Appendix

Algorithms and Networking for Computer Games Jouni Smed and Harri Hakonen
 ISBN: 0-470-01812-7

A

Pseudo-code Conventions

We describe the algorithms using a pseudo-code format, which, for most parts, follows the guidelines set by Cormen *et al*. (2001) closely. The conventions are based on common data abstractions (e.g. sets, sequences, and graphs) and the control structures resemble Pascal programming language (Jensen *et al*. 1985). Since our aim is to unveil the algorithmic ideas behind the implementations, we present the algorithms using pseudo-code instead of an existing programming language. This choice is backed by the following reasons:

- Although common programming languages (e.g. C, C++, Java, and scripting languages) share similar control structures, their elementary data structures differ significantly both on interface level and language philosophy. For example, if an algorithm uses a data structure from STL of C++, a Java programmer has three alternatives: reformulate the code to use Java's collection library, write a custom-made data structure to serve the algorithm, or buy a suitable third-party library. Apart from the last option, programming effort is unavoidable, and by giving a general algorithmic description we do not limit the choices on how to proceed with the implementation.

- Software development should account for change management issues. For instance, sometimes understandability of a code segment is more important than its efficiency. Because of these underlying factors affecting software development, we content ourselves with conveying the idea as clearly as possible and leaving the implementation selections to the reader.

- The efficiency of a program depends on the properties of its input. Often, code optimizations favouring certain kinds of inputs lead to 'pessimization', which disfavours other kinds of inputs. In addition, optimizing a code that is not the bottleneck of the whole system wastes development time, because the number of code lines increases and they are also more difficult to test. Because of these two observations, we give only a general description of a method that can be moulded so that it suits the reader's situation best.

Algorithms and Networking for Computer Games Jouni Smed and Harri Hakonen
 ISBN: 0-470-01812-7

- The implementation of an algorithm is connected to its software context not only through data representation but also through control flow. For example, time-consuming code segments are often responsible for reporting their status to a monitoring sub-system. This means that algorithms should be modifiable and easy to augment to respond to software integration forces, which tend to become more tedious when we are closer to the actual implementation language.
- Presenting the algorithms in pseudo-code has also a pedagogic rationale. There are two opposing views on how a new algorithm should be taught: First, the teacher describes the overall behaviour of the algorithm (i.e. its substructures and their relations), which often requires explanations in a natural language. Second, to guide on how to proceed with the implementation, the teacher describes the important details of the algorithm, which calls for a light formalism that can be easily converted to a programming code. The teacher's task is to find a balance between these two approaches. To support both approaches, the pseudo-code formalism with simple data and control abstractions allows the teacher to explain the topics in a natural language when necessary.

The pseudo-code notation tries to obey modern programming guidelines (e.g. avoiding global side effects). To clearly indicate what kind of an effect the algorithm has in the system, we have adopted the functional programming paradigm, where an algorithm is described as a function that does not mutate its actual parameters, and side effects are allowed only in the local structures within a function. For this reason, the algorithms are designed so that they are easy to understand – which sometimes means compromising on efficiency that could be achieved using the imperative programming paradigm (i.e. procedural programming with side effects). Nevertheless, immutability does not mean inefficiency but sometimes it is the key to manage object aliasing (Hakonen *et al*. 2000) or efficient concurrency (Hudak 1989). Immutability does not cause an extra effort in the implementation phase, because a functional description can be converted to a procedural one just by leaving out copy operations. The reader has the final choice on how to implement algorithms efficiently using the programming language of his or her preference.

Let us take an example of the pseudo-code notation. Assume that we are interested in changing a value so that some fraction α of the previous change also contributes to the outcome. In other words, we want to introduce inertia-like property to the change in the value of a variable. This can be implemented as linear momentum: If a change c affects a value v_t at time t, the outcome v_{t+1} is calculated using Equation (A.1).

$$v_{t+1} = v_t + c + \alpha(v_t - v_{t-1}) \iff \Delta v_{t+1} = c + \alpha \Delta v_t. \tag{A.1}$$

$\alpha \in [0, 1]$ is called a momentum coefficient and $\alpha \Delta v_t$ is a momentum term. To keep a record of the generated value, the history can be stored as a tail-growing sequence ⟨the first value, the second value, ..., the most recent value⟩. Algorithm A.1 describes this method as a function in the pseudo-code format.

If the use context of Algorithm A.1 assigns the returned sequence back to the argument variable, for example,

1: $V \leftarrow$ LINEAR-MOMENTUM(V, c, α)

the copying in line 1 can be omitted by allowing a side effect to the sequence V.

Algorithm A.1 Updating a value with a change value and a momentum term.

LINEAR-MOMENTUM(V, c, α)

in: sequence of n values $V = \langle V_0, V_1, \ldots, V_{n-1} \rangle$ $(2 \leq n)$; change c; momentum coefficient α $(0 \leq \alpha \leq 1)$

out: sequence of $n + 1$ values W where the first n values are identical to V and the last value is $W_n = W_{n-1} + c + \alpha(W_{n-1} - W_{n-2})$

1: $W \leftarrow$ **copy** V ▷ Make a local copy from V.
2: $W \leftarrow W \parallel \langle W_{n-1} + c + \alpha(W_{n-1} - W_{n-2}) \rangle$ ▷ Append a new value.
3: **return** W ▷ Publish W as immutable.

