Capítulo 1

Conceitos básicos

1.1 Algoritmos

Uma vez que este texto trata de análise de algoritmos, é conveniente iniciarmos esta discussão examinando com detalhes nosso objeto de estudo: o *algoritmo*. Um algoritmo é uma descrição detalhada dos passos necessários para solucionar um problema. Em geral, os texto básicos sobre programação ilustram esse conceito através de exemplos cotidianos, tais como a receita de um bolo, as instruções para a troca de um pneu furado ou as ações necessárias para fritar um ovo.

Nós iremos começar com um exemplo mais relevante, embora bastante antigo: o algoritmo de Euclides para o cálculo do máximo divisor comum de dois inteiros. O máximo divisor comum (mdc) é definido como o maior número inteiro que divide exatamente dois números inteiros dados[1]. O algoritmo de Euclides está incluído em sua monumental obra *Elementos*, que data de cerca de 300 A.C., mas provavelmente já era conhecido antes disso. O cálculo do mdc está descrito nas proposições 1 e 2 do Livro VII (que trata de conceitos do ramo da matemática que é conhecido hoje como *Teoria Elementar dos Números*) da seguinte forma:

> *Proposição 1*: quando dois números diferentes são dados, e o menor é continuamente subtraído do maior, se o número que sobra nunca mede o anterior até que sobre uma unidade, então os números originais são primos entre si.
>
> *Proposição 2*: encontrar a maior medida comum de dois números dados não primos entre si.
>
> *Corolário 1*: se um número mede dois números, então ele também mede sua maior medida comum.

Repare que Euclides enuncia o algoritmo na forma de proposições (teoremas) que devem ser demonstradas. A primeira proposição mostra como determinar se dois números são primos entre si (ou seja, não têm divisores comuns) e a segunda utiliza o algoritmo da anterior para calcular o mdc se os números não forem primos entre si. A linguagem é pouco familiar (em particular, Euclides utiliza "*mede*" e "*medida*" para indicar, respectivamente, "*divide*" e "*divisor*"), e o algoritmo está apresentado de forma desestruturada, através de uma descrição textual.

Assim, precisamos detalhar um pouco mais nossa definição de algoritmo e encontrar uma forma mais conveniente de representação. A definição abaixo[2] é baseada em Knuth (1973a, p. 4–6):

[1] Neste texto, não estamos preocupados em fornecer definições precisas para os conceitos matemáticos básicos. Assim, termos tais como *divisor* são confiados ao conhecimento prévio ou à intuição do leitor. Para um tratamento mais rigoroso, um texto sobre teoria dos números deve ser consultado. Uma boa referência é Shokranian, Soares e Godinho (1998, p. 15–26).

[2] A definição fornecida, embora mais detalhada, ainda é informal. Entretanto, uma definição mais rigorosa nos levaria para assuntos fora do contexto deste trabalho. Os interessados podem consultar as referências Diverio e Menezes (2003, p. 65–128) e Lewis e Papadimitriou (1998, p. 172–231). Um texto muito interessante, que trata do assunto de uma forma bastante acessível, é Berlinski (2000).

Algoritmo 1.1: `Mdc(`a`,`b`)`

Entrada: Inteiros não negativos a e b
Saída: Máximo divisor comum de a e b

```
1 dividendo ← a;
2 divisor ← b;
3 enquanto divisor ≠ 0 faça
4 |   resto ← dividendo mod divisor;
5 |   dividendo ← divisor;
6 |   divisor ← resto;
7 fim enqto
8 retorna dividendo;
```

Um algoritmo é um conjunto finito de regras ou *passos* que definem uma seqüência de operações para resolver um determinado problema, com as seguintes características:

1. Entrada: o algoritmo deve ter zero ou mais entradas, que representam os dados fornecidos a ele e sobre os quais irá trabalhar.
2. Saída: o algoritmo deve ter uma ou mais saídas, consistindo dos resultados encontrados e que devem ser devolvidos ao utilizador.
3. Definição: cada passo deve ser definido de maneira precisa e sem ambigüidades. Ou seja, não deve haver dúvidas sobre a operação a ser executada.
4. Efetividade: cada passo deve ser *efetivo*, ou seja, não deve exigir operações que não possam ser executadas. Em princípio, deve ser possível que um operador humano seja capaz de executá-las em um intervalo de tempo finito, utilizando apenas lápis e papel.
5. Correção: o algoritmo deve terminar em um número finito de passos, fornecendo o resultado correto.

Associada à idéia de algoritmo está a noção de *estrutura de dados*, que consiste na forma de organizar (ou estruturar) os dados que serão tratados pelo algoritmo, de forma a facilitar a sua operação.

Vamos reescrever o algoritmo de Euclides de uma forma mais estruturada, explicitando cada passo, e verificar se os critérios acima são atendidos. Veja o algoritmo 1.1 (que está "otimizado" para utilizar a operação de divisão no lugar da subtração). Todos os nossos algoritmos serão exibidos dessa forma:

- A linha inicial contém o número e o nome do algoritmo (que pode vir na forma de um nome de procedimento com os respectivos parâmetros, tal como neste exemplo). O número do algoritmo serve para que ele possa ser referenciado em qualquer parte do texto.
- A seguir, são mostrados os dados de entrada e saída. Freqüentemente, o algoritmo será mostrado na forma de uma função ou procedimento, e os dados de entrada corresponderão aos seus parâmetros. De qualquer forma, a entrada detalha os dados sobre os quais o algoritmo irá trabalhar e que devem ser fornecidos a ele. Já a saída mostra o que será obtido como resultado de sua execução.
- Os passos que especificam as operações a serem executadas são listados em linhas consecutivas e numeradas, de modo que possam ser referenciadas a partir do texto. Essas instruções estarão na forma de um pseudo-código cujas instruções são muito semelhantes às da maioria das linguagens de programação imperativas, mas sem a complexidade que poderia dificultar sua compreensão. As regras utilizadas serão as seguintes:

  1. Variáveis não serão declaradas (supõe-se que seus tipos ficarão claros pelo contexto), e serão sempre locais, a não ser que haja uma indicação explícita do contrário (através da palavra reservada **global**). Se marcada com **constante**, o valor inicial da variável não pode ser alterado.

2. Limites de vetores e matrizes serão indicados por uma notação semelhante à do Pascal ($A[1..5]$, $B[1..5, 1..3]$) ou através da declaração explícita dos elementos iniciais ($A = [1, 2, 3, 4, 5]$). Elementos dos vetores utilizarão índices entre colchetes ($A[i]$, $B[i, 2]$). O tamanho de um vetor (ou de uma dimensão de uma matriz) pode ser obtido com o atributo *comprimento* (A.comprimento, $B[i]$.comprimento), tal como em Java.

3. O acesso aos atributos de um registro usará uma notação semelhante à da linguagem C. Se R é um registro, $R.id$ indica o atributo id de R. Se r é uma referência (ponteiro) para um registro, $r{\rightarrow}id$ indica o atributo id do registro referenciado por r. O endereço de uma variável ou estrutura pode ser obtido com o atributo *endereço* (por exemplo, $r \leftarrow R$.endereço). O valor nulo para ponteiros é *NIL*.

4. Instruções simples serão colocadas em linhas individuais e encerradas com ";" (ponto-e-vírgula):

   i ← 0;

   Ocasionalmente, por questões de espaço, diversas instruções triviais poderão ocupar a mesma linha.

   i ← 0; j ← 1;

   Em qualquer ponto de um algoritmo, uma instrução individual pode ser substituída por um bloco de instruções que virão indentadas e delimitadas à esquerda por uma barra vertical, tal como mostrado nas linhas 4–6 do algoritmo 1.1; o bloco será encerrado por uma palavra indicativa (linha 7).

5. Atribuições serão indicadas pelo símbolo "←", tal como mostrado acima. Se o mesmo valor deve ser atribuído a diversas variáveis, pode-se empregar a seguinte notação:

   i ← j ← 0;

   A troca de valores entre duas variáveis é abreviada com o operador ↔:

   i ↔ j;

6. Expressões matemáticas utilizarão os operadores aritméticos usuais: + (adição), − (subtração), · ou × (multiplicação), ÷ ou / (divisão), mod (módulo), mas com um formato mais livre que as linguagens de programação, podendo usar funções e outros operadores tal como em um texto matemático[3]:

   $x \leftarrow a^2 + \sin(y) + \pi/2 - \lfloor \lg z \rfloor$

7. Expressões relacionais utilizarão os operadores usuais: = (igual), ≠ (diferente), > (maior), < (menor), ⩾ (maior ou igual), ⩽ (menor ou igual).

8. Expressões lógicas utilizarão os operadores **ou**, **e** e **não**. **ou** e **e** são operadores de curto-circuito (isto significa que, se o valor do primeiro operando de **ou** for "verdadeiro", o segundo não será avaliado. O mesmo acontece com **e** para o valor "falso").

9. Instruções condicionais poderão assumir os seguintes formatos:

   se *teste* **então** instrução;
   se *teste* **então** instrução **senão** instrução;

   cujo funcionamento deve ser conhecido pelos leitores.

10. Os seguintes laços serão utilizados (supõe-se que o seu funcionamento também é conhecido):

    enquanto *teste* **faça** instrução;
    repita instrução **até** *teste*;
    para *i=1* **até** *max* **faça** instrução;

    Repare que, ao final do último laço, a variável i continua acessível com uma valor *max+1*.

11. O valor retornado por uma função será indicado pela instrução **retorna** *valor;*. Se utilizado sem um valor, **retorna** apenas interrompe a execução do procedimento.

12. Procedimentos e funções utilizarão passagem de parâmetros por valor para variáveis simples, e por referência para vetores, matrizes e registros.

[3] O significado das notações mais utilizadas pode ser encontrado na página xv.

Execução	*dividendo*	*divisor*	*resto*
1	90	55	35
2	55	35	20
3	35	20	15
4	20	15	5
5	15	5	0

Tabela 1.1: Exemplo de execução de `Mdc(90,55)`

13. Comentários usarão uma sintaxe semelhante à do C:

    se *teste* **então** instrução; `// comentário`

    ou

    se *teste* **então** instrução; `/* comentário */`

Vejamos agora o algoritmo 1.1 em ação através de um exemplo. Suponha que seja necessário encontrar o *mdc* entre 90 e 55. Uma execução de `Mdc(90,55)` (ou seja, uma execução do algoritmo com 90 no lugar do parâmetro a e 55 no lugar de b) ocorreria da seguinte forma:

- Inicialmente, as linhas 1 e 2 atribuem as os valores 90 e 55 às variáveis de trabalho dividendo e divisor, respectivamente.
- Na linha 3 o valor de divisor é comparado com 0. Como os valores são diferentes, o laço inicia sua a execução.
- No corpo do laço (linhas 4–6), calcula-se o resto da divisão de dividendo por divisor, e armazena-se o resultado na variável de trabalho resto (neste exemplo, o valor armazenado será 35). A seguir, divisor torna-se o novo dividendo e resto o novo divisor, e o teste do laço volta a ser executado.
- O corpo do laço será repetidamente executado como mostrado acima até que o valor de divisor se torne nulo. Na tabela 1.1, temos os valores das variáveis dividendo, divisor e resto a cada execução do laço (os valores são aqueles encontrados após a execução da linha 4).
- Após o término do laço, o algoritmo retornará o último valor armazenado na variável dividendo, ou seja, 5 (lembre-se que os últimos valores de divisor e resto mostrados na tabela são copiados, respectivamente, para dividendo e divisor pelas instruções das linhas 5 e 6). Este valor corresponde ao *mdc* de 90 e 55.

