Capítulo 9: Números

Uau, num capítulo anterior falamos sobre “contagem”, e este capítulo é sobre “números”. Parece que estamos regredindo para o ensino fundamental, ou antes. E de fato, este capítulo vai conter uma repetição de alguns conceitos do ensino fundamental! Mas isto é para que possamos reexaminar os fundamentos e generalizá-los um pouco. Os processos mecânicos que você sempre usou com números — adicionar, subtrair, comparar, verificar se algo se divide uniformemente, trabalhar com o valor relativo — são todos corretos, mas todos eles foram codificados para números decimais. A palavra “decimal”, neste capítulo, não significará “um número com uma vírgula decimal, como 5,62”, mas sim um número expresso na base 10. E o que significa “expresso na base 10”? Significa que os dígitos, da direita para a esquerda, representam um “lugar de unidades”, um “lugar de dezenas”, um “lugar de centenas”, e assim por diante. Foi isto que todos nós aprendemos na escola primária, e talvez você pensasse que é assim que os números “eram”. Mas acontece que 1, 10, 100, 1000, … é apenas uma escolha de valores relativos à posição, e poderíamos igualmente escolher muitas outras coisas, como 1, 2, 4, 8, … ou 1, 16, 256, 4096, … ou mesmo 1, 23, 529, 12, 167, …, desde que esses valores sejam de um certo tipo (potências sucessivas da base).

É o conceito de bases, e especificamente bases diferentes de 10, que nos levará a repensar algumas coisas. A princípio não será natural, mas em breve você descobrirá que há aspectos de como você trabalha com números que são desnecessariamente específicos, e que é libertador tratá-los de uma forma mais geral.

9.1. O que é um “número”?

Antes de fazermos algo com bases, falemos sobre o conceito de **número**, em geral. A pergunta “o que é um número?” soa como a pergunta mais estúpida que eu poderia fazer-lhe. No entanto, prevejo que, a menos que já tenha estudado este material antes, tem um monte de pensamentos emaranhados na sua cabeça sobre o que são “números”, e esses pensamentos emaranhados são de dois tipos. Alguns deles são sobre números em si. Outros são sobre números de base 10. Se você for como a maioria das pessoas, pensa nestes dois conjuntos de conceitos como igualmente “primários”, ao ponto de um número parecer ser um número de base 10. É difícil concebê-lo de qualquer outra forma. É este preconceito que eu quero expor e erradicar para começar. A maioria das pessoas, se eu lhes pedisse para nomear um número, inventaria algo como “treze”. Isto está correto. Mas se eu lhes perguntasse qual era a sua imagem mental do número “treze”, eles formariam imediatamente a seguinte imagem inalterável:

13

Para elas, o número “treze” é intrinsecamente uma entidade com dois caracteres: o dígito 1 seguido do dígito 3. Este é o número. Se lhes dissesse que existem outras formas igualmente válidas de representar o número treze — utilizando mais, menos, ou o mesmo número de dígitos — ficariam muito confusos. No entanto, este é, de fato, o caso. E a única razão pela qual a imagem particular de dois dígitos “13” está tão gravada no nosso cérebro é que desde tenra idade fomos ensinados a pensar em núme-

ros decimais. Percorremos as nossas aulas sobre as quatro operações e sempre fizemos o nosso "vai um" e "empréstimo" na base 10, e no processo construímos uma quantidade incrível de inércia que é difícil de superar. Uma grande parte do seu trabalho neste capítulo será "desaprender" esta dependência dos números decimais, para que possa trabalhar com números noutras bases, particularmente os utilizados na concepção de computadores.

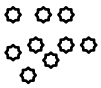
Quando pensar num número, quero que tente apagar a sequência de dígitos da sua mente. Pense num número como o que ele é: uma **quantidade**. Aqui está como o número treze *realmente* se parece:

É apenas uma *quantia*. Há mais objetos nessa figura do que em algumas figuras, e menos do que em outras. Mas de modo algum são "dois dígitos", nem tampouco os dígitos "1" e "3" em particular têm mais ou menos relação com essa quantia do que qualquer outro dígito.

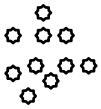
Vamos continuar a pensar sobre isto. Considere este número, que eu vou chamar de "A":

(A) 

Agora vamos adicionar-lhe outro objeto, criando um número diferente a que eu chamarei "B":

(B) 

E, finalmente, fá-lo-emos mais uma vez para obter "C":

(C) 

(Olhe cuidadosamente para essas imagens e convença-se de que adicionei um objeto a cada vez).

Ao passar de A para B, adicionei um objeto. Quando passei de B para C, adicionei também um objeto. Agora pergunto-lhe: ir de B para C é mais "significativo" do que ir de A para B? Aconteceu alguma coisa qualitativamente diferente?

A resposta é obviamente não. Acrescentar um objeto é acrescentar um objeto; não há nada além disso. Mas se tivesse escrito estes números como representações de base 10, como você está habituado a fazer, poderia ter pensado de forma diferente. Você teria ido de:

(A) 8

para

(B) 9

para

(C) 10

Ao passar de B para C, o seu “odômetro” deu uma volta. Teve de passar de um número de um dígito para um número de dois dígitos, simplesmente porque ficou sem espaço em um dígito. Isto pode levar à ilusão de que algo fundamentalmente diferente acontece quando se vai de B para C. *Isto é totalmente ilusório*. Nada diferente acontece ao *número* apenas porque a forma como o anotamos muda.

Os seres humanos têm o curioso hábito de pensar que as alterações do odômetro são significativas. Quando a temperatura atinge os 30 graus, de repente parece “mais quente” do que quando simplesmente subiu de 28 para 29. Quando a bolsa de valores atingiu os 100.000 pontos, quando um influenciador chegou a 100.000 seguidores, ou quando amanheceu o ano 2000, temos a tendência de pensar que algo verdadeiramente importante aconteceu. Mas, como veremos, o ponto em que estes marcos ocorrem é totalmente e até risivelmente arbitrário: tem simplesmente a ver com o número que escolhemos como nossa *base*. E, muito honestamente, poderíamos ter escolhido qualquer número.

9.2. Bases

Como mencionei, uma **base** é simplesmente um número que é uma espécie de âncora para o nosso sistema de valores relativos à posição. Representa quantos símbolos distintos utilizaremos para representar números. Isto estabelece implicitamente o valor da maior quantidade que podemos manter num só dígito, antes de precisarmos passar para dois dígitos.

