Algorithms

— Lecture Notes for the Summer Term 2013 —

Baden-Wuerttemberg Cooperative State University

Prof. Dr. Karl Stroetmann

May 21, 2013

Contents

1 Introduction **3**
1.1 Motivation . . . 3
1.2 Overview . . . 3
1.3 Algorithms and Programs . . . 5
1.4 Algorithms and Programs . . . 5
1.5 Literature . . . 6

2 Limits of Computability **8**
2.1 The Halting Problem . . . 8
2.2 Undecidability of the Equivalence Problem . . . 13
2.3 Concluding Remarks . . . 14
2.4 Further Reading . . . 14

3 Big $\mathcal{O}$ Notation **15**
3.1 Motivation . . . 15
3.2 A Remark on Notation . . . 23
3.3 Case Study: Efficient Computation of Powers . . . 24
3.4 The Master Theorem . . . 28
3.5 Variants of Big $\mathcal{O}$ Notation . . . 32
3.6 Further Reading . . . 33

4 Hoare Logic **34**
4.1 Preconditions and Postconditions . . . 34
4.1.1 Assignments . . . 35
4.1.2 The Weakening Rule . . . 37
4.1.3 Compound Statements . . . 37
4.1.4 Conditional Statements . . . 38
4.1.5 Loops . . . 40
4.2 The Euclidean Algorithm . . . 40
4.2.1 Correctness Proof of the Euclidean Algorithm . . . 40
4.3 Symbolic Program Execution . . . 43

5 Sorting **46**
5.1 Sortieren durch Einfügen . . . 47
5.1.1 Komplexität . . . 49
5.2 Sortieren durch Auswahl . . . 50
5.2.1 Komplexität . . . 52
5.3 Sortieren durch Mischen . . . 52
5.3.1 Komplexität . . . 55
5.3.2 Eine feldbasierte Implementierung . . . 57
5.3.3 Eine iterative Implementierung von *Sortieren durch Mischen* 59
5.4 Der *Quick-Sort*-Algorithmus . . . 60

5.4.1 Komplexität . . . 62
5.4.2 Eine feldbasierte Implementierung von *Quick-Sort* . . . 66
5.4.3 Korrektheit . . . 69
5.4.4 Mögliche Verbesserungen . . . 71
5.5 Eine untere Schranke für die Anzahl der Vergleiche . . . 72

6 Abstract Data Types **77**
6.1 A Formal Definition of Abstract Data Types . . . 77
6.2 Implementing Abstract Data Types in SetlX . . . 79
6.3 Implementierung eines Stacks mit Hilfe eines *Arrays* . . . 83
6.4 Eine Listen-basierte Implementierung von Stacks . . . 85
6.5 Auswertung arithmetischer Ausdrücke . . . 88
6.5.1 Ein einführendes Beispiel . . . 88
6.5.2 Ein Algorithmus zur Auswertung arithmetischer Ausdrücke . 91
6.6 Nutzen abstrakter Daten-Typen . . . 97

7 Mengen und Abbildungen **98**
7.1 Der abstrakte Daten-Typ der *Abbildung* . . . 98
7.2 Geordnete binäre Bäume . . . 100
7.2.1 Implementierung geordneter binärer Bäume in SetlX . . . 105
7.2.2 Analyse der Komplexität . . . 107
7.3 AVL-Bäume . . . 113
7.3.1 Implementierung von AVL-Bäumen in SetlX . . . 117
7.3.2 Analyse der Komplexität . . . 122
7.4 Tries . . . 125
7.4.1 Einfügen in Tries . . . 127
7.4.2 Löschen in Tries . . . 128
7.4.3 Implementierung in SetlX . . . 129
7.5 Hash-Tabellen . . . 134
7.6 Mengen und Abbildungen in Java . . . 141
7.6.1 Das Interface `Collection<E>` . . . 141
7.6.2 Anwendungen von Mengen . . . 147
7.6.3 Die Schnittstelle `Map<K,V>` . . . 148
7.6.4 Anwendungen . . . 151
7.7 Das Wolf-Ziege-Kohl-Problem . . . 152
7.7.1 Die Klasse `ComparableSet` . . . 154
7.7.2 Die Klasse `ComparableList` . . . 161
7.7.3 Lösung des Wolf-Ziege-Kohl-Problems in *Java* . . . 161

8 Prioritäts-Warteschlangen **168**
8.1 Definition des ADT *PrioQueue* . . . 168
8.2 Die Daten-Struktur *Heap* . . . 170
8.2.1 Implementierung der Methode *change* . . . 173
8.3 Implementierung in SetlX . . . 175

9 Daten-Kompression **181**
9.1 Der Algorithmus von Huffman . . . 182
9.2 Optimalität des Huffman'schen Kodierungsbaums . . . 188

10 Graphentheorie **193**
10.1 Die Berechnung kürzester Wege . . . 193
10.1.1 Der Algorithmus von Moore . . . 194
10.1.2 Der Algorithmus von Dijkstra . . . 195
10.1.3 Komplexität . . . 196

Chapter 1

Introduction

1.1 Motivation

The previous lecture in the winter term has shown us how interesting problems can be solved with the help of sets and relations. However, we did not discuss how sets and relations can be represented and how the operations on sets can be implemented in an efficient way. This course will answer these questions: We will see data structure that can be used to represent sets in an efficient way. Furthermore, we will discuss a number of other data structures and algorithms that should be in the toolbox of every computer scientist.

While the class in the last term has introduced the students to the theoretical foundations of computer science, this class is more application oriented. Indeed, it may be one of the most important classes for your future career: Stanford University regularly asks their former students to rank those classes that were the most useful for their professional career. Together with programming and databases, the class on algorithms consistently ranks highest. The practical importance of the topic of this class can also be seen by the availability of book titles like "Algorithms for Interviews" [AP10] or the Google job interview questions.

1.2 Overview

This lecture covers the design and the analysis of algorithms. We will discuss the following topics.

1. Undecidability of the halting problem.

   At the beginning of the lecture we discuss the limits of computability: We will show that there is no SetlX function `doesTerminate` such that for a given function f of one argument and a string s the expression

   `doesTerminate`(f, s)

   yields `true` if the evaluation of $f(s)$ terminates and yields `false` otherwise.

2. Complexity of algorithms.

   In general, in order to solve a given problem it is not enough to develop an algorithm that implements a function f that computes values $f(x)$ for some argument x. If the size of the argument x is big, then it is also important that the computation of $f(x)$ does not take too much time. Therefore, we want to have *efficient* algorithms. In order to be able to discuss the efficiency of algorithms we have to introduce two mathematical notions.

   (a) *Recurrence relations* are discrete analogues of differential equations. Recurrence relations occur naturally when analyzing the runtime of algorithms.
   (b) *Big O notation* offers a convenient way to discuss the growth rate of functions. This notation is useful to abstract from unimportant details when discussing the runtime of algorithms.

3. Abstract data types.

   Abstract data types are a means to describe the behavior of an algorithm in a concise way.

4. Sorting algorithms.

   Sorting algorithm are the algorithms that are most frequently used in practice. As these algorithms are, furthermore, quite easy to understand they serve best for an introduction to the design of algorithms. We discuss the following sorting algorithms:

   (a) insertion sort,
   (b) selection sort,
   (c) merge sort, and
   (d) quicksort.

5. Hoare logic.

   The most important property of an algorithm is its correctness. The *Hoare calculus* is a method to investigate the correctness of an algorithm.

6. Associative arrays.

   Associative arrays are a means to represent a function. We discuss various data structures that can be used to implement associative arrays efficiently.

7. Priority queues.

   Many graph theoretical algorithms use priority queues as a basic building block. Furthermore, priority queues have important applications in the theory of operating systems and in simulation.

8. Graph theory.

   There are many applications of graphs in computer science. The topic of graph theory is very rich and can easily fill a class of its own. Therefore, we can only cover a small subset of this topic. In particular, we will discuss Dijkstra's algorithm for computing the shortest path.

9. Monte Carlo Method

   Many important problems either do not have an exact solution at all or the computation of an exact solution would be prohibitively expensive. In these cases it is often possible to use simulation in order to get an approximate solution. As a concrete example we will show how certain probabilities in Texas hold 'em poker can be determined approximately with the help of the Monte Carlo method.

The primary goal of these lectures on algorithms is not to teach as many algorithms as possible. Instead, I want to teach how to think algorithmically: At the end of these lectures, you should be able to develop your own algorithms on yourself. This is a process that requires a lot of creativity on your side, as there are no cookbook recipes that can be followed. However, once you are acquainted with a fair number of algorithms, you should be able to discover algorithms on your own.

1.3 Algorithms and Programs

This is a lecture on algorithms, not on programming. It is important that you do not mix up programs and algorithms. An algorithm is an *abstract* concept to solve a given problem. In contrast, a program is a *concrete* implementation of an algorithm. In order to implement an algorithm by a program we have to cover every detail, be it trivial or not. On the other hand, to specify an algorithm it is often sufficient to describe the interesting aspects. It is quite possible for an algorithm to leave a number of questions open.

In the literature, algorithms are mostly presented as pseudo code. Syntactically, pseudo code looks similar to a program, but in contrast to a program, pseudo code can also contain parts that are only described in natural language. However, it is important to realize that a piece of pseudo code is <u>not</u> an algorithm but is only a *representation* of an algorithm. However, the advantage of pseudo code is that we are not confined by the randomness of the syntax of a programming language.

Conceptually, the difference between an algorithm and a program is similar to the difference between an *idea* and a *text* that describes the idea. If you have an idea, you can write it down to make it concrete. As you can write down the idea in English or French or any other language, the textual descriptions of the idea might be quite different. This is the same with an algorithm: We can code it in *Java* or COBOL or any other language. The programs will be very different but the algorithm will be the same.

