gt;</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>10</td>
<td>20</td>
</tr>
<tr>
<td>100</td>
<td>500</td>
</tr>
</tbody>
</table>
8. Uma P.G. (progressão geométrica) é determinada pela sua razão ($q$) e pelo primeiro termo ($a_1$). Faça um programa em *Pascal* que seja capaz de determinar o enésimo $n$ termo ($a_n$) de uma P.G., dado a razão ($q$) e o primeiro termo ($a_1$). Seu programa deve ler $a_1, q, n$ do teclado e imprimir $a_n$.

$$a_n = a_1 \times q^{(n-1)}.$$ 

<table>
<thead>
<tr>
<th>Exemplo de entrada</th>
<th>Saída esperada</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a_1$ $q$ $n$</td>
<td>$a_n$</td>
</tr>
<tr>
<td>1 1 100</td>
<td>1.00</td>
</tr>
<tr>
<td>2 2 10</td>
<td>1024.00</td>
</tr>
<tr>
<td>5 3 2</td>
<td>15.00</td>
</tr>
</tbody>
</table>

9. Dada a razão ($q$) de uma P.G. (progressão geométrica) e um termo qualquer, $k$ ($a_k$). Faça um programa em *Pascal* para calcular o enésimo termo $n$ ($a_n$). Seu programa deve ser $k, a_k, q, n$ do teclado e imprimir $a_n$.

<table>
<thead>
<tr>
<th>Exemplo de entrada</th>
<th>Saída esperada</th>
</tr>
</thead>
<tbody>
<tr>
<td>$k$ $a_k$ $q$ $n$</td>
<td>$a_n$</td>
</tr>
<tr>
<td>2 2 1 1</td>
<td>2</td>
</tr>
<tr>
<td>1 5 2 10</td>
<td>2560.00</td>
</tr>
<tr>
<td>2 100 10 20</td>
<td>100000000000000000000.00</td>
</tr>
</tbody>
</table>

10. Uma P.G. (progressão geométrica) é determinada pela sua razão ($q$) e pelo primeiro termo ($a_1$). Faça um programa em *Pascal* que seja capaz de determinar o enésimo termo ($a_n$) de uma P.G., dado a razão ($q$) e o primeiro termo ($a_1$). Seu programa deve ler $a_1, q, n$ do teclado e imprimir $a_n$.

<table>
<thead>
<tr>
<th>Exemplo de entrada</th>
<th>Saída esperada</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a_1$ $q$ $n$</td>
<td>$a_n$</td>
</tr>
<tr>
<td>1 1 100</td>
<td>1.00</td>
</tr>
<tr>
<td>2 2 10</td>
<td>1024.00</td>
</tr>
<tr>
<td>10 2 20</td>
<td>5242880.00</td>
</tr>
</tbody>
</table>

11. Considere que o número de uma placa de veículo é composto por quatro algarismos. Faça um programa em *Pascal* que leia este número do teclado e apresente o algarismo correspondente à casa das unidades.

<table>
<thead>
<tr>
<th>Exemplo de entrada</th>
<th>Saída esperada</th>
</tr>
</thead>
<tbody>
<tr>
<td>2569</td>
<td>9</td>
</tr>
<tr>
<td>1000</td>
<td>0</td>
</tr>
<tr>
<td>1305</td>
<td>5</td>
</tr>
</tbody>
</table>
12. Considere que o número de uma placa de veículo é composto por quatro algarismos. Faça um programa em *Pascal* que leia este número do teclado e apresente o algarismo correspondente à casa das dezenas.

<table>
<thead>
<tr>
<th>Exemplo de entrada</th>
<th>Saída esperada</th>
</tr>
</thead>
<tbody>
<tr>
<td>2569</td>
<td>6</td>
</tr>
<tr>
<td>1000</td>
<td>0</td>
</tr>
<tr>
<td>1350</td>
<td>5</td>
</tr>
</tbody>
</table>

13. Considere que o número de uma placa de veículo é composto por quatro algarismos. Faça um programa em *Pascal* que leia este número do teclado e apresente o algarismo correspondente à casa das centenas.

<table>
<thead>
<tr>
<th>Exemplo de entrada</th>
<th>Saída esperada</th>
</tr>
</thead>
<tbody>
<tr>
<td>2500</td>
<td>5</td>
</tr>
<tr>
<td>2031</td>
<td>0</td>
</tr>
<tr>
<td>6975</td>
<td>9</td>
</tr>
</tbody>
</table>

14. Considere que o número de uma placa de veículo é composto por quatro algarismos. Faça um programa em *Pascal* que leia este número do teclado e apresente o algarismo correspondente à casa do milhar.

<table>
<thead>
<tr>
<th>Exemplo de entrada</th>
<th>Saída esperada</th>
</tr>
</thead>
<tbody>
<tr>
<td>2569</td>
<td>2</td>
</tr>
<tr>
<td>1000</td>
<td>1</td>
</tr>
<tr>
<td>0350</td>
<td>0</td>
</tr>
</tbody>
</table>

15. Você é um vendedor de carros é só aceita pagamentos à vista. As vezes é necessário ter que dar troco, mas seus clientes não gostam de notas miúdas. Para agradá-los você deve fazer um programa em *Pascal* que receba o valor do troco que deve ser dado ao cliente e retorna o número de notas de R$100 necessárias para esse troco.

<table>
<thead>
<tr>
<th>Exemplo de entrada</th>
<th>Saída esperada</th>
</tr>
</thead>
<tbody>
<tr>
<td>500</td>
<td>5</td>
</tr>
<tr>
<td>360</td>
<td>3</td>
</tr>
<tr>
<td>958</td>
<td>9</td>
</tr>
</tbody>
</table>

16. Certo dia o professor de Johann Friederich Carl Gauss (aos 10 anos de idade) mandou que os alunos somassem os números de 1 a 100. Imediatamente Gauss achou a resposta – 5050 – aparentemente sem a soma de um em um. Supõe-se que já aí, Gauss, houvesse descoberto a fórmula de uma soma de uma progressão aritmética.

Faça um programa em *Pascal* que realize a soma de uma P.A. de *n* termos, dado o primeiro termo *a1* e o último termo *an*. A impressão do resultado deve ser formatada com duas casas na direita.
17. A sequência $A, B, C, \ldots$ determina uma Progressão Aritmética (P.A.). O termo médio ($B$) de uma P.A. é determinado pela média aritmética de seus termos, sucessor ($C$) e antecessor ($A$). Com base neste enunciado construa um programa em Pascal que calcule e imprima o termo médio ($B$) através de $A$ e $C$, que devem ser lidos do teclado.

$$B = \frac{A + B}{2}.$$ 

<table>
<thead>
<tr>
<th>Exemplo de entrada</th>
<th>Saída esperada</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n$</td>
<td>$a_1$</td>
</tr>
<tr>
<td>100</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>50</td>
<td>30</td>
</tr>
</tbody>
</table>

18. A sequência $A, B, C, \ldots$ determina uma Progressão Geométrica (P.G.), o termo médio ($B$) de uma P.G. é determinado pela média geométrica de seus termos, sucessor ($C$) e antecessor ($A$). Com base neste enunciado escreva um programa em Pascal que calcule e imprima o termo médio ($B$) através de $A$, $C$, que devem ser lidos do teclado.

<table>
<thead>
<tr>
<th>Exemplo de entrada</th>
<th>Saída esperada</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A$</td>
<td>$C$</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>100</td>
<td>500</td>
</tr>
</tbody>
</table>

19. O produto de uma série de termos de uma Progressão Geométrica (P.G.) pode ser calculado pela fórmula abaixo:

$$P = a_1^n \times q^{\frac{n(n-1)}{2}}.$$ 

Agora, faça um programa em Pascal para determinar o produto dos $n$ primeiros termos de uma P.G de razão $q$. Seu programa deverá ler $a_1$, $q$, $n$ do teclado e imprimir $P$. (ATENÇÃO PARA O TIPO DE VARIÁVEL!)
5.11. EXERCÍCIOS COMPLEMENTARES

<table>
<thead>
<tr>
<th>Exemplo de entrada</th>
<th>Saída esperada</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a_1$</td>
<td>$q$</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

20. A soma dos termos de uma Progressão Geométrica (P.G.) finita pode ser calculada pela fórmula abaixo:

$$S_n = \frac{a_1(q^n - 1)}{q - 1}$$

Agora, faça um programa em *Pascal* para determinar a soma dos $n$ termos de uma P.G de razão $q$, iniciando no termo $a_1$. Seu programa deverá ler $a_1, q, n$ do teclado e imprimir $S_n$.

<table>
<thead>
<tr>
<th>Exemplo de entrada</th>
<th>Saída esperada</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a_1$</td>
<td>$q$</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>150</td>
<td>30</td>
</tr>
</tbody>
</table>

21. Faça um programa em *Pascal* para calcular e imprimir o valor do volume de uma lata de óleo, utilizando a fórmula:

$$V = 3.14159 \times r^2 \times h,$$

onde $V$ é o volume, $r$ é o raio e $h$ é a altura. Seu programa deve ler $r, h$ do teclado e imprimir $V$.

<table>
<thead>
<tr>
<th>Exemplo de entrada</th>
<th>Saída esperada</th>
</tr>
</thead>
<tbody>
<tr>
<td>$r$</td>
<td>$h$</td>
</tr>
<tr>
<td>5</td>
<td>100</td>
</tr>
<tr>
<td>25</td>
<td>25.5</td>
</tr>
<tr>
<td>10</td>
<td>50.9</td>
</tr>
</tbody>
</table>

22. Faça um programa em *Pascal* que efetue o cálculo do salário líquido de um professor. Os dados fornecidos serão: valor da hora aula, número de aulas dadas no mês e percentual de desconto do INSS.

<table>
<thead>
<tr>
<th>Exemplo de entrada</th>
<th>Saída esperada</th>
</tr>
</thead>
<tbody>
<tr>
<td>valor hora aula</td>
<td>número de aulas</td>
</tr>
<tr>
<td>6.25</td>
<td>160</td>
</tr>
<tr>
<td>20.5</td>
<td>240</td>
</tr>
<tr>
<td>13.9</td>
<td>200</td>
</tr>
</tbody>
</table>
23. Em épocas de pouco dinheiro, os comerciantes estão procurando aumentar suas vendas oferecendo desconto aos clientes. Faça um programa em *Pascal* que possa entrar com o valor de um produto e imprima o novo valor tendo em vista que o desconto foi de 9%. Além disso, imprima o valor do desconto.

<table>
<thead>
<tr>
<th>Exemplo de entrada</th>
<th>Saída esperada</th>
</tr>
</thead>
<tbody>
<tr>
<td>valor do produto (R$)</td>
<td>novo valor (R$)</td>
</tr>
<tr>
<td>100</td>
<td>91.00</td>
</tr>
<tr>
<td>1500</td>
<td>1365.00</td>
</tr>
<tr>
<td>60000</td>
<td>54600.00</td>
</tr>
</tbody>
</table>

24. Todo restaurante, embora por lei não possa obrigar o cliente a pagar, cobra 10% de comissão para o garçom. Faça um programa em *Pascal* que leia o valor gasto com despesas realizadas em um restaurante e imprima o valor da gorjeta e o valor total com a gorjeta.

<table>
<thead>
<tr>
<th>Exemplo de entrada</th>
<th>Saída esperada</th>
</tr>
</thead>
<tbody>
<tr>
<td>75</td>
<td>82.50</td>
</tr>
<tr>
<td>125</td>
<td>137.50</td>
</tr>
<tr>
<td>350.87</td>
<td>385.96</td>
</tr>
</tbody>
</table>

25. Faça um programa em *Pascal* que leia um valor de hora (hora:minutos), calcule e imprima o total de minutos se passaram desde o início do dia (0:00h). A entrada será dada por dois números separados na mesma linha, o primeiro número representa as horas e o segundo os minutos.

<table>
<thead>
<tr>
<th>Exemplo de entrada</th>
<th>Saída esperada</th>
</tr>
</thead>
<tbody>
<tr>
<td>hora</td>
<td>minuto</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>14</td>
<td>30</td>
</tr>
<tr>
<td>23</td>
<td>55</td>
</tr>
</tbody>
</table>

26. Faça um programa em *Pascal* que leia o valor de um depósito e o valor da taxa de juros. Calcular e imprimir o valor do rendimento do depósito e o valor total depois do rendimento.

<table>
<thead>
<tr>
<th>Exemplo de entrada</th>
<th>Saída esperada</th>
</tr>
</thead>
<tbody>
<tr>
<td>depósito</td>
<td>taxa de juros</td>
</tr>
<tr>
<td>200</td>
<td>0.5</td>
</tr>
<tr>
<td>1050</td>
<td>1</td>
</tr>
<tr>
<td>2300.38</td>
<td>0.06</td>
</tr>
</tbody>
</table>

27. Para vários tributos, a base de cálculo é o salário mínimo. Faça um programa em *Pascal* que leia o valor do salário mínimo e o valor do salário de uma pessoa. Calcular e imprimir quantos salários mínimos essa pessoa ganha.
28. Faça um programa em *Pascal* que efetue o cálculo da quantidade de litros de combustível gastos em uma viagem, sabendo-se que o carro faz 12 km com um litro. Deverão ser fornecidos o tempo gasto na viagem e a velocidade média. 

\[ \text{Distância} = \text{Tempo} \times \text{Velocidade}. \]

\[ \text{Litros} = \frac{\text{Distância}}{12}. \] 

O algoritmo deverá apresentar os valores da Distância percorrida e a quantidade de Litros utilizados na viagem.

<table>
<thead>
<tr>
<th>Exemplo de entrada</th>
<th>Saída esperada</th>
</tr>
</thead>
<tbody>
<tr>
<td>tempo gasto</td>
<td>velocidade média</td>
</tr>
<tr>
<td>60</td>
<td>100</td>
</tr>
<tr>
<td>1440</td>
<td>80</td>
</tr>
<tr>
<td>5</td>
<td>90</td>
</tr>
</tbody>
</table>

29. Um vendedor de uma loja de sapatos recebe como pagamento 20\% de comissão sobre as vendas do mês e R$5.00 por cada par de sapatos vendidos. Faça um programa em *Pascal* que, dado o total de vendas do mês e o número de sapatos vendidos, imprima quanto será o salário daquele mês do vendedor.

<table>
<thead>
<tr>
<th>Exemplo de entrada</th>
<th>Saída esperada</th>
</tr>
</thead>
<tbody>
<tr>
<td>total de vendas (R$)</td>
<td>sapatos vendidos</td>
</tr>
<tr>
<td>50000.00</td>
<td>100</td>
</tr>
<tr>
<td>2000.00</td>
<td>30</td>
</tr>
<tr>
<td>1000000.00</td>
<td>500</td>
</tr>
</tbody>
</table>

30. Você está endividado e quer administrar melhor sua vida financeira. Para isso, faça um programa em *Pascal* que recebe o valor de uma dívida e o juros mensal, então calcule e imprima o valor da dívida no mês seguinte.

<table>
<thead>
<tr>
<th>Exemplo de entrada</th>
<th>Saída esperada</th>
</tr>
</thead>
<tbody>
<tr>
<td>valor da dívida (R$)</td>
<td>juros/mês</td>
</tr>
<tr>
<td>100.00</td>
<td>10</td>
</tr>
<tr>
<td>1500.00</td>
<td>3</td>
</tr>
<tr>
<td>10000.00</td>
<td>0.5</td>
</tr>
</tbody>
</table>

31. Antes de o racionamento de energia ser decretado, quase ninguém falava em quilowatts; mas, agora, todos incorporaram essa palavra em seu vocabulário. Sabendo-se que 100 quilowatts de energia custa um sétimo do salário mínimo, faça um programa em *Pascal* que receba o valor do salário mínimo e a quantidade de quilowatts gasta por uma residência e imprima:
• o valor em reais de cada quilowatt;
• o valor em reais a ser pago;

<table>
<thead>
<tr>
<th>Exemplo de entrada</th>
<th>Saída esperada</th>
</tr>
</thead>
<tbody>
<tr>
<td>salário mínimo (R$)</td>
<td>quilowatts</td>
</tr>
<tr>
<td>750.00</td>
<td>200</td>
</tr>
<tr>
<td>935.00</td>
<td>150</td>
</tr>
<tr>
<td>1200.00</td>
<td>250</td>
</tr>
</tbody>
</table>

5.11.2 Programas com cálculos e desvios condicionais

1. Faça um programa em *Pascal* que leia um número e o imprima caso ele seja maior que 20.

<table>
<thead>
<tr>
<th>Exemplo Entrada</th>
<th>Saída esperada</th>
</tr>
</thead>
<tbody>
<tr>
<td>30.56</td>
<td>30.56</td>
</tr>
<tr>
<td>20</td>
<td></td>
</tr>
<tr>
<td>20.05</td>
<td>20.05</td>
</tr>
</tbody>
</table>

2. Faça um programa em *Pascal* que leia dois valores numéricos inteiros e efetue a adição; se o resultado for maior que 10, imprima o primeiro valor. Caso contrário, imprima o segundo.

<table>
<thead>
<tr>
<th>Exemplo Entrada</th>
<th>Saída esperada</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>7</td>
</tr>
</tbody>
</table>

3. Faça um programa em *Pascal* que imprima se um dado número $N$ inteiro (recebido através do teclado) é PAR ou ÍMPAR.

<table>
<thead>
<tr>
<th>Exemplo Entrada</th>
<th>Saída esperada</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>impar</td>
</tr>
<tr>
<td>3</td>
<td>impar</td>
</tr>
<tr>
<td>2</td>
<td>par</td>
</tr>
</tbody>
</table>

4. Faça um programa em *Pascal* para determinar se um dado número $N$ (recebido através do teclado) é POSITIVO, NEGATIVO ou NULO.
5.11. **EXERCÍCIOS COMPLEMENTARES**

<table>
<thead>
<tr>
<th>Exemplo Entrada</th>
<th>Saída esperada</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>positivo</td>
</tr>
<tr>
<td>-3</td>
<td>negativo</td>
</tr>
<tr>
<td>0</td>
<td>nulo</td>
</tr>
</tbody>
</table>

5. Faça um programa em *Pascal* que leia dois números e efetue a adição. Caso o valor somado seja maior que 20, este deverá ser apresentado somando-se a ele mais 8; caso o valor somado seja menor ou igual a 20, este deverá ser apresentado subtraindo-se 5.

<table>
<thead>
<tr>
<th>Exemplo Entrada</th>
<th>Saída esperada</th>
</tr>
</thead>
<tbody>
<tr>
<td>13.14</td>
<td>13.14</td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>-3</td>
<td>-12.00</td>
</tr>
<tr>
<td>-4</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>20.00</td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>

6. Faça um programa em *Pascal* que imprima qual o menor valor de dois números *A* e *B*, lidos através do teclado.

<table>
<thead>
<tr>
<th>Exemplo Entrada</th>
<th>Saída esperada</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.35</td>
<td>4.00</td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>-3</td>
<td>-3.00</td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>6.00</td>
</tr>
<tr>
<td>15</td>
<td></td>
</tr>
</tbody>
</table>

7. Faça um programa em *Pascal* para determinar se um número inteiro *A* é divisível por um outro número inteiro *B*. Esses valores devem ser fornecidos pelo usuário.

<table>
<thead>
<tr>
<th>Exemplo Entrada</th>
<th>Saída esperada</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>nao</td>
</tr>
<tr>
<td>10</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>sim</td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>nao</td>
</tr>
<tr>
<td>21</td>
<td></td>
</tr>
</tbody>
</table>

8. Faça um programa em *Pascal* que leia um número inteiro e informe se ele é ou não divisível por 5.
<table>
<thead>
<tr>
<th>Exemplo Entrada</th>
<th>Saída esperada</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>sim</td>
</tr>
<tr>
<td>-5</td>
<td>sim</td>
</tr>
<tr>
<td>3</td>
<td>não</td>
</tr>
</tbody>
</table>

9. Faça um programa em *Pascal* que receba um número inteiro e imprima se este é múltiplo de 3.

<table>
<thead>
<tr>
<th>Exemplo Entrada</th>
<th>Saída esperada</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>não</td>
</tr>
<tr>
<td>-3</td>
<td>sim</td>
</tr>
<tr>
<td>15</td>
<td>sim</td>
</tr>
</tbody>
</table>

10. Faça um programa em *Pascal* que leia um número e imprima a raiz quadrada do número caso ele seja positivo ou igual a zero e o quadrado do número caso ele seja negativo.

<table>
<thead>
<tr>
<th>Exemplo Entrada</th>
<th>Saída esperada</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.00</td>
</tr>
<tr>
<td>4</td>
<td>2.00</td>
</tr>
<tr>
<td>-5</td>
<td>25.00</td>
</tr>
</tbody>
</table>

11. Faça um programa em *Pascal* que leia um número e informe se ele é divisível por 3 e por 7.

<table>
<thead>
<tr>
<th>Exemplo Entrada</th>
<th>Saída esperada</th>
</tr>
</thead>
<tbody>
<tr>
<td>21</td>
<td>sim</td>
</tr>
<tr>
<td>7</td>
<td>não</td>
</tr>
<tr>
<td>3</td>
<td>não</td>
</tr>
<tr>
<td>-42</td>
<td>sim</td>
</tr>
</tbody>
</table>

12. A prefeitura de Contagem abriu uma linha de crédito para os funcionários estatutários. O valor máximo da prestação não poderá ultrapassar 30% do salário bruto. Faça um programa em *Pascal* que permita entrar com o salário bruto e o valor da prestação, e informar se o empréstimo pode ou não ser concedido.

<table>
<thead>
<tr>
<th>Exemplo Entrada</th>
<th>Saída esperada</th>
</tr>
</thead>
<tbody>
<tr>
<td>500</td>
<td>não</td>
</tr>
<tr>
<td>200</td>
<td></td>
</tr>
<tr>
<td>1000.50</td>
<td></td>
</tr>
<tr>
<td>250.10</td>
<td>sim</td>
</tr>
<tr>
<td>1000</td>
<td></td>
</tr>
<tr>
<td>300</td>
<td>sim</td>
</tr>
</tbody>
</table>
13. Faça um programa em *Pascal* que dado quatro valores, \(A, B, C\) e \(D\), o programa imprima o menor e o maior valor.

<table>
<thead>
<tr>
<th>Exemplo Entrada</th>
<th>Saída esperada</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.00 4.00</td>
</tr>
<tr>
<td>2</td>
<td>1.00 4.00</td>
</tr>
<tr>
<td>3</td>
<td>1.00 4.00</td>
</tr>
<tr>
<td>4</td>
<td>1.00 4.00</td>
</tr>
<tr>
<td>-3</td>
<td>-3.00 1.00</td>
</tr>
<tr>
<td>0</td>
<td>-3.00 1.00</td>
</tr>
<tr>
<td>1</td>
<td>-3.00 1.00</td>
</tr>
<tr>
<td>1</td>
<td>-3.00 1.00</td>
</tr>
<tr>
<td>3.5</td>
<td>3.50 5.50</td>
</tr>
<tr>
<td>3.7</td>
<td>3.50 5.50</td>
</tr>
<tr>
<td>4.0</td>
<td>3.50 5.50</td>
</tr>
<tr>
<td>5.5</td>
<td>3.50 5.50</td>
</tr>
</tbody>
</table>

14. Dados três valores \(A, B\) e \(C\), faça um programa em *Pascal*, que imprima os valores de forma ascendente (do menor para o maior).

<table>
<thead>
<tr>
<th>Exemplo Entrada</th>
<th>Saída esperada</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2 1.5</td>
<td>1.00 1.50 2.00</td>
</tr>
<tr>
<td>-3 -4 -5</td>
<td>-5.00 -4.00 -3.00</td>
</tr>
<tr>
<td>6 5 4</td>
<td>4.00 5.00 6.00</td>
</tr>
</tbody>
</table>

15. Dados três valores \(A, B\) e \(C\), faça um programa em *Pascal*, que imprima os valores de forma descendente (do maior para o menor).

<table>
<thead>
<tr>
<th>Exemplo Entrada</th>
<th>Saída esperada</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2 1.5</td>
<td>2.00 1.50 1.00</td>
</tr>
<tr>
<td>-5 -4 -3</td>
<td>-3.00 -4.00 -5.00</td>
</tr>
<tr>
<td>5 6 4</td>
<td>6.00 5.00 4.00</td>
</tr>
</tbody>
</table>

16. Faça um programa em *Pascal* que leia dois números e imprimir o quadrado do menor número e raiz quadrada do maior número, se for possível.

<table>
<thead>
<tr>
<th>Exemplo Entrada</th>
<th>Saída esperada</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>9.00 2.00</td>
</tr>
<tr>
<td>3</td>
<td>9.00 2.00</td>
</tr>
<tr>
<td>4.35</td>
<td>4.35 2.09</td>
</tr>
<tr>
<td>3.50</td>
<td>12.25 2.09</td>
</tr>
<tr>
<td>-4</td>
<td>256.00</td>
</tr>
<tr>
<td>-16</td>
<td>256.00</td>
</tr>
</tbody>
</table>
17. Faça um programa em *Pascal* que indique se um número digitado está compreendido entre 20 e 90 ou não (20 e 90 não estão na faixa de valores).

<table>
<thead>
<tr>
<th>Exemplo Entrada</th>
<th>Saída esperada</th>
</tr>
</thead>
<tbody>
<tr>
<td>50.50</td>
<td>sim</td>
</tr>
<tr>
<td>20</td>
<td>não</td>
</tr>
<tr>
<td>90</td>
<td>não</td>
</tr>
</tbody>
</table>

18. Faça um programa em *Pascal* que leia um número inteiro e informe se ele é divisível por 10, por 5 ou por 2 ou se não é divisível por nenhum deles.

<table>
<thead>
<tr>
<th>Exemplo Entrada</th>
<th>Saída esperada</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>10 5 2</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>nenhum</td>
</tr>
</tbody>
</table>

19. Faça um programa em *Pascal* que leia um número e imprima se ele é igual a 5, a 200, a 400, se está no intervalo entre 500 e 1000, inclusive, ou se está fora dos escopos anteriores.

<table>
<thead>
<tr>
<th>Exemplo Entrada</th>
<th>Saída esperada</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>igual a 5</td>
</tr>
<tr>
<td>200</td>
<td>igual a 200</td>
</tr>
<tr>
<td>400</td>
<td>igual a 400</td>
</tr>
<tr>
<td>750.50</td>
<td>intervalo entre 500 e 1000</td>
</tr>
<tr>
<td>1000</td>
<td>intervalo entre 500 e 1000</td>
</tr>
<tr>
<td>1500</td>
<td>fora dos escopos</td>
</tr>
</tbody>
</table>

20. A CEF concederá um crédito especial com juros de 2% aos seus clientes de acordo com o saldo médio no último ano. Faça um programa em *Pascal* que leia o saldo médio de um cliente e calcule o valor do crédito de acordo com a tabela a seguir. Imprimir uma mensagem informando o valor de crédito.

<table>
<thead>
<tr>
<th>De 0 a 500</th>
<th>nenhum crédito</th>
</tr>
</thead>
<tbody>
<tr>
<td>De 501 a 1000</td>
<td>30% do valor do saldo medio</td>
</tr>
<tr>
<td>De 1001 a 3000</td>
<td>40% do valor do saldo medio</td>
</tr>
<tr>
<td>Acima de 3001</td>
<td>50% do valor do saldo medio</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Exemplo Entrada</th>
<th>Saída esperada</th>
</tr>
</thead>
<tbody>
<tr>
<td>300.50</td>
<td>0.00</td>
</tr>
<tr>
<td>571</td>
<td>171.30</td>
</tr>
<tr>
<td>1492.35</td>
<td>596.94</td>
</tr>
<tr>
<td>3001.20</td>
<td>1500.60</td>
</tr>
</tbody>
</table>
21. Faça um programa em *Pascal* que, dada a idade de uma pessoa, determine sua classificação segundo a seguinte tabela:

- Maior de idade;
- Menor de idade;
- Pessoa idosa (idade superior ou igual a 65 anos).

<table>
<thead>
<tr>
<th>Exemplo Entrada</th>
<th>Saída esperada</th>
</tr>
</thead>
<tbody>
<tr>
<td>18</td>
<td>maior</td>
</tr>
<tr>
<td>15</td>
<td>menor</td>
</tr>
<tr>
<td>65</td>
<td>idosa</td>
</tr>
</tbody>
</table>

22. Faça um programa em *Pascal* que leia a idade de uma pessoa e informe a sua classe eleitoral:

- não eleitor (abaixo de 16 anos);
- eleitor obrigatório (entre a faixa de 18 e menor de 65 anos);
- eleitor facultativo (de 16 até 18 anos e maior de 65 anos, inclusive).

<table>
<thead>
<tr>
<th>Exemplo Entrada</th>
<th>Saída esperada</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>não eleitor</td>
</tr>
<tr>
<td>16</td>
<td>facultativo</td>
</tr>
<tr>
<td>17</td>
<td>facultativo</td>
</tr>
<tr>
<td>18</td>
<td>obrigatorio</td>
</tr>
<tr>
<td>19</td>
<td>obrigatorio</td>
</tr>
</tbody>
</table>

23. A confederação brasileira de natação irá promover eliminatórias para o próximo mundial. Faça um programa em *Pascal* que receba a idade de um nadador e imprima a sua categoria segundo a tabela a seguir:

<table>
<thead>
<tr>
<th>Classe</th>
<th>Faixa de Idade</th>
</tr>
</thead>
<tbody>
<tr>
<td>Infantil A</td>
<td>5 – 7 anos</td>
</tr>
<tr>
<td>Infantil B</td>
<td>8 – 10 anos</td>
</tr>
<tr>
<td>Juvenil A</td>
<td>11 – 13 anos</td>
</tr>
<tr>
<td>Juvenil B</td>
<td>14 – 17 anos</td>
</tr>
<tr>
<td>Sênior</td>
<td>Maiores de 18 anos</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Exemplo Entrada</th>
<th>Saída esperada</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>INVALIDO</td>
</tr>
<tr>
<td>7</td>
<td>Infantil A</td>
</tr>
<tr>
<td>8</td>
<td>Infantil B</td>
</tr>
<tr>
<td>10</td>
<td>Infantil B</td>
</tr>
<tr>
<td>11</td>
<td>Juvenil A</td>
</tr>
<tr>
<td>13</td>
<td>Juvenil A</td>
</tr>
<tr>
<td>14</td>
<td>Juvenil B</td>
</tr>
<tr>
<td>17</td>
<td>Juvenil B</td>
</tr>
<tr>
<td>18</td>
<td>Sênior</td>
</tr>
</tbody>
</table>
24. Dados três valores $A$, $B$ e $C$, faça um programa em *Pascal* para verificar se estes valores podem ser valores dos lados de um triângulo, se é um triângulo ESCALENO, um triângulo EQUILÁTERO ou um triângulo ISÓSCELES. Caso não sejam válidos, imprimir: “INVALIDO”.

<table>
<thead>
<tr>
<th>Exemplo Entrada</th>
<th>Saída esperada</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>EQUILATERO</td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>ISOSCELES</td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>ESCALENO</td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>INVALIDO</td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td></td>
</tr>
</tbody>
</table>

25. Faça um programa em *Pascal* que leia as duas notas bimestrais de um aluno e determine a média das notas semestral. Através da média calculada o algoritmo deve imprimir a seguinte mensagem: APROVADO, REPROVADO ou em EXAME (a média é 7 para Aprovação, menor que 3 para Reprovação e as demais em Exame).

<table>
<thead>
<tr>
<th>Exemplo Entrada</th>
<th>Saída esperada</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>REPROVADO</td>
</tr>
<tr>
<td>2.5</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>EXAME</td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>EXAME</td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>APROVADO</td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>APROVADO</td>
</tr>
<tr>
<td>10</td>
<td></td>
</tr>
</tbody>
</table>

26. Depois da liberação do governo para as mensalidades dos planos de saúde, as pessoas começaram a fazer pesquisas para descobrir um bom plano, não muito caro. Um vendedor de um plano de saúde apresentou a tabela a seguir. Faça um programa em *Pascal* que entre com a idade de uma pessoa e imprima o valor que ela deverá pagar, segundo a seguinte tabela:
<table>
<thead>
<tr>
<th>Até 10 anos</th>
<th>R$ 30.00</th>
</tr>
</thead>
<tbody>
<tr>
<td>Acima de 10 até 29 anos</td>
<td>R$ 60.00</td>
</tr>
<tr>
<td>Acima de 29 até 45 anos</td>
<td>R$ 120.00</td>
</tr>
<tr>
<td>Acima de 45 até 59 anos</td>
<td>R$ 150.00</td>
</tr>
<tr>
<td>Acima de 59 até 65 anos</td>
<td>R$ 250.00</td>
</tr>
<tr>
<td>Maior do que 65 anos</td>
<td>R$ 400.00</td>
</tr>
</tbody>
</table>

10 R$ 30,00
29 R$ 60,00
45 R$ 120,00
59 R$ 150,00
65 R$ 250,00
66 R$ 400,00

27. Faça um programa em *Pascal* que leia um número inteiro entre 1 e 7 e escreva o dia da semana correspondente. Caso o usuário digite um número fora desse intervalo, deverá aparecer a seguinte mensagem: **INEXISTENTE**

<table>
<thead>
<tr>
<th>Exemplo Entrada</th>
<th>Saída esperada</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>INEXISTENTE</td>
</tr>
<tr>
<td>1</td>
<td>DOMINGO</td>
</tr>
<tr>
<td>2</td>
<td>SEGUNDA</td>
</tr>
<tr>
<td>3</td>
<td>TERÇA</td>
</tr>
<tr>
<td>4</td>
<td>QUARTA</td>
</tr>
<tr>
<td>5</td>
<td>QUINTA</td>
</tr>
<tr>
<td>6</td>
<td>SEXTA</td>
</tr>
<tr>
<td>7</td>
<td>SABADO</td>
</tr>
<tr>
<td>8</td>
<td>INEXISTENTE</td>
</tr>
</tbody>
</table>
Capítulo 6

Técnicas elementares

Toda solução computacional para um dado problema é baseada no correto arranjo dos conceitos básicos vistos no capítulo anterior. Mas eles não são suficientes sem que se aprendam técnicas elementares baseadas em um conceito fundamental em computação: a lógica de programação.

A arte de se conceber soluções computacionais depende fortemente da compreensão de como o computador funciona, de como é a estrutura de memória, de saber quais são suas limitações. Existe uma lógica por detrás de todo programa.

Um outro problema diz respeito às limitações das linguagens de programação. Exatamente porquê as gramáticas são extremamente rígidas, ocorre que as linguagens de programação são menos expressivas do que as linguagens naturais, como o português por exemplo.

Escrever algoritmos, e consequentemente programas, exige que o programador saiba explorar esta linguagem, digamos, mais pobre (comparada com as linguagens naturais) e neste sentido a lógica de programação é fundamental.

Fazendo uma analogia com carros, saber que pisar no acelerador faz o carro andar mais rápido e que pisar no freio faz o carro desacelerar não implica em ser um bom motorista, muito menos um excelente piloto de fórmula 1.

O objetivo deste capítulo é o domínio das técnicas elementares que possibilitarão posteriormente o desenvolvimento de algoritmos sofisticados.

6.1 Lógica de programação

Um computador pode ser definido como uma máquina que recebe uma entrada de dados, realiza um processamento deles e produz uma saída que é a solução do problema. Como sabemos, não existe computação sem um problema para ser resolvido.

O processamento dos dados é a parte que cabe ao programador: ele deve elaborar um conjunto de instruções que manipulam os dados que estão em memória. Para isto é preciso compreender como é o processo de manipulação do ponto de vista da estrutura do computador.

Tomemos um problema bem simples: queremos que o usuário digite alguns números inteiros, quantos ele queira, e ao final devemos imprimir quantos números foram digitados.
Aqui já temos um problema: como o computador saberá que o usuário terminou de digitar números? É preciso estabelecer uma espécie de contrato com este usuário para que ele informe que não quer mais digitar números. Então fica estabelecido que quando ele não mais quiser digitar números, vai digitar um zero. Este zero não deve contar como um número válido, ele só serve para terminar a entrada de dados. Logo, após a digitação do zero o programa deve informar quantos números o usuário tinha digitado antes.

O que devemos pensar antes de tentar escrever o programa?

- O computador possui uma memória RAM com muitos espaços para armazenamento, de fato hoje em dia, início do ano 2020, existem computadores com 1 terabyte de RAM;
- O programador só tem acesso à RAM através de variáveis;
- As variáveis devem ser declaradas enquanto o programador está digitando código no seu editor de textos.

A primeira questão que surge é:
Se o programador não sabe quantas informações o usuário vai digitar, como saber a quantidade certa de variáveis que ele precisa declarar no cabeçalho do programa?

A experiência sugere que esta pergunta parece intrigar os estudantes, aparentemente porque eles esquecem qual é o problema que está sendo resolvido! O problema é saber quantos números serão digitados pelo usuário, se soubéssemos quantos seriam não precisaríamos fazer um programa para isso!

O computador não funciona como a mente humana. Humanos são capazes de visualizar e processar os dados, desde que não sejam muitos, de uma maneira global, ele tem acesso a todos os dados de uma só vez.

Por exemplo, imagine que o usuário vai digitar o seguinte conjunto de informações:

\[
5 \ 8 \ 3 \ 9 \ 0
\]

Como são poucos dados, um ser humano não precisa fazer nenhuma conta, ele vê que existem cinco números, e sabe que o zero não conta, então já responde imediatamente: são 4 números.\footnote{Beremir, protagonista do livro “O Homem que Calculava” era capaz de dizer quantos pássaros haviam numa revoada no céu.}

Mas o computador não funciona assim. Ele tem que processar as informações uma a uma. E por causa disso e também do problema de que não sabemos quantas informações serão digitadas, só resta uma alternativa: usar uma única variável.

Como pode ser possível resolver este problema usando-se uma única variável? Usando-se lógica de programação!

O segredo é pensar como o computador e tentar extrair um algoritmo deste processo. Vamos lá.

- O usuário digita 5 e aperta ENTER;
6.1. LÓGICA DE PROGRAMAÇÃO

- O usuário digita 8 e aperta ENTER;
- O usuário digita 3 e aperta ENTER;
- O usuário digita 9 e aperta ENTER;
- O usuário digita 0 e aperta ENTER;

A cada ENTER a informação estará sendo processada por um comando `read(n)`, onde `n` é uma variável do tipo integer. Notem que é a mesma variável sempre. Significa que a cada novo ENTER o conteúdo da variável `n` foi alterado.

- Na primeira vez, `n` valerá 5;
- Na próxima vez, `n` valerá 8 e o 5 foi perdido;
- Na próxima vez, `n` valerá 3 e o 8 foi perdido;
- Na próxima vez, `n` valerá 9 e o 3 foi perdido;
- Na próxima vez, `n` valerá 0 e o 9 foi perdido;
- Como foi digitado um 0, devemos parar o processo e imprimir o resultado

Isto indica que, a cada ENTER, devemos `contar` um a mais. Porém, um a mais a partir de qual informação? Ora, nós humanos estamos habituados a iniciar nossas contagens com o número 1. É preciso também notar que isso naturalmente sugere o uso de um comando de repetição.

Qual processo deve ser repetido?

- Ler um número;
- Contar um a mais.

Sabemos também, do nosso aprendizado do capítulo anterior, que um processo repetitivo deve ter uma variável de controle e que também deve ter um critério de parada. Nosso critério de parada parece óbvio: é quando o usuário digitar um zero.

Uma reflexão adicional: devemos `lembrar` de duas informações diferentes, uma delas é o número que está sendo digitado, a outra é um contador que inicia em 1 e é incrementado a cada novo número informado.

Já temos todos os elementos necessários para um rascunho do algoritmo, vejamos na figura 6.1 como isso pode ser materializado em um programa em *Pascal*.

Observem a lógica de programação:

- inicialmente um número deve ser lido: `read(n);`
- devemos também inicializar o contador: `cont:= 1;`
- aqui entra o laço, que deve ter sua condição de parada:
  `while n <> 0 do;`
program contar_numeros;
var cont, n: integer;

begin
   read (n);
   cont:= 1;
   while n <> 0 do
      begin
         read (n);
         cont:= cont + 1;
      end;
   writeln (cont - 1);
end.

Figura 6.1: Contando números.

Façamos uma pausa para uma importante questão: qual o motivo da expressão booleana usada ser \( n <> 0 \)?

Por causa da semântica do comando while/do. A semântica é o significado do comando. O comando while/do repete um bloco de outros comandos enquanto uma condição for verdadeira. Quando a condição, ou a expressão booleana, for falsa, devemos parar o programa. Significa dizer em outras palavras que queremos parar quando a variável \( n \) tiver o valor zero, logicamente, para que os comandos do laço sejam executados, o teste deve ter o valor lógico contrário do que queremos como parada.

Em outras palavras, se queremos parar com \( n = 0 \), significa que a expressão booleana deve ser o teste contrário, isto é, a negação de \( n = 0 \), que é obviamente \( n <> 0 \).

Continuemos com a análise do código:

- caso a expressão \( n <> 0 \) seja verdadeira, os dois comandos do laço são executa-
- o primeiro faz a leitura do próximo valor;
- o segundo faz o incremento do contador \( cont \);
- com o próximo valor lido, o teste do laço, ou seja, a expressão booleana é reavaliada, mas com este novo valor digitado;
- e o processo se repete até que a expressão \( n <> 0 \) seja falsa;
- neste caso a repetição termina e o comando que vem depois do end; é executado;
- este comando é o de impressão do resultado: writeln(cont - 1).

Uma pergunta interessante: por quê foi impresso o valor de \( cont - 1 \)?

Porquê iniciamos o contador com 1. Faça as contas: imagine que, logo de cara, o usuário digite um zero. Os comandos no escopo do while/do não serão executados, mas a variável \( cont \) vale 1. Significa que, na prática, o usuário não quis digitar
nenhum valor, pois o combinado é que o zero é para terminar o programa e não deve ser processado. Não podemos imprimir 1 na tela, temos que imprimir cont - 1, que vale zero.

Do ponto de vista puramente lógico, podemos modificar este programa da maneira como é mostrado na figura 6.2. A diferença é que o contador cont inicia em zero, e a impressão final imprime cont, e não mais cont - 1.

```
program contar_numeros;
var cont, n: integer;

begin
  read (n);
  cont:= 1;
  while n <> 0 do
  begin
    read (n);
    cont:= cont + 1;
  end;
  writeln (cont - 1);
end.
```

Figura 6.2: Contando números.

Contar pode ser feito de várias maneiras. Por exemplo, para contar cinco números podemos fazer assim: 0, 1, 2, 3, 4. Ou então: 11, 12, 13, 14, 15. Para o computador, tanto faz, a lógica é a mesma e é baseado na matemática!

Observem também que os comandos no escopo do while foram invertidos de modo a atender à nova lógica da solução. Como a variável contadora (cont) vale inicialmente zero, a cada número não nulo lido, deve-se incrementar o contador antes da nova leitura.

### 6.2 O teste de mesa

Uma técnica básica para testar, a priori, o funcionamento de um programa é conhecido como teste de mesa. Ele consiste em acompanhar o programa segundo o fluxo das instruções e ir anotando em um papel, que supostamente está em uma mesa, os valores das variáveis. Com isso é possível observar se o programa contém algum erro de lógica.

A seguir mostraremos o teste de mesa para o programa da figura 6.2. No caso do aprendiz, servirá também para verificar se compreendeu bem os conceitos apresentados até este ponto do texto.
CAPÍTULO 6. TÉCNICAS ELEMENTARES

<table>
<thead>
<tr>
<th>situação</th>
<th>valor de ( n )</th>
<th>valor de ( i )</th>
<th>comentário</th>
</tr>
</thead>
<tbody>
<tr>
<td>antes do início</td>
<td>indefinido</td>
<td>indefinido</td>
<td>alocação de memória</td>
</tr>
<tr>
<td>executa ( \text{cont} := 0 )</td>
<td>0</td>
<td>indefinido</td>
<td></td>
</tr>
<tr>
<td>executa ( \text{read}(n) )</td>
<td>0</td>
<td>5</td>
<td>supondo que leu 5</td>
</tr>
<tr>
<td>testa ( n ) ( &lt;&gt; 0 )</td>
<td>0</td>
<td>5</td>
<td>( 5 \neq 0 ), entra no laço</td>
</tr>
<tr>
<td>executa ( \text{cont} := \text{cont} + 1 )</td>
<td>1</td>
<td>5</td>
<td>incrementou ( \text{cont} )</td>
</tr>
<tr>
<td>executa ( \text{read}(n) )</td>
<td>1</td>
<td>8</td>
<td>supondo que leu 8</td>
</tr>
<tr>
<td>testa ( n ) ( &lt;&gt; 0 )</td>
<td>1</td>
<td>8</td>
<td>( 8 \neq 0 ), permanece no laço</td>
</tr>
<tr>
<td>executa ( \text{cont} := \text{cont} + 1 )</td>
<td>2</td>
<td>5</td>
<td>incrementou ( \text{cont} )</td>
</tr>
<tr>
<td>executa ( \text{read}(n) )</td>
<td>2</td>
<td>3</td>
<td>supondo que leu 3</td>
</tr>
<tr>
<td>testa ( n ) ( &lt;&gt; 0 )</td>
<td>2</td>
<td>3</td>
<td>( 3 \neq 0 ), permanece no laço</td>
</tr>
<tr>
<td>executa ( \text{cont} := \text{cont} + 1 )</td>
<td>3</td>
<td>3</td>
<td>incrementou ( \text{cont} )</td>
</tr>
<tr>
<td>executa ( \text{read}(n) )</td>
<td>3</td>
<td>9</td>
<td>supondo que leu 9</td>
</tr>
<tr>
<td>testa ( n ) ( &lt;&gt; 0 )</td>
<td>3</td>
<td>9</td>
<td>( 9 \neq 0 ), permanece no laço</td>
</tr>
<tr>
<td>executa ( \text{cont} := \text{cont} + 1 )</td>
<td>4</td>
<td>9</td>
<td>incrementou ( \text{cont} )</td>
</tr>
<tr>
<td>executa ( \text{read}(n) )</td>
<td>4</td>
<td>0</td>
<td>supondo que leu 0</td>
</tr>
<tr>
<td>testa ( n ) ( &lt;&gt; 0 )</td>
<td>4</td>
<td>0</td>
<td>( 0 = 0 ), sai do laço</td>
</tr>
<tr>
<td>executa ( \text{writeln}(\text{cont}) )</td>
<td>4</td>
<td>0</td>
<td>imprime 4 na tela</td>
</tr>
</tbody>
</table>

6.3 Técnica do acumulador

A chamada **técnica do acumulador** é somente uma nomenclatura para generalizar o conteúdo visto até aqui neste capítulo. Na verdade o que fizemos para resolver o problema anterior foi justamente usar esta técnica para acumular valores na variável \( \text{cont} \). Especificamente, acumulamos nesta variável incrementos de 1 em 1 conforme os números eram lidos.

O problema a seguir permite entender a generalização.

**Problema:** Ler uma sequência de números positivos do teclado e imprimir a soma deles. O programa deve terminar quando o número lido do teclado for zero. Este zero não deve ser processado.

O programa deve ser desenvolvido conforme a figura 6.3.

Observem que é um problema muito parecido com o anterior, pelo menos parte dele, a que consiste em ler uma certa quantidade de números do teclado. A diferença é o que fazer com estes valores lidos. No problema anterior, nós os contávamos. Agora precisamos somá-los.

O algoritmo apresentado na figura 6.3 implementa a solução. Quando um número é digitado no teclado, somamos este valor na variável acumuladora, no caso \( \text{soma} \).

Observem que a variável \( \text{soma} \) deve ser inicializada com algum valor, pois na primeira vez não contém valor definido. O melhor valor inicial para ela é zero, pois é o elemento neutro da adição. Se fosse uma multiplicação de vários números, o acumulador deveria ser inicializado com o elemento neutro da multiplicação, isto é, o 1.
6.4. ÁRVORES DE DECISÃO

![Program code]

Figura 6.3: Técnica do acumulador.

6.3.1 Exemplo

O próximo problema é parecido com o anterior, a diferença é no critério de parada da entrada de dados.

**Problema:** Ler um número \( N > 0 \) do teclado e em seguida ler \( N \) números inteiros quaisquer. Ao final imprimir a soma deles.

Na figura 6.4 mostra-se como resolvê-lo usando a técnica dos acumuladores. É fácil perceber que esta maneira é genérica, resolve problemas similares para qualquer tamanho de entrada, bastando-se trocar o valor da constante \( MAX \).

Este programa faz uso de dois acumuladores, um deles, \( soma \), é utilizado como no programa anterior, para acumular a soma dos valores lidos. O segundo deles, \( i \), é usado para contar de 1 até \( MAX \). Observe que na parte interna do laço \( i \) é incrementado de uma unidade. Como antes do laço ele é inicializado com o valor 1, o resultado é que os comandos do laço serão executados exatamente \( MAX \) vezes.

6.4 Árvores de decisão

Árvores de decisão são utilizadas quando precisamos tomar uma série de decisões com diversos fluxos alternativos de código. Basicamente, temos que optar por um caminho inicialmente desconhecido, mas prevendo todas as possibilidades, de forma que, quando os dados forem conhecidos já equacionamos o problema e a resposta sai fácil.

**Problema:** Após ler três números no teclado, imprimir o menor deles.

A solução para este problema é apresentada na figura 6.5.

---

\(^2\) Estude sobre *constantes* no mini guia de referência da linguagem *Pascal* e na literatura recomendada.
CAPÍTULO 6. TÉCNICAS ELEMENTARES

```
program soma_valores;
const MAX=5;
var
    numero, i, soma: integer;
begin
    soma:= 0;
i:= 1; (* segundo acumulador, para contar os números lidos *)
while i <= MAX do
    begin
        read (numero);
soma:= soma + numero;
i:= i + 1; (* atualização do segundo acumulador *)
    end;
end.
```

Figura 6.4: Solução para o problema de somar $N$ números.

Lembrando que quando um comando está sob o controle de outro, diz-se que o comando mais interno está sob o escopo do mais externo. Neste caso, vamos usar um desvio condicional que controla outro. No jargão dos programadores, se diz que os comandos nesta forma estão aninhados.

```
program imprime_menor;
var
    a, b, c: integer;
begin
    writeln('entre com tres numeros inteiros: ');
read(a, b, c);
if a < b then
    if a < c then
        writeln ('o menor dos tres eh ',a)
    else
        writeln ('o menor dos tres eh ',c)
else (* entra neste else quando a >= b *)
    if b < c then
        writeln ('o menor dos tres eh ',b)
    else
        writeln ('o menor dos tres eh ',c);
end.
```

Figura 6.5: Imprimir o menor dentre 3 números lidos.

É importante observar que neste código um único comando writeln será executado. A escolha de qual deles depende de duas tomadas de decisão que dependem dos valores das variáveis lidas no teclado. A lógica deste programa, que orienta a elaboração do algoritmo, é a seguinte: Se for verificado que $a$ é menor do que $b$ e em seguida for verificado que $a$ também é menor do que $c$, segue logicamente que $a$ é o menor dos três. Observem que não se sabe neste ponto a ordem relativa de $b$ com relação à $c$,
6.5 Definir a priori e depois corrigir

Em muitas situações precisamos descobrir uma determinada informação que depende da entrada de dados completa, mas desde o início precisamos nos lembrar de uma informação importante sobre as primeiras informações já fornecidas.

É o caso, por exemplo, do problema de encontrar o menor número de uma sequência de números sem saber quantos números serão lidos. Portanto não podemos usar a técnica das árvores de decisão.

Problema: Ler uma sequência de números positivos terminada em zero do teclado e imprimir o menor deles. Este zero não deve ser processado pois serve apenas para marcar o final da entrada de dados.

Retomando a mesma discussão do início deste capítulo, o computador vai ler um único número de cada vez e a cada número lido em uma variável sobrescreve o valor da anteriormente lida. Isso dificulta encontrar o menor, a não ser que exploremos novamente a lógica de programação.

O raciocínio é o seguinte: quando lemos o primeiro número do teclado, pode ser que ele seja o menor de todos, mas também pode não ser. Sequer sabemos neste momento quantos números serão lidos, quanto menos o valor de cada um deles.

Mas, para todos os efeitos, de todos os números lidos até agora, e temos só este primeiro, ele é o menor de todos. Esta afirmação segue uma lógica absoluta. O menor de um conjunto de 1 elemento é este próprio elemento.

Usando este argumento, “chutamos” que este único elemento é o menor de todos. Depois, conforme formos conhecendo os outros valores que estão chegando do teclado, vamos checando se o nosso chute inicial ainda é válido. Caso não seja, corrigimos a informação. Ao final do processo teremos o menor valor de todos.

Este programa inicialmente testa se o primeiro número lido é zero, o que quer dizer que não há nada para ser feito pois não há entrada válida. Isto é necessário para podermos fazer o chute inicial da linha 7 do programa. Se isto não for feito, o programa retornaria o valor zero como sendo o menor, o que não faz sentido.

Mas desde que exista uma primeira informação válida, então o chute é dado na linha 7 e o laço que segue continua lendo os valores do teclado e testando para ver
program menor;
var n, menor: integer;
begin
  read (n);
  if n <> 0 then // se o primeiro for zero nao faz nada
  begin
    menor:= n; // aqui chutamos que o menor eh o primeiro lido
  while n <> 0 do
  begin
    read (n);
    if n < menor then // aqui testamos a nova informacao
      menor:= n; // e corrigimos se for o caso
  end;
  writeln (menor);
end.

Figura 6.6: Encontrando o menor elemento.

se algum novo número é menor do que aquele que, até o momento, acreditamos ser o menor. Caso a informação tenha que ser corrigida (teste na linha 11), isto é feito na linha 12. Quando o zero é digitado no teclado o laço termina e podemos imprimir o valor que é, garantidamente, o menor de todos.

6.6 Lembrar de mais de uma informação

A técnica anterior exigiu a memorização de uma única variável relevante para a determinação da solução. A que apresentaremos agora exige duas. Evidentemente ela pode ser generalizada para que lembremos de quantas quisermos, mas isto será abordado mais à frente neste texto.

Vamos usar como problema motivador uma das mais famosas sequências de números que existe. Trata-se da Sequência de Fibonacci³.

Esta sequência é gerada de modo bastante simples. Os dois primeiros valores são 1 e 1. Os seguintes são obtidos pela soma dos dois anteriores. Assim, a sequência de Fibonacci é: 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, . . .

Problema: Imprimir os primeiros \( n > 2 \) números da sequência de Fibonacci.

Como gerar e imprimir os elementos desta sequência é o nosso desafio. A solução exige que se guarde sempre os dois últimos elementos gerados, senão não é possível resolver o problema. Observamos que a frase do parágrafo anterior dá a dica: "os seguintes são obtidos pela soma dos dois anteriores".

Na solução apresentada na figura 6.7 consideramos duas variáveis importantes, uma para armazenar o último elemento já produzido pelo algoritmo, a outra para

guardar o penúltimo. Com estes dois, é possível produzir a soma deles, isto é, o próximo elemento.