Na base 10 (decimal), utilizamos dez símbolos: 0, 1, 2, 3, 4, 5, 6, 7, 8, e 9. Consequentemente, o número nove é o valor mais alto que podemos manter num único dígito. Uma vez adicionado outro elemento a um conjunto de nove, não temos outra escolha senão adicionar outro dígito para o expressar. Isto cria uma “casa de dezenas” porque representará o número de conjuntos de dez (que não conseguimos manter no lugar das unidades) que o valor contém.

Agora porque é que o próximo lugar em vez de, digamos, a “casa dos vinte” se chama a “casa das centenas”? Simplesmente porque vinte — assim como qualquer outro número inferior a cem — cabe confortavelmente em dois dígitos. Podemos ter até 9 na casa das unidades, e também até 9 na casa das dezenas, o que nos dá um valor máximo de noventa e nove antes de termos de ceder à utilização de três dígitos. O número cem é exatamente o ponto em que devemos passar para três dígitos; portanto, a sequência de dígitos 1-0-0 representa uma centena.

Se a base escolhida não for óbvia a partir do contexto (como muitas vezes não será neste capítulo), então quando escrevermos uma sequência de dígitos, anexaremos a base como um subscrito ao fim do número. Assim, o número “quatrocentos e trinta e sete” será escrito como 437_{10}.

A forma como interpretamos um número decimal, então, é contando os dígitos mais à direita como um número de indivíduos, o dígito à sua esquerda como o número de grupos de dez indivíduos, o dígito à sua esquerda como o número de grupos de cem

indivíduos, e assim por diante. 5472_{10} é apenas uma forma de escrever $5 \times 1000 + 4 \times 100 + 7 \times 10 + 2 \times 1$.

Se utilizarmos a notação exponencial (lembre-se de que qualquer coisa elevada à 0ª potência é 1), isto é equivalente a:

$$5472_{10} = 5 \times 10^3 + 4 \times 10^2 + 7 \times 10^1 + 2 \times 10^0$$

A propósito, utilizaremos frequentemente o termo **dígito menos significativo** para nos referirmos ao dígito mais à direita (2, no exemplo acima), e **dígito mais significativo** para nos referirmos ao mais à esquerda (5). "Significativo" refere-se simplesmente ao quanto esse dígito "vale" na magnitude global do número. Obviamente 249 é menos que 932, por isso dizemos que a casa das centenas é mais significativa do que os outros dígitos.

Tudo isto parece-lhe provavelmente bastante óbvio. Muito bem, então. Vamos usar uma base que não seja dez e ver como você se sai. Vamos escrever um número na base 7. Temos sete símbolos à nossa disposição: 0, 1, 2, 3, 4, 5, e 6. Espere, você pode perguntar — porque não 7? Porque não há um dígito para sete num sistema de base 7, tal como não há um dígito para dez num sistema de base 10. Dez é o ponto em que precisamos de dois dígitos num sistema decimal, e analogamente, sete é o ponto em que precisamos de dois dígitos no nosso sistema base 7. Como vamos escrever o valor sete? Assim mesmo: **10**. Agora olhe para esses dois dígitos e pratique dizendo "sete" enquanto olha para eles. Toda a sua vida você foi treinado para dizer o número "dez" quando vê os dígitos 1 e 0 impressos dessa forma. Mas esses dois dígitos só representam o número dez *se você estiver utilizando um sistema de base 10*. Se você estiver usando um sistema de base 34, "10" é como se escreve "trinta e quatro".

Muito bem, temos os nossos sete símbolos. Agora, como interpretamos um número como 6153_7? É isto:

$$6153_7 = 6 \times 7^3 + 1 \times 7^2 + 5 \times 7^1 + 3 \times 7^0.$$

Isso não parece tão estranho: é muito paralelo à cadeia decimal que expandimos, acima. Parece mais estranho quando na realidade multiplicamos os valores das casas:

$$6153_7 = 6 \times 343 + 1 \times 49 + 5 \times 7 + 3 \times 1.$$

Assim, na base 7, temos uma "casa de unidades", uma "casa de setes", uma "casa de quarenta e noves", e uma "casa de trezentos e quarenta e três". Isto parece incrivelmente bizarro — como poderia um sistema de números manter-se associado a tais valores relativos à posição? — mas aposto que não pareceria nada engraçado se tivéssemos nascido com 7 dedos. Tenha em mente que na equação acima, escrevemos os valores relativos das casas como números decimais! Se os tivéssemos escrito como números de base 7 (como certamente teríamos se a base 7 fosse o nosso sistema natural de numeração), teríamos escrito:

$$6153_7 = 6 \times 1000_7 + 1 \times 100_7 + 5 \times 10_7 + 3 \times 1_7.$$

Isto é exatamente equivalente em termos numéricos. Porque afinal de contas, 1000_7 *é* 343_{10}. Uma quantidade que parece uma coisa estranha num sistema em uma base parece o número mais redondo possível noutro sistema.

9.3. Hexadecimal (base 16)

Agora, objetivamente falando, verifica-se que dez é também uma base bastante estranha. Eu sei que não parece, mas isso é apenas porque estamos tão habituados a ela. Realmente, se você está repetidamente adicionando pequenos objetos a um desenho, dez é um lugar engraçado para decidir traçar a fronteira e ir para mais dígitos. Dez é somente divisível por 2 e 5 (de todas as coisas), não é um quadrado perfeito, e tudo isto faz dele uma escolha um pouco desapontadora.

Na informática, revela-se muito (muito) conveniente utilizar uma base que é uma potência de dois. Isto significa uma base que é "dois elevado a alguma coisa". Nos primeiros tempos da informática, octal (base 8) era uma escolha comum. Mas por várias razões, isso revela-se menos conveniente do que utilizar a base 16, ou hexadecimal.[1] Sempre que você estiver trabalhando com hardware, sistemas operacionais, controladores de dispositivos, máscaras de bits, ou qualquer outra coisa de baixo nível, encontrará números escritos na base 16 com muita frequência. Portanto, vamos estudar esta base em particular com algum detalhe.