Having discussed the difference between algorithms and programs, let us now decide how to present algorithms in this lecture.

1. We can describe algorithms using natural language. While natural language certainly is expressive enough, it also suffers from ambiguities. Furthermore, natural language descriptions of complex algorithms tend to be very hard to understand.

2. Instead, we can describe an algorithm by implementing it. There is certainly no ambiguity in a program, but on the other hand this approach would require us to implement every aspect of an algorithm and our descriptions of algorithms would therefore get longer than we want.

3. Finally, we can try to describe algorithms in the language of mathematics. This language is concise, unambiguous, and easy to understand, once you are accustomed to it. This is therefore our method of choice.

   However, after having presented an algorithm in the language of mathematics, it is often very simple to implement this algorithm in the programming language SetlX. The reason is that SetlX is based on set theory, which is the language of mathematics. We will see that SetlX enables us to present and implement algorithms on a very high abstraction level.

1.4 Algorithms and Programs

Before we start with our discussion of algorithms we should think about our goals when designing algorithms.

1. Algorithms have to be *correct*.

2. Algorithms should be as *efficient* as possible.

3. Algorithms should be *simple*.

The first goal in this list is so self-evident that it is often overlooked. The importance of the last goal might not be as obvious as the other goals. However, the reasons for the last goal are economical: If it takes too long to code an algorithm, the cost of the implementation might well be unaffordable. Furthermore, even if the budget is unlimited there is another reasons to strife for simple algorithms: If the conceptual complexity of an algorithm is too high, it may become impossible to check the correctness of the implementation. Therefore, the third goal is strongly related to the first goal.

1.5 Literature

These lecture notes are best read online at

http://wwwlehre.dhbw-stuttgart.de/˜stroetma/Algorithms/algorithms.pdf.

They are intended to be the main source for my lecture. Additionally, I want to mention those books that have inspired me most.

1. *Alfred V. Aho*, *John E. Hopcraft*, and *Jeffrey D. Ullman*: *Data Structures and Algorithms*, Addison-Wesley, 1987, [AHU87].

   This book is quite dated now but it is one of the classics on algorithms. It discusses algorithms at an advanced level.

2. *Thomas H. Cormen*, *Charles E. Leiserson*, *Ronald L. Rivest*, and *Clifford Stein*: *Introduction to Algorithms*, third edition, MIT Press, 2009, [CLRS09]

   Due to the level of detail and the number of algorithms given, this book is well suited as a lookup library.

3. *Robert Sedgewick*: *Algorithms in Java*, fourth edition, Pearson, 2011, [SW11].

   This book has a nice booksite containing a wealth of additional material. This book seems to be the best choice for the working practitioner. Furthermore, Professor Sedgewick teaches a course on algorithms on coursera that is based on this book.

4. *Einführung in die Informatik*, written by *Heinz-Peter Gumm* and `Manfred Sommer` [GS08].

   This German book is a very readable introduction to computer science and it has a comprehensive chapter on algorithms. Furthermore, this book is available electronically in our library.

5. Furthermore, there is a set of video lectures from Professor Roughgarden at coursera.

6. *Algorithms*, written by *Sanjoy Dasgupta, Christos H. Papadimitriou, and Unmesh V. Vazirani* [DPV08] is a short text book on Algorithms that is available online.

7. *Data Structures and Algorithms* written by *Kurt Mehlhorn and Peter Sanders* [MS08] is another good text book on algorithms that is available online.

As this is the first English edition of this set of lecture notes, it will undoubtedly contain some errors. If you spot any of these errors, I would like you to send me an email. It is of no use if you tell me any of these errors after the lecture because by the time I am back at my desk I have surely forgotten them. Furthermore,

when time permits, I will gladly answer all mailed questions concerning the theory explained in this script or in my lecture.

There is one final remark I would like to make at this point: Frequently, I get questions from students concerning the exams. While I will answer most of them, I should warn you that, 50% of the time, my answers will be lies. The other 50% my answer will be either dirty lies or accidentally true.

Chapter 2

Limits of Computability

Every discipline of the sciences has its limits: Students of the medical sciences soon realize that it is difficult to raise the dead, while adepts of theology have to admit that there is no way they can walk on water. But believe it or not, even computer science has its limits! We will discuss these limits next. First, we show that we cannot decide whether a computer program will eventually terminate or whether it will run forever. Second, we prove that it is impossible to check whether two programs are equivalent.

2.1 The Halting Problem

In this subsection we prove that it is not possible for a computer program to decide whether another computer program does terminate. This problem is known as the *halting problem*. Before we give a formal proof that the halting problem is undecidable, let us discuss one example that shows why it is indeed difficult to decide whether a program does always terminate. Consider the program shown in Figure 2.1 on page 9. This program contains an obvious infinite loop in line 6. However, this loop is only entered if the expression

$$\texttt{legendre}(n)$$

in line 5 evaluates to `false`. Given a natural number `n`, the expression `legendre`(n) tests whether there is a prime between $\texttt{n}^2$ and $(\texttt{n}+1)^2$. If, however, the intervall

$$[n^2, (n+1)^2] := \{k \in \mathbb{N} \mid n^2 \leq k \wedge k \leq (n+1)^2\}$$

does not contain a prime number, then `legendre`(n) evaluates to `false` for this value of n. The function `legendre` is defined in line 2. Given a natural number n, it returns `true` if and only if the formula

$$\exists k \in \mathbb{N} : n^2 \leq k \wedge k \leq (n+1)^2 \wedge \mathit{isPrime}(k)$$

holds true. The French mathematican Adrien-Marie Legendre (1752 – 1833) conjectured that for any natural number $n \in \mathbb{N}$ there is prime number p such that

$$n^2 \leq p \wedge p \leq (n+1)^2$$

holds. Although there are a number of reasons in support of Legendre's conjecture, to this day nobody has been able to prove it. The question, whether the invocation of the function f will terminate for every user input is, therefore, unknown as it depends on the truth of Legendre's conjecture: If we had some procedure that could check whether the function `main` does terminate for every number n that is input by the user, then this procedure would be able to decide whether Legendre's theorem

is true. Therefore, it should come as no surprise that such a procedure does not exist.

```
1  main := procedure() {
2      legendre := n |-> exists (k in [n**2..(n+1)**2] | isPrime(k));
3
4      n := read("input a natural number: ");
5      if (!legendre(n)) {
6          while (true) {
7              print("looping");
8          }
9      }
10     return true;
11 };
```

Figure 2.1: A program running into trouble if Legendre's was wrong.

Let us proceed to prove formally that the halting problem is not solvable. To this end, we need the following definition.

Definition 1 (Test Function) A string t is a *test function with name* n iff t has the form

n `:= procedure(x)` $\{\ \cdots\ \}$`;`

and, furthermore, the string t can be parsed as a SetlX statement, that is the evaluation of the expression

`parseStatements(t);`

does not yield an error. The set of all test functions is denoted as TF. If $t \in \mathit{TF}$ and t has the name n, then this is written as

$\texttt{name}(t) = n.$ □

Examples:

1. s_1 = "`simple := procedure(x) { return 0; };`"

   s_1 is a test function with the name `simple`.

2. s_2 = "`loop := procedure(x) { while (true) { x := x + 1; } };`"

   s_2 is a test function with the name `loop`.

3. s_3 = "`hugo := procedure(x) { return ++x; };`"

   s_3 is not a test function. The reason is that SetlX does not support the operator "`++`". Therefore,

   `parseStatements(s3)`

   yields an error message complaining about the two "`+`" characters.

In order to be able to formalize the halting problem succinctly, we introduce three additional notations.

Notation 2 ($\rightsquigarrow$, $\downarrow$, $\uparrow$) If n is the name of a SetlX function that takes k arguments $a_1, \cdots, a_k$, then we write

$$n(a_1, \cdots, a_k) \rightsquigarrow r$$

iff the evaluation of the expression $n(a_1, \cdots, a_k)$ yields the result r. If we are not concerned with the result r but only want to state that the evaluation terminates, then we will write

$$n(a_1, \cdots, a_k) \downarrow$$

and read this notation as "*evaluating* $n(a_1, \cdots, a_k)$ *terminates*". If the evaluation of the expression $n(a_1, \cdots, a_k)$ does not terminate, this is written as

$$n(a_1, \cdots, a_k) \uparrow.$$

This notation is read as "*evaluation of* $n(a_1, \cdots, a_k)$ *diverges*". □

Examples: Using the test functions defined earlier, we have:

1. `simple("emil")` $\rightsquigarrow 0$,
2. `simple("emil")` $\downarrow$,
3. `loop(2)` $\uparrow$.

The *halting problem* for SetlX functions is the question whether there is a `SetlX` function

`stops := procedure(`t`, `a`) { ··· };`

that takes as input a test function t and a string a and that satisfies the following specification:

1. $t \notin \mathit{TF} \quad \Leftrightarrow \quad \texttt{stops}(t, a) \rightsquigarrow 2.$

   If the first argument of `stops` is not a test function, then `stops(`t`, `a`)` returns the number 2.

2. $t \in \mathit{TF} \wedge \texttt{name}(t) = n \wedge n(a) \downarrow \quad \Leftrightarrow \quad \texttt{stops}(t, a) \rightsquigarrow 1.$

   If the first argument of `stops` is a test function and, furthermore, the evaluation of $n(a)$ terminates, then `stops(`t`, `a`)` returns the number 1.

3. $t \in \mathit{TF} \wedge \texttt{name}(t) = n \wedge n(a) \uparrow \quad \Leftrightarrow \quad \texttt{stops}(t, a) \rightsquigarrow 0.$

   If the first argument of `stops` is a test function but the evaluation of $n(a)$ diverges, then `stops(`t`, `a`)` returns the number 0.

