Capítulo 1

Introdução à Computação

Computadores são dispositivos que só sabem fazer um tipo de coisa: executar *algoritmos* para processar informação. Para cientistas da Computação, algoritmo é o conceito central da Computação.

Neste Tópico, introduziremos a noção de algoritmo, mostraremos alguns exemplos, abordaremos a relação algoritmo-computador e discutiremos sobre a Computação e suas áreas.

1.1 Algoritmo

Há tantas definições diferentes para o termo *algoritmo* quanto autores escrevendo sobre elas. Entretanto, todas estas definições concordam que um **algoritmo** é uma seqüência finita de instruções para resolver um problema, a qual possui as seguintes propriedades:

- **Garantia de término**: o problema a ser resolvido possui condições específicas que, quando satisfeitas, a execução do algoritmo é encerrada e o problema é então tido como "resolvido". Além disso, estas condições devem ser satisfeitas após uma quantidade finita de tempo, a ser contado a partir do início da execução do algoritmo.
- **Exatidão**: a intenção de cada instrução no algoritmo deve ser suficientemente clara, de forma que não haja ambigüidade na interpretação da intenção.
- **Efetividade**: cada instrução deve ser básica o suficiente para ser executada, pelo menos em princípio, por qualquer agente usando apenas lápis e papel.

Para exemplificar a noção de algoritmo, considere o problema de encontrar o máximo divisor comum (MDC) de dois números naturais quaisquer e a seguinte seqüência de instruções para resolver o problema:

1. Chame o maior número de a e o menor de b
2. Divida a por b e chame o resto de r
3. Se r é igual a zero então o MDC é igual a b e a execução das instruções encerra aqui. Caso contrário, siga para a próxima instrução.
4. Atribua o valor de b a a e o valor de r a b
5. Volte para a instrução 2.

Esta seqüência de instruções é um algoritmo para o problema de encontrar o MDC de dois números naturais quaisquer. Pois, se seguida, resolve qualquer ocorrência do problema[1]. A execução da seqüência sempre pára após uma quantidade finita de tempo. Isto é garantido pela instrução 3, que compara o valor de r a zero e termina a execução se r é igual a 0. Cada instrução da seqüência é clara e possível de ser executada por qualquer pessoa que saiba, pelo menos, dividir dois números.

Como era de se esperar, nem toda seqüência de instruções para resolver um determinado problema pode ser considerada um algoritmo. Por exemplo, se a instrução "Divida x por y se todo número inteiro par maior que 2 é a soma de dois números primos" estiver presente na seqüência, ela só poderá ser executada se soubermos se a proposição "todo número inteiro par maior que 2 é a soma de dois números primos" é verdadeira ou falsa. Entretanto, esta proposição, conhecida como **conjectura de Goldbach**, foi proposta em 1742 e continua sem solução até hoje. Logo, nossa instrução não pode ser executada por qualquer agente hoje em dia e, portanto, não pode fazer parte de um algoritmo.

Um outro exemplo de instrução que não pode fazer parte de um algoritmo é "Escreva todos os números ímpares". Neste caso, temos uma instrução que não pode ser executada porque a execução nunca terminará, apesar de sabermos exatamente como determinar os números ímpares. Observe que se modificássemos a instrução para "Escreva todos os números ímpares menores do que 100", ela poderia, perfeitamente, fazer parte de um algoritmo.

Um problema para o qual existe uma solução na forma de algoritmo é dito um **problema algorítmico**. O problema de encontrar o MDC de dois números naturais

[1] Se você não acredita nisso, pode começar a testar!

quaisquer é, portanto, um problema algorítmico. Problemas algorítmicos, em geral, possuem muitas *ocorrências*. Por exemplo, para o problema de encontrar o MDC de dois números naturais, cada ocorrência é uma dupla distinta de números naturais cujo MDC queremos encontrar. Um algoritmo é dito **correto** quando ele sempre termina e produz a resposta correta para *todas* as ocorrências de um dado problema.

Algoritmo, então, pode ser imaginado como a especificação de um processo "mecânico" que, quando executado, leva-nos à solução de algum problema. Embora o termo algoritmo esteja relacionado intimamente com Ciência da Computação, algoritmos tem sido parte de nossas vidas desde a primeira vez que uma pessoa explicou para outra como fazer alguma coisa. As pessoas utilizam algoritmos quando seguem receitas culinárias ou instruções para programar um vídeo cassete. Entretanto, nem todo algoritmo pode ser executado por um computador. Um computador pode executar apenas aqueles algoritmos cujas instruções envolvam tarefas que ele possa entender e executar. Este não é o caso, por exemplo, de instruções como "bata as gemas" e "ligue o vídeo cassete".

Computadores executam algoritmos que manipulam apenas *dados* e não coisas físicas, tais como gema de ovo e vídeo cassete. A execução de um algoritmo por um computador é denominada **processamento de dados** e consiste de três partes: uma *entrada*, um *processo* e uma *saída*. A **entrada** é um conjunto de informações que é requisitada para que as instruções do algoritmo possam ser executadas. O **processo** é a seqüência de instruções que compõe o algoritmo. A **saída** é o resultado obtido com a execução do processo para a entrada fornecida. Por exemplo, a entrada e a saída para uma computação do algoritmo para o problema de encontrar o MDC de dois números naturais são, respectivamente, dois números naturais e o MDC deles.

Quando escrevemos algoritmos para serem executados por computador, temos de fazer algumas suposições sobre o modelo de computação *entrada-processo-saída*. A primeira delas é que, a fim de realizar qualquer computação, o algoritmo deve possuir um meio de *obter* os dados da entrada. Esta tarefa é conhecida como **leitura** da entrada. A segunda, é que o algoritmo deve possuir um meio de *revelar* o resultado da computação. Isto é conhecido como **escrita** dos dados da saída. Todo e qualquer computador possui dispositivos através dos quais a leitura e a escrita de dados são realizadas.

1.2 Algoritmos e Resolução de Problemas

Todo algoritmo está relacionado com a solução de um determinado problema. Portanto, construir um algoritmo para um dado problema significa, antes de mais nada,

encontrar uma solução para o problema e descrevê-la como uma sequência finita de ações.

A tarefa de encontrar a solução de um problema qualquer é , em geral, realizada de forma empírica e um tanto quanto desorganizada; ocorrem vários procedimentos mentais, dos quais raramente tomamos conhecimento. A organização do procedimento de resolução de problemas é extremamente desejável, pois somente assim podemos verificar onde o procedimento não está eficiente. Identificadas as deficiências, procuramos formas de corrigi-las e, consequentemente, aumentamos a nossa capacidade de resolver problemas.

A capacidade para resolver problemas pode ser vista como uma habilidade a ser adquirida. Esta habilidade, como qualquer outra, pode ser obtida pela combinação de duas partes:

- Conhecimento: adquirido pelo estudo. Em termos de resolução de problemas, está relacionado a que táticas, estratégias e planos usar e quando usar;
- Destreza: adquirida pela prática. A experiência no uso do conhecimento nos dá mais agilidade na resolução de problemas.

Independente do problema a ser resolvido, ao desenvolvermos um algoritmo devemos seguir os seguintes passos:

- **Análise preliminar**: entender o problema com a maior precisão possível, identificando os dados e os resultados desejados;
- **Solução**: desenvolver um algoritmo para o problema;
- **Teste de qualidade**: executar o algoritmo desenvolvido com uma entrada para a qual o resultado seja conhecido;
- **Alteração**: se o resultado do teste de qualidade não for satisfatório, altere o algoritmo e submeta-o a um novo teste de qualidade;
- **Produto final**: algoritmo concluído e testado, pronto para ser aplicado.

1.3 Resolução de Problemas e Abstração

Talvez, o fator mais determinante para o sucesso em resolver um problema seja *abstração*. De acordo com o *Webster's New Dictionary of American Language* (segunda

edição), abstração "é alguma coisa independente de qualquer ocorrência particular" ou "o processo de identificar certas propriedades ou características de uma entidade material e usá-las para especificar uma nova entidade que representa uma simplificação da entidade da qual ela foi derivada". Esta "nova entidade" é o que chamamos de abstração.

Para entender o papel da abstração na resolução de problemas, considere a seguinte ocorrência de um problema que lembra nossos tempos de criança:

> *Maria tinha cinco maçãs e João tinha três. Quantas maçãs eles tinham juntos?*

Provavelmente, um adulto resolveria este problema fazendo uma abstração das maçãs como se elas fossem os números 5 e 3 e faria a soma de tais números. Uma criança poderia imaginar as cinco maçãs de Maria como cinco palitinhos e as três de João como três palitinhos. Daí, faria uma contagem dos palitinhos para chegar à solução. Em ambos os casos, os elementos do problema foram substituídos por outros (números e palitinhos) e a solução foi encontrada através da manipulação dos novos elementos.

O processo de abstração pode ser visto como constando de *níveis*. Isto diz respeito ao grau de simplificação de uma abstração. Por exemplo, minha avó cozinha desde garota, logo se alguém entregá-la uma receita culinária de uma lasanha ao molho branco e não mencionar como o molho branco é feito, provavelmente, isto não será um problema para ela. Entretanto, se a mesma receita for dada para mim, eu não saberei fazer a tal lasanha[2]. Isto quer dizer que a receita dada para mim deveria conter maiores detalhes do processo de preparo da lasanha do que aquela dada para minha avó. Aqui, a receita é uma abstração e o nível de detalhe da receita é proporcional ao nível de simplificação da abstração.

Algoritmos bem projetados são organizados em níveis de abstração, pois um mesmo algoritmo deve ser entendido por pessoas com diferentes graus de conhecimento. Quando um algoritmo está assim projetado, as instruções estão organizadas de tal forma que podemos entender o algoritmo sem, contudo, ter de entender os detalhes de todas as instrução de uma só vez. Para tal, o processo de construção de algoritmos conta com ferramentas, tais como módulos, que agrupam instruções que realizam uma determinada tarefa no algoritmo, independente das demais, tal como fazer a leitura da entrada, dispensando-nos de entender o detalhe de cada instrução separadamente, mas sim fornecendo-nos uma visão da funcionalidade do grupo de instruções.

[2] A menos que eu compre o molho branco no supermercado.

1.4 Algoritmos e Computadores

Desde que o homem começou a processar dados, ele tentou construir máquinas para ajudá-lo em seu trabalho. O computador moderno é o resultado dessas tentativas que vêm sendo realizadas desde o ano 1.200 a.C. com a invenção da calculadora denominada ábaco, a qual foi mais tarde aperfeiçoada pelos chineses. O computador é, sem dúvida alguma, um dos principais produtos da ciência do século XX.

O que é um computador? De acordo com o *Webster's New World Dictionary of the American Language* (segunda edição), um computador é "uma máquina eletrônica que, por meio de instruções e informações armazenadas, executa rápida e frequentemente cálculos complexos ou compila, correlaciona e seleciona dados". Basicamente, um computador pode ser imaginado como uma máquina que manipula informação na forma de números e caracteres. Esta informação é referenciada como **dado**. O que faz dos computadores uma máquina notável é a extrema rapidez e precisão com que eles podem armazenar, recuperar e manipular dados.

Quando desejamos utilizar um computador para nos auxiliar na tarefa de processamento de dados, deparamo-nos com alguns problemas inerentes a este processo: "Como informaremos ao computador o algoritmo que deve ser executado para obtermos o resultado desejado?", "Como forneceremos a entrada do algoritmo?" e "Como receberemos o resultado do algoritmo?"

O ato de instruir o computador para que ele resolva um determinado problema é conhecido como **programação**. Esta tarefa nada mais é do que inserir no computador as ações do algoritmo que corresponde à solução do problema e os dados referenciados pelas ações. Entretanto, antes de inserir as ações e os dados no computador, devemos reescrevê-las em uma linguagem apropriada para descrever algoritmos computacionais, ou seja, em uma **linguagem de programação**.

O termo **programa** é comumente empregado para designar o algoritmo em uma linguagem de programação. Entretanto, não há distinção conceitual entre algoritmo e programa no que diz respeito à linguagem em que eles estão escritos. A única diferença é que um programa não necessariamente termina. Um exemplo de programa que nunca termina é o sistema operacional de um computador. Como os programas a serem estudados neste curso sempre terminarão, nós utilizaremos os termos programa e algoritmo como sinônimos.

Cada computador possui uma linguagem de programação própria, denominada **linguagem de máquina**, e, em geral, distinta das linguagens de máquina dos demais modelos de computador. Esta é a única linguagem de programação que o computador realmente entende. No entanto, para evitar que nós tenhamos de aprender a linguagem

de máquina de cada computador diferente para o qual queremos programar, muitas linguagens de programação independentes de máquina foram criadas. Se você aprende uma linguagem independente de máquina, estará apto, pelo menos em princípio, a programar qualquer computador.

As linguagens de programação independentes de máquina não são compreendidas pelos computadores. Então, para que elas possam ser úteis para nós, um programa denominado **compilador** deve estar presente no computador. Um compilador para uma determinada linguagem de programação realiza a tradução automática de um programa para a linguagem de máquina. Tudo que nós temos a fazer para executar um programa escrito em uma linguagem de programação, que não é a linguagem de máquina do computador, é **compilar** o nosso programa com o compilador específico daquela linguagem.

Tanto o algoritmo quanto os seus dados de entrada são inseridos nos computadores por meio de equipamentos eletrônicos conhecidos como **periféricos de entrada**. O teclado e o *mouse* são exemplos de periféricos de entrada. As instruções e os dados inseridos no computador através de um periférico de entrada são armazenados em um dispositivo do computador denominado **memória**. Os dados de saída resultantes da execução do algoritmo pelo computador são apresentados também por meio de equipamentos eletrônicos denominados **periféricos de saída**. O vídeo e a impressora são exemplos de periféricos de saída.

O computador executa um determinado programa através de um dispositivo interno denominado **unidade central de processamento**, mais conhecido no mundo dos computadores pela sua abreviação em Inglês: **CPU** de *Central Processing Unit*. A CPU é responsável por buscar as instruções e os dados do programa que estão armazenados na memória do computador, decodificar as instruções e executar a tarefa descrita por elas com os respectivos dados. A CPU pode ser imaginada como o "cérebro" do computador.

No mundo dos computadores, você ouvirá as pessoas falarem sobre **hardware** e **software**. *Hardware* se refere à máquina propriamente dita e a todos os periféricos conectados a ela. *Software* se refere aos programas que fazem a máquina realizar alguma tarefa. Muitos "pacotes" de *software* estão disponíveis nos dias atuais. Eles incluem processadores de texto, sistemas gerenciadores de banco de dados, jogos, sistemas operacionais e compiladores. Você pode e aprenderá a criar seus próprios *softwares*.

Para aprender a criar *softwares*, você deverá adquirir capacidade para:

- Desenvolver algoritmos para solucionar problemas envolvendo transformação de informação;

- Usar uma linguagem de programação.

Inicialmente, você deve imaginar que aprender uma linguagem de programação, seja de máquina ou não, é a tarefa mais difícil porque seus problemas terão soluções relativamente fáceis. Nada poderia ser mais enganoso! **A coisa mais importante que você pode fazer como um estudante de Computação é desenvolver sua habilidade para resolver problemas**. Uma vez que você possui esta capacidade, você pode aprender a escrever programas em diversas linguagens de programação.

1.4.1 Hardware

Toda transmissão de dados, manipulação, armazenagem e recuperação é realmente realizada por um computador através de pulsos elétricos e magnéticos representando sequências de dígitos binários (**bits**), isto é, sequências de 0's e 1's. Cada sequência, por sua vez, é organizada em 1 ou mais **bytes**, que são grupos de oito bits. Neste contexto, as instruções e os dados manipulados pelo computador nada mais são do que sequências de bytes que possuem significado para o computador.

As instruções e os dados são armazenados na memória do computador. Um computador possui dois tipos de memória: a **primária** e a **secundária**. A primeira é também conhecida como **memória principal** ou **memória temporária** e tem como objetivo armazenar as instruções e os dados de um programa em execução. Esta memória se assemelha a um "gaveteiro", pois é uma sequência de posições de tamanho fixo, cada qual possuindo um identificador distinto denominado **endereço**.

Cada posição da memória primária armazena uma instrução ou parte dela ou um dado do programa. A CPU se comunica constantemente com a memória primária para obter a próxima instrução do programa a ser executada ou um dado necessário à execução da instrução. Tanto as instruções quanto os dados são localizados na memória através dos endereços das posições de memória que os contêm.

O tamanho de cada posição de memória é dado em *bytes*. Cada 1024 *bytes* representam 1 *quilobyte* (*1K*), cada 1024K representam 1 *megabyte* (*1M*), cada 1024M representam 1 *gigabyte* (*1G*) e cada 1024G representam 1 *terabyte* (*1T*). Se o tamanho de uma posição de memória de um computador mede 2 *bytes* e a memória possui 640 *Kbytes* de capacidade de armazenagem, o número de posições de memória é igual a 327.680.

A memória primária perde todo o seu conteúdo no momento em que o computador é desligado. Ela possui o propósito principal de armazenar instruções e dados de um programa em execução. Sua principal característica é a rapidez com que as informações

nela armazenadas são lidas e escritas pela CPU. Toda vez que um programa deve ser executado, ele é inserido antes na memória primária.

A memória secundária possui características opostas àquelas da memória primária. Ela é permanente, isto é, o computador pode ser desligado, mas ela não perde o seu conteúdo. Ela é mais lenta do que a memória principal e, em geral, possui muito mais capacidade para armazenagem de informação. O principal propósito da memória secundária é armazenar programas e dados que o computador pode executar e utilizar, respectivamente, em um dado instante.

Os discos rígidos, os discos flexíveis e os CD-ROM's são exemplos de memória secundária. Quando um programa armazenado em memória secundária precisa ser executado, o computador primeiro transfere o programa e os dados necessários a sua execução para a memória e, daí, inicia a execução do programa.

As memórias também podem ser classificadas quanto à permissão ou não para alterarmos o seu conteúdo. Os principais tipos nesta classificação são:

- Memória de acesso aleatório (*Random Access Memory* - RAM). Este tipo de memória permite a leitura e a escrita de seus dados em qualquer de suas posições. O acesso a qualquer posição é aleatório, isto é, podemos ter acesso a qualquer posição diretamente. A memória principal de um computador é uma memória do tipo RAM.
- Memória apenas para leitura (*Read Only Memory* - ROM). Este tipo de memória permite apenas a leitura de seus dados, como o próprio nome sugere. O conteúdo de uma ROM é gravado durante seu processo de fabricação, de acordo com a vontade do usuário. Uma vez que o usuário decidiu quais dados devem ser armazenados na ROM, ele os transmite ao fabricante da memória. Feita a gravação da ROM, o seu conteúdo não poderá mais ser alterado.

A CPU de um computador, devido a sua complexidade, é normalmente dividida, para fins de estudo e projeto, em duas partes:

- a Unidade de Controle (*Control Unit* - CU), onde as sequências de código binário, que representam as instruções a serem executadas, são identificadas e através da qual os dados são obtidos da memória; e
- a Unidade Lógica e Aritmética (*Arithmetic and Logic Unit* - ALU), onde as instruções são efetivamente executadas.

Toda instrução é codificada como uma sequência de *bits*. Alguns desses *bits* identificam a instrução propriamente dita e os demais contêm o endereço da posição de

memória dos dados usados pela instrução. A CU interpreta a sequência de *bits* e identifica qual é a instrução e, se houver referência a algum dado, realiza a busca do dado na memória. Estas operações são realizadas por um conjunto de circuitos lógicos que compõe a CU. A execução das instruções é realizada pela ALU.

A CPU também possui seus próprios elementos de memória. Eles são denominados **registradores**. Os registradores armazenam, em cada instante, os dados a serem imediatamente processados, isto é, os dados referenciados pela instrução processada no momento e que foram trazidos da memória principal. Os registradores possibilitam o aumento de velocidade na execução das instruções, pois os resultados intermediários da instrução não precisam ser armazenados na memória principal.

Com o avanço da microeletrônica é possível construir toda uma CPU em uma única pastilha de silício. Essa pastilha, ou *chip*, denomina-se **microprocesador**, sendo conhecido pelo nome de seu fabricante seguido de um determinado número, como por exemplo, Intel 80486. Os microprocessadores são classificados pelo tamanho da **palavra** - ou comprimento, em *bits*, da unidade de informação - que são capazes de processar de uma só vez. Os primeiros microprocessadores foram de 8 *bits*, seguidos pelos de 16 *bits*, depois pelos de 32 *bits* e, mais recentemente, pelos de 64 *bits*.

As **unidades de entrada**, que servem para introduzir programas ou dados no computador, e as **unidades de saída**, que servem para receber programas ou dados do computador, são denominadas **periféricos de entrada** e **periféricos de saída**, respectivamente. Os periféricos de entrada mais comuns são:

- Teclado;
- *Mouse*;
- Unidade de disco;
- *Scanner* e
- Leitora ótica.

E, alguns dos periféricos de saída mais comuns são:

- Vídeo;
- Impressora; e
- Unidade de disco.

1.4.2 Linguagens de Programação

A primeira geração de linguagens de programação remonta aos dias de codificação em linguagem de máquina. Esta linguagem é formada por instruções descritas como sequências de bytes, ou seja, ela é baseada em um alfabeto que possui apenas dois elementos, o *bit* 0 e o *bit* 1, e cada palavra (instrução) da linguagem é formada por grupos de oito *bits* denominados *bytes* que possuem significado para o computador. Portanto, um programa em linguagem de máquina poderia se parecer com a seguinte sequência de *bytes*:

01000011 00111010 00111011 01000001 00101011 01000100

O tamanho de uma instrução pode ser de 1 ou mais *bytes*, dependendo do número total de instruções da linguagem e do número máximo de operandos por instrução. Observe que com apenas 1 *byte* você pode codificar 256 instruções! Os dados utilizados por um programa também são codificados com 0's e 1's.

Para programar em linguagem de máquina, nós devemos conhecer a sequência de *bits* que determina cada instrução e também como codificar os dados em binário. Além disso, você deve conhecer os dispositivos internos do computador, pois as instruções de uma linguagem de máquina envolvem diretamente tais dispositivos.

Como a maioria dos problemas resolvidos por computadores não envolve o conhecimento dos dispositivos internos do computador, a programação em linguagem de máquina é, na maioria das vezes, inadequada, pois o desenvolvedor perde mais tempo com os detalhes da máquina do que com o próprio problema. Entretanto, para programas onde o controle de tais dispositivos é essencial, o uso de linguagem de máquina é mais apropriado ou, às vezes, indispensável.

O próximo passo na evolução das linguagens de programação foi a criação da linguagem montadora ou *assembly*. Nesta linguagem, as instruções da linguagem de máquina recebem nomes compostos por letras, denominados **mnemônicos**, que são mais significativos para nós humanos. Por exemplo, a instrução na linguagem montadora do processador 8088 que soma o valor no registrador CL com o valor no registrador BH e armazena o resultado em CL é dada por:

ADD CL,BH .

Esta instrução equivale a seguinte sequência de dois *bytes* na linguagem de máquina do 8088:

00000010 11001111 .

Para que o computador pudesse executar um programa escrito em linguagem montadora foi desenvolvido um compilador denominado **montador** ou **assembler**, o qual realiza a tradução automática de um código escrito em linguagem montadora para o seu correspondente em linguagem de máquina.

O sucesso da linguagem montadora animou os pesquisadores a criarem linguagens em que a programação fosse realizada através de instruções na língua inglesa, deixando para o próprio computador a tarefa de traduzir o código escrito em tais linguagens para sua linguagem de máquina. Isto foi possível devido à criação de compiladores mais complexos do que os montadores.

A primeira destas linguagens, que teve ampla aceitação, surgiu em 1957 e é ainda hoje utilizada. Trata-se da linguagem FORTRAN (*FORmula TRANslation*). A grande vantagem de linguagens como a FORTRAN é que o programador não necessita se preocupar com os detalhes internos do computador, pois as instruções da linguagem não envolvem os elementos internos do computador, tais como os registradores. Este fato também permitiu a execução do mesmo programa em computadores distintos sem haver alteração em seu "texto".

Na década de 70 surgiu a linguagem C e, na década de 80, a linguagem C++. Ambas constituem uma evolução na forma de estruturar as instruções de um programa e seus respectivos dados em relação as suas antecessoras. Outro aspecto importante da evolução das linguagens de programação diz respeito à quantidade de detalhe que o programador deve fornecer ao computador para que ele realize as tarefas desejadas.

1.4.3 Software

O *software* pode ser classificado como sendo de dois tipos: **básico** ou **aplicativo**. *Softwares* básicos são programas que administram o funcionamento do computador e nos auxiliam a usá-lo. *Softwares* aplicativos são programas que executam com o auxílio dos *softwares* básicos e realizam tarefas tipicamente resolvidas pelos computadores.

Os principais *softwares* básicos são:

- **Sistema Operacional**: conjunto de programas que gerencia o computador e serve de interface entre os programas do usuário e a máquina, isto é, controla o funcionamento do computador, as operações com os periféricos e as transferências de dados entre memória, CPU e periféricos.

Um sistema operacional pode ser classificado de acordo com a sua capacidade de execução de tarefas como:

  - **monotarefa**: sistema operacional que permite a execução de apenas um programa de cada vez. Por exemplo, o DOS;
  - **multitarefa**: sistema operacional que permite mais de um programa em execução simultaneamente. Por exemplo, o Unix e o Windows NT.

- **Utilitários**: programas de uso genérico que funcionam em conjunto com o sistema operacional e que têm como objetivo executar funções comuns em um computador. Por exemplo, formatadores de disco, programas que realizam transferência de dados entre computadores, entre outros.

- **Compiladores**: programas que traduzem um programa em uma linguagem de programação específica para seu equivalente em uma linguagem de máquina específica.

- **Interpretadores**: programas que, para cada instrução do programa, interpretam o seu significado e a executam imediatamente.

- **Depuradores**: programas que auxiliam o programador a encontrar erros em seus programas.

Um *software* aplicativo é aquele que realiza tarefas mais especializadas e que, apoiado nos *softwares* básicos, torna o computador uma ferramenta indispensável às organizações. Por exemplo, editores de texto, programas de desenho e pintura, programas de automação contábil, entre outros.

1.5 A Computação Como Disciplina

A disciplina de Computação é conhecida por vários nomes, tais como "Ciência da Computação", "Engenharia da Computação", "Informática" e assim por diante. Qualquer que seja a denominação, **Computação** pode ser entendida como " o estudo de processos sistemáticos que descrevem e transformam informação: suas teorias, análise, projeto, eficiência, implementação e a aplicação". A questão fundamental da Computação é "O que pode ou não ser automatizado?".

De acordo com o que vimos neste texto, os "processos sistemáticos que transformam a informação" são exatamente os algoritmos. Então, podemos dizer que a Computação é o estudo de algoritmos, mais especificamente, a teoria, análise, projeto, eficiência,

implementação e aplicação deles. Cada uma destas partes é abordada em uma área específica da Computação, a saber:

- **Arquitetura**. Esta área estuda as várias formas de fabricação e organização de máquinas nas quais os algoritmos possam ser efetivamente executados.
- **Linguagens de Programação**. Esta área estuda os métodos para projeto e tradução de linguagens de programação.
- **Teoria da Computação**. Aqui as pessoas perguntam e respondem questões tais como: "Uma determinada tarefa pode ser realizada por computador?" ou "Qual é o número mínimo de operações necessárias para qualquer algoritmo que execute uma certa tarefa?"
- **Análise de Algoritmos**. Esta área compreende o estudo da medida do tempo e do espaço que os algoritmos necessitam para realizar determinadas tarefas.
- **Projeto de algoritmos**. Esta área estuda os métodos para desenvolver algoritmos de forma rápida, eficiente e confiável.

As aplicações envolvendo algoritmos são responsáveis pelo surgimento de outras áreas da Computação, tais como Sistemas Operacionais, Bancos de Dados, Inteligência Artificial, entre outras.

Observe que a definição de Computação dada acima também menciona que a Computação compreende os "os processos sistemáticos que descrevem a informação". Isto é, a Computação envolve também o estudo de métodos para representar e armazenar dados a serem utilizados pelos algoritmos durante suas execuções. Sendo assim, podemos dizer que a Computação é o estudo de algoritmos e suas *estruturas de dados*.

Neste curso, nós estudaremos métodos para construir algoritmos e também algumas estruturas de dados simples para representar, no computador, os dados utilizados pelos algoritmos que construiremos. No ano seguinte, vocês estudarão algoritmos e estruturas de dados conhecidos e bem mais complexos do que aqueles que desenvolveremos neste ano.

Bibliografia

Este texto foi elaborado a partir dos livros abaixo relacionados:

1. Lambert, K.A., Nance, D.W., Naps, T.L. *Introduction to Computer Science with C++*. West Publishing Company, 1996.

2. Pothering, G.J., Naps, T.L. *Introduction to Data Structures and Algorithm Analysis with C++*. West Publishing Company, 1995.

3. Shackelford, R.L. *Introduction to Computing and Algorithms*. Addison-Wesley Longman, Inc, 1998.

4. Tucker, A., Bernat, A.P., Bradley, W.J., Cupper, R.D., Scragg, G.W. *Fundamentals of Computing I - Logic, Problem Solving, Programs, and Computers*. C++ Edition. McGraw-Hill, 1995.

Capítulo 2

Os Computadores HV-1, HV-2 e HIPO

Neste Tópico, estudaremos três computadores hipotéticos: HV-1, HV-2 e HIPO. O estudo do funcionamento destes três computadores nos auxiliará na compreensão do funcionamento dos computadores reais e também no aprendizado de conceitos fundamentais da programação de computadores, tais como os conceitos de variável e programa armazenado.

Uma pessoa que denominaremos **usuário** utilizará os computadores mencionados anteriormente para resolver seus problemas de processamento de dados. Cada um dos três computadores poderá funcionar sem a interferência do usuário até que a solução total do problema seja fornecida a ele.

2.1 O Computador HV-1

O computador HV-1 é formado pelos seguintes componentes:

- Um **gaveteiro** com 100 gavetas;
- Uma **calculadora** com mostrador e teclado;
- Um pequeno **quadro-negro** denominado EPI;
- Um **porta-cartões**;
- Uma **folha de saída**; e

- Um **operador** do sistema, uma pessoa chamada CHICO, com lápis, apagador de quadro-negro e giz.

2.1.1 Gaveteiro

O gaveteiro consiste numa sequência de gavetas numeradas de 00 a 99. O número de cada gaveta é denominado seu **endereço**. Cada gaveta contém um pequeno quadro-negro, onde é escrito um número sempre com 3 algarismos (por exemplo, 102, 003, etc.) e outras informações que veremos mais tarde. O gaveteiro é construído de tal maneira que valem as seguintes regras de utilização:

1. em qualquer momento, no máximo uma gaveta pode estar aberta;
2. a leitura do quadro-negro de uma gaveta não altera o que nele está gravado;
3. a escrita de uma informação no quadro-negro de uma gaveta é sempre precedida do apagamento do mesmo; e
4. somente o operador CHICO tem acesso ao gaveteiro.

2.1.2 Calculadora

Trata-se de uma calculadora usual, com teclado para entrada de números, teclas das quatro operações aritméticas básicas, tecla '=' e um mostrador, que denominaremos **acumulador**. Não há tecla de ponto (ou vírgula) decimal ou outra tecla adicional qualquer. Há dois tipos de operações efetuadas com essa calculadora:

1. carga de um número no acumulador. Para isso, pressiona-se a tecla '=' (garantindo-se assim o encerramento de alguma operação prévia) e a seguir "digitam-se" os algarismos do número a ser carregado, o qual aparece no acumulador;
2. operação aritmética. Esta é sempre feita entre o número que está no acumulador e um segundo número. Para isso, pressiona-se a tecla da operação desejada, digita-se o segundo número e pressiona-se a tecla '='. O resultado da operação aparece no acumulador.

Assim como o gaveteiro, a calculadora só pode ser utilizada pelo CHICO.

2.1.3 EPI

Trata-se de quadro-negro independente do gaveteiro, com a forma □□, onde será escrito um número entre 00 e 99, correspondendo a um endereço de gaveta do gaveteiro. O número nele escrito indica sempre o "Endereço da Próxima Instrução", donde sua abreviatura.

Somente o CHICO tem acesso ao EPI.

2.1.4 Porta-Cartões

É um dispositivo similar aos porta-cigarros onde são empilhados maços de cigarro a serem vendidos. O porta-cartões funciona de acordo com as seguintes regras:

1. cartões com informações são colocados exclusivamente pela parte superior, um a um; quando um cartão contém um número, este é sempre escrito com 3 algarismos, como por exemplo, 101, 003, etc;
2. cartões são retirados da extremidade inferior, um de cada vez, aparecendo na mesma ordem em que foram colocados no dispositivo;
3. a retirada de cartões só pode ser feita pelo CHICO; e
4. a colocação de cartões só pode ser feita pelo usuário.

2.1.5 Folha de Saída

Trata-se de uma folha de papel onde pode ser escrito um número em cada linha, utilizando-se sempre linhas consecutivas. Somente o CHICO pode escrever nessa folha; somente o usuário pode ler o que já foi escrito.