A atualização destas variáveis deve ser feita sempre que o próximo elemento for obtido, pois a soma do último com o penúltimo é agora o novo último elemento gerado. O que era o antigo último passa a ser o penúltimo. A ordem da atualização destes valores é relevante no código em função do esquema de funcionamento de memória do computador. Trocando-se a ordem dos comandos o algoritmo para de funcionar.

| program Fibonacci;   |
| var ultimo, penultimo, soma, cont, n: integer; |
| begin               |
| read (n);          | (* define quantos números serão impressos *) |
| ultimo:= 1;        | (* inicialização das variáveis principais *) |
| penultimo:= 1;     |
| writeln (penultimo); (* imprime os dois primeiros valores *) |
| writeln (ultimo);  |
| cont:= 3 ;         | (* calcula do terceiro em diante *) |
| while cont <= n do |
| begin              |
| soma:= penultimo + ultimo; |
| writeln (cont, ' ', soma); |
| penultimo:= ultimo;   | (* a ordem destes dois comandos *) |
| ultimo:= soma;      | (* é relevante no código *) |
| cont:= cont + 1;    |
| end;                |
| end.                |

Figura 6.7: Gerando números da sequência de Fibonacci.

O algoritmo da figura [6.7] é normalmente o mais utilizado para este problema, isto é, define-se os dois primeiros e depois sucessivamente soma-se os dois últimos e atualiza-se o último como sendo esta soma recentemente gerada e o penúltimo como sendo o que era o último. Existem vários outros que são apresentados como exercícios.

### 6.7 Processar parte dos dados de entrada

É muito frequente que os programas devam processar uma grande massa de dados sendo que parte da informação de entrada não interessa e consequentemente estes dados devem ser desprezados. O próximo problema ilustra esta situação.

**Problema:** Ler do teclado 30 números inteiros e imprimir na tela aqueles que são positivos, ignorando os negativos ou nulos.

Se o enunciado fosse simplesmente para ler e imprimir 30 números, como faríamos? Provavelmente como ilustrado na figura [6.8]. Observamos que este problema é muito similar a outros já estudados no capítulo anterior. Trata-se de um laço que precisa de um contador para se determinar a saída.
program lereimprimir;
var i , a: integer; (* i serve para contar quantos numeros foram lidos *)
begin
  i:= 1;
  while i <= 30 do
    begin
      read (a);
      writeln (a);
      i:= i + 1;
    end;
end.

Figura 6.8: Lendo e imprimindo 30 números.

O problema é que não queremos imprimir todos os números lidos, mas unicamente aqueles que são positivos: se o número lido for positivo, então queremos imprimir, senão não. Conforme já visto, o comando if faz exatamente isto, testa uma expressão booleana que, caso satisfeita, executa o comando subsequente, senão o pula, passando diretamente para o seguinte, tal como ilustrado na figura 6.9.

program lereimprimirpositivos;
var i , a: integer;
begin
  i:= 1;
  while i <= 30 do
    begin
      read (a);
      if a > 0 then
        writeln (a); (* so eh executado quando a eh positivo *)
      i:= i + 1;
    end;
end.

Figura 6.9: Lendo e imprimindo os positivos.

### 6.8 Processar parte dos dados de um modo e outra parte de outro

Nesta situação, diferentemente da seção anterior, quando parte dos dados deveria ser ignorada, agora queremos processar uma parte dos dados de uma maneira e a outra parte de outra maneira.

O comando de desvio condicional também permite executar exclusivamente uma ação alternativa, caso o teste da expressão booleana resulte em falso.

**Problema:** Ler do teclado 30 números inteiros e imprimir na tela aqueles que são
pares e imprimir o quadrado dos que não são, incluindo o zero.

Do ponto de vista lógico, seria o mesmo que dizer o seguinte: “se o número lido for positivo, imprimi-lo, caso contrário ele é zero ou negativo, então imprimir o quadrado dele”. Isto pode ser visto na figura [6.10]. Destacamos que a estrutura de controle do laço não mudou com relação ao problema anterior, os comandos que são executados no laço mudaram, caracterizando-se dois subproblemas: um para controlar o laço, outro para se decidir se o número é positivo ou não e o que fazer com ele.

```pascal
program lereimprimirpositivosequadrados;
var i, a: integer;
begin
  i:= 1;
  while i <= 30 do
    begin
      read (a);
      if a > 0 then
        writeln (a) (* so eh executado quando a for positivo *)
      else
        writeln (a*a) (* so eh executado quando a <= 0 *)
      i:= i + 1;
    end;
end.
```

Figura 6.10: Lendo e imprimindo os positivos e os quadrados dos outros.

Agora, o comando que imprime a só é executado quando a > 0. Se isto não for verdade, o que é impresso é o quadrado de a. Em ambos os casos o valor de i é incrementado, garantindo a correta contagem de 30 números lidos.

6.9 Múltiplos acumuladores

Um caso típico em que é preciso usar múltiplos acumuladores são os histogramas, também conhecidos como distribuição de frequências.

Histogramas servem basicamente para estudos estatísticos preliminares, no sentido de que os dados são previamente tabulados em algumas faixas que podem na sequência produzir gráficos de barras, por exemplo.

Vamos supor que os dados de entrada foram coletados em um hospital que recebeu pacientes com determinada doença. Os médicos podem ter interesse em saber qual, ou quais, faixas etárias são mais frequentes para esta determinada doença. Então querem tabular nas faixas: bebês (0-2 anos), crianças (3-12 anos), adolescentes (14-19 anos), adultos (20-60 anos) ou idosos (60 ou mais anos), por exemplo.

Problema: Ler do teclado uma sequência de números inteiros terminada em -1 e imprimir na tela o histograma classificado conforme as faixas acima. O -1 não deve
ser processado e serve para marcar o final da entrada de dados.

A solução deste problema requer o uso de tantas variáveis quantas forem as faixas etárias de interesse, no nosso caso são 5: bebês, crianças, adolescentes, adultos e idosos. Cada uma delas define um acumulador. Os acumuladores são incrementados em função de múltiplas escolhas, cada uma definindo uma das faixas de interesse. O programa que resolve este problema pode ser visto na figura 6.11, usando-se os tradicionais if/then/else aninhados em while/do.

```pascal
program histograma;
var bebe, crianca, adolescente, adulto, idoso, idade: integer;
begin
  bebe:= 0;
  crianca:= 0;
  adolescente:= 0;
  adulto:= 0;
  idoso:= 0;
  read (idade);
  while idade <> -1 do
  begin
    if idade <= 2 then bebe:= bebe + 1
    else if idade <= 12 then crianca:= crianca + 1
    else if idade <= 19 then adolescente:= adolescente + 1
    else if idade <= 60 then adulto:= adulto + 1
    else idoso:= idoso + 1
    read (idade);
  end;
  writeln ('bebes: ',bebe);
  writeln ('criancas: ',crianca);
  writeln ('adolescentes: ',adolescente);
  writeln ('adultos: ',adulto);
  writeln ('idosos: ',idoso);
end.
```

Figura 6.11: Um programa para construir um histograma.
Apêndice

Este apêndice pode ser tranquilamente ignorado pelo leitor, ele resolve o mesmo problema da última seção, mas usando um novo comando da linguagem *Pascal*: o `case`, que pode ser melhor estudado nos guias desta linguagem.

A figura [6.12] mostra esta nova versão, mas o programa é basicamente o mesmo. O `case` serve para economizar uma grande sequência de `if/then/elses`.

```pascal
program histograma;
var bebe, crianca, adolescente, adulto, idoso, idade: integer;
begin
  bebe:= 0;
  crianca:= 0;
  adolescente:= 0;
  adulto:= 0;
  idoso:= 0;
  read (idade);
  while idade <> -1 do
    begin
      case idade of
        0..2: bebe:= bebe + 1;
        3..12: crianca:= crianca + 1;
        13..19: adolescente:= adolescente + 1;
        20..60: adulto:= adulto + 1;
        else idoso:= idoso + 1;
      end;
      read (idade);
    writeln ('bebes: ',bebe);
    writeln ('criancas: ',crianca);
    writeln ('adolescentes: ',adolescente);
    writeln ('adultos: ',adulto);
    writeln ('idosos: ',idoso);
  end.
end.
```

Figura 6.12: O mesmo programa usando `case`.
6.10 Exercícios

1. Faça um programa em *Pascal* que, dados dois números inteiros positivos imprima o valor da maior potência do primeiro que divide o segundo. Se o primeiro não divide o segundo, a maior potência é definida como sendo igual a 1. Por exemplo, a maior potência de 3 que divide 45 é 9.

   **Entrada:**
   
   3 45
   
   **Saída:**
   
   9

2. Faça um programa em *Pascal* que, dadas as populações $P_A$ e $P_B$ de duas cidades $A$ e $B$ em 2009, e suas respectivas taxas de crescimento anual $X_A$ e $X_B$, receba estas informações como entrada e imprima:

   - se a população da cidade de menor população ultrapassará a de maior população;
   - e o ano em que isto ocorrerá.

3. Um inteiro positivo $N$ é perfeito se for igual a soma de seus divisores positivos diferentes de $N$.

   Exemplo: 6 é perfeito pois $1 + 2 + 3 = 6$ e 1, 2, 3 são todos os divisores positivos de 6 e que são diferentes de 6.

   Faça um programa em *Pascal* que receba como entrada um número positivo $K$ e imprima os $K$ primeiros números perfeitos.

4. Faça um programa em *Pascal* que, dado um inteiro positivo $n$, imprima todos os termos, do primeiro ao $n$-ésimo, da sequência abaixo. Você pode assumir que o usuário nunca digita valores menores que 1 para $n$.

   $5, 6, 11, 12, 17, 18, 23, 24, \ldots$

5. Faça um programa em *Pascal* que, dada uma sequência de números naturais positivos terminada por 0 (zero), imprima o histograma da sequência dividido em quatro faixas (o histograma é a contagem do número de elementos em cada faixa):

   - Faixa 1: 1 – 100;
   - Faixa 2: 101 – 250;
   - Faixa 3: 251 – 20000;
   - Faixa 4: acima de 20001.

   **Exemplo:**

   **Entrada:** 347 200 3 32000 400 10 20 25 0
   **Saída:**
   
   Faixa 1: 4
   Faixa 2: 1
   Faixa 3: 2
   Faixa 4: 1
6. Faça um programa em *Pascal* que leia uma sequência de pares de números inteiros quaisquer, sendo dois inteiros por linha de entrada. A entrada de dados termina quando os dois números lidos forem nulos. Este par de zeros não deve ser processado e serve para marcar o término da entrada de dados.

Para cada par \(A, B\) de números lidos, se \(B\) for maior do que \(A\), imprima a sequência \(A, A + 1, \ldots, B - 1, B\). Caso contrário, imprima a sequência \(B, B + 1, \ldots, A - 1, A\).

Exemplos:

<table>
<thead>
<tr>
<th>Entrada</th>
<th>Saída</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 6</td>
<td>4 5 6</td>
</tr>
<tr>
<td>-2 1</td>
<td>-2 -1 0 1</td>
</tr>
<tr>
<td>2 -3</td>
<td>-3 -2 -1 0 1 2</td>
</tr>
<tr>
<td>0 0</td>
<td></td>
</tr>
</tbody>
</table>

7. Faça um programa em *Pascal* que receba um número inteiro \(N\) como entrada e imprima cinco linhas contendo as seguintes somas, uma em cada linha:

\[
\begin{align*}
N \\
N + N \\
N + N + N \\
N + N + N + N \\
N + N + N + N + N
\end{align*}
\]

Exemplo:

<table>
<thead>
<tr>
<th>Entrada</th>
<th>Saída</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>3 6 9 12 15</td>
</tr>
</tbody>
</table>

8. Faça um programa em *Pascal* que imprima exatamente a saída especificada na figura 1 (abaixo) de maneira que, em todo o programa fonte, não apareçam mais do que três comandos de impressão.

\[
\begin{align*}
1 \\
121 \\
12321 \\
1234321 \\
123454321 \\
12345654321 \\
1234567654321 \\
123456787654321 \\
12345678987654321
\end{align*}
\]

Figura 1
9. Faça um programa em *Pascal* que imprima exatamente a mesma saída solicitada no exercício anterior, mas que use exatamente dois comandos de repetição.

10. Adaptar a solução do exercício anterior para que a saída seja exatamente conforme especificada na figura 2 (abaixo).

```
1
121
12321
1234321
123454321
12345654321
1234567654321
123456787654321
12345678987654321
```

Figura 2

11. Faça um programa em *Pascal* que leia do teclado uma sequência de \(N > 0\) números quaisquer. Para cada valor lido, se ele for positivo, imprimir os primeiros 10 múltiplos dele.

12. Sabe-se que um número da forma \(n^3\) é igual a soma de \(n\) números ímpares consecutivos.

Exemplos:
- \(1^3 = 1\)
- \(2^3 = 3 + 5\)
- \(3^3 = 7 + 9 + 11\)
- \(4^3 = 13 + 15 + 17 + 19\)

Dado \(M\), faça um programa em *Pascal* que imprima os ímpares consecutivos cuja soma é igual a \(n^3\) para \(n\) assumindo valores de 1 a \(M\).

13. Faça um programa em *Pascal* que, dados dois números naturais \(m\) e \(n\) imprima, dentre todos os pares de números naturais \((x, y)\) tais que \(x \leq m\) e \(y \leq n\), um par para o qual o valor da expressão \(xy - x^2 + y\) seja máximo e calcular também esse máximo.

14. Faça um programa em *Pascal* que leia uma sequência de números e imprima separadamente a soma dos que são pares e a soma dos que são ímpares. O programa deve terminar quando o número lido for o zero, que não deve ser processado pois serve para marcar o final da entrada de dados.
15. Faça um programa em Pascal que leia uma sequência de números inteiros do teclado terminada em zero e imprima os que são ao mesmo tempo múltiplos de 7 mas não são múltiplos de 2. O zero não deve ser processado pois serve para marcar o final da entrada de dados.

16. Faça um programa em Pascal que leia uma sequência de números inteiros do teclado terminada em zero e imprima os que forem ao mesmo tempo múltiplos de 3 maiores do que 50 e menores ou iguais a 201.

17. Faça um programa em Pascal que leia uma sequência de trincas de números inteiros do teclado terminada em três zeros e imprima, para cada trinca de entrada, o menor deles. Os três zeros não devem ser processados pois servem para marcar o final da entrada de dados.

18. (*) Escreva um programa em Pascal que leia uma sequência de números inteiros, terminada em $-1$. Para cada número inteiro lido, o programa deve verificar se este número está na base binária, ou seja, se é composto somente pelos dígitos 0 e 1. Caso o número esteja na base binária, o programa deve imprimir seu valor na base decimal. Caso contrário, deve imprimir uma mensagem indicando que o número não é binário. Ao final do programa deve ser impresso, em formato decimal, o maior número válido (binário) da sequência.

Dica: dado o número 10011 em base binária, seu valor correspondente em base decimal será dado por

$$1.2^4 + 0.2^3 + 0.2^2 + 1.2^1 + 1.2^0 = 19$$

Exemplo:

Entrada:     Saída:
10011         19
121          numero nao binario
1010          10
101010101      341
0             0
-1

O maior número binário foi 341
Capítulo 7

Aplicações das técnicas elementares

Neste capítulo escreveremos soluções para problemas cujo grau de dificuldade é similar aos daqueles do capítulo anterior, com o objetivo de fixar conceitos. Nestes problemas as técnicas devem ser combinadas com inteligência e lógica.

7.1 Convertendo para binário

Este é um bom problema para ajudar a fixar os conceitos estudados.

Problema: Dado um número inteiro entre 0 e 255 imprimir este número em seu formato binário.

A definição de números é uma uma série especial de potências de 2. O algoritmo para conversão normalmente apresentado em sala de aula é baseado em uma sequência de divisões por 2. O resultado se obtém tomando-se os restos das sucessivas divisões por 2 “ao contrário”, isto é, do último resto de divisão por 2 até o primeiro. Por exemplo, tomando o decimal 22 como entrada:

\[
\begin{array}{c}
22 \text{ div } 2 \\
0 \\
11 \text{ div } 2 \\
1 \\
5 \text{ div } 2 \\
1 \\
2 \text{ div } 2 \\
1 \\
1 \text{ div } 2 \\
1 \\
0
\end{array}
\]

Então o número binário correspondente ao 22 seria 10110. O programa ilustrado na figura 7.1 mostra o código para esta versão do algoritmo.

Ocorre que este algoritmo imprime o binário ao contrário, isto é, a saída para a entrada 22 seria 01101 e não 10110 como deveria. Na verdade, basta lembrar que desde o início deste texto se insiste em afirmar que para um mesmo problema existem diversas soluções. Neste caso, basta usar a definição de números binários.

\[1\text{Este exemplo nos ocorreu após trocas de emails com o Allan Neves, então aluno da disciplina. Agradecemos a ele por isto.}\]
program converteparabinario;
const max=128;
var n: integer;
begin
  write ('entre com um numero entre 0 e 255: ');
  read (n);
  while n <> 0 do
    begin
      write (n mod 2);
      n:= n div 2;
    end;
end.

Figura 7.1: Convertendo para binário, versão 1.

De fato, o número decimal 22 pode ser escrito numa série de potências de 2 como segue:

\[
22 = 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0
\]

A questão é saber quantas vezes cada potência de 2 cabe no número original. Este cálculo é simples e a solução é mostrada na figura 7.2.

program converteparabinario_v2;
const max=128;
var diferenca, n, pot2: integer;
begin
  write ('entre com um numero entre 0 e 255: ');
  read (n);
  pot2:= max;
  while pot2 <> 0 do
    begin
      diferenca:= n - pot2;
      if diferenca >= 0 then
        begin
          write (1);
          n:= diferenca;
        end
      else
        write (0);
      pot2:= pot2 div 2;
    end;
end.

Figura 7.2: Convertendo para binário, versão 2.
# 7.2 Calculo do MDC

**Problema:** Imprimir o Máximo Divisor Comum (MDC) entre dois números dados.

O conceito matemático de máximo divisor comum entre dois números dados $a$ e $b$ envolve a fatoração de cada número como um produto de fatores primos, escolhendo-se os fatores primos que se repetem com a potência mínima.

Exemplo: Calcular o MDC entre 72 e 135.

$$72 = 2^3 \times 3^2$$

$$135 = 3^3 \times 5$$

Da teoria conclui-se que o MDC entre 72 e 135 é $3^2$, pois o 3 é o único fator primo que se repete em ambos os números de entrada, e a menor potência comum é 2.

Implementar este algoritmo para encontrar o MDC é complicado no momento pois não sabemos ainda como obter uma sequência de primos. Também não sabemos ainda o quanto caro é calcular um número primo.

Euclides propôs um algoritmo eficiente para se obter o MDC entre dois números que não requer o uso da fatoração. Trata-se de um dos primeiros algoritmos conhecidos, pois foi proposto por volta do ano 300 a.c. O algoritmo pode ser visto atualmente em *Pascal* conforme está ilustrado na figura 7.3.

```
program mdcpeuclides;
var a, b, resto: integer;
begin
  read (a,b);
  if (a <> 0) AND (b <> 0) then
  begin
    resto := a mod b;
    while resto <> 0 do
    begin
      a := b;
      b := resto;
      resto := a mod b;
    end;
    writeln ('mdc = ', b);
  end
  else
  writeln ('o algoritmo nao funciona para entradas nulas.');
end.
```

Figura 7.3: Algoritmo de Euclides para cálculo do MDC.

Os cálculos do MDC são feitos somente se as entradas não forem nulas. Caso sejam nulas, a mensagem de aviso é impressa na tela.
7.3 Tabuada

**Problema:** Imprimir as tabuadas do 1 ao 10.

No código apresentado na figura 7.4 para cada valor de \( i \), os valores de \( j \) variam de 1 a 10, o que faz com que o comando `writeln` mais interno seja executado 100 vezes ao longo do programa, o que é diferente do outro comando `writeln` que é executado exatamente 10 vezes. Em termos genéricos, para uma entrada de tamanho \( n \), o comando mais interno é executado da ordem de \( n^2 \) vezes, por isto são conhecidos como algoritmos de complexidade quadrática. Nos programas anteriores, o máximo que tínhamos era uma complexidade linear com relação ao tamanho da entrada.

```plaintext
program tabuada;
var i, j: integer;
begin
  i := 1;
  while i <= 10 do
    begin
      j := 1;
      while j <= 10 do
        begin
          writeln (i, 'x', j, '=' , i*j); (* writeln mais interno *)
          j := j + 1;
        end;
      writeln; (* este aqui é executado apenas 10 vezes *)
      i := i + 1;
    end;
end.
```

**Figura 7.4:** Tabuadas do 1 ao 10.

É mais fácil perceber isso quando se vê a saída deste programa. A figura 7.5 ilustra isso. Por questões de espaço no texto a figura aparece em *modo econômico*, usando reticências. Cada quadro representa uma valoração para \( i \). A real saída do programa apresenta cada quadro abaixo do outro, separada por uma linha em branco. Mas aqui a ideia é o leitor perceber que esta saída contém 100 linhas, em 10 blocos de 10 linhas, exatamente compatível com os dois laços aninhados.

```
| 1 x 1 = 1 | 2 x 1 = 2 | ... | 10 x 1 = 10 |
| 1 x 2 = 2 | 2 x 2 = 4 | ... | 10 x 2 = 20 |
| 1 x 3 = 3 | 2 x 3 = 6 | ... | 10 x 3 = 30 |
| ... | ... | ... | ...
| 1 x 10 = 10 | 2 x 10 = 20 | ... | 10 x 10 = 100 |
```

**Figura 7.5:** Saída do programa da tabuada.
7.4 Fatorial

O problema seguinte é bastante relevante neste ponto, pois nos permitirá uma análise sobre a técnica usada para resolvê-lo. Vamos mostrar que combinar comandos sem compreender um problema as vezes pode resultar em um algoritmo pouco eficiente.

**Problema:** Imprimir o valor do fatorial de todos os números entre 1 e $n$, sendo $n$ fornecido pelo usuário.

Usando as técnicas já mencionadas, podemos observar inicialmente que este problema é muito parecido com vários outros já estudados. Se soubermos encontrar um fatorial, então pode-se colocar a solução para este subproblema em um laço e se obter a solução para o problema global.

O fatorial de $n$ é assim definido:

$$n = n \times (n - 1) \times (n - 2) \times \ldots \times 3 \times 2 \times 1$$

Logo, o cálculo do fatorial de um número tem complexidade linear com relação ao tamanho da entrada e pode ser facilmente implementado usando a técnica elementar dos acumuladores, conforme é ilustrado na figura 7.6, usando-se a técnica dos acumuladores. Formatamos a saída de maneira especial para facilitar o argumento que queremos com este exemplo.

```pascal
program fatorial;
var i, n, fat: integer;
begin
  read (n);
  fat:= 1; (* inicializacao do acumulador *)
  i:= n;
  write ('fat(',n,')= ');
  while i >= 1 do
  begin
    fat:= fat * i;
    if i > 1 then
      write (i , 'x')
    else
      write (i , '=');
    i:= i - 1;
  end;
  writeln ( fat )
end.
```

Saída do programa com entrada n=7:

fat(7)= 7x6x5x4x3x2x1= 120

Figura 7.6: Obtendo o fatorial de $n$. 
Esta versão resolve o problema para o cálculo do fatorial de um único número dado como entrada e portanto não resolve o problema especificado no enunciado. Porém, numa análise simplista, alguém poderia dizer que basta colocar este trecho sob o controle de outra repetição que se obtém o efeito desejado. Esta solução é apresentada na figura 7.7.

É verdade que o programa funciona, mas é extremamente ineficiente e repleto de cálculos redundantes. De fato, basta observar na saída apresentada para o programa para o valor de \( n = 7 \) que o programa calculou 6 vezes a multiplicação de \( 2 \times 1 \), calculou 5 vezes a multiplicação de \( 3 \times 2 \), calculou 4 vezes a de \( 4 \times 6 \) e assim por diante. O programador não explorou corretamente os cálculos já feitos anteriormente.

```pascal
program fatorial1_n;
var cont, i, n, fat: integer;
begin
    read (n);
    cont:= 1;
    while cont <= n do
      begin
          write ('fat(',cont,')= ');
          fat:= 1; (* inicialização do acumulador *)
          i:= cont;
          while i >= 1 do
            begin
                fat:= fat * i;
                if i > 1 then
                  write (i,'x')
                else
                  write (i,'= ');
                i:= i - 1;
            end;
      end;
  writeln (fat);
  cont:= cont + 1;
end.
```

Saída do programa com entrada \( n=7 \):

\[
\begin{align*}
\text{fat}(1) &= 1= 1 \\
\text{fat}(2) &= 2x1= 2 \\
\text{fat}(3) &= 3x2x1= 6 \\
\text{fat}(4) &= 4x3x2x1= 24 \\
\text{fat}(5) &= 5x4x3x2x1= 120 \\
\text{fat}(6) &= 6x5x4x3x2x1= 720 \\
\text{fat}(7) &= 7x6x5x4x3x2x1= 5040
\end{align*}
\]

Figura 7.7: Obtendo vários fatoriais.
7.5. NÚMEROS DE FIBONACCI REVISITADO

Em outras palavras, o programa pode ficar mais eficiente se for feito como ilustrado na figura 7.8, de maneira que o algoritmo, mais esperto, tem complexidade linear, e não quadrática como na versão anterior. Interessante observar que a solução é eficiente e usa uma única atribuição no escopo de uma repetição.

A ideia aqui é: se já temos o valor do fatorial de um número \( n \), então o cálculo para o valor \( n + 1 \) pode ser feito assim: \( \text{fat}(n + 1) = (n + 1) \times \text{fat}(n) \). Como \( \text{fat}(1) = 1 \), por definição, basta construir iterativamente o fatorial de \( n \) a partir do fatorial de 1.

```pascal
program fatorial1_n_v2;
var cont, n, fat: integer;
begin
  read (n);
  cont:= 1;
  fat:= 1; (* inicialization do acumulador *)
  while cont <= n do
    begin
      fat:= fat * cont;
      writeln ('fat(',cont,')= ', fat);
      cont:= cont + 1;
    end;
end.
```

Saída do programa com entrada \( n=7 \):

\[
\begin{align*}
\text{fat}(1)&= 1 \\
\text{fat}(2)&= 2 \\
\text{fat}(3)&= 6 \\
\text{fat}(4)&= 24 \\
\text{fat}(5)&= 120 \\
\text{fat}(6)&= 720 \\
\text{fat}(7)&= 5040 \\
\end{align*}
\]

Figura 7.8: Otimizando o cálculo de vários fatoriais.

7.5 Números de Fibonacci revisitado

Na seção 6.6 vimos como calcular números da sequência de Fibonacci. Nesta seção veremos outros problemas relacionados.

7.5.1 Alterando o critério de parada

Um problema similar mas alternativo a este é, por exemplo, saber qual é o primeiro número de Fibonacci maior do que um determinado valor. Uma pequena alteração no controle de parada e no local do comando de impressão resolve o novo problema.
Isto é apresentado na figura [7.9]. A diferença básica é que neste caso não é preciso contar os números, pois o critério de parada é diferente. Mas é importante observar que a parte da geração dos números da sequência não mudou.

```pascal
program Fibonacci_2;
const max=1000;
var ultimo, penultimo, soma, cont : integer;
begin
  ultimo:= 1; (* inicialização das variáveis principais *)
  penultimo:= 1;
  soma:= penultimo + ultimo;
  while soma <= max do (* calcula do terceiro em diante *)
  begin
    penultimo:= ultimo;
    ultimo:= soma;
    soma:= penultimo + ultimo;
  end;
  writeln (soma);
end.
```

Figura 7.9: Imprimindo o primeiro número de Fibonacci maior do que 1000.

### 7.5.2 O número áureo

Uma das maiores belezas dos números de Fibonacci é que a razão entre dois termos consecutivos converge para um número irracional conhecido como *número áureo*, denotado pela letra grega \( \varphi \), e é aproximadamente 1.6180339887499. De fato, vejamos a razão entre dois termos consecutivos para alguns números pequenos:

\[
\frac{1}{1} = 1, \quad \frac{2}{1} = 2, \quad \frac{3}{2} = 1.5, \quad \frac{5}{3} = 1.66, \quad \frac{8}{5} = 1.6, \quad \frac{13}{8} = 1.625, \quad \frac{21}{13} = 1.615, \ldots
\]

O algoritmo que calcula o número áureo com a precisão desejada, mostrando os valores intermediários, é apresentado na figura [7.10]. Notamos que ele verifica a convergência da sequência para a razão áurea, fazendo as contas até que o erro seja menor que a precisão desejada. A função `abs` é nativa do `Pascal` e retorna o valor absoluto do número dado como argumento.

Neste exemplo, novamente o cálculo central não mudou, isto é, os números continuam a ser gerados da mesma maneira. O que muda é *o que fazer com eles*. No caso, é preciso obter a razão do último pelo penúltimo.

Várias outras propriedades interessantes serão deixadas como exercício.
program numero_aureo;
const PRECISAO = 0.000000000000001;
var ultimo, penultimo, soma: integer;
    naureo, naureo_anterior: real;
begin
    ultimo := 1; (* inicialização das variáveis principais *)
    penultimo := 1;
    naureo_anterior := −1; (* para funcionar o primeiro teste *)
    naureo := 1;
    writeln (naureo:15:14);
    (* calcula do terceiro em diante *)
    while abs(naureo − naureo_anterior) >= PRECISAO do
        begin
            soma := penultimo + ultimo;
            naureo_anterior := naureo;
            naureo := soma/ultimo;
            writeln (naureo:15:14);
            penultimo := ultimo;
            ultimo := soma;
        end;
end.

Figura 7.10: Verificando a convergência do número áureo.
7.6 Inverter um número de três dígitos

Este problema vai nos permitir reforçar o uso da técnica do acumulador ao mesmo tempo em que continuamos a mostrar várias soluções para o mesmo problema de maneira a evoluirmos na compreensão da arte de se construir algoritmos.

**Problema:** Ler um número de 3 dígitos do teclado e imprimir este número invertido. Exemplo, se a entrada for “123” a saída deve ser “321”.

A primeira solução, apresentada na figura [7.11] foi feita pelos alunos em sala de aula em um curso no segundo semestre de 2002. É um bom exemplo de como particularizar a solução leva, quase que necessariamente, a dificuldades imensas na reimplementação do código para problemas similares.

Eles pensaram em separar os três dígitos e imprimir na ordem certa. Assim, os operadores de divisão e resto de divisão inteira são utilizados para a separação dos números em unidade, dezena e centena.

```plaintext
program inverte3_v0;
var numero, unidade, dezena, centena, inverso: integer;
begin
    write('entre com o numero de tres digitos: ');
    readln(numero);
    centena:= numero div 100;
    dezena:= (numero mod 100) div 10;
    unidade:= numero mod 10;
    inverso := unidade*100 + dezena*10 + centena;
    writeln(inverso);
end.
```

Figura 7.11: Primeira solução para inverter um número de 3 dígitos.

O programa funciona, mas tem um grave problema. Se o enunciado fosse sobre números de 4 dígitos teríamos um sério transtorno. Vejamos a figura [7.12] que preserva o raciocínio central da primeira solução, desta vez para 4 dígitos.

```plaintext
program inverte4;
var numero, unidade, dezena, centena, milhar, inverso: integer;
begin
    write('entre com o numero de quatro digitos: ');
    readln(numero);
    milhar:= numero div 1000;
    centena:= (numero mod 1000) div 100;
    dezena:= (numero mod 100) div 10;
    unidade:= numero mod 10;
    inverso := unidade*1000 + dezena*100 + centena*10 + milhar;
    writeln(inverso);
end.
```

Figura 7.12: Mesmo algoritmo, agora para 4 dígitos.
Além de ter que declarar mais uma variável (milhar), houve alteração de praticamente todas as linhas do código. É possível imaginar o estrago no código se for exigido que a entrada fosse constituída de 10 dígitos ou mais. De fato, o computador não “pensa” como o ser humano.

É preciso deixar que o computador faça o trabalho, não o programador. Se este último pensar o suficiente, fará um código baseado em um algoritmo mais elaborado e que possa ser adaptado para problemas similares.

O que queremos é tentar perceber algum raciocínio que possa ser repetido um número controlado de vezes. Deve-se tentar explorar uma mesma sequência de operações, que consiste em separar o último dígito e remover do número original este dígito que foi separado.

Desta forma, a construção da resposta não pode mais ser feita apenas no final do algoritmo. À medida que o número original vai sendo separado, o seu inverso já deve ir sendo construído. O código é apresentado na figura 7.13 e seu funcionamento é baseado em separar o último dígito com uma operação de \[\text{mod} \ 10\] seguida de outra operação \[\text{div} \ 10\].

Por exemplo, se temos o número 123, operamos \[123 \ \text{mod} \ 10\] e obtemos o último dígito, no caso 3. A próxima operação é \[123 \ \text{div} \ 10\] que resulta em 12. Assim vamos “destruindo” o número original a cada iteração, mas ao mesmo tempo usando os dígitos separados para ir construindo o número ao contrário. Quem controla isso são as sucessivas multiplicações por 10.

Esta técnica permite que com uma simples alteração no controle do laço possamos inverter números com qualquer quantidade de dígitos. Isto fica como exercício.

```pascal
program inverte3_v1;
var
i, numero, unidade, inverso, resto: integer;
begin
write('entre com o numero de tres digitos: ');
readln(numero);
inverso := 0;

i:= 1;
while (i <= 3) do
begin
unidade := numero mod 10;
resto := numero div 10;
inverso := inverso*10 + unidade;
numero := resto;
i:= i + 1;
end;
writeln(inverso);
end.
```

Figura 7.13: Solução com uso de acumuladores.
7.7 Palíndromos

Nesta seção generalizamos a ideia central apresentada no problema anterior.

**Problema:** Imprimir todos os números palíndromos entre 1 e 1000.

Números palíndromos são aqueles que são lidos da direita para a esquerda da mesma maneira que da esquerda para a direita. Por exemplo o número 12321 é palíndromo, enquanto que 123 não é.

Para testar se um número é palíndromo usamos a seguinte técnica: separamos os dígitos do número em questão, em seguida reconstruímos o número ao contrário, usando a técnica do acumulador, e finalmente comparamos com o número original.

Lembrando que um número da forma \(d_0d_1d_2\ldots d_n\) é escrito da seguinte forma:

\[
d_0 \times 10^0 + d_1 \times 10^1 + d_2 \times 10^2 + \ldots + d_n \times 10^n
\]

Para gerar o número ao contrário basta fazer multiplicações sucessivas por 10 invertendo-se os dígitos, assim:

\[
d_0 \times 10^n + d_1 \times 10^{n-1} + d_2 \times 10^{n-2} + \ldots + d_n \times 10^0
\]

O algoritmo da figura [7.14] generaliza o raciocínio usado no problema anterior e testa todos os números e imprime quando são palíndromos.

```pascal
program todospalindromos;
const max=1000;
var i, invertido, n: integer;
begin
  i:= 1;
  while i <= max do (* laco que controla os numeros entre 1 e max *)
  begin
    invertido:= 0; (* inicializa acumulador *)
    n:= i;
    while n <> 0 do
    begin
      invertido:= invertido*10 + n mod 10;
      n:= n div 10;
    end;
    (* imprime se for palindromo, senao nao faz nada *)
    if invertido = i then
      writeln (i);
    i:= i + 1;
  end.
end.
```

Figura 7.14: Imprimindo todos os palíndromos de 1 a 1000.
Testar palíndromos pode ser muito caro dependendo da aplicação. Em todo caso, apresentamos um novo problema que é no mínimo divertido: o de gerar os palíndromos. Ele servirá para o estudante perceber melhor o uso de contadores em diversos níveis de aninhamentos de comandos de repetição.

**Problema:** Gerar todos os números palíndromos entre 1 e 1000.

O algoritmo apresentado na figura 7.15 contém a solução para este problema. Ele tem por base o formato dos palíndromos, gerando todos os de um dígito, depois todos os de dois dígitos e finalmente todos os de 3 dígitos. Um problema interessante é generalizar este código, mas deixamos isto como exercício.

```pascal
program gerandopalindromos;
var i, j, pal: integer;
begnin
  i:= 1;                             (* gerando todos de um digito *)
  while i <= 9 do
    begin
      writeln (i);
      i:= i + 1;
    end;

  pal:= 11;
  i:= 2;                             (* gerando todos de 2 digitos *)
  while i <= 10 do
    begin
      writeln (pal);
      pal:= i * 11;
      i:= i + 1;
    end;

  i:= 1;                             (* gerando todos de 3 dígitos *)
  while i <= 9 do
    begin
      j:= 0;
      while j <= 9 do
        begin
          pal:= i*100 + j*10 + i;
          writeln (pal);
          j:= j + 1;
        end;
      i:= i + 1;
    end;
end.
```

Figura 7.15: Gerando todos os palíndromos de 1 a 1000.
7.8 Maior segmento crescente

Problema: Dada uma sequência de \( n \) números naturais, imprimir o valor do comprimento do segmento crescente de tamanho máximo dentre os números lidos.

Por exemplo, se a sequência for: 5, 10, 3, 2, 4, 7, 9, 8, 5, o comprimento crescente máximo é 4, já que o maior segmento é 2, 4, 7, 9. Para 10, 8, 7, 5, 2, o comprimento de um segmento crescente máximo é 1.

O algoritmo que resolve este problema é apresentado na figura 7.16.

```pascal
program maior_seg_crescente;
var n, tamanho, maiortam, a_anterior, i, a: integer;
begin
  read (n);                 (* inicializa as variáveis principais *)
  tamanho:= 0;
  maiortam:= 0;
  a_anterior:= 0;           (* zero eh o primeiro numero natural *)
  i:= 1;
  while i <= n do begin
    read (a);
    if a > a_anterior then (* continuamos na mesma sequencia *)
      tamanho:= tamanho + 1
    else (* foi lido um numero menor, ja eh uma outra sequencia *)
      begin
        if tamanho > maiortam then (* lembra o maior tamanho *)
          maiortam:= tamanho;
        tamanho:= 1;             (* reinicializa o contador *)
      end;
    a_anterior:= a;           (* lembra do numero anterior *)
    i:= i + 1;
  end;
  (* este ultimo if testa se a ultima sequencia nao eh a maior *)
  if tamanho > maiortam then (* lembra o maior tamanho *)
    maiortam:= tamanho;
  writeln ('maior tamanho encontrado: ',maior);  (* varia vel *)
end.
```

Figura 7.16: Maior segmento crescente.

A solução exige um conjunto de variáveis que controlam o estado da computação, isto é, que mantenho a memória de que tipo de números foi lido até o momento, segundo uma dada restrição. Os comentários no código explicam como guardar o tamanho da sequência sendo lida em comparação com o maior tamanho lido até o momento.

É preciso lembrar o valor lido anteriormente para comparar com o valor atual. Isto é necessário para saber se o valor atual é maior que o anterior. Caso seja, estamos em uma sequência crescente. Caso contrário, trata-se de outra sequência. A variável
tamanho armazena o tamanho da sequência crescente sendo lida, enquanto que maior registra a maior sequência crescente que já foi processada até o momento.

### 7.9 Séries

Nesta seção tratamos de problemas que envolvem o cálculo de séries, normalmente utilizadas para cálculos de funções tais como seno, cosseno, logaritmo, etc... A técnica utilizada é basicamente aquela dos acumuladores, porém, o cálculo dos novos termos somados é ligeiramente mais sofisticado e exige atenção do estudante para a boa compreensão da solução.

#### 7.9.1 Número neperiano

**Problema:** Calcular o valor no número \( e = 2.718281 \ldots \) pela série abaixo, considerando somente os vinte primeiros termos:

\[
e = \frac{1}{0!} + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \frac{1}{4!} + \frac{1}{5!} + \frac{1}{6!} + \frac{1}{7!} + \ldots
\]

A figura 7.17 ilustra uma solução baseada nas já estudadas técnicas dos acumuladores e no problema dos fatorial. Basicamente, o novo termo é calculado em função do fatorial, que, neste caso, aparece no denominador.

```program neperiano;
var e, novotermo: real;
    fat, i: longint;
const itmax=20;
begind\begin{verbatim}
e:= 0; (* inicializacao das variaveis *)
fat:= 1;
i:= 1; (* calculo da serie *)
while i <= itmax do
begin
    novotermo:= 1/fat;
    e:= e + novotermo;
    fat:= i*fat;
i:= i + 1;
end;
writeln ('e= ',e);
end.
\end{verbatim}

Figura 7.17: Cálculo do número neperiano.
```

O programa poderia ter dispensado o uso da variável novotermo, mas deixamos assim pois facilita a compreensão de que a grande e nova dificuldade neste problema
é “como gerar o novo termo a partir do anterior”? Isto vai ajudar na solução de
problemas similares, na sequência do texto.

Neste caso, o enunciado do problema determinou que o programa calculasse 20
termos da série. Alguns enunciados estabelecem como condição de parada um critério
de erro, isto é, se o cálculo em uma iteração difere do cálculo anterior por um valor
previamente estabelecido, isto é, uma precisão previamente determinada. Em alguns
casos, quando o cálculo da série não é convergente, este valor mínimo nunca é atingido,
obrigando também o programa a testar um número máximo de iterações. Segue a
mesma ideia discutida no caso de gerar o número áureo.

Suponhamos que o enunciado tivesse como condição de parada um destes dois
critérios: ou o erro é menor do que $10^{-4}$ ou o número máximo de iterações é 50. Se
o programa sair pelo critério de erro, então o valor de $e$ foi encontrado, senão, se
saiu pelo critério do número máximo de iterações, então o programa avisa que não
conseguiu encontrar a solução. A figura 7.18 ilustra a solução para este caso. É
importante observar que o que muda com relação à primeira solução é unicamente o
critério de parada. Os cálculos internos são os mesmos.

```
program neperiano,v2;
const itmax=50; precisao=0.0001;
var e , eanterior , novotermo: real;
    fat , i : integer;
begin
    e:= 0;
    eanterior:= -1;
    fat:= 1;
    i:= 1;
    while ( i <= itmax) and (e - eanterior > precisao) do 
     begin
         novotermo:= 1/fat;
         eanterior:= e;
         e:= e + novotermo;
         fat:= i*fat;
         i:= i + 1;
     end;
    if i > itmax then
     writeln ('solucao nao encontrada')
    else
     writeln ('e = ',e);
end.
```

Figura 7.18: Cálculo do número neperiano.
7.9. SÉRIES

7.9.2 Cálculo do seno

Problema: Calcular o valor de \( \text{seno}(x) \) pela série abaixo, considerando somente os vinte primeiros termos:

\[
\text{seno}(x) = \frac{x}{1!} - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!} - \frac{x^{11}}{11!} + \frac{x^{13}}{13!} - \frac{x^{15}}{15!} + \ldots
\]

Comparando esta série com a do número neperiano, observamos três diferenças básicas: a primeira é o sinal, que a cada termo muda de positivo para negativo e vice-versa e antes era sempre positivo; a segunda é o numerador, que antes era a constante 1, agora é uma potência de \( x \); a terceira é que os fatoriais não são consecutivos em cada termo, aumentam de dois em dois.

Trataremos de cada caso separadamente, construindo a solução para o seno baseada na do número neperiano. Vamos tratar primeiro o mais simples, que é a questão dos sinais. O algoritmo da figura [7.19] modifica o anterior pela introdução de uma nova variável que controla a mudança do sinal, produzindo como saída o cálculo da função \( tmp \).

\[
\text{tmp} = \frac{1}{0!} - \frac{1}{1!} + \frac{1}{2!} - \frac{1}{3!} + \frac{1}{4!} - \frac{1}{5!} + \frac{1}{6!} - \frac{1}{7!} + \ldots
\]

program serie_v1;
const itmax=20;
var seno, novotermo: real;
    fat, sinal, i: integer;
begin
    seno:= 0;
    fat:= 1;
    sinal:= 1;
    i:= 1;
    while i <= itmax do
        begin
            novotermo:= 1/fat;
            seno:= seno + sinal*novotermo;
            sinal:= ~sinal; (* nova variável para sinal trocado *)
            fat:= i*fat;
            i:= i + 1;
        end;
    writeln (’tmp= ’,seno);
end.

Figura 7.19: Série com troca de sinais.
Agora vamos corrigir o denominador, obtendo a seguinte série:

\[ \text{tmp2} = \frac{1}{1!} - \frac{1}{3!} + \frac{1}{5!} - \frac{1}{7!} + \frac{1}{9!} - \frac{1}{11!} + \frac{1}{13!} - \frac{1}{15!} + \ldots \]

A atualização da variável \( \text{fat} \) deve ser modificada, conforme apontado na figura 7.20:

```pascal
program serie_v2;
const itmax=20;
var seno, novotermo: real;
    fat, sinal, i: integer;
begin
    seno:= 0;
    fat:= 1;
    sinal:= 1;
    i:= 1;
    while i <= itmax do
    begin
        novotermo:= 1/fat;
        seno:= seno + sinal*novotermo;
        fat:= 2*i*(2*i+1)*fat; (* esta eh a principal modificacao *)
        sinal:= -sinal;
        i:= i + 1;
    end;
    writeln ('tmp2= ',seno);
end.
```

Figura 7.20: Série com troca de sinais e fatoriais no denominador corrigidos.

Agora só resta a atualização correta do numerador, que depende da leitura de um valor \( x \) do teclado. Uma variável adicional contém o valor de \( x^2 \), para evitar cálculos redundantes. O programa ilustrado na figura 7.21 implementa corretamente o cálculo do \( \text{seno}(x) \) para um \( x \) dado em radianos.

Existem várias outras séries que podem ser implementadas com estas mesmas ideias, vamos deixar isto como exercício. Basicamente, basta o estudante observar como os numeradores e denominadores podem ser atualizados em função do termo anteriormente calculado.
```pascal
program senox;
const itmax=20;
var seno, x, x_quadrado, numerador, novotermo: real;
    fat, i, sinal: integer;
begin
    seno:= 0;
    fat:= 1;
    sinal:= 1;
    read (x);
    x_quadrado:= x*x;
    numerador:= x;
    i:= 1;
    while i <= itmax do
    begin
        novotermo:= numerador/fat;  (* atualiza o termo *)
        seno:= seno + sinal*novotermo;
        numerador:= numerador*x_quadrado;  (* atualiza o numerador *)
        fat:= 2*i*(2*i+1)*fat;
        sinal:= -sinal;
        i:= i + 1;
    end;
    writeln ('seno(',x,')= ',seno);
end.
```

Figura 7.21: Cálculo do seno de \( x \).
7.10 Números primos

Este é um dos mais complexos problemas desta parte inicial da disciplina. Além disso, os números primos são de particular interesse em computação, pois estão por trás dos principais algoritmos de criptografia e transmissão segura na Internet. Nesta seção vamos estudar alguns algoritmos para se determinar se um número é ou não primo.

**Problema:** Dado um número natural \( n \), determinar se ele é primo.

Números naturais primos são aqueles que são divisíveis apenas por ele mesmo e por 1. Em outras palavras, se \( n \) é um número natural, então ele é primo se, e somente se, não existe outro número \( 1 < p < n \) que divida \( n \).

Aplicando-se diretamente a definição, temos que verificar se algum número entre \( 2 \) e \( n - 1 \) divide \( n \). Se \( p \) divide \( n \) então o resultado da operação \( n \mod p \) é zero.

O primeiro algoritmo, apresentado na figura 7.22, é bastante simples: basta variar \( p \) de \( 2 \) até \( n - 1 \) e contar todos os valores para os quais \( p \) divide \( n \). Se a contagem for zero, o número não tem divisores no intervalo e é portanto primo. Senão não é.

<table>
<thead>
<tr>
<th>program ehprimo;</th>
</tr>
</thead>
<tbody>
<tr>
<td>var n, cont, i: integer;</td>
</tr>
<tr>
<td>begin</td>
</tr>
<tr>
<td>read (n);</td>
</tr>
<tr>
<td>cont:= 0; (* contador de divisores de n *)</td>
</tr>
<tr>
<td>i:= 2;</td>
</tr>
<tr>
<td>while i &lt;= n-1 do</td>
</tr>
<tr>
<td>begin</td>
</tr>
<tr>
<td>if n mod i = 0 then</td>
</tr>
<tr>
<td>cont:= cont + 1; (* achou um divisor *)</td>
</tr>
<tr>
<td>i:= i + 1;</td>
</tr>
<tr>
<td>end;</td>
</tr>
<tr>
<td>if cont = 0 then</td>
</tr>
<tr>
<td>writeln (n,' eh primo')</td>
</tr>
<tr>
<td>end.</td>
</tr>
</tbody>
</table>

Figura 7.22: Verifica se \( n \) é primo contando os divisores.

Este algoritmo é bom para se determinar quantos divisores primos um dado número tem, mas não é eficiente para este problema pois basta saber se existe pelo menos um divisor. Logo, basta parar logo após o primeiro divisor ter sido encontrado.

A técnica utilizada é baseada em uma variável booleana inicializada como sendo verdadeira. O algoritmo “chuta” que \( n \) é primo mas, em seguida, se os cálculos mostrarem que o chute estava errado, a informação é corrigida.

O laço principal do programa deve ter duas condições de parada: (1) termina quando um divisor foi encontrado; (2) termina quando nenhum divisor foi encontrado, isto é, quando \( i \) ultrapassou \( n - 1 \). Um teste na saída do laço encontra o motivo da saída e imprime a resposta correta. Este algoritmo pode ser visto na figura 7.23.

A análise deste algoritmo deve ser feita em duas situações: (1) no pior caso, aquele em que o número é primo; (2) no melhor caso, aquele em que o número não é primo.


7.10. NÚMEROS PRIMOS

program ehprimo_v2;
var n, i: integer;
    eh_primo: boolean;
begin
    read (n);
    eh_primo:= true; (* inicia chutando que n eh primo *)
    i:= 2;
    while (i <= n−1) and eh_primo do
        begin
            if n mod i = 0 then
                eh_primo:= false; (* se nao for, corrige *)
                i:= i + 1;
        end;
    if eh_primo then
        writeln (n, ' eh primo')
end.

Figura 7.23: Testa se n é primo parando no primeiro divisor.

No segundo caso, o algoritmo vai terminar bem rápido. No outro, ele vai testar todas as possibilidades de ímpares. Mas o caso ótimo é raro. De fato, nos problemas envolvendo criptografia, estes números primos tem duzentos ou mais dígitos. Isto pode fazer com que o computador fique bastante tempo processando a informação.

Percebemos então que se o número for par então só tem uma chance dele ser também primo, justamente se o número for o 2. No caso dos ímpares é necessário que todos sejam testados.

O algoritmo testa inicialmente se o número é par. Se for, testa se é o 2, que é primo. Se for ímpar, testa todos os ímpares entre 1 e n − 1, eliminando metade dos cálculos, pois os pares foram descartados.

Melhorou, mas pode melhorar mais com um pouco mais de observação: não é necessário se testar todos os ímpares, basta que se teste até a raiz no número.

De fato, todo número natural pode ser decomposto como um produto de números primos. Se a entrada não for um primo, então pode ser decomposta, no melhor caso, assim: \( n = p \times p \), em que \( p \) é primo. O algoritmo que implementa esta solução é mostrado na figura 7.24.

Para se ter uma ideia do ganho, vejam na tabela seguinte o quanto se ganha com as três últimas versões do programa.

<table>
<thead>
<tr>
<th>( x )</th>
<th>( \sqrt{x} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000000</td>
<td>1000</td>
</tr>
<tr>
<td>100000000000</td>
<td>1000000</td>
</tr>
</tbody>
</table>

A tabela acima mostra que para entradas da ordem de \( 10^{12} \), o número de cálculos feitos com o programa da figura 7.23 pode ser da mesma ordem de \( 10^{12} \). Os cálculos do programa da última versão mostra que o programa pode fazer cálculos da ordem de “somente” \( 10^{6} \).
program eh_primo_v3;
var n, i: integer;
    eh_primo: boolean;
begin
    read (n);
    eh_primo:= true; (* inicia chutando que n eh primo *)
    if n mod 2 = 0 then (* n eh par *)
        if n <> 2 then
            eh_primo:= false (* n nao eh 2 *)
        else eh_primo:= true
    else begin (* n nao eh par, testar todos os impares *)
        i:= 3;
        while (i <= trunc(sqrt(n))) and eh_primo do
        begin
            if n mod i = 0 then
                eh_primo:= false; (* achamos um divisor impar *)
                i:= i + 2;
        end;
    end;
    if eh_primo then
        writeln (n, ' eh primo')
end.

Figura 7.24: Testa se n é primo parando na raiz de n.
7.11 Refinamentos sucessivos

Neste ponto consideramos que o estudante já domina as estruturas elementares de controle de fluxo de execução de um programa, então podemos aplicar estes conceitos em alguns algoritmos mais sofisticados.

A partir de agora vamos construir as soluções partindo de algoritmos escritos em uma linguagem de mais alto nível, chamada de pseudocódigo, que não pode ser compilada. Isto nos permitirá separar o problema global em subproblemas. Isolaremos estes subproblemas, apresentaremos soluções para eles e finalmente mostraremos como integrar cada solução parcial gera o código final para o problema inicial.

7.11.1 Primos entre si

Problema: Imprimir todos os pares de números \((a, b)\) que são primos entre si para todo \(2 \leq a \leq 100\) e \(a \leq b \leq 100\).

Este problema pode ser dividido em dois subproblemas: (1) dado um par de números quaisquer, como saber se eles são primos entre si? (2) como gerar todos os pares ordenados no intervalo desejado? Nosso conhecimento deveria ser suficiente para resolvermos o segundo subproblema trivialmente. Por este motivo vamos iniciar a solução por ele, ignorando completamente o primeiro subproblema, por enquanto.

A figura 7.25 contém um código global que gera todos os pares ordenados que interessam. Observe que o \(\{i e j sao primos entre si\} then writeln (i, j)\) na parte central do pseudocódigo é o subproblema que estamos ignorando no momento.

```
begin
  i:= 2;
  while i <= 100 do
    begin
      j:= i; (* para gerar pares com j sendo maior ou igual a i *)
      while j <= 100 do
        begin
          if \{i e j sao primos entre si\} then writeln (i, j);
          j:= j + 1;
        end;
    i:= i + 1;
  end;
end.
```

Figura 7.25: Pseudocódigo para o problema dos primos entre si.

Com o segundo subproblema resolvido podemos nos concentrar no primeiro, ou seja, escrever código para decidir se dois números são primos entre si. Agora, o foco da solução é neste pequeno subproblema.

Dois números \(a\) e \(b\) são números primos entre si quando o máximo divisor comum entre eles é 1. Isto é, \(mdc(a, b) = 1\).

Na seção 7.2 vimos como se calcula de forma eficiente o MDC entre dois números pelo método de Euclides (figura 7.3). Inserimos aquele código na nossa versão parcial
e teremos a solução global. Isto é mostrado na figura 7.26. O teste $b = 1$ no final do código significa que o mdc encontrado vale 1, isto é, os valores são primos entre si e o par em questão pode ser impresso.

```plaintext
program primosentresi;
var i, j, a, b, resto: integer;
begin
  i:= 2;
  while i <= 100 do
    begin
      j:= i;
      while j <= 100 do
        begin
          a:= i; b:= j; (* inicio do bloco euclides *)
          resto:= a mod b; (* * *)
          while resto <> 0 do (* * *)
            begin
              a:= b; (* * *)
              b:= resto; (* * *)
              resto:= a mod b; (* * *)
            end; (* termino do bloco euclides *)
          if b = 1 then writeln (i, j); (* b=1 era o oráculo *)
          j:= j + 1;
        end;
      i:= i + 1;
    end.
end.
```

Figura 7.26: Gerando todos os primos entre si.
7.11.2 Amigos quadráticos

Neste problema buscaremos por uma solução global usando vários níveis de detalhamento, começando pelo pseudocódigo de mais alto nível, passando por um outro intermediário e finalmente, com mais um refinamento, obteremos o código final.