If there was a SetlX function `stops` that did satisfy the specification given above, then the halting problem for SetlX would be decidable.

Theorem 3 (Alan Turing, 1936) The halting problem is undecidable.

Proof: In order to prove the undecidabilty of the halting problem we have to show that there can be no function `stops` satisfying the specification given above. This calls for an indirect proof also known as *proof by contradiction*. We will therefore assume that a function `stops` solving the halting problem does exist and we will then show that this assumption leads to a contradiction. This contradiction will leave us with the conclusion that there can be no function `stops` that satisfies the specification given above and that, therefore, the halting problem is not solvable.

In order to proceed, let us assume that a SetlX function `stops` satisfying the specification given above exists and let us define the string *turing* as shown in Figure 2.2 below.

```
1  turing := "alan := procedure(x) {
2                 result := stops(x, x);
3                 if (result == 1) {
4                     while (true) {
5                         print(\"... looping ...\");
6                     }
7                 }
8                 return result;
9             };";
```

Figure 2.2: Definition of the string *turing*.

Given this definition it is easy to check that *turing* is, indeed, a test function with the name "`alan`', that is we have

$$turing \in TF \ \wedge \ \texttt{name}(turing) = \texttt{alan}.$$

Therefore, we can use the string *turing* as the first argument of the function `stops`. Let us think about the following expression:

```
stops(turing, turing);
```

Since we have already noted that *turing* is test function, according to the specification of the function `stops` there are only two cases left:

$$\texttt{stops}(turing, turing) \leadsto 0 \quad \vee \quad \texttt{stops}(turing, turing) \leadsto 1.$$

Let us consider these cases in turn.

1. $\texttt{stops}(turing, turing) \leadsto 0$.

   According to the specification of `stops` we should then have

   $$\texttt{alan}(turing) \uparrow.$$

   Let us check whether this is true. In order to do this, we have to check what happens when the expression

   ```
   alan(turing)
   ```

   is evaluated:

   (a) Since we have assumed for this case that the expression $\texttt{stops}(turing, turing)$ yields 0, in line 2, the variable `result` is assigned the value 0.

   (b) Line 3 now tests whether `result` is 1. Of course, this test fails. Therefore, the block of the `if`-statement is not executed.

   (c) Finally, in line 8 the value of the variable `result` is returned.

   All in all we see that the call of the function `alan` does terminate with the argument *turing*. However, this is the opposite of what the function `stops` has claimed.

   Therefore, this case has lead us to a contradiction.

2. $\texttt{stops}(turing, turing) \leadsto 1$.

   According to the specification of `stops` we should then have

   $$\texttt{alan}(turing) \downarrow,$$

   i.e. the evaluation of $\texttt{alan}(turing) \downarrow$ should terminate.

Again, let us check in detail whether this is true.

(a) Since we have assumed for this case that the expression `stops`(*turing*, *turing*) yields 1, in line 2, the variable `result` is assigned the value 1.

(b) Line 3 now tests whether `result` is 1. Of course, this time the test succeeds. Therefore, the block of the `if`-statement is executed.

(c) However, this block contains an obvious infinite loop. Therefore, the evaluation of `alan`(*turing*) diverges. But this contradicts the specification of `stops`!

Therefore, the second case also leads to a contradiction.

As we have obtained contradictions in both cases, the assumption that there is a function `stops` that solves the halting problem is refuted. □

Remark: The proof of the fact that the halting problem is undecidable was given 1936 by Alan Turing (1912 – 1954) [Tur36]. Of course, Turing did not solve the problem for SetlX but rather for the so called *Turing machines*. A *Turing maschine* can be interpreted as a formal description of an algorithm. Therefore, Turing has shown, that there is no algorithm that is able to decide whether some given algorithm will always terminate.

Remark: Having read this far you might wonder whether there might be another programming language that is more powerful so that programming in this more powerful language it would be possible to solve the halting problem. However, if you check the proof given for SetlX you will easily see that this proof can be adapted to any other programming language that is as least as powerful as SetlX.

Of course, if a programming language is very restricted, then it might be possible to check the halting problem for this weak programming language. But for any programming language that supports at least `while`-loops, `if`-statements, and the definition of procedures the argument given above shows that the halting problem is not solvable.

Exercise 1: Show that if the halting problem would be solvable, then it would be possible to write a program that checks whether there are infinitely many *twin primes*. A *twin prime* is pair of natural numbers $\langle p, p+2\rangle$ such that both p and $p+2$ are prime numbers. The *twin prime conjecture* is one of the oldest unsolved mathematical problems.

Exercise 2: A set X is called *countable* iff there is a function

$$f : \mathbb{N} \to X$$

such that for all $x \in X$ there is a $n \in \mathbb{N}$ such that x is the image of n under f:

$$\forall x \in X : \exists n \in \mathbb{N} : x = f(n).$$

Prove that the set $2^{\mathbb{N}}$, which is the set of all subsets of $\mathbb{N}$ is not countable.

Hint: Your proof should be similar to the proof that the halting problem is undecidable. Proceed as follows: Assume that there is a function f enumerating the subsets of $\mathbb{N}$, that is assume that

$$\forall x \in 2^{\mathbb{N}} : \exists n \in \mathbb{N} : x = f(n)$$

holds. Next, and this is the crucial step, define a set `Cantor` as follows:

$$\texttt{Cantor} := \bigl\{ n \in \mathbb{N} \mid n \notin f(n) \bigr\}.$$

Now try to derive a contradiction.

2.2 Undecidability of the Equivalence Problem

Unfortunately, the halting problem is not the only undecidable problem in computer science. Another important problem that is undecidable is the question whether two given functions always compute the same result. To state this more formally, we need the following definition.

Definition 4 ($\simeq$) Assume n_1 and n_2 are the names of two SETLX functions that take arguments a_1, $\cdots$, a_k. Let us define

$$n_1(a_1,\cdots,a_k) \simeq n_2(a_1,\cdots,a_k)$$

if and only if either of the following cases is true:

1. $n_1(a_1,\cdots,a_k)\uparrow \quad\wedge\quad n_2(a_1,\cdots,a_k)\uparrow$,

   that is both function calls diverge.

2. $\exists r : \Big(n_1(a_1,\cdots,a_k) \leadsto r \quad\wedge\quad n_2(a_1,\cdots,a_k) \leadsto r\Big)$

   that is both function calls terminate and compute the same result.

If $n_1(a_1,\cdots,a_k) \simeq n_2(a_1,\cdots,a_k)$ holds, then the expressions $n_1(a_1,\cdots,a_k)$ and $n_2(a_1,\cdots,a_k)$ are called *partially equivalent*. □

We are now ready to state the *equivalence problem*. A SETLX function `equal` solves the *equivalence problem* if it is defined as

```
equal := procedure(p1, p2, a) { ... };
```

and, furthermore, satisfies the following specification:

1. $p_1 \notin TF \;\vee\; p_2 \notin TF \quad\Leftrightarrow\quad \texttt{equal}(p_1,p_2,a) \leadsto 2$.

2. If

   (a) $p_1 \in TF \;\wedge\; \texttt{name}(p_1) = n_1$,

   (b) $p_2 \in TF \;\wedge\; \texttt{name}(p_2) = n_2$ and

   (c) $n_1(a) \simeq n_2(a)$

   holds, then we must have:

   $$\texttt{equal}(p_1,p_2,a) \leadsto 1.$$

3. Otherwise we must have

   $$\texttt{equal}(p_1,p_2,a) \leadsto 0.$$

Theorem 5 (Henry Gordon Rice, 1953)
The equivalence problem is undecidable.

Proof: The proof is by contradiction. Therefore, assume that there is a function `equal` such that `equal` solves the equivalence problem. Assuming `equal` exists, we will then proceed to define a function `stops` that does solve the halting problem. Figure 2.3 shows how we construct the function `stops` that makes use of the function `equal`.

Notice that in line 2 the function `equal` is called with a string that is test function with name `loop`. This test function has the following form:

```
loop := procedure(x) { while (true) {} };
```

```
1    stops := procedure(p, a) {
2        f := "loop := procedure(x) { while (true) {} };";
3        e := equal(f, p, a);
4        if (e == 2) {
5            return 2;
6        } else {
7            return 1 - e;
8        }
9    }
```

Figure 2.3: An implementation of the function `stops`.

Obviously, the function `loop` does never terminate. Therefore, if the argument p of `stops` is a test function with name n, the function `equal` will return 1 if $n(a)$ diverges, and will return 0 otherwise. But this implementation of `stops` would then solve the halting problem as for a given test function p with name n and argument a the function `stops` would return 1 if and only the evaluation of $n(a)$ terminates. As we have already proven that the halting problem is undecidable, there can be no function `equal` that solves the equivalence problem either. □

2.3 Concluding Remarks

Although, in general, we cannot decide whether a program terminates for a given input, this does not mean that we should not attempt to do so. After all, we only have proven that there is no procedure that can always check whether a given program will terminate. There might well exist a procedure for termination checking that works most of the time. Indeed, there are a number of systems that try to check whether a program will always terminate. For example, for *Prolog* programs, the article "*Automated Modular Termination Proofs for Real Prolog Programs*" [MGS96] describes a successful approach. The recent years have seen a lot of progress in this area. The article "Proving Program Termination" [CPR11] reviews these developments. However, as the recently developed systems rely on both *automatic theorem proving* and *Ramsey theory* they are quite out of the scope of this lecture.

2.4 Further Reading

The book "*Introduction to the Theory of Computation*" by Michael Sipser [Sip96] discusses the undecidability of the halting problem in section 4.2. It also covers many related undecidable problems. The exposition in the book of Sipser is based on Turing machines.

