Algoritmos e Estruturas de Dados I

Marcos Castilho Fabiano Silva Daniel Weingaertner

Versão 0.9.8.1
Julho de 2020

Dedicatória

Este livro é dedicado (*in memorian*) ao professor Alexandre Ibrahim Direne para quem deixamos aqui nossa eterna gratidão pelos ensinamentos, discussões, críticas sempre construtivas e um exemplo de professor preocupado com o aprendizado dos alunos.

Nós procuramos respeitar suas preocupações neste livro. Por isso pedimos aos leitores que, no mesmo espírito, nos façam críticas e sugestões, sempre bem-vindas.

Na verdade, o Alexandre sempre foi o mentor nas nossas experiências e tentativas de construir um material de qualidade para os alunos em geral, mas de forma particular quanto aos calouros.

Uma de suas principais intervenções era que os exercícios deste livro deveriam ser cuidadosamente organizados. Esperamos ter chegado perto das suas expectativas.

Mas não destacamos apenas este fato, como também sua especialidade que era a informática na educação e sua visão de que esta disciplina é, sem dúvida, a mais importante disciplina de um curso de Computação, pois é ministrada para calouros, e, talvez, a mais difícil de ser ministrada, dada a complexidade inerente ao aprendizado de conceitos elementares de algoritmos.

Obrigado Alexandre por teus ensinamentos.

Os autores.

Sumário

Capítulo 1

Introdução

Neste livro 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;
- Análise de Algoritmos; 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 na 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, este 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 três partes bem definidas, a primeira 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 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, na terceira parte, oferecemos um desafio aos alunos. O objetivo é o de mostrar aplicações interessantes 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.

Uma característica importante deste livro são os exercícios propostos. Eles foram cuidadosamente inseridos em ordem de complexidade, os mais fáceis primeiro e os mais difíceis por último. Praticamente todos os enunciados têm casos de teste para que os estudantes possam testar suas soluções. Os casos de teste também têm o objetivo de facilitar a compreensão dos enunciados.

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 [Fre]. Finalmente, embora talvez de difícil compreensão para um iniciante, recomendamos pelo menos folhear o material em [Knu68].

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 dos capítulos iniciais.

Ao professor André Luiz Pires Guedes agradecemos especialmente por ter feito uma cuidadosa revisão de todo o livro, nos apontando não apenas pequenos erros, mas sugestões valiosas que enriqueceram muito este texto.

Também agradecemos aos alunos de iniciação científica que trabalharam com afinco na organização dos exercícios da parte 1: Arthur D. V. de Castro, Artur T. Coelho, Eduardo M. de Souza, Gabriel H. O. de Mello, Gabriel N. H. do Nascimento, Gustavo H. S. Barbosa, Jorge L. V. Jabczenski, Leonardo L. Dionízio, Pietro P. Cavassin, Tiago H. Conte, Vinicius T. V. Date.

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.

Parte I

Algoritmos

Introdução da parte 1

Nesta parte apresentaremos os conceitos básicos de algoritmos que estão presentes em qualquer linguagem de programação de alto nível.

Podemos dizer que é possível, com este conteúdo, programar uma vasta coleção de algoritmos, inclusive alguns com alta complexidade.

Apresentaremos:

- Uma discussão sobre problemas e soluções: para um determinado problema, pode existir mais de uma solução;
- O modelo básico do funcionamento do computador: sem entrar em detalhes de eletrônica, veremos em um nível de abstração mais baixo como é o funcionamento dos computadores.
- Conceitos elementares: quais são os elementos mais básicos possíveis que temos à disposição nas linguagens de programação em geral?
- Técnicas elementares: quais são as técnicas fundamentais que um programador deve conhecer?
- Aplicações das técnicas elementares: vamos resolver alguns problemas que generalizam os conteúdos vistos, habilitando o aprendiz a resolver problemas cada vez mais complexos;
- Noções importantes sobre modularidade, através de funções e procedimentos: não basta programar, tem que programar bem, com códigos legíveis, modulares, de simples manutenção. Devemos primar pela redação, pois quem programa e quem lê este programa é um ser humano.

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[1].

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

[1] A primeira aula, que chamamos de *aula inaugural*, é ministrada para todas as turmas da disciplina, por isso estava com 100 alunos. Cada turma não ultrapassa 40 alunos em média.

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 necessário 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 contagem 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

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 ímpar 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 deterioravam 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?

Com 4 pessoas, cada uma troca uma roda, divide-se o tempo por 4. Opa! Já estamos abaixo de 1 minuto.

Se tiver ainda a possibilidade de 3 pessoas por roda: um tira o parafuso, outro tira a roda velha, um terceiro coloca a roda nova e o primeiro aperta o parafuso. Mais 2 mecânicos para levantar e baixar o carro todo de uma vez e está feito.

Hoje em dia se trocam os quatro pneus de um carro de fórmula 1 em cerca de 2 segundos.

Piquet foi campeão naquele ano, pois também usou o truque de aquecer os pneus antes da prova e andar com o carro contendo metade da gasolina, já que ele ia ter que parar nos boxes de qualquer maneira para trocar os pneus... O cara é um gênio.

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.[2].

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 duas matrizes 3 por 3 específicas. Em geral, na ciência da computação, estamos interessados em resolver problemas, ou melhor dizendo, em *estudar* problemas, e não somente resolver instâncias dele. Saber que $2 + 2 = 4$ não interessa. Interessa saber como é o processo de obtenção da soma de dois números.

[2]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 e 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

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 houver gemas, junte uma gema e depois bata até obter um creme homogêneo.
     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.[1]

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

[1] O texto deste capítulo foi adaptado de outro escrito pelo prof. Renato Carmo para a disciplina CI-208 - Programação de Computadores ministrada para diversos cursos na UFPR.

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 usadas 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 Computer 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

[2]Quem assistiu ao filme Matrix pode imaginar a complicação.

Endereço	Conteúdo
0	1
1	54
2	2
3	1
4	50
5	4
6	7
7	46
8	4
9	47
10	46
11	46
12	7
13	48
14	4
15	49
16	50
17	48
18	3
19	51
20	47

Endereço	Conteúdo
21	49
22	6
23	52
24	51
25	3
26	53
27	46
28	52
29	5
30	55
31	53
32	54
33	8
34	55
35	2
36	56
37	46
38	52
39	5
40	57
41	56

Endereço	Conteúdo
42	54
43	8
44	57
45	9
46	33
47	2
48	76
49	67
50	76
51	124
52	14
53	47
54	235
55	35
56	23
57	78
58	243
59	27
60	88
61	12
62	12

Figura 4.1: Uma fotografia da memória.

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 uma notação. Seja p um endereço. Denotamos por $[p]$ o conteúdo do endereço p. Vejamos alguns exemplos com base na figura 4.1:

$$[0] = 1$$
$$[[0]] = [1] = 54$$
$$[[[0]]] = [[1]] = [54] = 235$$
$$[0] + 1 = 1 + 1 = 2$$
$$[0 + 1] = [1] = 54$$
$$[[0] + 1] = [1 + 1] = [2] = 2$$
$$[[0] + [1]] = [1 + 54] = [55] = 35$$

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.[3].

Código	**Mnemônico**	**Descrição**	**Notação**
1	`load`	escreva em $[p+1]$ o valor do número em $[p+2]$ e some 3 em p	$[p+1] \leftarrow [p+2]$.
2	`add`	escreva em $[p+1]$ o valor da soma dos números em $[[p+2]]$ e $[[p+3]]$ e some 4 em p	$[p+1] \leftarrow [[p+2]] + [[p+3]]$.
3	`sub`	escreva em $[p+1]$ o valor da subtração do número em $[[p+2]]$ pelo número em $[[p+3]]$ e some 4 em p	$[p+1] \leftarrow [[p+2]] - [[p+3]]$.
4	`mult`	escreva em $[p+1]$ o valor do produto dos números em $[[p+2]]$ e $[[p+3]]$ e some 4 em p	$[p+1] \leftarrow [[p+2]] \times [[p+3]]$.
5	`div`	escreva em $[p+1]$ o valor da divisão do número em $[[p+2]]$ pelo número em $[[p+3]]$ e some 4 em p	$[p+1] \leftarrow \frac{[[p+2]]}{[[p+3]]}$.
6	`sqrt`	escreva em $[p+1]$ o valor da raiz quadrada de $[[p+2]]$ e some 3 em p	$[p+1] \leftarrow \sqrt{[[p+2]]}$.
7	`read`	leia um número do teclado, escreva-o em $[p+1]$ e some 2 em p	$[p+1] \leftarrow \underline{\vee}$.
8	`write`	escreva $[[p+1]]$ na tala e some 2 em p	$\square \leftarrow [[p+1]]$.
9	`stop`	pare	$\bullet$

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, execute esta instrução, e pare somente quando a instrução for uma ordem de parar (instrução 9, `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 compreenda, 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 "`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 "`load`". Analogamente ao que foi feito no parágrafo anterior, o conteúdo de $[5] = 4$ é colocado como sendo o conteúdo da posição $[4] = 50$. Na posição 50 havia o valor 76. Após a execução da instrução o 76 dá lugar ao 4. Ao final o valor de p foi atualizado para 6.
4. Como $p = 6$ devemos interpretar $[6] = 7$. Logo, a instrução para ser executada agora é "`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.

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

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.

Instrução	Operando	Operando	Operando
1	54	2	
1	50	4	
7	46		
4	47	46	46
7	48		
4	49	50	48
3	51	47	49
6	52	51	
3	53	46	52
5	55	53	54
8	55		
2	56	46	52
5	57	56	54
8	57		
9			

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?

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.

Instrução	Operando	Operando	Operando
`load`	54	2	
`load`	50	4	
`read`	46		
`mult`	47	46	46
`read`	48		
`mult`	49	50	48
`sub`	51	47	49
`sqrt`	52	51	
`sub`	53	46	52
`div`	55	53	54
`write`	55		
`add`	56	46	52
`div`	57	56	54
`write`	57		
`stop`			

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.

$x = [p+1]$
$y = [p+2]$
$z = [p+3]$

$[p]$	**Instrução**	**Notação**
1	`load` $x\ y$	$x \leftarrow [y]$
2	`add` $x\ y\ z$	$x \leftarrow [y] + [z]$
3	`sub` $x\ y\ z$	$x \leftarrow [y] - [z]$
4	`mult` $x\ y\ z$	$x \leftarrow [y] \times [z]$
5	`div` $x\ y\ z$	$x \leftarrow \frac{[y]}{[z]}$
6	`sqrt` $x\ y$	$x \leftarrow \sqrt{[y]}$
7	`read` x	$x \leftarrow V$
8	`write` x	$V \leftarrow [x]$
9	`stop`	$\bullet$

Figura 4.6: Notação para as instruções.

54	$\leftarrow$	2
50	$\leftarrow$	4
46	$\leftarrow$	$\underline{\vee}$
47	$\leftarrow$	$[46] \times [46]$
48	$\leftarrow$	$\underline{\vee}$
49	$\leftarrow$	$[50] \times [48]$
51	$\leftarrow$	$[47] - [49]$
52	$\leftarrow$	$\sqrt{[[51]]}$
53	$\leftarrow$	$[46] - [52]$
55	$\leftarrow$	$\frac{[53]}{[54]}$
$\square$	$\leftarrow$	$[55]$
56	$\leftarrow$	$[46] + [52]$
57	$\leftarrow$	$\frac{[56]}{[54]}$
$\square$	$\leftarrow$	$[57]$
$\bullet$		

Figura 4.7: Programa escrito sob nova notação.

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):

- `http://www.youtube.com/watch?v=sShMA85pv8M`;
- `http://www.youtube.com/watch?v=Lr1DWkgUBTw`.

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.

Endereço	**Nome**
54	dois
50	quatro
46	B
47	quadradoB
48	C
49	quadruploC
51	discriminante
52	raizDiscriminante
53	dobroMenorRaiz
55	menorRaiz
56	dobroMaiorRaiz
57	maiorRaiz

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.

dois	$\leftarrow$	2
quatro	$\leftarrow$	4
B	$\leftarrow$	$\underline{\vee}$
quadradoB	$\leftarrow$	B $\times$ B
C	$\leftarrow$	$\underline{\vee}$
quadruploC	$\leftarrow$	quatro $\times$ C
discriminante	$\leftarrow$	quadradoB - quadruploC
raizDiscriminante	$\leftarrow$	$\sqrt{discriminante}$
dobroMenorRaiz	$\leftarrow$	B - raizDiscriminante
menorRaiz	$\leftarrow$	$\frac{dobroMenorRaiz}{dois}$
$\square$	$\leftarrow$	menorRaiz
dobroMaiorRaiz	$\leftarrow$	B + raizDiscriminante
maiorRaiz	$\leftarrow$	$\frac{dobroMaiorRaiz}{dois}$
$\square$	$\leftarrow$	maiorRaiz
$\bullet$		

Figura 4.9: Programa reescrito com nomes para variáveis.

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. São conhecidas como *gramáticas livre de contexto*. Uma subclasse delas viabiliza escrever programas que não são ambíguos.

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.

read B
read C
discriminante $\leftarrow$ B^2 - 4 $\times$ C
raizDiscriminante $\leftarrow$ $\sqrt{discriminante}$
menorRaiz $\leftarrow$ $\frac{B-raizDiscriminante}{2}$
write menorRaiz
maiorRaiz $\leftarrow$ $\frac{B+raizDiscriminante}{2}$
write maiorRaiz

read B
read C
raizDiscriminante $\leftarrow$ $\sqrt{B^2 - 4 \times C}$
write $\frac{\text{B - raizDiscriminante}}{2}$
write $\frac{\text{C + raizDiscriminante}}{2}$

Figura 4.10: Duas outras versões do programa.

Estas versões são compreensíveis para o ser humano, mas ainda não estão no formato ideal para servirem de entrada para o compilador, em particular por causa dos símbolos de fração ou do expoente. Os compiladores exigem um grau maior de rigidez, infelizmente. A disciplina Construção de Compiladores 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.

```
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 programa 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 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 (atribuição, 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[1] 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 encerra.

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

[1] `http://www.inf.ufpr.br/cursos/ci055/pascal.pdf`.

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 video.

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.

```
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 `read` 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`, recupera seu *conteúdo* e imprime na tela este conteúdo, ou este valor.

Observe que o uso de uma variável exigiu, pelas regras desta linguagem, a *declaração* no cabeçalho: `var a: real;`. Toda variável em *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 `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 `real`, que usa a codificação em *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 *sistema operacional* que reserve um espaço de memória para a variável `a`, que deve ser interpretada como um número real;

2. Inicia o programa pela primeira instrução (`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 `a` conterá o valor 2.

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

4. O programa termina.

5.4 O ciclo edição–compilação–testes

A partir deste ponto o leitor deve ter uma versão do *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 `ASCII`[2]. Isto exclui os pacotes tipo *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: *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 *linux* pode fazer assim, após o programa editado em um arquivo chamado `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
```

[2]https://pt.wikipedia.org/wiki/ASCII

```
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:

```
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:

```
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.

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

```
O valor digitado no teclado foi 2.00
```

Este processo é um ciclo porquê 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 grau: $x^2 - bx + c = 0$. A figura 5.3 é uma cópia daquela solução apresentada na figura 4.11.

```
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 5.3: Programa que implementa o método de Bhaskara.

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`:

```
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);`.

4. Um primeiro comando de impressão, de uma outra expressão aritmética:
   `write((b - raizdiscriminante)/2);`

5. Um segundo comando de impressão, de uma terceira expressão aritmética:
   `write((b + raizdiscriminante)/2);`

Os comandos de leitura (*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 b e 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 *ENTER*. A equação a ser resolvida será $x^2 - 5x + 6 = 0$.

O terceiro comando (:=) é uma *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 b por b e subtrair o resultado pela operação de multiplicar 4 por c). O resultado disso tudo é um valor numérico do tipo *real*. Este último valor é atribuído na variável $raizdiscriminante$.

Vamos detalhar o que ocorre, supondo que os valores de b e c sejam respectivamente 5 e 6. Maiores detalhes devem ser vistos no material complementar sobre a linguagem *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 b com b, denotado $b * b$. Isto resulta no valor 25. Em seguida, é feita a multiplicação de 4 pelo valor de c, denotado $4 * c$. Isto resulta no valor 24. O terceiro passo é a subtração de 25 por 24, resultando no valor 1. Finalmente, *sqtr* é uma função pré-definida em *Pascal* que calcula o valor da raiz quadrada do valor da expressão entre parênteses. O resultado de fazer $sqrt(1)$ resulta no valor 1. Finalmente, este valor, 1, é atribuído na variável $raizdiscriminante$.

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 *write*, que serve para imprimir alguma coisa na tela do usuário, neste caso, o resultado dos cálculos das expressões aritméticas $\frac{b-raizdiscriminante}{2}$ e $\frac{b+raizdiscriminante}{2}$, 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 *operadores* e estes operadores são executados segundo uma *ordem de precedência*. Por exemplo, multiplicação (*) e

divisão (`/`) têm precedência sobre a adição (`+`). Assim, se a conta a ser feita for

$$\frac{b+c}{2}$$

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:

$$b+\frac{c}{2}$$

Observação: quando usamos a *função* `sqrt` não previmos que seu *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 (*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 Uma atribuição simples e importante

Uma das operações mais comuns em programas é uma variável ser incremantada de uma unidade. Por exemplo, no seguinte programa da figura 5.4, a variável `i` recebe inicialmente o valor zero e imprime este zero na tela. O comando seguinte (`i:= i + 1;`) faz o incremento de uma unidade no valor de `i` e por isso o segundo comando de impressão imprime o valor `1` na tela.

```
program incremanta1;
var i: integer;

begin
        i:= 0;
        write (i);
        i:= i + 1;
        write (i);
end.
```

Figura 5.4: Incrementando uma unidade.

É preciso lembrar que ao fazer uma atribuição, o compilador deve primeiramente resolver a expressão aritmética que está do lado direito do `:=`, no caso é `i + 1`. Para isso, toma o valor atual de `i`, que vale zero, soma com `1`, o que resulta no valor `1` e somente após este cálculo estar completo é que o resultado é atribuído ao valor da variável que está do lado esquerdo. Esta variável é `i`, que recebe o valor `1`. Observe que a saída deste programa é um zero depois um `1`.

5.5.2 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 5.5 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 *expressão aritmética* faz o cálculo da soma de ambos e o resultado é impresso na tela.

```
program soma2;
var a,b: integer;

begin
        read (a);
        read (b);
        write (a+b);
end.
```

Figura 5.5: 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 5.6.

```
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 5.6: Mesma solução, agora com interface amigável.

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 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 endereçamentos de 2 bytes (16 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*b + 4*a+c) >= 0`: a raiz quadrada do discriminante de uma equação do segundo grau é positiva ou nula, isto é, não é negativa.

- `(a >= 3) AND (a <= 5)`: `a` está no intervalo $3 \leq a \leq 5$. Observação: não pode escrever `3 <= a <= 5`, o compilador rejeita.
- `NOT ((a >= 3) AND (a <= 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 ilustrada na figura 5.7:

```
program contar;

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

Figura 5.7: Primeira solução para contar de 1 a 5.

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 5.8.

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.

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

```
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 5.8: Primeira solução modificada para números de 1 a 20.

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 preciso introduzir um tipo de 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 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 5.9 ilustra o uso deste conceito.

```
program contar;
var i: integer;

begin
        i:= 1;
        while i <= 5 do
        begin
              writeln (i);
              i:= i + 1;
        end;
end.
```

Figura 5.9: Sexta solução.

O comando de repetição (*while/do*) executa os comandos aninhados no bloco entre o `begin` e o `end;` enquanto a *expressão booleana* `i <= 5` for verdadeira. No primeiro momento em que for falsa, o fluxo é alterado para depois do `end;`, ou seja, neste caso o programa termina.

Esta expressão vai resultar falso quando i for estritamente maior do que 5. O símbolo `<=` significa *menor ou igual*, logo, `i <= 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 que resultar em falso, pula o laço e termina o programa.

Neste exemplo, a variável i foi inicializada com o valor 1. O comando `while` então avalia *antes* se 1 é menor ou igual a 5. Como isto é verdadeiro, entramos no laço. 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 <= 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.

5.7.1 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 5.10 e imprime todos os valores *inteiros* de 1 a 30, juntamente com seus respectivos quadrados.

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*i`, que significa $i \times i$, ou seja, i^2.

```
program quadrados;
var i: integer;

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

Figura 5.10: Tabela com os quadrados dos números de 1 a 30.

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 <= 30` vir um dia a ser falsa, o que provocará o término deste laço.

- O teste `i <= 30` 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 se 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 5.5.2. 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*.

Apenas uma questão deve ser resolvida: quantos pares de números serão dados como entrada? O enunciado diz "vários pares", mas não estabelece o número preciso. Algoritmos não sabem lidar com isto.

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 5.11 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.

```
program soma2variasvezes_v1;
var a,b,cont: integer;
(* cont conta os numeros lidos *)
begin
      cont:= 1;
      while cont <= 30 do
      begin
            read (a);
            read (b);
            writeln (a+b);
            cont:= cont + 1;
      end;
end.
```

```
program soma2variasvezes_v2;
var a,b: integer;

begin
      read (a);
      read (b);
      while (a <> 0) or (b <> 0) do
      begin
            writeln (a+b);
            read (a);
            read (b);
      end;
end.
```

Figura 5.11: Duas formas para somar pares de números.

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 exatamente o mesmo, lê dois números e imprime a soma, o que muda é o controle do laço. Observem que nos dois casos o laço pode terminar pois as variáveis de controle

são passível de mudança de valor. No programa da esquerda, a variável `cont` sofre um incremento dentro do laço enquanto que no programa da direita os valores de a e b são lidos do teclado dentro do laço, portanto também podem ter o valor alterado.

A próxima seção apresentará a última estrutura básica que nos interessa.

5.8 Desviando fluxo condicionalmente

Problema: Ler um único número do teclado e imprimí-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 5.12 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.

```
program imprime_se_positivo;
var a,b: integer;

begin
      read (a);
      if a > 0 then
            writeln (a); (* so executa se a for positivo *)
end.
```

Figura 5.12: Imprime se for positivo.

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.

Considere o seguinte problema.

Problema: Ler um único número do teclado e imprimí-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 5.13. 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.

```
program imprime_se_positivo_v2;
var a,b: integer;

begin
      read (a);
      if a > 0 then
            writeln (a)                    (* so executa se a for positivo *)
      else
            writeln ('numero invalido'); (* executa se a for nulo ou negativo *)
end.
```

Figura 5.13: 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*);
- 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 grafia 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 5.14 ilustra o uso do desvio incondicional para resolver o problema de se imprimir todos os números de 1 a 30.

```
program contar;
label: 10;
var i: integer;

begin
        i:= 1;

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

        if i <= 30 then
            goto 10;
end.
```

Figura 5.14: Exemplo do uso do desvio incondicional.

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 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 5.9. A observação interessante é que, no nível de máquina, o que o compilador faz é gerar a partir do programa da figura 5.9, 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:

`http://www.inf.ufpr.br/cursos/ci055/pascal.pdf`,

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* :

```
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 inteiras 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 aritimética. 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:

$$X = \frac{\frac{A+B}{C-D}E}{\frac{F}{AB} + E}$$

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*.

Exemplo de entrada	Saída esperada
1 2 3 4 5 6	-1.8750000000000000E+000
1 -1 1 -1 1 -1	0.0000000000000000E+000
3 5 8 1 1 2	1.0084033613445378E+000

3. Faça um programa em *Pascal* que implemente 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 ($x^2 = x * x$).

   (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:

Exemplo de entrada	Saída esperada
12 52 7 13	12:52 + 7:13 = 20:05
20 15 1 45	20:15 + 1:45 = 22:00
0 0 8 35	0:0 + 8:35 = 8:35

   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.

Exemplo de entrada	Saída esperada
3600	1, 0, 0
3500	0, 58, 20
7220	2, 0, 20

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

Exemplo de entrada	Saída esperada
3 7	7 3
-5 15	15 -5
2 10	10 2

7. (*) Desafio: Faça um programa em *Pascal* que troque o conteúdo de duas variáveis *inteiras* 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:

$$a = 6; b = 9.5; d = 14; p = 4; q = 5; r = 10; z = 6.0; sim = TRUE.$$

(a) `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 aritimética. Adicionalmente, você precisa estudar sobre expressões lógicas (ou booleanas) e sobre a ordem de precedência dos operadores relacionais ($=, <>, \leq, \geq, >, <$) e lógicos (NOT, AND, OR).

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

$$a = 6; b = 9.5; d = 14; p = 4; q = 5; r = 10; z = 6.0; sim = TRUE.$$

(a) `NOT sim AND (z DIV b + 1 = r)`

(b) `(x + y > z) AND sim OR (d \ge b)`

(c) `(x + y <> z) AND (sim OR (d \ge b))`

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

$$a = 5; b = 3; d = 7; p = 4; q = 5; r = 2; x = 8; y = 4; z = 6; sim = TRUE.$$

(a) `(z DIV a + b * a) - d DIV 2`

(b) `p / r mod q - q / 2`

(c) `(z DIV y + 1 = x) 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.

