

# RESOLUÇÃO DE PROBLEMAS COM COMPUTADOR E FERRAMENTAS DE PROGRAMAÇÃO

---

## SUMÁRIO

2.1	Fases na resolução de problemas	REVISÃO DO CAPÍTULO
2.2	Programação modular	Conceitos-chave
2.3	Programação estruturada	Resumo
2.4	Conceito e características de algoritmos	EXERCÍCIOS
2.5	Escrevendo algoritmos	
ATIVIDADES DE PROGRAMAÇÃO		
RESOLVIDAS		

---

Este capítulo introduz a metodologia para a resolução de problemas com computadores e com uma linguagem de programação como C.

A resolução de um problema com um computador é feita escrevendo-se um programa, que exige pelo menos os seguintes passos:

1. Definição ou análise do problema.
2. Projeto do algoritmo.
3. Transformação do algoritmo em um programa.
4. Execução e validação do programa.

Entre os objetivos fundamentais deste livro estão a *aprendizagem* e o *projeto dos algoritmos*. Este capítulo introduz o conceito de algoritmo e de programa, assim como as ferramentas que permitem ao usuário "dialogar" com a máquina: *as linguagens de programação*.

---

## 2.1 FASES NA RESOLUÇÃO DE PROBLEMAS

O processo de resolução de um problema com um computador leva à escrita de um programa e à sua execução. Ainda que o processo de projetar programas seja — essencialmente — um processo criativo, pode-se considerar uma série de fases ou passos comuns que todos os programadores geralmente devem seguir.

As fases de resolução de um problema com computador são:

- Análise do problema
- Projeto do algoritmo
- Codificação
- Compilação e execução
- Verificação
- Depuração
- Manutenção
- Documentação

Constituem o ciclo de vida do software e as fases ou etapas usuais:

- **Análise.** O problema é analisado tendo presente a especificação dos requisitos dados pelo cliente da empresa ou pela pessoa responsável pelo programa.
- **Projeto.** Uma vez analisado o problema, projeta-se uma solução que conduzirá a um algoritmo que resolva o problema.
- **Codificação (implementação).** A solução é escrita na sintaxe da linguagem de alto nível (por exemplo, C) e se obtém um programa.
- **Compilação, execução e verificação.** O programa é executado, testado rigorosamente e são eliminados todos os erros (denominados *bugs*, em inglês) que possam aparecer.
- **Depuração e manutenção.** O programa é atualizado e modificado sempre que necessário de modo que sejam cumpridas todas as necessidades de mudança de seus usuários.
- **Documentação.** Escrita das diferentes fases do ciclo de vida do software, essencialmente a análise, o projeto e a codificação, com manuais do usuário e de referência, assim como normas para a manutenção.

As duas primeiras fases levam a um projeto detalhado escrito em forma de algoritmo. Durante a terceira etapa (*codificação*) é implementado<sup>1</sup> o algoritmo em um código escrito em uma linguagem de programação, refletindo as idéias desenvolvidas nas fases de análise e projeto.

A fase de *compilação e execução* traduz e executa o programa. Nas fases de *verificação e depuração* o programador busca erros das etapas anteriores e os elimina. Está comprovado que, quanto mais tempo se gasta na fase de análise e projeto, menos se gastará na depuração do programa. Por último, deve ser realizada a *documentação do programa*.

Antes de conhecer as tarefas realizadas em cada fase, vamos considerar o conceito e o significado da palavra **algoritmo**. A palavra *algoritmo* deriva da tradução do latim da palavra Alkhôwarizmi<sup>2</sup>, nome de um matemático e astrônomo árabe que escreveu um tratado sobre manipulação de números e equações no século IX. Um **algoritmo** é um método para resolver um problema mediante uma série de passos precisos, definidos e finitos.

### Característica de um algoritmo

- **preciso** (indicar a ordem de realização em cada passo),
- **definido** (se seguido duas vezes, obtém-se o mesmo resultado toda vez),
- **finito** (tem fim: um número determinado de passos).

<sup>1</sup> Na penúltima edição (21ª) do DRAE (Dicionário da Real Academia Espanhola) se acertou o termo implementar: (Informática) “Pôr em funcionamento, aplicar métodos e medidas, etc. para levar algo a cabo”.

<sup>2</sup> Foi escrito um tratado matemático famoso sobre a manipulação de números e equações intitulado *Kitab al-jabr w' almugabala*. A palavra álgebra derivou da sua semelhança sonora com *al-jabr*.

Um algoritmo deve produzir um resultado em um tempo finito. Os métodos que utilizam algoritmos são denominados *métodos algorítmicos*, em oposição aos métodos que implicam algum juízo ou interpretação, que são denominados *métodos heurísticos*. Os métodos algorítmicos podem ser *implementados* em computadores, entretanto, os processos *heurísticos* não foram convertidos facilmente nos computadores. Nos últimos anos, as técnicas de inteligência artificial têm possibilitado a *implementação* do processo heurístico em computadores.

Exemplos de algoritmos são: instruções para montar uma bicicleta, fazer uma receita culinária, obter o máximo divisor comum de dois números etc. Os algoritmos podem ser expressos por *fórmulas*, *diagramas de fluxo* ou *N-S* e *pseudocódigos*. Essa última representação é a mais utilizada em linguagens estruturadas como C.

### 2.1.1 Análise do problema

A primeira fase da resolução de um problema com computador é a *análise do problema*. Ela requer uma clara definição, na qual sejam observados exatamente o que deve fazer o programa e o resultado ou solução desejada.

Dado que se busca uma solução pelo computador, são necessárias especificações detalhadas de entrada e saída. A Figura 2.1 mostra os requisitos que devem ser definidos na análise.

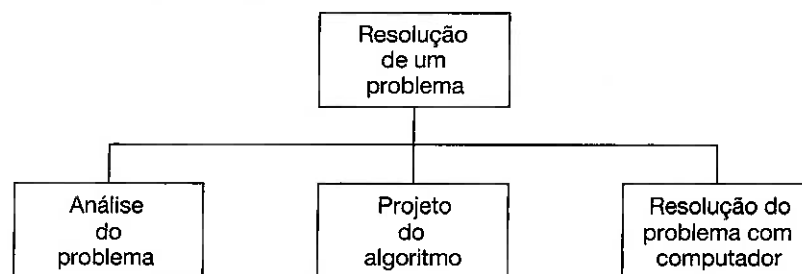


Figura 2.1 Análise do problema.

Para poder definir bem um problema, é conveniente responder as seguintes perguntas:

- Que entrada se requer? (tipo e quantidade).
- Qual é a saída desejada? (tipo e quantidade).
- Que método produz a saída desejada?

#### Problema 2.1

Desejamos obter uma tabela com as depreciações acumuladas e dois valores reais de cada ano de um automóvel comprado por 1.800.000 pesetas no ano de 1996, durante os seis anos seguintes, supondo um valor de recuperação ou resgate de 120.000. Realizar a análise do problema, conhecendo a fórmula da depreciação anual constante  $D$  para cada ano de vida útil.

$$D = \frac{\text{custo} - \text{valor de recuperação}}{\text{vida útil}}$$

$$D = \frac{1.800.000 - 120.000}{6} = \frac{1.680.000}{6} = 280.000$$

$$\text{Entrada} \left\{ \begin{array}{l} \text{custo original} \\ \text{vida útil} \\ \text{valor de recuperação} \end{array} \right.$$

Saída	<ul style="list-style-type: none"> <li>depreciação anual por ano</li> <li>depreciação acumulada em cada ano</li> <li>valor do automóvel em cada ano</li> </ul>
Processo	<ul style="list-style-type: none"> <li>depreciação acumulada</li> <li>cálculo da depreciação acumulada a cada ano</li> <li>cálculo do valor do automóvel em cada ano</li> </ul>

A Tabela 2.1 mostra a saída solicitada.

**Tabela 2.1**

Ano	Depreciação	Depreciação acumulada	Valor anual
1 (1996)	280.000	280.000	1.520.000
2 (1997)	280.000	560.000	1.240.000
3 (1998)	280.000	840.000	960.000
4 (1999)	280.000	1.120.000	680.000
5 (2000)	280.000	1.400.000	400.000
6 (2001)	280.000	2.180.000	120.000

### 2.1.2 Projeto do algoritmo

Na etapa de análise do processo de programação, é determinado o que o programa faz. Na etapa do projeto é determinado como o programa faz a tarefa solicitada. Os métodos mais eficazes para o processo de projeto se baseiam no conhecido por *dividir e conquistar*, ou seja, a resolução de um problema complexo é obtida dividindo-se o problema em subproblemas e esses subproblemas em outros de nível mais baixo, até que possa ser *implementada* uma solução no computador. Esse método é conhecido tecnicamente como **projeto descendente** (*top-down*) ou **modular**. O processo de quebrar o problema em cada etapa e expressar cada passo em forma mais detalhada é denominado *refinamento sucessivo*.

Cada subprograma é resolvido mediante um **módulo** (*subprograma*) que tem somente um ponto de entrada e um ponto de saída.

Qualquer programa bem projetado contém um *programa principal* (o módulo de nível mais alto) que chama subprogramas (módulos de nível mais baixo), que por sua vez podem chamar outros subprogramas. Para os programas estruturados dessa forma, dizemos que possuem um *projeto modular*, e o método de quebrar o programa em módulos menores denomina-se *programação modular*. Os módulos podem ser planejados, codificados, testados e depurados independentemente (inclusive por diferentes programadores) e depois combinados entre si. O processo implica a execução dos seguintes passos até que o programa termine:

1. Programar um módulo.
2. Testar o módulo.
3. Se for necessário, depurar o módulo.
4. Combinar o módulo com os módulos anteriores.

O processo que converte os resultados de análise do problema em um projeto-modular com refinamentos sucessivos que permitam uma tradução posterior para uma linguagem é denominado **projeto de algoritmo**.

O projeto de algoritmo é independente da linguagem de programação usada para posterior codificação.

### 2.1.3 Ferramentas de programação

As duas ferramentas mais utilizadas para projetar algoritmos são: *diagramas de fluxo* e *pseudocódigo*.

#### Diagramas de fluxo

Um **diagrama de fluxo** (*flowchart*) é uma representação gráfica de um algoritmo. Os símbolos utilizados foram padronizados pelo Instituto Norte-americano de Padronização (ANSI), e os mais empregados estão na Figura 2.2, junto com uma planilha utilizada para o projeto dos diagramas de fluxo (Figura 2.3). A Figura 2.4 representa o diagrama de fluxo que resolve o Problema 2.1.

#### Pseudocódigo

O **pseudocódigo** é uma ferramenta de programação na qual as instruções são escritas em palavras similares ao inglês ou português, que facilitam tanto a escrita como a leitura de programas. Essencialmente, o pseudocódigo pode ser definido como uma *linguagem de especificações de algoritmos*.

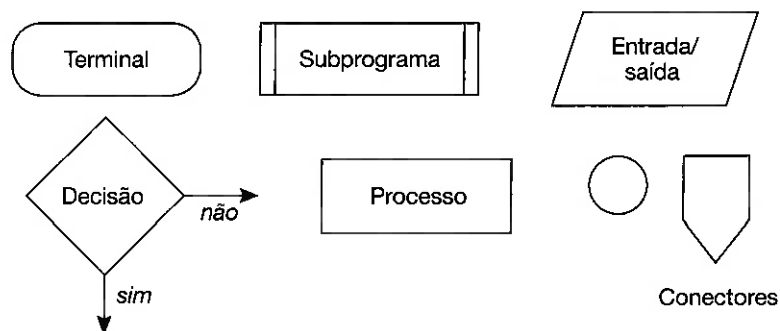


Figura 2.2 Símbolos mais utilizados nos diagramas de fluxo.

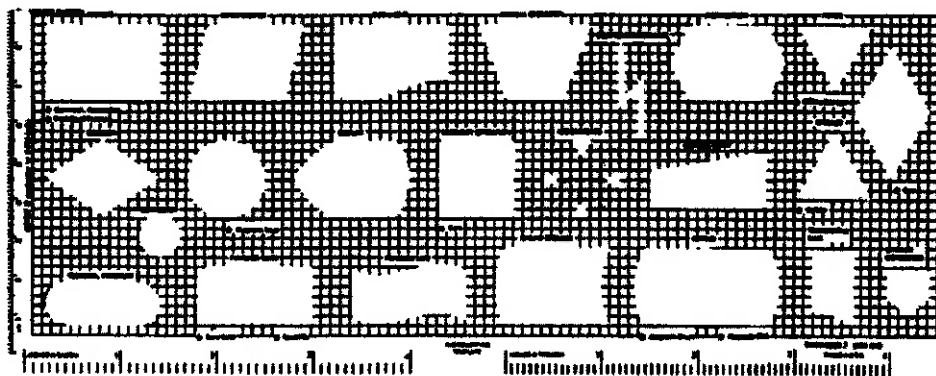


Figura 2.3 Planilha para projeto de diagramas de fluxo.

Mesmo não existindo regras para se escrever o pseudocódigo em português, é comum usar uma notação-padrão que será utilizada no livro e que é muito empregada nos livros de programação em português. As palavras reservadas basicamente são representadas por letras minúsculas em negrito. Essas palavras são traduções livres de palavras reservadas de linguagens como C, Pascal etc. Mais adiante serão indicados os pseudocódigos fundamentais que serão utilizados nesta obra.

O pseudocódigo que resolve o Problema 2.1 é:

```

Previsões de depreciação
Introduzir custo
    vida útil
    valor final de resgate (recuperação)

imprimir cabeçalhos
Estabelecer o valor inicial do Ano
Calcular depreciação
enquanto valor ano <= vida útil fazer
    calcular depreciação acumulada
    calcular valor atual
    imprimir uma linha na tabela
    incrementar o valor do ano
fim do enquanto
  
```

e seu diagrama de fluxo aparece na Figura 2.4.