Table A.1 Reserved words for algorithms.

all	**div**	**error**	**not**	**repeat**	**while**
and	**do**	**for**	**of**	**return**	**xor**
case	**else**	**if**	**or**	**then**	
copy	**end**	**mod**	**others**	**until**	

Let us take a closer look at the pseudo-code notation. As in any other formal programming language, we can combine primitive constants and operators to build up expressions, control the execution flow with statements, and define a module as a routine. To do this, the pseudo-code notation uses the reserved words listed in Table A.1.

Table A.2 lists the notational conventions used in the algorithm descriptions. The constants FALSE and TRUE denote the truth values, and value NIL is a placeholder for an entity that is not yet known. The assignment operator ← defines a statement that updates the structure on the left side by a value evaluated from the right side. Equality can be compared using the operator =. To protect an object from side effects, it can be copied (or cloned) by the prefix operator **copy**. In a formal sense, the trinity of assignment, equality, and copy can be applied to the identity, shallow structure, or deep structure of an object. However, a mixture of these structure levels is possible. Because the algorithms presented in this book

Table A.2 Algorithmic conventions.

Notation	Meaning
FALSE, TRUE	Boolean constants
NIL	Unique reference to non-existent objects
$x \leftarrow y$	Assignment
$x = y$	Comparison of equality
$x \leftarrow$ **copy** y	Copying of objects
▷ Read me.	Comment
primitive(x)	Primitive routine for object x
HELLO-WORLD(x)	Algorithmic function call with parameter x
mathematical(x)	Mathematical function with parameter x

Table A.3 Mathematical functions.

Notation	Meaning
$\lfloor x \rfloor$	The largest integer n so that $n \leq x$
$\lceil x \rceil$	The smallest integer n so that $x \leq n$
$\log_b x$	Logarithm in base b
$\ln x$	Natural logarithm ($b = e \approx 2.71828$)
$\lg x$	Binary logarithm ($b = 2$)
$\max C$	Maximum of a collection; similarly for $\min C$
$\tan x$	Trigonometric tangent; similarly for $\sin x$ and $\cos x$
$\arctan \alpha$	Inverse of tangent; similarly for $\arcsin \alpha$ and $\arccos \alpha$

do not have relationships across their software interfaces (e.g. classes in object-oriented languages), we use these operations informally, and if there is a possibility of confusion, we elaborate on it in a comment.

At first sight, the difference between primitive routines and algorithmic functions can look happenstance, but a primitive routine can be likened to an attribute of an object or a trivial operation. For example, when operating with linearly orderable entities, we can define *predecessor*(e) and *successor*(e) for the predecessor and successor of e. The *successor*$(\bullet)$ – where $\bullet$ denotes a dummy variable – can be seen just as a function that extracts its result from the given argument. A primitive routine that indicates a status can also be seen as an attribute that changes – and can be changed – during the execution of an algorithm. For this reason, we can assign a value to a primitive routine. For example, to mark a town t as visited, we can define a primitive routine *visited*$(\bullet)$ to characterize this status, and then assign

```
1: visited(t) ← TRUE
```

If towns are modelled as software objects, the primitive routine *visited*$(\bullet)$ can be implemented as a member variable with appropriate get and set functions.

Sometimes, the algorithms include functions originating from elementary mathematics. For example, we denote the sign of x with $\operatorname{sgn}(x)$, which is defined in Equation (A.2).

$$\operatorname{sgn}(x) = \begin{cases} -1, & \text{if } x < 0; \\ 0, & \text{if } x = 0; \\ 1, & \text{if } 0 < x. \end{cases} \tag{A.2}$$

Table A.3 is a collection of the mathematical functions used throughout this book.

A.1 Changing the Flow of Control

The algorithms presented in this book run inside one control flow or thread. The complete control command of the pseudo-code is a *statement*, which is built from other simpler statements or sub-parts called *expressions*. When a statement is evaluated, it does not yield value but affects the current state of the system. In contrast, the evaluation of an expression produces a value but does not change the visible state of the system.

A.1.1 Expressions

Anything that yields a value after it is evaluated can be seen as an expression. The fundamental expressions are constants, variables, and primitive routines. An algorithm also represents an expression, because it returns a value.

To change, aggregate, and compare values, we need operators (see Table A.4) that can be used to build up more descriptive expressions. Although the pseudo-code operators originate mainly from mathematics, some of them are more related to computer calculations. For example, if we have two integers x and y, expression x **div** y equals the integer part of x/y so that the outcome is truncated towards $-\infty$. Operator **mod** produces the remainder of this division, which means that the Boolean expression $x = (x \textbf{ div } y) \cdot y + (x \textbf{ mod } y)$ is always true. It should be noted that some mathematical conventions are context sensitive. For example, for a value x, the operator $|x|$ denotes its absolute value, but for a set S, the operator $|S|$ means its cardinality (i.e. the number of its members). If the meaning of a notation is ambiguous, we clarify it with a comment.