```
program questao1;
var
  m, x, y: integer;
begin
  read(x);
  y := 0;
  m := 1;
  while x > 0 do
    begin
      y := y + (x mod 2) * m;
      x := x div 2;
      m := m * 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:

Exemplo de entrada	Saída esperada
5	impar
4	par
15	impar

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:

Exemplo de entrada	Saída esperada
5 2	primeiro eh maior
2 5	segundo eh maior ou igual
5 5	segundo eh maior ou igual

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:

Exemplo de entrada	Saída esperada
3 5 8	esta no intervalo
3 8 8	nao esta no intervalo
4 12 5	nao esta no intervalo

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:

Exemplo de entrada	Saída esperada
3 5 8	sim
3 8 8	nao
4 12 5	nao

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:

$$X = \frac{\frac{A+B}{C-D}E}{\frac{F}{AB} + E}$$

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

Exemplos de entradas	Saídas esperadas
1 2 3 4 5 6	-1.8750000000000000E+000
0 0 0 0 0 0	divisao por zero
1 1 2 2 1 3	divisao por zero

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:

Exemplo de entrada	Saída esperada
12 6 26 86 0	12 14 6 8 26 28 86 88
-2 -5 -1 0	-2 0 -5 -3 -1 1
1 2 3 4 5 0	1 3 2 4 3 5 4 6 5 7

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.

Exemplo de entrada	Saída esperada
1.00 -1.00 -6.00	-3.00 2.00
1.00 0.00 -1.00	-1.00 1.00
1.00 0.00 0.00	0.00 0.00
0.00 0.00 0.00	

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.

Exemplo de entrada	Saída esperada
30 7	2
3 2	1
12 3	0

20. Faça um programa em *Pascal* que leia um número $n > 0$ do teclado e imprima a tabuada de n de 1 até 10.

Exemplo de entrada	Saída esperada
5	5 x 1 = 5 5 x 2 = 10 5 x 3 = 15 5 x 4 = 20 5 x 5 = 25 5 x 6 = 30 5 x 7 = 35 5 x 8 = 40 5 x 9 = 45 5 x 10 = 50

5.11 Exercícios complementares

Estes exercícios[3] 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.

Exemplo de entrada	Saída esperada
1	2 0
100	101 99
-3	-2 -4

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= ".

Exemplo de entrada	Saída esperada
1 2	SOMA= 3
100 -50	SOMA= 50
-5 -40	SOMA= -45

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.

Exemplo de entrada		Saída esperada
Valor original	Desconto	Valor na promoção
500.00	50.00	450.00
10500.00	500.00	10000.00
90.00	0.80	89.20

[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.

Exemplo de entrada	Saída esperada
1.2 2.3	1.75
750 1500	1125.00
8900 12300	10600.00

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

Exemplo de entrada	Saída esperada
3	1.00
10	3.33
90	30.00

6. Uma P.A. (progressão aritmética) é determinada pela sua razão (r) 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.A., dado a razão (r) e o primeiro termo (a_1). Seu programa deve ler n, r, a_1 do teclado e imprimir a_n.

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

Exemplo de entrada			Saída esperada
n	r	a_1	a_n
8	1	3	10
100	10	1	991
5	-2	0	-98

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

$$a_n = a_k + (n - r) \times r$$

Exemplo de entrada				Saída esperada
k	a_k	r	n	a_n
1	5	2	10	23
10	20	2	5	10
100	500	20	90	300

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.

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

Exemplo de entrada			Saída esperada
a_1	q	n	a_n
1	1	100	1.00
2	2	10	1024.00
5	3	2	15.00

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 (an). Seu programa deve ser k, a_k, q, n do teclado e imprimir a_n.

Exemplo de entrada				Saída esperada
k	a_k	q	n	a_n
2	2	1	1	2
1	5	2	10	2560.00
2	100	10	20	10000000000000000000.00

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.

Exemplo de entrada			Saída esperada
a_1	q	n	a_n
1	1	100	1.00
2	2	10	1024.00
10	2	20	5242880.00

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.

Exemplo de entrada	Saída esperada
2569	9
1000	0
1305	5

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.

Exemplo de entrada	Saída esperada
2569	6
1000	0
1350	5

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.

Exemplo de entrada	Saída esperada
2500	5
2031	0
6975	9

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.

Exemplo de entrada	Saída esperada
2569	2
1000	1
0350	0

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. i 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.

Exemplo de entrada	Saída esperada
500	5
360	3
958	9

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.

Exemplo de entrada			Saída esperada
n	a_1	a_n	*soma*
100	1	100	5050.00
10	1	10	55.00
50	30	100	3250.00

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.

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

Exemplo de entrada		Saída esperada
A	C	B
1	3	2.00
2	2	2.00
100	500	300.00

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.

Exemplo de entrada		Saída esperada
A	C	B
1	3	1.73
10	100	31.62
90	80	84.85

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!)

Exemplo de entrada			Saída esperada
a_1	q	n	P
5	1	10	9765625.00
1	1	10	1.00
2	2	5	32768.00

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.

Exemplo de entrada			Saída esperada
a_1	q	n	S_n
2	3	6	728.00
0	5	10	0.00
150	30	2	4650.00

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.

Exemplo de entrada		Saída esperada
r	h	V
5	100	7853.98
25	25.5	69704.03
10	50.9	15990.69

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.

Exemplo de entrada			Saída esperada
valor hora aula	número de aulas	percentual INSS	Salário bruto
6.25	160	1.3	987.00
20.5	240	1.7	4836.36
13.9	200	6.48	2599.86

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.

Exemplo de entrada		Saída esperada
valor do produto (R$)	novo valor (R$)	valor do desconto (R$)
100	91.00	9.00
1500	1365.00	135.00
60000	54600.00	5400.00

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 i valor da gorjeta e o valor total com a gorjeta.

Exemplo de entrada	Saída esperada
75	82.50
125	137.50
350.87	385.96

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.

Exemplo de entrada		Saída esperada
hora	minuto	total de minutos
1	0	60
14	30	870
23	55	1435

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.

Exemplo de entrada		Saída esperada	
depósito	taxa de juros	rendimento	total
200	0.5	1.00	201.00
1050	1	10.5	1060.5
2300.38	0.06	1.38	2301.38

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.

Exemplo de entrada		Saída esperada
salário mínimo (R$)	salário (R$)	salário em salários mínimos (R$)
450.89	2700.00	5.99
1000.00	1000.00	1.00
897.50	7800.00	8.69

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. $Distancia = Tempo \times Velocidade$. $Litros = Distancia/12$. O algoritmo deverá apresentar os valores da Distância percorrida e a quantidade de Litros utilizados na viagem.

Exemplo de entrada			Saída esperada
tempo gasto	velocidade média	distância percorrida	litros
60	100	6000.00	500.00
1440	80	115200.00	9600.00
5	90	450.00	37.50

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.

Exemplo de entrada		Saída esperada
total de vendas (R$)	sapatos vendidos	salário (R$)
50000.00	100	10500.00
2000.00	30	550.00
1000000.00	500	202500.00

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.

Exemplo de entrada		Saída esperada
valor da dívida (R$)	juros/mês	dívida (R$)
100.00	10	110.00
1500.00	3	1545.00
10000.00	0.5	10050.00

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;

Exemplo de entrada		Saída esperada	
salário mínimo (R$)	quilowatts	valor do quilowatt (R$)	valor pago (R$)
750.00	200	1.07	214.29
935.00	150	1.34	200.36
1200.00	250	1.71	428.57

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.

Exemplo Entrada	Saída esperada
30.56	30.56
20	
20.05	20.05

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.

Exemplo Entrada	Saída esperada
7 4	 7
7 2	 2
3 7	 7

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

Exemplo Entrada	Saída esperada
5	impar
3	impar
2	par

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

Exemplo Entrada	Saída esperada
5	positivo
-3	negativo
0	nulo

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.

Exemplo Entrada	Saída esperada
13.14 5	13.14
-3 -4	-12.00
16 5	20.00

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

Exemplo Entrada	Saída esperada
5.35 4	4.00
-3 1	-3.00
6 15	6.00

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.

Exemplo Entrada	Saída esperada
5 10	nao
4 2	sim
7 21	nao

8. Faça um programa em *Pascal* que leia um número inteiro e informe se ele é ou não divisível por 5.

Exemplo Entrada	Saída esperada
5	sim
-5	sim
3	nao

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

Exemplo Entrada	Saída esperada
5	nao
-3	sim
15	sim

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.

Exemplo Entrada	Saída esperada
0	0.00
4	2.00
-5	25.00

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

Exemplo Entrada	Saída esperada
21	sim
7	nao
3	nao
-42	sim

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.

Exemplo Entrada	Saída esperada
500 200	nao
1000.50 250.10	sim
1000 300	sim

13. Faça um programa em *Pascal* que dado quatro valores, A, B, C e D, o programa imprima o menor e o maior valor.

Exemplo Entrada	Saída esperada
1 2 3 4	1.00 4.00
-3 0 1 1	-3.00 1.00
3.5 3.7 4.0 5.5	3.50 5.50

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).

Exemplo Entrada	Saída esperada
1 2 1.5	1.00 1.50 2.00
-3 -4 -5	-5.00 -4.00 -3.00
6 5 4	4.00 5.00 6.00

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).

Exemplo Entrada	Saída esperada
1 2 1.5	2.00 1.50 1.00
-5 -4 -3	-3.00 -4.00 -5.00
5 6 4	6.00 5.00 4.00

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.

Exemplo Entrada	Saída esperada
4 3	9.00 2.00
4.35 3.50	12.25 2.09
-4 -16	256.00

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).

Exemplo Entrada	Saída esperada
50.50	sim
20	nao
90	nao

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.

Exemplo Entrada	Saída esperada
10	10 5 2
5	5
4	2
7	nenhum

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.

Exemplo Entrada	Saída esperada
5	igual a 5
200	igual a 200
400	igual a 400
750.50	intervalo entre 500 e 1000
1000	intervalo entre 500 e 1000
1500	fora dos escopos

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.

De 0 a 500	nenhum credito
De 501 a 1000	30% do valor do saldo medio
De 1001 a 3000	40% do valor do saldo medio
Acima de 3001	50% do valor do saldo medio

Exemplo Entrada	Saída esperada
300.50	0.00
571	171.30
1492.35	596.94
3001.20	1500.60

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).

Exemplo Entrada	Saída esperada
18	maior
15	menor
65	idosa

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).

Exemplo Entrada	Saída esperada
15	nao eleitor
16	facultativo
17	facultativo
18	obrigatorio
19	obrigatorio

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:

Infantil A	5 – 7 anos
Infantil B	8 – 10 anos
Juvenil A	11 – 13 anos
Juvenil B	14 – 17 anos
Sênior	Maiores de 18 anos

Exemplo Entrada	Saída esperada
4	INVÁLIDO
7	Infantil A
8	Infantil B
10	Infantil B
11	Juvenil A
13	Juvenil A
14	Juvenil B
17	Juvenil B
18	Sênior

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".

Exemplo Entrada	Saída esperada
5 5 5	EQUILATERO
7 7 5	ISOSCELES
3 4 5	ESCALENO
5 4 15	INVALIDO

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).

Exemplo Entrada	Saída esperada
3.1 2.5	REPROVADO
3 3	EXAME
10 0	EXAME
6 8	APROVADO
10 10	APROVADO

26. Faça um programa em *Pascal* que leia o 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

Exemplo Entrada	Saída esperada
0	INEXISTENTE
1	DOMINGO
2	SEGUNDA
3	TERÇA
4	QUARTA
5	QUINTA
6	SEXTA
7	SÁBADO
8	INEXISTENTE

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 porquê 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.[1]

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;

[1] Beremir, protagonista do livro "O Homem que Calculava" era capaz de dizer quantos pássaros havia numa revoada no céu.

- 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 executados;
- 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 `writeln` é executado;
- este comando é o de impressão do resultado: `writeln (cont - 1);`
- o programa termina.

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_novo;
var cont, n: integer;

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

situação	valor de *cont*	valor de n	comentário
antes do início	indefinido	indefinido	alocação de memória
executa `cont:= 0`	0	indefinido	
executa `read(n)`	0	5	supondo que leu 5
testa `n <> 0`	0	5	$5 \neq 0$, entra no laço
executa `cont:= cont + 1`	1	5	incrementou `cont`
executa `read(n)`	1	8	supondo que leu 8
testa `n <> 0`	1	8	$8 \neq 0$, permanece no laço
executa `cont:= cont + 1`	2	5	incrementou `cont`
executa `read(n)`	2	3	supondo que leu 3
testa `n <> 0`	2	3	$3 \neq 0$, permanece no laço
executa `cont:= cont + 1`	3	3	incrementou `cont`
executa `read(n)`	3	9	supondo que leu 9
testa `n <> 0`	3	9	$9 \neq 0$, permanece no laço
executa `cont:= cont + 1`	4	9	incrementou `cont`
executa `read(n)`	4	0	supondo que leu 0
testa `n <> 0`	4	0	$0 = 0$, sai do laço
executa `writeln (cont)`	4	0	imprime 4 na tela

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 `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 pois serve para marcar o final da entrada de dados.

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 `soma`.

Observem que a variável `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.

```
program soma_valores;
var
      numero, soma:   integer;

begin
      soma:= 0;                        (* inicializa o acumulador *)
      read (numero);
      while numero <> 0 do
      begin
            soma:= soma + numero;      (* atualiza o acumulador *)
            read (numero);
      end;
      writeln (soma);
end.
```

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.

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 n. 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 n vezes.

```
program soma_valores;
var
     n, numero, i, soma:  integer;

begin
    read (n);
    soma:= 0;            (* primeiro acumulador, para somar os numeros lidos *)
    i:= 1;               (* segundo acumulador, para contar os numeros lidos *)
    while i <= n do
    begin
        read (numero);
        soma:= soma + numero;    (* atualizacao do primeiro acumulador *)
        i:= i + 1;               (* atualizacao do segundo acumulador *)
    end;
    writeln (soma);
end.
```

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

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 completamente.

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

A solução para este problema é apresentada na figura 6.5.

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
     write('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, pois isto não foi testado. Isto é mais um exemplo de lógica de programação.

O teste que determina se a é menor do que c, por estar aninhado ao teste anterior, só é executado quando a é menor do que b. Caso a não seja menor do que b, obviamente ele é maior ou igual a b. Neste caso o comando executado é o que segue o else contendo o comentário, isto é, se faz o teste para determinar se b é menor do que c. Se for, então b é o menor garantidamente. Interessante observar que o primeiro teste garante

que b é menor *ou igual* a a. Então pode existir uma situação na qual $a = b$. Mas, por inferência lógica, imprimir o valor de b é suficiente, pois *no pior caso* ambos seriam iguais. Observem também que não se sabe a ordem relativa de a e c, pois isto não foi testado.

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 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.

```
program encontra_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
            if n < menor then // aqui testamos a nova informacao
                menor:= n;    // e corrigimos se for o caso
            read (n);
        end;
        writeln (menor);
    end;
end.
```

Figura 6.6: Encontrando o menor elemento.

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 Fibonacci[2].

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.

[2]Vale a pena ler: http://pt.wikipedia.org/wiki/Número_de_Fibonacci.

```
program Fibonacci;
var ultimo, penultimo, soma, cont, n: integer;
begin
    read (n);              (* define quantos numeros serao impressos *)
    ultimo:= 1;            (* inicializacao das variaveis 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;             (* eh relevante no codigo       *)
        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 devam 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.

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 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.

```
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, imprimí-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.

```
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 (13-19 anos), adultos (20-59 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 conhecidos `if/then/else` aninhados em `while/do`.

```
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 <= 59 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.

É interessante observar novamente a lógica de programação empregada neste aninhamento de `if/then/else`. Observem o primeiro deles, que testa se `idade <= 2`. Caso o teste seja verdadeiro então seguramente temos um bebê. Mas se for falso? Não podemos concluir nada, exceto que `idade >= 3`. Por isso o segundo `if` já usa esta informação, de que a idade é pelo menos 3, e faz o teste para se determinar se o paciente é uma criança, através do teste `idade <= 12`, e assim por diante. É exatamente por esta lógica que o último contador, o dos idosos, não precisa de outro `if`: já se testaram todas as outras possibilidades, e se a linha contendo o último `else` for atingida pelo programa, é porque, com toda certeza, a idade é de 60 anos ou mais (porquê ela é maior do que 59), portanto se trata de um idoso.

É também importante frisar que os testes são feitos de maneira a simplificar as linhas de código. De fato, se a segunda pergunta fosse para se determinar um idoso, logo depois de saber que o paciente não é um bebê, então a expressão booleana seria mais complexa e exigiria o uso de operadores lógicos do tipo `AND`, `OR` e `NOT`. Em resumo, a ordem das perguntas é importante para que o código fique mais simples.

Outra sequencia interessante seria testar dos idosos em direção aos bebês.

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/else`'s aninhados. Em outras palavras, ninguém é obrigado a usar um `case`, pode perfeitamente continuar usando `if/then/else`'s aninhados. Mas para aqueles alunos interessados, servirá para digitar menos código nos seus programas, basta aprender a usar.

```
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..59: adulto:= adulto + 1;
            else idoso:= idoso + 1;
        end;
        read (idade);
    end;
    writeln ('bebes: ',bebe);
    writeln ('criancas: ',crianca);
    writeln ('adolescentes: ',adolescente);
    writeln ('adultos: ',adulto);
    writeln ('idosos: ',idoso);
end.
```

Figura 6.12: O mesmo programa usando *case*.

6.10 Exercícios

1. Faça um programa em *Pascal* que leia uma sequência de números terminada em zero e imprima separadamente a soma dos que são pares e a soma dos que são ímpares. O zero não deve ser processado pois serve para marcar o final da entrada de dados.

Exemplos de entradas	Saídas esperadas
1 2 3 4 5 6 7 8 9 0	Soma dos pares: 20 Soma dos impares: 25
19 0	Soma dos pares: 0 Soma dos impares: 19

2. 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.

Exemplos de entradas	Saídas esperadas
7 14 15 21 18 35 70 0	7 21 35
19 0	

3. 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. O zero não deve ser processado pois serve para marcar o final da entrada de dados.

Exemplos de entradas	Saídas esperadas
3 60 100 201 333 0	60 201
19 0	

4. 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 servem 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 $B, B{+}1, \ldots, A{-}1, A$.

Exemplo de entrada	Saída esperada
4 6	4 5 6
-2 1	-2 -1 0 1
2 -3	-3 -2 -1 0 1 2
0 0	

5. 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.

Exemplo de entrada	Saída esperada
1 2 3	1
0 0 3	0
3 2 1	1
3 1 2	1
0 0 0	

6. Faça um programa em *Pascal* que leia uma sequência de números inteiros terminada em zero e imprima os dois maiores números lidos. Seu programa deve considerar que o usuário que digita os números vai sempre digitar pelo menos dois números diferentes de zero no início. Como sempre, o zero não deve ser processado pois serve para marcar o final da entrada de dados.

Exemplos de entradas	Saídas esperadas
1 2 3 4 5 6 0	6 5
6 5 3 1 0	6 5
2 1 0	2 1

7. 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.

Exemplo de entrada	Saída esperada
3 45	9

8. Faça um programa em *Pascal* que leia dois números inteiros representando respectivamente as populações P_A e P_B de duas cidades A e B em 2009, e outros dois números inteiros representando respectivamente suas taxas percentuais de crescimento anuais X_A e X_B. Com estas informações seu programa deve imprimir *sim* se a população da cidade de menor população ultrapassará a de maior população e *nao* em caso contrário. Adicionalmente, em caso afirmativo, também deve imprimir o ano em que isto ocorrerá. Faça todas as contas usando operadores inteiros.

Exemplos de entradas	Saídas esperadas
100 80 2 3	sim em 2036
80 100 3 2	sim em 2036
100 20 10 50	sim em 2015
100 20 50 10	nao

9. Faça um programa em *Pascal* que leia um número inteiro positivo n e imprima todos os termos, do primeiro ao n-ésimo, da sequência abaixo.

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

Exemplos de entradas	Saídas esperadas
5	5 6 11 12 17
6	5 6 11 12 17 18

10. Faça um programa em *Pascal* que leia um número inteiro positivo K e imprima os K primeiros números perfeitos. 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. A saída do programa deve ter um número perfeito em cada linha.

Exemplos de entradas	Saídas esperadas
2	6 28
4	6 28 496 8.128

11. Faça um programa em *Pascal* que leia um número inteiro positivo N como entrada e imprima cinco linhas contendo as seguintes somas, uma em cada linha:

```
N
N + N
N + N + N
N + N + N + N
N + N + N + N + N
```

Exemplo de entrada	Saída esperada
3	3 6 9 12 15

12. 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. Note que este programa não recebe dados do teclado, somente produz uma saída, que é sempre a mesma.

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

Figura 1

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

14. Adapte 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

15. Faça um programa em *Pascal* que leia um número inteiro positivo N e em seguida leia outros N números inteiros quaisquer. Para cada valor lido, se ele for positivo, imprimir os primeiros 5 múltiplos dele.

Exemplos de entradas	Saídas esperadas
2 7 3	 7 14 21 28 35 3 6 9 12 15
4 2 4 5 3	 2 4 6 8 10 4 8 12 16 20 5 10 15 20 25 3 6 9 12 15

16. Sabe-se que um número da forma n^3 é igual a soma de n números ímpares consecutivos. Por exemplo:

    - $1^3 = 1$
    - $2^3 = 3 + 5$
    - $3^3 = 7 + 9 + 11$
    - $4^3 = 13 + 15 + 17 + 19$

    Faça um programa em *Pascal* que leia um número inteiro positivo M e imprima os ímpares consecutivos cuja soma é igual a n^3 para n assumindo valores de 1 a M.

Exemplos de entradas	Saídas esperadas
1	1
2	1 3 5
4	1 3 5 7 9 11 13 15 17 19

17. Faça um programa em *Pascal* que leia uma sequência de números naturais positivos terminada m zero (que não deve ser processado pois serve para marcar o final da entrada de dados) e imprima o histograma da sequência dividida 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 de entrada	Saída esperada
347 200 3 32000 400 10 20 25 0	Faixa 1: 4 Faixa 2: 1 Faixa 3: 2 Faixa 4: 1

18. (*) Escreva um programa em *Pascal* que leia um número inteiro e verifique 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.

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

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

Dica 2: Use cálculos contendo divisões por 10 e restos de divisão por 10 para separar os dígitos.

Exemplos de entradas	Saídas esperadas
10011	19
101	5
1210	nao eh binario

Capítulo 7

Aplicações das técnicas elementares

Neste capítulo nós mostraremos como construir algoritmos um pouco mais elaborados, para os quais o grau de dificuldade é um pouco superior quando comparados com os problemas elementares dos capítulos anteriores.

Agora as técnicas devem ser combinadas com inteligência e lógica, mas também com criatividade, entendendo as limitações do computador, usando também nossos conhecimentos de matemática, com o objetivo de produzir soluções cada vez mais interessantes para problemas variados.

7.1 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. Ele vai ilustrar bem que nem sempre a primeira solução é a melhor de todas e também vai servir para aprendermos a resolver classes de problemas e não somente o problema particular apresentado. Em outras palavras, vamos aprender a generalizar soluções.

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 7.1, 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, à dificuldades imensas na reimplementação do código para problemas similares.

Os alunos 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. Observem no código da figura 7.1 que usamos expressões aritméticas baseadas em operadores de divisão inteira e resto de divisão inteira para separar os três dígitos.

Assim, a dígito da centena é obtido pela divisão do número de entrada (que tem três dígitos) por 100. A unidade pelo cálculo do resto da divisão inteira por 100. A

dezena é um pouco mais complicada, precisa de uma divisão inteira após o cálculo do resto de outra divisão inteira.

Vejamos, a linha com `numero div 100`, se o número é 123, resulta no valor 1. A linha com `(numero mod 100) div 10` nos retorna o valor 2. Finalmente a linha com `numero mod 100` nos dá o 3. Usando cálculos simples de aritmética básica, construímos o número invertido. Notem que `3*100 + 2*10 + 1` resulta em 321.

```
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 7.1: 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 7.2 que preserva o raciocínio central da primeira solução, desta vez para 4 dígitos.

```
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 7.2: 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 opera-

ções, que consiste em separar o último dígito e remover do número original este dígito que foi separado. É preciso imaginar como um comando `while` possa ser utilizado. Notaram que o algoritmo anterior não tem nenhum comando de repetição? É muito raro um programa não ter comandos de repetição.