### Exemplo 2.1

Calcular o pagamento líquido de um trabalhador conhecendo o número de horas trabalhadas, a tarifa horária e a alíquota de impostos.

*Algoritmo*

1. Ler Horas, Tarifa, alíquota
2. Calcular ValorBruto = Horas \* Tarifa
3. Calcular impostos = ValorBruto \* Alíquota
4. Calcular ValorLíquido = ValorBruto – Impostos
5. Visualizar ValorBruto, Impostos, ValorLíquido

### Exemplo 2.2

Calcular o valor da soma  $1+2+3 \dots + 100$ .

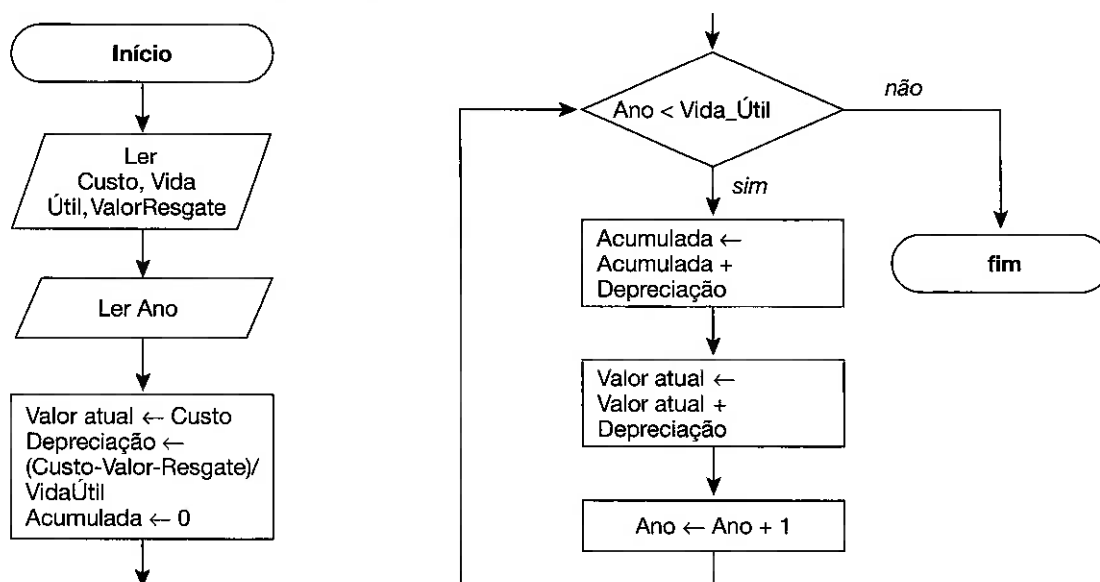


Figura 2.4 Diagrama de fluxo (Problema 2.1).

### Algoritmo

Utiliza-se uma variável *Contador* como um contador que gere os sucessivos números inteiros, e *Soma* para armazenar as somas parciais 1, 1+2, 1+2+3...

1. Estabelecer *Contador* a 1
  2. Estabelecer *Soma* a 0
  3. **enquanto** *Contador* <= 100 **fazer**  
     Somar *Contador* a *Soma*  
     Incrementar *Contador* em 1  
   **fim-enquanto**
  4. Visualizar *Soma*
- 

### 2.1.4 Codificação de um programa

*Codificação* é a escrita, em uma linguagem de representação da programação, do algoritmo desenvolvido nas etapas precedentes. Dado que o projeto de um algoritmo é independente da linguagem de programação utilizada para sua implementação, o código pode ser escrito com igual facilidade em uma linguagem ou em outra.

Para realizar a conversão do algoritmo em programas, devemos substituir as palavras reservadas em português por seus homônimos em inglês, e as operações/instruções indicadas em linguagem natural devem ser expressas na linguagem de programação correspondente.

```
/*
Este programa escrito em "C" obtém uma tabela de depreciações acumuladas e
valores reais de cada ano de um determinado produto
*/
# include <stdio.h> void main()
{
    double    Custo, Depreciação,
              Valor_Recuperação,
              Valor_Atual,
              Acumulado
              Valor_Anual;
    int Ano, Vida_Útil;
    puts ("Introduza custo, valor recuperação e vida útil");
    scanf ("%lf %lf %lf", &Custo, &Valor_Recuperação, &Vida_Útil);
    puts ("Introduza ano atual");
    scanf ("%d", &Ano);
    Valor_Atual = Custo;
    Depreciação = (Custo-Valor_Recuperação)/Vida_Útil;
    Acumulado = 0;
    puts ("Ano Depreciação Dep. Acumulada");
    while (Ano < Vida_Útil)
    {
        Acumulado = Acumulado + Depreciação;
        Valor_Atual = Valor_Atual - Depreciação;
        printf ("Ano: %d, Depreciação:%.2lf, 2lf Acumulada",
               Ano, Depreciação, Acumulado);
        Ano = Ano + 1;
    }
}
```

### Documentação interna

Como veremos mais adiante, a documentação de um programa é classificada em *interna e externa*. A *documentação interna* é a que se inclui dentro do código do programa-fonte mediante comentários que ajudam na compreensão do código. Todas as linhas de programas que comecem com um símbolo */\** são *comentários*. O programa não necessita deles e o computador os ignora. Essas linhas de comentários servem apenas para fazer os programas mais fáceis de compreender. O objetivo do programador deve ser escrever códigos simples e limpos.

Como as máquinas atuais suportam grande memória (256 MB ou 512 MB de memória central mínima em computadores pessoais), não é necessário recorrer a técnicas de economia de memória, portanto é recomendável que seja incluído o maior número de comentários possível, desde que significativos.

### 2.1.5 Compilação e execução de um programa

Uma vez que o algoritmo foi convertido em um programa-fonte, é preciso introduzi-lo na memória por meio do teclado e armazená-lo posteriormente em um disco. Essa operação é realizada com um programa editor, e em seguida o programa-fonte é convertido em um *arquivo de programa* que é guardado (*gravado*) em disco.

O **programa-fonte** deve ser traduzido para linguagem de máquina, processo realizado com o compilador e o sistema operacional, que se encarrega, na prática, da compilação.

Se a compilação apresenta erros (*erros de compilação*) no programa-fonte, é preciso voltar a editar o programa, corrigir os erros e compilar de novo. Esse processo se repete até que não se produzam erros, obtendo-se o **programa-objeto**, que ainda não é executável diretamente. Supondo que não existam erros no programa-fonte, deve-se instruir o sistema operacional para que realize a fase de **montagem ou enlace** (*link*), carga, do programa-objeto com as bibliotecas do programa do compilador. O processo de montagem produz um **programa executável**. A Figura 2.5 descreve o processo completo de compilação/execução de um programa.

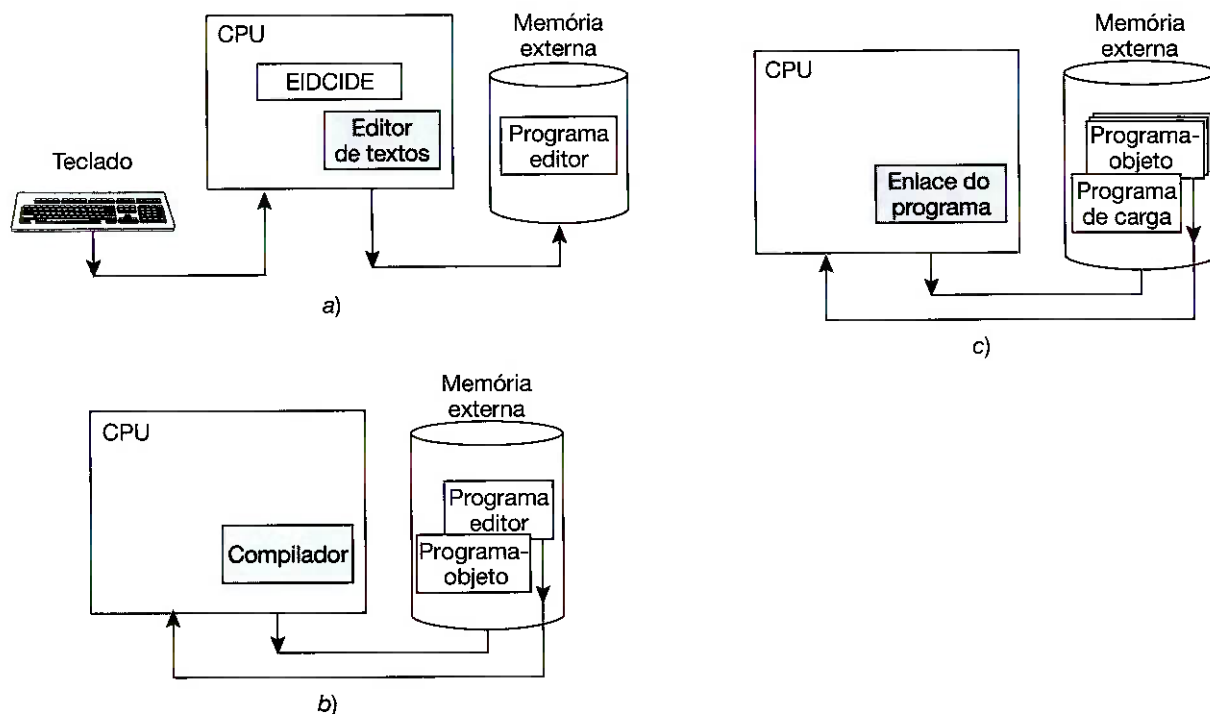


Figura 2.5 Fases da compilação/execução de um programa: a) edição; b) compilação; c) montagem ou enlace.

Quando o programa executável tiver sido criado, podemos executar (rodar) do sistema operacional por exemplo escrevendo seu nome (no caso de DOS). Supondo-se que não existem erros durante a execução (chamados **erros de tempo de execução**), será obtida uma saída de resultados do programa.

As instruções ou ordens para compilar e executar um programa em C podem variar segundo o tipo de compilador. Deve-se observar que o processo de Visual C++ 6 é diferente de C no Unix ou no Linux.

### 2.1.6 Verificação e depuração de um programa

A *verificação* de um programa é o processo de execução do programa com uma ampla variedade de dados de entrada, chamados *dados de teste ou prova*, que determinarão se o programa tem erros (*bugs*). Para realizar a verificação devemos desenvolver uma ampla gama de dados de teste: valores de entrada normais que testem aspectos especiais do programa.

A *depuração* é o processo de encontrar os erros do programa e corrigir ou eliminar esses erros.

Quando se executa um programa, podem ser produzidos três tipos de erros:

1. *Erros de compilação*. São produzidos normalmente pelo uso incorreto das regras da linguagem de programação e costumam ser *erros de sintaxe*. Se existe um erro de sintaxe, o computador não pode compreender a instrução, não será obtido o programa-objeto e o compilador imprimirá uma lista de todos os erros encontrados durante a compilação.
2. *Erros de execução*. São produzidos por instruções que o computador pode compreender mas não executar. Exemplos: divisão por zero e raízes quadradas de números negativos. Nestes casos a execução do programa é interrompida e uma mensagem de erro é impressa.
3. *Erros lógicos*. São produzidos na lógica do programa e a fonte do erro costuma ser o projeto do algoritmo. Estes erros são os mais difíceis de detectar, já que o programa pode funcionar e não produzir erros de compilação nem de execução, e somente podemos perceber o erro pela obtenção de resultados incorretos. Nesse caso, deve-se voltar à fase de projeto do algoritmo, modificar o algoritmo, mudar o programa-fonte e compilar e executar mais uma vez.

### 2.1.7 Documentação e manutenção

A documentação de um problema são descrições dos passos a serem dados no processo de resolução de um problema. A importância da documentação deve ser destacada pela sua influência decisiva no produto final. Programas mal documentados são difíceis de ler, mais difíceis de depurar e quase impossíveis de manter e modificar.

A documentação de um programa pode ser *interna* e *externa*. A *documentação interna* é a contida nas linhas de comentários. A *documentação externa* inclui análise, diagramas de fluxo e/ou pseudocódigos, manuais de usuário com instruções para executar o programa e para interpretar os resultados.

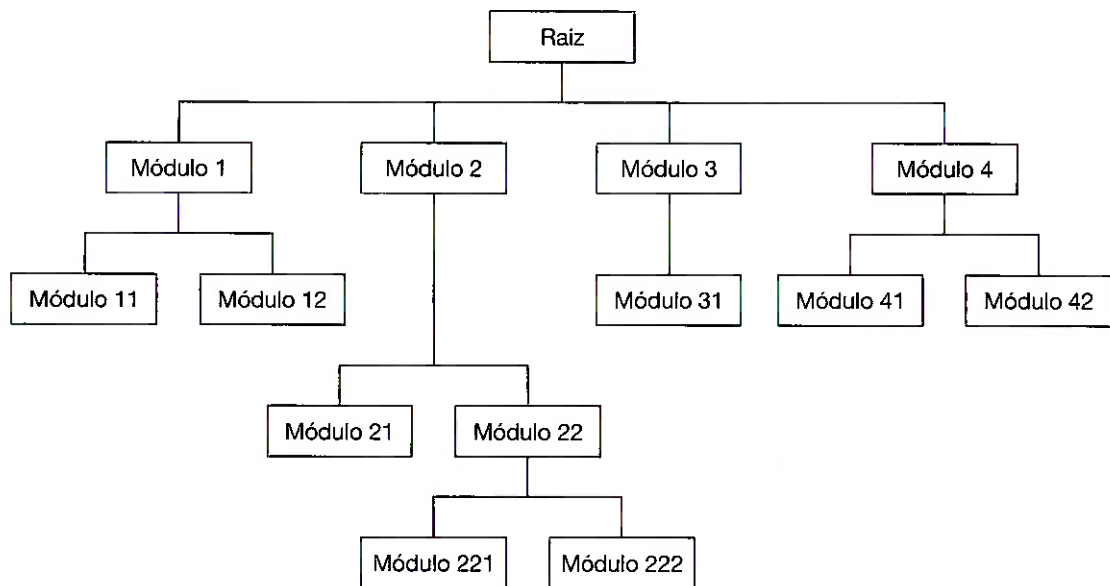
A documentação é vital quando se deseja corrigir possíveis erros futuros ou mudar o programa. Tais mudanças são denominadas *manutenção do programa*. Depois de cada mudança, a documentação deve ser atualizada para facilitar mudanças posteriores. Uma prática freqüente é enumerar as sucessivas versões dos programas **1.0, 1.1, 2.0, 2.1** etc. (Se as mudanças introduzidas são importantes, varia o primeiro dígito [**1.0, 2.0**,...]; no caso de pequenas mudanças, somente varia o segundo dígito [**2.0, 2.1**,...].)

