Department of Computer Science
Aarhus University

Algorithms and Data Structures

Transition systems

Mikkel Nygaard
Erik Meineche Schmidt
February 2014

Contents

1 Transition systems **1**
- 1.1 Introduction . . . 1
- 1.2 A football match . . . 2
- 1.3 Definitions . . . 4
- 1.4 Proof principles . . . 9
  - 1.4.1 Invariance . . . 10
  - 1.4.2 Termination . . . 12
- 1.5 Examples . . . 14
  - 1.5.1 Nim . . . 14
  - 1.5.2 Hanoi . . . 16
  - 1.5.3 Euclid . . . 18
  - 1.5.4 Expressions . . . 20
  - 1.5.5 Graph traversal . . . 22
  - 1.5.6 Red-black search trees . . . 23

2 Algorithm theory **31**
- 2.1 Commands . . . 31
- 2.2 Algorithms . . . 33
- 2.3 Correctness . . . 34
  - 2.3.1 Validity . . . 35
  - 2.3.2 Termination . . . 39
- 2.4 Examples . . . 40
  - 2.4.1 Extended Euclid . . . 41
  - 2.4.2 Factorial . . . 43
  - 2.4.3 Power sum . . . 45
- 2.5 Arrays . . . 47
- 2.6 Complexity . . . 49

3 Fundamental algorithms **51**
- 3.1 Exponentiation . . . 51
- 3.2 Scanning . . . 53

3.3 Searching 54
3.3.1 Linear search 54
3.3.2 Binary search 54
3.4 Sorting 56
3.4.1 Insertion sort 57
3.4.2 Merge sort 58
3.4.3 Quick sort 62
3.5 Selection 67
3.5.1 Quick select 67
3.5.2 Deterministic selection 69
3.6 Examples 70
3.6.1 Maximal subsum 71
3.6.2 Longest monotone sequence 73

Chapter 1

Transition systems

This chapter gives an introduction to transition systems, one of the simplest and most useful formalisms in computer science. The theory of transition systems presented here provides the techniques we'll need in the rest of these notes for reasoning about algorithms.

1.1 Introduction

A transition system consists of a set of ***configurations*** and a collection of ***transitions***. Transition systems are used to describe ***dynamic processes*** with configurations representing states and transitions saying how to go from state to state.

As a concrete example of these somewhat abstract notions, consider a game like chess. Here the configurations are the positions in the game (that is, the placements of pieces on the board) while the transitions describe the legal moves according to the rules of chess. A game is viewed as the dynamic process with the players (starting with white's first move) taking turns in moving and thereby transforming one configuration into another until a winner is found or the game has ended as a draw.

In computer science there are many situations where a process can be seen as a sequence of configurations, one following another in a systematic way as with chess. An important example is the computer itself which at a sufficiently high level of abstraction can be seen as a mechanism capable of transforming the contents of its registers and stores (the configurations) according to the rules built into the instruction set of the machine. Other examples are networks, communication systems, algorithms etc. Using suitable abstractions they may all be described in terms of transition systems.

The main purpose of describing processes formally is that this allows us to subject the processes to formal analysis, ie. it allows us to talk about their properties in a precise way. Traditionally, two kinds of properties are of interest, ***safety properties*** and ***liveness properties***. A safety property is used to express that nothing bad can happen in a process, ie. that it is impossible to enter an unwanted configuration. In contrast, liveness properties are used to express that something happens at all in a process—the safest way of avoiding errors is to do nothing, but systems like that tend to be rather boring.

The next section provides further intuition for the notions of configuration and process by means of a well-known example. Following this, we present techniques for expressing and reasoning about both safety properties and liveness properties, and we'll put the techniques to use in a range of different situations.

1.2 A football match

Consider as a simple example of a process a description of a football match between two teams which we'll call A and B. We may view the match as a sequence of situations where the teams take turns in having the ball.

If we use two configurations A and B to name the situation where team A, respectively team B, possesses the ball, and if we furthermore assume that team A kicks off, we may describe the match as the following process:

$$A, B, A, B, \dots, A, B, \dots$$

Surely, this description is not very interesting, mainly because the purpose of playing football (besides having fun) is to score goals, and the more the better. We can bring the goals into the description by extending the configurations, ie. the descriptions of the situations in the match, so that they contain the current score. A situation is now described by a configuration of the form $[A, 3, 2]$ or $[B, 1, 0]$, saying that team A has the ball and is leading by three goals to two, or that team B has the ball and is behind by one goal to nil. The description of the match is now a sequence starting with the configuration $[A, 0, 0]$, that is, a sequence of the form

$$[A, 0, 0], \dots$$

But now it is no longer clear how this sequence proceeds, ie. what processes are correct descriptions of the match. One could attempt to explicitly write down all possible processes according to the rules of football (eg.

no team may score twice against the opponent without handing over the ball in between). However, one would quickly realize that this is an impractical solution. Fortunately, it is often possible to describe processes *indirectly* (and much more compactly) by using *rules* for transitions between configurations. This is the technique used with transition systems. If we describe a football match by the rule

F_2: a situation where team A (team B) has the ball is followed by a situation where team B (team A) has the ball and where the score is either unchanged or where team A (team B) has scored another goal,

then the start configuration $[A, 0, 0]$ may be followed by the configurations $[B, 0, 0]$ and $[B, 1, 0]$ which then may be followed by $[A, 0, 0]$ and $[A, 0, 1]$, respectively $[A, 1, 0]$ and $[A, 1, 1]$ etc. A legal process is now a sequence starting with $[A, 0, 0]$, and in which all configuration changes obey the rule above. So examples of legal processes are

$[A, 0, 0], [B, 0, 0], [A, 0, 1], [B, 0, 1], [A, 0, 2], \ldots$
$[A, 0, 0], [B, 0, 0], [A, 0, 0], [B, 0, 0], \ldots$
$[A, 0, 0], [B, 1, 0], [A, 1, 1], [B, 2, 1], [A, 2, 2], \ldots$

Although our description of the football match has improved, it still suffers from non-termination: the match never ends! This is due to rule F_2 demanding that a configuration always has a successor so that all processes are infinite. We could modify the rule as follows:

F_3: a situation where team A (team B) has the ball is followed either by a situation where the referee has the ball and where the score is unchanged or by a situation where... (as for F_2).

If we represent the situation where the referee has the ball and the score is, say, three to two by the configuration $[R, 3, 2]$, the legal processes now include

$[A, 0, 0], [B, 1, 0], [A, 1, 0], [R, 1, 0]$
$[A, 0, 0], [B, 0, 0], [A, 0, 0], \ldots, [B, 0, 0], [R, 0, 0]$ (boring match!)
$[A, 0, 0], [B, 1, 0], [A, 1, 0], [B, 2, 0], \ldots, [A, 6, 0], [R, 6, 0]$

However, the *possibility* that processes may be infinite still exists, so the problem from before is still there. We can solve it by adding a *clock* to the description so that a configuration now also says how much time has elapsed. The situation that B has the ball, the score is one to nil, and twenty minutes has passed is expressed by the configuration $[20, B, 1, 0]$. We could now say that a team always possesses the ball in precisely one minute, but to make the description a bit more flexible, we'll allow them to keep the ball for one or two minutes. The description then obeys the following rule:

F_4: (a) a situation where team A (team B) has the ball and at most 89 minutes have passed is followed by a situation where team B (team A) has the ball and where the score is either unchanged or team A (team B) has scored another goal, *and* where one or two more minutes (but never more than 90 minutes total) have passed.

(b) a situation where team A (team B) has the ball and 90 minutes have passed is followed by a situation where the referee has the ball and where the time and score are unchanged.

Here is a legal process according to F_4:

$$[0, A, 0, 0], [2, B, 1, 0], [3, A, 1, 0], \ldots, [45, B, 1, 3], \ldots$$
$$\ldots, [88, B, 6, 7], [90, A, 6, 8], [90, R, 6, 8]$$

The description has again improved, but still valid points may be raised. One is that a team cannot score an own goal; another that the match consists of only one half. We leave it to the reader to modify the rule F_4 to address these (and any other) shortcomings.

As may be seen from the rule F_4 above, prose descriptions of situation changes can easily become a bit heavy. In the following section, we'll introduce a mathematical formalism which in many cases is easier to work with.

1.3 Definitions

Transition systems constitute a formalized method for description of processes, defined by rules for change of configuration as in the preceding discussion of the football example. A transition system is a mathematical object consisting of two parts, a set of configurations and a ***binary relation*** on this set:

Definition 1.3.1 A ***transition system*** S is a pair of the form

$$S = (C, T)$$

where C is the set of ***configurations*** and $T \subseteq C \times C$ is a relation, the ***transition relation***. □

When using transition systems to describe processes as in the football example, the configurations will name the situations and the transition relation will specify the legal transitions between the situations.

Example 1.3.2 A transition system capturing the first description of a football match (where all we said was that the two teams had to take turns in possessing the ball) looks as follows:

$$S_1 = (C_1, T_1)$$

where

$$C_1 = \{A, B\} \qquad \text{and} \qquad T_1 = \{(A, B), (B, A)\}.$$

□

The purpose of using transition systems is to describe *processes* which we'll define as certain sequences of configurations:

Definition 1.3.3 Let $S = (C, T)$ be a transition system. S ***generates a set of sequences***, $\mathcal{S}(S)$, defined as follows:

1. the finite sequence $c_0, c_1, \ldots, c_n$ (for $n \geq 0$) belongs to $\mathcal{S}(S)$ if
   - (ι) $c_0 \in C$
   - ($\iota\iota$) for all i with $1 \leq i \leq n$: $(c_{i-1}, c_i) \in T$
2. the infinite sequence $c_0, c_1, \ldots, c_n, \ldots$ belongs to $\mathcal{S}(S)$ if
   - (ι) $c_0 \in C$
   - ($\iota\iota$) for all $i \geq 1$: $(c_{i-1}, c_i) \in T$

□

Example 1.3.4 For the transition system S_1 we have that $\mathcal{S}(S_1)$ contains all finite sequences starting with A or B and where A's follow B's and vice versa, together with the two infinite sequences

$$A, B, A, \ldots, A, B, \ldots$$
$$B, A, B, \ldots, B, A, \ldots$$

□

In most applications of transition systems we are only interested in some of the sequences generated by the system, more precisely the sequences that are *maximal* in the sense that they cannot be extended:

Definition 1.3.5 Let $S = (C, T)$ be a transition system. The set of ***processes generated by*** S, written $\mathcal{P}(S)$, is the subset of $\mathcal{S}(S)$ containing

1. all infinite sequences of $\mathcal{S}(S)$
2. all finite sequences $c_0, c_1, \ldots, c_n$ $(n \geq 0)$ of $\mathcal{S}(S)$ for which it holds that there is no $c \in C$ with $(c_n, c) \in T$.

The final configuration of a finite process is called a ***dead configuration***. □

Example 1.3.6 The processes generated by S_1 are precisely the two infinite sequences of Example 1.3.4. □

We have now introduced enough "machinery" to enable us to talk about processes generated by general (abstract) transition systems. When using the systems for something concrete, eg. the football examples, the need arises for suitable notation for specifying configurations and especially transition relations. Here we'll be very liberal and allow practically any kind of description precise enough to unambiguously specify both configurations and transitions. As for the latter we'll often employ so-called ***transition rules*** which look as follows:

$$\textit{left-hand side} \quad \rhd \quad \textit{right-hand side}$$

Here, both sides are configurations and the meaning of the rule is that any configuration matching the left-hand side may be followed by the corresponding configuration on the right-hand side.

Example 1.3.7 For the system S_1 the following two rules constitute an alternative way of giving the transition relation $T_1 = \{(A, B), (B, A)\}$:

$$\begin{array}{lcl} A & \rhd & B \\ B & \rhd & A \end{array}$$

□

Consider now the second version of the football match description where the score is taken into account. The processes in such matches are generated by a transition system $S_2 = (C_2, T_2)$ where the configurations are

$$\begin{array}{rcl} C_2 & = & \{A, B\} \times \mathbf{N} \times \mathbf{N} \\ & = & \{[X, a, b] \mid X \in \{A, B\},\ a, b \in \mathbf{N}\}. \end{array}$$

($\mathbf{N}$ is the set of natural numbers, $0, 1, \ldots$)—but how do we specify the transition relation?

Since the relation is infinite (ie. infinitely many pairs of configurations may follow each other) we cannot list all its elements. However, we can specify them using ***parametrized transition rules*** of the following form:

$$[A, a, b] \quad \triangleright \quad [B, a+1, b].$$

This rule says that for arbitrary natural numbers a and b we have that the configuration $[A, a, b]$ may be followed by the configuration $[B, a+1, b]$. So the rule is an alternative representation of the following part of the transition relation

$$\{(c_1, c_2) \in C_2 \times C_2 \mid \exists c_1 = [A, a, b] \text{ and } c_2 = [B, a+1, b]\}.$$

In this way we can represent the rule F_2 from Section 1.2 using the following four transition rules:

$$\begin{array}{lll} [A, a, b] & \triangleright & [B, a, b] \\ [A, a, b] & \triangleright & [B, a+1, b] \\ [B, a, b] & \triangleright & [A, a, b] \\ [B, a, b] & \triangleright & [A, a, b+1] \end{array}$$

—whose meaning should be clear.

Actually, it is necessary to extend the transition rules by adding the ***conditions*** under which they may be used. The need for this becomes apparent if we consider the last (clocked) version of the football example. Here, the configurations are given by the set

$$C_4 = \{[t, X, a, b] \mid 0 \leq t \leq 90, X \in \{A, B, R\}, a, b \in \mathbf{N}\},$$

and now the rules must express that they may only be used in such a way that the t-value doesn't exceed 90. This is done by adding conditions to the rules as follows:

$$[t, A, a, b] \quad \triangleright \quad [t+2, B, a+1, b] \quad \textbf{if } t \leq 88$$

—saying that this rule may only "be used upon" a configuration whose t-value is less than or equal to 88.

The following example shows the template for specification of transition systems that we'll use in the following:

Example 1.3.8 A transition system describing the final version of the football match:

Transition system Football
Configurations: $\{[t, X, a, b] \mid 0 \leq t \leq 90, X \in \{A, B, R\}, a, b \in \mathbf{N}\}$

$[t, A, a, b]$	$\triangleright$	$[t+2, B, a, b]$	**if** $t \leq 88$
$[t, A, a, b]$	$\triangleright$	$[t+2, B, a+1, b]$	**if** $t \leq 88$
$[t, A, a, b]$	$\triangleright$	$[t+1, B, a, b]$	**if** $t \leq 89$
$[t, A, a, b]$	$\triangleright$	$[t+1, B, a+1, b]$	**if** $t \leq 89$
$[90, A, a, b]$	$\triangleright$	$[90, R, a, b]$	
$[t, B, a, b]$	$\triangleright$	$[t+2, A, a, b]$	**if** $t \leq 88$
$[t, B, a, b]$	$\triangleright$	$[t+2, A, a, b+1]$	**if** $t \leq 88$
$[t, B, a, b]$	$\triangleright$	$[t+1, A, a, b]$	**if** $t \leq 89$
$[t, B, a, b]$	$\triangleright$	$[t+1, A, a, b+1]$	**if** $t \leq 89$
$[90, B, a, b]$	$\triangleright$	$[90, R, a, b]$	

Clearly, this system behaves correctly. □

We end this section with a more precise description of the format and meaning of the transition rules. A rule of the form

left-hand side $\triangleright$ *right-hand side* **if** *condition*

in which the parameters $x_1, \ldots, x_n$ occur, has the following meaning:

1. For all values of $x_1, \ldots, x_n$ for which *left-hand side* specifies a configuration and *condition* is true, *right-hand side* also specifies a configuration.

2. The rule defines the relation consisting of the set of pairs of configurations for which such values of $x_1, \ldots, x_n$ exist.

A collection of transition rules specifies the transition relation consisting of the union of the relations defined by the individual rules.

Abusing notation slightly, we'll also write $\triangleright$ for the transition relation itself. As $\triangleright$ is then a binary relation, we may talk about its ***reflexive*** and ***transitive closure***, standardly written $\triangleright^*$. Two configurations c, c' are related by $\triangleright^*$ if $c = c'$ or there exists a sequence generated by the transition system in which c comes before c'.

Example 1.3.9 For the transition system Football we have eg.

$$\begin{array}{lll} [45, B, 1, 1] & \triangleright^* & [47, A, 1, 2] \\ [0, A, 0, 0] & \triangleright^* & [90, R, 0, 0] \\ [39, A, 7, 6] & \triangleright^* & [39, A, 7, 6] \end{array}$$

□

1.4 Proof principles

One of the purposes of introducing a precise mathematical formalism for different notions is that we want to be able to reason precisely about the properties of those notions. When we in the following sections want to reason precisely about sequences and processes generated by transition systems, a certain kind of mathematical proof principle will be used over and over again, the so-called ***induction principle***. In its simplest version, the induction principle is used to show the correctness of a collection of mathematical statements enumerated by the natural numbers. Take as an example the (hopefully) well-known formula for the sum of the natural numbers up to n:

$$0+1+2+\cdots+n=\tfrac{1}{2}n(n+1).$$

We may consider this formula as a short-hand version of the following (infinitely many) statements $P(0), P(1), P(2), \ldots, P(n), \ldots$

$$\begin{array}{lrcl}
P(0) & 0 & = & \frac{1}{2}\cdot 0\cdot 1 \\
P(1) & 1 & = & \frac{1}{2}\cdot 1\cdot 2 \\
P(2) & 1+2 & = & \frac{1}{2}\cdot 2\cdot 3 \\
P(3) & 1+2+3 & = & \frac{1}{2}\cdot 3\cdot 4 \\
\vdots & & \vdots & \\
P(n) & 1+2+\cdots+n & = & \frac{1}{2}\cdot n\cdot (n+1) \\
\vdots & & \vdots &
\end{array}$$

The induction principle now says that we can prove all these statements by proving only the following two things:

a) $P(0)$ is true

b) for any of the statements $P(n)$ with $n \geq 0$ it holds that *if* $P(n)$ is true, *then* $P(n+1)$ is true.

Correctness of this induction principle is not hard to accept, for when $P(0)$ is true, it follows from b) that $P(1)$ is true, from which it follows using b) again that $P(2)$ is true, and so on.

Let us now prove the formula above, that is, let us prove that for all $n \geq 0$ it holds that $P(n)$ is true. When proving something by induction, we'll call the two proof-steps a) and b) above for the ***base case*** and the ***induction step***, respectively, as in the following proof:

Base case We need to show that $P(0)$ is true, which is obvious.

Induction step Assuming that $P(n)$ is true for some $n \geq 0$, we need to show that $P(n+1)$ is true. In other words, we need to prove that

$$0+1+2+\cdots+n+(n+1) = \tfrac{1}{2}(n+1)(n+2).$$

We argue as follows:

$$\begin{aligned}
&0+1+2+\cdots+n+(n+1) \\
&= \big(0+1+2+\cdots+n\big)+(n+1) \\
&= \big(\tfrac{1}{2}n(n+1)\big)+(n+1) && \text{using } P(n) \\
&= \tfrac{1}{2}\big(n(n+1)+2(n+1)\big) \\
&= \tfrac{1}{2}(n+1)(n+2)
\end{aligned}$$

—which proves $P(n+1)$ as wanted.

The assumption that $P(n)$ is true is often called the ***induction hypothesis***. We highlight the induction principle as follows:

Induction principle Let $P(0), P(1), \ldots, P(n), \ldots$ be statements. If

a) $P(0)$ is true

b) for all $n \geq 0$ it holds that $P(n)$ implies $P(n+1)$,

then $P(n)$ is true for all $n \geq 0$.