Será que o algoritmo 1.1 obedece aos critérios propostos na página 2? Comparando, podemos notar o seguinte:

1. O algoritmo acima é constituído por um número finito de passos (a descrição é composta por um número finito de linhas).
2. A entrada é composta de um par de números inteiros não negativos[4]. Estes números devem ser fornecidos pelo usuário do algoritmo e serão utilizados para a execução dos cálculos.
3. A saída consiste de um número inteiro que corresponde ao máximo divisor comum dos dados de entrada.
4. As operações do algoritmo são bem definidas:
   (a) As linhas 1, 2, 4, 5, 6 pedem que seja feita a atribuição de um valor a uma variável.
   (b) Na linha 4 deve ser efetuada uma divisão de números inteiros de forma a obter o *resto* dessa divisão.

[4] O máximo divisor comum pode ser também definido para inteiros negativos, mas isso não será feito aqui. Os interessados podem encontrar detalhes em Buchmann (2002, p. 20–23).

(c) Na linha 3, o valor de uma variável deve ser comparado com o valor constante 0. Esta linha indica também que as operações das linhas 4 a 6 devem ser repetidas enquanto o resultado da comparação for verdadeiro.

(d) A linha 8 indica que o valor de uma das variáveis deve ser retornado como o resultado final da execução do algoritmo.

Todas esses passos estão perfeitamente definidos, desde que saibamos executar as operações solicitadas, que estamos aqui considerando como primitivas e conhecidas. Por exemplo, para executar esse algoritmo, supõe-se que sabemos como dividir dois números inteiros e tomar o resto. Se isso não fosse verdade, necessitaríamos de um outro algoritmo que nos ensinasse como dividir, utilizando operações mais básicas.

5. Todos os passos são efetivos. A operação mais complicada do algoritmo é a divisão da linha 4. Se os operandos forem muito grandes, um ser humano, utilizando apenas lápis e papel, poderia encontrar alguma dificuldade em realizar a operação, e o processo poderia ser muito sujeito a erros. Porém, *em princípio*, isso poderia ser feito.

Todos esses critérios são, portanto, atendidos pelo nosso algoritmo. O critério final de *correção*, entretanto, merece uma discussão mais detalhada. Não é claro que esse algoritmo calcule o *mdc* de dois inteiros; na verdade, este é um fato que precisa ser demonstrado[5]. Não podemos afirmar, portanto, que o algoritmo apresentado é correto. Isto nos leva à seguinte questão: quando podemos dizer que um algoritmo é correto?

1.2 Correção e Eficiência

Voltando à definição de algoritmo dada na página 2, notamos que o critério de correção exige que o algoritmo termine em um número finito de passos, fornecendo o resultado correto. Isto envolve dois fatos:

1. O mais óbvio é que o algoritmo não pode realizar cálculos incorretos. Utilizando o algoritmo 1.1 como exemplo, dados dois números inteiros que atendam as restrições especificadas pela entrada, o resultado nunca pode ser diferente do valor correto para o máximo divisor comum desses dois inteiros. Não poderíamos exigir respostas corretas se fornecêssemos números negativos. Mas se os números informados forem inteiros não negativos, nenhum erro deve ser tolerado, quaisquer que sejam os valores de entrada. A correção deve ser, portanto, demonstrada. Repare que não é suficiente fornecer um determinado número de exemplos para os quais o algoritmo funcione corretamente. Como há infinitos números inteiros, precisaríamos de infinitos exemplos para atender o critério. Assim, a correção deve ser demonstrada a partir de uma *prova*, ou seja, uma argumentação lógica que comprove que o algoritmo funciona corretamente em qualquer situação.

2. Um ponto mais sutil é que o algoritmo deve terminar em um número finito de passos, ou seja, ele deve sempre *parar*. É possível construir algoritmos que funcionem corretamente para a maioria dos valores de entrada, mas que continuem sendo executados indefinidamente para alguns outros. Para que o algoritmo esteja correto, ele deve sempre parar em algum ponto. Este fato também deve ser demonstrado através de uma prova. *Observação*: repare que o fato do algoritmo ser constituído de um número finito de passos não assegura que ele irá sempre terminar. O número de passos pode ser finito e, ainda assim, estes passos podem ser executados infinitas vezes.

As técnicas para a demonstração de correção de algoritmos serão estudadas no capítulo 3. Por enquanto, a correção do algoritmo 1.1 ficará em aberto.

[5] Os leitores com forte inclinação matemática poderiam questionar até mesmo a existência de algo como um "máximo divisor comum" de dois inteiros. Este fato também necessita de demonstração, mas esse nível de rigor matemático extrapola as intenções do texto. Interessados podem encontrar detalhes em Buchmann (2002, cap. 1).

Algoritmo 1.2: `Soma(`A`)`

Entrada: Vetor de inteiros A
Saída: Somatório dos elementos de A: $\sum_{i=1}^{n} A[i]$, onde n é o tamanho de A

```
1 soma ← 0;
2 para i = 1 até A.comprimento faça
3 |   soma ← soma + A[i];
4 fim para
5 retorna soma;
```

Outro aspecto muito importante no estudo dos algoritmos, embora não mencionado na definição, é a sua *eficiência*. Dado um problema, é possível construir diversos algoritmos diferentes para resolvê-lo. Se todos esses algoritmos são corretos, qual deles deve ser escolhido como o melhor? Obviamente, o algoritmo escolhido deve ser o mais eficiente.

Quando falamos da eficiência de um algoritmo, estamos nos referindo ao uso que ele faz de determinado recurso. O melhor algoritmo é aquele que, para resolver um problema, consome ou exige menos recursos que os demais. Em geral, os recursos mais importantes são a *memória* e o *tempo de processamento*.

A memória refere-se à área de armazenamento representada pelas variáveis e estruturas utilizadas pelo algoritmo durante a sua execução. É evidente que o algoritmo mais econômico é preferível. Por exemplo, se o algoritmo a_1 utiliza apenas algumas poucas variáveis inteiras, e o algoritmo a_2 precisa de um vetor inteiros de 1000 posições para realizar a mesma atividade, certamente a_1 é mais eficiente que a_2.

Entretanto, o aspecto que mais nos interessa quando comparamos a eficiência de algoritmos é seu tempo de execução. Preferimos sempre o algoritmo mais rápido, aquele que realiza a tarefa em menos tempo. Para diversos problemas, a diferença no tempo de execução dos algoritmos pode ser bastante significativa. Por vezes, encontrar um algoritmo melhor pode representar uma economia de tempo apreciável.

Para tornar esta discussão mais concreta, associaremos a cada algoritmo um custo C para representar o consumo que ele faz de um determinado recurso. Por exemplo, o algoritmo a_1 terá um custo C_1 representando seu tempo de execução e, da mesma forma, a_2 terá um custo C_2. Dizemos que a_1 é mais eficiente que a_2 se $C_1 < C_2$. Para calcular C_1 e C_2 precisamos, basicamente, de duas informações:

1. Quanto tempo é necessário para executar cada instrução?
2. Quantas vezes cada uma delas é executada?

De posse desses dados, podemos fazer o cálculo dos custos multiplicando o custo individual de cada instrução pelo número de vezes que ela será executada e somando todos os resultados. O custo de um algoritmo é denominado, mais precisamente, de *complexidade de tempo*.

Vamos agora fazer uma distinção entre um *problema* e uma *instância* de um problema. Algoritmos são construídos para sistematizar a solução de problemas. Por exemplo, já mostramos um algoritmo que permite encontrar o *mdc* de dois inteiros não negativos. Neste caso, o cálculo do *mdc* representa o problema tratado pelo algoritmo. Entretanto, no ato da execução, precisamos fornecer valores concretos, tais como 90 e 55, que possam ser manipulados. O cálculo do *mdc* de 90 e 55 é uma instância do problema; outra poderia ser o cálculo do *mdc* de 104 e 72.

Podemos agora entender um fato muito importante a respeito da eficiência dos algoritmos: a complexidade C depende do tamanho da instância. Seja, por exemplo, o algoritmo 1.2 que calcula a soma dos elementos de um vetor de inteiros. Quantas instruções serão executadas por ele? Isto depende do tamanho do vetor A fornecido, ou seja, do tamanho da instância do problema. Quanto maior o vetor, mais vezes as operações das linhas 2 e 3 serão executadas.

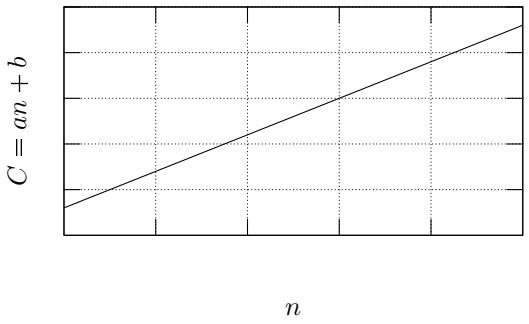

Figura 1.1: Complexidade do algoritmo 1.2

Desta forma, chegamos a um fato muito importante no cálculo da eficiência de um algoritmo: *o custo de execução depende do tamanho da entrada.* Ou, utilizando um jargão matemático, a complexidade é função do tamanho da entrada. Podemos escrever, com n representando o tamanho em questão:

$$C = f(n)$$

Colocando de outra forma: seja t_i o tempo de execução de uma determinada instrução i e k_i o número de vezes que ela é executada por determinado algoritmo. Então a complexidade C é:

$$C = \sum_i k_i t_i$$

onde o somatório é sobre todas as instruções utilizadas pelo algoritmo. Vejamos como exemplo a complexidade do algoritmo 1.2. Suponha que ele está trabalhando sobre um vetor A de tamanho n. São executados quatro tipos de instruções:

1. Atribuição: tempo de execução t_1. Há atribuições na linha 1 (executada uma vez) e na linha 3 (executada uma vez para cada posição do vetor, ou seja, n vezes).
2. Adição: tempo de execução t_2. Executada na linha 3, ou seja, n vezes.
3. Controle do laço **para**: tempo de execução t_3. Executada $n + 1$ vezes na linha 2 (uma vez para cada posição do vetor e mais uma no encerramento do laço).
4. Retorno da função: tempo de execução t_4. Executada apena uma vez na linha 5.

Assim, a complexidade é:

$$C = (n + 1)t_1 + nt_2 + (n + 1)t_3 + t_4 = n(t_1 + t_2 + t_3) + (t_1 + t_3 + t_4)$$

Fazendo $a = (t_1 + t_2 + t_3)$ e $b = (t_1 + t_3 + t_4)$, vem $C = an + b$, ou seja, a complexidade varia linearmente com n. A figura 1.1 mostra um gráfico desta função.

Desta discussão poderiam surgir duas questões:

1. Não seria possível comparar o tempo de execução de dois algoritmos simplesmente implementando-os, executando-os, cronometrando as execuções e confrontando os tempos obtidos?
2. O esforço de analisar os algoritmos é realmente necessário? Afinal, com a evolução galopante dos equipamentos de informática, não bastaria apenas utilizar um equipamento melhor para superar as deficiências de um determinado algoritmo, no lugar de construir outro?