## 2.2 PROGRAMAÇÃO MODULAR

A *programação modular* é um dos métodos de projeto mais flexíveis e potentes para melhorar a produtividade de um programa. Em programação modular, o programa é dividido em *módulos* (partes independentes), cada um dos quais executa uma atividade ou tarefa e é codificado independentemente de outros módulos. Cada um desses módulos é analisado, codificado e colocado em uso separadamente.

Cada programa contém um módulo denominado *programa principal*, que controla tudo o que acontece; transfere-se o controle para *subprodutos* (posteriormente denominados *subprogramas*), de modo que eles possam executar suas funções; entretanto, cada submódulo devolve o controle ao módulo principal quando completa sua tarefa. Se a tarefa alocada a cada submódulo é muito complexa, ele deverá ser quebrado em outros módulos menores. O processo sucessivo de subdivisão de módulos continua até que cada módulo tenha apenas uma tarefa específica para executar. Essa tarefa pode ser *entrada*, *saída*, *manipulação de dados*, *controle de outros módulos* ou alguma *combinação destes*. Um módulo pode transferir temporariamente (*ir para*) o controle para outro módulo, entretanto, cada módulo deve eventualmente devolver o controle ao módulo do qual recebeu originalmente o controle.

Os módulos são independentes no sentido de que nenhum módulo pode ter acesso direto a qualquer outro módulo exceto ao módulo que chama e a seus próprios submódulos. Entretanto, os resultados produzidos por um módulo podem ser utilizados por quaisquer outros módulos quando lhes é transferido o controle.



**Figura 2.6** Programação modular.

Dado que os módulos são independentes, diferentes programadores podem trabalhar simultaneamente em diferentes partes do mesmo programa. Isso reduzirá o tempo do projeto do algoritmo e posteriores codificações do programa. Além disso, um módulo pode ser modificado radicalmente sem afetar os outros módulos, inclusive sem alterar sua função principal.

A decomposição de um programa em módulos independentes mas simples é conhecida também como o método de “**dividir para conquistar**” (*divide and conquer*). Projetando-se cada módulo com independência dos demais e seguindo um método ascendente ou descendente, se chegará até a decomposição final do problema em módulos de maneira hierárquica.

## 2.3 PROGRAMAÇÃO ESTRUTURADA

Os termos *programação modular*, *programação descendente* e *programação estruturada* foram introduzidos na segunda metade da década de 1960 e costumam ser usados como sinônimos, embora não signifiquem a mesma coisa. As programações modular e descendente já foram examinadas. A *programação estruturada* significa escrever um programa conforme as seguintes regras:

- O programa tem um projeto modular.
- Os módulos são projetados de modo descendente.
- Cada módulo é codificado utilizando-se as três estruturas de controle básicas: *seqüência*, *seleção* e *repetição*.

Para quem está familiarizado com linguagens como BASIC, Pascal, FORTRAN ou C, a programação estruturada significa também *programação sem GOTO* (C não requer o uso da sentença **GOTO**).

A expressão *programação estruturada* se refere a um conjunto de técnicas desenvolvidas dos trabalhos fundamentais de Edgar Dijkstra. Essas técnicas ampliam consideravelmente a produtividade do programa, reduzindo em alto grau o tempo necessário para escrever, verificar, depurar e manter os programas. A programação estruturada utiliza um número limitado de estruturas de controle que minimizam a complexidade dos programas e, portanto, reduzem os erros; os programas ficam mais fáceis de escrever, verificar, ler e manter. Os programas devem ser dotados de uma estrutura.

A **programação estruturada** é o conjunto de técnicas que incorporam:

- *recursos abstratos,*
- *projeto descendente (top-down),*
- *estruturas básicas.*

### 2.3.1 Recursos abstratos

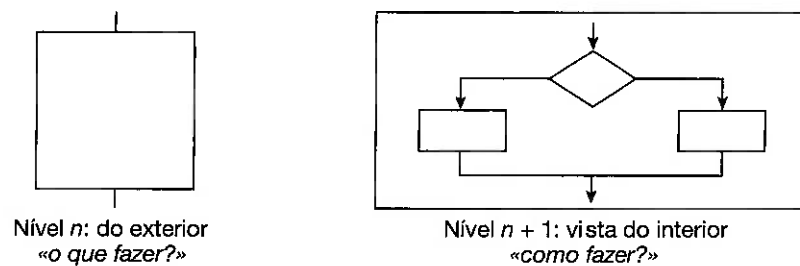
A programação estruturada faz uso de recursos abstratos em vez de recursos concretos que uma determinada linguagem de programação dispõe.

*Decompor um programa em termos de recursos abstratos* — segundo Dijkstra — consiste em decompor uma determinada ação complexa em termos de ações mais simples que podem ser executadas ou que constituam instruções disponíveis de computador.

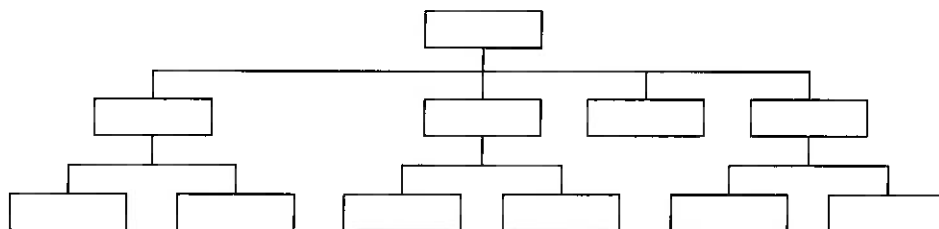
### 2.3.2 Projeto descendente (top-down)

O projeto descendente (*top-down*) é o processo mediante o qual um problema é decomposto em uma série de níveis ou passos sucessivos de refinamento (*stepwise*). A metodologia descendente consiste em efetuar um relacionamento entre as sucessivas etapas de estruturação de modo a se relacionarem mediante entradas e saídas de informação, ou seja, decomposição do problema em etapas ou estruturas hierárquicas, de maneira a se poder considerar cada estrutura dos pontos de vista do *que fazer?* e *como fazer?*

Para um nível  $n$  de refinamento, as estruturas são consideradas da seguinte maneira:



O projeto descendente pode ser visto na Figura 2.7.



**Figura 2.7** Projeto descendente.

### 2.3.3 Estruturas de controle

As *estruturas de controle* de uma linguagem de programação são métodos de especificar a ordem em que as instruções de um algoritmo são executadas. A ordem de execução das sentenças (linguagem) ou instruções determinam o *fluxo de controle*. Essas estruturas de controle são, conseqüentemente, fundamentais nas linguagens de programação e nos projetos de algoritmos, especialmente os pseudocódigos.

As três estruturas de controle básico são:

- *seqüência,*
- *seleção,*
- *repetição.*

e serão estudadas nos Capítulos 4 e 5.

A programação estruturada torna os programas mais fáceis de escrever, verificar, ler e manter; utiliza um número limitado de estruturas de controle, o que minimiza a complexidade dos problemas.

### 2.3.4 Teorema de programação estruturada: estruturas básicas

Em maio de 1966, Böhm e Jacopini demonstraram que um *programa próprio* pode ser escrito utilizando-se somente três tipos de estruturas de controle.

- *seqüenciais,*
- *seletivas,*
- *repetitivas.*

Um programa é definido como **próprio** se apresenta as seguintes características:

- *Possui somente um ponto de entrada e um de saída ou fim para controle de programa.*
- *Existem caminhos da entrada até a saída que podem ser seguidos e que passam por todas as partes do programa.*
- *Todas as instruções são executáveis e não existem laços infinitos (sem fim).*

Os Capítulos 4 e 5 são dedicados ao estudo das estruturas de controle seletivas e repetitivas.

*A programação estruturada significa:*

- O programa completo tem um projeto modular.
- Os módulos são projetados com metodologia descendente (pode ser também ascendente).
- Cada módulo é codificado utilizando as três estruturas de controle básicas: seqüenciais, seletivas e repetitivas (ausência total de sentenças **GOTO**).
- *Estruturação e modulação* são conceitos complementares.

## 2.4 CONCEITO E CARACTERÍSTICAS DE ALGORITMOS

O objetivo fundamental deste texto é ensinar a resolver problemas por meio de um computador. O programador de computador é acima de tudo uma pessoa que resolve problemas; portanto, para ser um programador eficaz, é necessário aprender a resolver problemas de um modo rigoroso e sistemático. Neste livro nos referiremos à *metodologia necessária para resolver problemas por meio de programas*, conceito que é denominado **metodologia da programação**, cujo ponto central é o conceito de algoritmo, já tratado.

Um algoritmo é um método para resolver um problema. Ainda que a popularização do termo tenha chegado com o advento da era da informática, **algoritmo** provém de *Mohammed al-Khowârizmi*, matemático persa que viveu durante o século IX e alcançou grande nome pelo enunciado das regras passo-a-passo para somar, subtrair, multiplicar e dividir números decimais; a tradução para o latim do sobrenome na palavra *algorismus* derivou posteriormente em algoritmo. Euclides, o grande matemático grego (do século IV a.C.) que inventou um método para encontrar o maior divisor comum de dois números, é considerado, ao lado de Al-Khowârizmi, o outro grande pai da ciência que trata dos algoritmos.

O professor Niklaus Wirth — inventor de Pascal, Modula-2 e Oberon — intitulou um de seus mais famosos livros como *Algoritmos + Estruturas de Dados = Programas*, significando que somente se pode realizar um bom programa com o projeto de um algoritmo e uma correta estrutura de dados. Essa equação será uma das hipóteses fundamentais consideradas nesta obra.

A resolução de um problema exige o projeto de um algoritmo que resolva o problema proposto.

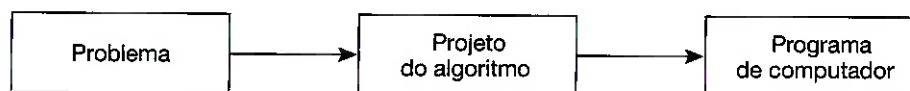


Figura 2.8 Resolução de um problema.

Os passos para a resolução de um problema são:

1. *Projeto do algoritmo*, que descreve a sequência ordenada de passos — sem ambigüidades — que leva à solução de um problema dado. (*Análise do problema e desenvolvimento do algoritmo.*)
2. Expressar o algoritmo como um *programa* em uma linguagem de programação adequada. (*Fase de codificação.*)
3. *Execução e validação* do programa pelo computador.

Para chegarmos à realização de um programa, é necessário o projeto prévio de um algoritmo, pois sem algoritmo não pode existir um programa.

Os algoritmos são independentes tanto da linguagem de programação em que se expressam como do computador que os executa. Em cada problema o algoritmo pode ser expresso em uma linguagem de programação diferente e executado em um computador distinto; entretanto, o algoritmo será sempre o mesmo. Assim, por exemplo, em uma analogia com a vida diária, uma receita culinária pode ser expressa em espanhol, inglês ou francês, mas, qualquer que seja a linguagem, os passos para a sua elaboração serão realizados sem que importe o idioma do cozinheiro.

Na ciência da computação e na programação, os algoritmos são mais importantes que as linguagens de programação ou os computadores. Uma linguagem de programação é tão somente um meio para expressar um algoritmo e um computador é apenas um processador para executá-lo. Tanto a linguagem de programação como o computador são os meios para obter um fim: conseguir que o algoritmo seja executado e que seja efetuado o processo correspondente.

Dada a importância do algoritmo na ciência da computação, um aspecto muito importante será o *projeto de algoritmos*. Ao ensinamento e à prática dessa tarefa dedicamos grande parte deste livro.

O projeto da maioria dos algoritmos requer criatividade e conhecimentos profundos da técnica da programação. Essencialmente, *a solução de um problema pode ser expressa por meio de um algoritmo*.

### 2.4.1 Características dos algoritmos

As características fundamentais de todo algoritmo são:

- Um algoritmo deve ser *preciso* e indicar a ordem de realização de cada passo.
- Um algoritmo deve estar *definido*. Seguindo um algoritmo duas vezes, devemos obter o mesmo resultado toda vez.
- Um algoritmo deve ser *finito*. Seguindo um algoritmo, devemos terminar em algum momento; ou seja, deve ter um número finito de passos.

A definição de um algoritmo deve descrever três partes: *Entrada*, *Processamento* e *Saída*. No algoritmo da receita citado anteriormente teremos:

*Entrada:* ingredientes e utensílios empregados.  
*Processamento:* elaboração da receita na cozinha.  
*Saída:* nome do prato (por exemplo, carneiro).

### Exemplo 2.3

Um cliente efetua um pedido a uma fábrica. A fábrica examina em seu banco de dados a ficha do cliente, se o cliente é bom a empresa aceita o pedido; caso contrário, recusa o pedido. Redigir o algoritmo correspondente.