The value of an arithmetic expression is stored to a variable or compared to another value as a Boolean expression. To construct expressions from truth values, we resort to mathematical logic. We use the logical operators listed in Table A.5 in the main text, and their algorithmic counterparts listed in Table A.6 in pseudo-codes. The conditional logical operators **and then** and **or else** differ in that that their evaluation, proceeding from left to right, is terminated immediately, when the result can be inferred. There are no reserved words for logical implication or equivalence, but, if necessary, they can be formed as $x \Rightarrow y \equiv \neg x \vee y$ and $x \Leftrightarrow y \equiv \neg(x \oplus y)$.

Table A.4 Arithmetic operators.

Notation	Meaning
$x + y$	Addition
$x - y$	Subtraction
x/y	Division ($y \neq 0$)
$x \cdot y$	Multiplication, also denoted as xy
n **div** m	Integer division
n **mod** m	Integer modulo

Table A.5 Logical operators in text.

Notation	Meaning
$\neg x$	Logical negation
$x \wedge y$	Logical and
$x \vee y$	Logical or
$x \oplus y$	Logical exclusive-or
$x \Rightarrow y$	Logical implication
$x \Leftrightarrow y$	Logical equivalence

Table A.6 Logical operators in algorithms.

Notation	Meaning
not x	Logical negation
x **and** y	Logical and
x **or** y	Logical or
x **xor** y	Logical exclusive-or
x **and then** y	Conditional logical and
x **or else** y	Conditional logical or

A.1.2 Control structures

Pseudo-code notations follow the widely accepted idea of structured programming, where the control flow is described using sequence, selection, and repetition structures (Dahl *et al*. 1972; Dijkstra 1968). However, we permit that this rule of 'single entry and single exit points of control' can be broken with an explicit return statement.

Sequence

A sequence of statements is indicated by writing the statements one after the other. If there is more than one statement in the line, the statements are separated by a semicolon (;). For example, swapping the values of variables x and y using a temporary variable t can be written as

1: $t \leftarrow x$; $x \leftarrow y$; $y \leftarrow t$

Line numbers are used only for reference purposes, and they do not imply any algorithmic structure.

Many programming languages include a compound structure that combines multiple statements into one. Because we do not scope the variables (e.g. define their lifetimes), this construct is expressed only implicitly: any statement can be replaced with a sequence of statements without stating it explicitly.

Selection

To describe a branch in the control flow, we have two selection structures. The first one, a **if–then–else** structure, proceeds according to the value of a Boolean expression: If the value is TRUE, the control flows to the **then** statements; otherwise, the control flows to the **else** statements. The **else** branch is optional.

1: **if** *Boolean expression* **then**
2: $\quad$ *statement*$_0$ ▷ Executed only for true case.
3: **else**
4: $\quad$ *statement*$_1$ ▷ Executed only for false case.
5: **end if**

The **case–of** construct defines a multi-selection that is based on the value of an arithmetic expression. To clearly indicate which control branch is executed, we require that

the branches are labelled with disjoint, constant-like values. Unlike in some programming languages, the control does not flow from one branch to another. A label **others** can be used to indicate the branch 'any other value not mentioned'. If the selection expression returns a truth value, we prefer the **if–then–else** structure.

```
1: case expression of
2:   constant_0: statement_0          ▷ Control branch for value constant_0.
3:   constant_1: statement_1
4:   ⋮
5:   others: default statement
6: end case
```

If none of the branching labels match with the expression, the control moves directly to the next statement following the **case–of** structure.

Repetition

To iterate statements, we introduce one definite loop structure and two indefinite loop structures. The definite loop is called **for–do** structure and it is used when the number of iteration cycles can be calculated before entering the loop body.

```
1: for iteration statement do
2:   statement
3: end for
```

The iteration statement has two variants. First, it can represent an enumeration by introducing a loop variable v that gets values sequentially from a given range $[f, t]$: $v \leftarrow f \ldots t$ (i.e. the initial value of v is f and the final value is t). Second, the iteration statement can represent a sequential member selection over a collection C: **all** $v \in C$. This loop variant bounds v once to each member of C in an unspecified order. To preserve clarity, C cannot be changed until the loop is finished.

As an example of the difference between these two **for** loops, let us find the maximum value from a sequence S of n values. We denote the ith member of S with S_i for $i \in [0, n-1]$. The most concrete algorithm for the problem is to define the order in which the sequence S is traversed:

```
1: c ← S_0
2: for i ← 1 ... (n − 1) do
3:   if c < S_i then c ← S_i end if
4: end for
5: ▷ Value in c is the maximum of S.
```

If there is no need to restrict the way the algorithm can traverse S, the iteration statement of the loop can be formed as a member selection:

```
1: c ← some member in S
2: S′ ← S \ {c}
3: for all m ∈ S′ do
4:   if c < m then c ← m end if
5: end for
6: ▷ Value in c is the maximum of S.
```

Of course, finding a maximum from a linear structure is so trivial that we can express it using the mathematical convention $c \leftarrow \max S$.

To find out the position of a maximum value, we can use the primitive function *indices*(S) that returns the set $\{0, 1, \ldots, |S| - 1\}$ of valid indices in S. The index set can be used for an iteration coordination:

```
1: I ← indices(S)
2: c ← some member in I
3: I′ ← I \ {c}
4: for all i ∈ I′ do
5:   if S_c < S_i then c ← i end if
6: end for
7: ▷ Value S_c is some maximum of S.
```

Using a mathematical notation, we express the same with $c \leftarrow \arg\max S$.