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 7.3 e seu funcionamento é baseado em separar o último dígito com uma operação de `mod 10` seguida de outra operação `div 10`. O `mod 10` resulta no último dígito de um número, enquanto que o `div 10` resulta no número anterior *sem* o último dígito.

Por exemplo, se temos o número 123, operamos `123 mod 10` e obtemos o último dígito, no caso 3. A próxima operação é `123 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. Quando começamos com o último dígito separado, inicialmente 3, quando ele é multiplicado por 10 se obtém 30, que quando somado com 2 resulta em 32. Quando se multiplica 32 por 10 se obtem 320, que quando somado com 1 resulta em 321. O problema global é resolvido pela determinação dos valores iniciais para as variáveis.

```
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 7.3: Solução com uso de acumuladores.

Esta técnica permite que com uma simples alteração no controle do laço possamos inverter números com qualquer quantidade de dígitos. Por exemplo, se o número tiver 7 dígitos, basta trocar o controle do laço de `i <= 3` para `i <= 7` que o programa vai funcionar.

Mas todas estas soluções exigem que saibamos quantos dígitos tem o número que o usuário digitou no teclado. E se não soubermos? Será que ainda assim é possível

resolver este problema? Sim! É possível.

Pense em realizar uma sequência de divisões por 10 para o número de entrada, a cada uma se incrementa um contador. Quando o número de entrada tiver sofrido divisões por 10 suficientes, chegaremos em zero e contador conterá o número de dígitos do número. Observem que tomamos o cuidado de "destruir" uma cópia do número original, para não perdê-lo.

Vejam esta técnica no programa da figura 7.4. Deixamos como exercício a integração destes dois algoritmos para produzir um algoritmo totalmente genérico para o problema original.

```
program conta_digitos;
var numero, copia, cont: longint;

begin
    read (numero);
    copia:= numero;
    cont:= 0;
    while copia <> 0 do
    begin
        copia:= copia div 10;
        cont:= cont + 1;
    end;
    writeln ('o numero ',numero,' tem ',cont,' digitos.');
end.
```

Figura 7.4: Contando quantos dígitos existem em um número dado.

7.2 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 binários envolve 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:

```
22 div 2
 0     11 div 2
        1     5 div 2
              1     2 div 2
                    0     1 div 2
                          1     0
```

Então o número binário correspondente ao 22 seria 10110. O programa ilustrado na figura 7.5 mostra uma tentativa de código para esta versão do algoritmo.

```
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 7.5: Convertendo para binário, versão 1.

Ocorre que este algoritmo imprime o binário ao contrário, isto é, a saída para a entrada 22 seria 01101 e não 10110 como deveria.

É possível usar a técnica explicada na seção 7.1 e se inverter o número. De fato, basta que, ao invés de imprimir na tela `n mod 2`, este número seja usado junto com um acumulador com a técnica da multiplicação por 10. Então um número parecido com binário estará sendo construído e poderá sem impresso logo antes do término do programa. O laço propriamente dito não teria comandos de impressão, só os cálculos. Deixamos isso como exercício.

Por outro lado, lembrando que para um mesmo problema existem várias soluções, vamos apresentar um outro algoritmo que é baseado na própria definição de números binários.[1]

De fato, o número decimal 22 pode ser escrito como uma 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 7.6.

Por exemplo, para o número 22, qual é a maior potência de 2 que é menor do que 22? É o $2^4 = 16$, pois $2^5 = 32$ é maior que 22. Então para saber o bit de mais alta ordem basta subtrair $22 - 16 = 6$. Como esta diferença não é negativa, significa que a potência 2^4 é usada na soma de potências de 2 que constituem o número 22, portanto o bit de mais alta ordem é 1.

Para continuidade dos cálculos, como já sabemos que 2^4 faz parte do número, temos que saber quais são as potências de 2 que constituem a diferença $22 - 16 = 6$. A próxima potência de 2 menor que $2^4 = 16$ é $2^3 = 8$. Mas 8 é maior que 6, e portanto

[1]Este exemplo nos ocorreu após trocas de emails com o Allan Neves, então aluno da disciplina. Agradecemos a ele por isto.

$6-8=-2<0$, o que significa que 2^3 não faz parte das potências de 2 que constituem o 22. Logo, este bit relativo ao 2^3 é zero.

O próximo bit deve ser relativo à potência $2^2=4$. A diferença $6-4=2\geq 0$, o que indica que 2^2 faz parte de 22, portanto este bit deve ser 1.

Para continuar, temos que saber a próxima potência de 2, que agora é $2^1=1$ com relação à diferença $6-4=2$. Ora, $2-2=0\geq 0$, e assim o bit relativo ao 2^1 deve ser 1.

Finalmente, resta saber o bit relativo à potência $2^0=1$. Mas $0-1=-1<0$, consequentemente este bit deve ser zero. Neste momento, não há mais potências de 2 a considerar e o algoritmo gerou todos os bits corretamente, na sequência certa, o que resultou no binário 10110.

Notem no algoritmo da figura 7.6 como se inicia pela potência de 2 mais alta ordem e a cada entrada no corpo do laço faz as contas e no final divide a potência de 2 "da vez" por 2. Quando este valor for nulo, o algoritmo termina.

É possível também construir esta maior potência de 2, bastaria um laço que inicia em 1 e vai multiplicando por 2 sucessivamente até encontrar um valor maior do que o número de entrada. Esta é a potência de 2 que deve ser usada no restante do código. Deixamos como exercício ao leitor, mas é muito parecido com o algoritmo de contar dígitos mostrado na figura 7.4.

```
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 7.6: Convertendo para binário, versão 2.

7.3 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 de Euclides faz divisões sucessivas de um número pelo outro e, baseado nos restos da divisões, calcula o MDC com pouquíssimos cálculos, permitindo um algoritmo extremamente simples de ser implementado. Vejamos o que Euclides nos ensinou. Os números de entrada são colocados em uma grade, assim:

135	72					

Então se procede a divisão inteira de 135 por 72, anotando-se o quociente desta divisão, obtido pelo cálculo `135 div 72 = 1`, acima do 72 e o resto da divisão inteira, obtido pelo cálculo `135 mod 72 = 63`, abaixo do 135, desta forma:

	1					
135	72					
63						

Neste momento é preciso testar se este resto é nulo, o que não é o caso. O algoritmo vai terminar quando este valor for nulo. Então procede-se da seguinte maneira: copia-se o 63 no lado direito do 72 e volta-se a calcular divisão inteira e resto de divisão inteira, mas desta vez não com 135 e 72, mas com 72 e 63, assim:

	1	1				
135	72	63				
63	9					

Observem que 9 é o resto da divisão inteira de 72 por 63 e 1 é o quociente. Como 9 ainda não é nulo, repete-se o processo, isto é, copia-se o 9 no lado direito do 63 e refaz-se os cálculos do mesmo modo, assim:

	1	1	7			
135	72	63	9			
63	9	0				

Desta vez o resto da divisão é nulo. Portanto, o valor do segundo operando da divisão, no caso 9, é o MDC entre 135 e 72! De fato, este algoritmo é correto e converge (encontra a solução) em pouquíssimos passos.

O algoritmo que implementa o processo pode ser visto atualmente em *Pascal* conforme está ilustrado na figura 7.7. Notem que a técnica utilizada é a de lembrar sempre dois números: inicia com A e B, o teste do laço é se o resto da divisão inteira de A por B é nulo. Enquanto não for, A recebe o valor de B e B recebe o valor do resto, nesta ordem. Quando o resto for nulo, o MCD estará contido na variável B.

```
program mdcporeuclides;
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 7.7: 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. O algoritmo funciona também se A for menor do que B, vejam:

72	135					

Após a primeira rodada do laço, vejam como fica:

	0					
72	135	72				
72						

7.4 Tabuada

Problema: Imprimir as tabuadas do 1 ao 10.

Este problema vai nos permitir pela primeira vez estudar o aninhamento de comandos de repetição. Já tínhamos visto aninhamento de desvios condicionais, na seção 6.4. O aninhamento de laços tende a ser mais complexo para um aprendiz, por este motivo escolhemos este problema da tabuada, que é muito simples do ponto de vista de matemática, e vai permitir uma boa compreensão por parte do estudante.

O problema que é simples é imprimir a tabuada de um único número, por exemplo o 7, cuja solução algoritmica pode ser vista na figura 7.9:

$7 \times 1 = 7$
$7 \times 2 = 14$
$7 \times 3 = 21$
$7 \times 4 = 28$
$7 \times 5 = 35$
$7 \times 6 = 42$
$7 \times 7 = 49$
$7 \times 8 = 56$
$7 \times 9 = 63$
$7 \times 10 = 70$

Figura 7.8: Tabuada do 7.

```
program tabuada_do_7;
var j: integer;

begin
    j:= 1;
    while j <= 10 do
    begin
        writeln (7,'x',j,'= ',7*j);
        j:= j + 1;
    end;
end.
```

Figura 7.9: Tabuadas do 7.

É preciso um laço que executa 10 vezes. Mas para o problema de imprimir a tabuada de todos os números entre 1 e 10, evidentemente não se imaginando 10 laços um após o outro, o que seria um absurdo de código repetido, deve-se usar dois laços *aninhados*, um no escopo do outro. Isto torna necessário termos duas variáveis de controle, uma para controlar qual é o número da vez no cálculo da sua tabuada, o outro controla as multiplicações deste número de 1 até 10.

No código apresentado na figura 7.10, para cada valor de i (tabuada do i), os valores de j variam de 1 a 10 (fazendo o cálculo da tabuada do i propriamente dita), 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.

```
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 eh executado apenas 10 vezes *)
              i:= i + 1;
        end;
end.
```

Figura 7.10: Tabuadas do 1 ao 10.

É mais fácil perceber isso quando se vê a saída deste programa. A figura 7.11 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.

Notem como a variável i no segundo programa substitui o 7 do primeiro programa. Ocorre que o i sofre variação de 1 até 10, o que produz como efeito a impressão de todas as tabuadas do 1 ao 10.

$1 \times 1 = 1$	$2 \times 1 = 2$	$\ldots$	$10 \times 1 = 10$
$1 \times 2 = 2$	$2 \times 2 = 4$	$\ldots$	$10 \times 2 = 20$
$1 \times 3 = 3$	$2 \times 3 = 6$	$\ldots$	$10 \times 3 = 30$
$\vdots$	$\vdots$	$\ldots$	$\vdots$
$1 \times 10 = 10$	$2 \times 10 = 20$	$\ldots$	$10 \times 10 = 100$

Figura 7.11: Saída do programa da tabuada.

7.5 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 conhecidas, 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 7.12. Formatamos a saída de maneira especial para facilitar o argumento que queremos com este exemplo.

```
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 7.12: 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. Isto foi feito no algoritmo da tabuada, na seção 7.10. Esta solução é apresentada na figura 7.13.

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 2×1, calculou 5 vezes a multiplicação de 3×2, calculou 4 vezes a de 4×6 e assim por diante. O programador não explorou corretamente os cálculos já feitos anteriormente.

```
program fatorial1_n;
var cont, i, n, fat: integer;

begin
        read (n);
        cont:= 1;
        while cont <= n do
        begin
             write ('fat(',cont,')= ');
             fat:= 1; (* inicializacao 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;
             writeln (fat);
             cont:= cont + 1;
        end;
end.
```

```
Saída do programa com entrada n=7:

fat(1)= 1= 1
fat(2)= 2x1= 2
fat(3)= 3x2x1= 6
fat(4)= 4x3x2x1= 24
fat(5)= 5x4x3x2x1= 120
fat(6)= 6x5x4x3x2x1= 720
fat(7)= 7x6x5x4x3x2x1= 5040
```

Figura 7.13: Obtendo vários fatoriais.

Em outras palavras, o programa pode ficar mais eficiente se for feito como ilustrado na figura 7.14, 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: $fat(n+1) = (n+1) * fat(n)$. Como $fat(1) = 1$, por definição, basta construir iterativamente o fatorial de n a partir do fatorial de 1.

```
program fatorial1_n_v2;
var cont, n, fat: integer;

begin
        read (n);
        cont:= 1;
        fat:= 1; (* inicializacao 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:

fat(1)= 1
fat(2)= 2
fat(3)= 6
fat(4)= 24
fat(5)= 120
fat(6)= 720
fat(7)= 5040
```

Figura 7.14: Otimizando o cálculo de vários fatoriais.

7.6 Números de Fibonacci revisitado

Na seção 6.6 vimos como calcular números da sequência de Fibonacci. Nesta seção veremos outros problemas relacionados. O objetivo é mostrar critérios alternativos de saída de um laço, pois as vezes, mesmo que os cálculos principais sejam os mesmos, o objetivo do problema é outro.

7.6.1 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 7.15. A diferença básica é que neste caso não é preciso contar os números, pois o critério de parada é diferente, pois o problema é diferente, mas similar. É importante observar que a parte da geração dos números da sequência não mudou.

```
program Fibonacci_2;
const max=1000;
var ultimo, penultimo, soma: integer;
begin
      ultimo:= 1;
      penultimo:= 1;
      soma:= penultimo + ultimo;
      while soma <= max do (* termina quando atinge o valor desejado *)
      begin
            penultimo:= ultimo;
            ultimo:= soma;
            soma:= penultimo + ultimo;
      end;
      writeln (soma);
end.
```

Figura 7.15: Imprimindo o primeiro número de Fibonacci maior do que 1000.

Observem que nesta versão não é necessário usar a variável `cont`, pois não queremos mais contar os números, queremos apenas saber quando o último número gerado atingiu o valor desejado.

7.6.2 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 φ, e é aproximadamente 1.6180339887499. De fato, vejamos a razão entre dois termos consecutivos para alguns números pequenos:

$$\frac{1}{1} = 1, \frac{2}{1} = 2, \frac{3}{2} = 1.5, \frac{5}{3} = 1.66, \frac{8}{5} = 1.60, \frac{13}{8} = 1.625, \frac{21}{13} = 1.615, \ldots$$

O algoritmo que calcula o número áureo com a precisão desejada, mostrando os valores intermediários, é apresentado na figura 7.16. 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.

```
program numero_aureo;
const PRECISAO=0.00000000000001;
var ultimo, penultimo, soma: integer;
    naureo, naureo_anterior: real;
begin
     ultimo:= 1;       (* inicializacao das variaveis 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 7.16: Verificando a convergência do número áureo.

Este tipo de critério de parada é extremamente importante, isto é, quando se quer encontrar um valor aproximado que depende de um *erro* pré estabelecido.

Isto é relacionado com a *convergência* do número, o que é um conceito muito estudado em Cálculo Diferencial e Integral, por exemplo. Convergir significa que os número intermediários obtidos nos cálculos estão cada vez mais próximos uns dos outros, o que nos leva a ter a oportunidade de definirmos, por exemplo, com quantas casas decimais estaremos satisfeitos.

Por exemplo, o número π pode ser matematicamente calculado, mas quantas casas decimais queremos? Seria suficiente aceitar 3.14? ou 3.14159?

O caso contrário são os algoritmos que não convergem e que tornariam a execução do programa infinita. Nestes casos é importante também que se tenha um critério alternativo de saída do programa: se o laço executar um certo número de vezes, digamos, mil, e não chegar no erro estabelecido, então o programa deve encerrar com uma mensagem de erro informando esta situação.

Este assunto será muito estudado nas disciplinas de Métodos Numéricos, na qual são estudados métodos importantes para calcular zeros de funções, integrais numéricas, soluções de sistemas de equações lineares e até mesmo soluções para equações diferenciais.

Várias outras propriedades interessantes da sequência de Fibonacci serão deixadas como exercício.

7.7 Palíndromos

Nesta seção generalizamos a ideia central apresentada no problema de se inverter um número de três dígitos (seção 7.1).

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 $d_0 d_1 d_2 \ldots d_n$ é escrito da seguinte forma:

$$d_o \times 10^0 + d_1 \times 10^1 + d_2 \times 10^2 + \ldots + d_n \times 10^n$$

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

$$d_o \times 10^n + d_1 \times 10^{n-1} + d_2 \times 10^{n-2} + \ldots + d_n \times 10^0$$

O algoritmo da figura 7.17 generaliza o raciocínio usado no problema anterior e testa todos os números e imprime quando são palíndromos.

```
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 aculumador *)
            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 7.17: 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 7.18 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.

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

      pal:= 11;                           (* gerando todos de 2 digitos *)
      i:= 2;
      while i <= 9 do
      begin
            writeln (pal);
            pal:= i * 11;
            i:= i + 1;
      end;


      i:= 1;                           (* gerando todos os de 3 digitos *)
      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 7.18: Gerando todos os palíndromos de 1 a 1000.

7.8 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.

7.8.1 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 7.19 ilustra uma solução baseada nas já estudadas técnicas dos acumuladores e no problema dos fatoriais. 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;
begin
      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.
```

Figura 7.19: 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 7.20 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 7.20: Cálculo do número neperiano.

7.8.2 Cálculo do seno

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

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

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 7.21 modifica o anterior pela introdução de uma nova variável que controla a mudança do sinal, produzindo como saída o cálculo da função *tmp*. A parte central do algoritmo agora tem a seguinte operação:
`seno:= seno + sinal*novotermo;` e na linha logo abaixo, o sinal muda a cada iteração, produzindo os cálculos desejados.

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

```
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 variavel para sinal trocado *)
           fat:= i*fat;
           i:= i + 1;
      end;
      writeln ('tmp= ',seno);
end.
```

Figura 7.21: Série com troca de sinais.

Agora vamos corrigir o denominador, obtendo a seguinte série:

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

A atualização da variável fat deve ser modificada, conforme apontado na figura 7.22:

```
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 princial modificacao *)
           sinal:= -sinal;
           i:= i + 1;
      end;
      writeln ('tmp2= ',seno);
end.
```

Figura 7.22: Série com troca de sinais e fatorais 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 7.23 implementa corretamente o cálculo do $seno(x)$ 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.

```
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 7.23: Cálculo do seno de x.

7.9 Maior segmento crescente

Os problemas agora se tornam bem complexos, vão exigir combinações das técnicas elementares e bastante raciocínio antes de se apresentar um algoritmo. Teremos que procurar entender quais são os subproblemas do problema principal e tratá-los pouco a pouco, progressivamente, para no final termos a solução. Isso foi feito na construção do cálculo do seno, na seção 7.8.2 e, a partir de agora, esta técnica será bem importante.

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.

Quais são os subproblemas deste problema? Podemos inicialmente elencar alguns:

- ler os números de entrada;
- para cada número, descobrir se estamos em um segmento crescente ou não;
- memorizar qual é o tamanho do segmento crescente "da vez";
- saber qual é o tamano do segmento crescente de tamanho máximo.

Vamos usar uma técnica de construção de algoritmos que extrapola a linguagem de programação, ela usa frases em português e chamaremos isso de *pseudo-código*. Iniciamos pela leitura dos dados de entrada, conforme a figura 7.24.

```
program maior_seg_crescente_v1;
var n, a: integer;
begin
    read (n);
    i:= 1;
    while i <= n do
    begin
        read (a);
        i:= i + 1;
    end;
end.
```

Figura 7.24: Maior segmento crescente: lendo os números da entrada.

A próxima questão é saber se estamos em um segmento crescente ou não. Isto exige que se saiba qual é o número lido na iteração anterior para poder compará-lo com o que foi lido na iteração atual. Para isso precisamos de uma outra variável, para guardar a informação anterior. Esta cópia tem que ser feita antes da nova leitura. Vejamos uma tentativa na figura 7.25. Este algoritmo tem um problema: `a` não tem valor definido na primeira vez! Isto faz com que: ou o primeiro `read(a)` tem que

estar antes do laço; ou `a_anterior` tem que ser definida antes. A primeira solução não parece boa, pois `n` pode ser zero. Optamos pela segunda via, usando os fatos de que os números lidos são naturais e de que zero é o menor natural.

```
program maior_seg_crescente_v2;
var n, a, a_anterior: integer;
begin
    read (n);
    i:= 1;
    while i <= n do
    begin
        a_anterior:= a;
        read (a);
        if a > a_anterir then
            {estamos em um segmento crescente}
        else
            {acabou o segmento crescente atual}
        i:= i + 1;
    end;
end.
```

Figura 7.25: Maior segmento crescente: lembrando o número anterior.

O código fica como na figura 7.26: após a leitura de `a` já se faz o teste de comparação com `a_anterior`, que é atualizado no final do laço.

```
program maior_seg_crescente_v3;
var n, a, a_anterior: integer;
begin
    read (n);
    i:= 1;
    a_anterior:= 0;
    while i <= n do
    begin
        read (a);
        if a > a_anterir then
            {estamos em um segmento crescente}
        else
            {acabou o segmento crescente atual}
        a_anterior:= a;
        i:= i + 1;
    end;
end.
```

Figura 7.26: Maior segmento crescente: lembrando o número anterior.

Quanto ao subproblema *estamos em um segmento crescente*, é preciso incrementar uma variável que conta o tamanho e inicializá-la, conforme a figura 7.27.

```
program maior_seg_crescente_v4;
var n, a, a_anterior, tamanho: integer;
begin
    read (n);
    i:= 1;
    a_anterior:= 0;
    tamanho:= 0;     (* inicializa o tamanho *)
    while i <= n do
    begin
        read (a);
        if a > a_anterir then
            tamanho:= tamanho + 1
        else
            {acabou o segmento crescente atual}
        a_anterior:= a;
        i:= i + 1;
    end;
end.
```

Figura 7.27: Maior segmento crescente: controlando o tamanho da sequencia.

Com relação ao terceiro subproblema, memorizar o tamanho do segmento crescente atual, quando o teste do `if` falha é porquê o segmento atual acabou, então lembramos disso na parte do `else`, como mostrado na figura 7.28. A variável `tamanho` deve ser zerada neste ponto, pois a subsequência atual acabou.

```
program maior_seg_crescente_v5;
var n, a, a_anterior, tamanho, tamanho_atual: integer;
begin
    read (n);
    i:= 1;
    a_anterior:= 0;
    tamanho:= 0;
    while i <= n do
    begin
        read (a);
        if a > a_anterir then
            tamanho:= tamanho + 1
        else
        begin
            tamanho_atual:= tamanho; (* temos o tamanho da seq. atual *)
            tamanho:= 0;
        end;
        a_anterior:= a;
        i:= i + 1;
    end;
end.
```

Figura 7.28: Maior segmento crescente: memorizando o tamanho da sequencia.

Agora resta resolver o último subproblema e consequentemente o problema global. Temos que saber qual dos segmentos é o de maior valor. Para isso vamos usar a técnica elementar vista na seção 6.5. Temos que definir um valor a priori, que pode muito bem ser o zero, já que é o menor tamanho possível para uma sequência. Depois, na parte do `else`. corrigimos a informação caso seja necessário.

O algoritmo que resolve este problema é apresentado na figura 7.29.

```
program maior_seg_crescente;
var n, tamanho, maiortam, a_anterior, i, a: integer;
begin
     read (n);                    (* inicializa as variaveis 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: ',maiortam);
end.
```

Figura 7.29: Maior segmento crescente.

A solução exige um conjunto de variáveis que controlam o estado da computação, isto é, que mantêm 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 *maiortam* registra a maior sequência crescente que já foi processada até o momento.

7.10 Primos entre si

Este problema ao mesmo tempo que serve para motivar para o próximo capítulo, serve também para mostrar como um problema já resolvido, e que é um subproblema de um problema novo, pode ajudar na solução deste último. Mas é preciso tomar cuidado com estas integrações de problemas, conforme já foi mostrado na seção 7.5. No problema aqui tratado este não é o caso, pois a solução é eficiente.

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 7.30 contém um código global que gera todos os pares ordenados que interessam. Observe que o `if {i e j sao primos entre si} then` 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 7.30: 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 7.3 vimos como se calcula de forma eficiente o MDC entre dois números pelo método de Euclides (figura 7.7). Então temos que adaptar aquele código neste,

inserindo-o com cuidado na nossa versão parcial e teremos a solução global, que pode ser visto na figura 7.31.

```
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 significa primos entre si *)
                j:= j + 1;
        end;
        i:= i + 1;
    end;
end.
```

Figura 7.31: Gerando todos os primos entre si.

7.11 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 7.32, 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.

```
program ehprimo;
var n, cont, i: integer;
begin
      read (n);
      cont:= 0; (* contador de divisores de n *)
      i:= 2;
      while i <= n-1 do
      begin
            if n mod i = 0 then
                  cont:= cont + 1; (* achou um divisor *)
            i:= i + 1;
      end;
      if cont = 0 then
            writeln (n,' eh primo')
end.
```

Figura 7.32: 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 7.33.

```
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 7.33: Testa se n é primo parando no primeiro divisor.

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.

No segundo caso, o algoritmo vai terminar bem rápido. No outro, ele vai testar todas as possibilidades. 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.

Mas é possível melhorar este algoritmo, lembrando que só existe um único número primo que é par, este número é o 2. Todos os outros primos são ímpares. Logo, pode-se testar o 2 separadamente e depois testar os ímpares. Com isso, somente os ímpares são testado, o que, em tese, deveria cortar pela metade os testes feitos.