Another good book discussing undecidability is the book "*Introduction to Automata Theory, Languages, and Computation*" written by John E. Hopcroft, Rajeev Motwani and Jeffrey D. Ullman [HMU06]. This book is the third edition of a classic text. In this book, the topic of undecidability is discussed in chapter 9.

Chapter 3

Big $\mathcal{O}$ Notation

This chapter introduces both the *big O notation* and the *tilde notation* advocated by Sedgewick [SW11]. These two notions are needed to analyse the running time of algorithms. In order to have some algorithms to talk about, we will also discuss how to implement the computation of powers efficiently, i. e. we discuss how to evaluate the expression a^b for given $a, b \in \mathbb{N}$ in a way that is significantly faster than the naive approach.

3.1 Motivation

Often it is necessary to have a precise understanding of the complexitity of an algorithm. In order to obtain this understanding we could proceed as follows:

1. We implement the algorithm in a given programming language.
2. We compute how many additions, multiplications, assignments, etc. are needed for an input of a given length.
3. We read the processor handbook to look up the amount of time that is needed for the different operations.
4. Using the information discovered in the previous two steps we can then predict the running time of our algorithm for given input.

As you might have already guessed by now, this approach is problematic for a number of reasons.

1. It is very complicated.
2. The execution time of the basic operations is highly dependend on the memory hierarchy of the computer system: For many modern computer architectures, adding two numbers that happen to be in a register is more than ten times faster than adding two numbers that reside in main memory.

   However, unless we peek into the machine code generated by our compiler, it is very difficult to predict whether a variable will be stored in memory or in a register. Even if a variable is stored in main memory, we still might get lucky if the variable is also stored in a cache.
3. If we would later code the algorithm in a different programming language or if we would port the program to a computer with a different processor we would have to redo most of the computation.

The final reason shows that the approach sketched above is not well suited to measure the complexity of an algorithm: After all, the notion of an algorithm is more abstract than the notion of a program and we really need a notion measuring the complexity of an algorithm that is more abstract than the notion of the running time of a program. This notion of complexity should satisfy the following specification:

- The notion of complexity should abstract from constant factors. After all, according to *Moore's law*, computers hitting the market 18 month from now will be about twice as powerful as todays computers.

- The notion should abstract from *insignificant terms.*

  Assume you have written a program that multiplies two $n \times n$ matrices. Assume, furthermore, that you have computed the running time $T(n)$ of this program as a function of the size n of the matrix as

  $$T(n) = 3 \cdot n^3 + 2 \cdot n^2 + 7.$$

  When compared with the total running time, the portion of running time that is due to the term $2 \cdot n^2 + 7$ will decrease with increasing value of n. To see this, consider the following table:

n	$\dfrac{2 \cdot n^2 + 7}{3 \cdot n^3 + 2 \cdot n^2 + 7}$
1	0.75000000000000
10	0.06454630495800
100	0.00662481908150
1000	0.00066622484855
10 000	6.6662224852 e-05

  This table clearly shows that, for large values of n, the term $2 \cdot n^2 + 7$ can be neglected.

- The notion of complexity should describe how the running time increases when the size of the input increases: For small inputs, the running time is not very important but the question is how the running time grows when input size is increased. Therefore the notion of complexity should capture the *growth* of the running time.

Let us denote the set of all positive real numbers as $\mathbb{R}_+$, i. e. let us define

$$\mathbb{R}_+ := \{x \in \mathbb{R} \mid x > 0\}.$$

Furthermore, the set of all functions defined on $\mathbb{N}$ yielding a positive real number is defined as:

$$\mathbb{R}_+^{\mathbb{N}} = \big\{f \mid f \text{ is a function of the form } f : \mathbb{N} \to \mathbb{R}_+\big\}.$$

Definition 6 ($\mathcal{O}(g)$) Assume $g \in \mathbb{R}_+^{\mathbb{N}}$ is given. Let us define the set of all functions that *grow at most as fast* as the function g as follows:

$$\mathcal{O}(g) \;:=\; \big\{f \in \mathbb{R}_+^{\mathbb{N}} \mid \exists k \in \mathbb{N}\colon \big(\exists c \in \mathbb{R}_+\colon \forall n \in \mathbb{N}\colon n \geq k \to f(n) \leq c \cdot g(n)\big)\big\}. \quad \Box$$

The definition of $\mathcal{O}(g)$ contains three nested quantifiers and may be difficult to understand when first encountered. Therefore, let us analyse this definition carefully. We take some function g and try to understand what $f \in \mathcal{O}(g)$ means.

1. Informally, the idea is that, for increasing values of the argument n, the function f does not grow faster than the function g.

2. The fact that $f \in \mathcal{O}(g)$ holds does not impose any restriction on small values of n. After all, the condition

   $$f(n) \leq c \cdot g(n)$$

   if only required for those values of n that are bigger than k and the value k can be any suitable natural number.

   This property shows that the big $\mathcal{O}$ notation captures the growth rate of functions.

3. Furthermore, $f(n)$ can be bigger than $g(n)$ even for arbitrary values of n but it can only be bigger by a constant factor: There must be some fixed constant c such that

   $$f(n) \leq c \cdot g(n)$$

   holds for all values of n that are sufficiently big. This implies that if $f \in \mathcal{O}(g)$ holds then, for example, the function $2 \cdot f$ will also be in $\mathcal{O}(g)$.

   This last property shows that the big $\mathcal{O}$ notation abstracts from constant factors.

I have borrowed Figure 3.1 from the wikipedia article on asymptotic notation. It shows two functions $f(x)$ and $c \cdot g(x)$ such that $f \in \mathcal{O}(g)$. Note that the function $f(x)$, which is drawn in red, is less or equal than $c \cdot g(x)$ for all values of x such that $x \geq k$. In the figure, we have $k = 5$, since the condition $f(x) \leq g(x)$ is satisfied for $x \geq 5$. For values of x that are less than $k = 5$, sometimes $f(x)$ is bigger than $c \cdot g(x)$ but that does not matter. In Figure 3.1 the functions $f(x)$ and $g(x)$ are drawn as if they were functions defined for all positive real numbers. However, this is only done to support the visualization of these functions. In reality, the functions f and g are only defined for natural numbers.

Next, we discuss some examples in order to further clarify the notion $f \in \mathcal{O}(g)$.

Example: We claim that the following holds:

$$3 \cdot n^3 + 2 \cdot n^2 + 7 \in \mathcal{O}(n^3).$$

Proof: We have to provide a constant c and another constant k such that for all $n \in \mathbb{N}$ satisfying $n \geq k$ the inequality

$$3 \cdot n^3 + 2 \cdot n^2 + 7 \leq c \cdot n^3$$

holds. Let us define $k := 1$ and $c := 12$. Then we may assume that

$$1 \leq n \tag{3.1}$$

holds and we have to show that this implies

$$3 \cdot n^3 + 2 \cdot n^2 + 7 \leq 12 \cdot n^3. \tag{3.2}$$

If we take the third power of both sides of the inequality (3.1) then we see that

$$1 \leq n^3 \tag{3.3}$$

holds. Let us multiply both sides of this inequality with 7. We get:

$$7 \leq 7 \cdot n^3 \tag{3.4}$$

Furthermore, let us multiply the inequality (3.1) with the term $2 \cdot n^2$. This yields

$$2 \cdot n^2 \leq 2 \cdot n^3 \tag{3.5}$$

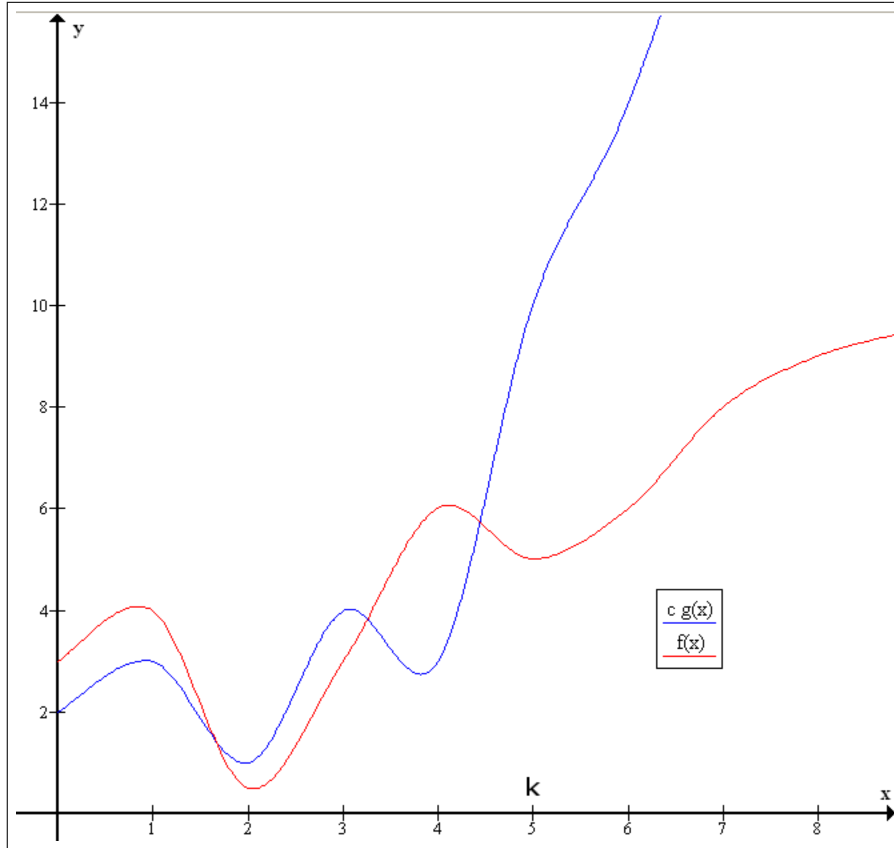

Figure 3.1: Example for $f \in \mathcal{O}(g)$.

