Desenvolvendo Programas com Correção

Uma introdução teórica e prática

(versão 0.9.2)

Fabiano S. Oliveira

Sumário

Parte I – Especificação

1. Introdução

> If debugging is the process of removing software bugs, then programming must be the process of putting them in.
>
> *Edsger W. Dijkstra*

Esse texto preliminar é inspirado no livro *The Science of Programming*, por David Gries. Embora haja no presente trabalho alguma pequena originalidade, com o ensaio de notação que divirja do primeiro em alguns pontos, bem como o estudo de novos problemas e a apresentação de outras soluções a problemas apresentados na obra original, é certo que diversos problemas e exercícios do original estão presentes também aqui, por sua eficácia em exemplificar a metodologia de desenvolvimento de programas com correção.

Esse texto foi escrito como material complementar ao estudo do livro original de Gries, no contexto da disciplina de Lógica e de Correção de Programas da Universidade do Estado do Rio de Janeiro (UERJ).

2. Lógica

2.1 Sintaxe

Uma *proposição* é uma afirmação, que pode ser verdadeira ou falsa. A lógica, de forma geral, se preocupa com a formalização da descrição de proposições e da relação entre elas.

Embora a forma de expressar uma proposição possa ser o emprego da linguagem natural, a lógica estabelece uma terminologia, em conjunto com uma sintaxe própria, para a expressão de proposições bem formadas, dada a seguir:

1. **V** e **F** são proposições bem formadas;
2. Um identificador (sequência de um ou mais símbolos alfanuméricos, começando por uma letra, e sem espaços) é uma proposição bem-formada; tal identificador é chamado de *variável lógica*;
3. se a, b são proposições bem formadas, então $(\neg a)$, $(a \wedge b)$, $(a \vee b)$, $(a \Rightarrow b)$, e $(a \Leftrightarrow b)$ são proposições bem-formadas. O operador lógico $\neg$ é chamado de *negação*, o $\wedge$ de *conjunção*, o $\vee$ de *disjunção*, o $\Rightarrow$ de *implicação* (no qual a é chamado de *antecedente* e b de *consequente*) e o $\Leftrightarrow$ de *equivalência*.

■ **Exemplo 2.1** As seguintes expressões são proposições bem formadas: **V**, **F**, x, $(x \vee \mathbf{V})$, $(\mathbf{F} \vee \mathbf{V})$, $((x \Leftrightarrow y) \wedge z)$ e $(((x \vee (a \wedge b)) \Rightarrow y) \Rightarrow (\neg a)) \Leftrightarrow b)$. As seguintes expressões não são proposições bem formadas: 123, $\mathbf{V}x$, $(x \vee)$, $(x \Leftrightarrow\Rightarrow y)$, $(x \Leftrightarrow y) \wedge z)$ e $x + 10$. ■

Uma proposição bem formada será chamada simplesmente de proposição.

Regras de Precedência e Eliminação de Parênteses

O uso demasiado de parênteses em proposições é inconveniente. A exemplo de expressões algébricas, há a possibilidade de deixar parênteses implícitos, pela adoção de regras de precedência de operadores lógicos. Com efeito, utilizaremos a convenção de que:

1. uma sequência de aplicação de um mesmo operador é resolvido da esquerda para direita;

2. a prioridade de aplicação dos operadores $\neg$, $\wedge$, $\vee$, $\Rightarrow$ e $\Leftrightarrow$ deve ocorrer nesta ordem (ou seja, $\neg$ é resolvido primeiramente, enquanto $\Leftrightarrow$ é o último).

■ **Exemplo 2.2** As seguintes proposições são equivalentes. Note que as expressões mais simplificadas mantiveram apenas os parenteses necessários para evitar que as regras de precedência resolvessem os operadores em uma ordem diferente da original.

- $((a \Rightarrow b) \Rightarrow c)$ e $a \Rightarrow b \Rightarrow c$;
- $(a \Rightarrow (b \Rightarrow c))$ e $a \Rightarrow (b \Rightarrow c)$;
- $(((x \wedge (a \vee b)) \Rightarrow y) \Rightarrow ((\neg a) \Leftrightarrow b))$ e $x \wedge (a \vee b) \Rightarrow y \Rightarrow (\neg a \Leftrightarrow b)$.

■

2.2 Semântica

A proposição **V** (resp. **F**) corresponde à afirmação que resulta em verdadeira (resp. falsa).

Uma variável lógica corresponde a um fenômeno que pode ou não ocorrer; caso ocorra, a variável é dita ser avaliada como **V** ou, no caso contrário, avaliada como **F**. Assume-se, a princípio, que os fenômenos associados às variáveis de uma proposição podem ocorrer ou não de forma independente uns dos outros. Sendo assim, se há k variáveis distintas em uma proposição, há 2^k maneiras distintas de avaliar o conjunto destas variáveis. Cada uma destas maneiras é chamada de *interpretação*. Uma proposição formada a partir de um operador lógico é avaliada da seguinte maneira:

1. $\neg a$ é avaliado como **V** se, e somente se, a é avaliado como **F**;
2. $a \wedge b$ é avaliado como **V** se, e somente se, ambos a e b são avaliados como **V**;
3. $a \vee b$ é avaliado como **V** se, e somente se, ao menos um entre a e b é avaliado como **V**;
4. $a \Rightarrow b$ é avaliado como **F** se, e somente se, a seja avaliado como **V** e b como **F**;
5. $a \Leftrightarrow b$ é avaliado como **V** se, e somente se, a e b sejam ambos avaliados como **V** ou ambos como **F**.

Assim, é possível determinar a avaliação de uma proposição composta por diversas proposições, primeiro avaliando-se tais proposições mais elementares para, em seguida, avaliar a proposição completa. Uma interpretação é dita *satisfazer* uma proposição se tal proposição é avaliada como **V** sob a dada interpretação.

Uma forma de conduzir a avaliação de uma proposição é através do uso da tabela-verdade, que descrevemos a seguir. Uma *tabela-verdade* é uma tabela onde cada linha corresponde a uma interpretação e cada coluna corresponde a uma proposição contida na proposição que queremos avaliar. Em cada célula, cuja coluna corresponde à uma proposição P e cada linha a uma interpretação, escrevemos **V** nesta célula se, e somente se, P for avaliado como tal na interpretação associada àquela linha.

■ **Exemplo 2.3** A tabela abaixo corresponde à tabela-verdade da proposição $a \Rightarrow b \Rightarrow b$:

a	b	$a \Rightarrow b$	$a \Rightarrow b \Rightarrow b$
F	**F**	**V**	**F**
F	**V**	**V**	**V**
V	**F**	**F**	**V**
V	**V**	**V**	**V**

■

■ **Exemplo 2.4** A tabela abaixo corresponde à tabela-verdade da proposição $a \Rightarrow b \Leftrightarrow \neg a \vee b$:

a	b	$a \Rightarrow b$	$\neg a$	$\neg a \vee b$	$a \Rightarrow b \Leftrightarrow \neg a \vee b$
F	**F**	**V**	**V**	**V**	**V**
F	**V**	**V**	**V**	**V**	**V**
V	**F**	**F**	**F**	**F**	**V**
V	**V**	**V**	**F**	**V**	**V**

■

■ **Exemplo 2.5** A tabela abaixo corresponde à tabela-verdade da proposição $a \Rightarrow b \Leftrightarrow a \wedge \neg b$:

a	b	$a \Rightarrow b$	$\neg b$	$a \wedge \neg b$	$a \Rightarrow b \Leftrightarrow a \wedge \neg b$
F	**F**	**V**	**V**	**F**	**F**
F	**V**	**V**	**F**	**F**	**F**
V	**F**	**F**	**V**	**V**	**F**
V	**V**	**V**	**F**	**F**	**F**

■

Uma *tautologia* é uma proposição que é avaliada como **V** em qualquer interpretação. Uma *contradição* é uma proposição que é avaliada como **F** em qualquer interpretação. Logo, o Exemplo 2.4 apresenta tautologia, enquanto o Exemplo 2.5 uma contradição.

2.3 Transformação de Proposições

Uma proposição pode ser escrita de maneiras distintas, sendo todas equivalentes entre si. Assim como ocorre nas expressões aritméticas, pode-se formalizar um conjunto de regras de transformações de proposições, que apresentamos a seguir.

- **Leis de Comutatividade**:
  - $a \wedge b = b \wedge a$
  - $a \vee b = b \vee a$
  - $a \Leftrightarrow b = b \Leftrightarrow a$
- **Leis de Associatividade**:
  - $a \wedge (b \wedge c) = (a \wedge b) \wedge c$
  - $a \vee (b \vee c) = (a \vee b) \vee c$
  - $a \Leftrightarrow (b \Leftrightarrow c) = (a \Leftrightarrow b) \Leftrightarrow c$
- **Leis de Distributividade**:
  - $a \wedge (b \vee c) = (a \wedge b) \vee (a \wedge c)$
  - $a \vee (b \wedge c) = (a \vee b) \wedge (a \vee c)$
- **Lei de De Morgan**:
  - $\neg(a \wedge b) = (\neg a) \vee (\neg b)$
  - $\neg(a \vee b) = (\neg a) \wedge (\neg b)$
- **Lei da Dupla Negação**: $\neg(\neg a) = a$
- **Lei do Meio Excluído**: $a \vee \neg a = \mathbf{V}$
- **Lei da Contradição**: $a \wedge \neg a = \mathbf{F}$
- **Lei da Implicação**: $a \Rightarrow b = \neg a \vee b$

- **Lei da Igualdade**: $a \Leftrightarrow b = (a \Rightarrow b) \wedge (b \Rightarrow a)$
- **Lei da Contra-Positiva**: $a \Rightarrow b = \neg b \Rightarrow \neg a$
- **Leis da Simplificação da Conjunção**:
  - $a \wedge \mathbf{F} = \mathbf{F}$
  - $a \wedge \mathbf{V} = a$
  - $a \wedge a = a$
  - $a \wedge (a \vee b) = a$
- **Leis da Simplificação da Disjunção**:
  - $a \vee \mathbf{F} = a$
  - $a \vee \mathbf{V} = \mathbf{V}$
  - $a \vee a = a$
  - $a \vee (a \wedge b) = a$

Estas leis são utilizadas com tal frequência que justifica-se o esforço para memorizá-las.

■ **Exemplo 2.6** A expressão $a \wedge \neg(\neg a \Leftrightarrow \neg b)$ é equivalente a $a \wedge \neg b$, pois $a \wedge \neg(\neg a \Leftrightarrow \neg b)$

$= a \wedge \neg((\neg a \Rightarrow \neg b) \wedge (\neg b \Rightarrow \neg a))$ (Lei da Igualdade)
$= a \wedge \neg(\neg a \Rightarrow \neg b) \vee \neg(\neg b \Rightarrow \neg a)$ (Lei de De Morgan)
$= a \wedge \neg(\neg(\neg a) \vee \neg b) \vee \neg(\neg(\neg b) \vee \neg a)$ (Lei da Implicação)
$= a \wedge (\neg(a \vee \neg b) \vee \neg(b \vee \neg a))$ (Lei da Dupla Negação)
$= a \wedge (\neg a \wedge \neg(\neg b) \vee \neg b \wedge \neg(\neg a))$ (Lei de De Morgan)
$= a \wedge (\neg a \wedge b \vee \neg b \wedge a)$ (Lei da Dupla Negação)
$= a \wedge (\neg a \wedge b) \vee a \wedge (\neg b \wedge a)$ (Lei da Distributividade)
$= a \wedge (\neg a \wedge b) \vee a \wedge (a \wedge \neg b)$ (Lei da Comutatividade)
$= a \wedge \neg a \wedge b \vee a \wedge a \wedge \neg b$ (Lei da Associatividade)
$= \mathbf{F} \wedge b \vee a \wedge \neg b$ (Leis da Contradição e da Simplificação da Conjunção)
$= \mathbf{F} \vee a \wedge \neg b$ (Lei da Simplificação da Conjunção)
$= a \wedge \neg b$ (Lei da Simplificação da Disjunção)

■

Atente para a transformação sistemática de uma expressão em outra, sempre aplicando alguma lei de equivalência sobre uma expressão para produzir a seguinte. Podem haver outras sequências que produzem a mesma transformação, até mesmo sequências mais curtas. Depois de certa prática, pode-se obter com mais facilidade uma sequência que produza a transformação desejada. Como prova real da operação, pode-se verificar, por meio de tabelas-verdade, que a expressão original e a transformada são avaliadas sempre do mesmo modo sob uma mesma interpretação.

Um *argumento* ou *prova* da proposição b sob a premissa de que as proposições $a_1, a_2, \ldots, a_k$ são satisfeitas, o que será denotado por

$$\textbf{de } a_1, a_2, \ldots, a_k \textbf{ inferir } b$$

consiste de uma sequência de transformações de equivalência a partir da proposição

$$a_1 \wedge a_2 \wedge \ldots \wedge a_k \Rightarrow b$$

que resulta em $\mathbf{V}$. Isto só será possível, naturalmente, se a proposição acima for uma tautologia. Uma implicação $a \Rightarrow b$ é dita *válida* se existe um argumento para **de** a **inferir** b, ou seja, a implicação for uma tautologia.

2.4 Predicados

O conceito de proposição será generalizada pela introdução das noções de predicados e de quantificadores. Um *predicado* sobre $x_1 \in X_1$, $x_2 \in X_2$, ..., $x_k \in X_k$ é uma afirmação que envolve as variáveis $x_1, x_2, \ldots, x_k$, representando respectivamente elementos quaisquer de $X_1, X_2, \ldots, X_k$, a qual só deve ser avaliada como verdadeira ou falsa após a determinação de valores concretos para $x_1, x_2, \ldots, x_k$. As variáveis $x_1, x_2, \ldots, x_k$ são chamadas *variáveis livres* do predicado. Um predicado é denotado por um identificador assim como uma variável, acrescentando-se as variáveis livres entre parênteses, como por exemplo, $\mathrm{P}(x_1, x_2, \ldots, x_k)$. Para se descrever a afirmação associada a um predicado comumente emprega-se linguagem natural, proposições lógicas formais, expressões algébricas, expressões de teoria dos conjuntos, ou uma mistura destas maneiras, desde que se entenda ter produzido uma afirmação que deve resultar em verdadeira ou falsa uma vez que uma valoração para as variáveis livres for estabelecida.

■ **Exemplo 2.7** Alguns exemplos de predicados:

- $\mathrm{P}_1(x, y) =$ "x é menor ou igual a y" $= x \leq y$
- $\mathrm{P}_2(x, y, z) =$ "x é o dobro de y ou o triplo de z" $= (x = 2y) \vee (x = 3z)$
- $\mathrm{P}_3(x) =$ "x é menor ou igual a 10" $= P_1(x, 10) = x \leq 10$
- $\mathrm{P}_4(x, S) =$ "x pertence ao conjunto S" $= x \in S$

■

É interessante notar que ao aplicar valores específicos às variáveis livres de um predicado, tal predicado se reduz, na prática, a uma variável lógica. Por exemplo, com relação ao exemplo anterior, $\mathrm{P}_1(2, 3)$ corresponde à variável lógica que é avaliada em **V** se $2 \leq 3$ (que é o caso, neste exemplo). Assim, um predicado pode ser visto informalmente como uma coleção de variáveis lógicas, cada uma associada a uma valoração específica das variáveis livres do predicado. Ou ainda, pode-se ver uma variável lógica como um predicado sem variáveis livres.

A próxima seção apresenta a outra forma de generalização de proposições, que é a introdução de quantificadores.

2.5 Quantificadores

É muito comum a necessidade de afirmar que certo predicado é verdadeiro quando suas variáveis livres são atribuídas a *alguma* valoração específica ou a *qualquer* valoração sobre certo domínio. Para afirmações desta natureza, o conceito de quantificadores é definido. O quantificador *existencial* representa a proposição que é avaliada como **V** se o predicado $\mathrm{P}(x_1, x_2, \ldots, x_k)$ é verdadeiro para algum $x_1 \in X_1, x_2 \in X_2, \ldots, x_k \in X_k$. Denotamos tal afirmação por

$$(\exists\, x_1 \in X_1, x_2 \in X_2, \ldots, x_k \in X_k : \mathrm{P}(x_1, x_2, \ldots, x_k)) \tag{2.1}$$

O quantificador *universal* representa a proposição que é avaliada como **V** se o predicado $\mathrm{P}(x_1, x_2, \ldots, x_k)$ é verdadeiro para cada $x_1 \in X_1, x_2 \in X_2, \ldots, x_k \in X_k$. Denotamos tal afirmação por

$$(\forall\, x_1 \in X_1, x_2 \in X_2, \ldots, x_k \in X_k : \mathrm{P}(x_1, x_2, \ldots, x_k)) \tag{2.2}$$

Para representar os conjuntos $X_1, X_2, \ldots, X_k$, empregamos além da terminologia da teoria dos conjuntos (para nos referenciar ao conjunto de naturais, inteiros, racionais, etc., além das operações sobre conjuntos), as notações seguintes, com i, j sendo números inteiros:

- $[i..j] = \{z \in \mathbb{Z} \mid i \leq z \leq j\} = \{i, i+1, \ldots, j\}$

- $[i..j) = \{z \in \mathbb{Z} \mid i \leq z < j\} = \{i, i+1, \ldots, j-1\}$
- $(i..j] = \{z \in \mathbb{Z} \mid i < z \leq j\} = \{i+1, i+2, \ldots, j\}$
- $(i..j) = \{z \in \mathbb{Z} \mid i < z < j\} = \{i+1, i+2, \ldots, j-1\}$

■ **Exemplo 2.8** São exemplos de proposições com quantificadores:

- $\mathrm{P}(x) = (x \leq 10) \wedge (\exists\, y \in \mathbb{N} : y < x)$
  $(\mathrm{P}(0) = \mathbf{F};\ \mathrm{P}(1..10) = \mathbf{V};\ \mathrm{P}(11..+\infty) = \mathbf{F})$
- $\mathrm{P}(B,N) = (\forall\, i \in [1..N] : B[i] = 0)$
  $(\mathrm{P}([0,0,0,1],3) = \mathbf{V};\ \mathrm{P}([0,0,0,1],4) = \mathbf{F})$
- $\mathrm{P}(B,N) = (\forall\, i \in [1..N) : B[i] \leq B[i+1])$
  $(\mathrm{P}([0,1,6,4],3) = \mathbf{V};\ \mathrm{P}([0,1,6,4],4) = \mathbf{F})$

■

As variáveis $x_1, x_2, \ldots, x_k$ nas proposições (2.1) e (2.2) são chamadas de *variáveis amarradas*, e seu uso é restrito às respectivas proposições, caso os quantificadores estejam inseridos como parte em outras proposições. Isto não significa, contudo, que o identificador fica proibido de ser usado em outras partes da proposição mais geral. O que ocorre é que o uso destas variáveis em outras partes possuem usos distintos em relação àqueles nas expressões do quantificador. Por exemplo, se

$$\mathrm{P}(x,y) = (x \leq y) \vee (\forall\, x \in [1..5] : x \leq y)$$

então $\mathrm{P}(4,2) = \mathbf{F}$ pois tanto $(4 \leq 2)$ quanto $(\forall\, x \in [1..5] : x \leq 2)$ são proposições avaliadas como $\mathbf{F}$, enquanto $\mathrm{P}(7,6) = \mathbf{V}$ pois $(\forall\, x \in [1..5] : x \leq 6) = \mathbf{V}$, o que é suficiente para simplificar a proposição $\mathrm{P}(7,6)$ para $\mathbf{V}$. Note o uso distinto da variável x na avaliação da proposição $(x \leq y)$ e em $(\forall\, x \in [1..5] : x \leq y)$ em $\mathrm{P}(x,y)$. Na primeira, o valor de x vem de x como variável livre, enquanto na segunda vem de x como variável amarrada.

Para a condução de provas envolvendo quantificadores, são necessárias algumas leis adicionais:

- **Leis do Quantificador Existencial**:
  - $(\exists\, x \in \emptyset : \mathrm{P}(x)) = \mathbf{F}$
  - $(\exists\, x \in X : \mathrm{P}(x)) = \mathrm{P}(x_1) \vee \mathrm{P}(x_2) \vee \cdots \vee \mathrm{P}(x_n)$, onde $X = \{x_1, x_2, \ldots, x_n\} \neq \emptyset$
- **Leis do Quantificador Universal**:
  - $(\forall\, x \in \emptyset : \mathrm{P}(x)) = \mathbf{V}$
  - $(\forall\, x \in X : \mathrm{P}(x)) = \mathrm{P}(x_1) \wedge \mathrm{P}(x_2) \wedge \cdots \wedge \mathrm{P}(x_n)$, onde $X = \{x_1, x_2, \ldots, x_n\} \neq \emptyset$

2.6 Definição de Conjuntos

Para a elaboração de predicados, se faz necessário a definição de conjuntos, sobre os quais as variáveis amarradas tomam seus valores. Para tanto, utilizaremos a convenção de que, a partir de certo conjunto previamente definido C, pode-se definir um novo conjunto X pela expressão

$$X = \{f(x) : x \in C \mid \mathrm{P}(x)\}$$

onde $f(x)$ é uma função arbitrária e $\mathrm{P}(x)$ um predicado. Quando $f(x) = x$, podemos simplificar a notação anterior para

$$X = \{x \in C \mid \mathrm{P}(x)\}$$

O conjunto X definido desta forma é tal que um elemento y pertence a X se, e somente se, existe $x \in C$ que satisfaz $P(x)$ e $y = f(x)$. De outra maneira, de uma forma computacional, X seria conceitualmente construído com o seguinte algoritmo:

$X \leftarrow \emptyset$
para cada $x \in C$ **faça**
  se $\mathrm{P}(x)$ **então**
    $X \leftarrow X \cup \{f(x)\}$

O algoritmo acima não é um processo efetivo de construção, uma vez que C não raramente é um conjunto infinito; o algoritmo visa definir alternativamente a notação de conjuntos em termos computacionais como auxílio para aqueles que melhor compreendem tal forma de expressão.

2.7 Traduzindo Linguagem Natural em Proposições Formais

É muito importante que as proposições formais correspondam corretamente a intenção da linguagem natural. Para servir de modelo, apresentamos diversas construções linguísticas comuns, bem como elas podem ser traduzidas em proposições formais.

Proposição em Linguagem Natural	**Proposição em Lógica Formal**
"não ocorre <proposição>"	$\neg$ "<proposição>"
"<proposição 1> e <proposição 2>"	"<proposição 1>" $\wedge$ "<proposição 2>"
"<proposição 1> ou <proposição 2>"	"<proposição 1>" $\vee$ "<proposição 2>"
"ou <proposição 1>, ou <proposição 2>"	"<proposição 1>" $\Leftrightarrow \neg$ "<proposição 2>"
"se <proposição 1>, então <proposição 2>"	"<proposição 1>" $\Rightarrow$ "<proposição 2>"
"quando <proposição 1>, então <proposição 2>"	"<proposição 1>" $\Rightarrow$ "<proposição 2>"
"<proposição 1> somente se <proposição 2>"	"<proposição 1>" $\Rightarrow$ "<proposição 2>"
"<proposição 1> se <proposição 2>"	"<proposição 2>" $\Rightarrow$ "<proposição 1>"
"<proposição 1> se, e somente se, <proposição 2>"	"<proposição 1>" $\Leftrightarrow$ "<proposição 2>"
"há algum x no conjunto X tal que <proposição sobre x>"	$(\exists\, x \in X$: "<proposição sobre x>")
"para todo x no conjunto X, <proposição sobre x>"	$(\forall\, x \in X$: "<proposição sobre x>")
"proposição sobre x"	$\mathrm{P}(x)$, onde $\mathrm{P}(x)$ = "proposição sobre x"

Se a expressão em linguagem natural não se encaixar em nenhuma estrutura acima, tente refraseá-la em termos destas estruturas, mantendo o significado original, para que a tradução lógica adequada se torne aparente.

■ **Exemplo 2.9** As seguintes proposições, antes de serem traduzidas para uma proposição formal com a ajuda da tabela anterior, são reformuladas como etapa intermediária:

- "nenhum x no conjunto X é tal que <proposição sobre x>"
  ="para todo x no conjunto X, não ocorre <proposição sobre x>"
  $=(\forall\, x \in X : \neg \mathrm{P}(x))$, onde $\mathrm{P}(x)$ = "proposição sobre x"

- "os elementos do vetor B entre as posições i e j são iguais a 0"
  ="para todo x no conjunto $[i..j]$, $B[x] = 0$"
  $=(\forall\, x \in [i..j] : B[x] = 0)$

- "não há um número primo que seja o maior de todos"

$=$"não ocorre que haja algum natural x tal que x é primo e o maior de todos"

$=\neg(\exists\, x \in \mathbb{N} :$ "x é primo" $\wedge$ "para todo natural y, se y é primo, então $y \leq x$")

$=\neg(\exists\, x \in \mathbb{N} : \text{Primo}(x) \wedge (\forall\, y \in \mathbb{N} : \text{Primo}(y) \Rightarrow y \leq x))$, onde $\text{Primo}(x) =$ "x é primo"

- "dois números que são múltiplos de certo natural maior que um não são primos entre si"

$=$"para todo x e y naturais, se há algum natural $z > 1$ tais que x, y são múltiplos de z, então x, y não são primos entre si"

$=(\forall\, x, y \in \mathbb{N} : (\exists\, z \in \mathbb{N} : z > 1 \wedge \text{Mult}(x,z) \wedge \text{Mult}(y,z)) \Rightarrow \neg\text{PrimSi}(x,y))$
onde $\text{Mult}(a,b) =$ "a é múltiplo de b" e $\text{PrimSi}(a,b) =$ "a e b são primos entre si"

■

2.8 Exercícios

1. Avalie cada proposição abaixo nas interpretações das variáveis indicadas em s_1 e s_2:

	proposições	s_1	s_2
a	$\neg(m \vee n)$	$m = V, n = V$	$m = V, n = F$
b	$m \vee n \Rightarrow p$	$m = V, n = V, p = F$	$m = F, n = F, p = F$
c	$m \vee (n \Rightarrow p)$	$m = V, n = V, p = F$	$m = F, n = F, p = F$
d	$m \Leftrightarrow (n \wedge p \Leftrightarrow q)$	$m = F, n = F, p = V, q = F$	$m = V, n = F, p = V, q = F$
e	$m \Leftrightarrow (n \wedge (p \Leftrightarrow q))$	$m = F, n = F, p = V, q = F$	$m = V, n = F, p = V, q = F$
f	$(m \Leftrightarrow n) \wedge (p \Leftrightarrow q)$	$m = F, n = F, p = V, q = F$	$m = V, n = F, p = V, q = F$
g	$(m \Rightarrow n) \Rightarrow (p \Rightarrow q)$	$m = F, n = F, p = F, q = F$	$m = V, n = V, p = V, q = V$
h	$((m \Rightarrow n) \Rightarrow p) \Rightarrow q$	$m = F, n = F, p = F, q = F$	$m = V, n = V, p = V, q = V$

2. Responda:
   (a) $A = \{\lfloor x/2 \rfloor : x \in \mathbb{N}\}$. $A = \mathbb{N}$?
   (b) $B = \{2x+1 : x \in A\}$. Determine os 5 menores elementos de B.
   (c) $C = \{2x+1 : x \in A \mid -2x^2 + 100 \geq 0\}$. Determine os 5 menores elementos de C.
   (d) $D = \{(x,y) : x, y \in C \mid x \text{ é par}\}$. Enumere os elementos de D.
   (e) $E = \{(x,y) : x \in C, y \in A \mid 2 \leq y \leq 3\}$. Enumere os elementos de E.
   (f) $F = \{(x,y) : x \in C, y \in E\}$. Determine $|F|$.
3. Escreva a tabela-verdade de cada uma das proposições abaixo:
   (a) $b \wedge (c \vee d)$
   (b) $b \vee (c \wedge d)$
   (c) $\neg b \Rightarrow (c \vee d)$
   (d) $\neg b \Leftrightarrow (b \vee c)$
   (e) $(b \vee c) \wedge (b \Rightarrow c) \wedge (c \Rightarrow b)$
   (f) $(b \Leftrightarrow c) \Leftrightarrow (b \Rightarrow c) \wedge (c \Rightarrow b)$
4. Defina recursivamente "$(\forall\, x \in X : \text{P}(x))$" e "$(\exists\, x \in X : \text{P}(x))$".
5. Prove:
   (a) **de** $(\forall\, x \in X : \text{F}(x))$ **inferir** $(\forall\, x \in X : \text{F}(x) \vee \text{G}(x))$
   (b) **de** $(\forall\, x \in X : \text{F}(x)) \wedge (\exists\, x \in X : \text{G}(x))$ **inferir** $(\exists\, x \in X : \text{F}(x) \wedge \text{G}(x))$
   (c) **de** $(\forall\, x \in \mathbb{N} : \text{F}(x)) \wedge (\text{F}(2) \Rightarrow (\forall\, y \in \mathbb{N} : \text{G}(y)))$ **inferir** $(\exists\, z \in \mathbb{N} : \text{G}(z))$
   (d) **de** $(\forall\, x \in X : \text{F}(x) \Rightarrow \text{G}(x)) \wedge (\forall\, x \in X : \text{G}(x) \Rightarrow \text{H}(x))$ **inferir** $(\forall\, x \in X : \text{F}(x) \Rightarrow \text{H}(x))$

6. O operador de implicação possui as propriedades comutativa e associativa? Justifique sua resposta usando tabelas-verdade.
7. Cada um dos teoremas abaixo podem ser provados usando exatamente uma regra do sistema de dedução natural inferência básica. Nomeie esta regra.
   (a) **de** a, b **inferir** $a \wedge b$
   (b) **de** $a \wedge b \wedge (q \vee r)$ **inferir** $q \vee v$
   (c) **de** $\neg a$ **inferir** $\neg a \vee a$
   (d) **de** $c \Leftrightarrow d$, $d \vee c$ **inferir** $d \Rightarrow c$
   (e) **de** $b \Rightarrow c$, b **inferir** $b \vee \neg b$
   (f) **de** $\neg a$, $\neg b$, c **inferir** $\neg a \vee c$
   (g) **de** $(a \Rightarrow b) \vee b$, a **inferir** $a \Rightarrow b$
   (h) **de** $a \vee b \Rightarrow c$, $c \Rightarrow a \vee b$ **inferir** $a \vee b \Leftrightarrow c$
   (i) **de** $a \wedge b$, $q \vee r$ **inferir** $(a \wedge b) \wedge (q \vee r)$
   (j) **de** $p \Rightarrow (q \Rightarrow r)$, p, $q \vee r$ **inferir** $q \Rightarrow r$
   (k) **de** $c \Rightarrow d$, $d \Rightarrow e$, $d \Rightarrow c$ **inferir** $c \Leftrightarrow d$
   (l) **de** $a \vee b$, $a \vee c$, $(a \vee b) \Rightarrow c$ **inferir** c
   (m) **de** $a \Rightarrow (d \vee c)$, $(d \vee c) \Rightarrow a$ **inferir** $a \Leftrightarrow (d \vee c)$
   (n) **de** $(a \vee b) \Rightarrow c$, $(a \vee d) \Rightarrow c$, $(a \vee b) \vee (a \vee d)$ **inferir** c
   (o) **de** $a \Rightarrow (b \vee c)$, $b \Rightarrow (b \vee c)$, $a \vee b$ **inferir** $b \vee c$
8. Forneça as seguintes provas:
   (a) **de** $p \wedge q$, $p \Rightarrow r$ **inferir** r
   (b) **de** $p \Leftrightarrow q$, q **inferir** p
   (c) **de** $p, q \Rightarrow r$, $p \Rightarrow r$ **inferir** $p \wedge r$
   (d) **de** $b \wedge \neg c$ **inferir** $\neg c$.
   (e) **de** b **inferir** $b \vee \neg c$
   (f) **de** $b \Rightarrow c \wedge d$, b **inferir** d
   (g) **de** $p \wedge q$, $p \Rightarrow r$ **inferir** r
   (h) **de** $p, q \wedge (p \Rightarrow s)$ **inferir** $q \wedge s$
   (i) **de** $p \Leftrightarrow q$ **inferir** $q \Leftrightarrow p$
   (j) **de** $b \Rightarrow (c \wedge d), b$ **inferir** d
9. Prove as seguintes leis usando somente as regras de inferência básicas da dedução natural:
   (a) Distributivas:
      i. **de** $a \wedge (b \vee c)$ **inferir** $(a \wedge b) \vee (a \wedge c)$
      ii. **de** $(a \wedge b) \vee (a \wedge c)$ **inferir** $a \wedge (b \vee c)$
      iii. **de** $a \vee (b \wedge c)$ **inferir** $(a \vee b) \wedge (a \vee c)$
      iv. **de** $(a \vee b) \wedge (a \vee c)$ **inferir** $a \vee (b \wedge c)$
   (b) De Morgan:
      i. **de** $\neg(a \vee b)$ **inferir** $\neg a \wedge \neg b$
      ii. **de** $\neg a \wedge \neg b$ **inferir** $\neg(a \vee b)$
      iii. **de** $\neg(a \wedge b)$ **inferir** $\neg a \vee \neg b$
      iv. **de** $\neg a \vee \neg b$ **inferir** $\neg(a \wedge b)$
   (c) Contra-positiva:
      i. **de** $a \Rightarrow b$ **inferir** $\neg b \Rightarrow \neg a$
      ii. **de** $\neg b \Rightarrow a$ **inferir** $a \Rightarrow b$

3. Especificação

Neste capítulo, concentraremos a atenção nas especificações. A Seção 3.1 discute o papel das especificações na verificação da correção. A Seção 3.2 apresenta as asserções. A Seção 3.3 define formalmente uma especificação e apresenta as convenções empregadas em sua escrita. A Seção 3.4 discute a relação entre predicado e conjunto de estados.

3.1 Introdução

O algoritmo abaixo está correto?

função RESTO(x, y)**:**
  $r \leftarrow x$; $q \leftarrow 0$
  enquanto $r \geq y$ **faça**
    $r \leftarrow r - y$; $q \leftarrow q + 1$
  retorna r

Se disse sim, reavalie. É certo que o algoritmo acima retorna a multiplicação de x por y?

Esta última pergunta talvez tenha causado estranheza, pois não era exatamente multiplicação entre dois números que você tinha em mente quando avaliou a correção da função. Mas note que, além do nome da função, não há nada que indique que esta função deveria retornar o resto da divisão de x por y. A função não é correta se a multiplicação de x por y for o resultado esperado.

Se determinar o resto da divisão de x por y é o objetivo da função, ainda assim ela não estaria correta caso a operação matemática de resto estivesse bem-definida para valores positivos de x e negativos de y (apesar de não usual, é possível e desejado em certas aplicações). Se supormos este tipo de entrada, tal função não é correta pois sua execução é infinita.

O exemplo anterior ilustra o fato que a correção de um algoritmo não pode pode ser avaliada apenas com sua descrição. É necessário também uma especificação sobre o objetivo deste algoritmo, além de uma especificação sobre os valores de entrada sobre os quais ele é suposto operar. Na próxima seção, estabeleceremos uma linguagem para a descrição de ambas as especificações.

3.2 Asserções

Durante uma execução de certo algoritmo A, o *estado de máquina*, ou simplesmente *estado*, consiste da valoração específica que cada variável de A possui em dado ponto desta execução. Um estado está bem definido antes de todos os comandos de um algoritmo, depois de todos eles, ou imediatamente após cada comando efetuado em sua execução e imediatamente antes do próximo. A bem conhecida atividade de "fazer o Chinês" de uma execução, normalmente utilizada para a depuração de um algoritmo, é um termo popular para a tarefa de manter registrado o estado de máquina a cada alteração, potencialmente produzida a cada novo comando efetuado.

Seja $A = S_1; S_2; \ldots; S_N$ um algoritmo, onde $S_1, S_2, \ldots, S_N$ representa a sequência de comandos definida por A, e considere o estado de um algoritmo em determinado ponto de A, digamos, entre os comandos S_i e S_{i+1} (permitimos $i = 0$ para indicar que o ponto de A de interesse é aquele anterior a qualquer comando). Uma *asserção* em tal ponto de A consiste de um predicado $\mathbf{P}(x_1, x_2, \ldots, x_n)$ no qual as variáveis livres $x_1, x_2, \ldots, x_n$ correspondem a variáveis de mesmo nome definidas em A. Tal asserção é denotada por

$$A = S_1; \ldots; S_i \ \{\ \mathbf{P}\ \}\ S_{i+1}; S_{i+2}; \ldots; S_N$$

Note que, por simplicidade, omitiremos a lista das variáveis livres do predicado $\mathbf{P}$, que pode ser obtida observando-se a definição do predicado. Naturalmente, para a expressão da asserção da maneira acima, a expressão de $\mathbf{P}$ pressupõe-se conhecida. Caso contrário, pode-se defini-la em conjunto com a asserção, na forma

$$A = S_1; \ldots; S_i \ \{\ \mathbf{P} = \langle predicado \rangle\ \}\ S_{i+1}; S_{i+2}; \ldots; S_N$$

Ainda, caso o nome do predicado não seja relevante e deseja-se evitar de nomeá-lo, podemos também usar

$$A = S_1; \ldots; S_i \ \{\ \langle predicado \rangle\ \}\ S_{i+1}; S_{i+2}; \ldots; S_N$$

■ **Exemplo 3.1** Alguns exemplos de asserções, descontextualizadas do algoritmo, com uma descrição dos estados que satisfazem tais asserções:

1. $\{\ x \geq 0\ \}$: x não-negativo (outras variáveis podem possuir quaisquer valores);
2. $\{\ (x \geq 0) \wedge (y \leq 10) \wedge (x = 2y)\ \}$: as variáveis x, y devem ser tais que $x \geq 0$, $y \leq 10$ e x é o dobro de y. Se as variáveis são do tipo inteiro, então seus respectivos valores (x, y) é um destes: $(0,0), (1,2), (2,4), (3,6), (4,8), (5,10)$;
3. $\{\ \forall\, i \in [1..N] : 0 \leq a[i] \leq 100\ \}$: cada elemento do vetor a entre as posições 1 e N (inclusive) (isto é, do subvetor $a[1..N]$) é um valor entre 0 e 100;
4. $\{\ \exists\, i \in [1..N] : a[i]$ é primo $\}$: algum elemento do subvetor $a[1..N]$ é primo;
5. $\{\ s = (\sum\, i \in [1..N] : a[i])\ \}$: a variável s é igual à soma dos elementos do subvetor $a[1..N]$.