A resposta à primeira pergunta é "sim", mas com ressalvas. O problema com essa estratégia de comparação é que diversos fatores alheios às características próprias de um algoritmo podem influenciar

	$n = 10$	$n = 20$	$n = 40$	$n = 80$	$n = 160$
a_1 $(C_1 = n)$	10	20	40	80	160
a_2 $(C_2 = n^2)$	100	400	1600	6400	25600
a_3 $(C_3 = 2^n)$	1024	1048576	$\approx 1,1 \times 10^{12}$	$\approx 1,2 \times 10^{24}$	$\approx 1,5 \times 10^{48}$

Tabela 1.2: Tempos de execução para diferentes algoritmos (ms)

no tempo de execução de sua implementação: equipamento e sistema operacional utilizados, ambiente de execução (o que mais está executando simultaneamente e competindo por recursos?), habilidade do programador, qualidade do código gerado pelo compilador, etc. A única forma de isolar um algoritmo desse "ambiente externo" e chegar a um custo que só leve em conta suas características intrínsecas é fazer uma análise matemática teórica, tal como descrito acima. Ainda assim, a análise experimental é útil, principalmente em determinadas situações onde a comparação teórica pode levar a resultados não muito conclusivos. Os capítulos 4 a 8 discutirão as técnicas necessárias para a realização da análise teórica, enquanto que a análise experimental será vista no capítulo 9.

A segunda questão será respondida através de um exemplo. Sejam três algoritmos a_1, a_2 e a_3 para o mesmo problema, com complexidades $C_1 = n$, $C_2 = n^2$ e $C_3 = 2^n$ (vamos supor que os custos são dados em ms [milisegundos]). Ou seja, a complexidade do primeiro algoritmo varia linearmente com o tamanho da entrada, a do segundo com seu quadrado, e a do terceiro exponencialmente em relação ao tamanho da entrada. Calculando as complexidades desses algoritmos para entradas de diversos tamanhos, chegamos aos resultados mostrados na tabela 1.2.

Os dados apresentados na tabela são muito significativos. Para entradas com tamanho $n = 10$, as velocidades variam entre 10 ms para a_1 e aproximadamente 1 s para a_3. A complexidade do algoritmo a_1 cresce linearmente, de forma que, para $n = 160$, $C_1 = 160$ ms, ou seja, ainda inferior a dois décimos de segundo.

Para a_2, o crescimento do tempo de execução é mais acentuado, uma vez que a variação é quadrática com o tamanho da entrada. Para $n = 160$, C_2 se aproxima da marca de 30 s, valor que já pode começar a incomodar um usuário. Entretanto, se conseguirmos um equipamento mil vezes mais eficiente, a solução pode voltar a ser encontrada em tempos inferiores a 1 s.

A situação se altera drasticamente para a_3. Apenas dobrando o valor inicial da entrada (de $n = 10$ para $n = 20$), a complexidade C_3 passa de 1 s para aproximadamente 17,5 min. O valor para $n = 40$ é de 34,8 anos, e os dois últimos valores ultrapassam a idade estimada do universo, que é de 11 bilhões de anos. Isto significa que a_3 não pode ser utilizado, a não ser para valores de n muito pequenos, ou não seremos capazes de ver o processamento do algoritmo terminar. Nenhuma evolução na capacidade de processamento dos equipamentos computacionais será capaz de vencer as limitações desse algoritmo[6]. Encontrar um algoritmo mais eficiente torna-se, então, imprescindível. Problemas para os quais só existem algoritmos de eficiência análoga à de a_3 são considerados *problemas difíceis*. A figura 1.2 mostra os gráficos das funções discutidas para que as taxas de variação possam ser comparadas. Repare como a taxa de crescimento de 2^n é significativamente maior que as das demais funções.

Ainda há um aspecto a discutir em relação à eficiência dos algoritmos. Se temos dois algoritmos para resolver um problema, podemos compará-los e escolher o de menor complexidade. Entretanto, uma questão permanece: será que existe um terceiro algoritmo que seja mais eficiente que ambos? E se esse algoritmo for encontrado, existiria um quarto ainda melhor? Quando deveríamos nos dar por satisfeitos e encerrar a busca pelo algoritmo mais eficiente?

A resposta a essas perguntas envolve questões muito importantes: como encontrar o *melhor* algoritmo que resolva um determinado problema? Será que existe um limite inferior para a complexidade dos algoritmos que resolvem o problema? Além disso, existem realmente programas intrinsecamente difíceis, para os quais não seja possível encontrar um algoritmo eficiente? Encontrar essas respostas exige a análise do *problema* a resolver, baseando-se apenas em suas características, e não em qualquer algoritmo particular. Técnicas para este tipo de análise encontram-se nos capítulos 10 e 11.

[6] Pelo menos até a criação de um computador quântico funcional...

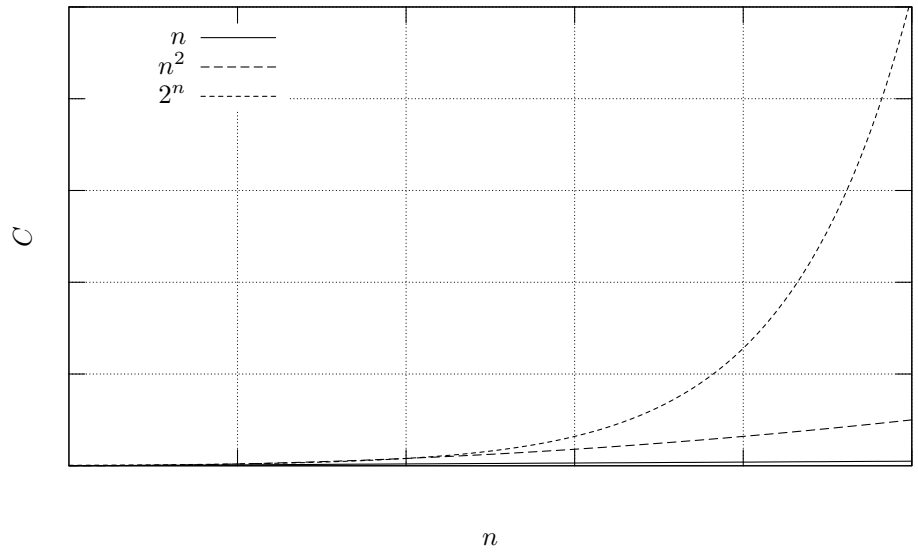

Figura 1.2: Taxa de variação para $C = n$, $C = n^2$ e $C = 2^n$

Em resumo, a disciplina que chamamos de *análise de algoritmos*, que é o objeto do nosso estudo, consiste, basicamente, em encontrar resposta para as perguntas abaixo (LEE et al., 2005, p. 17):

1. Como determinar a eficiência de um algoritmo?
2. Como saber se um algoritmo é ótimo para um determinado problema (ou seja, se é o melhor algoritmo que resolve esse problema)?
3. Como medir a dificuldade de um problema?

1.3 Implementação

Repare que, da forma como foi definido, um algoritmo não está relacionado, em princípio, a nenhum equipamento de execução tal como um computador. Na verdade, os primeiros algoritmos (por exemplo o de Euclides) surgiram alguns séculos antes de Cristo, e mesmo a disciplina denominada Teoria da Computação, tal como a conhecemos hoje, começou a se desenvolver antes da invenção dos primeiros computadores.

Nosso interesse, entretanto, é estudar algoritmos que serão processados por computadores. Para tanto, os algoritmos devem ser *implementados* em alguma linguagem de programação, ou seja, traduzidos para uma linguagem para a qual exista um compilador ou interpretador funcionando na máquina alvo.

Para que esta tradução possa ser feita com facilidade, é interessante que as instruções utilizadas no algoritmo correspondam de perto às instruções da linguagem desejada. Isto minimiza o esforço de implementação e reduz o risco de ocorrência de erros no processo de tradução. O pseudo-código descrito na página 2 é muito próximo das linguagens imperativas mais comuns, tais como Pascal e C, e pode ser traduzido para elas com facilidade. Estas duas linguagens serão utilizadas neste texto para implementar todos os algoritmos que analisarmos. Por exemplo, a implementação do algoritmo 1.1 em Pascal encontra-se no programa 1.1, e a implementação em C no programa 1.2. Para os demais algoritmos, as implementações podem ser encontradas nos apêndices A e B.

Programa 1.1: Mdc — (Pascal)

```
function Mdc(a,b:integer) : integer;
  var dividendo, divisor, resto : integer;
```

```
  begin

    dividendo := a;
    divisor := b;
    while divisor <> 0 do begin
      resto := dividendo mod divisor;
      dividendo := divisor;
      divisor := resto;
    end;

    Mdc := dividendo;
  end;
```

Programa 1.2: Mdc — (C)

```
int mdc(int a, int b) {
  int dividendo, divisor, resto;

  dividendo = a;
  divisor = b;

  while (divisor != 0) {
    resto = dividendo % divisor;
    dividendo = divisor;
    divisor = resto;
  }

  return dividendo;
}
```

1.4 Recursividade.

Definição e exemplos

Nesta seção, iremos estudar uma poderosa técnica de programação denominada recursividade. Informalmente, recursividade ocorre quando uma entidade é definida em termos de si mesma. Por exemplo, seja a seguinte definição para a função $f(x) = x^n$, com x real e n inteiro:

$$x^n = \begin{cases} 1 & \text{para } n = 0 \\ 1/x^{-n} & \text{para } n < 0 \\ (x^{n/2})^2 & \text{para } n > 0;\ n \text{ par} \\ x \cdot x^{n-1} & \text{para } n > 0;\ n \text{ ímpar} \end{cases}$$

Observe que, para os casos em que $n \neq 0$, x^n é aplicado em sua própria definição. Este exemplo simples já mostra as características necessárias para o correto emprego da recursividade:

1. Devem existir um ou mais *passos recursivos*, onde o elemento em questão é definido em termos de versões simplificadas dele mesmo. Por exemplo, para o caso de n par, a definição de x^n utiliza $x^{n/2}$. Os casos de n ímpar e $n < 0$ têm tratamento semelhante.

2. Devem existir também um ou mais *casos base*, que não utilizam recursividade e são, em geral, simples o suficiente para serem obtidos diretamente. Na definição acima, isto é exemplificado por $x^0 = 1$. Repare que um caso base especifica uma *condição de parada*, sem a qual nunca conseguiríamos escapar das definições recursivas.

Algoritmo 1.3: `Potencia(`x`,`n`)`

Entrada: Números $x \in \mathbb{R}$ e $n \in \mathbb{Z}$
Saída: x^n
Implementação: Programas A.1 e B.1

```
1 se n = 0 então retorna 1;
2 se n < 0 então retorna 1/ Potencia(x, −n);
3 se n mod 2 = 0 então retorna (Potencia(x, n/2))^2;
4 retorna x · Potencia(x, n − 1);
```

Vejamos como esta definição funciona. Suponha que queremos calcular 2^{-7}. Como -7 é menor que 0, utilizamos o passo recursivo correspondente a $n < 0$ para determinar que $2^{-7} = 1/2^7$, e agora devemos calcular 2^7. Voltando à definição, vemos que precisamos do caso para n ímpar, e assim sucessivamente, até que o caso base seja atingido. O cálculo completo, então, é:

$$\begin{aligned} 2^{-7} &= 1/2^7 = 1/(2 \cdot 2^6) = 1/(2(2^3)^2) = 1/(2(2 \cdot 2^2)^2) = 1/(2(2(2^1)^2)^2) = \\ &= 1/(2(2(2 \cdot 2^0)^2)^2) \end{aligned}$$

quando então atingimos o caso base $2^0 = 1$ que não necessita de mais recursões, e pode ser calculado diretamente. Prosseguindo:

$$\begin{aligned} 2^{-7} &= 1/(2(2(2 \cdot 1)^2)^2) = 1/(2(2 \cdot 2^2)^2) = 1/(2(2 \cdot 4)^2) = 1/(2 \cdot 8^2) = \\ &= 1/(2 \cdot 64) = 1/128 = 0,0078125 \end{aligned}$$

Em termos de programação, podemos ter módulos (funções, procedimentos) recursivos. O fato de um módulo recursivo ser definido em termos de si mesmo indica que, em seu corpo, podemos encontrar chamadas para o próprio módulo. Vamos exemplificar criando uma função recursiva que faça o cálculo de x^n utilizando nossa definição. Esta função será denominada `Potencia(`x`,`n`)`, com $x \in \mathbb{R}$ e $n \in \mathbb{Z}$, e está mostrada no algoritmo 1.3. Repare que a própria função que está sendo definida, `Potencia(`x`,`n`)`, contém chamadas para si mesma nas linhas 2, 3 e 4.