Finally, we obviously have

$$3 \cdot n^3 \leq 3 \cdot n^3 \tag{3.6}$$

Adding up the inequalities (3.4), (3.5), and (3.6) gives

$$3 \cdot n^3 + 2 \cdot n^2 + 7 \leq 12 \cdot n^3$$

and therefore the proof is complete. □

Example: We have $n \in \mathcal{O}(2^n)$.

Proof: We have to provide a constant c and a constant k such that

$$n \leq c \cdot 2^n$$

holds for all $n \geq k$. Let us define $k := 0$ and $c := 1$. We will then have to show that

$$n \leq 2^n \quad \text{holds for all } n \in \mathbb{N}.$$

We prove this claim by induction on n.

1. **Base case**: $n = 0$

   Obviously, $n = 0 \leq 1 = 2^0 = 2^n$ holds. Therefore, $n \leq 2^n$ holds for $n = 0$.

2. **I.S.**: $n \mapsto n + 1$

   By the induction hypothesis we have

$$n \leq 2^n. \tag{3.7}$$

Furthermore, we have

$$1 \leq 2^n. \tag{3.8}$$

Adding the inequalities (3.7) and (3.8) yields

$$n + 1 \leq 2^n + 2^n = 2^{n+1}. \qquad \square$$

Remark: To be complete, we should also have proven the inequality $1 \leq 2^n$ by induction. As this proof is straightforward, it is left to the student.

Exercise 3: Prove that

$$n^2 \in \mathcal{O}(2^n). \qquad \diamond$$

It would be very tedious if we would have to use induction every time we need to prove that $f \in \mathcal{O}(g)$ holds for some functions f and g. Therefore, we show a number of properties of the big $\mathcal{O}$ notation next. These properties will later enable us to prove a claim of the form $f \in \mathcal{O}(g)$ much quicker than by induction.

Proposition 7 (Reflexivity) For all functions $f\colon \mathbb{N} \to \mathbb{R}_+$ we have that

$$f \in \mathcal{O}(f) \quad \text{holds.}$$

Proof: Let us define $k := 0$ and $c := 1$. Then our claim follows immediately from the inequality

$$\forall n \in \mathbb{N}\colon f(n) \leq f(n). \qquad \square$$

Proposition 8 (Multiplication with Constants) Assume that we have functions $f, g\colon \mathbb{N} \to \mathbb{R}_+$ and a number $d \in \mathbb{R}_+$. Then we have that

$$g \in \mathcal{O}(f) \Rightarrow d \cdot g \in \mathcal{O}(f) \quad \text{holds.}$$

Proof: The premiss $g \in \mathcal{O}(f)$ implies that there are constants $c' \in \mathbb{R}_+$ and $k' \in \mathbb{N}$ such that

$$\forall n \in \mathbb{N}\colon \big(n \geq k' \rightarrow g(n) \leq c' \cdot f(n)\big)$$

holds. If we multiply this inequality with d, we get

$$\forall n \in \mathbb{N}\colon \big(n \geq k' \rightarrow d \cdot g(n) \leq d \cdot c' \cdot f(n)\big)$$

Let us therefore define $k := k'$ and $c := d \cdot c'$. Then we have

$$\forall n \in \mathbb{N}\colon \big(n \geq k \rightarrow d \cdot g(n) \leq c \cdot f(n)\big)$$

and by definition this implies $d \cdot g \in \mathcal{O}(f)$. $\square$

Remark: The previous proposition shows that the big $\mathcal{O}$ notation does indeed abstract from constant factors.

Proposition 9 (Addition) Assume that $f, g, h\colon \mathbb{N} \to \mathbb{R}_+$. Then we have

$$f \in \mathcal{O}(h) \wedge g \in \mathcal{O}(h) \;\rightarrow\; f + g \in \mathcal{O}(h).$$

Proof: The preconditions $f \in \mathcal{O}(h)$ and $g \in \mathcal{O}(h)$ imply that there are constants $k_1, k_2 \in \mathbb{N}$ and $c_1, c_2 \in \mathbb{R}$ such that both

$$\forall n \in \mathbb{N}\colon \big(n \geq k_1 \rightarrow f(n) \leq c_1 \cdot h(n)\big) \quad \text{and}$$

$$\forall n \in \mathbb{N}\colon \big(n \geq k_2 \rightarrow g(n) \leq c_2 \cdot h(n)\big)$$

holds. Let us define $k := \max(k_1, k_2)$ and $c := c_1 + c_2$. For all $n \in \mathbb{N}$ such that $n \geq k$ it then follows that both

$$f(n) \leq c_1 \cdot h(n) \quad \text{and} \quad g(n) \leq c_2 \cdot h(n)$$

holds. Adding these inequalities we conclude that

$$f(n) + g(n) \leq (c_1 + c_2) \cdot h(n) = c \cdot h(n)$$

holds for all $n \geq k$. □

Exercise 4: Assume that $f_1, f_2, h_1, h_2\colon \mathbb{N} \rightarrow \mathbb{R}_+$. Prove that

$$f_1 \in \mathcal{O}(h_1) \wedge f_2 \in \mathcal{O}(h_2) \;\rightarrow\; f_1 \cdot f_2 \in \mathcal{O}(h_1 \cdot h_2) \quad \text{holds.}$$ ◇

Exercise 5: Assume that $f_1, f_2, h_1, h_2\colon \mathbb{N} \rightarrow \mathbb{R}_+$. Prove or refute the claim that

$$f_1 \in \mathcal{O}(h_1) \wedge f_2 \in \mathcal{O}(h_2) \;\rightarrow\; f_1/f_2 \in \mathcal{O}(h_1/h_2) \quad \text{holds.}$$ ◇

Proposition 10 (Transitivity) Assume $f, g, h\colon \mathbb{N} \rightarrow \mathbb{R}_+$. Then we have

$$f \in \mathcal{O}(g) \wedge g \in \mathcal{O}(h) \;\rightarrow\; f \in \mathcal{O}(h).$$

Proof: The precondition $f \in \mathcal{O}(g)$ implies that there exists a $k_1 \in \mathbb{N}$ and a number $c_1 \in \mathbb{R}$ such that

$$\forall n \in \mathbb{N}\colon \big(n \geq k_1 \rightarrow f(n) \leq c_1 \cdot g(n)\big)$$

holds, while the precondition $g \in \mathcal{O}(h)$ implies the existence of $k_2 \in \mathbb{N}$ and $c_2 \in \mathbb{R}$ such that

$$\forall n \in \mathbb{N}\colon \big(n \geq k_2 \rightarrow g(n) \leq c_2 \cdot h(n)\big)$$

holds. Let us define $k := \max(k_1, k_2)$ and $c := c_1 \cdot c_2$. Then for all $n \in \mathbb{N}$ such that $n \geq k$ we have the following:

$$f(n) \leq c_1 \cdot g(n) \quad \text{and} \quad g(n) \leq c_2 \cdot h(n).$$

Let us multiply the second of these inequalities with c_1. Keeping the first inequality this yields

$$f(n) \leq c_1 \cdot g(n) \text{ and } c_1 \cdot g(n) \leq c_1 \cdot c_2 \cdot h(n).$$

However, this immediately implies $f(n) \leq c \cdot h(n)$ and our claim has been proven. □

Proposition 11 (Limit Proposition) Assume that $f, g\colon \mathbb{N} \rightarrow \mathbb{R}_+$. Furthermore, assume that the limit

$$\lim_{n \to \infty} \frac{f(n)}{g(n)}$$

exists. Then we have $f \in \mathcal{O}(g)$.

Proof: Define

$$\lambda := \lim_{n \to \infty} \frac{f(n)}{g(n)}.$$

According to the definition of the notion of a limit there exists a number $k \in \mathbb{N}$ such that for all $n \in \mathbb{N}$ satisfying $n \geq k$ the inequality

$$\left| \frac{f(n)}{g(n)} - \lambda \right| \leq 1$$

holds. Let us multiply this inequality with $g(n)$. This yields

$$|f(n) - \lambda \cdot g(n)| \leq g(n).$$

The triangle inequality $|a+b| \leq |a| + |b|$ for real numbers tells us that

$$f(n) \leq \bigl|f(n) - \lambda \cdot g(n)\bigr| + \lambda \cdot g(n)$$

holds. Combining the previous two inequalities yields

$$f(n) \leq g(n) + \lambda \cdot g(n) = (1+\lambda) \cdot g(n).$$

If we now define

$$c := 1 + \lambda,$$

then we have shown that $f(n) \leq c \cdot g(n)$ holds for all $n \geq k$. □

The following examples show how to put the previous propositions to good use.