■

■ **Exemplo 3.2** A seguir, ilustra-se um algoritmo com asserções definidas em certos pontos de sua execução. Note as diferentes maneiras de descrever asserções: a primeira delas define o predicado $\mathbf{P}(y)$; as asserções dentro do comando condicional definem predicados sem nomeá-los; a asserção ao final do algoritmo faz referência a um predicado $\mathbf{Q}$, que deveria estar previamente definido para que aquela asserção estivesse bem formada.

$\{\ \mathbf{P} = y > 0\ \}$
$x \leftarrow y^2$
se $x \geq 5$ **então**
  $\{\ x > 0\ \}$
  $y \leftarrow x$
senão
  $y \leftarrow x - 5$
  $\{\ y \geq 0\ \}$
$\{\ \mathbf{Q}\ \}$

■

Quando uma execução de certo algoritmo A atinge um ponto onde uma asserção está definida, tal asserção é dita *satisfeita* se o estado é tal que os valores das variáveis $x_1, x_2, \ldots, x_n$ satisfazem o predicado $\mathbf{P}(x_1, x_2, \ldots, x_n)$; no caso contrário, tal asserção é dita *violada*.

■ **Exemplo 3.3** Considere o trecho de programa no qual há uma asserção **P**:

$\ldots$
$x \leftarrow 2 * x$
$\{\ \mathbf{P}\ \}$
$x \leftarrow x + 1$
$\ldots$

Se **P** =“x é par”, então a asserção é satisfeita seja qual for a execução. Se **P** =“x é ímpar”, então a asserção é violada em qualquer execução. Finalmente, se **P** =“x não é primo”, então a asserção é violada se $x = 1$ no início deste trecho do algoritmo. Para todos os demais valores de x, ela é satisfeita. ■

O predicado especial que define a asserção localizada antes de qualquer comando de um algoritmo A é chamada de *pré-condição* de A. Analogamente, o predicado que define a asserção localizada após todos os comandos de A é chamada de *pós-condição* de A. No algoritmo do Exemplo 3.2, **P** (resp. **Q**) corresponde à pré-condição (resp. pós-condição) daquele algoritmo. Se considerarmos o trecho correspondente ao bloco “então” do algoritmo (isto é, o comando $y \leftarrow x$), então a pré-condição deste trecho é $x > 0$ e sua pós-condição encontra-se não especificada. Analogamente para o bloco “senão”, a pós-condição para este bloco é $y \geq 0$ enquanto sua pré-condição não está definida.

3.3 Especificação

A *especificação formal*, ou simplesmente *especificação*, de um algoritmo A consiste de sua pré-condição **P** e sua pós-condição **Q**. Uma especificação define implicitamente a seguinte afirmação:

- o algoritmo pressupõe que, ao iniciar a execução de A, o estado inicial satisfaz **P**;
- o algoritmo garante que termina sua execução em tempo finito e, quando o fizer, o estado final satisfaz **Q**.

Retornando à questão levantada na Seção 3.1, com relação à correção da função RESTO, uma maneira apropriada de se apresentar o algoritmo para em seguida se perguntar sobre sua correção seria, por exemplo, conforme se segue.

função RESTO(x, y)**:**
  $\{\ (x \geq 0) \wedge (y > 0)\ \}$
  $r \leftarrow x$; $q \leftarrow 0$
  enquanto $r \geq y$ **faça**
    $r \leftarrow r - y$; $q \leftarrow q + 1$
  $\{$ "r é igual ao resto da divisão de x por y" $\}$
  retorna r

Desta forma, a questão levantada naquela seção sobre o que a função RESTO intencionaria, premissa sobre a qual uma avaliação de correção deve se basear, estaria resolvida: a especificação introduzida junto com o algoritmo nesta seção descreve os estados iniciais aceitos pela função e o que esperar do estado final.

Uma observação importante é que a especificação deve descrever com precisão o *quê* se espera do estado final, e não *como* tal estado será atingido. A princípio, a maneira pela qual se atinge o estado final não é uma preocupação: a princípio, qualquer maneira de conduzir a computação de modo a satisfazer a pós-condição é igualmente válida, do ponto de vista da correção. Critérios como eficiência de tempo e espaço de fato são relevantes nesta escolha, mas são preocupações distintas da correção. Por exemplo, note que se trocássemos a pós-condição para

"r recebe inicialmente o valor x, e então debita-se de r o valor y até que $r < y$"

os estados finais atingidos são exatamente os mesmos; porém, a pós-condição desta vez caracteriza os estados finais desejados pelo próprio algoritmo, e não através da propriedade que deve ser satisfeita ao final. Desta forma, proíbe-se o emprego de algoritmos mais convenientes que este (como por exemplo, mais eficientes) para se produzir os mesmos estados finais.

Convenções nas Especificações

Grau de Formalismo

Como vimos, os predicados, fundamentos da especificação, podem ser escritos textualmente, em linguagem natural, ou formalmente, em notação lógica. Embora a linguagem natural possa inadvertidamente ser ambígua e, por isso, exige cuidado redobrado com sua redação, ela é conveniente no caso em que o formalismo exagerado não ilumina ao leitor o que se pretende capturar com o predicado. Uma notação intermediária entre formalismo detalhado e a linguagem natural, em geral, é a melhor forma de expressão. O Exemplo 3.4 ilustra esta diferença entre graus de formalismo.

■ **Exemplo 3.4** As asserções $\{\ ((m = a) \wedge (a \geq b)) \vee ((m = b) \wedge (b \geq a))\ \}$, $\{\ m = \max\{a, b\}\ \}$ e $\{$ "m é o maior valor entre a e b" $\}$ são equivalentes entre si. A primeira emprega terminologia elementar de álgebra e lógica, e demora a ser compreendida. A terceira forma emprega linguagem natural, que sempre corre o risco de ser ambígua ou imprecisa e exige cuidado extremo. A forma intermediária é precisa e utiliza um operador algébrico muito comum (de "máximo"), e das três, talvez seja a preferível. ■

O exemplo anterior conduz à seguinte convenção:

> **Convenção 3.1** Um predicado adequado a uma especificação deve ser fácil de compreender e tanto formal quanto possível.

Constantes e Valores Iniciais de Variáveis

Considere a especificação de um algoritmo A que precisa trocar de valores duas variáveis, x e y. A dificuldade aqui é que, na pós-condição precisamos nos referenciar ao valor de x e de y não naquele ponto, mas do início da execução. Para isto, faremos uso da contante lógica (conceito equivalente à

variável matemática): uma grandeza arbitrária, definida em outra asserção, cujo valor é invariante onde quer que seja referenciado no escopo do mesmo algoritmo. Veja o exemplo:

$\{ (x = \overline{x}) \wedge (y = \overline{y}) \}$
A
$\{ (x = \overline{y}) \wedge (y = \overline{x}) \}$

onde x, y representam os valores das variáveis x, y no estado onde a asserção está definida, e $\overline{x}, \overline{y}$ são constantes. Pela pré-condição, tais valores devem corresponder àqueles iniciais respectivamente das variáveis x, y. Logo, a pós-condição estabelece que o valor final de x (resp. y) seja aquele inicial de y (resp. x), conforme era de interesse. Seguindo este exemplo, estabelecemos a seguinte convenção:

Convenção 3.2 Variáveis são escritas sem a sobrelinha (por exemplo, Var), enquanto constantes são escritas com a sobrelinha (por exemplo, $\overline{Var}$).

Além disso, é muito comum que constantes lógicas sejam definidas para denotar valores iniciais de variáveis, a exemplo da pré-condição do exemplo anterior. Por conveniência, introduziremos também a seguinte convenção:

Convenção 3.3 Se houver apenas uma pré-condição $\mathbf{P}$ no contexto, e se $x_1, x_2, \ldots, x_k$ são as variáveis do algoritmo, a efetiva pré-condição $\mathbf{P}'$ a ser considerada é

$$\mathbf{P}' = \mathbf{P} \wedge (x_1 = \overline{x_1}) \wedge (x_2 = \overline{x_2}) \wedge \cdots \wedge (x_k = \overline{x_k})$$

Desta forma, o exemplo anterior poderia ser simplificado para

$\{ \mathbf{V} \}$
A
$\{ (x = \overline{y}) \wedge (y = \overline{x}) \}$

Variáveis Sem Modificação

Considere o seguinte algoritmo:

A
$\{ \mathbf{P} \}$

Se a pós-condição $\mathbf{P}$ for $z = a + b$, indica que, logo após a execução de A, a variável z deve ser igual a soma das variáveis a e b. Assim, se

$$A = z, a, b \leftarrow 3, 1, 2$$

$\mathbf{P}$ é satisfeito. No entanto, alguma experiência em resolução de problemas algorítmicos é suficiente para indicar que, embora o algoritmo consiga satisfazer a pós-condição, este não é o algoritmo que se estava procurando. Suponha, então, que há necessidade que z deva receber o valor das variáveis a, b originais. Neste caso, $\mathbf{P}$ é satisfeita por

$$A = z, a, b \leftarrow a + b, 0, 0$$

Novamente, a pós-condição é satisfeita, mas ainda o resultado não é satisfatório. Nesta última solução, o problema é que as variáveis a, b estão sendo anuladas, o que apesar de não violar a pós-condição, talvez contradiga a possível intuição do leitor. Mais provavelmente, espera-se que a, b mantenha-se com seus valores originais inalterados. Finalmente, podemos então revisar o algoritmo para

$$A = z \leftarrow a + b$$

e a questão natural é: como revisar a pós-condição para desqualificar as duas primeiras versões de A, aceitando a terceira? Isto pode ser conseguido através das constantes lógicas que referenciam os valores iniciais, desde modo:

$$P = (z = a + b) \wedge (a = \overline{a}) \wedge (b = \overline{b})$$

Por conveniência, introduziremos também a seguinte convenção:

Convenção 3.4 Se houver apenas uma pós-condição $\mathbf{P}$ no contexto e $\underline{y_1}, \underline{y_2}, \ldots, \underline{y_r}, x_1, x_2, \ldots, x_k$ são as variáveis livres de $\mathbf{P}$, a efetiva pós-condição $\mathbf{P}'$ a ser considerada é

$$\mathbf{P}' = \mathbf{P} \wedge (x_1 = \overline{x_1}) \wedge (x_2 = \overline{x_2}) \wedge \cdots \wedge (x_k = \overline{x_k})$$

removendo-se todos os sublinhamentos das variáveis de $\mathbf{P}$. Ou seja, apenas as variáveis $y_1, y_2, \ldots, y_r$ podem ter seu valor original alterado para que $\mathbf{P}$ seja atendida.

Desta forma, o predicado $\mathbf{P}$ que aceita apenas a terceira versão do algoritmo anterior pode, segundo a última convenção, ser simplificado para

$$P = (\underline{z} = a + b)$$

Generalização de Quantificadores

Note que o Exemplo 3.1(5), informalmente, generalizou o conceito dos quantificadores universal e existencial. Formalmente, estabelecemos agora a convenção para tal:

Convenção 3.5 A expressão

$$v_1 \oplus v_2 \oplus \cdots \oplus v_k$$

pode ser escrita, de forma mais precisa, por

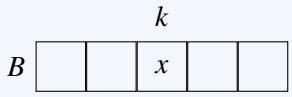
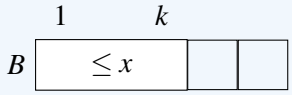
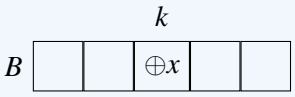
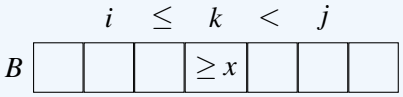
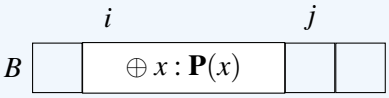
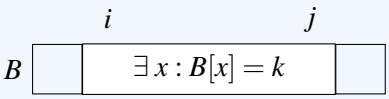
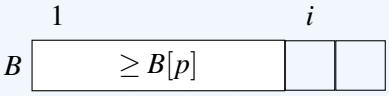
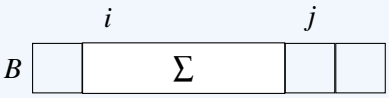
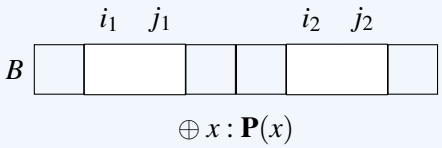
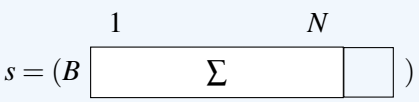
$$(\oplus\, i \in [1..k] : v_i)$$

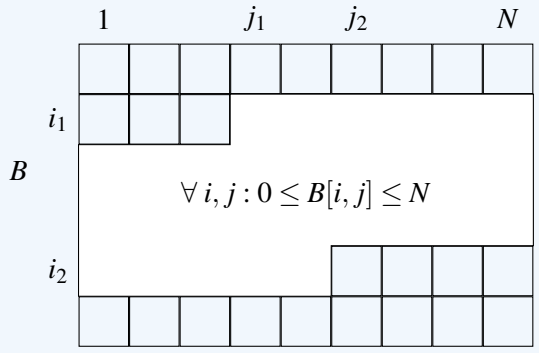
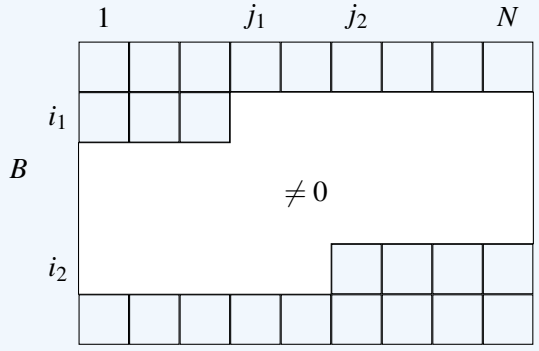
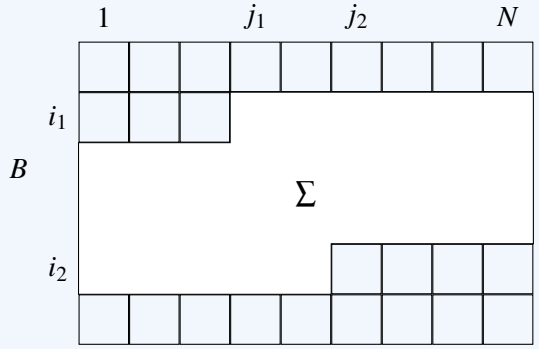
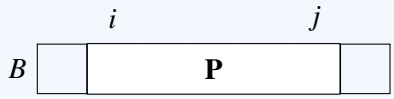
Desde modo, a expressão $1+2+3+4+5+6+7+8+9$ pode ser escrita como $(+\, i \in [1..9] : i)$ ou, dado que já existe um símbolo bem conhecido para o quantificador associado à operação de soma, pode-se escrever na forma do Exemplo 3.1(5), que resulta em $(\sum\, i \in [1..9] : i)$.

Vetores e Matrizes

Proposições sobre vetores e matrizes, não raramente, são melhor comunicadas se um desenho do vetor ou matriz for desenhado, em conjunto com algumas anotações. Embora seja uma tentativa natural de se fazer asserções sobre tais estruturas, lembramos do cuidado que se há de ter para que o significado esteja bem definido. Desta forma, esta seção tem por objetivo estabelecer convenções para tais desenhos, de modo que eles se tornem partes de proposições formais.

Convenção 3.6 A seguinte tabela apresenta algumas proposições formais e a forma equivalente gráfica.

Proposição Formal	Proposição Gráfica Equivalente
$B[k] = x$	
$\forall\, i \in [1..k] : B[i] \leq x$	
$B[k] \oplus x$	
$(i \leq k < j) \wedge (B[k] \geq x)$	
$(\oplus\, x \in [i..j) : \mathbf{P}(x))$	
$(\exists\, x \in [i..j] : B[x] = k)$	
$(\forall\, x \in [1..i) : B[x] \geq B[p])$	
$(\oplus\, x \in [i..j) : B[x])$	
$(\sum\, x \in [i..j) : B[x])$	
$(\oplus\, x \in [i_1..j_2] \cup [i_2..j_2] : \mathbf{P}(x))$	
$s = (\sum x \in [1..N] : B[x])$	

Proposição Formal	Proposição Gráfica Equivalente
$(\forall\, i \in [i_1..i_2], j \in [1..N] :$ $(i = i_1) \wedge (j \geq j_1) \vee (i_1 < i < i_2) \vee$ $(i = i_2) \wedge (j < j_2) \Rightarrow 0 \leq B[i,j] \leq N)$	
$(\forall\, i \in [i_1..i_2], j \in [1..N] :$ $(i = i_1) \wedge (j \geq j_1) \vee (i_1 < i < i_2) \vee$ $(i = i_2) \wedge (j < j_2) \Rightarrow B[i,j] \neq 0)$	
$(\sum (i,j) \in \{(i,j) : i \in [i_1..i_2], j \in [1..N] \mid$ $(i = i_1) \wedge (j \geq j_1) \vee (i_1 < i < i_2) \vee$ $(i = i_2) \wedge (j < j_2)\} :$ $B[i,j])$	
$\mathbf{P}(B,i,j)$	
Ordenado(B,i,j)	

3.4 Predicado vs. Conjunto de Estados

A um predicado **P** corresponde o conjunto de estados que tornam o predicado **P** satisfeito. Raciocinar sobre a asserção em termos deste conjunto associado mostra-se de utilidade, principalmente para aqueles que dominam a teoria de conjuntos. Por exemplo, para quaisquer dois predicados **P**, **Q**, temos que $\mathbf{P} \Rightarrow \mathbf{Q}$ se, e somente se, o conjunto de estados associado a **P** é um subconjunto daquele associado a **Q**. Daqui em diante, ora um predicado será visto como uma proposição lógica, ora como o conjunto de todos os estados que tornam este predicado satisfeito, conforme a necessidade.

3.5 Exercícios

1. Reescreva as seguintes asserções utilizando proposições gráficas equivalentes e forneça um estado que a asserção é satisfeita:
   (a) $\{\ (1 \le p \le q \le n) \wedge (\forall\, i \in [1..p] : b[i] < x) \wedge (\forall\, i \in (q..n] : x \le b[i])\ \}$
   (b) $\left\{ \begin{array}{l} (1 \le k-1 \le f \le h-1 < n) \wedge (\forall\, i \in [1..k) : b[i] = 1) \wedge \\ (\forall\, i \in [k..f) : b[i] = 2) \wedge (\forall\, i \in (h..n] : b[i] = 3) \end{array} \right\}$
2. Reescreva as seguintes asserções utilizando proposições equivalentes não gráficas e forneça um estado que a asserção é satisfeita:
   (a) $\left\{ \begin{array}{c} \begin{array}{cccccccc} & 1 & & i & \le & j & \le & n \end{array} \\ \begin{array}{cc} b & \end{array} \boxed{\le x} \boxed{x} \boxed{} \boxed{} \boxed{x} \boxed{\ge x} \wedge (0 \le i) \end{array} \right\}$
   (b) $\left\{ \begin{array}{c} \begin{array}{cccccccc} & 1 & < & k & < & z & & n \end{array} \\ \begin{array}{cc} a & \end{array} \boxed{\textbf{Ordenado}} \boxed{} \boxed{} \boxed{\ge x} \wedge (z \le n+1) \end{array} \right\}$
3. Para cada descrição de problema abaixo, elabore uma especificação que a traduza da melhor maneira possível, elaborando pré- e pós-condições para tais especificações.
   (a) Atribua a x o maior valor dentre os n primeiros elementos do vetor b
   (b) Atribua a p a posição de um maior valor dentre os n primeiros elementos do vetor b
   (c) Atribua a x o valor absoluto de y
   (d) Atribua a z o valor x/y
   (e) Atribua a z uma raiz de $f(x) = ax^2 + bx + c$
   (f) Atribua a p a posição do último maior valor dentre os n primeiros elementos do vetor b
   (g) Atribua a $p1$ e a $p2$ as posições do primeiro e do último maior valor dentre os n primeiros elementos do vetor b, respectivamente
   (h) Atribua a r o valor lógico correspondente a afirmação de que dado número n é primo
   (i) Atribua a r o n-ésimo número de Fibonacci, onde n é dado. O primeiro número de Fibonacci é 1, o segundo também é 1 e, a partir do terceiro, o número de Fibonacci é obtido pela soma dos dois anteriores. Os primeiros números de Fibonacci são, portanto, $1, 1, 2, 3, 5, 8, \ldots$.
   (j) Atribua a r o valor da afirmação de que um dado vetor a com n elementos possui a primeira metade de elementos ordenada ascendentemente e a segunda metade ordenada descendentemente.
   (k) Dados vetores a, b, atribua a z um elemento que esteja nas primeiras n posições de a e entre as primeiras m posições de b.
   (l) Dada uma matriz M, de n linhas por s colunas, onde a célula $M[i, j]$ representa a nota do aluno i no semestre j, atribua a r o número da linha com o aluno com a maior média de notas.
4. Considere um inteiro de n algarismos, cada um em uma posição do vetor $a[1..n]$, onde $a[1]$ é o algarismo de maior ordem. Como exemplo, se o número 123542 estivesse sendo representado, então $n = 6$ e $a = [1, 2, 3, 5, 4, 2]$. A próxima permutação maior é a representação do menor inteiro $n' > n$ tal que n' é uma permutação dos algarismos de n. No exemplo anterior, a próxima permutação maior seria $[1, 2, 4, 2, 3, 5]$. Escreva formalmente a especificação para a próxima permutação maior de um dado número de entrada.

Parte II – Verificação da Correção

4. Verificação de Correção

Neste capítulo, discutimos os fundamentos da verificação formal de algoritmos. Na Seção 4.1, define-se triplas de Hoare. Na Seção 4.2, define-se o conceito crucial de pré-condição mais fraca. Na Seção 4.3, apresenta-se o emprego da pré-condição mais fraca na verificação formal de algoritmos.

4.1 Tripla de Hoare

Uma *tripla de Hoare* (tH) consiste de um algoritmo A e dois predicados $\mathbf{P}$ e $\mathbf{Q}$, respectivamente pré-condição e pós-condição de A. Em outras palavras, uma tH é escrita como

$$\{\ \mathbf{P}\ \}\ A\ \{\ \mathbf{Q}\ \}$$

Tal tH é dita *válida* sob *correção parcial* se, a partir de qualquer estado satisfazendo $\mathbf{P}$, o algoritmo A executa e, se terminar sua execução, o faz em um estado que satisfaz $\mathbf{Q}$. Atente para o condicional sobre o término de A: sob correção parcial, não é necessário que A termine para que a tH seja válida. Se, adicionalmente, é requerido que A termine em tempo finito em qualquer execução a partir de um estado satisfazendo $\mathbf{P}$, então dizemos que a tripla é *válida* sob *correção total*. Quando uma tH for dita válida, sem menção explícita a um tipo de correção, pressupõe-se que é sob correção total.

■ **Exemplo 4.1** As seguintes tHs são válidas:

$$\begin{aligned}
\{\ (x=3)\wedge(y=2)\ \}\ z \leftarrow x/y\ \{\ z\geq 1\ \}\\
\{\ (x\geq 2)\wedge(y=2)\ \}\ z \leftarrow x/y\ \{\ z\geq 1\ \}\\
\{\ (x\geq y)\wedge(y>0)\ \}\ z \leftarrow x/y\ \{\ z\geq 1\ \}\\
\{\ (x/y\geq 1)\wedge(y\neq 0)\ \}\ z \leftarrow x/y\ \{\ z\geq 1\ \}
\end{aligned}$$

■

■ **Exemplo 4.2** As seguintes tHs são válidas:

$$\{\ z = 5\ \}\ z \leftarrow 2 * z\ \{\ 6 > 5\ \}$$
$$\{\ z = 5\ \}\ z \leftarrow 2 * z\ \{\ z > 5\ \}$$
$$\{\ z = 5\ \}\ z \leftarrow 2 * z\ \{\ (z = 10) \wedge (z > x)\ \}$$
$$\{\ z = 5\ \}\ z \leftarrow 2 * z\ \{\ z = 10\ \}$$

■

■ **Exemplo 4.3** As seguintes tHs não são válidas. Os respectivos estados iniciais em evidência comprovam tal fato.

$$\{\ (x \geq y) \wedge (y \neq 0)\ \}\ z \leftarrow x/y\ \{\ z \geq 1\ \} \quad \text{(para } x > 0 \text{ e } y < 0\text{)}$$
$$\{\ -1 \leq z \leq 5\ \}\ z \leftarrow 2 * z\ \{\ -1 \leq z \leq 10\ \} \quad \text{(para } -1 \leq z < -0.5\text{)}$$

■

■ **Exemplo 4.4** Considere os seguintes algoritmos:

$$A_1 = z, a, b \leftarrow 0, 0, 0$$
$$A_2 = z, a, b \leftarrow a * b, 0, 0$$
$$A_3 = z \leftarrow a * b$$

Analisemos, para cada uma das pré- e pós-condições a seguir, quais destes três algoritmos tornam válidas as respectivas tHs dadas a seguir:

1. $\{\ (a \geq 0) \wedge (b \geq 0)\ \}\ ?\ \{\ z = ab\ \}$: apenas A_1 e A_3;
2. $\{\ (a \geq 0) \wedge (b \geq 0)\ \}\ ?\ \{\ z = \overline{ab}\ \}$: apenas A_2 e A_3;
3. $\{\ (a \geq 0) \wedge (b \geq 0)\ \}\ ?\ \{\ \underline{z} = ab\ \}$: apenas A_3.

■

■ **Exemplo 4.5** Alguns exemplos de pré- e pós-condição, seguidos das respectivas descrições em linguagem natural dos algoritmos que tornariam cada tH válida:

1. $\{\ N \geq 0\ \}\ ?\ \{\ \underline{s} = (\sum i \in [1..N] : A[i])\ \}$: fazer a variável s ser igual a soma dos elementos do subvetor $A[1..N]$ dado de entrada, sem alterar os elementos deste vetor;
2. $\{\ N \geq 0\ \}\ ?\ \{\ \underline{r}^2 \leq N < (\underline{r} + 0.01)^2\ \}$: fazer a variável r ser igual a raiz quadrada de N dada de entrada, com precisão de 0.01, sem alterar o valor de N;
3. $\{\ N \geq 0\ \}\ ?\ \{\ \text{“}\underline{A}[1..N] \text{ é uma permutação de } \overline{A}[1..N]\text{”} \wedge (\forall\, i \in [1..N) : A[i] \leq A[i+1])\ \}$: ordenar $A[1..N]$ ascendentemente;
4. $\{\ \mathbf{ProgPython}(x)\ \}\ ?\ \{\ \mathbf{ProgJava}(\underline{y}) \wedge \mathbf{Equivalentes}(x, \underline{y})\ \}$: traduzir o programa de entrada, escrito em sintaxe Python como conteúdo da variável x, para um programa equivalente, escrito em sintaxe Java como conteúdo da variável y.

■

■ **Exemplo 4.6** Relembre que pode ser conveniente utilizar predicados graficamente representados em asserções sobre vetores e matrizes. Como ilustração, reformulamos as tHs do exemplo anterior que fazem referência a vetores.

1. $\{\, N \geq 0 \,\}$? $\left\{ \underline{s} = A \boxed{\begin{matrix}1 & N \\ \Sigma & \end{matrix}} \right\}$

3. $\{\, N \geq 0 \,\}$? $\left\{ \text{“}\underline{A}[1..N] \text{ é uma permutação de } \overline{A}[1..N]\text{”} \wedge \boxed{\begin{matrix}1 & & N & \\ \forall\, i : A[i] \leq A[i+1] & & & \end{matrix}} \right\}$

■

4.2 Pré-Condição Mais Fraca (PMF)

Reveja as tHs apresentadas no Exemplo 4.1. Como ilustrado, é possível a elaboração de múltiplas tHs válidas variando-se apenas a pré-condição (isto é, mantendo-se fixos o algoritmo e a pós-condição). Sendo assim, uma pergunta natural é se haveria alguma pré-condição que seria preferível para a apresentação da tH. A resposta a esta pergunta conduz à definição da pré-condição mais fraca, que desempenha papel central na verificação de algoritmos.

Lembre que em uma tH válida, a pré-condição define o conjunto de estados iniciais para os quais o algoritmo garante que a pós-condição será satisfeita ao término da execução. Sendo assim, fixados o algoritmo A e a pós-condição $\mathbf{Q}$, o mais interessante seria estabelecer uma pré-condição $\mathbf{P}$ que defina o conjunto de *todos* os estados iniciais que torne a tH $\{\, \mathbf{P} \,\}\, A \,\{\, \mathbf{Q} \,\}$ válida. Tal pré-condição é chamada de *pré-condição mais fraca* (PMF), e denotada por

$$\textsc{Pmf}(\text{“}A\text{”}, \mathbf{Q})$$

Como exemplo, considere novamente o Exemplo 4.1. Nele, o conjunto de estados definido por cada pré-condição é subconjunto próprio daquele definido pela pré-condição apresentada em seguida naquele exemplo. Além disso, é fácil se convencer que $(x/y \geq 1) \wedge (y \neq 0)$ corresponde à $\textsc{Pmf}(\text{“}z \leftarrow x/y\text{”}, z \geq 1)$.

Pode ser conveniente a interpretação do significado de uma PMF através da representação gráfica de conjuntos. Observe a Figura 4.1. O retângulo corresponde ao conjunto de todos as possíveis valorações das variáveis de certo algoritmo A, isto é, o conjunto universo de todos os possíveis estados de A. Cada círculo preto corresponde a um estado específico. Uma seta ligando o estado I ao estado F significa que A, ao executar a partir do estado inicial I, finaliza sua execução no estado F. Uma seta ligando o estado I a um símbolo de “$\times$” significa que A, ao executar a partir do estado inicial I, ou finaliza sua execução com erro, ou nunca termina. Dada uma pós-condição $\mathbf{Q}$, como vimos na Seção 3.4, $\mathbf{Q}$ também representa um conjunto de estados, a saber daqueles que satisfazem $\mathbf{Q}$. Tal conjunto é então um subconjunto do universo de estados possíveis, que é representado na figura como um oval. Portanto, graficamente, o subconjunto $\textsc{Pmf}(\text{“}A\text{”}, \mathbf{Q})$ que está demarcado na figura, também como um oval, é de fato o conjunto de todos os estados iniciais em que o algoritmo A termina em um estado satisfazendo $\mathbf{Q}$.

■ **Exemplo 4.7** A seguir, determinamos diversas PMFs a título de exemplo. As PMFs foram obtidas, e devem ser verificadas, puramente pela observação e raciocínio lógico, sem método sistemático. É altamente instrutivo se tentar, por meios próprios, obter as mesmas expressões. Tal empreendimento valorizará os métodos sistemáticos de se obter PMFs dos diversos comandos, a serem apresentados nos capítulos que se seguem.

1. $A_1 = v \leftarrow v + 1$; $\mathbf{Q1} = v \geq 0$;
   $\textsc{Pmf}(\text{“}A_1\text{”}, \mathbf{Q1}) = v \geq -1$

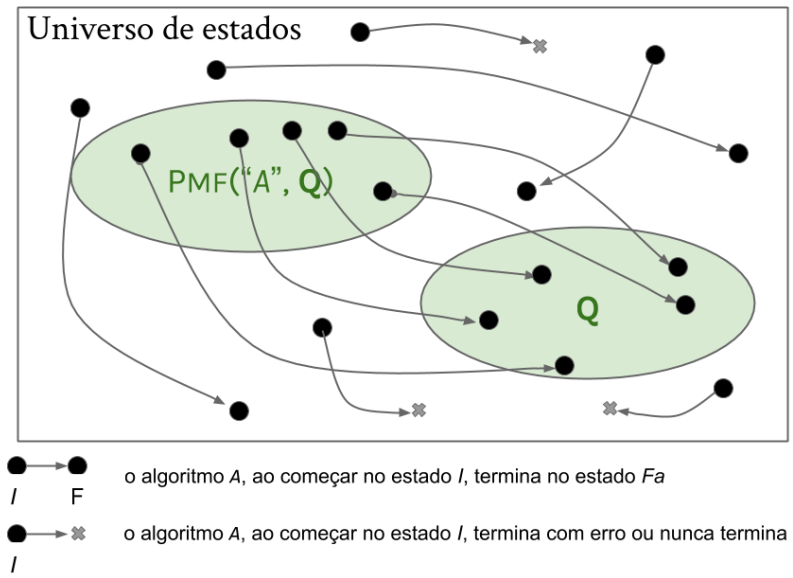


Figura 4.1: Representação em conjuntos do conceito de PMF("A", **Q**).

2. $A_2 = v \leftarrow v * 10$; $\mathbf{Q2} = 10 \leq v \leq x$;
   PMF("A_2", **Q2**) $= 1 \leq v \leq x/10$

3. $A_3 = v \leftarrow v + x$; $\mathbf{Q_3} = 10 \leq v \leq 10x$;
   PMF("A_3", $\mathbf{Q_3}$) $= 10 - x \leq v \leq 9x$

4. $A_4 = v \leftarrow v^2$; $\mathbf{Q_4} = v^2 \leq x$;
   PMF("A_4", $\mathbf{Q_4}$) $= v^4 \leq x$

5. $A_5 = b[i] \leftarrow 10 * x$; $\mathbf{Q_5} = b[z] = 10$;
   PMF("A_5", $\mathbf{Q_5}$) $= ((i = z) \wedge (x = 1)) \vee ((i \neq z) \wedge (b[z] = 10))$

6. $A_6 = x \leftarrow y$; $\mathbf{Q_6} = x \leq y$;
   PMF("A_6", $\mathbf{Q_6}$) $= \mathbf{V}$

7. $A_7 = x \leftarrow y$; $\mathbf{Q_7} = x > y$;
   PMF("A_7", $\mathbf{Q_7}$) $= \mathbf{F}$

8. $A_8 = x \leftarrow -x$; $\mathbf{Q_8} = x = |x|$;
   PMF("A_8", $\mathbf{Q_8}$) $= x \leq 0$

9. $A_9 =$
   se $x \geq y$ **então**
   | $x \leftarrow x + a$
   senão
   | $x \leftarrow x - b$

$\mathbf{Q_9} = x \geq 10$;
$\text{PMF}(\text{“}A_9\text{”}, \mathbf{Q_9}) = (x \geq \max\{y, 10-a\}) \vee (10+b \leq x < y)$

10. $A_{10} =$ **enquanto** $i \leq 3$ **faça**
    | $i, j \leftarrow i+1, j+1$

$\mathbf{Q_{10}} = j \geq 10$;
$\text{PMF}(\text{“}A_{10}\text{”}, \mathbf{Q_{10}}) = (i > 3) \wedge (j \geq 10) \vee (i \leq 3) \wedge (j \geq 6+i)$

■

4.3 Verificação Formal de Algoritmos

Esta seção trata do problema de verificar se um algoritmo A está correto, dadas sua pré-condição $\mathbf{P}$ e sua pós-condição $\mathbf{Q}$. Em outras palavras, trata-se de decidir se

$$\{\ \mathbf{P}\ \}\ A\ \{\ \mathbf{Q}\ \}$$

é uma tH válida. Para responder a esta questão, como dissemos, o conceito de PMF introduzido na Seção 4.2 terá papel central.

Comecemos por reanalisar a Figura 4.1. Como foi dito, o conjunto de estados correspondendo a $\text{PMF}(\text{“}A\text{”}, \mathbf{Q})$ são todos aqueles iniciais em que A termina em um estado satisfazendo Q. Segue diretamente desta definição, portanto, que para que a tH $\{\ \mathbf{P}\ \}\ A\ \{\ \mathbf{Q}\ \}$ seja válida, $\mathbf{P}$ deve ser um subconjunto de $\text{PMF}(\text{“}A\text{”}, \mathbf{Q})$ ou, caso contrário, há um estado que satisfaz $\mathbf{P}$ mas não a $\text{PMF}(\text{“}A\text{”}, \mathbf{Q})$, implicando que neste estado por inicial ou A não termina, ou termina com erros, ou termina em um estado que não satisfaz $\mathbf{Q}$. Ou seja, devemos verificar se

$$\mathbf{P} \subseteq \text{PMF}(\text{“}A\text{”}, \mathbf{Q})$$

ou, reinterpretando $\mathbf{P}, \mathbf{Q}$ como predicados e não como conjuntos, deve haver uma prova que

$$\mathbf{P} \Rightarrow \text{PMF}(\text{“}A\text{”}, \mathbf{Q})$$

Tamanha a importância desta conclusão, que a destacaremos a seguir e chamaremos de *Método #*1 tal maneira de se argumentar a correção de um algoritmo.

Método 4.1 — Método #1. Para verificar se a tH

$$\{\ \mathbf{P}\ \}\ A\ \{\ \mathbf{Q}\ \}$$

é válida, determina-se o predicado

$$\text{PMF}(\text{“}A\text{”}, \mathbf{Q})$$

e, em seguida, prova-se que

$$\mathbf{P} \Rightarrow \text{PMF}(\text{“}A\text{”}, \mathbf{Q})$$

O Método #1 é geral, isto é, sempre é possível considerar aplicá-lo. Contudo, nem sempre tal aplicação é um processo simples, pois tanto a determinação de $\text{PMF}(\text{“}A\text{”}, \mathbf{Q})$, quanto a prova de $\mathbf{P} \Rightarrow \text{PMF}(\text{“}A\text{”}, \mathbf{Q})$ pode exigir algum esforço. Por exemplo, revise a determinação das PMFs dadas no Exemplo 4.7 e confirme que as últimas duas PMFs em especial, que envolvem os comandos mais elaborados de condicional e de repetição, foram mais desafiadoras. Neste caso, há um método alternativo, que chamaremos de *Método #*2 e o delinearemos de forma geral como se segue:

Método 4.2 — Método #2. Para verificar se a tH

$$\{ \mathbf{P} \}\ A\ \{ \mathbf{Q} \}$$

é válida, observa-se que tipo de comando A consiste, que será um dos seguintes:

- vazio;
- atribuição;
- composição;
- alternativo (condicional);
- iterativo (repetição);
- chamada de função.

Dependendo do tipo, haverão propriedades $\pi_1, \pi_2, \ldots, \pi_k$ que, se satisfeitas, garantem a correção daquele tipo particular de comando. Verifica-se, então, se cada uma delas é satisfeita para a tH em questão.

Nos capítulos seguintes passaremos a estudar cada um dos possíveis comandos de que um algoritmo A consiste (vazio, atribuição, alternativo, iterativo, chamada de função ou composição). Para cada um deste comandos, o Método #2 descrito de forma geral acima será especializado, apresentando-se quais propriedades específicas $\pi_1, \pi_2, \ldots, \pi_k$ devem ser verificadas. Para comandos mais complexos, como o comando iterativo, veremos que o Método #1 quase não será usado. Neste caso, o emprego do Método #2 será de fato como será conduzida a verificação formal do algoritmo.