Vejamos como funciona a execução desse algoritmo quando é feita uma chamada `Potencia(`*2.0*`,-`*7*`)`. A execução começa com $x = 2.0$ e $n = -7$, e portanto o teste da linha 1 falha, mas o da linha 2 retorna *verdadeiro*, fazendo com que a expressão `1/ Potencia(`x`,`$-n$`)` seja avaliada. Isto provoca uma chamada de `Potencia(`*2.0*`,`*7*`)`. Neste ponto, como ocorre sempre que uma função qualquer é chamada por uma outra, a execução de `Potencia(`*2.0*`,-`*7*`)` é interrompida e `Potencia(`*2.0*`,`*7*`)` começa a ser processada. Quando esta última terminar, o processamento de `Potencia(`*2.0*`,-`*7*`)` prossegue normalmente na instrução seguinte (que é a divisão de 1 pelo resultado retornado por `Potencia(`*2.0*`,-`*7*`)`).

É importante entender que cada chamada recursiva terá seu próprio conjunto de parâmetros e variáveis locais. Por exemplo, na primeira chamada tem-se $x = 2.0$ e $n = -7$ e, na segunda, $x = 2.0$ e $n = 7$. Embora tenham o mesmo nome, os parâmetros n das duas chamadas são variáveis inteiramente distintas, e uma alteração feita em uma delas *não* terá efeito sobre a outra. O mesmo vale para todos os parâmetros e para todas as variáveis locais porventura utilizadas.

A figura 1.3 mostra a execução completa de `Potencia(`*2.0*`, -`*7*`)`, com todas as chamadas recursivas. Cada chamada está indentada em relação à sua respectiva chamadora, e os números de linha do algoritmo 1.3 que provocam cada uma delas estão indicados.

A partir desse exemplo simples, podemos estabelecer alguns passos básicos para a construção de funções ou procedimentos recursivos. Para evitar a repetição da expressão "função ou procedimento" no texto a seguir, utilizaremos apenas "função". Os passos são os seguintes (BARTLETT, 2005):

1. Inicialização: é necessário fornecer valores iniciais com os quais a função possa trabalhar. Em geral, isso é feito através de uma chamada não recursiva, a partir de outra função, que fornece os valores iniciais dos parâmetros (por exemplo, a chamada inicial de `Potencia(`*2.0*`,-`*7*`)`).
2. Caso base: deve-se verificar se os valores em processamento coincidem com um dos casos base, que podem ser calculados diretamente, sem novas chamadas recursivas. O resultado desse cálculo

```
Potencia(2.0,-7)
↳1/ Potencia(2.0,7) (linha 2)
    ↳2.0 × Potencia(2.0,6) (linha 4)
        ↳Potencia(2.0,3)^2 (linha 3)
          ↳2.0 × Potencia(2.0,2) (linha 4)
              ↳Potencia(2.0,1)^2 (linha 3)
                ↳2.0 × Potencia(2.0,0) (linha 4)
                    ↳1↴ (linha 1)
                ↳2.0 × 1 = 2↴
              ↳2^2 = 4↴
          ↳2.0 × 4 = 8↴
        ↳8^2 = 64↴
    ↳2.0 × 64 = 128↴
↳1/128 = 0.0078125↴
0.0078125
```

Figura 1.3: Execução de `Potencia(`*2.0*`,-7)`

Algoritmo 1.4: `Mdc(`a`,`b`)` — recursivo

Entrada: Inteiros não negativos a e b
Saída: Máximo divisor comum de a e b
Implementação: Programas A.2 e B.2

```
1 se b = 0 então retorna a;
2 retorna Mdc(b,a mod b);
```

deve ser retornado. No algoritmo 1.3, isto é feito na linha 1. Repare que, sem caso base, as funções recursivas entram em laço infinito.

3. Passo recursivo: o problema original deve ser redefinido em termos de um ou mais sub-problemas, que são versões mais simples ou menores de si mesmo. No exemplo, isto foi feito quando se redefiniu x^n em termos de $1/x^{-n}$, $x^{n/2}$ ou x^{n-1}.

4. Chamada recursiva: para cada um dos passos recursivos, construir uma chamada recursiva correspondente. Testar as condições para a execução de uma determinada chamada, executá-la e coletar o resultado. Veja as linhas 2, 3 e 4 do algoritmo. Em alguns casos, uma mesma condição pode provocar mais de uma chamada recursiva, embora isto não ocorra no nosso exemplo.

5. Finalização: combinar os resultados obtidos pela execução dos sub-problemas em um resultado final e retorná-lo. No algoritmo 1.3 isto ocorre quando, nas linhas 2, 3 e 4, efetuamos as operações necessárias para obter o valor final a partir dos resultados das chamadas recursivas. Por vezes, este passo pode ser trivial, apenas retornando o mesmo resultado obtido.

Vejamos mais um exemplo simples: uma versão recursiva do algoritmo para o cálculo do máximo divisor comum. Veja o algoritmo 1.4, que se baseia no seguinte fato: para $b \neq 0$, $\text{mdc}(a, b) = \text{mdc}(b, a \bmod b)$. Na verdade, é exatamente isto que é feito também pelo algoritmo iterativo (algoritmo 1.1), que em cada iteração do laço das linhas 3 a 7 faz com que o antigo divisor se torne o dividendo, e que o resto de dividendo/divisor se torne o novo divisor. A figura 1.4 ilustra a execução do algoritmo 1.4 para $a = 90$ e $b = 55$.

Um outro exemplo clássico de função recursiva é o cálculo automático das soluções de um quebra-cabeças denominado Torre de Hanói. Este jogo consiste em três pinos A, B e C e um conjunto de n discos de tamanhos diferentes, com um furo central de modo que possam ser introduzidos nos pinos. Inicialmente, todos os discos estão inseridos no pino A. A figura 1.5 mostra esta situação para um conjunto com $n = 3$ discos. O problema consiste em transferir, utilizando o menor número de movimentos, todos os discos de A para B, obedecendo às seguintes regras simples:

- Apenas um dos discos pode ser movido por vez.

```
Mdc(90,55)
↳Mdc(55,35) (linha 2)
  ↳Mdc(35,20) (linha 2)
    ↳Mdc(20,15) (linha 2)
      ↳Mdc(15,5) (linha 2)
        ↳Mdc(5,0) (linha 2)
          ↳5↴ (linha 1)
        ↳5↴
      ↳5↴
    ↳5↴
  ↳5↴
↳5↴
5
```

Figura 1.4: Execução de `Mdc(`*90*`,`*55*`)`

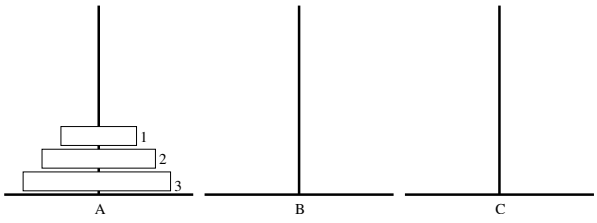

Figura 1.5: Torre de Hanói com 3 discos

- Um disco nunca pode ser colocado sobre outro de tamanho menor.

A seguinte lista de movimentos resolve o problema para o caso de $n = 3$ discos (com os discos numerados de 1 a 3 do menor para o maior):

1. Mover disco 1 de A para B.
2. Mover disco 2 de A para C.
3. Mover disco 1 de B para C.
4. Mover disco 3 de A para B.
5. Mover disco 1 de C para A.
6. Mover disco 2 de C para B.
7. Mover disco 1 de A para B.

Como podemos construir uma função recursiva que resolva esse problema? Pensando recursivamente, chegamos à seguinte conclusão: para transferir n discos do pino A para o pino B (usando o pino C como auxiliar), precisamos:

1. Transferir $n-1$ discos de A para C, usando B como auxiliar (passos 1–3 acima).
2. Mover o disco n de A para B (passo 4).
3. Transferir $n-1$ discos de B para C, usando A como auxiliar (passos 5–7 acima).

Entretanto, o problema de transferir $n-1$ discos de um pino de origem para um outro de destino é inteiramente análogo ao nosso problema original. Repare então que a solução do problema depende

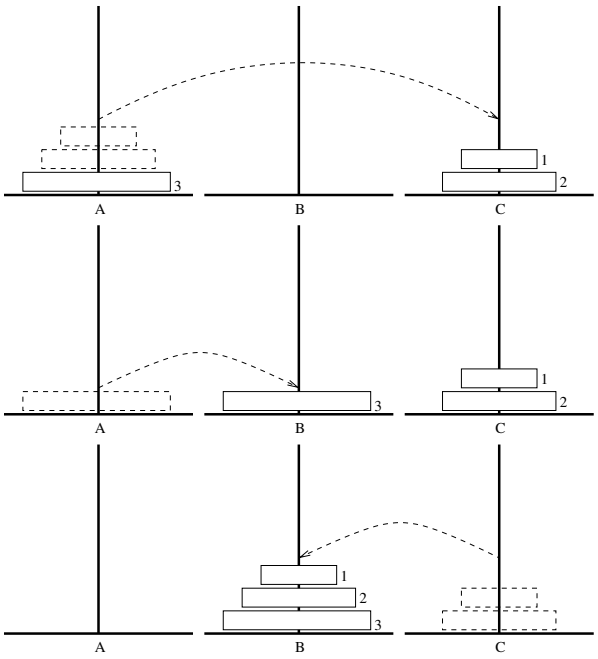

Figura 1.6: Solução (recursiva) da Torre de Hanói (3 discos)

Algoritmo 1.5: `Hanoi(`*disco*`,` *origem*`,` *destino*`,` *auxiliar*`)`

Entrada: Inteiro positivo *disco*: quantidade de discos a transferir. Caracteres *origem*, *destino* e *auxiliar*: identificadores dos pinos de origem, destino e auxiliar, respectivamente.
Saída: Lista de movimentos necessários para transferir *disco* discos de *origem* para *destino* (utilizando *auxiliar*).
Implementação: Programas A.3 e B.3

```
1 se disco > 1 então Hanoi(disco-1,origem,auxiliar,destino);
2 escreva "Mover disco"+ disco + "de"+ origem + "para"+ destino;
3 se disco > 1 então Hanoi(disco-1,auxiliar,destino,origem);
```

de conhecermos a solução para dois sub-problemas (passos 1 e 3 acima) que são versões simplificadas dele mesmo. Ou seja, se soubermos como transferir $n-1$ discos, saberemos também transferir n discos. Já temos, portanto, os passos recursivos. Falta-nos o caso base, que é trivial: para transferir 1 disco de um pino de origem para um pino de destino, basta apenas movê-lo da origem para o destino. Sistematizando, a solução recursiva para mover n discos, $n \geqslant 1$, do pino *origem* para o pino *destino* (utilizando o pino *auxiliar* como auxiliar) é:

1. Se $n > 1$, mover $n-1$ discos de *origem* para *auxiliar* (usando *destino* como auxiliar).
2. Mover o disco n de *origem* para *destino*.
3. Se $n > 1$, mover $n-1$ discos de *auxiliar* para *destino* (usando *origem* como auxiliar).

