Resumindo, como vimos, o conceito de algoritmo envolve quatro itens: dados de entrada e saída, a própria receita do algoritmo e o meio de suporte onde ele executa. No caso de algoritmos que executam em computadores, esses quatro itens são realizados de forma mais específica:

- Dados de entrada: podem ser materializados através do teclado, do *mouse*, ou lidos de um arquivo preparado com antecedência.
- Dados de saída: normalmente são comunicados através de um monitor, de uma impressão, ou podem ser escritos em arquivos de saída que serão examinados posteriormente.
- Meio de suporte: obviamente é todo o *hardware* do computador, onde o programa executa.
- Receita do algoritmo: é o *software*, ou seja, são os programas que executam no computador. Esses programas, resumidamente, são receitas escritas numa linguagem de máquina, que o *hardware* do computador pode interpretar diretamente, executando as instruções nas ordens especificadas. Normalmente, essa linguagem não é a mesma usada por um programador, como veremos. Mais precisamente, parte do *software* que controla o computador e seus periféricos também poderia ser classificado como um meio de suporte para a execução do programa.

Computadores e as quatro propriedades básicas

Vamos reinterpretar as quatro propriedades básicas de um algoritmo, agora dirigidas mais especificamente para algoritmos que executam em computadores.

- A finitude do texto: quer dizer, simplesmente, que os textos dos programas devem ser finitos. Esses textos, via de regra, residem em arquivos armazenados no disco rígido. É claro que devem ser finitos, uma vez que a capacidade dos discos é limitada.
- As instruções devem ser elementares: no caso, devem ser instruções que a máquina onde o programa vai executar consiga interpretar adequadamente. Programar usando essas instruções diretamente seria extremamente tedioso, trabalhoso e propenso a erros. Essas instruções são muito primitivas e podem variar de uma arquitetura de hardware para outra. Assim, o mesmo algoritmo teria que ser reescrito para executar em várias máquinas diferentes.

  A solução é usar uma *linguagem de programação de alto nível*. Essas linguagens dispõem de instruções básicas bem mais poderosas e, além disso, um mesmo programa escrito em determinada linguagem de programação pode vir a executar em várias máquinas diferentes. É claro que devemos traduzir o texto do algoritmo, escrito numa dessas linguagens, para um texto que contenha apenas instruções básicas da máquina, do contrário esta não seria capaz de executar a seqüência de instruções. Ocorre que, hoje, esse processo de tradução já foi automatizado. Ao processo de traduzir o texto em linguagem de alto nível para um texto equivalente em linguagem

de máquina dá-se o nome de *compilação*. Hoje existem programas – i.e. algoritmos, que tomam como entrada um texto em alto nível e produzem como saída um texto equivalente em linguagem de máquina. Tais programas são conhecidos como *compiladores*.

Existem centenas de linguagens de programação hoje em uso. Entre elas: C, C++, C#, Java, Pascal, FORTRAN, COBOL, e muitas outras. Cada uma delas carrega um conjunto de instruções elementares particular, embora, muitas vezes, esses conjuntos sejam bastante assemelhados. Hoje, já há compiladores para traduzir textos escritos nessas linguagens para virtualmente todas as arquiteturas de computadores existentes. Assim, podemos codificar algoritmos em qualquer uma delas e usar os compiladores correspondentes para criar listas de instruções de máquina que executem em várias arquiteturas diferentes.

Um dos objetivos maiores deste texto é apresentar a linguagem C.

- A lista de instruções deve ser metódica: essa propriedade é automaticamente garantida quando escrevemos algoritmos em uma linguagem de programação. Ou seja, a própria seqüência que compõe o texto já indica precisamente a ordem de execução das instruções. Algumas instruções ao longo do texto têm também o papel de influir na ordem de execução das instruções, especificando desvios para instruções adiante, ou atrás, na seqüência. Obviamente, o processo de compilação mantém a ordem de execução especificada originalmente.
- O algoritmo deve sempre terminar quando iniciado com dados de entrada válidos. Essa é uma propriedade crucial. Porém, em geral, é bastante difícil garantir que esteja satisfeita.

  O desafio está no fato de que o texto do programa, por si só, não deixa claro se a propriedade de terminação está ou não sendo obedecida. Se não estiver, então, para algum conjunto de dados de entrada válidos o programa vai executar indefinidamente, sem parar. É o que se conhece como "loops" de programação. Detectar esses casos, e corrigi-los, é uma tarefa que pode se provar delicada e árdua.