Verificação vs. Elaboração

Note que a seção de verificação, até aqui, trata do problema de verificar tHs nas quais um algoritmo A já está dado. O método de elaboração do algoritmo A neste caso, não nos importa: o importante é verificar se de fato A é correto.

É bem sabido que diversos algoritmos distintos podem ser empregados na resolução de um mesmo problema computacional. Dentre as possíveis várias alternativas, algumas podem ser particularmente difíceis de compreender e de se argumentar em favor de sua correção. Certos algoritmos podem ser escritos de maneiras que facilitem o argumento. Assim, diante do problema de elaborar um algoritmo A que torne válida certa tH, parece promissora a ideia de que A possa ser construído *com vistas à correção*, de modo de que a sua prova seja facilitada. Neste sentido é que descrevemos o Método #3.

Método 4.3 — Método #3. Para elaborar um algoritmo A de modo que a tH

$$\{ \mathbf{P} \}\ A\ \{ \mathbf{Q} \}$$

seja válida, escolhe-se que tipo de comando A provavelmente consiste (vazio, atribuição, alternativo, iterativo, chamada de função ou composição) e, dependendo do tipo, haverá uma estratégia específica que, se seguida, produz um algoritmo correto por construção.

Para enfatizar a importância deste método, quotamos Edsger Dijkstra, em seu discurso de recebimento do *Turing Award* do ano de 1972:

> "The only effective way to raise the confidence level of a program significantly is to give a convincing proof of its correctness. But one should not first make the program

> and then prove its correctness, because then the requirement of providing the proof would only increase the poor programmer's burden. On the contrary: the programmer should let correctness proof and program grow hand in hand."
> - em discurso intitulado *The Humble Programmer*, Edsger W. Dijkstra (1972)

Na apresentação de cada comando nos capítulos seguintes, também apresentaremos os detalhes das estratégias de cada comando, especializando o Método #3 de forma que ele possa ser utilizado concretamente.

4.4 Exercícios

1. Determine PMF("A", $\mathbf{Q}$) para os seguinte algoritmos e pós-condições. Considere todos os índices dentro dos vetores.

	A	$\mathbf{Q}$
(a)	$i \leftarrow i+1$	$i > 10$
(b)	$i \leftarrow i+2;\ j \leftarrow j-1$	$i+j > 0$
(c)	$i \leftarrow i+1;\ j \leftarrow j-1$	$ij = 1$
(d)	$a[i] \leftarrow 1$	$a[i] = a[j]$
(e)	$a[i] \leftarrow 1$	$a[r] = a[s]$
(f)	$x \leftarrow x*x$	$x < 100$
(g)	$z \leftarrow y$	$z = \max\{x,y\}$
(h)	$z \leftarrow$ y se x < y; x caso contrário	$z = \max\{x,y\}$
(i)	$z \leftarrow$ y se x < y; x caso contrário	$z = \min\{x,y\}$
(j)	$z \leftarrow 10$	$z = 0$

2. Verifique se cada afirmação é verdadeira ou falsa:
   (a) "{ **P** } A { **Q** }" e "{ PMF("A", **Q**) } A { **Q** }" são especificações equivalentes.
   (b) "{ PMF("A", **Q**) } A { **Q** }" é uma tH válida para quaisquer A, **Q**
   (c) Uma asserção com n variáveis define 2^n potenciais estados distintos
   (d) Para todo algoritmo A e asserção **Q**, temos que PMF("A", **Q**) $\vee$ PMF("A", $\neg$**Q**) é uma tautologia
   (e) Para todo algoritmo A e asserção **Q**, temos que se "{ **V** } A { **V** }" é uma tH válida, então PMF("A", **Q**) $\vee$ PMF("A", $\neg$**Q**) é uma tautologia

5. Comando Vazio

5.1 Descrição

A partir deste capítulo, e até o fim desta primeira parte, dedicaremos cada capítulo a um tipo de comando que um algoritmo A pode consistir. Uma observação importante é que consideraremos A como consistindo de um *único* comando entre aqueles a seguir:

- vazio;
- atribuição;
- composição;
- alternativo (condicional);
- iterativo (repetição);
- chamada de função.

Note que os comandos alternativo, iterativo e composição consistirão, por sua vez, de outros subcomandos, cada um com seus respectivos tipos.

O comando $\emptyset$ (lê-se: *comando vazio*) é o mais simples dentre aqueles que abordaremos e, por isso, começamos o estudo da correção por ele. Sua simplicidade, no entanto, não deve ser confundida com sua importância. Com efeito, diversos problemas de correção no desenvolvimento de programas práticos decorrem de seu uso inadvertido. Esta afirmação pode ser desacreditada a princípio por programadores experimentados em diversas linguagens de programação, pela dificuldade de sequer encontrar algum comando, qualquer que seja a linguagem, cujo nome se assemelhe a "$\emptyset$". Tal descrença se agravará com a definição do comando, dada a seguir.

Definição 1 — Comando $\emptyset$. O *comando* $\emptyset$ não altera o estado de máquina corrente. Isto é, todas as variáveis possuem os respectivos valores inalterados depois da execução do comando $\emptyset$.

Deste modo, para que a tH $\{ \mathbf{P} \}\ \emptyset\ \{ \mathbf{Q} \}$ seja válida, nenhuma alteração do estado deve ser necessária, isto é, quando $\mathbf{Q}$ é satisfeito sempre que $\mathbf{P}$ o for.

A definição do comando $\emptyset$ pode parecer se tratar de mero capricho intelectual. Com efeito, pode-se argumentar que o problema de encontrar um algoritmo que satisfaça um tripla de Hoare para a qual o comando $\emptyset$ seja uma solução, não é, propriamente, um problema real. Contudo, mesmo sob a hipótese de que nenhum problema real a é passível de ser resolvido com o comando $\emptyset$, tal comando ainda assim será útil. Ele será empregado para se resolver subproblemas gerados a partir de comandos que necessitam de subcomandos, como alternativo e iterativo. Este emprego do comando vazio será exemplificado nos capítulos seguintes.

5.2 Correção

Verificando o Comando Vazio

A verificação da correção do comando $\emptyset$ se procede como apresentada no Capítulo 4. Para tal, é necessário determinar a pré-condição mais fraca do comando, que é dada pelo Teorema 5.1.

Teorema 5.1 — PMF do Comando $\emptyset$. Seja $\mathbf{Q}$ uma proposição. Temos que

$$\text{PMF}(\text{“}\emptyset\text{”}, \mathbf{Q}) = \mathbf{Q}$$

■ **Exemplo 5.1** Para $\mathbf{Q} = (j \geq 10)$, temos que $\text{PMF}(\text{“}\emptyset\text{”}, \mathbf{Q}) = (j \geq 10)$. ■

■ **Exemplo 5.2** Para $\mathbf{Q} = (i \leq n) \wedge (j \leq i)$, temos que $\text{PMF}(\text{“}\emptyset\text{”}, \mathbf{Q}) = (i \leq n) \wedge (j \leq i)$. ■

Com a determinação da PMF, a verificação da correção pelo Método #1 pode então ser conduzida. Neste caso, tal processo se reduz trivialmente ao seguinte.

Método 5.2 — Método #2. A tH

$$\{ \mathbf{P} \}\ \emptyset\ \{ \mathbf{Q} \}$$

é válida se, e somente se,

$$\mathbf{P} \Rightarrow \mathbf{Q}$$

Elaborando o Comando Vazio

A estratégia de elaboração para o comando $\emptyset$ é formalizada a seguir.

Método 5.3 — Método #3. Para elaborar um algoritmo A de modo que a tH

$$\{ \mathbf{P} \}\ A\ \{ \mathbf{Q} \}$$

seja válida, verifique se

$$\mathbf{P} \Rightarrow \mathbf{Q}$$

e, neste caso, basta fazer

$$A = \emptyset$$

É interessante observar as respectivas ênfases de cada método, a saber:

- o Método #1 estabelece como se procede o cálculo da PMF do comando em questão;
- o Método #2 evidencia a verificação final que precisa ser conduzida para se garantir a correção do comando em questão; e

- o Método #3 sugere o processo de raciocínio a ser empregado para aplicar o comando em questão, de modo que o algoritmo esteja correto por construção.

Os três métodos apresentados para o comando vazio podem parecer, em última instância, um mesmo. E de fato o são, apenas com ênfases diferentes na apresentação. A distinção entre os métodos tornará mais evidente à medida que os comandos considerados se tornarem mais complexos.

5.3 Exercícios

1. Verifique a correção dos seguintes algoritmos:
   (a) $\{ x = 0 \}\ \emptyset\ \{ x \geq 0 \}$
   (b) $\{ x \geq 0 \}\ \emptyset\ \{ x = 0 \}$
   (c) $\{ x^2 + 2x = 0 \}\ \emptyset\ \{ x \geq 0 \}$
   (d) $\{$ "a[1..n] é ordenado ascendentemente" $\}\ \emptyset$
   $\{ (\exists\, i \in [1..n] : (\forall\, j \in [1..i] : a[j] \leq a[i]) \wedge (\forall\, j \in (i..n] : a[i] \leq a[j])) \}$
   (e) $\{ x^2 = (a+b)^2 \}\ \emptyset\ \{ x = a+b \}$
   (f) $\{ x = a+b \}\ \emptyset\ \{ x^2 = (a+b)^2 \}$
   (g) $\{ x = y-1 \}\ \emptyset\ \{ x < y \}$
   (h) $\{ x < y \}\ \emptyset\ \{ x = y-1 \}$

6. Comando de Atribuição

6.1 Descrição

Este capítulo descreve um comando central na elaboração de algoritmos: o comando de atribuição. Em conjunto com o comando de chamada a função, que veremos adiante, trata-se dos únicos comandos que alteram o estado de máquina.

Definimos formalmente o comando de atribuição da seguinte forma:

Definição 2 — Comando de Atribuição. A sintaxe do *comando de atribuição* é descrita por

$$v_1, v_2, \ldots, v_k \leftarrow e_1, e_2, \ldots, e_k$$

ou, equivalentemente, por

$$v_1 \leftarrow e_1 \mid v_2 \leftarrow e_2 \mid \cdots \mid v_k \leftarrow e_k$$

onde v_i é uma variável do algoritmo e e_i uma expressão que resulta em um valor do mesmo tipo de v_i, para $i = 1, 2, \ldots, k$. A semântica do comando é atribuir a cada variável v_i o valor da expressão e_i, primeiro determinando-se todos os valores de $e_1, e_2, \ldots, e_k$ para, em seguida, atribuí-los às respectivas variáveis $v_1, v_2, \ldots, v_k$.

■ **Exemplo 6.1** As seguintes tHs são válidas:

- $\{\ (x = 2) \wedge (y = 3)\ \}\ x, y \leftarrow y, x\ \{\ (x = 3) \wedge (y = 2)\ \}$
- $\{\ (x = 2) \wedge (y = 3)\ \}\ x \leftarrow y \mid y \leftarrow x\ \{\ (x = 3) \wedge (y = 2)\ \}$
- $\{\ (a = 2) \wedge (b = 3)\ \}\ a, b \leftarrow a + b, a - b\ \{\ (a = 5) \wedge (b = -1)\ \}$
- $\{\ (a = 2) \wedge (b = 3) \wedge (c = 4)\ \}\ a, b, c \leftarrow b, c, a\ \{\ (a = 3) \wedge (b = 4) \wedge (c = 2)\ \}$

■

O comando de atribuição, tal como definido, é chamado também de *atribuição múltipla*. Na grande maioria das linguagens de programação, o comando de atribuição é implementado em uma

versão mais restrita, na qual apenas uma única variável, e por consequência uma única expressão, pode ser usada no comando.

Substituição Textual

Para a verificação da correção do comando atribuição, a ser apresentada na sequência, é importante introduzir o conceito da substituição textual.

Definição 3 Seja $\mathbf{P}$ um predicado. A *substituição textual* de $x_1, x_2, \ldots, x_k$ por $e_1, e_2, \ldots, e_k$, onde $x_1, x_2, \ldots, x_k$ são identificadores e $e_1, e_2, \ldots, e_k$ expressões quaisquer, consiste na obtenção do predicado denotado por

$$\mathbf{P}^{x_1,x_2,\ldots,x_k}_{e_1,e_2,\ldots,e_k}$$

obtido a partir de $\mathbf{P}$ através da substituição de cada variável livre x_i, caso haja tal variável livre em $\mathbf{P}$, pela expressão e_i, para todo $i = 1, 2, \ldots, k$. Tais substituição devem seguir as seguintes regras:

- as substituições são consideradas simultâneas, de modo que uma variável x_j que porventura se encontre na expressão e_i sendo substituída não sofra o efeito da substituição de x_j por e_j;
- se necessário, envolver a expressão e_i que substitui x_i em parênteses; é necessário que no predicado resultante ocorra a avaliação de e_i;
- se necessário, renomeie variáveis amarradas em $\mathbf{P}$ antes da substituição; é necessário que não se crie variáveis amarradas em $\mathbf{P}^{x_1,x_2,\ldots,x_k}_{e_1,e_2,\ldots,e_k}$ que não existiam em $\mathbf{P}$;

■ **Exemplo 6.2** Considere o predicado $\mathbf{P} = (x > y) \wedge (x + y \leq z)$. São exemplos de substituições textuais:

- $\mathbf{P}^{x}_{4} = (4 > y) \wedge (4 + y \leq z)$
- $\mathbf{P}^{x,y}_{y,z} = (y > z) \wedge (y + z \leq z) = (z < y \leq 0)$
- $\mathbf{P}^{x,w,z}_{y+1,y,y} = (y + 1 > y) \wedge (y + 1 + y \leq y) = (y \leq -1)$

■

■ **Exemplo 6.3** Considere o predicado $\mathbf{P} = (x^2 + 3x \geq y) \vee (\forall\, x \in [y/2..y] : \text{“x não é primo”})$. São exemplos de substituições textuais:

- $\mathbf{P}^{x}_{4} = (4^2 + 3 \cdot 4 \geq y) \vee (\forall\, x \in [y/2..y] : \text{“x não é primo”})$
  $= (28 \geq y) \vee (\forall\, x \in [y/2..y] : \text{“x não é primo”})$
- $\mathbf{P}^{x,y}_{y,x} = (y^2 + 3y \geq x) \vee (\forall\, z \in [x/2..x] : \text{“z não é primo”})$
- $\mathbf{P}^{x,y}_{4+x,x^2} = ((4 + x)^2 + 3(4 + x) \geq x^2) \vee (\forall\, z \in [x^2/2..x^2] : \text{“z não é primo”})$
  $= (x \geq -28/11) \vee (\forall\, z \in [x^2/2..x^2] : \text{“z não é primo”})$

■

6.2 Correção

Verificando o Comando de Atribuição

A verificação da correção do comando de atribuição se procede como apresentada no Capítulo 4. Como vimos, é necessário apenas estabelecer como se determinar a pré-condição mais fraca do comando, que é dada pelo Teorema 6.1.

Teorema 6.1 — PMF do Comando de Atribuição. Sejam $\mathbf{Q}$ uma proposição, $x_1, x_2, \ldots, x_k$ variáveis e $e_1, e_2, \ldots, e_k$ expressões de valores dos respectivos tipos das variáveis. Temos que

$$\text{PMF}(\text{“}x_1,\ x_2,\ \ldots,\ x_k \leftarrow e_1,\ e_2,\ \ldots,\ e_k\text{”}, \mathbf{Q}) = \mathbf{Q}^{x_1,x_2,\ldots,x_k}_{e_1,e_2,\ldots,e_k}$$

■ **Exemplo 6.4** Para $\mathbf{Q} = (j \geq 10)$, temos que $\text{PMF}(\text{“}j \leftarrow 15\text{”}, \mathbf{Q}) = \mathbf{Q}^{j}_{15} = (15 \geq 10) = \mathbf{V}$. ■

■ **Exemplo 6.5** Para $\mathbf{Q} = (j \geq 10)$, temos que $\text{PMF}(\text{“}j \leftarrow 5\text{”}, \mathbf{Q}) = \mathbf{Q}^{j}_{5} = (5 \geq 10) = \mathbf{F}$. ■

■ **Exemplo 6.6** Para $\mathbf{Q} = (j \leq \sqrt{x})$, temos que $\text{PMF}(\text{“}x,\ j \leftarrow x/2,\ 2 * j\text{”}, \mathbf{Q}) = \mathbf{Q}^{x,j}_{x/2,2j} = 2j \leq \sqrt{x/2}) = (j \leq \sqrt{x/8})$. ■

Com a determinação da PMF, a verificação da correção pelo Método #1 pode então ser conduzida. Neste caso, tal processo se reduz trivialmente ao seguinte.

Método 6.2 — Método #2. A tH

$$\{\ \mathbf{P}\ \}\ x_1, x_2, \ldots, x_k \leftarrow e_1, e_2, \ldots, e_k\ \{\ \mathbf{Q}\ \}$$

é válida se, e somente se,

$$\mathbf{P} \Rightarrow \mathbf{Q}^{x_1,x_2,\ldots,x_k}_{e_1,e_2,\ldots,e_k}$$

■ **Exemplo 6.7** Considere a tH

$$\{\ \mathbf{P} = (x \geq 0)\ \}\ x \leftarrow 3 * (x+4)\ \{\ \mathbf{Q} = (x \geq 10)\ \}$$

Tal tH é válida se

$$\mathbf{P} \Rightarrow \mathbf{Q}^{x}_{3(x+4)}$$

o que resulta em provar que

$$(x \geq 0) \Rightarrow (3(x+4) \geq 10)$$

ou, simplificando o consequente,

$$(x \geq 0) \Rightarrow \left(x \geq -\frac{2}{3}\right)$$

o que se verifica. Portanto, a tH é válida. ■

■ **Exemplo 6.8** Considere a tH

$$\{\ \mathbf{P} = (x \geq y)\ \}\ z \leftarrow x\ \{\ \mathbf{Q} = (z = \max\{x,y\})\ \}$$

Tal tH é válida se

$$\mathbf{P} \Rightarrow \mathbf{Q}^{z}_{x}$$

o que resulta em provar que

$$(x \geq y) \Rightarrow (x = \max\{x,y\})$$

o que se verifica. Portanto, a tH é válida. ■

Elaborando o Comando de Atribuição

Em geral, quando um algoritmo pode satisfazer uma pós-condição através de um comando de atribuição, a solução usualmente não é desafiadora, como ilustram os exemplos anteriores. No entanto, nem sempre este é o caso. Embora seja simples determinar as variáveis que devem ser alteradas, consultando-se ambas as pré-condição e pós-condição, os valores para os quais tais variáveis devem ser alteradas por vezes não estão claras. Como ilustração, considere o problema de elaborar um comando de atribuição A de modo que seja válida a tH

$$\{\ ab = z\ \}\ A\ \{\ (\underline{ab} + 1 = z) \wedge (a = \overline{a} + 1)\ \}$$

Pela pós-condição, as variáveis a, b devem ser alteradas, enquanto a variável z deve permanecer constante. Além disso, tais alterações devem ser de modo que z seja igual ao produto ab antes da atribuição, e igual a uma unidade a mais que o produto após a atribuição. Ainda pela pós-condição, é fácil identificar a alteração necessária à variável a. Em outras palavras, sabemos que

$$A = a, b \leftarrow a + 1, ?$$

e não é evidente que valor b deve receber. Para auxiliar problemas desta natureza, o Método #3 vem ao auxílio.

Método 6.3 — Método #3. Para elaborar um algoritmo A de modo que a tH

$$\{\ \mathbf{P}\ \}\ x \leftarrow \square\ \{\ \mathbf{Q}\ \}$$

seja válida, onde "$\square$" representa uma incógnita, determine o valor de

$$\mathbf{Q}' = \textsc{Pmf}(\text{“}x \leftarrow \square\text{”}, \mathbf{Q})$$

que estará, naturalmente, em função de $\square$. Em seguida, determine um valor de $\square$ de modo que

$$\mathbf{P} \Rightarrow \mathbf{Q}'$$

■ **Exemplo 6.9** Procuramos determinar $\square$ de modo que a tH

$$\{\ \mathbf{P} = (ab = z)\ \}\ a, b \leftarrow a + 1, \square\ \{\ \mathbf{Q} = (ab + 1 = z)\ \}$$

seja válida. Temos que

$$\begin{aligned}\mathbf{Q}' &= \textsc{Pmf}(\text{“}a, b \leftarrow a + 1, \square\text{”}, \mathbf{Q}) = \mathbf{Q}^{a,b}_{a+1,\square} \\ &= ((a+1)\square + 1 = z)\end{aligned}$$

Assim, se

$$\square = \frac{z-1}{a+1}$$

temos que $\mathbf{Q}'$ é uma tautologia, e portanto $\mathbf{P} \Rightarrow \mathbf{Q}'$ independente de pré-condição. ■

■ **Exemplo 6.10** Um algoritmo mantém controle sobre certo intervalo de um vetor B. Para tal, o algoritmo registra em uma variável i o início do intervalo e em uma variável m a quantidade de células além da posição i pela qual o intervalo se estende. Em determinado ponto de um algoritmo, faz-se necessário que o início deste intervalo seja alterado para dada posição $p + 1$, localizada dentro do intervalo, sem que a posição final do intervalo se modifique. Como atualizar as variáveis i, m?

Em outras palavras, queremos determinar o valor de $\square$ de modo a tornar válida a tH

$$\{\ \mathbf{P} = (i+m=\overline{c})\ \}\ i, m \leftarrow p+1, \square\ \{\ \mathbf{Q} = (i+m=\overline{c})\ \}$$

Temos que

$$\begin{aligned}\mathbf{Q}' &= \text{PMF}(\text{“}i, m \leftarrow p+1, \square\text{”}, \mathbf{Q}) = \mathbf{Q}^{i,m}_{p+1,\square} \\ &= (p+1+\square = \overline{c})\end{aligned}$$

Para que seja válido $\mathbf{P} \Rightarrow \mathbf{Q}'$, conclui-se que

$$\square = i+m-p-1$$

■

■ **Exemplo 6.11** Procuramos determinar $\square$ de modo que a tH

$$\left\{\ \mathbf{P} = (s = \ B\ \begin{array}{|c|c|c|}\hline \square & \overset{i}{\underset{j}{\Sigma}} & \square \\ \hline \end{array}\ )\ \right\}$$

$A = j, s \leftarrow j+1, \square$

$$\left\{\ \mathbf{Q} = (s = \ B\ \begin{array}{|c|c|c|}\hline \square & \overset{i}{\underset{j}{\Sigma}} & \square \\ \hline \end{array}\ )\ \right\}$$

seja válida. Temos que

$$\begin{aligned}\mathbf{Q}' &= \text{PMF}(\text{“}A\text{”}, \mathbf{Q}) = \mathbf{Q}^{j,s}_{j+1,\square} \\ &= (\square = \ B\ \begin{array}{|c|c|c|}\hline \square & \overset{i}{\underset{j+1}{\Sigma}} & \square \\ \hline \end{array}\ )\end{aligned}$$

Assim, se

$$\square = s + A[j+1]$$

temos que $\mathbf{P} \Rightarrow \mathbf{Q}'$ é uma proposição válida. ■

6.3 Atribuição a Vetores e Matrizes

Relembre o cálculo da PMF do Exemplo 4.7(5). Não é possível determinar aquela PMF de modo sistemático, através do Teorema 6.1. Com efeito, pelas hipóteses do teorema, vê-se que os identificadores sendo atribuídos devem ser variáveis. Embora uma posição específica de um vetor possa, em diversos contextos práticos, ser considerado como uma variável, não são conceitos equivalentes. Por exemplo, a expressão “$B[i]$” pode estar associada a diferentes locais de memória, dependendo do valor de i, o que não ocorre com uma expressão que corresponde a uma variável escalar. A expressão B representando todo o vetor, por outro lado, é uma variável, cujo conteúdo é uma sequência de valores. Portanto, a atribuição

$$B[i] \leftarrow e$$

pode ser vista como uma atribuição ao vetor B de outro vetor B', tal que B' possui os mesmos elementos de B exceto aquele de posição i, cujo valor é igual a e. Será desta maneira que a atribuição de um elemento de um vetor será abordada para que o Teorema 6.1 possa ser aplicado. Para isso, é necessário introduzirmos uma notação para denotar mudanças a posições específicas de um vetor.

Definição 4 Seja B um vetor, $i_1, i_2, \ldots, i_k$ inteiros distintos correspondendo a posições de B e $e_1, e_2, \ldots, e_k$ expressões de valor do mesmo tipo dos elementos de B. A expressão

$$(B;\, i_1{:}e_1,\, i_2{:}e_2,\, \ldots,\, i_k{:}e_k)$$

denota o vetor tal que

$$(B;\, i_1{:}e_1,\, i_2{:}e_2,\, \ldots,\, i_k{:}e_k)[r] = \begin{cases} B[i] & \text{se } r \notin \{i_1, i_2, \ldots, i_k\} \\ e_j & \text{se } r = i_j \end{cases}$$

■ **Exemplo 6.12** Se $b = [3\ 1\ 4\ 0\ \text{-}1]$, então:

- $(b;\, 1{:}\text{-}2,\, 4{:}1)[1] = \text{-}2$
- $(b;\, 1{:}\text{-}2,\, 4{:}1)[3] = 4$
- $(b;\, 1{:}\text{-}2,\, 4{:}1) = [\text{-}2\ 1\ 4\ 1\ \text{-}1]$

■

■ **Exemplo 6.13** Sejam $A = B[i] \leftarrow 10 * x$ e $\mathbf{Q} = (b[z] = 10)$. Fazendo com que a atribuição de B na atribuição A seja na variável vetor, e não em uma posição deste vetor, reescrevemos A como

$$A = B \leftarrow (B;\, i{:}10 * x)$$

e, portanto, temos que

$$\begin{aligned} \text{PMF}(\text{“}A\text{”}, \mathbf{Q}) &= \mathbf{Q}^B_{(B;\, i:10*x)} \\ &= ((B;\, i{:}10 * x)[z] = 10) \end{aligned}$$

que já é, em si, uma expressão final. Caso se queira apresentar a expressão desta PMF sem a notação de modificação de vetor, podemos fazê-lo da seguinte forma. Dado que

$$(B;\, i{:}10 * x)[z] = \begin{cases} B[z] & \text{se } z \neq i \\ 10x & \text{se } z = i \end{cases}$$

então podemos reescrever a PMF como

$$\text{PMF}(\text{“}A\text{”}, \mathbf{Q}) = ((z = i) \wedge (x = 1)) \vee ((z \neq i) \wedge (B[z] = 10))$$

■

6.4 Exercícios

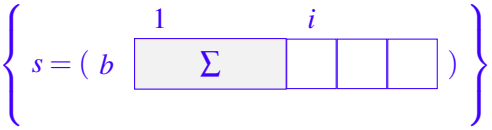
1. Determine $\text{PMF}(\text{“}A\text{”}, \mathbf{Q})$ para os seguintes algoritmos A e pós-condições $\mathbf{Q}$. Considere todos os índices dentro dos vetores.

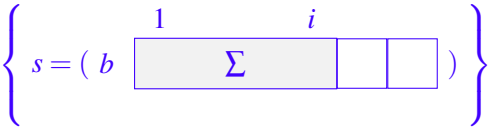
	A	$\mathbf{Q}$
(a)	$x \leftarrow 2*y+3$	$x = 13$
(b)	$x \leftarrow x+y$	$x < 2y$
(c)	$j \leftarrow j+1$	$b\ \overset{1}{\square}\ \overset{j}{\boxed{=5}}\ \square$
(d)	$todos5 \leftarrow (b[j] = 5)$	$todos5 = (\ b\ \overset{1}{\square}\ \overset{j}{\boxed{=5}}\ \square\ )$
(e)	$todos5 \leftarrow todos5 \wedge (b[j] = 5)$	$todos5 = (\ b\ \overset{1}{\square}\ \overset{j}{\boxed{=5}}\ \square\ )$
(f)	$x \leftarrow (x+y)*(x-y)$	$x+y^2 \neq 0$
(g)	$z, x, y \leftarrow 1, c, d$	$zxy = cd$
(h)	$i, s \leftarrow 1, b[0]$	$(1 \leq i < n) \wedge (s =\ b\ \overset{1}{\square}\ \overset{i}{\boxed{\Sigma}}\ \square\ \square\ )$
(i)	$a, n \leftarrow 0, 1$	$a^2 < n \leq (a+1)^2$
(j)	$i, s \leftarrow i+1, s+b[i]$	$(1 < i < n) \wedge (s =\ b\ \overset{1}{\square}\ \overset{i}{\boxed{\Sigma}}\ \square\ \square\ )$
(k)	$i, j \leftarrow i+1, j+1$	$i = j$
(l)	$z \leftarrow z*x \mid y \leftarrow y-1$	$(y \geq 0) \wedge (z*xy = c)$
(m)	$x \leftarrow x-y \mid y \leftarrow y-x$	$x+y = c$
(n)	$x \leftarrow x-y \mid y \leftarrow x+y$	$x+y = c$
(o)	$x \leftarrow 2*y \mid y \leftarrow 2*x$	$x = 2y$
(p)	$x \leftarrow 2*y \mid y \leftarrow y+4$	$(x = \bar{x}) \wedge (y = \bar{y})$
(q)	$a[k+1] \leftarrow \max\{a[k], a[k+1]\} \mid$ $a[k] \leftarrow \min\{a[k], a[k+1]\} \mid$ $k \leftarrow k+1$	$a[k] =\ b\ \overset{1}{\square}\ \overset{k}{\boxed{\max}}\ \square$
(r)	$b[j] \leftarrow i$	$b[b[i]] = i$
(s)	$b[i] \leftarrow 5$	$\exists\, j \in [i..n) : b[i] \leq b[j]$
(t)	$b[i] \leftarrow 5$	$\exists\, j \in [i..n) : b[i] < b[j]$
(u)	$b[i] \leftarrow 5$	$b = \bar{b}$
(v)	$b[i], b[j] \leftarrow b[j], b[i]$	$(b[i] = x) \wedge (b[j] = y)$
(x)	$b[i], b[j] \leftarrow b[j], b[i]$	$b[k] = z$

2. Verifique a validade das seguintes tHs:
   (a) $\{\ \mathbf{V}\ \}\ k \leftarrow k+1 \mid z \leftarrow k*k*k\ \{\ z = k^3\ \}$
   (b) $\{\ \mathbf{V}\ \}\ k \leftarrow k+1 \mid z \leftarrow (k+1)*(k+1)*(k+1)\ \{\ z = k^3\ \}$
   (c) $\{\ (z = k^3) \wedge (a = 3k^2) \wedge (b = 3k)\ \}$
   $k \leftarrow k+1 \mid z \leftarrow z+a+b+1 \mid a \leftarrow a+b+b+3 \mid b \leftarrow b+3$
   $\{\ (z = k^3) \wedge (a = 3k^2) \wedge (b = 3k)\ \}$
3. Determine o valor da expressão "□" de modo que cada uma das tHs a seguir seja válida.
   (a) $\{\ \mathbf{V}\ \}\ a, b \leftarrow a+1, \square\ \{\ b = a+1\ \}$
   (b) $\{\ i = j\ \}\ i, j \leftarrow i+1, \square\ \{\ i = j\ \}$

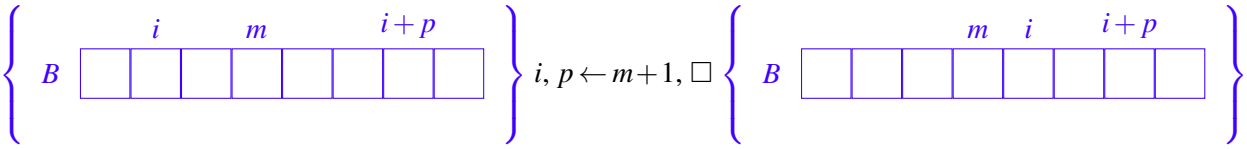
(c) $\{\ z+ab=c\ \}\ z, a \leftarrow z+b, \square\ \{\ z+ab=c\ \}$

(d) $\{\ \mathbf{Par}(a) \wedge (z+ab=c)\ \}\ a, b \leftarrow a/2, \square\ \{\ z+ab=c\ \}$

(e) $\{\ \mathbf{V}\ \}\ i, s \leftarrow 1, \square$ 

(f) $\{\ \mathbf{V}\ \}\ i, s \leftarrow 1, \square$ 

4. As variáveis i, m, p de um algoritmo são representadas pela figura abaixo, lado esquerdo. O algoritmo precisa fazer com que i aponte para a posição que sucede m, mas $i+p$ deve continuar resultando no mesmo valor, isto é, apontando a mesma posição. Como atualizar p, valor de "□" abaixo?



5. Uma abstração importante para algoritmos mais complexos é o conceito de estrutura (ou registro). Elabore uma proposta de como podemos lidar com estruturas com relação a correção de algoritmos, isto é, como pode ser calculada a PMF do comando

   $var.c \leftarrow$v

   onde *var* é uma variável do tipo estrutura, que possui o campo c definido, para o qual está sendo atribuído o valor v. (Dica: reveja como a atribuição a vetores é tratada, e aplique método análogo.)

7. Comando de Composição

7.1 Descrição

Este capítulo descreve um comando crucial na elaboração de algoritmos complexos que, paradoxalmente, pode passar desapercebido ao se estudar os comandos de uma linguagem: o comando de composição.

O comando de composição é aquele que permite que dois comandos A_1 e A_2 possam ser compostos para que o estado de máquina seja alterado sequencialmente, primeiro pelo efeito da execução de A_1, depois por aquele de A_2.

Definimos formalmente o comando de composição a seguir.

Definição 5 — Comando de Composição. A sintaxe do *comando de composição* é descrita por

$$A_1 ; A_2$$

onde A_1, A_2 são algoritmos arbitrários cujos efeitos de execução no estado devem ser compostos. Alternativamente, a composição pode ser escrita como

$$A_1$$
$$A_2$$

Além disso, caso haja alguma asserção **R** a ser descrita entre A_1 e A_2, a própria asserção elimina a necessidade de ";" de modo que a composição pode ser escrita também como

$$A_1 \ \{ \ \mathbf{R} \ \} \ A_2$$

A semântica do comando é alterar o estado de máquina pela execução do algoritmo A_1, seguida daquela de A_2.

Note que, pela definição, os algoritmos A_1 e A_2 podem ser, eles próprios, elaborados como comandos

de composição. Desta forma, pode-se produzir uma sequência com mais de dois algoritmos em composição, como ilustra o próximo exemplo.

■ **Exemplo 7.1** Uma maneira de denotar a composição dos algoritmos $A_1, A_2, \ldots, A_7$ é a seguinte

$A_1; A_2$
A_3
$\{ \mathbf{R_1} \}$
$A_4 \{ \mathbf{R_2} \} A_5; A_6$
A_7

■

7.2 Correção

Verificando o Comando de Composição

Para se proceder a verificação da correção do comando de composição tal como apresentada no Capítulo 4, estabelecemos a seguir como se determinar a pré-condição mais fraca do comando.

Teorema 7.1 — PMF do Comando de Composição. Sejam $\mathbf{Q}$ uma proposição e A_1, A_2 dois algoritmos. Temos que

$$\text{PMF}(\text{“}A_1; A_2\text{”}, \mathbf{Q}) = \text{PMF}(\text{“}A_1\text{”}, \text{PMF}(\text{“}A_2\text{”}, \mathbf{Q}))$$

■ **Exemplo 7.2** Para $\mathbf{Q} = (x \geq y)$, temos que

$$\begin{aligned}
\text{PMF}(\text{“}x \leftarrow x+y+5;\ y \leftarrow y-5\text{”}, \mathbf{Q}) &= \text{PMF}(\text{“}x \leftarrow x+y+5\text{”}, \text{PMF}(\text{“}y \leftarrow y-5\text{”}, \mathbf{Q})) \\
&= \text{PMF}(\text{“}x \leftarrow x+y+5\text{”}, \mathbf{Q}^y_{y-5}) \\
&= \text{PMF}(\text{“}x \leftarrow x+y+5\text{”}, x \geq y-5) \\
&= (x \geq y-5)^x_{x+y+5} \\
&= (x+y+5 \geq y-5) \\
&= (x \geq \text{-}10)
\end{aligned}$$

■

■ **Exemplo 7.3** Uma outra maneira de apresentar o cálculo do exemplo anterior é como a seguir:

$\{ \mathbf{Q_1} = \text{PMF}(\text{“}A_1\text{”}, \mathbf{Q_2}) = \mathbf{Q_2}^x_{x+y+5} = (x+y+5 \geq y-5) = (x \geq \text{-}10) \}$
$A_1 = x \leftarrow x+y+5$
$\{ \mathbf{Q_2} = \text{PMF}(\text{“}A_2\text{”}, \mathbf{Q}) = \mathbf{Q}^y_{y-5} = (x \geq y-5) \}$
$A_2 = y \leftarrow y-5$
$\{ \mathbf{Q} = (x \geq y) \}$

Note que as as asserções $\mathbf{Q_1}, \mathbf{Q_2}$ correspondem às respectivas PMFs do algoritmo no ponto onde estão definidas. Além disso, observe que é necessário que o cálculo das PMFs seja feito de baixo para cima, enquanto o algoritmo força a apresentação delas de cima para baixo. ■

Com a determinação da PMF, a verificação da correção pelo Método #1 pode então ser conduzida. Neste caso, tal processo se reduz trivialmente ao seguinte.