Os passos do algoritmo são:

1. Início.
2. Ler o pedido.
3. Examinar a ficha do cliente.
4. Se o cliente é solvente, aceitar pedido; caso contrário, recusar pedido.
5. Fim.

### Exemplo 2.4

Projetar um algoritmo para saber se um número é primo ou não.

Um número é primo se somente pode ser dividido por ele mesmo e pela unidade (ou seja, não possui mais divisores que ele mesmo e a unidade). Por exemplo, 9, 8, 6, 4, 12, 16, 20 etc. não são primos, já que são divisíveis por números distintos deles mesmos e da unidade. Assim, 9 é divisível por 3, 8 é por 2 etc. O algoritmo de resolução do problema passa por dividir sucessivamente o número por 2, 3, 4... etc.

1. Início.
2. X igual a 2 ( $X = 2$ , X variável que representa os divisores do número que se busca N).
3. Dividir N por X ( $N/X$ ).
4. Se o resultado de  $N/X$  é inteiro, então N não é um número primo e ir para o ponto 7; caso contrário, continuar o processamento.
5. Soma 1 a X ( $X \leftarrow X + 1$ ).
6. Se X é igual a N, então N é um número primo; caso contrário, ir para o ponto 3.
7. Fim.

Por exemplo, se N é 131, os passos anteriores seriam:

1. Início.
2.  $X = 2$ .
3.  $131/X$ . Como o resultado não é inteiro, continua o processamento.
5.  $X \leftarrow 2 + 1$ , logo  $X = 3$ .
6. Como X não é 131, continua o processamento.
3.  $131/X$  resultado não é inteiro.
5.  $X \leftarrow 3 + 1$ ,  $X = 4$ .
6. Como X não é 131, continua o processamento.
3.  $131/X$ ... etc.
7. Fim.

### Exemplo 2.5

Realizar a soma de todos os números pares entre 2 e 1.000.

O problema consiste em somar  $2 + 4 + 6 + 8 \dots + 1.000$ . Utilizaremos as palavras SOMA e NÚMERO (*variáveis*, serão denominadas depois) para representar as somas sucessivas  $(2 + 4)$ ,  $(2 + 4 + 6)$ ,  $(2 + 4 + 6 + 8)$  etc. A solução pode ser escrita com o seguinte algoritmo:

1. Início.
2. Estabelecer SOMA em 0.
3. Estabelecer NÚMERO em 2.
4. Somar NÚMERO a SOMA. O resultado será o novo valor da soma (SOMA).
5. Incrementar NÚMERO em 2 unidades.
6. Se NÚMERO  $\leq 1.000$  ir para o passo 4; caso contrário, escrever o último valor da SOMA e terminar o processamento.
7. Fim.

### 2.4.2 Projeto do algoritmo

Um computador não tem capacidade para solucionar problemas quando não lhe são proporcionados passos sucessivos para realizar. Esses passos sucessivos que indicam as instruções a serem executadas pela máquina são, como já conhecemos, o *algoritmo*.

A informação proporcionada ao algoritmo constitui sua *entrada* e a informação produzida pelo algoritmo constitui sua *saída*.

Os problemas complexos podem ser resolvidos mais eficazmente com o computador quando são quebrados em subproblemas mais fáceis de solucionar que o original. Esse método é denominado *dividir para conquistar* (*divide and conquer*) e consiste em dividir um problema complexo em outros mais simples. Assim, o problema de encontrar a superfície e o comprimento de um círculo pode ser dividido em três problemas mais simples ou *subproblemas* (Figura 2.9).

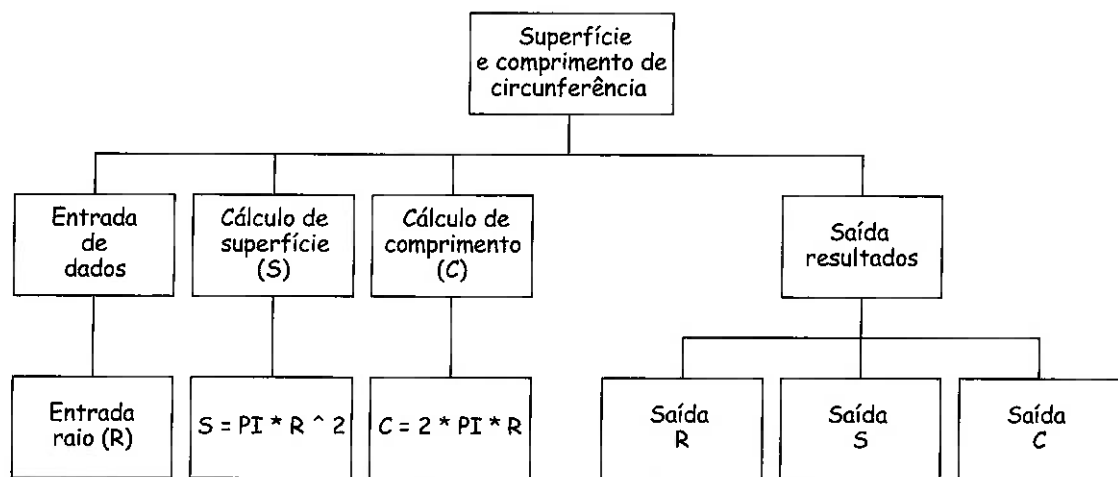


Figura 2.9 Refinamento de um algoritmo.

A decomposição do problema original em subproblemas mais simples e esses em outros ainda mais simples que podem ser implementados para sua solução no computador denomina-se *projeto descendente* (*top-down design*). Normalmente os passos projetados no primeiro esboço do algoritmo são incompletos e indicarão apenas uns poucos passos (máximo de 12 aproximadamente). Após essa primeira descrição, esses passos se ampliam em uma descrição mais detalhada com mais passos específicos. O processo é

denominado *refinamento do algoritmo (stepwise refinement)*. Problemas complexos necessitam com frequência de diferentes *níveis de refinamento* antes que possa ser obtido um algoritmo claro, preciso e completo.

O problema de cálculo da circunferência e superfície de um círculo pode ser decomposto em subproblemas mais simples: (1) ler dados de entrada, (2) calcular superfície e comprimento da circunferência e (3) escrever resultados (dados de saída).

Subproblemas	Refinamento
ler raio	ler raio
calcular superfície	$\text{superfície} = 3,141592 * \text{raio}^2$
calcular circunferência	$\text{circunferência} = 2 * 3,141592 * \text{raio}$
escrever resultados	escrever raio, circunferência, superfície

As *vantagens* mais importantes do projeto descendente são:

- O problema é compreendido mais facilmente ao ser dividido em partes mais simples denominadas *módulos*.
- As modificações nos módulos são mais fáceis.
- A verificação do problema pode ser feita com maior facilidade.

Após os passos anteriores (*projeto descendente e refinamento por passos*), é preciso representar o algoritmo por meio de uma determinada ferramenta de programação: *diagrama de fluxo*, *pseudocódigo* ou *diagrama N-S*.

Assim, o projeto do algoritmo se decompõe nas fases vistas na Figura 2.10.

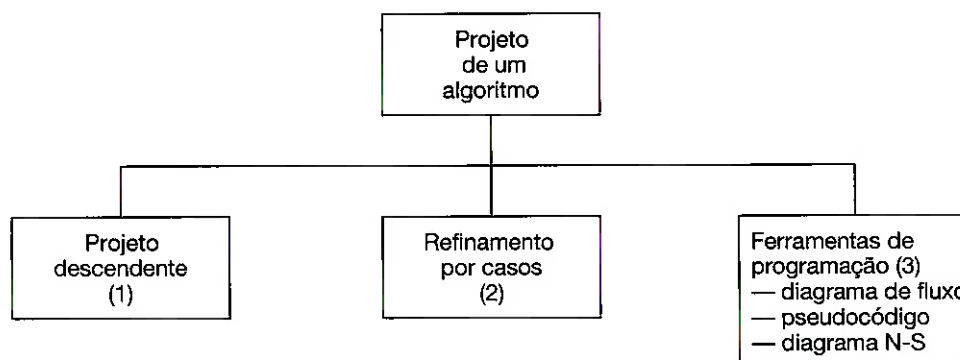


Figura 2.10 Fases do projeto de um algoritmo.

## 2.5 ESCRREVENDO ALGORITMOS

Conforme comentamos anteriormente, o sistema para descrever (“escrever”) um algoritmo consiste em realizar uma descrição passo-a-passo em uma linguagem natural do algoritmo. Recordemos que um algoritmo é um método ou conjunto de regras para solucionar um problema. Em cálculos elementares, essas regras têm as seguintes propriedades:

- deve ser seguida uma sequência definida de passos até que se obtenha um resultado coerente,
- somente pode ser executada uma operação por vez.

O fluxo de controle usual de um algoritmo é sequencial: consideremos o algoritmo que responde a pergunta:

*O que fazer para ver o filme O Homem Aranha?*

A resposta é muito simples e pode ser descrita em forma de algoritmo geral de modo similar a:

```
ir ao cinema
comprar uma entrada
ver o filme
voltar para casa
```

O algoritmo são quatro ações básicas, cada uma das quais deve ser executada antes da seguinte. Em termos de computador, cada ação será codificada em uma ou várias sentenças que executam uma tarefa particular.

O algoritmo descrito é muito simples, entretanto, como indicado em parágrafos anteriores, o algoritmo geral será decomposto em passos mais simples em um procedimento denominado *refinamento sucessivo*, já que cada ação pode ser decomposta por sua vez em outras ações simples. Assim, por exemplo, um primeiro refinamento do algoritmo ir ao cinema pode ser descrito da seguinte maneira:

```
1. início
2. ver os filmes que estão em cartaz pelo jornal
3. se não projetam "O Homem Aranha" então
    3.1 decidir outra atividade
    3.2 ir para o passo 7
    se não
    3.3 ir ao cinema
    fim_se
4. se tem fila então
    4.1 entrar nela
    4.2 enquanto há pessoas à frente fazer
        4.2.1. avançar na fila
    fim_enquanto
    fim_se
5. se há lugares então
    5.1 comprar uma entrada
    5.2 passar para a sala
    5.3 localizar a poltrona
    5.4 enquanto projetam o filme fazer
        5.4.1. ver o filme
    fim_enquanto
    5.5 sair do cinema
    se não
    5.6 reclamar
    fim_se
6. voltar para casa
7. fim.
```

No algoritmo anterior, há diferentes aspectos a considerar. Em primeiro lugar, certas palavras reservadas são escritas deliberadamente em negrito (**enquanto**, **senão** etc.). Essas palavras descrevem as estruturas de controle fundamentais e os processos de tomada de decisão no algoritmo, que incluem os conceitos importantes de *seleção* (expressos pelo **se-então-senão**, *if-then-else*) e de *repetição* (expressos com **enquanto\_fazer** ou às vezes **repetir-até** e **iterar-fim\_iterar**, em inglês *while-do* e *repeat-until*) que se encontram em quase todos os algoritmos, especialmente os do processamento de dados. A capacidade de decisão permite selecionar alternativas de ações a serem seguidas ou a repetição de operações básicas.

```

se projetam o filme selecionado ir ao cinema
  senão assistir à televisão, ir ao futebol ou ler o jornal

enquanto há pessoas na fila, ir avançando repetidamente
  até chegar à bilheteria

```

Outro aspecto a ser considerado é o método escolhido para descrever os algoritmos: emprego de *indentação* na escrita de algoritmo. Atualmente é tão importante a escrita do programa como sua posterior leitura. Ele é facilitado pela *indentação* das ações interiores às estruturas fundamentais citadas: seletivas e repetitivas. Em todo o livro, a indentação dos algoritmos será norma constante.

Para terminar essas considerações iniciais sobre algoritmos, descreveremos as ações necessárias para refinar o algoritmo de nosso estudo; para isso analisaremos a ação

Localizar a(s) poltrona(s)

Se os números dos assentos estão impressos na entrada, a ação composta é resolvida com o seguinte algoritmo:

```

1. início // algoritmo para encontrar a poltrona do espectador
2. caminhar até chegar à primeira fileira de poltronas
3. repetir
    comparar o número da fileira com o número impresso na entrada
    se não são iguais, então passar para a fileira seguinte
    até_que se localize a fileira correta
4. enquanto número de poltrona não coincidir com número de entrada
    fazer avançar pela fileira à poltrona seguinte
    fim_enquanto
5. sentar-se na poltrona
6. fim

```

Nesse algoritmo a repetição é mostrada de dois modos, utilizando ambas as notações, **repetir... até\_que** e **enquanto... fim\_enquanto**. Considerou-se também, como usual, que os números do assento e da fileira coincidem com os números marcados na entrada.

## 2.5.1 Representação gráfica dos algoritmos

Para representar um algoritmo, devemos utilizar algum método que permita tornar independente o algoritmo da linguagem de programação escolhida. Ele permitirá que um algoritmo possa ser codificado indistintamente em qualquer linguagem. Para conseguir esse objetivo, é preciso que o algoritmo seja representado gráfica ou numericamente, de modo que as sucessivas ações não dependam da sintaxe de nenhuma linguagem de programação, mas que a descrição possa servir facilmente para sua transformação em um programa, ou seja, *sua codificação*.

Os métodos usuais para representar um algoritmo são:

1. Diagrama de fluxo.
2. Diagrama N-S (Nassi-Schneiderman).
3. Linguagem de especificação de algoritmos: pseudocódigo.
4. Linguagem em espanhol, inglês...
5. Fórmulas.

