Capítulo 1

Introdução à Programação

It has often been said that a person does not really understand something until after teaching it to someone else. Actually, a person does not really understand something until after teaching it to a computer (...)

Donald E. Knuth, *Selected Papers in Computer Science*, 10 (1996)

1.1 Computadores

O que é um computador? As respostas que surgem com mais frequência são que um computador é um conjunto de circuitos integrados, uma ferramenta ou uma máquina programável. Todas estas respostas são verdadeiras, até certo ponto. No entanto, um rádio também é um conjunto de circuitos integrados, um martelo uma ferramenta, e uma máquina de lavar roupa uma máquina programável... Algo mais terá de se usar para distinguir um computador de um rádio, de um martelo, ou de uma máquina de lavar roupa. A distinção principal está em que um computador pode ser programado para resolver virtualmente qualquer problema ou realizar praticamente qualquer tarefa, ao contrário da máquina de lavar roupa em que existe um conjunto muito pequeno de possíveis programas à escolha e que estão pré-definidos. I.e., um computador tipicamente não tem nenhuma aplicação específica: é uma máquina genérica. Claro que um computador usado para meras tarefas de secretariado (onde se utilizam os ditos "conhecimentos de informática do ponto de vista do utilizador") não é muito diferente de uma máquina de lavar roupa ou de uma máquina de escrever electrónica.

Um computador é portanto uma máquina programável de aplicação genérica. Serve para resolver problemas de muitos tipos diferentes, desde o problema de secretaria mais simples, passando pelo apoio à gestão de empresas, até ao controlo de autómatos e à construção de sistemas inteligentes. O objectivo desta disciplina é ensinar os princípios da resolução de problemas através de um computador.

Os computadores têm uma arquitectura que, nos seus traços gerais, não parece muito complicada. No essencial, os computadores consistem num processador (o "cérebro") que, para além de dispor de um conjunto de registos (memória de curta duração), tem acesso a uma memória (memória de média duração) e a um conjunto de dispositivos de entrada e saída de dados, entre os quais tipicamente discos rígidos (memória de longa duração), um teclado e um ecrã. Esta é uma descrição simplista de um computador. Na disciplina de Arquitectura de Computadores os vários componentes de um computador serão estudados com mais profundidade. Na disciplina de Sistemas Operativos, por outro lado, estudar-se-ão os programas que normalmente equipam os computadores de modo a simplificar a sua utilização.

1.2 Programação

Como se programa um computador? Os computadores, como se verá na disciplina de Arquitectura de Computadores, entendem uma linguagem própria, usualmente conhecida por linguagem máquina. Esta linguagem caracteriza-se por não ter qualquer tipo de ambiguidades: a interpretação das instruções é única. Por outro lado, esta linguagem também se caracteriza por consistir num conjunto de comandos ou instruções que são executados "cegamente" pelo computador: é uma linguagem imperativa. Os vários tipos de linguagens (imperativas, declarativas, etc.), as várias formas de classificar as linguagens e as vantagens e desvantagens de cada tipo de linguagem serão estudados na disciplina de Linguagens de Programação.

A linguagem máquina caracteriza-se também por ser penosa de utilizar para os humanos: todas as instruções correspondem a códigos numéricos, difíceis de memorizar, e as instruções são muito elementares. Uma solução parcial passa por atribuir a cada instrução uma dada mnemónica, ou seja, substituir os números que representam cada operação por nomes mais fáceis de decorar. Às linguagens assim definidas chama-se linguagens *assembly* e aos tradutores de *assembly* para linguagem máquina *assemblers* ou assembladores.

Uma solução preferível passa por equipar os computadores com compiladores, isto é com programas que são capazes de traduzir para linguagem máquina programas escritos em linguagens mais fáceis de usar pelo programador e mais poderosas que o *assembly*. Infelizmente, não existem ainda compiladores para as linguagens naturais, que é o nome que se dá às linguagens humanas (e.g., português, inglês, etc.). Mas existem as chamadas linguagens de programação de alto nível (alto nível entre as linguagens de programação, bem entendido), que se aproximam um pouco mais das linguagens naturais na sua facilidade de utilização pelos humanos, sem no entanto introduzirem as imprecisões, ambiguidades e dependências de contextos externos que são características das linguagens naturais. Nesta disciplina usar-se-á o C++ como linguagem de programação[1].

Tal como o conhecimento profundo do português não chega para fazer um bom escritor, o conhecimento profundo de uma linguagem de programação não chega para fazer um bom programador, longe disso. O conhecimento da linguagem é necessário, mas não é de todo suficiente. Programar não é o simples acto de escrever ideias de outrem: é ter essas ideias, é ser criativo e engenhoso. Resolver problemas exige conhecimentos da linguagem, conhecimento das técnicas conhecidas de ataque aos problemas, inteligência para fazer analogias com outros

[1]Pode-se pensar num computador equipado com um compilador de uma dada linguagem de programação como um novo computador, mais poderoso. É de toda a conveniência usar este tipo de abstracção, que de resto será muito útil para perceber sistemas operativos, onde se vão acrescentando camada após camada de *software* para acrescentar "inteligência" aos computadores.

problemas, mesmo em áreas totalmente desconexas, criatividade, intuição, engenho, persistência, etc. Programar não é um acto mecânico. Assim, aprender a programar consegue-se através do estudo e, fundamentalmente, do treino.

1.3 Algoritmos: resolvendo problemas

Dado um problema que é necessário resolver, como desenvolver uma solução? Como expressá-la? À última pergunta responde-se muito facilmente: usando uma linguagem. A primeira é mais difícil. Se a solução desenvolvida corresponder a um conjunto de instruções bem definidas e sem qualquer ambiguidade, podemos dizer que temos um *algoritmo* que resolve um problema, i.e., que a partir de um conjunto de entradas produz determinadas saídas.

A noção de algoritmo não é simples de perceber, pois é uma abstracção. Algoritmos são métodos de resolver problemas. Mas à concretização de um algoritmo numa dada linguagem já não se chama algoritmo: chama-se *programa*. Sob esse ponto de vista, todas as versões escritas de um algoritmo são programas, mesmo que expressos numa linguagem natural (desde que não façam uso da sua característica ambiguidade). Abusando um pouco das definições, no entanto, chamaremos algoritmo a um método de resolução de um dado problema expresso em linguagem natural, e programa à concretização de um algoritmo numa dada linguagem de programação.

A definição de algoritmo é um pouco mais completa do que a apresentada. De acordo com Knuth [10] os algoritmos têm cinco características importantes:

Finitude Um algoritmo tem de terminar sempre ao fim de um número finito de passos. De nada nos serve um algoritmo se existirem casos em que não termina.