A base 16 vai precisar de dezesseis dígitos, claro. Infelizmente, nós, pessoas com dez dedos, apenas inventamos dez símbolos que são obviamente numéricos: os dígitos de 0 a 9. Então, o que fazemos para os outros seis? Acontece que os autores deste sistema adotaram talvez a abordagem mais óbvia: reutilizar as letras do alfabeto. Assim, adicionamos os "dígitos" de A a F (por vezes escritos em maiúsculas, por vezes em minúsculas) ao nosso conjunto de símbolos. Estas são, portanto, as quantidades que cada dígito representa individualmente:

`0` zero
`1` um
`2` dois
`3` três
`4` quatro
`5` cinco
`6` seis
`7` sete
`8` oito
`9` nove
`A` dez
`B` onze
`C` doze
`D` treze
`E` catorze
`F` quinze

Os inventores da notação hexadecimal não tinham que usar o alfabeto, claro; poderiam ter escolhido uma estrela para dez, um quadrado para onze, uma carinha feliz para doze etc., mas isso não teria sido muito fácil de digitar. Portanto, estamos presos às letras, para o bem ou para o mal. Pratique olhando para essa letra `A` e dizendo a

1 Muitas vezes nos referimos aos números hexadecimais simplesmente como números **hex**, ou **hexa**.

palavra “dez”. Porque é isso que ela significa. Em hexadecimal, a sequência de dígitos `10` não significa “dez”. Significa “dezesseis”.

Estes são os símbolos. Quais são os valores relativos para as casas? Bem, são (a partir da direita) a casa dos 16^0, a casa dos 16^1, a casa dos 16^2, e assim por diante. Escrito de maneira decimal, estas funcionam como o lugar das unidades, o lugar dos 16's, o lugar dos 256's, o lugar dos 4096's, e assim por diante. Mais uma vez, esses números parecem estranhos apenas porque quando escritos em forma decimal, não saem muito “redondos”.

O valor de um número como `72E3` é calculado como:

$$72E3_{16} = 7 \times 4096_{10} + 2 \times 256_{10} + 14 \times 16_{10} + 3 \times 1_{10} = 29{,}411_{10}.$$

Note que tratamos o “E” tal como outro dígito qualquer, o que ele é. Também chamamos ao 72E3 “um número”, o que ele é. Habitue-se à ideia de que os números — números totalmente legítimos — podem ter letras para alguns dos seus dígitos.

Em hexadecimal, qual é o valor mais alto que pode caber num só dígito? Resposta: F (que é quinze.) Qual é o valor mais alto que pode caber em dois dígitos? FF (que é duzentos e cinquenta e cinco.) E que tal três dígitos? FFF (que é sessenta e cinco mil quinhentos e trinta e cinco.) E assim por diante. Se contarmos em hexadecimal, fazemos a mesma coisa que em decimal, só “damos uma volta no odômetro” quando chegamos a F, não quando chegamos a 9.

9.3.1 Convertendo de/para decimal

Então sabemos como pegar num número hexadecimal (como $72E3_{16}$) e encontrar o seu equivalente decimal: simplesmente interpretamos o valor de cada casa como 1, 16, 256, 4096, e assim por diante. E que pensar no contrário? Se tivéssemos um número decimal, como escreveríamos o seu valor em hexadecimal?

Primeiro, vamos aprender duas operações (se ainda não as conhece) que são úteis quando se trabalha com números inteiros. A primeira chama-se o **operador módulo** (escrito “**mod**”), que simplesmente fornece o resto quando se dividem dois números. Este é um conceito que provavelmente aprendeu na escola primária, mas que poderá não ter utilizado desde então. À medida que envelhecemos (e usamos calculadoras), tendemos a pensar numa operação de divisão como 13 ÷ 3 como sendo 4,333… Mas isso é quando queremos uma resposta com valor real (em vez de valor inteiro). Se só quisermos números inteiros, então dizemos que 13 ÷ 3 é “4 com um resto de 1” (o “4” chama-se **quociente**). Isto significa que se tivermos 13 objetos, podemos tirar quatro grupos de 3 deles, e depois ter 1 objeto restante. A forma como se escreve matematicamente esta operação é “13 mod 3”. Neste caso, verifica-se que 13 mod 3 = 1.

Vamos pensar no que o operador mod produz para diferentes valores. Sabemos que 13 mod 3 = 1. E que tal 14 mod 3? Isso é igual a 2, uma vez que podemos (novamente) retirar quatro grupos de 3’s, mas depois sobrariam dois. E que tal 15 mod 3? Isso rende 0, uma vez que 3 distribui 15 uniformemente, não deixando qualquer resto. 16 mod 3 dá-nos novamente 1, tal como o 13 nos deu. Se pensarmos bem, perceberemos que 19 mod 3 também será 1, tal como 22 mod 3 e 25 mod 3. Estes números que dão o mesmo resto quando divididos por 3 são ditos “**congruentes** mod 3”. Os números

2, 5, 8, 11, 14, etc. são também todos congruentes (uns com os outros) mod 3, uma vez que todos eles dão resto 2.

Outra observação é que o valor de n mod k dá sempre um valor entre 0 e $k - 1$. Podemos não saber à primeira vista o que é 407.332.117 mod 3, mas sabemos que não pode ser 12, ou 4, ou mesmo 3, porque se tivéssemos tantos elementos depois de retirar grupos de 3, ainda podíamos retirar outro grupo de 3. O resto só nos dá o que resta depois de retirar grupos, portanto, por definição, não pode haver um grupo inteiro (ou mais) no resto.

A outra operação de que precisamos é simplesmente uma operação de "arredondamento para baixo", tradicionalmente chamada "**piso**" e escrita assim: "⌊ ⌋". O piso de um número inteiro é ele próprio. O piso de um não-inteiro é o inteiro logo abaixo dele. Portanto, ⌊7⌋ = 7 e ⌊4,81⌋ = 4. É tão simples assim[2].

A razão pela qual usamos o operador do piso é simplesmente para obter o número inteiro de vezes que um número aparece noutro. ⌊13 ÷ 3⌋ = 4, por exemplo. Ao utilizar o mod e o piso, obtemos o quociente e o resto de uma divisão, ambos números inteiros. Se os nossos números forem 25 e 7, temos ⌊25 ÷ 7⌋ = 3 e 25 mod 7 = 4. Note que isto é equivalente a dizer que 25 = 3 × 7 + 4. Perguntamos "quantos grupos de 7 estão em 25?" e a resposta é que 25 é igual a 3 grupos de 7, mais 4 extra.