Os métodos 4 e 5 não costumam ser fáceis de transformar em programas. Uma descrição em *espanhol narrativo* não é satisfatória, já que é demasiado prolixa e geralmente ambígua. Uma *fórmula*, entretanto, é um bom sistema de representação. Por exemplo, as fórmulas para a solução de uma equação de 2º grau são um meio sucinto de expressar o procedimento algorítmico que deve ser executado para a obtenção das raízes da equação.

$$x1 = (-b + \sqrt{b^2 - 4ac})/2a$$

$$x2 = (-b - \sqrt{b^2 - 4ac})/2a$$

e significa o seguinte:

1. Eleve *b* ao quadrado.
2. Tomar *a*; multiplicar por *c*; multiplicar por 4.
3. Subtrair o resultado obtido de 2 do resultado de 1 etc.

Entretanto, não é freqüente que um algoritmo possa ser expresso por meio de uma simples fórmula.

## 2.5.2 Diagramas de fluxo

Um **diagrama de fluxo** (*flowchart*) é uma das técnicas de representação de algoritmos mais antigas e a mais utilizada, mesmo que seu uso tenha diminuído consideravelmente, sobretudo depois da aparição de linguagens de programação estruturadas. Um diagrama de fluxo é um diagrama que utiliza os símbolos (caixas) padrão mostrados na Tabela 2.2 e que tem os passos de algoritmos escritos nessas caixas unidas por setas, denominadas *linhas de fluxo*, que indicam a seqüência que deve ser executada.

A Figura 2.11 é um diagrama de fluxo básico. Esse diagrama representa a resolução de um programa que deduz o salário líquido de um trabalhador partindo da leitura do nome, das horas trabalhadas, do valor da hora e sabendo que os impostos aplicados são 25% sobre o salário bruto.

Os símbolos-padrão padronizados pela ANSI (abreviatura de American National Standards Institute) são muito variados. A Figura 2.3 representa uma planilha de desenho típica na qual se observa a maioria dos símbolos utilizados no diagrama. Os símbolos mais empregados estão explicados a seguir.

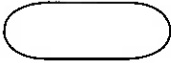
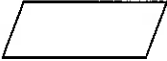
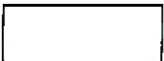
### Símbolos de diagramas de fluxo

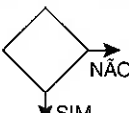
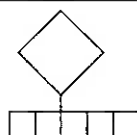



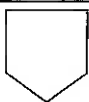


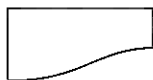
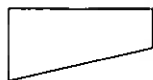
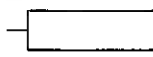
Cada símbolo visto anteriormente indica o *tipo de operação a ser executado* e o diagrama de fluxo ilustra graficamente a *seqüência na qual as operações são executadas*.

As *linhas de fluxo* (→) representam o fluxo seqüencial da lógica do programa.

Um retângulo (□) significa algum tipo de *processamento* no computador, ou seja, ações a serem realizadas (somar dois números, calcular a raiz quadrada de um número etc.).

**Tabela 2.2** Símbolos de diagrama de fluxo

Símbolos principais	Função
	Terminal (representa o começo, “início”, e o final, “fim” de um programa. Pode representar também uma parada ou interrupção programada que seja necessária em um programa.
	Entrada/Saída (qualquer tipo de introdução de dados na memória dos periféricos, “entrada”, ou registro da informação processada em um periférico, “saída”.
	Processamento (qualquer tipo de operação que possa originar mudança de valor, formato ou posição da informação armazenada na memória, operações aritméticas de transferência etc.).

	Decisão (indica operações lógicas ou de comparação entre dados — normalmente dois — e, em função do resultado, determina qual dos diferentes caminhos alternativos do programa seguir; normalmente tem duas saídas — respostas Sim ou Não — mas pode ter três ou mais, de acordo com o caso).
	Decisão múltipla (em função do resultado da comparação se seguirá um dos diferentes caminhos de acordo com o resultado).
	Conector (serve para ligar duas partes quaisquer de um organograma por meio de um conector na saída e outro conector na entrada. Refere-se à conexão na mesma página do diagrama).
	Indicador de sentido ou linha de fluxo (indica o sentido de execução das operações).
	Linha conectora (serve de união entre os símbolos).
	Conector (conexão entre os pontos do organograma situados em páginas diferentes).
	Chamada de sub-rotina ou de um processo predeterminado (sub-rotina é um módulo independente do programa principal, que recebe uma entrada procedente do programa, realiza uma tarefa determinada e regressa, ao terminar, ao programa principal).
	Tela (utiliza-se em casos específicos no lugar do símbolo de E/S).
	Impressora (utiliza-se em casos específicos no lugar do símbolo E/S).
	Teclado (utiliza-se em casos específicos no lugar do símbolo E/S).
	Comentários (utiliza-se para acrescentar comentários classificadores a outros símbolos do diagrama de fluxo. Pode ser desenhado em qualquer lado do símbolo).

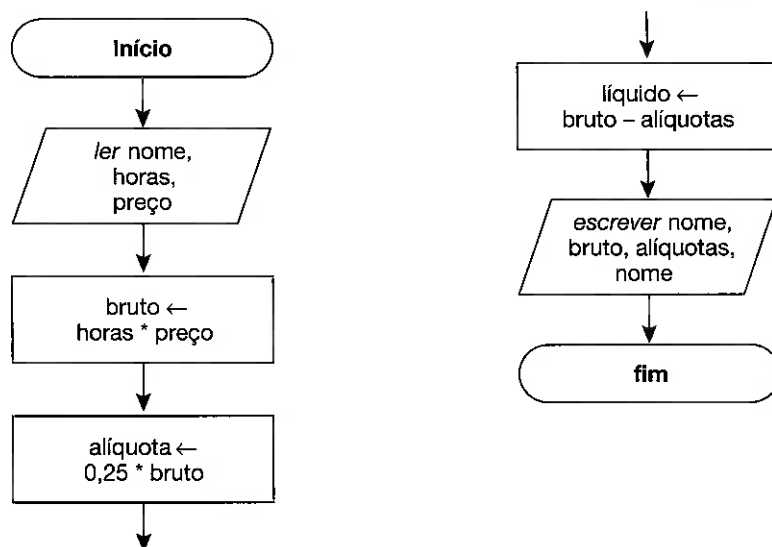





Figura 2.11 Diagrama de fluxo.

O paralelogramo () é um símbolo de entrada/saída que representa qualquer tipo de *entrada* ou *saída* do programa ou sistema; por exemplo, entrada do teclado, saída na impressora ou tela etc.

O símbolo losango () é uma caixa de decisão que representa respostas sim/não ou diferentes alternativas 1, 2, 3, 4... etc.

Cada diagrama de fluxo começa e termina com um símbolo terminal ()

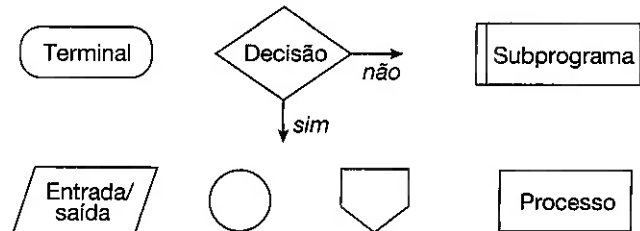










Figura 2.12 Símbolos mais utilizados em um diagrama de fluxo.

Um pequeno círculo () é um *conector* utilizado para conectar caminhos, seguindo rotas prévias do fluxo de algoritmos.

Outros símbolos de diagramas menos utilizados de mais detalhes que os anteriores são:

Um trapézio () indica que um *processo manual* vai ser executado em contraste com o retângulo que indica processo automático.

O símbolo geral de entrada/saída pode ser subdividido em outros símbolos: *teclado* () , *tela* () , *impressora* () , *disco magnético* () , *disquete* ou *disco flexível* () , *cassete* () .

O refinamento do algoritmo leva aos passos sucessivos necessários para a realização das operações de leitura, verificação do último dado, soma e média dos dados.

Se o primeiro dado lido é 0, a divisão  $S/C$  produzirá um erro no computador, já que não é permitida a divisão por zero.

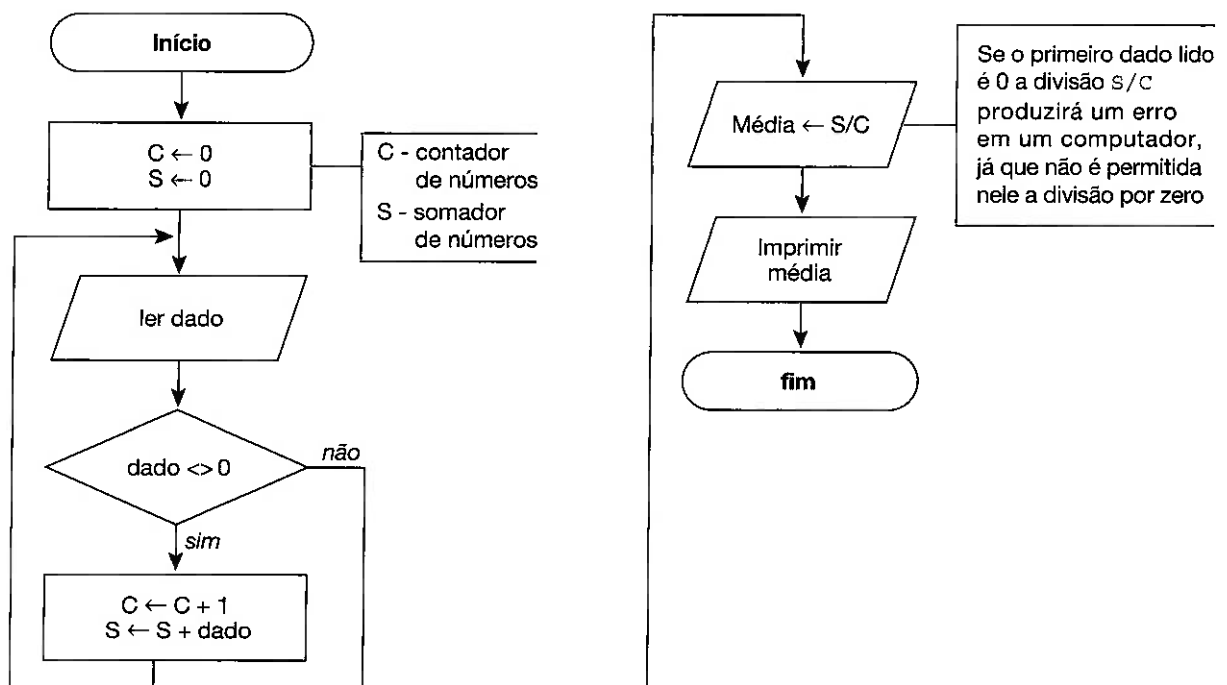
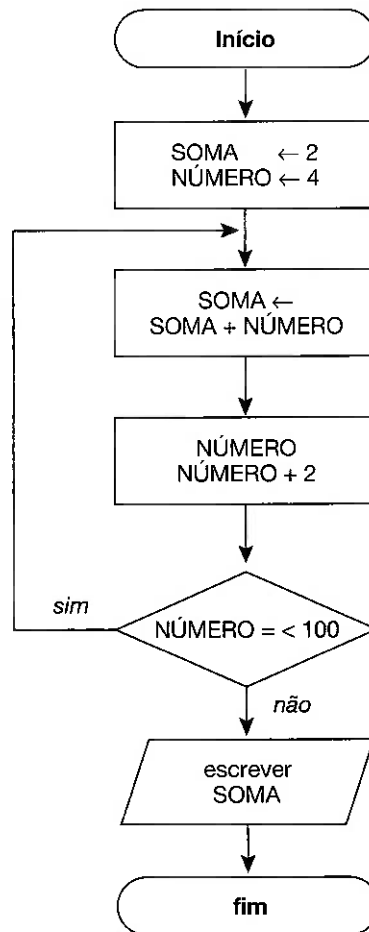


Figura 2.13

**Exemplo 2.6**

Soma dos números pares compreendidos entre 2 e 100.