2.1.6 Operador

Resumindo as diversas características descritas, vemos que o operador CHICO é a única pessoa que tem acesso ao gaveteiro, à calculadora, ao EPI, e é o único que pode retirar cartões do porta-cartões e escrever na folha de saída. Ele executa estritamente ordens recebidas, não podendo tomar nenhuma iniciativa própria, executando alguma ação fora da especificação dessas ordens.

O CHICO trabalha sempre em um de dois estados diferentes:

1. **Estado de carga**, onde ele exclusivamente transcreve informações de cartões, lidos do porta-cartões, para gavetas do gaveteiro.
2. **Estado de execução**, onde ele executa ordens gravadas nas gavetas. Os detalhes do funcionamento desses estados serão explicados adiante.

A comunicação entre o usuário e o operador é feita exclusivamente através das unidades porta-cartões e folha de saída. O CHICO sabe fazer "de cabeça" uma única operação aritmética: incrementar de 1 o conteúdo do EPI.

2.1.7 Programando o HV-1

Para resolver um problema usando o computador HV-1, o usuário deve planejar uma sequência de ordens (o programa) a serem executadas pelo CHICO. Cada uma dessas ordens é denominada instrução. Um exemplo de instrução é o seguinte: "some o conteúdo da gaveta de endereço 41 ao conteúdo do acumulador". A fim de se produzir a execução correta das instruções e na sequência adequada, elas são escritas nas gavetas do gaveteiro. Para executar uma instrução da sequência, o CHICO segue os seguintes passos:

1. consulta o EPI, onde está escrito o endereço E da próxima instrução;
2. incrementa de 1 o conteúdo do EPI, apagando o valor anterior e escrevendo o novo valor (o qual neste caso será E+1);
3. abre a gaveta de endereço E; nesta gaveta ele deve encontrar uma instrução I, que é lida;
4. fecha a gaveta E; e
5. executa I.

Após finalizados esses passos, o CHICO recomeça do passo 1, com exceção de um caso explicado a seguir. Se a execução da instrução I não acarretar alteração no conteúdo do EPI, a próxima instrução a ser executada será a da gaveta de endereço E+1, devido ao passo 2. Se uma instrução acarretar alteração no EPI, mudando o seu conteúdo para X, a próxima instrução a ser executada será a da gaveta de endereço X; diz-se que houve um **desvio** para X.

As instruções escritas nas gavetas do gaveteiro constituem um **programa armazenado**. Para conseguir a execução de um programa, o usuário deve produzir inicialmente o armazenamento desse programa no gaveteiro, passando portanto a constituir um programa armazenado. Isso é feito da seguinte forma:

1. o usuário escreve cada instrução em um cartão, precedida de um endereço; assim, cada cartão do programa contém um par ordenado (E,I), onde E é um endereço e I uma instrução;
2. o CHICO é colocado em **estado de carga de programa**;
3. o usuário coloca o conjunto de cartões do programa no porta-cartões, em qualquer ordem;
4. como o CHICO está em estado de carga, ele lê um cartão com um par (E,I); abre a gaveta de endereço E; escreve em seu quadro-negro a instrução I; fecha essa gaveta;
5. o CHICO repete o passo 4 até ler o último cartão de programa, após o que ele é colocado em **estado de execução de programa**.

Após o encerramento da carga do programa, o CHICO é colocado em estado de execução de programa. Isso é feito por meio de um cartão especial, que deve encerrar o conjunto de cartões de programa. A forma desse cartão é "EXECUTE X", onde X é um número escrito pelo usuário; será o endereço da gaveta onde se encontra a primeira instrução a ser executada.

Ao ler esse cartão, o CHICO apaga o EPI e escreve o mesmo valor X; a seguir, ele vai para o passo 1 da execução de uma instrução, como exposto no início deste item.

Para completar este quadro, resta descrever como o CHICO entra em estado de carga de programa. Vamos supor que, na verdade, esse estado seja o estado "normal" do CHICO; ele só pode sair desse estado ao tentar carregar um cartão "EXECUTE X". Estando em estado de execução, ele só sai desse estado nos dois casos seguintes:

1. através da execução da instrução "pare a execução";
2. se ocorrer algum erro durante a execução.

   Um exemplo do caso 2 é o de o CHICO tentar executar uma instrução inválida, isto é, não conhecida.

2.1.8 Instruções do HV-1

O conteúdo de uma gaveta de endereço E, isto é, o número gravado em seu quadro-negro, será representado por cE. Assim, c10 indicará o conteúdo da gaveta 10. Indicaremos por cAC o conteúdo do acumulador; este será abreviado por AC.

Cálculo Aritmético

- Instrução: "some/subtrai/multiplique/divida cAC com cE".
- Significado: some/subtrai/multiplique/divida o cAC com cE e coloque o resultado no AC; o cE não se altera.
- Execução: o CHICO efetua os seguintes passos:
  1. digita a tecla '+/-/*/ /' da calculadora;
  2. abre a gaveta de endereço E;
  3. lê o número escrito nessa gaveta (cE) e digita-o na calculadora;
  4. fecha a gaveta E; e
  5. digita '=' na calculadora.

Daqui em diante, em lugar de escrevermos "gaveta de endereço E", escreveremos simplesmente E. Também deixaremos de mencionar explicitamente que é o CHICO quem efetua os passos da execução.

Carga no AC

- Instrução: "carregue o cE no AC".
- Significado: copie o cE no AC; o cE não muda.
- Execução:
  1. digita '=';
  2. abre E;
  3. lê cE e digita-o; e
  4. fecha E.

Armazenamento do AC

- Instrução: "armazene o cAC em E".
- Significado: copie o cAC em E; o cAC não muda (oposto da instrução anterior).
- Execução:
  1. abre E;
  2. apaga o cE;
  3. lê o cAC e o escreve em E; e
  4. fecha a gaveta.

Impressão

- Instrução: "imprima o cE".
- Significado: o cE é transcrito na folha de saída.
- Execução:
  1. abre E;
  2. lê cE e escreve seu valor na próxima linha da folha de saída; e
  3. fecha a gaveta.

Note que supusemos haver espaço na folha de saída. No caso contrário, o CHICO aguarda até ser fornecida nova folha.

Leitura

- Instrução: "leia um cartão e guarde em E".
- Significado: o conteúdo do próximo cartão do porta-cartões é lido e transcrito para E;
- Execução:
  1. abre E;
  2. retira um cartão do porta-cartões;
  3. lê o conteúdo do cartão e escreve o seu valor em E;

4. joga fora o cartão; e
5. fecha E.

Note que supusemos haver cartão no porta-cartões. Em caso contrário, o CHICO aguarda a colocação de pelo menos um cartão no porta-cartões.

Desvio Condicional

- Instrução: "se cAC$\neq$ / = / > / < / $\geq$ / $\leq$cE_1, desvie para E_2".
- Significado: Compara cAC com cE_1, caso verdadeiro a próxima instrução a ser executada está em E_2, caso contrário não há nada a fazer (isto é, a próxima instrução a ser executada estará na gaveta seguinte à que contém esta instrução).
- Execução:
  1. lê o cAC e cE_1; e
  2. Compara cAC com E_1 e caso verdadeiro apaga o EPI e escreve E_2 no mesmo.

Pare

- Instrução: "pare".
- Significado: encerra a execução do programa.
- Execução:
  1. entrega a folha de saída para o usuário; e
  2. entra no estado de carga.

2.1.9 Exemplo de Programa

Considere o seguinte problema:

" É dada uma sequência de números inteiros positivos; determinar sua soma".

Suponhamos que o usuário do computador HV-1 planeje resolver o problema da seguinte maneira:

1. cada número da sequência é escrito em um cartão;

2. dois cartões adicionais contendo o número 0 são colocados um imediatamente antes do primeiro cartão da sequência, e o outro logo após o último cartão;

3. o programa é escrito em cartões já no formato de carga de programa como mostrado na tabela abaixo:

endereço	**instrução**
01	leia um cartão e guarde em 11
02	leia um cartão e guarde em 12
03	imprima o c12
04	carregue no AC o c11
05	some o c12 ao AC
06	armazene o cAC em 11
07	carregue o c12 no AC
08	se cAC$\neq$ 0, desvie para 02
09	imprima o c11
10	pare

4. é formada uma pilha de cartões com a seguinte ordem: programa - EXECUTE 01 - cartões conforme 1 e 2 acima. Essa pilha é colocada no porta-cartões. Teremos nesta unidade, portanto, os cartões denominados cartões de programa e cartões de dados, precedendo e seguindo, respectivamente, o cartão EXECUTE; e

5. terminada a execução, é recebida a folha de saída, onde estarão impressos os números da sequência, seguidos da soma procurada.

Para compreendermos como funciona o processo descrito pelo programa e pelos cartões de dados, vejamos um exemplo concreto.

Seja a sequência 100, 5 e 31. Os cartões de dados conterão, conforme 1 e 2, os seguintes valores, pela ordem: 000, 100, 005, 031 e 000. Suponhamos que o CHICO tenha carregado o programa e tenha encontrado o cartão EXECUTE 01. Como vimos na Sub-Seção 2.1.7, ele apaga o EPI, escreve nele o número 01, e começa a executar as instruções do programa conforme os passos 1 a 5 descritos no início daquela Sub-Seção.

Se nós fizermos um acompanhamento do papel do CHICO na execução do programa, obteremos a tabela de execução a seguir:

g	**pc**	**cAC**	**c11**	**12**	**fs**	**cEPI**
	000,100,005,031,000					01
01	100,005,031,000		000			02
02	005,031,000		000	100		03
03	005,031,000		000	100	100	04
04	005,031,000	000	000	100	100	05
05	005,031,000	100	000	100	100	06
06	005,031,000	100	100	100	100	07
07	005,031,000	100	100	100	100	08
08	005,031,000	100	100	100	100	02
02	031,000	100	100	005	100	03
03	031,000	100	100	005	100,005	04
04	031,000	100	100	005	100,005	05
05	031,000	105	100	005	100,005	06
06	031,000	105	105	005	100,005	07
07	031,000	005	105	005	100,005	08
08	031,000	005	105	005	100,005	02
02	000	005	105	031	100,005	03
03	000	005	105	031	100,005,031	04
04	000	105	105	031	100,005,031	05
05	000	136	105	031	100,005,031	06
06	000	136	136	031	100,005,031	07
07	000	031	136	031	100,005,031	08
08	000	031	136	031	100,005,031	02
02		031	136	000	100,005,031	03
03		031	136	000	100,005,031,000	04
04		136	136	000	100,005,031,000	05
05		136	136	000	100,005,031,000	06
06		136	136	000	100,005,031,000	07
07		000	136	000	100,005,031,000	08
08		000	136	000	100,005,031,000	09
09		000	136	000	100,005,031,000,136	10
10		000	136	000	100,005,031,000,136	

onde “g” é o número da gaveta com a instrução, “pc” é o porta-cartões, “fs” é a folha de saída e “cEPI” é o conteúdo do EPI.

2.2 O Computador HV-2

As instruções do computador HV-1 estão escritas por extenso, diferindo assim dos números armazenados em outras gavetas, como as de endereço 11 e 12 no programa exemplo da Sub-Seção 2.1.9. Conseguiremos uma grande simplificação de notação e de funcionamento se codificarmos as instruções, transformando-as também em número. Como veremos mais tarde, isso permitirá inclusive a substituição, com relativa facilidade, do operador CHICO por dispositivos eletrônicos. Para simplificar a compreensão, suponhamos que cada gaveta do gaveteiro contenha um quadro-negro da seguinte forma:

□□□□□,

onde o CHICO só pode escrever números de 5 algarismos, como 00001, 00015, 00152, etc. O novo computador assim obtido receberá a sigla HV-2.

No computador HV-2, as instruções deverão ser necessariamente codificadas como números de 5 algarismos, para podermos gravá-las no gaveteiro. Elas terão a forma CEE, onde C é um dígito de 0 a F (em hexadecimal!)e corresponde ao **código da instrução**; E é um número de 00 a 99 e corresponde ao endereço da gaveta empregada na execução da instrução, denominado **código de endereço**. As instruções vistas na Seção 2.1.8 serão codificadas conforme a tabela dada a seguir:

instrução codificada	**instrução**
$1E_100$	carregue o cE_1 no AC
$2E_100$	armazene o cAC em E_1
$3E_100$	leia um cartão e guarde em E_1
$4E_100$	imprima o cEE
$5E_100$	some o cEE ao AC
$6E_100$	subtraia o cE_1 de AC
$7E_100$	multiplique o cE_1 com AC
$8E_100$	divida cAC por cE_1
$9E_100$	resto de cAC por cE_1
AE_1E_2	se cAC=E_1 desvie para E_2
BE_1E_2	se cAC≠E_1 desvie para E_2
CE_1E_2	se cAC>E_1 desvie para E_2
DE_1E_2	se cAC<E_1 desvie para E_2
EE_1E_2	se cAC≥E_1 desvie para E_2
FE_1E_2	se cAC≤E_1 desvie para E_2
00000	pare

Lembremos que cE significa conteúdo (agora sempre com 5 dígitos) da gaveta de

endereço E. Note que nas instruções que não são desvios os últimos dois dígitos são E_2=00 e na instrução "pare" usamos sempre E_1=E_2=00.

Por exemplo, a instrução 51200 encontrada pelo CHICO em alguma gaveta é interpretada por ele como "some o conteúdo da gaveta 12 ao conteúdo do acumulador e guarde o resultado no acumulador". Na tabela dada a seguir apresentamos o programa da Sub-Seção 2.1.9 codificado para o computador HV-2:

endereço	**instrução codificada**
01	31100
02	31200
03	41200
04	11100
05	51200
06	21100
07	11200
08	B0002
09	41100
10	00000

Observe um fato muito importante: é impossível, no modelo HV-2, distinguir-se o conteúdo de uma gaveta como correspondendo a uma instrução codificada ou a um número manipulado por certas instruções, o que não era o caso do modelo HV-1. Por exemplo, seguindo a execução do programa exemplo da Sub-Seção 2.1.9, vemos, na décima quarta linha, que a gaveta 11 recebe o conteúdo 105, correpondendo ao número "cento e cinco" (resultado da soma até esse momento) e não à instrução "carregue no AC o c05". Como pode o CHICO distinguir esses dois significados? Na verdade, a distinção é feita através da situação em que o CHICO se encontra ao se utilizar de uma gaveta (no caso, a 11). Assim, se ele estiver abrindo uma gaveta (no caso, a 11) à procura da próxima instrução a ser executada, o seu conteúdo será interpretado como sendo uma instrução codificada (no caso, a instrução 105). Por outro lado, se essa gaveta for aberta **durante** a execução de uma instrução, o seu conteúdo será usado como um valor numérico (no caso, o número 105).

A idéia de se armazenar as instruções da mesma maneira que os dados é atribuída ao famoso matemático americano John Von Neumann, que em meados da década de 1940 propôs esse esquema. Esse conceito foi um dos motivos que possibilitou a rápida evolução dos computadores daí para frente.

A codificação, por meio de números, de instruções que manipulam números é, em essência, a idéia fundamental. Uma idéia análoga foi aplicada na década de 1930 pelo

matemático alemão Gödel, o qual codificou numericamente teoremas sobre números, permitindo assim se enunciar teoremas sobre teoremas, chegando ao seu famoso "teorema da incompletude" dos sistemas axiomáticos.

2.3 O Computador HIPO

O fabricante dos computadores HV-2 percebeu, em um dado instante, que eles tinham as seguintes desvantagens:

1. os valores numéricos que podiam ser representados nas gavetas eram muito restritos, permitindo apenas 3 algarismos, sem sinal;
2. o operador CHICO era muito lento;
3. da mesma maneira, o mecanismo das gavetas e de outros componentes também era muito lento. Assim, propôs-se a fabricar um outro computador, que denominou de HIPO (cuja sigla provém de "computador hipotético"). Podemos descrever a sua organização comparando-a com a do HV-2.

2.3.1 Memória

Em lugar do gaveteiro, o HIPO dispõe de um dispositivo eletrônico, denominado **memória**, com regras de funcionamento análogas às do gaveteiro do HV-2. Este dispositivo contém partes, denominadas **células**, que correspondem às gavetas no gaveteiro. Cada célula tem comprimento de 8 bits (1 byte). O modelo mais simples do HIPO é produzido com memória de 32 células, de endereços 00 a 31.

Em cada célula podem ser representadas instruções codificadas como especificado mais adiante ou um número inteiro de -128 a 127, representado por complemento de 2 (dois).

2.3.2 CPU

No caso do computador HV-2, um operador, o CHICO, acionava todos os dispositivos, seja gaveteiro ou calculadora. No HIPO, esse papel é desempenhado por um sistema de circuitos eletrônicos, cujo funcionamento equivale às ações executadas pelo CHICO

no HV-2. Esse sistema é denominado de **Unidade de Controle** ou simplesmente UC.

Na Seção 2.1.6 foi dito que o CHICO pode estar em um dos dois estados: "carga" e "execução". Analogamente, a unidade de controle do HIPO estará um um desses dois estados em qualquer instante. O CHICO interpretava as instruções escritas nas gavetas do HV-2. As instruções interpretadas por essa UC do HIPO e armazenadas na memória têm o formato mostrado a seguir:

C	C	C	E	E	E	E	E

onde os dígitos binários "CCC" representam o código da instrução e os dígitos binários "EEEEE" representam o endereço de uma célula de memória.

No lugar da calculadora, o computador HIPO realiza as operações aritméticas por meio de um conjunto de circuitos denominado **Unidade Lógica e Aritmética** ou simplesmente ULA. Para executar uma operação de soma, por exemplo, impulsos eletrônicos são encaminhados à seção apropriada do circuito da ALU, iniciando uma sequência de operações que resulta na obtenção do resultado da operação no acumulador. O acumulador é um registrador acessível eletronicamente, isto é, não possui exibição visual.

2.3.3 EPI

O endereço da próxima instrução é um registrador eletrônico, sem exibição visual, com formato de um número com 5 algarismos binários. Somente a CPU do HIPO tem acesso ao EPI, podendo consultá-lo ou alterar seu conteúdo.

2.3.4 Unidade de Entrada

Em lugar do porta-cartões, o HIPO contém uma unidade de entrada com capacidade para ler eletronicamente **linhas de entrada** com o formato apresentado a seguir:

I/D :□□□□□□□□ e E :□□□□□.

Isto é, cada linha de entrada contém duas partes. Se a unidade de controle está em estado de execução, só é utilizada a parte I/D (iniciais de Instrução/Dado). O usuário

especifica na mesma um número de 8 dígitos binários, onde o dígito mais significativo representa o sinal do número (0-não negativo e 1-negativo). Uma instrução de leitura executada pela unidade de controle provoca a transcrição do número dado pelo usuário na linha de entrada para uma célula da memória. O endereço dessa célula é especificado pela instrução de leitura.

No estado de carga, o usuário especifica em I/D uma instrução conforme o formato dado na Seção 2.3.2 e, em E, o endereço da célula de memória onde a instrução deve ser carregada. O mesmo esquema do HV-1 é usado para se mudar a unidade de controle do estado de carga para o estado de execução e vice-versa.

Vários dispositivos podem ser usados como unidade de entrada: cartões perfurados (onde furos codificam os elementos das linhas), cartões marcados a lápis ou com tinta magnética, teclado como de máquina de escrever, etc. Todos eles transformam a representação externa acessível ao usuário em impulsos eletrônicos que são enviados pela unidade de controle à memória, posicionando os circuitos desta a fim de que as células envolvidas tenham um conteúdo equivalente à representação externa.

2.3.5 Unidade de Saída

Em lugar da folha de saída do HV-2, o HIPO contém uma unidade de saída com capacidade de gravar **linhas de saída**. Cada uma destas consiste em um número com 8 dígitos binários, onde o dígito mais significativo indica o sinal do número (0-não negativo e 1-negativo).

Novamente, nenhum dispositivo de saída particular foi indicado, podendo o mesmo ser uma máquina de escrever, uma impressora de linha, um terminal de vídeo, etc.

2.3.6 Instruções do HIPO

Vejamos algumas das instruções do computador HIPO. Supomos que as instruções tenham código de endereço “EEEEE”. Lembramos que cAC abrevia “conteúdo do acumulador”.

instrução codificada	significado
001EEEEE	carrega o cEEEEE no AC
010EEEEE	armazena o cAC em EEEEE
011EEEEE	lê uma linha de entrada e põe seu conteúdo em EEEEE
100EEEEE	grava o cEEEEE em uma linha de saída
101EEEEE	soma o cEEEEE ao cAC e guarda o resultado em AC
110EEEEE	desvia para EEEEE se cAC$\neq$ 0
111EEEEE	pare
000EEEEE	inicia o estado de execução com a instrução em EEEEE

2.3.7 Exemplo de Programa

A tabela a seguir ilustra o programa da Seção 2.1.9 escrito na linguagem de máquina do computador HIPO.

endereço	instrução
00000	01101010
00001	01101011
00010	10001011
00011	00101010
00100	10101011
00101	01001010
00110	00101011
00111	11000001
01000	10001010
01001	11100000

2.4 Bibliografia

O texto apresentado neste Capítulo foi retirado e adaptado, com fins didático, do Capítulo 2, "O Computador à Gaveta", do livro "Introdução à Computação e à Construção de Algoritmos" de autoria dos professores Routo Terada e Waldemar W. Setzer, publicado pela editora Makron Books do Brasil em 1992.

Capítulo 3

Desenvolvimento de Algoritmos - Parte I

Neste tópico, você encontrará uma breve descrição dos componentes de um algoritmo e aprenderá a construir algoritmos utilizando os componentes mais simples e uma linguagem de programação virtual.

3.1 Componentes de um Algoritmo

Quando desenvolvemos algoritmos, trabalhamos, tipicamente, com sete tipos de componentes: *estruturas de dados*, *variáveis*, *constantes*, *instruções de manipulação de dados*, *expressões condicionais*, *estruturas de controle* e *módulos*. Cada um destes tipos de componentes estão descritos brevemente a seguir:

- **Estrutura de dados**, **variáveis** e **constantes**. Dados são representações de informações usadas por um algoritmo. Isto inclui dados de entrada e saída, bem como dados gerados pelo algoritmo para seu próprio uso. Quando um algoritmo é executado por um computador, os valores dos dados por ele manipulados devem ser armazenados em algum "depósito", de modo que tais valores estejam disponíveis para serem usados pelo algoritmo a qualquer momento. A definição da organização interna de tais "depósitos", bem como da relação entre suas partes, é conhecida como estrutura de dados.

  Na maior parte das vezes, desejamos que um "depósito" de dados seja *variável*, isto é, que o seu conteúdo possa ser alterado durante a execução do algoritmo.

Por exemplo, no algoritmo para calcular o MDC de dois números naturais (ver Capítulo 1), a, b e r são "depósitos" para valores de dados cujos conteúdos mudam durante a execução do algoritmo. Outras vezes, queremos que o conteúdo de um "depósito" de dados seja *constante*, ou seja, que ele não seja alterado durante a execução do algoritmo.

- **Instruções para Manipulação de Dados**. Qualquer algoritmo requer instruções que façam o seguinte: obtenham valores de dados fornecidos pelo usuário e armazenem-os em estruturas de dados; manipulem aqueles valores de dados, isto é, modifiquem o valor de uma variável através de operações aritméticas, copiem o valor de uma variável para outra, entre outras; comuniquem os valores de dados resultantes do algoritmo ao usuário.

- **Expressões condicionais**. Algoritmos também apresentam "pontos de decisão". A capacidade de um algoritmo de tomar decisões é o que o faz potente. Se um algoritmo não pudesse fazer mais do que seguir uma única lista de operação, então um computador nada mais seria do que uma máquina de calcular.

  Tais decisões são baseadas em expressões condicionais que, quando avaliadas, resultam em apenas um dos seguintes dois valores: *verdadeiro* ou *falso*. Por exemplo, no algoritmo para calcular o MDC de dois números naturais (ver Capítulo 1), a comparação "r é igual a zero" é um exemplo de expressão condicional. O resultado desta comparação pode ser apenas verdadeiro ou falso, dependendo se r é igual a zero ou não, respectivamente.

  Se uma expressão condicional é verdadeira, o algoritmo pode agir de diferentes formas, tal como executar certas instruções ou não executá-las. Portanto, a partir do resultado de uma expressão condicional, o algoritmo pode tomar decisões diferentes. No algoritmo para calcular o MDC, se a expressão condicional "r é igual a zero" é verdadeira, o algoritmo encerra sua execução. Do contrário, ele continua a execução.

- **Estruturas de controle**. Os elementos de um algoritmo que governam o que ocorre *depois* que um algoritmo toma uma decisão são denominados estruturas de controle. Sem estruturas de controle, as decisões não possuem qualquer valor. Isto é, o que adianta tomar uma decisão se você não pode efetuá-la? Sem estruturas de controle, o algoritmo é apenas uma lista de instruções que deve ser executada sequencialmente. Estruturas de controle permitem que um algoritmo possa tomar decisões e, a partir delas, executar algumas instruções ou não, repetir a execução de certas instruções e executar um grupo de instruções em detrimento de outro.

  No algoritmo para calcular o MDC de dois números naturais foi utilizada uma estrutura de controle condicional que permite encerrar ou não a execução do

algoritmo, dependendo do resultado da avaliação da expressão condicional "r é igual a zero": "se r é igual a zero então o MDC é igual a b e a execução encerra aqui. Caso contrário, siga para a próxima instrução".

- **Módulos**. Algoritmos podem se tornar muito complexos, de modo que agrupar todos os seus componentes em uma única unidade fará com que o algoritmo seja difícil de entender, difícil de manter e difícil de estender. Para evitar que isso ocorra, construímos nossos algoritmos utilizando módulos, que são trechos de algoritmos que agrupam instruções e dados necessários para a realização de uma dada tarefa lógica do algoritmo, que seja a mais independente possível das demais.

  A utilização de módulos facilita o entendimento do algoritmo, pois eles são menores do que o algoritmo como um todo e podem ser entendidos individualmente, isto é, para entendermos um deles não precisamos, necessariamente, entender os demais, visto que eles são partes independentes do algoritmo. A manutenção do algoritmo também é facilitada, pois a modificação de um módulo não deve afetar os demais, desde que o módulo continuará fazendo o que ele fazia antes. Por último, a extensão de um algoritmo pode ser feita mais facilmente se pudermos reutilizar módulos já existentes.

3.2 Uma Linguagem para Algoritmos

A experiência tem mostrado que é necessário expressar idéias algorítmicas em uma forma mais precisa e mais compacta do que aquela encontrada em uma "linguagem natural", tal como Português. Há muitas linguagens que poderíamos utilizar com o propósito de escrever algoritmos, incluindo qualquer linguagem de programação, as quais nos permitem escrever algoritmos em uma forma que os computadores podem entender.

A vantagem de usar uma linguagem de programação é que podemos, de fato, executar nossos algoritmos em um computador. Entretanto, para o propósito de introduzir conceitos algorítmicos, esta abordagem possui três sérias desvantagens:

- Linguagens de programação são criadas com algum propósito específico em mente e, desta forma, enfatizam algumas características em detrimento de outras. Não há linguagem de programação que possa ser considerada "a melhor". A opção por qualquer uma delas resultaria em compromissos que omitem a diferença entre propriedades importantes de algoritmos e as propriedades de certos tipos de programas.

- As linguagens de programação incluem muitos detalhes técnicos que introduzem dificuldades extras que não estão relacionadas com o aspecto lógico da resolução de problemas algorítmicos.
- Os praticantes se tornam tão envolvidos com os aspectos técnicos das linguagens que desviam a atenção da importância de um bom projeto de algoritmo.

Por estas razões, neste curso, os algoritmos serão descritos em pseudocódigo (uma linguagem virtual de programação). Entretanto, oportunamente, traduziremos alguns algoritmos que construiremos na sala de aula para seus equivalentes na linguagem de programação C++.

3.3 Tipos de Dados

Os dados manipulados por um algoritmo podem possuir natureza distinta, isto é, podem ser números, letras, frases, etc. Dependendo da natureza de um dado, algumas operações podem ou não fazer sentido quando aplicadas a eles. Por exemplo, não faz sentido falar em somar duas letras. Para poder distingüir dados de naturezas distintas e saber quais operações podem ser realizadas com eles, os algoritmos lidam com o conceito de tipo de dados.

O **tipo de um dado** define o conjunto de valores ao qual o valor do dado pertence, bem como o conjunto de todas as operações que podem atuar sobre qualquer valor daquele conjunto de valores. Por exemplo, o tipo de dados numérico pode ser imaginado como o conjunto de todos os números e de todas as operações que podem ser aplicadas aos números.

Os tipos de dados manipulados por um algoritmo podem ser classificados em dois grupos: *atômicos* e *complexos*. Os tipos atômicos são aqueles cujos elementos do conjunto de valores são indivisíveis. Por exemplo, o tipo numérico é atômico, pois os números não são compostos de partes. Por outro lado, os tipos complexos[1] são aqueles cujos elementos do conjunto de valores podem ser decompostos em partes mais simples. Por exemplo, o tipo cadeia é aquele cujos elementos do conjunto de valores são frases e frases podem ser decompostas em caracteres, os quais são indivisíveis.

Entre os tipos atômicos, os mais comuns são:

- **numérico**: inclui os números inteiros, racionais e irracionais e as operações aritméticas e relacionais para estes números. Na nossa linguagem de descrição

[1] Alguns autores utilizam o termo *estruturado* ao invés de complexo.

de algoritmos, os números inteiros são escritos apenas como a concatenação dos dígitos $0, 1, 2, 3, 4, 5, 6, 7, 8$ e 9. Por exemplo, 5, 100 e 1678. Os números com fração devem ser escritos, nos algoritmos, com a presença do ponto decimal. Por exemplo, 3.14159, 100.0 e 0.5. No caso do número ser negativo, adicionamos o sinal "−" na frente do número, tal como -23. Dentre os tipos numéricos distinguiremos os tipos **inteiro** e **real** que incluem, respectivamente, os números inteiros e reais.

- **caracter**: inclui os símbolos do alfabeto usado pela nossa linguagem de programação virtual. Estes símbolos consistem das letras do alfabeto romano, dos dígitos $0, 1, \ldots, 9$, dos caracteres de pontuação, tais como ?, ., ..., dos símbolos de operação aritmética + e −, entre outros. As operações definidas para este tipo são igualdade e desigualdade. Os elementos do conjunto de valores do tipo caracter devem ser escritos, nos algoritmos, entre aspas simples, como, por exemplo, 'a', '1' e '?'. Note a diferença entre escrever 1 e '1'. No primeiro caso, temos o número inteiro um; no segundo, o caracter representando o dígito um.

- **lógico**: inclui apenas os valores *verdadeiro* e *falso* e as operações lógicas. Nos algoritmos, estes valores são escritos como V e F.

Entre os tipos complexos, vamos utilizar inicialmente apenas o tipo **cadeia**, cujo conjunto de valores é formado por *frases* ou *cadeias*. Por frase, entendemos uma concatenação de valores do tipo caracter. As frases são escritas, nos algoritmos, entre aspas duplas, tal como "Entre com algum valor: ". Note que "a" e 'a' são valores pertencentes a tipos distintos. No primeiro caso, temos uma frase que contém apenas o caracter a; no segundo, temos o caracter a.

3.4 Variáveis

Como dito antes, um algoritmo manipula dados, que podem ser dados variáveis ou constantes. Por exemplo, em um algoritmo que calcula a área de um círculo, o raio do círculo é um dado de entrada variável, pois o valor do raio pode variar de círculo para círculo. Por outro lado, o valor do número π, utilizado no cálculo da área do círculo[2], é uma constante. Não importa qual seja o raio do círculo, o mesmo valor π é utilizado no cálculo da área.

Um algoritmo representa dados variáveis e dados constantes através de *variáveis* e *constantes*, respectivamente. Uma **variável** pode ser imaginada como um "depósito"

[2]A área do círculo é dada por πr^2, onde r é o raio do círculo.

para armazenar valores de dados, para o qual existe um nome, conhecido como *identificador*, e cujo conteúdo pode ser alterado pelo algoritmo. O identificador de uma variável deve ser distinto daquele das demais variáveis do algoritmo, pois é através do identificador da variável que o algoritmo a distingüe das demais e tem acesso ao seu conteúdo.

O ato de criar uma variável é conhecido como **declaração de variável**. Cada variável utilizada em um algoritmo deve ter sido declarada antes de ser utilizada pela primeira vez. Ao criarmos uma variável, temos de, explicitamente, associar-lhe um tipo de dados, pois o tipo determina o conjunto de valores que a variável pode armazenar e, conseqüentemente, a estrutura de dados que define o conteúdo da variável. Desta forma, se uma variável é declarada como sendo do tipo numérico, ela não poderá armazenar qualquer outro valor que não seja um número.

Para se declarar uma variável, segue-se o formato abaixo:

<tipo da variável> <lista de identificadores>

onde *tipo da variável* é o nome do tipo ao qual as variáveis estarão associadas e *lista de identificadores* é uma lista de identificadores de variáveis separados por vírgula.