O procedimento geral de conversão de uma base para outra é usar repetidamente o mod e o piso para tirar os dígitos da direita para a esquerda. Eis como se faz:

Expressando um valor numérico em uma base

1. Pegue no número mod a base. Escreva esse dígito.
2. Divida o número pela base e tome o piso.
   (a) Se obtiver zero, acabou.
   (b) Se obtiver um número diferente de zero, então faça deste número diferente de zero o seu novo valor, mova o seu lápis para a esquerda do(s) dígito(s) já anotado(s), e volte ao passo 1.

Como exemplo, voltemos ao número hexadecimal 72E3 como no nosso exemplo acima, que já calculamos e era igual a 29.411 em decimal. Começando com 29.411, então, seguimos o nosso algoritmo:

1. (Passo 1) Primeiro calculamos 29.411 mod 16. Isto nos fornece 3. Muitas calculadoras científicas podem realizar esta operação, assim como linguagens de programação como Java e linguagens de análise de dados como R. Ou, pode-se fazer uma divisão à mão (29.411 ÷ 16) e ver qual é o resto. Ou, pode-se dividir numa calculadora comum e ver se a parte após o ponto decimal é 0, ou $\frac{1}{16}$, ou $\frac{2}{16}$ etc. Ou, poderia você sentar-se calmamente e subtrair 16 após 16 de 29.411 até não haver mais 16 para tirar, e ver qual é a resposta. Em todo o caso, a resposta é 3. Por isso, anotamos 3:

$$3$$

2 No entanto, note "simplesmente eliminar a parte decimal" não funcionará com números negativos, uma vez que, por exemplo, ⌊−5,1⌋ = −6

2. (Passo 2) Dividimos agora 29,411 por 16 e tomamos o piso. Isto produz ⌊29.411 ÷ 16⌋ = 1838. Como isto não é zero, executamos o passo 2b: fazemos de 1838 o nosso novo valor, movemos o nosso lápis para a esquerda do 3, e voltamos ao passo 1.
3. (Passo 1) Agora compute 1838 mod 16. Isto dá-nos o valor 14, que é obviamente um número de base 10. O dígito hexagonal equivalente é E. Assim, escrevemos agora o E à esquerda do 3:

E3

4. (Passo 2) Dividindo 1838 por 16 e tomando o piso dá-nos 114. Uma vez que isto não é novamente zero, executamos o passo 2b: fazemos 114 o nosso novo valor, movemos o nosso lápis para a esquerda do E, e voltamos ao passo 1.
5. (Passo 1) A seguir calculamos 114 mod 16. Isto nos dá 2, por isso escrevemos um 2:

2E3

6. (Passo 2) A computação de ⌊114÷16⌋ produz 7, que mais uma vez não é zero, pelo que 7 se torna o nosso novo valor e voltamos mais uma vez ao passo 2b.
7. (Passo 1) 7 mod 16 é simplesmente 7, por isso escrevemo-lo:

72E3

8. (Passo 2) Finalmente, ⌊7 ÷ 16⌋ é zero, por isso passamos ao passo 2a e já terminamos. A página tem 72E3 escrito em letras grandes, que é a resposta correta.

9.3.2 Adicionando números hex

Suponha que temos dois números hexadecimais, e queremos adicioná-los a fim de obter um resultado hexadecimal. Como é que fazemos? Uma maneira é primeiro converter ambos em decimais, depois adicioná-los como se aprendeu na primeira série, e depois converter a resposta de volta para hexadecimal. Mas podemos permanecer "nativamente hexadecimais", desde que adicionemos cada par de dígitos corretamente.

Vamos experimentar isso. Suponha que queremos calcular esta soma:

$$\begin{array}{r} 48D4_{16} \\ +\ 5925_{16} \\ \hline ?_{16} \end{array}$$

Prosseguimos no caminho da primeira série, da direita para a esquerda. Acrescentando os valores das unidades, obtemos 4 + 5 = 9:

$$\begin{array}{r} 48D4_{16} \\ +\ 5925_{16} \\ \hline 9_{16} \end{array}$$

Muito fácil. Agora adicionamos o dígito seguinte à esquerda (a casa dos dezesseis, atenção, não o lugar das dezenas) e encontramos D + 2. Agora o que é "D + 2"? Na

verdade é fácil: tudo o que você tem que fazer é a mesma coisa que fez quando era criança e teve de acrescentar algo como 4 + 5. Ainda não tinha memorizado a resposta, por isso começou com quatro dedos levantados, e contou "1... 2... 3... 4... 5," levantando a cada vez mais um dedo. Depois, olhou para as suas mãos, e eis então: nove dedos!

Vamos fazer o mesmo aqui: comece com o número “D”, e conte dois lugares adicionais: “E... F.” A resposta é F. Esse é o número que é dois a mais que D. Felizmente para nós, ainda cabe em um dígito. Por isso, agora já temos:

$$\begin{array}{r} 48D4_{16} \\ +\ 5925_{16} \\ \hline F9_{16} \end{array}$$

Até agora, tudo bem. O próximo par de dígitos é 8 + 9. Aqui é onde se quer ter cuidado. É provável que olhe para “8 + 9” e diga imediatamente “17!”. Mas 8 + 9 *não* é 17 em hexadecimal. Para descobrir o que é, começamos com o número 8, e contamos: “9... A... B... C... D... E... F... 10... 11”. A resposta é “11”, que é obviamente como se escreve “dezessete” em hexadecimal. Assim, tal como na escola primária, escrevemos “1” e passamos o 1:

$$\begin{array}{r} {\color{red}1} \\ 48D4_{16} \\ +\ 5925_{16} \\ \hline 1F9_{16} \end{array}$$

Finalmente, o nosso último dígito é 4 + 5, mais o 1 herdado. Começamos com 4 e contamos 5: “5... 6... 7... 8... 9”. Depois acrescentamos o um, e contamos “... A”. A resposta é A, sem sobrar um, e por isso temos a nossa resposta final:

$$\begin{array}{r} {\color{red}1} \\ 48D4_{16} \\ +\ 5925_{16} \\ \hline A1F9_{16} \end{array}$$

9.4. Números Binários (Base 2)

A outra base que utilizamos habitualmente na informática é a base 2, ou binária. Isto porque a unidade básica de informação num computador é chamada de bit, que tem apenas dois valores, convencionalmente chamados de “verdadeiro” e “falso” ou “1” e “0”. Os números (assim como todas as outras coisas nos computadores) são, em última análise, representados como sequências enormes de 1's e 0's, que são, naturalmente, números binários.