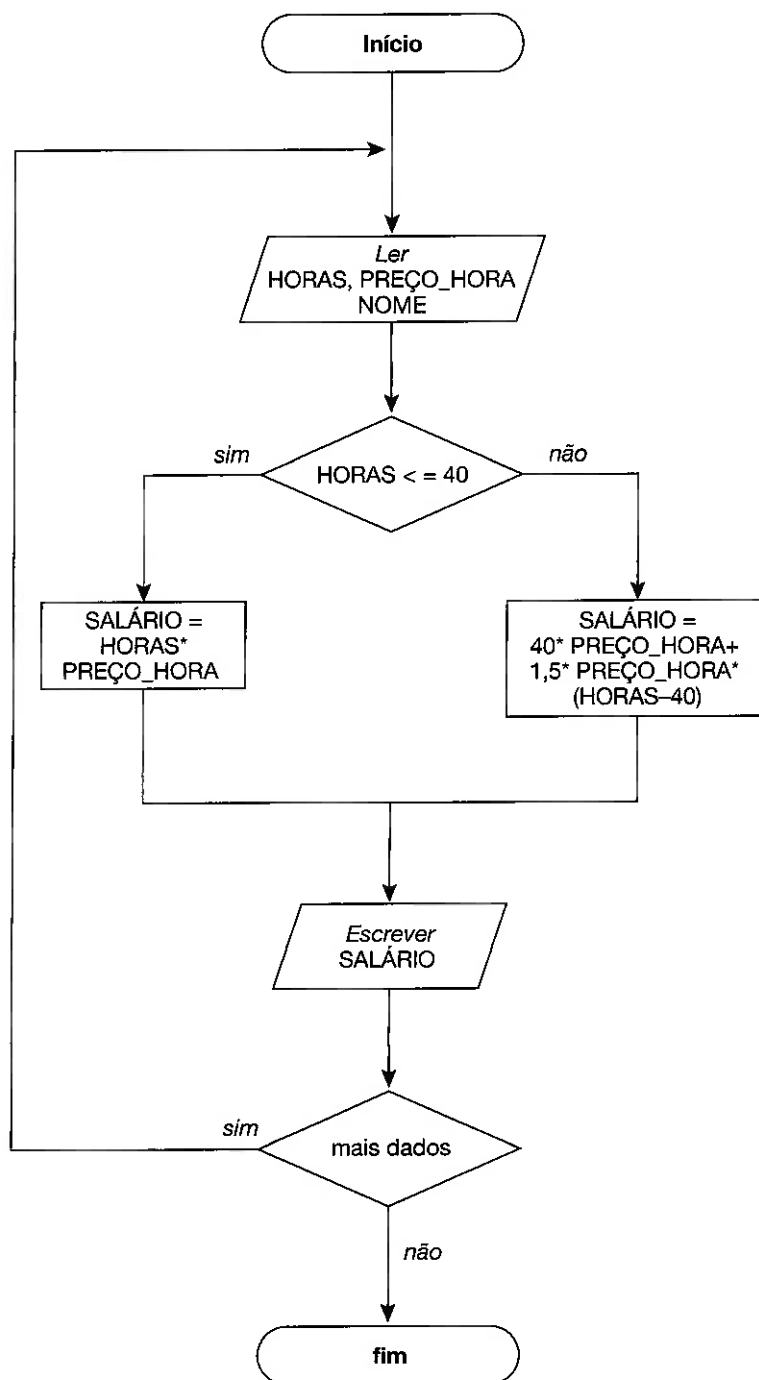
**Exemplo 2.7**

Desejamos construir o algoritmo que resolva o seguinte problema: cálculo dos salários mensais dos empregados de uma empresa, sabendo que são calculados com base nas horas semanais trabalhadas e de acordo com um preço especificado por hora. Se passarem de 40 horas semanais, as horas extras serão pagas na razão de 1,5 vez a hora normal.

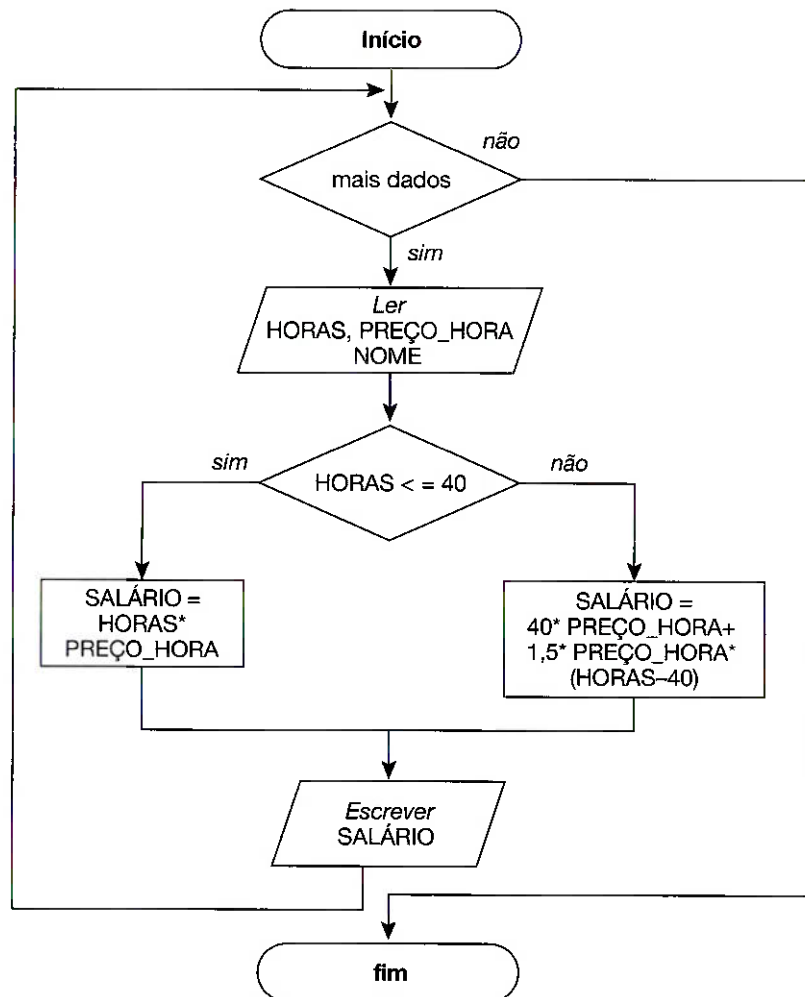
Os cálculos são:

1. Ler dados do arquivo da empresa, até que se encontre a ficha final do arquivo (HORAS, PREÇO\_HORA, NOME).
2. Se  $HORAS \leq 40$ , então SALÁRIO é o produto de horas por PREÇO\_HORA.
3. Se  $HORAS > 40$ , então SALÁRIO é a soma de 40 vezes PREÇO\_HORA mais 1,5 vez PREÇO\_HORA por  $(HORAS - 40)$ .

O diagrama de fluxo completo do algoritmo é apresentado a seguir:



Uma variante também válida para o diagrama de fluxo anterior é:



### Exemplo 2.8

A escrita de algoritmos para a realização de operações simples de contagem é uma das primeiras coisas que se pode aprender.

Suponhamos uma seqüência de números, como:

5 3 0 2 4 4 0 0 2 3 6 0 2

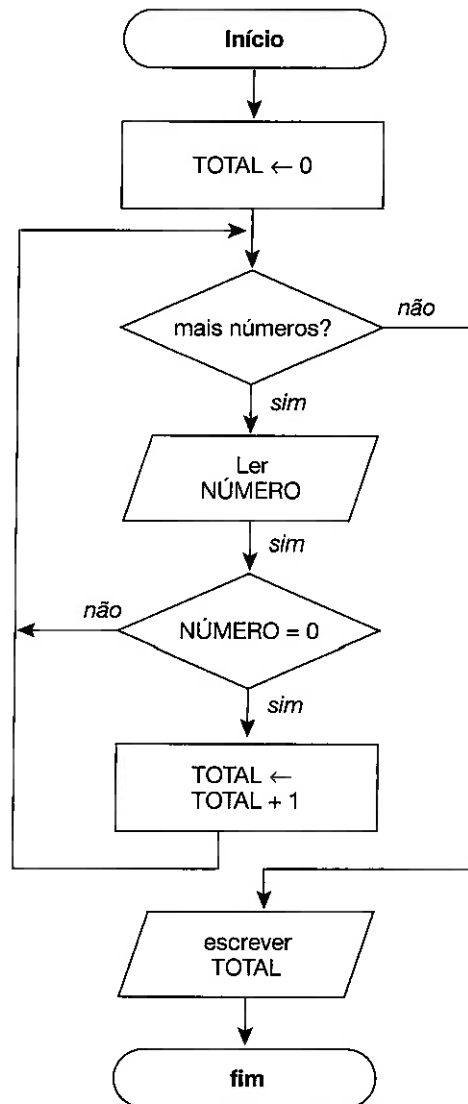
Deseja-se contar e imprimir o número de zeros da seqüência.

O algoritmo é muito simples, já que basta ler os números da esquerda para a direita, enquanto se contam os zeros. Utiliza como variável a palavra **NÚMERO** para os números examinados e **TOTAL** para o número de zeros encontrados. Os passos são:

1. Estabelecer **TOTAL** em zero.
2. Ficam mais números para examinar?
3. Se não ficam números, imprimir o valor do **TOTAL** e fim.

4. Se existem mais números, executar os passos 5 a 8.
5. Ler o seguinte número e dar seu valor à variável NÚMERO.
6. Se NÚMERO = 0, incrementar TOTAL em 1.
7. Se NÚMERO < > 0, não modificar TOTAL.
8. Retornar ao passo 2.

O diagrama de fluxo correspondente é:



### Exemplo 2.9

Dados três números, determinar se a soma de algum par é igual ao terceiro número. Se for cumprida essa condição, escrever "Iguais", caso contrário, escrever "Diferentes".

No caso de os números serem: 3 9 6

a resposta é "Iguais", já que  $3 + 6 = 9$ . Entretanto, se os números forem:

2 3 4

o resultado será "Diferentes".

Para resolver esse problema, podemos comparar a soma de cada par com o terceiro número. Com três números somente existem três pares diferentes, e o algoritmo de resolução do problema será fácil.

1. Ler os três valores, A, B e C.
2. Se  $A + B = C$  escrever "Iguais" e parar.
3. Se  $A + C = B$  escrever "Iguais" e parar.
4. Se  $B + C = A$  escrever "Iguais" e parar.
5. Escrever "Diferentes" e parar.

O diagrama de fluxo correspondente é a Figura 2.14.

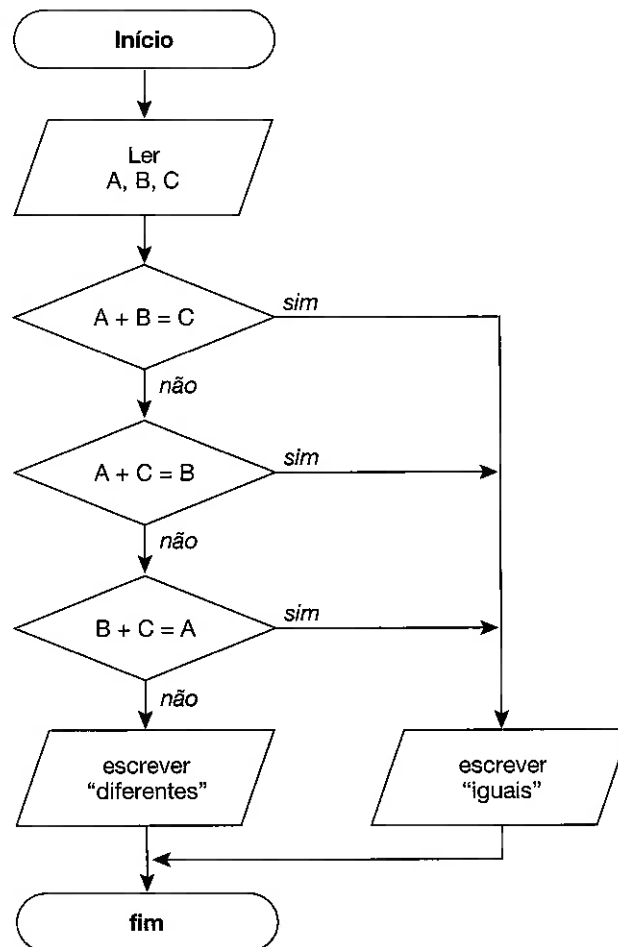


Figura 2.14 Diagrama de fluxo (Exemplo 2.9).

### Exemplo 2.10

Desejamos calcular o salário líquido semanal de um trabalhador em função do número de horas trabalhadas e da alíquota de impostos.

- as primeiras 35 horas são pagas em tarifa normal;
- as horas que passam de 35 horas são pagas 1,5 vez a tarifa normal;
- as alíquotas de impostos são:
  - a) os primeiros 60.000 reais são livres de impostos;

- b) os seguintes 40.000 reais têm 25% de imposto;
- c) os restantes, 45% de imposto;
- a tarifa horária é 800 reais.

Desejamos também escrever o nome, o salário bruto, as alíquotas e o salário líquido (*este exemplo é um exercício para o aluno*).

### 2.5.3 Pseudocódigo

O pseudocódigo é uma linguagem de especificação (*descrição*) de algoritmos. O uso dessa linguagem torna a codificação final (isto é, a tradução para uma linguagem de programação) relativamente fácil. As linguagens APL, Pascal e Ada são utilizadas às vezes como linguagem de especificação de algoritmos.

O pseudocódigo nasceu como uma linguagem similar ao inglês e era um meio de representar basicamente as estruturas de controle de programação estruturada que serão vistas em capítulos posteriores. É considerado um *primeiro rascunho*, dado que o pseudocódigo tem de ser traduzido depois para uma linguagem de programação. O pseudocódigo não pode ser executado por um computador. A *vantagem do pseudocódigo* é que em seu uso, no planejamento de um programa, o programador pode concentrar-se na lógica e nas estruturas de controle e não se preocupar com as regras de uma linguagem específica. É também fácil modificar o pseudocódigo quando são descobertos erros ou anomalias na lógica do programa, enquanto em muitas ocasiões pode ser difícil a mudança na lógica depois da codificação em uma linguagem de programação. Outra vantagem do pseudocódigo é que pode ser traduzido facilmente para linguagens estruturadas como Pascal, C, FORTRAN 77/90, C++, Java, C# etc.

O pseudocódigo original utiliza palavras reservadas em inglês para representar as ações sucessivas — similares a seus homônimos nas linguagens de programação —, como **start**, **end**, **stop**, **if-then-else**, **while-end**, **repeat-until** etc. A escrita do pseudocódigo exige normalmente a *indentação* de diferentes linhas.

A representação em pseudocódigo do diagrama de fluxo da Figura 2.11 é a seguinte:

```
start
  // cálculo de imposto e salários
  read nome, horas, preço_hora
  salário_bruto ← horas * preço_hora
  alíquotas ← 0,25 * salário_bruto
  salário_líquido ← salário_bruto - alíquotas
  write nome, salário_bruto, alíquotas, salário_líquido
end
```

O algoritmo começa com a palavra **start** e finaliza com a palavra **end**, em inglês (em português *início*, *fim*). Entre essas palavras, somente se escreve uma instrução ou ação por linha.

A linha precedida por **//** é denominada *comentário*. É uma informação para o leitor do programa e não realiza nenhuma instrução executável, apenas tem efeito de documentação interna do programa. Alguns autores costumam utilizar colchetes ou chaves.