1.4.1 Invariance

As mentioned we'll make extensive use of the induction principle in connection with transition systems, but we'll reformulate it in terms of configurations and transitions below. The new formulation, called the ***invariance principle***, is more convenient for our purpose, although it is equivalent in power to the induction principle.

If $S = (C, T)$ is a transition system, we'll often be interested in showing that all configurations in a sequence or process have some property. Consider again the transition system Football from Example 1.3.8 and recall that the sequences are of the form

$$[0, A, 0, 0], [1, B, 0, 0], [3, A, 0, 1], \ldots$$

It should be clear that in this model of a football match the number of goals is at most the number of minutes elapsed. Let's see how to prove this formally.

We want to show that for all configurations $[t, X, a, b]$ (where X is either A, B, or R) the following holds: in a sequence starting with $[0, A, 0, 0]$ we have $a + b \leq t \leq 90$. This can be done by showing that the statement $a + b \leq t \leq 90$ holds in the initial configuration and that its truth is preserved by all the transition rules. The latter is proved by showing that for any configuration $[t, X, a, b]$ such that $a + b \leq t \leq 90$, it holds that $a' + b' \leq t' \leq 90$ for any successor configuration $[t', X', a', b']$.

It is clear that the property I, defined by

$$I([t, X, a, b]) : a + b \leq t \leq 90,$$

is satisfied by the initial configuration $[0, A, 0, 0]$. In this example it is also easy to show that I is preserved by the transition rules. As for the first rule we must show that if the configuration $[t, A, a, b]$ satisfies I, then so does the successor configuration $[t + 2, B, a, b]$—in other words, we must show the implication

$$a + b \leq t \leq 90 \quad \Rightarrow \quad a + b \leq t + 2 \leq 90.$$

That

$$a + b \leq t \quad \Rightarrow \quad a + b \leq t + 2$$

is obvious, but it is not so clear that

$$t \leq 90 \quad \Rightarrow \quad t + 2 \leq 90;$$

in fact, it is not true. However, we should remember that the condition

if $t \leq 88$

in the transition rule ensures that the configuration $[t, A, a, b]$ is only followed by $[t + 2, B, a, b]$ if $t \leq 88$. Therefore we know *both* that $I([t, A, a, b])$ is true *and* that $t \leq 88$, and so the implication to be proved is

$$I([t, A, a, b]) \wedge (t \leq 88) \quad \Rightarrow \quad I([t + 2, B, a, b]),$$

or in other words,

$$a + b \leq t \leq 88 \quad \Rightarrow \quad a + b \leq t + 2 \leq 90$$

—which is obviously true. We leave it to the reader to check that the other transition rules preserve the truth of the statements $I([t, X, a, b])$.

That all configurations in the sequence above satisfy I follows from the invariance principle which says that if the initial configuration of a sequence satisfies I and I is preserved by the transition rules then all configurations in the sequence satisfy I.

A property which is preserved by the transition rules in a system is called an ***invariant*** for the system, and we may formulate the following principle:

Invariance principle for transition systems Let $S = (C, T)$ be a transition system and let $c_0 \in C$ be a configuration. If $I(c)$ is a statement about the configurations of the system, the following holds. If

a) $I(c_0)$ is true

b) for all $(c, c') \in T$ it holds that $I(c)$ implies $I(c')$

then $I(c)$ is true for any configuration c that occurs in a sequence starting with c_0.

Invariants are used to show safety properties for sequences and processes (cf. Section 1.1): a sequence whose initial configuration satisfies an invariant will never enter the (unwanted) situation that the invariant doesn't hold.

The invariance principle can be seen as a specialization of the induction principle in the following way: if one can show the demands a) and b), then one can also complete an induction proof for

$$P(n) : I(c) \text{ is true for all } c \in C \text{ at distance } n \text{ from } c_0.$$

The base case $P(0)$ follows from a), and b) is exactly what is needed to show correctness of the induction step $P(n) \Rightarrow P(n+1)$. The induction principle now says that for all $n \geq 0$, $P(n)$ holds, and since any c in a sequence starting with c_0 is at some distance from c_0, we clearly have $I(c)$ for all such c, as wanted.

Conversely, one can view the induction principle as a specialization of the invariance principle by considering the transition system with configurations the natural numbers and the transition rule $n \triangleright n + 1$. We leave the details to the reader.

1.4.2 Termination

Above we argued informally that adding the clock to the football system ensured that all matches come to an end. In more general terms, this

means that the system has the property that all processes terminate, that is, all processes are finite sequences. Termination is an example of a liveness property(!) and cannot be expressed in terms of an invariant. An invariant says that something *always* holds in a process, whereas with termination we are interested in concluding that something *eventually* holds, namely that the process terminates. So a formal proof of termination has to depend on another principle. Let's see how the reasoning might proceed for the system of Example 1.3.8.

We could introduce a measure $\mu([t, X, a, b])$ of how far a configuration $[t, X, a, b]$ is from ending. By this we mean an upper bound of how long processes starting in this configuration can be. In the football system the measure could be the number $90 - t$ of minutes remaining plus 1 for handing over the ball to the referee:

$$\mu([t, X, a, b]) = \begin{cases} (90 - t) + 1 & \text{if } X \in \{A, B\} \\ 0 & \text{if } X = R \end{cases}$$

From the definition of configurations in Example 1.3.8 it can be seen that μ is a function from configurations into $\mathbf{N}$,

$$\mu : C_4 \to \mathbf{N}$$

Furthermore, by inspection of the transition rules it can be seen that the value of μ is decremented by at least one in any transition, that is,

$$\begin{array}{rcl} & [t, X, a, b] \quad \rhd \quad [t', X', a', b'] \\ \Downarrow & & \\ & \mu([t, X, a, b]) \quad > \quad \mu([t', X', a', b']). \end{array}$$

This means that to any sequence of configurations

$$[t_0, X_0, a_0, b_0] \rhd [t_1, X_1, a_1, b_1] \rhd [t_2, X_2, a_2, b_2] \rhd \cdots \qquad (*)$$

is associated a decreasing sequence of natural numbers

$$\mu([t_0, X_0, a_0, b_0]) > \mu([t_1, X_1, a_1, b_1]) > \mu([t_2, X_2, a_2, b_2]) > \cdots .$$

Of course, since any such sequence has as most $\mu([t_0, X_0, a_0, b_0]) + 1$ elements, we may conclude that any sequence of the form $(*)$ is finite.

This discussion leads to the following general principle for showing termination:

Termination principle for transition systems Let $S = (C, T)$ be a transition system and let $\mu : C \to \mathbf{N}$ be a function. If

for all $(c, c') \in T$ it holds that $\mu(c) > \mu(c')$

then all processes in $\mathcal{P}(S)$ are finite.

If μ satisfies the requirements of the principle, it is called a ***termination function***. Like the invariance principle, the termination principle follows from the induction principle when we use the following hypothesis:

$P(n)$: For all $c \in C$ with $\mu(c) < n$ it holds that
any process starting in c is finite.

The base case $P(0)$ is true since no configuration has a negative measure. Assuming $P(n)$, we want to show $P(n+1)$. Then only new situations are processes

$$c \triangleright c' \triangleright \cdots ,$$

where $\mu(c) = n$. From the assumption of the termination principle we know that $\mu(c') < \mu(c) = n$. Therefore, the induction hypothesis says that the process is finite from c' and so also from c. Since $\mu(c)$ is finite for any $c \in C$ it follows from $P(n)$ being true for all $n \geq 0$ that the conclusion of the termination principle is correct.

1.5 Examples

To illustrate the broad application area for transition systems we devote this section to six different examples. Each shows how to use the notions of Section 1.3 and how to reason about the properties of the processes generated by the systems.

1.5.1 Nim

In the first example we show how transition systems can describe ***games*** where two or more persons play against each other. We consider a very simple such game called Nim where two persons A and B take turns in removing one or two matchsticks from a common pile. The player to take the last matchstick has lost.

We may describe the situations in the game using configurations of the form $[A, n]$ ($[B, n]$) representing that it is A's (B's) turn, and that there are

$n \geq 0$ matchsticks left in the pile. It should be clear that the game may be described by the following transition system:

Transition system Nim
Configurations: $\{A, B\} \times \mathbf{N}$

$[A, n]$	$\triangleright$	$[B, n-2]$	**if** $n \geq 2$
$[A, n]$	$\triangleright$	$[B, n-1]$	**if** $n \geq 1$
$[B, n]$	$\triangleright$	$[A, n-2]$	**if** $n \geq 2$
$[B, n]$	$\triangleright$	$[A, n-1]$	**if** $n \geq 1$

Obviously all processes generated by Nim are finite; we may use the termination function $\mu([A, n]) = n$. So any game has a winner and a loser. Let us now assume that we are only interested in games where A starts, that is in processes with initial configuration of the form $[A, n]$. We might want to know for which n (if any) the outcome of the game is determined beforehand to A's advantage, given that A plays optimally, and *no matter* how B plays (in this case, A is said to have a ***winning strategy***).

We immediately see that in the configuration $[A, 1]$, A has lost because at least one matchstick must be removed in each move. However, in both $[A, 2]$ and $[A, 3]$ A can produce the configuration $[B, 1]$ where B has lost. Inspired by this we define the notion of a configuration "lost for B" as follows:

Definition 1.5.1 A configuration $[B, n]$ is *lost for B*, if either

1. $n = 1$ or

2. $n > 1$ and for all transitions $[B, n] \triangleright [A, m]$ there exists another transition $[A, m] \triangleright [B, k]$ where $[B, k]$ is lost for B.

□

We can now prove

Proposition 1.5.2 *Any configuration of the form* $[B, 3i + 1]$ *for* $i \geq 0$ *is lost for B.*

Proof: The proof is by induction on i, that is by using the induction principle on the statements

$$P(i) : \text{the configuration } [B, 3i + 1] \text{ is lost for } B.$$

Base case $P(0)$ says that $[B, 1]$ is lost for B which follows directly from the definition.

Induction step Assuming that $P(i)$ is true, we want to show $P(i+1)$. For the configuration $[B, 3(i+1)+1]$ there are two possible successors, $[A, 3i+3]$ and $[A, 3i+2]$. By using the first transition rule in the former case and the second transition rule in the latter case, we in both cases produce the configuration $[B, 3i+1]$ which according to the induction hypothesis is lost for B.

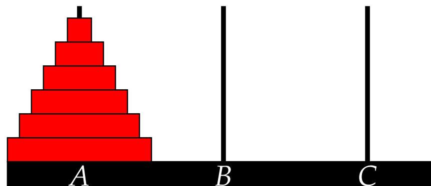
We conclude that $P(i)$ is true for all $i \geq 0$ as wanted. □

It should now be clear that if A from the outset can make a move that produces a configuration lost for B, then A can win.

In other words, we have shown that if A starts the game in a situation where there are $2, 3, 5, 6, 8, 9, \ldots$ matchsticks in the pile then A can always win.

1.5.2 Hanoi

As the second example, we let a transition system describe a ***combinatorial problem***. This class of problems include all kinds of puzzles and mazes, and we'll look at an ancient puzzle called ***Towers of Hanoi***, allegedly originating in a monastery in Tibet. Suppose we are given three pegs, A, B and C. Piled on peg A in descending order are n rings of sizes $1, \ldots, n$, while the other pegs are empty. Here is the situation with $n = 6$:



We are interested in moving the rings from A to C, perhaps using B along the way. The rules of the game say that rings are to be moved one at a time, from the top of one peg to another, and at no instant may a larger ring be placed atop a smaller one. In the original Tibetan version, $n = 64$, and according to the legend, the world will end when all 64 rings are correctly piled on peg C.

For each $n \geq 0$ we describe the Towers of Hanoi puzzle using a transition system, $\mathsf{Hanoi}(n)$, with configurations the legal placements of the n rings and transitions the legal movements of a single ring. Formally, we'll represent configurations using triples $[A, B, C]$ whose components are subsets of $\{1, \ldots, n\}$ such that each of these integers occurs in exactly one of

the subsets (we say that $\{A, B, C\}$ form a ***partition*** of $\{1, \ldots, n\}$). A ring r can be moved from peg A to peg B if r is the smallest ring in A and smaller than any ring in B, ie. if $r = \min A$ and $r < \min B$. With six combinations of source and target peg, we get:

Transition system $\mathsf{Hanoi}(n)$
Configurations: $\{[A, B, C] \mid \{A, B, C\}$ a partition of $\{1, \ldots, n\}\}$

$$
\begin{array}{lcll}
[A,B,C] & \triangleright & [A \setminus \{r\}, B \cup \{r\}, C] & \textbf{if } (r = \min A) \wedge (r < \min B) \\
[A,B,C] & \triangleright & [A \setminus \{r\}, B, C \cup \{r\}] & \textbf{if } (r = \min A) \wedge (r < \min C) \\
[A,B,C] & \triangleright & [A \cup \{r\}, B \setminus \{r\}, C] & \textbf{if } (r = \min B) \wedge (r < \min A) \\
[A,B,C] & \triangleright & [A, B \setminus \{r\}, C \cup \{r\}] & \textbf{if } (r = \min B) \wedge (r < \min C) \\
[A,B,C] & \triangleright & [A \cup \{r\}, B, C \setminus \{r\}] & \textbf{if } (r = \min C) \wedge (r < \min A) \\
[A,B,C] & \triangleright & [A, B \cup \{r\}, C \setminus \{r\}] & \textbf{if } (r = \min C) \wedge (r < \min B)
\end{array}
$$

Solving the puzzle means finding a sequence for this transition system of the form $[\{1, \ldots, n\}, \varnothing, \varnothing] \triangleright \cdots \triangleright [\varnothing, \varnothing, \{1, \ldots, n\}]$. We'll prove that any such sequence needs to be at least exponentially long in n; in fact:

Proposition 1.5.3 *At least $2^n - 1$ steps are needed to move n rings from one peg to another.*

Proof: By induction on n.

Base case With $n = 0$ clearly $0 = 2^0 - 1$ steps are needed.

Induction step To move $n + 1$ rings from source to target peg, one clearly needs to move the largest ring away from the source at some point, and we may without loss of generality assume that it is moved directly to the target. To accomplish this, it is necessary to first move the n smallest rings to the third peg which takes at least $2^n - 1$ steps by the induction hypothesis. Having moved also the largest ring, we've used 2^n moves. What remains is to move the n smallest rings back atop the largest ring, again using $2^n - 1$ steps, for a total of $2^n + 2^n - 1 = 2^{n+1} - 1$ steps as wanted.

By the induction principle, the proof is complete. □

In the Tibetan case, $2^{64} - 1$ or 18 billion billion moves are needed—so it probably shouldn't worry us that the world ends on completion.

We may turn the proof above into a recursive procedure for solving the Towers of Hanoi puzzle. Writing $\mathsf{Hanoi}(n, A, B, C)$ for the procedure for moving n rings from A to C via B, we have

$\mathsf{Hanoi}(n, A, B, C)$: if $n > 0$ then carry out $\mathsf{Hanoi}(n - 1, A, C, B)$
move the ring n from A to C
carry out $\mathsf{Hanoi}(n - 1, B, A, C)$

The number of moves performed by following this recipe is clearly a function of n, so let's write it as $H(n)$. It satisfies the equation

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

This kind of equation is called a ***recurrence equation***, because H recurs on the right-hand side. Such equations frequently arise when analyzing the running time of recursive algorithms, and we'll have more to say about them later. We want to find a closed expression for H, saying what $H(n)$ is without using H again. From the result above, we may guess that $H(n) = 2^n - 1$ and that can indeed be verified by a simple induction on n. This means that the procedure is optimal in the sense that is uses the least possible number of moves.

1.5.3 Euclid

The third example shows that an ***algorithm*** can be seen as a transition system. The following is a transition system formulation of the usual version of Euclid's algorithm for finding the greatest common divisor of two integers $m, n \geq 1$ (written $\gcd(m, n)$):

Transition system Euclid
Configurations: $\{[m, n] \mid m, n \geq 1\}$

$[m, n]$	$\triangleright$	$[m-n, n]$	**if** $m > n$
$[m, n]$	$\triangleright$	$[m, n-m]$	**if** $m < n$

The idea is that a process starting in the configuration $[m, n]$ ends up in a configuration that represents $\gcd(m, n)$. More precisely we have

Proposition 1.5.4 *All processes for* Euclid *are finite and for any process*

$$[m_0, n_0] \triangleright [m_1, n_1] \triangleright \cdots \triangleright [m_k, n_k]$$

it holds that $m_k = n_k = \gcd(m_0, n_0)$.

Proof: That all processes are finite follows from the termination principle by using the termination function

$$\mu([m, n]) = m + n.$$

Since a process

$$[m_0, n_0] \triangleright [m_1, n_1] \triangleright \cdots \triangleright [m_k, n_k]$$

is a sequence that cannot be extended we have that none of the transition rules may be used on $[m_k, n_k]$. Hence, we have neither $m_k > n_k$ nor $m_k < n_k$, and so clearly $m_k = n_k$. To show that m_k equals $\gcd(m_0, n_0)$, we show using the invariance principle that for *all* configurations $[m, n]$ in the process we have $\gcd(m, n) = \gcd(m_0, n_0)$:

$$I([m, n]) : \gcd(m, n) = \gcd(m_0, n_0).$$

So we are to prove the following:

a) $I([m_0, n_0])$ is true

b) $I([m, n])$ is preserved by $\triangleright$.

The proof of a) is trivial and for the proof of b) we first observe that if both m, n, and $m - n$ are positive integers, then for any (other) positive integer d it holds that

$$d \text{ divides both } m \text{ and } n \quad \Leftrightarrow \quad d \text{ divides both } m - n \text{ and } n \qquad (*)$$

We leave the easy proof of this to the reader. Now, because $\gcd(m, n)$ divides both m and n, it is also a common divisor of $m - n$ and n, according to $(*)$. Hence it is $\leq$ the *greatest* common divisor, $\gcd(m - n, n)$. So

$$\begin{array}{rcll} \gcd(m, n) & \leq & \gcd(m - n, n) & \text{and, symmetrically,} \\ \gcd(m - n, n) & \leq & \gcd(m, n) & \end{array}$$

which implies

$$\gcd(m, n) \quad = \quad \gcd(m - n, n).$$

To say that the truth of $I([m, n])$ is preserved by $\triangleright$ is the same as saying that these two implications are true:

$$\begin{array}{lcl} I([m, n]) \wedge (m > n) & \Rightarrow & I([m - n, n]) \\ I([m, n]) \wedge (n > m) & \Rightarrow & I([m, n - m]). \end{array}$$

Because of the symmetry, it is enough the show the former implication which is equivalent to

$$\begin{array}{l} (\gcd(m, n) = \gcd(m_0, n_0)) \wedge (m > n) \\ \Downarrow \\ \gcd(m - n, n) = \gcd(m_0, n_0) \end{array}$$

But this implication follows from the reasoning above because the condition $m > n$ ensures that all the numbers are positive. □

As the example suggests, any algorithm may be formulated as a transition system, whose configurations have a component for each of the variables of the algorithm, and where the transition relation specifies the individual steps of its execution. We'll return to this in Chapter 2.