Método 7.2 — Método #2. Sejam os algoritmos $A_1, A_2, \ldots, A_k$. A tH

$$\{\ \mathbf{P}\ \}\ A_1; A_2;\ \ldots; A_k\ \{\ \mathbf{Q}\ \}$$

é válida se, e somente se,

$$\begin{aligned}
&\mathbf{P} \Rightarrow \mathbf{Q_1}\ \text{, onde}\\
&\mathbf{Q_1} = \textsc{Pmf}(\text{“}A_1\text{”}, \mathbf{Q_2})\ \text{, onde}\\
&\mathbf{Q_2} = \textsc{Pmf}(\text{“}A_2\text{”}, \mathbf{Q_3})\ \text{, onde}\\
&\ldots\\
&\mathbf{Q_{k-1}} = \textsc{Pmf}(\text{“}A_{k-1}\text{”}, \mathbf{Q_k})\ \text{, onde}\\
&\mathbf{Q_k} = \textsc{Pmf}(\text{“}A_k\text{”}, \mathbf{Q})
\end{aligned}$$

■ **Exemplo 7.4** Considere a tH

$$\{\ \mathbf{P} = \mathbf{V}\ \}\ t \leftarrow x; x \leftarrow y; y \leftarrow t\ \{\ \mathbf{Q} = ((x = \overline{y}) \wedge (y = \overline{x}))\ \}$$

Assim, temos que

$$\begin{aligned}
\mathbf{Q_3} &= \textsc{Pmf}(\text{“}y \leftarrow t\text{”}, \mathbf{Q}) = ((x = \overline{y}) \wedge (t = \overline{x}))\\
\mathbf{Q_2} &= \textsc{Pmf}(\text{“}x \leftarrow y\text{”}, \mathbf{Q_3}) = ((y = \overline{y}) \wedge (t = \overline{x}))\\
\mathbf{Q_1} &= \textsc{Pmf}(\text{“}t \leftarrow x\text{”}, \mathbf{Q_2}) = ((y = \overline{y}) \wedge (x = \overline{x}))
\end{aligned}$$

A tH é válida se

$$\mathbf{P} \Rightarrow \mathbf{Q_1}$$

o que se verifica. Portanto, a tH é válida. ■

Elaborando o Comando de Composição

Há duas estratégias importantes na elaboração no comando de composição. A primeira, é a elaboração de pré-processamento, pós-processamento ou objetivos intermediários.

Um *pré-processamento* é um algoritmo que visa preparar os dados de entrada para se adequar melhor à solução algorítmica, em geral para permitir uma melhor eficiência de tempo ou espaço. Como exemplo, podemos citar a operação de ordenação de vetores, ou a computação de determinadas propriedades dos dados (como os graus dos vértices de um grafo) para que o algoritmo principal se utilize destas propriedades.

Um *pós-processamento* é um algoritmo que visa preparar os dados de saída para serem apresentados. Nestes casos, o algoritmo principal leva vantagem em utilizar uma representação dos dados bem diferente daquela esperada de sua saída, necessitando assim um algoritmo para conduzir a tradução.

Quando o algoritmo a ser desenvolvimento possui uma série de etapas bem definidas, a especificação do conjunto de estados que atendem ao fim de cada etapa é o que consiste um *objetivo intermediário*.

A segunda estratégia consiste em pensar no problema em questão de trás para frente, definindo qual deveria ser o último comando, que encerraria o algoritmo a ser desenvolvido.

Em ambas as estratégias, a elaboração da composição se reduz à elaboração de outros algoritmos, com complexidade menor que o original.

O seguinte método formaliza a primeira estratégia.

Método 7.3 — Método #3(a). Para elaborar um algoritmo A de modo que a tH

$$\{\,\mathbf{P}\,\}\,A\,\{\,\mathbf{Q}\,\}$$

seja válida, no caso em que $\mathbf{R}$ é um predicado que especifica

- uma pós-condição de um algoritmo de pré-processamento para A; ou
- uma pré-condição para um algoritmo de pós-processamento para A; ou
- um objetivo intermediário de A,

determine algoritmos A_1, A_2 de modo que

$$\{\,\mathbf{P}\,\}\,A_1\,\{\,\mathbf{R}\,\}$$
$$\{\,\mathbf{R}\,\}\,A_2\,\{\,\mathbf{Q}\,\}$$

sejam tHs válidas. Neste caso,

$$A = A_1; A_2$$

é uma solução para o problema.

O próximo método formaliza a segunda estratégia.

Método 7.4 — Método #3(b). Para elaborar um algoritmo A de modo que a tH

$$\{\,\mathbf{P}\,\}\,A\,\{\,\mathbf{Q}\,\}$$

seja válida, no caso em que A_2 é um algoritmo parte de A, que deve vir ao seu final. determine um algoritmo A_1 de modo que

$$\{\,\mathbf{P}\,\}\,A_1\,\{\,\textsc{Pmf}(\text{“}A_2\text{”}, \mathbf{Q})\,\}$$

seja uma tH válida. Neste caso,

$$A = A_1; A_2$$

é uma solução para o problema.

■ **Exemplo 7.5** Procuramos determinar A de modo que a tH

$$\left\{ \mathbf{P} = (s = B \overset{i}{\square}\Sigma\overset{j}{\square}) \right\}$$

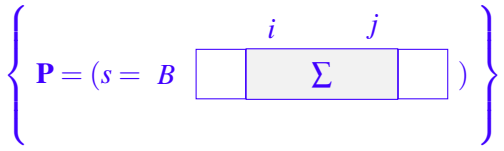
A

$$\left\{ \mathbf{Q} = ((\underline{s} = B \overset{i}{\square}\Sigma\overset{j}{\square}) \wedge (\underline{j} = \overline{j} + 1)) \right\}$$

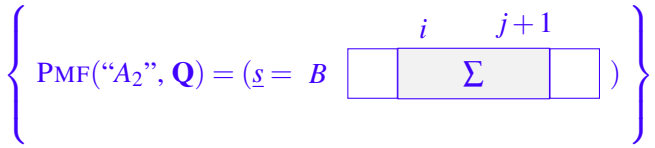
seja válida. Pela pós-condição, reconhecemos que as variáveis a serem alteradas devem ser s, j, e somente elas. A alteração de j é imediata a partir da pós-condição. Sendo assim, determinamos que

$$A_2 = j \leftarrow j+1$$

deve ser parte integrante de A e, sem perda de generalidade, pode vir ao seu final. Neste caso, o problema agora se reduz a terminar A_1 de modo que a tH



A_1



seja válida. Comparando-se pré- e pós-condições, determinamos que

$$A_1 = s \leftarrow s + A[j+1]$$

torna a tH anterior válida e, portanto,

$$A = s \leftarrow s + A[j+1];\ j \leftarrow j+1$$

resolve o problema original. ∎

7.3 Exercícios

1. Identifique quais das seguintes tHs são válidas:
   (a) $\{\ x,y > 0\ \}\ t \leftarrow x*x - y*y;\ x \leftarrow t/(x-y) - x;\ y \leftarrow \sqrt{t + x*x}\ \{\ (x = \overline{y}) \wedge (y = \overline{x})\ \}$
   (b) $\{\ x,y \neq 0\ \}\ x \leftarrow x*y;\ y \leftarrow x*y;\ x \leftarrow x/y\ \{\ (x = \overline{y}) \wedge (y = \overline{x})\ \}$
   (c) $\{\ x,y \neq 0\ \}\ x \leftarrow x*y;\ y \leftarrow x/y;\ x \leftarrow x/y\ \{\ (x = \overline{y}) \wedge (y = \overline{x})\ \}$
   (d) $\{\ \mathbf{V}\ \}\ x \leftarrow x\text{^}y;\ y \leftarrow x\text{^}y;\ x \leftarrow x\text{^}y\ \{\ (x = \overline{y}) \wedge (y = \overline{x})\ \}$
   onde a^b indica a operação de ou-exclusivo bit a bit entre as representações binárias de a e b. A operação de ou-exclusivo é tal que para os bits i, j, temos que $i\text{^}j = 1$ se exatamente um dos valores i ou j é igual a 1, e 0 caso contrário. Assim, 9^40 = 33 pois 001001^101000 = 100001
   (e) $\{\ \mathbf{V}\ \}\ x \leftarrow x\&y;\ y \leftarrow x\&y;\ x \leftarrow x\&y\ \{\ (x = \overline{y}) \wedge (y = \overline{x})\ \}$
   onde $a\&b$ indica a operação de e-exclusivo bit a bit entre as representações binárias de a e b. A operação de e-exclusivo é tal que para os bits i,j, temos que $i\&j = 1$ se ambos os valores i e j são iguais a 1. Assim, 9&40 = 8 pois 001001&101000 = 001000

2. Verifique a validade das seguintes tHs:
   (a) $\{\ z = k^3\ \}\ k \leftarrow k+1;\ z \leftarrow (k+1)*(k+1)*(k+1)\ \{\ z = k^3\ \}$
   (b) $\{\ z = k^3\ \}\ z \leftarrow (k+1)*(k+1)*(k+1);\ k \leftarrow k+1\ \{\ z = k^3\ \}$
   (c) $\{\ z = k^3\ \}\ z \leftarrow k*k*k;\ k \leftarrow k+1\ \{\ z = k^3\ \}$
   (d) $\{\ z = k^3\ \}\ k \leftarrow k+1;\ z \leftarrow k*k*k\ \{\ z = k^3\ \}$

(e) $\{\ (z = k^3) \wedge (a = 3k^2) \wedge (b = 3k)\ \}$
$b \leftarrow b+3;\ a \leftarrow a+b+b+3;\ k \leftarrow k+1;\ z \leftarrow z+a+b$
$\{\ (z = k^3) \wedge (a = 3k^2) \wedge (b = 3k)\ \}$

(f) $\{\ (z = k^3) \wedge (a = 3k^2) \wedge (b = 3k)\ \}$
$k \leftarrow k+1;\ z \leftarrow z+a+b;\ a \leftarrow a+b+b+3;\ b \leftarrow b+3$
$\{\ (z = k^3) \wedge (a = 3k^2) \wedge (b = 3k)\ \}$

3. Considere as atribuições:

$$A_1 = x \leftarrow e_1 \mid y \leftarrow e_2$$
$$A_2 = x \leftarrow e_1;\ y \leftarrow e_2$$
$$A_3 = y \leftarrow e_2;\ x \leftarrow e_1$$

   (a) Prove que, em geral, elas não são equivalentes, mostrando que as respectivas pré-condições mais fracas de A_1, A_2, A_3 em relação a alguma asserção $\mathbf{Q}$ e expressões e_1, e_2 são duas a duas distintas;
   (b) Prove que elas são equivalentes no caso em que x não ocorre em e_2 e y não ocorre em e_1.

4. Calcule $\text{PMF}(\text{“}x \leftarrow x+1;\ y \leftarrow x-1\text{”}, y \neq \overline{x})$ e $\text{PMF}(\text{“}x \leftarrow x+1 \mid y \leftarrow x-1\text{”}, y \neq \overline{x})$. Seria sensato encontrar em um programa real estas atribuições com tal pós-condição?

5. Repita o Exercício 3 do Capítulo 6, mas desta vez considerando que as atribuições são sequenciais, feitas na ordem de aparição (da esquerda para direita), ao invés de simultâneas.

8. Comando Alternativo

8.1 Descrição

O comando alternativo é aquele que permite que o fluxo de execução de um algoritmo possa continuar por diferentes sequências de comandos dependendo do valor de uma condição lógica. Definimos formalmente o comando alternativo conforme se segue.

Definição 6 — Comando Alternativo. A sintaxe do *comando alternativo* é descrita por

se C_1 **então** A_1 $\|$ C_2 **então** A_2 $\|$ $\cdots$ $\|$ C_k **então** A_k

ou, equivalentemente, por

se C_1 **então**
    A_1
$\|$ C_2 **então**
    A_2
$\cdots$
$\|$ C_k **então**
    A_k

onde C_1, C_2, ..., C_k são expressões lógicas e A_1, A_2, ..., A_k são algoritmos arbitrários. A expressão C_i é chamada de *condição guarda* de A_i, ou ainda dizemos que A_i é o *comando guardado* de C_i, para $i = 1, 2, \ldots, k$.

A semântica do comando alternativo é avaliar as condições guardas e, dentre as que resultarem em **V**, escolher uma condição C_i arbitrariamente. Em seguida, o fluxo de execução é conduzido para A_i e o comando termina após sua execução. Caso nenhuma condição guarda resulte em **V**, o comando alternativo termina imediatamente com um erro.

■ **Exemplo 8.1** No seguinte algoritmo, se, inicialmente,

- $x < 0$, o algoritmo termina com erro;
- $x = 0$, tanto o algoritmo A_1 quanto A_2 pode ser o executado;
- $0 < x < 5$, o algoritmo A_2 será o executado;
- $5 \leq x \leq 10$, tanto o algoritmo A_2 quanto A_3 pode ser o executado;
- $x > 10$, o algoritmo A_3 será o executado;

se $x = 0$ **então**
    A_1
$\|$ $0 \leq x \leq 10$ **então**
    A_2
$\|$ $x \geq 5$ **então**
    A_3

■

8.2 Correção

Verificando o Comando Alternativo

Para se proceder a verificação da correção do comando alternativo tal como apresentada no Capítulo 4, estabelecemos a seguir como se determinar a pré-condição mais fraca do comando.

Teorema 8.1 — PMF do Comando Alternativo. Sejam $\mathbf{Q}$ uma proposição e

$$A = \textbf{se } C_1 \textbf{ então } A_1 \ \| \ C_2 \textbf{ então } A_2 \ \| \ \cdots \ \| \ C_k \textbf{ então } A_k$$

um comando alternativo. Temos que

$$\begin{aligned} \text{PMF}(\text{“}A\text{”}, \mathbf{Q}) &= (\exists\, i \in [1..k] : C_i) \wedge (\forall\, i \in [1..k] : C_i \Rightarrow \text{PMF}(\text{“}A_i\text{”}, \mathbf{Q})) \\ &= (C_1 \vee C_2 \vee \cdots \vee C_k) \wedge \\ &\quad (C_1 \Rightarrow \text{PMF}(\text{“}A_1\text{”}, \mathbf{Q}) \wedge (C_2 \Rightarrow \text{PMF}(\text{“}A_2\text{”}, \mathbf{Q}) \wedge \cdots \wedge (C_k \Rightarrow \text{PMF}(\text{“}A_k\text{”}, \mathbf{Q}) \end{aligned}$$

A expressão da PMF acima pode ser simplificada caso os conjuntos de estados associados a $C_1, C_2, \ldots, C_k$ sejam dois a dois disjuntos, conforme mostrado a seguir.

Teorema 8.2 Sejam $C_1, C_2, \ldots, C_k$ predicados cujos conjuntos de estados associados são dois a dois disjuntos. Temos que

$$(C_1 \Rightarrow a_1) \wedge (C_2 \Rightarrow a_2) \wedge \cdots \wedge (C_k \Rightarrow a_k) = (C_1 \wedge a_1) \vee (C_2 \wedge a_2) \vee \cdots \vee (C_k \wedge a_k)$$

■ **Exemplo 8.2** Considere novamente o cálculo da PMF do Exemplo 4.7(9). Aplicando o cálculo da PMF do comando alternativo, e observando-se que os conjuntos de estados associados aos

predicados $(x \geq y)$ e $(x < y)$ são disjuntos, temos que

$$\begin{aligned}\textsc{Pmf}(\text{“}A_9\text{”}, \mathbf{Q_9}) &= ((x \geq y) \vee (x < y)) \wedge \\ &\quad ((x \geq y) \Rightarrow \textsc{Pmf}(\text{“}x \leftarrow x + a\text{”}, \mathbf{Q_9})) \wedge ((x < y) \Rightarrow \textsc{Pmf}(\text{“}x \leftarrow x - b\text{”}, \mathbf{Q_9})) \\ &= \mathbf{V} \wedge ((x \geq y) \wedge (x + a \geq 10) \vee ((x < y) \wedge (x - b \geq 10)) \\ &= (x \geq \max\{y, 10 - a\}) \vee (10 + b \leq x < y)\end{aligned}$$

■

Alternativamente, podemos conduzir a verificação da correção pelo Método #2, como apresentado a seguir.

Método 8.3 — Método #2. Sejam $C_1, C_2, \ldots, C_k$ expressões lógicas e $A_1, A_2, \ldots, A_k$ algoritmos. A tH

$$\{\ \mathbf{P}\ \}\ \textbf{se}\ C_1\ \textbf{então}\ A_1\ \|\ C_2\ \textbf{então}\ A_2\ \|\ \cdots\ \|\ C_k\ \textbf{então}\ A_k\ \{\ \mathbf{Q}\ \}$$

é válida se, e somente se,

1. $\mathbf{P} \Rightarrow C_1 \vee C_2 \vee \cdots \vee C_k$
2. a tH

   $$\{\ \mathbf{P} \wedge C_i\ \}\ A_i\ \{\ \mathbf{Q}\ \}$$

   é válida, para todo $i = 1, 2, \ldots, k$

■ **Exemplo 8.3** Considere verificar a tH

$\{\ \mathbf{V}\ \}$
se $x \geq 0$ **então**
  $y \leftarrow x$
$\|\ x \leq 0$ **então**
  $y \leftarrow -1 * x$
$\{\ y = |x|\ \}$

Pelo Método #2 de verificação de comandos alternativos, devemos verificar:

1. $\mathbf{V} \Rightarrow (x \geq 0) \vee (x < 0)$, e
2. a validade das tHs:
   (a) $\{\ \mathbf{V} \wedge (x \geq 0)\ \}\ y \leftarrow x\ \{\ y = |x|\ \}$
   (b) $\{\ \mathbf{V} \wedge (x \leq 0)\ \}\ y \leftarrow -1 * x\ \{\ y = |x|\ \}$

O item 1 se verifica, pois o consequente se reduz a $\mathbf{V}$. Para o item $2(a)$ e $2(b)$, podemos usar o Método #2 de verificação da atribuição. Logo, devemos verificar, respectivamente,

$$\begin{aligned}&(x \geq 0) \Rightarrow (y = |x|)^y_x \\ &= (x \geq 0) \Rightarrow (x = |x|)\end{aligned}$$

e

$$(x \leq 0) \Rightarrow (y = |x|)^{y}_{-1*x}$$
$$= (x \leq 0) \Rightarrow (-x = |x|)$$

e ambas as implicações são válidas. ∎

Elaborando o Comando Alternativo

A primeira estratégia para a elaboração de um comando alternativo pode ser descrita da seguinte maneira.

Método 8.4 — Método #3(a). Para elaborar um algoritmo A de modo que a tH

$$\{\ \mathbf{P}\ \}\ A\ \{\ \mathbf{Q}\ \}$$

seja válida, encontre um algoritmo A_1 que satisfaça $\mathbf{Q}$ para algum conjunto não vazio de entradas. Determine C_1 de forma que

$$\mathbf{P} \wedge C_1 \Rightarrow \text{PMF}(\text{“}A_1\text{”}, \mathbf{Q})$$

Repita o processo, obtendo novos pares (C_2, A_2), (C_3, A_3), ..., (C_k, A_k), até que

$$\mathbf{P} \Rightarrow C_1 \vee C_2 \vee \cdots \vee C_k$$

Neste caso,

$$A = \textbf{se } C_1 \textbf{ então } A_1 \ \|\ C_2 \textbf{ então } A_2 \ \|\ \cdots \ \|\ C_k \textbf{ então } A_k$$

é uma solução para o problema.

Note que, no processo acima, uma propriedade importante para C_1 de modo que

$$\mathbf{P} \wedge C_1 \Rightarrow \text{PMF}(\text{“}A_1\text{”}, \mathbf{Q})$$

é que C_1 seja de simples computação (onde “simples” significa que possui complexidade de tempo constante), pois será usado como guarda de um comando alternativo e, portanto, efetivamente computado. Se $\text{PMF}(\text{“}A_1\text{”}, \mathbf{Q})$ é de simples computação, então pode-se fazer $C_1 = \text{PMF}(\text{“}A_1\text{”}, \mathbf{Q})$. Caso contrário, outra expressão mais simples de ser computada, satisfazendo a implicação acima, deve ser procurada.

∎ **Exemplo 8.4** Considere determinar um algoritmo A que torne válida a tH

$$\{\ \mathbf{V}\ \}\ A\ \{\ \mathbf{Q} = (x = \min\{\overline{x}, \overline{y}\}) \wedge (y = \max\{\overline{x}, \overline{y}\})\ \}$$

Tomemos uma entrada arbitrária, digamos $x = 3$, $y = 2$. Um algoritmo A_1 que troque os valores de x, y é suficiente para que, ao executar A_1 sob tal entrada, a pós-condição seja satisfeita. Portanto,

$$A_1 = x, y \leftarrow y, x$$

$$\mathbf{Q}' = \text{PMF}(\text{“}A_1\text{”}, \mathbf{Q}) = \mathbf{Q}^{x,y}_{y,x} = (y = \min\{\overline{x}, \overline{y}\}) \wedge (x = \max\{\overline{x}, \overline{y}\})$$

Agora, precisamos determinar C_1 tal que

$$\mathbf{V} \wedge C_1 \Rightarrow \mathbf{Q}'$$

e não é difícil perceber que

$$C_1 = x \geq y$$

cumpre este objetivo. Como não se verifica

$$\mathbf{V} \Rightarrow C_1$$

o comando alternativo precisa de mais casos. Tomemos, então, um outro estado que não satisfaça C_1, digamos, $x = 2$, $y = 3$. Neste caso, constatamos que nenhuma alteração de estado é necessária para satisfazer a pós-condição. Em outras palavras, o comando $A_2 = \emptyset$ a partir do estado em questão satisfaz a pós-condição. Como

$$\mathbf{Q}'' = \textsc{Pmf}(\text{“}A_2\text{”}, \mathbf{Q}) = \mathbf{Q}$$

Para que

$$\mathbf{V} \wedge C_2 \Rightarrow \mathbf{Q}''$$

temos que

$$C_2 = x \leq y$$

cumpre este objetivo. Como

$$\mathbf{V} \Rightarrow C_1 \vee C_2$$

o algoritmo seguinte é uma solução para o problema:

$$A = \textbf{se } x \geq y \textbf{ então } x, y \leftarrow y, x \ \| \ x \leq y \textbf{ então } \emptyset$$

■

O próximo método formaliza uma segunda estratégia.

Método 8.5 — Método #3(b). Para elaborar um algoritmo A de modo que a tH

$$\{ \mathbf{P} \}\, A \,\{ \mathbf{Q} \}$$

seja válida, encontre um algoritmo A_1 que satisfaça $\mathbf{Q}$ para algum conjunto não vazio de entradas. Determine C_1 de forma que

$$\mathbf{P} \wedge C_1 \Rightarrow \textsc{Pmf}(\text{“}A_1\text{”}, \mathbf{Q})$$

Em seguida, encontre um algoritmo A_2 que torne válida a tH

$$\{ \mathbf{P} \wedge \neg C_1 \}\, A_2 \,\{ \mathbf{Q} \}$$

Neste caso,

$$A = \textbf{se } C_1 \textbf{ então } A_1 \ \| \ \neg C_1 \textbf{ então } A_2$$

é uma solução para o problema.

■ **Exemplo 8.5** Este exemplo se procede da mesma forma que o exemplo anterior, até a determinação do par (C_1, A_1). Contudo, continuamos o raciocínio da seguinte forma: determinamos

$$C_2 = \neg C_1 = x < y$$

e determinamos A_2 de modo a satisfazer

$$\{\ \mathbf{V} \wedge (x < y)\ \}\ A_2\ \{\ \mathbf{Q}\ \}$$

Neste caso, trivialmente percebemos que $A_2 = \emptyset$ resolve o problema. Assim, o algoritmo A procurado é

$$A = \textbf{se } x \geq y \textbf{ então } x, y \leftarrow y, x \ \|\ x < y \textbf{ então } \emptyset$$

■

8.3 Exercícios

1. Determine PMF("A", $\mathbf{Q}$) para os seguintes algoritmos A e pós-condições $\mathbf{Q}$.

	A	$\mathbf{Q}$
(a)	**se** $w \leq r$ **então** $r, q \leftarrow r - w, q + 1$ $\|$ $w < r$ **então** $\emptyset$	$(qw + r = x) \wedge (r \geq 0)$
(b)	**se** $a > b$ **então** $a \leftarrow a - b$ $\|$ $b > a$ **então** $b \leftarrow b - a$	$(a > 0) \wedge (b > 0)$
(c)	**se** $x \geq y$ **então** $x \leftarrow x - y$ $\|$ $x \leq y$ **então** $\emptyset$	$2x \leq y$
(d)	**se** $x \leq y$ **então** $z \leftarrow y$ $\|$ $y \leq x$ **então** $z \leftarrow x$	$z \geq 0$
(e)	**se** $x \geq 0$ **então** $y, x \leftarrow x, y$ $\|$ $x < 0$ **então** $\emptyset$	$(x < 0) \vee (y < 0) \Rightarrow (x < 0)$
(f)	**se** $x \geq 0$ **então** $y, x \leftarrow x, y$ $\|$ $x < 0$ **então** $\emptyset$	$x < 0$
(g)	**se** $x > 10$ **então** $y \leftarrow x/2$ $\|$ $x < 20$ **então** $y \leftarrow x - 100$	$y > 0$
(h)	**se** $x \geq y$ **então** $m \leftarrow x - y$ $\|$ $y \geq x$ **então** $m \leftarrow y - x$	$m > 0$

2. Para os problemas abaixo, aplique os métodos de elaboração do alternativo para construir os algoritmos propostos. Evidencie passo-a-passo a aplicação do método.
   (a) Uma variável x mantém o número de números pares compreendidos entre as posições 1 e k de vetor b. O objetivo é incrementar k, mantendo a propriedade enunciada de x.
   (b) Uma variável nn mantém o número de números negativos compreendido entre as posições 1 e k de um vetor b, e outra variável np mantém o número de números positivos dentre os mesmos elementos. O objetivo é incrementar k, mantendo a propriedade enunciada de nn e np.
   (c) Uma variável r mantém o resto da divisão de b por k. O objetivo é incrementar b, mantendo a propriedade enunciada de r sem refazer a divisão entre b e k.
3. Sejam a e b duas variáveis inteiras que representam o conjunto $\{a, a+1, \ldots, b\}$, que possui ao menos três valores e tais que o predicado

   $$\mathbf{P}(a,b) = (a^2 \leq n < b^2)$$

   seja satisfeito. É possível cortar o conjunto "ao meio", atribuindo ou à variável a ou à variável b o valor $\lfloor (a+b)/2 \rfloor$ de modo que $\mathbf{P}(a,b)$ seja mantido? Justifique tentando elaborar um algoritmo para fazê-lo.
4. Um aluno argumentou a seguinte estratégia para se elaborar um algoritmo A que torne

   $$\{\ \mathbf{P}\ \}\ A\ \{\ \mathbf{Q}\ \}$$

   uma tH válida com um comando alternativo: "Com o uso do Método #3(a) para se elaborar comandos alternativos, nem precisamos pensar muito que algoritmo resolve o problema. Basta elaborar qualquer algoritmo A (ou, quem sabe, sortear um aleatoriamente!) e protegê-lo com o guarda $\textsc{Pmf}(\text{"}A\text{"}, \mathbf{Q})$. Procedemos desta maneira iterativamente até que $\mathbf{P}$ implique na disjunção de todos os guardas.
   (a) Há algum erro nesta abordagem?
   (b) Há alguma limitação nesta abordagem?
5. Prove o Teorema 8.2.

9. Comando Iterativo

9.1 Descrição

O comando iterativo permite que determinado algoritmo possa ser repetido, isto é, executado em composição com si próprio, até que determinação condição seja satisfeita. O uso de comando iterativo é crucial em problemas mais complexos. Com efeito, é praticamente necessário que algoritmos não triviais formulados sem chamadas a funções consistam de um comando iterativo. Definimos formalmente o comando iterativo conforme se segue.

Definição 7 — Comando Iterativo. A sintaxe do *comando iterativo* é descrita por

INI; **enquanto** *C* **faça** *A*

ou, equivalentemente, por

INI
enquanto *C* **faça**
    A

onde *INI* e *A* correspondem a algoritmos, *INI* chamado de *inicialização* e *A* de *bloco da repetição*, e *C* corresponde a uma expressão lógica, chamada de *condição de parada*.

A semântica do comando iterativo é repetir o algoritmo *A* até que *C* seja igual a **V**; caso este já seja o caso após a execução de *INI*, nenhuma execução de *A* é feita. O algoritmo *INI* permite que o estado seja preparado para a repetição; em geral, consiste de apenas uma atribuição.

O próximo exemplo ilustra um algoritmo que consiste de um comando iterativo.

■ **Exemplo 9.1** A seguinte tH é válida:

$\{ N = 4 \}$
$i, j, s \leftarrow N, 1, 0$
enquanto $i > 0$ **faça**
    $i, j, s \leftarrow i - 1, j + 2, s + j$
$\{ s = 16 \}$

Verificar a correção deste algoritmo é trivial pois, dado que a entrada é única, basta acompanhar a execução do algoritmo para tal entrada e verificar a validade da pós-condição. Uma tarefa um pouco mais complicada, contudo, seria verificar a correção deste mesmo algoritmo sob uma especificação mais geral, como a seguir.

$\{ N \geq 0 \}$
$i, j, s \leftarrow N, 1, 0$
enquanto $i > 0$ **faça**
    $i, j, s \leftarrow i - 1, j + 2, s + j$
$\{ s = N^2 \}$

A seção seguinte, dedicada aos métodos de correção do comando iterativo, proverá meios para se argumentar a correção desta tH. ■

9.2 Correção

Verificando o Comando Iterativo

Como foi feito para os comandos anteriores, discutamos a obtenção da PMF de comandos iterativos. Em essência, um comando iterativo é uma composição da inicialização com a o bloco de repetição, este último em um número variável de vezes. Assim, tal PMF pode ser calculada como apresentado no estudo do comando de composição. Desta maneira, evidenciamos a seguir como se determinar a pré-condição mais fraca da repetição.

Teorema 9.1 — PMF do Comando Iterativo. Sejam $\mathbf{Q}$ uma proposição e

$$A = INI; A_R$$

um comando iterativo, onde

$$A_R = \textbf{enquanto } C \textbf{ faça } R$$

Temos que

$$\text{PMF}(\text{“}A\text{”}, \mathbf{Q}) = \text{PMF}(\text{“}INI\text{”}, \text{PMF}(\text{“}A_R\text{”}, \mathbf{Q}))$$

e

$$\text{PMF}(\text{“}A_R\text{”}, \mathbf{Q}) = (\exists\, k \in \mathbb{N} : \mathbf{P}(A_R, \mathbf{Q}, k))$$

onde $\mathbf{P}(A_R, \mathbf{Q}, k)$ é a PMF do algoritmo A_R com respeito a pós-condição $\mathbf{Q}$ para que A_R seja

executado em exatamente k vezes, que pode ser determinada por

$$\mathbf{P}(A_R, \mathbf{Q}, k) = \begin{cases} \neg C \wedge \mathbf{Q} & \text{se } k = 0 \\ C \wedge \textsc{Pmf}(\text{“}R\text{”}, \mathbf{P}(A_R, \mathbf{Q}, k-1)) & \text{se } k \geq 1 \end{cases}$$

■ **Exemplo 9.2** Considere novamente o cálculo da PMF do Exemplo 4.7(10). Para

$$\begin{aligned} \mathbf{Q} &= (j \geq 10) \\ A_R = \quad & \textbf{enquanto } i \leq 3 \textbf{ faça } A_R \\ R &= i, j \leftarrow i+1, j+1 \end{aligned}$$

temos que

$$\begin{aligned} \mathbf{P}(A_R, \mathbf{Q}, 0) &= \neg(i \leq 3) \wedge (j \geq 10) \\ &= (i > 3) \wedge (j \geq 10) \\ \mathbf{P}(A_R, \mathbf{Q}, 1) &= (i \leq 3) \wedge \textsc{Pmf}(\text{“}R\text{”}, \mathbf{P}(A_R, \mathbf{Q}, 0)) \\ &= (i \leq 3) \wedge \textsc{Pmf}(\text{“}R\text{”}, (i > 3) \wedge (j \geq 10)) \\ &= (i \leq 3) \wedge (i > 2) \wedge (j \geq 9) \\ &= (i = 3) \wedge (j \geq 9) \\ \mathbf{P}(A_R, \mathbf{Q}, 2) &= (i \leq 3) \wedge \textsc{Pmf}(\text{“}R\text{”}, \mathbf{P}(A_R, \mathbf{Q}, 1)) \\ &= (i \leq 3) \wedge \textsc{Pmf}(\text{“}R\text{”}, (i = 3) \wedge (j \geq 9)) \\ &= (i \leq 3) \wedge (i = 2) \wedge (j \geq 8) \\ &= (i = 2) \wedge (j \geq 8) \end{aligned}$$

ou, em geral para $k \geq 1$,

$$\begin{aligned} \mathbf{P}(A_R, \mathbf{Q}, k) &= (i = 4-k) \wedge (j \geq 10-k) \\ &= (i = 4-k) \wedge (j \geq 6+i) \end{aligned}$$

e, portanto,

$$\begin{aligned} \textsc{Pmf}(\text{“}A_R\text{”}, \mathbf{Q}) &= (\exists\, k \in \mathbb{N} : \mathbf{P}(A_R, \mathbf{Q}, k)) \\ &= \mathbf{P}(A_R, \mathbf{Q}, 0) \vee (\exists\, k \in \mathbb{N} : k \geq 1 \wedge \mathbf{P}(A_R, \mathbf{Q}, k)) \\ &= (i > 3) \wedge (j \geq 10) \vee (i \leq 3) \wedge (j \geq 6+i) \end{aligned}$$

■

Note que a determinação da PMF deste comando é demasiadamente mais complexa que aquela dos comandos anteriores. Além disso, a prova da validade de

$$\mathbf{P} \Rightarrow \textsc{Pmf}(\text{“}A\text{”}, \mathbf{Q})$$

na verificação de uma tH $\{\ \mathbf{P}\ \}\ A\ \{\ \mathbf{Q}\ \}$ envolvendo um comando iterativo A é um problema em geral indecidível. Isto se deve ao fato que, se a correção de um algoritmo puder ser verificada, isto implicaria reconhecer as entradas para as quais este algoritmo pára, um problema clássico de indecidibilidade. Assim, o Método #1 não será o método prático para a verificação da correção do comando iterativo. Em vez disso, o Método #2 será empregado para a verificação da correção, como apresentado a seguir.

Método 9.2 — Método #2. Seja

$\{\ \mathbf{P}\ \}$
INI
$\{\ \mathbf{INV}\ \}$ // $t =$ ... (definição de t)
enquanto C **faça**
    A
$\{\ \mathbf{Q}\ \}$

uma tH envolvendo um comando iterativo, onde *INI* e A são algoritmos, C é uma expressão lógica, **P**, **INV**, e **Q** são predicados, e t é uma expressão das variáveis do algoritmo que resulte em um inteiro. Tal tH é válida sob correção *parcial* se

1. a tH $\{\ \mathbf{P}\ \}$ *INI* $\{\ \mathbf{INV}\ \}$ é válida;
2. $\mathbf{INV} \wedge \neg C \Rightarrow \mathbf{Q}$
3. a tH $\{\ \mathbf{INV} \wedge C\ \}$ A $\{\ \mathbf{INV}\ \}$ é válida.

e válida sob correção *total* se, além de atender as propriedades 1, 2 e 3, atende às seguintes propriedades

4. a tH $\{\ t = \overline{\alpha}\ \}$ A $\{\ t < \overline{\alpha}\ \}$ é válida;
5. $\mathbf{INV} \wedge C \Rightarrow (t > 0)$

■ **Exemplo 9.3** Considere verificar a tH

$\{\ \mathbf{P} = (n \geq 0)\ \}$
$s, i \leftarrow 0, 0$
$\{\ \mathbf{INV} = (s = (\sum x \in [1..i] : B[x])) \wedge (i \leq n)\ \}$ // $t = n - i$
enquanto $i \neq n$ **faça**
    $s, i \leftarrow s + B[i+1], i+1$
$\{\ \mathbf{Q} = (s = (\sum x \in [1..n] : B[x]))\ \}$

Pelo Método #2 de verificação de comandos iterativos, devemos verificar:

1. $\{\ \mathbf{P}\ \}$ $s, i \leftarrow 0, 0$ $\{\ \mathbf{INV}\ \}$:
   Utilizando o Método #2 do comando de atribuição, verificamos a validade de

$$\begin{aligned} & \mathbf{P} \Rightarrow \mathbf{Q}^{s,i}_{0,0} \\ = \ & \mathbf{P} \Rightarrow (0 = (\sum x \in [1..0] : B[x])) \\ = \ & \mathbf{P} \Rightarrow \mathbf{V} \quad \checkmark \end{aligned}$$

2. $\mathbf{INV} \wedge \neg(i \neq n) \Rightarrow \mathbf{Q}$
   $= \mathbf{INV} \wedge (i = n) \Rightarrow \mathbf{Q}$ ✓;
3. $\{\ \mathbf{INV} \wedge (i = n)\ \}$ $s, i \leftarrow s + B[i+1], i+1$ $\{\ \mathbf{INV}\ \}$:
   Utilizando o Método #2 do comando de atribuição, verificamos a validade de

$$\begin{aligned} & \mathbf{INV} \wedge (i \neq n) \Rightarrow \mathbf{INV}^{s,i}_{s+B[i+1],i+1} \\ = \ & \mathbf{INV} \wedge (i \neq n) \Rightarrow (s + B[i+1] = (\sum x \in [1..i+1] : B[x])) \wedge (i < n) \quad \checkmark \end{aligned}$$

4. $\{\ n-i=\overline{\alpha}\ \}\ s, i \leftarrow s+B[i+1], i+1\ \{\ n-i<\overline{\alpha}\ \}$:
   Novamente, pelo Método #2 de verificação da atribuição,

$$(n-i=\overline{\alpha}) \Rightarrow (n-i<\overline{\alpha})^{s,i}_{s+B[i+1],i+1}$$
$$= (n-i=\overline{\alpha}) \Rightarrow (n-i-1<\overline{\alpha}) \ \checkmark$$

5. $\mathbf{INV} \wedge (i \neq n) \Rightarrow (n-i>0)$ ✓

Portanto, a tH é válida sob correção total. ∎

Elaborando o Comando Iterativo

Métodos para a elaboração do comando iterativo são apresentados a seguir. O primeiro trata de correção parcial, enquanto o segundo de correção total.

Método 9.3 — Método #3 (correção parcial). Para elaborar um comando iterativo A de modo que a tH

$$\{\ \mathbf{P}\ \}\ A\ \{\ \mathbf{Q}\ \}$$

seja válida sob correção parcial, encontre predicados lógicos **INV** e C de modo que

$$\mathbf{INV} \wedge C \Rightarrow \mathbf{Q}$$

e tais que:

1. haja um algoritmo *INI* para obter um estado satisfazendo **INV**;
2. C seja simples de computar.

Neste caso, definindo-se

$$A = INI;\ \textbf{enquanto}\ C\ \textbf{faça}\ R$$

é uma solução para o problema, onde R é um algoritmo que deve ser elaborado tal que a tH

$$\{\ \mathbf{INV} \wedge C\ \}\ R\ \{\ \mathbf{INV}\ \}$$

seja válida.

Neste método, o passo de obter predicados **INV** e C tais que $\mathbf{INV} \wedge C \Rightarrow \mathbf{Q}$ é o crucial e para o qual existe algumas técnicas padrão, apresentadas na Seção 9.3.