Infelizmente, não existe algoritmo que, dado um primo, diga quem é o próximo primo. Portanto testar todos os ímpares é inevitável. Mas existe ainda uma última melhoria: não é preciso testar *todos* os ímpares entre 1 e $n-1$. De fato, basta testar até a *raiz* do número de entrada.

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: $n = p * p$, em que p é primo. O algoritmo que implementa esta solução é mostrado na figura 7.34.

Para se ter uma ideia do ganho, vejam na tabela seguinte o quanto se ganha com as três últimas versões do programa.

x	$\sqrt{x}$
1000000	1000
1000000000	31622
1000000000000	1000000

A tabela acima mostra que para entradas da ordem de 10^{12}, o número de cálculos feitos com o programa da figura 7.33 pode ser da mesma ordem de 10^{12}. Os cálculos

```
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 7.34: Testa se n é primo parando na raiz de n.

do programa da última versão mostra que o programa pode fazer cálculos da ordem de "somente" 10^6.

Apêndice: cálculo do MDC pela definição

Neste apêndice 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 7.7. Também tem como objetivo mostrar que é possível escrever um código com uma técnica menos eficiente, o que não faz muito sentido a não ser praticar a arte de programação. O leitor pode pular este texto sem maiores problemas.

O MDC entre dois inteiros a e b foi definido matematicamente na seção 7.3 e envolve a fatoração de ambas as entradas como um produto de números primos. O algoritmo básico, em pseudo-código é apresentado na figura 7.35.

```
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 7.35: Pseudo-código para o calculo 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 7.36 e 7.37.

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 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
      (* inicializacao das variaveis 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 eh 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;
```

Figura 7.36: Calcula MDC entre a e b pela definição (caso primo=2).

```
     (* repete o processo para todos os impares *)
     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 eh 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 proximo impar *)
          primo:= primo + 2;
     end;

     (* imprime o resultado final *)
     writeln (mdc);
end.
```

Figura 7.37: 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 inteiro positovo N e imprima os N primeiros termos da progressão geométrica seguinte:

$$1, 2, 4, 8, 16, 32, \ldots$$

Exemplos de entradas	Saídas esperadas
3	1 2 4
6	1 2 4 8 16 32
8	1 2 4 8 16 32 64 128

2. 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 dele no conjunto.

Exemplos de entradas	Saídas esperadas
1026	4
1034	9
9876	4536
1243	72

3. Faça um programa em *Pascal* que leia um número inteiro positivo N e em seguida leia uma sequência de N números reais $x_1, \ldots, x_n$ e imprima o quociente da soma destes números reais pelo seu produto. Isto é, imprima o valor de q, com duas casas decimais, onde:

$$q = \frac{\sum_{i=1}^{N} x_i}{\prod_{i=1}^{N} x_i}$$

Como não pode haver divisão por zero, seu programa deve parar tão logo esta situação seja verificada indicando uma mensagem apropriada para o usuário.

Exemplos de entradas	Saídas esperadas
3 1 2 3	 1.00
4 1 2 0 4	 divisao por zero

4. Em *Pascal* o tipo *char* é enumerável, e portanto está na classe dos tipos chamados de *ordinais*, conforme o guia de referência da linguagem. A ordem de cada caractere é dada pela tabela ASCII. Assim é possível, por exemplo, escrever trechos de código tais como:

```
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:

```
FOR i:= 'a' TO 'z' DO
     WRITE (i);
```

que produziria como saída "abcdefghijklmnopqrstuvxwyz".

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:

Exemplos de entradas	Saídas esperadas
FABIANO SILVA.	Vogais: 6 Consoantes: 6
MARCOS ALEXANDRE CASTILHO.	Vogais: 9 Consoantes: 14

5. Faça um programa em *Pascal* que leia dois números naturais m e n e imprima, dentre todos os pares de números naturais (x, y) tais que $1 \leq x \leq m$ e $1 \leq y \leq n$, um par para o qual o valor da expressão $xy - x^2 + y$ seja máximo e imprima também o valor desse máximo.

Exemplos de entradas	Saídas esperadas
2 2	1 2 3
4 4	2 4 8
5 2	1 2 3

6. Faça um programa em *Pascal* que leia um inteiro positivo n e imprima a soma dos n primeiros termos da série:

$$\frac{1000}{1} - \frac{997}{2} + \frac{994}{3} - \frac{991}{4} + \ldots$$

7. Faça um programa em *Pascal* que calcule e imprima o valor de S:

$$S = \frac{37 \times 38}{1} + \frac{36 \times 37}{2} + \frac{35 \times 36}{3} + \cdots + \frac{1 \times 2}{37}$$

8. Faça um programa em *Pascal* que calcule e escreva o valor de S assim definido:

$$S = \frac{1}{1!} - \frac{2}{2!} + \frac{4}{3!} - \frac{8}{2!} + \frac{16}{1!} - \frac{32}{2!} + \frac{64}{3!} - \cdots$$

9. Faça um programa em *Pascal* que calcule e imprima o resultado da seguinte série:

$$S = \frac{x^0}{2!} - \frac{x^4}{6!} + \frac{x^8}{10!} - \frac{x^{12}}{14!} + \frac{x^{16}}{18!} - \ldots$$

10. Faça um programa em *Pascal* que leia um número real x, calcule e imprima o valor de $f(x)$:

$$f(x) = \frac{5x}{2!} - \frac{6x^2}{3!} + \frac{11x^3}{4!} - \frac{12x^4}{5!} + \frac{17x^5}{6!} - \frac{18x^6}{7!} + \ldots$$

O cálculo deve parar quando $abs(f(x_{n+1}) - f(x_n)) < 0.00000001$, onde $abs(x)$ é a função em *Pascal* que retorna o valor absoluto de x.

11. Sabe-se que o valor do cosseno de 1 (um) radiano pode ser calculado pela série infinita abaixo:

$$cosseno(x) = \frac{1}{0!} - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \ldots$$

Faça um programa em *Pascal* que leia um número real x do teclado, interpretado como sendo um ângulo em radianos, calcule e imprima o valor do cosseno de x obtido pela série acima considerando somente os primeiros 14 termos da mesma.

12. O número áureo φ (1,6180339...) pode ser calculado através de expressões com séries de frações sucessivas do tipo:

$$\begin{aligned}
\varphi_1 &= 1 + \frac{1}{1} = 2 \\
\varphi_2 &= 1 + \frac{1}{1 + \frac{1}{1}} = 1,5 \\
\varphi_3 &= 1 + \frac{1}{1 + \frac{1}{1 + \frac{1}{1}}} = 1,666 \\
\varphi_4 &= 1 + \frac{1}{1 + \frac{1}{1 + \frac{1}{1 + \frac{1}{1}}}} = 1,6
\end{aligned}$$

onde φ_i 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 inteiro N e imprima o valor da aproximação do número áureo φ_N, que usa uma série de N frações sucessivas.

13. Faça um programa em *Pascal* que leia um número de três dígitos e construa outro número de quatro dígitos com a seguinte regra:

- os três primeiros dígitos, contados da esquerda para a direita, são iguais aos do número dado;
- o quarto dígito é um dígito verificador 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 no formato mostrado nos exemplos.

Exemplos de entradas	Saídas esperadas
123	1231
001	0015
100	1001
101	1016

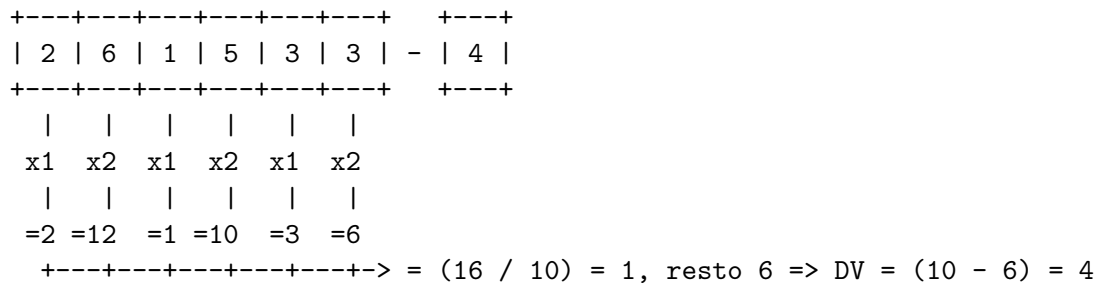
14. Faça um programa em *Pascal* que leia um número inteiro de cinco dígitos representando um número binário, determinar seu valor equivalente em decimal.

Exemplos de entradas	Saídas esperadas
10001	17
00001	1
11111	31

15. 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.

Exemplo de entrada	Saída esperada
2615334 2615332	2615332

16. 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, com duas casas decimais, dos múltiplos de 3 existentes entre A(inclusive) e B(inclusive).

Exemplos de entradas	Saídas esperadas
3 3 4 4 3 5	 8
9 1	6.00

17. 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. O zero não deve ser processado e serve para marca o final da entrada de dados.

Exemplos de entradas	Saídas esperadas
2 2 3 3 5 1 1 1	4
1 2 3 4 5	5
8 8 8 9 1 1 1 1	3

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 de entrada	Saída esperada
10011	19
121	numero nao binario
1010	10
101010101	341
0	0
-1	O maior numero foi 341

19. (*) Faça um programa em *Pascal* que leia um inteiro N e o imprima como um produto de primos.

Exemplos de entradas	Saídas esperadas
45	3 3 5
56	2 2 2 7

20. (*) Faça um programa em *Pascal* que leia um número inteiro N e imprima o maior divisor de N que é uma potência de um dos números primos fatorados. Exemplos:

    $N = 45 = 3^2.5^1$ escreve $9 = 3^2$

    $N = 145 = 5^2.7^1$ escreve $25 = 5^2$

    $N = 5616 = 2^4.3^3.13$ escreve $27 = 3^3$

Exemplo de entrada	Saída esperada
45	9
145	25
5616	27

21. (*) 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.

    Exemplos:

    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.

    Faça um programa *Pascal* que leia dois números inteiros m e n e imprima *sim* se são amigos quadráticos e *nao* caso contrário.

Exemplos de entradas	Saídas esperadas
13 16	sim
13 15	nao

22. (*) 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 leia dois números inteiros k e n e em seguida leia uma sequência de outros n números inteiros e verifique se esta sequência de tamanho n é k-alternante. É garantido que o usuário entrará um valor de n múltiplo de k. A saída do programa deve ser a mensagem *A sequencia eh k-alternante* caso a sequência seja k-alternante e *A sequencia não eh k-alternante*, caso contrário.

Exemplos de entradas	Saídas esperadas
2 12 1 3 6 8 9 11 2 4 1 7 6 8	a sequencia eh 2-alternante.
1 7 2 1 4 7 8 9 12	a sequencia eh 1-alternante.
3 3 1 3 5	a sequencia eh 3-alternante.
2 8 1 2 3 3 4 6 3 2	a sequencia nao eh 2 alternante

7.13 Exercícios de prova

No link abaixo você encontra diversas provas aplicadas que cobrem os conteúdos vistos até aqui:

- `http://www.inf.ufpr.br/cursos/ci055/prova1.html`

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.[1]

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, deveriam idealmente implicar em um código legível, isto é, compreensível para quem o lê e até mesmo para quem o escreveu.[2]

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.

---

[1] Niklaus Wirth, o autor da linguagem *Pascal*, criou uma linguagem modular chamada *Modula 2* e depois também criou outra orientada à objetos, chamada *Oberon*.

[2] Recomendamos a leitura do miniguia da linguagem *Pascal*, disponível no site oficial da disciplina CI055.

Neste capítulo 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 perguntas que os estudantes fazem que precisam ser esclarecidas:

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: $a \bmod 2 = 0$. 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 impar. 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:

```
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.

```
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 "$a \bmod 2 = 0$".

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.

```
(* primeira versao da funcao *)
if a mod 2 = 0 then
     a_eh_par:= true
else
     a_eh_par:= false;
```

```
(* segunda versao da funcao *)
if a mod 2 <> 0 then
     a_eh_par:= false
else
     a_eh_par:= true;
```

```
(* terceira versao da funcao *)
if a mod 2 <> 1 then
     a_eh_par:= true
else
     a_eh_par:= false;
```

```
(* quarta versao da funcao *)
if a mod 2 = 1 then
     a_eh_par:= false
else
     a_eh_par:= true;
```

```
(* quinta versao da funcao *)
a_eh_par:= false;
if a mod 2 = 0 then
     a_eh_par:= true
```

```
(* sexta versao da funcao *)
a_eh_par:= true;
if a mod 2 = 1 then
     a_eh_par:= false
```

```
(* setima versao da funcao *)
a_eh_par:= true;
if a mod 2 <> 1 then
    a_eh_par:= true
```

```
(* oitava versao da funcao *)
a_eh_par:= true;
if a mod 2 <> 0 then
    a_eh_par:= false
```

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*.

```
program imprime_pares;
var a: integer;

(* funcao que determina se o parametro n 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:

```
function eh_par (var n: integer): boolean;
```

A diferença sintática é caracterizada pela palavra *var* antes do nome do identificador do parâmetro.[3] 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.

[3]Com 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[4]:

[4]Um exercício interessante é passar os parâmetros por valor e compreender o efeito disso.

```
program imprimetrocado;
var x,y,temp: real; (* variaveis globais *)

(* procedimento que troca os conteudos da variaveis *)
procedure troca (var a, b: real);
begin
      temp:= a;
      a:= b;
      b:= temp;
end;

begin (* programa principal *)
      read (x,y);
      troca (x,y);
      writeln (x,y);
end.
```

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.

```
program imprimetrocado;
var x,y: real; (* variaveis globais *)

(* procedimento que troca os conteudos da variaveis *)
procedure troca (var a, b: real);
var temp: real; (* variavel local, temporaria para uso exclusivo neste procedimento *)
begin
      temp:= a;
      a:= b;
      b:= temp;
end;

begin (* programa principal *)
      read (x,y);
      troca (x,y);
      writeln (x,y);
end.
```

Figura 8.6: Versão com uma variável local.

8.3 Alguns exemplos

Nesta seção vamos implementar alguns problemas simples para exercitar.

8.3.1 Primos entre si, revisitado

Na seção 7.10 vimos como integrar o código do algoritmo de Euclides para resolvermos o problema se dados dos números, se eles eram primos entre si.

Pois bem, vamos mostrar novamente o pseudocódigo da figura 7.30 para podermos fazer a explicação do novo conceito, isto está na figura 8.7.

```
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 8.7: Pseudocódigo para o problema dos primos entre si.

Aquele pseudocódigo que consta na linha 8, aqui em destaque:

```
if {i e j sao primos entre si} then
```

foi subsitutído naquela capítulo pela injeção do código no próprio programa. Mas se, por outro lado, tivéssemos a disposição uma função que pudesse calcular o MDC entre dois números, poderíamos ter feito simplesmente assim:

```
    if primos_entre_si (i,j) then
                writeln (i,j);
```

Desta forma o programa pode ser escrito conforme a figura 8.8.
É importante observar alguns detalhes nesta versão:

- o código do cálculo do MDC ficou separado, não é misturado no meio do código do programa principal. Se algum dia alguém descobrir um método melhor do que o do Euclides, basta trocar na função e o programa principal não muda;
- os nomes de parâmetros a e b usados na função não importam, mas sim o fato de elas serem do tipo *iteger*, uma vez que elas recebem cópias dos valores de, respectivamente, i e j, que são passados pelo programa principal;

```
program primosentresi_v2;
var i, j: integer;

function primos_entre_si (a, b: integer): boolean;
var resto: integer;
begin
    primos_entre_si:= false;
    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;
        primos_entre_si:= true;
    end;
end;

begin
    i:= 2;
    while i <= 100 do
    begin
        j:= i;
        while  j <= 100 do
        begin
            if primos_entre_si (i,j) then
                writeln (i,j);
            j:= j + 1;
        end;
        i:= i + 1;
    end;
end.
```

Figura 8.8: Gerando todos os primos entre si, versão 2.

- a variável *resto*, que agora pode ser uma variável local à função, não precisando ser conhecida no programa principal.

8.3.2 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  x1  x2  x1  x2  x1
  |   |   |   |   |   |
 =4  =6  =2  =5  =6  =3
  +---+---+---+---+---+-> = 26
```

Como 26 tem dois dígitos, vamos repetir o processo acima até gerarmos um número de um único dígito. Assim:

```
+---+---+
| 2 | 6 |
+---+---+
  |   |
 x2  x1
  |   |
 =4  =6
  +---+ = 10
```

Como 10 ainda tem dois dígitos, o algoritmo roda ainda mais uma vez:

```
+---+---+
| 1 | 0 |
+---+---+
  |   |
 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 8.9 ilustra uma possível solução.

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.

```
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 8.9: Calculando dígito verificador.

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 estaríamos estragando o valor da variável que será ainda usada no programa principal.

8.3.3 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 8.10 contém o código que resolve este problema.

```
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 8.10: Calculando raízes de equação do segundo grau.

A figura 8.11 ilustra o programa principal modificado para se dar a ideia de que as funções se comportam como expressões aritméticas, ao contrário dos procedimentos, que se comportam como comandos. De fato, basta observar que as funções são chamadas dentro de um comando de impressão.

```
begin (* programa principal *)
      ler (a, b, c); (* garante-se que a nao eh nulo *)
      delta:= calcula_delta (a, b, c);
      if delta >= 0 then
            writeln (menor_raiz (a, b, delta), maior_raiz (a, b, delta))
      else
            writeln ('raizes complexas');

end.
```

Figura 8.11: 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 raízes[5]. Esta função é apresentada na figura 8.12.

```
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 8.12: 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 8.13.

[5]Agradecemos ao prof. Luis Carlos Erpen de Bona pela dica.

```
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 8.13: Versão final do programa.

8.3.4 Cálculo do MDC pela definição

Nesta seção vamos revisitar o cálculo do MDC pela definição estudado na seção 7.11. Deixamos propositalmente 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 7.36 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 7.37 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_que_divide(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 8.14.

```
program mdcpeladefinicao; (* pela definicao 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;
(* codigo da funcao num_vezes_que_divide *)

function potencia(n,p : longint): longint;
(* codigo da funcao 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 8.14: Calcula MDC entre a e b pela definição usando funções.

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 8.15 e 8.16 mostram sugestões de código para as funções.

```
function num_vezes_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_vezes_que_divide:= cont;
end;
```

Figura 8.15: 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 8.16: 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_vezes_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 parâmetros dois números inteiros não nulos e retorne *true* se um for o contrário do outro e *false* em caso contrário. Teste sua função usando este código:

```
program contrario;
var n,m: integer;

(* coloque aqui o codigo da sua funcao *)

begin
    read (n,m);
    if contrario (n,m) then
        writeln (n,'eh o contrario de ',m)
    else
        writeln (n,'nao eh o contrario de ',m);
end.
```

Exemplos de entradas	Saídas esperadas
123 321	123 eh o contrario de 321
123 231	123 nao eh o contrario de 231

2. Faça uma função em *Pascal* que receba como parâmetro um número inteiro e teste se ele é um número binário. Se ele for binário, imprima *sim* senão imprima *nao*. Teste sua função usando este código:

```
program testa_binario;
var n: integer;

(* coloque aqui o codigo da sua funcao que testa se eh binario *)

begin
    read (n);
    if eh_binario (n) then
        writeln ('sim')
    else
        writeln ('nao');
end.
```

Exemplos de entradas	Saídas esperadas
10001	sim
1020	nao

3. Faça uma função em *Pascal* que receba como parâmetro um número inteiro garantidamente binário e o converta para decimal. Teste sua função usando este código:

```
program converte;
var n: integer;

(* coloque aqui o codigo da sua funcao que testa se eh binario *)

(* coloque aqui o codigo da sua funcao que converte para binario *)

begin
    read (n);
    if eh_binario (n) then
        writeln (converte_em_binario (n))
    else
        writeln (n,'nao eh binario');
end.
```

Exemplos de entradas	Saídas esperadas
10001	17
1010	10

4. Faça uma função em *Pascal* que receba como parâmetro um número inteiro e retorne *true* se ele for primo e *false* em caso contrário. Teste sua função usando o código abaixo, que imprime todos os primos entre 1 e 10000.

```
program testa_se_primo;
var i: integer;

(* coloque aqui o codigo da sua funcao que testa se eh primo *)

begin
    read (n);
    for i:= 1 to 10000 do
        if eh_primo (i) then
            writeln (i);
end.
```

5. Faça duas funções em *Pascal* para calcular o seno e o cosseno de um número real lido do teclado representando um ângulo em radianos. Faça uma terceira função que calcule a tangente deste mesmo ângulo lido. Teste suas funções usando o código abaixo:

```
program calcula_tangente;
var angulo: real;

(* coloque aqui o codigo da sua funcao que calcula o seno *)

(* coloque aqui o codigo da sua funcao que calcula o cosseno *)

(* coloque aqui o codigo da sua funcao que calcula a tangente *)

begin
    read (angulo);
    writeln (tangente(angulo));
end.
```

6. Faça uma função em *Pascal* que receba como parâmetros seis inteiros *dia1, mes1, ano1, dia2, mes2, 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. Teste sua função usando o código abaixo.

```
program compara_datas;
var dia1, mes1, ano1, dia2, mes2, ano2: integer;

(* coloque aqui o codigo da sua funcao que compara as datas *)

begin
    read (dia1, mes1, ano1, dia2, mes2, ano2);
    if eh_anterior (dia1, mes1, ano1, dia2, mes2, ano2) then
        writeln ('a primeira data eh anterior')
    else
        writeln ('a primeira data nao eh anterior');
end.
```

7. Faça uma procedure em *Pascal* que receba como parâmetro um inteiro e retorne este número incrementado de uma unidade. Use esta procedure para fazer funcionar o código abaixo, que imprime todos os números de 1 a 10.

```
program incrementa_uma_unidade;
var n: integer;

(* coloque aqui o codigo da sua procedure que incrementa uma unidade *)

begin
    n:= 1;
    while n <= 10 do
    begin
        writeln (n);
        incrementa (n);
    end;
end.
```

8. Faça um programa em *Pascal* que receba os valores antigo e atual de um produto. Use uma função que determine o percentual de acréscimo entre esses valores. O resultado deverá ser mostrado pelo programa principal.

Exemplos de entradas	Saídas esperadas
10 15	0.34
100 110	0.10
134 134	0.00

9. Faça uma procedure em *Pascal* que receba como parâmetro uma string e que imprima na tela as palavras que ocorrem na string, uma por linha. As palavras podem estar separadas por brancos ou vírgulas, em qualquer quantidade. Você pode testar sua procedure usando o código abaixo:

```
program separa_palavras_das_frases;
var frase: string;

(* coloque aqui o codigo da sua procedure que separa palavras *)

begin
    readln (frase);
    separa_palavras(frase);
end.
```

Exemplo de entrada	Saída esperada
Estou, entendendo tudo,	Estou entendendo tudo

```
program media_alunos;
var n, p1, p2, p3, media: integer;

(* coloque aqui o codigo da sua funcao que calcula media ponderada *)

(* coloque aqui o codigo da funcao que decide pela aprovacao/reprovacao *)

begin
    read (n);
    for i:= 1 to n do
    begin
        read (p1, p2, p3);
        media:= media_ponderada (p1, p2, p3);
        if aprovado (media) then
            writeln ('aluno ',i,' aprovado com media: ', media)
        else
            writeln ('aluno ',i,' reprovado com media: ', media);
end.
```

10. Faça uma função em *Pascal* que calcule a média ponderada com pesos respectivamente 1, 2 e 3 para as provas 1, 2 e 3. Faça também outra função que decida se um aluno foi aprovado ou reprovado, sabendo que a aprovação deve ter a média final maior ou igual a 50. As notas das provas podem ser valores entre zero e 100. Você pode testar sua procedure usando o código abaixo:

```
program media_alunos;
var n, p1, p2, p3, media: integer;

(* coloque aqui o codigo da sua funcao que calcula media ponderada *)

(* coloque aqui o codigo da funcao que decide pela aprovacao/reprovacao *)

begin
    read (n);
    for i:= 1 to n do
    begin
        read (p1, p2, p3);
        media:= media_ponderada (p1, p2, p3);
        if aprovado (media) then
            writeln ('aluno ',i,' aprovado com media: ', media)
        else
            writeln ('aluno ',i,' reprovado com media: ', media);
end.
```

Exemplo de entrada	Saída esperada
3	
100 40 5	aluno 1 reprovado com media: 32
5 40 100	aluno 2 aprovado com media: 64
50 50 50	aluno 3 aprovado com media: 50

Parte II

Estruturas de Dados

Introdução da parte 2

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:

- vetores (ou *array* unidimencional);
- matrizes (ou *array* multidimencional);
- registros;

Nos capítulos seguintes explicaremos cada uma delas, sempre motivados por problemas que exigem seu uso ou que facilitam a implementação.

Também apresentaremos o conceito de *tipos abstratos de dados*, mostrando como os tipos básicos podem ser combinados para viabilizar um tipo de programação de alto nível.

É importante avisar ao leitor que, uma vez que os conceitos básicos já devem ter sido dominados (ou deveriam ter sido), então o texto flui com esta consideração. Vamos explorar conceitos vindos de outras áreas, sobretudo de matemática, e usar os conceitos de programação elementares já estudados na parte I deste livro.

Capítulo 9

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 aloca, 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-se 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.

9.1 Como funciona isto em memória?

A seguinte declaração em *Pascal* aloca 200 espaços em memória para números inteiros:

```
var v: array[1..200] of integer;
```

Em *Pascal* isto resulta na alocação de 200 vezes 2 bytes, dependendo de quanto espaço o compilador usa para um *integer*. 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 14×2 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.

9.2 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):

```
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:

```
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:

```
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:

```
read (v[1]); (* le do teclado e armazena na primeira posicao de v *)

i:= 10;
v[i+3]:= i * i; (* armazena o quadrado de i (100) na posicao 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 posicao 201 nao 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.

9.3 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.

9.3.1 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:

```
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.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	...	197	198	199	200
15	12	27	23	7	2	0	18	19	21	?	?	?	?	?	...	?	?	?	?