Definitude[2] Cada passo do algoritmo tem de ser definido com precisão; as acções a executar têm de ser especificadas rigorosamente e sem ambiguidade. No fundo isto significa que um algoritmo tem de ser tão bem especificado que até um computador possa seguir as suas instruções.

Entrada Um algoritmo pode ter zero ou mais entradas, i.e., entidades que lhe são dadas inicialmente, antes do algoritmo começar. Essas entidades pertencem a um conjunto bem definido (e.g., o conjunto dos números inteiros). Existem algoritmos interessantes que não têm qualquer entrada, embora sejam raros.

Saída Um algoritmo tem uma ou mais saídas, i.e., entidades que têm uma relação bem definida com as entradas (o problema resolvido pelo algoritmo é o de calcular as saídas correspondentes às entradas).

Eficácia Todas as operações executadas no algoritmo têm de ser suficientemente básicas para, em princípio, poderem ser feitas com exactidão e em tempo finito por uma pessoa usando um papel e um lápis.

Para além destas características dos algoritmos, pode-se também falar da sua eficiência, isto é, do tempo que demoram a ser executados para dadas entradas. Em geral, portanto, pretende-se que os algoritmos sejam não só finitos mas também suficientemente rápidos [10]. Ou seja, devem resolver o problema em tempo útil (e.g., enquanto ainda somos vivos para estarmos interessados na sua resolução) mesmo para a mais desfavorável combinação possível das entradas.

O estudo dos algoritmos, a algoritmia (*algorithmics*), é um campo muito importante da ciência da computação, que envolve por exemplo o estudo da sua correcção (verificação se de facto resolvem o problema), da sua finitude (será de facto um algoritmo, ou não passa de um método computacional [10] inútil na prática?) e da sua eficiência (análise de algoritmos). Estes temas serão inevitavelmente abordados nesta disciplina, embora informalmente, e serão fundamentados teoricamente na disciplina de Computação e Algoritmia. A disciplina de Introdução à Programação serve portanto de ponte entre as disciplinas de Arquitectura de Computadores e Computação e Algoritmia, fornecendo os conhecimentos necessários para o trabalho subsequente em outras disciplinas da área da informática.

1.3.1 Regras do jogo

No jogo de resolução de problemas que é o desenvolvimento de algoritmos, há duas regras simples:

1. as variáveis são os únicos objectos manipulados pelos algoritmos e
2. os algoritmos só podem memorizar valores em variáveis.

As *variáveis* são os objectos sobre os quais as instruções dos algoritmos actuam. Uma variável corresponde a um local onde se podem guardar valores. Uma variável tem três características:

1. Um nome, que é fixo durante a vida da variável, e pelo qual a variável é conhecida. É importante distinguir as variáveis sobre as quais os algoritmos actuam das variáveis matemáticas usuais. Os nomes das variáveis de algoritmo ou programa serão grafados com um tipo de largura fixa, enquanto as variáveis matemáticas serão grafadas em itálico. Por exemplo, `k` e `número_de_alunos` são variáveis de algoritmo ou programa enquanto k é uma variável matemática.
2. Um tipo, que também é fixo durante a vida da variável, e que determina o conjunto de valores que nela podem ser guardados e, sobretudo, as operações que se podem realizar com os valores guardados nas variáveis desse tipo. Para já, considerar-se-á que todas as variáveis têm tipo inteiro, i.e., que guardam valores inteiros suportando as operações usuais com números inteiros. Nesse caso dir-se-á que o tipo é das variáveis é inteiro ou $\mathbb{Z}$.
3. Um e um só valor em cada instante de tempo. No entanto, o valor pode variar no tempo, à medida que o algoritmo decorre.

Costuma-se fazer uma analogia entre algoritmo e receita de cozinha. Esta analogia é atractiva, mas falha em alguns pontos. Tanto uma receita como um algoritmo consistem num conjunto de instruções. Porém, no caso das receitas, as instruções podem ser muito vagas. Por exemplo, "ponha ao lume e vá mexendo até alourar". Num algoritmo as instruções tem de ser precisas, sem margem para interpretações diversas. O pior ponto da analogia diz respeito às variáveis. Pode-se dizer que as variáveis de um algoritmo correspondem aos recipientes usados para realizar uma receita. O problema é que um recipiente pode (a) estar vazio e (b) conter qualquer tipo de ingrediente, enquanto *uma variável contém sempre valores do mesmo tipo* e, além disso, *uma variável contém sempre um qualquer valor*: as variáveis não podem estar vazias! É absolutamente fundamental entranhar esta característica pouco intuitiva das variáveis.

É muito conveniente ter uma notação para representar graficamente uma variável. A Figura 1.1 mostra uma variável inteira de nome `idade` com valor 24. É comum, na linguagem corrente,

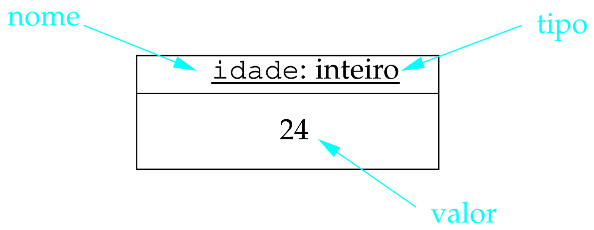


Figura 1.1: Notação para uma variável. A azul explicações sobre a notação gráfica usada.

não distinguir entre a variável, o seu nome e o valor que guarda. Esta prática é abusiva, mas simplifica a linguagem. Por exemplo, acerca da variável na Figura 1.1, é costume dizer-se que "a `idade` é 24". Em rigor dever-se-ia dizer que "a variável de nome `idade` guarda actualmente o valor 24".

Suponha-se um problema simples. São dadas duas variáveis `n` e `soma`, ambas de tipo inteiro. Pretende-se que a execução do algoritmo coloque na variável `soma` a soma dos primeiros `n` inteiros não-negativos. Admite-se portanto que a variável `soma` tem inicialmente um valor arbitrário enquanto a variável `n` contém inicialmente o número de termos da soma a realizar.

Durante a execução de um algoritmo, as variáveis existentes vão mudando de valor. Aos valores das variáveis num determinado instante da execução do algoritmo chama-se *estado*. Há dois estados particularmente importantes durante a execução de um algoritmo: o estado inicial e o estado final.

O estado inicial é importante porque os algoritmos só estão preparados para resolver os problemas se determinadas condições mínimas se verificarem no início da sua execução. Para o problema dado não faz qualquer sentido que a variável `n` possua inicialmente o valor -3, por exemplo. Que poderia significar a "soma dos primeiros -3 inteiros não-negativos"? Um algoritmo é uma receita para resolver um problema, mas apenas se as variáveis verificaram inicialmente determinadas condições mínimas, expressas na chamada *pré-condição* ou PC. Neste caso a pré-condição diz simplesmente que a variável `n` não pode ser negativa, i.e., $PC \equiv 0 \leq \mathtt{n}$.