∎ **Exemplo 9.4** Considere determinar um algoritmo A que torne válida a tH

$$\{\ \mathbf{P} = n \geq 1\ \}\ A\ \{\ \mathbf{Q} = (s = (\sum x \in [1..n] : x))\ \}$$

Fazendo-se

$$\mathbf{INV} = (s = (\sum x \in [1..i] : x) \wedge (i \leq n))$$
$$\neg C = (i = n)$$

temos que $\mathbf{INV} \wedge \neg C \Rightarrow \mathbf{Q}$. Além disso, **INV** é facilmente satisfeito por

$$INI = i, s \leftarrow 1, 1$$

(Verifique.) Portanto, temos que

$$C = (i \neq n)$$

e o algoritmo $A = INI$; **enquanto** C **faça** R é solução para o problema, onde $R = \emptyset$. ∎

No exemplo anterior, o uso do bloco de repetição $R = \emptyset$ é inaceitável na prática, embora o algoritmo seja válido sob correção parcial. O problema com R é que ele não progride o estado de modo a garantir que $\neg C$ seja eventualmente satisfeito, caso não seja inicialmente. Posterguemos até o próximo exemplo para elaborar uma segunda proposta para R, depois de apresentar o método que trata a correção total.

Método 9.4 — Método #3 (correção total). Para elaborar um comando iterativo A de modo que a tH

$$\{\ \mathbf{P}\ \}\ A\ \{\ \mathbf{Q}\ \}$$

seja válida sob correção total, proceda como indicado no Método 9.3 para obter um algoritmo

$$A = \ INI\ \{\ \mathbf{INV}\ \}\ \textbf{enquanto}\ C\ \textbf{faça}\ R$$

que esteja parcialmente correto e, no processo, tenha a preocupação adicional que R progrida em relação ao objetivo de eventualmente satisfazer $\neg C$. Para tal, defina uma expressão inteira t que mensure o quão "distante" está C de se tornar falso, e verifique se é válido que

- o valor de t decresça a cada iteração;
- sendo verdade tanto o invariante quanto C, vale $t > 0$.

∎ **Exemplo 9.5** Retornemos ao Exemplo 9.4, revisando o algoritmo R de modo a obter correção total. No algoritmo produzido, inicialmente $i = 1$ e para que $\neg C = (i = n)$. Assim, uma expressão que mede o quanto falta para $\neg C$ ser satisfeito seria

$$t = n - i$$

Para decrescer t a cada iteração, podemos incrementar i a cada iteração. Para garantir que isto sempre ocorra, decompomos R em

$$R = R'\ \{\ \mathbf{INV}'\ \}\ i \leftarrow i + 1$$

Fazendo-se

$$\mathbf{INV}' = \textsc{Pmf}(\text{“}i \leftarrow i+1\text{”}, \mathbf{INV}) = (s = (\sum x \in [1..i+1] : x) \wedge (i < n))$$

temos que

$$R' = s \leftarrow s + i + 1$$

torna válida a tH $\{\ \mathbf{INV} \wedge C\ \}\ R'\ \{\ \mathbf{INV}'\ \}$ (verifique), resultando no seguinte algoritmo:

$\{\ \mathbf{P} = (n \geq 1)\ \}$
$i, s \leftarrow 1, 1$
$\{\ \mathbf{INV} = (s = (\sum x \in [1..i] : x) \wedge (i \leq n)\ \}$ // $t = n - i$
enquanto $i \neq n$ **faça**
    $s \leftarrow s + i + 1$
    $i \leftarrow i + 1$
$\{\ \mathbf{Q} = (s = (\sum x \in [1..n] : x))\ \}$

■

9.3 Elaborando o Invariante

Para elaborar um comando iterativo A de modo que

$$\{\ \mathbf{P}\ \}\ A\ \{\ \mathbf{Q}\ \}$$

seja uma tH válida, é necessário encontrar predicados **INV** e C tais que

$$\mathbf{INV} \wedge C \Rightarrow \mathbf{Q}$$

como foi apresentado no Método 9.3. Em grande parte, a escolha destes predicados é a essência do comando iterativo, no sentido que o desenvolvimento restante do comando decorre quase que por consequência. Esta seção aborda um conjunto conhecido de técnicas para a elaboração de tais predicados.

De uma maneira geral, os métodos a seguir possuem uma estratégia geral. Cada um deles produz um invariante a partir de um enfraquecimento da pós-condição (isto é, pela ampliação do conjunto de estados que a satisfaz). A ideia é que, caso satisfazer a pós-condição a partir de um estado satisfazendo a pré-condição seja uma tarefa não trivial, o invariante é enfraquecido o suficiente para que:

- seja possível, de formal trivial, satisfazê-lo a partir da pré-condição (papel da inicialização), e
- seja possível, iteração a iteração, fortalecer o invariante (papéis da manutenção do invariante e do decréscimo do valor da expressão t) até que a pós-condição seja satisfeita (papel da condição de parada).

Eliminação de uma Conjunção

No caso em que **Q** é expresso na forma de uma conjunção, pode-se investigar se uma partição qualquer dos termos envolvidos na conjunção pode servir para estabelecer **INV** e C, como se segue.

Método 9.5 — Elimação de Conjunção. No caso em que **Q** é da forma

$$\mathbf{Q} = \alpha_1 \wedge \alpha_2 \wedge \cdots \wedge \alpha_k$$

defina, para algum $i = 1, 2, \ldots, k-1$, os predicados

$$\mathbf{INV} = \alpha_1 \wedge \alpha_2 \wedge \cdots \wedge \alpha_i$$

$$C = \neg(\alpha_{i+1} \wedge \alpha_{i+2} \wedge \cdots \wedge \alpha_k)$$

■ **Exemplo 9.6** Considere elaborar um comando iterativo que torne válida a tH

$$\{\ n \geq 0\ \}\ A\ \{\ r^2 \leq n < (r+1)^2\ \}$$

A pós-condição possui uma conjunção implícita, a saber, $(r^2 \leq n) \wedge (n < (r+1)^2)$. Aplicando a eliminação de conjunção, podemos fazer

$$\mathbf{INV} = (r^2 \leq n)$$

$$C = \neg(n < (r+1)^2) = (n \geq (r+1)^2)$$

ou

$$\mathbf{INV} = (n < (r+1)^2)$$

$$C = \neg(r^2 \leq n) = (r^2 > n)$$

Ambos invariantes são possíveis e derivam distintos algoritmos. Como exercício, termine de derivar os respectivos algoritmos. (Qual possui a melhor complexidade de tempo?) ∎

Substituição de uma Constante por uma Variável

Esta técnica consiste em substituir uma constante da pós-condição (um número, por exemplo), por uma variável nova no algoritmo. Além disso, estabelece-se o intervalo de valores que esta variável pode assumir, no qual deve estar incluído o valor da constante substituída. A expressão do invariante consiste desta pós-condição mais geral, que possui uma variável no lugar de uma constante, e a condição de parada consiste em testar se a variável é igual à constante substituída.

Método 9.6 — Substituição de uma Constante por uma Variável. No caso em que $\mathbf{Q}$ é da forma

$$\mathbf{Q} = \ldots c \ldots$$

onde c denota uma constante numérica que aparece como parte da expressão de $\mathbf{Q}$, defina o invariante como sendo a própria expressão de $\mathbf{Q}$, substituindo-se c por x, onde x é uma variável nova. Em seguida, acrescente uma conjunção para estabelecer um domínio S para x. Além disso, a repetição deve ocorrer até que $x = c$, isto é, enquanto $x \neq c$. Portanto,

$$\mathbf{INV} = (\ldots x \ldots) \wedge (x \in S)$$

$$C = (x \neq c)$$

∎ **Exemplo 9.7** Considere elaborar um comando iterativo que torne válida a tH

$$\{\, n \geq 0 \,\}\ A \left\{ \underline{s} = \ B \ \overset{1}{\underset{}{\boxed{\Sigma}}}\overset{n}{\underset{}{\boxed{\ }}} \right\}$$

A pós-condição especifica que apenas a variável s deve ter seu valor alterado por A, isto é, B e n são constantes com respeito a A. Além disso, há uma outra constante envolvida na pós-condição, que é o 1, início do intervalo em que o somatório atua. Aplicando a substituição de uma constante por uma variável, podemos produzir

$$\mathbf{INV} = (\underline{s} = \ B \ \overset{1}{\underset{}{\boxed{\Sigma}}}\overset{i}{\underset{}{\boxed{\ }}}\boxed{\ }\ ) \wedge (0 \leq \underline{i} \leq n)$$

$$C = \neg(i = n) = (i \neq n)$$

ou

$$\mathbf{INV} = (\underline{s} = \ B \ \boxed{\ }\overset{i}{\underset{}{\boxed{\Sigma}}}\overset{n}{\underset{}{\boxed{\ }}}\ ) \wedge (1 \leq \underline{i} \leq n+1)$$

$$C = \neg(i = 1) = (i \neq 1)$$

Ambos invariantes são possíveis e derivam distintos algoritmos. Como exercício, termine de derivar os respectivos algoritmos. ∎

Ampliando o Domínio de uma Variável

O próximo método se assemelha ao anterior, no sentido que consiste de permitir que uma certa grandeza possa assumir mais valores do que inicialmente a pós-condição especifica. Enquanto no método anterior tal grandeza é uma constante, neste é uma variável.

A técnica se caracteriza por obter o invariante a partir da pós-condição, pela ampliação do conjunto S dos valores que determinada variável pode ter. A ideia é tornar possível estabelecer o invariante por um algoritmo de inicialização, e a condição de parada ocorre quando o valor da variável está em S, estabelecendo a pós-condição.

Método 9.7 — Ampliando o Domínio de uma Variável. No caso em que $\mathbf{Q}$ é da forma

$$\mathbf{Q} = \ldots \wedge (x \in S)$$

onde x denota uma variável, defina o invariante como, para algum conjunto $S' \neq \emptyset$,

$$\mathbf{INV} = \ldots \wedge (x \in S \cup S')$$

$$C = (x \notin S)$$

■ **Exemplo 9.8 — Busca Linear.** Considere elaborar um comando iterativo que torne válida a tH

$$\left\{ B \;\boxed{\begin{array}{c}1 \qquad\qquad\qquad\qquad n\\ \exists\, i : B[i] = x\end{array}} \right\} A \left\{ B \;\boxed{\begin{array}{ccccccc}1 & & & & \underline{i} & & n\\ & & & & x & & \end{array}} \right\}$$

Note que i deve apontar para uma posição que contenha um valor x. Dentre as possíveis, digamos que escolhemos encontrar a primeira. Portanto, o objetivo é fazer o valor i ser igual à primeira posição $\overline{p}$ que contenha o valor x. Assim, revisamos a especificação para a seguinte:

$$\left\{ B \;\boxed{\begin{array}{cccc}1 & & \overline{p} & \\ \neq x & x & & \end{array}} \right\} A \; \{\, \underline{i} = \overline{p} \,\}$$

Assim, i admite um único valor na pós-condição. Usando o método de ampliar o domínio desta variável, elaboramos

$$\mathbf{INV} = (1 \leq i \leq \overline{p})$$

$$C = \neg(i = \overline{p}) = (i \neq \overline{p})$$

Note, contudo, que a expressão C não passa no teste de ter uma computação simples. Logo, precisamos reescrevê-la para uma condição equivalente mas que tenha esta propriedade. Recorrendo-se à definição de $\overline{p}$, e lembrando que **INV** vale quando C é avaliado, revisamos C para

$$C = (B[i] \neq x)$$

Como exercício, termine de derivar o algoritmo. ■

Combinando Pré- e Pós-Condição

Todas os métodos de elaboração de invariante apresentados anteriormente se baseiam na pós-condição e ignoram em grande parte a pré-condição. O próximo método ilustra o caso em que a pré-condição deve ser , ao menos parcialmente, considerada no invariante.

Método 9.8 — Combinando Pré- e Pós-Condição. No caso em que $\mathbf{Q}$ é da forma

$$\mathbf{P} = (\forall\, x \in S : \mathbf{R}(x))$$

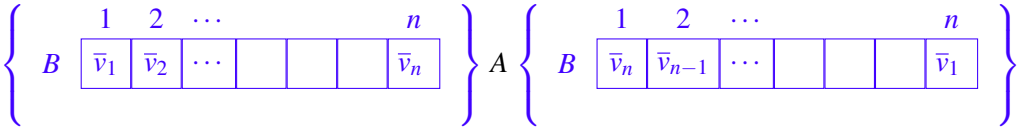
$$\mathbf{Q} = (\forall\, x \in S : \mathbf{S}(x))$$

onde $\mathbf{R}(x), \mathbf{S}(x)$ denotam predicados, fazemos

$$\mathbf{INV} = (\forall\, x \in S \setminus S' : \mathbf{R}(x)) \wedge (\forall\, x \in S' : \mathbf{S}(x))$$

$$C = (S' \neq S)$$

■ **Exemplo 9.9 — Inversão de Vetor.** Considere elaborar um comando iterativo que torne válida a tH

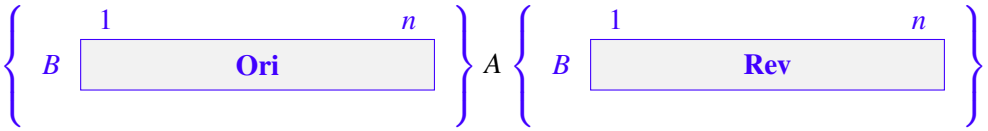


Em outras palavras, que seja revertido de posição os n primeiros elementos do vetor B. Alternativamente, poderíamos definir os predicados:

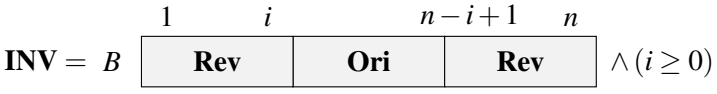
$$\mathbf{Ori}(B, i, j) = (\forall\, x \in [i..j] : B[x] = \bar{v}_x)$$

$$\mathbf{Rev}(B, i, j) = (\forall\, x \in [i..j] : B[x] = \bar{v}_{n-x+1})$$

e, assim, reescrever a tH como



Utilizando o método de combinar pré- e pós-condição, o invariante deve especificar que os i primeiros elementos de B já estão com os elementos corretos, enquanto os demais estão ainda em suas posições iniciais. Em uma segunda análise, constatamos que para não perder os primeiros i elementos originais, é conveniente afirmar também que os últimos i elementos de B já estão com os elementos corretos. Assim, derivamos o invariante e a condição de parada:



$$\wedge\, (i \geq 0)$$

$$C = \neg(i \geq n - i) = (i < n - i)$$

Como exercício, elabore o restante do algoritmo. ■

9.4 Exercícios

(nos slides)

Parte III – Elaboração Prática de Algoritmos

Introdução

Nesta parte, toda a teoria apresentada na Parte II será aplicada a diversos problemas. Estas aplicações serão apresentadas segundo uma estrutura fixa, a saber:

1. **Descrição:** A descrição do problema em linguagem natural, conforme ela normalmente é apresentada na vida real;
2. **Desenvolvendo o Algoritmo:** A aplicação do método formal, onde algoritmo e prova de correção vão sendo desenvolvidas ao mesmo tempo;
3. **Descrição do Algoritmo Desenvolvido:** Como no método de correção os comandos do algoritmo são desenvolvidos entremeado com a prova de correção, por conveniência, nesta seção uma descrição completa do algoritmo é apresentada;
4. **Considerações sobre Eficiência:** Um atributo importante de todo algoritmo, além de sua correção, é sua eficiência de tempo e espaço associadas. Embora este não seja o foco deste material, dedica-se esta seção a analisar este importante tema que não poderia ter sido deixado de fora de uma obra sobre algoritmos;
5. **Exercícios:** Em geral, durante o desenvolvimento, diversas decisões foram tomadas que poderiam ter sido diferentes, que produziriam outras soluções. Os exercícios exploram estas outras possibilidades, além de observar nuances da solução apresentada. A dedicação aos exercícios é de suma importância para se ter uma visão completa do processo de desenvolvimento.

10. Bandeira Nacional Holandesa

10.1 Descrição

Dados natural n, um vetor $B[1..n]$ onde cada elemento é igual a 1, 2 ou 3, elabore um algoritmo que ordene $B[1..n]$. Como exemplo, se

$$n = 10\ ,\ B = [2,1,3,1,1,2,3,2,1,3]$$

a modificação de B de modo que

$$B = [1,1,1,1,2,2,2,3,3,3]$$

é aquela que satisfaz ao problema. Este problema foi introduzido com o nome de *bandeira nacional holandesa*. A motivação para o nome veio da forma como o problema foi originalmente descrito. Nela, os elementos do vetor, ao invés de números, são as cores vermelho (correspondendo a 1), branco (a 2) e vermelho (a 3). Ao ordenar, e representando o vetor graficamente de cima para baixo (ao invés da forma usual esquerda para direita), veria-se as três listas da bandeira nacional da holanda, em suas respectivas cores. No entanto, esta lembrança à bandeira é um tanto quanto fraca, uma vez que não há garantia que as faixas tenham a mesma largura (isto é, que B inicialmente possua quantidades iguais ou similares dos elementos 1, 2 e 3), ou mesmo que haja todas as três faixas! A falta da cor azul, por exemplo, transformaria a interpretação do vetor final na bandeira da Indonésia! Assim, o nome do problema foi mantido pelo seu caráter histórico.

Por fim, este problema também inclui uma restrição em relação aos algoritmos. Deseja-se que os elementos de B sejam permutados comparando-se seus valores em pares e, eventualmente, trocando-os de posição. Assim, soluções que contem a quantidade de ocorrências de cada elemento para depois alterar B de modo a produzir estas mesmas quantidades, mas de maneira ordenada, estão descartadas.

Desenvolva seu algoritmo. Ao final, compare com aquele que será desenvolvido a seguir.

10.2 Desenvolvendo o Algoritmo

Em termos formais, procuramos elaborar um algoritmo A tal que a tH

$$\{\ \mathbf{P} = (n \geq 0)\ \}\ A\ \left\{\ \mathbf{Q} = \underline{B}\ \begin{array}{|c|c|c|}\hline \overset{1}{} & & \overset{n}{} \\ \hline = 1 & = 2 & = 3 \\ \hline \end{array}\ \right\}$$

seja válida. Um comando iterativo será empregado para se elaborar A, ou seja,

$$A = \mathit{INI}\ \{\ \mathbf{INV}\ \}\ \textbf{enquanto}\ C\ \textbf{faça}\ R;$$

Será aplicado o método de substituir as constante que indicam os pontos onde se começam os elementos 2 e 3, além da constante n, por variáveis. Assim, deriva-se o invariante

$$\mathbf{INV} = B\ \begin{array}{|c|c|c|c|c|c|}\hline \overset{1}{} & \leq & \overset{d}{} & \leq & \overset{t}{} & \leq \overset{f}{} \quad \overset{n}{} \\ \hline = 1 & & = 2 & & = 3 & \\ \hline \end{array}\ \wedge (f \leq n+1)$$

Para tal invariante, a condição de parada e o comando de inicialização são, respectivamente,

$$C = \neg(f = n+1) = (f \neq n+1)$$

$$\mathit{INI} = d, t, f \leftarrow 1, 1, 1$$

Em seguida, é necessário elaborar um algoritmo R que progrida em relação à terminação, enquanto mantém o invariante. Comparando-se a inicialização com a condição de parada, propomos

$$R = R'\ ;\ f \leftarrow f+1$$

e, portanto, devemos agora desenvolver R' de modo que a tH

$$\{\ \mathbf{INV} \wedge C\ \}\ R'\ \left\{\ \mathbf{INV}^{f}_{f+1} = B\ \begin{array}{|c|c|c|c|c|c|}\hline \overset{1}{} & \leq & \overset{d}{} & \leq & \overset{t}{} & \leq f+1 \quad \overset{n}{} \\ \hline = 1 & & = 2 & & = 3 & \\ \hline \end{array}\ \wedge (f \leq n)\ \right\}$$

seja válida Parece natural investigar o conteúdo da posição $B[f]$ e, de acordo com seu conteúdo, alargar o respectivo grupo dos elementos iguais a 1, 2 ou 3. Assim, elaboramos

$$R = \textbf{se}\ B[f] = 1\ \textbf{então}\ R_1\ \|\ B[f] = 2\ \textbf{então}\ R_2\ \|\ B[f] = 3\ \textbf{então}\ R_3$$

e procuramos elaborar algoritmos R_1, R_2, R_3 para cada caso. Se $B[f] = 3$, é trivial verificar que

$$R_3 = \emptyset$$

é suficiente. Se $B[f] = 2$, então pode-se resolver o problema com

$$R_2 = B[t], B[f], t \leftarrow B[f], B[t], t+1$$

Finalmente, se $B[f] = 1$, então pode-se resolver o problema com

$$R_1 = B[d], B[t], B[f], d, t \leftarrow B[f], B[d], B[t], d+1, t+1$$

completando o algoritmo.

10.3 Descrição do Algoritmo Desenvolvido

$\{\ \mathbf{P} = (n \geq 0)\ \}$
$d, t, f \leftarrow 1, 1, 1$

$$\left\{ \mathbf{INV} = \begin{array}{c} \begin{array}{ccccccccc} 1 & \leq & d & & \leq & t & \leq & f & \quad & & n \end{array} \\ \begin{array}{|c|c|c|c|c|c|} \hline = 1 & & = 2 & & = 3 & \quad & \quad & \quad \\ \hline \end{array} \end{array} \wedge (f \leq n+1) \right\} \ //\ t = (n+1) - i$$

enquanto $f \neq n+1$ **faça**
  se $B[f] = 1$ **então**
    $B[d], B[t], B[f], d, t \leftarrow B[f], B[d], B[t], d+1, t+1$
  $\|$ $B[f] = 2$ **então**
    $B[t], B[f], t \leftarrow B[f], B[t], t+1$
  $\|$ $B[f] = 3$ **então**
    $\emptyset$
  $f \leftarrow f+1$

$$\left\{ \mathbf{Q} = \begin{array}{c} \begin{array}{ccccc} 1 & & & & n \end{array} \\ \begin{array}{|c|c|c|} \hline = 1 & = 2 & = 3 \\ \hline \end{array} \end{array} \right\}$$

10.4 Considerações sobre Eficiência

A complexidade de tempo do algoritmo é de $\Theta(n)$. Note que empregar um algoritmo de ordenação por comparação geral, isto é, que assuma que os elementos de B são inteiros arbitrários, conduz a uma solução de tempo $\Theta(n \log n)$.

10.5 Observações Importantes

■ **Observação 10.1** Na solução, tanto a pós-condição como o invariante empregaram a notação gráfica. Lembre-se que, dadas as convenções estabelecidas para tal fim no Capítulo 3, esta é uma notação formal e pode ser transformada para o equivalente em notação textual. Por exemplo, a pós-condição poderia ter sido escrita como

$$\{\ \mathbf{Q} = (\forall\, i \in [1..n) : B[i] \leq B[i+1])\ \}$$

e o invariante como

$$\begin{aligned} \mathbf{INV} = {} & (\forall\, i \in [1..d) : B[i] = 1) \wedge (\forall\, i \in [d..t) : B[i] = 2) \wedge \\ & (\forall\, i \in [t..f) : B[i] = 3) \wedge (1 \leq d \leq t \leq f \leq n+1) \end{aligned}$$

Mas note como, quando bem empregada, a notação gráfica é de longe a melhor forma de raciocinar sobre o algoritmo. Isto foi particularmente verdade no passo de decidir cada um dos algoritmos R_1, R_2, R_3.

■ **Observação 10.2** O leitor atento deve ter percebido que à pós-condição falta, rigorosamente falando, o termo

$$\mathbf{Q} = \ldots \wedge \mathbf{Perm}(B, \overline{B}, n)$$

onde $\mathbf{Perm}(X, Y, n)$ é um predicado que afirma que os primeiros n elementos do vetor X é uma permutação daqueles de Y. No entanto, note que, dada a restrição do problema de que os elementos de B só podem ser permutados, através da comparação de seus valores em pares e, eventualmente, trocando-os de posição, tal predicado ausente seria trivialmente satisfeito. Sendo assim, opta-se por deixar na pós-condição *apenas as afirmações que caracterizam os algoritmos corretos dentre os algoritmos possíveis*.

10.6 Exercícios

Exercício 10.1 Elabore uma variante do algoritmo apresentado que seja desenvolvido tendo por base o invariante

$$\mathbf{INV} = \begin{array}{c} \begin{array}{ccccccccc} 1 & \leq & d & \leq & f & & \leq & t & n \end{array} \\ B \quad \boxed{=1}\boxed{=2}\boxed{}\boxed{}\boxed{}\boxed{=3} \end{array} \wedge (t \leq n+1)$$

Exercício 10.2 Elabore uma variante do algoritmo apresentado que seja desenvolvido tendo por base o invariante

$$\mathbf{INV} = \begin{array}{c} \begin{array}{ccccccccc} 1 & \leq & f & \leq & d & & \leq & t & n \end{array} \\ B \quad \boxed{=1}\boxed{}\boxed{}\boxed{=2}\boxed{=3} \end{array} \wedge (t \leq n+1)$$

Exercício 10.3 Elabore um invariante para o problema que afirme que, a partir de certa posição de B, os elementos já estão organizados ordenadamente (aqueles iguais a 1, seguido daqueles iguais a 2, e por fim seguido daqueles iguais a 3). A parte inicial do vetor fica reservada aos elementos ainda não inspecionados. Com base neste invariante, elabore o algoritmo correspondente.

11. Elementos Mínimos

11.1 Descrição

Dado um vetor V e naturais n, k, considere o problema de determinar os k menores elementos dentre os n primeiros de V. Tais elementos mínimos devem ser colocados em um vetor m, de modo que $m[i]$ seja o i-ésimo menor elemento de V, para todo $1 \leq i \leq k$. Como exemplo, se

$$n = 6,\ k = 3 \text{ e } V = [4,2,10,1,2,4]$$

a valoração

$$m = [1,2,2]$$

é aquela que satisfaz ao problema. Naturalmente, alguns elementos mínimos podem ser indefinidos, o que ocorre quando $n < k$. Neste caso, os respectivos $k - n$ valores de m devem ser iguais a $+\infty$.

Desenvolva seu algoritmo. Ao final, compare com aquele que será desenvolvido a seguir.

11.2 Desenvolvendo o Algoritmo

Em termos formais, procuramos elaborar um algoritmo A tal que a tH

$$\{\ \mathbf{P} = n \geq 0\ \}\ A\ \{\ \mathbf{Q} = \forall\, x \in [1..k] : \underline{m}[x] = \textsc{Min}(V,n,x)\ \}$$

seja válida, onde define-se a função

$$\textsc{Min}(V,n,x) = \text{“}x\text{-ésimo menor elemento de } V[1..n]\text{”}$$

Um comando iterativo será necessário para se elaborar A, ou seja,

$$A = \mathit{INI}\ \{\ \mathbf{INV}\ \}\ \textbf{enquanto}\ C\ \textbf{faça}\ R;$$

Usando o método de substituir uma constante (n) por uma variável (i), elaboramos o invariante

$$\mathbf{INV} = (0 \leq i \leq n) \wedge (\forall\, x \in [1..k] : m[x] = \textsc{Min}(V,i,x))$$

e, para tal invariante, a condição de parada e o comando de inicialização são, respectivamente,

$$C = \neg(i = n) = (i \neq n)$$

$$INI = i, m[1..k] \leftarrow 0, +\infty$$

Em seguida, é necessário elaborar um algoritmo R que progrida em relação à terminação, enquanto mantém o invariante. Comparando-se a inicialização com a condição de parada, propomos

$$R = R' ; i \leftarrow i+1$$

e, portanto, devemos agora desenvolver R' de modo que a tH

$$\{ \mathbf{INV} \wedge C \}\ R'\ \{ \mathbf{INV}^i_{i+1} = (0 \leq i+1 \leq n) \wedge (\forall\, x \in [1..k] : m[x] = \textsc{Min}(V, i+1, x)) \}$$

seja válida. Note que o algoritmo R' deve lidar com o caso em que o invariante sendo válido para o valor de i, precise se tornar válido para o valor de i quando incrementado. Isto sugere revisitarmos a definição da função MIN, formalizando-a por uma descrição recursiva. Com efeito, temos que

$$\textsc{Min}(V, i+1, x) = \begin{cases} +\infty & \text{, se } x > i+1 \\ -\infty & \text{, se } x = 0 \\ m_{x-1} & \text{, se } V[i+1] \leq m_{x-1} \\ V[i+1] & \text{, se } m_{x-1} \leq V[i+1] \leq m_x \\ m_x & \text{, se } m_x \leq V[i+1] \end{cases}$$

onde $m_r = \textsc{Min}(V, i, r)$.

Note que o valor de $-\infty$ para $x = 0$ foi convenientemente escolhido para que as expressões que envolvam m_{x-1} funcionem quando $x = 1$. Temos assim uma forma de atualizar os valores mínimos. Para qualquer valor de $1 \leq x \leq k$, a atualização pode ser feita por

$$\begin{array}{ll} & \{ (m[x-1] = \textsc{Min}(V, i, x-1)) \wedge (m[x] = \textsc{Min}(V, i, x)) \} \\ & \textbf{se } V[i+1] \leq m[x-1] \textbf{ então} \\ & \quad m[x] \leftarrow m[x-1] \\ R'' = & \| \ m[x-1] \leq V[i+1] \leq m[x] \textbf{ então} \\ & \quad m[x] \leftarrow V[i+1] \\ & \| \ m[x] \leq V[i+1] \textbf{ então} \\ & \quad \emptyset \\ & \{ m[x] = \textsc{Min}(V, i+1, x) \} \end{array}$$

se o invariante e o comando de inicialização forem redefinidos como

$$\mathbf{INV} = (0 \leq i \leq n) \wedge (\forall\, x \in [0..k] : m[x] = \textsc{Min}(V, i, x))$$

$$INI = i, m[1..k], m[0] \leftarrow 0, +\infty, -\infty$$

Agora, é necessário aplicar R'' para todo $1 \leq x \leq k$. Para que a pré-condição de R'' ser respeitada, temos que a ordem de aplicação deva ser para $x = k, k-1, \ldots 1$. Ou seja,

$$\begin{array}{ll} R' = & \textbf{para } x \leftarrow k, k-1, \ldots, 1 \textbf{ faça} \\ & \quad R'' \end{array}$$

11.3 Descrição do Algoritmo Desenvolvido

$\{\ \mathbf{P} = n \geq 0\ \}$
$i, m[1..k], m[0] \leftarrow 0, +\infty, -\infty$
$\{\ \mathbf{INV} = (0 \leq i \leq n) \wedge (\forall\, x \in [0..k] : m[x] = \textsc{Min}(V,i,x))\ \}$ // $t = n - i$
enquanto $i \neq n$ **faça**
  para $x \leftarrow k, k-1, \ldots, 1$ **faça**
    $\{\ (m[x-1] = \textsc{Min}(V,i,x-1)) \wedge (m[x] = \textsc{Min}(V,i,x))\ \}$
    se $V[i+1] \leq m[x-1]$ **então**
      $m[x] \leftarrow m[x-1]$
    $\|$ $m[x-1] \leq V[i+1] \leq m[x]$ **então**
      $m[x] \leftarrow V[i+1]$
    $\|$ $m[x] \leq V[i+1]$ **então**
      $\emptyset$
    $\{\ m[x] = \textsc{Min}(V,i+1,x)\ \}$
  $i \leftarrow i+1$
$\{\ \mathbf{Q} = \forall\, x \in [1..k] : m[x] = \textsc{Min}(V,n,x)\ \}$

11.4 Considerações sobre Eficiência

A complexidade de tempo do algoritmo desenvolvido é de $\Theta(kn)$. Outra solução comum para este problema é ordenar V para obter seus k primeiros elementos depois de ordenado, cuja complexidade de tempo resulta em $\Theta(n \log n)$. A solução original é preferível sempre que k é pequeno em relação a n ou, mais precisamente, quando $k = o(\log n)$, enquanto a segunda é preferível no caso contrário. Há uma solução contudo mais elaborada, de tempo ótimo de $\Theta(n + k \log k)$ para o problema. Nesta solução, emprega-se o algoritmo de obter o k-ésimo menor elemento de V, que pode ser feito em tempo $\Theta(n)$. Em seguida, pode-se particionar V nos elementos que estão abaixo e acima de tal valor, que também pode ser feito em tempo $\Theta(n)$. Por fim, ordena-se os k elementos da primeira parte de tal partição para se obter ordenadamente os k menores elementos de V, em tempo $\Theta(k \log k)$.

11.5 Observações Importantes

■ **Observação 11.1** O algoritmo foi desenvolvido em uma sequência que não corresponde àquela linear "de cima para baixo" correspondente a sua descrição textual. Tal *sequência de descrição* não deve ser confundida como necessária também para a *sequência de desenvolvimento*.

■ **Observação 11.2** Ao invés do desenvolvimento ser orientado pelo processo de execução (o "chinês" do algoritmo), ele é guiado pelas condições lógicas oriundas da correção de cada comando. Uma vez escolhidos ambos o comando e o respectivo método de prova, este último guia a elaboração das partes do primeiro, em um desenvolvimento conjunto de algoritmo e prova.

■ **Observação 11.3** O comando **para** foi desenvolvido, de maneira livre, sem aplicação do

método formal de desenvolvimento. Na prática, isto é permitido sempre que sua correção for evidente (veja Exercício 11.4). O método de correção visa guiar o desenvolvimento quando este é julgado como complexo, não para ser um fardo ao desenvolvimento de algoritmos (ou trechos de) cuja correção esteja suficientemente clara. Neste fino balanço entre formalidade e intuição reside a verdadeira eficácia do emprego do método de correção na prática.

11.6 Exercícios

Exercício 11.1 Redefina a pós-condição **Q** em termo da função:

$$\textsc{Ord}(V,n) = \text{“vetor com os primeiros } n \text{ elementos do vetor } V \text{ em ordem não-decrescente”}$$

Exercício 11.2 A definição de um valor para a função $\textsc{Min}(V,i+1,x)$ para $x = 0$, que implicou na criação da posição extra $m[0]$, foi feita por mera conveniência. O objetivo foi que o algoritmo se tornasse homogêneo para todas as posições do vetor. Se assim não fosse, qual deveria ser a definição recursiva da função $\textsc{Min}(V,i+1,x)$? Atualize o o algoritmo em conformidade com tal nova definição.

Exercício 11.3 Há duas melhorias que podem ser feitas no comando **para**. A primeira, é iniciar o valor de x com um valor em geral menor que k. A segunda, é finalizar a repetição em uma condição que em geral ocorre antes de x ser menor que 1. Quais são tais valor inicial e condição de parada? Modifique o algoritmo para realizar tais mudanças.

Exercício 11.4 O comando **para** do algoritmo constitui uma repetição para a qual não foi empregado o método de desenvolvimento formal, isto é, pulou-se as etapas de estabelecer um invariante, comando de inicialização, etc.. Desenvolva formalmente tal repetição, definindo o predicado

$$\mathbf{Z}(I,s) = (0 \leq s \leq n) \wedge (\forall\, x' \in I : m[x'] = \textsc{Min}(V,s,x'))$$

e para invariante, usando e técnica de combinar pré- e pós-condição, o predicado

$$\mathbf{INV}' = \mathbf{Z}([0..x],i) \wedge \mathbf{Z}((x,k],i+1)$$

Exercício 11.5 Outro algoritmo poderia ser derivado se o método para se obter o invariante fosse aquele de substituir a constante k por uma variável i. Formalize tal invariante e desenvolva o algoritmo que dele decorre.

Exercício 11.6 Suponha que deseja-se, após a elaboração do algoritmo, verificar agora formal-

mente sua correção. Tomemos, em particular, o comando

$$m[x] \leftarrow m[x-1]$$

Mostre qual a verificação formal que deve ser efetuada para provar que este comando está corretamente empregado. Em seguida, proceda tal verificação. Para este exercício, é necessário se trabalhar com a formalização do desenvolvimento do comando **para**, conforme solicitado no Exercício 11.4.

12. Exponenciação

12.1 Descrição

Dados naturais a, b, atribuir à variável e o valor

$$e = a^b$$

Como exemplo, se

$$a = 3,\ b = 4$$

a valoração

$$e = 243$$

é aquela que satisfaz ao problema. Naturalmente, o problema é trivial se for pressuposto que a função de exponenciação pertence ao conjunto de comandos básicos da linguagem de programação. No caso presente, procuramos por uma solução que empregue apenas as 4 operações básicas aritméticas (soma, subtração, multiplicação e divisão).

Desenvolva seu algoritmo. Ao final, compare com aquele que será desenvolvido a seguir.