É importante, neste momento, atentar para alguns fatos:

1. este vetor tem 200 elementos, pois seu tamanho foi definido em tempo de compilação. 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;

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: *read*(v[1]), *read*(v[2]), *read*(v[3]), ..., recaindo nos problemas do início do curso;

4. a título de exemplo, mostramos a versão deste programa usando os comando *repeat* e *for*[1]. 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 *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 *repeat*.

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.

9.3.2 Imprimindo vetores

O programa ilustrado na figura 9.4 mostra como ler 10 números reais do teclado e imprimí-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.

É importante observar que este programa poderia ter sido resolvido sem o uso de vetores, como mostrado na figura 9.5.

[1]Ver o guia de referência da linguagem para entender estes dois novos comandos de repetição que são permitidos em *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.

```
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 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.

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

```
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.

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:

```
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*.

```
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.

Agora estamos prontos para resolver o problema proposto no início deste capítulo, aquele de ler uma sequência de números e imprimí-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

```
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.

principal trocando a chamada *imprimir(v)* por *imprimir_ao_contrario(v)*.

```
procedure imprimir_ao_contrario (var v: vetor);
var i: integer;
begin
     for i:= 10 downto 1 do
          write (v[i]);
end;
```

Figura 9.8: Procedimento que imprime os elementos do vetor ao contrário.

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 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.

```
program ler_e_imprimir_ao_contrario;
const min=0; max=50;
type vetor= array [min..max] of real;
var v: vetor;
    n: integer;

procedure ler (var v: vetor; var tam: integer); (* leitura *)
var i: integer;
begin
     read (tam);  (* 1<= tam<= 200, define o tamanho util do vetor *)
     for i:= 1 to tam do
          read (v[i]);
end;

procedure imprimir_ao_contrario (var v: vetor; tam: integer); (* impressao *)
var i: integer;
begin
     for i:= tam downto 1 do
          write (v[i]);
end;

begin (* programa principal *)
     ler (v, n);
     imprimir_ao_contrario (v, n);
end.
```

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:

```
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.

9.3.3 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.

```
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.

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 esta diferença é fundamental no estudo de vetores. Consideremos o seguinte vetor v com tamanho 10 como exemplo:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	...	197	198	199	200
15	12	27	23	7	2	0	18	19	21	?	?	?	?	?	...	?	?	?	?

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):

```
if eh_par (i) then  // no lugar de: if eh_par (v[i]) 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.

9.3.4 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.

```
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.

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.

9.4 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.

```
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
            if v[i] < menor then
                  menor:= v[i];
      menor_dos_lidos:= menor;
end;
```

Figura 9.12: Encontrando o menor de N números lidos, com vetor.

9.4.1 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:

	1	2	3	4	5	6
v:	2	6	1	5	3	3
	+	+	+	+	+	+
w:	1	3	2	4	3	5
	=	=	=	=	=	=
v+w:	3	9	3	9	6	8

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.

```
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;
```

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 $tam\_v = tam\_w$):

```
procedure soma_vetores (var v: vetor_i; tam_v: integer;
                        var w: vetor_i; tam_w: integer;
                        var soma_v_w: vetor_i; tam_soma: integer);
```

9.4.2 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:

	1	2	3	4	5	6
v:	2	6	1	0	3	3
	×	×	×	×	×	×
w:	1	0	2	4	3	5
	=	=	=	=	=	=
	2	0	2	0	9	15

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.

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.

9.5 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

```
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.

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 de inadimplentes, e por ai 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.

```
function busca_simples (x: real; var v: vetor_r; 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.

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 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_r; 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 apos 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 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[3]. Mas, para tornar a busca mais eficiente é preciso impor algumas

[3]Em disciplinas avançadas de algoritmos se estudam métodos bastante eficientes para este problema. No nosso caso veremos somente um deles.

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. 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 começa 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" lexicograficamente 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.

Versão	$n = 10$	$n = 10^2$	$n = 10^4$	$n = 10^8$
Busca simples (fig. 9.16)	20	200	20000	200000000
Busca com sentinela (fig. 9.17)	10	100	10000	100000000
Busca em vetor ordenado (fig. 9.18)	10	100	10000	100000000
Busca binária (fig. 9.20)	3	5	10	19

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.

9.5.1 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.

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:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	...	197	198	199	200
0	2	7	12	15	18	19	21	23	27	?	?	?	?	?	...	?	?	?	?

```
function busca_binaria (x: real; var v: vetor_r; n: integer): integer;
var inicio, fim, meio: integer;

begin
     inicio:=1;                          (* aponta para o inicio do vetor *)
     fim:= n;                            (* aponta para o fim do vetor    *)
     meio:= (inicio + fim) div 2;        (* aponta para o meio do vetor   *)
     while (v[meio] <> x) and (fim >= inicio) do
     begin
          if v[meio] > x then      (* vai jogar uma das duas metades fora  *)
               fim:= meio - 1      (* metade superior foi jogada fora      *)
          else
               inicio:= meio + 1;(* metade inferior foi jogada fora       *)
          meio:= (inicio + fim) div 2;   (* recalcula apontador para meio *)
     end;
     (* o laco termina quando achou ou quando o fim ficou antes do inicio *)
     if inicio <= fim then
          busca_binaria:= meio
     else
          busca_binaria:= 0;
end;
```

Figura 9.20: Busca binária.

```
procedure remove_vetor_ordenado (pos: integer; var v: vetor_r; var n: integer);
var i: integer;

begin
     for i:= pos to n-1 do
          v[i]:= v[i+1];
     n:= n - 1;
end;
```

Figura 9.21: Removendo de um 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:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	...	197	198	199	200
0	2	7	15	18	19	21	23	27	27	?	?	?	?	?	...	?	?	?	?

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.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	...	197	198	199	200
0	2	7	12	15	18	19	21	23	27	?	?	?	?	?	...	?	?	?	?

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:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	...	197	198	199	200
0	2	7	12	15	18	18	19	21	23	27	?	?	?	?	...	?	?	?	?

Observe que agora temos duas vezes o 18 no vetor. O primeiro deles agora pode ser substituído pelo novo elemento, o 17, assim:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	...	197	198	199	200
0	2	7	12	15	17	18	19	21	23	27	?	?	?	?	...	?	?	?	?

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.

```
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;
```

Figura 9.22: Inserindo em um vetor ordenado.

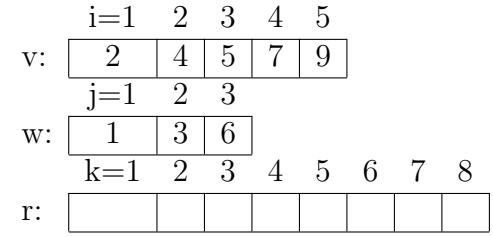
No caso de vetores não ordenados, os processos de inserção e remoção são mais simples. Pode-se sempre inserir um elemento novo no final do vetor e para remover,

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).

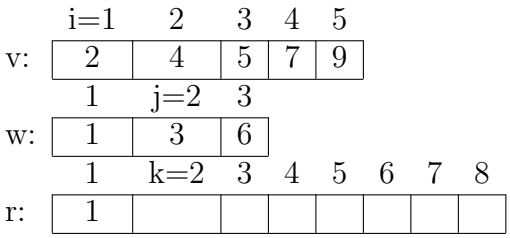
Há um último algoritmo interessante que trabalha com vetores ordenados: a ideia é fundir[4] dois vetores ordenados em um terceiro, obtendo-se um vetor ordenado como resultado contendo os mesmos elementos dos vetores anteriores.

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.

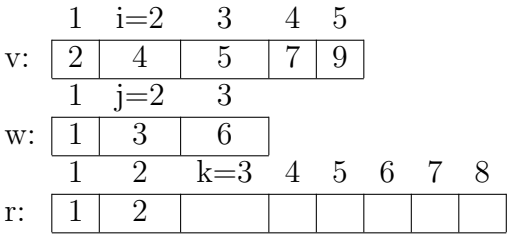
Vejamos isto através de um exemplo. Consideremos os seguintes dois vetores ordenados e um terceiro que vai receber a fusão dos dois.



comparam-se o elementos apontados por i e j. Como $1 < 2$, copia-se o 1 no vetor r e incrementa-se o apontador j, assim:

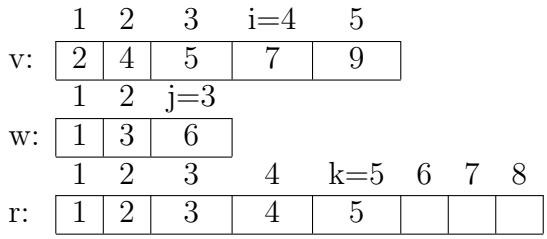


Agora comparam-se o elementos apontados por i e j. Como $2 < 3$, copia-se o 2 no vetor r e incrementa-se o apontador i, assim:

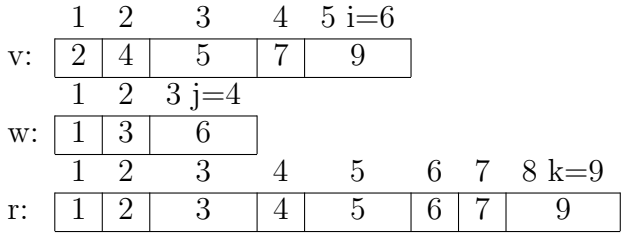


[4]Em inglês, *merge*

Repetindo-se este processo mais algumas vezes chegaremos no seguinte estado:



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:



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 dh 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.

9.6 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.

9.6.1 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.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	...	197	198	199	200
15	12	27	23	7	2	0	18	19	21	?	?	?	?	?	...	?	?	?	?

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.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	...	197	198	199	200
0	12	27	23	7	2	**15**	18	19	21	?	?	?	?	?	...	?	?	?	?

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:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	...	197	198	199	200
0	**2**	27	23	7	**12**	15	18	19	21	?	?	?	?	?	...	?	?	?	?
0	2	**7**	23	**27**	12	15	18	19	21	?	?	?	?	?	...	?	?	?	?
0	2	7	**12**	27	**23**	15	18	19	21	?	?	?	?	?	...	?	?	?	?
0	2	7	12	**15**	23	**27**	18	19	21	?	?	?	?	?	...	?	?	?	?
0	2	7	12	15	**18**	27	**23**	19	21	?	?	?	?	?	...	?	?	?	?
0	2	7	12	15	18	**19**	23	**27**	21	?	?	?	?	?	...	?	?	?	?
0	2	7	12	15	18	19	**21**	27	**23**	?	?	?	?	?	...	?	?	?	?
0	2	7	12	15	18	19	21	**23**	**27**	?	?	?	?	?	...	?	?	?	?

A figura 9.24 mostra a versão em *Pascal* deste algoritmo.

```
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 $N-1$.

9.6.2 Ordenação por inserção

A ordenação por inserção é na prática mais eficiente que a ordenação por seleção. Porém, 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
0	2	7	12	15	23	27	**18**	19	21	?	?	?	?	?	...	?	?	?	?

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 alguma 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
0	2	7	12	15	23	**23**	**27**	19	21	?	?	?	?	?	...	?	?	?	?

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
0	2	7	12	15	**18**	23	27	19	21	?	?	?	?	?	...	?	?	?	?

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.[5]

[5] O site `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 insercao (var v: vetor_r; n: integer);
var i, j: integer;
    aux: real;

begin
     for i:= 1 to n do
     begin
          aux:= v[i];

          (* abre espaco no vetor enquanto localiza a posicao 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.

9.7 Outros algoritmos com vetores

Nesta seção vamos apresentar alguns problemas interessantes que podem ser resolvidos usando-se a estrutura de vetores.

9.7.1 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.[6]

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 tem 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:

[6]Esta seção foi inspirada em uma preparatória para a maratona de programação da ACM da Ural State Universisy (Internal Contest October'2000 Junior Session) encontrada na seguinte URL: `http://acm.timus.ru/problem.aspx?space=1&num=1024`.

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 2 & 5 & 1 & 3 & 2 \end{pmatrix}$$

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:

1	2	3	4	5
4	1	5	2	3

1	2	3	4	5
2	5	1	3	2

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.

```
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[7] 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.

```
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

[7] Bruno Ricardo Sella

```
procedure gerar_permutacao (var v: vetor_i; 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_i; 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*).[8]

Em uma aula do segundo semestre de 2010 surgiu uma nova ideia para se gerar um vetor de permutação.[9]

A sugestão é modificar o algoritmo 9.29, fazendo com que um vetor auxiliar contenha 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 9.31 ilustra estas ideias. Ele foi implementado durante a aula e possibilitou gerar vetores de tamanhos incrivelmente grandes em tempo extremamente curto.[10]

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] 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] Créditos para o Felipe Z. do Nascimento.

[10] Código e testes feitos na aula por Renan Vedovato Traba, a partir da ideia do Felipe.

```
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; (* gera_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:

$$\begin{cases} P^1(n) = P(n); \\ P^k(n) = P(P^{k-1}(n)) \quad k \geq 2. \end{cases}$$

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 da *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[11]. 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 9.32 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, $P^{cont}(i) = x, P^{2\times cont}(i) = x, \ldots$. 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:

$$MMC(x_1, x_2, \ldots, x_n) = MMC(x_1, MMC(x_2, \ldots, x_n)).$$

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  mmc, cont: int64;
     p, i: integer;
begin
     mmc := 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;
          mmc := mmc * cont div mdc(mmc, cont);
     end;
     ordem_permutacao:= mmc;
end;
```

Figura 9.32: Calcula a ordem de uma permutação.

Euclides (figura 7.7, seção 7.3) 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).

9.7.2 Polinômios

Nesta seção vamos mostrar como representar e fazer cálculos com polinômios representados como vetores.

Para uma sucessão de termos $a_0, ..., a_n \in \mathbb{R}$, 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;
```

- $P(x) = 5 - 2x + x^2$:

0	1	2
5	-2	1

- $P(x) = 7 - 2x^2 + 8x^3 - 2x^7$

0	1	2	3	4	5	6	7
7	0	-2	8	0	0	0	-2

No restante desta seção iremos mostrar algoritmos que realizam operações costumeiras 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 $P\prime(x)$ a derivada de $P(x)$ assim definido:

$$P\prime(x) = na_n x^{n-1} + (n-1)a_{n-1}x^{n-2} + \ldots + 2a_2 x + a_1$$

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 $n >= m$:

$$P(x) = a_n x^n + \ldots + a_m x^m + \ldots + a_1 x + a_0$$

```
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 $P(x)$.

```
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 $P\prime(x)$ para um dado $x \in \mathbb{R}$.

$$Q(x) = b_n x^m + b_{n-1}x^{m-1} + \ldots + b_1 x + b_0$$

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 monômios de mesmo grau, somando-se os coeficientes, para cada monômio.

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 monômio à medida em que os índices dos dois comandos *for* variam, o que é um uso especial da técnica dos acumuladores, embora os acúmulos não sejam simultâneos para cada monômio 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 $Q(x)$.

9.8 Exercícios

9.8.1 Exercícios de aquecimento

1. Qual dos seguintes problemas requer o uso de vetores para uma solução elegante?
   - Ler cerca de duzentos números e imprimir os que estão em uma certa faixa;
   - Computar a soma de uma sequência de números;
   - Ler exatamente duzentos números e ordená-los em ordem crescente;
   - Encontrar o segundo menor elemento de uma sequência de entrada;
   - Encontrar o menor inteiro de uma sequência de inteiros.
2. Faça duas procedures em *Pascal* e teste-as no programa abaixo:
   - uma que leia vários números inteiros do teclado até que seja lido um zero. O zero não deve ser processado pois serve para marcar a entrada de dados. Cada número lido deve ser armazenado em um vetor de inteiros iniciando na posição 1 do vetor;
   - outra que imprima um vetor de inteiros que tem n elementos.

```
program ler_imprimir_vetores_v1;
const MAX = 200;
type vetor = array[1..MAX] of longint;
var v: vetor; tam: longint;

(* coloque aqui sua procedure para ler um vetor *)
(* use dois parametros, um para o vetor outro para o tamanho dele *)

(* coloque aqui sua procedure para imprimir um vetor *)
(* use dois parametros, um para o vetor outro para o tamanho dele *)

begin
    ler_vetor (v, tam);
    imprimir_vetor (v,tam);
end.
```

Exemplos de entradas	Saídas esperadas
4 7 3 1 9 12 5 3 0	4 7 3 1 9 12 5 3
7 8 6 2 9 9 3 2 1 2 0	7 8 6 2 9 9 3 2 1 2

3. Faça uma procedure que leia um número positivo n do teclado e em seguida leia exatamente outros n números inteiros quaisquer. Teste sua procedure no mesmo programa do exercício anterior.

Exemplos de entradas	Saídas esperadas
8 4 7 3 1 9 12 5 3	4 7 3 1 9 12 5 3
11 7 -8 -6 0 9 -9 3 3 -1 2 0	7 -8 -6 0 9 -9 3 3 -1 2 0

4. Faça uma procedure em *Pascal* que leia um número inteiro positivo n e inicialize um vetor de n inteiros de maneira que os conteúdos dos índices sejam respectivamente o dobro do índice para os índices pares e o triplo do índice para os índices ímpares. Teste sua procedure no programa abaixo:

```
program ler_imprimir_vetores_v1;
const MAX = 200;
type vetor = array[1..MAX] of longint;
var v: vetor; tam: longint;

(* coloque aqui sua procedure para gerar o vetor *)
(* use dois parametros, um para o vetor outro para o tamanho dele *)

(* coloque aqui sua procedure para imprimir um vetor *)
(* use dois parametros, um para o vetor outro para o tamanho dele *)

begin
    read (tam);
    gerar_vetor (v, tam);
    imprimir_vetor (v,tam);
end.
```

Exemplos de entradas	Saídas esperadas
5	3 4 9 8 15
8	3 4 9 8 15 12 21 16

9.8.2 Exercícios básicos

1. Faça uma função em *Pascal* que facilite a inicialização de um vetor de inteiros de modo que os elementos de índices ímpares recebam o valor inicial -2 e os elementos de índices pares recebam o valor inicial 7. Sua função deve fazer uso de um único comando de repetição, que incrementa de um em um, e de nenhum comando de desvio condicional.

Exemplos de entradas	Saídas esperadas
5	-2 7 -2 7 -2
8	-2 7 -2 7 -2 7 -2 7

2. Faça um programa em *Pascal* que leia um número inteiro n ($0 \leq n \leq 200$) e em seguida leia uma sequência de n valores reais e os insira em um vetor de reais. O programa deve imprimir na saída o valor absoluto da divisão da soma dos valores positivos pela soma dos valores negativos que estão armazenados no vetor. Cuidado com divisões por zero.

Exemplos de entradas	Saídas esperadas
4 -2.0 -7.0 7.0 2.0	 1.00
3 1 2 3	 divisao por zero
0	vetor vazio

3. Faça um programa em *Pascal* que leia uma sequência de inteiros terminada em zero e armazene esta sequência em um vetor. O zero não deve ser processado pois serve para marcar o final da entrada de dados. Em seguida o programa deve ler uma outra sequência de inteiros também terminada em zero e, para cada valor lido, o programa deve dizer se ele pertence ou não ao vetor armazenado previamente. Esta segunda sequência não precisa ser armazenada em vetores. Use ao máximo funções e procedimentos apropriados.

Exemplo de entrada	Saída esperada
-2 -6 7 2 0	
7	pertence
3	nao pertence
0	nao pertence
2	pertence

4. Faça um programa em *Pascal* que leia um inteiro positivo n e em seguida leia n valores inteiros quaisquer e imprima *sim* se o vetor estiver ordenado e *nao* em caso contrário. Use ao máximo funções e procedimentos apropriados.

Exemplos de entradas	Saídas esperadas
5 -2 -7 7 2 1	nao
7 1 3 4 8 8 10 15	sim
0	vetor vazio

5. Faça um programa em *Pascal* que leia um inteiro n e em seguida leia n números inteiros quaisquer. Seu programa deve então ler um número inteiro positivo p que esteja na faixa de índices válidos do vetor e remover o conteúdo deste índice do vetor, evidentemente reduzindo em uma unidade o tamanho do vetor. Ao final, deve imprimir o vetor resultante. Sabendo que o vetor não está ordenado, implemente uma procedure eficiente para fazer esta exclusão. Use a seguinte assinatura para o procedimento:

```
procedure remove(var v: vetor; var n: integer; p: integer);
```

Exemplos de entradas	Saídas esperadas
4 -2 -7 7 2 -7	-2 2 7
7 1 3 4 8 8 10 15 4	1 3 15 8 8 10

6. Este problema é muito parecido com o anterior, mas desta o vetor lido do teclado está garantidamente em ordem crescente de valores e você deve modificar o procedimento feito no exercício anterior para garantir que a saída continue ordenada.

Exemplos de entradas	Saídas esperadas
4 -2 -7 7 2 -7	 -2 2 7
7 1 3 4 8 8 10 15 4	 1 3 8 8 10 15

7. Faça um programa em *Pascal* que leia uma sequência de valores inteiros quaisquer terminada em zero. O zero não deve ser processado pois serva para marcar o final da entrada de dados. Em seguida, leia um número qualquer do teclado e imprima a posição em que ele se encontra no vetor, ou então a mensagem *nao esta presente* se ele não estiver presente no vetor. Você deve implementar uma variante da busca binária na qual, ao invés de achar a primeira ocorrência do valor na lista, imprima o menor índice do vetor no qual o valor ocorra. Use ao máximo funções e procedimentos apropriados. Lembre-se que a busca binária só pode ocorrer se o vetor estiver ordenado. Então, se a entrada de dados não estiver ordenada, ordene-a. Você pode usar funções e procedimentos de exercícios anteriores aqui.

Exemplos de entradas	Saídas esperadas
-2 -7 2 2 7 0 2	 3
1 8 8 8 8 10 15 0 8	 2

8. Faça um programa em *Pascal* que leia uma sequência de 10 letras (caracteres de 'A' a 'Z'), as armazene em um vetor de 10 posições e imprima a lista de letras repetidas no vetor. Use ao máximo funções e procedimentos adequados.

Exemplo de entrada	Saída esperada
A J G A D F G A	A G

9. Faça um programa em *Pascal* que leia um número inteiro positivo n e em sequida leia uma sequência de N números quaisquer e imprima quantos números distintos compõe a sequência e o número de vezes que cada um deles ocorre na mesma. Use ao máximo funções e procedimentos adequados.

Exemplo de entrada	Saída esperada
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

10. 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". Use ao máximo funções e procedimentos adequados.

Exemplos de entradas	Saídas esperadas
3 4 2101	65
4 2 35	-1

11. 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*. Use ao máximo funções e procedimentos adequados. Este exercício não tem caso de teste pois só tem uma saída possível.

12. 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 imprima 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. Use ao máximo funções e procedimentos apropriados.

Exemplos de entradas	Saídas esperadas	Comentário
719	53	pois 53 = 7 x 3! + 1 x 2! + 9 x 1!
1	1	pois 1 = 1 x 1!
15	7	pois 7 = 1 x 2! + 5 x 1!
110	8	pois 8 = 1 x 3! + 1 x 2! + 0 x 1!
102	8	pois 8 = 1 x 3! + 0 x 2! + 2 x 1!
0		

13. 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 número 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. O programa deve imprimir todos os vetores resultantes de cada operação e ao final deve imprimir o vetor resultante.

    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 (imprima a string `erro`);
- se for requisitada a remoção de um número não existente o programa deve dar uma mensagem de erro (imprima `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).
- use ao máximo funções e procedimentos apropriados.