O estado final é ainda mais importante que o estado inicial. O estado das variáveis no final de um algoritmo deve ser o necessário para que o problema que o algoritmo é suposto resolver esteja de facto resolvido. Para especificar quais os estados aceitáveis para as variáveis no final do algoritmo usa-se a chamada *condição-objectivo* ou CO. Neste caso a condição objectivo é $CO \equiv \mathtt{soma} = \sum_{j=0}^{\mathtt{n}-1} j$, ou seja, a variável `soma` deve conter a soma dos inteiros entre 0 e $\mathtt{n}-1$

inclusive[3].

O par de condições pré-condição e condição objectivo representa de forma compacta o problema que um algoritmo é suposto resolver. Neste caso, porém, este par de condições está incompleto: uma vez que o algoritmo pode alterar o valor das variáveis, pode perfeitamente alterar o valor da variável `n`, pelo que um algoritmo perverso poderia simplesmente colocar o valor zero quer em `n` quer em `soma` e declarar resolver o problema! Em rigor, portanto, dever-se-ia indicar claramente que `n` não pode mudar de valor ao longo do algoritmo, i.e., que `n` é uma *constante*, ou, alternativamente, indicar claramente na condição objectivo que o valor de `n` usado é o valor de `n` no *início* do algoritmo. Em vez disso admitir-se-á simplesmente que o valor de `n` não é alterado pelo algoritmo.

O problema proposto pode ser resolvido de uma forma simples. Considere-se uma variável adicional `i` que conterá os inteiros a somar (um de cada vez...). Comece-se por colocar o valor zero quer na variável `soma` quer na variável `i`. Enquanto o valor da variável `i` não atingir o valor da variável `n`, deve-se somar o valor da variável `i` ao valor da variável `soma` e guardar o resultado dessa soma na própria variável `soma` (que vai servindo para acumular o resultado), e em seguida deve-se aumentar o valor de `i` de uma unidade. Quando o valor de `i` atingir `n`, o algoritmo termina. Um pouco mais formalmente[4]:

$\{PC \equiv\ 0 \leq \mathtt{n}.\}$
$\mathtt{i} \leftarrow 0$
$\mathtt{soma} \leftarrow 0$
enquanto $\mathbf{i} \neq \mathbf{n}$ **faça-se:**
    $\mathtt{soma} \leftarrow \mathtt{soma} + \mathtt{i}$
    $\mathtt{i} \leftarrow \mathtt{i} + 1$
$\{CO \equiv\ \mathtt{soma} = \sum_{j=0}^{\mathtt{n}-1} j.\}$

O símbolo $\leftarrow$ deve ser lido "fica com o valor de" e chama-se a *atribuição*. Tudo o que se coloca entre chavetas são comentários, não fazendo parte do algoritmo propriamente dito. Neste caso usaram-se comentários para indicar as condições que se devem verificar no início e no final do algoritmo.

Um algoritmo é lido e executado pela ordem normal de leitura em português: de cima para baixo e da esquerda para a direita, excepto quando surgem construções como um **enquanto**, que implicam voltar atrás para repetir um conjunto de instruções.

É muito importante perceber a evolução dos valores das variáveis ao longo da execução do algoritmo. Para isso é necessário arbitrar os valores iniciais da variáveis. Suponha-se que `n` tem inicialmente o valor 4. O estado inicial, i.e., imediatamente antes de começar a executar o algoritmo, é o indicado na Figura 1.2. Dois aspectos são de notar. Primeiro que este estado verifica a pré-condição indicada. Segundo que os valores iniciais das variáveis `i` e `soma` são irrelevantes, o que é indicado através do símbolo '?'.

Neste caso é evidente que o estado final, i.e., imediatamente após a execução do algoritmo, é o indicado na Figura 1.3. Mas em geral pode não ser tão evidente, pelo que é necessário

[3]Mais tarde usar-se-á uma notação diferente para o somatório: $CO \equiv\ \mathtt{soma} = (\mathbf{S}\, j\ :\ 0 \leq j < \mathtt{n}\ :\ j)$.

[4]Para o leitor mais atento deverá ser claro que uma forma mais simples de resolver o problema é simplesmente colocar na variável `soma` o resultado de $\frac{\mathtt{n}(\mathtt{n}-1)}{2}$...

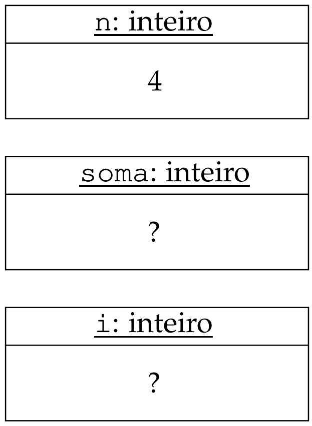


Figura 1.2: Estado inicial do algoritmo.

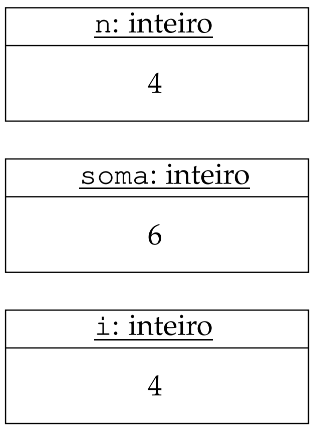


Figura 1.3: Estado final do algoritmo.

fazer o traçado da execução do algoritmo, i.e., a verificação do estado ao longo da execução do algoritmo.

O traçado da execução de um algoritmo implica, pois, registar o valor de todas as variáveis antes e depois de todas as instruções executadas. Para que isso se torne claro, é conveniente numerar as transições entre as instruções:

$\{PC \equiv 0 \leq \texttt{n}.\}$

1

$\texttt{i} \leftarrow 0$

2

$\texttt{soma} \leftarrow 0$

3

enquanto $\mathtt{i} \neq \mathtt{n}$ **faça-se:**

4

  $\texttt{soma} \leftarrow \texttt{soma} + \texttt{i}$

5

  $\texttt{i} \leftarrow \texttt{i} + 1$

6

fim do enquanto.

7

$\{CO \equiv \texttt{soma} = \sum_{j=0}^{\texttt{n}-1} j.\}$

Introduziu-se explicitamente o final do enquanto de modo a separar claramente os intervalos 6 e 7. O intervalo 6 está após o aumento de um da variável `i` e antes de se verificar se o seu valor atingiu já o valor de `n`. O intervalo 7, pelo contrário, está depois do enquanto, quando `i` atingiu já o valor de `n`, no final do algoritmo.