Example: Assume $k \in \mathbb{N}$. Then we have

$$n^k \in \mathcal{O}(n^{k+1}).$$

Proof: We have

$$\lim_{n\to\infty} \frac{n^k}{n^{k+1}} = \lim_{n\to\infty} \frac{1}{n} = 0.$$

Therefore, the claim follows from the limit proposition. □

Example: Assume $k \in \mathbb{N}$ and $\lambda \in \mathbb{R}$ where $\lambda > 1$. Then we have

$$n^k \in \mathcal{O}(\lambda^n).$$

Proof: We will show that

$$\lim_{n\to\infty} \frac{n^k}{\lambda^n} = 0 \tag{3.9}$$

is true. Then the claim is an immediate consequence of the limit proposition. According to L'Hôspital's rule, the limit can be computed as follows:

$$\lim_{n\to\infty} \frac{n^k}{\lambda^n} = \lim_{x\to\infty} \frac{x^k}{\lambda^x} = \lim_{x\to\infty} \frac{\frac{d\,x^k}{dx}}{\frac{d\,\lambda^x}{dx}}.$$

The derivatives can be computed as follows:

$$\frac{d\,x^k}{dx} = k \cdot x^{k-1} \quad \text{and} \quad \frac{d\,\lambda^x}{dx} = \ln(\lambda) \cdot \lambda^x.$$

We compute the second derivative and get

$$\frac{d^2\,x^k}{dx^2} = k \cdot (k-1) \cdot x^{k-2} \quad \text{and} \quad \frac{d^2\,\lambda^x}{dx^2} = \ln(\lambda)^2 \cdot \lambda^x.$$

In the same manner, we compute the k-th order derivative and find

$$\frac{d^k\,x^k}{dx^k} = k \cdot (k-1) \cdot \cdots \cdot 1 \cdot x^0 = k! \quad \text{and} \quad \frac{d^k\,\lambda^x}{dx^k} = \ln(\lambda)^k \cdot \lambda^x.$$

After k applications of L'Hôspital's rule we arrive at the following chain of equations:

$$\begin{aligned} \lim_{x\to\infty} \frac{x^k}{\lambda^x} &= \lim_{x\to\infty} \frac{\frac{d\,x^k}{dx}}{\frac{d\,\lambda^x}{dx}} = \lim_{x\to\infty} \frac{\frac{d^2\,x^k}{dx^2}}{\frac{d^2\,\lambda^x}{dx^2}} = \cdots \\ &= \lim_{x\to\infty} \frac{\frac{d^k\,x^k}{dx^k}}{\frac{d^k\,\lambda^x}{dx^k}} = \lim_{x\to\infty} \frac{k!}{\ln(\lambda)^k \lambda^x} = 0. \end{aligned}$$

Therefore the limit exists and the claim follows from the limit proposition. □

Example: We have $\ln(n) \in \mathcal{O}(n)$.

Proof: This claim is again a simple consequence of the limit proposition. We will use L'Hôpital's rule to show that we have

$$\lim_{n\to\infty} \frac{\ln(n)}{n} = 0.$$

We know that

$$\frac{d\,\ln(x)}{dx} = \frac{1}{x} \quad \text{and} \quad \frac{d\,x}{dx} = 1.$$

Therefore, we have

$$\lim_{n\to\infty} \frac{\ln(n)}{n} = \lim_{x\to\infty} \frac{\frac{1}{x}}{1} = \lim_{x\to\infty} \frac{1}{x} = 0.$$ □

Exercise 6: Prove that $\sqrt{n} \in \mathcal{O}(n)$ holds. ◇

Exercise 7: Assume $\varepsilon \in \mathbb{R}$ and $\varepsilon > 0$. Prove that $n \cdot \ln(n) \in \mathcal{O}\bigl(n^{1+\varepsilon}\bigr)$ holds. ◇

Example: We have $2^n \in \mathcal{O}(3^n)$, but $3^n \notin \mathcal{O}(2^n)$. ◇

Proof: First, we have

$$\lim_{n\to\infty} \frac{2^n}{3^n} = \lim_{n\to\infty} \left(\frac{2}{3}\right)^n = 0$$

and therefore we have $2^n \in \mathcal{O}\bigl(3^n\bigr)$. The proof of $3^n \notin \mathcal{O}(2^n)$ is a proof by contradiction. Assume that $3^n \in \mathcal{O}(2^n)$ holds. Then, there must be numbers c and k such that

$$3^n \leq c \cdot 2^n \quad \text{holds for } n \geq k.$$

Taking the logarithm of both sides of this inequality we find

$$\begin{array}{lll} & \ln(3^n) & \leq & \ln(c \cdot 2^n) \\ \leftrightarrow & n \cdot \ln(3) & \leq & \ln(c) + n \cdot \ln(2) \\ \leftrightarrow & n \cdot \bigl(\ln(3) - \ln(2)\bigr) & \leq & \ln(c) \\ \leftrightarrow & n & \leq & \dfrac{\ln(c)}{\ln(3) - \ln(2)} \end{array}$$

The last inequality would have to hold for all natural numbers n that are bigger than k. Obviously, this is not possible as, no matter what value c takes, there are natural numbers n that are bigger than

$$\frac{\ln(c)}{\ln(3) - \ln(2)}.$$ □

Exercise 8:

1. Assume that $b > 1$. Prove that $\log_b(n) \in \mathcal{O}(\ln(n))$.
2. Prove $3 \cdot n^2 + 5 \cdot n + \sqrt{n} \in \mathcal{O}(n^2)$.
3. Prove $7 \cdot n + \bigl(\log_2(n)\bigr)^2 \in \mathcal{O}(n)$.
4. Prove $\sqrt{n} + \log_2(n) \in \mathcal{O}\left(\sqrt{n}\right)$.
5. Assume that $f, g \in \mathbb{R}_+^{\mathbb{N}}$ and that, furthermore, $f \in \mathcal{O}(g)$. Proof or refute the claim that this implies
$$2^{f(n)} \in \mathcal{O}\bigl(2^{g(n)}\bigr).$$
6. Assume that $f, g \in \mathbb{R}_+^{\mathbb{N}}$ and that, furthermore,
$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0.$$
Proof or refute the claim that this implies
$$2^{f(n)} \in \mathcal{O}\bigl(2^{g(n)}\bigr).$$
7. Prove $n^n \in \mathcal{O}\bigl(2^{2^n}\bigr)$.

   Hint: The last one is difficult! ⋄

3.2 A Remark on Notation

Technically, for some function $g : \mathbb{N} \to \mathbb{R}_+$ the expression $\mathcal{O}(g)$ denotes a set. Therefore, for a given function $f : \mathbb{N} \to \mathbb{R}_+$ we can either have

$$f \in \mathcal{O}(g) \quad \text{or} \quad f \notin \mathcal{O}(g),$$

we can never have $f = \mathcal{O}(g)$. However, in the literature it has become common to abuse the notation and write

$$f = \mathcal{O}(g) \quad \text{instead of} \quad f \in \mathcal{O}(g).$$

Where convenient, we will also use this notation. However, you have to be aware of the fact that this really is an abuse of notation. For example, if we have two different functions f_1 and f_2 such that both

$$f_1 \in \mathcal{O}(g) \quad \text{and} \quad f_2 \in \mathcal{O}(g)$$

holds, we have to be aware that when we write instead

$$f_1 = \mathcal{O}(g) \quad \text{and} \quad f_2 = \mathcal{O}(g),$$

then we must not conclude that $f_1 = f_2$ as the functions f_1 and f_2 are merely members of the same set $\mathcal{O}(g)$.

Furthermore, for given functions f, g, and h we write

$$f = g + \mathcal{O}(h)$$

to express the fact that $(f - g) \in \mathcal{O}(h)$. For example, we have

$$n^2 + \tfrac{1}{2} \cdot n \cdot \log_2(n) + 3 \cdot n = n^2 + \mathcal{O}\bigl(n \cdot \log_2(n)\bigr).$$

This is true because

$$\tfrac{1}{2} \cdot n \cdot \log_2(n) + 3 \cdot n \in \mathcal{O}\bigl(n \cdot \log_2(n)\bigr).$$

The notation $f = g + \mathcal{O}(h)$ is useful because it is more precise than the pure big $\mathcal{O}$ notation. For example, assume we have two algorithms A and B for sorting a list of length n. Assume further that the number $count_A(n)$ of comparisons used by algorithm A to sort a list of length n is given as

$$count_A(n) = n \cdot \log_2(n) + 7 \cdot n,$$

while for algorithm B the corresponding number of comparisons is given as

$$count_B(n) = \frac{3}{2} \cdot n \cdot \log_2(n) + 4 \cdot n.$$

Then the big $\mathcal{O}$ notation is not able to distinguish between the complexity of algorithm A and algorithm B since we have

$$count_A(n) \in \mathcal{O}\big(n \cdot \log_2(n)\big) \quad \text{as well as} \quad count_B(n) \in \mathcal{O}\big(n \cdot \log_2(n)\big).$$

However, by writing

$$count_A(n) = n \cdot \log_2(n) + \mathcal{O}(n) \quad \text{and} \quad count_B(n) = \frac{3}{2} \cdot n \cdot \log_2(n) + \mathcal{O}(n)$$

we can abstract from lower order terms while still retaining the leading coefficient of the term determining the complexity.

3.3 Case Study: Efficient Computation of Powers

Let us study an example to clarify the notions introduced so far. Consider the program shown in Figure 3.2. Given an integer m and a positive natural number n, `power`(m, n) computes m^n. The basic idea is to compute the value of m^n according to the formula

$$m^n = \underbrace{m \cdot \ldots \cdot m}_{n}.$$

```
1   power := procedure(m, n) {
2       r := 1;
3       for (i in {1 .. n}) {
4           r := r * m;
5       }
6       return r;
7   };
```

Figure 3.2: Naive computation of m^n for $m, n \in \mathbb{N}$.