If we cannot determine a closed form equation for the number of loop cycles, it is preferable to use an indefinite loop structure instead. If it is possible that the loop body is not visited at all, we use the **while–do** structure. The loop exits when the control flow evaluates the Boolean expression to FALSE.

```
1: while Boolean expression do
2:   statement
3: end while
```

If the loop body is executed at least once, we use the **repeat–until** structure. The loop exits when the control flow evaluates the Boolean expression to TRUE.

```
1: repeat
2:   statement
3: until Boolean expression
```

Control shortcuts

As a general rule, the control structures with single entry and single exit points are easier to maintain than structures that use control shortcuts. For this reason, we use only two statement level mechanisms for breaking the control flow in the middle of an algorithm. Normally, an algorithm ends with a **return** statement that forwards the control back to the invoker of the algorithm possibly including a return value:

```
1: return expression
```

We permit multiple **return** statements to be placed at any pseudo-code line. When the control flow reaches a **return** statement, it exits the algorithm and forwards the evaluated value of the given expression immediately.

Another way to exit the algorithm is when an error has occurred and control flow cannot proceed normally:

```
1: error description
```

Because the algorithm cannot fulfil its operative contract with the invoker, the situation resembles exception handling, as is the way with many programming languages. The invoker can catch errors using a **case–of** structure:

```
1: v ← AVERAGE(S)
2: case v of
3:     error empty: v ← UNDEFINED        ▷ Unexpected situation: |S| = 0.
4: end case
```

A.2 Data Structures

Generality of the description of an algorithm follows from proper abstractions, which is why we have abstracted data structures to fundamental data collections such as sets, mappings, and graphs. For accessing data from these data collection, we use primitive routines and indexing abstractions.

A.2.1 Values and entities

The simplest datum is a value. Apart from the constants FALSE, TRUE, and NIL, we can define other literals for special purposes. A value is a result of an expression and can be stored to a variable. The values in the pseudo-code notation do not imply any particular implementation. For example, NIL can be realized using a null pointer, the integer value -1 or a sentinel object.

Values can be aggregated so that they form the attributes of an entity. These attributes can be accessed through primitive routines. For example, to define an entity e with physical attributes, we can attach primitive routines $location(e)$, $size(e)$ and $weight(e)$ to it. Because an attribute concerns only the entity given as an argument, the attribute can also be assigned. For example, to make e weightless, we can assign $weight(e) \leftarrow 0$. If an entity is implemented as a software record or an object, the attributes are natural candidates for member variables and the respective get and set member functions.

A.2.2 Data collections

A collection imposes relationships between its entities. Instead of listing all the commonly used data structures, we take a minimalist approach and use only a few general collections. A collection has characteristic attributes and it provides query operations. Moreover, it can be modified if it is a local structure in an algorithm. The elements of a data structure must be initialized, and an element that has not been given a value cannot be evaluated.

Sets

The simplest collection of entities (or values) is a set. The members of a set are unique (i.e. they have different values) and they are not ordered in any way. Table A.7 lists the usual set operations.

The set of natural numbers is $\mathbb{N} = \{0, 1, 2, \ldots\}$, the set of integer numbers is $\mathbb{Z} = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$, and the set of real numbers is $\mathbb{R}$. In a similar fashion, we can define the set $\mathbb{B} = \{0, 1\}$ for referring to binary numbers. We can now express, for example, a 32-bit word by denoting $w \in \mathbb{B}^{32}$, and refer to its ith bit as w_i.

We can also define a set by using an interval notation. For example, if it is clear from the context that a set contains integers, the interval $[0, 9]$ means the set $\{0, 1, \ldots, 9\}$. To

Table A.7 Notations for a set that are used in text and in pseudo-code.

Notation	Meaning
$e \in S$	Boolean query: is e a member of S
$\|S\|$	Cardinality (i.e. the number of elements)
$\emptyset$	Empty set
$\{x\}$	Singleton set
$R \cup S$	Union set
$R \cap S$	Intersection set
$R \setminus S$	Difference set
$R \subset S$	Boolean query: is R a proper subset of S
$R \times S$	Cartesian product
S^d	Set $S \times S \times \cdots \times S$ of d-tuples
$\wp(S)$	Power set of S

indicate that an interval notation refers to real numbers, we can denote $[0, 9] \subset \mathbb{R}$. The ends of the interval can be closed, marked with a bracket [or], or open, marked with a parenthesis (or).

The cardinality of a set is its attribute. If the final size of a (locally defined) set is known beforehand, we can emphasize it by stating

```
1: |S| ← n          ▷ Reserve space for n values.
```

This idiom does not have any effect in the algorithm; it is merely a hint for implementation.