**Problema:** Dois números naturais \( n \) e \( m \) são ditos amigos quadráticos quando \( n \) é igual a soma dos dígitos de \( m^2 \) e ao mesmo tempo \( m \) é igual a soma dos dígitos de \( n^2 \). Imprimir todos os pares \( n \) e \( m \) que são amigos quadráticos no intervalo \( 1 \leq 10000 \leq n \) e \( 1 \leq 10000 \leq m \).

Por exemplo, os números 13 e 16 são amigos quadráticos, pois \( 13^2 = 169 \) e \( 1+6+9=16 \). Por outro lado, \( 16^2 = 256 \) e \( 2+5+6=13 \).

Vamos iniciar gerando os pares de números dentro do intervalo desejado, conforme fizemos na seção anterior. Para cada par, alguns oráculos nos auxiliarão na busca pela solução. O pseudocódigo para isto está na figura 7.27.

```
begin
  n:= 1;
  while n <= 10000 do
    begin
      m:= 1;
      while m <= 10000 do
        begin
          if \{ \( m \) e \( n \) são amigos quadráticos \} then
            writeln(\( n \), ', e ', \( m \), ', são amigos quadráticos.'
        m:= m + 1;
        end;
      n:= n + 1;
    end;
end.
```

Figura 7.27: Pseudocódigo para o problema dos amigos quadráticos.

Agora nos concentraremos exclusivamente no problema de se determinar se dois números \( m \) e \( n \) são amigos quadráticos. Para isto vamos imaginarmos que sabemos como somar os dígitos de um número. Com esta crença o pseudocódigo para decidir sobre amigos quadráticos parece bastante trivial, como na figura 7.28.

```
begin \{ dados \( n \) e \( m \} 
  soma_n:= \{ oráculo que soma os dígitos de \( n \}
  soma_m:= \{ oráculo que soma os dígitos de \( m \}
  if soma_n = soma_m then
    writeln(\'são amigos quadráticos.
end.
```

Figura 7.28: Pseudo-código para decidir sobre amigos quadráticos.
Finalmente resta o terceiro subproblema: como separar os dígitos de um número inteiro qualquer dado, para depois somá-los? A técnica usada explora a operação aritmética de se pegar o resto da divisão inteira por 10. Vejamos um exemplo para o número 169.

\[
169 \div 10 \\
9 \quad 16 \div 10 \\
6 \quad 1 \div 10 \\
1 \quad 0
\]

No exemplo acima o número 169 foi sucessivamente dividido por 10 até virar zero. Os restos das divisões por 10 nos dão os dígitos do 169, a saber, 9, 6 e 1, obtidos nesta ordem. Isto está implementado conforme ilustrado na figura 7.29.

```plaintext
begin 
{ dado n inteiro }
while n <> 0 do 
begin 
dígito := n mod 10;
n := n div 10;
end;
end.
```

Figura 7.29: Separando os dígitos de um dado número \( n \).

A variável \( \text{dígito} \) contém, a cada iteração, um dígito que faz parte do número dado. Para se obter a soma, basta usar a técnica do acumulador, conforme a figura 7.30.

```plaintext
begin 
{ dado \( N \) inteiro }
soma := 0; 
while \( N <> 0 \) do 
begin 
dígito := N mod 10;
soma := soma + dígito;
N := N div 10;
end;
end.
```

Figura 7.30: Somando os dígitos de um dado número \( n \).

Por último, não resta mais nada a não ser juntar os diversos pedaços de código em um único programa que executa a tarefa com sucesso. Isto é mostrado na figura 7.31. O leitor deve estudar atentamente a maneira como os diversos códigos foram integrados.
program amigosquadraticos;
var n, m, n2, soma_dig_n2, digito, m2, soma_dig_m2: integer;
begin
  n:= 1;
  while n <= 10000 do
    begin
      m:= 1;
      while m <= 10000 do
        begin
          (* decompo n ao quadrado *)
          n2:= n*n;
          soma_dig_n2:= 0;
          while n2 <> 0 do
            begin
              digito:= n2 mod 10;
              soma_dig_n2:= soma_dig_n2 + digito;
              n2:= n2 div 10;
            end;
          (* decompo m ao quadrado *)
          m2:= m*m;
          soma_dig_m2:= 0;
          while m2 <> 0 do
            begin
              digito:= m2 mod 10;
              soma_dig_m2:= soma_dig_m2 + digito;
              m2:= m2 div 10;
            end;
          if (soma_dig_n2 = m) and (soma_dig_m2 = n) then
            writeln(n, ' e ', m, ' sao amigos quadraticos.');
        end;
      m:= m + 1;
    end;
  n:= n + 1;
end.

Figura 7.31: Verificando se números n e m são amigos quadráticos.
É possível contudo fazer menos cálculos e economizar um laço, reduzindo de um milhão para mil operações se, para cada \( n \), testarmos se a soma dos seus dígitos ao quadrado pode ser transformado novamente em \( n \). Veja o algoritmo da figura 7.32 e compare com a versão anterior.

Isto é, trocamos de algoritmo, a ideia central mudou e possibilítou um programa bastante mais eficiente. Isto mostra que vale a pena pensar bastante na melhor maneira de resolver um problema. Sempre existe mais de uma solução e é preciso encontrar a melhor antes de se iniciar a programação da primeira ideia que veio à cabeça.

```
capítulo 7. Aplicações das Técnicas Elementares

program amigosquad;
var n, m, n2, m2, soma_dig_n2, soma_dig_m2, unidade: integer;

begin
  n:= 1;
  while n <= 10000 do
  begin
    (* decompoe n ao quadrado *)
    n2:= n*n;
    soma_dig_n2:= 0;
    while n2 <> 0 do
    begin
      unidade:= n2 mod 10;
      soma_dig_n2:= soma_dig_n2 + unidade;
      n2:= n2 div 10;
    end;

    m:= soma_dig_n2;
    if soma_dig_n2 <= 1000 then (* se estiver na faixa permitida *)
    begin
      (* decompoe soma_dig_n2 *)
      m2:= m*m;
      soma_dig_m2:= 0;
      while m2 <> 0 do
      begin
        unidade:= m2 mod 10;
        soma_dig_m2:= soma_dig_m2 + unidade;
        m2:= m2 div 10;
      end;

      if soma_dig_m2 = n then
        writeln(n, ' e ', m, ' sao amigos quadraticos.');
    end;
  n:= n + 1;
  end.
```

Figura 7.32: Tornando amigos quadráticos mais eficiente.
7.11.3 Cálculo do MDC pela definição

Nesta seção apresentaremos a solução do problema do MDC calculado pela definição. O objetivo é motivar o capítulo seguinte, uma vez que sabemos que existe um algoritmo melhor, estudado na seção 7.3.

O MDC entre dois inteiros $a$ e $b$ foi definido matematicamente na seção 7.2 e envolve a fatoração de ambas as entradas como um produto de números primos. O algoritmo básico, em pseudo-código é apresentado na figura 7.33.

```
begin
    read (a,b);
    mdc := 1;

    (* descobre quantas vezes o 2 divide as duas entradas *)
    cont_a := {numero de vezes que o 2 divide a};
    cont_b := {numero de vezes que o 2 divide b};
    menor_cont := {menor entre cont_a e cont_b};
    mdc := mdc * {2 elevado a potencia menor_cont};
    a := a div mdc;
    b := b div mdc;

    (* repete o processo para todos os impares *)
    primo := 3;
    while (a <> 1) and (b <> 1) do
        begin
            cont_a := {numero de vezes que primo divide a};
            cont_b := {numero de vezes que primo divide b};
            menor_cont := {menor entre cont_a e cont_b};
            mdc := mdc * {primo elevado a potencia menor_cont};
            a := a div mdc;
            b := b div mdc;
            primo := primo + 2;  (* passa para o proximo impar *)
        end;
    writeln (mdc);
end.
```

Figura 7.33: Pseudo-código para o cálculo do MDC pela definição.

O princípio do algoritmo é verificar quantas vezes cada número primo divide as entradas e descobrir qual deles é o menor. O MDC é atualizado então para menor potência deste primo. É preciso separar o caso do 2 dos outros, por motivos que já discutimos. Os valores de $a$ e de $b$ são atualizados, para não haver cálculos inúteis com os ímpares múltiplos de primos que já foram previamente processados.

O programa que implementa este algoritmo não cabe em uma página, por isto é apresentado em duas partes nas figuras 7.34 e 7.35.

Neste ponto deixamos claro ao leitor o motivo da apresentação deste problema no final deste capítulo: este código tem um nível muito alto de trechos de código bastante parecidos. Observamos que, em quatro vezes, se calcula quantas vezes um
CAPÍTULO 7. APLICAÇÕES DAS TÉCNICAS ELEMENTARES

dado primo \( p \) divide um número \( n \). Ainda, a atualização do MDC também aparece em dois trechos diferentes mas bastante similares.

O reaproveitamento de código é uma das motivações para o uso de subprogramas nas linguagens de alto nível. Mas não é a única, existem outras motivações. No próximo capítulo vamos introduzir as importantes noções de procedure e function em Pascal, e poderemos reescrever o código acima com muito mais elegância.

```
program mdc_por_definicao;
var i, a, b, mdc, cont_a, cont_b, menor_cont, primo: integer;
begin
    (* inicialização das variáveis principais *)
    read (a,b);
    mdc:= 1;

    (* descobre quantas vezes o 2 divide as duas entradas *)
    cont_a:= 0;
    while a mod 2 = 0 do
    begin
        cont_a:= cont_a + 1;
        a:= a div 2;
    end;

    cont_b:= 0;
    while b mod 2 = 0 do
    begin
        cont_b:= cont_b + 1;
        b:= b div 2;
    end;

    (* descobre qual dos contadores é o menor *)
    if cont_a <= cont_b then
        menor_cont:= cont_a
    else
        menor_cont:= cont_b;

    (* atualiza o mdc para o 2 *)
    i:= 1;
    while i <= menor_cont do
    begin
        mdc:= mdc * 2;
        i:= i + 1;
    end;
end;
```

Figura 7.34: Calcula MDC entre \( a \) e \( b \) pela definição (caso primo=2).
7.11. REFINAMENTOS SUCESSIVOS

primo:= 3;
while \((a <> 1)\) and \((b <> 1)\) do
begin
    cont\_a:= 0;
    while \(a \mod primo = 0\) do
    begin
        cont\_a:= cont\_a + 1;
        a:= a \div primo;
    end;

    cont\_b:= 0;
    while \(b \mod primo = 0\) do
    begin
        cont\_b:= cont\_b + 1;
        b:= b \div primo;
    end;

    \(* descobre qual dos contadores é o menor *)
    if cont\_a <= cont\_b then
        menor\_cont:= cont\_a
    else
        menor\_cont:= cont\_b;

    \(* atualiza o mdc para o primo impar da vez *)
    i:= 1;
    while \(i <= menor\_cont\) do
    begin
        mdc:= mdc * primo;
        i:= i + 1;
    end;

    \(* passa para o próximo impar *)
    primo:= primo + 2;
end;

\(* imprime o resultado final *)
writeln (mdc);
end.

Figura 7.35: Calcula MDC entre \(a\) e \(b\) pela definição (caso primo é ímpar).
7.12 Exercícios

1. Faça um programa em *Pascal* que leia um número de três dígitos e construa outro número de quatro dígitos com a seguinte regra: a) os três primeiros dígitos, contados da esquerda para a direita, são iguais aos do número dado; b) o quarto dígito é um dígito de controle calculado da seguinte forma: primeiro dígito + 3*segundo dígito + 5*terceiro dígito; o dígito de controle é igual ao resto da divisão dessa soma por 7. Ao final imprima o número obtido.

2. Faça um programa em *Pascal* que leia um número inteiro de cinco dígitos representando um número binário, determinar seu valor equivalente em decimal. Por exemplo, se a entrada for 10001, a saída deve ser 17.

3. Considere o programa feito para resolução do cálculo do número neperiano (seção 7.9.1). Quantas operações de multiplicação são executadas no seu programa?
   
   (a) Considerando 20 termos.
   (b) Considerando N termos.

4. Considere a progressão geométrica 1, 2, 4, 8, 16, 32, ... e um inteiro positivo N. Faça um programa em *Pascal* que imprima os N primeiros termos desta PG e a soma deles.

5. Faça um programa em *Pascal* que leia um número inteiro N, tabelê N[k] para k variando de 1 até N. Considere que, por definição, \( X[k] = X(X - 1)(X - 2)(X - 3)...(X - k + 1) \), X sendo um número real, k um natural diferente de zero e \( X[0] = 1 \). Observe que se \( X = N = k \), então \( N[N] = N! \).

6. Considere o conjunto \( C \) de todos os números inteiros com quatro algarismos distintos, ordenados segundo seus valores, em ordem crescente:

   \[ C = \{1023, 1024, 1025, 1026, 1027, 1028, 1029, 1032, 1034, 1035, \ldots \} \]

   Faça um programa em *Pascal* que leia um número \( N \) pertencente a este conjunto e imprima a posição deste número no conjunto.

   Exemplos:
   
   - Entrada: 1026
     Saída: 4
   - Entrada: 1034
     Saída: 9
   - Entrada: 9876
     Saída: 4536
   - Entrada: 1243
     Saída: 72

7. Faça um programa em *Pascal* que leia um inteiro positivo \( N \) e dada uma sequência de \( N \) números reais \( x_1, \ldots, x_n \) calcule o quociente da soma dos reais pelo seu produto. Isto é:

   \[
   q = \frac{\sum_{i=1}^{N} x_i}{\prod_{i=1}^{N} x_i}
   \]
Como não pode haver divisão por zero, seu programa deve parar tão logo esta situação seja verificada indicando uma mensagem apropriada para o usuário.

8. Em *Pascal* o tipo *CHAR* é enumerável, e portanto está na classe dos tipos chamados de *ordinais*, conforme o guia de referência da linguagem estudado em aula. A ordem de cada caractere é dada pela tabela ASCII. Assim é possível, por exemplo, escrever trechos de código tais como:

```pascal
IF 'A' > 'B' THEN
  WRITE ('A eh maior que B')
ELSE
  WRITE ('A não eh maior que B');
```

que produziria a mensagem “A não eh maior que B”, pois na tabela ASCII o símbolo “A” tem ordem 64 enquanto que “B” tem ordem 65.

Ou ainda:

```pascal
FOR i:= 'a' TO 'z' DO
  WRITE (i);
```

que produziria como saída “abcdefghijklmnopqrstuvwxyz”.

Faça um programa em *Pascal* que leia seu nome completo (nomes completos em geral) constituídos por apenas letras maiúsculas entre “A” e “Z” e espaços em branco terminadas em “.” e que retorne o número de vogais e consoantes neste nome. Exemplos:

**Entrada:** FABIANO SILVA.

**Saída:**

Vogais: 6
Consoantes: 6

**Entrada:** MARCOS ALEXANDRE CASTILHO.

**Saída:**

Vogais: 9
Consoantes: 14

9. Faça um programa em *Pascal* que leia um inteiro positivo *n* e imprima a soma dos *n* primeiros termos da série:

$$
\frac{1000}{1} - \frac{997}{2} + \frac{994}{3} - \frac{991}{4} + \ldots
$$
10. Faça um programa em Pascal que calcule e imprima o valor de $S$: 

$$S = \frac{37 \times 38}{1} + \frac{36 \times 37}{2} + \frac{35 \times 36}{3} + \cdots + \frac{1 \times 2}{37}$$

11. Faça um programa em Pascal que calcule e imprima o resultado da seguinte série:

$$S = \frac{x^0}{2!} - \frac{x^4}{6!} + \frac{x^8}{10!} - \frac{x^{12}}{14!} + \frac{x^{16}}{18!} - \cdots$$

12. Faça um programa em Pascal que leia um inteiro $N$ e o imprima como um produto de primos. Exemplos: $45 = 3 \times 3 \times 5$. $56 = 2 \times 2 \times 2 \times 7$.

13. Faça um programa em Pascal que calcule e escreva o valor de $S$ assim definido:

$$S = \frac{1}{1!} - \frac{2}{2!} + \frac{4}{3!} - \frac{8}{2!} + \frac{16}{11!} - \frac{32}{2!} + \frac{64}{3!} - \cdots$$

14. Faça um programa em Pascal que calcule e imprima o valor de $f(x)$, onde $x \in \mathbb{R}$ é lido no teclado e:

$$f(x) = \frac{5x}{2!} - \frac{6x^2}{3!} + \frac{11x^3}{4!} - \frac{12x^4}{5!} + \frac{17x^5}{6!} - \frac{18x^6}{7!} + \cdots$$

O cálculo deve parar quando $\text{abs}(f(x_{n+1}) - f(x_n)) < 0.000000001$, onde $\text{abs}(x)$ é a função em Pascal que retorna o valor absoluto de $x$.

15. Sabe-se que o valor do cosseno de 1 (um) radiano pode ser calculado pela série infinita abaixo:

$$\cos(x) = \frac{1}{0!} - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \cdots$$

Faça um programa em Pascal que calcule o valor do cosseno de $x$ obtido pela série acima considerando somente os primeiros 14 termos da mesma.

16. O número áureo $\varphi$ (1.6180339...) pode ser calculado através de expressões com séries de frações sucessivas do tipo:

$$\varphi_1 = 1 + \frac{1}{1} = 2$$

$$\varphi_2 = 1 + \frac{1}{1 + \frac{1}{1}} = 1.5$$

$$\varphi_3 = 1 + \frac{1}{1 + \frac{1}{1 + \frac{1}{1}}} = 1.666$$

$$\varphi_4 = 1 + \frac{1}{1 + \frac{1}{1 + \frac{1}{1 + \frac{1}{1}}}} = 1.6$$
onde \( \varphi \), indica a aproximação do número áureo com \( i \) frações sucessivas. Estes valores variam em torno do número áureo, sendo maior ou menor alternadamente, mas sempre se aproximando deste quando o número de frações cresce.

Faça um programa em Pascal que leia um número \( N \) e imprima o valor da aproximação do número áureo \( \varphi_N \), que usa uma série de \( N \) frações sucessivas.

17. (*) Dizemos que uma sequência de inteiros é \( k \)-alternante se for composta alternadamente por segmentos de números pares de tamanho \( k \) e segmentos de números ímpares de tamanho \( k \).

Exemplos:

A sequência 1 3 6 8 9 11 2 4 1 7 6 8 é 2-alternante.
A sequência 2 1 4 7 8 9 12 é 1-alternante.
A sequência 1 3 5 é 3-alternante.

Faça um programa Pascal que verifique se uma sequência de tamanho \( n \) é 10-alternante. O programa deve ler \( n \), o tamanho da sequência, no início do programa e aceitar somente valores múltiplos de 10. A saída do programa deve ser a mensagem “A sequência eh 10-alternante” caso a sequência seja 10-alternante e “A sequência não eh 10-alternante”, caso contrário.

18. Faça um programa em Pascal que leia do teclado dois números inteiros positivos e imprima na saída um único número inteiro que é a soma dos dois primeiros. Entretanto, seu programa não pode utilizar o operador de soma (+) da linguagem para somar os dois inteiros lidos em uma única operação. Outrossim, o programa deve implementar a soma dos números dígito a dígito, iniciando pelo menos significativo até o mais significativo, considerando o “vai um”, conforme costumamos fazer manualmente desde o ensino fundamental.

<table>
<thead>
<tr>
<th>Exemplo 1</th>
<th>Exemplo 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>11 (&quot;vai um&quot;)</td>
<td>1111 (&quot;vai um&quot;)</td>
</tr>
<tr>
<td>40912 (primeiro número)</td>
<td>52986 (primeiro número)</td>
</tr>
<tr>
<td>1093 (segundo número)</td>
<td>1058021 (segundo número)</td>
</tr>
<tr>
<td>------</td>
<td>------</td>
</tr>
<tr>
<td>42005 (soma)</td>
<td>1111007 (soma)</td>
</tr>
</tbody>
</table>

19. Um agricultor possui 1 (uma) espiga de milho. Cada espiga tem 150 grãos, e cada grão pesa 1g (um grama). Faça um programa em Pascal que determine quantos anos serão necessários para que o agricultor colha mais de cem toneladas de milho (1T = 1000Kg, 1Kg = 1000g), sendo que:

- A cada ano ele planta todos os grãos da colheita anterior
- Há uma única colheita por ano
- 10% (dez por cento) dos grãos não germina (morre sem produzir)
- Cada grão que germina produz duas espigas de milho
Assuma que a quantidade de terra disponível é sempre suficiente para o plantio.

20. Modifique a questão anterior acrescentando na simulação os seguintes fatos:

- Há 8 (oito) CASAIS de pombas (16 pombas) que moram na propriedade do agricultor.
- Cada pomba come 30 grãos por dia, durante os 30 dias do ano em que as espigas estão formadas antes da colheita;
- A cada ano, cada casal gera 2 novos casais (4 pombas), que se alimentarão e reproduzirão no ano seguinte;
- Uma pomba vive três anos;

Ao final do programa, imprima também o número de pombas que vivem na propriedade quando o agricultor colher mais de 100T de milho.

21. Considere um número inteiro com 9 dígitos. Suponha que o último dígito seja o “dígito verificador” do número formado pelos 8 primeiros. Faça um programa em Pascal que leia uma sequência de números inteiros terminada em zero e imprima os números que não são bem formados, isto é, aqueles que não satisfazem o dígito verificador. O zero não deve ser processado pois serve somente para marcar o final da entrada de dados.

Implemente o seguinte algoritmo para gerar o dígito verificador:

Conforme o esquema abaixo, cada dígito do número, começando da direita para a esquerda (menos significativo para o mais significativo) é multiplicado, na ordem, por 2, depois 1, depois 2, depois 1 e assim sucessivamente.

Número exemplo: 261533-4

Ao invés de ser feita a somatória das multiplicações, será feita a somatória dos dígitos das multiplicações (se uma multiplicação der 12, por exemplo, será somado 1 + 2 = 3).

A somatória será dividida por 10 e se o resto (módulo 10) for diferente de zero, o dígito será 10 menos este valor. O zero não deve ser processado pois serve somente para marcar o final da entrada de dados.
22. Faça um programa *Pascal* que leia dois valores inteiros positivos A e B. Se A for igual a B, devem ser lidos novos valores até que sejam informados valores distintos. Se A for menor que B, o programa deve calcular e escrever a soma dos números ímpares existentes entre A(inclusive) e B(inclusive). Se A for maior que B, o programa deve calcular e escrever a média aritmética dos múltiplos de 3 existentes entre A(inclusive) e B(inclusive).

23. Faça um programa em *Pascal* que leia uma sequência de números inteiros terminada por zero e imprima quantos segmentos de números iguais consecutivos compõem essa sequência.

Ex.: A sequência 2,2,3,3,5,1,1 é formada por 4 segmentos de números iguais.

24. Faça um programa em *Pascal* que imprima a seguinte sequência de números: 1, 1, 2, 2, 3, 3, 3, 4, 4, 4, 5, 5, 5, 6, 6, 6, 7, 7, 7, 7, 8, 8, 8, 8, 8, 8, ... Este programa não tem entrada de dados.

25. Faça um programa em *Pascal* que leia um inteiro N e o imprima como um produto de primos. Exemplos: 45 = 3 × 3 × 5. 56 = 2 × 2 × 2 × 7.

26. Faça um programa em *Pascal* que leia um número inteiro N, escreva o maior divisor de N que é uma potência de um dos números primos fatorados. Ex:

   \[
   \begin{align*}
   N &= 45 = 3^2.5^1 \text{ escreve } 9 = 3^2 \\
   N &= 145 = 5^2.7^1 \text{ escreve } 25 = 5^2 \\
   N &= 5616 = 2^4.3^3.13 \text{ escreve } 27 = 3^3
   \end{align*}
   \]
Capítulo 8

Funções e procedimentos

Até agora vimos as noções básicas de algoritmos, fundamentalmente como funciona o computador de verdade (modelo Von Neumann) e como isto pode ter uma representação em um nível mais alto, guardadas as limitações da máquina.

A maneira de se resolver problemas do ponto de vista algorítmico envolve a elaboração de construções com somente quatro tipos de estruturas de controle de fluxo: comandos de entrada e saída, comandos de atribuição, comandos de repetição e comandos de desvio condicional, além do uso de expressões lógicas e aritméticas. Para a efetiva programação ainda é necessário dominar a arte de escrever os algoritmos em função das limitações dos compiladores.

O problema é que à medida em que os problemas exigem códigos com muitas linhas, os programas gerados vão se tornando cada vez mais complexos tanto para serem desenvolvidos quanto para serem mantidos em funcionamento. O exemplo do final do capítulo anterior dá uma ideia dessa dificuldade.

Por isto, os programas devem ser construídos de maneira a facilitar o trabalho dos programadores e analistas. É preciso que os códigos sejam elaborados com partes bem definidas, em módulos que contenham pedaços coerentes do problema geral que se está tentando modelar. As noções de funções e procedimentos são o caminho para se construir códigos elegantes, robustos e de fácil manutenção.

8.1 Motivação

Os procedimentos e funções são nada mais do que subprogramas, isto é, pedaços de programas dentro de programas. Mas, bem explorados, permitem a construção de códigos altamente elaborados. Existem três motivações para se usar subprogramas:

- modularidade;
- reaproveitamento de código;
- legibilidade do código.

Vamos detalhar cada tópico nos próximos itens.
8.1.1 Modularidade

A noção de modularidade é relacionada com a capacidade de se escrever programas em pedaços de código que executam operações bem definidas. Cada módulo possui variáveis e estruturas próprias, independentes do restante do programa. A ideia é que modificações em trechos de código (necessárias para manutenção e continuidade de desenvolvimento) não causem reflexos no comportamento do resto do programa.

É fácil conceber um programa dividido em três partes: entrada de dados, cálculos e impressão dos resultados. Uma entrada de dados textual poderia ser modificada para outra em modo gráfico sem que o pedaço que faz os cálculos perceba a modificação. Idem para a saída de dados. Mesmo nos cálculos, pode-se mudar toda uma estrutura do programa e garantir que a entrada e saída vão continuar a se comportar como antes. Isto é tarefa árdua sem o uso de funções e procedimentos.

É importante notar que a linguagem Pascal não fornece todos os meios para se implementar um programa totalmente modular, mas é o suficiente para os estudantes em primeiro curso de computação perceberem a importância do conceito. A evolução destas ideias deu origem ao conceito de Programação Orientada a Objetos, hoje na moda. Mas isto é tema para outras disciplinas.

8.1.2 Reaproveitamento de código

Vez por outra nos deparamos com situações onde temos que escrever códigos muito, mas muito, parecidos em trechos diferentes do programa. As vezes a diferença de um para outro é questão de uma ou outra variável que muda. Ocorre frequentemente que o trecho é exatamente o mesmo.

Então faz sentido que possamos estruturar o código repetido de maneira a constituir um subprograma e, no programa propriamente dito, fazer o código do subprograma ser executado para diferentes valores de variáveis. Isto provoca uma grande economia de código escrito, ao mesmo tempo em que facilita a manutenção do programa.

8.1.3 Legibilidade

Os dois aspectos acima, somados com o bom uso de nomes apropriados para os identificadores, identação e uso racional de comentários no código, implicam necessariamente em um código legível, isto é, compreensível para quem o lê e até mesmo para quem o escreveu.

De fato, é comum alguém fazer um programa, as vezes simples, e depois de alguns meses ler o código e não entender o que lá está escrito. Um código legível permite uma rápida compreensão e viabiliza sua manutenção, correção e expansão, seja pelo próprio programador ou por outras pessoas.

---

1Recomendamos a leitura do miniguia da linguagem Pascal, disponível no site oficial da disciplina CI055.
8.2. NOÇÕES FUNDAMENTAIS

8.1.4 Comentário adicional
Nesta seção, vamos tentar convencer o aprendiz a usar bem e a dar valor a estas noções, estudando exemplos simples, mas didáticos.

8.2 Noções fundamentais
Existem três conceitos que devem ser dominados pelo estudante:

1. quando usar função e quando usar procedimento?
2. quando usar variáveis locais ou variáveis globais?
3. quando usar passagem de parâmetros por valor ou por referência?

Nas próximas seções vamos detalhar cada um destes itens.

8.2.1 Exemplo básico
Vamos tomar como base um programa bem simples, estudado nas primeiras aulas desta disciplina. Trata-se do problema de se ler uma sequência de valores inteiros terminada por zero e que imprima somente aqueles que são pares.

Quando resolvemos este problema, o código foi escrito como na figura 8.1.

```
program imprime_pares;
var a: integer;
begin
  read (a);
  while a <> 0 do
    begin
      if a mod 2 = 0 then
        writeln (a);
      read (a);
    end;
end.
```

Figura 8.1: Programa que imprime os números da entrada que são pares.

8.2.2 O programa principal
É o código do programa propriamente dito. Tem início no `begin` e término no `end`.

No exemplo anterior, são todos os comandos que aparecem no programa, desde o de leitura da variável `a` até o `end`; do comando `while`. O resto que lá aparece é o cabeçalho do programa e a declaração de variáveis globais.
8.2.3 Variáveis globais

São todas as variáveis declaradas logo após o cabeçalho do programa, antes do begin do programa principal.

Como sabemos, variáveis são abstrações de endereços de memória. As variáveis globais são endereços visíveis em todo o programa, mesmo nos subprogramas, como veremos logo mais.

No exemplo acima, existe uma única variável global, ela tem o identificador \( a \) e é do tipo \( integer \).

8.2.4 Funções

No miolo do programa exemplo, existe um trecho onde se lê: \( a \mod 2 = 0 \). Hoje sabemos que isto é uma expressão booleana que calcula o resto da divisão inteira de \( a \) por 2, se o resultado for zero então \( a \) é par, senão é ímpar. Mas, esta expressão poderia ser bem mais complexa, exigindo muitas linhas de código, por exemplo, se quiséssemos imprimir os números que fossem primos ao invés dos pares.

O importante é que a expressão booleana resulta em um valor do tipo booleano. O mesmo efeito pode ser conseguido com uma função que resulta em um valor do tipo booleano.

Isto pode ser feito definindo-se um subprograma que é chamado pelo programa principal e que recebe os dados necessários para os cálculos (os parâmetros). Os cálculos devem computar corretamente se o número enviado pelo programa principal é par. Se for, de alguma maneira deve retornar um valor \( true \) para quem chamou. Se não for, deve retornar \( false \).

O que acaba de ser escrito estabelece uma espécie de “contrato” entre o programa principal e o subprograma. Em termos de linguagem de programação, este contrato é denominado de protótipo ou assinatura da função. Ele é constituído por três coisas:

- O nome da função;
- A lista de parâmetros, que são identificadores tipados (no caso da linguagem Pascal);
- O tipo do valor de retorno contendo o cálculo feito na função.

Para o problema em questão, vamos assumir que poderíamos estabelecer o seguinte protótipo para a função que calcula se um dado número é par retornando \( true \) em caso positivo e \( false \) em caso contrário. Vejamos como fica:

```pascal
function a_eh_par: boolean;
```

A palavra reservada \( function \) é para avisar o programa que se trata de uma função. O identificador \( a_eh_par \) procura representar o fato de que a variável \( a \) pode ser par.

O código da função não importa muito no momento, mas somente com o protótipo é possível reescrever o programa exemplo como apresentado na figura 8.2.
8.2. NOÇÕES FUNDAMENTAIS

```pascal
program imprime_pares;
var a: integer;

(* funcao que calcula se a variavel global a eh par *)
function a_eh_par: boolean;
begin    (* codigo da funcao, ainda nao escrito por razoes didaticas *)
end;

begin (* programa principal *)
    read (a);
    while a <> 0 do
    begin
        if a_eh_par then
            writeln (a);
        read (a);
    end;
end.
```

Figura 8.2: Programa que imprime os números da entrada que são pares.

Desde que o código da função seja corretamente implementado, o programa principal continua funcionando! Além disto é um código mais legível, já que é mais simples para alguém ler o programa principal e perceber o significado do identificador denominado a_eh_par ao invés do obscuro “a mod 2 = 0”.

Obviamente é necessário em algum momento se escrever o código da função. Vamos mostrar oito maneiras diferentes de se devolver o valor correto para o programa principal.

<table>
<thead>
<tr>
<th>(* primeira versao da funcao *)</th>
<th>(* segunda versao da funcao *)</th>
</tr>
</thead>
<tbody>
<tr>
<td>if a mod 2 = 0 then</td>
<td>if a mod 2 &lt;&gt; 0 then</td>
</tr>
<tr>
<td>a_eh_par:= true</td>
<td>a_eh_par:= false</td>
</tr>
<tr>
<td>else</td>
<td>else</td>
</tr>
<tr>
<td>a_eh_par:= false;</td>
<td>a_eh_par:= true;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(* terceira versao da funcao *)</th>
<th>(* quarta versao da funcao *)</th>
</tr>
</thead>
<tbody>
<tr>
<td>if a mod 2 &lt;&gt; 1 then</td>
<td>if a mod 2 = 1 then</td>
</tr>
<tr>
<td>a_eh_par:= true</td>
<td>a_eh_par:= false</td>
</tr>
<tr>
<td>else</td>
<td>else</td>
</tr>
<tr>
<td>a_eh_par:= false;</td>
<td>a_eh_par:= true;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(* quinta versao da funcao *)</th>
<th>(* sexta versao da funcao *)</th>
</tr>
</thead>
<tbody>
<tr>
<td>a_eh_par:= false;</td>
<td>a_eh_par:= true;</td>
</tr>
<tr>
<td>if a mod 2 = 0 then</td>
<td>if a mod 2 = 1 then</td>
</tr>
<tr>
<td>a_eh_par:= true</td>
<td>a_eh_par:= false</td>
</tr>
</tbody>
</table>
CAPÍTULO 8. FUNÇÕES E PROCEDIMENTOS

Em tempo, um detalhe da linguagem Pascal. Segundo a definição da linguagem, estabelecida por seu autor Niklaus Wirth ainda nos anos 1970, a maneira como o valor do retorno é feita para o programa principal exige que o nome da função apareça pelo menos uma vez no código da função do lado esquerdo de um comando de atribuição. Isto funciona diferente em outras linguagens de programação, para isto, se o estudante resolver mudar de linguagem, deve observar o respectivo guia de referência.

Ainda um comentário adicional sobre a linguagem Pascal, a função é executada até o final, isto é, até encontrar o comando end; que a termina. Isto pode ser diferente em outras linguagens.

8.2.5 Parâmetros por valor

Conforme dissemos, atendemos a questão da modularidade ao usarmos a função a_eh_par. Mas não atendemos a outra: a questão do reaproveitamento de código.

De fato, suponhamos que o enunciado tivesse estabelecido que seriam dados como entrada pares de números a e b e para imprimir os que fossem pares. Na versão atual do programa, teríamos que escrever uma outra função de nome provável b_eh_par que teria o código absolutamente idêntico ao da função a_eh_par exceto pela troca da variável a pela variável b. Isto é inadmissível em programação de alto nível!

Logo, deve existir uma maneira melhor para se programar a função. Felizmente existe: basta passar um parâmetro, isto é, basta informar à função que os cálculos serão feitos para um dado número inteiro, não importando o nome da variável que contém o valor sobre o qual será feito o cálculo da paridade. Isto é conseguido mudando-se o protótipo da função para o seguinte:

function eh_par (n: integer): boolean;

Em outras palavras, a função vai receber um número inteiro, no caso denominado n (mas poderia ser a, b ou qualquer outro identificador, o importante é que seja do tipo integer. O tipo do retorno é o mesmo, isto é, boolean.

Com isto, conseguimos escrever o programa (já com a função recebendo parâmetros por valor), da maneira como está apresentado na figura 8.3.

Esta maneira de passar parâmetros caracteriza uma passagem por valor, também conhecida como passagem de parâmetros por cópia. O que ocorre é que o identificador n da função recebe uma cópia do valor da variável a do programa principal. As alterações em n são feitas nesta cópia, mantendo-se intactos os dados da variável a no programa principal. Isto ocorre pois esta Cópia é feita em área separada de memória, denominada pilha.
8.2. NOÇÕES FUNDAMENTAIS

```pascal
program imprime_pares;
var a: integer;

(* funcao que calcula se a variavel global a eh par *)
function eh_par (n: integer): boolean;
begin
  if n mod 2 = 0 then
    eh_par:= true
  else
    eh_par:= false;
end;

begin (* programa principal *)
  read (a);
  while a <> 0 do begin
    if eh_par (a) then
      writeln (a);
    read (a);
  end;
end.

Figura 8.3: Programa que imprime os números da entrada que são pares.

8.2.6 Parâmetros por referência

Para passar parâmetros por referência, o protótipo da função difere ligeiramente daquele exibido acima. A chamada é assim:

```pascal
function eh_par (var n: integer): boolean;
```

A diferença sintática é caracterizada pela palavra `var` antes do nome do identificador do parâmetro. Pode parecer sutil, mas com uma semântica totalmente diferente daquela da passagem de parâmetros por valor.

O que antes era uma cópia, agora é uma referência ao endereço da variável do programa principal. Isto é, no momento da ativação da função, o identificador `n` da função é associado com o mesmo endereço da variável `a` no programa principal. Consequentemente, qualquer alteração associada a `n` na função provocará alteração do valor da variável `a` no programa principal.

Vejamos na figura 8.4 como fica a modificação no programa em estudo do ponto de vista meramente sintático. Deixamos o ponto de vista semântico para ser explicado no próximo exemplo.

---

2Com todo o respeito ao Niklaus Wirth, usar a palavra `var` para este fim não foi uma boa escolha, pois para um principiante é fácil confundir uma passagem de parâmetros por referência com uma declaração de uma variável, o que não é definitivamente o caso aqui.
program imprime_pares;
var a: integer;

(* funcao que calcula se a variavel global a eh par *)
function eh_par (var n: integer): boolean;
begin
  if n mod 2 = 0 then
    eh_par:= true
  else
    eh_par:= false;
end;

begin (* programa principal *)
  read (a);
  while a <> 0 do
    begin
      if eh_par (a) then
        writeln (a);
      read (a);
    end;
end.

Figura 8.4: Versão com parâmetros por referência.

8.2.7 Procedimentos

Um procedimento difere de uma função basicamente pois não tem um valor de retorno associado. Isto faz com que ele se comporte como um comando extra da linguagem ao passo que a função tem um comportamento mais parecido com o de uma expressão aritmética ou booleana. Um protótipo para um procedimento é definido então com base em duas informações:

1. o identificador do procedimento (nome);
2. a lista de parâmetros (que podem ser por valor ou referência).

Para explicar corretamente a noção de procedimentos, e também de variáveis locais, é importante mudar de exemplo. Vamos considerar o problema de se ler dois números reais \(x\) e \(y\) do teclado, fazer a troca dos conteúdos e depois imprimir.

Vamos considerar o seguinte protótipo para um procedimento que recebe os dois valores \(x\) e \(y\) e tem a finalidade de realizar a troca dos conteúdos destas variáveis:

procedure troca (var a, b: real);

Observe que, assim como no caso da função, não importa muito o nome dos identificadores dos parâmetros, mas importa que eles definam conteúdos do tipo REAL.

Podemos então tentar escrever um programa que leia dois valores e os imprima trocados, conforme ilustrado na figura 8.5.

\[^3\text{Um exercício interessante é passar os parâmetros por valor e compreender o efeito disso.}\]
8.2. NOÇÕES FUNDAMENTAIS

Figura 8.5: Versão com parâmetros por referência.

Este programa usa uma variável global (temp) para auxiliar a troca, o que não faz muito sentido. Os subprogramas devem funcionar independentemente de variáveis globais.

8.2.8 Variáveis locais

Variáveis locais são declaradas nos subprogramas e têm escopo local, isto é, elas só são conhecidas durante a execução do subprograma. Consequentemente não interferem no programa principal. O programa modificado da figura 8.6 faz uso da variável local temp e torna o código mais robusto.

Figura 8.6: Versão com uma variável local.
8.3 Alguns exemplos

Nesta seção vamos implementar dois problemas simples para exercitar.

8.3.1 Calculando dígito verificador

Vamos fazer um programa que recebe um número de \(N\) dígitos \((N > 0)\), sendo o último deles o “dígito verificador” do número formado pelos \(N−1\) primeiros. Devemos calcular se o dígito verificador fornecido pelo usuário está correto segundo o esquema de cálculo seguinte: cada dígito do número, começando da direita para a esquerda (menos significativo para o mais significativo) é multiplicado, na ordem, por 1, depois 2, depois 1, depois 2 e assim sucessivamente. O número de entrada do exemplo é 261533-4.

```
+---+---+---+---+---+---+ +---+
| 2 | 6 | 1 | 5 | 3 | 3 | - | 4 |
+---+---+---+---+---+---+ +---+
  |   x2    x2 x1 x2 x1   |
  |   =4   =6 =5 =6 =3   |
  +-----------------------> = 26
```

Como 26 tem dois dígitos, vamos repetir o processo acima até gerarmos um número de um único dígito. Assim:

```
+-----+
| 2   |
+-----+
  |     |
  | x2  x1 |
  | =4   =6 |
  +------> = 10
```

Como 10 ainda tem dois dígitos, o algoritmo roda ainda mais uma vez:

```
+-----+
| 1   |
+-----+
  |     |
  | x2  x1 |
  | =2   =0 |
  +------> = 2
```

Assim, o dígito verificador calculado (2) difere daquele fornecido (4) e o programa deveria acusar o erro. O programa da figura 8.7 ilustra uma possível solução.
program digitoverificador;
var numero, n: longint;
    dv_informado, dv_correto: integer;

procedure le (var n: longint);
(* este codigo garante que o numero lido eh positivo *)
begin
    repeat
        read (n);
        until n > 0;
    end;

function extrai_numero (n: longint): longint;
begin
    extrai_numero:= n div 10;
end;

function extrai_dv (n: longint): longint;
begin
    extrai_dv:= n mod 10;
end;

function calcula_dv_correto (n: longint): integer;
var soma, mult, ultimo: integer;
begin
    repeat
        soma:= 0;
        mult:= 1;
        while n <> 0 do
            begin
                ultimo:= n mod 10;
                n:= n div 10;
                soma:= soma + mult * ultimo;
                if mult = 1 then
                    mult:= 2
                else
                    mult:= 1;
            end;
        n:= soma;
    until (n >= 0) and (n <= 9);
    calcula_dv_correto:= soma;
end;

begin (* programa principal *)
    le (numero);
    n:= extrai_numero (numero);
    dv_informado:= extrai_dv (numero);
    dv_correto:= calcula_dv_correto (n);
    if dv_correto <> dv_informado then
        writeln ('digito verificador invalido.');
end.

Figura 8.7: Calculando dígito verificador.
O importante para se observar neste código é a clareza do algoritmo no programa principal. O leitor pode acompanhar este trecho e perceber claramente as diversas etapas em uma linguagem de bastante alto nível: leitura do número, separação deste em duas partes, uma contendo os primeiros dígitos a outra contendo o dv de entrada. Em seguida o cálculo do dígito verificador correto e finalmente a comparação dos dados calculados com o de entrada, gerando a mensagem final.

No programa principal pode-se ignorar completamente como são feitas todas as operações nas funções e procedimentos: não importa como os dados são lidos, nem como os dígitos são separados, e muito menos como é feito o cálculo do dígito verificador correto. No programa principal o importante é o algoritmo em alto nível.

É claro que em algum momento será necessário escrever código para cada uma das funções e procedimentos, mas quando isto for feito o programador estará resolvendo um subproblema de cada vez, o que facilita muito a construção do código para o problema global.

Por exemplo, a leitura poderia ser feita em uma interface gráfica ou textual. Foi escolhida nesta versão uma interface textual, mas que permite testes de consistência dos dados de entrada. Isto, feito separadamente, mantém o código global independente. No caso, o programador garante que o número lido será sempre maior do que zero. Importante notar que, para leitura de dados, o parâmetro tem que ser passado por referência, e não por valor, senão o valor seria lido em uma cópia da variável do programa principal.

Considerações similares são feitas para as três funções. Todas possuem código próprio e fazem com que a atenção do programador fique voltada exclusivamente para o subproblema da vez. Desde que os protótipos das funções estejam corretamente especificados, não há problema em se alterar o código interno. Notamos também que foram usadas variáveis locais para auxílio no cálculo do dígito verificador. Tudo para o bem da clareza do código principal. Se um dia mudarem a definição de como se calcula o dígito verificador, basta alterar a função que o programa principal continuará a funcionar corretamente.

Na última função é importante notar a passagem de parâmetros por valor. Isto permitiu o laço que controla o laço interno usar o próprio parâmetro como condição de parada, pois ele é dividido por 10 a cada iteração. Se o parâmetro fosse passado por referência isto não poderia ser feito, pois estariamos estragando o valor da variável que será ainda usada no programa principal.

8.3.2 Calculando raízes de equações do segundo grau

Para reforçarmos os conceitos em estudo, consideremos aqui o problema de se ler os coeficientes $a$, $b$ e $c$ que definem uma equação do segundo grau $ax^2 + bx + c = 0$ e imprimir as raízes calculadas pela fórmula de Bhaskara. O programa deve imprimir mensagens corretas no caso de não haverem raízes reais bem como não deve aceitar entradas cujo valor para o coeficiente $a$ sejam nulas. O programa da figura [8.8] contém o código que resolve este problema.

A figura [8.9] ilustra o programa principal modificado para se dar a ideia de que as funções se comportam como expressões aritméticas, ao contrário dos procedimen-
program bhaskara_v2;
var a, b, c, delta, x1, x2: real;

procedure ler (var a, b, c: real);
begin
    repeat
        read (a, b, c);
    until a <> 0; (* garante que a equacao eh de fato do segundo grau *)
end;

function calcula_delta (a, b, c: real): real;
begin
    calcula_delta:= b*b - 4*a*c;
end;

function menor_raiz (a, b, delta: real): real;
begin
    menor_raiz:= (-b - sqrt(delta))/(2*a);
end;

function maior_raiz (a, b, delta: real): real;
begin
    maior_raiz:= (-b + sqrt(delta))/(2*a);
end;

begin (* programa principal *)
    ler (a, b, c); (* garante-se que a nao eh nulo *)
    delta:= calcula_delta (a, b, c);
    if delta >= 0 then
        begin
            x1:= menor_raiz (a, b, delta);
            writeln (x1);
            x2:= maior_raiz (a, b, delta);
            writeln (x2);
        end
    else
        writeln ('raizes complexas');
end.

Figura 8.8: Calculando raízes de equação do segundo grau.
tos, que se comportam como comandos. De fato, basta observar que as funções são chamadas dentro de um comando de impressão.

```pascal
begin (* programa principal *)
    ler (a, b, c); (* garante-se que a não é nulo *)
    delta:= calcula_delta (a, b, c);
    if delta >= 0 then
        writeln (menor_raiz (a, b, delta), maior_raiz (a, b, delta))
    else
        writeln ('raízes complexas');
end.
```

Figura 8.9: Calculando raízes de equação do segundo grau.

Quando o discriminante é nulo, as duas raízes são iguais, o programa imprime os dois valores sempre. A função que calcula as raízes poderia devolver um inteiro contendo o número de raízes⁴. Esta função é apresentada na figura [8.10].

```pascal
function calcula_raiz (a,b,c: real; var x1, x2: real): integer;
var delta:real;
begin
    delta:= calcula_delta(a,b,c);
    if delta >= 0 then
        begin
            x1:= menor_raiz (a,b,delta);
            x2:= maior_raiz (a,b,delta);
            if delta = 0 then
                calcula_raiz:= 1
            else
                calcula_raiz:= 2;
        end
    else
        calcula_raiz:= 0;
end;
```

Figura 8.10: Função que calcula as raízes de equação do segundo grau.

Notem que as raízes são retornadas pela função ao programa principal pelo uso de dois parâmetros por referência. Esta função usa outras funções já definidas anteriormente. Isto torna o código modular e elegante. O programa principal que utiliza esta função está na figura [8.11].

⁴Agradecemos ao prof. Luis Carlos Erpen de Bona pela dica.
8.3. ALGUNS EXEMPLOS

begin
    ler (a, b, c);
    numraizes:=calcula_raiz(a, b, c, x1, x2);
    if numraizes > 0 then
        begin
            writeln(x1);
            if numraizes = 2 then
                writeln(x2);
        end
    else
        writeln('raizes complexas');
end.

Figura 8.11: Versão final do programa.

8.3.3 Cálculo do MDC pela definição

Nesta seção vamos revisitar o cálculo do MDC pela definição estudado na seção 7.11.3. Deixamos propositualmente em aberto a questão de que aquele código continha trechos de código similares que tinham que ser repetidos por causa de um número de entrada variante no programa.

De fato, naquele código exibido na figura 7.34 o cálculo para saber se o número $a$ era par difere do cálculo para saber se o número $b$ é par por causa do valor de $a$ ou $b$. O mesmo ocorre para o código da figura 7.35 no caso de números ímpares.

Os quatro trechos de código estão ali para se saber quantas vezes um dado número $n$ pode ser dividido por um número primo $p$, seja ele o 2 ou qualquer ímpar primo.

Este cálculo pode ser feito em um subprograma que recebe os números de entrada de alguma maneira e que devolva o valor correto também segundo uma convenção.

Esta convenção, em Pascal, é dada pelo protótipo de uma função que é constituído por três parâmetros: o nome da função, os números $n$ e $p$ envolvidos e, finalmente, o tipo do valor devolvido pelo subprograma, isto é, um tipo que contenha o número de vezes em que $n$ é dividido por $p$.

function num_vezes_quedivide(p, n : integer): integer;

Outro trecho de código repetido é a atualização do MDC para receber a menor potência do primo sendo calculado, na primeira vez é o 2, nas outras vezes é um primo ímpar. Basta criarmos um segundo protótipo de função que calcula a potência do primo elevado ao valor do menor contador. Este pode ser o seguinte:

function potencia(n, i : integer): integer;

Estas interfaces nos permitem modificar o programa do cálculo do MDC pela definição conforme mostra a figura 8.12.
program mdcpeladefinicao; (* pela definição de mdc *)
var
  a, b, primo, mdc, cont_a, cont_b, menor_cont : longint;
function num_vezes_que_divide(p: longint; var n: longint): longint;
  (* código da função num_vezes_que_divide *)
function potencia(n,p : longint): longint;
  (* código da função potencia *)
begin (* programa principal *)
  read (a,b);
  mdc:= 1;
  cont_a:= num_vezes_que_divide(2,a);
  cont_b:= num_vezes_que_divide(2,b);
  if cont_a <= cont_b then
    menor_cont:= cont_a
  else
    menor_cont:= cont_b;
  mdc:= mdc * potencia(2,menor_cont);
  primo:= 3;
  while (a <> 1) and (b <> 1) do
  begin
    cont_a:= num_vezes_que_divide(primo,a);
    cont_b:= num_vezes_que_divide(primo,b);
    if cont_a <= cont_b then
      menor_cont:= cont_a
    else
      menor_cont:= cont_b;
    mdc:= mdc * potencia(primo,menor_cont);
    primo:= primo + 2;
  end;
  writeln (mdc);
end.

Figura 8.12: Calcula MDC entre $a$ e $b$ pela definição usando funções.
8.3. ALGUNS EXEMPLOS

Observamos que o uso de funções e procedimentos permite muita flexibilidade ao programador, pois ele pode alterar o código das funções, se quiser, tornando-as mais eficientes (caso não fossem) sem que haja efeito colateral no programa principal. As figuras 8.13 e 8.14 mostram sugestões de código para as funções.

```
function num_vozes_que_divide(p: longint; var n: longint): longint;
var cont: longint;
begin
    (* descobre quantas vezes p divide n *)
    cont:= 0;
    while n mod p = 0 do
        begin
            cont:= cont + 1;
            n:= n div p;
        end;
    num_vozes_que_divide:= cont;
end;
```

Figura 8.13: Calcula quantas vezes um número divide outro.