Como exemplo, considere a declaração de duas variáveis, denominadas x do tipo inteiro e y do tipo real:

inteiro x
inteiro y

Quando executamos um algoritmo em um computador, a cada variável corresponde uma posição distinta de memória, em geral, o endereço da primeira célula de memória ocupada pelo conteúdo da variável. Variáveis, portanto, nada mais são do que representações simbólicas para posições de memória que armazenam dados. O uso de variáveis é uma característica das linguagens de alto-nível, pois, como pudemos constatar no capítulo anterior, quando programamos em linguagem de máquina, utilizamos os próprios endereços de memória para referenciar os dados manipulados pelo programa.

3.5 Constantes

Uma **constante** faz exatamente o que o nome sugere: representa um dado cujo valor não muda durante todo o algoritmo. Assim como uma variável, uma constante

também possui um identificador único e deve ser declarada antes de ser utilizada. No entanto, como o valor da constante é fixo, ele é atribuído à constante no momento de sua declaração, de modo que não há necessidade de explicitarmos o tipo do valor da constante.

Em nossa linguagem de escrita de algoritmos, uma constante é declarada da seguinte forma:

defina <identificador> <valor>

onde *defina* é uma palavra-chave, *identificador* é o identificador da constante e *valor* é o valor da constante. Entenda por **palavra-chave** qualquer palavra que tenha um significado específico para o algoritmo e que não deve ser utilizada para qualquer outro propósito, tal como para dar um nome a uma variável. Neste curso, as palavras-chaves sempre aparecerão sublinhadas.

Como exemplo, temos a declaração de duas constantes, PI e $MENSAGEM$:

defina PI 3.14159
Defina $MENSAGEM$ "A área do círculo é:"

Desta forma, PI é uma constante do tipo numérico e $MENSAGEM$ é uma constante do tipo cadeia, ou seja, $MENSAGEM$ é uma frase.

3.6 Operadores

Nós declaramos variáveis e constantes a fim de criar espaço para armazenar os valores dos dados manipulados pelo algoritmo. Uma vez que declaramos as variáveis e constantes, temos a nossa disposição vários tipos de *operadores*, com os quais podemos atribuir valor a uma variável e manipular os valores armazenados em variáveis e constantes. Há três categorias básicas de operadores: operadores de *atribuição*, operadores *aritméticos* e operadores de *entrada e saída*.

3.6.1 Operador de Atribuição

O ato de atribuir ou copiar um valor para uma variável é conhecido como **atribuição**. Utilizaremos o operador de atribuição (←) como um símbolo para esta operação.

Por exemplo, as seguintes instruções atribuem os valores 4 e 'a' às variáveis x e y, respectivamente:

$x \leftarrow 4$
$y \leftarrow$'a'

Após tais operações de atribuição serem executadas, x conterá o valor 4 e y, o valor 'a'. Lembre-se de que, para qualquer variável ser utilizada em um algoritmo, temos de declará-la antes de seu primeiro uso.

3.6.2 Operadores Aritméticos

Os operadores aritméticos básicos são quatro:

- **adição**, representado pelo símbolo +;
- **subtração**, representado pelo símbolo −;
- **multiplicação**, representado pelo símbolo ∗; e
- **divisão**, representado pelo símbolo /.

Estes operadores podem ser aplicados a expressões envolvendo valores numéricos quaisquer, não importando se o número é inteiro ou contém parte fracionária.

Como exemplo, suponha que as variáveis a, b e c tenham sido declaradas como segue:

inteiro a, b, c

Então, podemos escrever expressões como as seguintes:

$a + b + c$
$a - b * c/2$

A operação de divisão, representada pelo símbolo /, é a divisão real. Isto é, mesmo que os operandos sejam números inteiros, o resultado é real. Entretanto, para podermos trabalhar apenas com aritmética inteira, existem dois outros operadores: DIV e MOD. DIV é o operador de divisão para números inteiros. Se fizermos 4 DIV 3, obteremos 1 como resultado, ao passo que 4/3 resulta em 1.333 . . .. Já o operador MOD é utilizado para obtermos o resto de uma divisão inteira. Por exemplo, 5 MOD 2 é igual a 1, o resto da divisão inteira de 5 por 2.

3.6.3 Comando de Atribuição

Um **comando de atribuição** é qualquer comando que inclua um operador de atribuição. Este comando sempre envolve uma variável, que se localiza à esquerda do operador, e um valor ou uma expressão que se localiza à direita do operador. Quando um comando de atribuição é executado, o valor do lado direito do operador de atribuição, que pode ser uma constante, o conteúdo de uma variável ou o resultado de uma expressão, é copiado para a variável do lado esquerdo do operador.

Como exemplo, considere os seguintes comandos de atribuição:

$x \leftarrow a + 2 - c$
$y \leftarrow 4$
$z \leftarrow y$

onde a, c, x, y e z são variáveis.

3.6.4 Precedência de Operadores

Assim como na aritmética padrão, a precedência de operadores nas expressões aritméticas dos algoritmos também é governada pelo uso de parênteses. A menos que os parênteses indiquem o contrário, as operações de divisão e multiplicação são executadas primeiro e na ordem em que forem encontradas ao lermos a expressão da esquerda para a direita. Em seguida, temos as operações de adição e subtração.

Como exemplo, considere o seguinte trecho algorítmico:

inteiro a
inteiro b
inteiro c
inteiro x

$a \leftarrow 2$
$b \leftarrow 3$
$c \leftarrow 4$

$x \leftarrow a + b * c$

Este trecho de código armazenará o valor 14 na variável x. Se fosse desejado realizar a operação de adição primeiro, o comando de atribuição seria

$$x \leftarrow (a + b) * c$$

Neste caso, o uso dos parênteses fez com que a operação de adição fosse realizada primeiro. Na nossa linguagem para escrita de algoritmos, a utilização de parênteses possui o mesmo significado visto no primeiro grau. Além de possuírem prioridade máxima perante os demais operadores aritméticos, os parênteses também podem ocorrer de forma "aninhada", como na expressão abaixo:

$$((2 + 3) - (1 + 2)) * 3 - (3 + (3 - 2))$$

que resulta no valor 2.

3.6.5 Entrada e Saída

Qualquer algoritmo requer a obtenção de dados do "mundo" (entrada) e também um meio de comunicar ao "mundo" o resultado por ele obtido (saída). Para tal, existem duas operações, denominadas **entrada** e **saída**, realizadas, respectivamente, pelos operadores leia e escreva. O operador de leitura tem sempre um ou mais variáveis como operandos, enquanto o operador de saída pode possuir constantes, variáveis e expressões como operandos. A forma geral destes operadores é:

leia <lista de variáveis>
escreva <lista de variáveis e/ou constantes e/ou expressões >

onde *leia* e *escreva* são palavras-chave, *lista de variáveis* é uma lista de identificadores separados por vírgula, e *lsita de identificadores e/ou constantes e/ou expressões*, como o próprio nome já define é uma lista contendo identificadores (de constantes ou variáveis), constantes e expressões, independente de alguma ordem.

Como exemplo do uso dos operadores de entrada e saída, temos:

inteiro x

leia x
escreva x
escreva "O valor de x é:", x

3.7 Estrutura Geral de um Algoritmo

Nesta Seção, veremos algumas características que encontraremos nos algoritmos que estudaremos neste curso:

1. A *linha de cabeçalho*, que é a primeira linha do algoritmo, contém a palavra-chave algoritmo, seguida por um identificador, que é o nome escolhido para o algoritmo.
2. A *declaração de constantes e variáveis* será o próximo passo, sendo que a declaração de constantes precede a declaração de variáveis.
3. O *corpo do algoritmo* contém as instruções que faze parte da descrição do algoritmo.
4. A *linha final*, que é a última linha do algoritmo, contém a palavra-chave fimalgoritmo para especificar onde o algoritmo termina.

Temos ainda alguns detalhes que servirão para deixar o algoritmo mais claro e mais fácil de ler:

1. Os passos do algoritmo são especificados através da *identação* dos vários comandos entre a linha de cabeçalho e a linha final. A identação, visualmente, enquadra o corpo do algoritmo no olho humano e acaba com a confusão criada por não sabermos onde o algoritmo começa ou termina.
2. Dentro do corpo do algoritmo, *linhas em branco* são usadas para dividir o algoritmo em partes logicamente relacionadas e tornar o algoritmo mais legível.
3. O algoritmo pode vir seguido de *comentários*. Comentários são linhas que não são executadas e cujo propósito é apenas o de facilitar o entendimento do algoritmo por parte de quem desejar lê-lo. As linhas de comentário são iniciadas com duas barras (“//”) e podem abrigar qualquer texto.

Como exemplo, considere um algoritmo para calcular e escrever a área de um círculo, dado o raio do círculo.

```
// algoritmo para calcular a área do círculo
algoritmo area_do_circulo
```

```
    // declaração de constantes e variáveis
    defina PI 3.14159
    defina MENSAGEM "O valor da área do círculo é: "
    real raio, area

    // leitura do raio do círculo
    leia raio

    // cálculo da área do círculo
    area ← PI * raio * raio

    // comunicação do resultado
    escreva MENSAGEM, area

fimalgoritmo
```

Bibliografia

O texto deste capítulo foi elaborado a partir dos livros abaixo relacionados:

1. Pothering, G.J., Naps, T.L. *Introduction to Data Structures and Algorithm Analysis with C++*. West Publishing Company, 1995.

2. Shackelford, R.L. *Introduction to Computing and Algorithms*. Addison-Wesley Longman, Inc, 1998.

Capítulo 4

Desenvolvimento de Algoritmos - Parte II

Neste tópico, você estudará detalhadamente dois componentes dos algoritmos: expressões condicionais e estruturas de controle. Como visto no capítulo anterior, as expressões condicionais possibilitam os algoritmos tomarem decisões que são governadas pelas estruturas de controle.

4.1 Expressões Condicionais

Denomina-se **expressão condicional** ou **expresão lógica** a expressão cujos operandos são relações, constantes e/ou variáveis do tipo lógico.

4.1.1 Relações

Uma **expressão relacional**, ou simplesmente **relação**, é uma comparação entre dois valores do mesmo tipo básico. Estes valores são representados na relação através de constantes, variáveis ou expressões aritméticas.

Os operadores relacionais, que indicam a comparação a ser realizada entre os termos da relação, são conhecidos da Matemática:

Operador	Descrição
$=$	igual a
$\neq$	diferente de
$>$	maior que
$<$	menor que
$\geq$	maior ou igual a
$\leq$	menor ou igual a

O resultado da avaliação de uma relação é sempre um *valor lógico*, isto é, V ou F.

Como exemplo, considere as variáveis a, b e c definidas a seguir:

inteiro a, b, c

Agora, suponha as seguintes atribuições:

$a \leftarrow 2$
$b \leftarrow 3$
$c \leftarrow 4$

Então, as expressões $a = 2$, $a > b + c$, $c \leq 5 - a$ e $b \neq 3$ valem V, F, F e F, respectivamente.

4.1.2 Operadores Lógicos

Uma **proposição** é qualquer sentença que possa ser avaliada como verdadeira ou falsa. Por exemplo, a sentença "a população de Campo Grande é de 500 mil habitantes" pode ser classificada como verdadeira ou falsa e, portanto, é uma proposição. Já a sentença "feche a porta!" não pode e, consequentemente, não é uma proposição. No nosso contexto, uma proposição é uma relação, uma variável e/ou uma constante do tipo lógico.

As expressões condicionais ou lógicas são formadas por uma ou mais proposições. Quando há mais de uma proposição em uma expressão lógica, elas estão relacionadas

através de um **operador lógico**. Os operadores lógicos utilizados como conectivos nas expressões lógicas são os seguintes:

Operador	**Descrição**
E	para a conjunção
OU	para a disjunção
NÃO	para a negação

Duas proposições podem ser combinadas pelo conectivo E para formar uma única proposição denominada **conjunção** das proposições originais. A conjunção das proposições p e q é representada por $p \wedge q$ e lemos "p e q". O resultado da conjunção de duas proposições é verdadeiro se e somente se ambas as proposições são verdadeiras, como mostrado na tabela a seguir.

p	q	$p \wedge q$
V	V	V
V	F	F
F	V	F
F	F	F

Duas proposições quaisquer podem ser combinadas pelo conectivo OU (com o sentido de e/ou) para formar uma nova proposição que é chamada **disjunção** das duas proposições originais. A disjunção de duas proposições p e q é representada por $p \vee q$ e lemos "p ou q". A disjunção de duas proposições é verdadeira se e somente se, pelo menos, uma delas for verdadeira, como mostrado na tabela a seguir.

p	q	$p \vee q$
V	V	V
V	F	V
F	V	V
F	F	F

Dada uma proposição p qualquer, uma outra proposição, chamada **negação** de p, pode ser formada escrevendo "É falso que" antes de p ou, se possível, inserindo a palavra "não" em p. Simbolicamente, designamos a negação de p por $\neg p$ e lemos "não p". Desta forma, podemos concluir que se p é verdadeira, então $\neg p$ é falsa; se p é falsa, então $\neg p$ é verdadeira, como mostrado na tabela a seguir.

p	$\neg p$
V	F
F	V

Agora, vejamos alguns exemplos de expressões lógicas que utilizam os conectivos vistos antes. Considere as variáveis a, b, c e x definidas a seguir:

inteiro a, b, c
lógico x

Agora, suponha as seguintes atribuições:

$a \leftarrow 2$
$b \leftarrow 3$
$c \leftarrow 4$
$x \leftarrow$ F

Então, as expressões $a = 2$ E $a > b + c$, $c \leq 5 - a$ OU $b \neq 3$, e NÃO x valem F, F e V, respectivamente.

Na primeira expressão, $a = 2$ E $a > b + c$, $a = 2$ e $a > b + c$ são relações. Em particular, $a > b + c$ contém uma expressão aritmética, $b + c$, que devemos resolver primeiro para daí podermos avaliar a relação $a > b + c$. De forma análoga, devemos primeiro resolver as relações $a = 2$ e $a > b + c$ para podermos resolver a expressão lógica $a = 2$ E $a > b + c$. Isto significa que estamos realizando as operações em uma certa ordem: em primeiro lugar, fazemos as operações aritméticas, depois as operações relacionais e, por último, as operações lógicas. A tabela a seguir ilustra a prioridade de todos os operadores vistos até aqui.

Operador	**Prioridade**
/, $*$, DIV, MOD	1 (máxima)
$+$, $-$	2
$=$, $\neq$, $\geq$, $\leq$, $>$, $<$	3
NÃO	4
E	5
OU	6 (mínima)

Observe que entre os operadores lógicos existe níveis distintos de prioridade, assim como entre os operadores aritméticos. Isto significa que na expressão $a = 2$ OU $a >$

$b + c$ E $c \leq 5 - a$, a operação lógica $a > b + c$ E $c \leq 5 - a$ é realizada primeiro e seu resultado é, então, combinado através do operador de disjunção (OU) com aquele da relação $a = 2$. Se quiséssemos mudar a ordem natural de avaliação, poderíamos escrever a expressão com o uso de parênteses: $(a = 2$ OU $a > b + c)$ E $c \leq 5 - a$.

4.2 Estruturas de Controle

Uma vez que a expressão condicional foi avaliada, isto é, reduzida para um valor V ou F, uma estrutura de controle é necessária para governar as ações que se sucederão. Daqui em diante, veremos dois tipos de estruturas de controle: *estrutura condicional* e *estrutura de repetição.*

4.2.1 Estruturas Condicionais

A estrutura condicional mais simples é a se - então - fimse. A forma geral desta estrutura é mostrada a seguir:

se <expressão condicional> então
    comando$_1$
    comando$_2$
    ⋮
    comando$_n$
fimse

Esta estrutura funciona da seguinte forma: se *expressão condicional* for verdadeira, os comandos $1, 2, \ldots, n$, contidos dentro da estrutura, são executados e, depois disso, o comando imediatamente depois da estrutura de controle é executado. Caso contrário, os comandos $1, 2, \ldots, n$ não são executados e o fluxo de execução do algoritmo segue com o comando imediatamente depois da estrutura de controle. Isto significa que esta forma de estrutura de controle permite que um algoritmo execute ou não um determinado número de comandos, dependendo de uma dada condição ser ou não satisfeita.

Como exemplo, considere um algoritmo para ler dois números, a e b, e escrevê-los em ordem não decrescente:

```
// algoritmo para escrever dois números em ordem não decrescente
algoritmo ordena_dois_números

    // declaração de variáveis
    inteiro a, b, temp

    // lê os números
    leia a, b

    // ordena os números
    se a > b então
        temp ← a
        a ← b
        b ← temp
    fimse

    // escreve os números ordenados
    escreva a, b

fimalgoritmo
```

Tudo que este algoritmo faz é verificar se o valor de a, fornecido na entrada, é maior do que o valor de b; se sim, ele troca os valores de a e b; caso contrário, ele não altera o valor destas variáveis. Como a operação de troca deve ser realizada apenas quando $a > b$, ela foi inserida dentro da estrutura condicional se - então - fimse.

Um outro ponto importante deste exemplo é a operação de troca. Esta operação é realizada com a ajuda de uma outra variável, denominada *temp*, e através de três atribuições. Isto não pode ser feito de outra forma! A fim de atribuir o valor de a a b e o valor de b a a, temos de utilizar uma outra variável, pois ao executarmos $a \leftarrow b$, atribuímos o valor de b a a e, consequentemente, perdemos o valor que, anteriormente, estava retido em a. Logo, para podermos fazer com que este valor possa ser atribuído a b, antes de executarmos $a \leftarrow b$, guardamos o valor de a em um valor seguro: a variável *temp*. Depois, atribuímos o valor em *temp* a b.

Uma variação da estrutura de controle se - então - fimse é a estrutura se - então - senão - fimse. Esta estrutura permite ao algoritmo executar uma de duas seqüências mutuamente exclusivas de comandos.

A forma geral deste comando é dada a seguir.

se <expressão condicional> então
    comando$_1$
    comando$_2$
    $\vdots$
    comando$_n$
senão
    comando$'_1$
    comando$'_2$
    $\vdots$
    comando$'_n$
fimse

Neste caso, se o resultado da expressão condicional for verdadeiro, os comandos $1, 2, \ldots, n$ são executados e depois disso o primeiro comando logo após o fim da estrutura de controle é executado; caso contrário, se o resultado da expressão condicional for falso, os comandos $1', 2', \ldots, n'$ são executados e depois disso o próximo comando a ser executado é aquele logo após o fim da estrutura de controle. Notemos que as seqüências de comandos correspondentes a *então* e *senão* são mutuamente exclusivas, isto é, ou os comandos $1, 2, \ldots, n$ são executados ou os comandos $1', 2', \ldots, n'$ são executados.

Como exemplo considere um algoritmo para escrever uma mensagem com o resultado de um exame, dada a nota obtida pelo candidato no exame:

```
// algoritmo para escrever resultado de um exame
// baseado na nota obtida pelo candidato
algoritmo resultado_exame

    // declaração de constantes e variáveis
    defina NOTAMINIMA 6.0
    real nota

    // lê a nota do candidato
    leia nota

    // compara a nota lida e escreve resultado
    se nota ≥ NOTAMINIMA então
        escreva "candidato aprovado"
```

```
        senão
            escreva "candidato reprovado"
        fimse

    fimalgoritmo
```

As estruturas condicionais podem estar "aninhadas", isto é, elas podem ocorrer umas dentro de outras. Como exemplo disto, considere o seguinte algoritmo que lê três números e escreve-os em ordem não decrescente:

```
// algoritmo para ordenar três números
algoritmo ordena_três_números

    // declaração de variáveis
    inteiro a, b, c, temp

    // lê os três números
    leia a, b, c

    // encontra o menor dos três números e guarda em a
    se (a > b) OU (a > c) então
        se (b ≤ c) então
            temp ← a
            a ← b
            b ← temp
        senão
            temp ← a
            a ← c
            c ← temp
        fimse
    fimse

    // encontra o valor intermediário e guarda em b
    se (b > c) então
        temp ← b
        b ← c
        c ← temp
    fimse

    // escreve os números em ordem não descrescente
    escreva a, b, c
fimalgoritmo
```

4.2.2 Estruturas de Repetição

A **estrutura de repetição**, ou simplesmente **laço**, permite que um grupo de comandos seja executado repetidamente um número determinado de vezes ou até que uma determinada condição se torne verdadeira ou falsa.

Nesta Subseção, estudaremos três estruturas de repetição:

- a estrutura para - faça
- a estrutura enquanto - faça
- a estrutura repita - até

A estrutura de repetição para - faça possui a seguinte forma geral

para <variável> de <valor inicial> até <valor final> faça
    comando_1
    comando_2
    $\vdots$
    comando_n
fimpara

e funciona como segue:

1. atribui à *variável*, que é o nome de uma variável numérica, o valor numérico *valor inicial*;
2. compara o valor de *variável* com o valor numérico *valor final*. Se ele for menor ou igual a *valor final*, vai para o passo 3. Caso contrário, executa o comando imediatamente após a estrutura de repetição;
3. executa os comandos 1 a n;
4. incrementa o valor de *variável* de uma unidade.
5. volta para o passo 2.

Como exemplo, considere o seguinte algoritmo para calcular e exibir a soma de todos os números pares desde 100 até 200, inclusive:

```
// algoritmo para somar os pares de 100 a 200
algoritmo soma_pares_100_a_200

    // declaração de variáveis
    inteiro soma, i

    // inicializa com 0 a variável que guardará a soma
    soma ← 0

    // calcula a soma
    para i de 100 até 200 faça
        se i MOD 2 = 0 então
            soma ← soma + i
        fimse
    fimpara

    // escreve o valor da soma
    escreva "A soma dos pares de 100 a 200 é: ", soma

fimalgoritmo
```

Neste algoritmo, o laço para - faça inicia com a atribuição do valor 100 à variável i. Daí, compara o valor de i com 200 e, como $i < 200$, executa pela primeira vez os comandos dentro do laço e, ao final da execução, incrementa i de uma unidade. Deste ponto em diante, o valor de i é comparado com 200 e, caso seja menor ou igual a 200, uma nova iteração do laço é realizada e i é novamente incrementada de uma unidade ao final da iteração. Este processo se repete até que i seja igual a 201. Desta forma, i guarda um valor de 100 a 200 a cada repetição do laço. A cada iteração, o algoritmo verifica se o valor de i é par ou ímpar. Se o valor de i for par, o valor de i é adicionado ao valor na variável soma.

Observe que a variável *soma* recebe o valor 0 antes do laço ser executado. Esta atribuição é denominada **inicialização** da variável. Como você pode perceber, *soma* é utilizada como um *acumulador*, isto é, a cada iteração do laço, *soma* é incrementada de mais um número par e isto é feito somando o valor atual de *soma* ao número par em i e atribuindo o resultado novamente à variável *soma*: $soma \leftarrow soma + i$. Entretanto, se a inicialização não estivesse presente no algoritmo, quando o laço fosse executado pela primeira vez, não haveria um valor em *soma* e a atribuição $soma \leftarrow soma + i$

não faria sentido[1].

Uma variação da estrutura de repetição para - faça é aquela que nos possibilita controlar o valor do incremento da variável contadora do laço:

para <variável> de <valor inicial> até <valor final> passo <incremento> faça
    $comando_1$
    $comando_2$
    ⋮
    $comando_n$
fimpara

onde *incremento* é o valor adicionado ao valor de *variável* ao final de cada iteração. Quando a parte do comando passo <incremento> não está presente, o valor assumido para o incremento é 1 (um).

Esta variação da estrutura de repetição para - faça nos permite, por exemplo, resolver o problema de calcular a soma de todos os pares de 100 a 200 de uma forma mais elegante do que aquela vista anteriormente, como podemos constatar a seguir:

```
// algoritmo para somar os pares de 100 a 200
algoritmo soma_pares_100_a_200

    // declaração de variáveis
    inteiro soma, i

    // inicializa com 0 a variável que guardará a soma
    soma ← 0

    // calcula a soma
    para i de 100 até 200 passo 2 faça
            soma ← soma + i
    fimpara

    // escreve o valor da soma
    escreva "A soma dos pares de 100 a 200 é: ", soma

fimalgoritmo
```

[1]Se você executasse este algoritmo em computador sem a inicialização de *soma*, o resultado seria imprevisível, pois não poderíamos prever o valor que está na posição de memória do computador correspondente a *soma*.

A estrutura de repetição para - faça deve ser utilizada *apenas* quando queremos repetir a execução de um ou mais comandos um número conhecido de vezes, como no exemplo anterior. Entretanto, há problemas em que não é possível determinar, previamente, o número de repetições. Neste caso, devemos utilizar a estrutura enquanto - faça ou repita - até.

A estrutura enquanto - faça possui a seguinte forma geral

enquanto <expressão condicional> faça
    comando_1
    comando_2
    $\vdots$
    comando_n
fimenquanto

A lógica do funcionamento desta estrutura de repetição é a seguinte:

1. *expressão condicional* é avaliada. Se o resultado for verdadeiro, então o fluxo do algoritmo segue para o passo seguinte. Caso contrário, o comando imediatamente posterior à estrutura de repetição é executado;
2. executa os comandos de 1 a n;
3. volta para o passo 1.

Como exemplo, considere o seguinte algoritmo para ler um número inteiro n, que não contém dígito 0, e escrever um número inteiro m que corresponde a n invertido:

```
// algoritmo para inverter um nḿero inteiro sem dígito 0
algoritmo inverte_número

    // declaração de variáveis
    inteiro n, r, m

    // lê o valor de um inteiro
    leia n

    // inicializa a variável que conterá o inteiro invertido
    m ← 0

    // encontra o número invertido
    enquanto n > 0 faça
        r ← n MOD 10
        m ← m * 10 + r
        n ← n DIV 10
    fimenquanto

    // exibe o número invertido
    escreva m

fimalgoritmo
```

Observe que, neste algoritmo, o laço é repetido uma quantidade de vezes igual ao número de dígitos de n. Como n é desconhecido até a execução do algoritmo iniciar, não somos capazes de dizer, ao escrevermos o algoritmo, quantos dígitos n terá, logo não saberemos prever o número de repetições do laço. Entretanto, podemos determinar o número de dígitos de n ao longo da execução do algoritmo e isto é feito indiretamente pelo algoritmo ao dividir n, sucessivamente, por 10, o que fará n "perder" o dígito menos significativo a cada repetição do laço e se tornar 0 em algum momento. Neste ponto, podemos encerrar o laço.

A estrutura de repetição repita - até é semelhante à estrutura enquanto - faça, pois ambas são utilizadas quando não conhecemos, antecipadamente, o número de repetições. A diferença entre elas reside no fato que a seqüência de instruções da estrutura repita - até é executada pelo menos uma vez, independentemente da expressão condicional ser ou não verdadeira.

A estrutura repita - até tem a seguinte forma geral:

repita
    comando$_1$
    comando$_2$
    ⋮
    comando$_n$
até <expressão condicional>

A lógica do funcionamento desta estrutura de repetição é a seguinte:

1. executa a seqüência de instruções;
2. *expressão condicional* é avaliada. Se ela for falsa, então o fluxo de execução volta para o passo 1. Caso contrário, o comando imediatamente posterior à estrutura de repetição é executada.

Como exemplo, considere o seguinte trecho algorítmico:

⋮
repita
    escreva "entre com um número positivo: "
    leia n
até $n > 0$
⋮

Neste trecho de algoritmo, o objetivo do laço é garantir que a variável n receba um número positivo; enquanto o usuário entrar com um número menor ou igual a zero, o algoritmo continuará solicitando a entrada.

4.3 Problemas e Soluções

Nesta Seção, veremos quatro problemas e suas respectivas soluções. Associado a cada solução está um breve comentário.

Os problemas e suas respectivas soluções são os seguintes:

1. Uma pessoa aplicou seu capital a juros e deseja saber, trimestralmente, a posição de seu investimento inicial c. Chamando de i a taxa de juros do trimestre, escrever uma tabela que forneça, para cada trimestre, o rendimento auferido e o saldo acumulado durante o período de x anos, supondo que nenhuma retirada tenha sido feita.

```
// algoritmo para calcular rendimento e montante de aplicação
// trimestral
algoritmo calcula_rendimento

    // declaração de variáveis
    inteiro x, n
    inteiro c, i, r, j

    // lê o capital inicial, a taxa de juros e números de anos
    leia c, i, x

    // calcula o número de trimestres em x anos
    n ← x * 4

    // calcula e exibe rendimento e montante
    para j de 1 até n faça
        r ← i * c
        c ← c + r
        escreva "rendimento do trimestre ", j, ": ", r
        escreva "montante do trimestre ", j, ": ", c
    fimpara

fimalgoritmo
```

O algoritmo inicia com o cálculo do número de trimestres em x anos, já que o rendimento deve ser calculado por trimestre e a taxa de juros i também é dada em função do número de trimestres. O laço para - faça é repetido n vezes, onde n é o número de trimestres encontrado. A cada repetição do laço, o rendimento r é calculado e o capital c é somado a r para totalizar o montante do mês. Em seguida, os valores de r e c são exibidos para o usuário.

2. Em um frigorífico existem 90 bois. Cada boi traz em seu pescoço um cartão contendo seu número de identificação e seu peso. Faça um algoritmo que encontre e escreva o número e o peso do boi mais gordo e do boi mais magro.

```
// algoritmo para encontrar o número e o peso do boi mais gordo e
// do boi mais magro de um conjunto de 90 bois
algoritmo encontra_número_peso

    // declaração de variáveis
    inteiro num, boigordo, boimagro,
    real peso, maiorpeso, menorpeso

    // lê os dados do primeiro boi
    escreva "entre com o número de identificação do primeiro boi: "
    leia num
    escreva "entre com o peso do primeiro boi: "
    leia peso

    // inicializa as variáveis que conterão o resultado
    boigordo ← num
    boimagro ← num
    maiorpeso ← peso
    menorpeso ← peso

    // compara os pesos para encontrar o boi mais gordo e o boi mais magro
    para j de 2 até 90 faça
        escreva "entre com o número do próximo boi: "
        leia num
        escreva "entre com o peso do próximo boi: "
        leia peso

        // compara o peso lido com o maior peso até então
        se peso > maiorpeso então
            maiorpeso ← peso
            boigordo ← num
        senão
            se peso < menorpeso então
                menorpeso ← peso
                boimagro ← num
            fimse
        fimse
```

```
    fimpara

    // escreve o número e o peso do boi mais gordo e do boi mais magro
    escreva "o número do boi mais gordo é: ", boigordo
    escreva "o peso do boi mais gordo é: ", maiorpeso
    escreva "o número do boi mais magro é: ", boimagro
    escreva "o peso do boi mais magro é: ", menorpeso
fimalgoritmo
```

Neste algoritmo, a idéia é ler e processar as informações de cada boi simultaneamente, pois não necessitamos guardar as informações dos 90 bois para depois processá-las. Esta leitura é realizada através de um laço para - faça, pois sabemos exatamente quantos bois existem: 90. Para guardar o número do boi mais gordo, o número do boi mais magro, o peso do boi mais gordo e o peso do boi mais magro, usamos quatro variáveis, uma para cada dado.

Para encontrar o número e peso do boi mais gordo e o número e peso do boi mais magro, comparamos o peso do boi que acabamos de ler com os até então maior e menor pesos. O problema é que para realizarmos esta operação pela primeira vez, as variáveis que armazenam o maior e o menor peso precisam possuir algum valor. Para tal, lemos as informações sobre o primeiro boi separadamente e inicializamos ambas as variáveis com o peso do primeiro boi. Daí em diante, podemos prosseguir como imaginamos: comparando tais variáveis com os valores de cada boi, um a um.

3. Uma pesquisa sobre algumas características físicas da população de uma determinada região coletou os seguintes dados, referentes a cada habitante, para serem analisados:

   - idade em anos
   - sexo (masculino, feminino)
   - cor dos olhos (azuis, verdes, castanhos)
   - cor dos cabelos (louros, castanhos, pretos)

   Para cada habitante são informados os quatro dados acima. A fim de indicar o final da entrada, após a seqüência de dados dos habitantes, o usuário entrará com o valor -1 para a idade, o que deve ser interpretado pelo algoritmo como fim de entrada.

```
// algoritmo para encontrar a maior idade de um conjunto de indivíduos
// e o percentual de indivíduos do sexo feminino com idade entre 18 e
// 35 anos, inclusive, e olhos verdes e cabelos louros
```

```
algoritmo encontra_maior_idade_e_percentual

    // declaração de variáveis
    inteiro idade, maioridade, habitantes, totalhabitantes
     real porcentagem
    cadeia sexo, olhos, cabelos

    // inicializa algumas variáveis
    maioridade ← −1
    habitantes ← 0
    totalhabitantes ← 0

    // lê a idade do primeiro habitante
    escreva "entre com a idade do habitante ou -1 para encerrar: "
    leia idade

    // calcula os resultados
    enquanto idade ≠ −1 faça
        escreva "entre com o sexo do habitante: "
        leia sexo

        escreva "entre com a cor dos olhos do habitante: "
        leia olhos

        escreva "entre com a cor dos cabelos do habitante: "
        leia cabelos

        // compara a idade lida com a maior idade até então
        se idade > maioridade então
            maioridade ← idade
        fimse

        // conta o número total de habitantes
        totalhabitantes ← totalhabitantes + 1

        // conta o número total de habitantes que satisfazem as restrições
        // (sexo feminino, idade entre 18 e 35 anos, olhos verdes e cabelos louros)
        se   idade ≥ 18 E (idade ≤ 35) E (sexo ="feminino") E
            (olhos ="verdes") E (cabelos ="louros") então
            habitantes ← habitantes + 1
        fimse
```

```
        // lê a idade do próximo habitante
        escreva "entre com a idade do habitante ou -1 para encerrar: "
        leia idade
    fimenquanto

    // escreve a maior idade e a porcentagem pedida
    se totalhabitantes > 0 então
        porcentagem ← (habitantes/totalhabitantes) * 100
        escreva "a maior idade é: ", maioridade
        escreva "a porcentagem é: ", porcentagem
    fimse
fimalgoritmo
```

Este algoritmo difere do anterior em dois aspectos importantes: a entrada de dados e a forma pela qual encontramos a maior idade. Desta vez, não sabemos quantos habitantes serão processados, portanto não podemos realizar a leitura através de um laço para - faça. Entretanto, o problema diz que a entrada encerra quando for digitado o número -1 ao invés de uma idade, o que nos possibilita utilizar um laço do tipo enquanto - faça para ler a entrada.