Sequences

To impose a linear ordering to a collection of n elements we define a sequence as $S = \langle e_0, e_1, \ldots, e_{n-1} \rangle$. Unlike a set, a sequence differentiates its elements with an index, and, thus, it can contain multiple identical elements. We refer to the elements with subscripts. For example, the ith element of S is denoted S_i. The indexing begins from zero – and not from one – and the last valid index is $|S| - 1$. The cardinality of a sequence equals to its length (i.e. the number of elements in it). In addition to the notations presented in Table A.8, we have a primitive routine *enumeration*(C), which gives to its argument collection C some order in a form of a sequence. In other words, *enumeration*(C) returns a sequence S that is initialized by the following pseudo-code:

```
1: |S| ← |C|          ▷ Reserve space for |C| elements.
2: i ← 0
3: for all e ∈ C do
4:    S_i ← e
5:    i ← i + 1
6: end for
7: ▷ Sequence S is initialized.
```

We can declare the length of a sequence S before it is initialized using a pseudo-code idiom but – unlike with sets – the assignment affects the algorithm by defining a valid index range for S.

Table A.8 Notations for a sequence that are used in text and in pseudo-code.

Notation	Meaning
$e \in S$	Boolean query: is e a member of S
$\|S\|$	Length
$indices(S)$	Set $\{0, 1, \ldots, \|S\| - 1\}$ of valid indices
S_i	The ith element; $i \in indices(S)$
$\langle\ \rangle$	Empty sequence
$R \parallel S$	Catenation sequence
$sub(S, i, n)$	Subsequence $\langle S_i, S_{i+1}, \ldots, S_{i+n-1}\rangle$; $0 \leq n \leq \|S\| - i$

```
1: |S| ← n          ▷ Reserve space for n values.
```

The context of use can impose restrictions on the sequence structure. A sequence S of n elements can act in many roles:

- If S contains only unique elements, it can be seen as an *ordered set*.
- If the utilization of S does not depend on the element order, S can represent a *multiset* (or bag). A multiset consists possibly multiple identical elements and does not give any order to them.
- If the length of S is constant (e.g. it is not changed by a catenation), S stands for an *n-tuple*. This viewpoint is emphasized if the elements are of the same 'type', or the tuple is a part of a definition of some relation set.
- If S includes sequences, it defines a *hierarchy*. For example, a nesting of sequences $S = \langle a, \langle b, \langle c, \langle d, \langle\rangle\rangle\rangle\rangle\rangle$ defines a list structure as recursive pairs ⟨datum, sublist⟩. The element d can be accessed with the expression $(((S_1)_1)_1)_0$.
- If a sequence is not stored to a variable but we use it at the left side of the assignment operator, the sequence becomes a *nameless record*. This can be interpreted as a multi-assignment operator with pattern matching. For example, to swap two values in variables x and y, we can write

  ```
  1: ⟨x, y⟩ ← ⟨y, x⟩
  ```

  This *unification* mechanism originates from the declarative programming paradigm. However, this kind of use of sequences is discouraged, because it can lead to infinite structures and illegible algorithms. Perhaps, the only viable use for this kind of interpretation is receiving multiple values from a function:

  ```
  1: ⟨r, α⟩ ← AS-POLAR(x, y)
  2:     ▷ Variables r and α are assigned and can be used separately.
  ```

  This does away with the need for introducing an extra receiver variable and referring to its elements.

Although a sequence is a one-dimensional structure, it can be extended to implement tables or multidimensional arrays. For example, a hierarchical sequence $T = \langle\langle a, 0\rangle, \langle b, 1\rangle, \langle c, 2\rangle\rangle$ represents a table of three rows and two columns. An element can be accessed through a proper selection of subsequences (e.g. the element c is at $(T_2)_0$). However, this row-major notation is tedious and it is cumbersome to refer to a whole column. Instead of raising one dimension over another, we can make them equally important by generalizing the one-dimensional indexing mechanism of the ordinary sequences.

Arrays

An array allows to index an element in two or more dimensions. An element in a two-dimensional array A is referred as $A_{i,j}$, where $i \in \{0, \ldots, \mathit{rows}(A) - 1\}$ and $j \in \{0, \ldots, \mathit{columns}(A) - 1\}$. A single row can be obtained with $\mathit{row}(A, i)$ and a column with $\mathit{column}(A, j)$. These row and column projections are ordinary sequences. For the sake of convenience, we let $A_{\langle i,j\rangle} = A_{i,j}$, which allows us to refer to an element using a sequence.

For a t-dimensional array $A_{i_0,i_1,\ldots i_{t-1}}$, the size of the array in the dimension d ($0 \leq d \leq t - 1$) is defined as $\mathit{domain}(A, d)$. Hence, for a two-dimensional array A, we have $\mathit{rows}(A) = \mathit{domain}(A, 0)$ and $\mathit{columns}(A) = \mathit{domain}(A, 1)$. An array $A_{i_0,i_1,\ldots i_{t-1}}$ is always rectangular: if we take any dimension d of A, value $\mathit{domain}(A, d)$ does not change for any valid indices $i_0, i_1, \ldots, i_{d-1}, i_{d+1}, \ldots, i_{t-2}, i_{t-1}$.