```
function potencia(n,p : longint): longint;
var pot , i : longint ;
begin
    pot:= 1;
    i:= 1;
    while i <= p do
        begin
            pot:= pot * n;
            i:= i + 1;
        end;
    potencia := pot;
end;
```

Figura 8.14: Calcula a potência de um número elevado a outro.

Este código não é muito apropriado, pois exige um comportamento não muito elegante na função `num_vozes_que_divide`, ela tem o efeito colateral de alterar o valor da variável `n`, o que não é natural dado o nome da função e por isso `n` é passado por referência. Deveria apenas contar o número de vezes que divide e não fazer mais nada. O problema não pode ser resolvido com este mesmo algoritmo sem este efeito colateral. Em todo caso, sabemos que o algoritmo de Euclides é mais eficiente, mais simples de implementar e sobretudo mais elegante!

Terminamos aqui a primeira parte do curso, no qual as noções fundamentais sobre algoritmos estão estabelecidas. Nos próximos capítulos estudaremos as principais estruturas de dados básicas para um curso introdutório de algoritmos.
8.4 Exercícios

1. Faça uma função em *Pascal* que receba como parâmetro dois números inteiros não nulos e retorne TRUE se um for o contrário do outro e FALSE em caso contrário. Isto é, se os parâmetros forem 123 (cento e vinte e três) e 321 (trezentos e vinte e um), deve-se retornar TRUE. Usar apenas operações sobre inteiros.

2. Faça uma função em *Pascal* que receba como parâmetro um número inteiro representando um número binário e retorne seu valor equivalente em decimal. Por exemplo, se a entrada for 10001, a saída deve ser 17.

3. Faça uma função em *Pascal* que receba como parâmetro um número inteiro e retorne TRUE se ele for primo e FALSE em caso contrário. Use esta função para imprimir todos os números primos entre 0 e 1000.

4. Faça duas funções em *Pascal* que implementem os cálculos do seno e cosseno conforme definidos em capítulos anteriores e use-as em uma terceira função que calcule a tangente. O programa principal deve imprimir os valores de \( \tan(x) \) para um certo valor fornecido pelo usuário.

5. Faça uma função em *Pascal* que some dois números representando horas. A entrada deve ser feita da seguinte maneira:

\[
\begin{align*}
12 & \quad 52 \\
7 & \quad 13
\end{align*}
\]

A saída deve ser assim:

\[
12:52 + 7:13 = 20:05
\]

6. Faça uma função que receba como parâmetros seis variáveis DIA1, MES1 e ANO1, DIA2, MES2 e ANO2, todas do tipo integer. Considerando que cada trinca de dia, mês e ano representa uma data, a função deve retornar true se a primeira data for anterior à segunda e false caso contrário.
Capítulo 9

Estruturas de dados

Até aqui apresentamos as técnicas básicas para construção de algoritmos, incluindo as noções de funções e procedimentos. Podemos dizer que é possível, com este conteúdo, programar uma vasta coleção de algoritmos, inclusive alguns com alta complexidade.

Contudo, o estudo geral da disciplina de “Algoritmos e Estruturas de Dados” envolve algoritmos que trabalham dados organizados em memória de maneira mais sofisticada do que as simples variáveis básicas que foram estudadas até o momento. É algo mais ou menos parecido como manter um guarda-roupas organizado no lugar de um monte de coisas atiradas no meio do quarto de qualquer jeito.

A organização de dados em memória permite a construção de algoritmos sofisticados e eficientes. Neste texto estudaremos três estruturas de dados elementares. São elas:

- vetores (ou array unidimensional);
- matrizes (ou array multidimensional);
- registros.

Nas seções seguintes explicaremos cada uma delas, sempre motivados por problemas que exigem seu uso ou que facilitam a implementação.

9.1 Vetores

Para motivar, vamos considerar o problema seguinte: ler uma certa quantidade de valores inteiros e os imprimir na ordem inversa da leitura. Isto é, se os dados de entrada forem: 2, 5, 3, 4, 9, queremos imprimir na saída: 9, 4, 3, 5, 2.

Este tipo de problema é impossível de ser resolvido com o uso de uma única variável pois, quando se lê o segundo número, já se perdeu o primeiro da memória. Exigiria o uso de tantas variáveis quantos fossem os dados de entrada, mas notem que isto deve ser conhecido em tempo de compilação! Isso faz com que simplesmente não seja possível resolver este problema para uma quantidade indefinida de valores.

De fato, quando se alocata, por exemplo, um número inteiro em uma variável de nome $a$, o que ocorre é que o computador reserva uma posição de memória em algum
endereço da RAM (conforme sabemos pelo modelo Von Neumann). Um inteiro exige (dependendo da implementação) 2 bytes.

Mas, digamos que é preciso alocar espaço para 100 números inteiros. Sabendo que cada um deles ocupa 2 bytes, precisaríamos encontrar uma maneira de reservar \(100 \times 2 = 200\) bytes e fazer com que este espaço de memória pudesse ser acessado também por um único endereço, ou em outras palavras, por uma única variável.

Os vetores são estruturas de dados que permitem o acesso a uma grande quantidade de dados em memória usando-somente um nome de variável. Esta variável especial é declarada de tal maneira que o programador passa a ter acesso à muitas posições de memória, de maneira controlada.

Como funciona isto em memória?

A seguinte declaração em Pascal aloca 200 espaços em memória para números inteiros:

```pascal
var v: array[1..200] of integer;
```

Em Pascal isto resulta na alocação de 200 vezes 2 bytes. Pela variável \(V\) temos o controle deste espaço.

O problema é como se faz para se escrever um valor qualquer neste espaço. Outro problema é onde se escreve, já que temos 200 possibilidades de escolha. O simples uso da variável, como estávamos acostumados, não serve. É preciso uma outra informação adicional para se dizer em qual das 200 posições se quer escrever.

Na verdade, a variável \(V\) aponta para o início do segmento reservado, da mesma maneira que se fazia para variáveis básicas já estudadas. Para se escrever em algum lugar deste segmento, é preciso informar, além do nome da variável, uma segunda informação: a posição (ou o deslocamento) dentro do espaço reservado.

Ora, sabemos que foram reservadas 200 posições, cada uma delas com espaço para conter um número inteiro. Se quisermos escrever na quinta posição, basta informar ao computador que o início do segmento é dado pela variável \(V\) e que, antes de se escrever, é preciso realizar um deslocamento de 5 posições, cada uma delas para um inteiro. Isto dá um deslocamento de 10 bytes. Após esta informação, o valor pode ser escrito. Se o desejo é escrever na décima quarta posição, o deslocamento deve ser de \(14 \times 2\) bytes, isto é, 28 bytes.

Para se recuperar a informação, por exemplo para se imprimir, ou para se fazer algum cálculo com estes dados, basta usar o mesmo processo: os dados são acessados pelo nome da variável e pelo deslocamento.

Este processo foi apresentado em muito baixo nível. Como de costume, precisamos de uma outra forma de representação de mais alto nível. Isto é, cada linguagem de programação que implementa a noção de vetores tem que encontrar uma maneira para se mascarar para o programador este processo que é baseado em deslocamentos (ou somas de endereços).

Na próxima seção veremos como a linguagem Pascal lida com isto.
Vetores em *Pascal*

Como vimos, para se declarar um vetor de 200 posições inteiras, a linguagem *Pascal* usa a seguinte sintaxe (lembre-se que em outras linguagens a sintaxe pode ser diferente):

```pascal
var v: array [1..200] of integer;
```

A construção “1..200” é uma enumeração em *Pascal*, significa que os valores são 1, 2, 3, ..., 199, 200. Assim a construção `array[1..200]` indica que existem 200 posições controladas pela variável `v`. O “of integer” indica que cada posição é para se guardar um número inteiro, isto é, 2 bytes (dependendo da implementação).

Em *Pascal* o correto não é se referir à uma posição, mas a um rótulo. No exemplo acima, o rótulo da primeira posição é 1 e o rótulo da última posição é 200.

A rigor, a linguagem *Pascal* permite que se reserve 200 rótulos de várias maneiras. Basta que o intervalo “a..b” contenha 200 rótulos. Apresentamos 6 variantes dentre todas as possíveis:

```pascal
var v: array [0..199] of integer;
var v: array [201..400] of integer;
var v: array [−199..0] of integer;
var v: array [−300..−99] of integer;
var v: array [−99..100] of integer;
const min=11, max=210;
var v: array [min..max] of integer;
```

Para reforçar, se usarmos o intervalo 0..199, o rótulo da primeira posição agora é zero e o da última é 199.

Em todas estas variantes, o intervalo define 200 rótulos. Em termos gerais, existe uma restrição forte. O intervalo deve ser definido em termos de números de algum tipo *ordinal* (em *Pascal*), isto é, *integer, longint*, ..., até mesmo *char*. Também em *Pascal*, o limite inferior deve ser menor ou igual ao limite superior. Na sequência desta seção, vamos considerar a versão que usa o intervalo de 1 a 200.

Agora, para guardar um valor qualquer, digamos 12345, na posição 98 do vetor `v`, em *Pascal*, se usa um dos dois comandos seguintes:

```pascal
v[98]:= 12345;
read(v[98]);  (* e se digita 12345 seguido de ENTER no teclado *)
```
Em termos gerais, vejamos os seguintes exemplos, para fixar o conceito:

```pascal
read (v[1]); (* le do teclado e armazena na primeira posição de v *)
i := 10;
v[i+3] := i * i; (* armazena o quadrado de i (100) na posição 13 de v *)
write (i, v[i]); (* imprime o par (10, 100) na tela *)
write (v[v[13]]); (* imprime o valor de v[100] na tela *)
v[201] := 5; (* gera erro, pois a posição 201 não existe em v *)
v[47] := sqrt (4); (* gera erro, pois sqrt retorna um real, mas v eh de inteiros *)
var x: real;
v[x] := 10; (* gera erro, pois x eh do tipo real, deveria ser ordinal *)
```

Note que a construção ($v[v[13]]$) só é possível pois o vetor $v$ é do tipo `integer`. Se fosse um vetor de reais isto não seria possível (em `Pascal`), pois os rótulos devem ser de tipo `ordinal`, conforme já mencionado.

### 9.1.1 Primeiros problemas com vetores

Para iniciarmos nossa saga pelos algoritmos sofisticados que usam vetores, vamos apresentar uma série de problemas já conhecidos para vermos como eles podem ser resolvidos usando vetores. Aproveitaremos para fixar conceitos já ensinados sobre procedimentos e funções. Desta forma o estudante poderá, resolvendo exercícios simples, se concentrar na novidade, isto é, no uso de vetores.

**Lendo vetores**

Para resolvermos o problema apontado acima, isto é, um programa que leia 10 números reais e os imprima na ordem inversa da leitura, precisamos inicialmente ler os elementos do vetor. O código da figura [9.1] ilustra uma solução possível. Quando executado, considerando que no teclado foi digitada a sequência 15, 12, 27, 23, 7, 2, 0, 18, 19 e 21, teremos em memória algo como ilustrado na figura seguinte:

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>…</th>
<th>197</th>
<th>198</th>
<th>199</th>
<th>200</th>
</tr>
</thead>
</table>

É importante, neste momento, atentar para alguns fatos:

1. este vetor tem 200 elementos, pois seu tamanho foi definido em tempo de compileção. Como só foram lidos os 10 primeiros elementos, o restante do vetor contém valores indefinidos que já estavam em memória quando programa começou a executar. Isto é o que chamamos de *lixo* de memória e está representado com interrogações na figura acima;
program lendo_vetores;
var v: array [1..200] of real; (* define um vetor de reais *)
   i: integer;
begin
   i:= 1;
   while i <= 10 do
     begin
       read (v[i]);
       i:= i + 1;
     end;
end.

Figura 9.1: Lendo elementos e colocando no vetor.

2. o enunciado não especifica onde se armazenar os valores, como o vetor \( v \) tem 200 posições, poderíamos ter usado qualquer intervalo, mas normalmente se usa um vetor a partir da posição 1, e os algoritmos não podem deixar “buracos” no vetor;

3. o uso da variável auxiliar \( i \) no programa facilita muito a manipulação de vetores. Senão teríamos que usar comandos do tipo: \( \texttt{read}(v[1]), \texttt{read}(v[2]), \texttt{read}(v[3]) \), \ldots, recaíndo nos problemas do início do curso;

4. a títule de exemplo, mostramos a versão deste programa usando os comando \texttt{repeat e for}. Os trechos de código duas figuras 9.2 e 9.3 abaixo, ilustram estas duas variantes.

begin
  for i:= 1 to 10 do
     read (v[i]);
end.

Figura 9.2: Lendo elementos e colocando no vetor, usando \texttt{for}.

begin
  i:= 1;
  repeat
     read (v[i]);
     i:= i + 1;
  until i > 10;
end.

Figura 9.3: Lendo elementos e colocando no vetor, usando \texttt{repeat}.

\footnote{Ver o guia de referência da linguagem para entender estes dois novos comandos de repetição que são permitidos em \textit{Pascal}.}
Uma vez que se leu o vetor, pode-se agora manipular os dados da maneira necessária para se resolver o problema desejado. Nas seções seguintes vamos exemplificar usando diversos algoritmos já conhecidos.

**Imprimindo vetores**

O programa ilustrado na figura 9.4 mostra como ler 10 números reais do teclado e imprimi-los na tela. Usaremos o comando *for* nos exemplos seguintes pois ele facilita muito a redação dos programas que manipulam vetores. Os códigos ficam mais compactos e legíveis.

```pascal
program lendo_e_imprimindo_vetores;
var v: array [1..200] of real;
i: integer;
begin
  for i:= 1 to 10 do
  begin
    read (v[i]);
    writeln (v[i]);
  end;
end.
```

Figura 9.4: Lendo e imprimindo usando vetores.

É importante observar que este programa poderia ter sido resolvido sem o uso de vetores, como mostrado na figura 9.5

```pascal
program lendo_e_imprimindo;
var x: real; i: integer;
begin
  for i:= 1 to 10 do
  begin
    read (x);
    writeln (x);
  end;
end.
```

Figura 9.5: Lendo e imprimindo sem usar vetores.

Mostramos esta versão para o leitor poder comparar os dois códigos e perceber que a principal diferença entre as duas soluções é que, na versão com vetores, todos os números lidos do teclado continuam em memória após a leitura do último número digitado, enquanto que na versão sem vetores, a variável *x* teria armazenado apenas
9.1. VETORES

e tão somente o último número lido.

Uma outra maneira de escrever código para resolver o mesmo problema é separar o programa em duas partes: uma para a leitura e a outra para a impressão. O resultado é o mesmo, mas a maneira de fazer difere. A figura 9.6 ilustra esta solução.

```pascal
program lendo_e_imprimindo_vetores;
var v: array [1..200] of real;
i: integer;

begin
  (* este pedaco de codigo trata apenas da leitura dos dados *)
  for i:= 1 to 10 do
    read (v[i]);

  (* este outro pedaco de codigo trata apenas da impressao *)
  for i:= 1 to 10 do
    writeln (v[i]);
end.
```

Figura 9.6: Lendo e imprimindo: outra versão.

O código apresenta uma certa modularidade, pois pode ser facilmente visualizado como contendo uma parte que se ocupa da leitura dos dados separada da outra parte que se ocupa da impressão dos dados.

Apesar do fato deste programa funcionar, insistimos que ele merece ser escrito seguindo as boas técnicas de programação. Neste sentido o uso de funções e procedimentos pode ser explorado para que os dois módulos do programa (leitura e impressão) possam ficar claramente destacados de maneira independente um do outro.

O resultado final é o mesmo em termos de execução do programa e de seu resultado final na tela, exibido para quem executa o programa. Por outro lado, para o programador, o código é mais elegante. Por isto, vamos reescrever mais uma vez o programa, desta vez usando procedimentos.

Antes disto, porém, é importante destacar uma limitação da linguagem Pascal: infelizmente, o compilador não aceita um parâmetro do tipo array. Assim, a construção seguinte gera um erro de compilação:

```pascal
procedure ler (var v: array [1..200] of real);
```

Para contornar este problema, a linguagem Pascal permite a definição de novos tipos de dados baseados em outros tipos preexistentes. Isto se consegue com o uso da declaração type.

```pascal
type vetor= array [1..200] of real;
```
O que ocorre com o uso da declaração `type` é que o nome do tipo `vetor` passa a ser conhecido pelo compilador, que por `default` só conhece os tipos pré-definidos da linguagem. O compilador `Pascal` foi um dos primeiros a permitir que o programador pudesse definir seus próprios tipos.

Assim, para reescrevermos o programa da figura 9.1 usando todo o nosso arsenal de conhecimentos adquiridos sobre procedimentos, funções, uso de constantes no código, comentários no código, ... , faríamos como apresentado na figura 9.7.

```pascal
program ler_e_imprimir_com_procedures;
const min=1; max=200;
type vetor=array [min..max] of real;
var v: vetor;

procedure ler_vetor (var v: vetor);
var i: integer;
begin
  for i:= 1 to 10 do
    read (v[i]);
end;

procedure imprimir_vetor (var v: vetor); (* impressao dos dados *)
var i: integer;
begin
  for i:= 1 to 10 do
    write (v[i]);
end;

begin (* programa principal *)
  ler_vetor (v);
  imprimir_vetor (v);
end.
```

Figura 9.7: Lendo e imprimindo, agora com procedimentos.

Agora estamos prontos para resolver o problema proposto no início deste capítulo, aquele de ler uma sequência de números e imprimir-los ao contrário. Uma vez que os dados estão carregados em memória, após a execução do procedimento `ler(v)`, podemos manipular os dados da maneira que for necessário. No nosso caso, para imprimir ao contrário, basta modificar o procedimento de impressão para percorrer o vetor do final ao início. A figura 9.8 contém esta modificação. Basta agora modificar o programa principal trocando a chamada `imprimir(v)` por `imprimir_ao_contrario(v)`.

Algumas observações importantes:

1. A leitura é feita obrigatoriamente usando-se passagem de parâmetros por referência. A impressão pode usar passagem por valor. Contudo, conhecendo o
9.1. VETORES

Figura 9.8: Procedimento que imprime os elementos do vetor ao contrário.

fato de que existe uma cópia de elementos que é feita na pilha de memória, isto evidentemente provocará uma computação extra que pode custar caro, especialmente no caso em que os vetores são grandes. Imagine copiar, a toa, um milhão de elementos. Assim, em geral, vetores são passados sempre por referência.

2. O código seria generalizado facilmente se tivéssemos passado como parâmetro o tamanho usado (ou útil) do vetor, e não o número fixo 10, além do endereço dele. Neste caso, o código da figura 9.9 seria a solução mais elegante para o problema. Observar que o tamanho do vetor é lido dentro do procedimento, o que exige um parâmetro por referência.

Figura 9.9: Lendo e imprimindo ao contrário, versão final.

Neste ponto esgotamos o assunto de ler e imprimir vetores e estamos prontos para
novos problemas cujas soluções requerem o uso de vetores, ou tornam o código mais elegante.

Nas seções que seguem, vamos considerar dois vetores de tamanhos diferentes, um de inteiros o outro de reais. Nas apresentações dos algoritmos vamos omitir sempre que possível a redação dos cabeçalhos dos programas e nos concentrar na solução dos novos problemas, sempre usando funções e procedimentos. As seguintes definições serão usadas até o final deste capítulo:

```plaintext
const min_r=0; max_r=50; min_i=1; max_i=10;

type vetor_r= array [min_r..max_r] of real;
vetor_i= array [min_i..max_i] of integer;
```

Uma última observação, antes de continuarmos. Quando usamos vetores, estamos limitados ao tamanho dele, que deve ser conhecido em tempo de compilação! Isto pode causar dificuldades na resolução de problemas que envolvem um número desconhecido de valores de entrada. Mas não tem outro jeito a não ser, em tempo de compilação, se estimar um valor máximo para o número de elementos no vetor e, durante a execução, testar se este valor nunca foi ultrapassado. Se o número for maior, então deve-se modificar o tamanho do vetor e recompilar.

A questão de qual o tamanho ideal para ser o escolhido na hora de compilar é questão de bom-senso e envolve saber de qual aplicação se está falando. Por exemplo, se for um vetor para armazenar apostas da megassena, então o número 15 é suficiente. Se for para guardar saldos de clientes do banco, melhor saber quantos clientes existem hoje e estimar uma margem de erro que depende também do crescimento médio do número de clientes nos últimos anos. Um grande banco tem milhões de clientes.

**Imprimindo os que são pares**

Vamos retornar ao velho e conhecido problema de se ler uma massa de dados de quantidade indefinida e imprimir aqueles que são pares, ignorando os ímpares.

O programa da figura 9.10 ilustra uma procedure com uma possível solução. A leitura dos dados é muito similar ao que já foi mostrado no exemplo anterior, basta adaptar o tipo e dados vetor de reais para vetor de inteiros e por isto apresentamos apenas o que é diferente. Observemos a similaridade deste programa com relação ao código apresentado na figura 8.3.

Aqui se faz uso da função booleana “eh_par”, que foi estudada na seção 8.2.5. Com isto concluímos que os problemas são os mesmos, mas o uso deles é ligeiramente diferente por dois motivos: usa-se funções ou procedimentos, e também se resolve usando-se vetores. O resto não muda.

Um problema que pode parecer o mesmo, mas não é, seria imprimir os elementos dos rótulos pares do vetor, e não mais os elementos cujos conteúdos são pares. Perceber
procedure imprimir_pares (var v: vetor_i; tam: integer);
var i: integer;
begin
  for i:= 1 to tam do
    if eh_par (v[i]) then
      write (v[i], ', ');
  writeln;
end;

Figura 9.10: Imprimindo os elementos do vetor que são pares.

Esta diferença é fundamental no estudo de vetores. Consideremos o seguinte vetor $v$ com tamanho 10 como exemplo:

\[
\begin{array}{cccccccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & \ldots & 197 & 198 & 199 & 200 \\
\end{array}
\]

O código da figura [9.10] produziria como saída o seguinte: 12, 2, 0 e 18, que estão respectivamente nos rótulos 2, 6, 7 e 8 do vetor. Se, na linha 5 do programa, nós testarmos se o rótulo é par (e não o conteúdo):

\[
\text{if } \text{eh}_\text{par} (i) \text{ then } // \text{no lugar de: } \text{if } \text{eh}_\text{par} (v[i]) \text{ then}
\]

então a saída seria: 12, 23, 2, 18 e 21, respectivamente para os elementos das posições 2, 4, 6, 8 e 10 do vetor. Observe com atenção esta diferença, é importante.

Como antes, a função “eh_par” pode ser substituída por qualquer outra, como por exemplo, ser primo, se divisível por $n$, pertencer à sequência de Fibonacci, ou qualquer outra propriedade mais complexa que se queira.

**Encontrando o menor de uma sequência de números**

Vamos considerar o problema de se ler uma sequência de $N > 1$ números do teclado e imprimir o menor deles. Já sabemos resolvê-lo sem o uso de vetores, conforme ilustrado na figura [9.11].

Vamos reimplementá-lo em termos de uma função que considera que os elementos digitados no teclado já foram lidos em um vetor. A figura [9.12] permite percebermos que a técnica usada foi rigorosamente a mesma: o primeiro número lido é considerado o menor de todos. Na leitura dos dados subsequentes, quando um número ainda menor é encontrado, atualiza-se a informação de quem é o menor de todos.

**9.1.2 Soma e produto escalar de vetores**

Nesta seção vamos explorar algoritmos ligeiramente mais sofisticados, envolvendo noções novas sobre vetores.\(^2\)

\(^2\)Não tão novas, já que são conceitos estudados no ensino médio.
program menor_dos_lidos;
var N, i: integer; x, menor: real;
begin
  read (N);
  read (x);
  menor:= x;
  for i:= 2 to N do
    begin
      read (x);
      if x < menor then
        menor:= x;
    end;
  writeln (menor);
end.

Figura 9.11: Encontrando o menor de $N$ números lidos, sem vetor.

function menor_dos_lidos (var v: vetor_r; N: integer): real;
var i: integer; menor: real;
begin
  menor:= v[1];
  for i:= 2 to N do
    begin
    if v[i] < menor then
      menor:= v[i];
    menor_dos_lidos:= menor;
    end:
end;

Figura 9.12: Encontrando o menor de $N$ números lidos, com vetor.

Somando vetores

Nesta seção vamos implementar o algoritmo que soma dois vetores. Para isto precisamos antes entender como funciona o processo.

Sejam $v$ e $w$ dois vetores. Para somá-los, é preciso que eles tenham o mesmo tamanho. Isto posto, o algoritmo cria um novo vetor $v + w$ no qual cada elemento $(v + w)[i]$ do novo vetor é a soma dos respectivos elementos $v[i]$ e $w[i]$. O esquema é conforme a figura seguinte, que apresenta dois vetores de 6 elementos cada:

$$
\begin{array}{cccccc}
  v: & 1 & 2 & 3 & 4 & 5 & 6 \\
  w: & 1 & 3 & 2 & 4 & 3 & 5 \\
  v+w: & 3 & 9 & 3 & 9 & 6 & 8 \\
\end{array}
$$

Assim, o algoritmo que soma os dois vetores deverá, para cada $i$ fixo, somar os respectivos elementos em $v$ e $w$ e guardar em $v + w$. Variar $i$ de 1 até o tamanho do vetor resolve o problema. A função da figura 9.13 implementa esta ideia.
9.1. VETORES

**Figura 9.13:** Somando dois vetores.

Importante notar que \( i \) é variável local, isto é, serve para controlar o comando `for` interno do procedimento. Também digno de nota é a passagem de parâmetros: no caso de \( v \) e \( w \), poderíamos perfeitamente ter passado por valor, pois não são alterados no procedimento. Isto vale também para o tamanho do vetor. Mas \( soma_{v,w} \) deve ser obrigatoriamente passado por referência.

Se, por acaso, os dois vetores tivessem eventualmente tamanhos diferentes o protótipo mudaria um pouco e poderia ser assim (desde que na implementação se teste se \( tam_v = tam_w \)):

```pascal
procedure somar_vetores (var v, w, soma_v_w: vetor_i; tam: integer);
(* este procedimento considera que os dois vetores tem o mesmo tamanho *)
var i: integer;
begin
  for i:= 1 to tam do
    soma_v_w[i]:= v[i] + w[i];
end;
```

### Produto escalar

Nesta seção vamos implementar o algoritmo que calcula o produto escalar de dois vetores. Para isto precisamos antes entender como funciona o processo.

Sejam \( v \) e \( w \) dois vetores. Para se obter o produto escalar é preciso que eles tenham o mesmo tamanho. Isto posto, o algoritmo gera um número real (ou inteiro, depende do tipo de dados do vetor) obtido pela soma das multiplicações de cada elemento \( i \) dos vetores dados, respectivamente. O esquema é conforme a figura seguinte, que apresenta dois vetores de 6 elementos cada:

\[
\begin{array}{ccccccc}
  & 1 & 2 & 3 & 4 & 5 & 6 \\
  v: & 2 & 6 & 1 & 0 & 3 & 3 \\
  & \times & \times & \times & \times & \times \\
  w: & 1 & 0 & 2 & 4 & 3 & 5 \\
  & = & = & = & = & = \\
  & 2 & 0 & 2 & 0 & 9 & 15
\end{array}
\]

Os números obtidos a partir das multiplicações para todos os \( i \) fixos devem ser somados: \( 2 + 0 + 2 + 0 + 9 + 15 = 28 \). Logo, 28 é o produto escalar de \( v \) e \( w \).
Assim, o algoritmo que calcula o produto escalar de dois vetores deverá, para cada $i$ fixo, multiplicar os respectivos elementos em $v$ e $w$ e usar a técnica do acumulador para armazenar as somas parciais. Variar $i$ de 1 até o tamanho do vetor resolve o problema. A função que implementa esta ideia é apresentado na figura 9.14.

```pseudocode
function prod_escalar (var v, w: vetor_r; tam: integer): real;
var i : integer;
  soma: real;
begin
  soma:= 0;
  for i:= 1 to tam do
    soma:= soma + v[i] * w[i];
  prod_escalar:= soma;
end;
```

Figura 9.14: Produto escalar de dois vetores.

Como procuramos mostrar, programar usando vetores, funções e procedimentos não é muito diferente de se programar os algoritmos elementares tais como os que foram estudados até então. Pelo menos até agora. A próxima seção vai apresentar novas técnicas usando-se estes novos conceitos.

### 9.1.3 Busca em vetores

Nesta seção vamos estudar alguns algoritmos para o importante problema de busca em vetores. Em outras palavras, estamos interessados em saber se um dado elemento $x$ é um dos elementos de um vetor $v$ e, caso seja, também podemos querer saber o rótulo onde este elemento se encontra.

Este problema é extremamente importante em computação e não é difícil imaginar aplicações no nosso dia a dia. Por exemplo, localizar um livro na biblioteca, localizar um cliente de um banco, saber se um dado CPF está cadastrado em alguma lista negra, e por aí vai.

O tema de busca em vetores é tão importante que é estudado de diversas maneiras ao longo de um curso de ciência da computação. Em um curso introdutório iremos estudar os mais elementares para o caso de vetores de reais ou inteiros.

Ao longo desta seção, vamos considerar sempre que um dado vetor $v$ já foi lido em memória de alguma maneira. Assim, vamos elaborar algoritmos na forma de funções ou procedimentos. A ideia é, como tem sido até aqui, iniciar por algo trivial e evoluir a qualidade da solução.

O algoritmo mais ingênuo possível é mostrado na figura 9.15.

Este algoritmo sempre acha o elemento $x$ em $v$ se ele estiver presente, pois ele varre todo o vetor comparando cada elemento com $x$. Caso $x$ esteja presente duas vezes ou mais, ele retorna a posição da última ocorrência. O algoritmo sempre faz $n$
function busca_simples (x: real; var v: vetor; n: integer): integer;
var i: integer;
begin
  busca_simples:= 0;
  for i:= 1 to n do
    if v[i] = x then
      busca_simples:= i;
end;

Figura 9.15: Busca em vetores, primeira versão.

comparações, por causa do comando for. Por isto diz-se que o número de comparações que o algoritmo realiza é proporcional ao tamanho do vetor.

De fato, em qualquer programa sempre é interessante analisar o que custa mais caro no código. Entendemos por “custar caro” como sendo o comando que é executado o maior número de vezes durante uma rodada do programa.

Neste caso, temos um comando if, que faz um teste de igualdade (uma comparação), dentro do escopo de um comando for, o qual é controlado pela variável n. Logo, esta comparação será executada sempre n vezes.

Apesar do algoritmo funcionar, ele faz comparações demais: mesmo quando o elemento já foi encontrado o vetor é percorrido até o final. Obviamente ele poderia parar na primeira ocorrência, pois não foi exigido no enunciado que fossem localizadas todas as ocorrências (este é outro problema). Então podemos modificar o código para a versão apresentada na figura 9.16, simplesmente trocando-se o for por um while que termina a execução tão logo o elemento seja encontrado (se for encontrado).

function busca_simples_v2 (x: real; var v: vetor; n: integer): integer;
var i: integer;
  achou: boolean;
begin
  achou:= false;
  i:= 1;
  while (i <= n) and not achou do
    begin
      if v[i] = x then
        achou:= true;
      i:= i + 1;
    end;
  if achou then
    busca_simples_v2:= i - 1;
end;

Figura 9.16: Busca em vetores, segunda versão.

Este algoritmo é mais rápido, na média, do que o anterior, embora o número de comparações feitas possa ser a mesma no caso do elemento não estar presente no
vetor. Mas, se dermos sorte, o elemento pode estar no início do vetor e terminar bem rápido. Na média, espera-se que ele pare mais ou menos na metade do vetor, isto é, considerando-se uma distribuição uniforme dos elementos.

Mas como o laço faz dois testes, no caso do elemento não ser encontrado ele será um pouco mais lento. Notem que o duplo teste no laço é necessário pois deve parar ou porque achou o elemento ou porque o vetor terminou. Este segundo teste só vai dar true uma única vez (quando o elemento não está presente no vetor), o que é um desperdício.

Se pudesse haver garantia de que sempre o elemento procurado estivesse presente, então poderíamos construir um teste simples, o que pode nos economizar alguma computação. Esta garantia não pode existir, certo? Mais ou menos. Digamos que o programador deliberadamente coloque o elemento no vetor. Neste caso, há a garantia de que ele está presente. Mas alguém pode perguntar: assim não vale, se eu coloco não significa que ele já estava presente, eu vou sempre encontrar o elemento “falso”.

Novamente, depende de onde o programador coloque o elemento. Digamos que ele o coloque logo após a última posição. Então, das duas uma: ou o elemento não estava no vetor original, e neste caso a busca pode terminar pois o elemento será encontrado após a última posição; ou o elemento estava presente e será encontrado antes daquele que foi adicionado. Um teste final resolve a dúvida. Se for encontrado em posição válida, é porque estava presente, senão, não estava.

Este elemento adicionado logo após o final do vetor é denominado sentinela e seu uso é ilustrado na versão apresentada na figura 9.17.

| function busca_com_sentinela (x: real; var v: vetor_r; n: integer): integer; |
| var i: integer; |
| begin |
| v[n+1]:= x; (* sentinela: coloca x após a ultima posicao do vetor *) |
| i:= 1; |
| while v[i] <> x do |
| i:= i + 1; |
| if i <= n then (* achou nas posicoes validas *) |
| busca_com_sentinela:= i |
| else (* achou o sentinela, ou seja, x nao estava no vetor *) |
| busca_com_sentinela:= 0; |
| end; |

Figura 9.17: Busca em vetores com sentinela.

Apesar da melhoria, este algoritmo sempre faz um número de comparações que pode atingir n no pior caso, isto é, quando o elemento não está presente no vetor.

O caso ótimo e o caso médio não mudam, embora o algoritmo, conforme explicado, faça metade das comparações quando comparado com a versão anterior. Desta forma ele é ligeiramente melhor do que o anterior.

Ainda, o programador deve garantir que a posição usada pelo sentinela nunca seja usada como sendo um elemento válido, pois não é. O programador colocou o elemento ali de maneira controlada, mas não se trata de um elemento válido. Isto significa que
9.1. VETORES

o número de elementos úteis do vetor agora não é mais \( max_r \) (ou \( max_i \)), mas sempre um a menos. Normalmente se modifica a definição do tipo para prever este espaço adicional para o programador.

| const min_r=0; max_r=50; |
| min_i=1; max_i=10; |
| type vetor_r= array [min_r..max_r+1] of real; |
| vetor_i= array [min_i..max_i+1] of integer; |

É possível tornar o algoritmo mais eficiente?
A resposta é sim. Mas, para tornar a busca mais eficiente é preciso impor algumas restrições extra na forma como os dados são organizados em memória.

Esta maneira consiste em considerar que os dados estão de alguma forma respeitando uma ordem lexicográfica em memória. Por exemplo, se forem nomes, estão em ordem alfabética. Se forem números, estão em ordem crescente (ou decrescente). Porque isto é necessário? Pois pode-se explorar a informação de ordenação para tornar o método mais eficiente.

No caso, podemos modificar a solução anterior para que o algoritmo termine a busca sempre que encontrar um elemento que já é maior do que aquele que se está procurando, pois o vetor está ordenado. O programa da figura 9.18 foi implementado com esta ideia.

| function busca_vetor_ordenado (x: real; var v: vetor_r; n: integer): integer; |
| var i: integer; |
| begin |
| v[n+1]:= x; |
| i:= 1; |
| while v[i] < x do |
| i:= i + 1; |
| if (v[i] = x) and (i <= n) then |
| busca_vetor_ordenado:= i |
| else |
| busca_vetor_ordenado:= 0; |
| end; |

Figura 9.18: Busca em vetores ordenados.

Apesar de termos explorado uma propriedade adicional que faz com que no caso médio a busca seja mais eficiente, no pior caso, aquele em que o elemento procurado não pertence ao vetor, ainda temos um algoritmo que faz tantas comparações quanto o tamanho do vetor. É possível fazer melhor? A resposta novamente é sim.

Como exemplo, considere o problema de se encontrar um verbete no dicionário.

\(^3\)Em disciplinas avançadas de algoritmos se estudam métodos bastante eficientes para este problema. No nosso caso veremos somente um deles.
Sabemos que estes verbetes se encontram em ordem alfabética. Por este motivo, ninguém em sã consciência procura um nome em ordem sequencial desde o início, a menos que esteja procurando algo que comece com a letra “A”.

Suponha que o verbete inicia com a letra “J”. Normalmente se abre o dicionário mais ou menos no meio. Se o nome presente no início da página for “menor” lexicográficamente falando do que aquele que se busca, por exemplo, algo começando com a letra “D”, então, pode-se tranquilamente rasgar o dicionário, eliminando-se tudo o que está antes do “D” e continuar o processo com o que restou.

Na segunda tentativa abre-se novamente mais ou menos na metade do que sobrou. Suponha que caímos na letra “M”. Como estamos procurando o “J”, pode-se sem problemas rasgar novamente o dicionário eliminando-se tudo o que segue o “M”, até o fim. Resta procurar “somente” entre o “D” e o “M”.

Aplicando-se consistentemente este processo repetidas vezes, sempre teremos um dicionário dividido mais ou menos por 2. Se ele tinha 500 páginas no início, na segunda vez terá 250 e na terceira 125, na quarta algo próximo de 70 páginas. E assim por diante. Por isto que costumamos localizar rapidamente verbetes em um dicionário.

O algoritmo mostrado na figura 9.20, denominado de busca binária, implementa esta ideia. Procura-se o elemento no meio do vetor. Se encontrou, então pode parar. Se não encontrou, basta verificar se o valor encontrado é maior ou menor do que o procurado. Se for maior, joga-se a metade superior fora e trabalha-se com a metade inferior. Se for menor, joga-se fora a metade inferior e trabalha-se com a metade superior. Novamente procura-se na metade do que sobrou e assim por diante, até que se encontre o elemento ou até que se determine que não é mais possível encontrá-lo.

Vamos agora comparar estas últimas quatro versões para o algoritmo de busca. A tabela 9.19 resume o número de comparações feitas para diversos tamanhos de entrada, sempre analisando o pior caso. O caso médio exige um pouco mais de cuidado para se calcular e não será estudado aqui. O caso ótimo é sempre uma única comparação para os casos: o algoritmo acha o elemento na primeira tentativa.

<table>
<thead>
<tr>
<th>Versão</th>
<th>$n = 10$</th>
<th>$n = 10^2$</th>
<th>$n = 10^4$</th>
<th>$n = 10^8$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Busca simples (fig. 9.16)</td>
<td>20</td>
<td>200</td>
<td>20000</td>
<td>200000000</td>
</tr>
<tr>
<td>Busca com sentinela (fig. 9.17)</td>
<td>10</td>
<td>100</td>
<td>10000</td>
<td>100000000</td>
</tr>
<tr>
<td>Busca em vetor ordenado (fig. 9.18)</td>
<td>10</td>
<td>100</td>
<td>10000</td>
<td>100000000</td>
</tr>
<tr>
<td>Busca binária (fig. 9.20)</td>
<td>3</td>
<td>5</td>
<td>10</td>
<td>19</td>
</tr>
</tbody>
</table>

Figura 9.19: Tabela resumindo número de comparações para algoritmos de busca.

O importante do método da busca binária é que ela apresenta um caráter logarítmico para o pior caso com relação ao tamanho da entrada, o que é bastante significativo. Contudo, é absolutamente relevante destacar que este método só pode ser aplicado em vetores ordenados, senão não funciona. A questão é saber qual o custo de se ordenar, ou de se manter ordenado, um vetor. Isto será estudado à frente.
function busca_binaria \((x: \text{ real}; \ \text{var} \ v: \text{ vetor}_r; \ \text{n: integer})\): integer;

\textbf{var} \ \text{início, fim, meio: integer;}

\textbf{begin}
    \text{início}:=1; \ (* \ \text{aponta para o início do vetor} *)
    \text{fim}:= \text{n}; \ (* \ \text{aponta para o fim do vetor} *)
    \text{meio}:= (\text{início} + \text{fim}) \div 2; \ (* \ \text{aponta para o meio do vetor} *)
    \text{while} (v[\text{meio}] \neq x) \text{ and } (\text{fim} \geq \text{início}) \text{ do}
    \text{begin}
        \text{if} v[\text{meio}] > x \text{ then} \ (* \ \text{vai jogar uma das duas metades fora} *)
            \text{fim}:= \text{meio} - 1 \ (* \ \text{metade superior foi jogada fora} *)
        \text{else}
            \text{início}:= \text{meio} + 1; \ (* \ \text{metade inferior foi jogada fora} *)
            \text{meio}:= (\text{início} + \text{fim}) \div 2; \ (* \ \text{recalcula apontador para meio} *)
    \text{end;}
\ (* \ \text{o laço termina quando achou ou quando o fim ficou antes do início} *)
    \text{if} \ \text{início} \leq \text{fim} \text{ then}
        \text{busca\_binaria}:= \text{meio}
    \text{else}
        \text{busca\_binaria}:= 0;
\text{end;}

\textbf{Figura 9.20: Busca binária.}

\textbf{Manipulando vetores ordenados}

Quando operações de inserção e remoção são feitas em vetores ordenados é importante que estas alterações em seus elementos preservem a propriedade do vetor estar ordenado, por exemplo, para que se possa usar uma busca binária.

A seguir apresentamos soluções para se remover ou inserir um elemento de um vetor, iniciando pela remoção. O procedimento da figura 9.21 garante a remoção do elemento que está na posição \(pos\) do vetor \(v\) que tem tamanho \(n\).

\begin{verbatim}
procedure remove\_vetor\_ordenado \((\text{pos: integer}; \ \text{var} \ v: \text{ vetor}_r; \ \text{var n: integer})\);
\textbf{var} \ \text{i: integer;}
\textbf{begin}
    \text{for} \ i:= \text{pos} \ \text{to} \ n-1 \ \text{do}
        \text{v[i]}:= \text{v[i+1]};
    \text{n}:= n - 1;
\textbf{end:}
\end{verbatim}

\textbf{Figura 9.21: Removendo de um vetor ordenado.}

Remover elementos não é tão trivial quanto parece, pois não podemos deixar “buracos” no vetor. Este algoritmo então copia todos os elementos que estão à frente daquele que foi removido uma posição para trás. Tomemos como exemplo o seguinte vetor ordenado:
Para remover o elemento 12, que está na posição 4, todos os outros são copiados, um por um. Para não perdermos elementos, este processo tem que iniciar da posição onde houve a remoção até a penúltima posição. O vetor resultante é o seguinte:

O detalhe é que o elemento 27 que estava na posição 10 continua lá, mas, como o vetor agora tem tamanho 9 (pois um elemento foi removido) este último 27 agora é lixo de memória.

Com relação ao algoritmo de inserção de um elemento em um vetor ordenado, deve-se primeiro determinar a posição correta de inserção, logo, um processo de busca deve ser feito, iniciando do começo do vetor (posição 1) até que seja encontrado um elemento maior que aquele que está se inserindo. Para facilitar esta busca, usamos uma sentinela.

Tomemos novamente como exemplo o vetor abaixo e consideremos que vamos inserir o elemento 17.

A posição correta do 17 é logo após o elemento 15, isto é, na posição 6. Para abrirmos espaço no vetor e ao mesmo tempo não perdermos o 18 que lá está devemos copiar todos os elementos, um por um, uma posição para frente. Isto deve ser feito de trás para frente, isto é, copia-se o último uma posição adiante para abrir espaço para colocar o penúltimo, e assim por diante. O resultado seria um vetor assim:

Observe que agora temos duas vezes o 18 no vetor. O primeiro deles agora pode ser substituído pelo novo elemento, o 17, assim:

O procedimento da figura 9.22 insere um elemento x no vetor v que tem tamanho n de maneira que ele continue ordenado. Apenas uma passagem no vetor é feita: ao mesmo tempo em que ele está procurando a posição correta do elemento ele já vai abrindo espaço no vetor.
9.1. VETORES

procedure insere_vetor_ordenado (x: real; var v: vetor_r; var n: integer);
var i: integer;

begin
  v[0]:= x;
  i:= n;
  while x < v[i] do
  begin
    v[i+1]:= v[i];
    i:= i - 1;
  end;
  v[i+1]:= x;
  n:= n + 1;
end;


No caso de vetores não ordenados, os processos de inserção e remoção são mais simples. Pode-se sempre inserir um elemento novo no final do vetor e para remover, basta copiar o último sobre o que foi removido. Estes algoritmos de inserção e remoção em vetores não ordenados têm portanto um custo de uma única operação, enquanto que para vetores ordenados os custos são proporcionais ao tamanho do vetor (custo linear).

Há um último algoritmo interessante que trabalha com vetores ordenados: a ideia é fundir
dois vetores ordenados em um terceiro, obtendo-se um vetor ordenado como resultado contendo os mesmos elementos dos vetores anteriores.

O princípio deste algoritmo é: atribui-se a duas variáveis \( i \) e \( j \) o índice 1. Em seguida se compara os respectivos elementos das posições \( i \) em um vetor e \( j \) no outro. Se o elemento do primeiro vetor é menor do que o do segundo, ele é copiado no terceiro vetor e o apontador \( i \) é incrementado de 1, passando a observar o próximo elemento. Senão, o elemento do segundo vetor é menor e portanto este que é copiado para o terceiro vetor e o apontador \( j \) que é incrementado. Quando um dos dois vetores acabar, basta copiar o restante do outro no terceiro vetor. Um apontador \( k \) controla o terceiro vetor. Este é sempre incrementado a cada cópia.

Vejamos isto através de um exemplo. Consideremos os seguintes dois vetores ordenados e um terceiro que vai receber a fusão dos dois.

\[
\begin{array}{c|ccccc}
  i & 1 & 2 & 3 & 4 & 5 \\
  \hline
  v & 2 & 4 & 5 & 7 & 9 \\
  j & 1 & 2 & 3 \\
  w & 1 & 3 & 6 \\
  k & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
\end{array}
\]

comparam-se o elementos apontados por \( i \) e \( j \). Como \( 1 < 2 \), copia-se o 1 no vetor \( r \) e

\[\text{Em inglês, merge}\]
incrementa-se o apontador \( j \), assim:

\[
\begin{array}{cccccc}
\text{v:} & 2 & 4 & 5 & 7 & 9 \\
1 & j=2 & 3 \\
\end{array}
\]

\[
\begin{array}{cccccc}
\text{w:} & 1 & 3 & 6 \\
1 & k=2 & 3 & 4 & 5 & 6 & 7 & 8 \\
\end{array}
\]

\[
\begin{array}{cccccc}
\text{r:} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
\end{array}
\]

Agora comparam-se os elementos apontados por \( i \) e \( j \). Como 2 < 3, copia-se o 2 no vetor \( r \) e incrementa-se o apontador \( i \), assim:

\[
\begin{array}{cccccc}
\text{v:} & 2 & 4 & 5 & 7 & 9 \\
1 & j=2 & 3 \\
\end{array}
\]

\[
\begin{array}{cccccc}
\text{w:} & 1 & 3 & 6 \\
1 & k=3 & 4 & 5 & 6 & 7 & 8 \\
\end{array}
\]

\[
\begin{array}{cccccc}
\text{r:} & 1 & 2 & 3 & 4 & 5 \\
\end{array}
\]

Repetindo-se este processo mais algumas vezes chegaremos no seguinte estado:

\[
\begin{array}{cccccc}
\text{v:} & 2 & 4 & 5 & 7 & 9 \\
1 & j=3 \\
\end{array}
\]

\[
\begin{array}{cccccc}
\text{w:} & 1 & 3 & 6 \\
1 & 2 & 3 & 4 & k=5 & 6 & 7 & 8 \\
\end{array}
\]

\[
\begin{array}{cccccc}
\text{r:} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 9 \\
\end{array}
\]

Neste ponto o 6 será copiado em \( r \) e o vetor \( w \) termina. Basta copiar o que sobrou de \( v \) e terminar o processo, o que resultará no seguinte:

\[
\begin{array}{cccccc}
\text{v:} & 2 & 4 & 5 & 7 & 9 \\
1 & 2 & 3 & j=4 \\
\end{array}
\]

\[
\begin{array}{cccccc}
\text{w:} & 1 & 3 & 6 \\
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & k=9 \\
\end{array}
\]

\[
\begin{array}{cccccc}
\text{r:} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 9 \\
\end{array}
\]

O algoritmo da figura 9.23 implementa o que foi discutido, é eficiente e varre os dois vetores somente uma vez.
procedure merge_vetores (var v: vetor_r; n: integer; var w: vetor_r; m: integer; var r: vetor_r);
var i, j, k, l: integer;

begin
i:= 1; j:= 1; k:= 1;
(* i vai controlar os elementos de v *)
(* j vai controlar os elementos de w *)
(* k vai controlar os elementos do vetor resultante da fusao, r *)

while (i <= n) and (j <= m) do
begin
if v[i] <= w[j] then (* se o elemento de v eh menor que o de w *)
begin
r[k]:= v[i];
i:= i + 1;
end
else (* o elemento de w eh menor que o de v *)
begin
r[k]:= w[j];
j:= j + 1;
end;
k:= k + 1; (* k eh sempre incrementado *)
end;
(* quando sai do laco, um dos dois vetores acabou *)

if i <= n then (* w acabou, copiar o restante de v em r *)
for l:= i to n do
begin
r[k]:= v[l];
k:= k + 1;
end
else (* v acabou, copiar o restante de w em r *)
for l:= j to m do
begin
r[k]:= w[l];
k:= k + 1;
end;
end;

Figura 9.23: Fundindo dois vetores ordenados.
9.1.4 Ordenação em vetores

Ordenar vetores é extremamente importante em computação, pois é muito comum que uma saída de um programa seja dada com algum tipo de ordem sobre os dados. Nesta seção vamos apresentar dois algoritmos para este problema: os métodos da ordenação por seleção e por inserção.

Ordenação por seleção

A ordenação por seleção é um método bastante simples de se compreender e também de se implementar.

O princípio é selecionar os elementos corretos para cada posição do vetor, daí o nome do método. Para um vetor de \( N \) elementos, existem \( N - 1 \) etapas. Em cada etapa \( i \) o \( i \)-ésimo menor elemento é selecionado e colocado na posição \( i \).

Assim, na primeira etapa, o menor elemento de todos é localizado (selecionado) e colocado na primeira posição do vetor. Na segunda etapa localiza-se o segundo menor e coloca-se na segunda posição, e assim por diante, até que, por fim, o penúltimo menor elemento (isto é, o segundo maior) seja colocado na penúltima posição. Consequentemente, como já temos \( N - 1 \) elementos em seus devidos lugares, o último também está. Vejamos um exemplo de um vetor com 10 elementos.

Para localizarmos o menor elemento, que é o zero que está na posição 7 do vetor, só há uma maneira, que é percorrer todo o vetor e localizar o menor. Este deve ser colocado na primeira posição. Este primeiro (o 15), por sua vez, deve ser trocado de posição com o da posição 7. Por isto a busca pelo menor deve nos retornar o índice do menor elemento e não o elemento em si. O resultado da primeira etapa deste processo está mostrado na figura abaixo, com destaque para os elementos trocados.

Neste ponto precisamos do segundo menor. Por construção lógica do algoritmo, basta percorrer o vetor a partir da segunda posição, pois o primeiro já está em seu lugar. O menor de todos agora é o 2, que está na posição 6. Basta trocá-lo pelo segundo elemento, que é o 12. E o processo se repete:
9.1. VETORES

```pascal
procedure selecao (var v: vetor_r; n: integer);
var i, j, pos_menor: integer; aux: real;
begin
  for i:= 1 to n-1 do
    begin
      (* acha a posicao do menor a partir de i *)
      pos_menor:= i;
      for j:= i+1 to n do (* inicia a partir de i+1 *)
        if v[j] < v[pos_menor] then
          pos_menor:= j;
      aux:= v[pos_menor]; (* troca os elementos *)
      v[pos_menor]:= v[i];
      v[i]:= aux;
    end;
end;
```

Figura 9.24: Método de ordenação por seleção.


Este algoritmo tem algumas particularidades dignas de nota. A primeira é que, em cada etapa $i$, a ordenação dos primeiros $i - 1$ elementos é definitiva (dizemos ordenação total), isto é, constitui a ordenação final destes primeiros $i$ elementos do vetor.

A segunda é que a busca pelo menor elemento em cada etapa $i$ exige percorrer um vetor de $N - i$ elementos. Como isto é feito $N$ vezes, então o número de comparações feitas na parte mais interna do algoritmo é sempre $\frac{N(N-1)}{2}$, o que define um comportamento quadrático para as comparações.

A terceira é que trocas de posições no vetor ocorrem no laço mais externo, por isto o número total de trocas feitas pelo algoritmo é sempre $N - 1$.

Ordenação por inserção

A ordenação por inserção é na prática mais eficiente que a ordenação por seleção. Portém, embora o algoritmo em si seja simples, sua implementação é repleta de detalhes. Vamos inicialmente entender o processo.

O princípio do algoritmo é percorrer o vetor e a cada etapa $i$, o elemento da posição $i$ é inserido (daí o nome do método) na sua posição correta relativamente quando comparado aos primeiros $i - 1$ elementos. Muitas pessoas ordenam suas cartas em jogos de baralho usando este método.

Para melhor compreensão, faremos a apresentação de um exemplo sem mostrarmos o vetor, usaremos sequências de números. Consideremos que a entrada é a mesma do exemplo anterior, isto é:

```
15, 12, 27, 23, 7, 2, 0, 18, 19, 21.
```

Na primeira etapa o algoritmo considera que o primeiro elemento, o 15, está na
sua posição relativa correta, pois se considera apenas a primeira posição do vetor. Usaremos os negritos para mostrar quais as etapas já foram feitas pelo algoritmo.

Na segunda etapa deve-se inserir o segundo elemento em sua posição relativa correta considerando-se somente o vetor de tamanho 2. Como o 12 é menor que que o 15, deve-se trocar estes elementos de posição, nos resultando na sequência:

$$12, 15, 27, 23, 7, 2, 0, 18, 19, 21.$$ 

Neste ponto, os elementos 12 e 15 estão em suas posições relativas corretas considerando-se um vetor de 2 posições. Agora, deve-se colocar o 27 no vetor de 3 elementos. Mas o 27 já está em seu lugar relativo, então o algoritmo não faz nada:

$$12, 15, 27, 23, 7, 2, 0, 18, 19, 21.$$ 

Na quarta etapa deve-se inserir o 23 na sua posição relativa correta considerando-se um vetor de 4 elementos. O 23 tem que estar entre o 15 e o 27:

$$12, 15, 23, 27, 7, 2, 0, 18, 19, 21.$$ 

Na quinta etapa deve-se inserir o 7 na sua posição relativa a um vetor de 5 elementos. Ele deve ser inserido antes do 12, isto é, na primeira posição:

$$7, 12, 15, 23, 27, 2, 0, 18, 19, 21.$$ 

A situação para o 2 é similar, deve ser inserido antes do 7, isto é, no início:

$$2, 7, 12, 15, 23, 27, 0, 18, 19, 21.$$ 

Idem para o zero:

$$0, 2, 7, 12, 15, 23, 27, 18, 19, 21.$$ 

Agora é a vez de inserirmos o 18, entre o 15 e o 27:

$$0, 2, 7, 12, 15, 18, 23, 27, 19, 21.$$ 

Na penúltima etapa inserimos o 19 entre o 18 e o 23:

$$0, 2, 7, 12, 15, 18, 19, 23, 27, 21.$$ 

E por último o 21 entre o 19 e o 23:

$$0, 2, 7, 12, 15, 18, 19, 21, 23, 27.$$
Esta sequência de $N$ passos é de fácil compreensão. Se fôssemos executar com um conjunto de cartas na mão, por exemplo, com cartas de baralho, imaginando um maço de cartas virado na mesa, basta pegar as cartas uma a uma e encaixar no lugar certo. As cartas de baralho são facilmente manipuladas para permitir uma inserção em qualquer posição.

Infelizmente esta operação executada em um vetor não é tão simples. Vamos considerar como exemplo a etapa 8 acima, isto é, inserção do 18 no lugar certo. Retomemos este caso agora considerando um vetor para melhor ilustração, com destaque para o elemento 18 que deve nesta etapa ser inserido no lugar certo:

```
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 ... 197 198 199 200
```

A posição correta do 18, como vimos, é entre o 15 e o 23, isto é, na sexta posição do vetor. Significa que os elementos das posições 6 e 7 devem ser movidos um para frente para abrir espaço no vetor para inserção do 18. Os elementos das posições 9 em diante não vão mudar de lugar. Executando esta operação, e salvando o 18 em uma variável temporária, obteremos o seguinte vetor:

```
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 ... 197 198 199 200
```

Isto é, o 27 foi copiado da posição 7 para a posição 8 e o 23 foi copiado da posição 6 para a posição 7. Na figura acima destacamos os elementos que foram movidos de lugar. Observando que o 23 ficou repetido na posição 6, o que na prática resultou na posição 6 livre. Agora basta inserir o 18 aí:

```
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 ... 197 198 199 200
```

Esta etapa constitui o núcleo do algoritmo mostrado na figura 9.25. O laço externo garante que esta operação será executada para todos os elementos das posições de 1 até $N$.

O laço interno foi implementado de tal forma que, ao mesmo tempo em que se localiza o lugar certo do elemento da vez, já se abre espaço no vetor. O laço é controlado por dois testes, um deles para garantir que o algoritmo não extrapole o início do vetor, o outro que compara dois elementos e troca de posição sempre que for detectado que o elemento está na posição incorreta.

Analisar quantas comparações são feitas é bem mais complicado neste algoritmo, pois isto depende da configuração do vetor de entrada. Neste nosso exemplo, vimos que cada etapa teve um comportamento diferente das outras. Em uma vez o elemento já estava em seu lugar. Em duas outras vezes tivemos que percorrer todo o subvetor inicial, pois os elementos deveriam ser o primeiro, em cada etapa.

Aparentemente, no pior caso possível, que é quando o vetor está na ordem inversa da ordenação, haverá o maior número de comparações, que é quadrático. Mas, na prática, este algoritmo aparenta ser mais rápido que o método da seleção na maior parte dos casos, pois algumas vezes o elemento muda pouco de posição.

O site [http://cg.scs.carleton.ca/~morin/misc/sortalg](http://cg.scs.carleton.ca/~morin/misc/sortalg) permite visualizar o comportamento dos principais algoritmos de ordenação através de animações. Os dois algoritmos aqui explicados estão lá, entre outros.
procedure inserção (var v: vetor_r; n: integer);
var i, j: integer;
aux: real;
begin
  for i:= 1 to n do
    begin
      aux:= v[i];
      (* abre espaço no vetor enquanto localiza a posição certa *)
      j:= i - 1;
      while (j >= 1) and (v[j] > aux) do
        begin
          v[j+1]:= v[j];
          j:= j - 1;
        end;
      v[j+1]:= aux;
    end;
end;

Figura 9.25: Método de ordenação por inserção.

9.1.5 Outros algoritmos com vetores

Nesta seção vamos apresentar alguns problemas interessantes que podem ser resolvidos usando-se a estrutura de vetores.

Permutações

Vamos apresentar um problema matemático conhecido como permutação, propor uma representação computacional em termos de vetores, e, em seguida, estudar alguns problemas interessantes do ponto de vista de computação.\footnote{Esta seção foi inspirada em uma preparatória para a maratona de programação da ACM da Ural State University (Internal Contest October’2000 Junior Session) encontrada na seguinte URL: http://acm.timus.ru/problem.aspx?space=1&num=1024}

Os matemáticos definem uma permutação de algum conjunto como uma função bijetora de um conjunto nele mesmo. Em outras palavras, é uma maneira de reordenar os elementos do conjunto (que não têm elementos repetidos). Por exemplo, podemos definir uma permutação do conjunto \{1, 2, 3, 4, 5\} assim: \(P(1) = 4, P(2) = 1, P(3) = 5, P(4) = 2, P(5) = 3\). Esquematicamente temos:

\[
\begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 4 & 1 & 5 & 2 & 3 \end{pmatrix}
\]

Outra maneira seria: \(P(1) = 2, P(2) = 5, P(3) = 1, P(4) = 3, P(5) = 2\). Esquematicamente:
De fato, existem \( n! \) maneiras de se reordenar os elementos e obter uma permutação válida. Se \( n = 5 \) então existem 120 permutações.

**Modelando permutações**

O primeiro problema interessante do ponto de vista algorítmico é como representar uma permutação. Para isto pode-se usar um vetor de \( n \) posições inteiras, onde cada posição é um valor (sem repetição) entre 1 e \( n \).

Assim os dois vetores que representam as permutações acima são, respectivamente:

\[
\begin{pmatrix}
1 & 2 & 3 & 4 & 5 \\
4 & 1 & 5 & 2 & 3
\end{pmatrix}
\]

\[
\begin{pmatrix}
1 & 2 & 3 & 4 & 5 \\
2 & 5 & 1 & 3 & 2
\end{pmatrix}
\]

**Testando permutações válidas**

Uma vez resolvido o problema da representação, podemos estudar o próximo desafio, que é como testar se um dado vetor (lido do teclado) é uma permutação válida?

O algoritmo tem que testar se, para os índices de 1 a \( n \), seus elementos são constituídos por todos, e tão somente, os elementos entre 1 e \( n \), em qualquer ordem. A função da figura 9.26 apresenta uma possível solução.