Estes intervalos ficam mais claros se se recorrer a um diagrama de actividade[5] para representar o algoritmo, como se pode ver na Figura 1.4. É fácil agora seguir o fluxo de execução e analisar os valores das variáveis em cada transição entre instruções. O resultado dessa análise pode ser visto na Tabela 1.1.

A realização do traçado de um algoritmo, como se verá, não demonstra a sua correcção. Para o fazer há que usar técnicas um pouco mais formais, abordadas brevemente na próxima secção.

1.3.2 Desenvolvimento e demonstração de correcção

Seja o problema de, dados quaisquer dois inteiros positivos, calcular os seu máximo divisor comum[6].

O algoritmo que resolve este problema tem de começar por instruções dizendo para os dois inteiros serem pedidos a alguém (i.e., lidos) e guardados em duas variáveis, a que se darão

[5]Como estes diagramas mostram claramente o fluxo de execução do algoritmo, também são conhecidos por diagramas de fluxo ou fluxogramas.

[6]Este exemplo faz uso de alguma simbologia a que pode não estar habituado. Recomenda-se a leitura do Apêndice A.

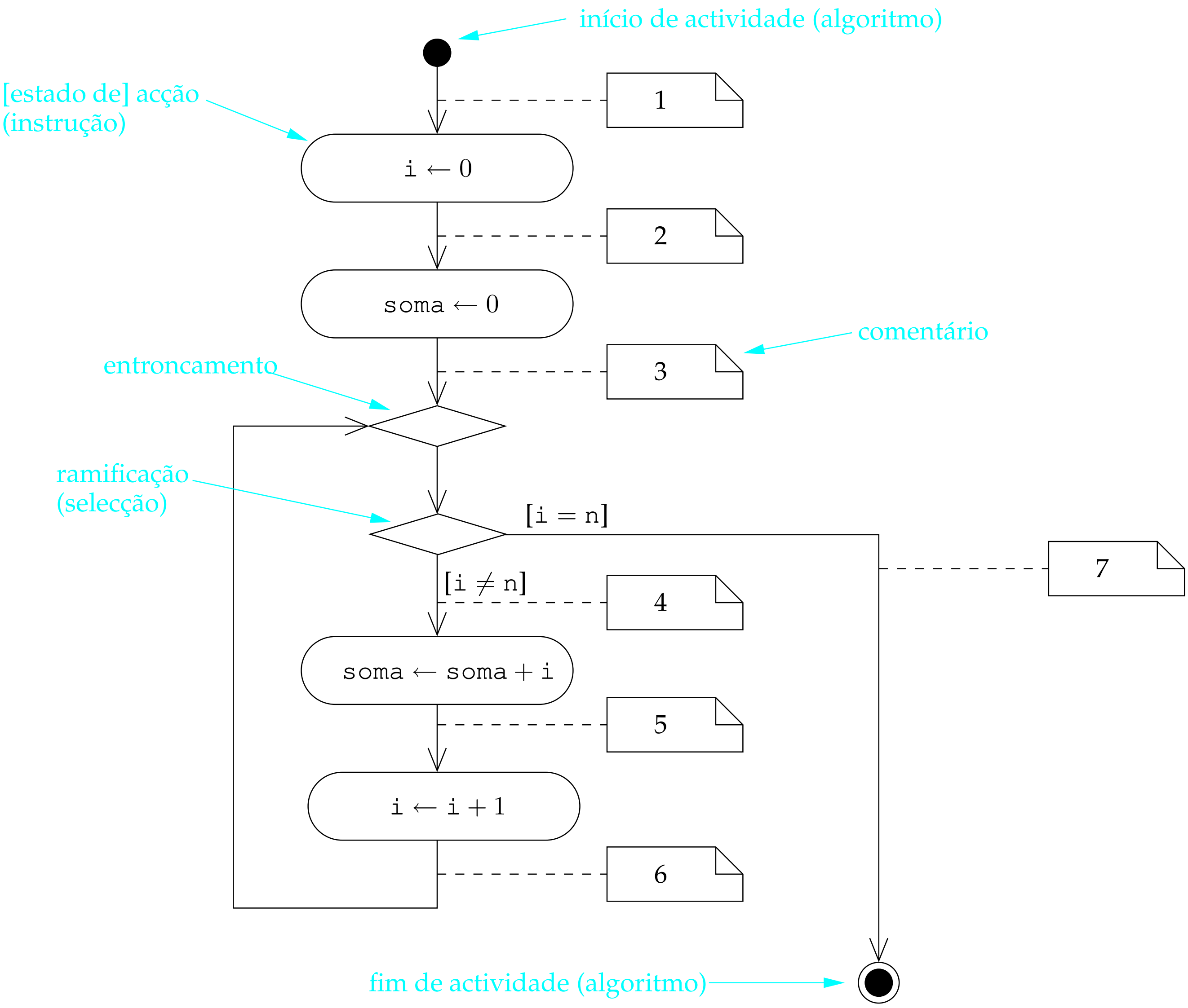


Figura 1.4: Diagrama de actividade do algoritmo. As setas indicam a sequência de execução das acções (instruções). Os losangos têm significados especiais: servem para unir fluxos de execução alternativos (entroncamentos) ou para os começar (ramificações). Em cada um dos ramos saídos de uma ramificação indicam-se as condições que têm de se verificar para que seja escolhido esse ramo. A azul encontram-se explicações sobre a notação gráfica usada. Os comentários contêm a numeração das transições usadas no algoritmo, para que se possa fazer uma mais fácil correspondência entre este e o diagrama.

Tabela 1.1: Traçado do algoritmo.

Transição	`n`	`i`	`soma`	Comentários
1	4	?	?	Verifica-se $PC \equiv 0 \leq \mathtt{n}$.
2	4	0	?	
3	4	0	0	
4	4	0	0	
5	4	0	0	
6	4	1	0	
4	4	1	0	
5	4	1	1	
6	4	2	1	
4	4	2	1	
5	4	2	3	
6	4	3	3	
4	4	3	3	
5	4	3	6	
6	4	4	6	
7	4	4	6	Verifica-se $CO \equiv \mathtt{soma} = \sum_{j=0}^{\mathtt{n}-1} j = \sum_{j=0}^{3} j = 0 + 1 + 2 + 3 = 6$.

os nomes `m` e `n`, e terminar por uma instrução dizendo para o máximo divisor comum ser comunicado a alguém (i.e., escrito). Para que o resultado possa ser comunicado a alguém, é necessário que esteja guardado em algum lado. Para isso é necessária uma variável adicional, a que se dará o nome de `k`. Os passos intermédios do algoritmo, entre a leitura dos dois inteiros e a escrita do resultado, indicam a forma de cálculo do máximo divisor comum, e são os passos que nos interessam neste momento. Assim, considerar-se-á inicialmente o problema mais simples de, dados dois valores inteiros positivos arbitrários colocados nas duas variáveis `m` e `n`, colocar na variável `k` o seu máximo divisor comum (mdc). Um exemplo é representado na Figura 1.5.