This program is obviously correct. The computation of m^n requires $n-1$ multiplication if the function `power` is implemented as shown in Figure 3.2. Fortunately, there is an algorithm for computing m^n that is much more efficient. Consider we have to evaluate m^4. We have

$$m^4 = (m \cdot m) \cdot (m \cdot m).$$

If the expression $m \cdot m$ is computed just once, the the computation of m^4 needs only two multiplications while the naive approach would already need 3 multiplications. In order to compute m^8 we can proceed according to the following formula:

$$m^8 = \big((m \cdot m) \cdot (m \cdot m)\big) \cdot \big((m \cdot m) \cdot (m \cdot m)\big).$$

If the expression $(m \cdot m) \cdot (m \cdot m)$ is computed only once, then we need just 3 multiplications in order to compute m^8. On the other hand, the naive approach would take 7 multiplications to compute m^8. The general case is implemented in the program

shown in Figure 3.3. In this program, the value of m^n is computed according to the *divide and conquer* paradigm. The basic idea that makes this program work is captured by the following formula:

$$m^n = \begin{cases} m^{n/2} \cdot m^{n/2} & \text{if } n \text{ is even;} \\ m^{n/2} \cdot m^{n/2} \cdot m & \text{if } n \text{ is odd.} \end{cases}$$

```
1  power := procedure(m, n) {
2      if (n == 0) {
3          return 1;
4      }
5      p := power(m, n \ 2);
6      if (n % 2 == 0) {
7          return p * p;
8      } else {
9          return p * p * m;
10     }
11 };
```

Figure 3.3: Berechnung von m^n für $m, n \in \mathbb{N}$.

It is by no means obvious that the program shown in 3.3 does compute m^n. We prove this claim by *computational induction*. Computational induction is an induction on the number of recursive invocations. This method is the method of choice to prove the correctness of a recursive procedure. The method of computational induction consists of two steps:

1. The *base case*.

   In the base case we have to show that the procedure is correct in all those cases were it does not invoke itself recursively.

2. The *induction step*.

   In the induction step we have to prove that the method works in all those cases were it does invoke itself recursively. In order to prove the correctness of these cases we may assume that the recursive invocations work correctly. This assumption is called the *induction hypotheses*.

Let us prove the claim

$$\texttt{power}(m, n) \leadsto m^n$$

by computational induction.

1. **Base case**:

   The only case were `power` does not invoke itself recursively is the case $n = 0$. In this case, we have

   $$\texttt{power}(m, 0) \leadsto 1 = m^0.$$

2. **Induction step**:

   The recursive invocation of `power` has the form $\texttt{power}(m, n\backslash 2)$. By the induction hypotheses we know that

   $$\texttt{power}(m, n\backslash 2) \leadsto m^{n\backslash 2}$$

   holds. After the recursive invocation there are two different cases:

(a) $n \,\%\, 2 = 0$, therefore n is even.
Then there exists a number $k \in \mathbb{N}$ such that $n = 2 \cdot k$ and therefore $n\backslash 2 = k$. Then, we have the following:

$$\begin{array}{lcl} \texttt{power}(m,n) & \leadsto & \texttt{power}(m,k) \cdot \texttt{power}(m,k) \\ & \stackrel{I.V.}{\leadsto} & m^k \cdot m^k \\ & = & m^{2\cdot k} \\ & = & m^n. \end{array}$$

(b) $n \,\%\, 2 = 1$, therefore n is odd.
Then there exists a number $k \in \mathbb{N}$ such that $n = 2 \cdot k + 1$ and we have $n\backslash 2 = k$, where $n\backslash 2$ denotes integer division of n by 2. In this case we have:

$$\begin{array}{lcl} \texttt{power}(m,n) & \leadsto & \texttt{power}(m,k) \cdot \texttt{power}(m,k) \cdot m \\ & \stackrel{I.V.}{\leadsto} & m^k \cdot m^k \cdot m \\ & = & m^{2\cdot k+1} \\ & = & m^n. \end{array}$$

As we have $\texttt{power}(m,n) = m^n$ in both cases, the proof is finished. □

Next, we want to investigate the computational complexity of this implementation of `power`. To this end, let us compute the number of multiplications that are done when $\texttt{power}(m,n)$ is called. If the number n is odd there will be more multiplications than in the case when n is even. Let us first investigate the *worst case*. The worst case happens if there is an $l \in \mathbb{N}$ such that

$$n = 2^l - 1$$

because then we have

$$n\backslash 2 = 2^{l-1} - 1 \quad \text{and} \quad n \,\%\, 2 = 1,$$

because in that case we have

$$2 \cdot (n\backslash 2) + n \,\%\, 2 = 2 \cdot (2^{l-1} - 1) + 1 = 2^l - 1 = n.$$

Therefore, if $n = 2^l - 1$ the exponent n will be odd on every recursive call. Therefore, let us assume $n = 2^l - 1$ and let us compute the number a_n of multiplications that are done when $\texttt{power}(m,n)$ is evaluated.

First, we have $a_0 = 0$, because if we have $n = 2^0 - 1 = 0$, then the evaluation of $\texttt{power}(m,n)$ does not require a single multiplication. Otherwise, we have in line 9 two multiplication that have to be added to those multiplications that are performed in the recursive call in line 5. Therefore, we get the following *recurrence relation*:

$$a_n = a_{n\backslash 2} + 2 \qquad \text{for all } n \in \left\{2^l - 1 \mid l \in \mathbb{N}\right\} \quad \text{and } a_0 = 0.$$

In order to solve this recurrence relation, let us define $b_l := a_{2^l-1}$. Then, the sequence $(b_l)_l$ satisfies the recurrence relation

$$b_l = a_{2^l-1} = a_{(2^l-1)\backslash 2} + 2 = a_{2^{l-1}-1} + 2 = b_{l-1} + 2 \qquad \text{für alle } l \in \mathbb{N}$$

and the initial term b_0 satisfies $b_0 = a_{2^0-1} = a_0 = 0$. It is quite obvious that the solution of this recurrence relation is given by

$$b_l = 2 \cdot l \qquad \text{for all } l \in \mathbb{N}.$$

Those who don't believe me should try to verify this claim by induction. Then, the

sequence a_n satisfies

$$a_{2^l-1} = 2 \cdot l.$$

Let us solve the equation $n = 2^l - 1$ for l. This yields $l = \log_2(n+1)$. Substituting this expression in the formula above gives

$$a_n = 2 \cdot \log_2(n+1) \in \mathcal{O}\bigl(\log_2(n)\bigr).$$

Next, we consider the best case. The computation of `power`(m, n) needs the least number of multiplications if the test `n % 2 == 0` always evaluates as true. In this case, n must be a power of 2. Therefore, there must exist an $l \in \mathbb{N}$ such that we have

$$n = 2^l.$$

Therefore, let us now assume $n = 2^l$ and let us again compute the number a_n of multiplications that are needed to compute `power`(m, n).

First, we have $a_{2^0} = a_1 = 2$, because if $n = 1$, the test `n % 2 == 0` fails and in this case line 9 yields 2 multiplications. Furthermore, in this case line 5 does not add any multiplications since the call `power`$(m, 0)$ immediately returns its result.

Now, if $n = 2^l$ and $n > 1$ then line 7 yields one multiplication that has to be added to those multiplications that are done during the recursive invocation of `power` in line 5. Therefore, we have the following recurrence relation:

$$a_n = a_{n\backslash 2} + 1 \qquad \text{for all } n \in \bigl\{2^l \mid l \in \mathbb{N}\bigr\} \quad \text{and } a_1 = 2.$$

Let us define $b_l := a_{2^l}$. Then the sequence $(b_l)_l$ satisfies the recurrence relation

$$b_l = a_{2^l} = a_{(2^l)\backslash 2} + 1 = a_{2^{l-1}} + 1 = b_{l-1} + 1 \qquad \text{for all } l \in \mathbb{N},$$

and the initial value is given as $b_0 = a_{2^0} = a_1 = 2$. Therefore, we have to solve the recurrence relation

$$b_{l+1} = b_l + 1 \qquad \text{for all } l \in \mathbb{N} \quad \text{with } b_0 = 2.$$

Obviously, the solution is

$$b_l = 2 + l \qquad \text{for all } l \in \mathbb{N}.$$

If we substitute this into the definition of b_l in terms of a_l we have:

$$a_{2^l} = 2 + l.$$

If we solve the equation $n = 2^l$ for l we get $l = \log_2(n)$. Substituting this values leads to

$$a_n = 2 + \log_2(n) \in \mathcal{O}\bigl(\log_2(n)\bigr).$$

Since we have gotten the same result both in the worst case and in the best case we may conclude that in general the number a_n of multiplications satisfies

$$a_n \in \mathcal{O}\bigl(\log_2(n)\bigr). \qquad \square$$

Remark: In reality, we are not interested in the number of multiplications but we are rather interested in the amount of computation time needed by the algorithm given above. However, this computation would be much more tedious because then we would have to take into account that the time needed to multiply to numbers depends on the size of these numbers.

Exercise 9: Implement a procedure `prod` that multiplies two numbers: For given natural numbers m and n, the expression `prod`(m, n) should compute the product $m \cdot n$. Of course, your implementation must not use the multiplication operator "`*`". However, you may use the operators "\" and "`%`" provided the second argument of

these operators is the number 2. The reason it that division by 2 can be implemented by a simple shift, while $n \,\%\, 2$ is just the last bit of n.

In your implementation, you should use the divide and conquer paradigm. Furthermore, you should use computational induction to prove the correctness of your implementation. Finally, you should provide an estimate for the number of additions needed to compute `prod`(m, n). This estimate should make use of the big $\mathcal{O}$ notation. ◇

3.4 The Master Theorem

In order to analyze the complexity of the procedure *power*(), we have first computed a recurrence relation, then we have solved this recurrence and, finally, we have approximated the result using the big $\mathcal{O}$ notation. If we are only interested in this last approximation then, in many cases, it is not necessary to solve the recurrence relation. Instead, we can use the *master theorem*. We present a simplified version of this theorem.

Theorem 12 (Master Theorem) Assume that

1. $\alpha, \beta \in \mathbb{N}$ such that $\alpha \geq 1$ and $\beta \geq 2$,
2. $\delta \in \mathbb{R}$ and $\delta \geq 0$,
3. the function $f : \mathbb{N} \to \mathbb{R}_+$ satisfies the recurrence relation
$$f(n) = \alpha \cdot f\,(n \backslash \beta) + \mathcal{O}(n^\delta),$$
where $n \backslash \beta$ denotes *integer division*[1] of n by β.