```pseudocode
function testa_permutacao (var v: vetor_i; n: integer): boolean;
var i, j: integer;
eh_permutacao: boolean;
begin
  eh_permutacao:= true;
i:= 1;
  while eh_permutacao and (i <= n) do
    begin
      j:= 1; (* procura se i esta no vetor *)
      while (v[j] <> i) and (j <= n) do
        j:= j + 1;
      if v[j] <> i then (* se nao achou nao eh permutacao *)
        eh_permutacao:= false;
      i:= i +1;
    end;
  testa_permutacao:= eh_permutacao;
end; (* testa_permutacao *)
```

Figura 9.26: Verifica se um vetor define uma permutação.
Este algoritmo testa para saber se cada índice entre i e n está presente no vetor. Para isto executa no pior caso algo da ordem do quadrado do tamanho do vetor.

No primeiro semestre de 2011 um estudante sugeriu usar um vetor auxiliar, inicialmente zerado, e percorrer o vetor candidato a permutação somente uma vez. Para cada índice, tentar inserir seu respectivo conteúdo no vetor auxiliar: se estiver com um zero, inserir, senão, não é permutação. Se o vetor auxiliar estiver totalmente preenchido então temos um vetor que representa uma permutação. Este processo é linear e está ilustrado na figura 9.27.

```pascal
function testa_permutacao_v2 (var v: vetor_i; n: integer): boolean;
var
  i : integer;
  aux: vetor_i;
  eh_permutacao: boolean;
begin
  zerar_vetor_i (aux,n);
  eh_permutacao := true;
  i := 1;
  while eh_permutacao and (i <= n) do
  begin
    if (v[i] >= 1) AND (v[i] <= n) AND (aux[v[i]] = 0) then
      aux[v[i]] := v[i]
    else
      eh_permutacao := false;
    i := i + 1;
  end;
  testa_permutacao_v2 := eh_permutacao;
end; (* testa_permutacao_v2 *)
```

Figura 9.27: Verifica linearmente se um vetor define uma permutação.

Outros estudantes sugeriram uma conjectura, não provada em sala, de que, se todos os elementos pertencem ao intervalo $1 \leq v[i] \leq n$ e $\sum_{i=1}^{n} v[i] = \frac{n(n+1)}{2}$ e ao mesmo tempo $\prod_{i=1}^{n} v[i] = n!$, então o vetor representa uma permutação. Também não encontramos contraexemplo e o problema ficou em aberto.

**Gerando permutações válidas**

O próximo problema é gerar aleatoriamente uma permutação. Para isto faremos uso da função `random` da linguagem *Pascal*.

O primeiro algoritmo gera um vetor de maneira aleatória e depois testa se o vetor produzido pode ser uma permutação usando o código da função `testa_permutacao` já implementado. A tentativa é reaproveitar código a qualquer custo. Este raciocínio está implementado no código da figura 9.28.

Este algoritmo é absurdamente lento quando $n$ cresce. Isto ocorre pois os vetores são gerados e depois testados para ver se são válidos, mas, conforme esperado, é muito provável que números repetidos sejam gerados em um vetor com grande número de

---

7Bruno Ricardo Sella
9.1. VETORES

procedure gerar_permutacao (var v: vetor; n: integer);
var i: integer;
begin
randomize;
repeat (* repete ate conseguir construir uma permutacao valida *)
for i:= 1 to n do
v[i]:= random (n) + 1; (* sorteia numero entre 1 e n *)
until testa_permutacao_v2 (v, n);
end; (* gera_permutacao *)

Figura 9.28: Gerando uma permutação, versão 1.

elementos. Um dos autores deste material teve paciência de esperar o código terminar até valores de \( n \) próximos de 14. Para valores maiores o código ficou infernalmente demorado, levando várias horas de processamento.

Numa tentativa de melhorar o desempenho, o que implica em abrir mão da comodidade de se aproveitar funções já existentes, este mesmo autor pensou que poderia gerar o vetor de modo diferente: gerar um a um os elementos e testar se eles já não pertencem ao conjunto já gerado até a iteração anterior. Isto garante que o vetor final produzido é válido. A procedure da figura [9.29] apresenta a implementação desta nova ideia.

procedure gerar_permutacao_v2 (var v: vetor; n: integer);
var i, j: integer;
begin
randomize;
v[1]:= random (n) + 1;
for i:= 2 to n do
repeat
v[i]:= random (n) + 1; (* gera um numero entre 1 e n *)
j:= 1; (* procura se o elemento ja existe no vetor *)
while (j < i) and (v[i] <> v[j]) do
j:= j + 1;
until j = i; (* descobre que o elemento eh novo *)
end; (* gera_permutacao_v2 *)

Figura 9.29: Gerando uma permutação, versão 2.

Este algoritmo executa na casa de 2 segundos para vetores de tamanho próximos de 1000, mas demora cerca de 30 segundos para entradas de tamanho que beiram os 30.000. Para se pensar em tamanhos maiores a chance do tempo ficar insuportável é enorme. Mas já é melhor do que o anterior.

Queremos gerar um vetor que represente uma permutação, provavelmente para fins de testes. Uma maneira possível seria a seguinte: inicializa-se um vetor de forma ordenada, depois faz-se alterações aleatórias de seus elementos um número também aleatório de vezes. Esta ideia foi implementada e é mostrada na figura [9.30].
procedure gerar_permutacao_v3 (var v: vetor_i; n: integer);
var i, j, k, aux, max: integer;
begin
  for i:= 1 to n do
    v[i] := i;

  randomize;
  max:= random (1000); (* vai trocar dois elementos de 0 a 999 vezes *)
  for i:= 1 to max do
    begin
      j:= random (n) + 1;
      k:= random (n) + 1;
      aux:= v[j];
      v[j]:= v[k];
      v[k]:= aux;
    end;
end; (* gera_permutacao_v3 *)

Figura 9.30: Gerando uma permutação, versão 3.

Este código produz corretamente vetores que representam permutações com bom grau de mistura dos números em tempo praticamente constante para entradas da ordem de um milhão de elementos (usando-se o tipo `longint`).

Em uma aula do segundo semestre de 2010 surgiu uma nova ideia para se gerar um vetor de permutação.

A sugestão é modificar o algoritmo fazendo com que um vetor auxiliar conte- nha os números ainda não colocados no vetor permutação. O sorteio deixa de ser sobre o elemento a ser inserido, mas agora sobre o índice do vetor auxiliar, cujo tamanho decresce na medida em que os números vão sendo sorteados. O código da figura ilustra estas ideias. Ele foi implementado durante a aula e possibilitou gerar vetores de tamanhos incrivelmente grandes em tempo extremamente curto.

Determinando a ordem de uma permutação

Antes de apresentarmos o próximo problema do ponto de vista algorítmico a ser tratado precisamos introduzi-lo do ponto de vista matemático.

Observem que, uma vez que a função que define a permutação é sobre o próprio conjunto, ocorre que, se \( P(n) \) é uma permutação, então \( P(P(n)) \) também é. Logo, é possível calcular o valor de expressões tais como \( P(P(1)) \). De fato, consideremos novamente a permutação:

---

\[ ^8 \text{Se alguém souber de um modo mais eficiente de gerar uma permutação, favor avisar. Não só daremos os créditos necessários como também mostraremos os resultados aqui neste material.} \]

\[ ^9 \text{Créditos para o Felipe Z. do Nascimento.} \]

\[ ^10 \text{Código e testes feitos na aula por Renan Vedovato Traba, a partir da ideia do Felipe.} \]
9.1. VETORES

```pascal
procedure gerar_permutacao_v4 (var v: vetor_i; n: longint);
var i, j, tam: longint;
    aux: vetor_i;
begin
    for i := 1 to n do
        aux[i] := i;

    randomize;
    tam := n;
    for i := 1 to n do
        begin
            j := random(tam) + 1;
            v[i] := aux[j];
            aux[j] := aux[tam];
            tam := tam - 1;
        end;
end; (* gerar_permutacao_v4 *)
```

Figura 9.31: Gerando uma permutação, versão 4.

\[
\begin{pmatrix}
1 & 2 & 3 & 4 & 5 \\
4 & 1 & 5 & 2 & 3
\end{pmatrix}
\]

Então pode-se calcular:

- \( P(P(1)) = P(4) = 2. \)
- \( P(P(2)) = P(1) = 4. \)
- \( P(P(3)) = P(5) = 3. \)
- \( P(P(4)) = P(2) = 1. \)
- \( P(P(5)) = P(3) = 5. \)

Desta maneira, definimos \( P^2(n) = P(P(n)). \) Em termos gerais, podemos definir o seguinte:

\[
\begin{align*}
P^1(n) &= P(n); \\
P^k(n) &= P(P^{k-1}(n)) \quad k \geq 2.
\end{align*}
\]

Dentre todas as permutações, existe uma especial:

\[
ID = \begin{pmatrix}
1 & 2 & 3 & 4 & 5 \\
1 & 2 & 3 & 4 & 5
\end{pmatrix}
\]

Isto é, \( P(i) = i, \forall i. \) Esta permutação recebe um nome especial \( ID. \) É possível demonstrar que, para quaisquer \( k \) e \( n, ID^k(n) = ID(n). \) Também é possível demonstrar que a sentença seguinte também é válida:
Seja $P(n)$ uma permutação sobre um conjunto de $n$ elementos. Então existe um número natural $k$ tal que $P^k = ID$. Este número natural é chamado de ordem da permutação.

Vamos considerar como exemplo a permutação acima. Podemos calcular para valores pequenos de $k$ como é $P^k$:

$$P = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 4 & 1 & 5 & 2 & 3 \end{pmatrix}$$

$$P^2 = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 2 & 4 & 3 & 1 & 5 \end{pmatrix}$$

$$P^3 = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 1 & 2 & 5 & 4 & 3 \end{pmatrix}$$

$$P^4 = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 4 & 1 & 3 & 2 & 5 \end{pmatrix}$$

$$P^5 = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 2 & 4 & 5 & 1 & 3 \end{pmatrix}$$

$$P^6 = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 1 & 2 & 3 & 4 & 5 \end{pmatrix}$$

Isto é, a ordem da permutação $P$ é 6.

Chegamos no ponto de apresentarmos o próximo problema. Dada uma permutação, encontrar sua ordem. Simular a sequência de operações e testar quando a identidade for encontrada, contando quantas operações foram feitas é muito caro. Tem que haver uma maneira melhor.

A função da figura implementa um algoritmo que recebe como entrada uma permutação (válida) e retorna sua ordem.

Este algoritmo parte da ideia de que cada elemento $P(i) = x$ do conjunto retorna à posição $i$ ciclicamente, de cont em cont permutações. Ou seja, $P^\text{cont}(i) = x$, $P^{2 \times \text{cont}}(i) = x$, ... O mesmo ocorre para todos elementos do conjunto, mas cada um possui um ciclo (valor de cont) próprio.

Para exemplificar, tomemos a permutação acima. Para o índice 1, temos que $P^3(1) = 1$. Isto quer dizer que para todo múltiplo de 3 (a cada 3 iterações) é verdade que $P^{3k}(1) = 1$. Isto também ocorre para os índices 2 e 4. Mas para os índices 3 e 5, o número de iterações para que ocorra uma repetição é de duas iterações. Logo, pode-se concluir que a permutação ID ocorrerá exatamente na iteração que é o mínimo múltiplo comum (MMC) entre o número que provoca repetição entre todos os índices. Observamos que:

$$\text{MMC}(x_1, x_2, \ldots, x_n) = \text{MMC}(x_1, \text{MMC}(x_2, \ldots, x_n)).$$

Infelizmente, não existe algoritmo eficiente para cálculo do MMC. Mas existe para o cálculo do MDC (máximo divisor comum). De fato, implementamos o algoritmo de

\(^{11}\)Este é o problema da Maratona da ACM.
function ordem_permutacao (var v: vetor_i; n: integer): int64;
var mmx, cont: int64;
p, i: integer;
begin
    mmx := 1;
    for i := 1 to n do
    begin
        cont := 1;
p := i;
        while (v[p] <> i) do
        begin
            cont := cont + 1;
p := v[p];
        end;
        mmx := mmx * cont div mdc(mmx, cont);
    end;
    ordem_permutacao := mmx;
end;

Figura 9.32: Calcula a ordem de uma permutação.

Euclides (figura 7.3, seção ??) e mostramos que ele é muito eficiente. Felizmente, a seguinte propriedade é verdadeira:

\[ MDC(a, b) = \frac{a \times b}{MMC(a, b)} \]

O programa acima explora este fato e torna o código muito eficiente para calcular a ordem de permutações para grandes valores de \( n \). O estudante é encorajado aqui a gerar uma permutação com os algoritmos estudados nesta seção e rodar o programa para valores de \( n \) compatíveis com os tipos de dados definidos (integer).

Polinômios

Nesta seção vamos mostrar como representar e fazer cálculos com polinômios representados como vetores.

Para uma sucessão de termos \( a_0, \ldots, a_n \in \mathbb{R} \), podemos para este curso definir um polinômio de grau \( n \) como sendo uma função que possui a seguinte forma:

\[ P(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0 \]

Vamos considerar ao longo desta seção que \( \forall k > n \), então \( a_k = 0 \). Também consideramos que \( a_n \neq 0 \) para um polinômio de grau \( n \).

Do ponto de vista computacional, uma possível representação para um polinômio é um vetor de \( n + 1 \) elementos cujos conteúdos são os coeficientes reais dos respectivos monômios. Consideremos então o tipo abaixo, podemos exemplificar alguns casos:

type polinomio = array [0..max] of real;

- \( P(x) = 5 - 2x + x^2 \):
• $P(x) = 7 - 2x^2 + 8x^3 - 2x^7$

No restante desta seção iremos mostrar algoritmos que realizam operações costumeadas sobre polinômios. A primeira é calcular o valor de um polinômio em um dado ponto $x \in \mathbb{R}$. Por exemplo, se $P(x) = 5 - 2x + x^2$ então, $P(1) = 5 - 2 \times 1 + 1^2 = 4$.

A função em *Pascal* que implementa este cálculo está apresentado na figura 9.33.

```
function valor_no_ponto (var p: polinomio; graup: integer; x: real): real;
var soma, potx: real;
i : integer;
begin
   potx:= 1;
soma:= 0;
   for i:= 0 to graup do
   begin
      soma:= soma + p[i] * potx;
potx:= potx * x;
   end;
   valor_no_ponto:= soma;
end;
```

Figura 9.33: Calcula o valor de $P(x)$ para um dado $x \in \mathbb{R}$.

O próximo algoritmo interessante é o cálculo do polinômio derivada de um polinômio $P$. Seja $P'(x)$ a derivada de $P(x)$ assim definido:

$$P'(x) = na_n x^{n-1} + (n-1)a_{n-1} x^{n-2} + \ldots + 2a_2 x + a_1$$

O programa que implementa este cálculo está na figura 9.34.

Por outro lado, para calcular o valor no ponto de uma derivada de um polinômio, não é necessário que se calcule previamente um vetor auxiliar contendo a derivada. Isto pode ser feito diretamente usando-se o polinômio $P$, bastando trabalhar corretamente os índices do vetor, conforme mostrado na figura 9.35.

Os próximos problemas vão nos permitir trabalhar um pouco com os índices do vetor. O primeiro problema é a soma de polinômios. O segundo é a multiplicação. Antes mostraremos a definição matemática para estes conceitos.

Sejam dois polinômios $P$ e $Q$ assim definidos, supondo que $n \geq m$:

$$P(x) = a_n x^n + \ldots + a_m x^m + \ldots + a_1 x + a_0$$

$$Q(x) = b_n x^n + b_{n-1} x^{n-1} + \ldots + b_1 x + b_0$$
9.1. VETORES

```pascal
procedure derivar (var p: polinomio; graup: integer;
var d: polinomio; var graud: integer);
var i : integer;
begin
  if graup = 0 then
    begin
      graud:= 0;
      d[0]:= 0;
    end
  else
    begin
      graud:= graup − 1;
      for i:= 0 to graud do
        d[i]:= (i+1) ∗ p[i+1];
    end;
end;
```

Figura 9.34: Calcula o polinômio derivada de $P(x)$.

```pascal
function valor_derivada_no_ponto (var p: polinomio; graup: integer; x: real): real;
var i : integer;
soma, potx: real;
begin
  soma:= 0;
potx:= 1;
  for i:= 1 to graup do
    begin
      soma:= soma + i ∗ p[i] ∗ potx;
potx:= potx ∗ x;
    end;
  valor_derivada_no_ponto:= soma;
end;
```

Figura 9.35: Calcula o valor de $P'(x)$ para um dado $x \in \mathbb{R}$.

Então o polinômio soma de $P$ e $Q$, denotado $P + Q$ é assim definido:

$$(P + Q)(x) = a_n x^n + \ldots + a_{m+1} x^{m+1} + (a_m + b_m) x^m + \ldots + (a_1 + b_1) x + (a_0 + b_0)$$

Basicamente é a mesma operação de soma de vetores estudada neste capítulo, embora naquele caso exigimos que os tamanhos dos vetores fossem iguais. No caso de polinômios os vetores podem ter tamanhos diferentes desde que se assuma que os coeficientes que faltam no polinômio de maior graus são nulos. A implementação deste processo está na figura 9.36.

Considerando os mesmos polinômios $P$ e $Q$ acima definidos, o produto de $P$ por $Q$, denotado $PQ$ é assim definida:
procedure somar (var p: polinomio; graup: integer;
var q: polinomio; grauq: integer;
var s: polinomio; var graus: integer);
var i,menorgrau: integer;
begin
(* o grau do pol soma eh o maior grau entre p e q *)
(* copiar os coeficientes que o maior pol tem a mais *)
if graup > grauq then
begin
graus:= graup;
menorgrau:= grauq;
for i:= menorgrau+1 to graus do
  s[i]:= p[i];
end
else
begin
graus:= grauq;
menorgrau:= graup;
for i:= menorgrau+1 to graus do
  s[i]:= q[i];
end;
for i:= 0 to menorgrau do
  s[i]:= p[i] + q[i];
end;

Figura 9.36: Calcula a soma de \( P(x) \) com \( Q(x) \).

\[
(PQ)(x) = (a_{n+m}b_0 + \ldots + a_n b_m + \ldots + a_0 b_{n+m})x^{n+m} + \ldots +
(a_k b_0 + a_{k-1} b_1 + \ldots + a_0 b_k)x^k + \ldots + (a_1 b_0 + a_0 b_1)x + (a_0 b_0)
\]

A operação matemática exige que sejam feitas todas as multiplicações e posterior agrupamento dos monônomos de mesmo grau, somando-se os coeficientes, para cada monônoimio.

O programa apresentado na figura 9.37 implementa os cálculos para obtenção do polinômio produto de \( P \) por \( Q \). O programa realiza os cálculos para cada monônoimio à medida em que os índices dos dois comandos for variam, o que é um uso especial da técnica dos acumuladores, embora os acumuladores não sejam simultâneos para cada monônoimio do resultado final da operação, eles são feitos aos poucos. Para isto é preciso zerar o vetor antes de começar.
procedure multiplicar (var p: polinomio; graup: integer;
var q: polinomio; grauq: integer;
var m: polinomio; var grauM: integer);

var i, j: integer;
begin
  grauM:= graup + grauq;
  for i:= 0 to grauM do 
m[i]:= 0;

  for i:= 0 to graup do 
    for j:= 0 to grauq do 
      m[i+j]:= m[i+j] + p[i]*q[j];

  if ((graup = 0) and (p[0] = 0)) or 
     ((grauq = 0) and (q[0] = 0)) then 
    grauM:= 0;
end;

Figura 9.37: Calcula o produto de $P(x)$ com $Q(x)$. 
9.1.6 Exercícios

1. Faça um programa em *Pascal* que leia e armazene em um vetor uma sequência de inteiros terminada em zero. Em seguida o programa deve ler uma outra sequência de inteiros terminada em zero e, para cada um deles, dizer se ele pertence ou não ao vetor armazenado previamente.

2. Faça um programa em *Pascal* que leia duas sequências de *n* inteiros terminadas em zero em dois vetores distintos, digamos, *v* e *w* e verifique se os dois vetores são idênticos.

3. Faça um programa em *Pascal* em que leia duas sequências de números inteiros e armazene em dois vetores. Seu programa deve descobrir se um deles é permutação do outro, isto é, se eles tem os mesmos elementos, ainda que em ordem diferente. A quantidade de elementos lidos em cada vetor é no máximo 100, e cada sequência termina quando o valor 0 é digitado. Por exemplo:

   
   [2, 2, 0, 3, 4] e [2, 2, 0, 3, 4]: sim.
   [2, 2, 0, 3, 4] e [4, 3, 2, 0, 2]: sim.
   [2, 2, 0, 3, 4] e [4, 3, 4, 0, 2]: não.
   
   [3, 0, 5] e [3, 0, 5, 3]: não.

Implemente três versões deste problema:

- ordenando os vetores para em seguida compará-los;
- sem ordenar os vetores;
- crie uma função que retorna 0 se *x* não pertence a *v* e caso contrário retorna o índice do vetor onde *x* se encontra. Use esta função para resolver este problema.

4. Faça um programa em *Pascal* que leia duas sequências de inteiros terminadas em zero, não necessariamente contendo a mesma quantidade de números. Seu programa deverá:

   - dizer se a segunda sequência está contida na primeira. Exemplo:
   ```plaintext
   v1: 7 3 2 3 2 6 4 7
   v2: 3 2 6
   Saída: sim
   ```
   
   - construir um terceiro vetor, sem destruir os originais, que é a concatenação do primeiro com o segundo;
   ```plaintext
   v1: 7 3 2 6
   v2: 5 1 8 4 9
   Saída: 1 2 3 4 5 6 7 8 9
   ```
   
   - ordená-los, e em seguida imprimir todos os números ordenados em ordem crescente. Exemplo:
9.1. VETORES

\[ v_1: 7 \ 3 \ 2 \ 6 \]
\[ v_2: 5 \ 1 \ 8 \ 4 \ 9 \]
Saída: 1 2 3 4 5 6 7 8 9

5. Crie uma função em *Pascal* que receba um vetor de inteiros de tamanho \( n \) e devolva o valor *true* se o vetor estiver ordenado e *false* em caso contrário.

6. Aproveitando as soluções dos problemas anteriores, faça um programa em *Pascal* que leia dois vetores de inteiros \( v \) e \( w \), de dimensões \( m \) e \( n \) respectivamente, verifique se eles estão ordenados, ordene-os em caso contrário e, em seguida, imprima a intercalação dos dois. Cada sequência termina quando um zero for digitado. Exemplo de intercalação: \( v: 1 \ 4 \ 6 \ 9 \); \( w: 2, 3, 5, 7 \).
Saída: 1, 2, 3, 4, 5, 6, 7, 9.

7. Dados dois números naturais \( m \) e \( n \), uma frase com \( m \) letras e uma palavra com \( n \) letras, faça um procedimento em *Pascal* que determine o número de vezes que a palavra ocorre na frase e a posição em que cada ocorrência inicia.
Exemplo:

Para \( M = 30, N = 3 \), a palavra ANA e a frase:

ANA E MARIANA GOSTAM DE BANANA

A palavra ANA ocorre 4 vezes, nas posições 1, 11, 26, 28.

8. Dada uma sequência de \( N \) números, faça um programa em *Pascal* que imprima quantos números distintos compõe a sequência e o número de vezes que cada um deles ocorre na mesma. Exemplo:

\( N=5 \ 1 \ 2 \ 3 \ 2 \ 3 \) a sequência tem três números distintos, 1, 2 e 3. Ocorrências: 1 ocorre 1 vez 2 ocorre 2 vezes 3 ocorre 2 vezes

9. Faça um programa em *Pascal* em que leia os seguintes valores: um inteiro \( B \), um inteiro \( N \) (\( 1 \leq N \leq 10 \)), e \( N \) valores inteiros. A ideia é que estes valores sejam entendidos como a representação de um número não negativo na base \( B \). Estes valores deverão ser inseridos em um vetor de tamanho \( N + 1 \), onde a primeira posição armazena a base \( B \) e as outras \( N \) posições o restante dos números lidos. Note que o intervalo de valores possíveis para cada dígito na base \( B \) é \([0, B - 1]\). Seu programa deve retornar o valor em decimal do número representado no vetor. Se o número representado no vetor não for válido na base \( B \) então deverá ser retornado o código de erro “-1”. Por exemplo, se \( B = 3 \) o número 2102 na base 3 equivale ao valor decimal 65; se \( B = 4 \) o número 35 é inválido na base 4.

10. Faça um programa em *Pascal* que, dadas duas sequências com \( N \) números inteiros entre 0 e 9, interpretadas como dois números inteiros de \( N \) algarismos, calcular a sequência de números que representa a soma dos dois inteiros, usando
o “vai-um”. Por exemplo:

\[ N=6, \]
\[ 4 \ 3 \ 4 \ 2 \ 5 \ 1 \]
\[ + \ 7 \ 5 \ 2 \ 3 \ 3 \ 7 \]
\[ 1 \ 1 \ 8 \ 6 \ 5 \ 8 \ 8 \]

11. Dada uma sequência \( x_1, x_2, \ldots, x_n \) de números inteiros terminadas em zero, faça um programa em *Pascal* que determine um segmento de soma máxima. Exemplo: na sequência 5, 2, -2, -7, 3, 14, 10, -3, 9, -6, 4, 1, a soma do maior segmento é 33, obtida pela soma dos números de 3 até 9.

12. Faça um programa em *Pascal* que leia um vetor de 1 milhão de inteiros em um vetor de inteiros. Gere um número aleatório e o procure neste vetor de duas maneiras diferentes: uma usando busca com sentinela e outra usando busca binária. Seu programa deve imprimir uma tabela com o número de comparações feitas em cada um dos casos acima (busca com sentinela e busca binária). Desconsidere o tempo gasto com ordenação no caso da busca binária. A busca com sentinela deve ser feita em um vetor não ordenado. Gere 200 números aleatórios e imprima a média de comparações para cada um dos dois algoritmos sendo testados.

13. Suponha que um exército tenha 20 regimentos e que eles estão em processo de formação. Inicialmente o primeiro tem 1000 homens, o segundo 950, o terceiro 900, e assim por diante, até o vigésimo que tem 50. Suponhamos que a cada semana 100 homens são enviados para cada regimento, e no final da semana o maior regimento é enviado para o *front*. Imaginemos que o general do quinto regimento é companheiro de xadrez do comandante supremo, e que eles estão no meio de uma partida. O comandante supremo então envia apenas 30 homens para o quinto regimento a cada semana, esperando com isto poder acabar o jogo com seu colega. Faça um programa em *Pascal* que diga, a cada semana, qual é o regimento enviado ao *front* e mostre o status dos outros regimentos. O programa deve também determinar exatamente quantas semanas levará o quinto regimento para ser deslocado ao *front*.


(a) 10, 9, 8, 7, 6, 5, 4, 3, 2, 1.
(b) 5, 4, 3, 2, 1, 10, 9, 8, 7, 6.
(c) 10, 1, 9, 2, 8, 3, 7, 4, 6, 5.
(d) 2, 3, 4, 5, 6, 7, 8, 9, 10, 1.
(e) 1, 10, 2, 9, 3, 8, 4, 7, 5, 6.
15. Suponha que você tenha uma variável do tipo vetor declarada como: `array [1..50] of real;`. Faça uma função em `Pascal` que inicialize o vetor de modo que os elementos de índices ímpares recebam o valor inicial -2.0 e os elementos de índices pares recebam o valor inicial 7.0. Sua função deve fazer uso de um único comando de repetição, que incrementa de um em um, e de nenhum comando de desvio condicional.

16. Qual dos seguintes problemas requer o uso de vetores para uma solução elegante?

(a) Ler cerca de duzentos números e imprimir os que estão em uma certa faixa;
(b) Computar a soma de uma sequência de números;
(c) Ler exatamente duzentos números e ordená-los em ordem crescente;
(d) Encontrar o segundo menor elemento de uma sequência de entrada;
(e) Encontrar o menor inteiro de uma sequência de inteiros.

17. Considere um vetor declarado como: `array [1..50] of integer` que tem a particularidade de todos os elementos estarem entre 1 e 30, sendo que nenhum é repetido. Faça um programa em `Pascal` que ordene o vetor de maneira eficiente explorando esta característica e fazendo o menor número possível de trocas.

18. Dada uma sequência \( x_1, x_2, \ldots, x_k \) de números reais, faça um programa em `Pascal` que verifique se existem dois segmentos consecutivos iguais nesta sequência, isto é, se existem \( i \) e \( m \) tais que:

\[
x_i, x_{i+1}, \ldots, x_{i+m-1} = x_{i+m}, x_{i+m+1}, \ldots, x_{i+2m-1}.
\]

Imprima, caso existam, os valores de \( i \) e de \( m \). Caso contrário, não imprima nada. Exemplo: Na sequência 7,9,5,4,5,4,8,6, existem \( i = 3 \) e \( m = 2 \).

19. Um coeficiente binomial, geralmente denotado \( \binom{n}{k} \), representa o número de possíveis combinações de \( n \) elementos tomados \( k \) a \( k \). Um “Triângulo de Pascal”, uma homenagem ao grande matemático Blaise Pascal, é uma tabela de valores de coeficientes combinatoriais para pequenos valores de \( n \) e \( k \). Os números que não são mostrados na tabela têm valor zero. Este triângulo pode ser construído automaticamente usando-se uma propriedade conhecida dos coeficientes binomiais, denominada “fórmula da adição”: \( \binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1} \). Ou seja, cada elemento do triângulo é a soma de dois elementos da linha anterior, um da mesma coluna e um da coluna anterior. Veja um exemplo de um triângulo de Pascal com 7 linhas:

\[
\begin{array}{ccccccc}
1 \\
1 & 1 \\
1 & 2 & 1 \\
1 & 3 & 3 & 1 \\
1 & 4 & 6 & 4 & 1 \\
1 & 5 & 10 & 10 & 5 & 1 \\
1 & 6 & 15 & 20 & 15 & 6 & 1 \\
\end{array}
\]
Faça um programa em *Pascal* que imprima na tela um triângulo de Pascal com 10 linhas. Seu programa deve obrigatoriamente fazer uso de exatamente dois vetores durante o processo de construção. Um deles conterá a última linha ímpar gerada, enquanto que o outro conterá a última linha par gerada. Lembre-se que os elementos que não aparecem na tabela tem valor nulo. Você deve sempre ter o controle do tamanho da última linha impressa (o tamanho útil dos vetores em cada passo). Você deve também usar um procedimento para imprimir o vetor. Observe que não há entrada de dados, os dois vetores são gerados, um a partir do outro. O único elemento da primeira linha tem o valor 1. Você deve obrigatoriamente declarar um tipo vetor com tamanho máximo $\text{Tam\_max\_vetor}$, e o seu programa deverá tomar cuidado para manipular corretamente vetores de tamanho menor do que o tamanho máximo, impedindo que haja uma atribuição em posição ilegal de memória.

20. Seja um polinômio $p(x) = a_0 + a_1 x + a_2 x^2 + \ldots + a_n x^n$ de grau $n \geq 2$. Uma possível maneira de calcular uma raiz do polinômio é pelo “método de Newton”. Este método consiste em se fornecer uma aproximação inicial para a raiz, isto é, um valor que não é a raiz exata, mas é um valor próximo. Assim, se $x_0$ é esta aproximação inicial, $p(x_0)$ não é zero mas espera-se que seja próximo de zero. A obtenção da raiz pelo método de Newton é feita pelo refinamento desta solução inicial, isto é, pela tentativa de minimizar o erro cometido. Isto é feito pela expressão seguinte:

$$x_{n+1} = x_n - \frac{p(x_n)}{p'(x_n)};$$

$n = 0, 1, 2, \ldots$, e onde $p'(x)$ é a primeira derivada de $p(x)$. Usualmente, repete-se este refinamento até que $|x_{n+1} - x_n| < \epsilon, \epsilon > 0$, ou até que $m$ iterações tenham sido executadas.

Faça um programa em *Pascal* que receba como dados de entrada um polinômio $p(x) = a_0 + a_1 x + a_2 x^2 + \ldots + a_n x^n$ e uma aproximação inicial $x_0$ da raiz de $p(x)$, $\epsilon > 0$ e o número máximo de iterações, e calcule uma aproximação da raiz de $p(x)$ pelo método de Newton. Utilize obrigatoriamente um procedimento que receba como parâmetro um polinômio $p(x)$ (incluindo a informação sobre o grau do polinômio) e que calcule e retorne a função derivada $p'(x)$. Utilize também uma função que receba como parâmetros um polinômio $p(x)$ e um valor real $\overline{x}$ e retorne o valor do polinômio no ponto $\overline{x}$, isto é $p(\overline{x})$. Use esta função para calcular, a cada iteração do método de Newton, os valores de $p(x_n)$ e de $p'(x_n)$.

21. Faça um programa em *Pascal* que leia uma sequência de 10 letras (caráteres de A a Z), as armazene em um vetor de 10 posições e imprima a lista de letras repetidas no vetor. Sendo assim, para os dados: A J G A D F G A A, a saída deve ser: A G.

22. Faça um programa em *Pascal* que implemente uma variante da busca binária. Dado um vetor de inteiros, ao invés de achar a primeira ocorrência do valor na lista, identifique e imprima o menor índice do vetor no qual o valor ocorra.
23. Faça um programa em *Pascal* que leia uma sequência de *código de operação* e *valor*, onde o *código de operação* é um inteiro com os seguintes valores:

- 0 (zero): fim
- 1 (um): inserção
- 2 (dois): remoção

O *valor* lido é um real que deve ser inserido em um vetor (caso a operação seja 1), ou removido do vetor (caso a operação seja 2). As inserções no vetor devem ser realizadas de forma que o vetor esteja sempre ordenado. No final do programa o vetor resultante deve ser impresso.

**Detalhamento:**

- a quantidade máxima de valores que pode ser inserida é 100;
- se a quantidade máxima for ultrapassada o programa deve dar uma mensagem de erro;
- se for requisitada a remoção de um número não existente o programa deve dar uma mensagem de erro;
- se o código de operação for inválido o programa deve continuar lendo um novo código até que ele seja 0 (zero), 1 (um) ou 2 (dois).

**Exemplo de execução:**

```plaintext
Entre com operacao (0=fim, 1=insercao, 2=remocao): 1
Valor: 45.3
Entre com operacao (0=fim, 1=insercao, 2=remocao): 1
Valor: 34.3
Entre com operacao (0=fim, 1=insercao, 2=remocao): 1
Valor: 40.8
Entre com operacao (0=fim, 1=insercao, 2=remocao): 2
Valor: 34.3
Entre com operacao (0=fim, 1=insercao, 2=remocao): 0
Vetor resultante
40.8 45.3
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>início</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>45.3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>após inserção de 45.3</td>
</tr>
<tr>
<td>34.3</td>
<td>45.3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>após inserção de 34.3</td>
</tr>
<tr>
<td>34.3</td>
<td>40.8</td>
<td>45.3</td>
<td></td>
<td></td>
<td></td>
<td>após inserção de 40.8</td>
</tr>
<tr>
<td>40.8</td>
<td>45.3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>após remoção de 34.3</td>
</tr>
</tbody>
</table>
24. Faça um programa em Pascal que leia duas sequências de caracteres e verifique se a segunda sequência é subpalavra da primeira. Por exemplo, todo é subpalavra de metodo e ar é subpalavra de farmacia. Porém, todo não é subpalavra de todavia. A leitura das seqüências deve ser feita caractere por caractere e o final de cada sequência é sinalizada pelo caractere ".". Se a segunda sequência é uma subpalavra, a saída do programa deve ser a posição na qual ela começa. Caso contrário, escrever a mensagem “Não eh subpalavra.”. Observações:

- cada sequência tem no máximo 80 caracteres.
- você não pode utilizar funções de manipulação de cadeias de caracteres existentes no compilador, mas somente as funções para o tipo char.

Exemplo de execução:

Entre com duas palavras terminadas por ponto:
metodo.todo.

A segunda subpalavra começa na posição 3 da primeira.

25. Faça um programa em Pascal que leia uma sequência de n valores reais (n ≤ 100) e os insira num vetor. A sequência termina quando o valor lido for 0. O programa deve escrever o valor da divisão da soma dos valores positivos pela soma dos valores negativos que estão armazenados no vetor. Cuidado com divisões por zero.

26. Faça uma função em Pascal que substitua em um texto a primeira ocorrência de uma palavra por outra. A função deve retornar true se a substituição for bem sucedida e false caso a palavra não seja encontrada no texto. O texto e as palavras são representados por vetores do tipo char. Por exemplo:

<table>
<thead>
<tr>
<th>texto1</th>
<th>e</th>
<th>x</th>
<th>e</th>
<th>m</th>
<th>p</th>
<th>r</th>
<th>o</th>
<th>u</th>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>palavra1</td>
<td>r</td>
<td>o</td>
<td>u</td>
<td>n</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>palavra2</td>
<td>l</td>
<td>o</td>
<td>d</td>
<td>o</td>
<td>i</td>
<td>s</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>texto2</td>
<td>e</td>
<td>x</td>
<td>e</td>
<td>m</td>
<td>p</td>
<td>l</td>
<td>o</td>
<td>d</td>
<td>i</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

A função recebe como parâmetros o texto, a palavra a ser substituída e a nova palavra. No exemplo, texto1 mostra o estado inicial do texto e texto2 o estado do texto após a substituição da palavra1 pela palavra2.

Você pode usar, caso seja necessário, a função:

`buscachar(texto, pos, letra);`
que busca um caractere (letra) a partir de uma determinada posição (pos) em um vetor que contém o texto (texto). A função buscaletra retorna a posição no vetor texto da primeira ocorrência de letra, se letra não aparece no texto a função retorna -1.

27. Um *algoritmo genético* é um procedimento computacional de busca, inspirado no processo biológico de evolução, que otimiza a solução de um problema. O problema é modelado por: uma população de indivíduos que representam possíveis soluções; uma função que avalia a qualidade da solução representada por cada indivíduo da população e um conjunto de operadores genéticos. Os indivíduos são dados por sequências de genes que representam características da solução do problema. O procedimento consiste em aplicar os operadores genéticos sobre a população, gerando novos indivíduos e selecionar os mais aptos para constituírem uma nova população. Esse processo é repetido até que uma solução adequada seja obtida. Dentre os operadores genéticos, o mais importante é o de recombinação genética (crossover) de dois indivíduos. Esse operador corta em duas partes as sequências de genes de dois indivíduos pais (pai1 e pai2) e gera dois novos indivíduos filhos (filho1 e filho2). filho1 é dado pela concatenação da primeira parte dos genes de pai1 com a segunda parte de pai2 e filho2 pela concatenação da primeira parte de pai2 com a segunda parte de pai1. O diagrama abaixo exemplifica a operação em indivíduos representados por vetores de números inteiros onde a primeira posição contém o tamanho do vetor:

Faça um procedimento em *Pascal* que execute a operação de recombinação descrita acima, usando a estrutura de dados vetor. O procedimento deve receber seis parâmetros, um vetor representando o primeiro pai, a posição de corte no primeiro pai, um vetor representando o segundo pai, a posição do corte no segundo pai, e dois vetores que receberão os novos indivíduos. No exemplo apresentado a chamada do procedimento seria:

```pascal
    corte1 := 4;
    corte2 := 6;
    crossover(pai1, corte1, pai2, corte2, filho1, filho2);
```
Note que os vetores devem iniciar na posição zero e essa posição é usada para armazenar o tamanho do vetor. No caso do exemplo, pai1[0]=11, pai2[0]=10, filho1[0]=8 e filho2[0]=13. Os pontos de corte devem estar dentro dos vetores: 1 < corte1 <= pai1[0] e 1 < corte2 <= pai2[0].

28. Faça um procedimento em Pascal que implemente a subtração de números binários. Considere que os números binários têm N bits e que os bits são armazenados em vetores de inteiros de N posições indexadas de 1 a N. O primeiro bit do vetor representa o sinal do número, sendo zero (0) para os números positivos e um (1) para negativos. Os demais bits representam o valor absoluto do número. Por exemplo, para N = 11, os números decimais −13, 12 e 1010 são representados pelos seguintes vetores:

```
-13:  | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 |
```

```
12:   | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 |
```

```
1010: | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 0 |
```

O procedimento recebe dois vetores do mesmo tamanho como parâmetros e deve gerar como resultado um vetor que contenha a subtração do primeiro pelo segundo. Por exemplo −12 − 1010 = −1022:

```
-998: | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 |
```

Caso o resultado tenha mais bits que o espaço disponível no vetor o procedimento deve descartar os bits excedentes.

29. Faça um programa em Pascal que leia uma sequência de N valores reais não nulos (N ≤ 100) e os insira em um vetor. A sequência termina quando o valor lido for 0. O elemento zero não faz parte do vetor. Leia do teclado um inteiro p (p ≤ N) e considere que o elemento V[p] como o pivô na operação de rearranjar o vetor de tal maneira que todos os elementos à esquerda de V[p] sejam menores que ele e todos os da direita sejam maiores ou iguais a ele. Por exemplo, considere o seguinte vetor dado como entrada:

```
| 99.7 | 32.6 | 2.45 | 13.4 | 26.7 | 12.2 | 0.51 |
```
e $P$ valendo 3, o programa deve gerar como resultado um vetor onde todos os elementos que estão à esquerda do valor 2.45 no vetor são menores que ele, enquanto que os da direita são maiores do que ele.

30. Faça uma função em *Pascal* que procure uma palavra dentro de um texto. A função deve receber como parâmetros:

- um vetor do tipo `texto` que contém um texto;
- o tamanho do vetor que contém o texto;
- a posição inicial da busca dentro do vetor que contém o texto;
- um vetor do tipo `texto` que contém uma palavra;
- o tamanho do vetor que contém a palavra.

A função deve retornar um número inteiro indicando a posição no texto onde a palavra foi encontrada pela primeira vez.

Caso a palavra não seja encontrada ou algum erro ocorra o valor retornado deve ser zero. A busca pela palavra no texto deve iniciar na posição passada como parâmetro para a função.

O tipo `texto` é dado por:

```pascal
const
  TAMMAX = 10000;

type
  texto = array [1..TAMMAX] of char;
```

31. Faça um programa em *Pascal* que gere e imprima um vetor de números reais de tamanho $N$, $1 \leq N \leq MAX$. A criação do vetor deve ser feita da seguinte maneira:

- O tamanho $N$ do vetor deve ser lido do teclado;
- Os $N$ números reais são gerados aleatoriamente no intervalo $[R_{min}, R_{max}]$, com $R_{min}$ e $R_{max}$ lidos do teclado;
- A posição em que cada elemento real é inserida no vetor também é gerada aleatoriamente;
- Se uma posição $i$ sorteada já estiver ocupada, seu algoritmo deve encontrar a primeira posição $j$ não ocupada, iniciando a partir de $i+1$ até o final do vetor. Se todas as posições entre $i+1$ e o final do vetor estiverem ocupadas, seu algoritmo deve pegar a primeira posição livre a partir do início do vetor.

Dica: a função `random` sem parâmetros retorna um número real no intervalo $[0, 1]$, e a função `random(n)` retorna um número inteiro no intervalo $[0, n]$.

32. Faça um procedimento em *Pascal* que remova um elemento de uma determinada posição $p$ de um vetor $v$ de $n$ números reais. O vetor não está ordenado. Use a seguinte assinatura para o procedimento:

```pascal
procedure remove(var v: vetor; var n: integer; p: integer);
```
33. Faça um procedimento em *Pascal* que altere um vetor de *N* números reais da seguinte forma: todos os elementos repetidos do vetor devem ir para o final do vetor, mas de maneira que estes últimos fiquem em ordem crescente. Exemplos:

```
ENTRADA: 5 3 8 2 3 9 8 9 7 5 3
SAÍDA : 5 3 8 2 9 7 3 3 5 8 9
```

```
ENTRADA: 4 4 3 3 2 2
SAÍDA : 4 3 2 2 3 4
```

34. Em uma festa estiveram presentes 150 pessoas. Cada uma delas recebeu um crachá na entrada com um número entre 1 e 150, número que representa a ordem de entrada de cada convidado.

Como em toda festa, cada um dos presentes cumprimentou outras pessoas com apertos de mão. Ao final da festa, cada convidado sabia exatamente quantas vezes tinha apertado a mão de outras pessoas.

Na saída, ao entregar o crachá ao recepcionista, cada convidado informou o número do seu crachá e quantas vezes trocou apertos de mão na festa.

Muito curioso, o recepcionista queria saber quantos convidados eram muito populares no encontro, isto é, queria saber o número de pessoas que apertaram a mão de pelo menos outros 120 convidados.

Faça um programa em *Pascal* que modele o problema do recepcionista e que produza como saída o número de celebridades (cumprimentadas pelo menos 120 vezes) presentes na festa.

35. Um procedimento chamado *nova_geracao* recebe como parâmetros dois vetores (origem e destino) e o tamanho dos vetores. Este procedimento constrói um novo vetor de valores 0 ou 1 (destino) a partir do conteúdo do primeiro vetor (origem). Seja *O* o vetor origem e *D* o vetor destino, a regra para a construção do novo vetor é dada por:

- se: *O*[i−1] = 0, *O*[i] = 0, *O*[i+1] = 0 então: *D*[i] = 0
- se: *O*[i−1] = 1, *O*[i] = 1, *O*[i+1] = 0 então: *D*[i] = 0

Onde *i* indica uma posição do vetor *D*. Considere o valor 0 para as bordas externas do vetor origem *O*. Faça em *Pascal* o procedimento *nova_geracao* e separe a regra descrita acima em uma função que dados os valores das 3 posições consecutivas do vetor origem (*O*[i−1], *O*[i], *O*[i+1]) calcula o valor correspondente no vetor destino (*D*[i]).

36. Faça um programa em *Pascal* que simule o tráfego em um trecho de uma rodovia de mão única, ou seja, uma rodovia na qual os veículos entram de um lado e saem do outro.
9.1. VETORES

- A rodovia é representada por um vetor com TAM_RODOVIA posições;
- A simulação ocorre durante MAX_TEMPO iterações;
- Através da chamada do procedimento
  
  detecta_entrada(VAR tipo, placa, velocidade:INTEGER),

  o programador é informado sobre a ocorrência ou não da entrada de um veículo na rodovia, bem como o tipo do veículo, sua placa e sua respectiva velocidade, onde:
  
  - tipo: 0 - nenhuma nova entrada, 1 - entrou automóvel, 2 - entrou caminhão;
  - placa: um número inteiro;
  - velocidade: a velocidade de deslocamento do veículo (em posições/unidade de tempo).

- Veículos do tipo automóvel ocupam uma posição da rodovia. Caminhões ocupam duas posições.
- Quando veículos mais rápidos alcançam veículos mais lentos, os primeiros devem andar mais devagar, pois não podem ultrapassar.

A cada unidade de tempo em que algum veículo sair da rodovia, seu programa deve imprimir esta unidade de tempo e o número da placa do veículo que saiu.

Exemplo: (TAM_RODOVIA=7, MAX_TEMPO=10)

- Entrada:
  
  - t=1: tipo = 2, placa = 35, velocidade = 1
  - t=2: tipo = 0
  - t=3: tipo = 1, placa = 27, velocidade = 4
  - t=4: tipo = 0
  - t=5: tipo = 0
  - t=6: tipo = 1, placa = 16, velocidade = 2
  - t=7: tipo = 0
  - t=8: tipo = 0
  - t=9: tipo = 0
  - t=10: tipo = 0

- Representação gráfica:
  
  - t=1: | 35_1 | 35_1 |
  - t=2: | 35_1 | 35_1 |
  - t=3: | 27_4 | 35_1 | 35_1 |
  - t=4: | 27_4 | 35_1 | 35_1 |
  - t=5: | 27_4 | 35_1 | 35_1 |
  - t=6: | 16_2 | | 27_4 | 35_1 | 35_1 |
  - t=7: | 16_2 | | 27_4 | 35_1 |
  - t=8: | | | 16_2 | 27_4 |
  - t=9: | | | | | 16_2 |
37. Você deve incluir no enunciado da questão anterior a existência de uma pista de ultrapassagem. Agora, veículos mais rápidos podem mover-se para a pista de ultrapassagem ao alcançarem veículos mais lentos, desde que não haja ninguém ocupando aquele trecho de pista. Eles devem retornar à pista original assim que tiverem completado a ultrapassagem, retomando a velocidade original. Você deve escrever unicamente os procedimentos modificados ou novos que levam em conta este novo fato.

Exemplo da nova saída para a entrada original:

- **Representação gráfica:**

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>35₁</td>
<td>35₁</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>35₁</td>
<td>35₁</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>27₄</td>
<td>35₁</td>
<td>35₁</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>27₄</td>
<td>35₁</td>
<td>35₁</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>35₁</td>
<td>35₁</td>
<td>35₁</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16₂</td>
<td></td>
<td>35₁</td>
<td>35₁</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16₂</td>
<td></td>
<td>35₁</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Saída:**
  - t=6: 27
  - t=8: 35
  - t=10: 16

38. Mateus, um engenheiro novato, está desenvolvendo uma notação posicional original para representação de números inteiros. Ele chamou esta notação de UMC (Um método curioso). A notação UMC usa os mesmos dígitos da notação decimal, isto
é, de 0 a 9. Para converter um número \( A \) da notação UMC para a notação decimal deve-se adicionar \( K \) termos, onde \( K \) é o número de dígitos de \( A \) (na notação UMC). O valor do \( i \)-ésimo termo correspondente ao \( i \)-ésimo dígito \( a_i \), contando da direita para a esquerda é \( a_i \times i! \).

Por exemplo, \( 719_{UMC} \) é equivalente a \( 53_{10} \), pois \( 7 \times 3! + 1 \times 2! + 9 \times 1! = 53 \).

Mateus está começando seus estudos em teoria dos números e provavelmente não sabe quais as propriedades que um sistema de numeração deve ter, mas neste momento ele está interessado em converter os números da notação UCM para a notação decimal. Você pode ajudá-lo? Caso possa, faça um programa em Pascal conforme instruções abaixo.

**Entrada:** cada caso de teste é fornecido em uma linha simples que contém um número não vazio de no máximo 5 dígitos, representando um número em notação UMC. Este número não contém zeros a esquerda. O último teste é seguido por uma linha contendo um zero.

**Saída:** para cada caso de teste imprimir uma linha simples contendo a representação em notação decimal do correspondente número em UMC seguido do cálculo feito para a conversão.

**O programa:** seu programa deve, para cada número da entrada, convertê-lo em um vetor de inteiros, sendo que cada dígito do número é um elemento do vetor, e fazer os cálculos usando este vetor.

Exemplos de entrada e saída:

<table>
<thead>
<tr>
<th>ENTRADA</th>
<th>SAÍDA</th>
</tr>
</thead>
<tbody>
<tr>
<td>719</td>
<td>( 53 = 7 \times 3! + 1 \times 2! + 9 \times 1! )</td>
</tr>
<tr>
<td>1</td>
<td>( 1 = 1 \times 1! )</td>
</tr>
<tr>
<td>15</td>
<td>( 7 = 1 \times 2! + 5 \times 1! )</td>
</tr>
<tr>
<td>110</td>
<td>( 8 = 1 \times 3! + 1 \times 2! + 0 \times 1! )</td>
</tr>
<tr>
<td>102</td>
<td>( 8 = 1 \times 3! + 0 \times 2! + 2 \times 1! )</td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

39. Sabemos que nos compiladores mais recentes, nos quais existe o tipo string, podemos realizar de maneira simples operações com palavras. Imagine, no entanto, que estamos usando um compilador Pascal para o qual não existe este tipo. Neste caso o programador deve implementar por sua própria conta os procedimentos com palavras. Neste exercício iremos considerar a seguinte declaração alternativa para o tipo string:

```pascal
type palavra = array[1..MaxTam] of char;
```

FAça uma função em Pascal que receba como parâmetros duas variáveis do tipo MeuString e retorne -1 se a primeira palavra for lexicograficamente menor que a segunda, 0 se forem iguais, e +1 no caso que resta.