1.5.4 Expressions

Our fourth example shows how transition systems can describe the way syntax is built up according to a ***grammar***. Here is a grammar describing simple expressions like $7-(3+4)$:

$$\begin{array}{lcl}\mathbf{Exp} & ::= & \mathbf{Term} \mid \mathbf{Term} + \mathbf{Exp} \mid \mathbf{Term} - \mathbf{Exp} \\ \mathbf{Term} & ::= & 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \mid (\mathbf{Exp})\end{array}$$

The grammar says that we can build the expression $7-(3+4)$ by going through the steps

$$\begin{array}{lclclcl}\mathbf{Exp} & \rightarrow & \mathbf{Term} - \mathbf{Exp} & \rightarrow & 7 - \mathbf{Exp} & \rightarrow \\ 7 - \mathbf{Term} & \rightarrow & 7 - (\mathbf{Exp}) & \rightarrow & 7 - (\mathbf{Term} + \mathbf{Exp}) & \rightarrow \\ 7 - (3 + \mathbf{Exp}) & \rightarrow & 7 - (3 + \mathbf{Term}) & \rightarrow & 7 - (3+4) & \end{array}$$

We'll see how to make these steps the transitions of a transition system. For conciseness we simplify the grammar a bit:

$$\begin{array}{lcl}\mathbf{E} & ::= & \mathbf{T} \mid \mathbf{T} + \mathbf{E} \\ \mathbf{T} & ::= & 0 \mid 1 \mid (\mathbf{E})\end{array}$$

We can view the grammar as a compact version of the following transition system:

Transition system Expressions
Configurations: $\{0, 1, +, \mathbf{E}, \mathbf{T}, (,)\}^*$

$$\begin{array}{lcl}\alpha\mathbf{E}\beta & \triangleright & \alpha\mathbf{T}\beta \\ \alpha\mathbf{E}\beta & \triangleright & \alpha\mathbf{T}{+}\mathbf{E}\beta \\ \alpha\mathbf{T}\beta & \triangleright & \alpha 0\beta \\ \alpha\mathbf{T}\beta & \triangleright & \alpha 1\beta \\ \alpha\mathbf{T}\beta & \triangleright & \alpha(\mathbf{E})\beta\end{array}$$

The transition rules express that in any configuration (which is a sequence of 0's, 1's, +'s, **E**'s, **T**'s, ('s and)'s) any occurrence of **E** and **T** may be replaced by the corresponding right-hand sides of the grammar (separated by |). The symbols **E** and **T** that may be replaced are called ***non-terminals*** and the other symbols are called ***terminals***. A grammar as this one where the replacement may take place independently of the surrounding symbols is called ***context-free***.

Context-free grammars are widely used to describe syntax of programming languages and as a basis for writing programs (interpreters and compilers) that among other things recognize those sequences of terminal

symbols that are legal programs, that is, those sequences that may be generated from a given non-terminal like **E** above. When a language is given in terms of a grammar, we may use the grammar to prove properties of the language. As an example, we may prove that in any legal expression, the number of left parentheses equals the number of right parentheses, and that the number of operators plus one equals the number of operands. We'll write $|\alpha|^a$ for the number of a's in the sequence α:

Proposition 1.5.5 *For any finite process of the form*

$$\mathbf{E} = \alpha_0 \rhd \alpha_1 \rhd \alpha_2 \rhd \cdots \rhd \alpha_n$$

we have

$$|\alpha_n|^{(} = |\alpha_n|^{)} \quad \textit{and} \quad |\alpha_n|^{+} + 1 = |\alpha_n|^{0} + |\alpha_n|^{1}$$

Proof: We use the invariance principle with the following invariant

$$\begin{array}{ll} I(\alpha): & |\alpha|^{(} = |\alpha|^{)} \quad \wedge \\ & |\alpha|^{+} + 1 = |\alpha|^{\mathbf{T}} + |\alpha|^{\mathbf{E}} + |\alpha|^{0} + |\alpha|^{1} \end{array}$$

Clearly, $I(\mathbf{E})$ is true and it is also easy to see that the truth of the invariant is preserved by all the transition rules of the system. The result now follows because since the process is finite, α_n is a dead configuration and so α_n contains neither **E**'s nor **T**'s, and so $|\alpha_n|^{\mathbf{T}} = |\alpha_n|^{\mathbf{E}} = 0$. □

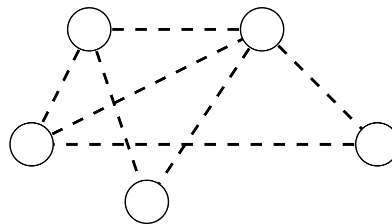
Although transition systems are more general than context-free grammars, the grammars are described more compactly. With the aim of getting the best of both worlds, we'll make our template for the description of transition systems more flexible in two ways, inspired by grammars: First, we allow rules with the same left-hand side and condition to be written as one rule with alternative right-hand sides separated by commas. Second, and more importantly, we allow the rules to be context-free whenever it makes sense. This means that left-hand sides need only match part of a configuration, and only the matched part is replaced by the right-hand side when taking the transition. Incorporating these improvements, we obtain the following alternative description of Expressions:

Transition system Expressions		
Configurations: $\{0, 1, +, \mathbf{E}, \mathbf{T}, (,)\}^*$		
E	$\rhd$	$\mathbf{T}, \mathbf{T}{+}\mathbf{E}$
T	$\rhd$	$0, 1, (\mathbf{E})$

We stress that this is mathematically the exact same transition system as above—only the description of it has changed. In the next section, we'll see another example of the power of context-freeness in the description of transition systems.

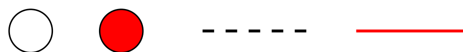
1.5.5 Graph traversal

The example of this section shows how to use transition systems to describe manipulation of ***graphs*** like

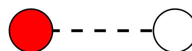


The blobs are called ***nodes*** and the lines connecting them are called ***edges***. Graphs are often used to describe structures of "places" (eg. cities) and "connections" (eg. airline routes). The graph above is ***undirected*** in the sense that we talk of connections *between* places rather than from one place to another.

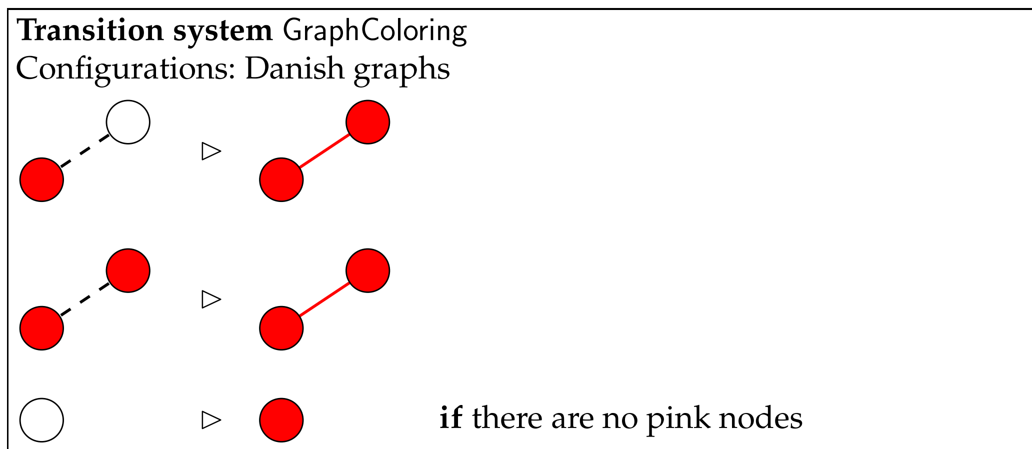
Below we'll describe ***traversal*** of such a graph as a *coloring process* where the nodes and edges of the graph, initially white, are gradually colored red. The red parts of the graph are the places and connections already visited during the traversal. If we call partially colored, undirected graphs *Danish* and a red node incident to at least one white edge *pink*, the transition system below specifies exactly what is meant by a traversal. In the description of the system,



denote white and red nodes, respectively white and red edges, and configurations are matched in a context-free way, such that eg.



matches an arbitrary Danish graph containing a red and a white node, connected by a white edge. The reader may like to consider how to describe this transition system without using context-freeness.



We'll now prove that the traversal, thus described, has the following properties:

Proposition 1.5.6 *Any process of* GraphColoring *starting with a finite graph is finite and ends with a configuration where all nodes and edges are red. The length of a process is at most $n+m$ where n (m) is the number of white nodes (edges) in the initial configuration.*

Proof: We show the first part, leaving the other to the reader. Termination is obtained by using the number of white nodes and edges as termination function. Now, consider a dead configuration for a process. It cannot contain any pink nodes because any such would allow us to take one of the first two transitions. And so it also cannot contain any white nodes, because that would allow the third transition to take place. But when there are no white or pink nodes, there can be no white edges either, because there are no nodes for them to connect. □

1.5.6 Red-black search trees

Our last example employs a transition system to describe operations of an ***abstract data type***. We presume familiarity with ***binary search trees*** and how they are used to implement the abstract data type Dictionary so that the methods Search, Insert, and Delete all run in time proportional to the height of the search tree.

We may describe this use of a search tree by a transition system whose configurations are binary search trees and where the transition relation contains the pair (T_1, T_2) of trees if T_2 is the result of using one of the above methods on T_1. This transition system has the following invariant

Invariant I_1 In each internal node there is a key k and all the keys in the left (right) subtree of the node are less (larger) than k.

It is clearly advantageous to keep the height of the search trees as small as possible, and this can be done by keeping each tree ***balanced***, such that all paths from the root to a leaf are of approximately the same length. There are many ways to ensure this, but all involve strengthening the invariant that the trees must satisfy. From this extra information one must be able to conclude that the trees have height logarithmic in the number of nodes.

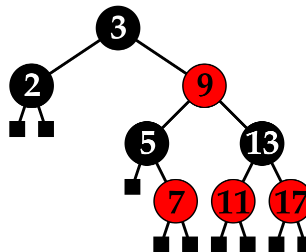
Definition 1.5.7 A ***red-black tree*** is a binary search tree in which all internal nodes are colored either red or black, in which the leaves are black, and

Invariant I_2 Each red node has a black parent.

Invariant I_3 There is the same number of black nodes on all paths from the root to a leaf.

□

The following is an example of a red-black tree:



Since at least every other node on a path must be black, the longest path from the root to a leaf is at most twice as long as the shortest. From this it follows that all paths in the tree have logarithmic lengths. In proof, let the tree have n internal nodes and let the number of edges on the shortest and longest paths be e_{min}, respectively e_{max}. Since the part of the tree lying "above level e_{min}" is a complete binary tree (that is, all possible nodes are present), and since the whole tree lies "above level e_{max}", it follows that

$$2^{e_{\mathrm{min}}} \leq n+1 \leq 2^{e_{\mathrm{max}}} \quad \text{and so} \quad e_{\mathrm{min}} \leq \log(n+1) \leq e_{\mathrm{max}},$$

so the shortest path has logarithmic length. As the longest path (which determines the height of the tree) is at most twice as long, it (and so any other path) is also of logarithmic length.

The problem remains of writing the three methods of Dictionary so that they preserve all three invariants, I_1, I_2, and I_3. Search needs no changes; a red-black tree is in particular a search tree. The problems lie in Insert and Delete: here, the old implementations may lead to violations of the invariants I_2 and I_3.

Insertion

After having inserted a new node in place of an old leaf in a red-black tree, we have a problem. We cannot just color this node; it cannot be red if it has a red parent (I_2) and it cannot be black because that generally gives one too many black nodes on the paths from the root to the two new leaves (I_3).

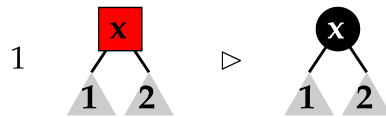
However, it turns out that we can always get away with either coloring the node or sending it further up the tree. The following is a transition system which shows us how to do this in different situations. On each side of $\triangleright$ in the rules there is (almost) a red-black tree. The idea is that if we have a tree matching the left-hand side, then we may replace it by the tree on the right-hand side, and for most of the rules, the matching is done context-freely so that we in fact only replace subtrees by other subtrees. The trees contain a "phony" node that remains to be colored. We'll call it an *illegitimate red node* and in the figures below it will look as follows:

Now, the algorithm for insertion proceeds by first inserting the element as in a normal search tree, but contained in an illegitimate red node. Then the rules in the transition system below are used on the tree, until we reach a dead configuration (which happens when the illegitimate node is properly colored). The invariant of the system is a weakened version of $I_1 \wedge I_2 \wedge I_3$ from above in that I_2 is replaced by

Invariant I_2': Each *legitimate* red node has a black parent.

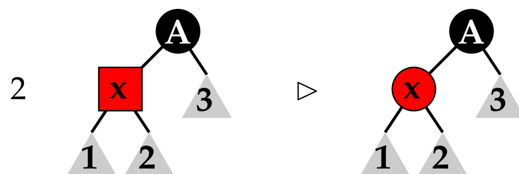
The transition relation is defined by case analysis on the different possible placements of the illegitimate node in the tree. We use the terminology of a (male) family tree:

The illegitimate node is the root of the tree In this case it is easy; we just use the following rule:



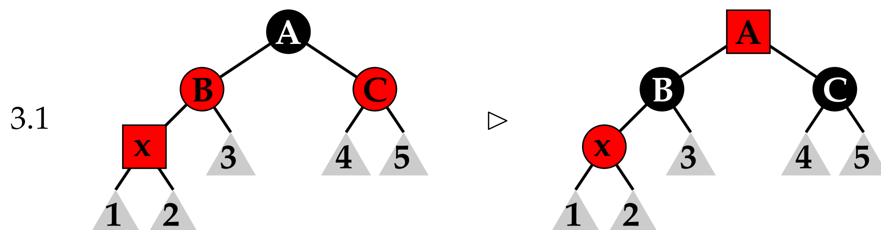
We may simply color the illegitimate node black because we then increase the number of black nodes on *all* paths. Notice that this rule is *not* context-free because its correctness depends on the illegitimate node being the root of the entire search tree. The rules below are all context-free.

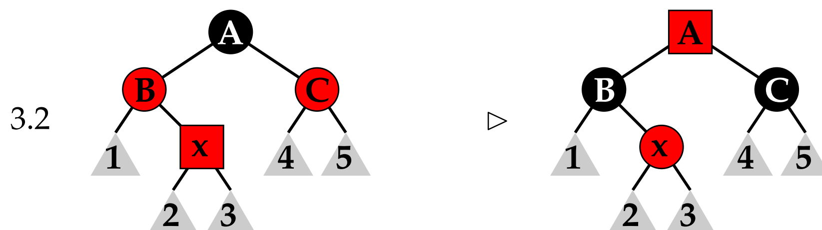
The illegitimate node has a black father Here, we may just color the illegitimate son red:



This is correct because it follows from the invariant that the roots of the subtrees 1 and 2 are black.

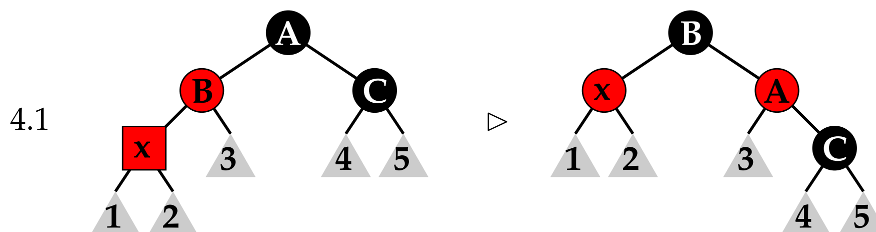
The illegitimate node has a red father and a red uncle Here, we perform a recoloring while passing the buck further up the tree. There are two sub-cases, depending on whether the illegitimate node is a left or a right son:



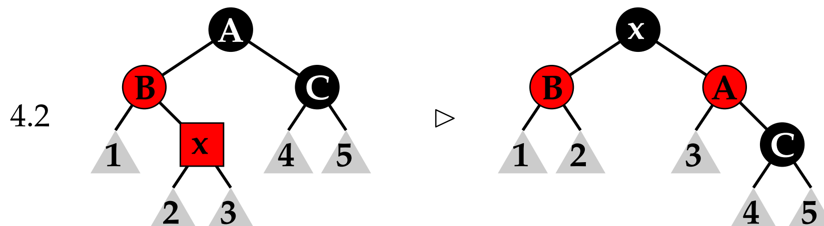


Since the nodes B and C are colored black, the paths through them contain an extra black node. In return for getting rid of it we make A into an illegitimate red node.

The illegitimate node has a red father and a black uncle In this case, a rearrangement of the tree is needed. There are again two sub-cases:



Notice that the tree is "rotated" about the axis x-B-A. Such a rotation will always preserve the order invariant I_1. It is easy to see that the other invariants are also preserved.



Being a rotation, this extensive transformation leaves I_1 intact, and the recoloring of nodes x and A ensures preservation of I_2' and I_3 also.

It is easy to see that the above six transitions (and their symmetric variants obtained by reflection in the vertical line through the root of the subtrees) are *complete* in the sense that as long as the tree contains an illegitimate node, we can always use (exactly) one of them. Furthermore, since we in each transition either eliminate the illegitimate node or push it further up the tree, it is also clear that it eventually disappears. But if all nodes in the tree are legitimate, the invariant I_2' is equivalent to I_2, so that after the rebalancing we end up with a correct red-black tree. The transition system is thus correct.

Deletion

When removing a node from a search tree, we only need to consider the situation where it has at most one non-trivial subtree (that is, a subtree which is not a leaf). Recall that if it had two non-trivial subtrees we would have swapped the element in it with its immediate successor (or predecessor), which cannot have a non-trivial left (right) subtree. We now mentally remove both the element in the node and the associated leaf and are at this point left with a "hole" with only one son. If the hole is red, we may remove it immediately. If the hole is black but has a red son, we may re-establish the invariant by removing the hole and color the son black. If the son is black we have to use a transition system as we did for insertion. We draw the hole as follows

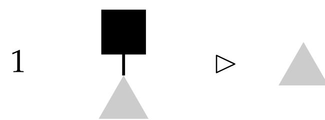
and call it an *illegitimate black node*. This time we change both I_1 and I_2:

Invariant I_1' The tree satisfies I_1 if we remove the illegitimate node.

Invariant I_2' Each red node has a *legitimate* black father.

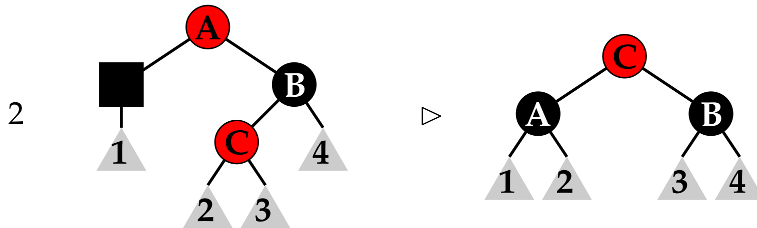
The transition system follows below. Again we can show that progress is made in each transition, and since there is always an applicable rule as long as the tree contains an illegitimate node, we have again that all processes terminate and that any dead configuration is a correct red-black tree.

The illegitimate node is the root It follows from I_2' that the son of the hole is black so we may simply remove the hole:



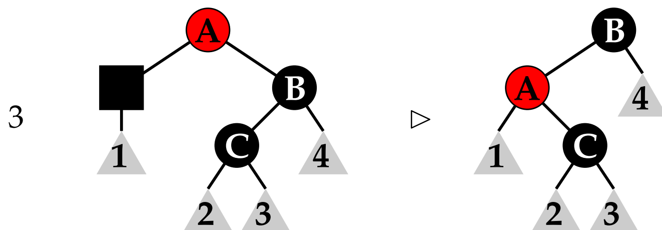
Contrary to the rules below, this rule is *not* context-free.