A abordagem deste problema passa em primeiro lugar pela identificação da chamada pré-condição (PC), i.e., pelas condições que se sabe que se verificam (ou que pelo menos se admite que se verificam) *a priori*. Neste caso a pré-condição é

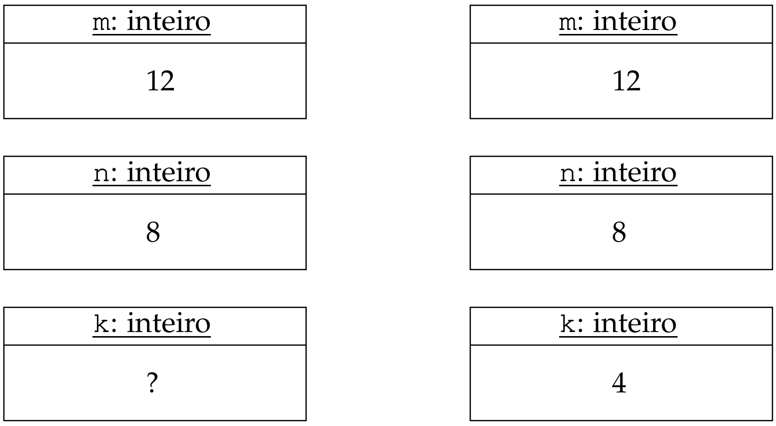
$$PC \equiv \mathtt{m} \text{ e } \mathtt{n} \text{ são inteiros positivos.}$$

Depois, deve-se formalizar a condição objectivo (CO). Neste caso a condição objectivo é

$$CO \equiv \mathtt{k} = \mathrm{mdc}(\mathtt{m}, \mathtt{n}),$$

onde se assume que `m` e `n` não mudam de valor ao longo do algoritmo. O objectivo do problema é, pois, encontrar um valor para `k` que verifique a condição objectivo CO.

Para que o problema faça sentido, é necessário que, quaisquer que sejam `m` e `n` tais que a PC é verdadeira, exista pelo menos um inteiro positivo `k` tal que a CO é verdadeira. No



(a) Antes de executar o algoritmo (b) Depois de executar o algoritmo

Figura 1.5: Exemplo de estado das variáveis antes 1.5(a) e depois 1.5(b) de executado o algoritmo.

problema em causa não há quaisquer dúvidas: todos os pares de inteiros positivos têm um correspondente máximo divisor comum.

Para que a solução do problema não seja ambígua, é necessário garantir mais. Não basta que exista solução: é necessário que seja única, ou seja, que quaisquer que sejam `m` e `n` tais que a PC é verdadeira, existe um e um só inteiro positivo `k` tal que a CO é verdadeira. Quando isto se verifica, pode-se dizer que o problema está bem colocado (a demonstração de que um problema está bem colocado muitas vezes se faz desenvolvendo um algoritmo: são as chamadas demonstrações construtivas). É fácil verificar que, neste caso, o problema está bem colocado: existe apenas um mdc de cada par de inteiros positivos.

Depois de se verificar que de facto o problema está bem colocado dadas a pré-condição e a condição objectivo, é de todo o interesse identificar propriedades interessantes do mdc. Duas propriedades simples são:

a) O mdc é sempre superior ou igual a 1, i.e., quaisquer que sejam m e n inteiros positivos, $1 \leq \mathrm{mdc}(m, n)$.

b) O mdc não excede nunca o menor dos dois valores, i.e., quaisquer que sejam m e n inteiros positivos, $\mathrm{mdc}(m, n) \leq \min(m, n)$, em que $\min(m, n)$ é o menor dos dois valores m e n.

A veracidade destas duas propriedades é evidente, pelo que não se demonstra aqui.

Seja $m \div k$ a notação para o resto da divisão inteira de m por k. Ou seja, dados dois inteiros $0 \leq m$ e $0 < k$, e sendo $m = qk + r$ com $0 \leq r < k$ (onde q e r são respectivamente o quociente e o resto da divisão de m por k), então $r = m \div k$. Nesse caso, dizer que $0 < k$ é divisor de $0 \leq m$ é o mesmo que dizer que $m \div k = 0$, i.e., que a divisão inteira de m por k tem resto zero.

Duas outras propriedades que se verificará serem importantes são:

c) Quaisquer que sejam k, m e n inteiros positivos,

$$\mathrm{mdc}(m,n) \leq k \wedge (m \div k = 0 \wedge n \div k = 0) \Rightarrow k = \mathrm{mdc}(m,n).$$

Ou seja, se k é superior ou igual ao máximo divisor comum de m e n e se k é divisor comum de m e n, então k é o máximo divisor comum de m e n.

A demonstração pode ser feita por absurdo. Suponha-se que k não é o máximo divisor comum de m e n. Como k é divisor comum, então conclui-se que $k < \mathrm{mdc}(m,n)$, isto é, há divisores comuns maiores que k. Mas então $\mathrm{mdc}(m,n) \leq k < \mathrm{mdc}(m,n)$, que é uma impossibilidade. Logo, $k = \mathrm{mdc}(m,n)$.

d) Quaisquer que sejam k, m e n inteiros positivos,

$$\mathrm{mdc}(m,n) \leq k \wedge (m \div k \neq 0 \vee n \div k \neq 0) \Rightarrow \mathrm{mdc}(m,n) < k.$$

Ou seja, se k é superior ou igual ao máximo divisor comum de m e n e se k não é divisor comum de m e n, então k é maior que o máximo divisor comum de m e n. Neste caso a demonstração é trivial.

Em que é que estas propriedades nos ajudam? As duas primeiras propriedades restringem a gama de possíveis divisores, i.e., o intervalo dos inteiros onde o mdc deve ser procurado, o que é obviamente útil. As duas últimas propriedades serão úteis mais tarde.

Não há nenhum método mágico de resolver problemas. Mais tarde ver-se-á que existem metodologias que simplificam a abordagem ao desenvolvimento de algoritmos, nomeadamente aqueles que envolvem iterações (repetições). Mesmo essas metodologias não são substituto para o engenho e a inteligência. Para já, usar-se-á apenas a intuição.