  Pode muito bem acontecer que, ao executar sem parada, o programa venha a consumir mais e mais recursos físicos da máquina, como, por exemplo, memória. Nesses casos, quando a capacidade de prover esses recursos se esgota, o programa é forçosamente interrompido e, usualmente, recebemos uma mensagem "ininteligível" ao usuário, tal como "abnormal end". O que ocorre é que todos os programas executam sob controle do sistema operacional da máquina (Windows, Linux, etc.). É o sistema operacional que disponibiliza os recursos necessários para o programa executar. Quando esses se esgotam, o sistema operacional termina o programa forçosamente e envia a mensagem ininteligível. Não há outra saída que reexaminar o texto do programa e procurar pelo erro.

Outro exemplo

Vamos desenvolver um algoritmo para o seguinte problema simples:

dado um número N ≥ 0, calcular 2^N.

Queremos já desenvolver um algoritmo de tal forma que a tarefa de codificá-lo em uma linguagem de programação seja facilitada. Para tal, devemos usar o conceito de "variável", ou "local de armazenagem de dados", discutido anteriormente. Isso porque, ao codificar o algoritmo em uma linguagem tal como C, esses locais serão facilmente mapeados para a memória do computador.

Qual seria uma boa idéia, a partir da qual poderíamos delinear o algoritmo? De novo, esse é um processo criativo, direcionado pela experiência e sagacidade do programador. Passando por cima desse processo, observamos o seguinte fato óbvio:

$$2^N = 1 \times 2 \times 2 \times \ldots \times 2,$$

onde há N parcelas do número 2 à direita do sinal de igual. A primeira parcela é o número 1 para levar em conta que N pode ser zero. Nesse caso, haveriam zero parcelas do multiplicando 2 e a equação se reduziria a:

$$2^0 = 1,$$

que é o resultado desejado.

As equações acima sugerem um método – i.e., um algoritmo -- para calcularmos 2^N. Podemos escrevê-lo como segue, onde P denota uma variável, i.e., distinta de N.

```
1. Inicie P com o valor 1;
2. Repita N vezes:
      a. Tomamos o valor que está em P e o multiplicamos por 2;
      b. O resultado é armazenado no próprio local P;
3. A potência desejada está armazenada em P; páre.
```

Podemos detalhar mais a linha 2, que controla o número de vezes que as linhas 2a e 2b são executadas. Para tal, usaríamos uma outra variável, seja Z. Iniciamos Z com o valor de N e, para cada execução das linhas 2a e 2b, subtraímos uma unidade do valor de Z, armazenando no próprio Z o novo valor. O processo deve parar, obviamente, quando o valor de Z chegar a zero. Ficaria assim:

```
1. Inicie P com o valor 1
2. Copie o valor de N para Z
3. Enquanto (Z > 0) faça
```

```
      a. Novo valor de P é 2x(valor atual de P)
      b. Novo valor de Z é (valor atual de Z)-1
   4. Resposta está em P; pare.
```

Vamos simular a execução do algoritmo, passo a passo, quando o valor inicial de N é 2. Obteríamos a tabela abaixo:

Passo	Linha	N	P	Z	Comentário
1	1	3	1	--	Inicializa P
2	2	3	1	3	Inicializa Z
3	3	3	1	3	Z>0, continua
4	3.a	3	2	3	Atualiza P
5	3.b	3	2	2	Atualiza Z
6	3	3	2	2	Z>0, continua
7	3.a	3	4	3	Atualiza P
8	3.b	3	4	1	Atualiza Z
9	3	3	4	1	Z>0, continua
10	3.a	3	8	1	Atualiza P
11	3.b	3	8	0	Atualiza Z
12	3	3	8	0	Z=0; páre

O valor computado é 8, e está correto, pois 2^3 = 8.

Convença-se de que este algoritmo corretamente computa o valor da potência 2^N, quando N não é negativo. Verifique o que ocorre se o algoritmo for executado com um dado inválido, i.e., quando o valor inicial de N é negativo.

É claro que este não é o único método para se calcular a potência desejada. Você pode investigar outras maneiras de se obter o mesmo resultado. Isso daria margem a algoritmos diferentes. Não há nada de surpreendente no fato de que possam existir muitos algoritmos para realizar uma mesma tarefa.

Adiantando, parte do algoritmo proposto acima, se transcrito na linguagem C, ficaria na forma:

```
. . . .
z = n;
p = 1;
while (z>0) do
{
      p = 2*p;
      z = z-1;
}
```

```
. . . .
```

Observe que o sinal de igual ("=") é usado para se armazenar novos valores nas variáveis. A execução do bloco entre chaves é controlada pelo comando `while (z > 0 )`. E assim por diante. Veremos todos os detalhes desses comandos, e muitos outros, mais adiante.