Quanto à idade, observe que, ao invés de lermos o primeiro habitante separadamente para inicializarmos o valor de *maioridade* com a idade do primeiro habitante, usamos um número negativo. Neste caso, isso é possível, pois não existe um valor de idade negativo e, além disso, quando *maioridade* for comparada com a idade do primeiro habitante, ela sempre vai receber este valor, pois ele sempre será maior do que o valor negativo de *maioridade*.

Finalmente, o algoritmo calcula a porcentagem pedida de habitantes e escreve os resultados dentro de uma estrutura condicional se - então - fimse. Isto é necessário para evitar que o cálculo de *porcentagem* seja realizado com o valor de *totalhabitantes* igual a 0, o que seria um comando impossível de ser executado.

4. Escrever um algoritmo para fazer uma tabela de seno de A, com A variando de 0 a 1.6 radiano de décimo em décimo de radiano, usando a série

$$\text{sen } A = A - \frac{A^3}{3!} + \frac{A^5}{5!} + \cdots$$

com erro inferior a 0.0001. Escrever também o número de termos utilizado.

```
// algoritmo para calcular o seno de um número por aproximação
algoritmo calcula_seno

    // declaração de variáveis
    real a, sena, t
    inteiro n

    // gera todos os valores de a para cálculo do seno
    para a de 0 até 1.6 passo 0.1 faça
        // cálculo do seno de a
        sena ← 0
        t ← a
        n ← 0
        repita
            sena ← sena + t
            n ← n + 1
            t ← −t * (a * a)/(2 * n * (2 * n + 1))
        até (t ≤ 0.0001) E (−t ≤ 0.0001)
    fimpara

    // escreve o seno obtido
    escreva "o valor de a é: ", a
    escreva "o valor do seno de a é: ", sena
    escreva "o número de termos usados no cálculo foi: ", n

fimalgoritmo
```

Neste algoritmo, temos um "aninhamento" de laços. O laço mais externo tem a finalidade de gerar todos os números de 0 a 1.6, com incremento de 0.1, dos quais desejamos calcular o seno. O laço mais interno, por sua vez, tem a finalidade de calcular o seno do valor atual de a gerado pelo laço mais externo. A cada repetição do laço mais interno, um repita - até, um termo da série é calculado e adicionado à variável *sena*.

Um termo da série é sempre calculado a partir do anterior, multiplicando-se por este o que falta para se obter aquele. O valor do termo é armazenado em t. Observe que se t é o n-ésimo termo da série, então o $(n+1)$-ésimo termo pode ser obtido da seguinte forma:

$$t = t \times \frac{a^2}{2 \times n \times (2 \times n + 1)}.$$

Por exemplo,

$$\frac{a^5}{5!} = -\frac{a^3}{3!} \times \frac{a^2}{2 \times 2 \times (2 \times 2 + 1)}.$$

A expressão condicional "$(t \leq 0.0001)$ E $(-t \leq 0.0001)$" garante que o laço repita - até encerra apenas quando o próximo termo t a ser adicionado a *sena* for inferior, em valor absoluto, a 0.0001, indicando que o valor até então em *sena* possui uma precisão de 0.0001.

Bibliografia

O texto deste capítulo foi elaborado a partir dos livros abaixo relacionados:

1. Farrer, H. et. al. *Algoritmos Estruturados*. Editora Guanabara, 1989.

2. Shackelford, R.L. *Introduction to Computing and Algorithms*. Addison-Wesley Longman, Inc, 1998.

Lista de Exercícios

1. Dados três números naturais, verificar se eles formam os lados de um triângulo retângulo.

2. Dados três números inteiros, imprimi-los em ordem crescente.

3. Dada uma coleção de números naturais terminada por 0, imprimir seus quadrados.

4. Dado n, calcular a soma dos n primeiros números naturais.

5. Dado n, imprimir os n primeiros naturais ímpares.

   Exemplo:

   Para $n = 4$ a saída deverá ser $1, 3, 5, 7$.

6. Durante os 31 dias do mês de março foram tomadas as temperaturas médias diárias de Campo Grande, MS. Determinar o número de dias desse mês com temperaturas abaixo de zero.

7. Dados x inteiro e n um natural, calcular x^n.

8. Uma loja de discos anota diariamente durante o mês de abril a quantidade de discos vendidos. Determinar em que dia desse mês ocorreu a maior venda e qual foi a quantidade de discos vendida nesse dia.

9. Dados o número n de alunos de uma turma de Programação de Computadores I e suas notas de primeira prova, determinar a maior e a menor nota obtidas por essa turma, onde a nota mínima é 0 e a nota máxima é 100.

10. Dados n e uma seqüência de n números inteiros, determinar a soma dos números pares.

11. Dado n natural, determinar $n!$.

12. Dados n e dois números naturais não nulos i e j, imprimir em ordem crescente os n primeiros naturais que são múltiplos de i ou de j ou de ambos.

    Exemplo:

    Para $n = 6, i = 2$ e $j = 3$ a saída deverá ser $0, 2, 3, 4, 6, 8$.

13. Dizemos que um número natural é **triangular** se é produto de três números naturais consecutivos.

    Exemplo:

    120 é triangular, pois $4 \cdot 5 \cdot 6 = 120$.

    Dado n natural, verificar se n é triangular.

14. Dado p inteiro, verificar se p é primo.

15. Uma pessoa aplicou um capital de x reais a juros mensais de y durante 1 ano. Determinar o montante de cada mês durante este período.

16. Dado um natural n, determine o número harmônico H_n definido por

$$H_n = \sum_{k=1}^{n} \frac{1}{k}.$$

17. Os pontos (x, y) que pertencem à figura H (veja a figura 4.1) são tais que $x \geq 0, y \geq 0$ e $x^2 + y^2 \leq 1$. Dados n pontos reais (x, y), verifique se cada ponto pertence ou não a H.

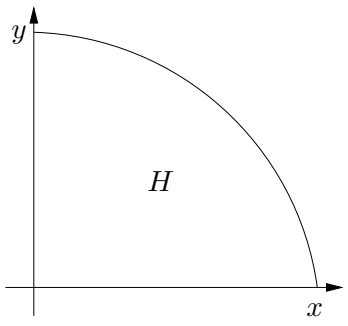

Figura 4.1: Área H de um quarto de um círculo. Veja o exercício 17.

18. Considere o conjunto $H = H_1 \cup H_2$ de pontos reais, onde

$$H_1 = \{(x, y) | x \leq 0, y \leq 0, y + x^2 + 2x - 3 \leq 0\}$$
$$H_2 = \{(x, y) | x \geq 0, y + x^2 - 2x - 3 \leq 0\}$$

Faça um algoritmo que lê uma seqüência de n pontos reais (x, y) e verifica se cada ponto pertence ou não ao conjunto H. O algoritmo deve também contar o número de pontos da seqüência que pertencem a H.

19. Dados números reais a, b e c, calcular as raízes de uma equação do segundo grau da forma $ax^2 + bx + c = 0$. Imprimir a solução em uma das seguintes formas:

a.	DUPLA	b.	REAIS DISTINTAS	c.	COMPLEXAS
	raiz		raiz 1		parte real
			raiz 2		parte imaginária

20. Para n alunos de uma determinada classe são dadas as 3 notas das provas. Calcular a média aritmética das provas de cada aluno, a média da classe, o número de aprovados e o número de reprovados, onde o critério de aprovação é média $\geq$ 5,0.

21. Dadas as populações de Caarapó (MS) e Rio Negro (MS) e sabendo que a população de Caarapó tem um crescimento anual de x e a população de Rio Negro tem um crescimento anual de y determine:

    (a) se a população da cidade menor ultrapassa a da maior;

    (b) quantos anos passarão antes que isso aconteça.

22. Dados dois números inteiros positivos, determinar o máximo divisor comum entre eles utilizando o algoritmo de Euclides.

    Exemplo:

	1	1	1	2
24	15	9	6	3
9	6	3	0	

    = mdc(24,15)

23. Dado n inteiro positivo, dizemos que n é **perfeito** se for igual à soma de seus divisores positivos diferentes de n.

    Exemplo:

    28 é perfeito, pois $1 + 2 + 4 + 7 + 14 = 28$

    Verificar se um dado inteiro positivo é perfeito.

24. Leonardo Fibonacci[2] conseguiu modelar o ritmo de crescimento da população de coelhos através de uma seqüência de números naturais que passou a ser conhecida como **seqüência de Fibonacci**. O n-ésimo número da seqüência de Fibonacci F_n é dado pela seguinte fórmula de recorrência:

$$\begin{cases} F_1 &= 1 \\ F_2 &= 1 \\ F_i &= F_{i-1} + F_{i-2} \text{ para } i \geq 3. \end{cases}$$

Faça um algoritmo que dado n calcula F_n.

25. Dizemos que um número i é **congruente módulo** m a j se $i \bmod m = j \bmod m$.

Exemplo:

35 é congruente módulo 4 a 39, pois $35 \bmod 4 = 3 = 39 \bmod 4$.

Dados n, j e m naturais não nulos, imprimir os n primeiros naturais congruentes a j módulo m.

26. Dado um número natural na base binária, transformá-lo para a base decimal.

Exemplo:

Dado 10010 a saída será 18, pois $1 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 = 18$.

27. Dado um número natural na base decimal, transformá-lo para a base binária.

Exemplo:

Dado 18 a saída deverá ser 10010.

28. Dado um número inteiro positivo n que não contém um dígito 0, imprimi-lo na ordem inversa de seus dígitos.

Exemplo:

Dado 26578 a saída deverá ser 87562.

29. Qualquer número natural de quatro algarismos pode ser dividido em duas dezenas formadas pelos seus dois primeiros e dois últimos dígitos.

Exemplos:

[2]Matemático italiano do século XII, conhecido pela descoberta dos números de Fibonacci e pelo seu papel na introdução do sistema decimal arábico para escrita e manipulação de números na Europa. O nome do matemático era Leonardo Fibonacci ($1170^{\star}$–$1250^{\dagger}$). Seu pai, Guglielmo, tinha o apelido de Bonacci, que quer dizer "pessoa simples". Leornardo foi postumamente conhecido como Fibonacci (*filius Bonacci*). Naquela época, uma pessoa ilustre tinha seu sobrenome associado ao lugar a que pertencia. Por isso, este matemático é mais conhecido como Leonardo de Pisa ou Leonardo Pisano.

1297 : 12 e 97.
5314 : 53 e 14.

Escreva um algoritmo que imprima todos os números de quatro algarismos cuja raiz quadrada seja a soma das dezenas formadas pela divisão acima.

Exemplo:

$\sqrt{9801} = 99 = 98 + 01$.
Portanto, 9801 é um dos números a ser impresso.

30. Dados n e uma seqüência de n números inteiros, determinar quantos segmentos de números iguais consecutivos compõem essa seqüência.

    Exemplo:

    A seqüência $\overbrace{5}, \overbrace{2,2}, \overbrace{4,4,4,4}, \overbrace{1,1}$ é formada por 5 segmentos de números iguais.

31. Dados um inteiro positivo n e uma seqüência de n números inteiros, determinar o comprimento de um segmento crescente de comprimento máximo.

    Exemplos:

    Na seqüência $5, 10, 3, \overbrace{2, 4, 7, 9}, 8, 5$ o comprimento do segmento crescente máximo é 4.
    Na seqüência $10, 8, 7, 5, 2$ o comprimento de um segmento crescente máximo é 1.

32. Dizemos que um número natural n com pelo menos 2 algarismos é **palíndrome** se

    o primeiro algarismo de n é igual ao seu último algarismo;
    o segundo algarismo de n é igual ao se penúltimo algarismo;
    e assim sucessivamente.

    Exemplos:

    567765 é palíndrome;
    32423 é palíndrome;
    567675 não é palíndrome.

    Dado um número natural n, $n \geq 10$, verificar se n é palíndrome.

33. Dado um inteiro positivo n, calcular e imprimir o valor da seguinte soma

$$\frac{1}{n} + \frac{2}{n-1} + \frac{3}{n-2} + \ldots + \frac{n}{1}$$

pelas seguintes maneiras:

(a) fórmula de recorrência;

(b) fórmula do termo geral.

34. Faça um algoritmo que calcula a soma

$$1 - \frac{1}{2} + \frac{1}{3} - \frac{1}{4} + \ldots + \frac{1}{9999} - \frac{1}{10000}$$

pelas seguintes maneiras:

(a) adição dos termos da direita para a esquerda;

(b) adição dos termos da esquerda para a direita;

(c) adição separada dos termos positivos e dos termos negativos da esquerda para a direita;

(d) adição separada dos termos positivos e dos termos negativos da direita para a esquerda;

(e) fórmula de recorrência;

(f) fórmula do termo geral.

35. Uma maneira de calcular o valor do número π é utilizar a seguinte série:

$$\pi = 4 - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} + \frac{4}{9} - \frac{4}{11} + \ldots$$

Fazer um algoritmo para calcular e imprimir o valor de π através da série acima, com precisão de 4 casas decimais. Para obter a precisão desejada, adicionar apenas os termos cujo valor absoluto seja maior ou igual a 0,0001.

36. Dados x real e n natural, calcular uma aproximação para $\cos x$ através dos n primeiros termos da seguinte série:

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

Compare com os resultados de sua calculadora.

37. Dados x e ε reais, $\varepsilon > 0$, calcular uma aproximação para sen x através da seguinte série infinita

$$\sin x = \frac{x}{1!} - \frac{x^3}{3!} + \frac{x^5}{5!} - \ldots + (-1)^k \frac{x^{2k+1}}{(2k+1)!} + \ldots$$

incluindo todos os termos $T_i = (-1)^k \frac{x^{2k+1}}{(2k+1)!}$, até que $T_i < \varepsilon$. Compare com os resultados de sua calculadora.

38. Dadas n seqüências de números inteiros, cada qual terminada por 0, calcular a soma dos números pares de cada seqüência.

39. Dado um número inteiro positivo n, determinar todos os inteiros entre 1 e n que são comprimento de hipotenusa de um triângulo retângulo com catetos inteiros.

40. Dados dois naturais m e n, determinar, entre todos os pares de números naturais (x, y) tais que $x \leq m$ e $y \leq n$, um par para o qual o valor da expressão $xy - x^2 + y$ seja máximo e calcular também esse máximo.

41. Dados n números inteiros positivos, calcular a soma dos que são primos.

42. Sabe-se que um número da forma n^3 é igual à soma de n números ímpares consecutivos.

Exemplo:

$$\begin{aligned} 1^3 &= 1 \\ 2^3 &= 3 + 5 \\ 3^3 &= 7 + 9 + 11 \\ 4^3 &= 13 + 15 + 17 + 19 \\ &\vdots \end{aligned}$$

Dado m, determine os ímpares consecutivos cuja soma é igual a n^3 para n assumindo valores de 1 a m.

43. Dado um número inteiro positivo, determine a sua decomposição em fatores primos, calculando também a multiplicidade de cada fator.

44. Dados n inteiros positivos, determinar o máximo divisor comum entre eles.

Capítulo 5

Estruturas de Dados

Como visto no Capítulo 3, tipos de dados elementares são caracterizados pelo fato que seus valores são atômicos, isto é, não admitem decomposição. Os tipos de dados numérico, caracter e lógico estão entre os tipos de dados elementares suportados pela maioria das linguagens de programação. Entretanto, se os valores de um tipo de dados admitem decomposição em valores mais simples, então o tipo de dados é dito complexo ou estruturado, ao invés de elementar, e a organização de cada componente e as relações entre eles constitui o que chamamos de *estrutura de dados*.

Neste Capítulo, estudaremos as estruturas de dados mais elementares, tais como vetores, matrizes e registros, mas que nos permitirão resolver problemas mais complexos do que aqueles vistos até agora.

5.1 Vetores

Considere o seguinte problema:

> Calcule a média aritmética das notas de 5 alunos de uma disciplina e determine o número de alunos que tiveram nota superior à média calculada.

Como você bem sabe, o cálculo da média aritmética das notas de 5 alunos de uma disciplina pode ser resolvido através de uma estrutura de repetição como a que segue:

```
⋮
soma ← 0
para i de 1 até 5 faça
    leia nota
    soma ← soma + nota
fimpara
media ← soma/5
⋮
```

Entretanto, se seguirmos com este trecho de algoritmo, como determinaremos quantos alunos obtiveram nota superior à média calculada? Isto porque não temos as notas de cada um dos 5 alunos depois que o trecho acima for executado. Logo, devemos optar por outro caminho, que pode ser este que segue abaixo:

```
// algoritmo para calcular a média de 5 notas e escrever
// quantos alunos obtiveram nota superior à média
algoritmo calcula_média_e_notas_superiores

    // declaração de constantes e variáveis
    inteiro soma, media, nota1, nota2, nota3
    inteiro nota4, nota5, num

    // leitura das notas
    leia nota1, nota2, nota3, nota4, nota5

    // cálculo a soma das notas
    soma ← nota1 + nota2 + nota3 + nota4 + nota5

    // cálculo da média
    media ← soma/5

    // cálculo das notas superiores à média
    num ← 0
    se nota1 > media então
        num ← num + 1
    fimse

    se nota2 > media então
        num ← num + 1
    fimse
```

```
se nota3 > media então
    num ← num + 1
fimse

se nota4 > media então
    num ← num + 1
fimse

se nota5 > media então
    num ← num + 1
fimse

// escreve o número de alunos com nota superior à média
escreve "o número de alunos com nota superior à média é: ", num

fimalgoritmo
```

Como podemos constatar, não fomos capazes de utilizar uma estrutura de repetição para calcular quantas notas eram superiores à média, mesmo embora houvesse 5 estruturas condicionais idênticas, a menos da variável com o valor da nota, para realizar o cálculo desejado. No caso do problema acima, esta redundância não chega a ser um "fardo", mas se tivéssemos 100, 1000, ou mesmo 1000000 de notas, esta solução seria inviável, uma vez que teríamos de escrever, respectivamente, 100, 1000 ou 1000000 de estruturas condicionais semelhantes, uma para cada nota. Felizmente, para problemas como este, temos uma forma eficaz de solução, que utiliza uma estrutura de dados denominada *vetor*.

5.1.1 Definindo uma Estrutura de Dados Vetor

A estrutura de dados **vetor** é uma estrutura de dados linear utilizada para armazenar uma lista de valores do mesmo tipo. Um dado vetor é definido como tendo algum *número fixo* de *células* idênticas. Cada célula armazena um, e somente um, dos valores de dados do vetor. Cada uma das células de um vetor possui seu próprio endereço, ou *índice*, através do qual podemos referenciá-la. Por exemplo, a primeira célula de um vetor possui índice 1, a quarta possui índice 4, e assim por diante.

Em geral, ao definirmos estruturas de dados complexas devemos especificar a estrutura de dados de um novo tipo de dados pois este tipo, ao contrário dos tipos primitivos, não está "pronto para uso" e, portanto, deve ser definido explicitamente

dentro do algoritmo. Temos também que a definição de uma estrutura de dados é parte da especificação de um tipo de dados complexo, que, como sabemos, também consiste na especificação das operações sobre o conjunto de valores do tipo. Entretanto, no momento, veremos a definição de tipos complexos como composta apenas da especificação da estrutura de dados.

No caso particular de estruturas de dados compostas homogêneas, como é o caso de vetores, podemos de fato especificar a estrutura de dados do novo tipo de dados, o tipo de dados vetor, e a partir do tipo definido declararmos as variáveis deste tipo ou declararmos diretamente a variável do tipo vetor.

Nós podemos definir um vetor através da especificação de um identificador para um *tipo* vetor, suas dimensões e o tipo de dados dos valores que ele pode armazenar. Isto é feito da seguinte forma:

definatipo vetor[l_i..l_s] de <tipo dos elementos> <nome do tipo>

onde

- *definatipo* e *vetor* são palavras-chave;
- *tipo dos elementos* é o nome do tipo dos elementos do vetor;
- *nome do tipo* é um identificador para o tipo sendo definido;
- l_i e l_s são respectivamente os limites inferior e superior do vetor.

Por exemplo, para criarmos um vetor de 5 elementos inteiros chamado *vetor_notas*, fazemos:

definatipo vetor[1..5] de inteiro *vetor_notas*

O número de elementos (células) de um vetor é dado por $l_s - l_i + 1$ e o índice do primeiro elemento (célula) é l_i, do segundo é $l_i + 1$, e assim por diante. Isto significa que dois vetores a e b com valores l_i e l_s sejam 1 e 5 e 7 e 11, respectivamente, possuem o mesmo número de elementos: $5 = 5 - 1 + 1 = 11 - 7 + 1$; entretanto, o primeiro elemento de a possui índice 1, enquanto o primeiro elemento de b possui índice 7.

5.1.2 Declarando e Manipulando Variáveis do Tipo Vetor

Há duas formas de declararmos variáveis tipo vetor: uma usando a definição de tipo visto acima e a outra e a declaração de forma direta que veremos abaixo. Para declararmos uma variável de um tipo vetor definido previamente, procedemos exatamente da mesma forma que para um tipo simples. Por exemplo, considere a criação de uma variável denominada *notas* do tipo *vetor_notas*:

$$vetor\_notas\ notas$$

Esta declaração significa que estamos criando uma variável *notas* que é do tipo *vetor_notas*, ou seja, um vetor de inteiros com 5 posições de índices 1 a 5.

Esta é uma forma bastante útil de declararmos vetores, principalmente quando tivermos mais de uma variável deste mesmo tipo. Quando tivermos apenas uma variável tipo vetor de um tamanho específico podemos declarar a variável diretamente com o mesmo efeito das etapas anteriores. Para isto usamos a seguinte sintaxe:

$$\underline{\text{vetor}}[l_i..l_s]\ \underline{\text{de}}\ \text{<tipo dos elementos> <nome da variável>}$$

No exemplo acima a variável *notas* poderia ser declarada alternativamente da seguinte forma

$$\underline{\text{vetor}}[1..5]\ \underline{\text{de}}\ \underline{\text{inteiro}}\ notas$$

Uma vez declarada a variável *notas*, podemos atribuir qualquer conjunto de 5 valores numéricos à variável. Entretanto, isto **não** é feito da forma mais natural, tal como

$$notas \leftarrow (-1, 0, 1, 33, -10)$$

mas sim individualmente; ou seja, um valor por vez a cada célula do vetor. Para tal, usamos o nome da variável, o índice da célula e uma sintaxe própria. Por exemplo, $notas[0] \leftarrow -1$, atribui o valor -1 à célula de índice 1 do vetor.

De forma geral, as células, e não a variável do tipo vetor como um todo, é que são utilizadas em qualquer operação envolvendo a variável, sejam elas leitura, escrita, atribuição, expressão, e assim por diante. No caso particular do vetor *notas*, a célula

de índice i de $notas$, $notas[i]$, pode ser usada em qualquer parte de um algoritmo em que uma variável inteira possa ser usada. Isto significa que podemos ter comandos tais como

$$\underline{\text{leia}}\ notas[1],\ \underline{\text{escreva}}\ notas[1] \text{ e } notas[2] \leftarrow notas[1] + 1.$$

5.1.3 Problemas e Soluções Envolvendo Vetores

Vamos iniciar esta subseção resolvendo de forma eficaz o problema estabelecido no início deste Capítulo:

1. Calcule a média aritmética das notas de 5 alunos de uma disciplina e determine o número de alunos que tiveram nota superior à média calculada.

```
// algoritmo para calcular a média de 5 notas e escrever
// quantos alunos obtiveram nota superior à média
algoritmo calcula_média_e_notas_superiores

    // declaração de constantes
    defina LI 1
    defina LS 5

    // declaração de tipos
    definatipo vetor[LI..LS] de real vetor_notas

    // declaração de variáveis
    vetor_notas notas
    real soma, media
    inteiro num, i

    // lê e calcula a média das notas
    soma ← 0
    para i de LI até LS faça
        leia notas[i]
        soma ← soma + notas[i]
    fimpara

    // cálculo da média
    media ← soma/(LS − LI + 1)
```

```
// cálculo das notas superiores à média
num ← 0
para i de LI até LS faça
    se notas[i] > media então
        num ← num + 1
    fimse
fimpara

// escreve o número de alunos com nota superior à média
escreve "o número de alunos com nota superior à média é: ", num

fimalgoritmo
```

2. Escreva um algoritmo que declare uma variável de um tipo vetor de 10 elementos inteiros, leia 10 valores para esta variável e então escreva o maior e o menor valor do vetor e suas respectivas posições no vetor.

```
// algoritmo para ler um vetor de 10 elementos e depois escrever
// o maior e o menor elemento e suas respectivas posições
algoritmo encontra_menor_maior

    // declaração de constantes
    defina LI 1
    defina LS 10

    // cria um tipo vetor de números
    definatipo vetor[LI..LS] de inteiro vetor10

    // declaração de variáveis
    vetor10 n
    inteiro i, maior, menor, pmenor, pmaior

    // lê 10 números e armazena-os em um vetor
    para i de LI até LS faça
        escreva "Entre com o elemento ", i − LI + 1, " do vetor: "
        leia n[i]
    fimpara

    // determina menor e maior e suas posições
    menor ← n[LI]
```

```
    maior ← n[LI]
    pmenor ← 1
    pmaior ← 1
    para i de LI + 1 até LS faça
        se menor > n[i] então
            menor ← n[i]
            pmenor ← i − LI + 1
        senão
            se maior < n[i] então
                maior ← n[i]
                pmaior ← i − LI + 1
            fimse
        fimse
    fimpara

    // escreve o menor e o maior valor e suas posições
    escreva "O menor valor é: ", menor
    escreva "A posição do menor valor é: ", pmenor
    escreva "O maior valor é: ", maior
    escreva "A posição do maior valor é: ", pmaior

fimalgoritmo
```

3. O Departamento de Computação e Estatística (DCT) da UFMS deseja saber se existem alunos cursando, simultaneamente, as disciplinas Programação de Computadores e Introdução à Ciência da Computação. Para tal, estão disponíveis os números de matrícula dos alunos de Programação de Computadores (no máximo 60) e de Introdução à Ciência da Computação (no máximo 80).

   Escreva um algoritmo que leia cada conjunto de números de matrícula dos alunos e escreva as matrículas daqueles que estão cursando as duas disciplinas ao mesmo tempo. Considere que cada conjunto de números de matrícula encerre com a matrícula inválida 9999, de forma que o algoritmo saiba quando o conjunto de números já foi lido.

```
// algoritmo para determinar e escrever as matrícula iguais de duas
// de duas disciplinas
// algoritmo determina_matrículas_iguais

    // declaração de constantes
    defina LI 1
    defina LSPC 60
```

```
defina LSICC 80

// declaração de tipos
definatipo vetor[LI..LSPC] de inteiro vetorpc
definatipo vetor[LI..LSICC] de inteiro vetoricc

// declaração de variáveis
vetorpc vpc
vetoricc vicc
inteiro i, j, k, l, num

// lê as matrículas dos alunos de Programação de Computadores
i ← LI − 1
leia num
enquanto num = 9999 faça
    i ← i + 1
    vpc[i] ← num
    leia num
fimenquanto

// lê as matrículas dos alunos de Introdução
// à Ciência da Computação
j ← LI − 1
leia num
enquanto num = 9999 faça
    j ← j + 1
    vicc[j] ← num
    leia num
fimenquanto

// verifica se existem matrículas iguais. Se existirem, escreve
// as matrículas iguais.
para k de LI até i faça
    l ← LI
    enquanto l ≤ j faça
        se vpc[k] = vicc[l] então
            escreva vpc[k]
            l ← j + 1
        senão
            l ← l + 1
        fimse
```

```
        fimenquanto
    fimpara

fimalgoritmo
```

4. Escreva um algoritmo que recebe um inteiro $0 < n \leq 100$ e um vetor de n números inteiros cuja primeira posição é 1 e inverte a ordem dos elementos do vetor sem usar outro vetor.

```
// algoritmo para inverter a ordem dos elementos de um vetor sem
// utilizar vetor auxiliar
algoritmo inverte

    // declaração de constantes
    defina LI 1
    defina LS 100

    // cria um tipo vetor de números
    definatipo vetor[LI..LS] de inteiro vet_int

    // declaração de variáveis
    vet_int v
    inteiro i, temp

    // entrada de dados
    leia n
    para i de 1 até n faça
        leia v[i]
    fimpara

    // troca os elementos com seus simétricos
    para i de 1 até n DIV 2 faça
        temp ← v[i]
        v[i] ← v[n − i + 1]
        v[n − i + 1] ← temp
    fimpara

    // saída dos resultados
    para i de 1 até n faça
        escreva v[i]
```

fimpara

fimalgoritmo

5.2 Matrizes

Os vetores que estudamos até então são todos unidimensionais. Mas, podemos declarar e manipular vetores com mais de uma dimensão, os quais denominamos **matrizes**. Por exemplo, podemos definir uma estrutura de dados matriz 4 por 5 (uma *tabela* com 4 linhas e 5 colunas), denominada *tabela*, escrevendo as seguintes linhas:

```
defina LI_1 1
defina LS_1 4
defina LI_2 1
defina LS_2 5

definatipo vetor[LI_1..LS_1] de inteiro coluna
definatipo vetor[LI_2..LS_2] de coluna tabela
```

Neste exemplo, *coluna* é um tipo vetor de números, isto é, um tipo cuja estrutura de dados é um vetor unidimensional, como estamos acostumados a criar. Entretanto, *tabela* é um tipo vetor de *coluna*, ou seja, um tipo cuja estrutura de dados é um vetor bidimensional (uma matriz), pois cada um de seus elementos é do tipo vetor de números do tipo *coluna*!

Uma forma alternativa para definir o tipo *tabela* acima (e preferida pelos desenvolvedores de algoritmo) é a seguinte:

```
defina LI_1 1
defina LS_1 4
defina LI_2 1
defina LS_2 5

definatipo vetor[LI_1..LS_1, LI_2..LS_2] de inteiro tabela
```

Tanto em uma forma de definição quanto na outra, as constantes *LI_1*, *LS_1*, *LI_2* e *LS_2* estabelecem o número de elementos da tabela. Isto é, $LS\_1 - LI\_1 + 1$ é número de linhas da tabela, enquanto $LS\_2 - LI\_2 + 1$, o número de colunas.

Uma vez definido o tipo *tabela*, podemos declarar uma variável deste tipo da mesma maneira que declaramos variáveis dos demais tipos. Por exemplo, a linha abaixo declara uma variável denominada *mat* do tipo *tabela*:

tabela mat

A partir daí, podemos manipular a variável *mat* utilizando dois índices, em vez de apenas 1, para referenciar cada elemento desta matriz. O primeiro índice identifica a posição do elemento na primeira dimensão, enquanto o segundo identifica a posição do elemento na segunda dimensão. Se imaginarmos *mat* como uma tabela, então o primeiro índice corresponde à linha da tabela em que o elemento se encontra, enquanto o segundo, à coluna. Por exemplo, suponha que desejemos atribuir o valor 0 a todos os elementos da matriz *mat*. Isto pode ser feito com o seguinte trecho de código:

```
⋮
para i de LI_1 até LS_1 faça
    para j de LI_2 até LS_2 faça
        mat[i, j] ← 0
    fimpara
fimpara
⋮
```

Observe que um aninhamento de laços foi utilizado para "varrer" toda a matriz. O laço mais externo é responsável por variar o índice que representa a posição do elemento na primeira dimensão da matriz, enquanto o laço mais interno é utilizado para variar o índice que representa a posição do elemento na segunda dimensão da matriz. A sintaxe $mat[i, j]$ é usada para referenciarmos o elemento da linha i e coluna j da matriz *mat*.