Onde se deve procurar o mdc? A intuição diz que a procura deve ser feita a partir de $\min(\mathtt{m},\mathtt{n})$, pois de outra forma, começando por 1, é fácil descobrir divisores comuns, mas não é imediato se eles são ou não o máximo divisor comum: é necessário testar todos os outros potenciais divisores até $\min(\mathtt{m},\mathtt{n})$. Deve-se portanto ir procurando divisores comuns partindo de $\min(\mathtt{m},\mathtt{n})$ para baixo até se encontrar algum, que será forçosamente o mdc.

Como transformar estas ideias num algoritmo e, simultaneamente, demonstrar a sua correcção? A variável `k`, de acordo com a CO, tem de terminar o algoritmo com o valor do máximo divisor comum. Mas pode ser usada entretanto, durante o algoritmo, para conter inteiros que são candidatos a máximo divisor comum. Assim, se se quiser começar pelo maior valor possível, deve-se atribuir a `k` (ou seja, registar na "caixa" `k`) o menor dos valores `m` e `n`:

se $\mathtt{m} < \mathtt{n}$ **então:**
    $\mathtt{k} \leftarrow \mathtt{m}$
senão:
    $\mathtt{k} \leftarrow \mathtt{n}$

Depois destas instruções, é evidente que se pode afirmar que $\mathrm{mdc}(\mathtt{m},\mathtt{n}) \leq \mathtt{k}$, dada a propriedade b).

Em seguida, deve-se verificar se `k` divide `m` e `n` e, no caso contrário, passar ao próximo candidato, que é o valor inteiro imediatamente abaixo:

enquanto $\mathtt{m} \div \mathtt{k} \neq 0 \vee \mathtt{n} \div \mathtt{k} \neq 0$ **faça-se:**
    $\mathtt{k} \leftarrow \mathtt{k} - 1$

A condição que controla o ciclo (chama-se ciclo porque é uma instrução que implica a repetição cíclica de outra, neste caso $\mathtt{k} \leftarrow \mathtt{k} - 1$) chama-se guarda ($G$) do ciclo. Neste caso a guarda é

$$G \equiv \mathtt{m} \div \mathtt{k} \neq 0 \vee \mathtt{n} \div \mathtt{k} \neq 0,$$

que é verdadeira se k não for divisor comum a m e n.

A instrução $\mathtt{k} \leftarrow \mathtt{k} - 1$ chama-se o progresso (*prog*) do ciclo pois, como se verá, é ela que garante que o ciclo progride em direcção ao seu fim.

Quando este ciclo terminar, $\mathtt{k}$ é o mdc. Como demonstrá-lo? Repare-se que, dadas as propriedades c) e d), quer a inicialização de $\mathtt{k}$, quer a guarda e o progresso, garantem que há uma condição que se mantém sempre verdadeira, desde o início ao fim do ciclo, e que por isso recebe o nome de condição invariante (CI) do ciclo:

$$CI \equiv \mathrm{mdc}(\mathtt{m}, \mathtt{n}) \leq \mathtt{k}.$$

Esta condição diz simplesmente que k é sempre superior ou igual ao máximo divisor comum que se procura.

Dada a inicialização do ciclo

se $\mathtt{m} < \mathtt{n}$ **então:**
    $\mathtt{k} \leftarrow \mathtt{m}$
senão:
    $\mathtt{k} \leftarrow \mathtt{n}$

e a pré-condição, tem-se que $\mathtt{k} = \min(\mathtt{m}, \mathtt{n})$, o que, conjugado com a propriedade b), garante que $\mathrm{mdc}(\mathtt{m}, \mathtt{n}) \leq \mathtt{k}$. Ou seja, a inicialização leva à veracidade da condição invariante.

Durante o ciclo, que acontece se a G for verdadeira? Diminui-se $\mathtt{k}$ de uma unidade, isto é progride-se[7]. Mas, admitindo que a condição invariante é verdadeira e sendo a guarda também verdadeira, conclui-se pela propriedade d) que

$$\mathrm{mdc}(\mathtt{m}, \mathtt{n}) < \mathtt{k}$$

logo após a verificação da guarda e imediatamente antes do progresso.

Como o valor guardado em $\mathtt{k}$ é inteiro, isso é o mesmo que dizer que $\mathrm{mdc}(\mathtt{m}, \mathtt{n}) \leq \mathtt{k} - 1$. Donde, depois do progresso $\mathtt{k} \leftarrow \mathtt{k} - 1$, a veracidade da condição invariante é recuperada, i.e., $\mathrm{mdc}(\mathtt{m}, \mathtt{n}) \leq \mathtt{k}$. Ou seja, demonstrou-se que se a condição invariante for verdadeira antes do progresso, também o será depois.

A demonstração da veracidade da condição invariante do ciclo após a inicialização do ciclo e da preservação da sua veracidade durante a execução do ciclo correspondem, no fundo, à demonstração por indução de que a condição invariante se verifica durante todo o ciclo, *inclusive quando este termina*. I.e., demonstrou-se que a condição invariante é, de facto, invariante.

[7] É uma progressão porque se fica mais próximo do final do ciclo, ou seja, fica-se mais próximo do real valor do $\mathrm{mdc}(\mathtt{m}, \mathtt{n})$.

Que acontece se, durante o ciclo, a guarda for falsa? Nesse caso o ciclo termina, concluindo-se que há duas condições verdadeiras: a condição invariante CI e também a negação da guarda, i.e., $\neg G$. A conjunção destas condições, pela propriedade c), implica que a condição objectivo é verdadeira, i.e., que `k` termina de facto com o máximo divisor comum de `m` e `n`.

A demonstração da correcção (parcial) do algoritmo passa, pois, por demonstrar que a condição invariante é verdadeira do início ao fim do ciclo (i.e., que é de facto invariante) e que, quando o ciclo termina (i.e., quando a guarda é falsa), se tem forçosamente que a condição objectivo é verdadeira, i.e.,

$$CI \wedge \neg G \Rightarrow CO.$$

Assim, o algoritmo completo, "decorado" com comentários onde se indica claramente quais as condições verdadeiras em cada transição entre intruções, é:

$\{PC \equiv\ 0 < \mathtt{m} \wedge 0 < \mathtt{n}.\}$
se $\mathtt{m} < \mathtt{n}$ **então:**
    $\mathtt{k} \leftarrow \mathtt{m}$
senão:
    $\mathtt{k} \leftarrow \mathtt{n}$
$\{$*Aqui sabe-se que* $\mathrm{mdc}(m, n) \leq \min(m, n) = k$, *ou seja, que a* CI *é verdadeira..*$\}$
enquanto $\mathtt{m} \div \mathtt{k} \neq 0 \vee \mathtt{n} \div \mathtt{k} \neq 0$ **faça-se:**
    $\{$*Aqui a* G *é verdadeira. Logo, pela propriedade d),*
    a CI *mantém-se verdadeira depois do seguinte progresso:*$\}$
    $\mathtt{k} \leftarrow \mathtt{k} - 1$
$\{$*Aqui a* G *é falsa. Logo, pela propriedade c),*
$k = \mathrm{mdc}(m, n)$, *ou seja a* CO *é verdadeira..*$\}$

Uma questão importante, que ficou por demonstrar, é se é garantido que o ciclo termina sempre. Se não se garantir a terminação, não se pode chamar a esta sequência de instruções um algoritmo, como se viu. Esta demonstração é essencial para garantir a correcção *total* do algoritmo.