Veja um exemplo com 3 discos na figura 1.6. A implementação encontra-se no algoritmo 1.5. Você pode verificar que a execução de `Hanoi(`*3*`,`*'A'*`,`*'B'*`,`*'C'*`)` leva à lista de movimentos da na página anterior.

Utilização

Através da recursividade, podemos construir algoritmos mais simples, claros, concisos e fáceis de entender e manter[7]. Compare, por exemplo, as duas versões de algoritmos para o cálculo do máximo divisor

[7] Uma vez que já estejamos acostumados com o conceito, que não é intuitivo.

comum: a iterativa (algoritmo 1.1) e recursiva (algoritmo 1.4). Você pode também implementar uma solução iterativa para o problema da Torre de Hanói, e compará-la com o algoritmo 1.5. Entretanto, algoritmos recursivos tendem a ser menos eficientes que seus equivalentes iterativos e, portanto, não devem ser usados indiscriminadamente. Para entender a razão disto, é necessário compreender como a recursividade é implementada.

Em geral, chamadas de procedimentos, sejam recursivas ou não, são implementadas com o auxílio de uma pilha[8], onde são guardados diversos dados importantes sobre elas. Cada chamada provoca a criação e o empilhamento de um "ambiente de execução" denominado *registro de ativação*. A lista a seguir mostra exemplos de dados que são guardados em um registro de ativação (DROZDEK, 2002, p. 156):

- Valores de todos os parâmetros do procedimento no caso de passagem por valor, ou ponteiros no caso de passagem por referência.
- Variáveis locais ao procedimento.
- Endereço de retorno para que o procedimento chamador possa prosseguir sua execução no ponto correto ao final da execução do procedimento chamado.
- Ponteiro para o registro de ativação do procedimento chamador.
- Espaço para o valor de retorno, caso exista.

Diferentes chamadas do mesmo procedimento são representadas por registros de ativação diferentes, e o último registro empilhado corresponde à chamada que está sendo processada. Isto explica porque cada chamada tem seu próprio conjunto de parâmetros e variáveis locais que, embora tenham o mesmo nome, não se confundem com os das demais.

Para ilustrar, vamos voltar ao algoritmo 1.3 e acompanhar a execução de `Potencia(2.0,2)`. A chamada inicial cria o registro de ativação mostrado na parte (a) da figura 1.7[9]. O algoritmo prossegue para o cálculo da expressão aritmética da linha 3, que provoca uma chamada de `Potencia(2.0,1)`. O registro de ativação desta última chamada é então construído e empilhado, tal como mostrado na figura 1.7(b). Como agora n é ímpar, o cálculo da expressão da linha 4 faz com que seja necessário chamar `Potencia(2.0,0)`, cujo registro de ativação é também criado e empilhado tal como na figura 1.7(c).

Neste ponto, terminam as chamadas recursivas, e o valor de retorno da última chamada de `Potencia` é estabelecido tal como indicado na linha 1 (ver figura 1.7(d)). O término do procedimento provoca o desempilhamento de seu registro de ativação, e a execução de `Potencia(2.0,1)` pode prosseguir com o cálculo da expressão da linha 4, que estabelece o valor de retorno (figura 1.7(e)) e encerra, desempilhando também seu registro de ativação. Finalmente, a chamada original pode fazer o cálculo da linha 3 e estabelecer seu valor de retorno (figura 1.7(f)), que será devolvido ao chamador original quando este último registro de ativação for desempilhado.

É evidente que a manipulação dessas estruturas consome memória e tempo de processamento e, portanto, o uso exagerado de recursividade pode provocar uma queda apreciável no desempenho do algoritmo. Podemos falar do *nível de recursividade* como correspondendo ao número de registros de ativação empilhados em um determinado momento. Se representarmos as chamadas recursivas através de uma árvore, o nível de recursividade será dado pela altura dessa árvore (um exemplo de árvore de recursividade pode ser visto na figura 1.8). É importante que o nível de recursividade seja não apenas finito (o que é garantido pela execução eventual do caso base), mas também pouco profundo.

Quando então se deve usar recursividade? Não existe uma regra precisa, mas segundo Wirth (1989, p. 112), algoritmos recursivos são especialmente adequados quando o problema a ser resolvido já está definido de forma recursiva. Mesmo assim, deve-se proceder com cautela. Veja, por exemplo, o

[8] Uma pilha é uma estrutura de dados na qual todas as inserções, remoções e acessos são feitos em uma única extremidade denominada *topo*. Assim, o último elemento a ser inserido é o primeiro a ser retirado. Para detalhes, consulte Cormen et al. (2002, p. 163–164), Ziviani (2004, p. 72–80) e Szwarcfiter e Markenzon (1994, p. 28–31).

[9] Na figura 1.7, o conteúdo da pilha exibe apenas os parâmetros das chamadas e os valores de retorno.

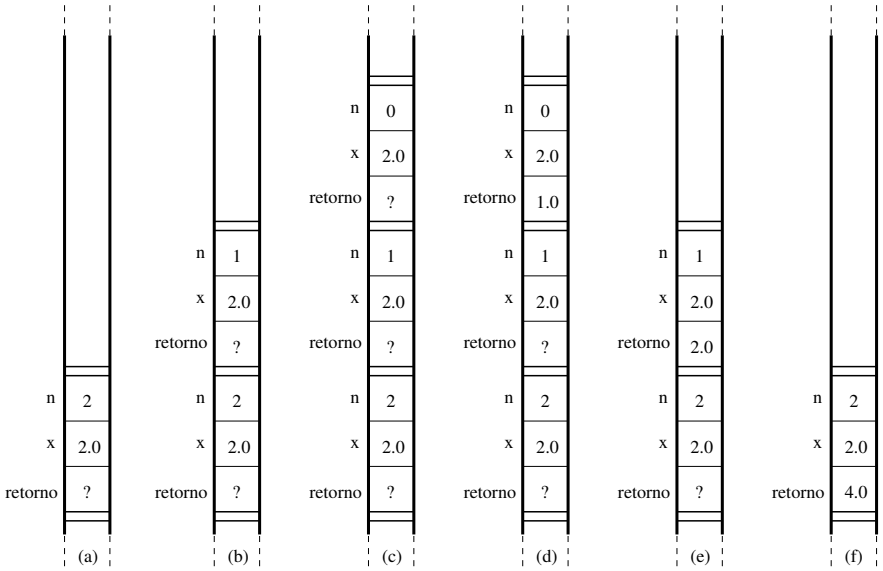

Figura 1.7: Pilha da execução de `Potencia(2.0,2)`

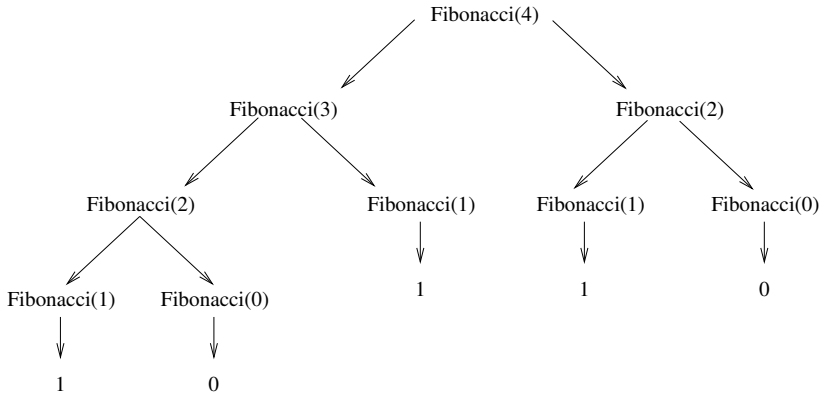

Figura 1.8: Árvore de recursividade de `Fibonacci(4)`

cálculo da série de Fibonacci. Este problema é muito utilizado para ensinar recursividade. Esta série é uma seqüência de números inteiros definida da seguinte forma:

- O primeiro elemento da série é igual a 0.
- O segundo elemento da série é igual a 1.
- Os elementos subseqüentes são sempre iguais à soma dos dois anteriores.

Os primeiros elementos da série de Fibonacci são:

$$0\ 1\ 1\ 2\ 3\ 5\ 8\ 13\ 21\ 34\ 55\ 89\ \ldots$$

Mais precisamente, a série pode ser considerada como uma função $F : \mathbb{N} \mapsto \mathbb{N}$ definida como:

$$F(n) = \begin{cases} 0 & \text{para } n = 0 \\ 1 & \text{para } n = 1 \\ F(n-1) + F(n-2) & \text{para } n > 1 \end{cases}$$

Note a definição recursiva, que leva diretamente ao algoritmo 1.6. Entretanto, cada chamada da função

Algoritmo 1.6: `Fibonacci(`n`)` — recursivo

Entrada: Inteiro não negativo n.
Saída: n-ésimo elemento da série de Fibonacci.
Implementação: Programas A.4 e B.4

```
1 se n ⩽ 1 então retorna n;
2 retorna Fibonacci(n-1) + Fibonacci(n-2);
```

Algoritmo 1.7: `Fibonacci(`n`)` — iterativo

Entrada: Inteiro não negativo n.
Saída: n-ésimo elemento da série de Fibonacci.
Implementação: Programas A.5 e B.5

```
1 se n ⩽ 1 então retorna n;
2 penultimo ← 0;
3 ultimo ← 1;
4 para i ← 2 até n faça
5 |   soma ← ultimo + penultimo;
6 |   penultimo ← ultimo;
7 |   ultimo ← soma;
8 fim para
9 retorna soma;
```

provoca, na linha 2, duas novas chamadas recursivas, o que leva a uma proliferação de chamadas (veja a figura 1.8, que mostra a árvore de recursividade para a uma chamada de `Fibonacci(4)`) e torna o algoritmo extremamente ineficiente. A versão iterativa (algoritmo 1.7) é ainda simples e muito mais eficiente. Isto torna o algoritmo 1.6 um excelente exemplo de onde *não* se deve empregar recursividade. Como sugestão, você deve implementar ambas as versões em sua linguagem de programação predileta e comparar seus tempos de execução. Utilize valores de n entre 40 e 50.

Resumindo, recursividade deve ser usada quando (ZIVIANI, 2004, p. 40):

1. O problema é naturalmente recursivo e o custo pode ser mantido baixo.
2. Não existe solução óbvia por iteração.

Uma última observação: na discussão acima, ficou implícita a idéia de que a todo procedimento recursivo corresponde um outro não-recursivo e vice-versa. Laços e chamadas recursivas podem substituir uns aos outros. Algumas vezes, como no caso da série de Fibonacci, pode-se encontrar um algoritmo iterativo correspondente ao recursivo de forma intuitiva. Freqüentemente, entretanto, a conversão exige a manipulação explícita de uma pilha, o que pode tornar o algoritmo bastante complexo e obscurecer seu entendimento. Existem algumas técnicas para sistematizar o procedimento de conversão, mas não iremos tratar delas neste texto. Para detalhes, consulte Horowitz e Sahni (1987, p. 149–150), Liu e Stoller (1999) e Bille (2001).