Exemplos de entradas	Saídas esperadas
1 45.3	 45.3
1 34.3	 34.3 45.3
4 9 1 40.8	 34.3 40.8 45.3
2 11.9	 erro
2 34.3	 40.8 45.3
0	40.8 45.3

9.8.3 Exercícios de dificuldade média

1. Faça um programa em *Pascal* que leia um número inteiro n e em seguida leia uma sequência $x_1, x_2, \ldots, x_n$ de números reais e 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. Use ao máximo funções e procedimentos adequados.

Exemplos de entradas	Saídas esperadas
8 7 9 5 4 5 4 8 6	 3 2
4 1 2 3 4	
10 6 1 2 3 4 1 2 3 4 5	 2 4

2. Faça um programa em *Pascal* que leia um número inteiro n e em seguida leia uma sequência $x_1, x_2, \ldots, x_n$ de números reais e imprima o valor do segmento de soma máxima. Use ao máximo funções e procedimentos adequados.

Exemplo de entrada	Saída esperada	Comentário
12 5 2 -2 -7 3 14 10 -3 9 -6 4 1	33.00	soma de 3 até 9.

3. Suponha que você esteja usando o método da ordenação por seleção. Qual das sequências abaixo requerirá o menor número de trocas? Quantas? Qual requerirá o maior número de trocas? Quantas? Explique.

   (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.

4. Faça um programa em *Pascal* que leia um número inteiro n e em seguida leia uma sequência de n números inteiros quaisquer. Seu programa deve imprimir esta sequência de forma que todos os elementos repetidos da sequência devem ir para o final, mas de maneira que estes últimos fiquem em ordem crescente. Os outros elementos devem ficar na mesma ordem. Use ao máximo funções e procedimentos adequados.

Exemplos de entradas	Saídas esperadas
11 5 3 8 2 3 9 8 9 7 5 3	5 3 8 2 9 7 3 3 5 8 9
6 4 4 3 3 2 2	4 3 2 2 3 4
3 1 1 1	1 1 1
0	

5. Faça um programa em *Pascal* que leia um certo número indefinido de sequências e as armazene em um vetor. Seu programa deve imprimir na saída a sequência original e a sequência gerada após um processo de compactação que consiste na eliminação de todos os elementos repetidos de cada sequência. Considere que a entrada de dados é feita em uma sequência por linha, sendo que o primeiro elemento da linha é o tamanho da sequência e os elementos restantes da linha são os elementos da sequência propriamente ditos. Quando o tamanho for zero significa que terminou a entrada de dados. Use ao máximo funções e procedimentos adequados.

Exemplo de entrada	Saída esperada
5 2 4 7 -1 2	2 4 7 -1 2
	2 4 7 -1
3 1 1 1	1 1 1
	1
7 3 4 5 3 4 5 1	3 4 5 3 4 5 1
	3 4 5 1
0	

6. Considere um vetor declarado como:
`vetpr = array [1..50] of integer;` e considere também que seus elementos 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. Use ao máximo funções e procedimentos adequados. Este programa não tem caso de teste pois a saída é única, que é o vetor ordenado.

9.8.4 Aplicações de vetores

1. Faça um programa em *Pascal* que receba como entrada dois números inteiros n e m e que em seguida leia duas sequências de números inteiros contendo respectivamente n e m elementos inteiros quaisquer. Seu programa deve verificar se a segunda sequência ocorre na primeira. Em caso afirmativo, a saída do programa deve ser a posição na qual ela começa na primeira. Caso contrário, a saída deve ser o valor zero. Pense que este problema é parecido com o de encontrar uma palavra em um texto. Use ao máximo funções e procedimentos adequados.

Exemplos de entradas	Saídas esperadas
6 4 1 2 3 4 5 6 3 4 5 6	 3
8 2 3 7 3 2 1 5 7 7 3 2	 2
3 1 4 7 2 6	 0

2. 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 diferentes 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": $\binom{r}{k} = \binom{r-1}{k} + \binom{r-1}{k-1}$. 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:

```
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
```

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). 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. Use ao máximo funções e procedimentos adequados. Este programa não tem caso de teste pois a saída e única.

3. Seja um polinômio $p(x) = a_0 + a_1x + a_2x^2 + \ldots + a_nx^n$ 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:

$$x_{n+1} = x_n - \frac{p(x_n)}{p\prime(x_n)},$$

$n = 0, 1, 2, \ldots$, e onde $p\prime(x)$ 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 número inteiro n positivo representando o grau de um polinômio e em seguida leia n números reais $a_0, a_1, a_2, \ldots, a_n$ representando os coeficientes do polinômio $p(x) = a_0 + a_1x + a_2x^2 + \ldots + a_nx^n$. Leia também um valor real x_0 como sendo uma aproximação inicial da raiz de $p(x)$, leia um real $\epsilon > 0$ e finalmente leia o número máximo de iterações, que é um valor inteiro. Seu programa deve calcular 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

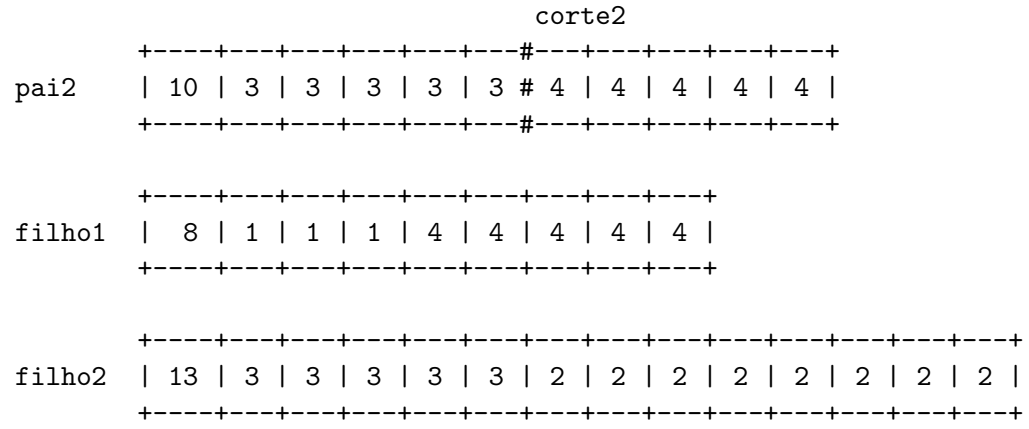
$p\prime(x)$. Utilize também uma função que receba como parâmetros um polinômio $p(x)$ e um valor real $\overline{x}$ e retorne o valor do polinômio no ponto $\overline{x}$, isto é $p(\overline{x})$. Use esta função para calcular, a cada iteração do método de Newton, os valores de $p(x_n)$ e de $p\prime(x_n)$.

4. Faça um programa em *Pascal* que simule a operação de localizar e substituir comumente usada em editores de texto. Leia três números inteiros positivos n_1, n_2 e n_3 e em seguida leia três sequências de números inteiros quaisquer contendo respectivamente n_1, n_2 e n_3 elementos. Seu programa deve localizar a segunda sequência na primeira e, caso encontre, deve fazer a substituição dos elementos da primeira pelos da terceira. Caso não encontre não deve fazer nada. Ao final seu programa deve imprimir a primeira sequência modificada pela operação. Use ao máximo funções e procedimentos adequados.

Exemplos de entradas	Saídas esperadas
10 5 7 5 9 5 6 7 8 4 0 1 3 7 8 4 0 1 3 1 2 3 4 5 6 7 8	5 9 5 6 1 2 3 4 5 6 7 8
5 5 0 1 2 3 4 5 1 2 3 4 5	

5. Um *algoritmo genético* é um procedimento computacional de busca inspirado no processo biológico de evolução, que otimiza a solução de um problema, que 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:

```
                          corte1
        +---+---+---+---#---+---+---+---+---+---+---+---+
pai1    | 11 | 1 | 1 | 1 # 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 |
        +---+---+---+---#---+---+---+---+---+---+---+---+
```



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:

```
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]`.

Faça também um programa completo em *Pascal* para testar seu procedimento, incluindo a leitura e a impressão dos dados. Primeiro leia os dois vetores de entrada e em seguida os dois inteiros que são os pontos de corte. Imprima na saída os dois filhos obtidos pelas operações, lembrando que o tamanho de cada vetor está na primeira posição dele.

Exemplo de entrada	Saída esperada
11 1 1 1 2 2 2 2 2 2 2 10 3 3 3 3 3 4 4 4 4 4 4 6	8 1 1 1 4 4 4 4 4 13 3 3 3 3 3 2 2 2 2 2 2 2

6. 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.

   - 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

  - **t=10:**

- Saída:
  - **t=8:** 35
  - **t=9:** 27
  - **t=10:** 16

Exemplo de entrada	Saída esperada
2 35 1 0 1 27 4 0 0 1 16 2 0 0 0	 8 35 9 27 10 16

7. 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. os procedimentos modificados ou novos que levam em conta este novo fato.

   Exemplo da nova saída para a entrada original:

   - 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:**

				35_1	35_1	27_4

  - **t=6:**

16_2					35_1	35_1

  - **t=7:**

		16_2				35_1

  - **t=8:**

				16_2		

  - **t=9:**

						16_2

  - **t=10:**

- Saída:
  - **t=6:** 27
  - **t=8:** 35
  - **t=10:** 16

Exemplo de entrada	Saída esperada
2 35 1 0 1 27 4 0 0 1 16 2 0 0 0	 6 27 8 35 10 16

9.8.5 Exercícios difíceis

1. Faça um programa em *Pascal* que leia um número inteiro positivo n e em seguida leia duas sequências de n inteiros quaisquer. Seu programa deve imprimir *sim* caso as duas sequências sejam iguais e *nao* em caso contrário. Use ao máximo funções e procedimentos adequados.

Exemplos de entradas	Saídas esperadas
6 2 4 3 1 3 5 2 4 3 1 3 5	 sim
5 4 3 1 3 1 4 3 1 3 5	 nao

2. Faça um programa em *Pascal* que leia dois números inteiros positivos n e m e em seguida leia duas sequências de inteiros quaisquer contendo respectivamente n e m números. 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.

Exemplos de entradas	Saídas esperadas
5 5 2 2 0 3 4 2 2 0 3 4	sim
5 5 2 2 0 3 4 4 3 2 0 2	sim
5 5 2 2 0 3 4 1 3 2 0 2	nao
3 4 3 0 5 3 0 5 3	nao

   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.

3. Faça um programa em *Pascal* que leia duas sequências de inteiros terminadas em zero, não necessariamente contendo a mesma quantidade de números. Os zeros não deverão ser processados e servem para marcar o final da entrada de dados. Seu programa deverá construir um terceiro vetor, sem destruir os originais, que é a concatenação do primeiro com o segundo e imprimir o vetor resultante. Use ao máximo funções e procedimentos adequados.

Exemplo de entrada	Saída esperada
7 3 2 6 0 5 1 8 4 9 0	7 3 2 6 5 1 8 4 9

4. Faça um programa em *Pascal* que leia duas sequências de inteiros terminadas em zero, não necessariamente contendo a mesma quantidade de números. Os zeros não deverão ser processados e servem para marcar o final da entrada de dados. Seu programa deverá ordená-los, e em seguida imprimir todos os números ordenados em ordem crescente. Use ao máximo funções e procedimentos adequados.

Exemplo de entrada	Saída esperada
7 3 2 6 0 5 1 8 4 9 0	 1 2 3 4 5 6 7 8 9

9.8.6 Desafios

1. Faça um programa em *Pascal* que leia dois números inteiros positivos m e n, leia também uma frase com m letras e uma palavra com n letras e imprima o número de vezes que a palavra ocorre na frase e a posição em que cada ocorrência inicia. Use ao máximo funções e procedimentos adequados.

Exemplo de entrada	Saída esperada
30 3 ANA E MARIANA GOSTAM DE BANANA	 4 1 11 26 28

   A saída indica que a palavra *ANA* ocorre 4 vezes na frase, nas posições 1, 11, 26, 28.

2. Faça um programa em *Pascal* que leia três números inteiros:

   - O tamanho n do vetor no intervalo $[1..MAX]$, onde MAX é o tamanho máximo do vetor definido no seu tipo `vetor`;
   - Os números inteiros min, max, $0 < min < max$, que definirão um intervalo, explicado abaixo.

   Seu programa deverá gerar um vetor de n elementos que serão gerados aleatoriamente a cada passo no intervalo $[min, max]$ da seguinte forma:

   - A posição na qual cada elemento é inserido no vetor também é gerada aleatoriamente, no intervalo $[1..n]$, pois deve ser uma posição válida no vetor;
   - 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 posição entre $i+1$ e o final do vetor estiverem ocupadas, seu algoritmo deve pegar a primeira posição livre a partir do início do vetor.

   O seu programa deve imprimir a cada passo os números i e *valor* gerados aleatoriamente e a impressão do vetor. Use espaços em branco para posições que não têm elementos válidos, isto é, aqueles inseridos pelo programa. Note que os números válidos são todos maiores do que 1.

   Dica: A função `random(n)` retorna um número inteiro no intervalo $[0, n[$.

Exemplo de entrada	Saída esperada
10 50 99	73 4
	73
	82 8
	73 82
	54 1
	54 73 82
	65 8
	54 73 82 65
	97 8
	54 73 82 65 97
	81 8
	54 81 73 82 65 97
	59 6
	54 81 73 59 82 65 97
	62 1
	54 81 62 73 59 82 65 97
	70 7
	54 81 62 73 59 70 82 65 97
	96 7
	54 81 62 73 96 59 70 82 65 97

3. Faça um programa em *Pascal* que leia um número inteiro positivo e depois leia uma sequência de n valores reais não nulos ($n \leq 100$) e os insira em um vetor. Considere o elemento $p = V[1]$ como o *pivô* na operação de rearranjar o vetor de tal maneira que, ao final do processo, todos os elementos à esquerda de p sejam menores que ele e todos os da direta sejam maiores ou iguais a ele. No final, o elemento p pode não estar mais na posição 1.

Exemplo de entrada	Saída esperada
6 13 32 99 2 26 12	 2 12 13 32 99 26

O programa gerou como resultado um vetor onde todos os elementos que estão à esquerda do valor 13 no vetor são menores que ele, enquanto que os da direita são maiores do que ele.

9.8.7 Exercícios de maratona de programação

Os exercícios de maratona são feitos de maneira que, em geral, um algoritmo qualquer que resolve o problema não serve, mas tem que ser um algoritmo eficiente. Por exemplo, se existe a chance de você usar um algoritmo de complexidade proporcial $log(n)$ e você usou um de complexidade proporcional a n, então você ganha do sistema de competição a resposta `time limit exceeded`.

Logo, nos próximos exercícios, cuide para que seu algoritmo seja eficiente!

Nesta seção os enunciados serão os mais fiéis possíveis aos que foram colocados na maratona.

1. 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.

9.9 Exercícios de prova

No link abaixo você encontra diversas provas aplicadas que cobrem os conteúdos vistos até aqui:

- `http://www.inf.ufpr.br/cursos/ci055/prova2.html`

Capítulo 10

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).

10.1 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):

```
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:

```
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:

```
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:

```
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:

```
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.

10.2 Exemplos elementares

Do mesmo modo como fizemos para os vetores, vamos iniciar o estudo das matrizes apresentando problemas simples.

10.2.1 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 10.1 permite visualizar o código para a leitura de uma matriz.

```
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 10.1: 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:

```
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 10.2 apresenta um procedimento que lê uma matriz de dimensões $n \times m$.

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 10.3 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.

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:

```
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 10.2: Procedimento para ler uma matriz.

```
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]); (* muda de linha a cada fim de coluna, sem o branco no final
            *)
    end;
end;
```

Figura 10.3: Procedimento para imprimir uma matriz.

	1	2	3	4	5	...	40
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	?		?
⋮							
50	?	?	?	?	?		?

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 10.4. Basta inverter i e j no comando de impressão! Observem que a matriz não mudou na memória, somente a impressão é diferente.

Outros procedimentos interessantes são os de impressão de uma certa linha (figura 10.5) ou de uma certa coluna (figura 10.6).

Considerando o exemplo acima e fazendo $K = 2$ teríamos a seguinte saída:

```
9 0 0 2
```

```
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 10.4: Procedimento para imprimir a transposta de uma matriz.

```
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
            write (w[K,j],' '); (* K fixo na primeira posicao *)
      writeln;
end;
```

Figura 10.5: Procedimento para imprimir uma única linha da matriz.

```
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 posicao *)
end;
```

Figura 10.6: 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 10.7.

Considerando novamente a nossa matriz exemplo, teríamos a seguinte saída:

```
4 6 2 0 0 2 8 2 4 0 0
```

```
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;
```

Figura 10.7: Procedimento para imprimir os elementos pares matriz.

Para finalizarmos esta seção inicial, apresentamos na figura 10.8 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
```

```
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],' ');
          writeln;
     end;
end;
```

Figura 10.8: Procedimento para imprimir os elementos cujos índices são pares.

10.2.2 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 10.9 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.

```
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 10.9: Encontrando o menor elemento de uma matriz.

10.2.3 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.

Desta 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 10.10.

```
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 10.10: Somando duas matrizes.

10.2.4 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 10.11.

```
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 10.11: Produto escalar de uma linha da matriz por uma coluna da outra.

Vejamos isto de forma ilustrada considerando as duas matrizes seguintes:

$$A: \begin{array}{cccc} 4 & 6 & 2 & 1 \\ 9 & 0 & 0 & 2 \\ \boxed{8} & \boxed{7} & \boxed{3} & \boxed{9} \\ 1 & 2 & 3 & 4 \\ 0 & 1 & 0 & 1 \end{array} \qquad B: \begin{array}{ccc} 2 & \boxed{3} & 4 \\ 5 & \boxed{0} & 0 \\ 0 & \boxed{7} & 7 \\ 1 & \boxed{0} & 9 \end{array}$$

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 a 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 10.12.

```
procedure multiplicar_matrizes (var A: matriz;      lin_A, col_A: integer;
                                var B: matriz;      lin_B, col_B: integer;
                                var AB: matriz; var lin_AB, col_AB: integer);
var i, j, k: integer;
begin
     lin_AB:= lin_A; col_AB:= col_B;
     for i:= 1 to lin_A do
          for j:= 1 to col_B do
          begin
               AB[i,j]:= 0;
               for k:= 1 to lin_B do
                    AB[i,j]:= AB[i,j] + A[i,k] * B[k,j];
          end;
end;
```

Figura 10.12: Multiplicação de duas matrizes.

10.3 Procurando elementos em matrizes

Nesta seção vamos apresentar alguns problemas de busca de elementos em matrizes.

10.3.1 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 10.13 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 se 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.

```
function busca (var w: matriz; n,m: integer; x: integer): boolean;
var i,j: integer; achou: boolean;
begin
      i:= 1; achou:= false;
      while (i <= n) and not achou do
      begin
            j:= 1;
            while (j <= m) and not achou do
            begin
                  if w[i,j] = x then achou:= true;
                  j:= j + 1;
            end;
            i:= i + 1;
      end;
      busca:= achou;
end;
```

Figura 10.13: Busca em uma matriz.

Para completar o grau de dificuldade, queremos parar o processamento tão logo um elemento repetido seja encontrado. O código final está ilustrado na figura 10.14.

10.4 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
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

1	2	3	4	5
7	6	5	4	3

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 5×5 (vetor inserido está em negrito na figura):

	1	2	3	4	5
1	4	6	2	1	**7**
2	9	0	0	2	**6**
3	8	7	3	9	**5**
4	1	2	3	4	**4**
5	0	1	0	1	**3**

```
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 10.14: Verifica se uma matriz tem elementos repetidos.

O procedimento mostrado na figura 10.15 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.

	1	2	3	4	5
1	4	**7**	6	2	1
2	9	**6**	0	0	2
3	8	**5**	7	3	9
4	1	**4**	2	3	4
5	0	**3**	1	0	1

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

```
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 10.15: Insere um vetor como última coluna de uma matriz.

várias colunas para frente, cada uma delas move n elementos para frente. O algoritmo apresentado na figura 10.16 mostra estes dois passos, um que abre espaço o outro que insere o vetor no espaço aberto.

```
procedure insere_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 espaco *)
     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 10.16: Insere um vetor como K-ésima coluna de uma matriz.

Para inserir linhas em uma matriz o procedimento é análogo.

10.5 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.

10.5.1 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*[1]. As imagens estão ar-

[1] `https://en.wikipedia.org/wiki/Netpbm_format`

mazenadas 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* tem 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 10.17 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.

```
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
```

Figura 10.17: Exemplo de imagem no formato PGM.

Nesta seção vamos considerar as seguintes declarações:

```
const MAXTAM = 1000;

type imagem = array [1..MAXTAM,1..MAXTAM] of integer;

var
    l,              (* numero de linhas da imagem  *)
    c,              (* numero de colunas da imagem *)
    max: integer;   (* valor do pixel maximo       *)
    O: imagem       (* matriz que contera 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 10.18 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 10.19 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.

```
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,l);
                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 10.18: Leitura de uma imagem PGM.

```
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 10.19: 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 *smartfones*. A figura 10.20 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 10.21 considera um valor constante *cte*, que é o quanto queremos clarear a imagem. O procedimento é muito simples, ele simplesmente soma este valor *cte* em todos os pixels, mas ele leva em conta que

Figura 10.20: Clareando uma imagem.

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 10.21: 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 10.22 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 10.23 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

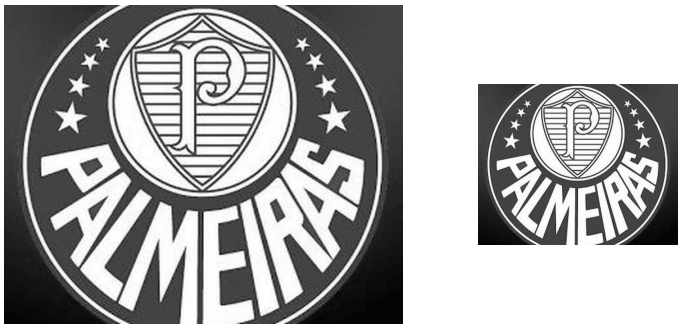


Figura 10.22: Reduzindo uma imagem.

$$\begin{matrix} 4 & 6 & 2 & 1 \\ 9 & 0 & 0 & 2 \\ 8 & 7 & 3 & 9 \\ 1 & 2 & 3 & 4 \end{matrix} \qquad \left\|\begin{matrix} 4 & 6 \\ 9 & 0 \end{matrix}\right\| \left\|\begin{matrix} 2 & 1 \\ 0 & 2 \end{matrix}\right\| \left\|\begin{matrix} 8 & 7 \\ 1 & 2 \end{matrix}\right\| \left\|\begin{matrix} 3 & 9 \\ 3 & 4 \end{matrix}\right\|$$

Figura 10.23: Visualizando uma matriz 4×4 em quatro blocos 2×2.

a seguinte ilustração da submatriz 2×2 do canto superior esquerdo da submatriz destacada no lado direito da figura:

$$\begin{matrix} 4 & 6 \\ 9 & 0 \end{matrix}$$

A média destes elementos é $\frac{(4+6+9+0)}{4} = 4.75$.

Já para a submatriz do canto superior direito:

$$\begin{matrix} 2 & 1 \\ 0 & 2 \end{matrix}$$

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 2×2 resultante é esta aqui:

$$\begin{matrix} 4.75 & 1.25 \\ 4.50 & 4.75 \end{matrix}$$

Evidentemente queremos o valor inteiro destes números, o que daria a seguinte matriz:

$$\begin{matrix} 4 & 1 \\ 4 & 4 \end{matrix}$$

O procedimento que realiza este cálculo é ilustrado na figura 10.24, 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 2×2 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 10.25.

```
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 10.24: Procedure para fazer zoom em uma imagem PGM.

A figura 10.25 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 $(2 * i - 1, 2 * j - 1)$.

```
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 10.25: 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 10.9, modificado para se retornar um valor do tipo *integer* e não do tipo *real*, o que resulta na função da figura 10.26.

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?

```
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 10.26: Cálculo do valor do maior pixel.

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 10.27 ilustram esta problemática.

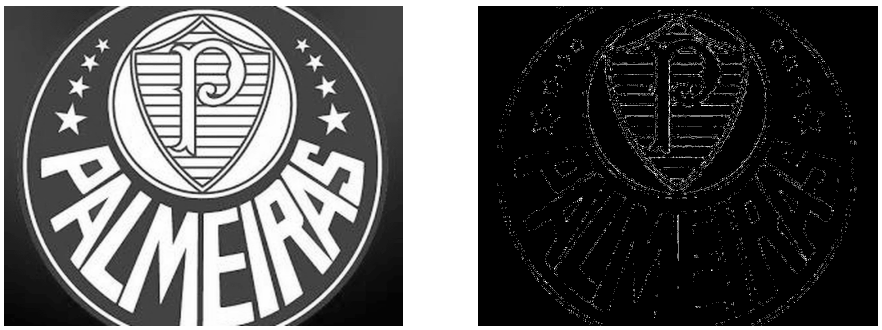


Figura 10.27: Detecção de bordas em uma imagem.

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:

0	2	1	3	1	2	3	2
1	1	1	50	50	50	5	4
1	2	2	50	0	50	4	2
1	1	3	50	6	50	3	1
1	1	2	50	50	50	2	4
3	4	5	3	2	1	0	2
1	4	5	2	3	5	0	1

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. Se somarmos 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.

Agora consideremos o pixel da posição $(4, 2)$: seus vizinhos, sempre no sentido horário, são 2, 3, 1 e 1. Somando estes valores resulta em 7. O valor do próprio pixel, vezes 4, é 4. Isto é, a diferença entre o pixel em questão e seus vizinhos é 1-7=6.

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 10.28 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.