As regras para a interpretação do valor relativo em base 2 são as mesmas:

$$\begin{aligned} 110101_2 &= 1 \times 2^5 + 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 \\ &= 1 \times 32 + 1 \times 16 + 0 \times 8 + 1 \times 4 + 0 \times 2 + 1 \times 1 \\ &= 53_{10}. \end{aligned}$$

Assim, em binário, temos a casa de unidades, de dois, quatros, oitos, e assim por diante. Chamamos ao lugar mais à direita o **bit menos significativo (LSB** – *do inglês*

*least significant bit***)** e ao mais à esquerda o **bit mais significativo (MSB** – *do inglês most significant bit***)**.

A contagem a partir do zero é de fato a mesma que em qualquer outra base, embora pareça um pouco estranho em binário porque se "dar voltas" com tanta frequência:

0	zero
1	um
10	dois
11	três
100	quatro
101	cinco
110	seis
111	sete
1000	oito
1001	nove
1010	dez
1011	onze
1100	doze
1101	treze
1110	catorze
1111	quinze
⋮	⋮

9.4.1 Conversão de/para decimal

A conversão de binário para decimal foi exemplificada acima (com $110101_2 = 53_{10}$.) Para ir no sentido contrário, seguimos o algoritmo da página FIXME. Vamos testá-lo para o número decimal 49:

1. (Passo 1) Primeiro calculamos 49 mod 2. Fazer o "mod 2" é fácil: basta ver se o número é par ou ímpar. Neste caso, é ímpar, portanto o resto é 1:

$$1$$

2. (Passo 2) Agora dividimos 49 por 2 e tomamos o piso, o que dá $\lfloor 49 \div 2 \rfloor = 24$. Não é zero, por isso executamos o passo 2b: fazemos 24 o nosso novo valor, movemos o nosso lápis para a esquerda do 1, e voltamos ao passo 1.

3. (Passo 1) Calcule 24 mod 2. Como 24 é par, o resto é zero, que escrevemos à esquerda do 1:

$$01$$

4. (Passo 2) Dividir 24 por 2 e tomar o piso, o que dá $\lfloor 24 \div 2 \rfloor = 12$. Fazemos 12 o nosso novo valor, movemos o nosso lápis para a esquerda do 0, e voltamos ao passo 1.

5. (Passo 1) Calcule 12 mod 2. Como 12 é par, isto é zero, o que nós anotamos:

$$001$$

6. (Passo 2) Dividir 12 por 2 e tomar o piso, o que dá $\lfloor 12 \div 2 \rfloor = 6$. Fazemos 6 o nosso novo valor, movemos o nosso lápis para a esquerda do 0, e voltamos ao passo 1.
7. (Passo 1) Calcule 6 mod 2. Como 6 é par, isto é zero, o que nós anotamos:

$$0001$$

8. (Passo 2) Dividir 6 por 2 e tomar o piso, o que dá $\lfloor 6 \div 2 \rfloor = 3$. Faça 3 o nosso novo valor, mova o nosso lápis para a esquerda do 0, e volte ao passo 1.
9. (Passo 1) Calcule 3 mod 2. Uma vez que 3 é ímpar, este é um, que nós escrevemos:

$$10001$$

10. (Passo 2) Dividir 3 por 2 e tomar o piso, o que dá $\lfloor 3 \div 2 \rfloor = 1$. Isto ainda não é zero, portanto faça 1 o nosso novo valor, mova o nosso lápis para a esquerda do 0, e volte ao passo 1.
11. (Passo 1) Calcule 1 mod 2. Uma vez que 1 é ímpar, este é um, que nós escrevemos:

$$110001$$

12. (Passo 2) Dividir 1 por 2 e tomar o piso, o que dá $\lfloor 1 \div 2 \rfloor = 0$. Terminamos. A resposta final é 110001_2. Verificando o nosso trabalho, vemos que de fato um 32 mais um 16 mais um 1 dá 49, que é aquilo com que começamos.

9.4.2 Conversão de/para hexadecimal

Isso foi bastante enfadonho. Mas a conversão de binário para hexadecimal e vice-versa é muito rápida. Isto porque 16 é exatamente 2^4, e assim um dígito hexadecimal é exatamente igual a quatro dígitos binários. Este não é o caso da base 10, onde um dígito decimal é igual a três dígitos binários… e mais um pequeno extra. Este "não é bem um número redondo de dígitos" é o que torna a conversão de decimal para binário (e também de decimal para hexadecimal, aliás) tão incômoda.

Lidamos mais frequentemente com conjuntos de oito bits de cada vez, que é chamado um **byte**. (Esta é a unidade fundamental de armazenamento em praticamente todos os computadores no planeta). Suponha que eu tivesse o seguinte byte:

$$10000110_2$$

Como um dígito hex é exatamente igual a quatro bits, este byte é exatamente igual a:

$$86_{16}$$

Isto porque o byte pode ser ordenadamente dividido em duas partes: 1000, que corresponde ao dígito hexadecimal 8, e 0110, que corresponde ao dígito hexadecimal 6. Estas duas metades são chamadas de **nibbles** — um byte tem dois nibbles, e cada nibble é um dígito hexadecimal. Num relance, portanto, sem multiplicação ou adição, podemos converter de binário para hexadecimal.

Ir na outra direção é igualmente fácil. Se tivermos:

$$3E_{16}$$

nós simplesmente convertemos cada dígito hexadecimal no nibble correspondente:

$$00111110_2$$

Depois de fazer isto durante algum tempo, chega-se ao ponto em que se pode reconhecer instantaneamente qual o dígito hexadecimal que corresponde a cada nibble. Até lá, no entanto, aqui está uma tabela útil:

Nibble	Dígito hex
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	A
1011	B
1100	C
1101	D
1110	E
1111	F

Caso esteja a perguntar-se, sim, isto vale a pena memorizar.