A demonstração é simples. Em primeiro lugar, a inicialização garante que o valor inicial de `k` é superior ou igual a 1 (pois `m` e `n` são inteiros positivos). O progresso, por outro lado, faz o valor de `k` diminuir a cada iteração do ciclo. Como 1 é divisor comum de qualquer par de inteiros positivos, o ciclo, na pior das hipóteses, termina com $\mathtt{k} = 1$, i.e., na pior das hipóteses o ciclo termina ao fim de $\min(\mathtt{m}, \mathtt{n}) - 1$ iterações. Assim, o algoritmo é de facto um algoritmo, pois verifica a condição de finitude: está totalmente correcto.

Resumindo, o conjunto de instruções que se apresentou é um algoritmo porque:

1. é finito, terminando sempre ao fim de no máximo $\min(\mathtt{m}, \mathtt{n}) - 1$ iterações do ciclo;

2. está bem definido, pois cada passo do algoritmo está definido com precisão e sem ambiguidades;

3. tem duas entradas, que são os valores colocados inicialmente em `m` e `n`, pertencentes ao conjunto dos inteiros positivos;

4. tem uma saída que é o valor que se encontra em k no final do algoritmo e que verifica $\mathtt{k} = \mathrm{mdc}(\mathtt{m}, \mathtt{n})$; e

5. é eficaz, pois todas as operações do algoritmo podem ser feitas com exactidão e em tempo finito por uma pessoa usando um papel e um lápis (e alguma paciência).

Quanto à sua eficiência, pode-se desde já afirmar que é suficientemente rápido para se poder usar em muitos casos, embora existam algoritmos consideravelmente mais eficientes, que serão abordados posteriormente.

Mais uma vez se poderia fazer o traçado deste algoritmo admitindo, por exemplo, o estado inicial indicado na Figura 1.5(a). Esse traçado fica como exercício para o leitor, que pode recorrer ao diagrama de actividade correspondente ao algoritmo mostrado na Figura 1.6.

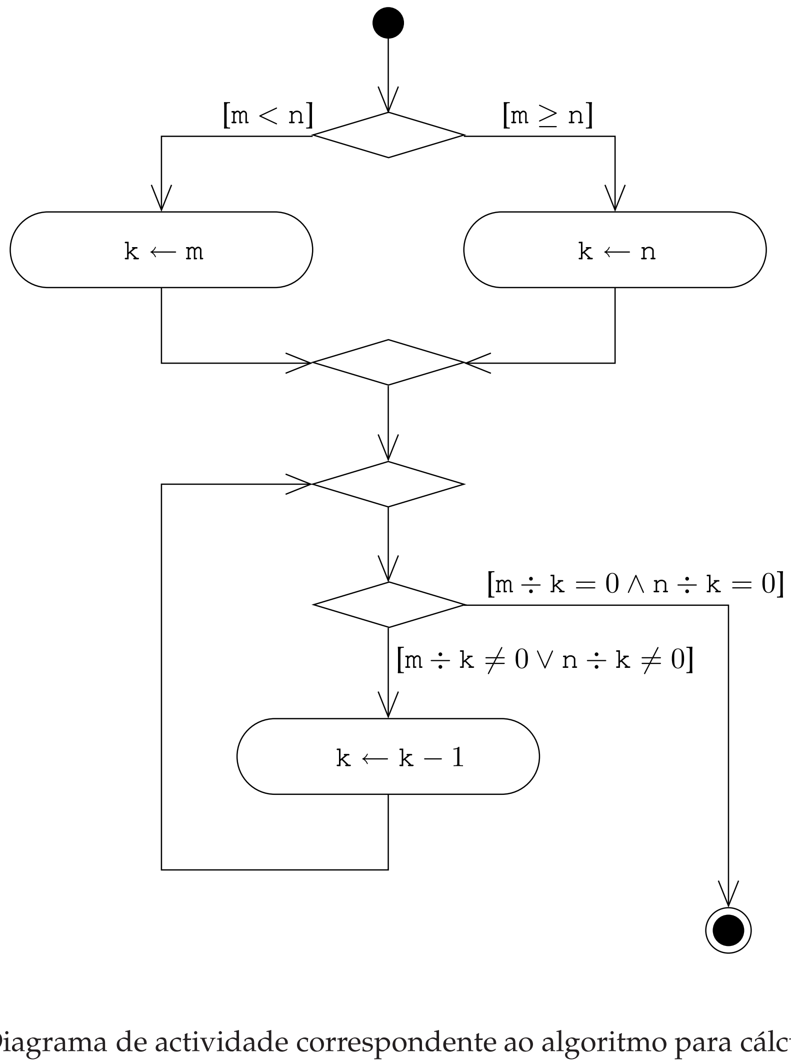


Figura 1.6: Diagrama de actividade correspondente ao algoritmo para cálculo do mdc.

Observações

O desenvolvimento de algoritmos em simultâneo com a demonstração da sua correcção é um exercício extremamente útil, como se verá no Capítulo 4. Mas pode-se desde já adiantar a uma das razões: não é possível demonstrar que um algoritmo funciona através de testes, excepto se se testarem rigorosamente todas as possíveis combinações de entradas válidas (i.e., entradas que verificam pré-condição do algoritmo). A razão é simples: o facto de um algoritmo funcionar correctamente para n diferentes combinações da entrada, por maior que n seja, não garante que não dê um resultado errado para uma outra combinação ainda não testada. Claro está que o oposto é verdadeiro: basta que haja uma combinação de entradas válidas para as quais o algoritmo produz resultados errados para se poder afirmar que ele está errado!