12.2 Desenvolvendo o Algoritmo

Em termos formais, procuramos elaborar um algoritmo A tal que a tH

$$\{\ \mathbf{P} = (a \geq 1) \wedge (b \geq 0)\ \}\ A\ \left\{\ \mathbf{Q} = (\underline{e} = a^b)\ \right\}$$

seja válida. Um comando iterativo será necessário para se elaborar A, ou seja,

$$A = \mathit{INI}\ \{\ \mathbf{INV}\ \}\ \textbf{enquanto}\ C\ \textbf{faça}\ R;$$

Usando o método de substituir uma constante (b) por uma variável (i), elaboramos o invariante

$$\mathbf{INV} = (0 \leq i \leq b) \wedge (e = a^i)$$

e, para tal invariante, a condição de parada e o comando de inicialização são, respectivamente,

$$C = \neg(i = b) = (i \neq b)$$

$$INI = i, e \leftarrow 0, 1$$

Em seguida, é necessário elaborar um algoritmo R que progrida em relação à terminação, enquanto mantém o invariante. Comparando-se a inicialização com a condição de parada, propomos

$$R = R' \; ; i \leftarrow i + 1$$

e, portanto, devemos agora desenvolver R' de modo que a tH

$$\{ \; \mathbf{INV} \wedge C \; \} \; R' \; \{ \; \mathbf{INV}^{i}_{i+1} = (0 \leq i+1 \leq b) \wedge (e = a^{i+1}) \; \}$$

seja válida. Note que o algoritmo R' deve lidar com o caso em que o invariante sendo válido para o valor de i, precise se tornar válido para o valor de i quando incrementado. Isto sugere revisitarmos a definição da exponenciação, formalizando-a por uma descrição recursiva. Com efeito, temos que

$$a^{i+1} = \begin{cases} 1 & \text{, se } i+1 = 0 \\ a \cdot a^{i} & \text{, se } i+1 > 0 \end{cases}$$

Portanto,

$$R' = e \leftarrow a * e$$

completa o algoritmo (Solução #1). Consideraremos, agora, uma outra solução. Nesta próxima, também empregamos o método de substituir uma constante por uma variável. Na verdade, serão substituídas duas constantes de uma só vez, as constantes de valor 1 tornadas explícitas na pós-condição

$$\mathbf{Q} = (1 \cdot e^{1} = a^{b})$$

Elaboramos então o invariante

$$\mathbf{INV} = (x \geq 1) \wedge (y \geq 0) \wedge (xe^{y} = a^{b})$$

e, para tal invariante, a condição de parada e o comando de inicialização são, respectivamente,

$$\neg C = \neg((x = 1) \wedge (y = 1)) = ((x \neq 1) \vee (y \neq 1))$$

$$INI = x, e, y \leftarrow 1, a, b$$

Em seguida, é necessário elaborar um algoritmo R que progrida em relação à terminação, enquanto mantém o invariante. Comparando-se a inicialização com a condição de parada, devemos diminuir o valor de y, exceto quando $y = 0$. Assim, decidirmos tratar de forma especial esta exceção, para que os demais casos se preocupem apenas em reduzir o valor de y. Uma ideia para se obter uma melhor eficiência, motivação para a elaboração desta nova solução, é fazer com que y diminua não somente de unidade a unidade, mas que seja cortado pela metade. Isto será possível nos casos em que y seja par. Como conclusão, os argumentos anteriores sugerem o emprego de um comando alternativo, isto é,

$$R = \textbf{se } y = 0 \textbf{ então } R_1 \;\; \| \;\; (y \text{ é par}) \wedge (y \neq 0) \textbf{ então } R_2 \;\; \| \;\; y \text{ é ímpar } \textbf{então } R_3$$

Para o caso em que $y = 0$, parece ser possível escolher o valor adequado para e e terminar o algoritmo. Supondo estarmos incertos de qual seria tal valor adequado, deixamos o método de correção guiar o desenvolvimento. Estabelecemos R_1 à forma geral

$$R_1 = x, e, y \leftarrow 1, \square, 1$$

e desejamos saber para que valor da incógnita $\square$ o comando de atribuição está correto. Calculamos que

$$\text{PMF}(\text{“}R_1\text{”}, \textbf{INV}) = (1 \geq 1) \wedge (1 \geq 0) \wedge (\square = a^b)$$

Portanto, para que seja válido

$$\textbf{INV} \wedge C \wedge (y = 0) \Rightarrow \text{PMF}(\text{“}R_1\text{”}, \textbf{INV})$$

é suficiente que

$$\square = x$$

Em relação a R_2, a atualização de y consistirá, como discutimos, de seu corte pela metade. Para que o invariante se mantenha, naturalmente os valores de x, e devem ser atualizados de forma apropriada. Estabelecemos para R_2 a forma geral

$$R_2 = x, e, y \leftarrow \square_1, \square_2, y/2$$

calculamos que

$$\text{PMF}(\text{“}R_2\text{”}, \textbf{INV}) = (\square_1 \geq 1) \wedge (y/2 \geq 0) \wedge (\square_1 \square_2^{y/2} = a^b)$$

Portanto, para que seja válido

$$\textbf{INV} \wedge C \wedge (y \text{ é par}) \wedge (y \neq 0) \Rightarrow \text{PMF}(\text{“}R_2\text{”}, \textbf{INV})$$

é suficiente que

$$\square_1 = x, \square_2 = e^2$$

Assim, R_2 resulta, após ser simplificado, em

$$R_2 = e, y \leftarrow e * e, y/2$$

Para o caso em que y é ímpar, intenciona-se apenas decrementar y. Isto é, a forma geral de R_3 é

$$R_3 = x, e, y \leftarrow \square_1, \square_2, y - 1$$

e calculamos que

$$\text{PMF}(\text{“}R_3\text{”}, \textbf{INV}) = (\square_1 \geq 1) \wedge (y - 1 \geq 0) \wedge (\square_1 \square_2^{y-1} = a^b)$$

Portanto, para que seja válido

$$\textbf{INV} \wedge C \wedge (\text{“}y \text{ é ímpar”}) \Rightarrow \text{PMF}(\text{“}R_3\text{”}, \textbf{INV})$$

é suficiente que

$$\square_1 = x * e, \square_2 = e$$

Assim, R_3 resulta, após ser simplificado, em

$$R_3 = x, y \leftarrow x * e, y - 1$$

completando o algoritmo (Solução #2).

12.3 Descrição do Algoritmo Desenvolvido

- Solução #1:

$\{\ \mathbf{P} = (a \geq 1) \wedge (b \geq 0)\ \}$
$i, e \leftarrow 0, 1$
$\{\ \mathbf{INV} = (0 \leq i \leq b) \wedge (e = a^i)\ \}$ // $t = b - i$
enquanto $i \neq b$ **faça**
  $e \leftarrow a * e$
  $i \leftarrow i + 1$
$\{\ \mathbf{Q} = (e = a^b)\ \}$

- Solução #2:

$\{\ \mathbf{P} = (a \geq 1) \wedge (b \geq 0)\ \}$
$x, e, y \leftarrow 1, a, b$
$\{\ \mathbf{INV} = (x \geq 1) \wedge (y \geq 0) \wedge (xe^y = a^b)\ \}$ // $t = (x-1) + (y-1)$
enquanto $(x \neq 1) \vee (y \neq 1)$ **faça**
  se $y = 0$ **então**
    $x, e, y \leftarrow 1, x, 1$
  || $(y$ *é par*$) \wedge (y \neq 0)$ **então**
    $e, y \leftarrow e * e, y/2$
  || y *é ímpar* **então**
    $x, y \leftarrow x * e, y - 1$
$\{\ \mathbf{Q} = (e = a^b)\ \}$

12.4 Considerações sobre Eficiência

A complexidade de tempo do algoritmo da Solução #1 é de $\Theta(b)$, enquanto o algoritmo da Solução #2 é de $\Theta(\log b)$. Note que, em cada iteração do segundo algoritmo, ou bem y é cortado ao meio, ou se ele for decrementado de uma unidade, será certamente cortado ao meio na iteração seguinte.

12.5 Observações Importantes

■ **Observação 12.1** Enquanto o algoritmo desenvolvido na Solução #1 é bem familiar aos já iniciados no estudo de algoritmos, aquele da Solução #2 parece requerer um tempo maior para ser digerido. Na verdade, este algoritmo é um bom exemplo da aplicação dos métodos de correção. Ao final de seu desenvolvimento, confiamos no algoritmo pelo resultado positivo da verificação formal , sendo mais difícil de se convencer de sua correção por outras maneiras *adhoc*. É um exemplo de "desenvolvimento sendo navegado por instrumentos", fazendo alusão à condição de profissionais da aviação que, em condições climáticas adversas, devem conduzir a aeronave confiando primordialmente nas medições e verificações reportadas pela instrumentação, já que a visão encontra-se obstruída.

12.6 Exercícios

Exercício 12.1 Generalize os algoritmos, permitindo que as variáveis a, b de entrada sejam inteiros arbitrário (isto é, um valor positivo, negativo ou nulo), desde que ao menos um deles seja diferente de zero. Não se esqueça de reescrever também a pré-condição.

Exercício 12.2 Uma possibilidade natural para o problema seria tentar o invariante obtido de substituição da constante a da pós-condição por uma variável. Tente desenvolver o algoritmo que decorre deste invariante e explique em que passo do desenvolvimento apresenta-se uma dificuldade (e qual seria tal dificuldade).

Exercício 12.3 Elabore um algoritmo para o problema de, dado uma matriz A de dimensão $n \times n$, determine a matriz A^k, dada por

$$A^k = A \times A \times \cdots \times A \qquad (k \text{ vezes})$$

Em termos de complexidade, sua solução deve ter complexidade $O(n^3 \log n)$.

Exercício 12.4 A *sequencia de Fibonacci* é a sequência infinita de naturais $f_1, f_2, \ldots$ tal que o n-ésimo termo f_n é definido por

$$f_n = \begin{cases} 1 & \text{, se } n = 1 \text{ ou } n = 2 \\ f_{n-1} + f_{n-2} & \text{, se } n \geq 3 \end{cases}$$

Como ilustração, os primeiros exemplos da sequência são: 1, 1, 2, 3, 5, 8, 13, ... Mostre que, para todo $n \geq 3$,

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{n-2} \times \begin{pmatrix} 1 \\ 1 \end{pmatrix} = \begin{pmatrix} f_n \\ f_{n-1} \end{pmatrix}$$

e, empregando tal fato, elabore um algoritmo para o problema de, dado um natural n, determinar f_n. Em termos de complexidade, sua solução deve ter complexidade $O(\log n)$.

13. Média Aritmética

13.1 Descrição

Dados n naturais, armazenados em um vetor $B[1..n]$, atribuir à variável m a média aritmética destes elementos. Isto é, ao final do algoritmo, deve valer

$$m = \frac{\sum_{k=1}^{n} B[k]}{n}$$

Como exemplo, se

$$B = [10, 1, 23, 53, 32, 22],\ n = 6$$

a valoração

$$m = 23{,}5$$

é aquela que satisfaz ao problema.

Desenvolva seu algoritmo. Ao final, compare com aquele que será desenvolvido a seguir.

13.2 Desenvolvendo o Algoritmo

Note que, pela sua definição, a média aritmética está definida somente para todo $n \geq 1$. Em termos formais, procuramos elaborar um algoritmo A tal que a tH

$$\{\ \mathbf{P} = (n \geq 1)\ \}\ A\ \left\{\ \mathbf{Q} = \left(\underline{m} = \frac{\sum_{k=1}^{n} B[k]}{n}\right)\ \right\}$$

seja válida. Observando-se que a computação de $\sum_{k=1}^{n} B[k]$ já foi estudada (Exemplo 9.7), podemos usar um comando de composição, estabelecendo um objetivo intermediário, como a seguir

$$A = A_1\ \left\{\ m = \sum_{k=1}^{n} B[k]\ \right\}\ A_2$$

O algoritmo A_2 é dado por

$$A_2 = m \leftarrow m/n$$

completando a Solução #1. Na próxima solução, um comando iterativo será empregado para se elaborar A, ou seja,

$$A = \mathit{INI}\ \{\ \mathbf{INV}\ \}\ \textbf{enquanto}\ C\ \textbf{faça}\ R;$$

Usando o método de substituir uma constante (n) por uma variável (i), deriva-se o invariante

$$\mathbf{INV} = \left(m = \frac{\sum_{k=1}^{i} B[k]}{n} \right)$$

e, para tal invariante, a condição de parada e o comando de inicialização são, respectivamente,

$$C = \neg(i = n) = (i \neq n)$$

$$\mathit{INI} = i, m \leftarrow 0, 0$$

Em seguida, é necessário elaborar um algoritmo R que progrida em relação à terminação, enquanto mantém o invariante. Comparando-se a inicialização com a condição de parada, propomos

$$R = R'\ ;\ i \leftarrow i+1$$

e, portanto, devemos agora desenvolver R' de modo que a tH

$$\{\ \mathbf{INV} \wedge C\ \}\ R'\ \left\{\ \mathbf{INV}^{i}_{i+1} = \left(m = \frac{\sum_{k=1}^{i+1} B[k]}{n} \right)\ \right\}$$

seja válida, o que se consegue com

$$R' = m \leftarrow m + B[i+1]/n$$

completando o algoritmo (Solução #2). Consideraremos, ainda, uma terceira solução, novamente com o uso do comando iterativo. Nesta próxima, também empregamos o método de substituir a constante n pela variável i, com a diferença que ambas as ocorrências de n serão substituídas, resultando no invariante

$$\mathbf{INV} = \left(m = \frac{\sum_{k=1}^{i} B[k]}{i} \right)$$

e, para tal invariante, a condição de parada e o comando de inicialização são, respectivamente,

$$C = \neg(i = n) = (i \neq n)$$

$$\mathit{INI} = i, m \leftarrow 1, B[1]$$

(Agora, não é mais possível inicializar i com 0.) Em seguida, é necessário elaborar um algoritmo R que progrida em relação à terminação, enquanto mantém o invariante. Da mesma forma, temos

$$R = R'\ ;\ i \leftarrow i+1$$

e desenvolve-se R' de modo que a tH

$$\{\ \mathbf{INV} \wedge C\ \}\ R'\ \left\{\ \mathbf{INV}^{i}_{i+1} = \left(m = \frac{\sum_{k=1}^{i+1} B[k]}{i+1} \right)\ \right\}$$

seja válida. Naturalmente, deve-se alterar o valor de m, porém, não é claro exatamente para qual valor. Utilizando a técnica de deixar o método guiar o desenvolvimento, estabelecemos R' como

$$R' = m \leftarrow \square$$

e desejamos saber para que valor da incógnita $\square$ o comando de atribuição está correto. Calculamos que

$$\text{PMF}(\text{“}R'\text{”}, \mathbf{INV}^{i}_{i+1}) = \left(\square = \frac{\sum_{k=1}^{i+1} B[k]}{i+1} \right)$$

Como

$$\square = \frac{\sum_{k=1}^{i+1} B[k]}{i+1} = \left(\frac{\sum_{k=1}^{i} B[k]}{i} \right) \left(\frac{i}{i+1} \right) + \frac{B[i+1]}{i+1}$$

para que seja válido

$$\mathbf{INV} \wedge C \Rightarrow \text{PMF}(\text{“}R'\text{”}, \mathbf{INV}^{i}_{i+1})$$

é suficiente que

$$\square = m \left(\frac{i}{i+1} \right) + \frac{B[i+1]}{i+1}$$

completando o algoritmo (Solução #3).

13.3 Descrição do Algoritmo Desenvolvido

- Solução #1:

$\{ \mathbf{P} = (n \geq 1) \}$
$i, m \leftarrow 0, 0$
$\{ \mathbf{INV} = \left(m = \sum_{k=1}^{i} B[k]\right) \}$ // $t = n - i$
enquanto $i \neq n$ **faça**
  $m \leftarrow m + B[i+1]$
  $i \leftarrow i + 1$
$\{ m = \sum_{k=1}^{n} B[k] \}$
$m \leftarrow m/n$
$\left\{ \mathbf{Q} = \left(\underline{m} = \frac{\sum_{k=1}^{n} B[k]}{n} \right) \right\}$

- Solução #2:

$\{ \mathbf{P} = (n \geq 1) \}$
$i, m \leftarrow 0, 0$
$\left\{ \mathbf{INV} = \left(m = \frac{\sum_{k=1}^{i} B[k]}{n} \right) \right\}$ // $t = n - i$
enquanto $i \neq n$ **faça**
  $m \leftarrow m + B[i+1]/n$
  $i \leftarrow i + 1$
$\left\{ \mathbf{Q} = \left(\underline{m} = \frac{\sum_{k=1}^{n} B[k]}{n} \right) \right\}$

- Solução #3:

$\{\ \mathbf{P} = (n \geq 1)\ \}$
$i, m \leftarrow 1, B[1]$
$\left\{\ \mathbf{INV} = \left(m = \frac{\sum_{k=1}^{i} B[k]}{i}\right)\ \right\}$ // $t = n - i$
enquanto $i \neq n$ **faça**
  $m \leftarrow m\left(\frac{i}{i+1}\right) + \frac{B[i+1]}{i+1}$
  $i \leftarrow i + 1$
$\left\{\ \mathbf{Q} = \left(\underline{m} = \frac{\sum_{k=1}^{n} B[k]}{n}\right)\ \right\}$

13.4 Considerações sobre Eficiência

A complexidade de tempo dos algoritmos das Soluções #1, #2 e #3 são todos de $\Theta(n)$. No entanto, eles possuem uma característica que os diferem, em relação aos valores intermediários de m até seu valor final com a média aritmética. Sejam b_{min} $b_{\text{máx}}$ respectivamente os valores mínimo e máximo dentre os elementos $B[1..n]$. Note que na Solução #1, os valores de m são crescentes durante a iteração, atingindo um valor máximo potencialmente muito maior que $b_{\text{máx}}$ e, em seguida, decresce de uma só vez para o valor exato da média. Já na Solução #2, os valores de m partem de um valor mínimo, potencialmente muito menor que b_{min}, e são crescentes a partir daí, até atingir o valor exato da média. Na Solução #3, note que o invariante especifica que m será a média aritmética dos elementos $B[1..i]$. Assim, para toda iteração tal que $1 \leq i \leq n$, o valor de m estará entre b_{min} e $b_{\text{máx}}$. Esta última propriedade é importante para garantir que o emprego do algoritmo na prática não causará nem instabilidades numéricas, que ocorrem quando valores intermediários de cálculo atingem valores muito pequenos, nem estouros de variáveis, que ocorrem quando valores intermediários ultrapassam o maior valor representável pelo tipo da variável.

13.5 Observações Importantes

■ **Observação 13.1** Na Solução #3, foi necessário encontrar um valor para a incógnita $\square$ para que fosse válido

$$\mathbf{INV} \wedge C \Rightarrow \text{PMF}(\text{“}R'\text{”}, \mathbf{INV}^{i}_{i+1})$$

e uma pergunta interessante que pode surgir seria a seguinte: dado que

$$\text{PMF}(\text{“}R'\text{”}, \mathbf{INV}^{i}_{i+1}) = \left(\square = \frac{\sum_{k=1}^{i+1} B[k]}{i+1}\right)$$

a forma mais fácil de atribuir um valor a incógnita não seria tornar o consequente da implicação em **V**? Aqui, mais fácil, se refere ao fato que toda implicação cujo consequente seja **V** é válida, independente do antecedente. E isto poderia ser conseguido fazendo

$$\square = \frac{\sum_{k=1}^{i+1} B[k]}{i+1}$$

Isto estaria correto? Bem, do ponto de vista lógico, estaria. Mas do ponto de vista computacional, não é uma solução adequada. Note que $\square$ corresponde, no problema, ao valor que deve ser

atribuído a m. Basta ver que, quando $i = n$ na última iteração, estaríamos dizendo que deve-se atribuir a m a média dos elementos $B[1..n]$, que é justamente o problema que estamos tentando resolver! E é justamente aqui que entra o papel do antecedente: devemos utilizá-lo para que a expressão de $\square$ seja de simples computação. Como o antecedente da implicação é

$$\mathbf{INV} \wedge C = \left(m = \frac{\sum_{k=1}^{i} B[k]}{i} \right) \wedge (i \neq n)$$

reescrevemos o consequente para que o resultados do antecedente possa ser utilizado para simplificar a computação de $\square$. Isto justifica a reescrita como

$$\square = \frac{\sum_{k=1}^{i+1} B[k]}{i+1} = \left(\frac{\sum_{k=1}^{i} B[k]}{i} \right) \left(\frac{i}{i+1} \right) + \frac{B[i+1]}{i+1}$$

com o objetivo de permitir que o valor de m garantido pelo antecedente possa ser empregado na expressão de $\square$. De forma geral, esta é a estratégia que deve sempre ser seguida na determinação de tais incógnitas: *o emprego do antecedente para simplificar a expressão da incógnita do consequente*.

13.6 Exercícios

Exercício 13.1 A média geométrica de um conjunto de elementos é aquela em que o produto e não a soma dos elementos é empregada para o seu cálculo. Dado um vetor $B[1..n]$, a *média geométrica* m_{G} de seus elementos é definida como

$$m_{\mathrm{G}} = (B[1]B[2]\cdots B[n])^{\frac{1}{n}}$$

Elabore um algoritmo que atribua o valor de m_{G} a uma variável m. Assuma a disponibilidade ao algoritmo de uma função $\textsc{Pot}(x,y)$, que computa o valor de x^y, para todo racional x, y.

Exercício 13.2 A média harmônica de um conjunto de elementos é o inverso da média aritmética dos inversos destes elementos. Isto é, dado um vetor $B[1..n]$, a *média harmônica* m_{H} de seus elementos é definida como

$$m_{\mathrm{H}} = \frac{n}{\frac{1}{B[1]} + \frac{1}{B[2]} + \cdots + \frac{1}{B[n]}}$$

Elabore um algoritmo que atribua o valor de m_{H} a uma variável m.

Exercício 13.3 Dado um vetor $B[1..n]$ e inteiro p, a *média de potências* m_p de seus elementos é definida como

$$m_p = \left(\frac{B[1]^p + B[2]^p + \cdots + B[n]^p}{n} \right)^{\frac{1}{p}}$$

Elabore um algoritmo que atribua o valor de m_p a uma variável m.

Exercício 13.4 Dado um vetor $B[1..n]$ e função $f : \mathbb{R} \to \mathbb{R}$, a *$f$-média* m_f de seus elementos é definida como

$$m_f = f^{-1}\left(\frac{f(B[1]) + f(B[2]) + \cdots + f(B[n])}{n}\right)$$

A importância desta média é que generaliza outras médias. Com efeito, note que a f-média se reduz à média aritmética para $f(x) = x$, à harmônica para $f(x) = 1/x$, e à geométrica para $f(x) = \ln x$. Elabore um algoritmo que atribua o valor de m_f a uma variável m. Assuma a disponibilidade ao algoritmo de funções $\mathrm{F}(x)$ e $\mathrm{F}^{-1}(x)$, que computa respectivamente os valores de $f(x)$ e $f^{-1}(x)$.

Exercício 13.5 Na Solução #3, a definição do valor a ser atribuído a m, no contexto do algoritmo R', foi empregado uma incógnita $\square$ e, para sua determinação, foi empregado o método de correção (veja Observação 13.1). Aplique técnica análoga para determinar os valores de atualização de m nas Soluções #1 e #2.

14. Raiz Quadrada

14.1 Descrição

Dado natural n, atribuir à variável r a raiz quadrada inteira de n. Isto é, ao final do algoritmo, deve valer

$$r = \lfloor \sqrt{n} \rfloor$$

Como exemplo, se

$$n = 23$$

a valoração

$$r = 4$$

é aquela que satisfaz ao problema, uma vez que $\lfloor \sqrt{23} \rfloor = 4$.

Desenvolva seu algoritmo. Ao final, compare com aquele que será desenvolvido a seguir.

14.2 Desenvolvendo o Algoritmo

Em termos formais, procuramos elaborar um algoritmo A tal que a tH

$$\{ \mathbf{P} = (n \geq 0) \}\ A\ \{ \mathbf{Q} = (r = \lfloor \sqrt{n} \rfloor) \}$$

seja válida. Um comando iterativo será empregado para se elaborar A, ou seja,

$$A = \mathit{INI}\ \{ \mathbf{INV} \}\ \textbf{enquanto}\ C\ \textbf{faça}\ R;$$

Usando o método de substituir uma constante (n) por uma variável (i), deriva-se o invariante

$$\mathbf{INV} = (r = \lfloor \sqrt{i} \rfloor) \wedge (0 \leq i \leq n)$$

No entanto, o desenvolvimento de um algoritmo a partir deste invariante oferecerá dificuldades (Exercício 14.1). Não parece que as outras técnicas de elaboração de invariante funcionem para a pós-condição apresentada. Assim, redefiniremos a pós-condição, de modo a eventualmente permitir outras elaborações de invariante. No Exemplo 9.6, apresentou-se outras possibilidades de invariante. Considere a redefinição da pós-condição como

$$\{\ \mathbf{Q} = \underline{r}^2 \leq n < (r+1)^2\ \}$$

Esta pós-condição possui implícita a conjunção $(r^2 \leq n) \wedge (n < (r+1)^2)$. Aplicando o método de eliminação de conjunção, podemos derivar

$$\mathbf{INV} = (n < (r+1)^2)$$

Para tal invariante, a condição de parada e o comando de inicialização são, respectivamente,

$$C = \neg(r^2 \leq n) = (r^2 > n)$$

$$INI = r \leftarrow n$$

Em seguida, é necessário elaborar um algoritmo R que progrida em relação à terminação, enquanto mantém o invariante. Comparando-se a inicialização com a condição de parada, propomos

$$R = R' \ ; r \leftarrow r-1$$

e, portanto, devemos agora desenvolver R' de modo que a tH

$$\{\ \mathbf{INV} \wedge C\ \}\ R'\ \{\ \mathbf{INV}^r_{r-1} = (n < r^2)\ \}$$

seja válida, o que se consegue com

$$R' = \emptyset$$

completando o algoritmo (Solução #1). Note que, retornando à elaboração do invariante, pode-se também escolher a outra cláusula para ser eliminada, produzindo o invariante

$$\mathbf{INV} = (r^2 \leq n)$$

e, para tal invariante, a condição de parada e o comando de inicialização são, respectivamente,

$$C = \neg(n < (r+1)^2) = (n \geq (r+1)^2)$$

$$INI = r \leftarrow 0$$

Em seguida, é necessário elaborar um algoritmo R que progrida em relação à terminação, enquanto mantém o invariante. Da mesma forma, temos

$$R = R' \ ; r \leftarrow r+1$$

e desenvolve-se R' de modo que a tH

$$\{\ \mathbf{INV} \wedge C\ \}\ R'\ \{\ \mathbf{INV}^r_{r+1} = ((r+1)^2 \leq n)\ \}$$

seja válida. Novamente, isto é conseguido por

$$R' = \emptyset$$

completando o algoritmo (Solução #2). Apresentaremos agora uma terceira solução. Nesta, usando o método de ampliação do domínio de uma variável, elabora-se o invariante

$$\mathbf{INV} = (r^2 \leq n < s^2)$$

Para tal invariante, a condição de parada e o comando de inicialização são, respectivamente,

$$C = \neg(s = r+1) = (s \neq r+1)$$

$$INI = r, s \leftarrow 0, n+1$$

Em seguida, é necessário elaborar um algoritmo R que progrida em relação à terminação, enquanto mantém o invariante. Observa-se que, para qualquer valor arbitrário m, ocorre que ou $m^2 \leq n$, ou $n < m^2$. Isto significa que, para qualquer valor arbitrário m, este serve para atualizar ou r, ou s, respectivamente às condições anteriores. Sendo assim, por questão de eficiência, surge a ideia de usar como valor m a média de r e s, cortando a diferença de r e s pela metade (a grosso modo) e, a cada iteração, aproximar-se da condição de término. Isto justifica que

$$R = m \leftarrow \lfloor (r+s)/2 \rfloor \text{ ; } \textbf{se } m^2 \leq n \textbf{ então } r \leftarrow m \;\; \| \;\; n < m^2 \textbf{ então } s \leftarrow m$$

completa o algoritmo (Solução #3).

Finalmente, revisitaremos a Solução #2 para introduzir uma importante técnica. Note que, a cada iteração, uma operação de multiplicação é efetuada ($(r+1) \times (r+1)$). Uma operação que certamente possui um custo computacional maior do que aquele de uma adição. (Se isto não for verdade no modelo de computação considerado, é certamente em computações à mão.) A próxima técnica permitirá o uso do invariante para transformar o algoritmo que troque uma multiplicação por uma soma a cada iteração.

Neste técnica, observa-se a operação que se deseja economizar que, neste caso, é $(r+1)^2$, e supõe-se que tal valor já estará previamente computado em alguma nova variável, digamos, z. Sendo assim, ao redefinirmos o invariante da Solução #2 para

$$\mathbf{INV} = (r^2 \leq n) \wedge (z = (r+1)^2)$$

e, para tal invariante, a condição de parada e o comando de inicialização são, respectivamente,

$$C = \neg(n < (r+1)^2) = (n \geq (r+1)^2) = (n \geq z)$$

$$INI = r, z \leftarrow 0, 1$$

Em seguida, é necessário elaborar um algoritmo R que progrida em relação à terminação, enquanto mantém o invariante. Da mesma forma, temos

$$R = R' \text{ ; } r \leftarrow r+1$$

e desenvolve-se R' de modo que a tH

$$\{ \mathbf{INV} \wedge C \}\ R'\ \{ \mathbf{INV}^r_{r+1} = ((r+1)^2 \leq n) \wedge (z = (r+2)^2) \}$$

seja válida. É necessário atualizar z para o valor $(r+2)^2$. Se isto for feito sem maiores cuidados, naturalmente o objetivo de evitar uma multiplicação não foi atingido. Lembrando da Observação 13.1, devemos usar verdades garantidas pelo invariante para tornar esta atualização menos custosa. Com efeito, como

$$(r+2)^2 = (r+1)^2 + 2r + 3$$

temos que

$$R' = z \leftarrow z + 2 * r + 3$$

o que completa o algoritmo se a multiplicação $2 * r$ for aceita como menos custosa, uma vez que, ao invés do uso de um algoritmo geral de multiplicação, tal operação necessita tão somente uma operação de deslocamento dos bits da representação binária de r à esquerda, tão eficientemente implementável quanto uma soma geral. No entanto, se o objetivo é ser purista em relação à redução a soma, basta observar que

$$R' = z \leftarrow z + r + r + 3$$

Ainda é possível diminuir as 3 somas necessárias para se atualizar z para apenas 2. Para tanto, novamente, definimos a expressão $2r+3$ como "custosa", e a introduzimos no invariante, em uma nova variável z'. Resulta na redefinição

$$\mathbf{INV} = (r^2 \leq n) \wedge (z = (r+1)^2) \wedge (z' = 2r+3)$$

e, para tal invariante, o comando de inicialização é

$$INI = r, z, z' \leftarrow 0, 1, 3$$

Como

$$\mathbf{INV}^r_{r+1} = ((r+1)^2 \leq n) \wedge (z = (r+2)^2) \wedge (z' = 2(r+1)+3)$$

e

$$2(r+1)+3 = (2r+3)+2$$

temos que

$$R' = z, z' \leftarrow z + z', z' + 2$$

o que completa o algoritmo (Solução #4).

14.3 Descrição do Algoritmo Desenvolvido

- Solução #1:

$\{\ \mathbf{P} = (n \geq 0)\ \}$
$r \leftarrow n$
$\{\ \mathbf{INV} = n < (r+1)^2\ \}$ // $t = r - \lfloor \sqrt{n} \rfloor$
enquanto $r^2 > n$ **faça**
  $r \leftarrow r - 1$
$\{\ \mathbf{Q} = r^2 \leq n < (r+1)^2\ \}$

- Solução #2:

$\{\ \mathbf{P} = (n \geq 0)\ \}$
$r \leftarrow 0$
$\{\ \mathbf{INV} = r^2 \leq n\ \}$ // $t = \lfloor \sqrt{n} \rfloor - r$
enquanto $n \geq (r+1)^2$ **faça**
  $r \leftarrow r + 1$
$\{\ \mathbf{Q} = r^2 \leq n < (r+1)^2\ \}$

- Solução #3:

$\{\ \mathbf{P} = (n \geq 0)\ \}$
$r, s \leftarrow 0, n+1$
$\{\ \mathbf{INV} = (r^2 \leq n < s^2)\ \}$ // $t = s-(r+1)$
enquanto $s \neq r+1$ **faça**
  $m \leftarrow \lfloor \frac{r+s}{2} \rfloor$
  se $m^2 \leq n$ **então**
    $r \leftarrow m$
  $\|\ n < m^2$ **então**
    $s \leftarrow m$
$\{\ \mathbf{Q} = r^2 \leq n < (r+1)^2\ \}$

- Solução #4:

$\{\ \mathbf{P} = (n \geq 0)\ \}$
$r, z, z' \leftarrow 0, 1, 3$
$\{\ \mathbf{INV} = (r^2 \leq n) \wedge (z = (r+1)^2) \wedge (z' = 2r+3)\ \}$ // $t = \lfloor \sqrt{n} \rfloor - r$
enquanto $n \geq z$ **faça**
  $z, z' \leftarrow z+z', z'+2$
  $r \leftarrow r+1$
$\{\ \mathbf{Q} = r^2 \leq n < (r+1)^2\ \}$

14.4 Considerações sobre Eficiência

A complexidade de tempo do algoritmo da Solução #1 é $\Theta(n-\sqrt{n}) = \Theta(n)$. A complexidade de tempo do algoritmo da Solução #2 é melhor, de $\Theta(\sqrt{n})$. A complexidade de tempo do algoritmo da Solução #3 é $\Theta(\log n)$, a melhor dentre as quatro. A complexidade de tempo do algoritmo da Solução #4 não difere daquela da Solução #2, pois para esta seção de análise de complexidade adotamos um modelo de computação no qual tanto somas quanto multiplicações são executadas em tempo constante. Mesmo sobre este modelo, talvez seja mais verossímil assumir que a constante associada ao algoritmo da Solução #4 ser menor na prática que aquela associada ao algoritmo da Solução #2.

14.5 Observações Importantes

■ **Observação 14.1** Considere o seguinte experimento mental. Suponha que o leitor viaje pelo tempo, retornando a um momento anterior ao contato com a apresentação do desenvolvimento do algoritmo da Solução #4. (Se está apenas passando os olhos pelos texto, sem ainda ter estudado o material, este é um excelente momento para fazer este exercício!) Considere que aquele exato algoritmo o fosse apresentado neste seu retorno ao passado. Por conveniência, repetimos o algoritmo:

```
// assuma n ≥ 0
r, z, z' ← 0, 1, 3
enquanto n ≥ z faça
    z, z' ← z + z', z' + 2
    r ← r + 1
// garante r = ⌊√n⌋
```

Apresente um argumento *adhoc* convincente sobre a correção. Se encontrar dificuldades, considere agora que o algoritmo foi apresentado com o invariante especificado. Utilizando o Método #2 de verificação de correção do comando iterativo, como fica a tarefa de provar a correção? Este experimento visa demonstrar, na prática, o benefício da declaração do invariante como parte integrante da especificação de um comando iterativo.

■ **Observação 14.2** O algoritmo da Solução #4 é um outro bom exemplo do "desenvolvimento guiado por instrumentos" (veja Observação 12.1).

14.6 Exercícios

Exercício 14.1 Tente desenvolver um algoritmo para o problema a partir do invariante

$$\mathbf{INV} = (r = \lfloor \sqrt{i} \rfloor) \wedge (0 \leq i \leq n)$$

e descreva a dificuldade encontrada. No caso de ter conseguido completar o desenvolvimento, contraste-a com as outras soluções desenvolvidas. Ela seria uma versão piorada de alguma delas?

Exercício 14.2 Prove formalmente que as tHs

$$\{\ \mathbf{INV} \wedge C\ \}\ R' = \emptyset\ \{\ \mathbf{INV}^r_{r-1}\ \}$$

$$\{\ \mathbf{INV} \wedge C\ \}\ R' = \emptyset\ \{\ \mathbf{INV}^r_{r+1}\ \}$$

que aparecem no desenvolvimento respectivamente das Soluções #1 e #2 são válidas.

Exercício 14.3 Suponha que, no contexto da Solução #3, o invariante

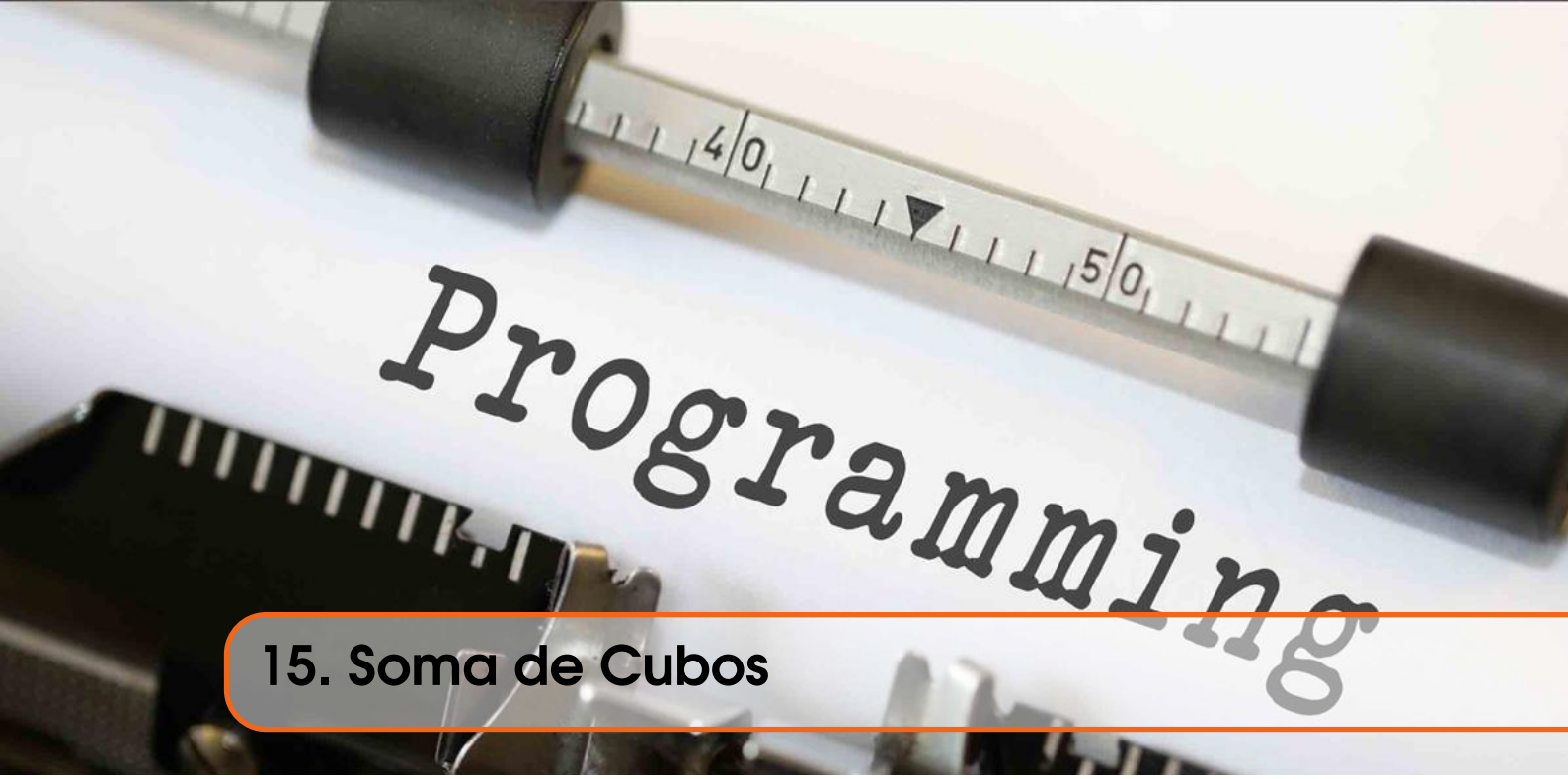
$$\mathbf{INV} = (r^2 \leq n < (s+1)^2)$$

fosse empregado. Desenvolva o restante do algoritmo com base neste invariante.

Exercício 14.4 Utilizando o Método #2 de verificação de correção do comando iterativo, proceda a verificação formal do algoritmo da Solução #1.