5.2.1 Problemas e Soluções Envolvendo Matrizes

1. Escreva um algoritmo que declare uma variável de um tipo matriz de 4 por 5 elementos numéricos, leia valores para esta variável e escreva a soma dos elementos de cada linha da matriz, bem como a soma de todos os elementos.

```
// algoritmo para determinar e escrever a soma dos elementos das linhas
// de uma matriz 4 por 5 e a soma de todos os elementos da matriz
algoritmo soma_por_linha_e_de_linhas_de_matriz

    // declaração de constantes
    defina LI_1 1
    defina LS_1 4
```

```
defina LI_2 1
defina LS_2 5

// declaração de tipos
definatipo vetor[LI_1..LS_1, LI_2..LS_2] de inteiro tabela

// declaração de variáveis
tabela mat
inteiro i, j, somalin, somatot

// leitura dos elementos da matriz
para i de LI_1 até LS_1 faça
    para j de LI_2 até LS_2 faça
        escreva "entre com o elemento ", i - LI_1 + 1," e ", j - LI_2 + 1,
        " da matriz"
        leia mat[i, j]
    fimpara
fimpara

// soma elementos por linha e totaliza
somatot ← 0
para i de LI_1 até LS_1 faça

    // calcula a soma da linha i - LI_1 + 1
    somalin ← 0
    para j de LI_2 até LS_2 faça
        somalin ← somalin + mat[i, j]
    fimpara

    // exibe a soma da linha i - LI_1 + 1
    escreva "A soma dos elementos da linha ", i - LI_1 + 1,": ", somalin

    // acumula a soma das linhas para encontrar a soma total
    somatot ← somatot + mat[i, j]
fimpara

// exibe a soma total
escreva "A soma de todos os elementos é: ", somatot

fimalgoritmo
```

2. Dadas duas matrizes $A_{n\times m}$ e $B_{m\times p}$, com $n \leq 50$, $m \leq 50$ e $p \leq 50$. Obter a matriz matriz $C_{m\times p}$ onde $C = AB$.

```
// algoritmo para multiplicar duas matrizes
algoritmo produto_de_matrizes

// definição de constantes
    defina LI 1
    defina LS 50

// definição de tipos
    definatipo vetor[LI..LS, LI..LS] de inteiro matriz

// declaração de variáveis
    matriz A, B, C
    inteiro i, j, k, n, m, p

// entrada de dados
    leia n, m, p

    para i de 1 até n faça
        para j de 1 até m faça
            leia A[i, j]
        fimpara
    fimpara

    para i de 1 até m faça
        para j de 1 até p faça
            leia B[i, j]
        fimpara
    fimpara

// Cálculo da matriz produto
    para i de 1 até n faça
        para j de 1 até p faça
            C[i, j] ← 0
            para k de 1 até m faça
                C[i, j] ← A[i, k] * B[k, j] + C[i, j]
            fimpara
        fimpara
    fimpara
```

```
// escrita da matriz C
    para i de 1 até n faça
        para j de 1 até p faça
            escreva C[i, j]
        fimpara
    fimpara
fimalgoritmo
```

5.3 Registros

Um **registro** é uma estrutura de dados que agrupa dados de tipos distintos ou, mais raramente, do mesmo tipo. Um registro de dados é composto por um certo número de **campos de dado**, que são itens de dados individuais. Por exemplo, suponha que desejemos criar um algoritmo para manter um cadastro dos empregados de uma dada companhia. Neste cadastro, temos os seguintes dados para cada empregado:

- Nome do empregado.
- CPF do empregado.
- Salário do empregado.
- Se o empregado possui ou não dependentes.

Então, cada ficha deste cadastro pode ser representada por um registro que contém os campos nome, CPF, salário e indicação de existência de dependentes. Neste caso, a natureza dos campos é bem diversificada, pois nome pode ser representado por uma cadeia, CPF e salário por valores numéricos e existência de dependentes por um valor lógico.

5.3.1 Definindo uma Estrutura de Registro

A sintaxe para definição de um novo tipo registro é a seguinte:

definatipo registro
    < tipo do $campo_1$ > < $campo_1$ >
    < tipo do $campo_2$ > < $campo_2$ >
    ⋮
    < tipo do $campo_n$ > < $campo_n$ >
fimregistro <nome do tipo>

onde *nome do tipo* é o nome do tipo registro cuja estrutura está sendo criada, $campo_i$ é o nome do i-ésimo campo do registro e *tipo do* $campo_i$ é o tipo do i-ésimo campo do registro.

Como exemplo, vamos criar um registro para representar uma ficha do cadastro de empregados dado como exemplo anteriormente:

definatipo registro
    cadeia *nome*
    inteiro *CPF*
    real *salario*
    lógico *temdep*
fimregistro *regficha*

5.3.2 Criando e Manipulando Variáveis de Registros

Uma vez que um tipo registro tenha sido definido, podemos criar tantas variáveis daquele tipo quanto desejarmos, assim como fazemos com qualquer outro tipo complexo visto até então. Por exemplo, para criarmos três fichas de empregado para o nosso exemplo do cadastro, escrevemos:

regficha ficha1, ficha2, ficha3

Cada uma dessas três variáveis é um registro do tipo *regficha* e, portanto, cada uma delas possui quatro campos de dado: *nome*, *CPF*, *salario* e *temdep*.

Assim como variáveis de vetores e matrizes, variáveis registros são manipuladas através de suas partes constituintes. Em particular, uma variável registro é manipulada através de seus campos. Então, se desejarmos atribuir um valor a uma variável registro, temos de efetuar a atribuição de valores para seus campos constituintes, um de cada vez. Como exemplo, considere a seguinte atribuição do conjunto de valores "Beltrano de Tal", 123456789, 1800000 e falso à variável $ficha1$:

$ficha1.nome \leftarrow$ "Beltrano de Tal"
$ficha1.CPF \leftarrow 123456789$
$ficha1.salario \leftarrow 180.00$
$ficha1.temdep \leftarrow$ falso

A partir deste exemplo, podemos verificar que o acesso a cada campo de uma variável registro é realizado através da escrita do nome da variável registro seguido de um ponto (.) que, por sua vez, é seguido pelo nome do campo.

5.3.3 Vetores de Registros

Registros nos fornecem uma forma de agrupar dados de natureza distinta. Entretanto, criarmos uma única variável de um registro complexo não parece ser muito diferente de criarmos uma variável para cada campo do registro e tratá-las individualmente. Isto é, criar uma única variável de um registro complexo não nos ajudaria a resolver nossos problemas mais facilmente do que com o que aprendemos antes. A grande força dos registros reside no uso deles combinado com vetores. Por exemplo, um grupo de 100 empregados iria requerer um conjunto de 100 variáveis registros, o que pode ser conseguido através da criação de um vetor de 100 elementos do tipo registro em questão!

Um vetor de registros é criado da mesma forma que criamos vetores de qualquer dos tipos que aprendemos até então. Suponha, por exemplo, que desejemos criar um vetor de 100 elementos do tipo *regficha*. Isto é feito da seguinte forma:

```
defina LI 1
defina LS 100

definatipo registro
    cadeia nome
    inteiro CPF
    real salario
    lógico temdep
fimregistro regficha

definatipo vetor[LI..LS] de regficha vetcad
```

Agora, considere a declaração de uma variável do tipo *vetcad*:

```
vetcad cadastro
```

Para manipular um registro individual do vetor *cadastro*, utilizamos o nome da variável vetor e o índice do elemento correspondente ao registro que queremos acessar, como fazemos com um vetor de qualquer outro tipo. Daí em diante, procedemos como aprendemos na Subseção 5.3.2. Por exemplo, se quisermos atribuir o conjunto de valores "Sicrano de Tal", 987654321, 540,00 e verdadeiro ao primeiro registro de *cadastro*, fazemos:

$cadastro[1].nome \leftarrow$ "Sicrano de Tal"
$cadastro[1].CPF \leftarrow 987654321$
$cadastro[1].salario \leftarrow 540,00$
$cadastro[1].temdep \leftarrow$ verdadeiro

5.3.4 Registro de Tipos Complexos

Assim como combinamos vetores e registros para criar vetores de registro, podemos ter um registro de cujo um ou mais campos são de um outro tipo de registro ou vetores. Suponha, por exemplo, que queremos adicionar um campo ao nosso registro *regficha* para representar a conta bancária do empregado. Este campo pode ser um outro registro contendo os campos nome do banco, número da agência e número da conta bancária. Vejamos como ficaria a nova definição de *regficha*:

```
definatipo registro
    cadeia banco
    inteiro agencia
    inteiro numcc
fimregistro regbanco

definatipo registro
    cadeia nome
    inteiro CPF
    real salario
    lógico temdep
    regbanco conta
fimregistro regficha
```

O fato do campo *conta* de *regbanco* ser um outro registro faz com que a manipulação deste campo seja realizada, também, através de suas partes constituintes. Então, se criarmos uma variável *ficha* do tipo *regficha*, temos de manipular o campo *conta* de *ficha* como segue:

$ficha.conta.banco \leftarrow$ "Banco de Praça"
$ficha.conta.agencia \leftarrow 666$
$ficha.conta.numcc \leftarrow 4555$

5.3.5 Um Problema Envolvendo Registros

Suponha que você tenha sido contratado pela Copeve para construir parte de um programa para calcular o número de acertos em prova dos candidatos ao vestibular da UFMS. Para tal, você deverá escrever um algoritmo que lerá os dados de cada candidato e o gabarito das provas, calculará o número de acertos de cada candidato em cada prova e escreverá o número de acertos dos candidatos em cada prova.

Considere a resposta a cada questão de prova como sendo um dos cinco caracteres 'a', 'b', 'c', 'd' e 'e'.

Vamos iniciar a construção da solução deste problema pela definição dos tipos e estruturas de dados necessários. De acordo com a descrição do problema, temos um gabarito, que é representado por uma matriz, e um registro que contém uma matriz e um vetor. Cada candidato possui um tal registro. Logo, o conjunto de todos os candidatos pode ser representado por um vetor de registros.

O algoritmo é dado a seguir:

```
// algoritmo para calcular o número de acertos de
// candidatos ao vestibular
algoritmo vestibular

// definição de constantes
        defina NUMQUEST 40
        defina NUMPROVAS 5
        defina NUMCAND 5000

// definição de tipos
        definatipo vetor[1..NUMPROVAS] de inteiro vet_ac
        definatipo vetor[1..NUMQUEST, 1..NUMPROVAS] de caracter mat_gab

        definatipo registro
            inteiro codigo
            mat_gab X
            vet_ac A
        fimregistro regcand

        definatipo vetor[1..NUMCAND] de regcand vet_cand
```

```
//declaração de variáveis
        mat_gab G //matriz contendo o gabarito
        vet_cand candidato // vetor de registros contendo os dados dos candidatos
        inteiro i, j, k, n, soma

//leitura da matriz de gabarito
        para j de 1 até NUMPROVAS faça
            para i de 1 até NUMQUEST faça
                escreva "Resposta da questão", i, "da prova", j
                leia G[i, j]
            fimpara
        fimpara

//leitura da quantidade de candidatos
        escreva "Quantidade de candidatos"
        leia n

//leitura dos dados dos candidatos
        para k de 1 até n faça
            escreva "Código do candidato:"
            leia candidato[k].codigo
            para j de 1 até NUMPROVAS faça
                para i de 1 até NUMQUEST faça
                    escreva "Resposta da questão", i, "da prova", j
                    leia candidato[k].X[i, j]
                fimpara
            fimpara
        fimpara

//Cálculo dos acertos de cada candidato
        para k de 1 até n faça
            para j de 1 até NUMPROVAS faça
                soma ← 0
                para i de 1 até NUMQUEST faça
                    se G[i][j] = candidato[k].X[i][j] então
                        soma ← soma + 1
                    fimse
                fimpara
                candidato[k].A[j] ← soma
            fimpara
        fimpara
```

```
//Saída dos resultados
        para k de 1 até n faça
            escreva "Código:", candidato[k].codigo
            para j de 1 até NUMPROVAS faça
                escreva candidato[k].A[j], "acertos na prova", j
            fimpara
        fimpara

fimalgoritmo
```

Bibliografia

O texto deste capítulo foi elaborado a partir dos livros abaixo relacionados:

1. Farrer, H. et. al. *Algoritmos Estruturados.* Editora Guanabara, 1989.

2. Shackelford, R.L. *Introduction to Computing and Algorithms.* Addison-Wesley Longman, Inc, 1998.

Lista de exercícios

Vetores

1. Dada uma seqüência de n números, imprimi-la em ordem inversa à de leitura.

2. Deseja-se publicar o número de acertos de cada aluno em uma prova em forma de testes. A prova consta de 30 questões, cada uma com cinco alternativas identificadas por A, B, C, D e E. Para isso são dados:

   o cartão gabarito;
   o cartão de respostas de cada aluno, contendo o seu número e suas respostas.

3. Tentando descobrir se um dado era viciado, um dono de cassino o lançou n vezes. Dados os n resultados dos lançamentos, determinar o número de ocorrências de cada face.

4. Um jogador viciado de cassino deseja fazer um levantamento estatístico simples sobre uma roleta. Para isso, ele fez n lançamentos nesta roleta. Sabendo que uma roleta contém 37 números (de 0 a 36), calcular a freqüência de cada número desta roleta nos n lançamentos realizados.

5. Dados dois vetores x e y, ambos com n elementos, determinar o produto escalar desses vetores.

6. São dados dois números inteiros positivos p e q, sendo que o número de dígitos de p é menor ou igual ao número de dígitos de q. Verificar se p é um **subnúmero** de q.

   Exemplos:

   $p = 23$, $q = 57238$, p é subnúmero de q;
   $p = 23$, $q = 258347$, p não é subnúmero de q.

7. São dadas as coordenadas reais x e y de um ponto, un número natural n e as coordenadas reais de n pontos, com $1 \leq n \leq 100$. Deseja-se calcular e imprimir sem repetição os raios das circunferências centradas no ponto (x, y) que passem por pelo menos um dos n pontos dados.

   Exemplo:

   $$\begin{cases} (x, y) = (1.0, 1.0); \\ n = 5; \\ \text{Pontos:}(-1.0, 1.2), (1.5, 2.0), (0.0, -2.0), (0.0, 0.5), (4.0, 2.0) \end{cases}$$

   Nesse caso há três circunferências de raios $1.12, 2.01$ e 3.162.

   Observações:

   (a) A distância entre os pontos (a, b) e (c, d) é $\sqrt{(a-c)^2 + (b-d)^2}$.

   (b) Dois pontos estão na mesma circunferência se estão à mesma distância do centro.

8. Dadas duas cadeias (uma contendo uma frase e outra contendo uma palavra), determine o número de vezes que a palavra ocorre na frase.

   Exemplo:

   Para a palavra ANA e a frase:

   ANA E MARIANA GOSTAM DE BANANA.

   Temos que a palavra ocorre 4 vezes na frase.

9. Dada uma seqüência de n números reais, determinar os números que compõem a seqüência e o número de vezes que cada um deles ocorre na mesma.

   Exemplo:

   $n = 8$
   Seqüência: $-1.7, 3.0, 0.0, 1.5, 0.0, -1.7, 2.3, -1.7$
   Saída:

-1.7	ocorre	3	vezes
3.0	ocorre	1	vez
0.0	ocorre	2	vezes
1.5	ocorre	1	vez
2.3	ocorre	1	vez

10. Dizemos que uma seqüência de n elementos, com n par, é **balanceada** se as seguintes somas são todas iguais:

a soma do maior elemento com o menor elemento;
a soma do segundo maior elemento com o segundo menor elemento;
a soma do terceiro maior elemento com o terceiro menor elemento;
e assim por diante ...

Exemplo:

2 12 3 6 16 15 é uma seqüência balanceada, pois 16+2 = 15+3 = 12+6.

Dados n (n par) e uma seqüência de n números inteiros, verificar se essa seqüência é balanceada.

11. Dados dois números naturais m e n e duas seqüências ordenadas com m e n números inteiros, obter uma única seqüência ordenada contendo todos os elementos das seqüências originais sem repetição.

    Sugestão: imagine uma situação real, por exemplo, dois fichários em uma biblioteca.

12. Dadas duas seqüências com n números inteiros entre 0 e 9, interpretadas como dois números inteiros de n algarismos, calcular a seqüência de números que representa a soma dos dois inteiros.

    Exemplo:

    $n = 8$,

1^a seqüência		8	2	4	3	4	2	5	1
2^a seqüência	+	3	3	7	5	2	3	3	7
	1	1	6	1	8	6	5	8	8

13. Calcule o valor do polinômio $p(x) = a_0 + a_1x + \ldots + a_nx^n$ em k pontos distintos. São dados os valores de n (grau do polinômio), de $a_0, a_1, \ldots a_n$ (coeficientes reais do polinômio), de k e dos pontos $x_1, x_2, \ldots, x_n$.

14. Dado o polinômio $p(x) = a_0 + a_1x + \ldots a_nx^n$, isto é, os valores de n e de $a_0, a_1, \ldots, a_n$, determine os coeficientes reais da primeira derivada de $p(x)$.

15. Dados dois polinômios reais

$$p(x) = a_0 + a_1x + \ldots + a_nx^n \text{ e } q(x) = b_0 + b_1x + \ldots + b_mx^m$$

    determinar o produto desses dois polinômios.

16. Chama-se **seqüência de Farey relativa à** n a seqüência de frações racionais irredutíveis, dispostas em ordem crescente, com denominadores positivos e não maiores que n.

    Exemplo:

Se $n = 5$, os termos α da seqüência de Farey, tais que $0 \leq \alpha \leq 1$, são:

$$\frac{0}{1}, \frac{1}{5}, \frac{1}{4}, \frac{1}{3}, \frac{2}{5}, \frac{1}{2}, \frac{3}{5}, \frac{2}{3}, \frac{3}{4}, \frac{4}{5}, \frac{1}{1}.$$

Para gerarmos os termos α de uma seqüência de Farey tais que $0 \leq \alpha \leq 1$, podemos utilizar o seguinte processo. Começamos com as frações $\frac{0}{1}$ e $\frac{1}{1}$, e entre cada duas frações consecutivas $\frac{i}{j}$ e $\frac{k}{m}$, introduzimos a fração $\frac{i+k}{j+m}$ e assim sucessivamente enquanto $j + m \leq n$. Quando não for mais possível introduzir novas frações teremos gerados todos os termos α da seqüência de Farey relativa a n, tais que $0 \leq \alpha \leq 1$. Utilizando o método descrito, determine os termos α, $0 \leq \alpha \leq 1$, da seqüência de Farey relativa a n, com n inteiro positivo.

Sugestão: gere os numeradores e os denominadores em dois vetores.

17. Dada uma seqüência $x_1, x_2, \ldots, x_k$ de números inteiros, verifique se existem dois segmentos consecutivos iguais nesta seqüência, 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 m.

Exemplo:

Na seqüência $7, 9, \underline{5, 4}, \underline{5, 4}, 8, 6$ existem $i = 3$ e $m = 2$.

18. Dada uma seqüência $x_0, x_1, \ldots, x_{k-1}$ de k números inteiros, determinar um segmento de soma máxima.

Exemplo:

Na seqüência $5, 2, -2, -7, \underline{3, 14, 10, -3, 9}, -6, 4, 1$, a soma do segmento é 33.

5.4 Matrizes

1. Seja a seguinte matriz A:

175	225	10	9000	37	475
98	100	363	432	156	18
40	301	302	6381	1	0
402	4211	7213	992	442	7321
21	3	2	1	9000	2000

(a) Quantos elementos fazem parte do conjunto?

(b) Qual o conteúdo do elemento identificado por $A[4, 5]$?

(c) Qual o conteúdo da variável x após a execução do comando $x = A[3, 2] + A[4, 1]$?

(d) O que aconteceria caso fosse referenciado o elemento $A[5, 1]$ em um programa em linguagem C?

(e) Somar os elementos da quarta coluna ($A[1, 3] + A[1, 3] + A[2, 3] + A[3, 3] + A[4, 3]$).

(f) Somar os elementos da terceira linha ($A[2, 1] + A[2, 1] + A[2, 2] + A[2, 3] + A[2, 4] + A[2, 5]$).

2. Dada uma matriz real B, de 100 linhas e 200 colunas, escrever um programa que calcule o somatório dos elementos da quadragésima coluna e que calcule o somatório da trigésima linha.

3. Dadas duas matrizes A e B, de dimensões $n \times m$, fazer um programa que calcule a matriz $C_{n \times m} = A + B$.

4. Fazer um programa que dada uma matriz $A_{n \times m}$, determine A^t.

5. Dada uma matriz real A com m linhas e n colunas e um vetor real V com n elementos, determinar o produto de A por V.

6. Um vetor real x com n elementos é apresentado como resultado de um sistema de equações lineares $Ax = b$ cujos coeficientes são representados em uma matriz real $A_{m \times n}$ e os lados direitos das equações em um vetor real b de m elementos. Verificar se o vetor x é realmente solução do sistema dado.

7. Dadas duas matrizes reais $A_{m \times n}$ e $B_{n \times p}$, calcular o produto de A por B.

8. Dada uma matriz real $A_{m \times n}$, verificar se existem elementos repetidos em A.

9. Seja $A_{26 \times 10}$ uma matriz fornecida, cujo conteúdo é a população dos 10 municípios mais populosos dos 26 estados brasileiros ($A[i, j]$ representa a população do j-ésimo município do i-ésimo estado). Determinar o número do município mais populoso e o número do estado a que pertence. Considerando que a primeira coluna sempre contém a população da capital do estado, calcular a média da população das capitais dos 26 estados.

10. Deseja-se atualizar as contas correntes dos clientes de uma agência bancária. É dado o cadastro de n clientes contendo, para cada cliente, o número de sua conta e o seu saldo; o cadastro está ordenado pelo número da conta. Em seguida, é

dado o número m de operações efetuadas no dia e, para cada operação, o número da conta, uma letra **C** ou **D** indicando se a operação é de crédito ou débito e o valor da operação. Emitir o extrato atualizado dos clientes.

11. Deseja-se fazer a emissão da folha de pagamento de uma empresa. Para cada um dos n funcionários são dadas as seguintes informações:

Código	Descrição
NOME	Nome do funcionário
SAL	Salário do funcionário
HED	Horas extras diurnas
HEN	Horas extras noturnas
ND	Número de dependentes
FAL	Faltas em horas
DE	Descontos eventuais
REF	Gastos com refeições feitas na empresa
VAL	Vales retirados durante o mês

Para cada funcionário, emitir as seguintes informações:

Nome,
Salário,
Horas Extras = HED $*$ SAL/160 + HEN $*$ 1,2 $*$ SAL/160,
Salário Família = ND $*$ 0,05 $*$ Salário Mínimo vigente,
Salário Bruto = Salário + Horas Extras + Salário Família.

e os descontos efetuados:

INSS = 0,08 $*$ SAL,
Faltas = FAL $*$ SAL/160,
Refeições,
Vales,
Descontos Eventuais,
Imposto de Renda = 0,08 $*$ Salário Bruto.

e finalmente o seu Salário Líquido:

Salário Líquido = Salário Bruto - Descontos.

12. Dizemos que uma matriz inteira $A_{n\times n}$ é uma **matriz de permutação** se em cada linha e em cada coluna houver $n-1$ elementos nulos e um único elemento 1.

Exemplo:

A matriz abaixo é de permutação

$$\begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Observe que

$$\begin{pmatrix} 2 & -1 & 0 \\ -1 & 2 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

não é de permutação.

Dada uma matriz inteira $A_{n\times n}$, verificar se A é de permutação.

13. Dada uma matriz $A_{m\times n}$, imprimir o número de linhas e o número de colunas nulas da matriz.

    Exemplo:

    $m = 4$ e $n = 4$

    $$\begin{pmatrix} 1 & 0 & 2 & 3 \\ 4 & 0 & 5 & 6 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

    tem 2 linhas nulas e 1 coluna nula.

14. Dizemos que uma matriz quadrada inteira é um **quadrado mágico**[1] se a soma dos elementos de cada linha, a soma dos elementos de cada coluna e a soma dos elementos da diagonal principal e secundária são todas iguais.

    Exemplo:

    A matriz

    $$\begin{pmatrix} 8 & 0 & 7 \\ 4 & 5 & 6 \\ 3 & 10 & 2 \end{pmatrix}$$

    é um quadrado mágico.

    Dada uma matriz quadrada inteira $A_{n\times n}$, verificar se A é um quadrado mágico.

15. (a) Imprimir as n primeiras linhas do triângulo de Pascal[2].

[1] O primeiro registro conhecido de um quadrado mágico vem da China e data do segundo século antes de Cristo.

[2] Descoberto em 1654 pelo matemático francês Blaise Pascal.

$$
\begin{array}{cccccc}
1 & & & & & \\
1 & 1 & & & & \\
1 & 2 & 1 & & & \\
1 & 3 & 3 & 1 & & \\
1 & 4 & 6 & 4 & 1 & \\
1 & 4 & 10 & 10 & 5 & 1 \\
\vdots & & & & &
\end{array}
$$

(b) Imprimir as n primeiras linhas do triângulo de Pascal usando apenas um vetor.

16. Um jogo de palavras cruzadas pode ser representado por uma matriz $A_{m\times n}$ onde cada posição da matriz corresponde a um quadrado do jogo, sendo que 0 indica um quadrado branco e -1 indica um quadrado preto. Indicar na matriz as posições que são início de palavras horizontais e/ou verticais nos quadrados correspondentes (substituindo os zeros), considerando que uma palavra deve ter pelo menos duas letras. Para isso, numere consecutivamente tais posições.

Exemplo:

Dada a matriz

$$
\begin{pmatrix}
0 & -1 & 0 & -1 & -1 & 0 & -1 & 0 \\
0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 \\
0 & 0 & -1 & -1 & 0 & 0 & -1 & 0 \\
-1 & 0 & 0 & 0 & 0 & -1 & 0 & 0 \\
0 & 0 & -1 & 0 & 0 & 0 & -1 & -1
\end{pmatrix}
$$

a saída deverá ser

$$
\begin{pmatrix}
1 & -1 & 2 & -1 & -1 & 3 & -1 & 4 \\
5 & 6 & 0 & 0 & -1 & 7 & 0 & 0 \\
8 & 0 & -1 & -1 & 9 & 0 & -1 & 0 \\
-1 & 10 & 0 & 11 & 0 & -1 & 12 & 0 \\
13 & 0 & -1 & 14 & 0 & 0 & -1 & -1
\end{pmatrix}
$$

17. Uma matriz $D_{8\times 8}$ pode representar a posição atual de um jogo de damas, sendo que 0 indica uma casa vazia, 1 indica uma casa ocupada por uma peça branca e -1 indica uma casa ocupada por uma peça preta. Supondo que as peças pretas estão se movendo no sentido crescente das linhas da matriz D, determinar as posições das peças pretas que:

(a) podem tomar peças brancas;

(b) podem mover-se sem tomar peças;

(c) não podem se mover.

18. Um campeonato de futebol foi disputado por n times identificados pelos seus nomes. Para cada time são considerados os seguintes dados:

PG	número de pontos ganhos (3 por vitória, 1 por empate e 0 por derrota);
GM	número de gols marcados;
GS	número de gols sofridos;
S	saldo de gols (GM - GS);
V	número de vitórias;
GA	*goal average* ou média de gols (GM / GS).

    (a) Dados os resultados de m jogos, imprima uma tabela com todos os dados (PG, GM, GS, S, V, GA, igual àquela que sai no jornal) dos n times. Cada resultado é representado na forma (t_1, t_2, n_1, n_2) cuja interpretação é a seguinte: no jogo $t_1 \times t_2$ o resultado foi $n_1 \times n_2$.

    Exemplo:

    São Paulo, Milan, 3, 2
    Palmeiras, Criciúma, 1, 2
    ⋮

    (b) Com os mesmos dados do item (a), imprima a classificação dos times no campeonato (do primeiro para o último). A classificação é pelo número de pontos ganhos (PG) e em segundo lugar pelo saldo de gols (S). Se houver empate segundo os dois critérios, classifique os times envolvidos como quiser.

    (c) Um grupo de t torcedores organizou um *bolão* sobre os resultados dos m jogos. Cada resultado certo vale 5 pontos (inclusive o placar) ou 3 pontos (apenas o vencedor ou empate). Com os dados do item (a) e mais os palpites que são compostos de m pares de números inteiros $(p_1, q_1), (p_2, q_2), \ldots, (p_m, q_m)$, onde o i-ésimo par representa o palpite do i-ésimo jogo, descubra o nome do ganhador do bolão.

19. Os elementos a_{ij} de uma matriz inteira $A_{n\times n}$ representam os custos de transporte da cidade i para a cidade j. Dados n itinerários, cada um com k cidades, calcular o custo total para cada itinerário.

    Exemplo:

    $$\begin{pmatrix} 4 & 1 & 2 & 3 \\ 5 & 2 & 1 & 400 \\ 2 & 1 & 3 & 8 \\ 7 & 1 & 2 & 5 \end{pmatrix}$$

    O custo do itinerário 0 3 1 3 3 2 1 0 é

    $a_{03} + a_{31} + a_{13} + a_{33} + a_{32} + a_{21} + a_{10} = 3 + 1 + 400 + 5 + 2 + 1 + 5 = 417.$

20. Considere n cidades numeradas de 0 a $n-1$ 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_{ij} assumem o valor 1 ou 0, conforme exista ou não estrada direta que saia da cidade i e chegue à cidade j. Assim, os elementos da coluna j indicam as estradas que chegam à cidade j. Por conveção $l_{ii} = 1$.

    A figura abaixo mostra um exemplo para $n = 4$.

    (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:

        i. As cidades isoladas, isto é, as que não têm ligação com nenhuma outra;

        ii. As cidade das quais não há saída, apesar de haver entrada;

        iii. As cidades das quais há saída sem haver entrada.

    (f) Dada uma seqüência de m inteiros cujos valores estão entre 0 e $n-1$, verificar se é possível realizar o roteiro correspondente. No exemplo dado, o roteiro representado pela seqüência 2 3 2 1 0, com $m = 5$, é impossível.

    (g) Dados k e p, determinar se é possível ir da cidade k para a cidade p pelas estradas existentes. Você consegue encontrar o menor caminho entre as duas cidades?

Capítulo 6

Modularização

Os algoritmos que temos construído até então são muito simples, pois resolvem problemas simples e apresentam apenas os componentes mais elementares dos algoritmos: constantes, variáveis, expressões condicionais e estruturas de controle. Entretanto, a maioria dos algoritmos resolve problemas complicados, cuja solução pode ser vista como formada de várias subtarefas ou *módulo*, cada qual resolvendo uma parte específica do problema.

Neste tópico, veremos como escrever um algoritmo constituído de vários módulos e como estes módulos trabalham em conjunto para resolver um determinado problema algorítmico.

6.1 O Quê e Por Quê?

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

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

- **Torna o algoritmo mais fácil de escrever**. O desenvolvedor pode focalizar pequenas partes de um problema complicado e escrever a solução para estas

partes, uma de cada vez, ao invés de tentar resolver o problema como um todo de uma só vez.

- **Torna o algoritmo mais fácil de ler**. O fato do algoritmo estar dividido em módulos permite que alguém, que não seja o seu autor, possa entender o algoritmo mais rapidamente por tentar entender os seus módulos separadamente, pois como cada módulo é menor e mais simples do que o algoritmo monolítico correspondente ao algoritmo modularizado, entender cada um deles separadamente é menos complicado do que tentar entender o algoritmo como um todo de uma só vez.

- **Eleva o nível de abstração**. É possível entender o que um algoritmo faz por saber apenas o *que* os seus módulos fazem, sem que haja a necessidade de entender os detalhes internos aos módulos. Além disso, se os detalhes nos interessam, sabemos exatamente onde examiná-los.

- **Economia de tempo, espaço e esforço**. Freqüentemente, necessitamos executar uma mesma tarefa em vários lugares de um mesmo algoritmo. Uma vez que um módulo foi escrito, ele pode, como veremos mais adiante, ser "chamado" quantas vezes quisermos e de onde quisermos no algoritmo, evitando que escrevamos a mesma tarefa mais de uma vez no algoritmo, o que nos poupará tempo, espaço e esforço.

- **Estende a linguagem**. Uma vez que um módulo foi criado, podemos utilizá-lo em outros algoritmos sem que eles sejam modificados, da mesma forma que usamos os operadores `leia` e `escreva` em nossos algoritmos.

A maneira mais intuitiva de proceder à modularização de problemas é realizada através da definição de um módulo principal, que organiza e coordena o trabalho dos demais módulos, e de módulos específicos para cada uma das subtarefas do algoritmo. O módulo principal solicita a execução dos vários módulos em uma dada ordem, os quais, antes de iniciar a execução, recebem dados do módulo principal e, ao final da execução, devolvem o resultado de suas computações.

6.2 Componentes de um módulo

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

interface de um módulo é tudo o que é necessário para a utilização correta do módulo em um algoritmo.

A interface de um módulo é definida em termos de parâmetros. Um **parâmetro** é um tipo especial de variável que o valor de um dado seja passado entre um módulo e qualquer algoritmo que possa usá-lo. Existem três tipos de parâmetros:

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

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

6.3 Ferramentas para Modularização

Há dois tipos de módulos:

- **função**: uma função é um módulo que produz um único valor de saída. Ela pode ser vista como uma expressão que é avaliada para um único valor, sua saída, assim como uma função em Matemática.
- **procedimento**: um procedimento é um tipo de módulo usado para várias tarefas, não produzindo valores de saída.