The illegitimate node has a red father and a red closer nephew The hole disappears by rearrangement:

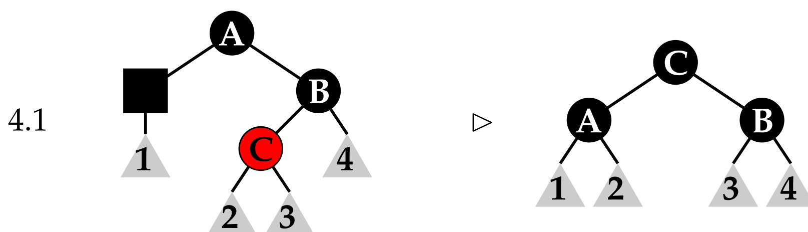


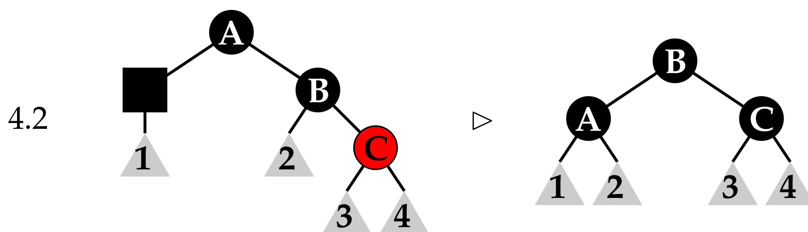
The invariants are clearly preserved.

The illegitimate node has a red father and a black closer nephew Again the hole disappears by an invariant-preserving rearrangement:

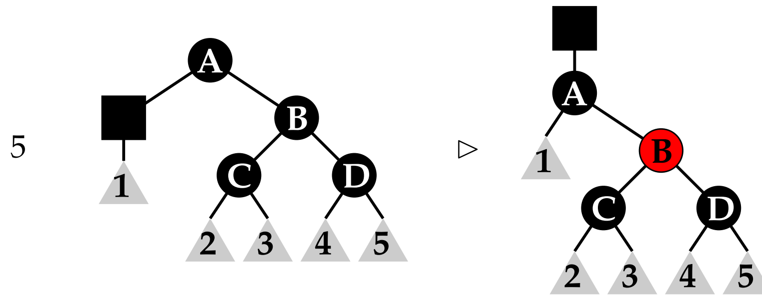


The illegitimate node has a black father, a black sibling and one red nephew We rearrange; there are two cases:

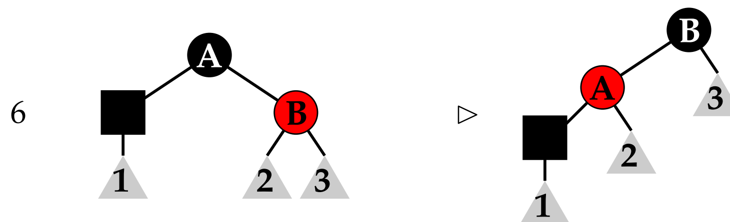




The illegitimate node has a black father, a black sibling and two black nephews This time, we cannot remove the hole; instead we send it further up the tree:



The illegitimate node has a black father and a red sibling In this last case, the transition doesn't look like we're making any progress:



—but since the root of subtree 2 must be black, we can finish up with either rule 2 or 3.

Chapter 2

Algorithm theory

In this chapter, we go into details with the idea of Section 1.5.3 to view algorithms as transition systems. The general theory of transition systems provides us with tools to reason formally about qualitative aspects like correctness and quantitative aspects like running time.

2.1 Commands

There are many ways to describe algorithms, ranging from informal prose or pictures to formal programming language code. Our approach will be tilted towards the formal side, but we want to avoid complicating our descriptions by the intricacies of a real programming language. We therefore use a small, semi-formal language of ***pseudo-code***. To start with, pseudo-code will be sequences of the form $C = c_1; c_2; \cdots; c_k$ where each c_i is one of these three ***commands***:

$x \leftarrow e$	(assignment)
if b **then** C_1 **else** C_2	(selection)
while b **do** C	(loop)

Here, C_1, C_2 and C are again sequences of commands, while x is a variable and e and b are expressions (we demand that b has type bool). We'll use standard mathematical notation for expressions and assume that they are evaluated in the same way as in Java. In particular, the boolean operators $\wedge$ and $\vee$, corresponding to Java's `&&` and `||`, are ***short-circuiting*** meaning that eg. $b_1 \wedge b_2$ is evaluated by first evaluating b_1 and only if the result is true is b_2 evaluated. So an expression like $(x \neq 0) \wedge (y/x > 2)$ evaluates to false when $x = 0$ rather than giving a division-by-zero error. We'll write **if** b **then** C as a shorthand for **if** b **then** C **else** λ where λ is the empty sequence.

At any stage during the execution of a command-sequence, a number of the variables of the sequence have been assigned a value. We'll call this assignment of values to variables a ***state***, and we can describe it using a table like the following:

Variable	Value
x	35
y	true
z	3.1416
$\vdots$	$\vdots$

One may view the state as a function mapping variables to values. If we call the state in the table above σ, we have $\sigma(x) = 35$. Given a state we may compute the value of an expression containing the variables of the state. For example, the expression $x < 19 \vee y$ will have the value true in the state σ. We'll use the notation $\sigma(e)$ for the value of the expression e in σ and thus write $\sigma(x < 19 \vee y) = \mathsf{true}$. When b is a boolean expression and $\sigma(b) = \mathsf{true}$ we say that σ ***satisfies*** b. This notion will be useful later when we want to specify the behaviour of algorithms. We shall then allow arbitrary mathematical notation, saying for example that σ satisfies $z > \pi$. Here, π is a mathematical number, not an expression in pseudo-code. We call such generalized boolean expressions ***assertions***.

We'll now describe how commands and command-sequences are executed in a given state σ:

- The assignment $x \leftarrow e$ changes the value of x to $\sigma(e)$ and leaves σ unchanged in other variables. We'll write $\sigma\langle x{:}e\rangle$ for the new state.

- The selection **if** b **then** C_1 **else** C_2 is executed by executing C_1 in σ if $\sigma(b) = \mathsf{true}$, and C_2 in σ if $\sigma(b) = \mathsf{false}$.

- The loop **while** b **do** C has no effect if $\sigma(b) = \mathsf{false}$, but if $\sigma(b) = \mathsf{true}$, the sequence C;**while** b **do** C is executed in σ.

- The command-sequence $c_1; c_2; \cdots; c_k$ is executed by first executing c_1 in σ which yields a new state σ' in which we execute c_2 etc. The empty sequence, λ, has no effect.

Starting from this description, it is fairly easy to construct a transition system capturing the effect of executing command-sequences. The configurations are pairs of the form $[C, \sigma]$ where C is the command-sequence left to be executed and σ is the current state. The configuration $[\lambda, \sigma]$ will be written just σ.

A process for the transition system will describe how the state "runs through" the command-sequence and changes along the way. The possible transitions from a given configuration depends on the first command of the sequence and on the state:

Transition system Commands
Configurations: $\{[C, \sigma] \mid C$ a command-sequence, σ a state$\}$

$[x \leftarrow e;\ C', \sigma]$	$\triangleright$	$[C', \sigma\langle x{:}e\rangle]$	
$[\textbf{if } b \textbf{ then } C_1 \textbf{ else } C_2;\ C', \sigma]$	$\triangleright$	$[C_1;\ C', \sigma]$	**if** $\sigma(b) =$ true
$[\textbf{if } b \textbf{ then } C_1 \textbf{ else } C_2;\ C', \sigma]$	$\triangleright$	$[C_2;\ C', \sigma]$	**if** $\sigma(b) =$ false
$[\textbf{while } b \textbf{ do } C;\ C', \sigma]$	$\triangleright$	$[C;\ \textbf{while } b \textbf{ do } C;\ C', \sigma]$	**if** $\sigma(b) =$ true
$[\textbf{while } b \textbf{ do } C;\ C', \sigma]$	$\triangleright$	$[C', \sigma]$	**if** $\sigma(b) =$ false

Suppose C is the (one-element) command-sequence

while $m \neq n$ **do**
  if $m > n$ **then**
    $m \leftarrow m - n$
  else
    $n \leftarrow n - m$

and that σ is a state with $\sigma(m) = 35$ and $\sigma(n) = 77$. A process generated by the transition system Commands starting in the configuration $[C, \sigma]$ will correspond to a process generated by the transition system Euclid starting in $[35, 77]$. Each transition of the latter will require three transitions of the former, making up a full iteration of the while-loop. The reader should try this out.

2.2 Algorithms

Here is a formulation of Euclid's algorithm in the notation we'll be using in the rest of this note:

Algorithm Euclid(m, n)
Input : $m, n \geq 1$
Output : $r = \gcd(m_0, n_0)$
Method : **while** $m \neq n$ **do**
    if $m > n$ **then**
      $m \leftarrow m - n$
    else
      $n \leftarrow n - m$;
  $r \leftarrow m$

So an algorithm consists of

- a ***name*** and some ***parameters***, used in the same way as with Java methods,
- a ***specification*** of the input and output of the algorithm, and
- a ***method***—which is a command-sequence—implementing the algorithm.

For Euclid we have specified that if we execute its method in a state where m and n are both positive integers, we'll end up in a state where the value of r is the greatest common divisor of the original values of m and n. We'll always use subscript 0 to denote the original value of a variable, and we'll use "result variables" like r rather than adding a **return** command to the language. Output specifications are always given in terms of the parameters and the result variables.

Notice that the specification does not involve explicit type information. The types of variables will always be clear from context. For Euclid, the variables are of integer type, because the notion of greatest common divisor is defined only for integers (in fact, only for positive integers, but restrictions like that are captured by the input specification instead).

2.3 Correctness

The formalization above allows a precise definition of correctness. Let

Algorithm A($\cdots$)	
Input	: *In*
Output	: *Out*
Method	: C

be an algorithm with input specification *In*, output specification *Out*, and method C.

Definition 2.3.1 The algorithm A is ***correct*** if any process for the transition system Commands starting in a configuration $[C, \sigma]$, where σ satisfies *In*, is finite and ends with a configuration σ' satisfying *Out*. □

So there are two aspects of correctness: the algorithm must do the right thing, and it must finish doing it. Below, we'll see how the invariance and termination principles can be used in these two parts of a correctness proof.

2.3.1 Validity

In practice it is difficult to carry out a correctness proof for an algorithm without thoroughly understanding the algorithm, and since loops are often the hardest parts to come to terms with, we'll start with them. Consider again the loop of Euclid. As we have seen, we can describe its effect with the transition system of the same name. It seems reasonable then that the invariant for this system,

$$I : \gcd(m, n) = \gcd(m_0, n_0),$$

should provide some information about the loop. In fact, the invariant captures the whole idea of Euclid's algorithm: that the greatest common divisor of m and n doesn't change although m and n do. In more general terms, an invariant provides a handle for describing the effect of a loop without referring to the number of iterations. Because of this, invariants will be important stepping stones in our correctness proofs.

Copying the proof of Proposition 1.5.4 we can show that I is satisfied before and after each iteration of the loop of Euclid. Accordingly, we call it a ***loop invariant***, and we'll document it as follows:

$\{I\}$**while** $m \neq n$ **do**
  if $m > n$ **then**
    $m \leftarrow m - n$
  else
    $n \leftarrow n - m$

An algorithm whose method contains assertions like I about the state will be called a ***decorated algorithm***. The intention is that whenever execution reaches an assertion, the assertion should be satisfied by the state at that point. Accordingly, we want loop invariants to show up before and after each iteration, and we therefore extend our transition system Commands to deal with decorated commands, so that it has transitions like eg.

$$[\{I\}\textbf{while } b \textbf{ do } C; C', \sigma] \triangleright [C; \{I\}\textbf{while } b \textbf{ do } C; C', \sigma] \textbf{ if } \sigma(b) = \text{true}.$$

Most of our assertions will be invariants for loops, but we'll also use assertions in other places as needed for clarity—well-placed assertions provide good documentation of an algorithm. In particular, we'll implicitly regard the input and output specifications of an algorithm as assertions placed at the beginning and end of the method, respectively.

Definition 2.3.2 An assertion U of a decorated algorithm is ***valid for a process*** if for all configurations of the form $[\{U\}C, \sigma]$ in the process, the assertion U is satisfied by the state σ. □

Consider again the generic algorithm A from before.

Definition 2.3.3 The algorithm A is ***valid*** if all its assertions are valid for all processes starting in a configuration $[C, \sigma]$ where σ satisfies In. □

Since we insist that the output specification is implicitly added as an assertion after the last command of the method, *a valid algorithm is correct if and only if it terminates*. What termination means will be made clear in the next section. Here, we'll see how to prove validity in practice.

We can prove validity by showing that it is preserved from assertion to assertion following the control-flow of the algorithm, and starting with the input specification. Suppose we have an algorithm in which two assertions U and V are separated by a command-sequence C. Showing that validity is preserved then amounts to the following so-called ***proof-burden***

> For any state σ, if σ satisfies U and the execution of C in σ leads to σ' (ie. $[C, \sigma] \rhd^* \sigma'$), then σ' must satisfy V.

We'll write such a proof-burden in the same way as it appears in the decorated pseudo-code:

$$\{U\}C\{V\}$$

If C is just a sequence of assignments, we can prove the proof-burden directly in the following way.

We start by writing the values of all variables after the assignments as functions of the values before. For example, if C is the (rather silly) sequence

$$x \leftarrow x + y; \quad y \leftarrow x + z; \quad z \leftarrow x; \quad x \leftarrow 3$$

we'll write (using primed variables for the values after the assignments):

$$x' = 3, \qquad y' = x + y + z, \qquad \text{and} \qquad z' = x + y.$$

This says that after the assignments, x holds the value 3, y holds the sum of the previous values of x, y and z, and z holds the sum of the previous values of x and y. Now suppose that our assertions are

$$U : z = x + y \qquad \text{and} \qquad V : y = 2z.$$

Proving the proof-burden $\{U\}C\{V\}$ is then simply a matter of showing that if U is true for x, y, z then V is true for x', y', z', in other words that

$$z = x + y \quad \Rightarrow \quad y' = 2z'.$$

Inserting the expressions above for the primed variables, this is the same as showing

$$z = x + y \quad \Rightarrow \quad x + y + z = 2(x + y).$$

—which is clearly true. We highlight this method as a proof principle:

Proof principle for simple proof-burdens Let $C = c_1; \cdots ; c_k$ be a sequence of assignments and let $x_1, \ldots, x_n$ be the variables of the algorithm. Suppose that C executed in σ leads to σ'. Then the proof-burden $\{U\}C\{V\}$ is proved by proving the implication

$$U(x_1, \ldots, x_n) \Rightarrow V(x'_1, \ldots, x'_n)$$

—where $x'_1, \ldots, x'_n$ are the values of the variables in σ' expressed as functions of their values in σ.

Notice that if C is the empty sequence λ, we just need to show the implication $U \Rightarrow V$. We'll therefore write $U \Rightarrow V$ as a synonym for the proof-burden $\{U\}\lambda\{V\}$.

If C is more complicated than just a sequence of assignments, we'll introduce new assertions to divide up the proof-burden $\{U\}C\{V\}$ into proof-burdens with less complicated command-sequences. We can repeat this process until we end up with a collection of proof-burdens, all of the simple form above. The division of proof-burdens can be done in a systematic way according to the following proof principle:

Proof principle for compound proof-burdens A proof-burden of the form $\{U\}C_1; C_2\{V\}$ gives rise to the proof-burdens

$$\{U\}C_1\{W\} \quad \text{and} \quad \{W\}C_2\{V\}.$$

A proof-burden of the form $\{U\}$**if** b **then** C_1 **else** $C_2\{V\}$ gives rise to the proof-burdens

$$\{U \wedge b\}C_1\{V\} \quad \text{and} \quad \{U \wedge \neg b\}C_2\{V\}.$$

A proof-burden of the form $\{U\}$**while** b **do** $C\{V\}$ gives rise to the proof-burdens

$$\begin{array}{ll} U \Rightarrow I & \text{(basis)} \\ \{I \wedge b\}C\{I\} & \text{(invariance)} \\ I \wedge \neg b \Rightarrow V & \text{(conclusion).} \end{array}$$

For proof-burdens of the last form, I is a loop invariant, shown to be true the first time the loop is reached (given that U is) by proving the basis, shown to be preserved by the loop body by proving invariance, and used to establish V after termination by proving the conclusion.

The principle provides no guidance as to how to come up with the assertions W and I in the principles for sequences and loops. However, as we shall see in Chapter 3, the algorithms and their correctness proofs may be developed hand in hand (with the proof ideas leading the way!), so that invariants and other central assertions are already there when needed. Supposing that we had developed Euclid and its invariant $I : \text{gcd}(m, n) = \text{gcd}(m_0, n_0)$ in this way, we would proceed as follows:

Example 2.3.4 Initially, the input specification, method, and output specification of Euclid give us the following compound proof-burden:

$$1: \quad \{m, n \geq 1\}\textsf{while } m \neq n \textsf{ do } \cdots ;\ r \leftarrow m\{r = \text{gcd}(m_0, n_0)\}$$

We divide it using the proof principle for sequences. What should the assertion W be? Since it is placed immediately after a while-loop with invariant I and condition $m \neq n$, we can use $W : I \wedge (m = n)$ because this is what we know holds after termination. We get:

$$2: \quad \{m, n \geq 1\}\textsf{while } m \neq n \textsf{ do } \cdots\{I \wedge (m = n)\}$$
$$3: \quad \{I \wedge (m = n)\}r \leftarrow m\{r = \text{gcd}(m_0, n_0)\}$$

Proof-burden 3 is simple. We use the principle for loops to divide proof-burden 2 into

$$4: \quad m, n \geq 1 \Rightarrow I$$
$$5: \quad \{I \wedge (m \neq n)\}\textsf{if } m > n \textsf{ then } m \leftarrow m - n \textsf{ else } n \leftarrow n - m\{I\}$$
$$6: \quad I \wedge \neg(m \neq n) \Rightarrow I \wedge (m = n)$$

Proof-burden 6 is trivial because of our choice of W above, so we'll ignore it. Proof-burden 4 is simple (it is the same as $\{m, n \geq 1\}\lambda\{I\}$). We split up proof-burden 5 using the principle for selections:

$$7: \quad \{I \wedge (m \neq n) \wedge (m > n)\}m \leftarrow m - n\{I\}$$
$$8: \quad \{I \wedge (m \neq n) \wedge \neg(m > n)\}n \leftarrow n - m\{I\}$$

Both are simple. In total, we get the following four simple proof-burdens:

$$3: \quad \{I \wedge (m = n)\}r \leftarrow m\{r = \text{gcd}(m_0, n_0)\}$$
$$4: \quad m, n \geq 1 \Rightarrow I$$
$$7: \quad \{I \wedge (m > n)\}m \leftarrow m - n\{I\}$$
$$8: \quad \{I \wedge (m < n)\}n \leftarrow n - m\{I\}$$

Proof-burdens 4, 7, and 8 are already shown as part of the proof of Proposition 1.5.4 (as for proof-burden 4, remember that $m = m_0$ and $n = n_0$ holds initially). To prove proof-burden 3, we must show the implication

$$I \wedge (m = n) \quad \Rightarrow \quad r' = \text{gcd}(m_0, n_0).$$

Inserting $r' = m$, $m' = m$, and $n' = n$, this is

$$(\text{gcd}(m,n) = \text{gcd}(m_0,n_0)) \wedge (m = n) \quad \Rightarrow \quad m = \text{gcd}(m_0,n_0)$$

So assume $\text{gcd}(m,n) = \text{gcd}(m_0,n_0)$ and $m = n$. We must prove $m = \text{gcd}(m_0,n_0)$ and argue as follows:

$$\begin{array}{lll} & m = \text{gcd}(m_0,n_0) & \\ \Leftrightarrow & m = \text{gcd}(m,n) & \text{since } \text{gcd}(m,n) = \text{gcd}(m_0,n_0) \\ \Leftrightarrow & m = \text{gcd}(m,m) & \text{since } m = n \end{array}$$

Ending up with something trivially true, we have proved proof-burden 3.
□

2.3.2 Termination

Consider once again our generic algorithm A:

Algorithm A$(\cdots)$	
Input	: *In*
Output	: *Out*
Method	: *C*

Even if A is valid it need not be correct, because validity only says that the assertion *Out* at the end of the method *C* holds *if we ever reach it*. Reaching this assertion is exactly what termination is about:

Definition 2.3.5 We say that A ***terminates*** if any process starting in a configuration $[C, \sigma]$, where σ satisfies *In*, is finite. □

Clearly, algorithms with just assignments and selections always terminate, so the problem lies with the loops. Again we look to the loop of Euclid and the corresponding transition system of Section 1.5.3 for a hint on how to solve it. The transition system has termination function $\mu([m,n]) = m + n$, so it is reasonable to expect the value $\sigma(m+n)$ to be a non-negative integer that decreases with every full iteration of the loop.