Classificação

Existem diferentes tipos de algoritmos recursivos. Iremos agora estudar alguns deles.

Recursividade direta e indireta

Todos os algoritmos vistos nesta seção, até agora, são exemplos de recursividade direta, em que um procedimento contém uma chamada explícita para si mesmo. Podemos também ter situações em que as chamadas recursivas são feitas de forma indireta. Por exemplo, seja um procedimento A que contém uma chamada para um procedimento B e este, por sua vez, chama outro procedimento C. Se este

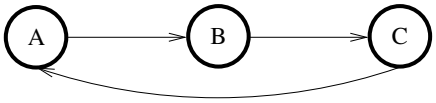

Figura 1.9: Recursividade indireta

Algoritmo 1.8: `Sen(`x`)` — recursivo

Entrada: Ângulo x (em radianos).
Saída: Seno de x.
Implementação: Programas A.6 e B.6

```
1  Sen(x) início
2  |  se x ⩽ 0.2 então retorna x · (1 − x²/6);
3  |  t ← (Tg(x/3))²;
4  |  retorna Sen(x/3) · (3 − t)/(1 + t);
5  fim
6  Tg(x) início
7  |  retorna Sen(x)/Cos(x);
8  fim
9  Cos(x) início
10 |  retorna 1 − 2 · (Sen(x/2))²;
11 fim
```

último tiver uma chamada para A, temos uma ocorrência de *recursividade indireta*, tal como mostrado na figura 1.9.

Qualquer número de procedimentos pode estar envolvido em um esquema de recursividade indireta. Se forem apenas dois, eles são chamados de *mutuamente recursivos*.

Vejamos um exemplo emprestado de Drozdek (2002, p. 167). Suponha que queremos calcular o valor do seno para um determinado ângulo x. Para um valor de x suficientemente pequeno, a seguinte equação é uma boa aproximação:

$$\sin x \approx x - \frac{x^3}{6} \tag{1.1}$$

com x em radianos[10]. Por exemplo, para $x \leqslant 0.2\,\text{rad}$, a equação 1.1 fornece uma aproximação correta até pelo menos a quinta casa decimal. Para valores maiores que o limite estabelecido, podemos utilizar as seguintes equações auxiliares[11] (que reduzem o valor de x):

$$\sin x = \sin\left(\frac{x}{3}\right) \cdot \frac{\left(3 - \tan^2\left(\frac{x}{3}\right)\right)}{\left(1 + \tan^2\left(\frac{x}{3}\right)\right)} \tag{1.2}$$

$$\tan x = \frac{\sin x}{\cos x} \tag{1.3}$$

$$\cos x = 1 - 2\sin^2\frac{x}{2} \tag{1.4}$$

As equações 1.2–1.4 permitem efetuar o cálculo de $\sin x$ através de recursividade direta e indireta, e a equação 1.1 atua como caso base. O algoritmo 1.8 mostra uma implementação desse esquema, e a figura 1.10 mostra a árvore de recursividade supondo que o valor de x seja tal que $x/3$ é inferior ao limite estabelecido pelo caso base.

O algoritmo mostrado é um tanto artificial, mas não é fácil encontrar exemplos simples e didáticos de recursividade indireta que tenham maior relevância. Este tipo de recursividade é muito empregado em uma técnica de compilação denominada *análise sintática descendente recursiva*. Não iremos detalhar esta técnica aqui, mas você deve consultar Aho, Sethi e Ullman (1988, p. 25–82) e Schildt (1997,

[10] Os leitores com inclinação matemática podem reconhecer na equação 1.1 os dois primeiros termos da expansão de $\sin x$ em uma série de Taylor (ver, por exemplo, Spiegel (1973, p. 111)).
[11] Para diversas fórmulas de manipulação trigonométrica, consulte Spiegel (1973, p. 11–20).

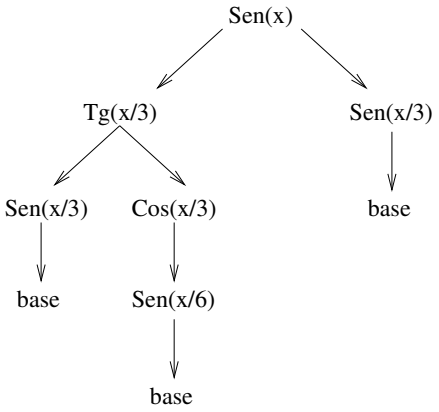

Figura 1.10: Árvore de recursividade de `Sen(x)`

p. 739–810) para obter maiores detalhes e inteirar-se de uma aplicação importante dos procedimentos recursivos.

Recursividade linear e em árvore

Pode-se classificar as chamadas recursivas em relação à forma de crescimento da árvore de recursividade. Existe *recursividade linear* quando, após o retorno da chamada recursiva, nenhuma operação pendente na função chamadora envolve novas chamadas recursivas. Ou seja, o crescimento do número de chamadas recursivas é linear (cada chamada recursiva gera, no máximo, uma nova) e a árvore de recursividade tem a forma de uma lista. Dentre os algoritmos apresentados, são lineares o `Potencia(`x`,`n`)` (1.3) e o `Mdc(`a`,`b`)` (1.4) . Por exemplo, após o retorno de todas as chamadas recursivas de `Potencia(`x`,`n`)`, (linhas 2, 3 e 4, ver p. 11), as únicas instruções executadas são operações aritméticas. A árvore de recursividade para a chamada `Potencia(2.0,2)` encontra-se na figura 1.11.

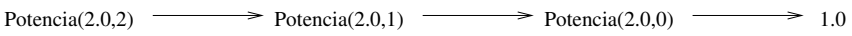

Figura 1.11: Árvore de recursividade de `Potencia(2.0,2)`

Na *recursividade em árvore*, as operações pendentes no retorno da chamada recursiva envolvem novas chamadas recursivas. Os algoritmos `Hanoi()` (1.5), `Fibonacci(`n`)` (1.6) e `Sen(`x`)` (1.8) são exemplos de recursividade em árvore. Neste caso, a árvore de recursividade é, literalmente, uma árvore, tal como mostrado nas figuras 1.8 e 1.10. Deve-se tomar especial cuidado no uso de recursividade em árvore, pois o crescimento do número de chamadas não é linear e pode tornar-se exponencial, levando a um desempenho muito ruim.

Recursividade de cauda

Um outro tipo importante de recursividade é a *recursividade de cauda*. Existe recursividade de cauda quando:

1. A chamada recursiva é a última operação executada pela função.
2. Não existem outras chamadas recursivas anteriores, diretas ou indiretas.

Por exemplo, o algoritmo `Potencia(`x`,`n`)` (1.3) não apresenta recursividade de cauda, pois existem sempre operações a serem executadas com o valor retornado pela chamada recursiva. Já o algoritmo `Mdc(`a`,`b`)` (1.4) apenas devolve o resultado retornado pela chamada recursiva, não realizando nenhuma outra operação, e representa um caso de recursividade de cauda. Toda recursividade de cauda é linear, mas o inverso não é verdadeiro.

Algoritmo 1.9: `Fibonacci(`n`)` — recursividade de cauda

Entrada: Inteiro não negativo n.
Saída: n-ésimo elemento da série de Fibonacci.
Implementação: Programas A.7 e B.7

```
1 Fib(n,ultimo,penultimo) início
2 |  se n = 0 então retorna penultimo;
3 |  retorna Fib(n − 1, ultimo + penultimo, ultimo);
4 fim
5 Fibonacci(n) início
6 |  retorna Fib(n,1,0);
7 fim
```

A importância da recursividade de cauda reside no fato de que ela pode ser facilmente substituída por um laço. Ou seja, é muito fácil converter um procedimento com recursividade de cauda para uma versão iterativa. Alguns compiladores são capazes de realizar automaticamente este tipo de otimização.

A razão disso é que, como a chamada recursiva é a última instrução executada, não é necessário guardar o estado do procedimento chamador (ou seja, seu registro de ativação) para que possa ser utilizado após o retorno. Por exemplo o algoritmo 1.4 pode ser facilmente convertido para um laço da forma:

enquanto $b \neq 0$ **faça início** $r \leftarrow a \bmod b$; $a \leftarrow b$; $b \leftarrow r$ **fim**;
retorna a;

que é, essencialmente, o algoritmo 1.1.

Por vezes, é possível converter uma função recursiva não de cauda em uma função recursiva de cauda. Para isso, você deve utilizar um ou mais parâmetros auxiliares que são usados para formar o resultado. As operações pendentes são incorporadas a esses parâmetros (e, possivelmente, também aos antigos) de forma que nenhuma chamada recursiva deixe qualquer operação pendente. Para esconder a utilização desses parâmetros, pode-se usar também uma função auxiliar para a primeira chamada.

Vejamos um exemplo. O algoritmo 1.6 para o cálculo da série de Fibonacci pode ser convertido para recursividade de cauda através da utilização de dois parâmetros auxiliares que se encarregam de transportar, de uma chamada para outra, os valores dos dois últimos elementos da série. A conversão resulta no algoritmo 1.9. Repare que a função `Fibonacci(`n`)` nas linhas 5–7 é apenas uma função auxiliar que deixa a chamada mais limpa, e que a verdadeira função recursiva, que realiza o cálculo, é `Fib(`n`,`*ultimo*`,`*penultimo*`)` nas linhas 1–4. Os parâmetros auxiliares são *ultimo* e *penultimo*, que carregam os resultados calculados para a próxima chamada.

Mais um exemplo: vamos converter a função `Potencia(`x`,`n`)` (algoritmo 1.3) para recursividade de cauda. Precisaremos de um parâmetro auxiliar r para nos ajudar a transportar os resultados de uma chamada para a outra, e as operações pendentes serão distribuídas pelos parâmetros r e x. Veja o algoritmo 1.10. Repare que também utilizamos uma função auxiliar `Potencia(`x`,`n`)` (linhas 7–9), e que a verdadeira função recursiva é `Pot(`x`,`n`,`r`)` (linhas 1–6).