Mappings

A mapping is a data structure that behaves like a function (i.e. it associates a single result entity to a given argument entity). To distinguish mappings from primitive functions, algorithms, and mathematical functions, they are named with Greek letters. The definition also includes the domain and codomain of the mapping. For example, $\tau : [0, 7] \times [0, 3] \rightarrow \mathbb{B} \cup \{ \text{FALSE}, \text{TRUE} \}$ defines a two-dimensional function that can contain a mix of bits and truth values (e.g. $\tau(6, 0) = 1$ and $\tau(4, 2) = \text{FALSE}$). It is worth noting that a sequence S that has elements from the set R can be seen as a mapping $S : [0, |S| - 1] \rightarrow R$. In other words, we denote $S : i \mapsto r$ simply with an access notation $S_i = r$. Similarly, arrays can be seen as multi-argument functions. However, the difference between $\tau(\bullet, \bullet)$ and an array with eight rows and four columns is that the function does not have to be rectangular.

Because a mapping is a data structure, it can be accessed and modified. A mapping $\mu(k) = v$ can be seen as an associative memory, where μ binds a search key k to the resulting value v. This association can be changed by assigning a new value to the key. This leads us to define the following three categories of functions: A function $\mu : K \rightarrow V$ is *undefined* if it does not has any associations, which means that it cannot be used. When μ is a local structure of an algorithm and its associations are under change, μ is *incomplete*. A function is *complete* after it is returned from an algorithm in which it was incomplete.

To define *partial functions*, we assume that NIL can act as a placeholder for any entity but cannot be declared into the codomain set explicitly. If mapping $\mu : K \rightarrow V$ is undefined, it can be made 'algorithmically' partial:

```
1: for all k ∈ K do
2:    μ(k) ← NIL
3: end for
```

Now, each search key is bound to NIL but not to any entity in the codomain V. The separation of undefined and partial functions allows us to have explicit control over incomplete functions: accessing an unbound search key implies fault in the algorithm, but we can refer to the members of the set $\{ k \mid k \in K \wedge \mu(k) = \text{NIL} \}$.

Mappings are useful when describing self-recursive structures. For example, if we have $V = \{a, b, c, d\}$, a cycle can be defined with a successor mapping $\sigma : V \to V$ so that $\sigma(a) = b$, $\sigma(b) = c$, $\sigma(c) = d$, and $\sigma(d) = a$.

Graphs

To describe discrete elements and their relationships, we use graphs. Graphs provide us with a rich terminology that can be used to clarify a vocabulary for problem and solution descriptions. Informally put, an *undirected graph* $G = (V, E)$ (or a graph for short) comprises a finite set of vertices V and a set of edges $E \subseteq V \times V$. A vertex is illustrated with a circle and an edge with a line segment. An edge $e = (u, v) \in E$ is undirected and it is considered identical to (v, u). An edge (v, v) is called a *loop*. The ends of an edge $e = (u, v) \in E$ are returned by the primitive routine $ends(e) = \{u, v\}$. If a vertex u is connected to another vertex v ($u \neq v$) by an edge, u is said to be *adjacent* to v. The set of adjacent vertices of a vertex v is called a *neighbourhood*, and it is returned by the routine $neighbourhood(v)$. A sequence $W = \langle e_0, e_1, \ldots, e_{n-1} \rangle$ is called a *walk* of length n if $e_i = (v_i, v_{i+1}) \in E$ for $i \in [0, n-1]$. If we are not interested in the intermediate edges of a walk but only in its starting vertex and ending vertex, we denote $v_0 \rightsquigarrow v_n$. If $v_0 = v_n$, the walk W is *closed*. The walk W is called a *path* if all of its vertices differ (i.e. $v_i \neq v_j$ when $i \neq j$) and it does not contain loops. A closed walk that is a path, except for $v_0 = v_n$, is a *cycle*. A graph without cycles is *acyclic*.

A *directed graph* (or digraph) changes the definition of the edge: An edge has a direction, which means that $(u, v) \neq (v, u)$ when $u \neq v$. In this case, an edge $e = (u, v)$ is illustrated with an arrow from u to v. Vertex v is the *head* and u is the *tail* of the edge, and we have routines $head(e)$ and $tail(e)$ to retrieve them. Naturally, $ends(e) = \{head(e)\} \cup \{tail(e)\}$. In a directed graph, the successors of vertex v are in a set returned by routine $successors(v)$, and if v has no successors, then $successors(v) = \emptyset$. Similarly, the predecessors of vertex v are given by $predecessors(v)$. The neighbourhood is the union of adjacent vertices: $neighbourhood(v) = successors(v) \cup predecessor(v)$. Because we allow loops, a vertex can be its own neighbour. The definition of the concepts *directed walk*, *directed path*, and *directed cycle* are similar to their respective definitions in the undirected graphs.

In a *weighted graph*, derived from an undirected or a directed graph, each edge has an associated weight given by a weight function $weight : E \to \mathbb{R}_+$. We let $weight(e)$ and $weight(u, v)$ to denote the weight of the edge $e = (u, v) \in E$.

A *tree* is an undirected graph in which each possible vertex pair u and v is connected with a unique path. In other words, the tree is acyclic and $|E| = |V| - 1$. A *forest* is a disjoint collection of trees. We are often interested in a *rooted tree*, where one vertex is called a *root*. We can call a vertex of a rooted tree as a *node*. The root can be used as a base for traversing the other nodes and the furthermost nodes from the root are *leaves*. The non-leaf nodes, the root included, are called *internal nodes*. The adjacent nodes of node n away from the root are called the *children* of node n, denoted by $children(n)$. The unique node in $neighbourhood(n) \setminus children(n)$ is called the *parent* of node n. If $parent(n) = \emptyset$, n is the root node.