No caso do algoritmo do mdc é evidente que jamais se poderá mostrar a sua correcção através de testes: há infinitas possíveis combinações das entradas. Ainda que se limitasse a demonstração a entradas inferiores ou iguais a $1\,000\,000$, ter-se-iam $1\,000\,000\,000\,000$ testes para realizar. Admitindo que se conseguiam realizar $1\,000\,000$ de testes por segundo, a demonstração demoraria $1\,000\,000$ de segundos, cerca de 12 dias. Bastaria aumentar os valores das entradas até $10\,000\,000$ para se passar para um total de 1157 dias, cerca de três anos. Pouco prático, portanto...

O ciclo usado no algoritmo não foi obtido pela aplicação directa da metodologia de Dijkstra que será ensinada na Secção 4.7, mas poderia ter sido. Pede-se ao leitor que regresse mais tarde a esta secção para verificar que o ciclo pode ser obtido usando a factorização da condição objectivo

$$CO \equiv \overbrace{\mathtt{m} \div \mathtt{k} = 0 \land \mathtt{n} \div \mathtt{k} = 0}^{\neg G} \land \overbrace{0 < \mathtt{k} \land (\mathbf{Q}\, i\, :\, 0 \leq i < \mathtt{k}\, :\, \mathtt{m} \div i \neq 0 \lor \mathtt{n} \div i \neq 0)}^{CI},$$

onde $\mathbf{Q}$ significa "qualquer que seja".

Finalmente, para o leitor mais interessado, fica a informação de que algoritmos mais eficientes para o cálculo do mdc podem ser obtidos pela aplicação das seguintes propriedades do mdc:

- Quaisquer que sejam a e b inteiros positivos $\mathrm{mdc}(a, b) = \mathrm{mdc}(b \div a, a)$.
- Se $a = 0 \land 0 < b$, então $\mathrm{mdc}(a, b) = b$.

Estas propriedades permitem, em particular, desenvolver o chamado algoritmo de Euclides para obtenção do máximo divisor comum.

1.4 Programas

Como se viu, um programa é a concretização, numa dada linguagem de programação, de um determinado algoritmo (que resolve um problema concreto). Nesta disciplina ir-se-á utilizar uma linguagem de alto nível chamada C++. Esta linguagem tem o seu próprio léxico, a sua sintaxe, a sua gramática e a sua semântica (embora simples). Todos estes aspectos da linguagem serão tratados ao longo deste texto. Para já, no entanto, apresenta-se sem mais explicações a tradução do algoritmo do mdc para C++, embora agora já com instruções explícitas para leitura das entradas a partir de um teclado e de escrita da saída no ecrã. O programa resultante é suficientemente simples para que possa ser entendido quase na sua totalidade. No entanto, é normal que o leitor fique com dúvidas: serão esclarecidas no próximo capítulo.

```
// Todo o texto após // são comentários, não fazendo parte do programa.

// Os programas começam tipicamente por um conjunto de inclusões.
// Neste caso incluem-se ferramentas de entradas e saídas (Input/Output) a partir
// de canais (stream):
#include <iostream>

// Para evitar ter de escrever std::cout, std::cin, etc.
using namespace std;

/// Este programa calcula o máximo divisor comum de dois números.
int main()
{
    // Instruções de inserção de informação no canal de saída cout, ligado ao ecrã.
    // Mostra ao utilizador informação sobre o programa e pede-lhe para inserir os
    // dois inteiros dos quais quer obter o máximo divisor comum:
    cout << "Máximo divisor comum de dois números."
         << endl;
    cout << "Introduza dois inteiros positivos: ";

    // Definição de duas variáveis inteiras:
    int m, n;

    // Instrução de extracção de informação do canal de entrada, ligado ao teclado.
    // Serve para obter do utilizador os valores dos quais ele pretende saber o mdc:
    cin >> m >> n; // Assume-se que m e n são positivos!

    // Como um divisor é sempre menor ou igual a um número, escolhe-se
    // o mínimo dos dois!
    int k;
    if(m < n)
        k = m;
    else
        k = n;
    // Neste momento sabe-se que mdc(m, n) ≤ k.

    while(m % k != 0 or n % k != 0) {
        // Se o ciclo não parou, então k não divide m e n, logo,
        // k ≠ mdc(m, n) ∧ mdc(m, n) ≤ k. Ou seja, mdc(m, n) < k.
        --k; // Decrementa k. É o progresso do ciclo.
        // Neste momento, mdc(m, n) ≤ k outra vez! É o invariante do ciclo!
```

```
        }

        // Como mdc(m, n) ≤ k (invariante do ciclo) e k divide m e n
        // (o ciclo terminou, não foi?), conclui-se que k = mdc(m, n)!

        // Insere no canal de saída uma mensagem para o utilizador dizendo-lhe qual o
        // resultado:
        cout << "O máximo divisor comum de " << m
             << " e " << n << " é " << k << '.' << endl;
}
```

1.5 Resumo: resolução de problemas

Resumindo, a resolução de problemas usando linguagens de programação de alto nível tem vários passos:

1. Especificação do problema, feita por humanos.
2. Desenvolvimento de um algoritmo que resolve o problema, feito por humanos. É neste passo que se faz mais uso da inteligência e criatividade.
3. Concretização do algoritmo na linguagem de programação: desenvolvimento do programa, feito por humanos. O resultado deste passo é um programa, consistindo numa sequência de instruções, à qual também se chama *código*. Este passo é mecânico e relativamente pouco interessante.
4. Tradução do programa para linguagem máquina, feita pelo computador, ou melhor, por uma programa chamado compilador.
5. Execução do programa para resolver um problema particular (e.g., cálculo de $\mathrm{mdc}(131, 47)$), feita pelo computador.

Podem ocorrer erros em todos estes passos. No primeiro, se o problema estiver mal especificado. No segundo ocorrem os chamados *erros lógicos*: o algoritmo desenvolvido na realidade não resolve o problema tal como especificado. É aqui que os erros são mais graves, pelo que é extremamente importante assegurar a correcção dos algoritmos desenvolvidos. No terceiro passo, os erros mais benévolos são os que conduzem a *erros sintácticos* ou gramaticais, pois o compilador, que é o programa que traduz o programa da linguagem de programação em que está escrito para a linguagem máquina, assinala-os. Neste passo os piores erros são *gralhas* que alteram a semântica do programa sem o invalidar sintacticamente. Por exemplo, escrever `l` (letra 'l') em vez de `1` pode ter consequências muito graves se `l` for o nome de uma variável. Estes erros são normalmente muito difíceis de detectar[8]. Finalmente, a ocorrência de erros nos dois últimos passos é tão improvável que mais vale assumir que não podem ocorrer de todo.

[8]Estes erros assemelham-se ao "não" introduzido (acidentalmente?) pelo revisor de provas em "História do cerco de Lisboa", de José Saramago.