Recursividade aninhada

O último tipo de recursividade que será analisado é a *recursividade aninhada*. Ela ocorre quando um procedimento recebe um parâmetro que inclui, ele mesmo, uma chamada recursiva. A recursividade aninhada é mais rara, aparecendo mais freqüentemente na definição de algumas funções matemáticas tais como a função de Ackermann $A(m, n)$ (m e n inteiros não negativos):

$$A(m,n) = \begin{cases} n+1 & \text{para } m = 0 \\ A(m-1,1) & \text{para } n = 0 \text{ e } m > 0 \\ A(m-1, A(m, n-1)) & \text{caso contrário} \end{cases}$$

Algoritmo 1.10: `Potencia(`x`,`n`)` — recursividade de cauda

Entrada: Números $x \in \mathbb{R}$ e $n \in \mathbb{Z}$
Saída: x^n
Implementação: Programas A.8 e B.8

```
1 Pot(x, n, r) início
2 |  se n = 0 então retorna r;
3 |  se n < 0 então retorna Pot(1/x, −n, r);
4 |  se n mod 2 = 0 então retorna Pot(x · x, n/2, r);
5 |  retorna Pot(x, n − 1, x · r);
6 fim
7 Potencia(x, n) início
8 |  retorna Pot(x, n, 1);
9 fim
```

Algoritmo 1.11: `Ackermann(`m`,`n`)`

Entrada: Inteiros não negativos men
Saída: $A(m, n)$
Implementação: Programas A.9 e B.9

```
1 se m = 0 então retorna n + 1;
2 se m > 0 e n = 0 então retorna Ackermann(m − 1,1);
3 retorna Ackermann(m − 1,Ackermann(m,n − 1));
```

O algoritmo 1.11 mostra uma implementação desta função. Note que a linha 3 tem, na verdade, duas chamadas recursivas, sendo que uma delas é passada como parâmetro da outra. Esta função pode gerar uma enorme árvore de recursividade. Por exemplo, o desenvolvimento de $A(1,2)$ é:

$$\begin{aligned} A(1,2) &= A(0,A(1,1)) = \\ &= A(0,A(0,A(1,0))) = \\ &= A(0,A(0,A(0,1))) = \\ &= A(0,A(0,2)) = \\ &= A(0,3) = \\ &= 4 \end{aligned}$$

Uma observação: a função de Ackermann tem um crescimento extremamente rápido, e mesmo valores baixos de m e n podem esgotar, rapidamente, a faixa de valores representáveis dos tipos primitivos das linguagens de programação. Por exemplo, sabe-se que:

$$A(3,n) = 2^{n+3} - 3 \tag{1.5}$$

$$A(4,n) = 2^{2^{\cdot^{\cdot^{\cdot^{2^{16}}}}}} - 3 \tag{1.6}$$

onde, na equação 1.6, tem-se uma pilha com n 2's no expoente, ou seja, $A(4,1) = 2^{2^{16}} - 3$, $A(4,2) = 2^{2^{2^{16}}} - 3$, etc. Estes números são gigantescos. Por exemplo, $A(4,1) = 2^{2^{16}} - 3 = 2^{65536} - 3$, número que tem uma ordem de grandeza de 10^{19718}. Se isto não parece impressionante, saiba que o número estimado de átomos no universo é da ordem de 10^{80}.

Exemplo: algoritmos de busca com retrocesso

Como exemplo final de utilização de recursividade, veremos uma técnica muito útil denominada *busca com retrocesso*. Algumas vezes, é muito difícil encontrar uma solução geral ou analítica para certos problemas, e a única técnica possível de resolução é o método de tentativa e erro. Suponha que temos diversas alternativas de caminhos de pesquisa de solução. Tentamos uma delas e prosseguimos. Se conseguimos encontrar a solução, o problema está resolvido; caso contrário, devemos voltar atrás

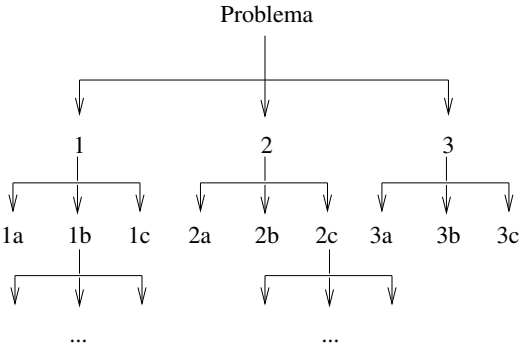

Figura 1.12: Árvore de busca com retrocesso

(retroceder), e tentar uma nova alternativa. Procedemos desta forma até que a solução final seja encontrada ou que todas as alternativas tenham sido tentadas sem sucesso.

Na verdade, a técnica pode ser um pouco mais complexa do que sugere a explicação acima. É possível que a solução do nosso problema possa ser decomposta em sub-tarefas, cada uma delas com o seu próprio conjunto de alternativas de pesquisa de solução. Por exemplo, podemos inicialmente ter as alternativas *1*, *2* e *3*. Começamos com a alternativa *1,* que pode levar a um ponto onde devemos decidir entre as alternativas *1a*, *1b*, *1c*, etc. Selecionamos *1a* e prosseguimos. Esta por sua vez, pode levar a novas alternativas, e assim por diante. Se esgotarmos todas as possibilidades de *1a*, devemos tentar com *1b*, que também pode levar a novas escolhas de caminho. Suponha que nenhuma das alternativas do caminho *1* foi bem sucedida. Devemos então repetir o processo com a alternativa *2*, e prosseguir desta forma até que a solução seja encontrada ou que os caminhos acabem. Graficamente, este processo assume a forma de uma árvore de tentativas, tal como mostrado na figura 1.12, e a técnica de recursividade pode auxiliar no percorrimento dessa árvore.

Vamos exemplificar com um antigo problema conhecido como o *passeio do cavalo* de xadrez. Este quebra-cabeças consiste em encontrar uma forma de cobrir todo o tabuleiro (ou seja, passar pelas sessenta e quatro casas) com sessenta e três movimentos do cavalo (e, portanto, sem passar duas vezes pela mesma casa). No xadrez, o cavalo tem um movimento em forma de "L": duas casas na direção vertical ou horizontal e uma casa na outra direção. Por exemplo, na figura 1.13, o cavalo branco pode atingir qualquer uma das casas ocupadas pelos peões negros. A figura 1.14 mostra uma das possíveis soluções do problema. O cavalo começa no canto superior esquerdo do tabuleiro e o número em cada casa corresponde ao movimento em que esta foi visitada.

Uma forma de resolver esse problema está descrita em Wirth (1989, p. 121): divide-se o problema inicial em sub-tarefas que consistem em, estando o cavalo em uma posição no tabuleiro, encontrar o próximo movimento ou determinar que este não é possível. Por exemplo, posicionamos o cavalo em uma casa inicial no tabuleiro. A partir desta posição, temos um determinado número de alternativas de movimento. Selecionamos uma delas, realizamos e registramos o movimento. Com o cavalo na nova posição, repetimos o processo (o que se refletirá em uma chamada recursiva), e assim sucessivamente. Eventualmente, chegaremos a uma posição onde teremos encontrado a solução (quando então o algoritmo pode ser encerrado) ou não existirão mais alternativas de movimento. Neste último caso, devemos retroceder à posição anterior e tentar uma nova alternativa partindo dela. Ou seja, estamos procedendo por tentativa e erro, percorrendo toda a árvore busca do problema até que encontremos a solução ou não haja mais caminhos a percorrer.

Em um primeiro nível de refinamento, o algoritmo 1.12 formaliza o processo descrito. O laço das linhas 2–13 percorre as alternativas de movimentos até esgotá-las ou realizar um movimento bem sucedido. Na linha 3 selecionamos uma das alternativas e, em seguida (linha 4), verificamos se ela leva a um movimento válido (para uma casa dentro do tabuleiro e não visitada). Se ainda não solucionamos o problema (ainda há movimentos a realizar, linha 6), uma chamada recursiva (linha 7) tenta o próximo movimento. Em caso de falhas, as linhas 9–11 efetuam o retrocesso.

Para detalhar este algoritmo, precisamos escolher as estruturas de dados que serão utilizadas. A escolha mais natural para representação do tabuleiro é uma matriz quadrada de inteiros. Cada

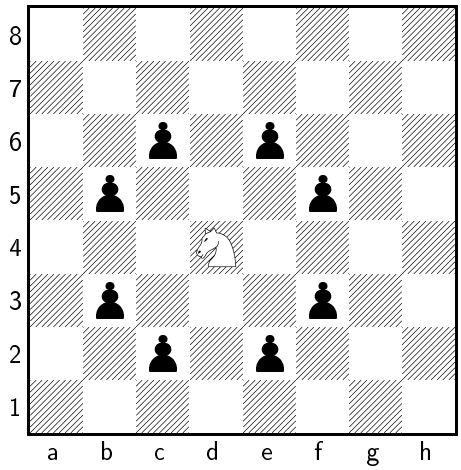

Figura 1.13: Movimentos do cavalo no xadrez

Algoritmo 1.12: `Tenta próximo movimento`

```
1  Inicializa seleção de movimentos;
2  repita
3  |  Seleciona próximo movimento;
4  |  se movimento selecionado é válido então
5  |  |  Registra movimento;
6  |  |  se tabuleiro não foi totalmente preenchido então
7  |  |  |  Tenta próximo movimento;
8  |  |  fim se
9  |  |  se tentativa não foi bem sucedida então
10 |  |  |  Elimina movimento;
11 |  |  fim se
12 |  fim se
13 até (movimento bem sucedido) ou (sem mais movimentos) ;
```

elemento da matriz representará uma casa, com a seguinte interpretação:

- Se o conteúdo do elemento é 0, a casa ainda não foi visitada pelo cavalo.
- Se o conteúdo do elemento é m, a casa foi visitada no m-ésimo movimento.

Assim, a linha 5 (*Registra movimento*) será representada por:

$$T[u, v] \leftarrow m;$$

onde T é a matriz representando o tabuleiro, u e v são as coordenadas da casa em questão e m é o número do movimento que está sendo realizado. Analogamente, a linha 10 (*Elimina movimento*) será representada por:

$$T[u, v] \leftarrow 0;$$

A operação em si será chamada `TentaMovimento` e receberá os seguintes parâmetros:

- Um inteiro m representando o movimento que está sendo analisado;

01	54	39	48	59	44	31	50
38	47	56	53	32	49	60	43
55	02	33	40	45	58	51	30
34	37	46	57	52	25	42	61
03	20	35	24	41	62	29	14
36	23	18	11	26	15	08	63
19	04	21	16	09	06	13	28
22	17	10	05	12	27	64	07

Figura 1.14: Solução do passeio do cavalo de xadrez

- Inteiros x e y representando a posição do cavalo no tabuleiro;
- Matriz de inteiros T representando o tabuleiro;

e a definição da função será:

`TentaMovimento`(m,x,y,T)

Precisaremos ainda de algumas estruturas para nos auxiliar na seleção do próximo movimento a partir de uma determinada posição (x, y). Analisando a figura 1.13, vemos que, se o cavalo está em (x, y) (casa d4), a nova posição será (u, v) com $u = x + dx$, $v = y + dy$ e (dx, dy) valendo:

- $(1, -2)$: casa e6;
- $(2, -1)$: casa f5;
- $(2, 1)$: casa f3;
- $(1, 2)$: casa e2;
- $(-1, 2)$: casa c2;
- $(-2, 1)$: casa b3;
- $(-2, -1)$: casa b5;
- $(-1, -2)$: casa c6.

Assim, construímos dois vetores dx e dy com os seguintes valores:

$$dx = [1, 2, 2, 1, -1, -2, -2, -1]$$
$$dy = [-2, -1, 1, 2, 2, 1, -1, -2]$$

e utilizamos uma variável inteira k para indexar estes valores.

Além disso, utilizaremos também uma variável lógica *rslt* para registrar o resultado de nossas tentativas. Assim teremos:

1. A linha 1, *Inicializa seleção de movimentos*, resume-se a
   rslt ←**falso**; $k \leftarrow 0$;

2. A linha 3, *Seleciona próximo movimento*, é:
   $k \leftarrow k + 1$; $u \leftarrow x + dx[k]$; $v \leftarrow y + dy[k]$;
3. Na linha 4, a verificação de validade do movimento é:
   $1 \leqslant u \leqslant T$.comprimento **e** $1 \leqslant v \leqslant T$.comprimento **e** $T[u, v] = 0$
4. A linha 5, *Registra movimento*, é:
   $rslt \leftarrow$ **verdadeiro**; $T[u, v] \leftarrow m$;
5. A verificação do preenchimento do tabuleiro na linha 6 é:
   $m < (T.\text{comprimento})^2$
6. A linha 7, *Tenta próximo movimento*, é:
   $rslt \leftarrow$ `TentaMovimento(`$m + 1$`,` u`,` v`,` T`)`;
7. A verificação da falha do movimento na linha 9 é:
   não *rslt*
8. O teste para finalização do laço na linha 13 é:
   rslt **ou** $k = dx$.comprimento
9. Ao final, devemos retornar o resultado obtido:
   retorna *rslt*;

O processo completo encontra-se no algoritmo 1.13. O algoritmo 1.14 exibe a inicialização das estruturas utilizadas (linhas 1–5) e uma tentativa de resolução com o cavalo na casa superior esquerda (linha 6), exibindo a solução encontrada ou uma mensagem informativa de falha (linhas 7–11).