A.3 Format of Algorithms

Algorithm A.2 is an example of an algorithm written using pseudo-code. The algorithm iteratively solves the Towers of Hanoi, and the solution can be generated with the following procedure:

TOWERS-OF-HANOI(n)
in: number of discs n $(0 \leq n)$
out: sequence S of states from the initial state to final state

1: $S \leftarrow \langle$INITIAL-STATE(n)$\rangle$
2: **while** $turn(S) \neq 2^n - 1$ **do**
3: $S \leftarrow S \parallel \langle$NEXT-MOVE($S$)$\rangle$
4: **end while**
5: **return** S

The details of how this algorithm works are left as an exercise for the interested reader. However, we encourage the casual reader to study the used notations and identify the conventions described in this appendix.

The *signature* of an algorithm includes the name of the algorithm and the arguments passed to it. It is followed by a *preamble*, which may include the following descriptions:

in: This section describes the call-by-value arguments passed to the algorithm. The most important preconditions concerning an argument are given in parenthesis. Because an algorithm behaves as a function from the caller's perspective, there is no need for preconditions about the state of the system. If the algorithm has multiple arguments, their descriptions are separated by a semicolon.

out: This section outlines the result passed to the caller of the algorithm. In most cases, it is sufficient to give the post-condition in a natural language. Because the algorithms are functions, each algorithm must include a description about its return values.

constant: If an algorithm refers to constant values or structures through a symbolic name, they are described in this section. The constraints are given in parentheses, and multiple constants are separated by a semicolon. The difference between an argument and a constant of an algorithm depends on the point of view, and the constants do not necessarily have to be implemented using programming language constants.

local: Changes are allowed only to the entities created inside the local scope of the algorithm. This section describes the most important local variables and structures.

The preamble of an algorithm is followed by enumerated lines of pseudo-code. The line numbering serves only for reference purposes and it does not impose any structure on the pseudo-code. For example, we can elaborate that line 3 of NEXT-MOVE in Algorithm A.2 can be implemented in $O(1)$ time by introducing an extra variable.

Algorithm A.2 An iterative solution to Towers of Hanoi.

INITIAL-STATE(n)
in: number of discs n ($0 \leq n$)
out: triplet $S = \langle s_0, s_1, s_2 \rangle$ representing the initial state
1: $s_0 \leftarrow \langle n, n-1, \ldots, 1 \rangle$; $s_1 \leftarrow s_2 \leftarrow \langle\,\rangle$
2: $S \leftarrow \langle s_0, s_1, s_2 \rangle$ ▷ Start s_0, goal s_1, aid s_2.
3: $turn(S) \leftarrow 0$
4: $direction(S) \leftarrow 1$ ▷ Clockwise rotation.
5: **if** n is even **then** ▷ Counter-clockwise rotation.
6: $direction(S) \leftarrow -1$
7: **end if**
8: **return** S

NEXT-MOVE(S)
in: triplet $S = \langle s_0, s_1, s_2 \rangle$ representing the current game state
out: triplet $R = \langle r_0, r_1, r_2 \rangle$ representing the new game state
local: pole indices $a, b, z \in \{0, 1, 2\}$; disc numbers $g, h \in [2, n]$; $last(Q) = Q_{|Q|-1}$, if $1 \leq |Q|$, otherwise, $last(Q) = +\infty$
1: $R \leftarrow$ **copy** S ▷ Now $r_i = s_i$, $0 \leq i \leq 2$.
2: $direction(R) \leftarrow direction(S)$
3: $a \leftarrow$ the index of the pole where $1 \in r_a$
4: $b \leftarrow (3 + a + direction(R)) \bmod 3$
5: $z \leftarrow (3 + a - direction(R)) \bmod 3$
6: **if** $turn(R)$ is even **then** ▷ Move the smallest disc.
7: $r_b \leftarrow r_b \parallel \langle 1 \rangle$
8: $r_a \leftarrow sub(r_a, 0, |r_a| - 1)$
9: **else** ▷ Move the non-smallest disc.
10: $g \leftarrow last(r_b)$ ▷ $+\infty$, **if** $|r_b| = 0$.
11: $h \leftarrow last(r_z)$ ▷ $+\infty$, **if** $|r_z| = 0$.
12: **if** $g < h$ **then**
13: $r_z \leftarrow r_z \parallel \langle g \rangle$
14: $r_b \leftarrow sub(r_b, 0, |r_b| - 1)$
15: **else if** $h < g$ **then**
16: $r_b \leftarrow r_b \parallel \langle h \rangle$
17: $r_z \leftarrow sub(r_z, 0, |r_z| - 1)$
18: **else**
19: **error** already in the final state
20: **end if**
21: **end if**
22: $turn(R) \leftarrow turn(S) + 1$
23: **return** R

A.4 Conversion to Existing Programming Languages

To concretize how an algorithm written in pseudo-code can be implemented with an existing programming language, let us consider the problem of converting a given Arabic number to the equivalent modern Roman number. Modern Roman numerals are the letters M (for the value 1000), D (500), C (100), L (50), X (10), V (5), and I (1). For example, $1989 = 1000 + (1000 - 100) + 50 + 3 \cdot 10 + (10 - 1)$ is written as MCMLXXXIX. Algorithm A.3 solves the conversion problem by returning a sequence R of multipliers of 'primitive' numbers in $P = \langle 1000, 900, 500, 400, 100, 90, 50, 40, 10, 9, 5, 4, 1 \rangle$. In our example, 1989 becomes $R = \langle 1, 1, 0, 0, 0, 0, 1, 0, 3, 1, 0, 0, 0 \rangle$.