```
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 10.28: Procedure para detectar bordas de uma imagem PGM.

10.6 Exercícios

Nesta seção a formatação dos casos de testes sai um pouco do padrão que adotamos neste livro para alguns problemas, pois entendemos que como as entradas e saídas são muito grandes o efito visual é melhor.

10.6.1 Exercícios de aquecimento

1. Dada uma matriz A, define-se como a *matriz transposta* de A, denotada A^T, como aquela que é o resultado da troca de linhas por colunas em A. Faça um programa em *Pascal* que leia dois números inteiros n e m representando as dimensões da matriz A, sendo n o número de linhas e m o número de colunas, e leia também os elementos inteiros de uma matriz A, linha por linha, da esquerda para a direita, e da mesma forma leia as dimensões e os elementos inteiros de uma matriz B. Seu programa deve imprimir *sim* se $B = A^T$ e *nao* caso contrário. Use um procedimento para a leitura dos dados e uma função para decidir sobre ser ou não transposta.

Exemplo de entrada	Saída esperada
2 3 1 2 3 4 5 6 3 2 1 3 5 2 4 6	 sim

2. Dizemos que uma matriz inteira A de dimensões $n \times n$ 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. Faça um programa em *Pascal* que leia a dimensão n de uma matriz A e em seguida leia seus elementos inteiros, linha por linha, coluna por coluna e em seguida imprima *sim* caso A seja uma matriz de permutação e *nao* caso contrário.

Exemplos de entradas	Saídas esperadas
4 0 1 0 0 0 0 1 0 1 0 0 0 0 0 0 1	 sim
0 1 0 0 0 0 1 0 1 0 0 0 0 0 0 2	 nao

3. Uma matriz B é dita ser a *matriz 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 na qual os elementos da diagonal principal são iguais a 1 e os demais são iguais a zero. Assim, $I[i, j] = 1$ se $i = j$ e $I[i, j] = 0$ se $i \neq j$. Faça um programa em *Pascal* que leia dois números inteiros positivos n, m e em seguida leia os elementos reais de uma matriz A. Faça o mesmo para ler uma matriz B de reais (use um procedimento para isso). Seu programa deve testar se B é a inversa de A.

Exemplos de entradas	Saídas esperadas
2 2 2.0 1.0 4.0 3.0 2 2 1.5 -0.5 -2 1	 sim

10.6.2 Exercícios básicos

1. Faça um programa em *Pascal* que leia dois números inteiros n e m representando respectivamente o número de linhas e o número de colunas de uma matriz $A_{n \times m}$ e em seguida leia os elementos inteiros da matriz. Seu programa deve imprimir o número de linhas e o número de colunas nulas da matriz. Exemplo: a matriz abaixo tem duas linhas e uma coluna nulas.

$$\begin{matrix} 0 & 0 & 0 & 0 \\ 1 & 0 & 2 & 2 \\ 4 & 0 & 5 & 6 \\ 0 & 0 & 0 & 0 \end{matrix}$$

Exemplo de entrada	Saída esperada
4 0 0 0 0 1 0 2 2 4 0 5 6 0 0 0 0	 2 1

2. Um vetor real x com n elementos é apresentado como resultado de um sistema de equações lineares $Ax = y$ cujos coeficientes são representados em uma matriz real $A_{n \times n}$ e os lados direitos das equações em um vetor real y de n elementos. Faça um programa em *Pascal* que leia a dimensão n da matriz quadrada, seus elementos reais, linha por linha, e depois leia o vetor y de n elementos e finalmente leia o vetor x de n elementos que seria a resposta do sistema de equações linerares. Seu programa deve verificar se o vetor x é realmente solução do sistema dado.

Por exemplo, para o sistema abaixo a solução é o vetor $[1, -2]$:

$$\begin{aligned} x + 2y &= -3 \\ 2x + y &= 0 \end{aligned}$$

Exemplo de entrada	Saída esperada
2 1 2 2 1 -3 0 1 -2	sim
2 1 2 2 1 -3 0 1 2	nao

3. Considere o seguinte programa em *Pascal* :

```
program maior_sequencia;

const MAX=100;
type matriz = array [1..MAX,1..MAX] of integer;
var n_lin, n_col: integer; (* dimensoes da matriz *)
    m: matriz;             (* matriz *)

(* espaco 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 (l_ini, c_ini);
   writeln (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.

Exemplos de entradas	Saídas esperadas
4 3 1 2 3 2 2 1 3 2 5	 1 2 3 2
4 5 1 2 3 1 2 1 2 2 2 3 2 3 4 5 6 8 7 6 4 2	 2 2 2 4

10.6.3 Exercícios de dificuldade média

1. Faça um programa em *Pascal* que leia dois números inteiros n e m representando as dimensões de uma matriz $A_{n\times m}$. Em seguida leia os elementos inteiros de A e imprima uma segunda matriz B de mesmas dimensões de A em que cada elemento $B[i, j]$ seja constituído pela soma de todos os 8 elementos vizinhos do elemento $A[i, j]$, excetuando-se o próprio $A[i, j]$. Observe que alguns elementos não tem os 8 vizinhos, os que ficam nas bordas e nos cantos têm menos.

Exemplos de entradas	Saídas esperadas
2 2 0 1 2 0	 3 2 1 3
3 3 1 2 0 0 1 2 2 0 1	3 4 5 6 8 4 1 6 3

2. Faça um programa em *Pascal* que leia dois números inteiros n e m representando as dimensões de uma matriz $A_{n\times m}$. Em seguida leia os elementos inteiros desta matriz. Leia em seguida um número inteiro p, $1 \leq p \leq n$ e $1 \leq p \leq m$, e um vetor de p inteiros. Finalmente leia outros dois inteiros lin, col que serão utizados como índices na matriz.

   Seu programa deve imprimir *sim* se o vetor está contido na matriz a partir das coordenadas lin, col e imprima *nao* em caso contrário. Imagine que você está simulando um jogo de caça-palavras usando números, e que a "palavra" representada pelo vetor de números pode estar na horizontal ou na vertical, tanto de frente para trás quando de trás para frente.

   Para facilitar, implemente quatro funções, cada uma delas recebe como parâmetros a matriz, as dimensões da matriz, o vetor de inteiros e as coordenadas de início da procura:

   - A primeira procura na horizontal, da esquerda para direita;
   - A segunda procura na horizontal, da direita para esquerda;

- A terceira procura na vertical, da cima para baixo;
- A quarta procura na vertical, da baixo para cima.

Por exemplo, considere a matriz 10×10 abaixo e suponha que você queira procurar se o vetor [50 56 41 37 58] está nela. A resposta do seu programa deve ser sim, pois este vetor é encontrado a partir das coordenadas $(5, 8)$ de trás para frente.

```
42 45 52 60 46 61 42 60 35 47
48 33 36 27 63 13 67 26 33 38
57 58 41 38 44 33 65 60 14 30
15 48 11 32 59 67 56 18 62 62
68 38 57 58 37 41 56 50 17 53
48 44 18 42 66 55 41 16 34 38
25 21 56 54 37 22 44 18 11 29
47 18 46 23 47 33 66 64 50 36
31 46 36 64 51 15 13 68 50 49
50 20 22 31 17 55 28 46 31 29
```

Exemplo de entrada	Saída esperada
10 10 42 45 52 60 46 61 42 60 35 47 48 33 36 27 63 13 67 26 33 38 57 58 41 38 44 33 65 60 14 30 15 48 11 32 59 67 56 18 62 62 68 38 57 58 37 41 56 50 17 53 48 44 18 42 66 55 41 16 34 38 25 21 56 54 37 22 44 18 11 29 47 18 46 23 47 33 66 64 50 36 31 46 36 64 51 15 13 68 50 49 50 20 22 31 17 55 28 46 31 29 5 50 56 41 37 58 5 8	sim

3. Dizemos que uma matriz quadrada 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. Por exemplo, a matriz abaixo 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.

$$\begin{matrix} 8 & 0 & 7 \\ 4 & 5 & 6 \\ 3 & 10 & 2 \end{matrix}$$

   Faça um programa em *Pascal* que leia um número inteiro n representando as dimensões de uma matriz $A_{n \times n}$ e em seguida leia os elementos inteiros desta matriz. Seu programa deve imprimir *sim* se esta matriz é um quadrado mágico e *nao* caso contrário.

Exemplos de entradas	Saídas esperadas
3 8 0 7 4 5 6 3 10 2	 sim
3 1 0 7 4 5 6 3 10 2	 nao

4. Considere que os elementos $A[i, j]$ de uma matriz $A_{n\times n}$ representam os custos de transporte da cidade i para a cidade j. Faça um programa em *Pascal* que leia um número inteiro n e em seguida leia os dados da matriz de inteiros $M_{n\times n}$. Leia também um número inteiro k que representa o tamanho de um itinerário e em seguida leia k inteiros que são as cidades deste itinerário. Seu programa deve calcular e imprimir o custo total deste itinerário.

   Por exemplo, considere a matriz abaixo e o itinerário: $[1, 4, 2, 4, 4, 3, 2, 1]$. O custo dele é 417, pois A[1,4] + A[4,2] + A[2,4] + A[4,4] + A[4,3] + A[3,2] + A[2,1] = 3 + 1 + 400 + 5 + 2 + 1 + 5 = 417.

```
4 1 2 3
5 2 1 400
2 1 3 8
7 1 2 5
```

Exemplo de entrada	Saída esperada
4 4 1 2 3 5 2 1 400 2 1 3 8 7 1 2 5 8 1 4 2 4 4 3 2 1	 417

10.6.4 Aplicações de matrizes

1. Uma matriz pode ser utilizada como uma estrutura de dados para suportar um jogo de damas. Por exemplo, considere uma matriz $X_{8\times 8}$ cujos elementos só podem ser 1, -1 e 0. O valor 1 pode ser usado para indicar uma casa ocupada por uma peça branca, o -1 para uma casa ocupada por peça preta e o 0 para indicar uma casa vazia. Faça um programa em *Pascal* que, supondo que as peças pretas estão se movendo no sentido crescente das linhas da matriz X, isto é, a partir da linha 1 em direção à linha 8, determine as posições das peças pretas que:

- podem tomar peças brancas;
- podem mover-se sem tomar peças brancas;
- não podem se mover.

Para isso seu programa deve ler os números inteiros que representam uma configuração do tabuleiro, supondo que é uma configuração válida, e deve imprimir a resposta no formato exemplificado nos casos de teste abaixo. Não precisa ler as dimensões do tabuleiro, já que ele será sempre 8×8.

Não se preocupe em promoção de peças que podem se tornar damas. Você está apenas calculando as possibilidades de uma única jogada das peças pretas. Use ao máximo funções e procedimentos para facilitar o código.

Por exemplo, a configuração inicial de um tabuleiro de damas é a seguinte:

	P		P		P		P
P		P		P		P	
	P		P		P		P
B		B		B		B	
	B		B		B		B
B		B		B		B	

- podem tomar peças brancas: nenhuma;
- podem mover-se sem tomar peças brancas: (3,2), (3,4), (3,6) e (3,8);
- não podem se mover: (1,2), (1,4), (1,6), (1,8), (2,1), (2,3), (2,5), (2,7), (3,2), (3,4), (3,6), e (3,8).

Exemplo de entrada	Saída esperada
0 -1 0 -1 0 -1 0 -1	
-1 0 -1 0 -1 0 -1 0	
0 -1 0 -1 0 -1 0 -1	
0 0 0 0 0 0 0 0	
0 0 0 0 0 0 0 0	
1 0 1 0 1 0 1 0	nenhuma
0 1 0 1 0 1 0 1	3 2 3 4 3 6 3 8
1 0 1 0 1 0 1 0	1 2 1 4 1 6 1 8 2 1 2 3 2 5 2 7 3 2 3 4 3 6 3 8

2. Considere o tipo PGM para imagens como definido na seção 10.5.1. Faça um programa em *Pascal* que leia duas imagens no formato PGM: imagem original ($imgO$) e a imagem do padrão ($imgP$). Conforme explicado, isto consiste na leitura de, para cada matriz:

   - uma *string* que deve ser igual à `P2`;

- um par de inteiros *col*, *lin* que são o número de colunas e o número de linhas da matriz;
- um inteiro que é o pixel de maior intensidade;
- os elementos inteiros da matriz.

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. Use ao máximo funções e procedimentos de forma adequada.

Exemplo de entrada	Saída esperada
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 P2 3 3 20 20 20 -1 -1 20 -1 -1 20 20	 2 2 7 4 5 8

3. 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 de entrada	Saída esperada
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 P2 3 3 20 20 20 -1 -1 20 -1 -1 20 20	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 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 20 20 11 40 5 5 5 5 5 5 40 11 20 11 40 5 5 0 40 5 5 40 11 20 20 40 5 40 5

4. 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:

```
4
3
2    +---+
1  +-x---x-------+
 1 2 3 4 5 6 7 8 9
```

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  +-------x---x-+
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. Este problema não tem casos de teste pois existem muitas saídas possíveis que estão corretas. Os exemplos acima devem ser suficientes.

5. 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
```

Este problema não tem casos de teste pois existem muitas saídas possíveis que estão corretas. Os exemplos acima devem ser suficientes.

10.6.5 Exercícios difíceis

1. Faça um programa em *Pascal* que leia um número inteiro positivo n e em seguida leia n datas e as coloque em uma matriz $n \times 3$. A primeira coluna desta matriz corresponde ao dia, a segunda ao mês e a terceira ao ano, coloque essas datas em ordem cronológica crescente.

Exemplo de entrada	Saída esperada
5	
5 1 1996	16 3 1951
25 6 1965	25 6 1965
16 3 1951	5 11 1965
15 1 1996	5 1 1996
5 11 1965	15 1 1996

2. 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 $n = 4$.

	1	2	3	4
1	1	1	1	0
2	0	1	1	0
3	1	0	1	1
4	0	0	1	1

Por exemplo, existe um caminho direto da cidade 1 para a cidade 2 mas não de 1 para 4. Faça um programa em *Pascal* que implemente funções e/ou procedimentos e que imprima respostas para as seguintes questões:

(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.

3. 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. Seu programa deve ler dois inteiros positivos n e m e em seguida ler os elementos da matriz que representam o jogo. O problema que deve ser resolvido é o de 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:

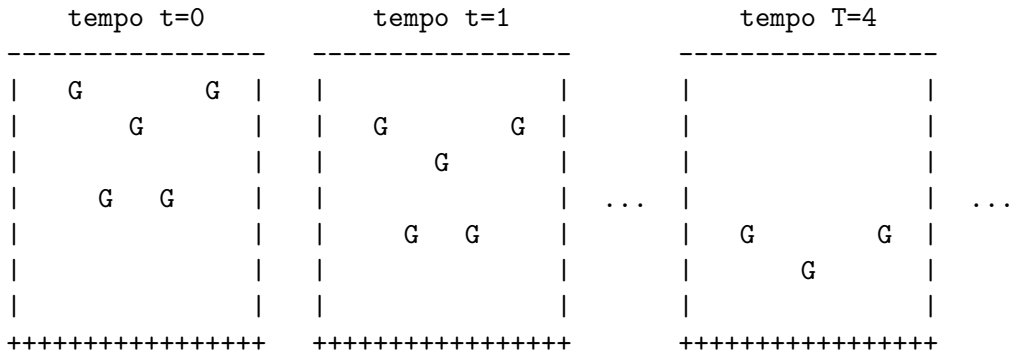
0	-1	0	-1	-1	0	-1	0
0	0	0	0	-1	0	0	0
0	0	-1	-1	0	0	-1	0
-1	0	0	0	0	-1	0	0
0	0	-1	0	0	0	-1	-1

   A saída deveria ser:

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

4. Considere uma matriz A de tamanho $n \times m$ utilizada para representar gotas de água (caractere 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:



Faça um programa em *Pascal* que:

(a) Leia as dimensões da matriz, dois números inteiros positivos n e m e em seguida 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 $t = 0$):

```
1 4
1 13
4 6
2 8
100 98
4 10
0 0
```

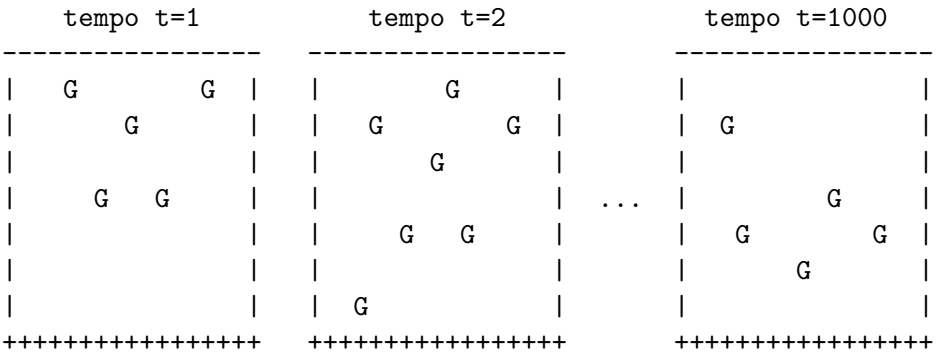
Note que a entrada $(100, 98)$ deve ser descartada pois é inválida para a matriz do exemplo.

(b) Imprima, a cada unidade de tempo t, o conteúdo da matriz A, atualizando a posição das gotas G até que não reste nenhuma gota na janela.

5. 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.

6. 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:



Faça um programa em *Pascal* que:

(a) Leia o número de linhas (L) e o número de colunas (C) da matriz A, 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 A (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 A, 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.

7. Faça um programa em *Pascal* que:

- leia um inteiro positivo 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:

	1	2	3	4	5	6
1	0	1	0	1	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

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).

10.6.6 Desafios

1. Uma matriz é chamada de *esparsa* quando possui uma grande quantidade de elementos que valem zero. Por exemplo, a matriz de ordem 5×4 seguinte é esparsa, pois contém somente 4 elementos não nulos.

	1	2	3	4
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

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:

```
const
    MAX = 6;        (* um valor bem menor que 5 x 4, dimensao da matriz *)
type
    vetor_coordenadas = array [1..MAX] of integer;   (* coordenadas  *)
    vetor_elementos   = array [1..MAX] of real;      (* dados        *)
var
    L, C: vetor_coordenadas;  (* L: linhas, C: colunas                *)
    D: vetor_elementos;       (* D: dados                             *)
    N_lin, N_col: integer;    (* para armazenar as dimensoes da matriz *)
    N_elementos: integer      (* numero de elementos nao nulos        *)
```

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
L	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.

10.6.7 Exercícios de maratona de programação

Os exercícios de maratona são feitos de maneira que, em geral, um algoritmo qualquer que resolve o problema não serve, mas tem que ser um algoritmo eficiente. Por exemplo, se existe a chance de você usar um algoritmo de complexidade proporcial $log(n)$ e você usou um de complexidade proporcional a n, então você ganha do sistema de competição a resposta `time limit exceeded`.

Logo, nos próximos exercícios, cuide para que seu algoritmo seja eficiente!

Nesta seção os enunciados serão os mais fiéis possíveis aos que foram colocados na maratona.

1. Faça um programa em *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 *perdida*.

Entrada

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:
1 1 1
1 1 1 1                {fim do primeiro conjunto de testes}
2 2 2
1 1 1 2
1 1 2 1                {fim do segundo conjunto de testes }
493 182 3
349 148 363 146
241 123 443 147
303 124 293 17         {fim do terceiro conjunto de testes}
0 0 0                  {fim do conjunto de testes}

Saída
Não há posições perdidas.
Existe uma posição perdida.
Existem 83470 posições perdidas.
```

2. 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 verifica 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:

```
 1   2   3   4        1   2   3   4   5
12  13  14   5       16  17  18  19   6
11  16  15   6       15  24  25  20   7
10   9   8   7       14  23  22  21   8
                     13  12  11  10   9
```

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".

Capítulo 11

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 estruturas de dados, 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.

11.1 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:

```
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:

```
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:

```
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 11.1 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 11.1: Imprimindo registros.

Normalmente é mais comum ver os registros integrados a outras estruturas, tais como vetores, matrizes ou arquivos em disco[1]. Nas próximas duas seções veremos como integrar registros com vetores.

[1]O tipo *file* está fora do escopo desta disciplina.

11.2 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 11.2.

```
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 11.2: 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 12.

11.3 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:

```
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 11.3.

```
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 11.3: 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 11.4.

```
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 11.4: 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 convencionar 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 11.5 ilustra tal situação.

```
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 11.5: Ordena pelo CPF.

A procedure da figura 11.6 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 11.6: Troca os elementos do banco de dados.

11.4 Exercícios

11.4.1 Exercícios conceituais

1. 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.

2. 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.

3. Esta definição é legal? Porque não?

```
TYPE
     Ponto = RECORD
                 quantidade: integer;
                 corte: Tamanho;
             END;
     Tamanho = (mini, medio, maxi);
```

11.4.2 Exercícios básicos

1. Declare um vetor em *Pascal* onde cada elemento é um registro com os campos: nome (uma *string*), DDD (um *integer*) e um telefone (*string*). Faça um programa que leia os dados de várias pessoas e imprima todos os nomes dos que são de Curitiba (DDD 41) ou do Rio de Janeiro (DDD 21).

Exemplo de entrada	Saída esperada
Fulano	
41	
90001823	
Beltrano	
16	
98989876	
Sicrano	
21	Fulano
96666666	Sicrano

2. Acrescente o campo `data_nascimento` no tipo do exercício anterior. Este campo deve ser um registro contendo dia, mês e ano, cada um deles sendo um número inteiro. Considerarando que o vetor não está ordenado, faça um programa em *Pascal* que encontre e imprima o nome do cliente mais jovem.

Exemplo de entrada	Saída esperada
Fulano 41 90001823 25 10 2001 Beltrano 16 98989876 13 8 1994 Sicrano 21 96666666 12 3 1999	 Beltrano

3. Considerando a estrutura definida no exercício anterior, faça um programa em *Pascal* que ordene os dados por ordem de nome e imprima, evidentemente por ordem de nome, todos os dados de todas as pessoas cadastradas, um por linha, dados separados por vírgula, sendo que as datas de nascimento devem ser impressas no formato padrão brasileiro.

Exemplos de entrada	Saída esperada
Fulano 41 90001823 25 10 2001 Beltrano 16 98989876 13 8 1994 Sicrano 21 96666666 12 3 1999	 Beltrano, 16, 98989876, 13/8/1994 Fulano, 41, 90001823, 25/10/2001 Sicrano, 21, 96666666, 12/3/1999

4. Considerando ainda a mesma estrutura de dados do exercício anterior, faça um programa que leia nome, DDD, telefone, e data de nascimento de outra pessoa e insira estes dados no vetor, lembrando que ele já está ordenado por ordem alfabética por nomes. A saída deve ser como no exercício anterior.

Exemplo de entrada	Saída esperada
Fulano 41 90001823 25 10 2001 Beltrano 16 98989876 13 8 1994 Sicrano 21 96666666 12 3 1999	 Beltrano, 16, 98989876, 13/8/1994 Fulano, 41, 90001823, 25/10/2001 Sicrano, 21, 96666666, 12/3/1999
Machin 14 96969696 27 4 1957	Beltrano, 16, 98989876, 13/8/1994 Fulano, 41, 90001823, 25/10/2001 Machin, 14, 96969696, 27/4/1957 Sicrano, 21, 96666666, 12/3/1999

11.4.3 Exercícios de dificuldade média

1. 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:

```
codDepto nomeDepto      inicio
1        vendas         0
2        contabilidade  8
3        estoque        7
4        entrega        2
```

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;
- 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).

11.4.4 Aplicações de registros

1. Considere o tipo PGM para imagens como definido na seção 10.5.1. 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;
                 m: matriz;
     end;

     imgcompactada = record
                 tam: integer;
                 v: vetor;
     end;

function calcula_valor_medio (var I: imagem): integer;
(* funcao que retorna o valor medio dos pixels da imagem, isto eh
   a soma de todos os elementos dividido pelo numero 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 eh uma representacao 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:

- **Imagem binarizada**:

```
P2
11 10
1
1  1  1  1  0  1  1  1  1  1  0
1  1  0  1  1  1  1  1  1  1  1
0  0  1  0  0  0  1  1  1  0  0
1  1  1  1  1  1  1  1  1  1  1
0  0  0  1  1  1  1  1  0  0  0
0  0  0  1  1  1  1  1  1  1  1
1  1  1  0  1  1  1  1  1  1  1
1  1  1  1  1  1  1  0  1  1  1
1  1  1  1  1  1  1  1  1  0  0
0  1  1  1  1  1  1  1  1  1  1
```

- **Imagem compactada (em duas linhas para caber na página)**:

```
36
4 0 5 0 2 0 8 0 0 1 0 0 0 3 0 0 11 0 0 0
5 0 0 0 0 0 0 11 0 14 0 12 0 0 0 10
```

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.

11.4.5 Exercícios dificeis

1. 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:

$$\mathbf{M} = \begin{bmatrix} 0 & 0 & 0 & 1.2 \\ 7.3 & 0 & 99 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 17 & 0 & 0 \end{bmatrix} \Longleftrightarrow \mathbf{M_e} = \left[\begin{array}{|c|c|} \hline 1 & 4 \\ \hline 1.2 \\ \hline \end{array} \begin{array}{|c|c|} \hline 2 & 1 \\ \hline 7.3 \\ \hline \end{array} \begin{array}{|c|c|} \hline 2 & 3 \\ \hline 99 \\ \hline \end{array} \begin{array}{|c|c|} \hline 3 & 2 \\ \hline 2 \\ \hline \end{array} \begin{array}{|c|c|} \hline 4 & 2 \\ \hline 17 \\ \hline \end{array} \right]$$

Para representar estas matrizes em *Pascal* podemos definir as seguintes estruturas de dados:

```
Const MAX = 1000; MAXESP = MAX*MAX/10;
Type t_matriz = record
        lin, col : integer;
        dados : array [1..MAX, 1..MAX] of real;
     end;
     elemento = record
        l, c : integer;
        val : real;
     end;
     t_matrizesp = record
        tam : integer;
        dados : array [1..MAXESP] of elemento;
     end;
```

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.

Capítulo 12

Tipos abstratos de dados

Um Tipo Abstrato de Dados (TAD)[1] 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 em um primeiro momento.

Evidentemente que em um segundo 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.

12.1 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.

[1]Agradecemos aos professores André Luiz Pires Guedes, Carmem Satie Hara, Daniel Alfonso de Oliveira e Luis 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:

```
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.

```
function conjunto_vazio (c: conjunto): boolean;
// Retorna true se o conjunto c eh vazio e false caso contrario.

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, mantem os elementos ordenados.

procedure remover_conjunto (x: longint; var c: conjunto);
// Remove o elemento x do conjunto c.

function uniao (c1, c2: conjunto): conjunto;
// Obtem a uniao dos conjuntos c1 e c2.

function interseccao (c1, c2: conjunto); conjunto;
// Obtem a interseccao dos conjuntos c1 e c2.

function diferenca (c1, c2: conjunto): conjunto
// Obtem a diferenca 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 contrario.

function sao_iguais (c1, c2: conjunto): boolean;
// Retorna true se o conjunto c1 = c2 e false caso contrario.

function contido (c1, c2: conjunto): boolean;
// Retorna true se o conjunto c1 esta contido no conjunto c2 e false caso contrario.

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.