Exercício 14.5 Utilizando o Método #2 de verificação de correção do comando iterativo, proceda a verificação formal do algoritmo da Solução #3.

Exercício 14.6 Utilizando o Método #2 de verificação de correção do comando iterativo, proceda a verificação formal do algoritmo da Solução #4.



15. Soma de Cubos

15.1 Descrição

Dado natural n, atribuir à variável s o somatório dos cubos 1^3, 2^3, ..., n^3. Isto é, ao final do algoritmo, deve valer

$$s = \sum_{k=1}^{n} k^3$$

Como exemplo, se

$$n = 3$$

a valoração

$$s = 36$$

é aquela que satisfaz ao problema, uma vez que $1^3 + 2^3 + 3^3 = 36$.

Desenvolva seu algoritmo. Ao final, compare com aquele que será desenvolvido a seguir.

15.2 Desenvolvendo o Algoritmo

Em termos formais, procuramos elaborar um algoritmo A tal que a tH

$$\{ \mathbf{P} = (n \geq 0) \}\ A\ \left\{ \mathbf{Q} = \left(s = \sum_{k=1}^{n} k^3 \right) \right\}$$

seja válida. Um comando iterativo será empregado para se elaborar A, ou seja,

$A =$ *INI* { **INV** } **enquanto** C **faça** R;

Usando o método de substituir uma constante (n) por uma variável (i), deriva-se o invariante

$$\mathbf{INV} = \left(s = \sum_{k=1}^{i} k^3 \right) \wedge (0 \leq i \leq n)$$

Para tal invariante, a condição de parada e o comando de inicialização são, respectivamente,

$$C = \neg(i = n) = (i \neq n)$$

$$INI = s, i \leftarrow 0, 0$$

Em seguida, é necessário elaborar um algoritmo R que progrida em relação à terminação, enquanto mantém o invariante. Comparando-se a inicialização com a condição de parada, propomos

$$R = R' \; ; i \leftarrow i + 1$$

e, portanto, devemos agora desenvolver R' de modo que a tH

$$\{ \; \mathbf{INV} \wedge C \; \} \; R' \left\{ \; \mathbf{INV}^{i}_{i+1} = \left(s = \sum_{k=1}^{i+1} k^3 \right) \wedge (0 \leq i+1 \leq n) \right\}$$

seja válida, o que se consegue com

$$R' = s \leftarrow s + (i+1)^3$$

completando o algoritmo. Porém, o algoritmo será agora aprimorado, empregando-se a técnica de trocar uma computação com maior custo (multiplicação) por outra de menor custo (soma), análogo aquele praticado na Solução #4 do Capítulo 14. Note que, a cada iteração, duas operações de multiplicação são efetuadas ($(i+1) \times (i+1) \times (i+1)$). A técnica consiste em supor que tal valor estará previamente computado em alguma nova variável, digamos, z. Assim, fazendo

$$\mathbf{I} = \left(s = \sum_{k=1}^{i} k^3 \right) \wedge (0 \leq i \leq n)$$

redefine-se o invariante para

$$\mathbf{INV} = \mathbf{I} \wedge (z = i^3)$$

e, para tal invariante, o comando de inicialização é redefinido para

$$INI = s, i, z \leftarrow 0, 0, 0$$

Note que, agora, temos que

$$\mathbf{INV}^{i}_{i+1} = \mathbf{I}^{i}_{i+1} \wedge (z = (i+1)^3)$$

e, portanto, é necessário redefinir R' para também atualizar o valor de z para $(i+1)^3$. Deve-se empregar fatos conhecidos no invariante para que tal atualização tenha um menor custo computacional. Como

$$(i+1)^3 = i^3 + 3i^2 + 3i + 1$$

temos que

$$R' = z \leftarrow z + 3i^2 + 3i + 1 \; ; s \leftarrow s + z$$

finaliza o algoritmo corretamente, mas não resolve muito a motivação de diminuir as multiplicações. Indo adiante, definimos a expressão $3i^2+3i+1$ como "custosa", e a introduzimos no invariante, em uma nova variável z' onde se supõe armazenar tal valor. Resulta na redefinição

$$\mathbf{INV} = \mathbf{I} \wedge (z = i^3) \wedge (z' = 3i^2+3i+1)$$

cujo comando de inicialização correspondente passa a ser

$$INI = s,\, i,\, z,\, z' \leftarrow 0,\, 0,\, 0,\, 1$$

Como

$$\mathbf{INV}^i_{i+1} = \mathbf{I}^i_{i+1} \wedge (z = (i+1)^3) \wedge (z' = 3(i+1)^2+3(i+1)+1)$$

e

$$3(i+1)^2+3(i+1)+1 = (3i^2+3i+1)+(6i+6)$$

Temos que

$$R' = z,\, z' \leftarrow z+z',\, z'+6i+6\ ;\ s \leftarrow s+z$$

o que já reduz para apenas uma multiplicação em cada iteração. Mas ainda é possível fazer mais um ciclo de otimização, definimos a expressão $6i+6$ como "custosa", e a introduzimos no invariante, em uma nova variável z'' onde se supõe armazenar tal valor. Resulta na redefinição

$$\mathbf{INV} = \mathbf{I} \wedge (z = i^3) \wedge (z' = 3i^2+3i+1) \wedge (z'' = 6i+6)$$

cujo comando de inicialização correspondente passa a ser

$$INI = s,\, i,\, z,\, z',\, z'' \leftarrow 0,\, 0,\, 0,\, 1,\, 6$$

Como

$$\mathbf{INV}^i_{i+1} = \mathbf{I}^i_{i+1} \wedge (z = (i+1)^3) \wedge (z' = 3(i+1)^2+3(i+1)+1) \wedge (z'' = 6(i+1)+6)$$

e

$$6(i+1)+6 = (6i+6)+6$$

Temos que

$$R' = z,\, z',\, z'' \leftarrow z+z',\, z'+z'',\, z''+6\ ;\ s \leftarrow s+z$$

completa o algoritmo, eliminando todas as multiplicações.

15.3 Descrição do Algoritmo Desenvolvido

$\{\ \mathbf{P} = (n \geq 0)\ \}$
$s,\, i,\, z,\, z',\, z'' \leftarrow 0,\, 0,\, 0,\, 1,\, 6$
$\left\{\ \mathbf{INV} = \begin{array}{l} (s = \sum_{k=1}^{i} k^3) \wedge (0 \leq i \leq n) \wedge \\ (z = i^3) \wedge (z' = 3i^2+3i+1) \wedge (z'' = 6i+6) \end{array}\ \right\}$
enquanto $i \neq n$ **faça**
  $z,\, z',\, z'' \leftarrow z+z',\, z'+z'',\, z''+6$
  $s \leftarrow s+z$
  $i \leftarrow i+1$
$\{\ \mathbf{Q} = (s = \sum_{k=1}^{n} k^3)\ \}$

15.4 Considerações sobre Eficiência

A complexidade de tempo do algoritmo é $\Theta(n)$, sem o uso de operações de multiplicações.

15.5 Observações Importantes

■ **Observação 15.1** O algoritmo é um outro bom exemplo do "desenvolvimento guiado por instrumentos" (veja Observação 12.1).

15.6 Exercícios

Exercício 15.1 Utilizando o Método #2 de verificação de correção do comando iterativo, proceda a verificação formal do algoritmo desenvolvido.

Exercício 15.2 Elabore um algoritmo para, dado natural n, atribua a s o valor $\sum_{k=1}^{n} k^2$. Troque eventuais multiplicações por um número constante de somas, a exemplo do algoritmo apresentado neste capítulo.

Exercício 15.3 Elabore um algoritmo para, dado natural n e inteiros a, b, c, d, atribua a s o valor $\sum_{k=1}^{n} (ak^3 + bk^2 + ck + d)$. Troque eventuais multiplicações por um número constante de somas, a exemplo do algoritmo apresentado neste capítulo.

Exercício 15.4 Elabore um algoritmo para, dado natural n, atribua a s o valor $\sum_{k=1}^{n} k^4$. Troque eventuais multiplicações por um número constante de somas, a exemplo do algoritmo apresentado neste capítulo.

16. Avaliação de Polinômio

16.1 Descrição

Dados natural n, um vetor $a[0..n]$ representando o polinômio de grau n

$$f(x) = a_n x^n + a_{n-1}x^{n-1} + \cdots + a_1 x + a_0$$

onde $a_i = a[i]$, e um inteiro x, atribuir à variável r a avaliação de f em x, isto é, $r = f(x)$. Como exemplo, se

$$n = 4\ ,\ a = [-5,-1,2,1]\ ,\ x = 3$$

a valoração

$$r = 43$$

é aquela que satisfaz ao problema, uma vez que $f(3) = 1(3)^3 + 2(3)^2 - 1(3) - 5 = 43$.

Desenvolva seu algoritmo. Ao final, compare com aquele que será desenvolvido a seguir.

16.2 Desenvolvendo o Algoritmo

Em termos formais, procuramos elaborar um algoritmo A tal que a tH

$$\{\ \mathbf{P} = (n \geq 0)\ \}\ A\ \left\{\ \mathbf{Q} = \left(r = \sum_{k=0}^{n} a[k]x^k\right)\ \right\}$$

seja válida. Um comando iterativo será empregado para se elaborar A, ou seja,

$A =$ *INI* { **INV** } **enquanto** C **faça** R;

Usando o método de substituir uma constante (n) por uma variável (i), deriva-se o invariante

$$\mathbf{INV} = \left(r = \sum_{k=0}^{i} a[k]x^k\right) \wedge (0 \leq i \leq n)$$

Para tal invariante, a condição de parada e o comando de inicialização são, respectivamente,

$$C = \neg(i = n) = (i \neq n)$$

$$INI = r, i \leftarrow 0, 0$$

Em seguida, é necessário elaborar um algoritmo R que progrida em relação à terminação, enquanto mantém o invariante. Comparando-se a inicialização com a condição de parada, propomos

$$R = R' \,;\, i \leftarrow i+1$$

e, portanto, devemos agora desenvolver R' de modo que a tH

$$\{\ \mathbf{INV} \wedge C\ \}\ R'\ \left\{\ \mathbf{INV}^{i}_{i+1} = \left(r = \sum_{k=0}^{i+1} a[k]x^k\right) \wedge (0 \leq i+1 \leq n)\ \right\}$$

seja válida, o que se consegue com

$$R' = r \leftarrow r + a[i+1] * x^{i+1}$$

Empregando a técnica de pré-computar operações custosas no invariante, supomos que o valor x^i esteja armazenado na variável z, e revisamos o invariante para

$$\mathbf{INV} = \left(r = \sum_{k=0}^{i} a[k]x^k\right) \wedge (0 \leq i \leq n) \wedge (z = x^i)$$

e o respectivo comando de inicialização para

$$INI = r, i, z \leftarrow 0, 0, 1$$

O desenvolvimento de R' deve ser de modo que a tH

$$\{\ \mathbf{INV} \wedge C\ \}\ R'\ \left\{\ \mathbf{INV}^{i}_{i+1} = \left(r = \sum_{k=0}^{i+1} a[k]x^k\right) \wedge (0 \leq i+1 \leq n) \wedge (z = x^{i+1})\ \right\}$$

seja válida, o que se consegue com

$$R' = z \leftarrow z * x \,;\, r \leftarrow r + a[i+1] * z$$

completando o algoritmo (Solução #1). Esta solução emprega duas multiplicações por iteração, mas há como melhorar. Na próxima solução, também usa-se o método de substituir uma constante (0) por uma variável (i), que deriva o invariante (veja Observação 16.1)

$$\mathbf{INV} = \left(r = \sum_{k=i}^{n} a[k]x^{k-i}\right) \wedge (0 \leq i \leq n)$$

Para tal invariante, a condição de parada e o comando de inicialização são, respectivamente,

$$C = \neg(i = 0) = (i \neq 0)$$

$$INI = r, i \leftarrow a[n], n$$

Em seguida, é necessário elaborar um algoritmo R que progrida em relação à terminação, enquanto mantém o invariante. Comparando-se a inicialização com a condição de parada, propomos

$$R = R' \,;\, i \leftarrow i-1$$

e, portanto, devemos agora desenvolver R' de modo que a tH

$$\{\ \mathbf{INV} \wedge C\ \}\ R'\ \left\{\ \mathbf{INV}^{i}_{i-1} = \left(r = \sum_{k=i-1}^{n} a[k]x^{k-i+1} \right) \wedge (0 \leq i-1 \leq n)\ \right\}$$

seja válida, o que se consegue com

$$R' = r \leftarrow r * x + a[i-1]$$

completando o algoritmo (Solução #2). Este algoritmo constitui a famosa *regra* ou *esquema de Horner* para avaliação de polinômios.

16.3 Descrição do Algoritmo Desenvolvido

- Solução #1:

$\{\ \mathbf{P} = (n \geq 0)\ \}$
$r, i, z \leftarrow 0, 0, 1$
$\{\ \mathbf{INV} = \left(r = \sum_{k=0}^{i} a[k]x^k\right) \wedge (0 \leq i \leq n) \wedge (z = x^i)\ \}$ // $t = n - i$
enquanto $i \neq n$ **faça**
  $z \leftarrow z * x$
  $r \leftarrow r + a[i+1] * z$
  $i \leftarrow i + 1$
$\{\ \mathbf{Q} = \left(r = \sum_{k=0}^{n} a[k]x^k\right)\ \}$

- Solução #2:

$\{\ \mathbf{P} = (n \geq 0)\ \}$
$r, i \leftarrow a[n], n$
$\{\ \mathbf{INV} = \left(r = \sum_{k=i}^{n} a[k]x^{k-i}\right) \wedge (0 \leq i \leq n)\ \}$ // $t = i$
enquanto $i \neq 0$ **faça**
  $r \leftarrow r * x + a[i-1]$
  $i \leftarrow i - 1$
$\{\ \mathbf{Q} = \left(r = \sum_{k=0}^{n} a[k]x^k\right)\ \}$

16.4 Considerações sobre Eficiência

A complexidade de tempo de ambos os algoritmos é de $\Theta(n)$. Aquele da Solução 1 emprega duas multiplicações por iteração, enquanto aquele da Solução 2 apenas uma.

16.5 Observações Importantes

■ **Observação 16.1** Note que, para produzir o invariante da Solução 2, a pós-condição $\mathbf{Q}$ foi tomada como sendo

$$\mathbf{Q} = \left(r = \sum_{k=0}^{n} a[k]x^{k-0} \right)$$

e a substituição da constante 0 foi feita em ambas as ocorrências. Considerar a existência da

constante 0 no expoente de x é surpreendente, e também libertadora: abre a mente em relação às inúmeras possibilidades com que um invariante pode ser gerado. Com efeito, há inúmeras possibilidades de se gerar constantes implícitas, além daquelas explícitas, na expressão de **Q**. Na verdade, esta substituição de constantes implícitas por variáveis não é novidade: esta técnica já foi empregada para a Solução 2 do Capítulo 12. Observe que lá, por exemplo, duas constantes implícitas 1 foram substituídas por duas variáveis diferentes. Estes exemplos ilustram a diversidade de invariantes possíveis.

16.6 Exercícios

Exercício 16.1 Utilizando o Método #2 de verificação de correção do comando iterativo, proceda a verificação formal do algoritmo desenvolvido na Solução 1.

Exercício 16.2 Utilizando o Método #2 de verificação de correção do comando iterativo, proceda a verificação formal do algoritmo desenvolvido na Solução 2.

Exercício 16.3 Elabore um algoritmo para, dados natural n e inteiro x, atribua a r o valor $f(x,n)$, onde

$$f(x,n) = 1 + x + x^2 + \cdots + x^n$$

Para obter uma melhor complexidade, tente usar o fato que

$$f(x,n) = \begin{cases} f\left(x^2, \frac{n-1}{2}\right)(1+x) & \text{, se } n \text{ é ímpar} \\ f\left(x^2, \frac{n}{2}\right)(1+\frac{1}{x}) - \frac{1}{x} & \text{, se } n \text{ é par} \end{cases}$$

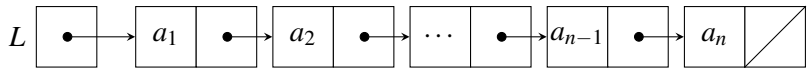
17. Listas Encadeadas

17.1 Descrição

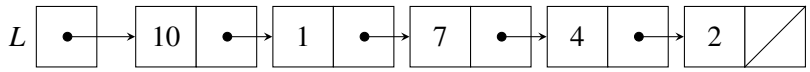
A metodologia do correção até aqui tratou de teoria e prática das variáveis escalares e vetor/matrizes. Programas reais, contudo, empregam estruturas de dados de mais alto nível, necessárias para se trabalhar com problemas complexos. Neste capítulo, trataremos de mostrar como a metodologia de correção pode ser estendida para uma fundamental estrutura de dados: as listas encadeadas.

Uma variável p do tipo *ponteiro* é uma variável que guarda um endereço de memória como conteúdo. Tal conteúdo consiste ou de um valor especial denotado por $\emptyset$, ou de um endereço de memória que armazena um dado, que é chamado de *valor apontado* e acessado pela sintaxe $p\uparrow$. Uma variável v do tipo *estrutura* é uma variável cujo dado armazenado é acessado de forma compartimentada, tal que a cada compartimento corresponde uma variável distinta definida interna à estrutura; se uma variável interna à estrutura é chamada de *campo*, o dado correspondente é acessado através da sintaxe *v.campo*.

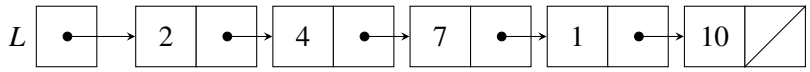
Listas encadeadas são baseadas nas noções de estruturas e ponteiros, e projetadas da seguinte maneira. Um *nó de lista encadeada* é uma estrutura que consiste das variáveis *valor* e *próx*. A primeira armazena um valor arbitrário de interesse de uma aplicação. A segunda é empregada para que os nós sejam sequenciados, de modo que cada nó indique o próximo nó que o segue. O valor $\emptyset$ é empregado para indicar a inexistência de um próximo nó. Uma *lista encadeada* é uma variável L do tipo ponteiro que ou possui o valor $\emptyset$ ou seu valor apontado é um nó de lista encadeada. Diz-se que a *sequência de valores de uma lista encadeada* L é uma sequência vazia se $L = \emptyset$, ou a sequência $a_1, a_2, \ldots, a_n$ tal que (i) a_1 é o dado armazenado no campo *valor* do nó que é valor apontado por L, e (ii) se a_i é o dado armazenado no campo *valor* de certo nó, então $i = n$ se o campo $próx = \emptyset$ ou, caso contrário, a_{i+1} é o dado armazenado no campo *valor* do próximo nó. As listas encadeadas possuem a representação gráfica clássica mostrada a seguir. Note que, graficamente, as variáveis estruturas são representadas por "caixas" de memória compartimentadas, representando suas variáveis internas, e as variáveis ponteiros são representadas por setas que indicam o valor apontado ou, no caso de inexistente, por uma diagonal cortando o conteúdo da variável ponteiro.



O problema que discutiremos é o seguinte. Dado uma lista encadeada L cuja sequência de valores seja $a_1, a_2, \ldots, a_n$ para algum $n \geq 0$, modificar L de modo que a sequência de valores de L passe a ser $a_n, a_{n-1}, \ldots, 1$. Como exemplo, se



a valoração

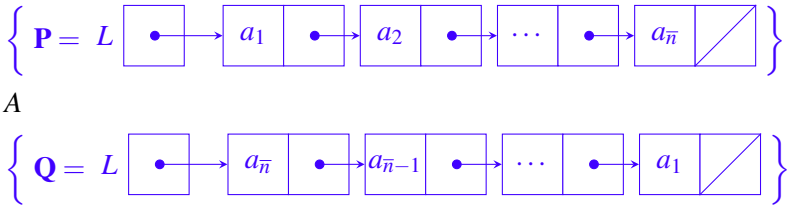


é aquela que satisfaz ao problema.

Desenvolva seu algoritmo. Ao final, compare com aquele que será desenvolvido a seguir.

17.2 Desenvolvendo o Algoritmo

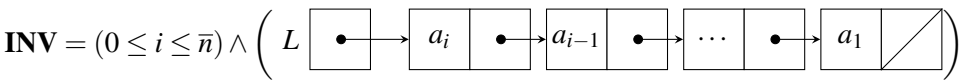
Em termos formais, procuramos elaborar um algoritmo A tal que a tH



seja válida. Um comando iterativo será necessário para se elaborar A, ou seja,

$A =$ *INI* { **INV** } **enquanto** C **faça** R;

Usando o método de substituir uma constante ($\overline{n}$) por uma variável (i), elaboramos o invariante



e a condição de parada e o comando de inicialização parecem ser, respectivamente,

$$C = \neg(i = \overline{n}) = (i \neq \overline{n})$$

$$INI = i, L \leftarrow 0, \emptyset$$

Contudo, há dois problemas a serem contornados:

1. A computação de C requer a ciência do valor de $\overline{n}$, que não é um dado disponível de entrada. Embora tal valor possa ser previamente determinado por um outro algoritmo, claramente aumentará a complexidade da solução. Há condição equivalente para C de fácil computação?
2. Ao se fazer o exercício de elaborar R de modo a progredir em relação à condição de parada, percebe-se logo que o invariante não estabelece onde estão armazenados os valores $a_{i+1}, a_{i+2}, \ldots, a_{\overline{n}}$, que serão necessários.

O problema #2 motiva a redefinição do invariante para

$$\mathbf{INV} = (0 \le i \le \overline{n}) \wedge \left(\begin{array}{l} L \bullet\!\!\rightarrow a_i \bullet\!\!\rightarrow a_{i-1} \bullet\!\!\rightarrow \cdots \bullet\!\!\rightarrow a_1 \,/ \\ S \bullet\!\!\rightarrow a_{i+1} \bullet\!\!\rightarrow a_{i+2} \bullet\!\!\rightarrow \cdots \bullet\!\!\rightarrow a_{\overline{n}} \,/ \end{array} \right) \wedge$$

E agora verifica-se que ambos os problemas são resolvidos por esta invariante, pois para tal invariante, a condição de parada e o comando de inicialização são, respectivamente,

$$C = \neg(i = \overline{n}) = \neg(S = \emptyset) = (S \neq \emptyset)$$

$$INI = i, L, S \leftarrow 0, \emptyset, L$$

Em seguida, é necessário elaborar um algoritmo R que progrida em relação à terminação, enquanto mantém o invariante. Comparando-se a inicialização com a condição de parada, propomos

$$R = R' \,;\, i \leftarrow i+1$$

e, portanto, devemos agora desenvolver R' de modo que a tH

$$\{\ \mathbf{INV} \wedge C\ \}$$

$$R'$$

$$\left\{ \mathbf{INV}^i_{i+1} = (0 \le i+1 \le \overline{n}) \wedge \left(\begin{array}{l} L \bullet\!\!\rightarrow a_{i+1} \bullet\!\!\rightarrow a_i \bullet\!\!\rightarrow \cdots \bullet\!\!\rightarrow a_1 \,/ \\ S \bullet\!\!\rightarrow a_{i+2} \bullet\!\!\rightarrow a_{i+3} \bullet\!\!\rightarrow \cdots \bullet\!\!\rightarrow a_{\overline{n}} \,/ \end{array} \right) \wedge \right\}$$

seja válida, o que se consegue com

$$R' = T \leftarrow S \,;\, S \leftarrow S{\uparrow}.prox \,;\, T{\uparrow}.prox \leftarrow L \,;\, L \leftarrow T$$

onde T é uma variável introduzida para armazenar um valor temporário. Isto completa o algoritmo. Finalmente, nota-se que a variável i é supérflua no algoritmo, e assim pode ser desprezada. A variável passa a corresponder a uma constante lógica no invariante.

17.3 Descrição do Algoritmo Desenvolvido

$$\left\{ \mathbf{P} = \; L \bullet\!\!\rightarrow a_1 \bullet\!\!\rightarrow a_2 \bullet\!\!\rightarrow \cdots \bullet\!\!\rightarrow a_{\overline{n}} \,/ \right\}$$

$$L, S \leftarrow \emptyset, L$$

$$\left\{ \mathbf{INV} = (0 \le i \le \overline{n}) \wedge \left(\begin{array}{l} L \bullet\!\!\rightarrow a_i \bullet\!\!\rightarrow a_{i-1} \bullet\!\!\rightarrow \cdots \bullet\!\!\rightarrow a_1 \,/ \\ S \bullet\!\!\rightarrow a_{i+1} \bullet\!\!\rightarrow a_{i+2} \bullet\!\!\rightarrow \cdots \bullet\!\!\rightarrow a_{\overline{n}} \,/ \end{array} \right) \wedge \right\}$$

// $t = n - i$

enquanto $S \neq \emptyset$ **faça**

  $T \leftarrow S$

  $S \leftarrow S{\uparrow}.prox$

  $T{\uparrow}.prox \leftarrow L$

  $L \leftarrow T$

$$\left\{ \mathbf{Q} = \; L \bullet\!\!\rightarrow a_{\overline{n}} \bullet\!\!\rightarrow a_{n-1} \bullet\!\!\rightarrow \cdots \bullet\!\!\rightarrow a_1 \,/ \right\}$$

17.4 Considerações sobre Eficiência

A complexidade de tempo do algoritmo desenvolvido é de $\Theta(n)$, executado em uma única leitura dos elementos de L e emprega memória auxiliar constante. Portanto, o algoritmo resultante é ótimo.

17.5 Observações Importantes

■ **Observação 17.1** Na determinação de C, é importante destacar a troca de uma condição lógica $(i = \overline{n})$ pela expressão $(S = \emptyset)$, equivalente à anterior no contexto do invariante. A troca é importante para que a expressão lógica seja de fácil computação.

■ **Observação 17.2** O problema de reversão de uma lista encadeada não requer nem a alocação de áreas de memória para novos nós, nem desalocação da memória dos existentes por desuso. Porém, ambas operações são muito comuns em problemas envolvendo listas encadeadas. Assim, os Exercícios 17.5 e 17.6 tratam o assunto, apresentando comandos específicos para tais fins.

17.6 Exercícios

Exercício 17.1 Elabore um algoritmo para que, dada uma lista encadeada L, determine o número de elementos de L.

Exercício 17.2 Elabore um algoritmo que, dada uma lista encadeada L, determine a soma e o produto dos elementos de L.

Exercício 17.3 Elabore um algoritmo que, dada uma lista encadeada L, determine as quantidades de elementos não-positivos e não-negativos de L.

Exercício 17.4 Elabore um algoritmo que, dados uma lista encadeada L e um valor c, atribua a uma variável ponteiro b o nó cujo valor seja igual a c ou, no caso da sua inexistência, atribua a b o valor $\emptyset$.

Exercício 17.5 Elabore um algoritmo que, dada uma lista encadeada L, crie uma outra lista encadeada S de modo que a sequência de valores de S seja uma permutação daquela de L ascendentemente ordenada. Para este algoritmo, portanto, será necessário a alocação de novos nós. Assuma que, se p é uma variável ponteiro, a criação de uma nova memória que armazena um nó de lista encadeada é feita através do comando

alocar(p)

fazendo parte deste comando tornar tal memória criada para o nó o valor apontado por p.

Exercício 17.6 Elabore um algoritmo que, dada uma lista encadeada L, remova todos os elementos de L, desalocando apropriadamente a memória dos nós. Assuma que a memória associada ao valor apontado por uma variável ponteiro p é apropriadamente desalocada através do comando

desalocar(p)

Exercício 17.7 Elabore um algoritmo que, dada uma lista encadeada L, remova todos os elementos duplicados de L, mantendo uma única cópia de cada elemento original. Os nós da lista encadeada que não forem mais utilizados devem ser apropriadamente desalocados.

Exercício 17.8 Para cada problema em grafos abaixo, pesquise e estude algoritmos que resolvem o problema. Em seguida, elabore o invariante que está implícita em cada algoritmo e verifique formalmente o algoritmo.

1. determinação do centro de uma árvore;
2. distância mais curta entre um vértice e todos os demais.

18. Árvore Geradora Mínima

18.1 Descrição

Neste capítulo, a metodologia do correção será estendida para uma fundamental e complexa estrutura de dados: os grafos.

Um grafo é uma estrutura que modela um conjunto de elementos e alguma relação de interesse entre pares de tais elementos. Por exemplo, um grafo pode representar uma rede social na qual o conjunto de elementos é o conjunto dos usuários desta rede e a relação de interesse é o conhecimento mútuo entre tais usuários, que é a informação mais básica de uma rede social. Formalmente, um *grafo* (V,E) consiste do conjunto de elementos V e um conjunto de relações E. Cada elemento de V é denominado *vértice* e cada elemento de E é denominado de *aresta*. Cada vértice é elemento arbitrário, enquanto cada aresta consiste de um par de vértices, denotado por um par não-ordenado (isto é, uma aresta, digamos, (u,v) será considerada a mesma que (v,u)).

Considere um grafo G que representa uma mapa de rotas de veículos de determinada região. Nele, cada vértice representa um ponto distinto de destino neste mapa, e cada aresta (u,v) representa que os pontos do mapa u e v estão interconectados por uma via de comunicação (uma rua, uma estrada, etc.). Na Figura 18.1, vê-se uma instância concreta de G, como ilustração. Emprega-se na figura uma notação gráfica usual de grafos, na qual os vértices são pontos do plano e cada aresta é representada por uma ligação entre os dois pontos associados aos vértices que constituem tal aresta.

É comum que os problemas acerca de mapas associem a cada aresta um valor, que servirá de dado de entrada para os problemas. Tais grafos, que associam valores às arestas, são chamados de *ponderados*. Por exemplo, considere a atribuição de valores às arestas dada na figura, representando algum tipo de custo de travessia por aquele caminho (como distância de percurso, ou tempo de percurso, ou custo de pedágios, etc.). Um problema de interesse clássico, por exemplo, é determinar para cada vértice o caminho de menor custo entre este vértice e todos os demais, onde o custo de um caminho corresponde à soma dos custos associados às arestas que o compõe. Um *caminho* de um grafo G é uma sequência $v_0, v_1, \ldots, v_k$ tal que $(v_i, v_{i+1}) \in E(G)$, para todo $i = 1, 2, \ldots, k-1$.

Um problema também clássico, mas cuja motivação é menos direta, é o problema da determinação de uma árvore geradora mínima, que discutiremos neste capítulo. Mas antes, introduziremos conceitos fundamentais acerta de grafos, que serão empregados a seguir.

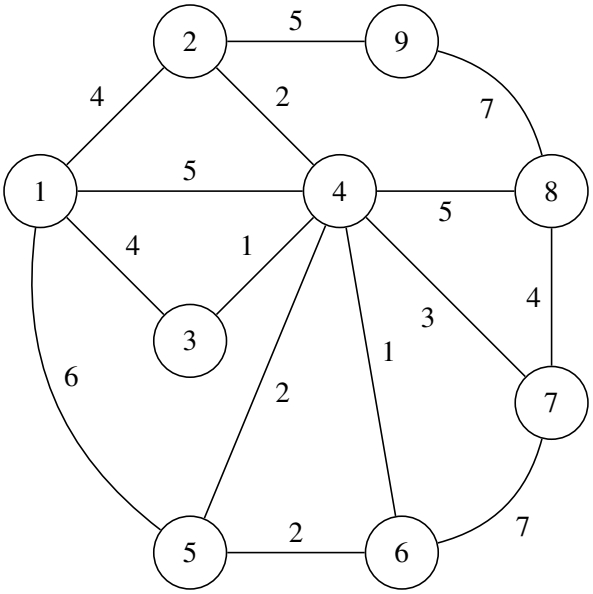


Figura 18.1: Exemplo de um grafo G ponderado.

Um grafo em que quaisquer dois de seus vértices estejam conectados por um caminho é chamado de *conexo*. Um *ciclo* de um grafo é um caminho fechado, isto é, um caminho $v_0, v_1, \ldots, v_k$ tal que $v_0 = v_k$ e $k \geq 3$. Um grafo *acíclico* é um grafo que não possui ciclos. Uma *árvore* é um grafo que seja conexo e acíclico. Dados dois grafos H, G, diz-se que $H \subseteq G$ se $V(H) \subseteq V(G)$ e $E(H) \subseteq E(G)$. Uma *árvore geradora* de um grafo G é uma árvore T tal que $T \subseteq G$ e $V(T) = V(G)$.

O problema da *árvore geradora mínima* (AGM) de dado grafo ponderado G é aquele de determinar uma árvore geradora de G de mínimo custo. O custo $c(T)$ de uma árvore T é a soma do custo associado às arestas de T, isto é,

$$c(T) = \sum_{e \in E(T)} c(e)$$

E, portanto, uma AGM T de G é tal que

$$T = \text{argmin}\{c(T) : T \text{ é árvore geradora de } G\}$$

Como exemplo, se o grafo G e os custos associados a cada aresta de G dados de entrada são aqueles da Figura 18.1, a árvore T apresentada na Figura 18.2, que consiste dos vértices e arestas em vermelho, satisfaz ao problema. Verifica-se que $c(T) = 22$ e não há árvore de custo menor.

Desenvolva seu algoritmo. Ao final, compare com aquele que será desenvolvido a seguir.

18.2 Desenvolvendo o Algoritmo

Em termos formais, procuramos elaborar um algoritmo A tal que a tH

$$\{\ \mathbf{P} = \mathbf{Conexo}(G)\ \}\ A\ \{\ \mathbf{Q} = \mathbf{AGM}(T, G)\ \}$$

seja válida, onde

$\mathbf{Conexo}(G) =$ "G é conexo"

$\mathbf{AGM}(T, G) =$ "T é uma árvore geradora mínima de G"

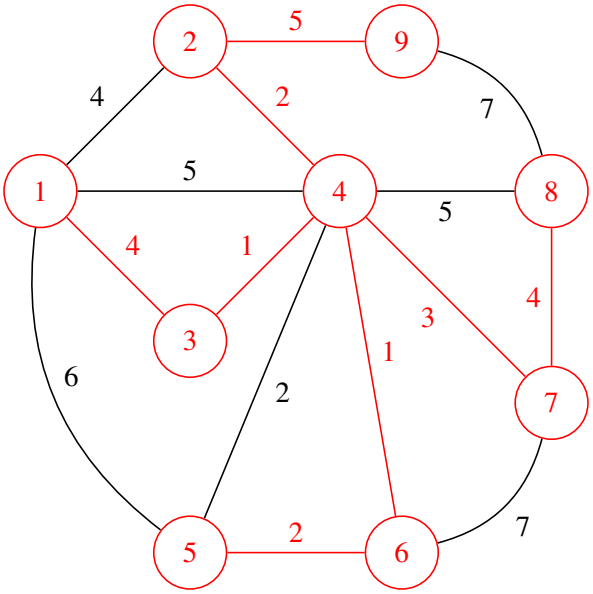


Figura 18.2: Exemplo de uma árvore geradora.

Um comando iterativo será necessário para se elaborar A, ou seja,

$$A = \mathit{INI}\ \{\ \mathbf{INV}\ \}\ \textbf{enquanto}\ C\ \textbf{faça}\ R;$$

Claramente, tal comando iterativo deve construir T incrementalmente. Assim, o invariante deve estabelecer que T seja parte de uma árvore geradora mínima de G. Isto pode ser formalmente descrito através de

$$\mathbf{INV} = (T \subseteq \overline{T}) \wedge \mathbf{AGM}(\overline{T}, G)$$

Note que T deve ser portanto uma floresta. Uma *floresta* é um grafo acíclico. Esta necessidade de T ser, durante toda a iteração, uma parte de uma árvore geradora de G, que é um grafo acíclico. A condição de parada e o comando de inicialização para tal invariante são, respectivamente,

$$C = \neg(T = \overline{T}) = \neg(n = |E(T)| + 1) = (n \neq |E(T)| + 1)$$

$$\mathit{INI} = T \leftarrow (V(G), \emptyset)$$

Em seguida, é necessário elaborar um algoritmo R que progrida em relação à terminação, enquanto mantém o invariante. Comparando-se a inicialização com a condição de parada, propomos

$$R = R'\ ;\ T \leftarrow T \cup \{e\}$$

onde e é uma variável nova que armazenará a próxima aresta a ser introduzida em T. Portanto, devemos agora desenvolver R' de modo que a tH

$$\{\ \mathbf{INV} \wedge C\ \}$$
$$R'$$
$$\left\{\ \mathbf{INV}^{T}_{T \cup \{e\}} = ((T \cup \{e\}) \subseteq \overline{T}) \wedge \mathbf{AGM}(\overline{T}, G)\ \right\}$$

seja válida. Assim, dado uma floresta T que está contida em uma AGM de G, é necessário atribuir a e uma aresta tal que $T \cup \{e\}$ também esteja contida em uma AGM de G. Naturalmente, $T \cup \{e\}$ deve ser floresta, isto é, um grafo acíclico. Entre as arestas que, quando adicionada a T, mantém o grafo acíclico, parece razoável verificar a possibilidade de ser aquela de menor custo. Com efeito, esta escolha é justificada pelo seguinte teorema.

Teorema 18.1 Sejam $\overline{T}$ uma AGM de um grafo G e $T \subset \overline{T}$. Se $e \in E(G)$ é uma aresta de menor custo tal que $T \cup \{e\}$ é floresta, então $T \cup \{e\}$ está contida em uma AGM de G.

Demonstração: Suponha, por absurdo, que $T' = T \cup \{e\}$ não esteja contida em nenhuma AGM de G. Seja $\overline{T}$ uma AGM de G em que T esteja contida. Seja $T'' = \overline{T} \cup \{e\}$. Seja $e = (u,v)$. Como u, v são vértices de $\overline{T}$, então em T'' há um ciclo C, no qual certamente e está contida. Como T' não contém ciclos, há certamente uma aresta e' de C que não esteja contida em T'. Tal aresta e' está contida, naturalmente, em $\overline{T}$. Seja $T''' = T'' \setminus \{e'\}$. Note os seguintes fatos:

- O ciclo C em T'' é único. Caso houvesse ciclo distinto C', então e também deve estar contido em C'. Assim, a união de C e C', removidos da aresta e, constituiria um ciclo em $\overline{T}$, o que seria um absurdo;
- T''' é acíclico, pois embora T'' contenha o único ciclo C, ele foi desfeito com a remoção de e';
- T''' é conexo, pois para todo caminho em $\overline{T}$ entre dois vértices que passava por e', há certamente outro que se utiliza as demais arestas de C.