Não é recomendável o uso de apóstrofes ou aspas simples para os comentários como no BASIC da Microsoft, já que esse caractere é representativo de abertura ou encerramento de cadeias de caracteres em linguagens como Pascal ou FORTRAN, e daria lugar a confusão.

Outro exemplo que deixa claro o uso do pseudocódigo poderia ser um simples algoritmo de partida matinal de um carro.

```

início
  // partida matinal de um carro
  introduzir a chave de contato
  girar a chave de contato
  pisar no acelerador
  ouvir o ruído do motor
  pisar de novo no acelerador
  esperar uns instantes para que o motor esquente
fim

```

Embora o pseudocódigo tenha nascido como um substituto da linguagem de programação e, conseqüentemente, suas palavras reservadas se tenham conservado muito similares às dessas linguagens, praticamente o inglês, o uso do pseudocódigo estendeu-se na comunidade de língua portuguesa com termos em português como **início**, **fim**, **parada**, **ler**, **escrever**, **sim-então-sim\_não**, **enquanto**, **fim\_enquanto**, **repetir**, **até\_que** etc. Sem dúvida, o uso da terminologia do pseudocódigo em português facilitou e facilitará consideravelmente a aprendizagem e o uso diário da programação. Nesta obra, assim como em outras nossas, utilizaremos o pseudocódigo em português e daremos a cada momento as estruturas equivalentes em inglês, para facilitar a tradução do pseudocódigo para a linguagem de programação selecionada.

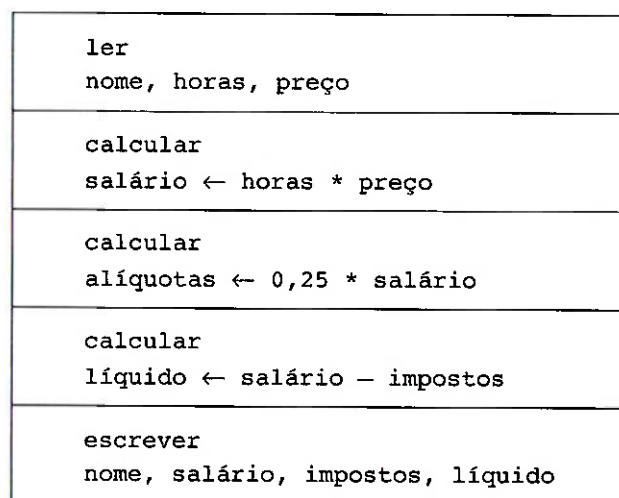
Assim, nos pseudocódigos citados deveriam ser substituídas as palavras **start**, **end**, **read**, **write** por **início**, **fim**, **ler**, **escrever**, respectivamente.

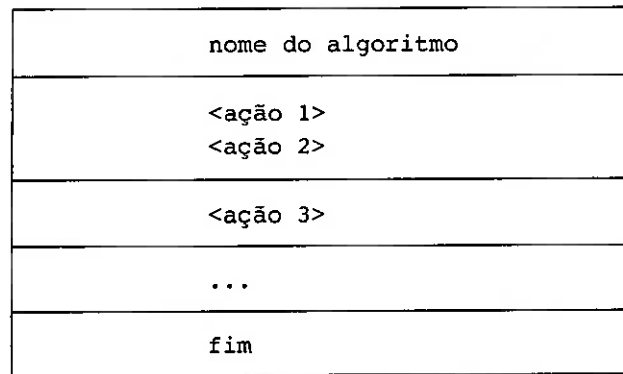
<i>início</i>	<i>start</i>	<i>ler</i>	<i>read</i>
.			
.			
.			
.			
.			
<i>fim</i>	<i>end</i>	<i>escrever</i>	<i>write</i>

## 2.5.4 Diagramas de Nassi-Schneiderman (N-S)

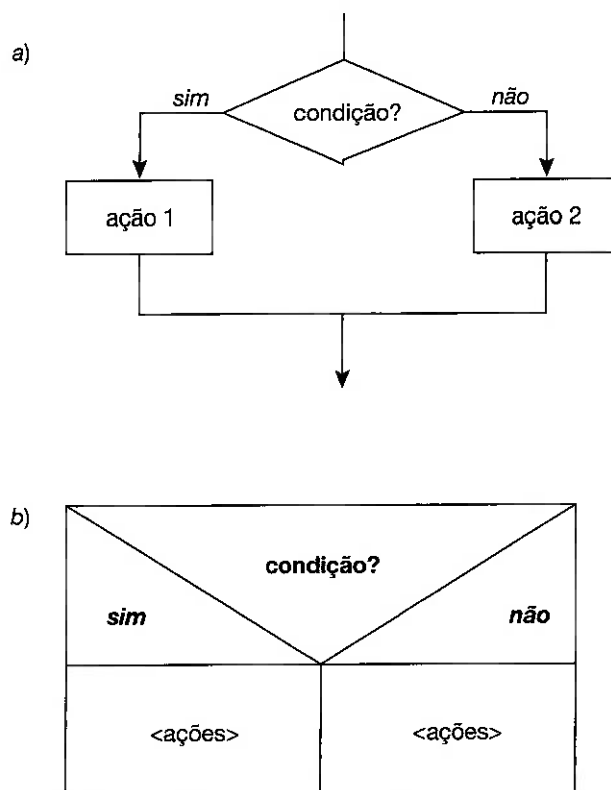
O diagrama N-S de Nassi-Schneiderman — também conhecido como diagrama de Chapin — é como um diagrama de fluxo no qual são omitidas as setas de união e as caixas são contínuas. As ações sucessivas são escritas em caixas sucessivas e, como nos diagramas de fluxo, podemos escrever diferentes ações em uma caixa.

Um algoritmo é representado com um retângulo em que cada lado é uma ação a ser realizada:



**Figura 2.15** Representação gráfica N-S de um algoritmo.

Outro exemplo é a representação da estrutura condicional (Figura 2.16).

**Figura 2.16** Estrutura condicional ou seletiva: a) diagrama de fluxo; b) diagrama N-S.

## ATIVIDADES DE PROGRAMAÇÃO RESOLVIDAS

Desenvolva os algoritmos que resolvam os seguintes problemas:

### 2.1 Ir ao cinema.

*Análise do problema*

DADOS DE SAÍDA: ver o filme  
 DADOS DE ENTRADA: nome do filme, endereço da sala, hora da projeção  
 DADOS AUXILIARES: entrada, número de poltronas

Para solucionar o problema, devemos seleccionar um filme em cartaz no jornal, ir à sala e comprar a entrada para, finalmente, poder assistir ao filme.

*Projeto do algoritmo*

**início**

```
< seleccionar o filme >
ver o jornal
enquanto não chegamos à programação em cartaz
  virar a folha
enquanto não se acabe a programação em cartaz
  ler sobre o filme
se gostamos, recordá-lo
fim enquanto
escolher um dos filmes seleccionados
ler o endereço da sala e a hora de projeção
< comprar a entrada >
dirigir-se à sala
se não há entradas, ir ao fim
se há fila então
  ser o último
enquanto não chegamos à bilheteria
  avançar
  se não há entradas, ir ao fim
fim enquanto
comprar a entrada
< ver o filme >
ler o número de poltrona na entrada
buscar a poltrona
sentar-se
ver o filme
```

**fim.**

### 2.2 Comprar uma entrada para ir à tourada.

*Análise do problema*

DADOS DE SAÍDA: a entrada  
 DADOS DE ENTRADA: tipo de entrada (sol, sombra, arquibancada...)  
 DADOS AUXILIARES: disponibilidade da entrada

Tem de ir à bilheteria e escolher a entrada desejada. Se há entradas, comprar (na bilheteria ou em postos de revenda). Se não há, podemos seleccionar outro tipo de entrada ou desistir, repetindo essa ação até que se tenha conseguido a entrada ou o possível comprador tenha desistido.

#### *Projeto de algoritmo*

```
início  
ir à bilheteria  
se não há entradas na bilheteria  
se nos interessa comprá-la em postos de revenda  
    ir comprar a entrada  
se não ir ao fim  
< comprar a entrada >  
seleccionar sol ou sombra  
seleccionar geral, arquibancada, ou palco  
seleccionar número do assento  
solicitar a entrada  
se não há disponibilidade então  
    adquirir a entrada  
    se não  
        se queremos outro tipo de entrada então  
            ir comprar a entrada  
fim.
```

### 2.3 Fazer uma xícara de chá.

#### *Análise do problema*

DADOS DE SAÍDA:      xícara de chá  
DADOS DE ENTRADA:    saquinho de chá, água  
DADOS AUXILIARES:    apito da chaleira, aspecto da infusão

Depois de despejar a água na chaleira, levamos ao fogo e esperamos que a água ferva (até que soe o apito da chaleira). Introduzimos o chá e deixamos um tempo até que fique pronto.

#### *Projeto do algoritmo*

```
início  
    pegar a chaleira  
    colocar água  
    acender o fogo  
    colocar a chaleira no fogo  
    enquanto não ferve a água  
        esperar  
    fim enquanto  
    pegar o saquinho de chá  
    introduzi-lo na chaleira  
    enquanto não está pronto o chá  
        esperar  
    fim enquanto  
    despejar o chá na xícara  
fim.
```

## 2.4 Fazer uma chamada telefônica. Considerar os casos: a) chamada manual com operador; b) chamada automática; c) chamada a cobrar.

### Análise do problema

Para decidir o tipo de chamada que será efetuado, primeiro devemos considerar se dispomos de dinheiro ou não — para realizar a chamada a cobrar. Tendo o dinheiro, devemos ver se o lugar para onde vamos fazer a chamada está conectado à rede automática ou não.

Para uma chamada com operador, temos de chamar a central e solicitar a chamada, esperando até que a comunicação seja estabelecida. Para uma chamada automática, lemos os prefixos do país e cidade se for necessário e realizamos a chamada, esperando até que atendam ao telefone. Para chamar a cobrar, devemos chamar a central, solicitar a chamada e esperar que o proprietário do telefone chamado dê sua autorização, com isso a comunicação será estabelecida.

Como dados de entrada, teríamos as variáveis que condicionam o tipo de chamada, o número de telefone e, em caso de chamada automática, os prefixos. Como dado auxiliar, poderíamos considerar os casos a) e c) o contato com a central.

### Projeto do algoritmo

#### início

```

se temos dinheiro então
  se podemos fazer uma chamada automática então
    ler o prefixo do país e localidade
    marcar o número
  se não
    < chamada manual >
    chamar a central
    solicitar a comunicação
  fim enquanto não atendem fazer
    esperar
  fim enquanto
  estabelecer comunicação
se não
  < realizar uma chamada a cobrar >
  chamar a central
  solicitar a chamada
  esperar até ter a autorização
  estabelecer comunicação
fim se

```

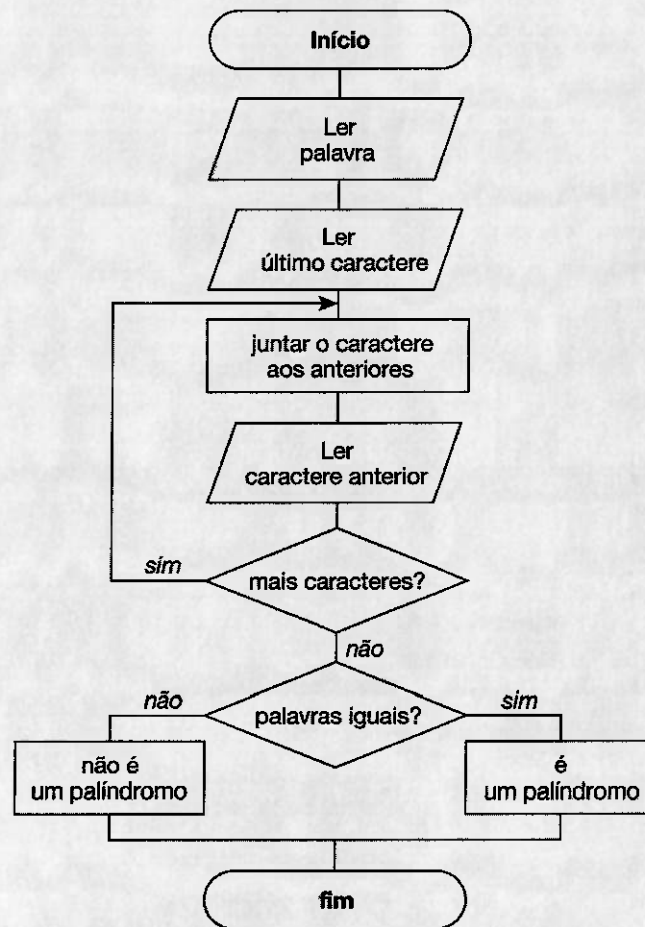
**fim.**

## 2.5 Averiguar se uma palavra é um palíndromo. Um palíndromo é uma palavra lida da mesma maneira da esquerda para a direita e da direita para a esquerda, por exemplo, "radar".

### Análise do problema

DADOS DE SAÍDA: a mensagem que nos diz se é ou não um palíndromo  
 DADOS DE ENTRADA: palavra  
 DADOS AUXILIARES: cada caractere da palavra, palavra ao contrário

Para comprovar se uma palavra é um palíndromo, podemos formar uma palavra com os caracteres invertidos com respeito à original e comprovar se a palavra ao contrário é igual à original. Para se obter essa palavra ao contrário, serão lidos em sentido inverso os caracteres da palavra inicial e os juntaremos sucessivamente até chegar ao primeiro caractere.

*Projeto do algoritmo*

2.6 Projetar um algoritmo para calcular a velocidade (em metros/segundo) dos corredores de uma corrida de 1.500 metros. A entrada serão pares de números (minutos, segundos) que darão o tempo de cada corredor. Para cada corredor, usaremos o tempo em minutos e segundos, assim como a velocidade média. O laço será executado até que tenhamos uma entrada de 0,0 que será a marca de fim de entrada de dados.