As diferenças entre as definições de função e procedimento permitem determinar se um módulo para uma dada tarefa deve ser implementado como uma função ou procedimento. Por exemplo, se um módulo para determinar o menor de dois números é necessário, ele deve ser implementado como uma função, pois ele vai produzir um valor de saída. Por outro lado, se um módulo para determinar o maior e o menor

valor de uma seqüência de números é requerido, ele deve ser implementado como um procedimento, pois ele vai produzir dois valores de saída.

Vamos considerar que a produção de um valor de saída por uma função é diferente da utilização de parâmetros de saída ou de entrada e saída. Veremos mais adiante que os parâmetros de saída e de entrada e saída modificam o valor de uma variável do algoritmo que a chamou, diferentemente do valor de saída produzido por uma função que será um valor a ser usado em uma atribuição ou evolvido em alguma expressão.

6.4 Criando Funções e Procedimentos

A fim de escrevermos uma função ou procedimento, precisamos construir as seguintes partes: interface e corpo. Como dito antes, a interface é uma descrição dos parâmetros do módulo, enquanto corpo é a seqüência de instruções do módulo. A interface de uma função tem a seguinte forma geral:

função <tipo de retorno> <nome> (<lista de parâmetros formais>)

onde

- *nome* é um identificador único que representa o nome da função;
- *lista de parâmetros formais* é uma lista dos parâmetros da função;
- *tipo de retorno* é o tipo do valor de retorno da função.

Por exemplo:

função real *quadrado* (real r)

Logo após a descrição da *interface* podemos descrever o corpo do algoritmo. Para delimitar os comandos que fazem parte da função, utilizamos fimfunção.

O corpo de uma função é uma seqüência de comandos que compõe a função e que sempre finaliza com um comando especial denominado **comando de retorno**. O comando de retorno é da seguinte forma

retorna <valor de retorno>

onde *valor de retorno* é o valor de saída produzido pela função. Por exemplo:

retorna x

Vejamos um exemplo de uma função. Suponha que desejemos construir uma função para calcular o quadrado de um número. O número deve ser passado para a função, que calculará o seu quadrado e o devolverá para o algoritmo que a solicitou. Vamos denominar esta função de *quadrado*, como mostrado a seguir:

```
// função para calcular o quadrado de um número
função real quadrado (real r)

    // declaração de variáveis
    real x

    // calcula o quadrado
    x ← r * r

    // retorna o quadrado calculado
    retorna x

fimfunção
```

A interface de um procedimento tem a seguinte forma geral:

procedimento <nome> (<lista de parâmetros formais>)

onde

- *nome* é um identificador único que representa o nome do procedimento;
- *lista de parâmetros formais* é uma lista dos parâmetros do procedimento.

Por exemplo,

procedimento *ler_número* (ref real *val*)

O corpo de um procedimento é uma seqüência de comandos que não possui comando de retorno, pois apenas funções possuem tal tipo de comando. Para delimitar os comandos que fazem parte do procedimento, utilizamos fimprocedimento.

Vejamos um exemplo de um procedimento. Suponha que desejemos construir um procedimento para ler um número e passar o número lido para o algoritmo que solicitou a execução do algoritmo através de um parâmetro de saída. Denominemos este procedimento de *ler_número*, como mostrado a seguir:

```
// procedimento para ler um número
procedimento ler_número (ref real val)

    // Solicita a entrada do número
    escreva "Entre com um número:"
    leia val

fimprocedimento
```

Aqui, *val* é o parâmetro de saída que conterá o valor de entrada que será passado para o algoritmo que solicitar a execução do procedimento *ler_número*. A palavra-chave ref, que antecede o nome da variável na lista de parâmetros formais do procedimento, é utilizada para definir *val* como parâmetro de saída ou entrada e saída.

6.5 Chamando Funções e Procedimentos

Funções e procedimentos não são diferentes apenas na forma como são implementados, mas também na forma como a solicitação da execução deles, ou simplesmente **chamada**, deve ser realizada. A chamada de uma função é usada como um valor constante que deve ser atribuído a uma variável ou como parte de uma expressão, enquanto a chamada de um procedimento é realizada como um comando a parte.

Como exemplo, considere um algoritmo para ler um número e exibir o seu quadrado. Este algoritmo deve utilizar o procedimento *ler_número* e a função *quadrado*, vistos antes, para ler o número e obter o seu quadrado, respectivamente. O algoritmo segue abaixo:

```
// algoritmo para calcular e exibir o quadrado de um número
// fornecido como entrada
algoritmo calcula_quadrado

    // declaração de variáveis
    real num, x

    // lê um número
    ler_número(num)

    // calcula o quadrado do número lido
    x ← quadrado(num)

    // escreve o quadrado
    escreva "O quadrado do número é:", x

fimalgoritmo
```

Note a diferença da chamada do procedimento *ler_número* para a chamada da função *quadrado*. Na chamada do procedimento, temos uma instrução por si só. Por outro lado, a chamada da função ocupa o lugar de um valor ou expressão em uma instrução de atribuição. Isto porque a chamada

$$quadrado(num)$$

é substituída pelo valor de retorno da função.

Tanto na chamada do procedimento *ler_número* quanto na chamada da função *quadrado*, temos um parâmetro: a variável *num*. Na chamada do procedimento *ler_número*, *num* é um parâmetro de saída, como definido na interface do procedimento, e, portanto, após a execução do procedimento, *num* conterá o valor lido dentro do procedimento. Já na chamada da função *quadrado*, *num* é um parâmetro de entrada, como definido na interface da função, e, assim sendo, o valor de *num* é passado para a função.

A variável *num* é denominada **parâmetro real**. Um parâmetro real especifica um valor passado para o módulo pelo algoritmo que o chamou ou do módulo para o algoritmo que o chamou. Os parâmetros formais são aqueles da declaração do módulo. A lista de parâmetros reais deve concordar em número, ordem e tipo com a lista de prâmetros formais.

6.6 Passagem de Parâmetros

Os valores dos parâmetros reais de entrada são passados por um mecanismo denominado **cópia**, enquanto os valores dos parâmetros reais de saída e entrada e saída são passados por um mecanismo denominado **referência**.

O mecanismo de passagem por cópia funciona da seguinte forma. O valor do parâmetro real (uma constante ou o valor de uma variável ou expressão) de entrada é atribuído ao parâmetro formal quando da chamada do procedimento/função. Por exemplo, na chamada

$$x \leftarrow quadrado(num)$$

o valor da variável *num* é atribuído ao parâmetro formal *r* da função *quadrado*.

No mecanismo de passagem por referência, quando da chamada do procedimento/função, o parâmetro formal passa a compartilhar a mesma área de armazenamento de parâmetro real, assim, qualquer alteração no valor do parâemtro formal feita pelo procedimento/função acarretará uma modificação no parâmetro real quando do término do procedimento/função. Sendo assim, quando a chamada

ler_número(num)

é executada, a variável real *num* e a variável formal *val* (que está precedida da palavra chave ref) compartilham uma mesma área de armazenamento, assim, *num* e *val* contêm o mesmo valor. Quando é feita a leitura do dado e armazenada em *val*, esta atualização afetará o valor de *num*. Ao término do procedimento a variável *num* conterá o valor atualizado.

Devemos observar que os parâmetros reais de saída e de entrada e saída devem *obrigatoriamente* ser variáveis, uma vez que não faz sentido modificar o valor de uma constante ou de uma expressão.

6.7 Escopo de Dados e Código

O **escopo** de um módulo (ou variável) de um algoritmo é a parte ou partes do algoritmo em que o módulo (ou variável) pode ser referenciado. Quando iniciamos o estudo de modularização é natural nos perguntarmos qual é o escopo de um dado módulo e

das constantes ou variáveis nele presentes. Em particular, o escopo de um módulo determina quais são os demais módulos do algoritmo que podem chamar-lhe e quais ele pode chamar.

Os módulos de um algoritmo são organizados por níveis. No primeiro nível, temos apenas o algoritmo principal. Aqueles módulos que devem ser acessados pelo algoritmo principal devem ser escritos dentro dele e, nesta condição, são ditos pertencerem ao segundo nível. Os módulos escritos dentro de módulos de segundo nível são ditos módulos de terceiro nível. E assim sucessivamente. Por exemplo:

```
// algoritmo para calcular o quadrado de um número fornecido
// como entrada
algoritmo calcula_quadrado

    // módulo para cálculo do quadrado de um número
    função real quadrado (real r)

        // declaração de variáveis
        real x

        // calcula o quadrado
        x ← r * r

        // passa para o algoritmo chamador o valor obtido
        retorna x

    fimfunção

    // módulo para ler um número
    procedimento ler_número (ref real val)

        // solicita um número ao usuário
        escreva "Entre com um número: "
        leia val

    fimprocedimento

    // declaração de constantes e variáveis
    real num, x
```

```
    // lê um número
    ler_número(num)

    // calcula o quadrado
    x ← quadrado(num)

    // escreve o quadrado
    escreva "O quadrado do número é: ", x

fimalgoritmo
```

Aqui, desejávamos que a função *quadrado* e o procedimento *ler_número* fossem chamados pelo módulo principal e, por isso, escrevemos tais módulos dentro do módulo principal, no segundo nível da hierarquia modular do algoritmo.

A regra para determinar o escopo de um módulo é bastante simples: um módulo X escrito dentro de um módulo A qualquer pode ser acessado apenas pelo módulo A ou por qualquer módulo escrito dentro de A ou descendente (direto ou não) de algum módulo dentro de A.

Variáveis podem ser locais ou globais. Uma variável (constante) é dita **local** a um módulo se ela é declarada naquele módulo. Por exemplo, a variável x da função *quadrado* é uma variável local a esta função. Uma variável é dita **global** a um módulo quando ela não está declarada no módulo, mas pode ser referenciada a partir dele. Neste curso, consideraremos que variáveis são sempre locais, isto é, nenhum módulo poderá referenciar uma variável declarada em outro módulo.

6.8 Problemas e Soluções

Nesta seção, veremos três problemas envolvendo funções e procedimentos e suas respectivas soluções.

1. Escreva um algoritmo para ler dois números e exibir o menor dos dois. A verificação de qual deles é o menor deve ser realizada por uma função.

```
    // algoritmo para o encontrar e exibir o menor de dois números
    algoritmo encontra_menor_de_dois

        // módulo para encontrar o menor de dois números
```

```
função inteiro menor_de_dois (inteiro a, b)

    // declaração de variáveis
    inteiro menor

    menor ← a
    se a > b então
        menor ← b
    fimse

    retorna menor

fimfunção

// declaração de constantes e variáveis
inteiro x, y, z


// lê dois números
escreva "Entre com dois números: "
leia x, y

// obtém o menor dos dois
z ← menor_de_dois(x, y)

// escreve o menor dos dois
escreva "O menor dos dois é: ", z

fimalgoritmo
```

2. Escreva um algoritmo para ler três números e exibir o maior e o menor dos três. A obtenção do maior e do menor número deve ser realizada por um procedimento.

```
// algoritmo para o encontrar e exibir o menor e o maior de três
// números
algoritmo encontra_min_max

    // módulo para encontrar o menor e o maior de três números
    procedimento min_max(inteiroa, b, c,
                                ref inteiro min, max)
```

```
        // encontra o maior dos três
        se a ≥ b e a ≥ c então
            max ← a
        senão
            se b ≥ c então
                max ← b
            senão
                max ← c
            fimse
        fimse

        // encontra o menor dos três
        se a ≤ b e a ≤ c então
            min ← a
        senão
            se b ≤ c então
                min ← b
            senão
                min ← c
            fimse
        fimse

    fimprocedimento

    // declaração de constantes e variáveis
    inteiro x, y, z, menor, maior

    // lê três números
    escreva "Entre com três números: "
    leia x, y, z

    // obtém o menor e o maior dos três
    min_max(x, y, z, menor, maior)

    // escreve o menor e o maior dos três
    escreva "O menor dos três é: ", menor
    escreva "O maior dos três é: ", maior

fimalgoritmo
```

3. Escreva um algoritmo para ler três números e escrevê-los em ordem não decres-

cente. Utilize, obrigatoriamente, um procedimento para trocar o valor de duas variáveis.

```
// algoritmo para ler três números e escrevê-los em ordem
// não decrescente
algoritmo ordena_três

    // módulo para trocar o valor de duas variáveis
    procedimento troca(ref inteiro a, b)

        // declaração de variáveis
        inteiro aux

        // troca os valores
        aux ← a
        a ← b
        b ← aux

    fimprocedimento

    // declaração de constantes e variáveis
    inteiro l, m, n

    // lê os três números
    escreva "Entre com três números: "
    leia l, m, n

    // encontra o menor e põe em l
    se l > m ou l > n então
        se m ≤ n então
            troca(l, m)
        senão
            troca(l, n)
        fimse
    fimse

    se m > n então
        troca(m, n)
    fimse

    escreva "Os números em ordem não decrescente: ", l, m, n
```

fimalgoritmo

Neste algoritmo, os parâmetros do procedimento *troca* são parâmetros de entrada e saída, pois para trocar os valores dos parâmetros reais dentro de um procedimento, estes valores devem ser copiados para os parâmetros formais e as mudanças nos parâmetros formais devem ser refletidas nos parâmetros reais, uma vez que as variáveis precisam retornar do procedimento com os valores trocados.

Bibliografia

O texto deste capítulo foi elaborado a partir dos livros abaixo relacionados:

1. Farrer, H. et. al. *Algoritmos Estruturados*. Editora Guanabara, 1989.
2. Shackelford, R.L. *Introduction to Computing and Algorithms*. Addison-Wesley Longman, Inc, 1998.

Lista de Exercícios

1. Para se determinar o número de lâmpadas necessárias para cada cômodo de uma residência, existem normas que fornecem o mínimo de potência de iluminação exigida por metro quadrado (m^2) conforme a utilização deste cômodo. Suponha que só serão usadas lâmpadas de 60W.

   Seja a seguinte tabela:

Utilização	Classe	Potência/m^2
quarto	1	15
sala de TV	1	15
salas	2	18
cozinha	2	18
varandas	2	18
escritório	3	20
banheiro	3	20

   (a) Faça um módulo (função ou um procedimento) que recebe a classe de iluminação de um cômodo e suas duas dimensões e devolve o número de lâmpadas necessárias para o cômodo.

   (b) Faça um algoritmo que leia um número indeterminado de informações, contendo cada uma o nome do cômodo da residência, sua classe de iluminação e as suas duas dimensões e, com base no módulo anterior, imprima a área de cada cômodo, sua potência de iluminação e o número total de lâmpadas necessárias. Além disso, seu algoritmo deve calcular o total de lâmpadas necessárias e a potência total para a residência.

2. A comissão organizadora de um *rallye* automobilístico decidiu apurar os resultados da competição através de um processamento eletrônico. Um dos programas necessários para a classificação das equipes é o que emite uma listagem geral do desempenho das equipes, atribuindo pontos segundo determinadas normas.

   (a) Escreva um módulo (função ou procedimento) que calcula os pontos de cada equipe em cada uma das etapas do *rallye*, seguindo o seguinte critério. Seja

Δ o valor absoluto da diferença entre o tempo-padrão e o tempo despendido pela equipe numa etapa (fornecidos como parâmetros):

$\Delta < 3$ minutos	atribuir 100 pontos à etapa
$3 \leq \Delta \leq 5 minutos$	atribuir 80 pontos à etapa
$\Delta > 5$	atribuir $80 - \frac{\Delta - 5}{5}$ pontos à etapa

(b) Faça um algoritmo que leia os tempos-padrão (em minutos decimais) para as três etapas da competição, leia um número indeterminado de informações, contendo para cada equipe o seu número de inscrição e os tempos (em minutos decimais) despendidos para cumprir as três etapas e, utilizando o módulo anterior, calcule os pontos obtidos por cada equipe em cada etapa, a soma total de pontos por equipe, e a equipe vencedora.

3. (a) Faça uma função *quantosdias* que recebe o dia, o mês e o ano de uma data e retorna um valor que contém o número de dias do ano até a data fornecida.

(b) Faça um algoritmo que recebe n pares de datas, onde cada par deve ser fornecido no formato *dia1, mês1, ano1, dia2, mês2, ano2*, verifique se as datas estão corretas e mostre a diferença, em dias, entre essas duas datas. Utilize a função anterior para o cálculo do total de dias de cada data.

4. (a) Escreva uma função que recebe dois números inteiros positivos e determina o produto dos mesmos, utilizando o seguinte método de multiplicação:

i. dividir, sucessivamente, o primeiro número por 2, até que se obtenha 1 como quociente;

ii. paralelamente, dobrar, sucessivamente, o segundo número;

iii. somar os números da segunda coluna que tenham um número ímpar na primeira coluna. O total obtido é o produto procurado.
Exemplo: 9×6

9	6	$\rightarrow$	6
4	12		
2	24		
1	48	$\rightarrow$	48
			54

(b) Escreva um programa que leia n pares de números e calcule os respectivos produtos, utilizando a função anterior.

5. Um número a é dito ser *permutação* de um número b se os dígitos de a formam uma permutação dos dígitos de b.

Exemplo:

5412434 é uma permutação de 4321445, mas não é uma permutação de 4312455.

Observação: considere que o dígito 0 (zero) não aparece nos números.

(a) Faça uma função *contadígitos* que, dados um inteiro n e um inteiro d, $0 < d \leq 9$, devolve quantas vezes o dígito d aparece em n;

(b) Utilizando a função do item anterior, faça um algoritmo que leia dois números a e b e responda se a é permutação de b.

6. Um número b é dito ser *sufixo* de um número a se o número formado pelos últimos dígitos de a são iguais a b.

Exemplo:

a	b		
567890	890	→	sufixo
1234	1234	→	sufixo
2457	245	→	não é sufixo
457	2457	→	não é sufixo

(a) Construa uma função *sufixo* que dados dois números inteiros a e b verifica se b é um sufixo de a.

(b) Utilizando a função do item anterior, escreva um algoritmo que leia dois números inteiros a e b e verifica se o menor deles é subseqüência do outro.

Exemplo:

a	b		
567890	678	→	b é subseqüência de a
1234	2212345	→	a é subseqüência de b
235	236	→	Um não é subseqüência do outro

7. Uma seqüência de n números inteiros não nulos é dita m-alternante se é constituída por m segmentos: o primeiro com um elemento, o segundo com dois elementos e assim por diante até a m-ésima, com M elementos. Além disso, os elementos de um mesmo segmento devem ser todos pares ou todos ímpares e para cada segmento, se seus elementos forem todos pares (ímpares), os elementos do segmento seguinte devem ser todos ímpares (pares).

Por exemplo:

A seqüência com $n = 10$ elementos: 8 3 7 2 10 4 5 13 4 11 é 4-alternante.

A seqüência com $n = 3$ elementos: 7 2 8 é 2-alternante.

A seqüência com $n = 8$ elementos: 1 12 4 2 13 5 12 6 não é alternante, pois o último segmento não tem tamanho 4.

(a) Escreva uma função *bloco* que recebe como parâmetro um inteiro n e lê n inteiros do teclado, devolvendo um dos seguintes valores:
  0, se os n números lidos forem pares;
  1, se os n números lidos forem ímpares;
 -1, se entre os n números lidos há números com paridades diferentes.

(b) Utilizando a função do item anterior, escreva um algoritmo que, dados um inteiro n ($n \geq 1$) e uma seqüência de n números inteiros, verifica se ela é m-alternante. O algoritmo deve imprimir o valor de m ou dar uma resposta dizendo que a seqüência não é alternante.

8. Considere as seguintes fórmulas de recorrência:

$$\begin{cases} F_1 = 2; \\ F_2 = 1; \\ F_i = 2 * F_{i-1} + G_{i-2} \ i \geq 3. \end{cases} \qquad \begin{cases} G_1 = 1; \\ G_2 = 2; \\ G_i = G_{i-1} + 3 * F_{i-2} \ i \geq 3. \end{cases}$$

Podemos então montar a seguinte tabela:

i	1	2	3	4	5	...
F_i	2	1	3	8	24	...
G_i	1	2	8	11	20	...

Este exercício está dividido em três partes.

(a) Só para ver se você entendeu as fórmulas, qual é o valor de F_6 e G_6?

(b) Faça um procedimento de nome *valor* que recebe um inteiro $K \geq 1$ e devolve F_k e G_k.
Exemplo:

> Para $k = 2$, o procedimento deve retornar os valores 1 e 2. Para $k = 3$, a função deve devolver os valores 3 e 8. Para $k = 4$, a função deve devolver os valores 8 e 11.

Observação: não utilize vetores neste exercício.

(c) Faça um algoritmo que lê um inteiro $n > 2$ e imprime os valores $F_{n-2} + G_{n+200}$ e $F_{n+200} + G_{n-2}$. Seu algoritmo deve obrigatoriamente utilizar o procedimento do item anterior.

9. Um conjunto pode ser representado por um vetor da seguinte forma: $V[0]$ é o tamanho do conjunto; $V[1], V[2], \ldots$ são os elementos do conjunto (sem repetições).

(a) Faça uma função *intersecção* que dados dois conjuntos de números inteiros A e B, constrói um terceiro conjunto C que é a intersecção de A e B. Lembre-se de que em $C[0]$ a sua função deve colocar o tamanho da intersecção.

(b) Faça um algoritmo que leia um inteiro $n \geq 2$ e uma seqüência de n conjuntos de números inteiros (cada um com no máximo 100 elementos) e construa e imprima o vetor INTER que representa a intersecção dos n conjuntos.

NOTE que NÃO é preciso ler todos os conjuntos de uma só vez. Você pode ler os dois primeiros conjuntos e calcular a primeira instersecção. Depois, leia o próximo conjunto e calcule uma nova intersecção entre esse conjunto lido e o conjunto da intersecção anterior, e assim por diante.

10. (a) Escreva uma função que recebe como parâmetros um vetor real A com n elementos e um vetor B com m elementos, ambos representando conjuntos, e verifica se A está contido em B (A $\subset$ B).

    (b) Utilizando a função anterior verifique se dois conjuntos são iguais ($A = B$ se e somente se $A \subset B$ e $B \subset A$).

11. (a) Escreva uma função que troca o conteúdo de duas variáveis.

    (b) Escreva uma função que recebe dois inteiros, i e j, uma matriz real $A_{m \times n}$ e troca a linha i pela linha j. Utilize o procedimento do item anterior.

12. Dizemos que uma matriz $A_{n \times n}$ é um *quadrado latino* de ordem n se em cada linha e em cada coluna aparecem todos os inteiros $1, 2, 3, \ldots, n$ (ou seja, cada linha e coluna é permutação dos inteiros $1, 2, \ldots, n$).

    Exemplo:

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

    A matriz acima é um quadrado latino de ordem 4.

    (a) Escreva uma função que recebe como parâmetro um vetor inteiro V com n elementos e verifica se em V ocorrem todos os inteiros de 1 a n.

    (b) Escreva uma função que recebe como parâmetros uma matriz inteira $A_{n \times n}$ e um índice j e verifica se na coluna j de A ocorrem todos os inteiros de 1 a n.

    (c) Utilizando as funções acima, verifique se uma dada matriz inteira $A_{n \times n}$ é um quadrado latino de ordem n.

Capítulo 7

Ponteiros

A memória do computador é constituída de muitas posições (geralmente milhões), cada qual pode armazenar um pedaço de dado. Cada posição de memória é denominada célula. Estas posições são muito similares às variáveis que temos utilizado, com uma diferença: ao escrever nossos algoritmos, criamos nossas variáveis simplesmente dando nome a elas, enquanto que no computador estas células existem fisicamente.

Estas células de memória são numeradas seqüencialmente. Quando o processador do computador precisa acessar uma célula de memória, ele a referencia pelo seu número, que também conhecemos como *endereço*. Quando um algoritmo é escrito em uma linguagem de programação para ser usado em um computador, cada identificador de variável usado no algoritmo é associado com o endereço da célula de memória que será alocada para aquela variável. Nas linguagens de programação modernas, o programador não precisa se preocupar com isto. No ambiente da linguagem de programação é mantida uma lista dos identificadores usados pelo programador e que associa automaticamente estes com os endereços das células de memória.

Um ponteiro é uma célula que armazena o endereço de outra célula de dados. Olhando o valor do ponteiro, o computador determina onde olhar na memória pelo valor do dado apontado pelo ponteiro. De fato, um ponteiro é um tipo especial de variável cujo conteúdo é um endereço de memória. Veja um exemplo na Figura 7.

Figura 7.1: Ponteiro p armazenando o endereço 8900 e sua representação esquemática

Usaremos a seguinte notação para declarar uma variável tipo ponteiro

tipododado ^<identificador>

Este comando declara uma variável de nome identificador que aponta para uma célula armazenando um dado *tipodado*. Por exemplo, na declaração

inteiro $\hat{}p, \hat{}q$

as variáveis p e q são do tipo ponteiro para inteiro, isto é, ambas poderão apontar para uma variável do tipo inteiro. Como a quantidade de células de memória utilizada por um tipo varia conforme o tipo, a variável ponteiro aponta para a primeira posição e pelo tipo ele sabe quantas células a partir daquela posição contêm o dado.

Para podermos entender a base de ponteiros, consideraremos a Figura 7, onde mostramos as variáveis p e q e seu respectivo conteúdos, bem como sua representação esquemática. Nesta figura, mostramos duas variáveis p e q, ambos são variáveis do tipo *ponteiro*, isto é armazenam endereços, e ambas apontam para uma variável inteira que armazena o valor 4.

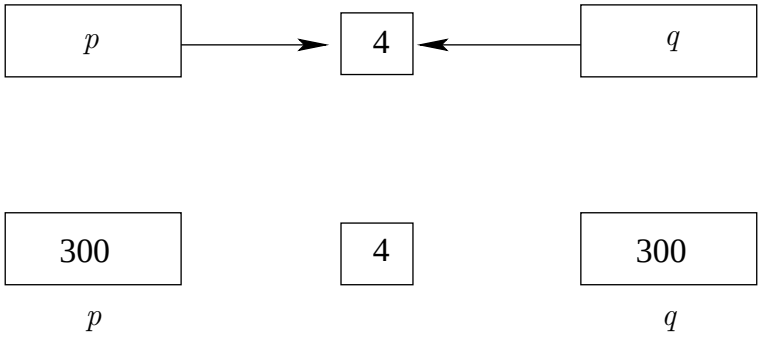

Figura 7.2: A representação esquemática dos ponteiros p e q e seus respectivos conteúdo: o endereço 300

Assim, o algoritmo pode acessar o valor númerico 4 referenciando o valor apontado por p ou referenciando o valor apontado por q. Isto é feito através do operador unário ^. Ao referenciarmos um ponteiro estamos seguindo o ponteiro e vendo para onde ele aponta. Como p é um ponteiro para o tipo inteiro, isto é armazena o valor do endereço de memória onde este valor inteiro se encontra, precisamos de um mecanismo para que ele possa alterar o valor da célula de memória para onde aponta. Assim $\hat{}p \leftarrow 9$ altera o valor da variável para a qual p aponta. Neste caso, o valor da variável apontada por q foi também alterada, uma vez que ambos apontam para o mesmo endereço.

Na Figura 7 mostramos o que ocorre se atribuirmos o valor 9 à variável apontada por p: estaremos mudando também o valor o qual q aponta.

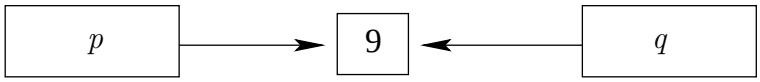

Figura 7.3: Alteração do conteúdo para onde p aponta acarreta mudança no conteúdo para onde q aponta

A variável ponteiro contém um endereço. Para alterar o valor de um ponteiro devemos atribuir um novo valor a ele. Por exemplo, quando mudamos o valor de q para fazê-lo apontar para a mesma célula que r aponta, utilizaremos o seguinte comando:

$$q \leftarrow r$$

Este comando significa que "nós mudamos o valor de q de forma que ele aponta para o endereço de memória para o quel r aponta." Isto tem o mesmo significado de "copie o endereço armazenado em r para q."

A Figura 7.4(a) mostra o que acontecerá se a variável q apontar para a variável inteira apontada por r. O algoritmo poderá obter o valor armazenado nela referenciando tanto r quanto q.

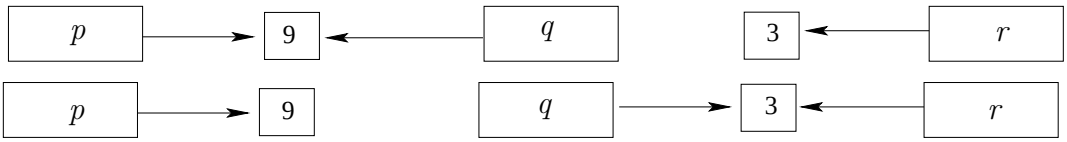

Figura 7.4: Manipulando ponteiros

Se agora quisermos que r aponte para a posição para a qual p aponta, usaremos o comando

$$r \leftarrow p.$$

Devemos notar que q não só referenciou diferentes valores mas também armazenou diferentes células de memória (Figura 7).

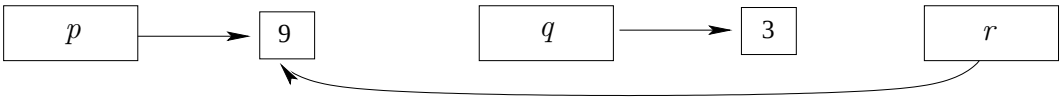

Figura 7.5: Manipulando ponteiros

Se um ponteiro não estiver apontando para nenhuma célula, diremos que seu valor é nulo. O valor nulo não é a mesma coisa que nenhum valor. Ao invés disso, ele é um valor particular que significa que "este ponteiro não está apontando para nada".

Atribuímos o valor nulo para uma variável apontador r através do seguinte comando:

$$r \leftarrow \text{nulo}$$

Na Figura 7 desenhamos um ponteiro nulo como um aterramento (eletricidade).

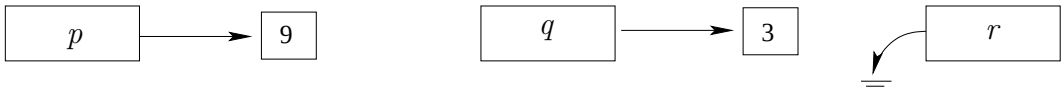

Figura 7.6: O ponteiro r apontando para nulo

Ao referenciarmos um ponteiro ele deve estar apontando para alguma célula. Assim, neste contexto, o seguinte comando é incorreto $\hat{}r \leftarrow 5$ uma vez que r não aponta para nenhuma posição de memória. Para corrigir isto poderíamos fazer

$$r \leftarrow p$$

$\hat{}r \leftarrow 5$ e teríamos p e r apontando para o endereço de memória cujo valor é 5 (Figura 7).

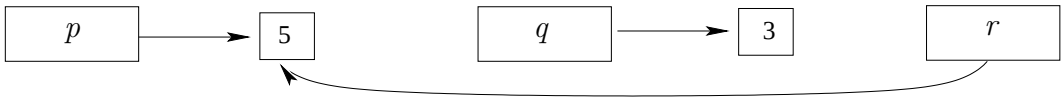

Figura 7.7: Resultado depois dos comandos $r \leftarrow p$ e $\hat{}r \leftarrow 5$

Se quisermos obter o endereço de memória de uma variável devemos utilizar o operador unário &. Assim se tivermos uma variável do tipo inteiro *val*, contendo o valor 7, o resultado do comando

$$p \leftarrow \&val$$

é a atribuição do endereço de variával *val* à p (Figura 7).

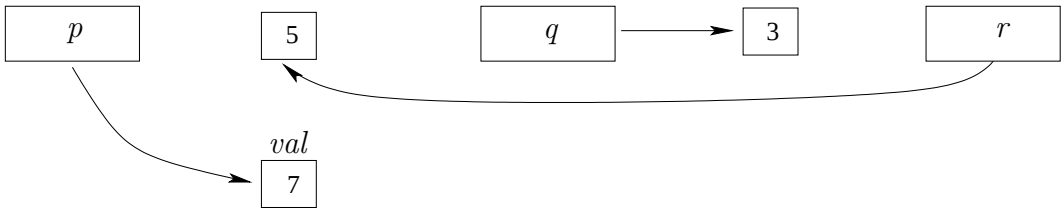

Figura 7.8: Atribuição do endereço de uma variável tipo inteiro para um ponteirp p

7.1 Tipos Base

Até o presente momento, consideramos que cada variável ocupa uma região da memória e não nos preocupamos com a quantidade de memória ocupada por cada um dos tipos. Vimos que um ponteiro aponta para a primeira posição de memória que contém o dado. Para sabermos quantas posições de memória, a partir da inicial, contêm o valor armazenado devemos olhar o tipo base da variável ponteiro.