40. Faça um programa em Pascal que leia um certo número indefinido de vetores e que imprima o vetor original (O) e um vetor gerado (G) após um processo de compactação que consiste na eliminação de todos os elementos repetidos em cada vetor. Considere
que a entrada de dados é feita em um vetor por linha, sendo que o primeiro elemento da linha é o tamanho de cada vetor e os elementos restantes da linha são os elementos do vetor. Quando o tamanho for zero significa que terminou a entrada de dados. Por exemplo, considere a seguinte entrada:

5 2 4 7 -1 2  
3 1 1 1 7 3 4 5 3 4 5 1 0

Deverá produzir como saída o seguinte:

O: 2 4 7 -1 2  
G: 2 4 7 -1  
O: 1 1 1  
G: 1  
O: 3 4 5 3 4 5 1  
G: 3 4 5 1

41. Considere uma sequência de dígitos binários como:

011100011

Uma maneira de criptografar essa sequência de bits é adicionar à cada dígito a soma dos seus dígitos adjacentes. Por exemplo, a sequência acima se tornaria:

123210122

Se P é a sequência original e Q é a sequência criptografada, então Q[i] = P[i − 1] + P[i] + P[i + 1] para todas as posições i da sequência. Considerando uma sequência de tamanho n e seus índices variando de 0 a n − 1, os dígitos P[−1] e P[n] não fazem parte da sequência original e são tratados como zeros na operação de codificação.

Assumindo P[0] = 0 temos:


Agora repetindo o mesmo processo para P[0] = 1 temos:

Note que este algoritmo pode gerar ou decodificar uma sequência criptografada em até duas possíveis sequências originais, uma iniciando com 0 e outra iniciando com 1.

Faça um procedimento em *Pascal* que receba como parâmetros um vetor de números inteiros contendo a sequência criptografada e a decodifica em dois outros vetores de números inteiros. Caso uma das decodificações não seja possível, como no caso do exemplo para $P[0] = 1$, o vetor correspondente deve ser preenchido com -1 na posição inicial.

Outros exemplos:

- $123210122 = 011100111, -1$
- $11 = 01, 10$
- $22111 = -1, 11001$
- $123210120 = -1, -1$
- $3 = -1, -1$
- $122211112222211111111112221111 = 01101001101101001101001001001101001, 10110010110110010110010010110010$

42. Faça um programa em *Pascal* que leia um texto e imprima uma distribuição de frequências para palavras do texto (quantas palavras de uma letra, quantas de duas letras, etc.).

43. Faça um programa em *Pascal* que leia do teclado o gabarito de uma prova de 20 questões de múltipla escolha, onde as respostas são inteiros de 1 a 5. Em seguida, o programa deve ler o número de alunos que prestaram a prova e, para cada aluno, a sua matrícula (um inteiro) e as respectivas respostas. O programa deve calcular e escrever:

- a relação de alunos ordenados pela nota, supondo que cada questão vale 5 pontos;
- para cada questão: quantos alunos acertaram a questão.
9.2 Matrizes

Assim como os vetores, as matrizes são arrays. Os vetores são estruturas unidimensionais enquanto que as matrizes são bidimensionais. Isto é, o acesso às posições de memória de um vetor são feitas com base em duas informações: o nome da variável e o deslocamento. Para se acessar os elementos de uma matriz, precisa-se do nome da variável, do deslocamento horizontal e do deslocamento vertical. Os elementos de uma matriz também são todos do mesmo tipo (estrutura uniforme).

9.2.1 Matrizes em Pascal

Para se declarar uma matriz de 200 posições inteiras, sendo 20 na vertical e 10 na horizontal, a linguagem Pascal usa a seguinte sintaxe (lembre-se que em outras linguagens a sintaxe pode ser diferente):

```pascal
var m: array[1..20,1..10] of integer;
```

A construção “1..20,1..10” indica que existem 20 posições na horizontal (linhas) e 10 na vertical (colunas). O “of integer” indica que cada posição é para se guardar um número inteiro, isto é 2 bytes (dependendo da implementação).

Todas as restrições e variantes que usamos para os vetores valem também para as matrizes. Isto é, as declarações seguintes também são válidas:

```pascal
var m: array[0..19,0..9] of integer;
var m: array[21..40,−19..−10] of integer;
var m: array[−19..0,51..60] of integer;
const maxLin=20, maxCol=10;
var m: array[1..maxLin,1..maxCol] of integer;
```

Agora, para escrever um valor qualquer, digamos 12345, na linha 15 e coluna 7 da matriz m, em Pascal, se usa um dos dois comandos seguintes:

```pascal
m[15,7]:= 12345;
read(m[15,7]); (* e se digita 12345 no teclado *)
```

Por exemplo, a matriz abaixo tem 5 linhas e 4 colunas e pode ser visualizada do modo padrão:

```
1 2 3 4
1 4 6 2 1
2 9 0 0 2
3 8 7 3 9
4 1 2 3 4
5 0 1 0 1
```
A declaração do tipo matriz em *Pascal* é assim:

```pascal
type matriz = array [1..5,1..4] of integer;
```

No exemplo acima, a primeira linha é constituída pelos elementos seguintes: 4, 6, 2 e 1. A terceira coluna pelos elementos 2, 0, 3, 3 e 0. Podemos destacar a título de exemplo alguns elementos da matriz. Consideremos uma variável *m* do tipo matriz. Então: \(m[3, 2] = 7\), \(m[2, 3] = 0\), \(m[3, 4] = 9\), e assim por diante.

Notem que, isoladamente, cada linha ou coluna completa pode ser imaginada como sendo um vetor. De fato, uma outra maneira de se ver uma matriz é como sendo um vetor de vetores! Confira a declaração seguinte:

```pascal

type vetor = array [1..4] of integer;
matriz = array [1..5] of vetor;

var m: matriz;
```

Em *Pascal*, é correto dizer que, para o exemplo acima: \(m[3][2] = 7\), \(m[2][3] = 0\), \(m[3][4] = 9\), e assim por diante. Nós usaremos a construção apresentada inicialmente.

### 9.2.2 Exemplos elementares

Do mesmo modo como fizemos para os vetores, vamos iniciar o estudo das matrizes apresentando problemas simples.

**Lendo e imprimindo matrizes**

A leitura dos elementos de uma matriz é bastante parecida com a leitura de vetores. Imaginando que cada linha da matriz é um vetor, basta fixar uma linha e ler todos os elementos das colunas. Agora é só repetir o procedimento para todas as linhas. Isto leva naturalmente a um código com um laço duplo: um para variar as linhas o outro para as colunas. A figura 9.38 permite visualizar o código para a leitura de uma matriz.

```pascal
program ler_matriz;
var w: array [0..50,1..10] of real;
i, j: integer;

begin
  for i:= 0 to 50 do
    for j:= 1 to 10 do
      read (w[i,j]);
end.
```

Figura 9.38: Lendo uma matriz.

Da forma como foi construído, este código exige que se digite os elementos da primeira linha, depois os da segunda e assim por diante. No laço mais interno, controlado pelo *j*, observem que o *i* é fixo, apenas o *j* varia, desta forma, a leitura das
colunas de uma linha fixa é idêntico a ler um vetor. O laço mais externo, controlado pelo i, faz variar todas as linhas.

Da mesma forma como apresentamos os vetores, vamos mostrar os códigos em termos de procedimentos e funções. Para isto vamos precisar considerar as seguintes definições:

```pascal
const maxLin = 50; maxCol = 40;
type matriz = array [0..maxLin, 1..maxCol] of integer;
```

Também vamos considerar que apenas uma parte da matriz será usada, logo precisaremos sempre fazer a leitura das dimensões de uma matriz. Neste sentido, a figura 9.39 apresenta um procedimento que lê uma matriz de dimensões \( n \times m \).

```pascal
procedure ler matriz (var w: matriz; var n,m: integer);
var i,j: integer;
begin
  read (n); (* n deve estar no intervalo 1..maxLin *)
  read (m); (* m deve estar no intervalo 1..maxCol *)
  for i:= 1 to n do
    for j:= 1 to m do
      read (w[i,j]);
end;
```

Figura 9.39: Procedimento para ler uma matriz.

As mesmas observações sobre passagem de parâmetros que foram feitas sobre os vetores se aplicam aqui, isto é, pela sobrecarga, sempre passamos matrizes por referência. A figura 9.40 apresenta um procedimento para impressão de uma matriz usando passagem de parâmetros por referência, embora não fosse estritamente necessário. O procedimento foi construído para imprimir a matriz linha por linha.

```pascal
procedure imprimir matriz (var w: matriz; n,m: integer);
var i,j: integer;
begin
  for i:= 1 to n do
    begin
      for j:= 1 to m-1 do
        write (w[i,j], ’ ’);
      writeln (w[i,m]); (* mudar de linha a cada fim de coluna, sem o branco no final *)
    end;
end;
```


Tomando como exemplo a matriz do início desta seção, se forem digitados todos os
valores, linha por linha, para cada uma do início até o final de cada coluna, teríamos em memória algo como ilustrado na figura seguinte:

A título de exemplo, poderíamos construir um procedimento que imprime a matriz em sua forma transposta, isto é, com as linhas e colunas invertidas. Isto é apresentado na figura [9.41]. Basta inverter i e j no comando de impressão! Observem que a matriz não mudou na memória, somente a impressão é diferente.

```pascal
procedure imprimir_transposta (var w: matriz; n,m: integer);
var i,j: integer;
begin
  for i:= 1 to m do
    begin
      for j:= 1 to n-1 do
        write (w[j,i], ' ');
      writeln (w[j,n]);
    end;
end;
```

Figura 9.41: Procedimento para imprimir a transposta de uma matriz.

Outros procedimentos interessantes são os de impressão de uma certa linha (figura [9.42]) ou de uma certa coluna (figura [9.43]).

```pascal
procedure imprimir_uma_linha (var w: matriz; n,m: integer; K: integer);
(* imprime a linha K da matriz *)
var j: integer;
begin
  for j:= 1 to m do
    begin
      write (w[K,j], ' ');
      (* K fixo na primeira posição *)
      writeln;
    end;
end;
```


Considerando o exemplo acima e fazendo $K = 2$ teríamos a seguinte saída:

9 0 0 2
CAPÍTULO 9. ESTRUTURAS DE DADOS

procedure imprimir_uma_coluna(var w: matriz; n,m: integer; K: integer);
(* imprime a coluna K da matriz *)
var i: integer;
begin
  for i:= 1 to n do 
    writeln (w[i,K]); (* K fixo na segunda posição *)
end;

Figura 9.43: Procedimento para imprimir uma única coluna da matriz.

Considerando o exemplo acima e fazendo K = 2 teríamos a seguinte saída:

6
0
7
2
1

Outro exemplo interessante seria imprimir somente os elementos da matriz que são pares, conforme mostrado na figura 9.44.

procedure imprimir_os_pares(var w: matriz; n,m: integer);
var i,j : integer;
begin
  for i:= 1 to n do 
    begin
      for j:= 1 to m do
        if eh_par (w[i,j]) then
          write (w[i,j], ' ');
        end;
      writeln;
    end;
end;

Figura 9.44: Procedimento para imprimir os elementos pares matriz.

Considerando novamente a nossa matriz exemplo, teríamos a seguinte saída:

4 6 2 0 0 2 8 2 4 0 0

Para finalizarmos esta seção inicial, apresentamos na figura 9.45 um código que imprime os elementos cujos índices das linhas e das colunas são pares. Considerando novamente a nossa matriz exemplo, teríamos a seguinte saída:

0 2
2 4
9.2. **MATRIZES**

```pascal
procedure imprimir_as_linhas_e_colunas_pares(var w: matriz; n,m: integer);
var i, j : integer;
begin
  for i:= 1 to n do
    begin
      for j:= 1 to m do
        if eh_par (i) and eh_par (j) then
          write (w[i,j], ' '); write ('');
      writeln;
    end;
end;
```

Figura 9.45: Procedimento para imprimir os elementos cujos índices são pares.

**Encontrando o menor elemento de uma matriz**

Vamos retornar ao velho e conhecido problema de se determinar qual é o menor elemento de um conjunto de números em memória, considerando que, desta vez, eles estarão armazenados em uma matriz. A figura [9.46] contém o código que faz isto. Note a semelhança com os programas das seções anteriores. A técnica é a mesma: o primeiro elemento é considerado o menor de todos e depois a matriz tem que ser toda percorrida (todas as linhas e todas as colunas) para ver se tem outro elemento ainda menor.

```pascal
function acha_menor_matriz (var w: matriz; n,m: integer): integer;
var i, j: integer;
  menor: integer;
begin
  menor:= w[1,1];
  for i:= 1 to n do
    for j:= 1 to m do
      if w[i,j] < menor then
        menor:= w[i,j];
  acha_menor_matriz:= menor;
end;
```

Figura 9.46: Encontrando o menor elemento de uma matriz.

**Soma de matrizes**

Nesta seção vamos implementar o algoritmo que soma duas matrizes. Para isto precisamos antes entender como funciona o processo.

Sejam $v$ e $w$ duas matrizes. Para somá-las, é preciso que elas tenham o mesmo tamanho. Isto posto, o algoritmo cria uma nova matriz $v + w$ onde cada elemento $i, j$ da nova matriz é a soma dos respectivos elementos $v[i, j]$ e $w[i, j]$. O esquema é muito parecido com a soma de vetores já estudada, apenas trata-se também da segunda dimensão.
Destina forma, o algoritmo que soma os dois vetores deverá, para cada par \( i, j \) fixo, somar os respectivos elementos em \( v \) e \( w \) e guardar em \( v + w \). Variando \( i \) de 1 até o número de linhas e \( j \) de 1 até o número de colunas resolve o problema. O programa que implementa esta ideia é apresentado na figura 9.47.

```pascal
procedure somar_matrizes (var v, w, soma_v_w: matriz; n,m: integer);
var i,j: integer;
begin
  (* n e m sao o numero de linhas e colunas, respectivamente *)
  for i:= 1 to n do
    for j:= 1 to m do
      soma_v_w[i,j]:= v[i,j] + w[i,j];
end;
```

Figura 9.47: Somando duas matrizes.

### Multiplicação de matrizes

Agora vamos resolver o problema da multiplicação de matrizes. Inicialmente vamos recordar o embasamento teórico.

Do ponto de vista matemático cada elemento da matriz resultado da multiplicação de duas matrizes pode ser encarado como sendo o produto escalar de dois vetores formados por uma linha da primeira matriz e por uma coluna da segunda.

Vamos considerar duas matrizes \( A_{n \times m} \) e \( B_{m \times p} \). A multiplicação só pode ocorrer se o número de colunas da matriz \( A \) for igual ao número de linhas da matriz \( B \). Isto por causa do produto escalar de vetores, que exige que os vetores tenham o mesmo tamanho. O resultado é uma matriz \( n \times p \). O produto escalar de vetores, conforme já estudado, é o resultado da seguinte somatória:

\[
\sum_{k=1}^{m} V[k] \times W[k].
\]

Vamos considerar duas matrizes \( A \) e \( B \), fixando uma linha \( I \) na matriz \( A \) e uma coluna \( J \) na matriz \( B \). Conforme já mencionado, fixar linhas ou colunas em matrizes é o mesmo que trabalhar com vetores. Então, neste caso, o produto escalar da linha \( I \) da matriz \( A \) pela coluna \( J \) da matriz \( B \) é dado pela seguinte somatória:

\[
\sum_{k=1}^{m} A[I, k] \times B[k, J].
\]

O programa que realiza esta operação é uma adaptação simples do código exibido na figura 9.14 que para fins didáticos é mostrado na figura 9.48.
9.2. MATRIZES

begin (* considerando I e J fixos *)
soma:= 0;
for i:= 1 to m do
    soma:= soma + A[i,k] * B[k,j];
prod_escalar:= soma;
end;

Figura 9.48: Produto escalar de uma linha da matriz por uma coluna da outra.

Vejamos isto de forma ilustrada considerando as duas matrizes seguintes:

\[
\begin{align*}
A: & \begin{bmatrix}
4 & 6 & 2 & 1 \\
9 & 0 & 0 & 2 \\
8 & 7 & 3 & 9 \\
1 & 2 & 3 & 4 \\
0 & 1 & 0 & 1
\end{bmatrix} & \quad \begin{bmatrix}
2 & 3 & 4 \\
5 & 0 & 0 \\
0 & 7 & 7 \\
1 & 0 & 9
\end{bmatrix} = B.
\end{align*}
\]

O produto escalar da linha (terceira) pela coluna (segunda) em destaque é o resultado da seguinte operação: \(8 \times 3 + 7 \times 0 + 3 \times 7 + 9 \times 0 = 45\). Este resultado é o valor da linha 3 coluna 2 da matriz produto, isto é, é o elemento \(AB[3,2]\).

Para se obter o resultado completo do produto de \(A\) por \(B\), basta variar \(I\) nas linhas de \(A\) e \(J\) nas colunas de \(B\). Isto é apresentado na figura 9.49.

figura 9.49: Multiplicação de duas matrizes.

9.2.3 Procurando elementos em matrizes

Nesta seção vamos apresentar alguns problemas de busca de elementos em matrizes.

Busca em uma matriz

O problema de busca em matrizes é similar ao caso dos vetores. O procedimento agora é quadrático, pois no pior caso a matriz inteira deve ser percorrida (caso em que o elemento não está na matriz). A figura 9.50 contém o código para este problema.
Um problema interessante é saber se uma matriz contém elementos repetidos. Basta varrer a matriz e, para cada elemento, saber se ele existe na matriz em posição diferente. Isto exige um aninhamento de quatro laços!

Um laço duplo é necessário para se percorrer a matriz por completo, e depois um outro laço duplo para cada elemento para saber se ele se repete. Isto resulta em um algoritmo de ordem de $n^4$ para uma matriz quadrada de ordem $n$, para o pior caso. Para completar o grau de dificuldade, queremos parar o processamento tão logo um elemento repetido seja encontrado. O código final está ilustrado na figura 9.51.

### 9.2.4 Inserindo uma coluna em uma matriz

Vamos considerar o problema de receber uma matriz de dimensões $n \times m$ e um vetor de $n$ elementos e inserir o vetor como uma coluna adicional na matriz, que ficará com dimensões $n \times m + 1$.

Por exemplo, consideremos a nossa matriz exemplo e o seguinte vetor:

```
 1 2 3 4 5
1 4 6 2 1
2 9 0 0 2
3 8 7 3 9
4 1 2 3 4
5 0 1 0 1
```

Inicialmente vamos inserir o vetor após a última coluna, isto é, o vetor será a última coluna da nova matriz, tal como na figura seguinte, de ordem $5 \times 5$ (vetor inserido está em negrito na figura):
9.2. MATRIZES

function tem_repetidos (var w: matriz; n,m: integer): boolean;
var i, j, p, q: integer;
  repetiu: boolean;
begin
  repetiu:= false;
  i:= 1;
  while (i <= n) and not repetiu do
  begin
    j:= 1;
    while (j <= m) and not repetiu do
    begin
      p:= 1;
      while (p <= n) and not repetiu do
      begin
        q:= 1;
        while (q <= m) and not repetiu do
        begin
          if (w[p,q] = w[i,j]) and ((p <> i) or (q <> j)) then
            repetiu:= true;
        q:= q + 1;
        end;
        p:= p + 1;
      end;
      j:= j + 1;
    end;
  i:= i + 1;
  end;
  tem_repetidos:= repetiu;
end;

Figura 9.51: Verifica se uma matriz tem elementos repetidos.

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
<td>6</td>
<td>2</td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td>9</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>7</td>
<td>3</td>
<td>9</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>3</td>
</tr>
</tbody>
</table>

O procedimento mostrado na figura 9.52 faz esta operação.

Um problema mais difícil seria se quiséssemos inserir o vetor em alguma coluna que não fosse a última da matriz. O exemplo seguinte mostra nossa matriz de exemplo com o vetor inserido na coluna 2.
**PROCEDURE** inserir_coluna_no_fim (var w: matriz; v: vetor_i; var n,m: integer);  
(* recebe uma matriz e um vetor e insere o vetor como ultima coluna da matriz *)  
var i: integer;  
begin  
for i:= 1 to n do  
w[i,m+1] := v[i]; (* m+1 eh fixo, queremos sempre a ultima coluna *)  
m:= m + 1; (* altera o numero de colunas *)  
end;  

Figura 9.52: Insere um vetor como ultima coluna de uma matriz.

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
<td>7</td>
<td>6</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>9</td>
<td>6</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>5</td>
<td>7</td>
<td>3</td>
<td>9</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>4</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Neste caso, tal como no caso dos vetores, teríamos que abrir espaço na matriz antes de inserir o vetor. Esta é uma operação bastante custosa, pois temos que mover várias colunas para frente, cada uma delas move n elementos para frente. O algoritmo apresentado na figura 9.53 mostra estes dois passos, um que abre espaço o outro que insere o vetor no espaço aberto.

**PROCEDURE** inser_coluna_k (var w: matriz; var v: vetor_i; var n,m: integer; K: integer);  
(* recebe uma matriz e um vetor e insere o vetor na coluna K da matriz *)  
var i,j: integer;  
begin  
(* primeiro abre espaço *)  
for j:= m downto K do (* para cada coluna, iniciando na ultima *)  
  for i:= 1 to n do (* move elementos uma coluna para frente *)  
    w[i,j+1]:= w[i,j];  
(* depois insere na coluna K *)  
for i:= 1 to n do  
  w[i,K] := v[i]; (* K eh fixo, queremos sempre a K-esima coluna *)  
m:= m + 1; (* altera o numero de colunas *)  
end;  

Figura 9.53: Insere um vetor como K-ésima coluna de uma matriz.

Para inserir linhas em uma matriz o procedimento é análogo.
9.2.5 Aplicações de matrizes em imagens

Nosso objetivo nesta seção é mostrar como podemos fazer modificações em imagens digitais no formato PGM, que são essencialmente matrizes.

Matrizes que representam imagens

Um dos formatos reconhecidos pelos computadores atuais para imagens é o padrão PGM, que é na verdade um dos tipos do formato Netpbm[12]. As imagens estão armazenadas em um arquivo ASCII que pode ser editado normalmente em qualquer editor de textos, mas quando abertas em aplicativos apropriados a imagem pode ser visualizada, como por exemplo, uma fotografia.

Aplicativos de manipulação de imagens podem carregar as informações do arquivo em memória e fazer diversos tipos de conversão, como por exemplo clareamento de imagens, zoom, etc. Nesta seção mostraremos como isso é possível, mas antes vamos entender este formato.

O formato PGM

Um arquivo em formato PGM tem o seguinte conteúdo:

- a primeira linha contém um identificador, que é “P2” para formatos PGM. Outros formatos da família Netpbm têm outras identificações;
- a segunda linha contém a largura e a altura de uma matriz, isto é, o número de colunas e de linhas;
- a terceira linha contém um número que indica o maior valor da matriz da parte que segue;
- o restante do arquivo contém uma matriz de elementos (bytes) que representam um pixel da imagem em tons de cinza.

Os valores da matriz são números entre 0 e 255, sendo que 0 representa o preto e 255 representa o branco. Os valores intermediários representam tons de cinza, quanto mais perto do 0 é um cinza cada vez mais escuro, quanto mais perto do 255 é um cinza bem claro.

Para cada imagem, não é obrigatório que o branco seja representado por 255. Este é o valor máximo que o branco pode assumir. Para uma imagem particular, o branco é o valor contido na terceira linha do arquivo.

Na área de imagens estes números são chamados de pixels, que são na verdade um ponto da imagem quando exibidos em um aplicativo.

A figura 9.54 mostra um exemplo de um arquivo que é uma imagem em PGM: a primeira linha tem “P2”; a segunda linha contém a dimensão da matriz (10 × 11, observe que por definição o número de colunas vem antes); a terceira linha contém o maior elemento (40) da matriz que constitui o restante do arquivo.

Figura 9.54: Exemplo de imagem no formato PGM.

Nesta seção vamos considerar as seguintes declarações:

```pascal
const MAXTAM = 1000;

type imagem = array [1..MAXTAM, 1..MAXTAM] of integer;

var
  l, (* número de linhas da imagem *)
  c, (* número de colunas da imagem *)
  max: integer; (* valor do pixel máximo *)
  O: imagem (* matriz que conterá a imagem *)
```

**Lendo arquivos PGM**

Para carregar os dados de um arquivo PGM, basta fazer a leitura do cabeçalho da imagem (as três primeiras linhas) e depois ler a matriz propriamente dita.

A procedure da figura 9.55 faz a leitura dos dados, consumindo a primeira linha, que contém P2. Isto serve para os aplicativos saberem que se trata de uma imagem em formato PGM. Esta informação é irrelevante neste texto, mas de todo modo esta informação tem que ser lida, embora não seja utilizada. Em seguida deve-se ler o número de colunas e linhas, o valor do pixel máximo e finalmente a matriz propriamente dita.

**Imprimindo arquivos PGM**

A procedure da figura 9.56 faz a impressão de uma imagem, para isso tem que imprimir uma linha contendo P2, para garantir o padrão do formato, depois o número de colunas e linhas, o valor do pixel máximo e finalmente imprimindo uma matriz.
9.2. MATRIZES

```pascal
procedure ler_pgm (var O: imagem; var l,c,max: integer);
var i,j: integer;
  s: string[2];
begin
  readln (s);
  if s = 'P2' then
    begin
      readln (c,1);
      readln (max);
      for i:= 1 to l do
        for j:= 1 to c do
          readln (O[i,j]);
    end
  else
    writeln ('Formato invalido');
end;
```

Figura 9.55: Leitura de uma imagem PGM.

```pascal
procedure imprimir_pgm (var O: imagem; l,c,max: integer);
var i,j: integer;
begin
  writeln ('P2');
  writeln (c,' ', l);
  writeln (max);
  for i:= 1 to l do
    begin
      for j:= 1 to c-1 do
        write (O[i,j], ' ');
      writeln (O[i,c]);
    end;
end;
```

Figura 9.56: Impressão de uma imagem PGM.

Clareando uma imagem

Um dos problemas mais simples que podemos resolver quando temos uma fotografia é o seu clareamento (ou escurecimento). Esta funcionalidade está disponível em aplicativos tais como photoshop ou gimp, ou mesmo em aplicativos de smartphones. A figura mostra o efeito que queremos, a imagem da direita é mais clara do que a da esquerda.

Como vimos, os valores dos pixels variam de 0 a 255, sendo que o zero representa o preto e o 255 representa o branco. Se queremos clarear uma imagem, basta somar um determinado valor fixo nos pixels. Ao contrário, se queremos escurecer uma imagem, basta subtrair um determinado valor fixo nos pixels.

O procedimento apresentado na figura considera um valor constante , que é o quanto queremos clarear a imagem. O procedimento é muito simples, ele simplesmente soma este valor em todos os pixels, mas ele leva em conta que um pixel
nunca pode exceder o valor do pixel máximo (aquele da terceira linha do cabeçalho da imagem), por isso existe um if que testa se após uma soma o valor ultrapassa este valor max, então o valor do pixel recebe este mesmo valor.

```
procedure clarear_pgm (var O: imagem; l,c,max,cte: integer);
var i,j: integer;
begin
  for i:= 1 to l do
    for j:= 1 to c do
      begin
        O[i,j]:= O[i,j] + cte;
        if O[i,j] > max then
          O[i,j]:= max;
      end;
end;
```

Figura 9.58: Procedure para clarear uma imagem PGM.

**Fazendo zoom em uma imagem**

O objetivo é ler uma imagem PGM e gerar uma outra com a metade do tamanho da original, produzindo assim um efeito de zoom. A figura 9.59 mostra o efeito que queremos, a imagem da direita é obtida a partir da imagem da da esquerda.

Existem várias técnicas para se fazer isso, a nossa será a de se dividir a matriz original em blocos de 4 células, tal como mostrado na figura 9.60 que mostra no lado esquerdo a matriz original e no lado direito um destaque para os blocos que queremos separar.

Agora é fácil entender o que queremos fazer: Cada bloco 2 × 2 dará origem a um único valor, que será a média dos 4 valores do bloco. Desta maneira obteremos uma matriz 2 × 2. A motivação de se calcular a média é obter um tom de cinza que seja o mais próximo possível dos tons de cinza destes quatro elementos.

Resumindo, vamos então obter uma segunda matriz a partir da primeira onde cada elemento é a média do elemento do canto superior esquerdo de um bloco mais os seus três vizinhos com relação à matriz original. Para exemplificar o que queremos, vejamos
9.2. MATRIZES

Figura 9.59: Reduzindo uma imagem.

\[
\begin{array}{cccc}
4 & 6 & 2 & 1 \\
9 & 0 & 0 & 2 \\
8 & 7 & 3 & 9 \\
1 & 2 & 3 & 4 \\
\end{array}
\]

\[
\begin{array}{cccc}
4 & 6 & 2 & 1 \\
9 & 0 & 0 & 2 \\
8 & 7 & 3 & 9 \\
1 & 2 & 3 & 4 \\
\end{array}
\]

Figura 9.60: Visualizando uma matriz $4 \times 4$ em quatro blocos $2 \times 2$.

A seguinte ilustração da submatriz $2 \times 2$ do canto superior esquerdo da submatriz destacada no lado direito da figura:

\[
\begin{array}{cc}
4 & 6 \\
9 & 0 \\
\end{array}
\]

A média destes elementos é $\frac{4+6+9+0}{4} = 4.75$.

Já para a submatriz do canto superior direito:

\[
\begin{array}{cc}
2 & 1 \\
0 & 2 \\
\end{array}
\]

A média destes elementos é $\frac{2+1+0+2}{4} = 1.25$. Os mesmos cálculos são feitos para as outras duas submatrizes.

Com isso, a matriz $2 \times 2$ resultante é esta aqui:

\[
\begin{array}{cc}
4.75 & 1.25 \\
4.50 & 4.75 \\
\end{array}
\]

Evidentemente queremos o valor inteiro destes números, o que daria a seguinte matriz:

\[
\begin{array}{cc}
4 & 1 \\
4 & 4 \\
\end{array}
\]
O procedimento que realiza este cálculo é ilustrado na figura 9.61, considerando que as dimensões \( n \) e \( m \) da matriz são sempre pares. Este procedimento percorre a matriz destino (a que está sendo gerada) e para cada par \( i, j \) calcula o canto superior esquerdo da respectiva submatriz \( 2 \times 2 \) da matriz original. A partir deste elemento é fácil saber quem são os seus três vizinhos para fazer o cálculo da média. Isto é feito como mostrado na figura 9.62.

```plaintext
procedure zoom_pgm (var O: imagem; lO, cO: integer;
    var D: imagem; var lD, cD, maxD: integer);
var i, j: integer;
begin
    lD := lO div 2;
    cD := cO div 2;
    for i := 1 to lD do
        for j := 1 to cD do
            D[i, j] := media_4_vizinhos (O, i, j);
    maxD := maior_valor (D, lD, cD);
end;
```

Figura 9.61: Procedure para fazer zoom em uma imagem PGM.

A figura 9.62 mostra os cálculos que são feitos para a obtenção da média dos 4 elementos de um bloco. Observem que para um par \((i, j)\) da matriz destino, é possível calcular o canto superior esquerdo do bloco de interesse, que é exatamente \((2 \times i - 1, 2 \times j - 1)\).

```plaintext
function media_4_vizinhos (var O: imagem; i, j: integer): integer;
var x, y: integer;
begin
    x := 2*i - 1;
    y := 2*j - 1;
    media_4_vizinhos := (O[x, y] + O[x+1, y] + O[x, y+1] + O[x+1, y+1]) div 4;
end;
```

Figura 9.62: Função que calcula média dos quatro vizinhos de um pixel.

Se esta matriz resultante for impressa, teremos uma imagem reduzida da matriz original. Porém, antes de imprimir, temos que calcular o novo número de linhas e colunas (que é metade do original) e também o valor do novo pixel de maior valor.

Notem que o procedimento para se achar o maior pode ser facilmente adaptado do programa da figura 9.46, modificado para se retornar um valor do tipo integer e não do tipo real, o que resulta na função da figura 9.63.

Detectando bordas em arquivos PGM

Pode parecer natural para usuários de redes sociais, em especial aqueles que nasceram no século 21, observar que as fotografias mostram retângulos nos rostos das pessoas. Mas se você acompanhou o nosso estudo até agora como é que é possível identificar um rosto em uma matriz que só tem números?
Os pesquisadores da área de Processamento de Imagens desenvolveram diversas técnicas para isso e nesta seção vamos abordar uma técnica que já está ultrapassada, segundo estes mesmos pesquisadores, a partir do descobrimento das pesquisas em aprendizado de máquina (Deep Learning), que surgiram por volta do ano 2016. Portanto, no texto que segue, vamos apresentar uma das técnicas antigas, mas que é apropriada para estudantes do primeiro período de um curso de ciência da computação, pois basicamente se trata de manipular matrizes com um efeito, digamos, lúdico, pois se pode ver efetivamente o resultado dos cálculos feitos.

Muito bem, antigamente o processo de detecção de imagens seguia várias etapas, todas elas de custo computacional relativamente alto. A primeira destas etapas era a detecção de bordas, para posteriormente se aplicar diversas outras técnicas com o objetivo de extrair o maior número de informações possíveis desta matriz de números.

Detectar bordas significa identificar os contornos principais de uma imagem, ou seja, os pixels que caracterizem mudanças radicais das cores, ou dos tons de cinza. As imagens da figura 9.64 ilustram esta problemática.

Existem várias técnicas para se detectar bordas, a que apresentaremos aqui tem a seguinte ideia: uma borda é um local na imagem no qual os pixels vizinhos tem grande variação no tom de cinza. Como exemplo introdutório vamos usar a seguinte matriz:

```haskell
function maior_valor (var O: imagem; l, c: integer): integer;
var i, j, m: integer;
begin
  m := O[1,1];
  for i:= 1 to l do
    for j:= 1 to c do
      if O[i,j] > m then
        m := O[i,j];
  maior_valor := m;
end;
```

Figura 9.63: Cálculo do valor do maior pixel.
É óbvio que existe um quadrado com valores iguais a 50 no, digamos, meio desta matriz. O objetivo é identificar este formato de quadrado, que é o que chamamos de borda. Como fazer isso?

A ideia tem início pela observação de que os elementos ao redor dos valores 50 são bem menores do que 50, tipicamente valores entre 0 e 6. Isto dá origem à proposta de algoritmo de detecção de bordas, a qual, uma vez compreendida, é de fácil implementação.

Consideremos um valor denominado gradiente, que é calculado para cada pixel (ou em outras palavras, um elemento da matriz). Este pixel deve ser comparado com seus vizinhos, considerando-se somente as direções horizontal e vertical (ignora-se as diagonais). Se o valor deste pixel, multiplicado por 4, que é o número de vizinhos, for maior do que a soma destes vizinhos, significa que existe uma grande diferença entre este pixel e seus vizinhos e ele é portanto uma borda.

Vejamos no exemplo acima, tomando o caso do pixel da posição (2,4), que vale 50. Seus vizinhos são, no sentido horário, 3, 50, 50, 1. Somando este valor o resultado é 104. O valor do próprio pixel, que é 50, vezes 4, é $4 \times 50 = 200$. Isto é, a diferença entre o pixel em questão e seus vizinhos é 200-104=96.


Logo, existe uma diferença significativa entre o pixel (2,4) quando comparada com o pixel (4,2), o que sugere que, dependendo do limiar que se estabelece, (2,4) é uma borda, enquanto que (4,2) não é.

O algoritmo apresentado na figura 9.65 implementa esta ideia baseada no valor de um limiar, que nesta procedure considera limiar como sendo uma constante estabelecida, digamos, 50, decide se o ponto é uma borda ou não. Se for borda, gera em uma nova matriz o valor zero (preto), senão gera o valor 255 (branco), criando assim uma imagem preto-e-branco na qual o branco identifica uma borda. O algoritmo faz as extremidades da matriz serem pretas (zero), para facilitar.

O interessante é que o algoritmo é muito simples quando pensado do ponto de vista de matrizes. Basta percorrer a matriz somando os vizinhos e comparando com o limiar. Se o limiar for ultrapassado, é borda, senão não é. Simples assim.

<table>
<thead>
<tr>
<th>0 2 1 3 1 2 3 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 50 50 50 50 4</td>
</tr>
<tr>
<td>1 2 2 50 0 50 4 2</td>
</tr>
<tr>
<td>1 1 3 50 6 50 3 1</td>
</tr>
<tr>
<td>1 1 2 50 50 50 2 4</td>
</tr>
<tr>
<td>3 4 5 3 2 1 0 2</td>
</tr>
<tr>
<td>1 4 5 2 3 5 0 1</td>
</tr>
</tbody>
</table>
procedure detectar_bordas_pgm (var O,D: imagem; l,c: integer; var max: integer);
var i,j,grad: integer;
begin
  (* extremidades recebem zero *)
  for i:= 1 to l do
    begin
      D[i,1]:= 0;
      D[i,c]:= 0;
    end;
  for i:= 1 to c do
    begin
      D[1,i]:= 0;
      D[l,i]:= 0;
    end;
  for i:= 2 to l-1 do
    for j:= 2 to c-1 do
      begin
        grad:= abs (O[i,j]*4 - (O[i-1,j] + O[i+1,j] + O[i,j-1] + O[i,j+1]));
        if grad > limiar then
          D[i,j]:= 255
        else
          D[i,j]:= 0;
      end;
  max:= 255;
end;

Figura 9.65: Procedure para detectar bordas de uma imagem PGM.
9.2.6 Exercícios

1. Dada uma matriz real $A$ com $m$ linhas e $n$ colunas e um vetor real $V$ com $n$ elementos, faça um programa em *Pascal* que determine o produto de $A$ por $V$.

2. Um vetor real $X$ com $n$ elementos é apresentado como resultado de um sistema de equações lineares $Ax = B$ cujos coeficientes são representados em uma matriz real $A(m \times n)$ e os lados direitos das equações em um vetor real $B$ de $m$ elementos. Faça um programa em *Pascal* que verifique se o vetor $X$ é realmente solução do sistema dado.

3. Dizemos que uma matriz inteira $A(n \times n)$ é uma matriz de permutação se em cada linha e em cada coluna houver $n - 1$ elementos nulos e um único elemento igual a 1. Dada uma matriz inteira $A(n \times n)$, faça um programa em *Pascal* que verifique se $A$ é de permutação. Exemplos:

\[
\begin{bmatrix}
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
1 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 \\
\end{bmatrix}
\]

é de permutação, enquanto que esta outra não é:

\[
\begin{bmatrix}
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
1 & 0 & 0 & 0 \\
0 & 0 & 0 & 2 \\
\end{bmatrix}
\]

4. Faça um programa em *Pascal* que, dada uma matriz $A(n \times m)$ imprima o número de linhas e o número de colunas nulas da matriz. Exemplo: a matriz abaixo tem duas linhas e uma coluna nulas.

\[
\begin{bmatrix}
0 & 0 & 0 & 0 \\
1 & 0 & 2 & 2 \\
4 & 0 & 5 & 6 \\
0 & 0 & 0 & 0 \\
\end{bmatrix}
\]

5. Dizemos que uma matriz quadrada inteira é um quadrado mágico se a soma dos elementos de cada linha, a soma dos elementos de cada coluna e a soma dos elementos das diagonais principal e secundária são todos iguais. Exemplo:

\[
\begin{bmatrix}
8 & 0 & 7 \\
4 & 5 & 6 \\
3 & 10 & 2 \\
\end{bmatrix}
\]
é um quadrado mágico pois 

\[8 + 0 + 7 = 4 + 5 + 6 = 3 + 10 + 2 = 8 + 4 + 3 = 0 + 5 + 10 = 7 + 6 + 2 = 8 + 5 + 2 = 3 + 5 + 7 = 15.\]

Faça um programa em *Pascal* que:

(a) Dada uma matriz quadrada \(A(n \times m)\), verificar se \(A\) é um quadrado mágico.

(b) Informe quantas são e quais são as submatrizes não triviais (isto é, não pode ser a matriz constituída por umicamente um elemento, uma linha ou uma coluna) que definem quadrados mágicos. Por exemplo, a matriz do exemplo acima tem 4 submatrizes de tamanho 2 \(\times\) 2, duas submatrizes de tamanho 2 \(\times\) 3, etc, e uma única submatriz de dimensão 3 \(\times\) 3;

- Armazene de alguma maneira as informações necessárias sobre a localização precisa de cada uma das submatrizes não triviais que definem quadrados mágicos;
- Imprima a qualquer tempo algum quadrado mágico armazenado;
- Dado uma dimensão qualquer, digamos \(N\), imprima todas os quadrados mágicos de dimensão \(N\) contidos na matriz original.

6. Faça um programa em *Pascal* que implemente o quadrado “quase magico”. Um quadrado quase mágico é aquele em que as somas das linhas e a somas das colunas resultam em um mesmo valor, mas a soma dos elementos das diagonais não. O programa deve pedir a dimensão do quadrado a ser impresso, que deve ser um número ímpar entre 1 e 99.

7. Faça um programa em *Pascal* que resolva o seguinte problema: um jogo de palavras cruzadas pode ser representado por uma matriz \(A(n \times m)\) onde cada posição da matriz corresponde a um quadrado do jogo, sendo que 0 indica um quadrado em branco e -1 indica um quadrado preto. Colocar as numerações de início de palavras horizontais e/ou verticais nos quadrados correspondentes (substituindo os zeros), considerando que uma palavra deve ter pelo menos duas letras.

Exemplo: Dada a matriz:

\[
\begin{array}{cccccccc}
0 & -1 & 0 & -1 & -1 & 0 & -1 & 0 \\
0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 \\
0 & 0 & -1 & -1 & 0 & 0 & -1 & 0 \\
-1 & 0 & 0 & 0 & 0 & -1 & 0 & 0 \\
0 & 0 & -1 & 0 & 0 & 0 & -1 & -1 \\
\end{array}
\]

A saída deveria ser:

\[
\begin{array}{cccccccc}
1 & -1 & 2 & -1 & -1 & 3 & -1 & 4 \\
5 & 6 & 0 & 0 & -1 & 7 & 0 & 0 \\
8 & 0 & -1 & -1 & 9 & 0 & -1 & 0 \\
-1 & 10 & 0 & 11 & 0 & -1 & 12 & 0 \\
13 & 0 & -1 & 14 & 0 & 0 & -1 & -1 \\
\end{array}
\]
8. Uma matriz \( D(8 \times 8) \) pode representar a posição atual de um jogo de damas, sendo que 0 indica uma casa vazia, 1 indica uma casa ocupada por uma peça branca e -1 indica uma casa ocupada por uma peça preta. Faça um programa em *Pascal* que, supondo que as peças pretas estão se movendo no sentido crescente das linhas da matriz \( D \), determine as posições das peças pretas que:

- podem tomar peças brancas;
- podem mover-se sem tomar peças brancas;
- não podem se mover.

9. Faça um programa em *Pascal* que ajude o gerente de um banco de deseja atualizar as contas correntes dos clientes de uma agência bancária. É dado o cadastro de \( N \) clientes contendo para cada cliente o número de sua conta e seu saldo. O cadastro está ordenado pelo número da conta. Em seguida é dado o número de operações realizadas no dia, e, para cada operação, o número da conta, uma letra \( C \) ou \( D \) indicando se a operação é de crédito ou débito, e o valor da operação. Emitir o cadastro de clientes atualizado. Pode ser modelado como uma matriz \( N \times 2 \).

10. Reordenar a matriz do exercício anterior por ordem de saldo, do maior para o menor.

11. Os elementos \( M[i,j] \) de uma matriz \( M(n \times n) \) representam os custos de transporte da cidade \( i \) para a cidade \( j \). Faça um programa em *Pascal* que, dados \( n \) itinerários lidos do teclado, cada um com \( k \) cidades, calcular o custo total para cada itinerário. Exemplo:

\[
\begin{align*}
4 & \quad 1 & \quad 2 & \quad 3 \\
5 & \quad 2 & \quad 1 & \quad 400 \\
2 & \quad 1 & \quad 3 & \quad 8 \\
7 & \quad 1 & \quad 2 & \quad 5
\end{align*}
\]


12. Considere \( n \) cidades numeradas de 1 a \( n \) que estão interligadas por uma série de estradas de mão única. As ligações entre as cidades são representadas pelos elementos de uma matriz quadrada \( L(n \times n) \) cujos elementos \( L[i,j] \) assumem o valor 0 ou 1 conforme exista ou não estrada direta que saia da cidade \( i \) e chegue na cidade \( j \). Assim, os elementos da \( i \)-ésima linha indicam as estradas que saem da cidade \( i \) e os elementos da \( j \)-ésima coluna indicam as estradas que chegam à cidade \( j \). Por convenção, \( L[i,i] = 1 \). A figura abaixo ilustra um exemplo para \( n = 4 \).
9.2. **MATRIZES**

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>C</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>D</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Por exemplo, existe um caminho direto de A para B mas não de A para D. Faça um programa em *Pascal* que implemente soluções para os problemas abaixo.

(a) Dado *k*, determinar quantas estradas saem e quantas chegam à cidade *k*.
(b) A qual das cidades chega o maior número de estradas?
(c) Dado *k*, verificar se todas as ligações diretas entre a cidade *k* e outras são de mão dupla;
(d) Relacionar as cidades que possuem saídas diretas para a cidade *k*;
(e) Relacionar, se existirem:
   - As cidades isoladas, isto é, as que não têm ligação com nenhuma outra;
   - As cidades das quais não há saída, apesar de haver entrada;
   - As cidades das quais há saída sem haver entrada;
(f) Dada uma sequência de *m* inteiros cujos valores estão entre 1 e *n*, verificar se é possível realizar o roteiro correspondente. No exemplo dado, o roteiro representado pela sequência *(m = 5)* 3 4 3 2 1 é impossível;
(g) Dados *k* e *p*, determinar se é possível ir da cidade *k* até a cidade *p* pelas estradas existentes. Você consegue encontrar o menor caminho entre as duas cidades?
(h) Dado *k*, determinar se é possível, partindo de *k*, passar por todas as outras cidades uma única vez e retornar a *k*.

13. Uma matriz transposta $M^T$ é o resultado da troca de linhas por colunas em uma determinada a matriz $M$. Faça um programa em *Pascal* que leia duas matrizes $(A$ e $B$), e testa se $B = A + A^T$.

14. Faça procedimentos em *Pascal* que recebam três parâmetros: uma matriz e dois inteiros representando as dimensões da matriz. Retornar:
   (a) a transposta de matriz;
   (b) a soma das duas matrizes;
   (c) a multiplicação das duas matrizes.

15. Faça um programa em *Pascal* que leia duas matrizes $A$ e $B$ quaisquer e imprima a transposta de $B$ se a transposta de $A$ for igual a $B$.

16. Faça uma função em *Pascal* que receba como parâmetros: dois vetores de reais e dois inteiros representando as dimensões dos vetores e retorne o produto escalar dos dois vetores (real). Refazer a multiplicação de matrizes usando esta função.
17. Faça um programa em Pascal que, dadas N datas em uma matriz DATAS_{N×3}, onde a primeira coluna corresponde ao dia, a segunda ao mês e a terceira ao ano, coloque essas datas em ordem cronológica crescente. Por exemplo:

\[
DATAS = \begin{pmatrix}
5 & 1 & 1996 \\
25 & 6 & 1965 \\
16 & 3 & 1951 \\
15 & 1 & 1996 \\
5 & 11 & 1965 \\
\end{pmatrix}
\]  
\[
DATAS = \begin{pmatrix}
16 & 3 & 1951 \\
25 & 6 & 1965 \\
5 & 11 & 1965 \\
5 & 1 & 1996 \\
15 & 1 & 1996 \\
\end{pmatrix}
\]

18. Faça um programa em Pascal que verifique se a matriz A é simétrica, isto é, se \(A[i, j] = A[j, i], \forall i, j \leq M\). Faça uma função em Pascal que retorne 1 em caso afirmativo, 0 caso contrário.

19. Uma matriz B é dita inversa da matriz A quando \(A \times B = I\), onde I é a matriz identidade e \(\times\) é a operação de multiplicação de matrizes. A matriz identidade é a matriz quadrada onde os elementos da diagonal principal são 1 e os demais 0 (\(I[i, j] = 1\) se \(i = j\) e \(I[i, j] = 0\) se \(i \neq j\)). Faça um programa em Pascal que leia duas matrizes e teste se a segunda é a inversa da primeira.

20. Considere uma matriz M de tamanho \(N \times M\) utilizada para representar gotas de água (caráter G) em uma janela. A cada unidade de tempo T, as gotas descem uma posição na matriz, até que atinjam a base da janela e desapareçam. Considere que a chuva parou no momento em que seu programa iniciou.

Exemplo:

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-------</td>
<td>-------</td>
<td>-------</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-------</td>
<td>-------</td>
<td>-------</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-------</td>
<td>-------</td>
<td>-------</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Faça um programa em Pascal que:

(a) Leia as coordenadas iniciais das gotas de água na matriz. O canto superior esquerdo da matriz (desconsiderando as bordas) possui coordenada (1, 1). A coordenada (0, 0) indica o término da leitura. Coordenadas inválidas devem ser desconsideradas.

Exemplo de entrada para a matriz acima (em \(T = 0\)):
Note que a entrada (100, 98) deve ser descartada pois é inválida para a matriz do exemplo.

(b) Imprima, a cada unidade de tempo $T$, o conteúdo da matriz $M$, atualizando a posição das gotas $G$ até que não reste nenhuma gota na janela.

21. Modifique seu programa da questão anterior de modo que as gotas que estão inicialmente na primeira linha da janela desçam com o dobro da velocidade das outras gotas. Ou seja, as gotas que iniciam na primeira linha descem duas linhas na matriz a cada instante $T$. As gotas mais rápidas podem encontrar gotas mais lentas pelo caminho, neste caso a gota mais lenta desaparece ficando somente a mais rápida.

22. Modifique novamente o programa da questão anterior considerando que, desta vez, a cada unidade de tempo $T$, $NG$ novas gotas são inseridas na matriz. Além disso, as gotas descem na matriz até que atinjam a base da janela e desapareçam. Inicialmente não há gotas na janela, pois a chuva começa quando $T = 1$.

Exemplo:

<table>
<thead>
<tr>
<th>Passo $T=1$</th>
<th>Passo $T=2$</th>
<th>Passo $T=1000$</th>
</tr>
</thead>
<tbody>
<tr>
<td>G G</td>
<td>G</td>
<td></td>
</tr>
<tr>
<td>G</td>
<td>G G</td>
<td>G</td>
</tr>
<tr>
<td></td>
<td>G</td>
<td></td>
</tr>
<tr>
<td>G G</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>G G</td>
<td>G G</td>
</tr>
<tr>
<td></td>
<td>G</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
+-------------+-------------+----------------|
+-------------+-------------+----------------|
+-------------+-------------+----------------|

Faça um programa em Pascal que:

(a) Leia o número de linhas ($L$) e o número de colunas ($C$) da matriz $M$, a quantidade de novas gotas a serem criadas a cada iteração ($NG$), e o número de iterações ($TMAX$) do programa.

Exemplo de entrada para a matriz acima:

```
7 15 5 1000
```

(b) A cada unidade de tempo $T$, insira $NG$ novas gotas na matriz. A posição de uma nova gota é dada por um procedimento cujo protótipo é:
Procedure coordenada_nova_gota(L,C:integer; VAR x,y:integer);

Este procedimento recebe quatro parâmetros: os dois primeiros indicam o número de linhas e colunas da matriz $M (L, C)$. Os dois últimos retornam as coordenadas $(x, y)$ da nova gota na matriz.

(c) A cada unidade de tempo $T$, imprima o conteúdo da matriz $M$, atualizando a posição das gotas $G$ seguindo os seguintes critérios:

i. Quando uma gota cai sobre outra, forme-se uma gota “dupla”, ou seja, ela desce duas posições a cada instante $T$. Caso uma nova gota caia sobre uma gota “dupla”, surge uma gota “tripla”, que desce três posições a cada instante $T$, e assim por diante.

ii. As gotas mais rápidas podem encontrar gotas mais lentas pelo caminho, neste caso a velocidade delas é somada.

23. Considere o tipo PGM para imagens como definido na seção 9.2.5. Faça um programa em Pascal que leia da entrada padrão (teclado) duas imagens no formato PGM: imagem original ($imgO$) e a imagem do padrão ($imgP$).

Em seguida, o programa deve procurar se a imagem $imgP$ está contida na imagem $imgO$ e imprimir na tela as coordenadas (coluna, linha) do canto superior esquerdo de cada ocorrência da imagem $imgP$ encontrada na imagem $imgO$.

Observações:

- A imagem $imgP$ pode aparecer mais de uma vez na imagem $imgO$;
- Na imagem $imgP$, pontos com o valor $-1$ devem ser ignorados, isto é, representam pontos transparentes da imagem e não devem ser comparados com a imagem $imgO$.
- Estruture seu código. A solução parcial ou a indicação de chamadas a funções não implementadas serão consideradas.

Exemplo:

- Imagem original:

  P2
  11 10
  40
  40 5 5 5 5 5 5 5 5 40 0
  5 20 20 5 5 5 5 5 5 5 5
  5 5 20 5 5 5 0 0 0 0 0
  5 5 20 20 5 5 20 20 0 0 5
  5 5 5 5 5 5 0 20 0 0 0
  5 5 5 5 5 5 0 20 20 0 5
  5 5 5 5 11 11 11 0 0 0 0
  5 5 5 5 20 20 11 5 5 5 5
  5 5 5 5 11 20 11 5 5 5 0
  40 5 5 5 11 20 20 5 5 40 5
• Imagem do padrão:

P2
3 3
20
20 20 -1
-1 20 -1
-1 20 20

• Resultado do Programa:

2 2
7 4
5 8

24. Modifique o programa anterior de forma que, ao invés de imprimir as coordenadas, seja impressa uma nova imagem, que consiste de uma cópia da imagem original $imgO$ na qual as ocorrências da imagem $imgP$ estejam circunscritas por uma borda de um ponto de largura, com o valor máximo da imagem $imgO$ (3ª linha do arquivo PGM). Você não precisa se preocupar com possíveis sobreposições das bordas.

Exemplo da nova saída para a entrada original:

• Imagem resultante:

P2
11 10
40
40 40 40 40 40 5 5 5 5 40 0
40 20 20 5 40 5 5 5 5 5 5
40 5 20 5 40 40 40 40 40 40 0
40 5 20 20 40 40 20 20 0 40 5
40 40 40 40 40 0 20 0 40 0
5 5 5 5 5 40 0 20 20 40 5
5 5 5 40 40 40 40 40 40 40 0
5 5 5 40 40 20 20 11 40 5 5
5 5 5 40 11 20 11 40 5 5 0
40 5 5 40 11 20 20 40 5 40 5

25. Uma matriz é chamada de *esparsa* quando possui uma grande quantidade de elementos que valem zero. Por exemplo, a matriz de ordem $5 \times 4$ seguinte é esparsa, pois contém somente 4 elementos não nulos.
Obviamente, a representação computacional padrão para matrizes é ineficiente em termos de memória, pois gasta-se um espaço inútil para se representar muitos elementos nulos.

Nesta questão, vamos usar uma representação alternativa que vai permitir uma boa economia de memória.

A proposta é representar somente os elementos não nulos. Para isto usaremos três vetores, dois deles ($L$ e $C$) para guardar as coordenadas dos elementos não nulos e o terceiro ($D$) para guardar os valores dos elementos daquelas coordenadas. Também usaremos três variáveis para representar o número de linhas e colunas da matriz completa e o número de elementos não nulos da matriz.