Indeed, consider $\mu(m,n,r) = m + n$ as a function of the variables of Euclid in states satisfying the loop-invariant $I : \text{gcd}(m,n) = \text{gcd}(m_0,n_0)$. It has the following properties:

- It is integer-valued by the typings of m and n.
- It is non-negative, since by the invariant, $\gcd(m, n)$ is well-defined, meaning that $m, n \geq 1$.
- Each iteration of the loop makes it decrease, because an iteration is only performed when the invariant holds and $m \neq n$ so that either m is larger (and then $n \geq 1$ is subtracted) or n is larger (and then $m \geq 1$ is subtracted).

We can formulate the following general proof principle:

Termination principle for algorithms Let $x_1, \ldots, x_n$ be the variables of an algoritm A. A terminates if for every loop

$$\{I\}\textbf{while } b \textbf{ do } C$$

in its method with I as valid invariant, there exists an integer-valued function $\mu(x_1, \ldots, x_n)$ satisfying

a) $I \Rightarrow \mu(x_1, \ldots, x_n) \geq 0$

b) $I \wedge b \Rightarrow \mu(x_1, \ldots, x_n) > \mu(x'_1, \ldots, x'_n) \geq 0$

—where $x'_1, \ldots, x'_n$ are the values of the variables after an iteration expressed as functions of their values before.

This completes the theory we need so far for specifying algorithms and proving them correct. Even though we have based the development on the general theory of transition systems, *there is no need to refer to transition systems when writing an algorithm or conducting a correctness proof*. All that's required is an understanding of how pseudo-code is executed and the proof principles of this section.

2.4 Examples

In this section, we go through three correctness proofs in quite some detail. The first concerns an extension of Euclid's algorithm while the second and third are prototypical examples of correctness proofs for simple loops.

2.4.1 Extended Euclid

Euclid's algorithm is easily extended so that it also computes the least common multiplum (lcm) of its input. For integers $m, n \geq 1$ the number $\text{lcm}(m, n)$ is defined as the least positive number divisible by both m and n. One can easily show the following:

$$m \cdot n = \gcd(m, n) \cdot \text{lcm}(m, n) \qquad (*)$$

Here is the algorithm:

Algorithm ExtendedEuclid(m, n)
Input : $m, n \geq 1$
Output : $(r = \gcd(m_0, n_0)) \wedge (s = \text{lcm}(m_0, n_0))$
Method : $p \leftarrow m;\ q \leftarrow n;$
$\{I\}$**while** $m \neq n$ **do**
  if $m > n$ **then**
    $m \leftarrow m - n;\ p \leftarrow p + q$
  else
    $n \leftarrow n - m;\ q \leftarrow q + p;$
$r \leftarrow m;\ s \leftarrow (p + q)/2$

It is probably not immediately clear that this algorithm is correct. So let's prove it. We can use the following invariant

$$I : (mq + np = 2m_0n_0) \wedge (\gcd(m, n) = \gcd(m_0, n_0)).$$

This may seem intimidating at first, especially if one is to come up with it just by looking at the algorithm. But again: I was there to start with and then the algorithm has been written to make I a valid invariant.

The form of the method is almost the same as for Euclid. An extra use of the principle for division of sequences with $W = I$ separates the initializations of p and q from the loop. We get the following simple proof-burdens:

$$\begin{aligned}
1: &\quad \{m, n \geq 1\} p \leftarrow m;\ q \leftarrow n \{I\} \\
2: &\quad \{I \wedge (m > n)\} m \leftarrow m - n;\ p \leftarrow p + q \{I\} \\
3: &\quad \{I \wedge (m < n)\} n \leftarrow n - m;\ q \leftarrow q + p \{I\} \\
4: &\quad \{I \wedge (m = n)\} r \leftarrow m;\ s \leftarrow (p + q)/2 \\
&\qquad\qquad\qquad \{(r = \gcd(m_0, n_0)) \wedge (s = \text{lcm}(m_0, n_0))\}.
\end{aligned}$$

Let's show them one by one:

Proof-burden 1 We must show

$$m, n \geq 1 \quad \Rightarrow \quad I(m', n', \ldots).$$

Inserting $m' = m = m_0$, $n' = n = n_0$, $p' = m$, and $q' = n$, this is

$$m, n \geq 1 \quad \Rightarrow \quad (mn + nm = 2mn) \wedge (\gcd(m, n) = \gcd(m, n)).$$

So assume that $m, n \geq 1$. $mn + nm = 2mn$ trivially holds. $\gcd(m, n) = \gcd(m, n)$ holds whenever it makes sense, that is, whenever $m, n \geq 1$. But this is exactly what we assumed.

Proof-burden 2 We must show

$$I \wedge (m > n) \quad \Rightarrow \quad I(m', n', \ldots).$$

Inserting $m' = m - n$, $n' = n$, $p' = p + q$, and $q' = q$, this is

$$\begin{aligned}&(mq + np = 2m_0n_0) \wedge (\gcd(m, n) = \gcd(m_0, n_0)) \wedge (m > n) \quad \Rightarrow \\ &((m - n)q + n(p + q) = 2m_0n_0) \wedge (\gcd(m - n, n) = \gcd(m_0, n_0)).\end{aligned}$$

So assume $mq + np = 2m_0n_0$ and $\gcd(m, n) = \gcd(m_0, n_0)$ and $m > n$. To prove $(m - n)q + n(p + q) = 2m_0n_0$ we argue as follows:

$$\begin{aligned}&(m - n)q + n(p + q) = 2m_0n_0 \\ \Leftrightarrow\ &mq - nq + np + nq = 2m_0n_0 \\ \Leftrightarrow\ &mq + np = 2m_0n_0\end{aligned}$$

—and end up with our first assumption. To prove that $\gcd(m - n, n) = \gcd(m_0, n_0)$ we argue as follows:

$$\begin{aligned}&\gcd(m - n, n) = \gcd(m_0, n_0) \\ \Leftrightarrow\ &\gcd(m, n) = \gcd(m_0, n_0) \qquad \text{since } m > n\text{; cf. Proposition 1.5.4}\end{aligned}$$

—and end up with our second assumption.

Proof-burden 3 Symmetrical to proof-burden 2.

Proof-burden 4 We must show

$$I \wedge (m = n) \quad \Rightarrow \quad (r' = \gcd(m_0, n_0)) \wedge (s' = \mathrm{lcm}(m_0, n_0)).$$

Inserting $r' = m$ and $s' = (p + q)/2$ (other variables unchanged), this is

$$\begin{aligned}&(mq + np = 2m_0n_0) \wedge (\gcd(m, n) = \gcd(m_0, n_0)) \wedge (m = n) \Rightarrow \\ &(m = \gcd(m_0, n_0)) \wedge ((p + q)/2 = \mathrm{lcm}(m_0, n_0))\end{aligned}$$

So assume $mq + np = 2m_0n_0$ and $\gcd(m,n) = \gcd(m_0,n_0)$ and $m = n$. The equation $m = \gcd(m_0,n_0)$ is shown exactly as for Euclid. To prove $(p+q)/2 = \mathrm{lcm}(m_0,n_0)$ we argue as follows:

$$\begin{array}{lll} & (p+q)/2 = \mathrm{lcm}(m_0,n_0) & \\ \Leftrightarrow & (mq+mp)/2 = m \cdot \mathrm{lcm}(m_0,n_0) & \text{since } m \neq 0 \\ \Leftrightarrow & (mq+np)/2 = m \cdot \mathrm{lcm}(m_0,n_0) & \text{since } m = n \\ \Leftrightarrow & m_0n_0 = m \cdot \mathrm{lcm}(m_0,n_0) & \text{since } mq+np = 2m_0n_0 \\ \Leftrightarrow & m_0n_0 = \gcd(m_0,n_0) \cdot \mathrm{lcm}(m_0,n_0) & \text{since } m = \gcd(m_0,n_0) \end{array}$$

—and end up with a true equation according to $(*)$.

So ExtendedEuclid is valid. To show correctness we just need a termination function and here we may use $\mu(m,n,p,q,r,s) = m+n$. The proof is the same as for Euclid. We conclude that ExtendedEuclid is correct.

2.4.2 Factorial

The following algorithm computes the factorial

$$n! = 1 \cdot 2 \cdot \cdots \cdot (n-1) \cdot n$$

of a natural number n, with the understanding that $0! = 1$:

```
Algorithm Factorial(n)
Input   : n ≥ 0
Output  : r = n0!
Method  : r ← 1;
           {I}while n ≠ 0 do
                 r ← r * n;
                 n ← n − 1;
```

As a loop invariant we may use

$$I : (r = n_0!/n!) \wedge (n_0 \geq n \geq 0).$$

The shape of this invariant is typical: it expresses how far we have come in the computation by saying that r is what we want, except for a factor $n!$. Proving validity of the algorithm, we are faced with a proof-burden of the form

$$\{n \geq 0\}C'; \textbf{while } n \neq 0 \textbf{ do } C\{r = n_0!\}.$$

According to the proof principles for division of a sequence (with $W = I$) and for loops, this gives rise to three simple proof-burdens:

$$\begin{array}{rl} 1: & \{n \geq 0\} r \leftarrow 1 \{I\} \\ 2: & \{I \wedge (n \neq 0)\} r \leftarrow r * n;\ n \leftarrow n - 1 \{I\} \\ 3: & I \wedge (n = 0) \Rightarrow r = n_0! \end{array}$$

(together with the trivial proof-burden $I \Rightarrow I$). We show them one by one:

Proof-burden 1 We must show

$$n \geq 0 \quad \Rightarrow \quad I(n', r').$$

Inserting $n' = n = n_0$ and $r' = 1$, this is

$$n \geq 0 \quad \Rightarrow \quad (1 = n!/n!) \wedge (n \geq n \geq 0)$$

So assume $n \geq 0$. $1 = n!/n!$ is true when $n!$ makes sense which it does by the assumption. The inequalities $n \geq n \geq 0$ follow directly from the assumption.

Proof-burden 2 We must show

$$I \wedge (n \neq 0) \quad \Rightarrow \quad I(n', r').$$

Inserting $r' = rn$ and $n' = n - 1$, this is

$$\begin{array}{l} (r = n_0!/n!) \wedge (n_0 \geq n \geq 0) \wedge (n \neq 0) \quad \Rightarrow \\ (rn = n_0!/(n-1)!) \wedge (n_0 \geq n - 1 \geq 0) \end{array}$$

So assume $r = n_0!/n!$ and $n_0 \geq n \geq 0$ and $n \neq 0$. To prove that $rn = n_0!/(n-1)!$ we argue as follows:

$$\begin{array}{ll} \quad rn = n_0!/(n-1)! & \\ \Leftrightarrow rn = (n_0!/n!)n & \text{since } n_0 \geq n \neq 0 \\ \Leftrightarrow r = n_0!/n! & \text{since } n \neq 0 \end{array}$$

—and end up with our first assumption. The inequalities $n_0 \geq n - 1 \geq 0$ follow from the second and third assumptions.

Proof-burden 3 We must show

$$I \wedge (n = 0) \quad \Rightarrow \quad r = n_0!$$

which is

$$(r = n_0!/n!) \wedge (n_0 \geq n \geq 0) \wedge (n = 0) \quad \Rightarrow \quad r = n_0!$$

Using the first and third assumption, the wanted conclusion follows immediately.

So Factorial is valid. To show correctness we just need a termination function and here we may use $\mu(n, r) = n$, because

a) $I \Rightarrow \mu(n, r) \geq 0$ because I says that $n \geq 0$.

b) $I \wedge (n \neq 0) \Rightarrow \mu(x, n, r, m) > \mu(x', n', r', m')$ simply because $n' = n - 1$ (we don't use the assumptions here).

We conclude that Factorial is correct.

2.4.3 Power sum

Consider an algorithm for computing the sum

$$1 + x + x^2 + \cdots + x^{n-1} + x^n$$

of the powers of $x \neq 0$ up to x^n with n a natural number. One might expect the output specification for this algorithm to be something like $r = \Sigma_{i=0}^{n} x^i$, but this is not quite good enough. A trivial way to establish it is

$$n \leftarrow 0;\ r \leftarrow 1.$$

—and clearly, this is not the intention of the algorithm; x and n should not be changed. So formally, we need to add $(x = x_0) \wedge (n = n_0)$ to all assertions used, including the output specification. Since this is a lot of bureaucracy for saying something simple, we'll introduce a convenient shorthand in the form of a ***constant clause*** in the specification of the algorithm:

```
Algorithm PowerSum(x, n)
Input     : (x ≠ 0) ∧ (n ≥ 0)
Constants: x, n
Output    : r = Σ_{i=0}^{n} x^i
Method    : r ← 1; m ← 0;
            {I}while m ≠ n do
                r ← r * x + 1;
                m ← m + 1;
```

We stress that this is merely a shorthand for writing assertions—we still need to show that $x = x_0$ and $n = n_0$ hold at each of them. However, this can be done once and for all: the method implementing the algorithm doesn't have x or n on the left-hand side of any assignment commands, so they cannot change. Accordingly, we'll treat x and n as constants in the correctness proof below.

The loop invariant is

$$I : (r = \Sigma_{i=0}^{m} x^i) \wedge (n \geq m \geq 0).$$

Again, this expresses how far we have come in the computation. Proving validity of the algorithm, we have a compound proof-burden of the same form as for Factorial, so we get these three simple proof-burdens:

$$\begin{array}{ll} 1: & \{(x \neq 0) \wedge (n \geq 0)\} r \leftarrow 1;\ m \leftarrow 0 \{I\} \\ 2: & \{I \wedge (m \neq n)\} r \leftarrow r * x + 1;\ m \leftarrow m + 1 \{I\} \\ 3: & I \wedge (m = n) \Rightarrow r = \Sigma_{i=0}^{n} x^i \end{array}$$

We show them one by one:

Proof-burden 1 We must show

$$(x \neq 0) \wedge (n \geq 0) \quad \Rightarrow \quad I(x, n, r', m').$$

Inserting $r' = 1$ and $m' = 0$, this is

$$(x \neq 0) \wedge (n \geq 0) \quad \Rightarrow \quad (1 = \Sigma_{i=0}^{0} x^i) \wedge (n \geq 0 \geq 0)$$

So assume $x \neq 0$ and $n \geq 0$. That $1 = \Sigma_{i=0}^{0} x^i$ is true by our first assumption, and $n \geq 0 \geq 0$ is true by our second assumption.

Proof-burden 2 We must show

$$I \wedge (m \neq n) \quad \Rightarrow \quad I(x, n, r', m').$$

Inserting $r' = rx + 1$ and $m' = m + 1$, this is

$$\begin{array}{l} (r = \Sigma_{i=0}^{m} x^i) \wedge (n \geq m \geq 0) \wedge (m \neq n) \quad \Rightarrow \\ (rx + 1 = \Sigma_{i=0}^{m+1} x^i) \wedge (n \geq m + 1 \geq 0) \end{array}$$

So assume $r = \Sigma_{i=0}^{m} x^i$ and $n \geq m \geq 0$ and $m \neq n$. To prove that $rx + 1 = \Sigma_{i=0}^{m+1} x^i$ we argue as follows:

$$\begin{array}{ll} rx + 1 = \Sigma_{i=0}^{m+1} x^i & \\ \Leftrightarrow rx = \Sigma_{i=1}^{m+1} x^i & \\ \Leftrightarrow rx = \Sigma_{i=0}^{m} x^{i+1} & \\ \Leftrightarrow r = \Sigma_{i=0}^{m} x^i & \text{since } x \neq 0 \end{array}$$

—and end up with our first assumption. The inequalities $n \geq m + 1 \geq 0$ follows from the second and third assumptions.

Proof-burden 3 We must show

$$I \wedge (m = n) \quad \Rightarrow \quad r = \Sigma_{i=0}^{n} x^i$$

which is

$$(r = \Sigma_{i=0}^{m} x^i) \wedge (n \geq m \geq 0) \wedge (m = n) \quad \Rightarrow \quad r = \Sigma_{i=0}^{n} x^i$$

Using the first and third assumption, the wanted conclusion follows immediately.

So PowerSum is valid. To show correctness we just need a termination function and here we may use $\mu(x, n, r, m) = n - m$, because

a) $I \Rightarrow \mu(x, n, r, m) \geq 0$ because I says that $n \geq m$.

b) $I \wedge (m \neq n) \Rightarrow \mu(x, n, r, m) > \mu(x', n', r', m')$ simply because $m' = m + 1$ (we don't use the assumptions here).

We conclude that PowerSum is correct.

The theory above allows us to reason about the correctness of simple algorithms working on simple data like integers. However, we also want to reason about algorithms that compute on structured data, like arrays, as well as about quantitative aspects of algorithms, like their execution time.

2.5 Arrays

Our concept of an array will be the same as in Java, although the notation will be more user-friendly. When reasoning about arrays, we'll write $|A|$ for the length of the array A, and $A[i]$ for its i'th entry, $0 \leq i < |A|$. Further, we'll use the notation $A[i..j]$ for the subarray of A with elements $A[i], \ldots, A[j-1]$. In using this notation we'll always have $i \leq j$, and if $i = j$, $A[i..j]$ is the empty array. We'll often need to say that something holds about all elements of an array or subarray—eg. that $A[i], \ldots, A[j-1]$ are all less than 35—and we'll write this as $A[i..j] < 35$. We'll write AB for the concatenation of A and B.

We add arrays to our pseudo-code as follows:

Indexing If e is an integer expression whose value is a valid index into an array A, we treat $A[e]$ as a variable in itself, so that we may assign to it.

Length We'll use $|A|$ as an integer expression for the length of the array A.

Allocation If e is an integer expression, the command

$$A \leftarrow \mathsf{allocate}\ e$$

executes in one step and leads from a state σ to a state σ' such that $\sigma'(A)$ is a ***reference*** to a newly allocated array of length $\sigma(e)$.