Algoritmo 1.13: `TentaMovimento(`m,x,y,T`)`

Entrada: Inteiros m (número do movimento), x e y (posição do cavalo no tabuleiro) e matriz de inteiros T (tabuleiro).
Saída: retorno **falso** indicando falha ou **verdadeiro** indicando sucesso. Neste último caso, T está preenchida com a seqüência de movimentos representando a solução.
Implementação: Programas A.10 e B.10

```
1  global constante dx = [1, 2, 2, 1, −1, −2, −2, −1];
2  global constante dy = [−2, −1, 1, 2, 2, 1, −1, −2];
3  TentaMovimento(m,x,y,T) início
4  |  rslt ← falso; k ← 0;
5  |  repita
6  |  |  k ← k + 1; u ← x + dx[k]; v ← y + dy[k];
7  |  |  se 1 ⩽ u ⩽ T.comprimento e 1 ⩽ v ⩽ T.comprimento e T[u, v] = 0 então
8  |  |  |  rslt ← verdadeiro; T[u, v] ← m;
9  |  |  |  se m < (T.comprimento)² então
10 |  |  |  |  rslt ← TentaMovimento(m + 1,u,v,T);
11 |  |  |  fim se
12 |  |  |  se não rslt então
13 |  |  |  |  T[u, v] ← 0;
14 |  |  |  fim se
15 |  |  fim se
16 |  até rslt ou k = dx.comprimento ;
17 |  retorna rslt;
18 fim
```

Repare que o algoritmo 1.12 exibe um padrão que é comum a todos os algoritmos que utilizam a mesma técnica, e pode ser generalizado tal como mostrado no algoritmo 1.15 (WIRTH, 1989, p. 124). Obviamente, existem variações. Para obter maiores detalhes, você pode consultar Wirth (1989, p. 120–138) e Ziviani (2004, p. 375–378).

Algoritmo 1.14: `Passeio`

Saída: Solução do problema do passeio do cavalo de xadrez.
Implementação: Programas A.11 e B.11

```
1  para i ← 1 até t.comprimento faça
2  |  para j ← 1 até T[i].comprimento faça T[i, j] ← 0
3  fim para
4  x ← 1; y ← 1;
5  T[x, y] ← 1;
6  se TentaMovimento(2,x,y,T) então
7  |  para i ← 1 até T.comprimento faça
8  |  |  para j ← 1 até T[i].comprimento faça imprime T[i, j];
9  |  fim para
10 senão
11 |  imprime "Caminho não encontrado";
12 fim se
```

Algoritmo 1.15: `Tenta`

```
1  Inicializa seleção de alternativas;
2  repita
3  |  Seleciona próxima alternativa;
4  |  se alternativa válida então
5  |  |  Registrar;
6  |  |  se solução está incompleta então
7  |  |  |  Tenta;
8  |  |  fim se
9  |  |  se insucesso então
10 |  |  |  Cancelar registro;
11 |  |  fim se
12 |  fim se
13 até (sucesso) ou (sem mais alternativas) ;
```

1.5 Exercícios

Em todos os exercícios a seguir, sempre que for pedida uma implementação, você deverá desenvolver um algoritmo e implementá-lo em sua linguagem de programação predileta[To do (2)] .

Ex. 1 — Analise o algoritmo 1.1 e responda: que resultado será retornado se b for informado como 0? E se tanto a quanto b forem 0? Estes resultados são coerentes com a definição de máximo divisor comum?

Ex. 2 — Procure informações sobre o *algoritmo de Euclides estendido* e implemente-o. Verifique se o seu algoritmo possui as características mencionadas na página 2 (não se preocupe com a correção).

Ex. 3 — Seja um algoritmo para resolver um problema com uma complexidade $C = f(n)$ dada em μs (microssegundos). Determine o tamanho máximo da entrada que pode ser tratada nos tempos de 1 segundo, 1 minuto, 1 hora, 1 dia, 1 mês e 1 ano, supondo que $f(n)$ pode ser cada uma das funções abaixo:

1. $f(n) = \lg n$
2. $f(n) = n$

3. $f(n) = n \lg n$
4. $f(n) = n^2$
5. $f(n) = n^3$
6. $f(n) = 2^n$
7. $f(n) = n!$

Ex. 4 — Implemente duas funções que calculem a soma e o produto de duas matrizes quadradas. Calcule a complexidade $C(n)$ dessas funções, onde n é a dimensão da matriz. Considere os seguintes custos para as operações individuais:

- A atribuição tem custo k_1.
- A adição tem custo k_2.
- A multiplicação tem custo k_3.
- O controle de um laço **para** tem custo k_4.

Se você utilizar outras operações, estabeleça para elas um custo k_{op} adequado.

Ex. 5 — Observe a definição da função $f : \mathbb{N}^* \mapsto \mathbb{Z}$:

$$f(n) = \begin{cases} 1 & \text{se } n = 1 \\ f(n/2) & \text{se } n \text{ é par} \\ f((3n+1)/2) & \text{se } n > 1 \text{ e } n \text{ é ímpar} \end{cases}$$

Descreva o seu comportamento. Você consegue explicá-lo?

Ex. 6 — Implemente uma função recursiva que inverta os caracteres de uma *string*.

Ex. 7 — Implemente uma função recursiva que verifique se uma *string* é um palíndromo. Um palídromo é uma palavra (ou frase) que não se altera se lida da esquerda para a direita ou da direita para a esquerda.

Ex. 8 — Implemente uma função recursiva que calcule a representação binária de um número inteiro não negativo.

Ex. 9 — Implemente uma função recursiva para retornar o maior elemento de um vetor de inteiros. Faça duas versões dessa função:

1. Na primeira versão, trate apenas o primeiro elemento do vetor, comparando-o com o maior elemento do vetor constituído pelos elementos restantes.

2. Na segunda versão, divida o vetor em duas metades aproximadamente iguais, encontre o maior elemento de cada uma delas e depois compare-os.

Ex. 10 — Implemente uma função recursiva que realize uma pesquisa binária de um valor dentre os elementos de um vetor de inteiros previamente ordenado. Pesquisas binárias estão descritas em Ziviani (2004, p. 156–157); veja também a seção 4.3.

Ex. 11 — Implemente uma função recursiva para gerar todas as $n!$ permutações possíveis dos n elementos de um vetor V de inteiros sem utilizar um vetor auxiliar. Sugestão: considere a tarefa de gerar todas as permutações dos elementos $V[1] \ldots V[n]$ como consistindo de n sub-tarefas que gerem todas as permutações de $V[1] \ldots V[n-1]$, seguidas por $V[n]$, onde, na i-ésima sub-tarefa, os dois elementos $V[i]$ e $V[n]$ tenham sido inicialmente trocados (lembre-se de destrocá-los ao final).

Ex. 12 — Classifique todas as funções recursivas desenvolvidas nos exercícios anteriores como diretas ou indiretas, lineares ou em árvore, de cauda ou não de cauda, aninhadas ou não aninhadas. Converta as que forem não de cauda para uma versão de cauda.

Ex. 13 — Veja a definição da função $M(n)$, definida para n inteiro positivo e denominada "função McCarthy 91":

$$M(n) = \begin{cases} n - 10 & \text{se } n > 100 \\ M(M(n+11)) & \text{se } n \leqslant 100 \end{cases}$$

Analise-a, justifique o seu nome e depois implemente-a.

Ex. 14 — Implemente uma versão não recursiva da função de Ackermann descrita na página 20. Você precisará, provavelmente, fazer uso de pilhas.

Ex. 15 — Implemente uma função recursiva que permita encontrar um caminho através de um labirinto. O labirinto deve ser representado por uma matriz de inteiros M com l linhas e c colunas. A entrada é em $M[1,1]$ e a saída em $M[l,c]$. Um obstáculo na posição (i,j) é representada por $M[i,j] = 9999$, e um caminho livre por $M[i,j] = 0$ (você pode utilizar outros valores para representar quaisquer informações que julgar necessárias). Os movimentos só devem ser permitidos nas direções horizontal e vertical, em qualquer sentido, e o caminho encontrado deve ser exibido no final. Utilize diversos labirintos para testar sua função, inclusive alguns sem saída.

Ex. 16 — O *problema das oito rainhas* consiste em colocar, em um tabuleiro de xadrez, oito rainhas de modo que elas não se ataquem mutuamente. Implemente uma função recursiva que encontre uma solução para este problema. Observações:

1. Uma rainha ataca qualquer peça que esteja na mesma coluna, linha, ou diagonal.
2. Embora a representação mais óbvia para o tabuleiro seja uma matriz quadrada de 8×8 posições, Wirth (1989, p. 125–6) sugere que sejam utilizados vetores para marcar a ocupação das linhas, colunas e diagonais pelas rainhas.
3. Repare que a soma das coordenadas de todas as casas de uma diagonal com inclinação $\diagup$ é constante, e o mesmo acontece com a diferença das coordenadas das casas das diagonais com inclinação $\diagdown$.

Ex. 17 — Implemente uma função recursiva que seja capaz de contar as construções em uma imagem de uma zona rural gerada por satélite. A imagem é representada por um reticulado de células com l linhas e c colunas, tal como mostrado na figura 1.15. Células que não contêm nenhum material construtivo recebem código numérico 0 (não mostrado na figura para não sobrecarregá-la), e células com algum material recebem código 1.

As construções não se sobrepõem, e construções distintas são separadas por uma distância mínima de uma célula. Assim, uma célula com código 1 pertence a uma única construção, e células adjacentes pertencem à mesma construção. Células adjacentes são vizinhas imediatas nas direções vertical,

		1	1	1	1	1	1	1					1	
		1							1					
		1			1	1				1				
		1			1	1			1					
		1						1				1		
	1	1	1	1	1	1	1				1			
1											1	1		
	1													

Figura 1.15: Imagem de satélite

horizontal ou diagonal. Uma construção pode ser circundada por outra; neste caso, as construções devem ser consideradas distintas. Por exemplo, a imagem da figura 1.15 contém quatro construções.

Sua função deve analisar uma imagem com a representação descrita e retornar o número de construções encontradas.

Ex. 18 — Consulte as referências mencionadas na página 18 e construa um interpretador de expressões aritméticas que faça a análise de expressões descritas pela gramática abaixo:

<expressão> ::= <termo> + <expressão> | <termo> − <expressão> | <termo>
<termo> ::= <fator> * <termo> | <fator> / termo | <fator>
<fator> ::= <primário> ^ <fator> | <primário>
<primário> ::= NÚMERO | (<expressão>)

onde NÚMERO representa um número inteiro não negativo, "^" é o operador de exponenciação e os demais operadores (+, − , *, /) têm o significado usual.