9.4.3 Adicionando números binários

Adicionar dois números binários é o mesmo que adicionar em decimal, hexadecimal, ou qualquer outra base: basta saber quando "dar a volta no odômetro", o que neste caso é quase instantâneo, uma vez que o valor mais alto que um bit pode conter é 1! Vamos tentar:

$$\begin{array}{r} 111001_2 \\ +\ 011010_2 \\ \hline ?_2 \end{array}$$

Uma criança poderia seguir as regras: quando adicionamos dois zeros, obtemos zero. Acrescentar um a um zero dá um. Adicionar dois uns dá zero, e "vai um" para o dígito significativo seguinte. E adicionar dois 1's mais um 1 recebido dá 1 e ainda vai um. Veja se consegue seguir o fluxo:

$$
\begin{array}{r}
{\scriptstyle 11} \\
111001_2 \\
+\ 011010_2 \\
\hline
1010011_2
\end{array}
$$

9.4.4 Capacidade

Qual o maior valor que um byte pode armazenar? Existem 8 bits, e cada um pode ter independentemente um de dois valores (0 ou 1), portanto pelo Teorema Fundamental da Contagem, existem 2^8 combinações diferentes. Isto dá 256, mas não podemos realmente armazenar o número 256 num byte se estivermos utilizando o padrão de bits 00000000_2 (ou 00_{16}) para representar o zero. O valor mais alto seria 111111111_2 (ou FF_{16}), que é 255_{10}.

Como é que armazenamos um número maior do que isso? Basta usar mais do que um byte, claro. Se utilizássemos dois bytes de memória, e os tratássemos como concatenados um após o outro, isso dar-nos-ia 16 bits, permitindo-nos armazenar até ao número $1111111111111111_2 = FFFF_{16} = 65.535_{10}$. Chamaríamos um destes bytes — aquele que representa o lugar dos 2^0 até o lugar dos 2^7 — o byte menos significativo, e o outro — que contém os lugares 2^8 até 2^{15} — o byte mais significativo. A extensão para mais de dois bytes para acomodar números ainda maiores é feita da maneira óbvia.

9.4.5 Esquemas de representação binária

Já vimos o principal sobre representação binária de números inteiros. Mas há uma coisa que ainda não discutimos, e isto são os números negativos. Sabemos como representar qualquer número positivo (ou zero) com um esquema comum de valor relativo à posição. Mas como é que armazenamos um número como -5?

Existem três esquemas diferentes para tratar números negativos, cada um com os seus pontos fortes e fracos.

Sem sinal (unsigned)

O esquema mais simples é chamado de **sem sinal** (em inglês, **unsigned**), e significa simplesmente que não permitimos números negativos. Para um byte, temos à nossa disposição 256 padrões de bits diferentes, e podemos simplesmente optar por atribuir todos eles para representar números positivos, de modo a obter a maior amplitude possível. Isto faz sentido para, digamos, uma variável de um programa em C++ chamada `alturaEmCm`, que sabemos nunca pode ter sentido ser negativa (ninguém tem uma altura negativa).

A vantagem deste esquema é simplesmente que podemos representar a maior gama possível de números positivos, o que por vezes é o objetivo. Cada um dos esquemas alternativos arranca um pedaço destes padrões de bits disponíveis e dedica-os a representar números negativos, deixando menos sobra para os números positivos. Não há almoço grátis: é preciso decidir como se quer "gastar" os seus padrões de bits disponíveis, dependendo dos valores que é preciso representar.

Sinal-Magnitude

O esquema de **sinal-magnitude** é provavelmente a primeira coisa em que se pensaria para resolver o problema da representação de números negativos. Precisamos armazenar o sinal do número de alguma forma, e um sinal é inerentemente uma coisa com dois valores (positivo ou negativo), então porque não tirar um dos bits e usá-lo para representar o sinal? Os bits restantes podem então ser utilizados da forma habitual para representar a magnitude do número.

A forma mais frequente de fazer isto é pegar o bit mais à esquerda e usá-lo como **bit de sinal**. Este bit agora *não tem outro significado além disso*. Ele não pode "ter dupla função" como a casa dos 128, porque então não haveria maneira de distinguir entre, por exemplo, 129 e −129 (ambos seriam representados por 10000001). Não, o bit de sinal deve ser considerado "dinheiro gasto", e o seu poder de expressão não pode ser reivindicado para representar também parte da magnitude. Por convenção, se o bit de sinal for 0, isto representa um número *positivo*, e um bit de sinal 1 representa um número *negativo* (isso pode parecer contraintuitivo, mas é assim que as coisas são).

Portanto, este número em sinal-magnitude:

00100110

representa o número decimal 38. Isto porque o bit de sinal (em negrito, na extrema esquerda) é 0, o que significa que o número é positivo. A magnitude do número está contida nos outros 7 bits, o que dá 32 + 4 + 2 = 38. Este número, por outro lado:

10100110

representa −38. A magnitude é a mesma, mas o bit de sinal é 1, portanto este padrão agora "significa" um número negativo.

É evidente que reduzimos a nossa gama de números positivos em troca da capacidade de armazenar também os negativos. Temos 7 bits de amplitude em vez de 8, portanto, em vez de 255, o nosso valor mais alto possível é apenas 127. No outro extremo, o valor mais baixo possível é de −127.

Se você tiver visão aguçada, poderá ter notado uma discrepância na contagem. Com a abordagem sinal-magnitude, podemos manter números no intervalo de −127 a 127. Mas espere: isso são apenas 255 valores diferentes, não 256! Porque é que perdemos um valor de poder de expressividade? A resposta é que o esquema de sinal-magnitude tem *duas formas* de representar *zero*. O padrão de bits 00000000 é obviamente zero, mas também o é 10000000 (a que se pode chamar de "zero negativo"). Usar dois padrões diferentes para representar o mesmo valor é um pouco de desperdício, mas a situação é na realidade pior do que isso. Ter de prestar contas de ambos os padrões significa que o hardware computacional que utiliza o esquema de sinal-magnitude é inevitavelmente mais complicado. Para comparar dois bytes para ver se são iguais, seria de se esperar que compararíamos simplesmente cada posição de bit, e se fossem todos iguais, os bytes seriam declarados iguais, caso contrário, não. Infelizmente, isto já não é assim tão simples. Os dois padrões zero devem ser considerados numericamente iguais, portanto a nossa lógica digital tem agora de conter um caso especial. "Para serem iguais, todos os bits têm de ser todos iguais… oh, mas na ver-

dade não se os sete mais à direita forem todos zeros em ambos os bytes. Nesse caso, não importa o que o bit mais à esquerda contém". Que loucura.

Complemento de dois

Esta deficiência no esquema de sinal-magnitude é remediada com o esquema de **complemento de dois**, que é o mais frequentemente utilizado na prática. Ele parecerá estranho no início — certamente não tão intuitivo como os dois primeiros — mas ele conduz a uma característica de importância crítica que analisaremos em breve.