Actually, we'll never use the allocation command explicitly in our algorithms, since we'll always want to initialize the array. Rather, we'll use these abbreviations:

Construct If e_1, e_2 are expressions with e_1 of integer type, we'll write

$$A \leftarrow [e_1 : e_2] \qquad \text{for} \qquad \begin{array}{l} A \leftarrow \mathsf{allocate}\ e_1;\ i \leftarrow 0; \\ \textbf{while}\ i \neq |A|\ \textbf{do} \\ \qquad A[i] \leftarrow e_2;\ i \leftarrow i+1 \end{array}$$

So $A \leftarrow [e_1 : e_2]$ executes in $3e_1 + 3$ steps and constructs an array with e_1 entries, all initialized to e_2.

Copy If e_1, e_2 are integer expressions and B is an array variable, we'll write

$$A \leftarrow B[e_1..e_2] \qquad \text{for} \qquad \begin{array}{l} A \leftarrow \mathsf{allocate}\ e_2 - e_1;\ i \leftarrow 0; \\ \textbf{while}\ i \neq |A|\ \textbf{do} \\ \qquad A[i] \leftarrow B[e_1 + i];\ i \leftarrow i+1 \end{array}$$

So $A \leftarrow B[e_1..e_2]$ executes in $3(e_2 - e_1) + 3$ steps and makes a copy of the subarray $B[e_1..e_2]$.

Example 2.5.1 As a simple example, here is an algorithm for computing the maximum, written $\max A$, of an array A of numbers. If A is empty, we define $\max A = -\infty$:

```
Algorithm ArrayMax(A)
Input     : true
Constants: A
Output    : r = max A
Method    : r ← −∞; i ← 0;
            {I}while i ≠ |A| do
                if r < A[i] then r ← A[i];
                i ← i + 1
```

Using the invariant $I : (0 \leq i \leq |A|) \wedge (r = \max A[0..i])$ and the termination function $\mu(A, i, r) = |A| - i$, it is not hard to show that the algorithm is correct. □

2.6 Complexity

In the preceding sections we have been concerned with a *qualitative* aspect of algorithms, namely correctness. Of course, algorithms also have *quantitative* aspects, and among these the questions of how long it takes to execute them (***time complexity***) and how much storage they use (***space complexity***) are the most important ones. In these notes, we'll only discuss time complexity formally. Consider the algorithm

Algorithm $\mathsf{A}(x_1,\ldots,x_n)$	
Input	: *In*
Output	: *Out*
Method	: C

Definition 2.6.1 The ***time complexity*** of A is the function $T[\![\mathsf{A}]\!]$ taking a state σ satisfying *In* to the length of the process starting at $[C,\sigma]$.[1] □

Notice that this definition says that the evaluation of any expression takes just one "time unit". This is reasonable because all our expressions are simple: they can be implemented on a real machine so that their evaluation takes constant time. Also notice that $T[\![\mathsf{A}]\!](\sigma)$ will only depend on the values in σ of the input parameters $x_1,\ldots,x_n$ of A, and so we can consider $T[\![\mathsf{A}]\!]$ as a function of $x_1,\ldots,x_n$.

Example 2.6.2 Consider the algorithm ArrayMax of Example 2.5.1. The initialization of r and i takes 2 steps, and each of the $|A|$ iterations of the loop takes 3 or 4 steps, depending on whether r is updated or not. Further, the loop test is evaluated an extra time (when $i = |A|$). So the total number of steps will be between

	$3\|A\|+3$	if A is empty or all elements of A equal $-\infty$, so that r is never updated;
and	$4\|A\|+3$	if A is sorted in increasing order, so that r is updated in every iteration.

We therefore have $3|A|+3 \le T[\![\mathsf{ArrayMax}]\!](A) \le 4|A|+3$. More precisely, $T[\![\mathsf{ArrayMax}]\!](A)$ equals $3|A|+3$ plus the number of times $A[i]$ is strictly larger than $\max A[0..i]$, with i running through the indices $0,\ldots,|A|-1$. □

[1]For the algorithms we have seen so far, only one process starts at $[C,\sigma]$ for each σ. Later we'll allow our algorithms to make random choices and then each possible choice gives rise to a different process. We then define $T[\![\mathsf{A}]\!](\sigma)$ as the maximum of their lengths and $E[\![\mathsf{A}]\!](\sigma)$ as the expected length, taken over all random choices.

Although $T[\![\mathsf{A}]\!]$ is a well-defined mathematical function, it can sometimes be extremely hard to find a nice expression for it. For example, the time complexity of Euclid is quite erratic judging from the table below:

m	n	T	m	n	T	m	n	T	m	n	T	m	n	T	$\cdots$
1	1	2	2	1	5	3	1	8	4	1	11	5	1	14	$\cdots$
1	2	5	2	2	2	3	2	8	4	2	5	5	2	11	$\cdots$
1	3	8	2	3	8	3	3	2	4	3	11	5	3	11	$\cdots$
1	4	11	2	4	5	3	4	11	4	4	2	5	4	14	$\cdots$
1	5	14	2	5	11	3	5	11	4	5	14	5	5	2	$\cdots$
1	6	17	2	6	8	3	6	5	4	6	8	5	6	17	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	

Although we may be able to find a succinct way of expressing $T[\![\mathsf{Euclid}]\!]$, algorithms exist that are complicated enough that such a table is the best we can do. Fortunately, $T[\![\mathsf{A}]\!]$ often provides more information than needed in practice anyway, and so approximating it will be sufficient.

The crucial step here is to view $T[\![\mathsf{A}]\!]$ as a function of the *size of the input*, rather than as a function of the input itself. A reasonable notion of input size for ArrayMax would be the length of the array A. Of course, there are many possible inputs with the same size n, namely all arrays of length n, and $T[\![\mathsf{ArrayMax}]\!]$ will now assign the same execution time to all of them. Recall that *in the worst case*, when A is sorted in increasing order, ArrayMax uses $4|A| + 3$ steps. We want to be as pessimistic as possible and so we take $T[\![\mathsf{ArrayMax}]\!]$ to be the function mapping $n \geq 0$ to $4n + 3$. The reason for this pessimism is a pragmatic one: we aim for algorithms that perform well on *all* possible inputs. This justifies the following definition:

Definition 2.6.3 Let *size* be a function mapping states satisfying *In* to non-negative integers. The ***worst-case time complexity*** of A is the function mapping $n \geq 0$ to the maximum of $T[\![\mathsf{A}]\!](\sigma)$ for states σ (satisfying *In*) with $size(\sigma) = n$.[2] □

How we define *size* will depend on the algorithm and will be made clear in each case, but again, we may view it as a function of the input parameters of the algorithm. For ArrayMax above, we have $size(A) = |A|$ while for PowerSum of Section 2.4.3, we may reasonably take $size(x, n) = n$. Accordingly, we can write $T[\![\mathsf{ArrayMax}]\!](A) = 4|A| + 3$ and $T[\![\mathsf{PowerSum}]\!](n) = 3n + 3$ for the worst-case time complexities. In fact, we'll use $\mathcal{O}$-notation to suppress the constants involved, and simply say that ArrayMax runs in $\mathcal{O}(|A|)$ time, while PowerSum runs in $\mathcal{O}(n)$ time.

[2] The ***worst-case expected time complexity*** of A is obtained by replacing $T[\![\mathsf{A}]\!]$ by $E[\![\mathsf{A}]\!]$.

Chapter 3

Fundamental algorithms

The formal techniques of the previous chapter can not only be used to prove existing algorithms correct; more importantly, they provide the basis of a programming methodology. In particular, to solve an algorithmic problem by iteration, one comes up with an invariant and builds an algorithm around it. In this chapter, we'll apply this programming methodology to a range of fundamental algorithmic problems.

3.1 Exponentiation

As an easy start in the application of our methodology, let us use it to improve an existing algorithm. Consider the following way of computing x^p for a number x and a natural number p:

Algorithm LinExp(x, p)
Input : $p \geq 0$
Constants: x, p
Output : $r = x^p$
Method : $r \leftarrow 1$; $q \leftarrow p$;
$\{I\}$**while** $q > 0$ **do**
$r \leftarrow r * x$; $q \leftarrow q - 1$

With the invariant $I : (rx^q = x^p) \wedge (q \geq 0)$ and the termination function $\mu(x, p, r, q) = q$ a correctness proof is easy. The execution time is linear in p, hence the name. Intuitively, we may improve LinExp by making sure that q decreases faster; for example, we could try to obtain logarithmic running time by repeatedly dividing q by two instead of subtracting 1.

Consider the first part of the invariant above, $rx^q = x^p$. If this equation is true, and q is even, then we can divide q by two and square the x of

x^q, and the resulting equation, $r(x^2)^{q/2} = x^p$ will also be true. Since x is specified to be constant we introduce a new variable h which we may modify as needed. The invariant should now look as follows:

$$I : (rh^q = x^p) \wedge (q \geq 0).$$

To establish it we must start by initializing h to x. This gives us a new initialization-sequence, which we'll call $\ll$ init $\gg$:

$$\ll \text{init} \gg = r \leftarrow 1;\ q \leftarrow p;\ h \leftarrow x.$$

The body of the loop must update r and h, and so we'll call it $\ll$ update $\gg$. If q is even, we can divide q by two and square h without spoiling the invariant, but otherwise we need to do something else. Therefore, let's make $\ll$ update $\gg$ a selection:

$$\ll \text{update} \gg = \textbf{if } q \text{ even } \textbf{then} \ll \text{q even} \gg \textbf{else} \ll \text{q odd} \gg,$$

—where the **then**-branch is

$$\ll \text{q even} \gg = q \leftarrow q/2;\ h \leftarrow h * h.$$

We may now prove $\{I \wedge (q > 0) \wedge (q \text{ even})\} \ll \text{q even} \gg \{I\}$. An obvious solution for the other branch is to subtract 1 from q because then q will be even in at least every other iteration (which is good for the execution time). The simplest way to update the other variables is to leave h unchanged and multiply r by h:

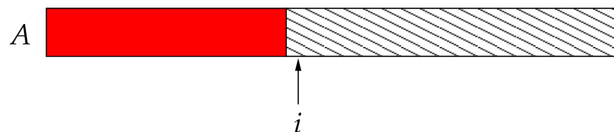
$$\ll \text{q odd} \gg = q \leftarrow q - 1;\ r \leftarrow r * h.$$

Then $\{I \wedge (q > 0) \wedge (q \text{ odd})\} \ll \text{p odd} \gg \{I\}$ can be shown, and so can clearly also the last proof-burden, $I \wedge (q \leq 0) \Rightarrow r = x^p$. The algorithm is thus valid. Since q is decreased in every iteration and since I says that q is non-negative, we can use $\mu(x, p, r, q, h) = q$ as a termination function. So the algorithm below is correct and runs in time $\mathcal{O}(\log p)$.

Algorithm LogExp(x, p)
Input : $p \geq 0$
Constants: x, p
Output : $r = x^p$
Method : $r \leftarrow 1;\ q \leftarrow p;\ h \leftarrow x;$
$\{I\}$**while** $q > 0$ **do**
if q even **then**
$q \leftarrow q/2;\ h \leftarrow h * h$
else
$q \leftarrow q - 1;\ r \leftarrow r * h$

3.2 Scanning

A standard way to do computations on arrays—say finding a maximal element in an array A of numbers—is to write a loop that scans through the array from left to right using an index i. An invariant for such a loop can be pictured as follows:



The red part of the array, $A[0..i]$, represents the elements we have already scanned, and so know some information about—say their maximum—while the shaded part contains elements that we have not yet looked at. Formally, the invariant looks like

$$I : (0 \leq i \leq |A|) \wedge I'$$

—where I' says what information we have about the red part, which for the computation of maximum would be, say, $r = \max A[0..i]$. We can write a general outline of the algorithm's method as follows:

$\ll$ init $\gg$; $i \leftarrow 0$;
$\{I\}$**while** $i \neq |A|$ **do**
  $\ll$ update $\gg$; $i \leftarrow i + 1$;
$\ll$ end $\gg$

The three unspecified command-sequences are specific to the computation, but notice that if $\ll$ update $\gg$ does not change i or the length of A, then we already have two elements of a correctness proof: the first part of I is a valid invariant and $\mu(A, i, \ldots) = |A| - i$ is a termination function.

Of course, sometimes there's no need to scan the whole array, and one can then use a loop like

while $(i \neq |A|) \wedge b$ **do** $\cdots$

—where b is a boolean expression which is only true as long as we have to keep scanning. The LinearSearch algorithm below is of this kind.

Scanning algorithms are often simple and should be your first attempt at a solution when faced with an algorithmic problem on arrays.

3.3 Searching

Searching for an element s in an array A is a truly fundamental computation that in one form or other happens almost every time we need to retrieve data stored in a computer. Still, it is not obvious what output a search algorithm should give. For some applications it suffices to know whether s occurs in A or not. In others, one needs an index of s if it occurs, and some other (non-index) value if it doesn't. Further, if there are multiple occurrences of s, one might want the first or the last or all of them. To accommodate for these different needs, we'll specify our algorithms below such that they provide the basic computation needed in any case.

3.3.1 Linear search

Our first search algorithm finds the smallest index r such that $A[r] = s$. If no such index exists, it returns $r = |A|$. A simple scanning solution is immediate; it uses r as the scanning index and scans only as long as $A[r] \neq s$:

Algorithm LinearSearch(A, s)
Input : true
Constants: A, s
Output : $(0 \leq r \leq |A|) \wedge (s \notin A[0..r]) \wedge (r = |A| \vee A[r] = s)$
Method : $r \leftarrow 0$;
$\{I\}$**while** $(r \neq |A|) \wedge (A[r] \neq s)$ **do**
$r \leftarrow r + 1$;

The invariant

$$I : (0 \leq r \leq |A|) \wedge (s \notin A[0..r])$$

is valid and we can use the termination function $\mu(A, s, r) = |A| - r$ to conclude that the algorithm is correct. It clearly runs in time $\mathcal{O}(|A|)$.

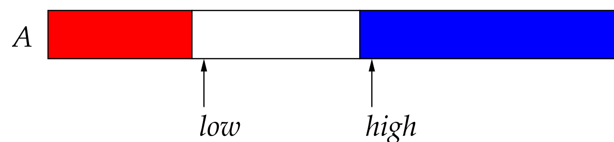
3.3.2 Binary search

The linear search algorithm is slow if a long array is to be searched many times, and in such cases, it is more efficient to sort the array first. This is exactly why telephone books are sorted. Sorting makes sense for any kind of value, including numbers and strings, for which an ordering $\leq$ can be defined. In the following, we describe an efficient algorithm that searches

for an element s in a sorted array A of integers. Since A is sorted, it makes sense to find the first index r where s must appear if it occurs in A at all:

Algorithm BinarySearch(A, s)
Input : A sorted
Constants: A, s
Output : $(0 \le r \le |A|) \wedge (A[0..r] < s) \wedge (s \le A[r..|A|])$

Since we want to do better than linear search, we cannot use scanning. Then what? It seems obvious that we need to compare s with elements of A. Doing so provides knowledge about A of the form



The intended meaning of this picture is that the red elements are all $< s$ and the blue elements are all $\geq s$, whereas we don't know how the white elements relate to s.

If we narrow the gap between *low* and *high* until they are equal, we get the wanted result by putting $r = low = high$. Therefore, we can use the above picture as an invariant and write

$low \leftarrow 0$; $high \leftarrow |A|$;
$\{I\}$**while** $low \neq high$ **do**
  $\ll$ narrow the gap $\gg$;
$r \leftarrow low$

Formally, the invariant is

$$I : (0 \le low \le high \le |A|) \wedge (A[0..low] < s) \wedge (s \le A[high..|A|]).$$

Now, consider an index m between *low* and *high*.

- If $A[m] < s$ we can extend the red part by setting *low* to $m + 1$.
- If $s \le A[m]$ we can extend the blue part by setting *high* to m.

So clearly, choosing m in the middle of the range, we can approximately halve the distance between *low* and *high*. Doing so in each iteration will quickly make *low* equal *high*:

Algorithm BinarySearch(A, s)
Input : A sorted
Constants: A, s
Output : $(0 \leq r \leq |A|) \wedge (A[0..r] < s) \wedge (s \leq A[r..|A|])$
Method : $low \leftarrow 0$; $high \leftarrow |A|$;
$\{I\}$**while** $low \neq high$ **do**
$m \leftarrow (low + high)/2$;
$\{U\}$**if** $A[m] < s$ **then**
$low \leftarrow m + 1$
else
$high \leftarrow m$;
$r \leftarrow low$

With

$$U : I \wedge (low \leq m < high)$$

it is quite easy to prove the proof-burdens

$$\{I \wedge (low \neq high)\} m \leftarrow (low + high)/2 \{U\}$$
$$\{U \wedge (A[m] < s)\} low \leftarrow m + 1 \{I\}$$
$$\{U \wedge (s \leq A[m])\} high \leftarrow m \{I\}$$

establishing the invariant. As for termination, we can use the termination function

$$\mu(A, s, low, high, m, r) = high - low.$$

It is non-negative because by the invariant $low \leq high$ and it decreases in each iteration because either *low* increases or *high* decreases. In fact, the value $high - low + 1$ is at least halved in each iteration, and so the running time is $\mathcal{O}(\log |A|)$.

3.4 Sorting

BinarySearch clearly demonstrates that sorting is an important problem, and so we'll develop three different sorting algorithms in the following. We start with a simple scanning algorithm, called InsertionSort, whose running time is quadratic in the length n of the input array. Next, by using a common problem solving technique, we obtain the MergeSort algorithm with optimal running time $n \log n$. Although this second algorithm is optimal, there is room for improvement in the constant factors involved and our third sorting algorithm, QuickSort, achieves greater speed on the average by using random choices.

3.4.1 Insertion sort

A scanning algorithm for sorting would have an invariant like this:

$$I : (A \text{ perm } A_0) \wedge (0 \leq i \leq |A|) \wedge (A[0..i] \text{ sorted})$$

where A perm A_0 means that A is a ***permutation*** of its original elements; it holds the same elements, but possibly in a different order.

In each iteration we need to perform some actions that allow us to increment i without I becoming incorrect. A simple way to achieve this is by repeatedly swapping the element at $A[i]$ with its predecessor until $A[0..i+1]$ is in sorted order. This calls for a new loop with the invariant

$$J : (A \text{ perm } A_0) \wedge (0 \leq j \leq i < |A|) \wedge (A[0..j]A[j+1..i+1] \text{ sorted})$$
$$\wedge (A[j..i+1] \text{ sorted})$$

Here, j is the current index of the element originally at $A[i]$. Clearly we are done when $j = 0$ or $A[j-1] \leq A[j]$, because then the second half of J says that $A[0..i+1]$ is sorted. Here's the full algorithm:

```
Algorithm InsertionSort(A)
Input     : true
Output    : (A perm A0) ∧ (A sorted)
Method    : i ← 0;
            {I}while i ≠ |A| do
                j ← i;
                {J}while (j ≠ 0) ∧ (A[j − 1] > A[j]) do
                    A[j − 1] ↔ A[j];
                    j ← j − 1;
                i ← i + 1
```

We can use $\mu_I(A, i, j) = |A| - i$ as termination function for the outer loop and $\mu_J(A, i, j) = j$ for the inner loop. The running time of the algorithm is $\mathcal{O}(|A|^2)$ in the worst case which happens when A is sorted in backwards order to begin with. In that case, the inner loop is executed first $|A| - 1$ times, then $|A| - 2$ times and so on. This gives the sum

$$(|A| - 1) + (|A| - 2) + \cdots + 1 = \frac{|A|(|A| - 1)}{2}.$$

Notice, on the other hand, that the element originally at index i is only swapped with those elements at indices $j < i$ that are strictly larger than it. Such pairs of indices (j, i) are called ***inversions*** and the total number

of inversions in A gives a measure of how far from being sorted it is. If there are k inversions in A, then InsertionSort takes $\mathcal{O}(|A| + k)$ time, which means that this algorithm might be a good choice if you know that your array is almost sorted.