*Análise do problema*

DADOS DE SAÍDA:  $v$  (velocidade média)

DADOS DE ENTRADA:  $mm, ss$  (minutos e segundos)

DADOS AUXILIARES: *distância* (distância percorrida, que no exemplo é de 1.500 metros) e *tempo* (os minutos e os segundos gastos no percurso)

Devemos efetuar um laço até que  $mm$  seja 0 e  $ss$  seja 0. Dentro do laço calculamos o tempo em segundos com a fórmula  $\text{tempo} = ss + mm * 60$ . A velocidade será encontrada com a fórmula.

$$\text{velocidade} = \text{distância} / \text{tempo}$$

*Projeto do algoritmo*

```

início
    distância ← 1500
    ler (mm,ss)
    enquanto mm = 0 e ss = 0 fazer
        tempo ← ss + mm * 60
        v ← distância/tempo
        escrever (mm, ss, v)
        ler (mm, ss)
    fim enquanto
fim

```

**2.7** Escrever um algoritmo que calcule a superfície de um triângulo em função da base e da altura.

*Análise do problema*

DADOS DE SAÍDA: s (superfície)

DADOS DE ENTRADA: b (base) a (altura)

Para calcular a superfície se aplica a fórmula:

$$S = \text{base} * \text{altura}/2$$

*Projeto do algoritmo*

```

início
    ler (b, a)
    s = b * a/2
    escrever (s)
fim

```

**2.8** Construir um algoritmo que calcule a soma dos inteiros entre 1 e 10, ou seja,  $1 + 2 + 3 + \dots + 10$ .

*Análise do problema*

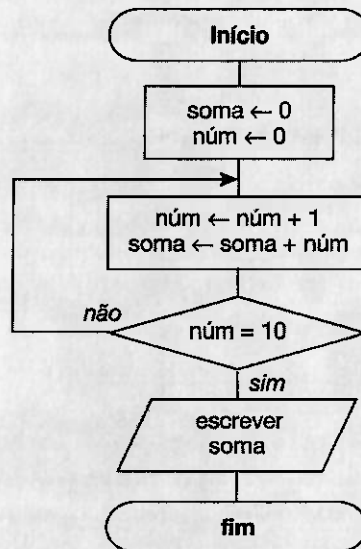
DADOS DE SAÍDA: soma (contém a soma requerida)

DADOS AUXILIARES: núm (será uma variável que recebe valores entre 1 e 10 e se acumulará em soma)

Devemos executar um laço que se realize dez vezes. Nele será incrementado em 1 a variável núm, e será acumulado seu valor na variável soma. Uma vez fora do laço, se visualizará o valor da variável soma.

*Projeto do algoritmo*

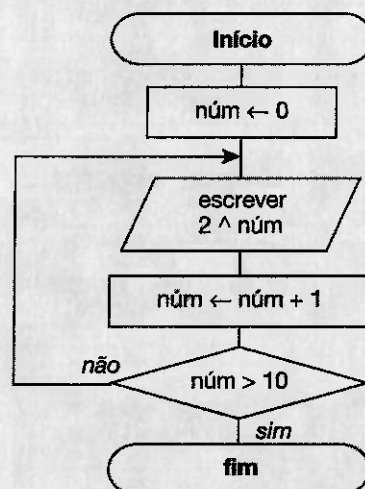
## TABELA DE VARIÁVEIS

**inteiro:** soma, núm**2.9** Construir um algoritmo que calcule e visualize as potências de 2 entre 0 e 10.*Análise do problema*

Devemos implementar um laço que seja executado 11 vezes e dentro dele ir incrementando uma variável que recebe valores entre 0 e 10 e que se chamará núm. Também dentro dele será visualizado o resultado da operação  $2^{\text{núm}}$ .

*Projeto do algoritmo*

## TABELA DE VARIÁVEIS

**inteiro:** núm

2.10 Desejamos obter o salário líquido de um trabalhador conhecendo o número de horas trabalhadas, o salário\_hora e a alíquota de impostos que devemos aplicar como deduções.

As *entradas* do algoritmo são:

horas trabalhadas, salário\_hora, alíquotas

As *saídas* do algoritmo são:

pagamento bruto, total de impostos e pagamento líquido

O algoritmo geral é:

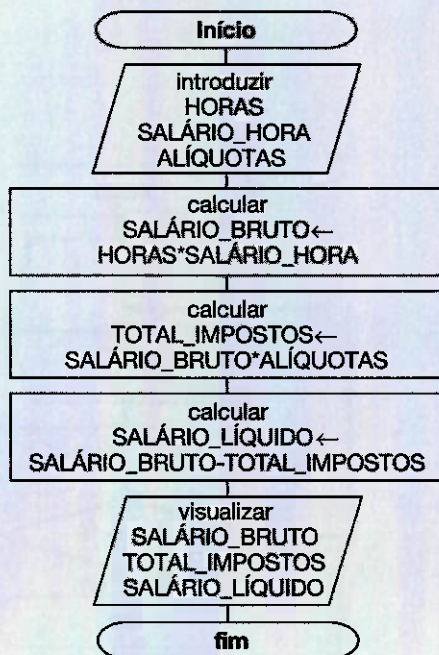
1. Obter valores de horas trabalhadas, salário\_hora e alíquotas.
2. Calcular salário\_bruto, total de impostos e salário\_líquido.
3. Visualizar salário\_bruto, total de impostos e salário\_líquido.

O refinamento do algoritmo em passos de nível inferior é:

1. Obter valores de horas trabalhadas, salário bruto e alíquotas.
2. Calcular salário\_bruto, total de impostos e pagamento líquido.
  - 2.1 Calcular salário\_bruto multiplicando as horas trabalhadas pelo salário\_hora.
  - 2.2 Calcular o total de impostos multiplicando salário\_bruto por alíquotas (tanto por cento de impostos).
  - 2.3 Calcular o salário\_líquido retirando o total de impostos do pagamento bruto.
3. Visualizar salário\_bruto, total de impostos, salário\_líquido.

O diagrama de fluxo a seguir representa esse algoritmo.

*Diagrama de fluxo*



## 2.11 Definir o algoritmo necessário para intercambiar os valores de duas variáveis numéricas.

### Análise do problema

Para realizar essa análise, utiliza-se uma variável denominada auxiliar que recebe um dos valores dados de modo temporário.

Variáveis: A B AUX

O método consiste em atribuir uma das variáveis à variável auxiliar.

AUX  $\leftarrow$  A

A seguir se atribui o valor da outra variável B à primeira:

A  $\leftarrow$  B

Por último, se atribui o valor da variável auxiliar à segunda variável B:

B  $\leftarrow$  AUX

Variáveis:	A	primeiro valor,
	B	segundo valor,
	AUX	variável auxiliar.

### Projeto do algoritmo

#### início

ler (A, B)

AUX  $\leftarrow$  A

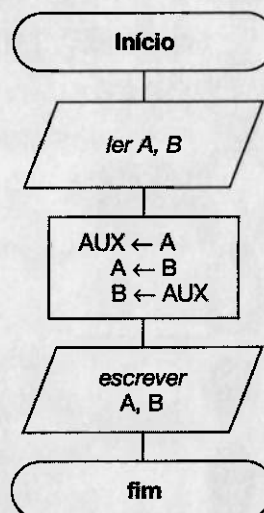
A  $\leftarrow$  B

B  $\leftarrow$  AUX

escrever (A, B)

#### fim

### Diagrama de fluxo



## REVISÃO DO CAPÍTULO

### Conceitos-chave

- Algoritmo
- Ciclo de vida
- Diagrama *Nassi-Schneiderman*
- Diagramas de fluxo
- Programação modular
- Projeto
- Programação estruturada
- Projeto descendente
- Testes
- *Pseudocódigo*
- Fatores de qualidade
- Verificação

### Resumo

Um método geral para a resolução de um problema com computador tem as seguintes fases:

1. *Análise do programa*
2. *Projeto do algoritmo*
3. *Codificação*
4. *Compilação e execução*
5. *Verificação e depuração*
6. *Documentação e manutenção*

O sistema mais seguro para resolver um problema é decompô-lo em módulos mais simples e,

por meio de projetos descendentes e refinamento sucessivo, chegar a módulos facilmente codificáveis. Esses módulos devem ser codificados com as estruturas de controle de programação estruturada.

1. *Sequenciais*: as instruções são executadas uma depois da outra.
2. *Repetitivas*: uma série de instruções são repetidas até que se cumpra uma certa condição.
3. *Seletivas*: permite escolher entre duas alternativas (dois conjuntos de instruções) dependendo de uma condição determinada.

## EXERCÍCIOS

2.1 Deduzir os resultados obtidos do seguinte algoritmo:

```
inteiro: x, y, z
início
  x ← 5
  y ← 20
  z ← x + y
  escrever (x, y)
  escrever (z)
fim
```

2.2 Que resultados produzirá este algoritmo?

```
inteiro: nx, duplo
início
  NX ← 25
  Duplo ← NX * 2
  escrever (NX)
  escrever (DUPLO)
fim
```

- 2.3 Escrever um algoritmo que calcule e escreva o quadrado de 243.
- 2.4 Escrever um algoritmo que leia um número e escreva seu quadrado.
- 2.5 Determinar a área e o volume de um cilindro cujas dimensões de raio e altura sejam lidas do teclado.
- 2.6 Calcular o perímetro e a superfície de um quadrado dado o comprimento de seu lado.
- 2.7 Construir o algoritmo que some dois números.
- 2.8 Calcular a superfície de um círculo.
- 2.9 Calcular o perímetro e a superfície de um retângulo dadas a base e a altura.
- 2.10 Escrever um algoritmo que leia um nome de uma marca de automóveis seguida do nome de seu modelo e informe do modelo seguido do nome.
- 2.11 Determinar a hipotenusa de um triângulo retângulo conhecidos os comprimentos dos catetos.
- 2.12 Projetar um algoritmo que realize a seguinte conversão: uma temperatura dada em graus Celsius para graus Fahrenheit.  
*Nota:* A fórmula de conversão é:  $F = (9/5)C + 32$ .
- 2.13 Projetar um algoritmo que calcule a área de um retângulo em função dos comprimentos de seus lados:  
$$Área = \sqrt{p(p-a)(p-b)(p-c)}$$
onde  $p = (a + b + c)/2$  (semiperímetro).
- 2.14 Deseja-se um algoritmo para converter metros para pés e polegadas (1 metro = 39,37 polegadas, 1 pé = 12 polegadas).
- 2.15 A cotação de moedas na Bolsa de Madri no dia 25 de agosto de 1987 foi a seguinte:
- |                         |               |
|-------------------------|---------------|
| 100 chilins austríacos  | = 14,76 reais |
| 1 dólar norte-americano | = 1,89 reais  |
| 100 dracmas gregos      | = 1,37 reais  |
| 100 francos belgas      | = 4,99 reais  |
| 1 franco francês        | = 0,31 reais  |
| 1 libra esterlina       | = 2,76 reais  |
| 100 liras italianas     | = 0,14 reais  |
- 2.16 Desenvolver algoritmos que realizem as seguintes conversões:
- Ler uma quantidade em chilins austríacos e imprimir o equivalente em pesetas.
  - Ler uma quantidade em dracmas gregos e imprimir o equivalente em francos franceses.
  - Ler uma quantidade em pesetas e imprimir o equivalente em dólares e em liras italianas.
- 2.17 Projetar uma solução para resolver cada um dos seguintes problemas e refinar suas soluções por meio de algoritmos adequados:
- Realizar uma chamada telefônica de um telefone público.
  - Cozinhar uma omelete.
  - Consertar um pneu furado de uma bicicleta.
  - Fritar um ovo.
- 2.18 Escrever um algoritmo para:
- Somar dois números inteiros.
  - Subtrair dois números inteiros.
  - Multiplicar dois números inteiros.
  - Dividir um número inteiro por outro.
- 2.19 Escrever um algoritmo para determinar o máximo divisor comum de dois números inteiros (mdc) pelo algoritmo de Euclides:
- Dividir o maior dos dois inteiros positivos pelo menor.
  - Em seguida, dividir o divisor pelo seu resto.

- Continuar o processo e dividir o último divisor pelo último resto até que a divisão seja exata.
  - O último divisor é o mdc.
- 2.20** Projetar um algoritmo que leia e imprima uma série de números distintos de zero. O algoritmo deve terminar com um valor zero que não deve ser impresso. Visualizar o número de valores lidos.
- 2.21** Projetar um algoritmo que imprima e some a série de números 3, 6, 9, 12..., 99.
- 2.22** Escrever um algoritmo que leia quatro números e imprima o maior dos quatro.
- 2.23** Projetar um algoritmo que leia três números e encontre se um deles é a soma dos outros dois.
- 2.24** Escrever um algoritmo que conte o número de ocorrências de cada letra em uma palavra lida como entrada. Por exemplo, "Mortimer" contém dois "m", um "o", dois "r", um "i" um "t" e um "e".
- 2.25** Muitos bancos e caixas de poupança calculam os juros das quantidades depositadas pelos clientes diariamente usando as seguintes premissas. Um capital de 1.000 reais, com uma taxa de juros de 6%, rende juros em um dia de 0,06 multiplicado por mil e dividido por 365. Esta operação produzirá 0,16 reais de juros e o capital acumulado será 1.000,16. Os juros para o segundo dia serão calculados multiplicando-se 0,06 por 1.000 e dividindo o resultado por 365. Projetar um algoritmo que receba três entradas: o capital para depósito, a taxa de juros e a duração do depósito em semanas, e calcular o capital total acumulado no final do período especificado.