```
procedure inicializar_conjunto (var c: conjunto);
// Cria um conjunto vazio. Deve ser chamado antes de qualquer operacao no conjunto.

function retirar_um_elemento (var c: conjunto): longint;
// Escolhe um elemento qualquer do conjunto para ser removido, remove, e o retorna.

procedure iniciar_proximo (var c: conjunto);
// Inicializa o contador que sera usado na funcao incrementar_proximo.

function incrementar_proximo (var c: conjunto; var x: longint): boolean;
// Incrementa o contador e retorna x por referencia. Retorna false se acabou conjunto
```

- *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.

```
// trecho de codigo para imprimir todos os elementos.
// o efeito eh que o conjunto c estara vazio ao final.
    while not conjunto_vazio (c) do
    begin
        x:= retirar_um_elemento (c);
        writeln (x);
    end;
```

- *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`.

```
// outro trecho de codigo para imprimir todos os elementos.
// o efeito eh que o conjunto c estara intacto ao final.
    iniciar_proximo (c);
    while incrementar_proximo (c,x) do
        writeln (x);
```

12.1.1 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 numero 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 12.1.

É importante é observar o *uso* da procedure *intersecção* e da função *cardinalidade*. Se estas implementarem corretamente a definição dos conceitos, o programa funciona perfeitamente.

A figura 12.2 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 12.1: 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 12.2: Lendo os números apostados ou sorteados.

Encontrando celebridades

Uma *celebridade* 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 celebridade pode não ter ido à festa. A questão é saber quem é a celebridade 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 celebridade é a pessoa representada na linha 2:

$$\begin{matrix} 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 \end{matrix}$$

É possível observar que a celebridade, 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 celebridade podemos procurar na matriz uma linha nula e depois verificar se a coluna só tem 1's, o que custaria N^2. Porem, podemos usar o TAD conjunto, pois além de se mais elegante, é bem mais eficiente, como veremos.

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 celebridade, mas B ainda pode ser, por isso reinserimos B de volta no conjunto. Caso B não conheça A, então A não pode ser celebridade, 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 celebridade, pois ela pode não estar presente na festa. Toda esta operação custa duas vezes o número de presentes e é portanto linear.

A figura 12.3 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 *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 12.4 mostra o código da função que faz o teste final, isto é, se o único elemento que sobrou no conjunto é de fato a celebridade.

```
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 12.3: 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 12.4: Confirmando se um presente é uma celebridade.

12.1.2 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[2].

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 11.2.

```
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 12.5, 12.6 e 12.7. 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 12.5. Isto torna trivial testar se um conjunto é vazio (figura 12.6) e saber a cardinalidade do conjunto (figura 12.7.), as informações dependem apenas do tamanho do vetor. A função que verifica se um elemento pertence a um conjunto, mostrada na figura 12.9, usa uma função interna ao TAD conjunto para fazer uma busca binária, conforme ilustrado na figura 12.8. 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.

[2]Este tipo de alocação não será coberto neste livro.

```
procedure inicializar_conjunto (var c: conjunto);
// Cria um conjunto vazio. Deve ser chamado antes de qualquer operacao no conjunto.
// Custo: constante.
begin
    c.tam:= 0;
end;
```

Figura 12.5: Inicializar um conjunto.

```
function conjunto_vazio (c: conjunto): boolean;
// Retorna true se o conjunto c eh vazio e false caso contrario.
// Custo: constante.
begin
    conjunto_vazio:= c.tam = 0;
end;
```

Figura 12.6: Testa se um conjunto é vazio.

```
function cardinalidade (c: conjunto): longint;
// Retorna a cardinalidade do conjunto c.
// Custo: constante.
begin
    cardinalidade:= c.tam;
end;
```

Figura 12.7: Encontra a cardinalidade do conjunto c.

```
function busca_binaria (x: longint; c: conjunto): longint;
// Retorna o indice do vetor que contem x, senao retorna zero.
// Custo: proporcial ao logaritmo do tamanho do conjunto.
var ini, fim, meio: longint;
begin
    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;
```

Figura 12.8: Função interna ao TAD: busca binária.

```
function pertence (x: longint; c: conjunto): boolean;
// Retorna true se x pertence ao conjunto c e false caso contrario.
// Custo: proporcial ao logaritmo do tamanho do conjunto.
begin
    // a funcao busca_binaria retorna o indice do vetor que contem x
    // se x nao esta no vetor, retorna zero.
    pertence:= busca_binaria (x,c) <> 0;
end;
```

Figura 12.9: Tenta encontrar um elemento no conjunto usando busca binária.

As procedures para inserção e remoção, ilustradas nas figuras 12.10 e 12.11 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.

```
procedure inserir_conjunto (x: longint; var c: conjunto);
// Insere o elemento x no conjunto c, mantem 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 espaco
        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 12.10: Procedure para inserir elemento em conjunto ordenado.

As figuras 12.12 e 12.13 mostram como são implementadas as funções *iniciar_proximo* e *incrementar_proximo*, respectivamente.

```
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 12.11: Procedure para remover elemento em conjunto ordenado.

```
procedure iniciar_proximo (var c: conjunto);
// Inicializa o contador que sera usado na funcao incrementar_proximo.
// Custo: constante.
begin
    c.proximo:= 1;
end;
```

Figura 12.12: Inicializa o contador próximo.

O código para o incremento foi feito como uma função que retorna *false* quando o contador chegou no 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 12.14. 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 12.15. 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 12.13: 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 12.14: Retira um elememento 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 12.15: 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 12.16.

```
function uniao (c1, c2: conjunto): conjunto;
// Obtem a uniao dos conjuntos c1 e c2.
// Custo: como estao ordenados, proporcial 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 // avanca apontador do primeiro vetor
        begin
            uni.v[k]:= c1.v[i]; i:= i + 1;
        end
        else if c1.v[i] > c2.v[j] then // avanca apontador do segundo vetor
        begin
            uni.v[k]:= c2.v[j]; j:= j + 1;
        end
        else // descarta um dos repetidos e avanca os dois apontadores
        begin
            uni.v[k]:= c1.v[i];
            i:= i + 1; j:= j + 1;
        end;
    end; (* while *)
    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 12.16: 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. Observem 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 12.17.

```
function interseccao (c1, c2: conjunto): conjunto;
// Obtem a interseccao dos conjuntos c1 e c2.
// Custo: como estao ordenador, proporcional ao tamanho do vetor c1.
// Possivel modificacao: 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 esta 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 12.17: 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 12.18 contém o código da operação de diferença entre conjuntos.

```
function diferenca (c1, c2: conjunto): conjunto;
// Obtem a diferenca dos conjuntos c1 e c2 (c1 - c2).
// Custo: linear por causa da ordenacao.
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 // sao 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 12.18: 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, apresentado na figura 12.19 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 12.20 mostra como copiar um conjunto em outro.

```
function contido (c1, c2: conjunto): boolean;
// Retorna true se o conjunto c1 esta contido no conjunto c2 e false caso contrario.
// 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 12.19: 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 12.20: 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 12.21 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.

```
function pertence (x: longint; c: conjunto): boolean;
// Retorna true se x pertence ao conjunto c e false caso contrario.
// Custo: proporcial ao tamanho do conjunto, isto eh, 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 12.21: Tenta encontrar um elemento no conjunto usando busca com sentinela.

O algoritmo para *união* também custará mais caro sem ordenação. Desta vez copiaremos o primeiro conjunto na união com custo linear, pois sabemos que não existem elementos repetidos nesse. O problema é o segundo conjunto: para cada elemento dele deve-se primeiro saber se ele já não ocorre na união. Esta pesquisa é feita em tempo linear para cada elemento resultando em um custo total quadrático. A figura 12.22, mostra este código.

```
function uniao (c1, c2: conjunto) conjunto;
// Obtem a uniao dos conjuntos c1 e c2.
// Custo: quadratico por causa do teste de pertinencia.
var i: longint; uni: conjunto;
begin
    inicializar_conjunto (uni);
    for i:= 1 to c1.tam do              // copia primeiro vetor
        inserir_conjunto (c1.v[i],uni);
    uni.tam:= c1.tam;
    for i:= 1 to c2.tam do              // insere elementos do segundo vetor
        inserir_conjunto (c2.v[i],uni);
    uniao:= uni;
end;
```

Figura 12.22: União de conjuntos com vetores não ordenados.

A intersecção também teve seu custo aumentado de linear para quadrático. Para cada elemento do primeiro vetor é preciso pesquisar se ele também está no segundo. Esta pesquisa é linear para cada elemento. A figura 12.23.

```
function interseccao (c1, c2: conjunto): conjunto;
// Obtem a interseccao dos conjuntos c1 e c2.
// Custo: quadratico, por causa do teste de pertinencia.
var i: longint; intersec: conjunto;
begin
    inicializar_conjunto (intersec);
    for i:= 1 to c1.tam do
        if pertence (c1.v[i],c2) then          // insere na interseccao
            inserir_conjunto (c1.v[i],intersec);
    intereseccao:= intersec;
end;
```

Figura 12.23: 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 12.24 mostra este código.

```
function diferenca (c1, c2: conjunto): conjunto;
// Obtem a diferenca dos conjuntos c1 e c2 (c1 - c2).
// Custo: quadratico, por causa do teste de pertinencia.
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 diferenca
            inserir_conjunto (c1.v[i],dif);
    diferenca:= dif;
end;
```

Figura 12.24: Diferença entre conjuntos com vetores não ordenados.

Da mesma forma, a função que testa se um conjunto está contido em outro, apresentado na figura 12.25 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 esta contido no conjunto c2 e false caso contrario.
// Custo: quadratico, por causa do teste da pertinencia.
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 12.25: Verifica se um conjunto está contido em outro com vetores não ordenados.

As procedures para inserção e remoção, ilustradas nas figuras 12.26 e 12.27 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 posicao do vetor.
// Custo: linear por causa do teste de pertinencia.
var i: longint;
begin
    if not pertence (x,c) then
    begin
        c.tam:= c.tam + 1;
        c.v[c.tam]:= x;
    end;
end;
```

Figura 12.26: 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 posicao 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 ultimo neste lugar.
        c.tam:= c.tam - 1;
    end;
end;
```

Figura 12.27: 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 12.28.

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.

```
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
        if contido (c1,c2) and contido (c2,c1) then
            sao_iguais:= true
        else
            sao_iguais:= false;
end;
```

Figura 12.28: Verifica se dois conjuntos são iguais em vetores não ordenados.

12.2 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:

pilha 1	pilha 2
2	
8	
5	2
5	1
4	3

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, embora elas sejam diferentes. 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 processamento 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 *o que é* uma pilha, e isso nos é suficiente no momento. Declaramos então uma pilha de números inteiros:

```
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.

```
procedure inicializar_pilha (var p: pilha);
// Cria uma pilha vazia.

function pilha_vazia (p: pilha): boolean;
// Retorna true se a pilha esta vazia e false caso contrario.

function pilha_unitaria (p: pilha): boolean;
// Retorna true se a pilha tem um unico elemento.

procedure empilhar (x: longint; var p: pilha);
// Insere x no inicio da pilha.

function desempilhar (var p: pilha): longint;
// Retorna o elemento do inicio da pilha e o remove dela.

function topo (p: pilha): longint;
// Retorna o elemento do inicio 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.

12.2.1 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 "abre 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 12.29 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 sobraram 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 12.29: 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 12.1.1. Ele pode também ser resolvido usando-se o TAD pilha.

A implementação é muito similar, conforme podemos ver na figura 12.30, 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 12.4 é o mesmo.

```
procedure eliminar_nao_celebridade (var p: pilha; m: matriz; n: longint);
var a,b: longint;
begin
      while not pilha_unitaria (p) do
      begin
            a:= desempilhar (p);
            b:= desempilhar (p);
            if conhece(a,b,m,n) then
                  empilhar (b,p)
            else
                  empilhar(a,p);
      end;
end;
```

Figura 12.30: Encontrando uma celebridade usando pilha.

12.2.2 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 computacionalmente, 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

```
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 `p.ini` como sendo zero. O código é mostrado na figura 12.31. A operação têm custo constante[3].

```
procedure inicializar_pilha (var p: pilha);
// Inicializa a estrutura.
// Custo: constante.
begin
    p.ini:= 0;
end;
```

Figura 12.31: Cria uma pilha vazia.

Nossa decisão torna trivial o teste de pilha vazia, é só testar se `p.ini` é zero ou não, conforme a figura 12.32 ilustra. Esta operação também tem custo constante.

```
function pilha_vazia (p: pilha): boolean;
// Retorna true se a pilha esta vazia e false caso contrario.
// Custo: constante.
begin
    pilha_vazia:= p.ini = 0;
end;
```

Figura 12.32: Testa se uma pilha é vazia.

Também trivial é a função mostrada na figura 12.33, já que, pela nossa implementação, o tamanho da pilha coincide com o valor de `p.ini`, fazendo que que o custo seja também constante.

```
function pilha_unitaria (p: pilha): boolean;
// Retorna true se a pilha tem um unico elemento e false caso contrario.
// Custo: constante.
begin
    pilha_unitaria:= p.ini = 1;
end;
```

Figura 12.33: 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 12.34.

[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

```
procedure empilhar (x: longint; var p: pilha);
// Insere x no inicio da pilha.
// Custo: constante.
begin
      p.ini:= p.ini + 1;
      p.v[p.ini]:= x;
end;
```

Figura 12.34: 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 inicio da pilha e o remove dela.
// Custo: constante.
begin
      desempilhar:= p.v[p.ini];
      p.ini:= p.ini - 1;
end;
```

Figura 12.35: 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 12.36.

```
function topo (p: pilha): longint;
// Retorna o elemento do inicio da pilha, sem remove-lo.
// Custo: constante.
begin
      topo:= p.v[p.ini];
end;
```

Figura 12.36: Retorna o elemento do topo da pilha, sem removê-lo.

Em resumo, todas as seis operações sobre o TAD *pilha* têm 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 celebridade 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.

12.3 Exercícios

Os exercícios desta seção não têm casos de teste, pois estamos em um capítulo avançado da disciplina. Você pode testar os seus códigos rodando testes que você julgar adequados.

1. Defina em *Pascal* um Tipo Abstrato de Dados *racional* que vai procurar abstrair números da forma $\frac{A}{B}$, onde A eB 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. As 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 polinomio. A estrutura tem que ser capaz de armazenar o grau e os coeficientes de um polinômio

$$a_0 + a_1 x + a_2 x^2 + \ldots + a_n x^n.$$

   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 $x \in R$, calcular o valor de $P(x)$;
   (d) obter o polinômio derivada de P, P'.
   (e) dado $x \in R$, 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:
`http://www.inf.ufpr.br/cursos/ci055/tad_conjunto/enunciado_tad.pdf`

9. Resolva o exercício que está na seguinte URL:
`http://www.inf.ufpr.br/cursos/ci055/Provas_antigas/final-20191.pdf`

Parte III

Aplicações

Introdução da parte 3

Agora estamos prontos para resolver problemas bastante interessantes, de relativa complexidade e, também, bastante motivadores para estudantes de programação.

Mostraremos através de algumas aplicações em jogos como se pode programar bem usando todos os conceitos fundamentais de algoritmos e estruturas de dados estudados até aqui.

De extrema importância, veremos como escrever programas usando uma técnica clássica: o desenvolvimento *top-down*, também conhecido como *refinamentos sucessivos*.

Capítulo 13

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 detalharmos as estruturas de dados e algoritmos necessários para a resolução do problema. Para isto vamos explorar ao

[1] `http://pt.wikipedia.org/wiki/Campo_minado`, em 12/07/2019.

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 13.1.

```
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 13.1: 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:

```
procedure inicializar_campo (var c: campo_minado);
// inicializa variaveis e outras estruturas necessarias para consistencia

procedure imprimir_campo (c: campo_minado);
// imprime o campo minado conforme as regras do jogo

procedure ler_jogada (var x,y: longint; c: campo_minado);
// (x,y) eh a coordenada escolhida, o campo serve para consistencia dos dados

procedure executar_jogada (x,y: longint; var c: campo_minado);
// dada a coordenada (x,y) concretiza a jogada segundo as regras do jogo

function ganhou (c: campo_minado): boolean;
// retorna true quando nao falta abrir nenhum quadrado sem mina

function perdeu (c: campo_minado): boolean;
// retorna true quando jogador clicou em uma mina
```

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 13.2 mostra a estrutura inicial.

```
const
    MAX = 9;                    // tamanho maximo da matriz

type
    quadrado = record
        info: longint;          // ou tem mina ou o numero de minas vizinhas
        revelado: boolean;      // quando true nao imprime o conteudo
    end;

    matriz = array [1..MAX,1..MAX] of quadrado;
```

Figura 13.2: 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. E 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 13.3.

```
    campo_minado = record
        x,y: longint;           // numero de linhas e colunas
        total_minas: longint;   // numero total de minas no campo
        falta_abrir: longint;   // numero de quadrados que faltam abrir
        status: longint;        // indica se o jogador ganhou ou perdeu o jogo
        m: matriz;
    end;
```

Figura 13.3: 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 `MAX = 9`), já definida anteriormente:

```
const
    MINA = -1;
    NADA = 0;
    GANHOU = 1;
    PERDEU = 2;
    EMANDAMENTO = 3;
```

Figura 13.4: Constantes para legibilidade e flexibilidade em alterações futuras.

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 13.5.

```
procedure inicializar_campo (var c: campo_minado);
// chamas as funcoes e procedimentos que iniciam a estrutura
begin
    write ('Defina numero de linhas (entre 1 e ',MAX,'): '); readln (c.x);
    write ('Defina numero de colunas (entre 1 e ',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:= EMANDAMENTO;                  // para inicializar variavel
    zerar_campo (c);
    gerar_minas (c);
    contar_vizinhos_com_mina (c);
end;
```

Figura 13.5: Inicializar campo minado.

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 (`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 `MAX+1`.

```
matriz = array [0..max+1,0..max+1] of quadrado;
```

Utilizaremos também uma nova constante para representar o valor da borda:

```
const BORDA = -2;
```

O procedimento para zerar o campo é apresentado na figura 13.6. Ele usa a função *eh_borda* (figura 13.7), tanto por legibilidade quanto por reaproveitamento de código, pois ela é utilizada mais de uma vez.

```
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:= BORDA
            else
                c.m[i,j].info:= NADA;
            c.m[i,j].revelado:= false;
        end;
end;
```

Figura 13.6: 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.

```
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 13.7: 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 13.8, 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 posicoes aleatorias 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 13.8: 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 adoção da borda, pois ela facilitou imensamente o código desta função, que é mostrado na figura 13.9.

```
procedure contar_vizinhos_com_mina (var c: campo_minado);
// para os quadrados que nao sao minas, conta os vizinhos que sao 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 13.9: Contando vizinhos com minas.

Esta procedure faz uso de uma função (figura 13.10) que efetivamente conta as minas vizinhas para cada quadrado e foi usada para deixar o código legível e modular.

```
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 13.10: 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 13.11.

```
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 13.11: 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:

```
const
    SIMBOLO_MINA = '@ ';              // representa uma mina
    SIMBOLO_SEM_MINA_VIZ = '  ';      // representa quadrado sem minas vizinhas
    SIMBOLO_ESCONDIDO = '* ';         // representa quadrado nao revelado
```

A figura 13.12 ilustra como queremos ver a tela em uma fase do jogo, enquanto que a figura 13.13 mostra como obtê-la.

```
   1 2 3 4 5 6 7 8 9
 +-------------------
1|
2|
3|              * * 1
4|            * * * 1
5| 1 * *      1 * * *
6| * * 2    1 * * * *
7| * * 2    * * 2 * *
8|          * * 2 * *
9|            * * * 1
```

Figura 13.12: 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_ESCONDIDO) // nao exibe o quadrado
            else
                case c.m[i,j].info of
                    MINA: write (SIMBOLO_MINA);
                    NADA: write (SIMBOLO_SEM_MINA_VIZ);
                     else
                         write (c.m[i,j].info,' ');
                end;
        writeln;
    end;
end;
```

Figura 13.13: Imprimindo o campo minado.

Ela funciona apenas para retângulos de no máximo 9×9. 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:

```
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:

```
type
    coordenada = record
        x,y: longint;
    end;
    pilha = record
        ini: longint;
        v: array [1..TAMMAXPILHA] 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 13.14. 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 13.15.

```
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 13.14: Procedure que executa as regras do jogo.

```
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 13.15: 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 13.16. 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 nulos, armazena na pilha. Assim, enquanto a pilha contiver elementos, pode haver quadrados para serem revelados ainda.

```
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 13.16: 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`.

Referências Bibliográficas

[Car82] S. Carvalho. *Introdução à Programação com Pascal*. Editora Campus, 1982.

[Feo99] H. Farrer and e outros. *PASCAL Estruturado*. Editora Guanabara Dois, 1999. 3a edição Guanabara Dois.

[Fre] FreePascal.org. Manuais on-line do freepascal. Disponíveis juntamente com o compilador em `http://www.freepascal.org`.

[Knu68] D. E Knuth. *The Art of Computer Programming*, volume 1–3. Addison Wessley, 1968.

[MF05] M.A. Medina and C. Fertig. *Algoritmos e Programação: Teoria e Prática*. Novatec, 2005.

[SB98] D.D. Salveti and L.M. Barbosa. *Algoritmos*. Makron Books, 1998.

[TLA95] A.M. Tenenbaum, Y. Langsam, and M.J. Augenstein. *Estruturas de Dados Usando C*. Makron Books, 1995.

[Tre83] P. Tremblay. *Ciência dos Computadores*. McGraw-Hill, 1983.

[Wir78] N. Wirth. *Programação Sistemática em PASCAL*. Editora Campus, 1978.