3.4.2 Merge sort

To improve on the quadratic worst case running time of the insertion sort algorithm, we employ a powerful problem-solving technique, called ***divide-and-combine***. Using this technique, one solves a big problem by first *dividing* it into smaller problems, then *solving* those smaller problems, and finally *combining* the results into a solution to the original problem. Obviously, there has to be some way of solving small problems directly, but by using recursion, we can keep on dividing problems into subproblems until they are trivial to solve.

Applied to the problem of sorting an array, it is at least clear how to handle small problems: arrays of length ≤ 1 are already sorted. As for big problems—longer arrays—one can divide them into, say, two halves and then sort those recursively. Combining two sorted arrays into one is called ***merging***, hence the name of the algorithm:

Algorithm MergeSort(A)
Input : true
Output : $(A \text{ perm } A_0) \wedge (A \text{ sorted})$
Method : **if** $|A| > 1$ **then**
$B \leftarrow A[0..|A|/2]$;
$C \leftarrow A[|A|/2..|A|]$;
$\{U\}$MergeSort(B);
$\{V\}$MergeSort(C);
$\{W\}$Merge(A, B, C)

The algorithm Merge is specified by

Algorithm Merge(A, B, C)
Input : $(|A| = |B| + |C|) \wedge (B, C \text{ sorted})$
Constants: B, C
Output : $(A \text{ perm } BC) \wedge (A \text{ sorted})$

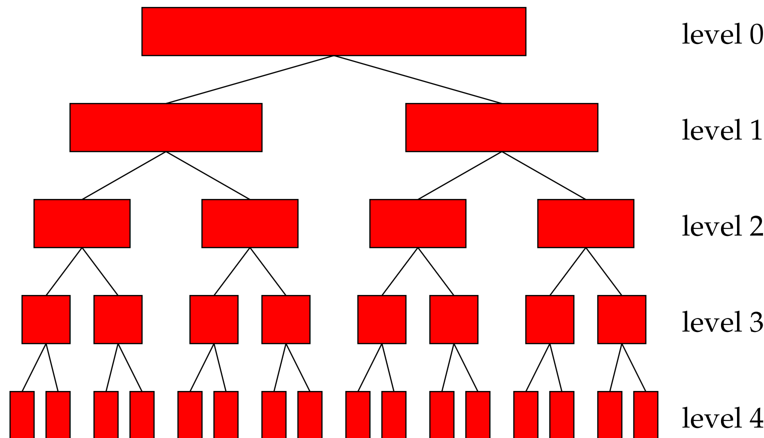
We develop this algorithm below, but let's first look at the correctness and running time of MergeSort. Write A_1, A_2 for $A[0..|A|/2]$ and $A[|A|/2..|A|]$,

respectively. Validity of the algorithm follows from validity of assertions

$$\begin{array}{ll} U: & (A = A_0) \wedge (|A| > 1) \wedge (B = A_1) \wedge (C = A_2) \\ V: & (A = A_0) \wedge (|A| > 1) \wedge (B \text{ perm } A_1) \wedge (B \text{ sorted}) \wedge (C = A_2) \\ W: & (A = A_0) \wedge (B \text{ perm } A_1) \wedge (C \text{ perm } A_2) \wedge (B, C \text{ sorted}) \end{array}$$

Since we use recursion we had better find a termination function: with $|A| > 1$ both $|A_1|$ and $|A_2|$ are strictly smaller than $|A|$ and so $\mu(A) = |A|$ can be used. The algorithm is correct.

As we shall see, Merge(A, B, C) takes time linear in $|A|$, and so we can characterize the running time of MergeSort as follows: Suppose for simplicity that $|A| = n$ is a power of 2. Then the call-tree will have height $\log n$ and there will be a total of 2^i subarrays of length $n/2^i$ at level i, for $0 \le i \le \log n$, as shown in the figure below for $n = 16$:



Now, for some reasonable time unit, we may say that dividing an array of length n into two subarrays each of length $n/2$ and to merge two such subarrays once they are sorted takes a total of n time units. It is then easy to see that the total time spent at level i of the call-tree is $2^i \cdot n/2^i = n$ time units. The total time is therefore $n \log n$ time units which means that MergeSort takes $\mathcal{O}(n \log n)$ time on an array of length n.

Merge

We want an algorithm

Algorithm Merge(A, B, C)
Input : $(\|A\| = \|B\| + \|C\|) \wedge (B, C \text{ sorted})$
Constants: B, C
Output : $(A \text{ perm } BC) \wedge (A \text{ sorted})$

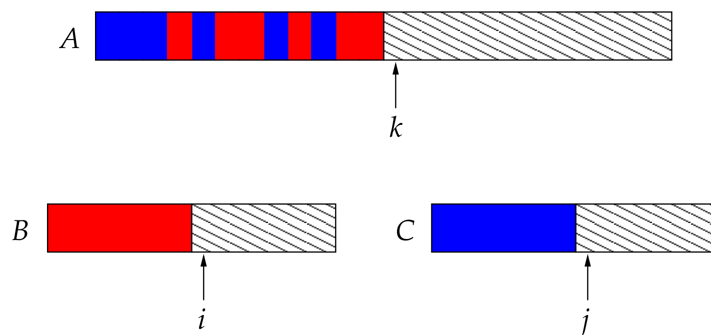
so that given, say, A of length 11 and

$$B = [1, 1, 4, 7, 8] \qquad \text{and} \qquad C = [1, 2, 2, 3, 4, 7]$$

we should obtain

$$A = [1, 1, 1, 2, 2, 3, 4, 4, 7, 7, 8].$$

The algorithm must fill the array A with the right elements. A first guess (as always) is to scan through A from left to right, maintaining the invariant that we are finished with the part of A that we have already scanned. Clearly, this will also involve scanning the arrays B and C from left to right. So we have the following picture of the invariant:



From the picture it seems that we need three indices i, j and k, but notice that k is redundant since $k = i + j$. So formally, the invariant is

$$\begin{aligned} I: \quad & (|A| = |B| + |C|) \wedge (0 \leq i \leq |B|) \wedge (0 \leq j \leq |C|) \wedge \\ & (A[0..i+j] \text{ perm } B[0..i]C[0..j]) \wedge (A[0..i+j] \text{ sorted}) \end{aligned}$$

We can now write the usual outline of the algorithm:

Algorithm Merge(A, B, C)
Input : $(|A| = |B| + |C|) \wedge (B, C \text{ sorted})$
Constants: B, C
Output : $(A \text{ perm } BC) \wedge (A \text{ sorted})$
Method : $\ll$ init $\gg$;
$\{I\}$**while** $i + j \neq |A|$ **do**
$\ll$ update $\gg$;
$\ll$ end $\gg$

According to the invariant, the initialization should create a picture with all three arrays shaded and so we put

$$\ll \text{init} \gg = i \leftarrow 0;\ j \leftarrow 0.$$

Now the proof-burden $\{(|A| = |B| + |C|) \wedge (B, C \text{ sorted})\} \ll \mathsf{init} \gg \{I\}$ can be proved.

In each iteration of the loop, we would like to fill in one more element of A. It should be either $B[i]$ or $C[j]$ but which one? The requirements that it may be $B[i]$ are as follows: First, of course, $B[i]$ must be defined, that is we must have $i < |B|$. Second, $B[i]$ must be $\leq$ any element of $C[j..|C|]$. This trivially happens if $j = |C|$, and otherwise it happens if $B[i] \leq C[j]$ since C is sorted. In all other cases, we must use $C[j]$. So we can take $\ll \mathsf{update} \gg$ to be the selection

if $(i < |B|) \wedge (j = |C| \vee B[i] \leq C[j])$ **then**
  $\ll \mathsf{use\ B[i]} \gg$
else
  $\ll \mathsf{use\ C[j]} \gg$

Knowing that we can use $B[i]$, it is quite easy to restore the invariant. According to the picture, we should increment i and leave j as it is. With

$$\ll \mathsf{use\ B[i]} \gg = A[i+j] \leftarrow B[i];\ i \leftarrow i+1$$

we can prove the proof-burden

$\{I \wedge (i + j \neq |A|) \wedge (i < |B|) \wedge (j = |C| \vee B[i] \leq C[j])\}$
$\ll \mathsf{use\ B[i]} \gg$
$\{I\}$.

The command-sequence $\ll \mathsf{use\ C[j]} \gg$ is symmetric. Notice that the loop must terminate because the value $i + j$ increases in every iteration and cannot exceed $|A|$. In other words, $\mu(A, B, C, i, j) = |A| - (i + j)$ is a termination function.

When the loop finishes, we have $i + j = |A|$ by the loop condition and so by the invariant, $i = |B|$ and $j = |C|$. It immediately follows from the invariant that A is the merge of B and C as wanted. So $\ll \mathsf{end} \gg$ may be empty. Here is the complete algorithm:

Algorithm Merge(A, B, C)
Input : $(|A| = |B| + |C|) \wedge (B, C \text{ sorted})$
Constants: B, C
Output : $(A \text{ perm } BC) \wedge (A \text{ sorted})$
Method : $i \leftarrow 0;\ j \leftarrow 0;$
  $\{I\}$**while** $i + j \neq |A|$ **do**
    if $(i < |B|) \wedge (j = |C| \vee B[i] \leq C[j])$ **then**
      $A[i+j] \leftarrow B[i];\ i \leftarrow i+1$
    else
      $A[i+j] \leftarrow C[j];\ j \leftarrow j+1$

Clearly, the running time of this algorithm is proportional to the number of iterations of the loop. Hence, the algorithm runs in time $\mathcal{O}(|A|)$.

3.4.3 Quick sort

One can prove that MergeSort is as fast as any comparison-based sorting algorithm, if we ignore constant factors. However, because sorting is such an important problem, it makes sense to try to improve the constants involved. To this end, notice that MergeSort spends a lot of time copying arrays to auxiliary storage (the arrays B and C). Avoiding that may give a faster algorithm.

In this section, we describe the algorithm QuickSort which performs the sorting ***in-place***, that is, it sorts the input array A just by swapping elements. The central idea is to choose an element $s = A[r]$ of A and then swap the elements of A around to obtain a picture like this:

The red elements are all strictly smaller than s, the white elements equal s, and the blue elements are strictly larger than s. Having done so, we may sort the red and blue parts by applying QuickSort recursively to these subarrays, after which the whole array is sorted. Notice that this means that QuickSort must be parameterized with two indices, saying where to start and end the sorting, and of course, it should not alter the array outside this range. Formally,

Algorithm QuickSort$(A, low, high)$
Input : $0 \leq low \leq high \leq |A|$
Constants: $low, high, A[0..low], A[high..|A|]$
Output : $(A[low..high] \text{ perm } A_0[low..high]) \wedge (A[low..high] \text{ sorted})$

For simplicity we shall delegate the rearrangement of $A[low..high]$ to an auxiliary algorithm, called the "Dutch flag" algorithm because of the picture above—it looks like a Dutch flag rotated 90 degrees. In addition to rearranging A in this range, DutchFlag must give us the boundaries of the red and blue parts. The formal specification is as follows:

Algorithm DutchFlag$(A, low, high, s)$
Input : $0 \leq low \leq high \leq |A|$
Constants: $low, high, A[0..low], A[high..|A|], s$
Output : $(A[low..high] \text{ perm } A_0[low..high]) \wedge (low \leq w \leq b \leq high) \wedge$
$(A[low..w] < s) \wedge (A[w..b] = s) \wedge (A[b..high] > s)$

The red part is then given by $A[low..w]$ and the blue part by $A[b..high]$. We shall later develop DutchFlag as a scanning algorithm with linear running time. At this point, the method of QuickSort looks as follows:

if $high - low > 1$ **then**
  $\ll$ choose r $\gg$;
  $(w, b) \leftarrow$ DutchFlag$(A, low, high, A[r])$;
  QuickSort(A, low, w);
  QuickSort$(A, b, high)$

Notice that in the case of ranges containing less than 2 elements, the algorithm does nothing, thus providing a bottom for the recursion. We still have to say how to choose the index r of s. For the correctness of the algorithm it doesn't matter what r is as long as it belongs to the range $low..high$ so that the white part becomes non-empty. This restriction makes $\mu(A, low, high) = high - low$ a termination function.

So one possibility would be to take $r = low$. But what about the running time? At each level of the call-tree for QuickSort we spend $\mathcal{O}(|A|)$ time doing DutchFlag on all the red and blue parts. Because all other computations take constant time at each level, the total running time of QuickSort is therefore $\mathcal{O}(|A| \cdot \text{height of call-tree})$.

Now, suppose that A is already sorted and has all elements distinct. Then during the execution all red parts will be empty (because with $r = low$ the smallest element in the range is $s = A[r]$), all white parts will contain a single element, s, and the blue parts will occupy the rest. So there will be approximately $|A|$ levels in the call-tree, corresponding to blue parts of length $|A| - 1, |A| - 2, \ldots$, giving a quadratic time behavior.

A similar argument can be given for any other *fixed* way of choosing r: we cannot with a fixed choice of r guarantee that neither the red nor the blue part becomes very long—and so we cannot guarantee a shallow call-tree. To obtain a better behavior, we therefore choose r *uniformly at random* in the legal range:

$\ll$ choose r $\gg$ $= r \leftarrow$ random$(low, high)$

Let $n = high - low$ and call a choice of r *good* if it results in both the red and blue parts having size at least $\frac{n}{4}$ (and so at most $\frac{3n}{4}$). Since we can only remove one quarter of n (that is, divide n by $\frac{4}{3}$) approximately $\log_{\frac{4}{3}} n$ times before the result is ≤ 1, the recursion continues only until we have seen $\log_{\frac{4}{3}} n$ good choices of r. The question then is: how long should we expect to wait for a good choice?

Assuming that all elements in A are distinct, there is a probability of $\frac{1}{n}$ for each of the possible ways that our choice of r divides the range. Since r is good for half of the possibilities, r is good with probability $\frac{1}{2}$. But then probability theory tells us that the expected number of divisions performed before r is good is 2.

In conclusion, the expected height of the call-tree for QuickSort is at most $2\log_{\frac{4}{3}}|A|$ and so the expected running time of QuickSort on an array of length n is $\mathcal{O}(n \log n)$. Here is the completed algorithm:

Algorithm QuickSort$(A, low, high)$
Input : $0 \leq low \leq high \leq |A|$
Constants: $low, high, A[0..low], A[high..|A|]$
Output : $(A[low..high]$ perm $A_0[low..high]) \wedge (A[low..high]$ sorted$)$
Method : **if** $high - low > 1$ **then**
    $r \leftarrow$ random$(low, high)$;
    $(w, b) \leftarrow$ DutchFlag$(A, low, high, A[r])$;
    QuickSort(A, low, w);
    QuickSort$(A, b, high)$

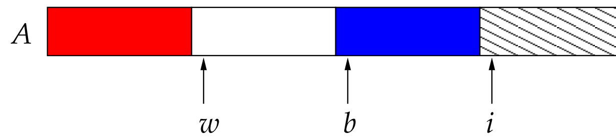
Timing experiments on concrete implementations will show that although QuickSort has worst-case quadratic running time, it is fast enough on the average to outperform MergeSort in many applications. Beware though, that if A is too large to fit in internal storage, then MergeSort is the better choice. The somewhat erratic indexing done by DutchFlag incurs far too many accesses to external storage (eg. a hard disk) and such operations are extremely slow compared to processor speed.

Dutch flag

The Dutch flag algorithm is a classic application of our methodology and we'll therefore give a rather detailed development of it. To keep notation manageable, we shall simplify the specification above by requiring the algorithm to rearrange the whole array rather than a particular range:

Algorithm DutchFlag(A, s)
Input : true
Constants: s
Output : $(A$ perm $A_0) \wedge (0 \leq w \leq b \leq |A|) \wedge$
$(A[0..w] < s) \wedge (A[w..b] = s) \wedge (A[b..|A|] > s)$

As usual, our first guess would be a scanning algorithm. The invariant should then look like this:



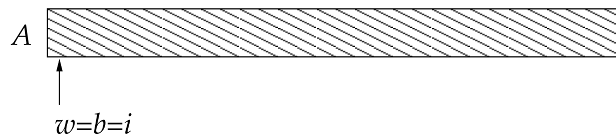
Formally, it can be expressed as

$$I: \quad (A \text{ perm } A_0) \wedge (0 \le w \le b \le i \le |A|) \wedge \\ (A[0..w] < s) \wedge (A[w..b] = s) \wedge (A[b..i] > s)$$

Our problem is now to write the sequences $\ll$ init $\gg$, $\ll$ update $\gg$, and $\ll$ end $\gg$ so that the command-sequence

$\ll$ init $\gg$; $i \leftarrow 0$;
$\{I\}$**while** $i \ne |A|$ **do**
    $\ll$ update $\gg$; $i \leftarrow i+1$;
$\ll$ end $\gg$

makes I valid and the algorithm correct. The sequence $\ll$ init $\gg$ must initialize the variables w and b. From the outset we have not looked at any element of A and so the picture of the invariant looks like this:



It is thus easy to see that if we put $\ll$ init $\gg$ $= w \leftarrow 0;\ b \leftarrow 0$ we can prove the proof-burden $\{\text{true}\}\ll$ init $\gg$; $i \leftarrow 0\{I\}$.

Next, we must write the sequence $\ll$ update $\gg$ so that we can prove the proof-burden $\{I \wedge i \ne |A|\}\ll$ update $\gg$; $i \leftarrow i+1\{I\}$. Here it is intuitively clear that we need to consider three cases, depending on how $A[i]$ relates to s. So we let $\ll$ update $\gg$ be the selection

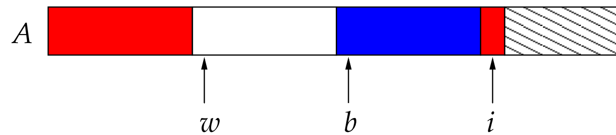
if $A[i] < s$ **then**
    $\ll$ red $\gg$
else if $A[i] = s$ **then**
    $\ll$ white $\gg$
else
    $\ll$ blue $\gg$

The proof-burden for $\ll$ update $\gg$ now splits into

$$1: \quad \{I \wedge (i \ne |A|) \wedge (A[i] < s)\}\ll \text{red} \gg;\ i \leftarrow i+1\{I\}$$
$$2: \quad \{I \wedge (i \ne |A|) \wedge (A[i] = s)\}\ll \text{white} \gg;\ i \leftarrow i+1\{I\}$$
$$3: \quad \{I \wedge (i \ne |A|) \wedge (A[i] > s)\}\ll \text{blue} \gg;\ i \leftarrow i+1\{I\}.$$

We look at each case in turn:

Case 1: In the case $A[i] < s$ we have the situation

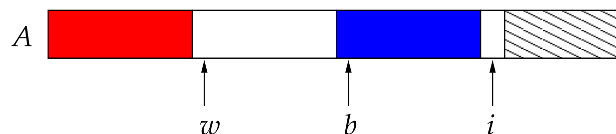


and so $\ll$ red $\gg$ first swaps $A[i]$ and $A[b]$, then swaps $A[w]$ and $A[b]$, and finally increments b and w. This way of doing it also works in the case where two or more of w, b, and i are equal:

$$\ll \text{red} \gg = \quad A[i] \leftrightarrow A[b];\ A[b] \leftrightarrow A[w];$$
$$w \leftarrow w + 1;\ b \leftarrow b + 1.$$

This enables us to prove proof-burden 1.