Em primeiro lugar, as regras. Para interpretar um número em complemento de dois, você deve seguir as regras a seguir.

Interpretação de um número representado em complemento de dois

1. Olhe para o bit mais à esquerda (tal como em sinal-magnitude). Se for um 0, você tem um número positivo. Se for um 1, tem um número negativo.
2. Se for um número positivo, os outros 7 bits dão-lhe a magnitude (tal como em sinal-magnitude).
3. Se, no entanto, for um número negativo, então para descobrir a magnitude desse número negativo você deve *inverter todos os bits e adicionar um*. Isto dar-lhe-á um número positivo, que é o valor absoluto do seu número negativo.

Exemplo fácil: considere o byte 00100110. O bit mais à esquerda é um 0, o que significa que é um número positivo, e como descobrimos acima, os 7 bits restantes dão uma magnitude de 38. Portanto, este é o número 38.

Exemplo mais difícil: considere o byte 10100110. O bit mais à esquerda é um 1, o que significa que é negativo. Certo: negativo *quanto*? Como é que encontramos a magnitude? Bem, "viramos" todos os bits (ou seja, invertemos cada um dos 0 para 1 e vice-versa) para obter:

$$01011001$$

e depois adicionamos um ao resultado:

$$\begin{array}{r} 1 \\ 01011001 \\ +\quad\quad 1 \\ \hline 01011010 \end{array}$$

Esta mágica produz o valor 01011010_2, que se converte em 90_{10}. **Isto significa que o número original, 10100110, corresponde ao valor −90.**

"Inverter todos os bits e adicionar um" é a receita de bolo para obter o complemento (negativo) de um número no esquema de complemento de dois. Também funciona em sentido inverso. Comecemos com 90 desta vez e voltemos ao processo, assegurando-nos de que obtemos −90.

Comecemos com a representação binária de 90_{10}:

$$01011010$$

Inverta todos os bits para obter:

10100101

e finalmente adicione um ao resultado:

$$\begin{array}{r} 1 \\ 10100101 \\ +\quad\quad\quad 1 \\ \hline 10100110 \end{array}$$

Obtemos 10100110, que foi precisamente o número com que começamos inicialmente, e que já vimos que representa -90.

Agora você pode se perguntar o que ganhamos com tudo isto. Certamente que este esquema é consideravelmente mais complicado do que a simples ideia de reservar um bit como bit de sinal, e tratar o resto como uma magnitude. Mas acontece que existe de fato uma razão para essa loucura. Pode soar estranho, mas um esquema de representação de complemento de dois permite-nos *realizar adição e subtração com uma única operação*.

Na primeira série (ou similar), você aprendeu o procedimento de adição de números de vários dígitos, que seguimos várias vezes neste capítulo. Implica acrescentar os dígitos da direita para a esquerda e possivelmente "transportar" (o "vai um"). Depois, na segunda série (ou algo assim), aprendeu o procedimento de subtração de números com vários dígitos. Implica subtrair os algarismos da direita para a esquerda e possivelmente "tomar emprestado". Se for como eu, achou mais fácil adicionar do que subtrair. É fácil apenas levar o um, mas para pedir emprestado é necessário olhar o dígito à esquerda, certificando-se de que se pode pedir emprestado (ou seja, que não já é 0), pedindo emprestado mais à esquerda até encontrar de fato um valor não zero disponível, se necessário, esperando que o número na parte de baixo seja de fato menor do que o número na parte de cima (porque de outra forma terá de mudar a ordem e depois adicionar um sinal negativo ao resultado), e mantendo tudo isso em ordem à medida que você avança no processo.

Mesmo que para você não tenha sido mais difícil subtrair do que adicionar, você não pode negar que trata-se de um algoritmo completamente diferente, com regras diferentes a seguir. No hardware de computador, temos de implementar circuitos diferentes para executar cada operação, o que é mais difícil, mais dispendioso, mais sujeito a erros e consome mais energia.

No entanto, o que é maravilhoso no complemento de dois, é que com este esquema nunca precisamos realmente utilizar o algoritmo de subtração. Se quisermos subtrair dois números — digamos, $24 - 37$ — podemos, em vez disso, pegar o complemento do segundo número e depois adicioná-los. Em vez de $24 - 37$, calculamos $24 + (-37)$.

Vamos vê-lo em ação. Utilizando os procedimentos de conversão, podemos calcular que 24_{10} é:

00011000

e o 37_{10} *positivo* é:

00100101

Se quiséssemos computar 24 + 37, simplesmente adicionaríamos estes. Mas em vez disso estamos à procura de 24 − 37, por isso vamos tomar o complemento de 37 para encontrar −37. Inverta todos os bits de 37:

$$11011010$$

e acrescente um:

$$\begin{array}{r} 11011010 \\ +\quad\quad\quad 1 \\ \hline 11011011 \end{array}$$

e agora determinamos que no esquema de complemento de dois, −37 é representado por 11011011_2.

Estamos agora prontos para calcular 24 + (−37):

$$\begin{array}{rl} & 11 \\ & 00011000 \quad \leftarrow \text{isto é } 24_{10} \\ + & 11011011 \quad \leftarrow \text{isto é } -37_{10} \\ \hline & 11110011 \end{array}$$

Assim, temos a nossa resposta em complemento de dois, 11110011. A que valor isso corresponde? Bem, o bit mais à esquerda é um 1, por isso é um número negativo. Para descobrir qual é o número negativo, invertemos todos os bits e adicionamos um:

$$\begin{array}{rl} & 00001100 \quad \leftarrow \text{inverta os bits para obter} \\ + & \quad\quad\quad 1 \quad \leftarrow \text{adicione um} \\ \hline & 00001101 \end{array}$$

Isto é 13 positivo, o que significa que o número que invertemos para obtê-lo — 11110011 — deve representar −13. E esta é de fato a resposta correta, para 24 − 37 = −13.

Uma última palavra sobre o complemento de dois: qual é a faixa de números que podemos representar? Acontece que é de −128 a 127. O valor mais alto é 01111111, que é 127. Pode-se pensar que o valor mais baixo seria representado como 11111111, mas se o calcularmos, descobriremos que este é na realidade o número −1. O menor número é na verdade o padrão de bits 10000000, que é −128.