Pelos dois últimos fatos, conclui-se que T''' é uma árvore geradora de G. Além disso,

$$\begin{aligned} c(T''') &= c(T'') - c(e') \\ &= c(\overline{T}) + c(e) - c(e') \\ &\leq c(\overline{T}) \end{aligned}$$

A última desigualdade vem do fato que $c(e) \leq c(e')$, pela escolha de e. Como $c(\overline{T})$ é mínimo, então $c(T''') = c(\overline{T})$. Logo, T''' é uma AGM de G. Como $T' \subset T'''$, então há uma contradição com a suposição que T' não esteja contida em nenhuma AGM. □

Portanto, o teorema anterior justifica que o algoritmo

$$R' = e \leftarrow \text{argmin}\{c(e) : e \in E(G) \mid T \cup \{e\} \text{ é floresta}\}$$

completa o algoritmo (Solução #1). Esta solução é conhecida como *algoritmo de Kruskal*.

Há uma outra solução bem conhecida para o problema, apresentada a seguir. Para ela, o invariante especifica que T será de um tipo mais restrito que floresta: T será uma árvore. Uma motivação para perseguir esta alternativa é que, do ponto de vista de complexidade, pode ser mais simples determinar uma próxima aresta que mantém T uma árvore do que manter T uma floresta. Mas a discussão do aspecto de complexidade será apresentada na seção correspondente. Prosseguindo o desenvolvimento do algoritmo, considere o invariante

$$\textbf{INV} = (T \subseteq \overline{T}) \wedge \textbf{Árv}(T) \wedge \textbf{AGM}(\overline{T}, G)$$

onde

$$\textbf{Árv}(T) = \text{“}T \text{ é uma árvore”}$$

A condição de parada e o comando de inicialização para tal invariante são, respectivamente,

$$C = \neg(T = \overline{T}) = \neg(|V(T)| = n) = (|V(T)| \neq n)$$

$INI = T \leftarrow (\{v\}, \emptyset)$, onde v é um vértice qualquer de G

Em seguida, é necessário elaborar um algoritmo R que progrida em relação à terminação, enquanto mantém o invariante. Comparando-se a inicialização com a condição de parada, propomos

$$R = R' \,;\, T \leftarrow T \cup \{e\}$$

onde e é uma variável nova que armazenará a próxima aresta a ser introduzida em T. Portanto, devemos agora desenvolver R' de modo que a tH

$$\{\ \mathbf{INV} \wedge C\ \}$$
$$R'$$
$$\left\{\ \mathbf{INV}^{T}_{T\cup\{e\}} = ((T \cup \{e\}) \subseteq \overline{T}) \wedge \textbf{Árv}(T \cup \{e\}) \wedge \mathbf{AGM}(\overline{T}, G)\ \right\}$$

seja válida. Assim, dado uma árvore T que está contida em uma AGM de G, é necessário atribuir a e uma aresta tal que $T \cup \{e\}$ também esteja contida em uma AGM de G. Entre as arestas que, quando adicionada a T mantém o grafo conexo e acíclico, parece novamente razoável verificar a possibilidade de ser aquela de menor custo. E novamente, esta escolha é correta.

Teorema 18.2 Sejam $\overline{T}$ uma AGM de um grafo G e $T \subset \overline{T}$ uma árvore. Se $e \in E(G)$ é uma aresta de menor custo tal que $T \cup \{e\}$ é uma árvore, então $T \cup \{e\}$ está contida em uma AGM de G.

Demonstração: Suponha, por absurdo, que $T' = T \cup \{e\}$ não esteja contida em nenhuma AGM de G. Seja $\overline{T}$ uma AGM de G em que T esteja contida. Seja $T'' = \overline{T} \cup \{e\}$. Seja $e = (u, v)$. Como u, v são vértices de $\overline{T}$, então em T'' há um ciclo C, no qual certamente e está contida. Como T' não contém ciclos, há certamente uma aresta $e' = (u', v')$ de C que não esteja contida em T'. Dentre as possíveis, há uma para a qual $u' \in V(T)$ e $v' \notin V(T)$. Tal aresta e' está contida, naturalmente, em $\overline{T}$. Seja $T''' = T'' \setminus \{e'\}$. A demonstração agora segue rigorosamente aquela do Teorema 18.1 a partir dos fatos que estão lá pontuados, e a obtenção consequente da mesma contradição. □

Portanto, o teorema anterior justifica que o algoritmo

$$R' = e \leftarrow \text{argmin}\{c(e) : e \in E(G) \mid \textbf{Árv}(T \cup \{e\})\}$$

completa o algoritmo (Solução #2). Esta solução é conhecida como *algoritmo de Prim*.

18.3 Descrição do Algoritmo Desenvolvido

- Solução #1 (Algoritmo de Kruskal):

$\{\ \mathbf{P} = \mathbf{Conexo}(G)\ \}$
$T \leftarrow (V(G), \emptyset)$
$\{\ \mathbf{INV} = (T \subseteq \overline{T}) \wedge \mathbf{AGM}(\overline{T}, G)\ \}$ // $t = |E(\overline{T} \setminus T)|$
enquanto $n \neq |E(T)| + 1$ **faça**
  $e \leftarrow \text{argmin}\{c(e) : e \in E(G) \mid T \cup \{e\}$ é floresta$\}$
  $T \leftarrow T \cup \{e\}$
$\{\ \mathbf{Q} = \mathbf{AGM}(T, G)\ \}$

- Solução #2 (Algoritmo de Prim):

$\{\ \mathbf{P} = \mathbf{Conexo}(G)\ \}$
$T \leftarrow (\{v\}, \emptyset)$, onde v é um vértice qualquer de G
$\left\{\ \mathbf{INV} = (T \subseteq \overline{T}) \wedge \mathbf{\acute{A}RV}(T) \wedge \mathbf{AGM}(\overline{T}, G)\ \right\}$ // $t = |E(\overline{T} \setminus T)|$
enquanto $|V(T)| \neq n$ **faça**
  $e \leftarrow \text{argmin}\{c(e) : e \in E(G) \mid \mathbf{\acute{A}RV}(T \cup \{e\})\}$
  $T \leftarrow T \cup \{e\}$
$\{\ \mathbf{Q} = \mathbf{AGM}(T, G)\ \}$

18.4 Considerações sobre Eficiência

As complexidades de tempo dos algoritmos desenvolvidos não são tão diretas de serem determinadas, como nos exemplos apresentados anteriormente. O problema reside no fato de que o algoritmo que parte da definição em como se determinar a aresta e, em ambos os casos, possuem uma complexidade que a intuição suspeita haver desperdício, levando-se em conta que a mesma operação é aplicada na iteração seguinte em um estado quase idêntico ao anterior. Sendo mais específico, pode-se afirmar com segurança que a complexidade de computar

$$e \leftarrow \text{argmin}\{c(e) : e \in E(G) \mid T \cup \{e\} \text{ é floresta}\}$$

é $O(mn)$. Isto se deve à implementação em que cada aresta $e \in E(G)$ é percorrida, e dentre aquelas em que se verifica que $T \cup \{e\}$ é floresta, registra-se aquela de menor custo. Por sua vez, a determinação desta última propriedade pode ser efetuada em $O(|V(T)| + |E(T)|) = O(n)$ através do bem conhecido algoritmo de busca em grafos, adaptado para a detecção de ciclos. Levando-se em conta que há $n-1$ iterações, a complexidade do algoritmo da Solução #1 resulta em $O(mn^2)$. A complexidade do algoritmo da Solução #2 também resulta em $O(mn^2)$ por raciocínio análogo. No entanto, tais complexidades podem ser sensivelmente reduzidas, refinando-se as estruturas de dados para a computação da determinação de e em cada iteração de maneira mais eficiente. Em seguida, trataremos de apresentar os respectivos refinamentos em ambas as soluções.

18.4.1 Refinamento da Solução #1 (Kruskal)

Observando-se o algoritmo, fica claro que uma forma de implementar a determinação de e é ordenar e percorrer as arestas por custo ascendente e, para cada aresta e sendo investigada, determinar se $T \cup \{e\}$ é floresta. Caso seja, a aresta é adicionada a T. No caso negativo, ela é ignorada e a próxima na ordenação é considerada. Isto resulta no refinamento a seguir.

$\{\ \mathbf{P} = \mathbf{Conexo}(G)\ \}$
$T \leftarrow (V(G), \emptyset)$
$E \leftarrow E(G)$; OrdenarPorCusto(E) ; $i \leftarrow 1$
$\{\ \mathbf{INV} = (T \subseteq \overline{T}) \wedge \mathbf{AGM}(\overline{T}, G)\ \}$ // $t = |E(\overline{T} \setminus T)|$
enquanto $|V(G)| \neq |E(T)| + 1$ **faça**
  $e, i \leftarrow E[i], i+1$
  se $T \cup \{e\}$ *é floresta* **então**
    $T \leftarrow T \cup \{e\}$
$\{\ \mathbf{Q} = \mathbf{AGM}(T, G)\ \}$

A ordenação das arestas por custo pode ser feita em $\Theta(m \log m) = O(m \log n^2) = \Theta(m \log n)$. Assim, a complexidade desta solução refinada é de $O(m \log n + mn) = O(mn)$, que já representa

uma melhoria. Mas ainda por ser melhorada. Note que a operação de maior custo é a verificação se $T \cup \{e\}$ é uma floresta. Para o refinamento de tal operação, será empregado uma estrutura de dados chamada de *união disjunta*, que representa um conjunto $\mathscr{C}$ de conjuntos disjuntos de naturais. Cada conjunto $C \in \mathscr{C}$ é identificado por um natural. As operações disponíveis nesta estrutura de dados são:

- DEFINIR$(\mathscr{C}, I)$: define a estrutura de união disjunta $\mathscr{C}$, de modo que cada elemento $i \in I$ seja um conjunto $\{i\}$ em $\mathscr{C}$, identificado pelo natural i;
- IDENT$(\mathscr{C}, x)$: obtém a identificação do conjunto que contém x;
- UNIR$(\mathscr{C}, r, s)$: une em $\mathscr{C}$ os conjuntos identificados por r e s e identifica este conjunto que representa a união disjunta com r ou s.

Esta estrutura de dados pode ser implementada de modo que a primeira operação seja executada em tempo $O(|I|)$ e as duas últimas em tempo $O(\log|\mathscr{C}|)$. Resta saber como tal estrutura de dados poderia ser útil para detectar se $T \cup \{e\}$ é uma floresta. Note que T, inicialmente, possui n componentes conexos distintos, cada um consistindo de um único vértice. Cada nova aresta introduzida em T faz com que dois componentes conexos de T se unam, reduzindo o número de tais componentes em uma unidade. Assim, pode-se empregar uma união disjunta $\mathscr{C}$ de modo que cada $C \in \mathscr{C}$ represente os vértices em uma mesma componente conexa de T, assumindo $V(G) = \{1, 2, \ldots, n\}$. Isto conduz ao novo refinamento a seguir.

$\{$ **P** $=$ **Conexo**(G) $\}$
$T \leftarrow (V(G), \emptyset)$
$E \leftarrow E(G)$; ORDENARPORCUSTO(E) ; $i \leftarrow 1$
DEFINIR$(\mathscr{C}, V(G))$
$\{$ **INV** $= (T \subseteq \overline{T}) \wedge$ **AGM**$(\overline{T}, G)$ $\}$ // $t = |E(\overline{T} \setminus T)|$
enquanto $|V(G)| \neq |E(T)| + 1$ **faça**
    $e, i \leftarrow E[i], i+1$
    Seja $e = (u, v)$; $r \leftarrow$ IDENT$(\mathscr{C}, u)$; $s \leftarrow$ IDENT$(\mathscr{C}, v)$
    se $r \neq s$ **então**
        $T \leftarrow T \cup \{e\}$
        UNIR$(\mathscr{C}, r, s)$
$\{$ **Q** $=$ **AGM**(T, G) $\}$

A complexidade de tempo deste refinamento resulta em $O(m \log n)$.

18.4.2 Refinamento da Solução #2 (Prim)

Para este refinamento, a estrutura de dados de *fila de prioridade* será empregada. Esta estrutura agrupa um conjunto de elementos, cada um associado a uma prioridade. As operações disponíveis nesta estrutura de dados são:

- DEFINIR$(\mathscr{F})$: define a estrutura de dados de fila de prioridade, inicialmente vazia;
- ADICIONAR$(\mathscr{F}, x, p)$: adiciona na fila de prioridade $\mathscr{F}$ o elemento x associado à prioridade p;
- REMOVER$(\mathscr{F})$: retorna o elemento associado à maior prioridade, e o remove da fila de prioridade $\mathscr{F}$. A maior prioridade é aquela associada ao menor natural (a prioridade igual a 0 será portanto a maior prioridade possível).

A fila de prioridade pode ser implementada de modo que todas as operações sejam executadas em tempo $O(|\mathscr{F}|)$. Para a determinação da próxima aresta e, é necessário encontrar a de menor custo dentre aquelas que possuam uma ponta em T e outra fora. Assim, pode-se organizar uma fila de prioridade com todas as arestas nesta condição, cujas respectivas prioridade são seus custos. Para manter tal estrutura, note que, quando uma nova aresta e for adicionada a T,

- todas as arestas w incidentes à extremidade de e que não pertence a T, tais que w não possui extremidade em T, devem ser adicionadas à fila;
- todas as arestas w incidentes à extremidade de e que não pertence a T, tais que w possui extremidade em T, deveriam ser removidas da fila. Porém, note que a única remoção existente na estrutura é por prioridade, o que não é o caso neste momento. A solução, assim, será ignorar a remoção das arestas e ignorá-la quando ela finalmente sair da fila.

Esta estratégia conduz ao seguinte refinamento.

$\{$ **P** $=$ **Conexo**(G) $\}$
$EmT[1..n] \leftarrow$ **F**
$T \leftarrow (\{v\},\emptyset)$, onde v é um vértice qualquer de G; $\quad EmT[v] \leftarrow$ **V**
DEFINIR$(\mathscr{F})$
para cada $e = (v,w) \in E(G)$ **faça**
  ADICIONAR$(\mathscr{F}, e, c(e))$
$\{$ **INV** $= (T \subseteq \overline{T}) \wedge$ **ÁRV**$(T) \wedge$ **AGM**$(\overline{T}, G)$ $\}$ // $t =$ $|E(\overline{T} \setminus T)|$
enquanto $|V(T)| \neq n$ **faça**
  $e \leftarrow$ REMOVER$(\mathscr{F})$; Seja $e = (u,v)$
  se $EmT[v] =$ **V então**
    $u, v \leftarrow v, u$
  se $EmT[v] =$ **F então**
    $T \leftarrow T \cup \{e\}$; $\quad EmT[v] \leftarrow$ **V**
    para cada $e = (v,w) \in E(G)$ **faça**
      se $EmT[w] =$ **F então**
        ADICIONAR$(\mathscr{F}, e, c(e))$
$\{$ **Q** $=$ **AGM**(T,G) $\}$

Cada aresta pode ser adicionada no máximo uma vez na fila e, portanto, o número de iterações não excede m. Como em cada iteração uma operação de remoção da fila é executada, esta operação possui custo total de $O(m \log m) = O(m \log n^2) = \Theta(m \log n)$. Por outro lado, cada novo vértice que é adicionado em T tem sua vizinhança percorrida e, para cada um deles, uma operação de adição na fila é potencialmente executada. Assim, esta parte do algoritmo possui complexidade de $O(\sum_{v \in V(G)}(d(v) \log m)) = O(\log m \sum_{v \in V(G)} d(v)) = O(m \log n^2) = \Theta(m \log n)$. A complexidade resultante da soma das duas complexidades anteriores é de $O(m \log n)$, a mesma complexidade daquela do refinamento do algoritmo de Kruskal.

18.5 Observações Importantes

■ **Observação 18.1** Na determinação de C, é importante destacar a troca de uma condição lógica $T = \overline{T}$ pela expressão $n = |E(T)| + 1$ na Solução #1 e a troca de $T = \overline{T}$ por $|V(T)| = n$ na Solução #2, equivalente às anteriores no contexto do invariante. A troca é importante para

que a expressão lógica seja de fácil computação.

■ **Observação 18.2** Note que propriedades mais complexas em relação aos grafos foram especificas com o uso de predicados, como foi o caso de **Conexo**(G), **AGM**(G) e **Árv**(T). Isto é importante para manter a descrição dos predicados onde eles se encontram simples, e eles são suficientes para se validar as implicações lógicas que são exigidas no processo de correção.

■ **Observação 18.3** Note que os algoritmos produzidos no refinamento são bem diferentes dos originais. No entanto, não se pode perder de vista que são, em essência, a mesma solução dos originais. Na apresentação do texto, não se incorporou o refinamento ao método formal da correção. Ao invés, proveu-se a parte argumentos de que os algoritmos refinados são equivalentes aos respectivos originais. Este balanço, entre método formal e argumentação complementar, é importante para que o método formal seja empregado com sucesso na prática. E um cenário para tal separação é este portanto: empregar o método formal de correção para produzir a *essência* do algoritmo, materializada na estratégia da solução, e prover argumentos complementares para justificar os *refinamentos das estruturas de dados* com objetivo de melhorar a eficiência. (Se esta omissão de incorporar todo o algoritmo refinado no método formal causar incômodo ao leitor, veja o Exercício 18.1.)

18.6 Exercícios

Exercício 18.1 Redefina o invariante de ambos os algoritmos refinados de modo a incorporar todo o algoritmo no método formal e, em seguida, verifique formalmente ambos os algoritmos.

19. Busca de Padrões em Cadeias

19.1 Descrição

Dados naturais n e m e dois vetores $T[1..n]$ e $P[1..m]$ de caracteres, procura-se determinar se a cadeia formada pelos caracteres $P[1..m]$ ocorre como subcadeia da cadeia formada pelos $T[1..n]$. Isto é, procura-se saber se existe $1 \leq i \leq n-m+1$ tal que

$$T[i..i+m-1] = P[1..m]$$

Em caso afirmativo, pede-se atribuir à variável i o valor que torna válida a igualdade acima. Caso contrário, a variável i deve ser atribuída a $n+1$. Como exemplo, se

$$n = 8\ ,\ m = 3\ ,\ T = [b,c,b,c,c,b,a,b]\ ,\ P = [b,c,c]$$

a valoração

$$i = 3$$

é aquela que satisfaz ao problema, uma vez que

$$T[3..5] = P[1..3]$$

Desenvolva seu algoritmo. Ao final, compare com aquele que será desenvolvido a seguir.

19.2 Desenvolvendo o Algoritmo

Em termos formais, procuramos elaborar um algoritmo A tal que a tH

$$\{\ \mathbf{P} = (n \geq 0) \wedge (m \geq 0)\ \}$$
$$A$$
$$\{\ \mathbf{Q} = ((1 \leq i \leq n) \Rightarrow (P[1..m] = T[i..i+m-1])) \wedge ((i = n+1) \Rightarrow (P[1..m] \not\subseteq T[1..n]))\ \}$$

seja válida. Note que a pós condição é precisa, mas de alguma forma, a pós-condição é de difícil leitura pois é longa. Porém, note que a primeira metade dela diz respeito a um resultado que deve ser satisfeito em uma dada condição (a saber, se o padrão existe no texto) e a segunda metade em outra condição (se o padrão não for encontrado no texto). Isto é relativamente comum, e colocar as duas condições na mesma linha é a razão para que a proposição fique um pouco menos clara. Assim, pode-se pensar em apresentar a pós-condição de uma forma que favoreça o entendimento. Assim, ela será apresentada por

$$\mathbf{Q} = \begin{cases} P[1..m] = T[i..i+m-1] & \text{, se } 1 \leq i \leq n \\ P[1..m] \not\subseteq T[1..n] & \text{, se } i = n+1 \end{cases}$$

Um comando iterativo será empregado para se elaborar A, ou seja,

$$A = \ INI \ \{ \ \mathbf{INV} \ \} \ \textbf{enquanto } C \textbf{ faça } R;$$

Seja $\bar{i}$ o menor valor de i que torna válido $\mathbf{Q}$. Ou seja, sendo

$$\bar{i} = \min\{i \in [1..n+1] \mid \begin{cases} P[1..m] = T[i..i+m-1] & \text{, se } 1 \leq i \leq n \\ P[1..m] \not\subseteq T[1..n] & \text{, se } i = n+1 \end{cases} \}$$

Esta constante tem a vantagem de simplificar a pós-condição, que mais uma vez, será redefinida por

$$\mathbf{Q} = (i = \bar{i})$$

Usando o método de ampliar o domínio de uma variável (i), deriva-se o invariante

$$\mathbf{INV} = (1 \leq i \leq \bar{i}) \wedge (P[1..j] = T[i..i+j-1]) \wedge (0 \leq j \leq m)$$

Para tal invariante, a condição de parada e o comando de inicialização são, respectivamente,

$$C = \neg((j = m) \vee (i = n+1)) = (j \neq m) \wedge (i \neq n+1)$$

$$INI = i, \ j \leftarrow 1, 0$$

Em seguida, é necessário elaborar um algoritmo R que progrida em relação à terminação, enquanto mantém o invariante. Comparando-se a inicialização com a condição de parada, vê-se que é necessário incrementar j ou r a cada iteração. Isto sugere o emprego de um comando condicional, e define-se R como

$$R = \textbf{se } C_1 \textbf{ então } (R_1 \ ; \ j \leftarrow j+1) \ \| \ C_2 \textbf{ então } (R_2 \ ; \ i \leftarrow i+1)$$

e, portanto, devemos agora desenvolver C_1, R_1, C_2, R_2 de modo que as tHs

$$\{ \ \mathbf{INV} \wedge C \wedge C_1 \ \} \ R_1 \ \left\{ \ \mathbf{INV}^j_{j+1} = (1 \leq i \leq \bar{i}) \wedge (P[1..j+1] = T[i..i+j]) \wedge (0 \leq j+1 \leq m) \ \right\}$$

$$\{ \ \mathbf{INV} \wedge C \wedge C_2 \ \} \ R_2 \ \{ \ \mathbf{INV}^i_{i+1} = (1 \leq i+1 \leq \bar{i}) \wedge (P[1..j] = T[i+1..i+j]) \wedge (0 \leq j \leq m) \ \}$$

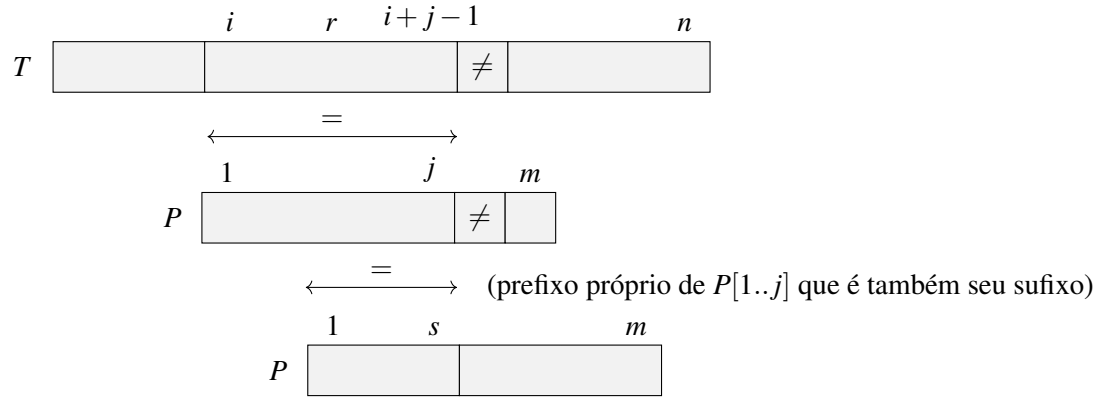
sejam válidas, o que se consegue com

$$C_1 = (P[j+1] = T[i+j]) \ , \quad R_1 = \emptyset$$

$$C_2 = (i+j > n) \vee (P[j+1] \neq T[i+j]) \ , \quad R_2 = j \leftarrow 0$$

completando o algoritmo (Solução #1). A determinação, contudo, de C_2, R_2 pode levantar uma questão interessante: há alguma maneira de atribuir um valor correto para j sem empregar o trivial

"$j \leftarrow 0$"? Note que, ao executar R_2, já se conhece todos os caracteres $T[i..i+j-1]$ e, sendo assim, ao incrementar i, talvez se possa determinar um valor de j maior que 0. Ou, ainda de maneira mais geral, talvez seja possível incrementar i por mais de uma unidade, e ainda escolher um valor compatível correspondente para j maior do que 0. Se isto for possível, isto sem dúvida traria benefícios ao tempo de execução. Veja a figura abaixo e, em seguida, detalharemos tal estratégia.



Os vetores mais acima e no meio da figura corresponde aos vetores T e P no invariante, que estabelece que as porções $T[i..i+j-1]$ e $P[1..j]$ são iguais. A situação que foi levantada anteriormente ocorre quando $P[j+1] \neq T[i+j]$, como indicado na figura por um símbolo "$\neq$" nas respectivas posições dos vetores. Na solução anterior, avança-se i de uma unidade e atribui-se $j = 0$, o que certamente é o mais seguro de se fazer sem maior análise. A proposta desta segunda solução é verificar em que condições é possível atribuir $i = r$, avançando i de uma ou mais posições, e atribuir $j = s \geq 0$. Assumindo que tais atribuições ocorram, o vetor mais abaixo na figura corresponde a P realinhando-se da posição i para a posição r de T. Note que, como já se conhece, neste ponto do algoritmo, todos os caracteres $T[r..i+j-1]$, procuramos avançar j para uma posição s tal que $P[1..s] = T[r..i+j-1]$, que é o máximo de informação que se dispõe. Comparando-se a relação que se impõe do vetor P anterior e o vetor P atualizado, percebe-se que é necessário que $P[1..s] = P[j-s+1..j]$. Note que $P[1..s]$ é um prefixo próprio de $P[1..j]$, enquanto $P[j-s+1..j]$ é um sufixo de $P[1..j]$. É possível que haja diversos valores possíveis para s de realinhamento, isto é, prefixos próprios de $P[1..j]$, de diferentes comprimentos, que também são sufixos de $P[1..j]$ (ver Exercício 19.1). Por este motivo, para se garantir que o avanço mantenha o invariante (em particular, mantenha a condição $i \leq \bar{i}$), é necessário que o avanço seja o menor daqueles possíveis, isto é, que s corresponda ao comprimento do maior prefixo próprio de $P[1..j]$ que seja também seu sufixo.

Assim, a próxima solução adotará um comando de composição, com o objetivo intermediário de pré-processar, para cada valor possível de j, tais comprimentos do maior prefixo próprio de $P[1..j]$ que seja também seu sufixo, de modo que tal informação seja utilizada no algoritmo, conforme análise anterior. Portanto, definindo-se a função

$$\mathrm{PS}(j) = \text{"comprimento do maior prefixo próprio de } P[1..j] \text{ que seja também sufixo de } P[1..j]\text{"}$$

procuramos por algoritmos A_1, A_2 que satisfaçam

$$\{\ \mathbf{P} = (n \geq 0) \wedge (m \geq 0)\ \}$$
$$A_1$$
$$\{\ \mathbf{PProc} = \mathbf{P} \wedge (\forall\, j \in [1..m-1] : V[j] = \mathrm{PS}(j))\ \}$$
$$A_2$$
$$\{\ \mathbf{Q} = (i = \bar{i})\ \}$$

O algoritmo A_2 é análogo ao algoritmo da Solução #1, exceto pela redefinição de R para

$$R = \textbf{se } C_1 \textbf{ então } (R_1 \ ; \ j \leftarrow j+1) \ \| \ C_2 \textbf{ então } R_2$$

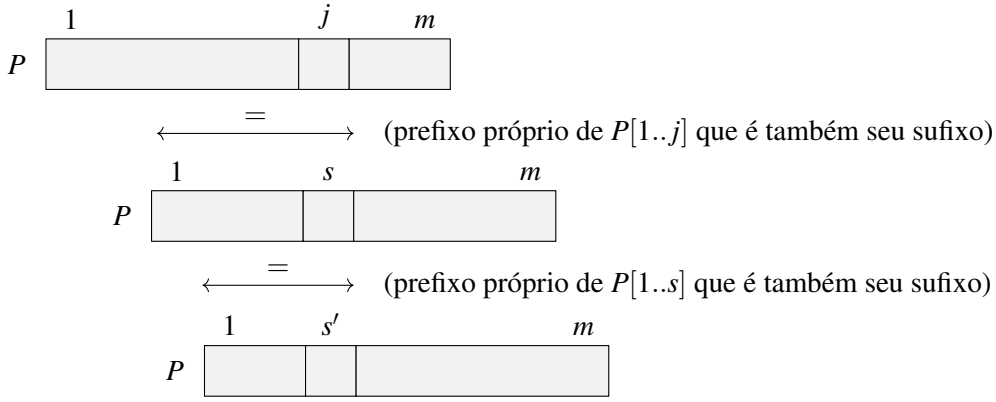
onde C_1, R_1, C_2 são os mesmos de anterior, enquanto R_2 é redefinido para

$$R_2 = j, i \leftarrow V[j], i + j - V[j]$$

conforme a nova estratégia de avanço de i e atualização de j apresentada (ver Exercício 19.4).

Resta então elaborar A_1. Para tanto, será necessário caracterizar quando certo valor de s é igual a $P(j)$. Veja os dois primeiros da próxima figura. Pelo desenho, fica claro que, se $s = PS(j)$, então:

- $P[s] = P[j]$, e
- $P[1..s-1]$ é prefixo próprio de $P[1..j-1]$, isto é, $P[1..s-1] = P[j-s+1..j-1]$.



Pela observação de que qualquer prefixo próprio $P[1..s-1]$ de $P[1..j-1]$ que seja seu sufixo tal que $P[s] = P[j]$ serve para compor um prefixo próprio $P[1..s]$ de $P[1..j]$ que seja seu sufixo, conclui-se que $s = PS(j)$ se $P[1..s-1]$ é o maior prefixo próprio de $P[1..j-1]$ que seja seu sufixo e *para o qual* $P[s] = P[j]$. Esta restrição em destaque é importante para entender que não necessariamente $s - 1 = PS(j-1)$.

Usando o método de ampliar o domínio de uma variável (i), deriva-se o invariante

$$\begin{aligned}\textbf{INV} =& \textbf{P} \wedge (\forall\, x \in [1..j-1] : V[x] = \mathrm{PS}(x)) \wedge (V[j] \geq PS(j)) \\ & \wedge (P[1..V[j]-1] = P[j - V[j] + 1..j-1])\end{aligned}$$

Para tal invariante, a condição de parada e o comando de inicialização são, respectivamente,

$$\begin{aligned}C &= \neg((j = m) \wedge ((V[j] = 0) \vee (P[V[j]] = P[j]))) \\ &= (j \neq m) \vee ((V[j] \neq 0) \wedge (P[V[j]] \neq P[j]))\end{aligned}$$

$$INI = j \leftarrow 1 \ ; V[j] \leftarrow 0$$

Em seguida, é necessário elaborar um algoritmo R que progrida em relação à terminação, enquanto mantém o invariante. Comparando-se a inicialização com a condição de parada, vê-se que é necessário incrementar j ou decrementar $V[j]$ a cada iteração. Isto sugere o emprego de um comando condicional, e define-se R como

$$R = \textbf{se } C_1 \textbf{ então } R_1 \ \| \ C_2 \textbf{ então } R_2$$

onde R_1 será encarregado de incrementar j, enquanto R_2 de decrementar $V[j]$. Pela análise anterior, temos que

$$C_1 = (j \neq m) \wedge ((V[j] = 0) \vee (P[V[j]] = P[j])) \;, \quad R_1 = j \leftarrow j+1 \;;\; V[j] \leftarrow V[j-1]+1$$
$$C_2 = (V[j] \neq 0) \wedge (P[V[j]] \neq P[j]) \;, \quad R_2 = V[j] \leftarrow V[V[j]]$$

Note que, de fato,

$$C \Rightarrow C_1 \vee C_2$$

(ver Exercício 19.6.) Isto completa o algoritmo (Solução #2), conhecido como *algoritmo KMP*, iniciais dos três autores que o descobriram: Knuth, Morris, Pratt.

19.3 Descrição do Algoritmo Desenvolvido

- Solução #1:

$\{\ \mathbf{P} = (n \geq 0) \wedge (m \geq 0)\ \}$
$i,\ j \leftarrow 1,\ 0$
$\{\ \mathbf{INV} = (1 \leq i \leq \bar{i}) \wedge (P[1..j] = T[i..i+j-1]) \wedge (0 \leq j \leq m)\ \}$ // $t = n - i$
enquanto $(j \neq m)$ ***e*** $(i \neq n+1)$ **faça**
  se $P[j+1] = T[i+j]$ **então**
    $j \leftarrow j+1$
  $\|$ $(i+j > n)$ ***ou*** $(P[j+1] \neq T[i+j])$ **então**
    $j \leftarrow 0$
    $i \leftarrow i+1$
$\{\ \mathbf{Q} = (i = \bar{i})\ \}$

onde

$$\bar{i} = \min\{i \in [1..n+1] \mid \left\{ \begin{array}{ll} P[1..m] = T[i..i+m-1] & \text{, se } 1 \leq i \leq n \\ P[1..m] \not\subseteq T[1..n] & \text{, se } i = n+1 \end{array} \right. \}$$

- Solução #2 (Algoritmo KMP):

$\{\ \mathbf{P} = (n \geq 0) \wedge (m \geq 0)\ \}$
$j \leftarrow 1\ ;\ V[j] \leftarrow 0$
$\left\{ \mathbf{INV} = \begin{array}{l} \mathbf{P} \wedge (\forall\, x \in [1..j-1] : V[x] = \mathrm{PS}(x)) \wedge (V[j] \geq PS(j)) \\ \wedge (P[1..V[j]-1] = P[j-V[j]+1..j-1]) \end{array} \right\}$ // $t = (m-j, V[j])$
enquanto $(j \neq m)$ ***ou*** $((V[j] \neq 0)$ ***e*** $(P[V[j]] \neq P[j]))$ **faça**
  se $(j \neq m)$ ***e*** $((V[j] = 0)$ ***ou*** $(P[V[j]] = P[j]))$ **então**
    $j \leftarrow j+1\ ;\ V[j] \leftarrow V[j-1]+1$
  $\|$ $(V[j] \neq 0)$ ***e*** $(P[V[j]] \neq P[j])$ **então**
    $V[j] \leftarrow V[V[j]]$
$\{\ \mathbf{PProc} = \mathbf{P} \wedge (\forall\, j \in [1..m-1] : V[j] = \mathrm{PS}(j))\ \}$
$i,\ j \leftarrow 1,\ 0$
$\{\ \mathbf{INV} = (1 \leq i \leq \bar{i}) \wedge (P[1..j] = T[i..i+j-1]) \wedge (0 \leq j \leq m)\ \}$ // $t = n - i$
enquanto $(j \neq m)$ ***e*** $(i \neq n+1)$ **faça**
  se $P[j+1] = T[i+j]$ **então**
    $j \leftarrow j+1$
  $\|$ $(i+j > n)$ ***ou*** $(P[j+1] \neq T[i+j])$ **então**
    $j,\ i \leftarrow V[j],\ i+j-V[j]$
$\{\ \mathbf{Q} = (i = \bar{i})\ \}$

onde

$\mathrm{PS}(j) =$ "comprimento do maior prefixo próprio de $P[1..j]$ que seja também sufixo de $P[1..j]$"

19.4 Considerações sobre Eficiência

A complexidade de tempo de do algoritmo da Solução #1 é de $O(n^2)$. Isto ocorre pois, no caso do padrão não ocorrer, todas as n posições iniciais serão investigadas e, para cada uma, pode ser que haja $\Theta(n)$ comparações entre os caracteres do padrão e do texto, sem que haja incremento de i, supondo que $m = \Theta(n)$ (ver Exercício 19.5). Por outro lado, a complexidade daquele da Solução 2 é de $\Theta(n)$, justificado da seguinte forma. (...)

19.5 Observações Importantes

■ **Observação 19.1** Na Solução #2, na determinação de R_2, o valor de i não é tão trivial de ser determinado. É um exemplo onde o emprego da verificação da correção do comando de atribuição, em conjunto com o uso de incógnita, pode tornar a determinação dos valores corretos a serem atribuídos uma tarefa menos árdua. O Exercício 19.4 pede que tal estratégia seja empregada para confirmar o valor utilizado no algoritmo.

■ **Observação 19.2** O algoritmo KMP é particularmente de difícil explicação. Como exercício, procure outras fontes em que o algoritmo é apresentando, prestando particular atenção à argumentação empregada. O nível de confiança da explicação se aproxima daquela suportada pela verificação da correção? A explicação alternativa poderia ser utilizada para *produzir* o algoritmo, ou apenas para *justificar* o algoritmo pronto?

19.6 Exercícios

Exercício 19.1 Elabore um exemplo de padrão para mostrar que, para certo valor e j, pode haver diversos prefixos próprios de $P[1..j]$ que são sufixos de $P[1..j]$.

Exercício 19.2 Execute passo-a-passo os algoritmos apresentados para a entrada

$$n = 16 \quad T = [A,B,B,A,A,C,A,A,D,A,A,B,A,A,B,A]$$

$$m = 4 \quad P = [A,A,B,A]$$

Exercício 19.3 Note que, no Exercício 19.2, o padrão P é encontrado na posição $i = 10$ da cadeia T. O padrão também ocorre na posição 13, mas ele não é encontrado pois o algoritmo é projetado para parar na primeira ocorrência do padrão. Modifique ambos os algoritmos para que eles determinarem todas as posições iniciais em que o padrão ocorra. Mais especificamente, considere atribuir a uma variável p o número de ocorrências do padrão. Além disso, na posição $k = 1,2,\ldots,p$ de um vetor $i[1..n]$, atribuir o início da k-ésima ocorrência de P em T. Para o

exemplo do Exercício 19.2, o algoritmo deveria produzir $p = 2$ e $i[1] = 10$ e $i[2] = 13$.

Exercício 19.4 Na Solução #2, na determinação de R_2, o valor de i não é tão direto e exige alguma reflexão. Empregue uma incógnita no método de atribuição de modo que a própria verificação de correção confirme o valor utilizado na atribuição. Note que o valor de $i + j$ é um invariante, isto é, é igual a um mesmo valor antes e depois da atribuição. Sendo assim, procurar determinar um valor apropriado para a incógnita $\square$ de modo que

$$\{\ i + j = \overline{\alpha}\ \}\ j, i \leftarrow V[j], \square\ \{\ i + j = \overline{\alpha}\ \}$$

Exercício 19.5 Mostre uma instância, em função do valor de n, que comprove a complexidade de pior caso do primeiro algoritmo de $\Theta(n^2)$.

Exercício 19.6 Prove formalmente que $C \Rightarrow C_1 \vee C_2$ no contexto da Solução #2.