Considere as seguintes definições de tipos:

<table>
<thead>
<tr>
<th>Tipo</th>
<th>Declaração</th>
<th>Descrição</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CONST</strong></td>
<td>MAX = 6;</td>
<td>(* um valor bem menor que 5 x 4, dimensão da matriz *)</td>
</tr>
<tr>
<td><strong>TYPE</strong></td>
<td>vetor_coordenadas = array [1..MAX] of integer;</td>
<td>(* coordenadas *)</td>
</tr>
<tr>
<td></td>
<td>vetor_elementos = array [1..MAX] of real;</td>
<td>(* dados *)</td>
</tr>
<tr>
<td><strong>VAR</strong></td>
<td>L, C: vetor_coordenadas;</td>
<td>(* L: linhas, C: colunas *)</td>
</tr>
<tr>
<td></td>
<td>D: vetor_elementos;</td>
<td>(* D: dados *)</td>
</tr>
<tr>
<td></td>
<td>N_lin, N_col: integer;</td>
<td>(* para armazenar as dimensões da matriz *)</td>
</tr>
<tr>
<td></td>
<td>N_elementos: integer</td>
<td>(* número de elementos não nulos *)</td>
</tr>
</tbody>
</table>

**Definição 1** Um elemento $M[i,j]$ da matriz completa pode ser obtido da representação compactada:

- se existe um $k$ tal que $L[k] = i$ e $C[k] = j$, então $M[i,j] = D[k]$;
- caso contrário, $M[i,j] = 0$.

A matriz do exemplo anterior pode então ser assim representada:

```
N_elementos:= 4; N_lin:= 5; N_col:= 4;

1 2 3 4 5 6
1 0 17 0 0
2 0 0 0 0
3 13 0 -12 0
4 0 0 25 0
5 0 0 0 0

L   1  2  3  4  5  6
  1  3  3  4

C   2  1  3  3

D  17 13 -12 25
```
(a) Faça um procedimento em Pascal que leia da entrada padrão:
- dois inteiros, representando as dimensões da matriz (linha, coluna);
- trincas de elementos l, c, d, onde l e c são inteiros e d é real, representando respectivamente a linha, a coluna e o valor de um elemento não nulo da matriz. A leitura termina quando for lido uma trinca 0, 0, 0. Para cada trinca, devem ser criados os três vetores que representam a matriz conforme descrito acima. Veja o exemplo de entrada de dados, abaixo.

Exemplo para a entrada de dados:

5 4
1 2 17
3 1 13
3 3 -12
4 3 25
0 0 0

(b) Faça uma função em Pascal que, dada uma coordenada (l, c), respectivamente para uma linha e coluna, retorne o valor de elemento M[l, c], conforme a definição [1].

(c) Faça um procedimento em Pascal que, dadas duas matrizes no formato compactado descrito acima, obtenha uma terceira matriz compactada que é a soma das duas primeiras.

(d) Faça um procedimento em Pascal que, dada uma matriz no formato compactado, imprima na tela uma matriz no formato padrão, contendo os zeros.

26. Faça um programa em Pascal que tenha declarada uma matriz $M \times N$ de caracteres do tipo char. Implemente quatro funções que, dados como parâmetros a matriz, uma palavra do tipo string e um par de coordenadas na matriz, isto é, dois inteiros representando uma linha e uma coluna, descubram se, na matriz, a palavra ocorre iniciando na posição indicada pelas coordenadas. A primeira função procura na horizontal, da esquerda para direita; a segunda função procura na horizontal, da direita para esquerda; a terceira função procura na vertical, da cima para baixo; a quarta função procura na vertical, da baixo para cima.

27. Os incas construíram pirâmides de base quadrada em que a única forma de se atingir o topo era seguir em espiral pela borda, que acabava formando uma escada em espiral. Faça um programa em Pascal que leia do teclado uma matriz quadrada $N \times N$ de números inteiros e verifique se a matriz é inca; ou seja, se partindo do canto superior esquerdo da matriz, no sentido horário, em espiral, a posição seguinte na ordem é o inteiro consecutivo da posição anterior. Por exemplo, as matrizes abaixo são incas:
O programa deve ler do teclado a dimensão da matriz (um inteiro $N$, $1 \leq N \leq 100$) e em cada uma das próximas $N$ linhas, os inteiros correspondentes às entradas da matriz naquela linha. A saída do programa deve ser “A matriz eh inca” ou “A matriz nao eh inca”.


29. Nesta questão você deve providenciar ligações par-a-par entre diversos pontos distribuídos ao longo de uma rota qualquer. A entrada de dados consiste de um conjunto de pares $(x, y)$, $1 \leq x, y \leq MAX$, sendo que o último par a ser lido é o $(0,0)$, que não deve ser processado.

Para cada par $(x, y)$ dado como entrada, você deve providenciar uma conexão física entre eles. As linhas de uma matriz podem representar a “altura” das linhas de conexão, enquanto que as colunas da matriz podem representar os pontos $(x, y)$ sendo conectados. Um símbolo de “+” pode ser usado para se representar alteração na direção de uma conexão. O símbolo “|” pode ser usado para representar um trecho de conexão na vertical. Finalmente o símbolo “-” pode ser usado para se representar um trecho de conexão na direção horizontal. Quando um cruzamento de linhas for inevitável, deve-se usar o símbolo “x” para representá-lo. Considere que não existem trechos de conexões na diagonal.

Por exemplo, suponha que a entrada é dada pelos seguintes pares:

```
3 5
2 9
0 0
```

Uma possível saída para seu programa seria a impressão da seguinte matriz:

```
4
3
2 +-------------------+
1 | +----+       |
  1 2 3 4 5 6 7 8 9
```

Outra possível matriz solução para este problema seria esta:
Note que nesta última versão foi preciso inserir dois cruzamentos.

Ainda como exemplo, se o par (6,8) também fosse dado como entrada no exemplo anterior, a saída do programa poderia ser assim exibida:

4
3
2 +----+
1 | +---+ +---+ |
1 2 3 4 5 6 7 8 9

Faça um programa em *Pascal* que seja capaz de ler uma sequência de pares terminada em (0,0) (como no exemplo acima) e que imprima o desenho das conexões como saída, também conforme o diagrama acima.

30. Modifique o programa anterior com o objetivo de minimizar o número de cruzamentos da matriz gerada como solução do problema anterior. Assim, a matriz ideal para ser dada como resposta do último exemplo seria a seguinte:

4
3
2 +-----+
1 | +---- | +---+ |
1 2 3 4 5 6 7 8 9

31. Considere o seguinte programa:

```pascal
program prova_final;

CONST MAX=100;
TYPE matriz = array [1..MAX,1..MAX] of integer;
VAR n_lin, n_col: integer; (* dimensoes da matriz *)
m: matriz; (* matriz *)

(* espacio reservado para os procedimentos *)

begin
  read (n_lin, n_col);
  le_matriz (m, n_lin, n_col);
  acha_maior_sequencia (m, n_lin, n_col, l_ini, c_ini, l_fim, c_fim);
  writeln ('A maior sequencia de numeros repetidos na matriz ');
  writeln ('inicia na coordenada ', l_ini, c_ini);
  writeln (' e termina na coordenada ', l_fim, c_fim);
end.
```
Faça em *Pascal* os procedimentos indicados para que o programa leia uma matriz de inteiros e imprima as coordenadas de início e término da maior sequência de números repetidos da matriz. Esta sequência pode estar tanto nas linhas quanto nas colunas. No caso de existir mais de uma sequência repetida de mesmo tamanho, você pode imprimir as coordenadas de qualquer uma delas, desde que imprima as de uma só.

<table>
<thead>
<tr>
<th>Exemplo 1:</th>
<th>Exemplo 2:</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Entrada:</strong></td>
<td><strong>Entrada:</strong></td>
</tr>
<tr>
<td>4 3 1 2</td>
<td>4 5 2 2</td>
</tr>
<tr>
<td>1 2 3 3 2</td>
<td>1 2 3 1 2</td>
</tr>
<tr>
<td>2 2 1</td>
<td>2 4</td>
</tr>
<tr>
<td>3 2 5</td>
<td>2 3 4 5 6</td>
</tr>
<tr>
<td><strong>Saída:</strong></td>
<td><strong>Saída:</strong></td>
</tr>
<tr>
<td>1 2</td>
<td>2 4</td>
</tr>
</tbody>
</table>

32. Faça um programa em *Pascal* que:

- leia uma sequência de polinômios \( P_i(x) = a_{i_0} + a_{i_1}x + a_{i_2}x^2 + ... + a_{i_n}x^n, i = 1, 2, ..., k; \)
- A leitura deve considerar que cada linha de entrada contém um polinômio \( P_i \). A primeira informação é o seu respectivo grau \( n_i \). As outras informações são os \( n_i \) coeficientes \( (a_{i_0}, a_{i_1}, ..., a_{i_n}) \); Exemplo:
  \[ P(x) = 8.1 - 3.4x + x^2 \implies 2 \quad 8.1 \quad -3.4 \quad 1.0 \]
- A sequência de polinômios se encerra quando for fornecido um polinômio de grau zero;
- Após a leitura de todos os polinômios, o programa deve ler uma sequência de números reais \( x \). Para cada número real lido, o programa deve imprimir o resultado de \( P_i(x) \), para todos os polinômios lidos anteriormente \( (i = 1, 2, ..., k) \);
- A sequência de números reais se encerra quando for lido o número 0.0, para o qual não se deve calcular os valores de \( P_i(x) \).

<table>
<thead>
<tr>
<th>Exemplo:</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Entrada:</strong></td>
</tr>
<tr>
<td>2 \quad -1.0 \quad 0.0 \quad 1.0</td>
</tr>
<tr>
<td>3 \quad 1.0 \quad 2.0 \quad 0.0 \quad -1.0</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>4.5</td>
</tr>
<tr>
<td>1.0</td>
</tr>
<tr>
<td>0.0</td>
</tr>
<tr>
<td><strong>Saída:</strong></td>
</tr>
<tr>
<td>( P_1(2.0) = 3.0 )</td>
</tr>
<tr>
<td>( P_2(2.0) = -3.0 )</td>
</tr>
</tbody>
</table>
9.2. MATRIZES

\[ P_{1}(1.0) = 0.0 \]
\[ P_{2}(1.0) = 2.0 \]

33. Faça um programa em \textit{Pascal} que:

- leia um inteiro \( N \) e uma matriz quadrada de ordem \( N \) contendo apenas 0’s e 1’s.
- encontre a maior submatriz quadrada da matriz de entrada que contém apenas 1’s.
- imprima as coordenadas dos cantos superior esquerdo e inferior direito da submatriz encontrada no item anterior. Havendo mais de uma submatriz máxima, imprimir as coordenadas de qualquer uma delas.

Exemplo: Considere a seguinte matriz quadrada de ordem 6:

\[
\begin{array}{cccccc}
1 & 2 & 3 & 4 & 5 & 6 \\
1 & 0 & 1 & 0 & 1 & 1 \\
2 & 0 & 1 & 1 & 1 & 1 & 0 \\
3 & 0 & 1 & 1 & 1 & 0 & 1 \\
4 & 1 & 1 & 1 & 1 & 0 & 1 \\
5 & 0 & 0 & 1 & 0 & 1 & 0 \\
6 & 0 & 1 & 0 & 1 & 0 & 1 \\
\end{array}
\]

A título de ilustração, esta matriz tem:

- 22 submatrizes quadradas de ordem 1 que contém apenas 1’s;
- 5 submatrizes quadradas de ordem 2 que contém apenas 1’s. Por exemplo, para duas delas: uma é dada pelas coordenadas (1,4) e (2,5) e outra pelas coordenadas (2,2) e (3,3);
- 1 submatriz quadrada de ordem 3 que contém apenas 1’s, as coordenadas são (2,2) e (4,4).

Como a maior submatriz quadrada que contém apenas 1’s é a de ordem 3, então a saída do programa deve imprimir, para este exemplo, as coordenadas (2,2) e (4,4).

34. Faça um programa em \textit{Pascal} que, dado um tabuleiro e uma lista de subpartes retangulares do tabuleiro, retorna o número de posições que não pertencem a nenhuma subparte. Quando uma posição não pertence a nenhuma subparte dizemos que ela está \textit{perdida}.

\textbf{Entrada}
CAPÍTULO 9. ESTRUTURAS DE DADOS

A entrada consiste de uma série de conjuntos de teste.

Um conjunto de teste começa com uma linha com três números \(W, H\) e \(N\), indicando, respectivamente, a largura e a altura do tabuleiro e o número de subpartes deste. Estes valores satisfazem as seguintes restrições: \(1 \leq W, H \leq 500\) e \(0 \leq N \leq 99\).

Seguem \(N\) linhas, compostas de quatro inteiros \(X_1, Y_1, X_2\) e \(Y_2\), tais que \((X_1, Y_1)\) e \((X_2, Y_2)\) são as posições de dois cantos opostos de uma subparte. Estes valores satisfazem as seguintes restrições: \(1 \leq X_1, X_2 \leq W\) e \(1 \leq Y_1, Y_2 \leq H\).

O fim da entrada acontece quando \(W = H = N = 0\). Esta última entrada não deve ser considerada como um conjunto de teste.

Saída

O programa deve imprimir um resultado por linha, seguindo o formato descrito no exemplo de saída.

Exemplo

Entrada:

\[
\begin{align*}
  &1 \ 1 \ 1 \\
  &1 \ 1 \ 1 \ 1 \quad \text{\{fim do primeiro conjunto de testes\}} \\
  &2 \ 2 \ 2 \\
  &1 \ 1 \ 1 \ 2 \\
  &1 \ 1 \ 2 \ 1 \quad \text{\{fim do segundo conjunto de testes\}} \\
  &493 \ 182 \ 3 \\
  &349 \ 148 \ 363 \ 146 \\
  &241 \ 123 \ 443 \ 147 \\
  &303 \ 124 \ 293 \ 17 \quad \text{\{fim do terceiro conjunto de testes\}} \\
  &0 \ 0 \ 0 \quad \text{\{fim do conjunto de testes\}}
\end{align*}
\]

Saída

Não há posições perdidas.

Existe uma posição perdida.

Existem 83470 posições perdidas.
9.3 Registros

Até agora vimos, como estruturas de dados, somente vetores e matrizes. Estas estruturas são ditas *homogêneas*, no sentido que as diversas posições de memória alocadas são sempre do mesmo tipo.

Para completarmos nosso estudo básico de algoritmos, resta ainda introduzir a noção de *registros*, que são estruturas *heterogêneas*, isto é, pode-se alocar várias posições de memória cada uma delas de um tipo potencialmente diferente.

9.3.1 Introdução aos registros

Suponhamos que seja necessário implementar um cadastro de um cliente de um banco. Normalmente este cadastro contém: nome do cliente, telefone, endereço, idade, RG e CPF. Usando-se um registro, podemos agrupar todos os dados diferentes em uma só variável. Por exemplo, em *Pascal* podemos declarar tal variável assim:

```pascal
var r: record
  nome: string[50];
  fone: longint;
  endereco: string;
  idade: integer;
  rg: longint;
  cpf: qword;
end;
```

Cada linguagem de programação tem sua sintaxe própria para a declaração e acesso aos dados. Nos vetores e matrizes, o acesso é feito usando-se o nome da variável e um índice (ou um par no caso das matrizes). Para os registros, em *Pascal*, usa-se o nome da variável, um ponto, e o nome do campo, que é escolhido pelo programador.

Por exemplo, é válido em *Pascal* a seguinte sequência de comandos:

```pascal
r.nome := 'Fulano de Tal';
r.fone := 32145678;
r.endereco := 'Rua dos bobos, no 0';
r.idade := 75;
r.rg := 92346539;
r.cpf := 11122233344;
```

Também seria válido ler a partir do teclado da seguinte maneira:

```pascal
  read (r.nome);
  read (r.fone);
  read (r.endereco);
  read (r.idade);
  read (r.rg);
  read (r.cpf);
```

Contudo, assim como se dá para o tipo *array*, para se passar um parâmetro de procedimento ou função em *Pascal* é necessário antes a declaração de um novo tipo, que poderia ser desta maneira:
type cliente = record
    nome: string[50];
    fone: longint;
    endereco: string;
    idade: integer;
    rg: longint;
    cpf: qword;
end;

var r: cliente;

Na verdade a linguagem Pascal permite uma facilidade para se economizar alguma digitação através do comando with. A figura 9.66 ilustra uma forma de se imprimir todo o conteúdo de um registro usando-se um procedimento. O comando with pode ser usado para leitura ou atribuição também.

procedure imprime_reg (r: cliente);
begin
    with r do
    begin
        writeln (nome);
        writeln (fone);
        writeln (endereco);
        writeln (idade);
        writeln (rg);
        writeln (cpf);
    end;
end;

Figura 9.66: Imprimindo registros.

Normalmente é mais comum ver os registros integrados a outras estruturas, tais como vetores, matrizes ou arquivos em disco[13]. Nas próximas duas seções veremos como integrar vetores com registros.

9.3.2 Registros com vetores

Nesta sessão vamos ver como implementar um registro que tem, como um dos campos, um vetor. Para isto, vamos considerar as seguintes definições:

const MAX = 10000;
type vetor = array [1..MAX] of real;
tipo_vetor = record
    tam: integer;
    dados: vetor;
end;

var v: tipo_vetor;

A ideia é encapsular o tamanho do vetor junto com o próprio vetor. Isto facilita na hora de passar parâmetros, entre outras coisas. Em uma figura de linguagem, é como se o vetor “soubesse” seu tamanho, sem precisar passar um parâmetro indicando isto.

O conceito é simples, vamos ver como pode ser feita a leitura de um vetor nesta nova estrutura de dados. Isto é apresentado na figura 9.67.

```pascal
procedure ler vetor (var v: tipo_vetor);
var i: integer;
begin
  readln (v.tam); (* tamanho do vetor *)
  for i:= 1 to v.tam do
    readln (v.dados[i]) (* elementos do vetor *)
end;
```

Figura 9.67: Lendo vetores implementados em registros.

É importante observar o correto uso dos símbolos de ponto (.) e dos colchetes, eles têm que estar no lugar certo. Uma vez que, no exemplo acima, v é uma variável do tipo registro, ela deve receber inicialmente um ponto para se poder acessar um dos dois campos. Se quisermos acessar o tamanho, então a construção é v.tam. Se quisermos acessar o vetor de reais, então a construção correta é v.dados. Ocorre que v.dados é um vetor, logo, deve-se indexar com algum inteiro, por isto a construção final correta é v.dados[i].

Esta estrutura será muito útil no capítulo 10.

### 9.3.3 Vetores de registros

Considerando novamente o exemplo do cliente do banco. Uma maneira ainda um pouco precária de se manipular muitos clientes é usando a estrutura de vetores em combinação com a de registros. A título de exemplo, consideremos então as seguintes definições:

```pascal
const MAX = 10000;
type cliente = record
  nome: string[50];
  fone: longint;
  endereco: string;
  idade: integer;
  rg: longint;
  cpf: qword;
end;

bd = array [1..MAX] of cliente;
var
  r: cliente;
  v: bd;
  tam_v: integer;
```
Isto é, temos um vetor de tam_v clientes!
Vamos imaginar que o banco é novo na praça e que é preciso criar o banco de dados contendo os clientes. Podemos usar o procedimento que é mostrado na figura 9.68.

```
procedure ler_cliente (var r: cliente);
begin
  with r do
  begin
    readln (nome);
    readln (fone);
    readln (endereco);
    readln (idade);
    readln (rg);
    readln (cpf);
  end;
end;
```

```
procedure carregar_todos_clientes (var v: bd; var tam_v: integer);
begin
  readln (tam_v);
  for i:= 1 to tam_v do
    ler_cliente (v[i]);
end;
```

Figura 9.68: Lendo os clientes do banco.

Os algoritmos para busca, ordenação e outros tipos de manipulação desta nova estrutura devem levar em conta agora qual é o campo do registro que deve ser utilizado. Por exemplo, se quisermos imprimir o telefone do cliente do banco cujo CPF seja 1234567899, então é no campo r.cpf que devemos centrar atenção durante a busca, mas na hora de imprimir, deve-se exibir o campo r.fone. Vejamos um exemplo na figura 9.69.

```
procedure busca_telefone (var v: bd; tam_v: integer; cpf_procurado: string);
var i: integer;
begin
  i:= 1;
  while (i <= tam_v) and (v[i].cpf <> cpf_procurado) do
    i:= i + 1;
  if i <= tam_v then
    writeln ('O telefone do cliente com CPF ',v[i].cpf,' eh: ',v[i].fone)
  else
    writeln (' Cliente nao localizado na base.' );
end;
```

Figura 9.69: Imprime o telefone do cliente que tem um certo CPF.

O campo do registro de interesse para um algoritmo normalmente é denominado chave. Por exemplo, vamos tentar ordenar o banco de dados. Por qual chave devemos
fazer isto, já que temos uma estrutura contendo 6 campos diferentes? Vamos conven-cionar que a ordenação se dará pelo CPF do cliente. O algoritmo de ordenação pode ser, por exemplo, o método da seleção. Mas deve-se observar que, durante as trocas, *todo o registro deve ser trocado*, sob pena de misturarmos os dados dos clientes! A figura 9.70 ilustra tal situação.

```plaintext
procedure ordena_por_cpf (var v: bd; tam: integer);
var i, j, pos_menor: integer;
begin
  for i:= 1 to tam−1 do
    begin
      (* acha o menor elemento a partir de i *)
      pos_menor:= i;
      for j:= i+1 to tam do
        if v[j].cpf < v[pos_menor].cpf then
          pos_menor:= j;
      troca (bd, i, pos_menor); (* troca os elementos *)
    end;
end;
```

Figura 9.70: Ordena pelo CPF.

A procedure da figura 9.71 apresenta o código que faz a troca dos registros nas posições corretas.

O estudante é encorajado a praticar vários exercícios até compreender bem estas noções. Uma vez compreendido, não haverá dificuldades em prosseguir no estudo de algoritmos e estruturas de dados. A maior parte das estruturas sofisticadas são variantes das construções estudadas nesta seção.
procedure troca (var v: bd; k,m: integer);
var aux: cliente;
begin
  with aux do
  begin
    nome:= v[k].nome;
    fone:= v[k].fone;
    endereco:= v[k].endereco;
    idade:= v[k].idade;
    rg:= v[k].rg;
    cpf:= v[k].cpf;
  end;

  with v[k] do
  begin
    nome:= v[m].nome;
    fone:= v[m].fone;
    endereco:= v[m].endereco;
    idade:= v[m].idade;
    rg:= v[m].rg;
    cpf:= v[m].cpf;
  end;

  with v[m] do
  begin
    nome:= aux.nome;
    fone:= aux.fone;
    endereco:= aux.endereco;
    idade:= aux.idade;
    rg:= aux rg;
    cpf:= aux.cpf;
  end;

Figura 9.71: Troca os elementos do banco de dados.
9.3.4 Exercícios

1. Latitude e longitude são especificados em graus (°), minutos (′), segundos (″), e direção (N, S, L, O). Por exemplo, a cidade A fica na latitude 22°17′34″N e longitude 53°41′9″O.

Defina um tipo em Pascal cujo nome seja localização, e que seja constituído de longitude e latitude, ambas do tipo coordenadas. Para isto você terá que definir o tipo coordenadas como sendo constituído de grau, minutos, segundos e direção.

2. Declare um vetor em Pascal onde cada elemento é um registro com os campos: nome, DDD, telefone, CPF, Idade.

3. Considerando o vetor não ordenado, encontrar e imprimir o nome do cliente mais jovem. Faça um procedimento para isto.

4. Ordenar por ordem de nome. Faça um procedimento para isto.

5. Dado um CPF, localizar se o nome está no vetor e imprimir todos os dados. Faça um procedimento para isto.

6. Faça um procedimento que receba por valor parâmetros para nome, DDD, telefone, CPF, Idade e o insira no vetor (que já está ordenado) em ordem alfabética. Não vale usar um vetor auxiliar. Retornar por referência o vetor alterado.

7. Considere o arquivo de uma empresa (chamado de “func.dat” – um arquivo de registros) contendo para cada funcionário seu número, seu nível salarial e seu departamento. Como a administração desta empresa é feita a nível departamental é importante que no arquivo os funcionários de cada um dos departamentos estejam relacionados entre si e ordenados sequencialmente pelo seu número. Como são frequentes as mudanças interdepartamentais no quadro de funcionários, não é conveniente reestruturar o arquivo a cada uma destas mudanças. Desta maneira, o arquivo poderia ser organizado da seguinte forma:

```
linha  numFunc  nivel  departamento  proximo
0     123      7      1           5
1     8765     12     1           -1
2     9210     4      2           -1
3     2628     4      3           6
4     5571     8      2           -1
5     652      1      1           9
6     7943     1      3           -1
7     671      5      3           12
8     1956     11     2           11
9     1398     6      1           10
10    3356     3      1           1
11    4050     2      2           4
12    2468     9      3           3
```
Em um segundo arquivo (chamado “depto.dat” – um arquivo de registros) temos as seguintes informações:

<table>
<thead>
<tr>
<th>codDepto</th>
<th>nomeDepto</th>
<th>inicio</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>vendas</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>contabilidade</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>estoque</td>
<td>7</td>
</tr>
<tr>
<td>4</td>
<td>entrega</td>
<td>2</td>
</tr>
</tbody>
</table>

Assim, o primeiro funcionário do departamento de vendas é o registro 0 do arquivo de funcionários e os demais funcionários do mesmo departamento são obtidos seguindo o campo proximo. Ou seja, os funcionários do departamento de vendas são os funcionários nos registros: 0, 5, 9, 10 e 1. Os funcionários do departamento de contabilidade são os funcionários nos registros: 8, 11 e 4.

Faça um programa em *Pascal* que realize as seguintes operações:

- admissão de novo funcionário
- demissão de funcionário
- mudança de departamento por um funcionário

Para estas operações devem ser lidas as informações:

- código do tipo da operação: 0 para fim, 1 para admissão, 2 para demissão e 3 para mudança de departamento
- número do funcionário
- nível salarial (somente no caso de admissão)
- número do departamento ao qual o funcionário passa a pertencer (no caso de admissão e mudança)
- número do departamento do qual o funcionário foi desligado (no caso de demissão e mudança)

O programa deve escrever as seguintes informações:

- os valores iniciais lidos dos arquivos
- para cada operação: o tipo da operação realizada, os dados da operação e a forma final dos dados (de funcionários e departamentos)

No final do programa novos arquivos “func.dat” e “depto.dat” são gerados com os dados atualizados.

Detalhamento:

- a quantidade máxima de funcionários é 1000;
- a quantidade máxima de departamentos é 20;
9.3. REGISTROS

- se a quantidade máxima for ultrapassada o programa deve dar uma mensagem de erro;
- se for requisitada a remoção ou mudança de um funcionário não existente no departamento especificado o programa deve dar uma mensagem de erro;
- quando for requisitada a inserção de um novo funcionário é preciso verificar se um funcionário com o mesmo número já existe.
- se o código de operação for inválido o programa deve continuar lendo um novo código até que ele seja 0 (zero), 1 (um), 2 (dois) ou 3 (três).

8. Usando as estruturas de dados abaixo faça um procedimento em Pascal que receba como parâmetro uma estrutura do tipo TENDA e a ordene de forma crescente o vetor pessoa dessa estrutura tomando como referência para a ordenação o campo nome da estrutura TPESSOA. Ou seja, ordena uma agenda pessoal de telefones e endereços em ordem crescente do nome das pessoas presentes na agenda. Você deve usar a função compara(r, s), que recebe dois parâmetros do tipo string e retorna 0 se r e s são iguais, 1 se r é lexicograficamente maior que s e −1 se r é lexicograficamente menor que s. Um nome n₁ é lexicograficamente maior que um nome n₂ se n₁ aparece depois de n₂ numa ordenação alfabética crescente desses nomes.

Const
MAX = 1000;
Type
TPESSOA = record
nome: string;
telefone: string;
endereco: string
end;
TAGENDA = record
pessoa: array [1..MAX] of TPESSOA;
tamanho: integer
end;

9. Uma matriz é dita esparsa quando a maioria dos seus elementos possui valor 0.0 (zero). Neste caso, a representação da matriz sob a forma tradicional (um array bidimensional) implica em uma utilização ineficiente da memória. Por isso, matrizes esparsas são frequentemente representadas como vetores de elementos não nulos, sendo que cada elemento contém suas coordenadas e seu valor.

Exemplo:

\[
M = \begin{bmatrix}
0 & 0 & 0 & 1.2 \\
7.3 & 0 & 99 & 0 \\
0 & 2 & 0 & 0 \\
0 & 17 & 0 & 0
\end{bmatrix}
\]

\[\iff M_e = \begin{bmatrix}
1 & 4 \\
1.2 & 7.3 \\
2 & 3 \\
3 & 2 \\
4 & 2
\end{bmatrix}
\]

Para representar estas matrizes em Pascal podemos definir as seguintes estruturas de dados:
Utilizando as estruturas de dados definidas acima, faça em *Pascal*:

(a) uma função que transforme uma matriz do tipo `t_matriz` em uma matriz do tipo `t_matrizesp`.

(b) uma função que transforme uma matriz do tipo `t_matrizesp` em uma matriz do tipo `t_matriz`.

(c) uma função que receba duas matrizes do tipo `t_matrizesp` e imprima o resultado da *soma* destas matrizes. O resultado deve ser impresso na forma bidimensional, com os valores de cada linha separados por espaços.

10. Verdadeiro ou falso:

- um record deve ter pelo menos dois campos;
- os campos de um record tem que ter nomes diferentes;
- um record deve ter um nome diferente de qualquer um dos seus campos.

11. Suponha que a linguagem *Pascal* foi modificada para permitir que o símbolo ponto `.”` possa fazer parte de identificadores. Qual problema isto poderia causar? Dê um exemplo.

12. Esta definição é legal? Porque não?

```pascal
TYPE
  Ponto = RECORD
    quantidade: integer;
    corte: Tamanho;
  END;
  Tamanho = (mini, medio, maxi);
```

13. Latitudes e longitudes são especificadas em graus (°), minutos (’), segundos (") e direção (norte, sul, leste, oeste). Suponha que uma cidade esteja na latitude 22°17’34" norte e longitude 53o41’9”oeste. Faça um programa em *Pascal* que leia estas informações e a armazene na variável *Cidade*, como declarada abaixo. Em seguida imprima as coordenadas no formato especificado.
9.3. REGISTROS

TYPE
    TipoDirecao = (norte, sul, leste, oeste);
    Coordenadas = RECORD
        graus: 0..180;
        minutos, segundos: 0..60;
        direcao: TipoDirecao;
    END;
    Localizacao = RECORD
        latitude, longitude: Coordenadas;
    END;

VAR
    Cidade: Localizacao;

14. Suponha que temos as seguintes definições e declarações abaixo:

    TYPE
        NumTelefone = RECORD
            Area, Prefixo, Numero: integer;
        END;
    VAR
        Casa, Escritorio, Celular: NumTelefone;

Faça um programa em Pascal que teste se Escritório e Celular estão sobre o mesmo código de área e para atribuir o mesmo número do Celular como sendo o do Escritório.

15. Criar em Pascal registros para as seguintes configurações da vida real:

- Pessoas com nome, endereço, telefone, sexo, idade e salário;
- Planetas com massa, distância média do sol, número de satélites, nome, diâmetro.
- Cursos, com alunos e suas notas, períodos em que são oferecidos, professor, horário.

16. Considere o tipo PGM para imagens como definido na seção 9.2.5. Nas questões que seguem, considere as seguintes estruturas de dados e assinaturas de funções e procedimentos:

```
const MAX=10000;

type
    matriz = array [1..MAX, 1..MAX] of integer;
    vetor = array [1..MAX] of integer;
    imagem = record
        col, lin, maior: integer;
    end;
```
m: matriz;

imgcompactada = record
tam: integer;
v: vetor;
end;

function calcula_valor_medio (var I: imagem): integer;
(* função que retorna o valor médio dos pixels da imagem, isto é, a soma de todos os elementos dividido pelo número de elementos *)

procedure ler (var I: imagem);
(* procedimento que le uma imagem no formato PGM *)

procedure imprime_imagem (var I: imagem);
(* procedimento que imprime uma imagem no formato PGM *)

procedure binariza (var I: imagem; limiar: integer);
(* procedimento que transforma a imagem de tons de cinza para preto e branco para isto, os pixels que forem maiores que o limiar devem se tornar brancos e os que forem menores ou iguais a este mesmo limiar devem se tornar pretos *)

procedure compacta_imagem (var I: imagem; var C: imgcompactada);
(* procedimento que recebe uma imagem no formato PGM e cria um vetor C que é uma representação compactada desta *)

procedure imprime_img_compactada (var C: imgcompactada);
(* procedure que recebe uma imagem compactada e a imprime no formato PGM *)

(a) Implemente estas funções e procedimentos em Pascal e faça um programa que receba um certo número $N$ de imagens PGM em tons de cinza (onde 0 representa preto e branco é representado pelo maior valor da imagem) e imprima a imagem binarizada, isto é, em preto e branco (onde 0 representa preto e 1 representa branco). Note que o limiar é obtido pelo valor médio dos pixels.

(b) Implemente um procedimento em Pascal que gere um vetor que representa a matriz binarizada de forma compacta. Para isto, use a seguinte ideia: como a matriz só tem zeros e uns, vamos substituir sequências de uns pelo número de uns consecutivos. Os elementos vão sendo colocados no vetor, de maneira linear, cada linha seguinte é concatenada à anterior. Veja o exemplo: Exemplo:

- **Imagem binarizada:**

```
P2
11 10
1
1 1 1 1 0 1 1 1 1 0
1 1 0 1 1 1 1 1 1 1
0 0 1 0 0 0 1 1 1 0
1 1 1 1 1 1 1 1 1 1
```
Isto é, a primeira linha da matriz possui 4 uns consecutivos seguido de um zero e outros 5 uns consecutivos, por isto, o vetor contém seus primeiros elementos “4, 0 e 5”. Preste atenção antes de escrever o código. Você pode definir, se precisar, funções, procedimentos ou estruturas de dados adicionais.
Capítulo 10

Tipos abstratos de dados

Um Tipo Abstrato de Dados (TAD) é basicamente abstrair o dado que está em memória pela definição de operações abstratas que se preocupam fundamentalmente em o que fazer e não em como fazer.

O conjunto de operações sobre os dados são definidos em termos dos protótipos das funções e procedimentos, para os quais os códigos que as implementam são absolutamente irrelevantes.

Evidentemente que em algum momento deverá haver código escrito que irá manipular os dados em sua forma concreta. Mas para o usuário de um TAD o que importa é apenas o uso das operações sobre um dado abstrato.

O tipo em si juntamente com os códigos das funções e procedimentos pode ser feita de várias maneiras, mas na verdade, para quem usa, não faz diferença alguma, a não ser o comportamento destas funções e procedimentos.

Vamos apresentar dois TAD’s básicos para que a ideia seja melhor compreendida, o TAD Conjunto e o TAD Pilha. A forma de apresentação será:

1. Definição do conceito conjunto/pilha;
2. Solução de problemas que podem ser resolvidos com estes conceitos;
3. Por último, implementações alternativas concretas dos tipos, com os respectivos custos computacionais.

10.1 Tipo Abstrato de Dados Conjunto

Um conjunto é um conceito matemático bem conhecido. Trata-se de uma coleção de elementos sem repetição. Operações importantes sobre conjuntos incluem saber se o conjunto é vazio, se um elemento pertence ou não a um conjunto, inserir ou remover elementos ou realizar operações de união e intersecção, por exemplo.

\footnote{Agradecemos aos professores André Luiz Pires Guedes, Carmem Satie Hara, Daniel Alfonso de Oliveira e Luís Carlos Erpen de Bona pelas valiosas contribuições para este capítulo.}
Vamos considerar a existência de um tipo denominado **conjunto**, mas neste momento não estamos interessados em saber como este tipo é de fato definido. Sabemos o que é um conjunto, e isso nos é suficiente no momento. Declaramos então:

```pascal
var c: conjunto;
```

Neste capítulo, um conjunto será constituído de números inteiros do tipo `longint` da linguagem `Pascal`.

Primeiramente, vamos definir as interfaces, deixando a implementação para um segundo momento. Por interfaces queremos dizer qual é o comportamento das operações que manipulam o TAD conjunto.

Reforçamos que não é preciso conhecer concretamente como um conjunto é implementado, basta saber que as funções e procedimentos estarão corretamente implementados e poderemos resolver vários tipos de problemas apenas conhecendo o comportamento das operações.

As principais operações sobre conjuntos são definidas em termos dos protótipos dos procedimentos e funções abaixo.

```pascal
function conjunto_vazio (c: conjunto): boolean;
// Retorna true se o conjunto c é vazio e false caso contrário.

function cardinalidade (c: conjunto): longint;
// Retorna a cardinalidade do conjunto c.

procedure inserir_conjunto (x: longint; var c: conjunto);
// Insere o elemento x no conjunto c, mantém os elementos ordenados.

procedure remover_conjunto (x: longint; var c: conjunto);
// Remove o elemento x do conjunto c.

function uniao (c1, c2: conjunto): conjunto;
// Obtém a uniao dos conjuntos c1 e c2.

function interseccao (c1, c2: conjunto): conjunto;
// Obtém a interseccao dos conjuntos c1 e c2.

function diferenca (c1, c2: conjunto): conjunto
// Obtém a diferença dos conjuntos c1 e c2 (c1 \( c2\)).

function pertence (x: longint; c: conjunto): boolean;
// Retorna true se x pertence ao conjunto c e false caso contrário.

function sao_iguais (c1, c2: conjunto): boolean;
// Retorna true se o conjunto c1 = c2 e false caso contrário.

function contido (c1, c2: conjunto): boolean;
// Retorna true se o conjunto c1 está contido no conjunto c2 e false caso contrário.

function copiar_conjunto (c1: conjunto): conjunto;
// Copia os elementos do conjunto c1 para outro conjunto.
```
Estas funções implementam operações óbvias, tais como: teste de conjunto vazio, retornar a cardinalidade de conjunto, inserir ou remover elementos, testar pertinência, fazer união, interseção e diferença de conjuntos.

As que seguem abaixo não são básicas e merecem alguma explicação adicional.

- **inicializar_conjunto**: serve para inicializar a estrutura do conjunto e sua chamada deve anteceder qualquer outra;

- **retirar_um_elemento**: função útil quando se deseja fazer uma determinada operação sobre todos os elementos do conjunto. Normalmente é utilizada dentro de um laço controlado pela função conjunto_vazio. Por exemplo: enquanto o conjunto não for vazio, retire os elementos um a um e os processe.

- **iniciar_proximo e incrementar_proximo**: estas funções também permitem que se opere sobre todos os elementos do conjunto, mas sem removê-los. Por exemplo, acesse todos os elementos, um por um, e os processe. Para isso é necessário iniciar o contador do próximo usando-se a primeira função. A segunda serve para pegar o próximo. Quando se processou todos os elementos a segunda função retorna *false*. O sentido disso é mais ou menos o mesmo de quando queremos percorrer um vetor: usamos uma inicialização de um contador, por exemplo *i := 1*, em combinação com um incremento, por exemplo *i := i + 1*.

```
10.1.1 Usando o TAD Conjunto para resolver problemas

Nesta seção apresentamos dois problemas que podem ser resolvidos usando-se o TAD conjunto. O primeiro trata do problema de conhecer os vencedores da megassena enquanto que o segundo é conhecido como o problema da celebridade.

Encontrando apostadores vencedores da megassena

A Caixa Econômica Federal tem que descobrir duas vezes por semana se existem vencedores do concurso da megassena. A megassena é um concurso que paga um prêmio milionário para os apostadores que acertarem seis dezenas que são sorteadas. Cada apostador pode apostar de 6 a 15 dezenas. As dezenas são valores inteiros de 01 a 60.

Como entrada de dados vamos considerar que serão digitados no teclado:

1. O programa deve ler o sorteio feito em cerimônia pública pela Caixa Econômica Federal. Devemos fazer a carga de exatamente 6 números entre 01 e 60;
2. Em seguida, devemos ler um número \( N \) contendo o total de apostadores;
3. Na sequência, devemos ler \( N \) linhas de dados, cada uma com a aposta de um apostador. O primeiro valor da linha é o número \( M \) de apostas deste apostador, isto é, um número entre 6 e 15. O restante da linha são \( M \) números de 01 a 60;
4. Finalmente, para cada aposta, devemos compará-la com o sorteio, gerando como saída as apostas que fizeram a megassena (acertou 6 dezenas), ou a quina (acertou 5 dezenas) ou a quadra (acertou quatro dezenas).
5. Encerrar o programa.

Como o TAD conjunto pode nos ajudar? Se definirmos as variáveis _sorteio_ e _aposta_ como sendo do tipo _conjunto_, então o número de acertos de uma aposta com relação ao sorteio é simplesmente a cardinalidade da intersecção entre estes conjuntos! Podemos agora implementar o código conforme mostrado na figura 10.1.

É importante observar o uso da procedure _intersecção_ e da função _cardinalidade_. Se estas implementarem corretamente a definição dos conceitos, o programa funciona perfeitamente.

A figura 10.2 mostra como ler as apostas e o sorteio, na qual enfatizamos o uso das procedures _inicializar_conjunto_ e _inserir_conjunto_. A primeira garante a consistência dos dados e a segunda cuida de inserir os elementos no conjunto, para que não existam elementos repetidos e ao mesmo tempo garante a informação da cardinalidade deste conjunto. Em outras palavras, é possível usar o TAD conjunto sem sequer conhecer sua estrutura. Os dados estão abstraídos.
program megasena;
// usa um TAD conjunto ainda nao definido
// usa as funcoes e procedimentos acima definidos
var sorteio, aposta, intersec: conjunto;
    N, num_acertos: longint;
begin
    ler (sorteio,6);
    read (N);
    for i:= 1 to N do
    begin
        read (tam_aposta);
        ler (aposta, tam_aposta);
        intersec:= interseccao (sorteio,aposta);
        num_acertos:= cardinalidade (intersec);
        if num_acertos = 6 then
            writeln ('aposta ',i,' ganhou a megasena!')
        else if num_acertos = 5 then
            writeln ('aposta ',i,' ganhou a quina!')
        else if num_acertos = 4 then
            writeln ('aposta ',i,' ganhou a quadra!')
        // else nao imprime nada, o acerto foi no maximo 3 e eh perdedora
    end;
end.

Figura 10.1: Programa que usa o TAD conjunto para o problema da megassena.

procedure ler (var c: conjunto; tam_aposta: longint);
var i, num: longint;
begin
    inicializar_conjunto (c);
    for i:= 1 to tam_aposta do
    begin
        read (num);
        inserir_conjunto (num,c);
    end;
end;

Figura 10.2: Lendo os números apostados ou sorteados.
Encontrando celebridades

Uma celebidade é definida como uma pessoa que quando está presente em uma festa com \(N\) pessoas, apenas ela é conhecida por todos os outros presentes, mas não conhece ninguém. Um detalhe complica o problema: a celebidade pode não ter ido à festa. A questão é saber quem é a celebidade ou se ela não está presente nesta festa. Podemos apenas fazer perguntas do tipo: \(A\) conhece \(B\).

Vamos considerar que o organizador da festa possui uma matriz \(N \times N\) na qual as linhas e colunas representam os presentes, que são numerados de 1 a \(N\). A matriz contém um zero na coordenada \((i,j)\) se \(i\) não conhece \(j\) e contém um 1 caso \(i\) conheça \(j\). Consideramos \((i,i) = 0\). A matriz abaixo é um exemplo para uma festa com 5 pessoas e cuja celebidade é a pessoa representada na linha 2:

\[
\begin{pmatrix}
0 & 1 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 \\
1 & 1 & 0 & 1 & 0 \\
0 & 1 & 0 & 0 & 1 \\
0 & 1 & 0 & 1 & 0 \\
\end{pmatrix}
\]

É possível observar que a celebidade, caso esteja presente, é o índice de uma linha que só contém zeros com a respectiva coluna contendo todos os elementos iguais a 1, menos o elemento da diagonal principal.

Para encontrar a celebidade podemos procurar na matriz uma linha nula e depois verificar se a coluna só tem 1’s, o que custaria \(N^2\). É mais elegante usar o TAD conjunto.

Basta inicialmente inserir todos os presentes em um conjunto. Em seguida, retiramos dois elementos quaisquer, digamos \(A\) e \(B\). Se \(A\) conhece \(B\), ele não pode ser celebidade, mas \(B\) ainda pode ser, por isso reinserimos \(B\) de volta no conjunto. Caso \(B\) não conheça \(A\), então \(A\) não pode ser celebidade, mas \(B\) ainda pode ser, por isso reinserimos \(A\) de volta no conjunto.

Se repetirmos este raciocínio até o conjunto ser unitário, então este elemento é candidato a ser celebidade, pois ela pode não estar presente na festa. Toda esta operação custa duas vezes o número de presentes e é portanto linear.

A figura [10.3] mostra os principais trechos do programa que implementa esta ideia. Ele considera que, previamente, todos os presentes foram inseridos no conjunto \(c\) e que existe uma função \(\text{conhece}\) que retorna \(true\) caso \(a\) conheça \(b\) na matriz \(m\) que tem dimensão \(n\), e \(false\) caso contrário. Também é necessário, previamente, inicializar o conjunto \(c\) antes de inserir os presentes.

A figura [10.4] mostra o código da função que faz o teste final, isto é, se o único elemento que sobrou no conjunto é de fato a celebidade.
procedure eliminar_nao_celebridade (var c: conjunto; m: matriz; n: longint);
var a,b: longint;
begin
    while cardinalidade (c) > 1 do
        begin
            a:= retirar_um_elemento (c);
            b:= retirar_um_elemento (c);
            if conhece(a,b,m,n) then
                inserir (b,c)
            else
                inserir (a,c);
        end;
end;

Figura 10.3: Encontrando uma celebridade usando conjuntos.

function eh_celebridade (candidato: longint; m: matriz; n: integer): boolean;
var i: integer; ok: boolean;
begin
    ok:= true;
    i:= 1;
    while ok and (i <= n) do
        begin
            if (not conhece (i,candidato,m,n) or conhece (candidato,i,m,n)) and
                (i <> candidato) then
                ok:= false;
                i:= i + 1;
        end;
    eh_celebridade:= ok;
end;

Figura 10.4: Confirmando se um presente é uma celebridade.
10.1.2 Implementações do TAD conjunto

Chegamos no momento em que as implementações do TAD conjunto podem ser vistas. Era preciso antes o estudante entender a abstração do dado.

Existem várias maneiras de se implementar o TAD conjunto, elas incluem variantes que usam alocação estática ou até mesmo dinâmica.¹

Nesta seção mostraremos duas variantes que usam vetores, na primeira os vetores terão seus elementos ordenados. Na segunda, não. O importante é que independentemente da implementação a semântica das funções e procedimentos sejam mantidos, isto é, o funcionamento das funções e procedimentos não podem ser alterados de uma implementação para outra.

Implementação 1

Nesta implementação vamos usar um vetor com elementos ordenados. A primeira tarefa é definir o dado propriamente dito, para depois podermos implementar as funções e procedimentos. Vamos encapsular o tamanho do vetor juntamente com o vetor em uma estrutura de registros, tal como mostrado na seção 9.3.2.

```
const max = 101;

type
  conjunto = record
    tam: longint;
    proximo: longint;
    v: array [0..MAX+1] of longint;
  end;

// Primeira e a ultima posicoes (c.v[0] e c.v[MAX+1]): sentinelas.
```

Agora podemos implementar algumas funções e procedimentos, iniciando pelas mais simples, conforme as figuras 10.5, 10.6 e 10.7. Muitas destas implementações já foram feitas no capítulo 9, porém optamos por mostrar todos os códigos para termos completude e também para mostrarmos a sintaxe que usa o conceito de registro.

Para inicializar um conjunto é suficiente definir o tamanho do vetor como sendo zero, conforme a figura 10.5. Isto torna trivial testar se um conjunto é vazio (figura 10.6) e saber a cardinalidade do conjunto (figura 10.7), as informações dependem apenas do tamanho do vetor. A função que verifica se um elemento pertence a um conjunto, mostrada na figura 10.9 usa uma função interna ao TAD conjunto para fazer uma busca binária, conforme ilustrado na figura 10.8. Fizemos esta função separadamente por dois motivos: (1) ela também é usada na função de retirar um elemento; e (2) para mostrar que na implementação de um TAD podemos ter funções e procedimentos que não são públicos, são de uso interno da estrutura.

¹Este tipo de alocação não será coberto neste livro.
10.1. TIPO ABSTRATO DE DADOS CONJUNTO

**Figura 10.5:** Inicializar um conjunto.

```pascal
procedure inicializar_conjunto (var c: conjunto);
// Cria um conjunto vazio. Deve ser chamado antes de qualquer operação no conjunto.
// Custo: constante.
begin
  c.tam := 0;
end;
```

**Figura 10.6:** Testa se um conjunto é vazio.

```pascal
function conjunto_vazio (c: conjunto): boolean;
// Retorna true se o conjunto c é vazio e false caso contrário.
// Custo: constante.
begin
  conjunto_vazio := c.tam = 0;
end;
```

**Figura 10.7:** Encontra a cardinalidade do conjunto c.

```pascal
function cardinalidade (c: conjunto): longint;
// Retorna a cardinalidade do conjunto c.
// Custo: constante.
begin
  cardinalidade := c.tam;
end;
```

**Figura 10.8:** Função interna ao TAD: busca binária.

```pascal
function busca_binaria (x: longint; c: conjunto): longint;
// Retorna o índice do vetor que contém x, senão retorna zero.
// Custo: proporcional ao logaritmo do tamanho do conjunto.
var ini, fim, meio: longint;
beg
  ini := 1;
  fim := c.tam;
  meio := (ini + fim) div 2;
  while (ini <= fim) and (x <> c.v[meio]) do
    begin
      if x < c.v[meio] then
        fim := meio - 1
      else
        ini := meio + 1;
      meio := (ini + fim) div 2;
    end;
  if fim < ini then
    busca_binaria := 0
  else
    busca_binaria := meio;
end;
```
**CAPÍTULO 10. TIPOS ABSTRATOS DE DADOS**

```pascal
function pertence (x: longint; c: conjunto): boolean;
// Retorna true se x pertence ao conjunto c e false caso contrário.
// Custo: proporcional ao logaritmo do tamanho do conjunto.
begin
  // a função busca_binaria retorna o índice do vetor que contém x
  // se x não está no vetor, retorna zero.
  pertence := busca_binaria (x, c) <> 0;
end;
```

Figura 10.9: Tenta encontrar um elemento no conjunto usando busca binária.

As procedures para inserção e remoção, ilustradas nas figuras 10.10 e 10.11 têm custo linear. Em ambos é preciso deslocar todo os elementos do vetor no pior caso. Para inserir, é preciso encontrar o ponto de inserção, para garantir a ordenação, ao mesmo tempo deslocando todos para frente criando o espaço para o novo elemento. Para remover, antes é necessário encontrar o elemento a ser removido no vetor para em seguida deslocar para trás todos os elementos subsequentes.

```pascal
procedure inserir_conjunto (x: longint; var c: conjunto);
// Insere o elemento x no conjunto c, mantém os elementos ordenados.
// Custo: para garantir o conjunto ordenado, proporcional ao tamanho do conjunto.
var
  i: longint;
begin
  if not pertence (x, c) then
  begin
    c.v[0]:= x; // sentinela
    i:= c.tam;
    while x < c.v[i] do // procura ponto de insercao abrindo espaço
    begin
      c.v[i+1]:= c.v[i];
      i:= i - 1;
    end;
    c.v[i+1]:= x;
    c.tam:= c.tam + 1;
  end;
end;
```

Figura 10.10: Procedure para inserir elemento em conjunto ordenado.

As figuras 10.12 e 10.13 mostram como são implementadas as funções iniciar_proximo e incrementar_proximo, respectivamente.
10.1. TIPO ABSTRATO DE DADOS CONJUNTO

```pascal
procedure remover_conjunto (x: longint; var c: conjunto);
// Remove o elemento x do conjunto c.
// Custo: para garantir o conjunto ordenado, proporcional ao tamanho do conjunto.
var i, indice: longint;
begin
  indice:= busca_binaria(x,c); // primeiro acha a posicao do elemento
  if indice <> 0 then // achou o elemento
    begin // puxa os elementos para tras
      for i:= indice to c.tam-1 do
        c.v[i]:= c.v[i+1];
      c.tam:= c.tam - 1;
    end;
end;
```

Figura 10.11: Procedure para remover elemento em conjunto ordenado.

```pascal
procedure iniciar_proximo (var c: conjunto);
// Inicializa o contador que sera usado na funcao incrementar_proximo.
// Custo: constante.
begin
  c.proximo:= 1;
end;
```

Figura 10.12: Inicializa o contador próximo.

O código para o incremento foi feito como uma função que retorna `false` quando o contador chegou ao fim do vetor e devolve o elemento usando-se um parâmetro por referência. Os algoritmos que vão usar esta função devem saber quando o contador chegou ao final e devem chamar o iniciador do contador quando precisarem percorrer o conjunto novamente.

A função `retirar_um_elemento` é útil para se manipular conjuntos, ela permite a retirada de um elemento qualquer. Optamos por retirar o último elemento, mantendo o conjunto ordenado de modo simples. O código está na figura 10.14. Ela foi pensada para ser usada em combinação com o teste de conjunto vazio para percorrer todo o vetor. A diferença desta para a combinação que usa o contador é que ela retira os elementos do conjunto, enquanto que com o contador o conjunto permanece inalterado.

A última função serve para testar se dois conjuntos são iguais e é apresentada na figura 10.15. O algoritmo é eficiente, linear, pois os vetores estão ordenados.
function incrementar_proximo (var c: conjunto; var x: longint): boolean;
// Incrementa o contador e retorna x por referencia. Retorna false se acabou conjunto
begin
  if c.proximo <= c.tam then
    begin
      x:= c.v[c.proximo];
      c.proximo:= c.proximo + 1;
      incrementar_proximo:= true;
    end
  else
    incrementar_proximo:= false;
end;

Figura 10.13: Incrementa o contador próximo.

function retirar_um_elemento (var c: conjunto): longint;
// Escolhe um elemento qualquer do conjunto para ser removido e o remove
// Custo: constante, pois optamos por devolver o ultimo elemento.
begin
  retirar_um_elemento:= c.v[c.tam];
  c.tam:= c.tam - 1;
end;

Figura 10.14: Retira um elemento qualquer do conjunto.

function sao_iguais (c1, c2: conjunto): boolean;
// Retorna true se o conjunto c1 = c2 e false caso contrario.
// Custo: dada a ordenacao, linear.
var i: longint;
begin
  if c1.tam <> c2.tam then
    sao_iguais:= false
  else
    begin
      i:= 1;
      while (i <= c1.tam) and (c1.v[i] = c2.v[i]) do
        i:= i + 1;
      if i <= c1.tam then
        sao_iguais:= false
      else
        sao_iguais:= true;
    end;
end;

Figura 10.15: Verifica se dois conjuntos são iguais em vetores ordenados.
O algoritmo para união também leva em conta que os vetores estão ordenados e usa uma fusão dos vetores de modo bem eficiente, conforme a figura 10.16.