Algorithm A.3 Conversion from an Arabic number to a modern Roman number.

ARABIC-TO-ROMAN(n)
in: decimal number n ($0 \leq n$)
out: sequence $R = \langle s_0, s_1, \ldots, s_{12} \rangle$ representing the structure of the Roman number ($R_i =$ number of primitives V_i in n for $i \in [0, 12]$)
constant: sequence $P = \langle 1000, 900, 500, 400, 100, 90, 50, 40, 10, 9, 5, 4, 1 \rangle$ of primitive Roman numbers
local: remainder x to be converted ($0 \leq x \leq n$); coefficient c for a primitive Roman numbers (for other than P_0, $0 \leq c \leq 3$)

1: $|R| \leftarrow |P|$ ▷ Reserve space for $|P| = 13$ values.
2: $x \leftarrow n$
3: **for** $i \leftarrow 0 \ldots (|P| - 1)$ **do**
4: $c \leftarrow x$ **div** P_i ▷ Number of multiplicands P_i in x.
5: $R_i \leftarrow c$
6: $x \leftarrow x - c \cdot P_i$
7: **end for**
8: **return** R

A Java programmer could implement Algorithm A.3 by first modelling the primitive numbers with the enumeration type `RomanNumeral`. Each **`enum`** constant (`I`, `IV`, ..., `M`) is declared with its decimal value, which can be accessed with the function `getValue()`.

```
public enum RomanNumeral {
   I(   1),
   IV(  4), V(   5), IX(  9), X(  10),
   XL( 40), L(  50), XC( 90), C( 100),
   CD(400), D( 500), CM(900), M(1000);

   private int value;
   private RomanNumber(int v) { value = v; }

   public int getValue() { return value; }
   }
```

The actual conversion is implemented as a static function `toRoman(int)` in the class `ArabicToRomanNumber`. Note that the original algorithm has been modified as follows:

- The conversion returns a string instead of a sequence of integers. Because a Roman number does not include zeros, the for-loop at lines 3–7 is replaced by two nested while-loops. The inner loop takes care of possible repetitions of the same primitive number.
- The precondition is strengthened to $1 \leq n$.
- To emphasize that the values $4000 \leq n$ are cumbersome to express in Roman numerals, the post-condition gives an estimate of how long the result string will be.

The actual Java code looks like this:

```
public class ArabicToRomanNumber {
   /** Convert an Arabic number to a modern Roman number.
    *  @.pre  1 <= n
    *  @.post result.length() <= (n div 1000) + (3 * 4)
    */
   public static String toRoman(int n) {
      RomanNumeral[] primitives = {
         RomanNumeral.M,  RomanNumeral.CM, RomanNumeral.D,
         RomanNumeral.CD, RomanNumeral.C,  RomanNumeral.XC,
         RomanNumeral.L,  RomanNumeral.XL, RomanNumeral.X,
         RomanNumeral.IX, RomanNumeral.V,  RomanNumeral.IV,
         RomanNumeral.I
         };
      int remainder = n;
      StringBuffer result = new StringBuffer();
      int i = 0;
      while ( remainder != 0 ) {
         while ( primitives[i].getValue() <= remainder ) {
            result.append(primitives[i]);
            remainder -= primitives[i].getValue();
            }
         ++i;
         }
      String res = result.toString();
      return res;
      }
   }
```

A programmer more accustomed to the quirks of C programming language could implement Algorithm A.3 following the original form more closely. However, the primitive sequence P has a regular structure and it can be compressed to four values by introducing a scaling variable. To include a possibility for memory allocation optimizations, the caller must provide the storage buffer for the Roman number.

```
#include <string.h>

/* Convert an Arabic number to a modern Roman number.
 * Pre:  (the length of buffer is at least 13) and (0 <= n).
 * Post: (result == buffer) and (result[0..12] represents
 *       Roman number).
 */
int* arabicToRoman(int* buffer, int n) {
   memset(buffer, 0, 13 * sizeof(int));
   /* Here: For all i: buffer[i] == 0. */
   int conversions[] = { 1000, 900, 500, 400 };
   int divider = 1;
   int i = 0;
   int value;
   while ( n != 0 ) {
      value = conversions[i % 4] / divider;
      buffer[i] = n / value;
      n -= buffer[i] * value;
      ++i;
      if ( i % 4 == 0 ) divider *= 10;
      }
   return buffer;
   }
```

As we can see, the Java and C implementations include numerous language-specific details that can be omitted from the pseudo-code representation. When the syntax and semantics of C, C++, and Java seem as peculiar as Algol68, Cobol, and Fortran do today, descriptions resembling Algorithm A.3 are likely to remain understandable in the future too and can be re-implemented using the favourite programming language of that time.