Case 2: In the case $A[i] = s$ we have the situation

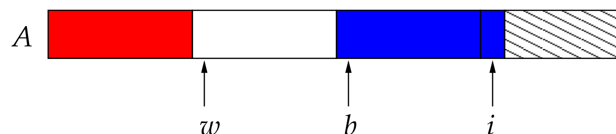


and so $\ll$ white $\gg$ must swap $A[i]$ and $A[b]$ and then increment b to extend the white block with $A[i]$:

$$\ll \text{white} \gg = A[i] \leftrightarrow A[b];\ b \leftarrow b + 1.$$

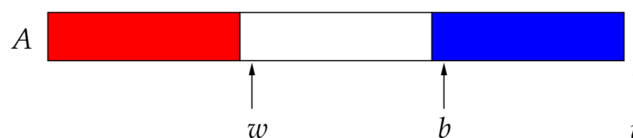
This enables us to prove proof-burden 2.

Case 3: In the case $A[i] > s$ we have the situation



and so $\ll$ blue $\gg$ doesn't need to do anything and we take $\ll$ blue $\gg = \lambda$.

When the loop terminates, we have $i = |A|$ and the picture of the invariant looks as follows:



—which is just what we want, so we can put $\ll$ end $\gg = \lambda$. Assembling the different pieces, we get the completed algorithm:

Algorithm DutchFlag(A, s)
Input : true
Constants: s
Output : $(A \text{ perm } A_0) \wedge (0 \leq w \leq b \leq |A|) \wedge$
$(A[0..w] < s) \wedge (A[w..b] = s) \wedge (A[b..|A|] > s)$
Method : $w \leftarrow 0$; $b \leftarrow 0$; $i \leftarrow 0$;
$\{I\}$**while** $i \neq |A|$ **do**
  if $A[i] < s$ **then**
    $A[i] \leftrightarrow A[b]$; $A[b] \leftrightarrow A[w]$;
    $w \leftarrow w + 1$; $b \leftarrow b + 1$
  else if $A[i] = s$ **then**
    $A[i] \leftrightarrow A[b]$; $b \leftarrow b + 1$
  $i \leftarrow i + 1$

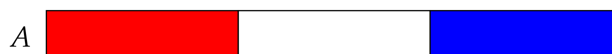
Since the initialization as well as each iteration by itself takes constant time, the time complexity of the algorithm is given by the number of iterations—which for any scanning algorithm is just the length of the array. So DutchFlag runs in time $\mathcal{O}(|A|)$ as wanted.

3.5 Selection

In some statistical applications, it is of interest to compute, say, the minimum or the median of a dataset of numbers. The algorithmic problem underlying such computations is called ***selection***: Given a non-empty array A of integers and a number k in $0..|A|$, rearrange A such that $A[0..k] \leq A[k] \leq A[k..|A|]$. The minimum is then obtained as $A[k]$ by using $k = 0$ and the median using $k = |A|/2$.

3.5.1 Quick select

Selection resembles the problem solved by the Dutch flag algorithm. But notice that A must now be rearranged with respect to some *unknown* element (with k predecessors). Still, we can use DutchFlag for this problem in very much the same way it was used in QuickSort. First, select at random an index r of A and use DutchFlag on $s = A[r]$ to obtain the situation



—with red elements $A[0..w] < s$, white elements $A[w..b] = s$, and blue elements $A[b..|A|] > s$.
There are now three possibilities depending on how k relates to w and b:

Case $k < w$**:** The element of A with k predecessors must be red because the white and blue elements all have at least w predecessors. If we recurse on the red part, we obtain $A[0..k] \leq A[k] \leq A[k..w]$ and so we're done because all other elements of A are larger.

Case $w \leq k < b$**:** The element of A with k predecessors must be white, because the red elements have strictly less than w predecessors and the blue elements have at least b predecessors. We are therefore done because $A[w..k] = A[k] = A[k..b]$ with the red elements smaller and the blue elements larger.

Case $b < k$**:** The element of A with k predecessors must be blue, because the red and white elements all have at most b predecessors. If we recurse on the blue part, we obtain $A[b..k] \leq A[k] \leq A[k..|A|]$ and so we're done because all other elements of A are smaller.

As for QuickSort we need indices *low* and *high* to bound the rearrangements and we can use $\mu(A, low, high, k) = high - low$ as a termination function:

Algorithm QuickSelect$(A, low, high, k)$
Input : $(0 \leq low \leq k < high \leq |A|)$
Constants: $low, high, A[0..low], A[high..|A|], k$
Output : $(A[low..high] \text{ perm } A_0[low..high]) \wedge$
$(A[low..k] \leq A[k] \leq A[k..high])$
Method : $r \leftarrow$ random$(low, high)$;
$(w, b) \leftarrow$ DutchFlag$(A, low, high, A[r])$;
if $k < w$ **then**
QuickSelect(A, low, w, k)
else if $k \geq b$ **then**
QuickSelect$(A, b, high, k)$

The expected running time on a range of length $n = high - low$ is $\mathcal{O}(n)$ which can be seen as follows: If we do not count the time spend on recursive calls, it takes n time units to run QuickSelect on an array of length n for some suitable time unit. For $n = 1$ there will be no recursive calls and so the running time is just 1 time unit. For larger arrays the running time depends on r. Let's call the choice of r *good* if it results in both the red and blue parts having size at most $\frac{3n}{4}$. Write $g(n)$ for the random function

giving the number of recursive calls needed before r is good. The running time $T(n)$ for QuickSelect is then bounded by the recurrence

$$T(n) \leq \begin{cases} 1 & \text{if } n = 1 \\ n \cdot g(n) + T(\frac{3n}{4}) & \text{if } n > 1 \end{cases}$$

We're interested in the expected running time $E(T(n))$. Assuming the elements of A are distinct, r is good with probability $\frac{1}{2}$ and so $E(g(n)) = 2$. By linearity of the expectation, we therefore get

$$E(T(n)) \leq \begin{cases} 1 & \text{if } n = 1 \\ 2n + E(T(\frac{3n}{4})) & \text{if } n > 1 \end{cases}$$

We can prove by induction that $E(T(n)) \leq 8n$ for all $n \geq 1$. The base case is trivial. In the step we calculate as follows:

$$\begin{aligned} E(T(n+1)) &\leq 2(n+1) + E(T(\tfrac{3(n+1)}{4})) \\ &\leq 2(n+1) + 8 \cdot \tfrac{3(n+1)}{4} && \text{by the induction hyp.} \\ &= 2n + 2 + 6n + 6 \\ &= 8(n+1) \end{aligned}$$

We conclude that QuickSelect runs in expected linear time.

3.5.2 Deterministic selection

We'll now show that selection can be done in deterministic worst-case linear time. The algorithm presented is mainly of theoretical interest since the constant involved in its $\mathcal{O}(|A|)$ running time is rather high compared to QuickSelect. For simplicity, we shall assume that $n = |A|$ is a power of 5. The algorithm then proceeds in the following steps:

Step 1: Scan through A while considering the elements in groups of 5. Find the median of each group and copy these "baby-medians" to another array B of length $\frac{n}{5}$.

Step 2: Apply the selection algorithm recursively to B with $k = \frac{n}{10}$ and find the median m of the baby-medians.

Step 3: Use m instead of $A[r]$ and proceed with DutchFlag and recursive calls as in QuickSelect.

We are not going to write this out in detail, but we'll analyze the running time as follows: First, as there are $\frac{n}{10}$ baby-medians less than m, there are at

least $\frac{3n}{10}$ elements of A less than m and symmetrically at least $\frac{3n}{10}$ elements of A larger than m. Therefore, both the red and blue parts will have size at most $\frac{7n}{10}$ after DutchFlag. Now, for some suitable time unit, it takes n time units to perform the algorithm not counting the recursive calls. For $n = 1$ there are no recursive calls so the running time here is just 1 time unit. For larger n there are in the worst case two recursive calls, on arrays of sizes $\frac{n}{5}$ and $\frac{7n}{10}$. So we get the recurrence

$$T(n) \leq \begin{cases} 1 & \text{if } n = 1 \\ n + T(\frac{n}{5}) + T(\frac{7n}{10}) & \text{if } n > 1 \end{cases}$$

We can prove by induction that $T(n) \leq 10n$ for all $n \geq 1$. The base case is trivial. In the step we calculate as follows:

$$\begin{array}{rcll} T(n+1) & \leq & n+1+T(\frac{n+1}{5})+T(\frac{7(n+1)}{10}) & \\ & \leq & n+1+10\cdot\frac{n+1}{5}+10\cdot\frac{7(n+1)}{10} & \text{by the induction hyp.} \\ & = & n+1+2(n+1)+7(n+1) & \\ & = & 10(n+1) & \end{array}$$

We conclude that deterministic selection runs in time $\mathcal{O}(n)$ on an array of length n.

3.6 Examples

In the remainder of this chapter we'll solve two algorithmic problems on arrays using our programming methodology. We claim that while the development is reasonably systematic, the resulting solutions are not at all obvious, thus providing evidence of the strength of a methodological approach.

Problem 1: Given an array A of integers, find the maximal subsum of A, that is, the largest sum found in any subarray of A. If

$$A = [3, -4, 6, 3, -2, 5, -9, 4],$$

then the maximal subsum is 12, the sum of the subarray $[6, 3, -2, 5]$. We define the empty sum to be 0 and so if $A = [-4, -2, -9]$, the maximal subsum is 0.

Problem 2: Given an array A of integers, find the length of the longest monotone sequence in A. If

$$A = [6, 2, 0, 0, 3, 8, 7, 3, 4, 9],$$

then the longest monotone sequence $0, 0, 3, 3, 4, 9$ has length 6.

Note that it is not too hard to come up with quadratic solutions (in $|A|$). What we are looking for are *efficient* solutions, which in this case means something better than quadratic in $\mathcal{O}$-notation.

3.6.1 Maximal subsum

Given an array A of integers, we'll write $\text{ms}(A)$ for the maximal subsum of A. Once again, we will develop a scanning algorithm and our task will therefore be to write command-sequences $\ll$ init $\gg$, $\ll$ update $\gg$, and $\ll$ end $\gg$ so that the following algorithm becomes valid and correct:

Algorithm MaxSubsum(A)
Input : true
Constants: A
Output : $r = \text{ms}(A)$
Method : $\ll$ init $\gg$; $i \leftarrow 0$;
$\{I\}$**while** $i \neq |A|$ **do**
$\ll$ update $\gg$; $i \leftarrow i + 1$;
$\ll$ end $\gg$

The invariant has the form

$$I : (0 \leq i \leq |A|) \wedge I'$$

where I' describes the information gathered until now, that is, the information we have about $A[0..i]$. What do we need to remember about these elements? It would be natural to try storing $\text{ms}(A[0..i])$ in r because we then have the wanted result when $i = |A|$. But it is not immediately possible to update r correctly in the loop since we lack the information needed to determine whether the element $A[i]$ is part of $\text{ms}(A[0..i+1])$ or not. Now, $\text{ms}(A[0..i+1])$ is the maximum of

> the maximal subsum obtained by not using $A[i]$—this is just $\text{ms}(A[0..i])$ which we've got stored in r

and

> the maximal subsum obtained by using $A[i]$—we call this the *maximal right subsum* of $A[0..i+1]$, in short $\text{mrs}(A[0..i+1])$.

If we store $\text{mrs}(A[0..i])$ in a variable h and are able to update it correctly in the loop, we can thus also update r correctly. Updating h is easy since $\text{mrs}(A[0..i+1])$ is clearly either $\text{mrs}(A[0..i]) + A[i]$ or 0, whichever is greater.

Therefore, we are able to update both r and h without further information. The invariant now looks as follows:

$$I : (0 \leq i \leq |A|) \wedge (r = \text{ms}(A[0..i])) \wedge (h = \text{mrs}(A[0..i])).$$

The command-sequence $\ll \mathsf{init} \gg$ is followed by the assignment $i \leftarrow 0$ and so must initialize r and h to $\text{ms}(A[0..0])$ and $\text{mrs}(A[0..0])$, respectively. Both values are zero, and so

$$\ll \mathsf{init} \gg = r \leftarrow 0;\ h \leftarrow 0.$$

We can now prove the proof-burden

$$\{\mathsf{true}\} \ll \mathsf{init} \gg;\ i \leftarrow 0 \{I\}.$$

According to the discussion above, the sequence $\ll \mathsf{update} \gg$ must assign to r the maximum of $\text{ms}(A[0..i])$ and $\text{mrs}(A[0..i+1])$ and to h the maximum of $\text{mrs}(A[0..i]) + A[i]$ and 0. That is most easily done in the opposite order:

$$\ll \mathsf{update} \gg = h \leftarrow \max\{h + A[i], 0\};\ r \leftarrow \max\{r, h\}.$$

We are now able to prove the proof-burden

$$\{I \wedge i < |A|\} \ll \mathsf{update} \gg;\ i \leftarrow i + 1 \{I\}.$$

By termination of the loop we have $I \wedge (i = |A|)$ and so in particular $r = \text{ms}(A)$. The sequence $\ll \mathsf{end} \gg$ may therefore be empty. The algorithm

```
Algorithm MaxSubsum(A)
Input      : true
Constants: A
Output    : m = ms(A)
Method    : r ← 0; h ← 0; i ← 0;
             {I}while i ≠ |A| do
                   h ← max{h + A[i], 0};
                   r ← max{r, h};
                   i ← i + 1
```

is valid and correct and runs in time $\mathcal{O}(|A|)$.

3.6.2 Longest monotone sequence

We write llms(A) for the **l**ength of the **l**ongest **m**onotone **s**equence of A. We attempt to compute it using yet another scanning algorithm:

Algorithm LLMS(A)
Input : true
Constants: A
Output : $r = \text{llms}(A)$
Method : $\ll$ init $\gg$; $i \leftarrow 0$;
$\{I\}$**while** $i \neq |A|$ **do**
$\ll$ update $\gg$; $i \leftarrow i+1$;
$\ll$ end $\gg$

The natural choice of an invariant is

$$I : (0 \leq i \leq |A|) \wedge (r = \text{llms}(A[0..i])) \wedge I'.$$

where I' will specify the information needed about $A[0..i]$ to update r correctly. At a first glance, this seems to involve every monotone sequence from $A[0..i]$ since all of them could be the start of a longest sequence in all of A. But notice that if there are several sequences from $A[0..i]$ of equal length, then we only need to remember one with minimal last element—because if any of the sequences of that length can be extended, so can those whose last element is smaller.

During the loop we therefore only need to store information about a single sequence of each length. Since there are a total of $|A|+1$ possible lengths we can use an array B with $|A|+1$ elements of which $B[0]$ will represent the empty sequence. In the i'th iteration of the loop, $B[l]$ for $1 \leq l \leq |A|$ must contain

> the necessary information about a monotone sequence from $A[0..i]$ of length l whose last element is minimal (if there is no sequence of length l, we just record that).

What information is necessary? First notice that if we are able to maintain B during the loop then we are also able to maintain r—it is given by the largest index in B that represents a sequence. So whenever we update $B[l]$, we can just update r to $\max\{r, l\}$. Thus, we need no information except for what it takes to maintain B itself.

In order to maintain B through the i'th iteration we as a minimum need to change the first element of B that represents a sequence whose last element is strictly larger than $A[i]$. Assume that this element has index $l \geq 1$.

The element $B[l-1]$ then represents a sequence of length $l-1$ which is either empty or whose last element is at most $A[i]$, and so it may be extended to a monotone sequence of length l, whose last element is $A[i]$. This new sequence must therefore be represented by $B[l]$ instead.

No other elements of B should be changed because the elements $B[l']$ for $l' < l$ represent sequences whose last elements are smaller than $A[i]$. And the elements $B[l']$ for $l' > l$ represent sequences whose last element is larger than $A[i]$ and so they cannot be extended with $A[i]$. So we only need to update $B[l]$ and we can find the index l using just $A[i]$ and the last elements of the represented sequences. We conclude that the only information we need in B are these last elements.

Since we have to search for the least element in B which is strictly larger than $A[i]$, it would be nice if B was sorted. Fortunately, we can make sure that it is. Consider two consecutive entries of B, containing last elements $B[l-1] = x$ and $B[l] = y$. If we had $x > y$ the first $l-1$ elements of the sequence represented by $B[l]$ would constitute a monotone sequence whose last element is less than x. This is a contradiction and therefore $B[1..l_{\max}+1]$ is sorted by definition (where $l_{\max}$ is the length of the longest sequence we have found). So if we define $B[0]$ to be $-\infty$ and $B[l']$ for $l' > l_{\max}$ to be ∞ then all of B is sorted. The invariant now looks as follows:

$$\begin{aligned} I:\quad & (0 \le i \le |A|) \wedge (r = \text{llms}(A[0..i])) \wedge \\ & (B[l] = \text{ms}_l(A[0..i]) \text{ for } 0 \le l \le |A|) \end{aligned}$$

where we define

$$\text{ms}_l(A[0..i]) = \begin{cases} -\infty & \text{if } l = 0 \\ \text{minimal last element in a monotone sequence of length } l \text{ from } A[0..i] & \text{if such exists} \\ \infty & \text{otherwise} \end{cases}$$

We must write $\ll \mathsf{init} \gg$ so that $\{\mathsf{true}\}\ll \mathsf{init} \gg;\ i \leftarrow 0\{I\}$ can be shown. Since $A[0..i]$ with $i = 0$ only contains the empty sequence, r must be initialized to 0 and B must be initialized to $[-\infty, \infty, \ldots, \infty]$ where the length of the array is $|A| + 1$.

$$\begin{aligned} \ll \mathsf{init} \gg = \quad & B \leftarrow [|A|+1 : \infty];\ B[0] \leftarrow -\infty; \\ & r \leftarrow 0 \end{aligned}$$

$\ll \mathsf{update} \gg$ must first conduct a search in B for an index l such that the inequations $B[l-1] \le A[i] < B[l]$ are satisfied. Since B is sorted we can achieve this by a variant of binary search which instead of searching for s searches for the smallest element larger than s:

Algorithm BinarySearch$'(B, s)$
Input : B sorted
Constants: B, s
Output : $(0 \leq l \leq |B|) \wedge (B[0..l] \leq s) \wedge (s < B[l..|B|])$

We leave the modification of the method of BinarySearch to the reader. After the search, $B[l]$ must be updated to $A[i]$, and we can then update r to be the larger of r and l:

$$\ll \text{update} \gg = \begin{array}{l} l \leftarrow \text{BinarySearch}'(B, A[i]); \\ B[l] \leftarrow A[i]; \\ r \leftarrow \max\{r, l\}. \end{array}$$

Finally, at termination we have $i = |A|$, and so $r = \text{llms}(A)$ as wanted. We can therefore take $\ll \text{end} \gg = \lambda$.

The complete algorithm below is valid and correct. Its running time is $\mathcal{O}(|A| \log |A|)$ because there are $|A|$ iterations of the loop, and the running time of each iteration is dominated by the binary search algorithm which takes time $\mathcal{O}(\log |B|) = \mathcal{O}(\log |A|)$.

Algorithm LLMS(A)
Input : true
Constants: A
Output : $r = \text{llms}(A)$
Method : $B \leftarrow [|A| + 1 : \infty];\ B[0] \leftarrow -\infty;$
$r \leftarrow 0;\ i \leftarrow 0;$
$\{I\}$**while** $i \neq |A|$ **do**
$l \leftarrow \text{BinarySearch}'(B, A[i]);$
$B[l] \leftarrow A[i];$
$r \leftarrow \max\{r, l\};$
$i \leftarrow i + 1$