Cada linguagem de programação define uma quantidade de memória utilizada por cada um de seus tipos básicos. O conceito de ponteiros é implementado em linguaguem de programação considerando esta quantidade de memória, tanto para efeito de alocação quanto para movimentação de ponteiros. Neste texto utilizaremos a seguinte convenção

Tipo	Unidades de Memória
caracter	1
lógico	1
inteiro	2
real	4

O objetivo de fixarmos uma quantidade de memória que os diversos tipos utilizam será visto mais adiante.

7.2 Operações com Ponteiros

Podemos utilizar duas operações com ponteiros: soma e subtração. Devemos lembrar que ponteiros armazenam endereços de memória onde um dado de um determinado tipo base está armazenado. Assim estas operações levam em conta a quantidade de

memória necessária para armazenar o tipo base. Por exemplo, se considerarmos um ponteiro p que aponta para uma variável do tipo inteiro armazenada no endereço 200, a atribuição

$$p \leftarrow p + 1$$

fará com que p aponte para a primeria posição de memória após o inteiro. Como convencionamos que um inteiro utiliza duas unidades de memória, o conteúdo de p será 202 (e não 201!). O mesmo raciocínio vale para a operação de subtração. Se o valor inicial de p for 200, o comando

$$p \leftarrow p - 1$$

fará com que o valor de p seja 198.

Cada vez que o valor de um ponteiro é incrementado ele aponta para o próximo elemento de seu tipo base. Cada vez que um ponteiro é decrementado ela aponta para o elemento anterior. Em ambos os casos leva-se em considerção a quantidade utilizada pelo tipo base. Veja Figura 7.2 como exemplo.

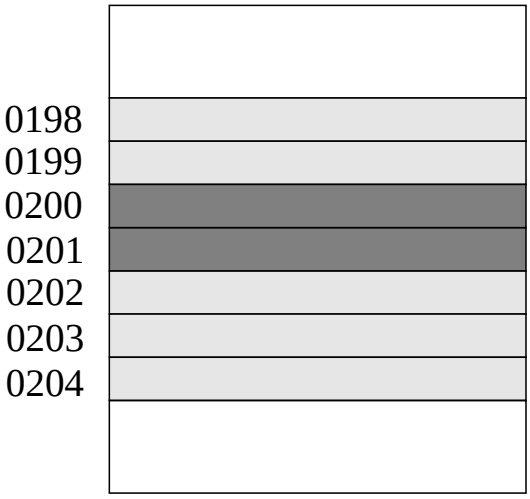

Figura 7.9: Memória ocupada por um inteiro e seus respectivos endereços

As operações aritméticas de ponteiros acima somente podem ser feitas entre ponteiros e inteiros(constante, variáveis ou expressões). Nenhuma outra operação é permitida com ponteiros.

7.3 Ponteiros e Vetores

Nesta seção vamos considerar os ponteiros de maneira similar ao que ocorre na linguagem C. Quando utilizamos somento o nome de um vetor estamos, de fato, referenciando o endereço da primeira posição do vetor. Assim, se considerarmos um vetor A que contém elementos do tipo inteiro e uma variável p do tipo ponteiro para inteiro, a atribuição

$$p \leftarrow A$$

é válida e neste caso p armazenará o endereço da primeira posição do vetor A. Diante disto e considerando as operações aritméticas de ponteiros podemos ver que $p+1$ será um apontador para a próxima posição do vetor. Além disso, se considerarmos que A começa com índice 1, $A[5]$ e $\hat{}(p+4)$ acessam o mesmo elemento, neste caso o quinto elemento do vetor.

Observamos que $\hat{}(p+4)$ é diferente de $\hat{}p+4$, uma vez que o operador $\hat{}$ tem precedência sobre os demais.

Vimos duas formas de acessar os elementos de um vetor: usando a indexação da matriz ou usando a artimética de ponteiros. Surge a questão: qual delas utilizar?

7.4 Alocação Dinâmica de Memória

Relembremos das seções anteriores que, quando fazemos as declarações de variáveis, estamos fazendo alocação estática de memária, ou ainda que estamos alocando memária em tempo de compilação. Quando executamos um programa, a ele é destinado um bloco de memória disposto, em geral, como ilustrado na Figura 7.4.

O código do programa gerado pelo compilador e ligador é carregado na parte mais baixa do bloco de memória. AS variáveis globais e a outras variáveis temporárias criados pelo compilador ocupam uma área de memória imediatamente acima da área de código. O restante do bloco de memória é dividido em duas áreas: pilha e heap. A pilha é usada para armazenar variáveis locais, parâmetros, valores de retorno e outras informações necessárias durante chamadas a módulos realizadas no programa. A pilha e o heap são colocadas em posições opostas na área restante do bloco de memória, e cada uma delas cresce de tamanho em direções opostas (uma em direção aos endereços mais altos e outra em direção aos endereços mais baixos), de tal forma que cada uma delas possa mudar de tamanho sem afetar a outra. O único problema que ocorre é

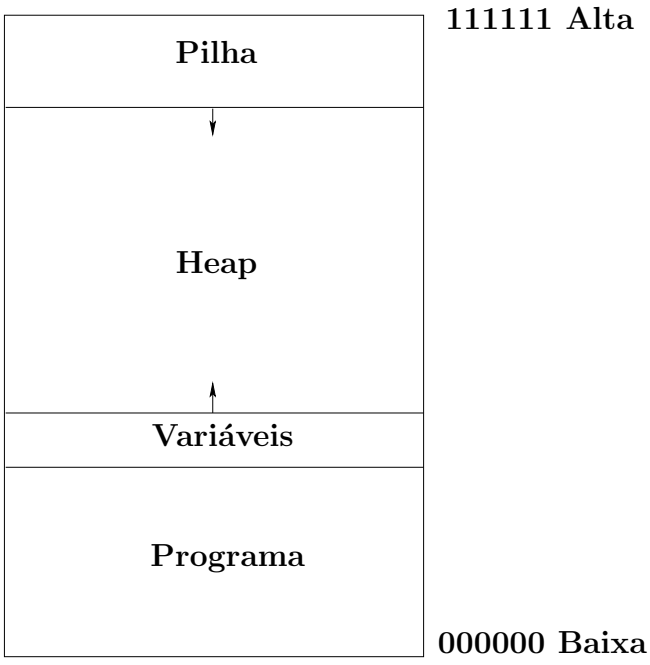

Figura 7.10: Organização da Memória

quando a quantidade de memória livre no bloco de memória não é suficiente para comportar o crescimento da pilha e do heap.

Em muitas situações, a quantidade de memória que um programa necessita não é conhecida em tempo de compilação. Considere o seguinte exemplo bastante simples que ilustra essa situação. Seja S um conjunto de números inteiros. Deseja-se um programa para determinar os elementos de S que são maiores que a média aritmética dos inteiros em S. São dados de entrada do programa: a quantidade de elementos em S e os elementos de S. Para resolver este problema é necessário que os elementos de S sejam armazenados. O problema aqui é que desconhecemos, em tempo de compilação, a quantidade de memória que será necessária para armazenar S (essa informação só será conhecida em tempo de execução). Até então, temos resolvido esse tipo de problema, considerando a existência de um limitante superior para o tamanho de S e então alocando memória grande o suficiente para comportar o maior tamanho esperado para S. Por exemplo, se o limitante para o tamanho de S é 100, usamos um vetor de tamanho 100 para armazenar os elementos de S. Obviamente, se o tamanho de S for menor que 100 o programa teria alocado mais memória do que necessitava.

A alocação dinâmica de memória resolve este problema. Ela consiste no processo de alocar memória em tempo de execução conforme a necessidade do programa. O heap é a memória reservada para ser usada pela alocação dinâmica. Para que a quantidade de memória disponível no heap não sobreponha a área de pilha, é recomendado que

a memória alocada dinamicamente seja liberada pelo programa à medida que não é mais necessária. As seguintes operações são usadas pelo algoritmo para alocação e liberação de memória dinâmica:

- Aloque (p,n)

  p é um ponteiro e n é a quantidade de memória, em unidades relativas ao tipo base de p, a ser alocada dinamicamente. Aloque irá alocar n posições consecutivas de memória, cada uma delas do tamanho necessário para armazenar um valor do tipo base de p, e atribuir a p o endereço inicial da memória alocada. Pode ocorrer falha na alocação, quando a quantidade de memória requerida não pode ser alocada. Nesses casos, Aloque atribui o valor nulo a p. O valor de p sempre deve ser verificado após uma chamada a Aloque, para que haja garantia de que houve sucesso na alocação.

- Libere (p)

  p é um ponteiro. Libere irá liberar a memória anteriormente alocada, cujo endereço inicial está armazenado em p. O espaço de memória desalocado (liberado) torna-se livre para ser usado por futuras chamadas a Aloque.

Desta forma, podemos evitar o desperdício de memória alocando exatamente a quantidade de memória que o programa necessita. Em nosso exemplo anterior, podemos usar um ponteiro para inteiro no lugar do vetor de 100 posições e, então, após ler o tamanho de S efetuar uma chamada a Aloque para alocar a quantidade de memória necessária para armazenar os elementos de S. Essa idéia está ilustrada no seguinte algoritmo:

```
Algoritmo Exemplo
inteiro ^p,^q, n, i, soma, maiores
real media
leia n
Aloque (p,n)
se p = nulo então
   escreva "Não há memória suficiente"
senão
   soma ← 0
   para i de 0 até n − 1 faça
      leia ^(p + i) // o mesmo que p[i]
      soma ← soma+ ^(p + i)
   fim para
   media ← soma/n
   maiores ← 0
```

```
    para i de 0 até n − 1 faça
      se ^(p + i) > media então
        maiores ← maiores + 1
      fim se
    fim para
    Aloque (q,maiores)
    se q ≠ nulo então
      para i de 0 até n − 1 faça
        se ^(p + i) > media então
          q[k] ← ^(p + i)
          k ← k + 1
        fim se
      fim para
    fim se
    para i de 0 até maiores − 1 faça
      escreva ^(q + i)
    fim para
  fim se
  Libere (p)
  Libere (q)
  Fimalgoritmo
```

7.5 Ponteiros para Ponteiros

Em algumas situações é necessário conhecer e manipular o endereço de uma variável ponteiro. Um exemplo, é quando o sistema operacional precisa da capacidade de mover blocos de memória na pilha. Um ponteiro para ponteiro é como se você anotasse em sua agenda o número do telefone de uma empresa que fornecesse número de telefones residenciais. Quando você precisar do telefone de uma pessoa você consulta sua agenda, depois consulta a empresa e ela te fornece o número desejado. Entender ponteiro para ponteiro requer conhecimento de ponteiro. Chama-se nível de indireção ao número de vezes que temos que percorrer para obter o conteúdo da variável final. As variáveis do tipo ponteiro têm nível de indireção um. As variáveis do tipo ponteiro para ponteiro têm nível de indireção dois. Se tivermos uma variável que é um ponteiro para ponteiro para inteiro, para obtermos o valor do inteiro precisamos fazer dois níveis de indireções.

Podemos declarar um ponteiro para um ponteiro com a seguinte notação:

tipodado ^^*identificador*

Figura 7.11: Exemplo onde p é um ponteiro para ponteiro

onde: ^^ identificador é o conteúdo da variável apontada e ^identificador é o conteúdo ponteiro intermediário.

Algoritmo Exemplo 2
inteiro a, b, $\hat{}p$, $\hat{}\hat{}pp$
$a \leftarrow 2$
$b \leftarrow 3$
$p \leftarrow \&a$ // p recebe endereço de a
$pp \leftarrow \&p$ // pp aponta para o ponteiro p ou seja "a"
$\hat{}\hat{}pp \leftarrow 5$ // "a"recebe valor 5 (duas indireções)
$\hat{}pp \leftarrow \&b$ // pp aponta para b
$\hat{}\hat{}pp \leftarrow 6$ // "b"recebe valor 6
Fimalgoritmo

Ressaltamos que ponteiro para inteiro e ponteiro para ponteiro para inteiro tem níveis de indireções diferentes. Em nosso, exemplo, o conteúdo de p é um endereço de um inteiro e o conteúdo de pp é um endereço de um ponteiro para inteiro. Em outras palavras são tipos diferentes. Assim um comando da forma $pp \leftarrow p$ não é válido.

Algoritmo Exemplo 3
inteiro $\hat{}\hat{}p$, $\hat{}q$
Aloque(p, 1);
Aloque($\hat{}p$, 1)
$\hat{}\hat{}$p$\leftarrow$ 10
$q \leftarrow \hat{}p$
escreva $\hat{}q$
Libere(q)
Libere(p)
Fimalgoritmo

Um fato interessante é que o número de indireções é, teoricamente, indeterminado e depende da linguagem de programação o número de indireções permitidas, no entanto a lógica permanece a mesma. Assim podemos fazer a seguinte declaração

inteiro $\hat{}\hat{}\hat{}\hat{}p$

Que significa que o ponteiro para ponteiro p possui quatro indireções.

7.6 Ponteiros para Registro

Em algumas situações é necessário uma variável ponteiro apontar para um tipo definido pelo usuário. Um exemplo é um ponteiro para regristro. De fato, pode-se criar um ponteiro para qualquer tipo, incluindo tipos definidos pelo usuário. É extremamente comum criar ponteiros para estas estruturas. Um trecho de algoritmo é dado abaixo como exemplo:

definatipo registro
    cadeia *nome*
    inteiro *CPF*
    real *salario*
fimregistro *regficha*
regficha ^*r*
Aloque($r, 1$)

O pointeiro r é um ponteiro para registro. Observe que, como r é um ponteiro, a instrução Aloque aloca memória do heap suficiente para todos os campos. O conteúdo ára onde r aponta, $\hat{}r$, é uma estrutura como qualquer estrutura de tipo regficha. A codificação seguinte mostra os usos freqüentes de ponteiro para registro:

$(\hat{}r).nome \leftarrow$ "João Ninguem"
$(\hat{}r).CPF \leftarrow 12345678900$
$(\hat{}r).salario \leftarrow 480{,}01$
escreva $(\hat{}r).nome$
Libera(r)

Podemos opera com $\hat{}r$ da mesma forma como operamos com uma variável estática de registro. O uso de parênteses é importante para evitar ambiguidade em alguns casos. Uma forma igualmente válida e mais intuitiva de manipular ponteiro de regisro é dada abaixo e faz exatamente a mesma coisa que o trecho acima:

$r{->}nome \leftarrow$ "João Ninguem"
$r{->}CPF \leftarrow 12345678900$
$r{->}salario \leftarrow 480,01$
escreva $r{->}nome$
Libera(r)

Capítulo 8

Algoritmos de Ordenação

Um dos problemas mais tradicionais da área de computação é o problema da **ordenação**. Nesse problema, dada uma seqüência de números, precisamos colocá-la em uma certa ordem (numérica ou lexicográfica). Na prática, os números a serem ordenados raramente são valores isolados. Em geral, cada um deles faz parte de uma coleção de dados (**registro**). Cada registro contém uma chave, que é o valor a ser ordenado, e o restante do registro consiste em dados satélites. Assim, na prática, quando permutamos os números, também devemos permutar os dados satélites. Por considerarmos a permutação dos dados satélites uma operação mecânica, suporemos que a entrada do problema consiste somente de números.

Definindo o problema da ordenação formalmente, temos: dada uma seqüência de n números $\langle a_1, a_2, \ldots, a_n \rangle$, encontrar uma permutação $\langle a'_1, a'_2, \ldots, a'_n \rangle$ dessa seqüência tal que $a'_1 \leq a'_2 \leq \ldots \leq a'_n$.

A seguir descrevemos alguns dos algoritmos simples de ordenação. Nessa descrição, assumiremos que já existe um tipo denominado *vet_num* que define um vetor de inteiros cujo limite inferior é 1 e limite superior é 100. Em outras palavras, omitiremos nos algoritmos a seguir uma linha do tipo

definatipo vetor[1..100] de inteiro *vet_num*

8.1 *Bubble-sort*

O algoritmo *bubble-sort* consiste em passar seqüencialmente várias vezes pelos elementos $a_1, a_2, \ldots, a_n$ da seqüência de tal forma que, a cada passo, o elemento a_j seja

comparado com o elemento a_{j+1}. Se $a_j > a_{j+1}$, então eles são trocados. O algoritmo funciona como uma bolha que coloca nesta bolha o maior elemento entre dois adjacentes e sobe até a última posição do vetor, daí o seu nome.

```
Algoritmo bubble_sort
inteiro i, j, n, temp
vet_num a
leia n
para i de 1 até n faça
  leia a[i]
fim para
para i de 1 até n − 1 faça
  para j de 1 até n − 1 faça
    se a[j] > a[j + 1] então
      temp ← a[j]
      a[j] ← a[j + 1]
      a[j + 1] ← temp
    fim se
  fim para
fim para
Fimalgoritmo
```

No exemplo abaixo vemos o funcionamento do laço interno do algoritmo para um vetor com 6 elementos posicionando o maior elemento no final do vetor.

5	6	2	1	4	3
5	**6**	2	1	4	3
5	2	**6**	1	4	3
5	2	1	**6**	4	3
5	2	1	4	**6**	3
5	2	1	4	3	**6**

E abaixo as demais rodadas do laço externo algoritmo que completam a ordenação.

5	2	1	4	3	6
2	1	4	3	**5**	6
1	2	3	**4**	5	6
1	2	**3**	4	5	6
1	**2**	3	4	5	6

Observe que no algoritmo acima, em cada rodada do laço interno um elemento encontra sua posição correta na sequencia ordenada. Na primeira rodada é o maior elemento que encontra sua posição, na segunda rodada é o segundo maior e assim sucessivamente. Assim, como cada vez um elemento é colocado na sua posição correta do final para o início do vetor, o algoritmo pode ser melhorado reduzindo o número de rodadas do laço interno conforme o código abaixo.

```
Algoritmo bubble_sort
inteiro i, j, n, temp
vet_num a
leia n
para i de 1 até n faça
  leia a[i]
fim para
para i de 1 até n − 1 faça
  para j de 1 até n − i faça
    se a[j] > a[j + 1] então
      temp ← a[j]
      a[j] ← a[j + 1]
      a[j + 1] ← temp
    fim se
  fim para
fim para
Fimalgoritmo
```

8.2 *Insertion-sort*

Para compreensão do algoritmo *Insertion-sort*, imagine a seguinte situação: você tem um conjunto de cartas em sua mão esquerda e com a mão direita tenta ordenar as cartas. A idéia é colocar cada carta, da esquerda para a direita, em sua posição correta, fazendo inserções. Neste algoritmo, a cada passo k, o elemento a_k é inserido em sua posição correta.

Abaixo ilustramos o funcionamento do algoritmo para o mesmo vetor com 6 ele-

mentos do algoritmo anterior.

5	**6**	2	1	4	3
5	6	**2**	1	4	3
5	6	6	1	4	3
5	5	6	1	4	3

2	5	6	**1**	4	3
2	5	6	6	4	3
2	5	5	6	4	3
2	2	6	6	4	3
1	2	5	6	**4**	3
1	2	5	6	6	3
1	2	5	5	6	3
1	2	4	5	6	3
1	2	4	5	6	**3**
1	2	4	5	6	6
1	2	4	5	5	6
1	2	4	4	5	6
1	2	3	4	5	6

A seguir detalhamos o código do algoritmo.

```
Algoritmo insertion_sort
inteiro i, j, c, n
vet_num a
leia n
para i de 1 até n faça
   leia a[i]
fim para
para i de 2 até n faça
   c ← a[i]
   j ← i − 1
   enquanto j > 0 E a[j] > c faça
      a[j + 1] ← a[j]
      j ← j − 1
```

```
    fim enquanto
    a[j + 1] ← c
  fim para
  Fimalgoritmo
```

8.3 *Selection-sort*

O método de ordenação por seleção tem o seguinte princípio de funcionamento: pegue o maior elemento da seqüência e troque-o com o elemento que está na última posição. A seguir, repita essa operação para os $n - 1$ elementos. Ou seja, encontre o maior elemento entre eles e coloque-o na penúltima posição. Depois, repita a operação para os $n - 2$ elementos e assim sucessivamente. Ao final da execução, a seqüência estará em ordem não-decrescente.

```
Algoritmo selection_sort
inteiro imax, max, i, j
vet_num a
leia n
para i de 1 até n faça
  leia a[i]
fim para
para i de n até 2 passo −1 faça
  max ← a[1]
  imax ← 1
  para j de 2 até i faça
    se a[j] > max então
      max ← a[j]
      imax ← j
    fim se
  fim para
  a[imax] ← a[i]
  a[i] ← max
fim para
Fimalgoritmo
```

Ilustramos abaixo o funcionamento deste algoritmo.

$$
\begin{array}{cccccc}
5 & \mathbf{6} & 2 & 1 & 4 & \mathbf{3} \\
5 & 3 & 2 & 1 & 4 & 6 \\
\mathbf{5} & 3 & 2 & 1 & \mathbf{4} & 6 \\
4 & 3 & 2 & 1 & 5 & 6 \\
\mathbf{4} & 3 & 2 & \mathbf{1} & 5 & 6 \\
1 & 3 & 2 & 4 & 5 & 6
\end{array}
$$

$$
\begin{array}{cccccc}
1 & \mathbf{3} & \mathbf{2} & 4 & 5 & 6 \\
1 & 2 & 3 & 4 & 5 & 6 \\
1 & \mathbf{2} & 3 & 4 & 5 & 6 \\
1 & 2 & 3 & 4 & 5 & 6
\end{array}
$$

Capítulo 9

Um pouco sobre complexidade de algoritmos

Até agora, estivemos interessados apenas no quesito corretude de um algoritmo. Ou seja, dado um problema específico, focalizamos as nossas atenções apenas no desenvolvimento e estudo de algoritmos que resolvesse o problema, sem nos preocuparmos com a sua eficiência.

Para podermos classificar um algoritmo como eficiente ou não, precisamos definir precisamente um critério para medirmos a sua eficiência. Esse critério é o seu tempo de execução, que pode ser medido por meio da implementação e execução do algoritmo utilizando-se várias entradas distintas. Apesar de válida, essa estratégia experimental apresenta vários problemas. O principal deles é o fato dela ser dependente da máquina (*software* e *hardware*) em que o algoritmo está sendo executado. Precisamos então encontrar uma forma de medir o tempo de execução que nos permita comparar dois algoritmos independentemente da máquina onde estão sendo executados. Uma forma de se fazer isso é contando o número de operações primitivas do algoritmo. Dentre as operações primitivas estão: atribuir um valor a uma variável, executar uma operação aritmética, chamar um módulo, comparar dois números, avaliar uma expressão e retornar de uma função.

Em um ambiente computacional, cada operação primitiva corresponde a uma ou mais instruções de baixo-nível, as quais possuem tempo de execução que depende da máquina, embora seja constante. Desde que o número de instruções de baixo nível correspondentes a cada operação primitiva é constante, e desde que cada operação elementar possui tempo de execução constante, os tempos de execução de duas operações primitivas quaisquer diferem entre si por um fator constante. Isso nos permite considerar o tempo de execução de cada operação primitiva como sendo o mesmo, o que

implica podermos nos concentrar apenas na quantidade e freqüência de tais operações.

Tomemos então como exemplo o seguinte algoritmo que lê um vetor de inteiro a, com $n \geq 1$ posições, um inteiro x e procura por x dentro de A.

```
1: Algoritmo busca
2: definatipo vetor[1..10] de inteiro vet_num
3: inteiro i, j, x, n
4: lógico achou
5: vet_num a
6: achou ← F
7: i ← 1
8: leia n, x
9: para j de 1 até n faça
10:    leia a[j]
11: fim para
12: enquanto achou = F E i ≤ n faça
13:    se x = a[i] então
14:       achou ← V
15:    senão
16:       i ← i + 1
17:    fim se
18: fim enquanto
19: se achou = V então
20:    escreva "O elemento de valor ", x, "está na posição ", i, "do vetor."
21: senão
22:    escreva "O elemento de valor ", x, "não se encontra no vetor"
23: fim se
24: Fimalgoritmo
```

Nesse algoritmo, as linhas 6 e 7 contêm uma operação primitiva (atribuição do valor "falso" à variável *achou* e do valor 1 à variável i).

A linha 9 é um laço para..faça que se repete exatamente n vezes. Na primeira iteração desse laço, temos uma operação de atribuição ($j \leftarrow 1$), que corresponde a uma operação primitiva. No início de cada uma das iterações desse laço, a condição $j \leq n$, que também corresponde a uma operação primitiva, é avaliada uma vez. Como essa avaliação é feita $n + 1$ vezes, a condição $j \leq n$ contribui com $n + 1$ operações primitivas. Ao final de cada iteração, temos duas operações primitivas: uma soma $(j + 1)$ e uma atribuição ($j \leftarrow j + 1$). Como essas duas operações estão dentro do laço, elas se repetem n vezes. Uma outra operação que se encontra dentro do laço é a indexação ($a[i]$) necessária durante a leitura do vetor. Essa indexação corresponde

a uma soma que se repete n vezes. Com tudo isso, o laço em questão corresponde a $4n + 2$ operações primitivas.

A linha 12 determina o início de um laço que será repetido, no mínimo, uma vez e, no máximo, n vezes. No início de cada iteração desse laço, a condição $achou = \underline{F}$ $\underline{\text{E}}$ $i < n$ é avaliada, no mínimo, duas vezes e, no máximo, $n + 1$ vezes. A condição do laço contém três operações primitivas: a comparação $i < n$, a avaliação da expressão "$achou = \underline{F}$" e a verificação do resultado da expressão "$achou = \underline{F}$ $\underline{\text{E}}$ $i < n$". Logo, a linha 12 corresponde a, no mínimo, 6 e, no máximo, $3 \times (n + 1)$ operações primitivas.

A linha 13 contém duas operações primitivas: uma indexação ($a[i]$) e uma comparação ($x = a[i]$). Como essa linha está no corpo do laço, ela se repetirá, no mínimo, um vez e, no máximo, n vezes. Logo, a linha 13 corresponde a, no mínimo duas e, no máximo $2n$ operações primitivas.

A linha 14 ($achou \leftarrow \underline{V}$) contém uma operação primitiva, que será executada, no máximo, uma vez. A linha 16 ($i = i + 1$) contém duas operações primitivas: a adição em si e a atribuição do valor $i + 1$ à variável i. Além disso, ela é executada no máximo n vezes, podendo não ser executada nenhuma vez. Finalmente, a linha 19 $\underline{\text{se}}$ ... $\underline{\text{então}}$ contém uma operação primitiva: a verificação da condição da estrutura condicional. Ela será executada uma única vez.

Pelas observações acima podemos notar que o tempo de execução do algoritmo *busca* depende de n, o tamanho do vetor a ser processado. Iremos então representá-lo como uma função $f(n)$. Note que, se fixarmos n, $f(n)$ atingirá seu menor valor quando x estiver na primeira posição do vetor (ou seja, quando $A[1] = x$), e seu maior valor quando x não estiver em A ($x \notin A$). Isto é, no melhor caso, temos que $t(n) = 1 + 1 + 4n + 6 + 2 + 1 + 1 = 4n + 12$. No pior caso temos que $t(n) = 1 + 1 + 4n + 2 + 3(n + 1) + 2n + 2n + 1 = 11n + 8$. Portanto, *busca* executa de $4n + 12$ a $11n + 8$ operações primitivas em uma entrada de tamanho $n \geq 1$.

Nós usualmente consideramos o pior caso de tempo de execução para fins de comparação da eficiência de dois ou mais algoritmos. Uma análise de pior caso, em geral, ajudá-nos a identificar o "gargalo" de um algoritmo e pode nos dar pistas para melhorar a sua eficiência.

A análise de tempo de execução vista até aqui é conhecida como análise de complexidade de tempo, ou simplesmente, análise de complexidade.

Considere O problema de encontrar o maior elemento de um vetor de inteiros $v[1..n]$, $n \geq 1$. Um algoritmo para este problema é dado a seguir. Neste caso a função

Max devolve o valor desejado.

```
1: inteiro Função Max(vet_num v, inteiro n)
2: inteiro i, aux
3: aux ← v[1]
4: para i de 2 até n faça
5:    se aux < v[i] então
6:       aux ← v[i]
7:    fim se
8: fim para
9: devolva aux
10: Fimfunção
```

Muitas vezes fazemos o cálculo da complexidade olhando apenas as operações primitivas mais relevantes. No nosso algoritmo esta operação é a comparação. Assim a complexidade de tempo da função Max é $T(n) = n - 1$. Podemos mostrar que nosso algoritmo é ótimo.

Teorema 9.1 *Qualquer algoritmo para encontrar o maior elemento em um vetor com n elementos, $n \geq 1$, faz no mímino $n - 1$ comparações.*

Prova: Cada um dos $n - 1$ elementos deve ser comparado com algum outro elemento a fim de se concluir que ele não é o maior. Portanto $n - 1$ comparações são necessárias.

Com este teorema mostramos que qualquer outro algoritmo para o problema vai gastar pelo menos tantas comparações com o algoritmo Max. Podemos concluir então que ele é ótimo.

Vamos considerar agora o problema de encontrar o maior e o menor elemento de um vetor de inteiros $v[1..n]$, $n \geq 1$. Um algoritmo para este problema é dado a seguir. Neste caso a função MinMax1 devolve o valor desejado.

```
1: Procedimento MinMax(vet_num v, inteiro n, ref inteiro min, max)
2: inteiro i
3: min ← v[1]
4: max ← v[1]
5: para i de 2 até n faça
6:    se max < v[i] então
7:       max ← v[i]
8:    senão
```

```
 9:       se min > v[i] então
10:          min ← v[i]
11:       fim se
12:    fim se
13: fim para
14: Fimprodecimento
```

Melhor caso: $T(n) = n - 1$, quando o vetor está em ordem crescente.

Pior caso: $T(n) = 2(n - 1)$, quando o vetor está em ordem decrescente.

Caso Médio:$T(n) = 3n/2 - 3/2$ quando $v[i]$ é maior que max a metade das vezes.

Logo,

$$T(n) = n - 1 + \frac{n-1}{2} = \frac{3n}{2} - 3/2$$

Este algoritmo pode ser melhorado usando a seguinte idéia. Compare os elementos aos pares, separando-os em dois subconjuntos, colocando os maiores em um subconjunto e os menores em outros. O máximo é obtido do subconjunto dos maiores e o mínimo é obtido do subconjunto dos menores. Quando n for ímpar o elemento $v[n]$ é duplicado em $v[n + 1]$, por conveniência. O código deste algoritmo é dado abaixo:

```
 1: Procedimento MinMax2(vet_num v,inteiro n, ref inteiro min, max)
 2: inteiro i, fim
 3: se n  mod 2 > 0 então
 4:    v[n+1] ← v[n]
 5:    fim ← n
 6: senão
 7:    fim ← n − 1
 8: fim se
 9: se v[1] > v[2] então
10:    max ← v[1]
11:    min ← v[2]
12: senão
13:    max ← v[2]
14:    min ← v[1]
15: fim se
16: para i de 3 até fim Passo 2 faça
17:    se v[i] > v[i + 1] então
18:       se v[i] > max então
```

```
19:          max ← v[i]
20:       fim se
21:       se v[i + 1] < min então
22:          min ← v[i + 1]
23:       fim se
24:    senão
25:       se v[i] < min então
26:          min ← v[i]
27:       fim se
28:       se v[i + 1] > max então
29:          max ← v[i + 1]
30:       fim se
31:    fim se
32: fim para
33: Fimprodecimento
```

A complexidade de tempo (número de comparações) deste algoritmo é dada por:

$$T(n) = \frac{n}{2} + \frac{n-2}{2} + \frac{n-2}{2} = \frac{3n}{2} - 2$$

Tanto para o pior caso, melhor caso e caso médio.

Compare este algoritmo com o teorema abaixo.

Teorema 9.2 *Qualquer algoritmo para encontrar o maior e o menor elemento de um vetor com n elementos, faz no mínimo* $\lceil 3n/2 \rceil - 2$ *comparações.*

Na tabela abaixo veja a comparação entre os dois procedimento para encontrar o mínimo e o máximo.

	Melhor caso	Pior caso	Caso Médio
minmax	$n-1$	$2(n-1)$	$3n/2-2$
minmax2	$3n/2-2$	$3n/2-2$	$3n/2-2$

Exercícios

1. Descreva um procedimento $somamatriz(A, B, C, n)$ que recebe duas matrizes A e B, soma e coloca o resultado na matriz C. Calcule a quantidade de operações primitivas que o procedimento realiza.

2. Descreva um procedimento $multimatriz(A, B, C, n)$ que recebe duas matrizes A e B, multiplica e coloca o resultado na matriz C. Calcule a quantidade de operações primitivas que o procedimento realiza.

3. Considerando os dois procedimentos $somamatriz(A, B, C, n)$ e $multimatriz(A, B, C, n)$ dos exercícos anteriores, calcule a complexidade de cada um considerando como operações mais relevantes apenas as somas no primeiro procedimento e as multiplicações no segundo.