Overflow

Um último detalhe delicado que precisamos abordar tem a ver com o **overflow** (**estouro**). Quando adicionamos dois números, há a possibilidade de o resultado conter um dígito a mais do que os números originais. Provavelmente já viu isto numa calculadora portátil ao apertar "=" e obter um "E" (para "erro") no visor. Se houver apenas dez dígitos no visor, adicionar dois números de dez dígitos resultará (por vezes) num número de onze dígitos que a sua calculadora não pode exibir, e ela está alertando-o para esse fato para que não interprete mal o resultado. Aqui, podemos adicionar duas quantidades de 8 bits e acabar com uma quantidade de 9 bits que não cabe em um byte. Esta situação chama-se *overflow* (ou *estouro*), e precisamos detectar quando isto ocorre.

As regras para detectar o overflow são diferentes, dependendo do esquema de representação. Para números sem sinal, a regra é simples: se um 1 for transportado a par-

tir do MSB (extremo esquerdo), então temos um overflow. Portanto, se eu tentasse acrescentar 155_{10} e 108_{10}:

$$\begin{array}{rl} & \texttt{1111} \\ & \texttt{10011011} \quad \leftarrow \text{isto é } 155_{10} \\ \underline{+} & \underline{\texttt{01101100} \quad \leftarrow \text{isto é } 108_{10}} \\ & \texttt{100001111} \end{array}$$

eu acabaria tendo um "vai um" à esquerda para o 9º dígito. Uma vez que só conseguimos manter oito dígitos no nosso resultado, obteríamos uma resposta incorreta (15_{10}), que podemos detectar como errada porque o 1 para o 9º dígito indica o estouro.

Em sinal-magnitude funciona da mesma maneira, exceto que tenho um bit a menos quando estou a adicionar e armazenar resultados. (Em vez de um byte de bits representando a magnitude, o bit da extremidade esquerda foi reservado para um propósito especial: indicar o sinal do número. Portanto, se eu adicionar as quantidades restantes de 7 bits e conseguir um "vai um" para o oitavo dígito, isso indicaria um overflow).

Agora com o complemento de dois, as coisas não são (previsivelmente) assim tão fáceis. Mas acontece que são quase tão fáceis. Ainda há uma regra simples para detectar o estouro, é apenas uma regra diferente. A regra é: se o bit que seria transportado depois de efetuar a soma no último bit (mais à esquerda) for *diferente* do bit que foi transportado para o último bit (ao somar a penúltima posição), então temos o overflow.

Vamos tentar adicionar 103_{10} e 95_{10} em complemento de dois, dois números que cabem no nosso intervalo de -128 a 127, mas cuja soma não cabe:

$$\begin{array}{rl} \text{bit transportado recebido} \rightarrow & \texttt{1}\texttt{1111111} \\ & \texttt{01100111} \leftarrow \text{isto é } 103_{10} \\ \underline{+} & \underline{\texttt{01011111} \leftarrow \text{isto é } 95_{10}} \\ \text{bit transportado fornecido} \rightarrow & \texttt{0}\texttt{11000110} \end{array}$$

O bit recebido na extremidade esquerda foi 1, mas o bit a ser transportado (para a nona posição) seria 0, por isso, para o complemento de dois, isto significa que detectamos um overflow. É bom que o tenhamos detectado, visto que 11000110 em complemento de dois representa -57_{10}, o que certamente não é 103 + 95.

Essencialmente, se o bit recebido não for igual ao bit fornecido, isso significa que adicionamos dois números positivos e chegamos a um número negativo, ou que adicionamos dois negativos e obtivemos um positivo. É evidente que este é um resultado errado, e a simples comparação diz-nos isso. Basta ter o cuidado de perceber que a regra para detectar o overflow depende totalmente do esquema de representação particular que estamos utilizando. Um "vai um" final de 1 significa sempre um overflow… no esquema sem sinal. Para o complemento de dois, podemos facilmente obter um "vai um" final de 1 sem qualquer erro, desde que o último bit também tenha recebido um bit 1.

“Tudo é relativo”

Finalmente, depois de sairmos para tomar um ar depois desta quantidade imensa de detalhes, vale a pena salientar que não existe uma forma intrinsecamente “certa” de interpretar um número binário. Se eu lhe mostrar um padrão — digamos, 11000100 — e lhe perguntar que valor representa, você não pode me dizer sem saber como interpretá-lo.

Se eu disser “oh, isso é um número sem sinal”, então você trataria cada bit como um dígito num esquema simples de numeração base 2. Você somaria $2^7 + 2^6 + 2^2$ para obter 196, e depois responderia, “ah, então esse é o número 196_{10}”. E estaria certo.

Mas se eu disser, “oh, isso é um número em sinal-magnitude”, você primeiro olharia para o bit mais à esquerda, veria que é um 1, e perceberia que tem um número negativo. Depois pegaria os sete bits restantes e tratá-los-ia como dígitos num esquema simples de numeração de base 2. Acrescentaria $2^6 + 2^2$ para obter 68, e depois responderia, “ah, então esse é o número -68_{10}”. E você estaria certo.

Mas, mais uma vez, se eu disser, “oh, esse é um número em complemento de dois”, primeiro você olharia para o bit mais à esquerda, veria que é um 1, e perceberia que está lidando com um número negativo. De que valor é que é o negativo? Você inverteria todos os bits e adicionaria um para descobrir. Isto dar-lhe-ia 00111100, que você interpretaria como um número de base 2 e obteria 60_{10}. Depois responderia, “ah, então esse é o número -60_{10}”. E você estaria certo.

O que representa então o padrão 11000100? É 196, −68, ou −60? A resposta é qualquer um dos três, dependendo do esquema de representação que estiver utilizando. Nenhum dos dados em computadores ou sistemas de informação tem um significado intrínseco: tudo tem de ser interpretado de acordo com as regras sintáticas e semânticas que inventamos. Em matemática e informática, tudo pode ser feito para significar qualquer coisa: afinal de contas, somos nós que inventamos as regras.

9.5. Créditos

Todas as seções, foram adaptadas (traduzidas e modificadas) de [1], que está disponível sob a licença Creative Commons Attribution-ShareAlike 4.0 International.

9.6. Referências

1. Davies, Stephen. *A Cool Brisk Walk Through Discrete Mathematics*. Disponível em http://www.allthemath.org/vol-i/

9.7. Licença