```pascal
function uniao (c1, c2: conjunto): conjunto;
    // Obtém a união dos conjuntos c1 e c2.
    // Custo: como estão ordenados, proporcional a soma dos tamanhos dos vetores
var
    i, j, k, l: longint; uni: conjunto;
begin
    inicializar_conjunto (uni);
    i:= 1; j:= 1; k:= 0;
    while (i <= c1.tam) and (j <= c2.tam) do
    begin
        k:= k + 1;
        if c1.v[i] < c2.v[j] then // avança apontador do primeiro vetor
            begin
                uni.v[k]:= c1.v[i]; i:= i + 1;
            end
        else if c1.v[i] > c2.v[j] then // avança apontador do segundo vetor
            begin
                uni.v[k]:= c2.v[j]; j:= j + 1;
            end
        else // descarta um dos repetidos e avança os dois apontadores
            begin
                uni.v[k]:= c1.v[i];
                i:= i + 1; j:= j + 1;
            end;
    end;
    for l:= i to c1.tam do
    begin
        k:= k + 1; uni.v[k]:= c1.v[l];
    end;
    for l:= j to c2.tam do
    begin
        k:= k + 1; uni.v[k]:= c2.v[l];
    end;
    uni.tam:= k;
    uniao:= uni;
end;
```

Figura 10.16: União de conjuntos usando fusão.

Usamos dois apontadores para os vetores de entrada, como os vetores estão ordenados, comparamos os elementos apontados por i e j. O menor deles é copiado no vetor união e seu respectivo apontador é incrementado. Se os elementos são iguais, apenas um deles é copiado, mas os dois apontadores são incrementados. O contador k serve para controlar o vetor união. Se um dos vetores terminar antes do outro um for final termina a cópia para o novo vetor. Observe que somente um for será executado pela semântica dele na linguagem Pascal. O custo é linear pois os dois vetores são percorridos exatamente uma vez.
A intersecção, também muito eficiente pois explora a ordenação, é mostrada na figura 10.17.

```pascal
function interseccao (c1, c2: conjunto): conjunto;
// Obtém a interseccao dos conjuntos c1 e c2.
// Custo: como estão ordenados, proporcional ao tamanho do vetor c1.
// Possível modificação: o custo pode ser proporcional ao tamanho do menor conjunto.
var i, j, k: longint; intersec: conjunto;
begin
  inicializar_conjunto (intersec);
  i := 1;
  j := 1;
  k := 0;
  while (i <= c1.tam) and (j <= c2.tam) do
    if c1.v[i] < c2.v[j] then
      i := i + 1
    else
      if c1.v[i] > c2.v[j] then
        j := j + 1
      else // elemento está nos dois conjuntos
        begin
          k := k + 1;
          intersec.v[k] := c1.v[i];
          i := i + 1;
          j := j + 1;
        end;
  intersec.tam := k;
  interseccao := intersec;
end;
```

Figura 10.17: Intersecção de conjuntos.

Usamos novamente dois apontadores para os vetores de entrada, explorando novamente a ordenação. Comparamos os elementos apontados por i e j e inserimos uma das cópias no vetor intersecção quando eles são iguais. Neste caso ambos os apontadores são incrementados. Quando um é menor do que o outro significa que ocorrem em um conjunto somente e apenas um dos apontadores é incrementado. O custo também é linear pois os dois vetores são percorridos exatamente uma vez.
A figura [10.18] contém o código da operação de diferença entre conjuntos.

```plaintext
function diferenca (c1, c2: conjunto): conjunto;
// Obtém a diferença dos conjuntos c1 e c2 (c1 − c2).
// Custo: linear por causa da ordenação.
var i, j, k: longint; dif: conjunto;
begin
  inicializar_conjunto (dif);
  i:= 1; j:= 1; k:= 0;
  while (i <= c1.tam) and (j <= c2.tam) do
    if c1.v[i] < c2.v[j] then
      begin
        k:= k + 1;
        dif.v[k]:= c1.v[i];
        i:= i + 1;
      end
    else if c1.v[i] > c2.v[j] then
      j:= j + 1
    else // são iguais
      begin
        i:= i + 1;
        j:= j + 1;
      end;
  // se ainda tem elementos em c1, copia
  for j:= i to c1.tam do
    begin
      k:= k + 1;
      dif.v[k]:= c1.v[j];
    end;
  dif.tam:= k;
  diferenca:= dif;
end;
```

Figura 10.18: Diferença entre conjuntos.

Mais uma vez a estratégia de usar os dois apontadores para os vetores de entrada foi utilizada. Desta vez o primeiro vetor deve ser percorrido na busca de elementos que sejam menores do que os do segundo. São os elementos que ocorrem apenas no primeiro e devem ser copiados no vetor diferença. Os outros são descartados. Caso o segundo vetor termine antes, os elementos do primeiro vetor são copiados para o destino. O custo é linear pois os dois vetores são percorridos exatamente uma vez.

Da mesma forma, a função que testa se um conjunto está contido em outro, apresentada na figura [10.19] também explora a ordenação do conjunto e é bastante eficiente. O objetivo é encontrar todos os elementos do primeiro no segundo. O custo é linear: os dois vetores são percorridos uma única vez.

Uma falha ocorre quando um elemento do primeiro vetor é menor do que o do segundo, significando que ele não está presente nesse ou quando chegamos no final do segundo vetor antes de chegar no final do primeiro.

A figura [10.20] mostra como copiar um conjunto em outro.
function contido (c1, c2: conjunto): boolean;
// Retorna true se o conjunto c1 está contido no conjunto c2 e false caso contrário.
// Custo: proporcional ao tamanho do conjunto c1.
var i, j: longint; ok: boolean;
begin
  if c1.tam > c2.tam then
    contido := false
  else
    begin
      contido := false; ok := true; i := 1; j := 1;
      while (i <= c1.tam) and (j <= c2.tam) and ok do
        if c1.v[i] < c2.v[j] then
          ok := false
        else if c1.v[i] > c2.v[j] then
          j := j + 1
        else
          begin
            i := i + 1; j := j + 1;
          end;
      if ok and (i > c1.tam) then
        contido := true
    end;
end;

Figura 10.19: Verifica se um conjunto está contido em outro.

function copiar_conjunto (c1: conjunto): conjunto;
// Copia os elementos do conjunto c1 para o conjunto c2.
// Custo: proporcional ao tamanho do conjunto c1.
var i: longint; c2: conjunto;
begin
  inicializar_conjunto (c2);
  c2.tam := c1.tam;
  for i := 0 to c1.tam do
    c2.v[i] := c1.v[i];
  copiar_conjunto := c2;
end;

Figura 10.20: Copia um conjunto em outro.
Implementação 2

Nesta implementação vamos usar um vetor com elementos não ordenados. Detalharemos apenas as funções e procedimentos que se alteram, sem necessariamente mostrar todos os códigos.

As seguintes funções e procedimentos não se alteram pelo fato dos elementos não estarem ordenados:

- inicializar_conjunto
- conjunto_vazio
- cardinalidade
- copiar_conjunto
- iniciar_proximo
- incrementar_proximo
- retirar_um_elemento

A função pertence não pode mais usar busca binária, pois perdemos a ordenação. A figura [10.21] mostra o código que usa uma busca simples com sentinela. Com isso o custo foi alterado, na implementação 1 era logarítmico, agora é linear, portanto mais cara.

```plaintext
function pertence (x: longint; c : conjunto): boolean;
// Retorna true se x pertence ao conjunto c e false caso contrário.
// Custo: proporcional ao tamanho do conjunto, isto é, linear.
var
  i : longint;
begin
  pertence:= true;
  c.v[c.tam+1]:= x;
  i:= 1;
  while c.v[i] <> x do
    i:= i + 1;
  if i > c.tam then
    pertence:= false;
end;
```

Figura 10.21: Tenta encontrar um elemento no conjunto usando busca com sentinela.

```pascal
function uniao (c1, c2: conjunto) conjunto;
// Obtém a união dos conjuntos c1 e c2.
// Custo: quadrático por causa do teste de pertinência.
var i: longint; uni: conjunto;
begin
  inicializar_conjunto (uni);
  for i:= 1 to c1.tam do 
    inserir_conjunto (c1.v[i], uni);
  uni.tam:= c1.tam;
  for i:= 1 to c2.tam do 
    inserir_conjunto (c2.v[i], uni);
  uniao:= uni;
end;
```

Figura 10.22: União de conjuntos com vetores não ordenados.


```pascal
function interseccao (c1, c2: conjunto) : conjunto;
// Obtém a intersecção dos conjuntos c1 e c2.
// Custo: quadrático, por causa do teste de pertinência.
var i: longint; intersec: conjunto;
begin
  inicializar_conjunto (intersec);
  for i:= 1 to c1.tam do 
    if pertence (c1.v[i],c2) then 
      inserir_conjunto (c1.v[i],intersec);
  interseccao:= intersec;
end;
```

Figura 10.23: Intersecção de conjuntos com vetores não ordenados.

O algoritmo da diferença de conjuntos é outro que teve seu custo aumentado de linear para quadrático. Para cada elemento do primeiro conjunto é preciso saber, a custo linear, se ele pertence ao outro, o que resulta no custo total quadrático. A figura 10.24 mostra este código.
10.1. TIPO ABSTRATO DE DADOS CONJUNTO

function diferença (c1, c2: conjunto): conjunto;
// Obtém a diferença dos conjuntos c1 e c2 (c1 − c2).
// Custo: quadrático, por causa do teste de pertinência.
var i: longint; dif: conjunto;
begin
    inicializar_conjunto (dif);
    for i:= 1 to c1.tam do
        if not pertence (c1.v[i],c2) then // insere na diferença
            inserir_conjunto (c1.v[i],dif);
    diferenca:= dif;
end;

Figura 10.24: Diferença entre conjuntos com vetores não ordenados.

Da mesma forma, a função que testa se um conjunto está contido em outro, apresentada na figura[10.25], também teve seu custo alterado de linear para quadrático pela perda da garantida de ordenação.

Para cada elemento do primeiro conjunto devemos testar se ele existe no segundo. Como isto tem custo linear para cada elemento, o custo total é quadrático.

function contido (c1, c2: conjunto): boolean;
// Retorna true se o conjunto c1 está contido no conjunto c2 e false caso contrário.
// Custo: quadrático, por causa do teste de pertinência.
var i: longint;
    ok: boolean;
begin
    if c1.tam > c2.tam then
        contido:= false
    else
        begin
            i:= 1;
            ok:= true;
            while ok and (i <= c1.tam) do
                begin
                    if not pertence (c1.v[i],c2) then
                        ok:= false;
                    i:= i + 1;
                end;
            contido:= ok;
        end;
end;

Figura 10.25: Verifica se um conjunto está contido em outro com vetores não ordenados.

As procedures para inserção e remoção, ilustradas nas figuras [10.26] e [10.27], continuam com custos lineares, no primeiro caso por conta do teste de pertinência. Como os conjuntos não podem ter elementos repetidos é necessário testar antes. No código anterior a busca era binária, portanto logarítmica, mas desta vez ela é linear.

O algoritmo da inserção explora a não ordenação e insere no final do vetor, en-
quanto que a procedure de remoção ainda temos que encontrar a posição do elemento a ser removido, que tem custo linear. Uma vez encontrada, basta copiar o último neste lugar.

```
procedure inserir_conjunto (x: longint; var c: conjunto);
// Insere o elemento x no conjunto c, na ultima posição do vetor.
// Custo: linear por causa do teste de pertinência.
var i: longint;
begin
  if not pertence (x, c) then
    begin
      c.tam:= c.tam + 1;
      c.v[c.tam]:= x;
    end;
end;
```

Figura 10.26: Procedure para inserir elemento em conjunto não ordenado.

```
procedure remover_conjunto (x: longint; var c: conjunto);
// Remove o elemento x do conjunto c.
// Custo: linear, tanto por causa
var i, indice : longint;
begin
  indice:= 1; // primeiro acha a posição do elemento
  c.v[c.tam+1]:= x; // sentinela
  while x <> c.v[indice] do
    indice:= indice + 1;
  if indice > c.tam + 1 then // achou o elemento
    begin
      c.v[indice]:= c.v[c.tam]; // sobrecreve o último neste lugar.
      c.tam:= c.tam - 1;
    end;
end;
```

Figura 10.27: Procedure para remover elemento em conjunto não ordenado.

O algoritmo que testa igualdade de conjunto no caso não ordenado é bem mais caro do que o anterior. Ele tem que testar se o primeiro está contido no segundo e se o segundo está contido no primeiro, conforme a figura 10.28.

Conclusão: esta segunda implementação é bastante pior do que a primeira. Ela serviu apenas para mostrarmos que um TAD pode ser implementado de mais de uma maneira. Quando os estudantes aprenderem alocação dinâmica, poderão exercitar outras implementações, quem sabe mais eficientes do que as nossas.
### 10.2 TIPO ABSTRATO DE DADOS PILHA

Um pilha é provavelmente o Tipo Abstrato de Dados mais simples que podemos pensar, mas por outro lado é extremamente útil em uma grande variedade de problemas. Podemos pensar em uma pilha de pratos ou em uma pilha de cartas de baralho. Nestas pilhas, o que interessa é o elemento que está no topo, não temos acesso aos elementos abaixo dele. Quando empilhamos pratos, colocamos o próximo prato na pilha sobre o último. Quando retiramos cartas de um monte de baralho, sempre retiramos a carta que está por cima de todas as outras.

Uma pilha é definida pelo seu topo, isto é, pelo elemento que está por cima de todos os outros. Vejamos estes dois exemplos de pilhas de números:

<table>
<thead>
<tr>
<th>pilha 1</th>
<th>pilha 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
</tr>
</tbody>
</table>

A primeira tarefa é definir qual é o início da pilha, isto é, se vamos encarar as pilhas de baixo pra cima ou de cima pra baixo. No exemplo acima, o topo da pilha da esquerda é 2 ou é 4? Nós vamos definir agora que nossa visão é de que o topo é o 2. Assim, para efeitos lógicos, as duas pilhas acima tem o topo igual ao elemento 2. Pensando na pilha de cartas de baralho, não importa o tamanho do monte, o jogador que tirar a carta vai ter que se virar com um 2.

É comum pensarmos em uma pilha como uma estrutura na qual o último elemento que entra é o primeiro que sai, por isso são por vezes denominadas de LIFO, do inglês *Last In, First Out*.

As pilhas são amplamente utilizadas em computação para uma série de algoritmos importantes. Normalmente ela funciona como uma espécie de memória de alguma decisão que tomamos mas que pode ser revertida.
Vamos imaginar que queremos sair de um labirinto. Uma estratégia é: sempre que tivermos uma bifurcação, tomamos uma decisão e lembramos dela empilhando esta decisão na pilha. Após uma série de decisões chegamos em um beco sem saída. Voltamos atrás até a última bifurcação e para não repetirmos o mesmo erro, desempilhamos o topo da pilha e tomamos o caminho contrário.

Pilhas são usadas também por compiladores, por exemplo o do Pascal: cada `begin` que foi digitado deve ter um respectivo `end`. Então, quando o compilador encontra um `begin` ele empilha. Quando encontra um `end`, desempilha. O `end` não pode ocorrer se a pilha estiver vazia, pois significa que não houve um `begin` antes deste `end`.

Pilhas também podem ser utilizadas para lembrarmos de fazer um certo processo futuro em alguns elementos que não podemos fazer agora. Enfim, o uso é variado mas quase sempre significativo.

Com esta motivação, vamos considerar a existência de um tipo denominado `pilha`, mas, novamente, neste momento não estamos interessados em saber como este tipo é de fato implementado. Sabemos que é uma pilha, e isso nos é suficiente no momento. Declaramos então uma pilha de números inteiros:

```pascal
var p: pilha;
```

Primeiramente, vamos definir as interfaces, deixando a implementação para um segundo momento. Por interfaces queremos dizer qual é o comportamento das operações que manipulam o TAD pilha.

As únicas operações sobre pilhas são definidas em termos dos protótipos dos procedimentos e funções abaixo.

```pascal
procedure inicializar_pilha (var p: pilha);
// Cria uma pilha vazia.

function pilha_vazia (p: pilha): boolean;
// Retorna true se a pilha está vazia e false caso contrário.

function pilha_unitaria (p: pilha): boolean;
// Retorna true se a pilha tem um único elemento.

procedure empilhar (x: longint; var p: pilha);
// Insere x no início da pilha.

function desempilhar (var p: pilha): longint;
// Retorna o elemento do início da pilha e o remove dela.

function topo (p: pilha): longint;
// Retorna o elemento do início da pilha, sem remove-lo.
```

A função `pilha_unitaria` não é padrão. Mas nós entendemos que em alguns algoritmos é útil saber se a pilha está para acabar ou não. Pense no jogo de baralho, a decisão de um jogador pode ser diferente se ele percebe que a pilha de cartas vai acabar na jogada dele. Por isso manteremos esta função no nosso TAD.
10.2.1 Usando o TAD Pilha para resolver um problema

Nesta seção resolveremos um problema que pode ser elegantemente modelado e resolvido usando-se o TAD pilha.

Verificando expressões bem formadas com parênteses

Neste problema vamos estudar como usar pilhas para decidir se expressões com vários tipos de “parênteses” estão bem formadas.

A título de exemplo, considere que temos três tipos de “abra parênteses”: (, { e [, e três tipos de “fecha parênteses”: ), } e ].

Chamamos de expressões com parênteses bem formada se um mesmo tipo de parênteses aberto é fechado corretamente após a sua ocorrência, mas sem que haja um outro parênteses aberto que não foi fechado entre eles.

Por exemplo, as três expressões seguintes estão bem formadas:

- ({{}))(((){})
- ({{()}})
- {{}}(())][]

Por outro lado, as seguintes não estão:

- (}
- }{
- }{}

Para facilitar a programação vamos considerar que os tipos de parênteses são números que podem ser lidos do teclado. Assim, ( = 1, [ = 2 e { = 3 e assim por diante. Os fecha parênteses são os mesmos números na sua forma negativa. Assim: ) = -1, ] = -2 e } = -3.

O problema: receber uma sequência de valores do teclado, esta sequência é terminada em zero para indicar o término da leitura dos dados. Os valores lidos são números entre –N e N, onde N é o número máximo de tipos de parênteses. Decidir se a expressão é bem formada ou não.

Como o TAD pilha pode nos ajudar? É de fato muito simples: a cada valor lido, se ele for positivo empilhamos ele. Isto servirá para guardar a ordem em que os parênteses foram abertos. Mas se o número lido for negativo significa que temos que saber se ele está fechando o parênteses do mesmo tipo dele ou então se estamos tentando fechar um parênteses sem ele ter sido aberto.

Então temos que desempilhar o topo e comparar os tipos: eles tem que ter o mesmo valor em módulo! A figura [10.29] mostra este código. Ao final, se a expressão lida estiver bem formada com a pilha vazia significa que está tudo certo. Se a pilha estiver com algum elemento é porque sobram alguns parênteses abertos que não foram fechados.
program testa_parenteses_bem_formados;
// Aqui definimos o TAD pilha, ainda com implementacao desconhecida.
// Esta definicao vem juntamente com as funcoes e procedimentos acima definidos.
var p: pilha;
  n, x: longint;
  bem_formada: boolean;
begin
  inicializar_pilha (p);
  bem_formada:= true;
  read (n);
  while (n <> 0) and bem_formada do // pode parar assim que encontrar erro
    begin
      if n > 0 then
        empilhar (n,p)
      else if pilha_vazia (p) or (-n <> desempilhar (p)) then
        bem_formada:= false;
        read (n);
    end;
  if bem_formada and pilha_vazia(p) then
    writeln ('a expressao lida eh bem formada')
  else
    writeln ('a expressao lida nao eh bem formada')
end;

Figura 10.29: Programa que testa se os parênteses lidos são bem formados.
Encontrando celebridades usando o TAD pilha

Vamos retomar o problema de encontrar celebridades, visto na seção 10.1.1. Ele pode também ser resolvido usando-se o TAD pilha.

A implementação é muito similar, conforme podemos ver na figura 10.30, mas a diferença principal é na eficiência da implementação, conforme veremos na seção seguinte. Lembrando que, inicialmente, todos os convidados da festa são empilhados e apenas em seguida a função é chamada. O teste final mostrado na figura 10.4 é o mesmo.

![Figura 10.30: Encontrando uma celebridade usando pilha.](image)

### 10.2.2 Implementações do TAD pilha

Implementar o TAD pilha sem usar alocação dinâmica requer o uso de vetores. Como ordenação aqui não faz sentido, só existem duas maneiras de fazer esta implementação: quando empilhar insere o elemento no final, e portanto retira do final; ou então quando empilhar insere no início do vetor, e portanto retirando do início.

A segunda maneira é extremamente custosa, pois devemos movimentar os elementos do vetor a cada inserção ou remoção, com custo linear, o que não é aceitável computationalmente, embora possa ser implementado.

Nesta seção mostramos apenas uma forma de implementação de pilhas com vetores, a que insere e remove do final. Quando o estudante aprender alocação dinâmica ele poderá fazer diferente.

### Implementação

```plaintext
const MAX = 101;

type
  pilha = record
    ini: longint;
    v: array [1..MAX] of longint;
  end;
```
Conforme explicado, o principal é decidir qual é o início da pilha. Nós optamos pela última posição útil do vetor.

Agora podemos implementar todas as funções e procedimentos, cada um deles com sua respectiva análise de custo.

Para inicializar uma pilha, da mesma forma como foi feito para o TAD conjunto, é suficiente definir \texttt{p.ini} como sendo zero. O código é mostrado na figura 10.31. A operação têm custo constante\textsuperscript{3}

```plaintext
procedure inicializar_pilha (var p: pilha);
// Inicializa a estrutura.
// Custo: constante.
begin
    p.ini:= 0;
end;
```

Figura 10.31: Cria uma pilha vazia.

Nossa decisão torna trivial o teste de pilha vazia, é só testar se \texttt{p.ini} é zero ou não, conforme a figura 10.32 ilustra. Esta operação também tem custo constante.

```plaintext
function pilha_vazia (p: pilha) : boolean;
// Retorna \texttt{true} se a pilha esta vazia e \texttt{false} caso contrario.
// Custo: constante.
begin
    pilha_vazia:= p.ini = 0;
end;
```

Figura 10.32: Testa se uma pilha é vazia.

Também trivial é a função mostrada na figura 10.33, já que, pela nossa implementação, o tamanho da pilha coincide com o valor de \texttt{p.ini}, fazendo que que o custo seja também constante.

```plaintext
function pilha_unitaria (p: pilha) : boolean;
// Retorna \texttt{true} se a pilha tem um único elemento e \texttt{false} caso contrario.
// Custo: constante.
begin
    pilha_unitaria:= p.ini = 1;
end;
```

Figura 10.33: Retorna se a pilha tem um único elemento.

Empilhar um elemento é muito fácil e barato: inserimos o elemento após a última posição do vetor, com custo constante conforme podemos ver na figura 10.34.

\textsuperscript{3}Custo constante significa que não há laços no algoritmo, ele é constituído somente de um número fixo de operações triviais como atribuições e testes simples
10.2. TIPO ABSTRATO DE DADOS PILHA

**procedure** empilhar (x: longint; var p: pilha);
// Insere x no início da pilha.
// Custo: constante.
begin
    p.ini:= p.ini + 1;
    p.v[p.ini]:= x;
end;

Figura 10.34: Insere um elemento na pilha.

Desempilhar é tão trivial quanto empilhar, basta retornar o valor do último elemento e acertar o valor de `p.ini`. Isto também tem custo constante.

**function** desempilhar (var p: pilha): longint;
// Retorna o elemento do início da pilha e o remove dela.
// Custo: constante.
begin
    desempilhar:= p.v[p.ini];
    p.ini:= p.ini - 1;
end;

Figura 10.35: Remove, e retorna, o elemento do topo da pilha.

A última função é muito similar à anterior, a diferença é que o elemento não é removido, também com custo constante, conforme mostrado na figura 10.36.

**function** topo (p: pilha): longint;
// Retorna o elemento do início da pilha, sem removê-lo.
// Custo: constante.
begin
    topo:= p.v[p.ini];
end;

Figura 10.36: Retorna o elemento do topo da pilha, sem removê-lo.

Em resumo, todas as seis operações sobre o TAD *pilha* tem custo constante.

Observação: Não fizemos nenhuma checagem de consistência do vetor no caso dele estar cheio ou vazio. O estudante pode modificar os códigos para que isto seja feito, por exemplo, ao desempilhar um elemento, o vetor não pode estar vazio. Similar para empilhar, o vetor não pode estar cheio, e assim por diante. Uma excelente apresentação de Tipos Abstratos de Dados pode ser encontrada em [TLA95]. Os mesmos argumentos valem para o TAD conjunto.

Para encerrarmos este capítulo um comentário merece ser feito. A implementação do TAD *conjunto* usando o vetor ordenado é de fato mais eficiente do que a outra implementação, mas é preciso lembrar que a cada inserção é preciso saber se o elemento
se repete e isso certamente gera um custo. Por isso a implementação do problema da celebidade é melhor quando se usa uma pilha, pois este teste adicional não é preciso ser feito. A pilha admite repetição de elementos, mas no caso deste problema os elementos não se repetem por garantia do próprio enunciado.
10.3 Exercícios

1. Defina em *Pascal* um Tipo Abstrato de Dados *racional* que vai procurar abstrair números da forma $\frac{A}{B}$, onde $A$ e $B$ são números inteiros. Em seguida implemente as funções e procedimentos necessários para realizar as operações abaixo de maneira que os resultados sejam sempre simplificados, isto é, o resultado final deve indicar $\frac{3}{2}$ e não $\frac{6}{4}$, por exemplo. Suas implementações não podem permitir que ocorram divisões por zero.

   (a) adição de duas frações;
   (b) subtração de duas frações;
   (c) multiplicação de duas frações;
   (d) divisão de duas frações.

2. Sem olhar os códigos escritos neste capítulo, implemente em *Pascal* sua própria biblioteca que implementa o TAD pilha. Os algoritmos que estão ali são triviais mas servem para ver sua prática em programação.

3. Sem olhar os códigos escritos neste capítulo, implemente em *Pascal* sua própria biblioteca que implementa o TAD conjunto. Alguns dos algoritmos que estão ali não são triviais e é um ótimo exercício de programação.

4. Considere um Tipo Abstrato de Dados *Fila*, que define uma estrutura na qual o primeiro que entra é o primeiro que sai (FIFO, do inglês *First In, First Out*). Pense em uma fila de uma padaria, por exemplo. As principais operações que manipulam filas são as seguintes:

   • criar uma fila (vazia);
   • inserir elementos;
   • remover elementos;
   • retornar o tamanho da fila;
   • saber se a fila está vazia/cheia;
   • imprimir a fila como está;
   • retornar o elemento do início da fila, removendo-o;
   • retornar o elemento do início da fila, sem removê-lo.

   Faça uma biblioteca em *Pascal* que implemente as operações acima.

5. Considere um Tipo Abstrato de Dados *lista*, que define uma estrutura que pode conter elementos em qualquer ordem ou mesmo repetidos. Pense por exemplo em uma lista de compras ou de afazeres. As principais operações que manipulam listas são as seguintes:

   • criar uma lista (vazia);
• inserir no início;
• inserir no fim;
• inserir na posição p;
• remover (do início, do fim, da posição p);
• retornar o tamanho da lista;
• saber se a lista está vazia/cheia;
• imprimir a lista como está;
• imprimir a lista ordenada;
• fundir duas listas;
• intercalar duas listas;
• pesquisar o elemento da posição p na lista;
• copiar uma lista em outra;

Faça uma biblioteca em *Pascal* que implemente as operações acima.

6. Faça um programa em *Pascal* que encontre bilhetes premiados do jogo da megassena, usando a sua implementação do tipo lista acima definido.

7. Defina em *Pascal* o Tipo Abstrato de Dados *polinômio*. A estrutura tem que ser capaz de armazenar o grau e os coeficientes de um polinômio $a_0 + a_1x + a_2x^2 + \ldots + a_nx^n$. Implemente o conjunto de operações abaixo na forma de funções e procedimentos:

   (a) inicializar a estrutura polinômio;
   (b) ler um polinômio de grau $n$;
   (c) dado $x \in \mathbb{R}$, calcular o valor de $P(x)$;
   (d) obter o polinômio derivada de $P$, $P'$.
   (e) dado $x \in \mathbb{R}$, calcular o valor de $P'(x)$;
   (f) obter o polinômio soma de $P$ e $Q$;
   (g) obter o polinômio multiplicação de $P$ e $Q$;

Faça um programa em *Pascal* que utilize o tipo abstrato de dados definido, leia dois polinômios $p$ e $q$, calcule o produto $r$ de $p$ e $q$, imprima o polinômio resultante, leia um certo número real $x$, calcule o valor de $r(x)$ e o imprima.

8. Resolva o exercício que está na seguinte URL:


9. Resolva o exercício que está na seguinte URL:


10. Resolva o exercício que está na seguinte URL:

Capítulo 11

Aplicações em jogos

Neste ponto já temos todas as noções fundamentais de algoritmos e estruturas de dados que são necessárias para a construção de programas complexos. O objetivo agora é mostrar como se desenvolver programas de maior porte que, não por acaso, envolvem implementar jogos.

11.1 Campo minado

O primeiro problema que vamos tentar resolver é o conhecido jogo do *Campo Minado*, disponível gratuitamente em diversos sistemas operacionais atuais. Uma descrição do funcionamento deste jogo pode ser encontrada na Wikipédia[^1]:

A área de jogo consiste num campo de quadrados retangular. Cada quadrado pode ser revelado clicando sobre ele, e se o quadrado clicado contiver uma mina, então o jogo acaba. Se, por outro lado, o quadrado não contiver uma mina, uma de duas coisas poderá acontecer:

- Um número aparece, indicando a quantidade de quadrados adjacentes que contêm minas;
- Nenhum número aparece. Neste caso, o jogo revela automaticamente os quadrados que se encontram adjacentes ao quadrado vazio, já que não podem conter minas;

O jogo é ganho quando todos os quadrados que não têm minas são revelados.

Opcionalmente, o jogador pode marcar qualquer quadrado que acredita que contém uma mina com uma bandeira, bastando para isso clicar sobre ele com o botão direito do mouse. Em alguns casos, carregar com ambos os botões do mouse num número que contenha tantas bandeiras imediatamente à sua volta quanto o valor desse número revela todos os quadrados sem minas que se encontrem adjacentes a ele. Em contrapartida, o jogo acaba se se efectuar essa acção quando os quadrados errados estiverem marcados com as bandeiras.

Implementaremos a versão ASCII do jogo, contudo ignorando as bandeiras que se podem colocar para marcação de prováveis minas. Isso ficará como exercício para o estudante.

Desenvolvermos o código final usando-se a técnica de refinamentos sucessivos, isto é, vamos partir de um código geral e a cada etapa detalharemos as estruturas de dados e algoritmos necessários para a resolução do problema. Para isto vamos explorar ao máximo as noções de funções e procedimentos da linguagem Pascal. Também usaremos a noção de Tipos Abstratos de Dados para facilitar a implementação, também deixando-a elegante.

Da mesma forma que nossa apresentação sobre Tipos Abstratos de Dados, iniciaremos considerando a existência de um TAD denominado campo_minado que será posteriormente definido segundo as nossas necessidades conforme o processo de refinamentos sucessivos avance. Mas podemos considerar no momento a existência de algumas funções e procedimentos que vão nos ajudar a iniciar por um código de programa principal que represente de forma geral a solução do problema do jogo Campo Minado. Este código está ilustrado na figura 11.1.

```pascal
program mines;
// aqui vem as declaracoes iniciais, no momento omitidas
var
c   : campo_minado; // usa o TAD campo_minado, a ser definido
x,y : longint; // coordenadas do quadrado que usuario clicou
begin (* programa principal *)
  inicializar_campo (c);
  imprimir_campo (c);
  repeat
    ler_jogada (x,y,c);
    executar_jogada (x,y,c);
    imprimir_campo (c);
  until ganhou (c) or perdeu (c);
  if perdeu (c) then
    writeln ('Voce perdeu...')
  else
    writeln ('Parabens, voce ganhou !!!');
end.
```

Figura 11.1: Programa principal para o jogo campo minado.

Este parece ser um bom código inicial para se resolver jogos do tipo tabuleiro com um único jogador. Se for necessário em refinamentos futuros outras funções e procedimentos poderão ser utilizados.

Esta implementação considera que a variável `c`, do tipo `campo_minado` contém todas as informações necessárias para a boa execução das funções e procedimentos. A parte crítica é a execução da jogada, que depende da correta inicialização da estrutura do campo minado. As funções `ganhou` e `perdeu` têm semântica óbvia, bem como a procedure de leitura das coordenadas escolhidas pelo jogador.

Desta forma consideramos os seguintes protótipos com suas respectivas semânticas:
11.1. CAMPO MINADO

A estrutura de dados campo_minado deve ser definida agora, porém, como ainda não sabemos tudo o que é necessário, ela pode ser modificada posteriormente. Parece intuitivo definir o campo minado como sendo uma matriz de inteiros, porém, um dos detalhes mais importantes da interface do jogo é que os quadrados não são todos exibidos, somente os que já foram clicados pelo usuário e eventualmente aqueles que foram expandidos quando se clica em quadrados com zero minas vizinhas.

Então a pergunta é: como representar adequadamente a estrutura de maneira a facilitar a procedure de impressão da matriz? Outra boa pergunta é: como representar em uma matriz de inteiros a informação a ser exibida? Isto é, alguns quadrados não são revelados e, quando são, uns mostram o número de minas ao seu redor e outros mostram um vazio, no caso daquele quadrado ter zero minas. Ainda há que se armazenar a informação se um quadrado contém uma mina.

Primeira decisão: Como representar o TAD campo_minado?

Usaremos um registro para um quadrado, que inicialmente conterá duas informações:

- uma delas vai armazenar a informação se aquele quadrado deve ou não ser revelado na tela.
- a outra conterá um número inteiro que significa:
  - -1: o quadrado contém uma mina;
  - número não negativo: é o número de minas ao redor deste quadrado.

Desta forma, se o quadrado contiver um zero, é porque seus vizinhos não têm minas. Se tiver 3, seus vizinhos têm três minas, e assim por diante. Se tiver um -1 é porque contém uma mina. A matriz é definida como tendo seus elementos do tipo deste registro. A figura 11.2 mostra a estrutura inicial.
CAPÍTULO 11. APLICAÇÕES EM JOGOS

const
MAX = 9; // tamanho máximo da matriz

type
quadrado = record
  info: longint; // ou tem mina ou o número de minas vizinhas
  revelado: boolean; // quando true não imprime o conteúdo
end;

matriz = array [1..MAX, 1..MAX] of quadrado;

Figura 11.2: Estrutura inicial para o retângulo do campo minado.

Próximas decisões: Como finalizar o TAD campo_minado?

Antes de continuar precisamos tomar algumas decisões:

• Como armazenar as dimensões do retângulo que definem a matriz?

• Como decidir se o jogador ganhou ou perdeu? Esta pergunta depende da próxima.

• Quantas minas serão distribuídas neste retângulo?

A primeira decisão é análoga àquela que tomamos nas implementações dos TADs conjunto e pilha. A matriz será encapsulada juntamente com suas dimensões.

Para decidir se o jogador ganhou: a regra diz que é quando todos os quadrados que não contêm minas foram revelados. Para não ter que percorrer a matriz com custo quadrático manteremos a informação de quantos quadrados restam a ser revelados. É para saber esta informação dependemos do número total de minas espalhadas no jogo.

O jogador perde se ele clicou em um quadrado que possui uma mina. Isto provavelmente será decidido na procedure executar_jogada. Para facilitar criaremos uma nova entrada no registro do TAD: a informação do status se ele ganhou ou perdeu. Em resumo temos a versão inicial do TAD campo_minado, que pode ser visto na figura 11.3.

campo_minado = record
  x,y: longint; // número de linhas e colunas
  total_minas: longint; // número total de minas no campo
  falta_abrir: longint; // número de quadrados que faltam abrir
  status: longint; // indica se o jogador ganhou ou perdeu o jogo
  m: matriz;
end;

Figura 11.3: Primeiro refinamento para o TAD
Assim o campo minado é um registro que contém uma matriz, as suas dimensões, o total de minas no campo, a informação de quantos quadrados faltam ser descobertos pelo jogador, além do status do jogo (ganhou ou perdeu).

Cada elemento da matriz é um registro que armazena o número de minas vizinhas ou o número -1, que representa a mina, e um campo booleano para se representar um quadrado que já foi revelado.

Algumas constantes são úteis em programas grandes, elas facilitam a leitura e também permitem modificações em versões futuras. Inicialmente utilizaremos estas aqui (além de \( \text{MAX} = 9 \)), já definida anteriormente:

```plaintext
const
  MINA = -1;
  NADA = 0;
  GANHOU = 1;
  PERDEU = 2;
  EM_ANDAMENTO = 3;
```

Consequências da decisão quanto ao TAD: Como as decisões sobre como representar o TAD implicam na implementação das funções e procedimentos?

Iniciaremos pela procedure que faz a inicialização da estrutura de dados: *inicializar_campo*. Este código é exibido na figura [11.5]

```plaintext
procedure inicializar_campo (var c: campo_minado);
// chamas as funções e procedimentos que iniciam a estrutura
begin
  write ('Defina numero de linhas (entre 1 e ', \text{MAX}','): '); readln (c.x);
  write ('Defina numero de colunas (entre 1 e ', \text{MAX}','): '); readln (c.y);
  write ('Defina numero de minas (entre 1 e ', c.x*c.y,'): '); readln (c.total_minas);
  c.falta_abrir:= c.x*c.y - c.total_minas; // para teste se ganhou jogo
  c.status:= EM_ANDAMENTO; // para inicializar variavel
  zerar_campo (c);
  gerar_minas (c);
  contar_vizinhos_com_mina (c);
end:
```

O procedimento inicialmente solicita ao jogador que defina as dimensões do retângulo e o número de minas que serão distribuídas, desta forma escolhendo a dificuldade do jogo. Com estas informações é possível calcular qual é a quantidade de quadrados que devem ser revelados para podermos determinar se o jogador ganhou. Como status, como não sabemos ainda se houve vitória ou derrota, inicializamos com uma informação diferente (\text{EM_ANDAMENTO}).
Finalmente, é necessário preparar a matriz, o que é feito em três etapas: (1) zerar toda a matriz, limpando a memória de possíveis lixos; (2) distribuir todas as minas pelo campo minado; e (3) contar o número de minas vizinhas de todos os quadrados.

Zerar o campo é necessário pois haverá uma distribuição de minas e a posterior contagem dos vizinhos que contém minas, logo, o programador deve inicializar todas as estruturas para que o programa seja robusto.

Antes de apresentar o código cabe uma consideração: a procedure que conta os vizinhos com minas deve se preocupar com a contagem dos quadrados que estão nas bordas da matriz. O problema é que estes quadrados não têm os oito vizinhos e o programa que faz a contagem deve conter uma quantidade grande de comandos if-then-else’s, tornando o código extenso.

Portanto, vamos modificar a estrutura da matriz prevendo uma borda adicional que será inicializada com valores apropriados e que facilitarão a contagem. Agora os índices iniciam-se em zero e terminam em \( \text{MAX}+1 \).

```plaintext
matriz = array [0..\text{MAX}+1,0..\text{MAX}+1] of quadrado;
```

Utilizaremos também uma nova constante para representar o valor da borda:

```plaintext
const \text{BORDA} = -2;
```

O procedimento para zerar o campo é apresentado na figura 11.6. Ele usa a função \text{eh_borda} (figura 11.7), tanto por legibilidade quanto por reaproveitamento de código, pois ela é utilizada mais de uma vez.

```plaintext
procedure zerar_campo (var c: campo_minado); // prepara o campo: define as bordas, zera o tabuleiro e inicia info revelado
var i,j: longint;
begin
  for i:= 1 to c.x do // zera a matriz e inicia revelado false
    for j:= 1 to c.y do
      begin
        if eh_borda (i,j,c) then
          c.m[i,j].info:= \text{BORDA}
        else
          c.m[i,j].info:= \text{NADA};
          c.m[i,j].revelado:= false;
      end;
  end;
end;
```

Figura 11.6: Zerar o campo minado.

Este procedimento percorre a matriz, zera o número de minas e informa que, inicialmente, nenhum quadrado do campo pode ser revelado ainda na impressão.
11.1. CAMPO MINADO

function eh_borda (i,j: longint; c: campo_minado): boolean;
// retorna true quando uma linha ou coluna eh uma borda
begin
  eh_borda:= (i = 0) or (i = c.x+1) or (j = 0) or (j = c.y+1);
end;

Figura 11.7: Testa se uma coordenada faz parte da borda.

Agora podemos distribuir as minas no campo. Elas são aleatoriamente distribuídas até o número máximo indicado pelo usuário, conforme a figura [11.8] no qual um laço garante que não serão geradas minas nos mesmos lugares.

procedure gerar_minas (var c: campo_minado);
// coloca o total de minas definido em posições aleatórias do campo
var i,x,y: longint;
begin
  randomize;
  for i:= 1 to c.total_minas do
  begin
    repeat
      x:= random (c.x) + 1;
      y:= random (c.y) + 1;
    until c.m[x,y].info = NADA; // garante mina em lugar vazio
    c.m[x,y].info:= MINA;
  end;
end;

Figura 11.8: Distribuindo as minas.

Neste ponto temos uma matriz com as minas distribuídas e com todas as outras posições zeradas. Podemos prosseguir com a contagem dos vizinhos.

O procedimento para contar vizinhos é, a princípio, simples. Basta varrer a matriz e para cada quadrado diferente de MINA percorrer todos os oito vizinhos e contar as minas. Podemos ver o efeito da adção da borda, pois ela facilitou imensamente o código desta função, que é mostrado na figura [11.9].

procedure contar_vizinhos_com_mina (var c: campo_minado);
// para os quadrados que não são minas, conta os vizinhos que são minas
var i, j: longint;
begin
  for i:= 1 to c.x do
    for j:= 1 to c.y do
      if c.m[i,j].info = NADA then // conta vizinhos com mina
        c.m[i,j].info:= num_vizinhos_com_mina (i,j,c);
end;

Figura 11.9: Contando vizinhos com minas.
Esta procedure faz uso de uma função (figura 11.10) que efetivamente conta as minas vizinhas para cada quadrado e foi usada para deixar o código legível e modular.

```pascal
function num_vizinhos_com_mina (x,y: longint; c: campo_minado): longint;
// para um quadrado valido conta as minas nos 8 vizinhos
// se entrou aqui eh porque o proprio quadrado nao tem mina
var i,j, cont: longint;
begin
  cont:= 0;
  for i:= x-1 to x+1 do
    for j:= y-1 to y+1 do
      if c.m[i,j].info = MINA then
        cont:= cont + 1;
  num_vizinhos_com_mina:= cont;
end;
```

Figura 11.10: Contando os vizinhos com minas para um quadrado.

Neste ponto temos um campo minado pronto para ser jogado!

Para ler uma coordenada escolhida pelo jogador optamos por uma interface bem simples, ela apenas garante uma leitura da linha e da coluna de maneira que seja um quadrado que ainda não foi revelado. O código é apresentado na figura 11.11.

```pascal
procedure ler_jogada (var x,y: longint; c: campo_minado);
// le uma jogada, nao testa consistencia, mas testa se eh quadrado novo.
begin
  repeat
    write ('Escolha a linha para jogar: ');
    readln (x);
    write ('Escolha a coluna para jogar: ');
    readln (y);
  until not c.m[x,y].revelado; // nao pode jogar no mesmo lugar novamente
end;
```

Figura 11.11: Lendo uma jogada.

Implementaremos uma versão muito simples da procedure de impressão do campo usando símbolos da tabela ASCII. O jogador deve poder identificar as coordenadas desejadas para jogar. Devemos escolher símbolos para imprimir um quadrado não revelado. Os revelados devem mostrar um símbolo qualquer para uma mina, quando for o caso, ou um número que representa a quantidade de minas vizinhas. Escolhemos os seguintes símbolos:

```pascal
const
  SIMBOLO_MINA = '@'; // representa uma mina
  SIMBOLO_SEM_MINA_VIZ = ' '; // representa quadrado sem minas vizinhas
  SIMBOLO_ESCONDIDO = '*'; // representa quadrado nao revelado
```

A figura 11.12 ilustra como queremos ver a tela em uma fase do jogo, enquanto que a figura 11.13 mostra como obtê-la.
11.1. CAMPO MINADO

Figura 11.12: Tela do jogo.

procedure imprimir_campo (c: campo_minado);
    // imprime o campo minado com base na info revelado, se o quadrado ja foi aberto
var i, j: longint;
begin
    clrscr;
    write (' ');
    for i:= 1 to c.x do
        write (i, ' ');
    writeln;
    write (' +');
    for i:= 1 to 2 * c.x do
        write ('-');
    writeln;
    for i:= 1 to c.x do
    begin
        write (i, '| ');
        for j:= 1 to c.y do
            if not c.m[i,j].revelado then
                write (SIMBOLO_ESCONCIDO) // nao exibe o quadrado
            else
                case c.m[i,j].info of
                    MINA: write (SIMBOLO_MINA);
                    NAIDA: write (SIMBOLO_SEM_MINA_VIZ);
                    else
                        write (c.m[i,j].info, ' ');
                    end;
        writeln;
    end;
end;

Figura 11.13: Imprimindo o campo minado.
CAPÍTULO 11. APLICAÇÕES EM JOGOS

Ela funciona apenas para retângulos de no máximo \( 9 \times 9 \). Mas o leitor já sabe que neste texto não primamos pela interface, mas pelos algoritmos que jogam o jogo. O comando `clrscr` faz parte da biblioteca CRT, que deve ser invocada com a chamada:

```plaintext
uses CRT;
```

Este comando limpa a tela antes de imprimir algo, causando a impressão de movimento. Na verdade, este efeito é obtido pela impressão da matriz no mesmo local de onde ela tinha sido impressa anteriormente, criando a ilusão.

Aqui terminamos a parte mais simples da implementação. O procedimento que executa a jogada é o mais complexo de todos a implementar pois ele é exatamente a essência do jogo.

Quando se revela um quadrado que tem zero minas vizinhas, toda sua vizinhança deve ser revelada também. O problema é que um destes vizinhos (ou mais de um) pode também contém um valor zero. Por isto usamos um procedimento que revela todos os vizinhos que têm zero minas e os vizinhos destes e assim por diante, até não sobrar mais nenhum (`executar_jogada`).

A solução mais elegante para este problema envolve a noção de `recursividade` que está fora do escopo deste curso, normalmente ela é vista em Algoritmos e Estruturas de Dados II.

A solução adotada foi, para cada coordenada que possui zero minas vizinhas, armazenar esta coordenada para ser revelada posteriormente em outro procedimento. Para isto usaremos o TAD Pilha, que é adequado para isso. Nossa pilha implementada no capítulo anterior é de inteiros mas agora precisamos que sejam empilhadas coordenadas. A modificação feita é essa:

```plaintext
type
  coordenada = record
    x,y: longint;
  end;

  pilha = record
    ini: longint;
    v: array [1..TAM_MAX_Pilha] of coordenada;
  end;
```

Todas as outras funções e procedimentos foram adaptadas e não serão mostradas aqui pois, como sabemos, o que interessa é o uso do TAD.

Nós separamos em duas procedures, a principal, que executa a jogada, é mostrada na figura [11.14]. O algoritmo testa se a coordenada escolhida é uma mina, neste caso o jogador perdeu. Caso contrário, revela o quadrado. Se este quadrado contém alguma mina vizinha, apenas revela este quadrado. Mas se ele contém zero minas vizinhas, ativa a função que revela todos os vizinhos sem minas, e para isto ele se limita a empilhar o quadrado escolhido pelo jogador e em seguida chama a função que revelará todos os outros.

Para efeitos visuais, quando o jogo é perdido uma procedure cuida de revelar todos os quadrados, para mostrar o campo minado ao jogador para ele se convencer que perdeu mesmo. Isto é mostrado na figura [11.15].
**11.1. CAMPO MINADO**

```pascal
procedure executar_jogada (x,y: longint; var c: campo_minado);
  // dada a coordenada (x,y) concretiza a jogada
var coord: coordenada;
  cont: longint;
  p: pilha;
begin
  if c.m[x,y].info = MINA then // achou mina...
     begin
       c.status:= DERROTA;
       revelar_tudo (c); // para revelado todo o campo no final
     end
  else // nao eh mina
     begin
       cont:= 1;
       c.m[x,y].revelado:= true;
       if c.m[x,y].info = NADA then // quadrado sem vizinhos com mina
          begin
            coord.x:= x;
            coord.y:= y;
            inicializar_pilha (p);
            empilhar (coord, p);
            cont:= abrir_vizinhos_sem_mina (c, p);
          end;
       c.falta_abrir:= c.falta_abrir - cont;
       if c.falta_abrir = 0 then // ganhou
          c.status:= VITORIA;
     end;
end;
```

Figura 11.14: Procedure que executa as regras do jogo.

```pascal
procedure revelar_tudo (var c: campo_minado);
  // usada quando o jogador perde, para mostrar o campo minado todo aberto
var i,j: longint;
begin
  for i:= 1 to c.x do
     for j:= 1 to c.y do
        c.m[i,j].revelado:= true;
end;
```

Figura 11.15: Revela todos os quadrados para impressao final.
A última procedure que falta ser implementada é a parte da revelação dos vizinhos que tem zero minas nas vizinhanças, o que é apresentado na figura \[11.16\]. Como a função foi chamada com apenas um elemento na pilha, ela entra em um laço que desempilha o topo e testa seus vizinhos. Para cada um deles que contiver zero minas vizinhas, empilha estes. Quando a pilha estiver vazia é porque todos os vizinhos do quadrado original que não continham minas foram revelados.

A ideia é termos um procedimento que pega um elemento da pilha, sabidamente um quadrado que tem zero minas vizinhas, e que abre todos os vizinhos, mas para os que são mulos, armazena na pilha. Assim, enquanto a pilha contiver elementos, pode haver quadrados para serem revelados ainda.

```pascal
function abrir_vizinhos_sem_mina (var c: campo_minado; var p: pilha): longint;
// usa a pilha de coordenadas para abrir todos os vizinhos que nao tem mina
// retorna o numero de quadrados abertos
var quadrado, vizinho: coordenada; i, j, cont : longint;
begin
  cont:= 1;
  while not pilha_vazia (p) do
  begin
    quadrado:= desempilhar (p);
    for i:= quadrado.x−1 to quadrado.x+1 do
      for j:= quadrado.y−1 to quadrado.y+1 do
        if not c.m[i,j].revelado and (c.m[i,j].info = NADA) and not eh_borda (i,j,c) then
        begin
          vizinho.x:= i;
          vizinho.y:= j;
          empilhar (vizinho, p);
          c.m[i,j].revelado:= true;
          cont:= cont + 1;
        end;
  end;
  abrir_vizinhos_sem_mina:= cont;
end;
```

Figura 11.16: Revela todos os vizinhos sem minas ao redor.

O código completo pode ser visto na página web da disciplina:

http://www.inf.ufpr.br/cursos/ci055/Util/mines.pas
11.2 Exercícios

1. O jogo da vida foi desenvolvido por um matemático chamado John Conway. Sua intenção foi prover um modelo para a vida, morte e sobrevivência entre organismos simples que habitam uma matriz $N \times M$.

A população atual é considerada como uma geração. A partir de uma determinada população inicial (geração zero) as gerações seguintes são calculadas segundo quatro regras, que seguem:

- Toda célula vazia com três vizinhos vivos nascerá na próxima geração;
- Qualquer célula com um ou zero vizinhos vivos morrerá de solidão, enquanto que qualquer célula com quatro ou mais vizinhos vivos morrerá sufocado;
- Qualquer célula com dois ou três vizinhos continuará vivo na próxima geração.
- Todos os nascimentos e mortes ocorrem ao mesmo tempo.

Por quê vida é um jogo? Bem, apesar de certas populações iniciais morrerem rapidamente, outras formam padrões interessantes de repetição, crescem, ou se movem ao longo do tabuleiro conforme passam de geração para geração.

Faça um programa em Pascal que jogue vida:

- Deixe que o usuário especifique a localização da população inicial e também o número de gerações que devem ser mostradas na saída;
- Inicialize uma matriz retangular $N \times M$ chamada Geracao que conterá a configuração inicial das células vivas;
- Repetir os próximos passos enquanto desejar:
  (a) Para cada uma das células na matriz Geracao faça:
    - Conte o número de células vizinhas vivas;
    - Se a contagem for 0, 1, 4, 5, 6, 7 ou 8 então defina a célula correspondente de outra matriz chamada NovaGeracao como sendo morta;
    - Se a contagem for 3, então defina a célula correspondente em NovaGeracao como sendo viva;
    - Se a contagem for 2, então defina a célula correspondente em NovaGeracao como sendo igual à da matriz Geracao.
  (b) Atualize a matriz Geracao com o conteúdo da matriz NovaGeracao (Faça isso de modo esperto, sem ter que copiar todos os elementos);
  (c) Imprima a matriz Geracao, limpando a tela antes.

Observe com atenção:

- As dimensões das matrizes devem ser informadas na linha de comando;
• Faça uma boa apresentação do código (legibilidade, identação);
• Use bons nomes para as variáveis;
• Faça uso de constantes que facilitem a manutenção do código, bem como a legibilidade;
• Use corretamente as variáveis locais e globais;
• Bons comentários no código são fundamentais, mas não exagere no número deles, use com bom senso em situações pertinentes.

2. Fazendo uso das boas técnicas de programação vistas durante o curso, faça um programa em *Pascal* que implemente um *jogo da velha*:

• O jogo possui um tabuleiro composto de nove posições, na forma de uma matriz de tamanho 3 por 3; cada posição pode estar vazia ou pode ser ocupada pelo símbolo de um dos jogadores.
• Dois jogadores participam do jogo, sendo que a cada um destes é associado um símbolo distinto, por exemplo: “X” e “0”.
• A primeira jogada é efetuada pelo jogador X; em cada jogada um dos jogadores ocupa uma posição vazia do tabuleiro; os jogadores se alternam a cada jogada.
• Um dos jogadores vence quando ocupa uma posição que completa uma sequência de três símbolos iguais em uma linha, coluna ou diagonal.
• O jogo termina empatado quando todas as posições do tabuleiro foram ocupadas e não houve vencedor.

3. Considere um jogo de Batalha Naval em que cada participante disporá seus 5 barcos de 1, 2, 3, 4 e 5 células, respectivamente, no espaço de uma matriz de $N \times N$ ($N \geq 100$).

Os barcos deverão estar ou na direção horizontal ou na vertical, deverão ser retos e não poderão se tocar. Não poderá haver barcos que passem pelas linhas e colunas marginais da matriz.

Faça em *Pascal* o programa principal para montar um jogo e fazer uma disputa entre dois adversários, especificando as chamadas às diferentes funções e procedimentos.

Escrever as funções e procedimentos necessários para o bom funcionamento do seu programa principal acima.

Você deve documentar a lógica da solução de forma precisa. Em particular, descreva as estruturas de dados que você utilizar e a forma como elas serão usadas para resolver o problema.
Referências Bibliográficas