4. O que é um algoritmo ótimo? Dê um exemplo.

5. Descreva um algoritmo ótimo para encontrar o maior e o menor elemento de um vetor com n elementos.

6. Escreva a função $fatorial(i)$, $i \in N$, e calcule sua complexidade.

7. Escreva uma função para encontrar o segundo maior elemento de de um vetor com n elementos. Calcule sua complexidade.

Capítulo 10

Listas

No capítulo 5, onde falamos sobre vetores, definimos *e*struturas de dados como a forma com que os valores componentes de um tipo complexo (estruturado) se organizam as relações entre eles. Vetores (unidimensionais e bidimensionais) são considerados estruturas de dados primitivas. Neste capítulo estudaremos uma estrutura de dados não primitiva, que recebe o nome de *l*ista linear.

10.1 Definições básicas

Uma lista linear agrupa informações referentes a um conjunto de elementos que, de alguma forma, se relacionam entre si. Ela pode se constituir, por exemplo, de informações sobre funcionários de uma empresa, notas de compras, itens de estoque, notas de alunos, etc.

Definindo formalmente, uma lista linear é um conjunto de $n > 0$ elementos (nós) $L_1, L_2, \ldots, L_n$ com as seguintes propriedades estruturais, que decorrem unicamente das posições relativas de seus nós:

1. L_1 é o primeiro nó;
2. para $1 < k \leq n$, o nó L_k é precedido pelo nó L_{k-1};
3. L_n é o último nó.

Em nossos estudos assumiremos que cada nó é formado por vários campos, que armazenam as características dos elementos da lista. Um desses campos constitui o

que chamamos de *c*have do nó. O conceito de campo chave é de extrema importância pois ele constitui o principal objeto a ser manipulado pelos algoritmos sobre listas. Para se evitar ambigüidades, iremos supor que todas as chaves são distintas.

As operações mais freqüentes em listas são a *b*usca, *i*nserção de um nó, e *r*emoção de um nó da lista. Além dessas operações vale mencionar outras de relativa importância como a combinação de duas ou mais listas lineares em uma única, a ordenação dos nós, a determinação do tamanho da lista, etc.

Podemos citar casos particulares de listas com base nas operações de inserção e remoção. Se as inserções e remoções são relizadas somente em um extremo, a lista é chamada *p*ilha. Se as inserções são relizadas em um extremo, e as remoções em outro, a lista é chamada *f*ila.

Existem duas formas de armazenar os nós de uma lista no computador:

- alocação seqüencial: os nós são armazendos em posições consecutivas da memória;
- alocação encadeada: os nós são armazendos em posições não consecutivas da memória.

A melhor maneira de armazenar listas lineares depende, basicamente, das operações mais freqüentes a serem executadas sobre ela. Não existe, em geral, uma única representação que atende, de maneira eficiente, todas as operações acima mencionadas.

10.2 Alocação seqüencial

A maneira mais simples de manter uma lista no computador é alocando posições consecutivas da memória a cada um de seus nós (vetor). Assim, poderíamos definir uma lista L, com no máximo 1000 nós de um certo tipo, utilizando os seguintes comandos de nossa linguagem algorítmica:

defina MAX 1000
definatipo registro
inteiro *chave*
tipo$_1$ *campo*$_1$
tipo$_2$ *campo*$_2$
...
tipo$_m$ *campo*$_m$

fimregistro *tipo_registro*
definatipo vetor[1..MAX] de *tipo_registro lista*
lista L

Note nas linhas acima que tipo_registo é um tipo que define um registro que possui um número qualquer de campos, sendo um deles o campo *chave*. Em nosso exemplo, ele está definido como inteiro, mas pode ser de qualquer tipo elementar conhecido.

Armazenada de forma seqüencial, a seguinte função pode ser utilizada para a busca, por meio da chave, de um elemento na lista L. Ela recebe como parâmetro uma variável do tipo *lista* (um vetor), a chave x e o tamanho n da lista. Ela devolve o índice na lista onde o elemento se encontra ou -1 caso o elemento não esteja na lista.

```
função inteiro Busca_Lista_Seq(lista L, inteiro x, inteiro n)
inteiro i
i ← 1
enquanto L[i].chave ≠ x E i ≤ n faça
   i ← i + 1
fim enquanto
se i ≠ n + 1 então
   retorne i
senão
   retorne -1
fim se
fimfunção
```

A função acima executa, no mínimo, 8 (quando o elemento encontra-se na primeira posição do vetor) e, no máximo 1 + 3(n+1) + 2n + 2 + 1 = 5n + 6 operações primitivas.

Sobre os procedimentos de inserção (insere um novo registro no final da lista) e remoção (remove um registro da lista cuja chave é x) em uma lista linear alocada seqüencialmente, eles encontram-se detalhadas abaixo. Note que ambos se utilizam da função Busca_Lista_Seq. A função de inserção insere um novo resgitra

```
procedimento Inserção(lista L, tipo_registro novo, ref inteiro n)
se n < MAX então
   se BUSCA_Lista_Seq(L, novo.chave) = -1 então
      L[n + 1] ← novo
      n ← n + 1
   senão
      escreva "Elemento já existe na lista"
```

```
    fim se
  senão
    imprima "lista cheia (overflow)"
  fim se
  fimprocedimento
```

O procedimento acima executa, no mínimo, 1 operação primitiva (quando a lista está cheia) e, no máximo $1 + 2 + 2 = 5$ operações primitivas (note que o número de operações primtivas realizadas por esse procedimento não depende de n).

```
procedimento Remoção(lista L, inteiro x, ref inteiro n)
inteiro k
tipo_registro valor
se n > 0 então
  k ← BUSCA_Lista_Seq(L, x)
  se k ≠ −1 então
    valor ← L[k]
    para i de k até n − 1 faça
      L_k ← L_k + 1
    fim para
    n ← n − 1
  senão
    imprima "Elemento não se encontra na lista"
  fim se
senão
  imprima "underflow"
fim se
fimprocedimento
```

A função acima executa, no mínimo, 1 operação primitiva (quando a lista está vazia) e, no máximo 1 + 2 + 1 + 1 + 1 + n + 2(n-1) + 2(n-1) + 1 = 5n + 3 operações primitivas.

10.2.1 Pilhas e Filas

Em geral, o armazenamento seqüencial de listas é empregado quando as estruturas sofrem poucas inserções e remoções ao longo do tempo, ou quando inserções e remoções não acarretam movimentação dos nós. Esse último caso ocorre quando os elementos a serem inserido e removidos estão em posiçõe especiais da lista, como a primeira ou a última. Isso nos leva à definição de pilhas e filas.

Pilhas

Uma pilha é uma lista linear tal que as operações de inserção e remoção são realizadas em um único extremo.

Em geral, a implementação de uma pilha se faz por meio de um registro P contendo dois campos: $P.topo$, um inteiro indicando a quantidade de elementos na pilha, e um vetor $P.L$, de tamanho MAX, contendo os $P.topo$ elementos. Assim, poderíamos definir uma pilha, com no máximo 1000 nós de um certo tipo, utilizando os seguintes comandos de nossa linguagem algorítmica:

```
defina MAX 1000
definatipo registro
inteiro chave
tipo_1 campo_1
tipo_2 campo_2
...
tipo_m campo_m
fimregistro tipo_registro
definatipo vetor[1..MAX] de tipo_registro lista
definatipo registro
tipo_1 campo_1
lista L
fimregistro tipo_pilha
tipo_pilha P
```

Os seguintes procedimento podem então ser utilizados para inserir e remover elementos de uma pilha, respectivamente. Quando falamos nessa estrutura de dados, é comum nos referenciarmos às operações de inserção e remoção de elementos como *e*mpilhar e *d*esempilhar

```
procedimento Empilha(ref tipo_pilha P, tipo_registro novo)
se P.topo ≠ MAX então
   P.topo ← P.topo + 1
   P.L[P.topo] ← novo
senão
   imprima "overflow"
fim se
fimprocedimento
```

O procedimento acima executa, no mínimo, 1 operação primitiva (quando a pilha está cheia) e, no máximo, 5 operações primitivas.

```
função tipo_registro Desempilha(ref tipo_pilha P, removido)
tipo_registro removido
se P.topo ≠ 0 então
   removido ← P.L[P.topo]
   P.topo ← P.topo − 1
senão
   imprima "underflow"
fim se
fimfunção
```

A função acima executa, no mínimo, 1 operação primitiva (quando a pilha está vazia) e, no máximo, 5 operações primitivas.

Filas

Uma fila é uma lista linear tal que as operações de inserção são realizadas em um extremo e as de remoção são realizadas no outro extremo.

Em geral, a implementação de uma fila se faz por meio de um registro F contendo três campos: $F.inicio$, um inteiro indicando o início da fila, $F.fim$, um inteiro indicando o fim da fila, e um vetor $F.L$, de tamanho MAX, contendo os $F.inicio-F.fim+1$ elementos da fila. Assim, poderíamos definir uma fila, com no máximo 1000 nós de um certo tipo, utilizando os seguintes comandos de nossa linguagem algorítmica:

```
defina MAX 1000
definatipo registro
inteiro chave
tipo_1 campo_1
tipo_2 campo_2
...
tipo_m campo_m
fimregistro tipo_registro
definatipo vetor[1..MAX] de tipo_registro lista
definatipo registro
inteiro inicio
tipo_1 fim
lista L
fimregistro tipo_fila
tipo_fila F
```

Os seguintes procedimento podem ser utilizados para inserir (enfileirar) e remover (desinfileirar) elementos de uma fila.

Note que, após qualquer operação, deve-se sempre ter o ponteiro *inicio* indicando o início da fila e fim o final da fila. Isso implica incrementar *inicio* quando de uma inserção e fim quando de uma remoção da fila. Isso faz com que a fila se "mova", dando a falsa impressão de memória esgotada. Para eliminar esse problema, consideram-se os n nós alocados como se eles estivessem em círculo, com $F.L[1]$ seguindo $F.L[n]$. No algoritmo de inserção em uma fila, a variável auxiliar *prov* armazena provisoriamente a posição de memória calculada de forma a respeitar a circularidade, sendo que o indíce fim é movimentado somente se a posição for possível. Os índices *inicio* e fim são ambos inicializados com zero.

```
procedimento Enfileira(ref tipo_fila F, tipo_registro novo)
prov ← (F.fim MOD MAX) + 1
se prov ≠ F.inicio então
   F.fim ← prov
   F.L[F.fim] ← novo
   se F.inicio = 0 então
      F.inicio = 1
   fim se
senão
   imprima "overflow"
fim se
fimprocedimento
```

O procedimento acima executa, no mínimo, 4 operação primitiva (quando a fila está cheia) e, no máximo, 9 operações primitivas (quando o primeiro elemento está sendo inserido).

```
função tipo_registro Desenfileira(ref fila F)
tipo_registro recuperado
se F.inicio ≠ 0 então
   recuperado ← F.L[F.inicio]
   se F.inicio = F.fim então
      F.inicio ← 0
      F.fim ← 0
   senão
      F.inicio ← (F.inicio MOD MAX) + 1
   fim se
   retorne recuperado
senão
   imprima "overflow"
```

```
fim se
fimfunção
```

A função acima executa, no mínimo, 1 operação primitiva (quando a fila está vazia) e, no máximo, 7 operações primitivas (quando o único elemento está sendo removido).

10.2.2 Um exemplo de aplicação de uma pilha

Dentre as várias aplicações de uma pilha, existe uma de grande interesse na área de compiladores. Suponha que queremos decidir se uma dada seqüência de parênteses está bem formada ou não. Uma seqüência desse tipo é dita bem formada quando os parênteses e colchetes se fecham perfeitamente. Por exemplo, a seqüência (( ) [( )]) está bem formada, enquanto que a seqüência ([ )] não está bem formada.

Vamos então escrever um algoritmo que leia um vetor de caracteres s contendo a seqüência de parênteses e colchetes e determine a se a seqüência está bem formada ou não.

```
algoritmo
definatipo vetor[1..1000] de caracteres vet_carac
definatipo registro
inteiro topo
vet_carac L
fimrgistro tipo_pilha
vet_carac s
tipo_pilha P
lógico temp
inteiro i, n
temp ← V
P.topo ← 0
leia n
para i de 1 até n faça faça
  leia s[i]
fim para
i ← 1
enquanto i ≤ n AND temp = V faça
  se s[i] =')' então
    se P.topo ≠ 0 AND P.L[P.topo] =' (' então
      P.topo ← P.topo − 1
    senão
      temp = F
```

```
      fim se
    senão
      se s[i] =']' então
        se P.topo > 1 AND P.L.[P.topo − 1] =' [' então
          P.topo ← P.topo − 1
        senão
          temp ← F
        fim se
      senão
        P.topo ← P.topo + 1
        P.L[P.topo] ← s[i]
      fim se
    fim se
    i ← i + 1
  fim enquanto
  se temp = V então
    escreva "A seqüência é bem formada!"
  senão
    "A seqüência não é bem formada!"
  fim se
  fimalgoritmo
```

Lista de Exercícios

ponteiros

1. Escreva um algoritmo que leia um vetor A de n números inteiros e ordene esse vetor. O vetor deve ser alocado dinamicamente e ordenado utilizando apenas a notação de ponteiros.

2. Escreva um algoritmo que leia dois vetores A e B com n números inteiros, e crie um vetor C resultado da soma de A e B. Os vetores devem ser alocados dinamicamente e todas as operações envolvendo os seus elementos devem ser feitas utilizando a notação de ponteiros.

3. Escreva o algoritmo para uma função que recebe como parâmetros dois vetores de inteiros A e B e retorne um terceiro vetor C tal que:

   - C é a diferença entre X e Y;
   - C é o produto entre X e Y;
   - C é a intersecçaõ entre X e Y;
   - C é a união de X com Y.

Ordenação

1. Um algoritmo de ordenação é dito **estável** se não altera a posição relativa de elementos com mesmo valor. Por exemplo, se o vetor de inteiros v tiver dois elementos iguais a 222, primeiro um azul e depois um vermelho, um algoritmo de ordenação estável mantém o 222 azul antes do vermelho. Com base nessa definição, para cada um dos algoritmos vistos em sala de aula (bolha, seleção, inserção), determine se o algoritmo é estável ou não. Em caso afirmativo, escreva "sim".Em caso negativo, escreva "não" e forneça uma seqüência de números (entrada para o algoritmo) que comprove o fato do algoritmo não ser estável.

2. Seja S uma seqüência de n números tal que cada elemento representa um voto diferente para presidente do centro acadêmico do curso de Bacharelado em Análise de Sistemas da UFMS. O voto é um número inteiro que representa, de forma única, o estudante escolhido para o cargo. Escreva um algoritmo que receba S e n como entrada e determine o número que representa o candidato mais votado.

   dica: ordene a seqüência

Listas

1. Escreva um módulo que, dados como parâmetros uma lista alocada seqüencialmente L, o seu tamanho n e um valor x, devolva o número de elementos da lista cuja chave possui valor maior ou igual a x.

2. Escreva um módulo que, dados como parâmetros uma lista ORDENADA alocada seqüencialmente L, o seu tamanho n e um elemento *novo* cujo valor da chave é x, insira *novo* dentro da lista (logicamente, a lista deve permanecer ordenada após a inserção).

3. Escreva um módulo que, dados como parâmetros uma lista ORDENADA alocada seqüencialmente L, o seu tamanho n e o valor x, remova o elemento cujo valor da chave é x.

4. Escreva um módulo que, dados como parâmetros uma lista alocada seqüencialmente L e o seu tamanho n, inverta a ordem dos elementos dessa lista.

5. Escreva um módulo que, dados como parâmetros duas listas alocadas seqüencialmente L_1 e L_2 e o tamanho de cada lista, (m e n, respectivamente), devolva uma terceira lista L_3 resultado da combinação das listas L_1 e L_2.

6. Escreva um módulo que, dados como parâmetros duas listas ORDENADAS alocadas seqüencialmente L_1 e L_2 e o tamanho de cada lista, (m e n, respectivamente), devolva uma terceira lista L_3, também ORDENADA, resultado da combinação das listas L_1 e L_2.

Pilhas

1. **Utilizando uma pilha** (e as funções *Empilha* e *Desempilha* associadas), escreva um algoritmo que leia uma seqüência de caracteres e imprima essa seqüência de forma invertida.

2. Uma palavra construída sob o alfabeto $\Sigma = \{a, b\}$ é dita *bacana* se ela contém o mesmo número de $a's$ e $b's$. A palavra *abab*, por exemplo, é bacana, enquanto que a palavra *abb* não é bacana. Escreva um algoritmo que leia uma palavra e determine se ela é *bacana* ou não.

3. Na notação usual de expressões aritméticas, os operadores são escritos **entre** os operandos; por isso, a notação é chamada *infixa*. Na notação polonesa, ou *posfixa*, os operadores são escritos **depois** dos operandos. Exemplo:

Infixa	Posfixa
(A+B*C)	ABC*+
(A*(B+C)/D-E)	ABC+*D/E-
(A+B*(C-D*(E-F)-G*H)-I*3)	ABCDEF-*-GH*-*+I3*-

   Escreva um algoritmo que leia uma expressão em notação infixa e a traduza para a expressão posfixa. Para simplificar, suponha que a expressão infixa está correta e consiste apenas de letras, abre-parêntese, fecha-parêntese e símbolos para as quatro operações aritméticas. Além disso, suponha que a expressão toda está "embrulhada" em um par de parênteses.

4. Seja $1, 2, \ldots, n$ uma seqüência de elementos que serão inseridos e posteriormente removidos de uma pilha P, um de cada vez. A ordem de inserção dos elementos na pilha é $1, 2, \ldots, n$, enquanto que a remoção depende da ordem na qual as operações de remoção são realizadas.

   Exemplo:

   Com $n = 3$, a seqüência de operações

   incluir em P
   incluir em P
   remover de P
   incluir em P
   remover de P
   remover de P

   produzirá uma permutação $2, 3, 1$ a partir da entrada $1, 2, 3$.

   Representando por I e R, respectivamente, as operações de inserção e remoção, a permutação $2, 3, 1$ do exemplo acima pode ser denotada por $IIRIRR$. De modo geral, uma permutação é chamada **admissível** quando puder ser obtida mediante uma sucessão de inserções e remoções em uma pilha a partir da permutação $1, 2, \ldots, n$. Assim, a permutação $2, 3, 1$ do exemplo acima é admissível.

(a) Determine a permutação correspondente a $IIIRRIRR$, com $n = 4$.

(b) Dê um exemplo de uma permutação não admissível.

(c) Escreva uma relação de permutações admissíveis de $1, 2, 3, 4$.

Filas

(a) Mostre o estado de uma fila cujos elementos são inteiros e onde cabem, no máximo, 10 elementos, após a seguinte seqüência de operações, insere o elemento 10, insere o elemento 9, retira um elemento, insere o elemento 6, insere o elemento 7, insere o elemento 13, retira um elemento, insere o elemento 14, insere o elemento 15.

(b) Repita o último exercício da seção anterior utilizando uma fila (ao invés de uma pilha).

Capítulo 11

Arquivos

Nos capítulos anteriores manipulamos informações armazenadas na memória principal e os dados de entrada e saída eram fornecidos pelos meios padrões. Pela característica volátil da memória principal após desligarmos a máquina estes dados eram perdidos. Dados que precisam ser armazenados de forma mais duradoura (mesmo com a máquina desligada) são mantidos em **arquivos**.

Um arquivo é um conjunto de dados armazenado na memória secundária. Como o acesso ao disco é duas ou três ordem de magnitude mais lento que o acesso a memória principal, os dados dos arquivos são acessados em blocos (buffer) e se utilizam de estruturas de dados específicas para serem acessados. Neste capítulo estudaremos inicialmente as formas mais elementares de tratamento de arquivo e em seguida forneceremos uma visão mais conceitual das diversas formas de organização de estruturas de dados complexas em arquivos.

11.1 Comandos de manipulação de arquivo

Para se ter acesso ao dados de um arquivo é necessário abrí-lo. Antes de abrir um arquivo deve-se declarar um ponteiro especial que representará o arquivo nas operações de leitura e escrita subsequentes no arquivo.

O funcionamento da manipulação de arquivos se dá da seguinte forma: inicialmente, um comando abre o arquivo e cria um ponteiro para o arquivo. Além disso, um cursor é mantido indicando a posição corrente dentro do arquivo onde o cursor está posicionado. Os comando de leitura e escrita usam esta variável ponteiro bem como o cursor para manipular dados do arquivo. Enquanto o arquivo estiver aberto o

ponteiro representará o arquivo. No final o arquivo deve ser fechado.

Sintaxe: arquivo ^ponteiroparaarquivo

Exemplo

arquivo ^F

Para abrir e fechar um arquivo

No momento da abertura do arquivo o cursor indicará a posição inicial do mesmo. O comando abra realiza esta tarefa.

Sintaxe: abra nome, modo, ponteiro

onde:

nome: nome do arquivo (pode conter diretórios e extensão)

modo: especifica como o arquivo aberto será utilizado

ponteiro: ponteiro para arquivo

Os modos são:

r: para leitura e/ou

w: para escrita

que podem ser combinados com

b: para arquivos binários

t: para arquivos de texto

Caso o arquivo não possa ser aberto o ponteiro receberá valor NULO.

Exemplo:

abra "clientes.dat", rwt, F

Neste caso o arquivo `clientes.dat` do diretório corrente será aberto para leitura e escrita no modo texto e será apontado por F, caso exista.

Para fazermos o fechamento de uma arquivo previamente aberto usamos o comando

feche com a seguinte sintaxe:

feche ponteiro

No exemplo acima, para fecharmos o arquivo `clientes.dat` faremos

feche F

Para gravar dados em um arquivo

O comando de escrita em arquivo grava os dados solicitados a partir da posição corrente do arquivo indicada pelo cursor e automaticamente atualiza a posição do cursor para a posição imediatamente após os dados gravados.

sintaxe: escreva em ponteiro, valor1, valor2, ...

onde:

ponteiro: ponteiro que representa um arquivo aberto para escrita

valor1, etc: valores de dados reais que o algoritmo enviará ao arquivo.

Para leitura de dados de um arquivo

Comando leia de
sintaxe: leia de ponteiro, valor1, valor2, ...

onde:

ponteiro: ponteiro que representa um arquivo aberto para leitura

valor1, etc: valores de dados reais que o algoritmo receberá do arquivo.

Cada vez que um algoritmo ler dados de uma arquivo automaticamente o cursor mantém atualizada a posição de leitura corrente no arquivo aberto. Como resultado, o algoritmo lê sequencialmente do início para o fim o arquivo. Quando o algoritmo encontra o fim do arquivo o ponteiro do arquivo no comando receberá valor NULO.

Para remover um arquivo

Para apagar uma arquivo do disco temos o comando
sintaxe: remove ponteiro

Esta função remove o arquivo do disco apontado por ponteiro.

Exemplo: remove F

11.1.1 Para acesso aleatório a uma posição do arquivo

Podemos manipular o cursor de um arquivo mudando sua posição dentro do arquivo tanto para frente quanto para trás usando um comando específico para esta finalidade.

Comando posiciona
sintaxe: posiciona ponteiro, deslocamento, origem

Este comando posiciona o cursor do arquivo representado por ponteiro a um lugar determinado do arquivo a partir do ponto definido por origem. O deslocamento é um valor inteiro, e origem pode assumir: I, deslocamento a partir do início do arquivo; F, deslocamento a partir do final do arquivo; e A, deslocamento a partir da posição atual do cursor.

11.2 Estruturas de arquivos

Quando escrevemos um programa para gerar um arquivo de dados (e.g. cadastro de pessoa física), costumamos agrupar os campos que fornecem informações sobre um mesmo indivíduo em um registro. Como o conceito de registro é lógico e não físico, a integridade dos campos e a integridade dos registros podem ser perdidas quando o arquivo é gravado em disco, impossibilitando a recuperação dos dados. Técnicas de organização dos campos e dos registros são usadas para manter a integridade deles em disco.

11.2.1 Organização dos campos

Considere um programa para gravar um arquivo ASCII em disco com o primeiro nome e o último nome de cada indivíduo. Observe que a integridade de cada campo é

perdida, pois os dados são armazenados como uma sequência de bytes. Se tentarmos ler de volta os campos e imprimí-los na tela, vamos observar que não conseguimos separá-los. Portanto, precisamos de alguma técnica para identificar os campos do arquivo.

- Forçar que cada campo ocupe um tamanho fixo em bytes. O tamanho do arquivo pode aumentar bastante caso algumas informações requeram campos compridos, mas na maioria dos casos os campos não sejam completamente preenchidos.
- Reservar um certo número de bytes antes de cada campo para indicar o seu comprimento. Se existirem campos com mais do que 256 bytes, por exemplo, a informação de comprimento requererá mais do que 1 byte. Se o arquivo for grande, este adicional de memória pode ser significativo.
- Inserir um delimitador separando os campos. Este delimitador pode ser um caracter, o qual ocupa apenas 1 byte, mas temos que cuidar para não selecionar um caracter comum aos campos
- Utilizar uma palavra chave para identificar o conteúdo de cada campo (ex. nome=Alexandre). Este método tem a vantagem de prover informações sobre o conteúdo do arquivo, o que os outros métodos não oferecem, mas desperdiça bastante memória em disco.

11.2.2 Organização dos registros

- Registros de tamanhos fixos
  - Campos de tamanhos fixos
    Não precisamos nos preocupar com os delimitadores, pois a escrita e leitura vão gravar e ler o mesmo número de bytes definido para cada campo. Porém, esta opção desperdiça muita memória nos campos do registro (fragmentação).
  - Campos de tamanhos variáveis
    O número de campos pode ser variável no registro com delimitadores entre campos, não teremos desperdício no campo, mas podemos desperdiçar bytes do final do registro (fragmentação).

  Uma alternativa é definir o tamanho do registro supondo que os tamanhos máximos possíveis dos campos não ocorrem simultaneamente para um mesmo registro.

- Registros de tamanhos variáveis
  Os campos podem ser de tamanhos variáveis separados por delimitadores, porém precisamos identificar os registros também.

  – Reservar um certo número de bytes antes de cada registro para indicar o seu comprimento.
  – Inserir um delimitador separando os registros.
  – Usar um outro arquivo com o endereço em disco (offset) de cada registro.

A escolha do método depende da natureza e da utilização dos dados. Por exemplo, vamos supor o método de registros de tamanhos variáveis com campos de tamanhos variáveis, que indica o comprimento de cada registro e separa os campos por delimitadores. Dois problemas neste método são: saber o comprimento do registro antes de gravá-lo em disco e saber o comprimento máximo dos registros para decidir se usamos um inteiro (4 bytes) ou uma string de n caracteres (n bytes) para gravar esta informação. Para ler o registro, evitando acesso byte a byte ao arquivo, os bytes que correspondem ao registro podem ser lidos de uma só vez e carregados em um buffer na memória principal. O buffer é então interpretado byte a byte. O processo inverso pode ser feito para a gravação.

11.3 Acesso aos dados

Cada registro pode ser unicamente identificado por uma "chave de acesso" (i.e. chave primária), que pode ser um número ou uma sequência de caracteres. Caso a unicidade não seja exigida, um grupo de registros poderá ser acessado com uma mesma chave, denominada secundária.

No caso de uma sequência de caracteres, chaves primárias e secundárias devem ser representadas em forma canônica (padrão).

Ex: As chaves facão, Facão, FACÃO terão representação única FACAO para indicar um mesmo registro ou um grupo de registros. Partimos do princípio que o arquivo de dados é muito grande e não cabe na memória principal. Seus registros devem ser acessados no disco.

11.3.1 Acesso sequencial

A forma mais simples de acesso é a sequencial, isto é, o arquivo é lido registro por registro até encontrarmos o(s) registro(s) que possue(m) a mesma chave de acesso. Ex: Quando usamos o comando grep no UNIX/LINUX.

Normalmente esta é a forma menos eficiente de acesso, porém algumas situações favorecem o acesso sequencial. Ex: Quando o arquivo possui poucos registros, quando se trata de uma chave secundária que recupera um alto número de registros, e quando o acesso é uma operação rara (ex. atualização de um dado registro em um arquivo de backup).

11.3.2 Acesso direto

Visto que o acesso ao disco é computacionalmente bem mais caro do que o acesso à memória principal, o acesso sequencial passa a ser um problema cuja solução requer acesso direto. O acesso direto a um dado registro (ou grupo de registros) em disco requer conhecer o(s) endereço(s) do(s) registro(s) no arquivo de dados. Assim, comandos como fseek, podem ser usados para localizar diretamente o(s) registro(s) para leitura/gravação. Esses endereços e suas respectivas chaves são armazenados em uma estrutura de dados, denominada índice. Dependendo do seu tamanho, o índice pode caber ou não na memória principal. O último caso requer que o índice seja armazenado em arquivo separado no disco e carregado por partes na memória principal, durante a busca.

11.3.3 Endereçamento

O endereço de um registro é o deslocamento em bytes (offset) desde o início do arquivo, ou final do cabeçalho do arquivo (registro que armazena informações gerais tais como comprimento dos registros, data da última atualização, número de registros, etc.). No caso de registros de tamanho fixo, podemos armazenar no índice apenas o número de registros que antecedem cada registro (Relative Record Number - RRN). Neste caso, o endereço de um dado registro é obtido multiplicando-se o seu RRN pelo comprimento dos registros.

11.4 Gerenciamento de espaço disponível em arquivo

Suponha que um registro de tamanho variável é modificado de tal forma que o novo registro fica mais longo do que o original. Como resolver o problema?

1. Colocar os dados extras no final do arquivo e usar um campo no registro para indicar o endereço desses dados.
2. Gravar todo o registro no final do arquivo e disponibilizar o espaço original para um novo registro menor.

A opção 1 demanda muito processamento. A opção 2 é mais atraente, mas requer a solução de dois novos problemas.

1. Como reconhecer que um certo espaço no arquivo está disponível? Podemos colocar uma marca (*) no primeiro campo do registro cujo tamanho indica agora o número de bytes disponíveis.
2. Como reutilizar o espaço disponível?
   (a) Solução estática: Podemos marcar os registros disponíveis e rodar de tempos em tempos um programa para copiar os registros ativos para um novo arquivo.
   (b) Solução dinâmica: Podemos marcar os registros disponíveis e criar uma lista ligada com seus endereços. Esta solução é mais atraente, pois pode usar o próprio espaço disponível no arquivo para armazenar a lista, provendo formas imediatas de saber se existe um espaço disponível (i.e. lista não vazia) e de acessar o endereço disponível. O nó cabeça pode ser armazenado no cabeçalho do arquivo e os demais nos espaços disponíveis.

O gerenciamento de espaços disponíveis também ocorre no caso de remoção de registros, variando apenas o tratamento para registros de tamanho fixo e de tamanho variável.

11.4.1 Gerenciamento de memória com registros de tamanho fixo

A lista deve ser uma pilha, onde cada nó tem a marca e o RRN do próximo registro disponível. Usa-se sempre o primeiro disponível na pilha.

11.4.2 Gerenciamento de memória com registros de tamanho variável

Cada nó tem a marca, o endereço do próximo espaço disponível e o tamanho do espaço corrente. Esta abordagem traz dois novos problemas.

1. O espaço disponível tem que ter tamanho mínimo necessário para acomodar o novo registro (i.e. não podemos usar uma pilha e devemos buscar na lista o espaço a ser usado).
2. Se o novo registro for menor que o espaço disponível para ele, teremos fragmentação no final deste espaço. Caso o espaço que sobra seja colocado de volta na lista ele pode ser pequeno demais para ser reutilizado no futuro (i.e. a fragmentação persistente). Neste caso, podemos ainda fazer a união de espaços disponíveis adjacentes, mas isto requer processamento adicional, na inserção, para identificar esses espaços.

Esses problemas levam as seguintes técnicas de gerenciamento de memória.

1. First-fit: Varre-se a lista (pilha) e o primeiro espaço disponível com tamanho suficiente é usado. Não requer processamento adicional e é a opção mais indicada no caso de registros com mais ou menos o mesmo tamanho. A remoção insere o registro no início da lista.
2. Worst-fit: Mantém-se os nós da lista em ordem decrescente de tamanho. A vantagem é evitar varrer o resto da lista quando o primeiro nó já não tem tamanho suficiente para acomodar o novo registro. É a melhor opção se os espaços disponíveis são pequenos. Porém, desperdiça mais espaço no registro do que seria necessário e requer processamento adicional.
3. Best-fit: Mantém-se os nós da lista em ordem crescente de tamanho. Minimiza o desperdício de espaço no registro e é uma boa opção no caso de registros com mais ou menos o mesmo tamanho. Porém, requer processamento adicional.

Exercícios

1. Faça um programa que efetue a cópia de um arquivo existente para outro lugar e/ou outro nome. Ele deve funcionar como o programa cp do Linux ou o copy do DOS, assim: programa arquivo_origem arquivo_destino

2. Faça um programa de agenda que ofereça um menu para o usuário com as opções Inserir, Listar, Buscar e Sair. O programa deve armazenar os dados (nome, telefone e e-mail) em um arquivo binário.

3. Faça um programa que troque os espaços de um arquivo texto por tabulações. O nome do arquivo deve ser passado como argumento pela linha de comando.

4. Faça um programa que concatene dois arquivos texto, e o arquivo resultante tenha um nome diferente dos dois arquivos usados na concatenação. Use argumentos da linha de comando.