Then we have the following:

1. $\alpha < \beta^\delta \to f(n) \in \mathcal{O}\big(n^\delta\big)$,
2. $\alpha = \beta^\delta \to f(n) \in \mathcal{O}\big(\log_\beta(n) \cdot n^\delta\big)$,
3. $\alpha > \beta^\delta \to f(n) \in \mathcal{O}\big(n^{log_\beta(\alpha)}\big)$.

Proof: We will only discuss the case where n is a power of β, that is n has the form

$$n = \beta^k \quad \text{for some } k \in \mathbb{N}.$$

The general case is similar, but it is technically much more involved. Observe that the equation $n = \beta^k$ implies $k = \log_\beta(n)$. We will need this equation later. We define $a_k := f(n) = f\big(\beta^k\big)$. Then the recurrence relation for the function f is transformed into a recurrence relation for the sequence a_k as follows:

$$\begin{aligned} a_k &= f\big(\beta^k\big) \\ &= \alpha \cdot f\big(\beta^k \backslash \beta\big) + \mathcal{O}\Big(\big(\beta^k\big)^\delta\Big) \\ &= \alpha \cdot f\big(\beta^{k-1}\big) + \mathcal{O}\big(\beta^{k \cdot \delta}\big) \\ &= \alpha \cdot a_{k-1} + \mathcal{O}\big(\beta^{k \cdot \delta}\big) \\ &= \alpha \cdot a_{k-1} + \mathcal{O}\Big(\big(\beta^\delta\big)^k\Big) \end{aligned}$$

[1] For given integers $a, b \in \mathbb{N}$, the *integer division* $a \backslash b$ is defined as the biggest number $q \in \mathbb{N}$ such that $q \cdot b \leq a$. It can be implemented via the formula $a \backslash b = \mathit{floor}(a/b)$, where $\mathit{floor}(x)$ rounds x down to the nearest integer.

In order to simplify this recurrence relation, let us define

$$\gamma := \beta^\delta.$$

Then, the recurrence relation for the sequence a_k can be written as

$$a_k = \alpha \cdot a_{k-1} + \mathcal{O}\big(\gamma^k\big).$$

Let us substitute $k-1$ for k in this equation. This yields

$$a_{k-1} = \alpha \cdot a_{k-2} + \mathcal{O}\big(\gamma^{k-1}\big).$$

Next, we plug the value of a_{k-1} into the equation for a_k. This yields

$$\begin{aligned} a_k &= \alpha \cdot a_{k-1} + \mathcal{O}\big(\gamma^k\big) \\ &= \alpha \cdot \Big(\alpha \cdot a_{k-2} + \mathcal{O}\big(\gamma^{k-1}\big)\Big) + \mathcal{O}\big(\gamma^k\big) \\ &= \alpha^2 \cdot a_{k-2} + \alpha \cdot \mathcal{O}\big(\gamma^{k-1}\big) + \mathcal{O}\big(\gamma^k\big) \end{aligned}$$

Next, we can observe that

$$a_{k-2} = \alpha \cdot a_{k-3} + \mathcal{O}\big(\gamma^{k-2}\big)$$

holds and substitute the right hand side of this equation into the previous equation. This yields

$$\begin{aligned} a_k &= \alpha^2 \cdot a_{k-2} + \alpha \cdot \mathcal{O}\big(\gamma^{k-1}\big) + \mathcal{O}\big(\gamma^k\big) \\ &= \alpha^2 \cdot \Big(\alpha \cdot a_{k-3} + \mathcal{O}\big(\gamma^{k-2}\big)\Big) + \alpha \cdot \mathcal{O}\big(\gamma^{k-1}\big) + \mathcal{O}\big(\gamma^k\big) \\ &= \alpha^3 \cdot a_{k-3} + \alpha^2 \cdot \mathcal{O}\big(\gamma^{k-2}\big) + \alpha \cdot \mathcal{O}\big(\gamma^{k-1}\big) + \alpha^0 \cdot \mathcal{O}\big(\gamma^k\big) \end{aligned}$$

Proceeding in this way we get the general formula

$$\begin{aligned} a_k &= \alpha^{i+1} \cdot a_{k-(i+1)} + \alpha^i \cdot \mathcal{O}\big(\gamma^{k-i}\big) + \alpha^{i-1} \cdot \mathcal{O}\big(\gamma^{k-(i-1)}\big) + \cdots + \alpha^0 \cdot \mathcal{O}\big(\gamma^k\big) \\ &= \alpha^{i+1} \cdot a_{k-(i+1)} + \sum_{j=0}^{i} \alpha^j \cdot \mathcal{O}\big(\gamma^{k-j}\big) \end{aligned}$$

If we take this formula and set $i+1 := k$, i. e. $i := k-1$, then we arrive at

$$\begin{aligned} a_k &= \alpha^k \cdot a_0 + \sum_{j=0}^{k-1} \alpha^j \cdot \mathcal{O}\big(\gamma^{k-j}\big) \\ &= \alpha^k \cdot a_0 + \mathcal{O}\left(\gamma^k \cdot \sum_{j=0}^{k-1} \big(\tfrac{\alpha}{\gamma}\big)^j\right) \end{aligned}$$

At this point we have to remember the formula for the geometric series. This formula reads

$$\sum_{j=0}^{n} q^j = \frac{q^{n+1}-1}{q-1} \quad \text{provided } q \neq 1, \text{ while}$$

$$\sum_{j=0}^{n} q^j = n+1 \quad \text{if } q = 1.$$

For the geometric series given above, $q = \frac{\alpha}{\gamma}$. In order to proceed, we have to make a case distinction:

1. Case: $\alpha < \gamma$, i. e. $\alpha < \beta^\delta$.

   In this case, the series $\sum_{j=0}^{k-1} \big(\frac{\alpha}{\gamma}\big)^j$ is bounded by the value

$$\sum_{j=0}^{\infty} \left(\frac{\alpha}{\gamma}\right)^j = \frac{1}{1 - \frac{\alpha}{\gamma}}.$$

Since this value does not depend on k and the big $\mathcal{O}$ notation abstracts from constant factors, we are able to drop the sum. Therefore, we have

$$a_k = \alpha^k \cdot a_0 + \mathcal{O}\big(\gamma^k\big).$$

Furthermore, let us observe that, since $\alpha < \gamma$ we have that

$$\alpha^k \cdot a_0 \in \mathcal{O}\big(\gamma^k\big).$$

Therefore, the term $\alpha^k \cdot a_0$ is subsumed by $\mathcal{O}\big(\gamma^k\big)$ and we have shown that

$$a_k \in \mathcal{O}\big(\gamma^k\big).$$

The variable γ was defined as $\gamma = \beta^\delta$. Furthermore, by definition of k and a_k we have

$$k = \log_\beta(n) \quad \text{and} \quad f(n) = a_k.$$

Therefore we have

$$f(n) \in \mathcal{O}\Big(\big(\beta^\delta\big)^{\log_\beta(n)}\Big) = \mathcal{O}\Big(\big(\beta^{\log_\beta(n)}\big)^\delta\Big) = \mathcal{O}\big(n^\delta\big).$$

Thus we have shown the following:

$$\alpha < \beta^\delta \rightarrow f(n) \in \mathcal{O}\big(n^\delta\big).$$

2. Case: $\alpha = \gamma$, i. e. $\alpha = \beta^\delta$.

   In this case, all terms in the series $\sum\limits_{j=0}^{k-1} \left(\frac{\alpha}{\gamma}\right)^j$ have the value 1 and therefore we have

$$\sum_{j=0}^{k-1} \left(\frac{\alpha}{\gamma}\right)^j = \sum_{j=0}^{k-1} 1 = k.$$

   Therefore, we have

$$a_k = \alpha^k \cdot a_0 + \mathcal{O}\big(k \cdot \gamma^k\big).$$

   Furthermore, let us observe that, since $\alpha = \gamma$ we have that

$$\alpha^k \cdot a_0 \in \mathcal{O}\big(k \cdot \gamma^k\big).$$

   Therefore, the term $\alpha^k \cdot a_0$ is subsumed by $\mathcal{O}\big(k \cdot \gamma^k\big)$ and we have shown that

$$a_k \in \mathcal{O}\big(k \cdot \gamma^k\big).$$

   We have $\gamma = \beta^\delta$, $k = \log_\beta(n)$, and $f(n) = a_k$. Therefore,

$$f(n) \in \mathcal{O}\Big(\log_\beta(n) \cdot \big(\beta^\delta\big)^{\log_\beta(n)}\Big) = \mathcal{O}\big(\log_\beta(n) \cdot n^\delta\big).$$

   Thus we have shown the following:

$$\alpha = \beta^\delta \rightarrow f(n) \in \mathcal{O}\big(\log_\beta(n) \cdot n^\delta\big).$$

3. Case: $\alpha > \gamma$, i. e. $\alpha > \beta^\delta$.

   In this case we have

$$\sum_{j=0}^{k-1} \left(\frac{\alpha}{\gamma}\right)^j = \frac{\left(\frac{\alpha}{\gamma}\right)^k - 1}{\frac{\alpha}{\gamma} - 1} \in \mathcal{O}\Big(\left(\frac{\alpha}{\gamma}\right)^k\Big).$$

   Therefore, we have

$$a_k = \alpha^k \cdot a_0 + \mathcal{O}\Big(\gamma^k \cdot \left(\frac{\alpha}{\gamma}\right)^k\Big) = \alpha^k \cdot a_0 + \mathcal{O}\big(\alpha^k\big).$$

Since $\alpha^k \cdot a_0 \in \mathcal{O}\big(\alpha^k\big)$, we have shown that

$$a_k \in \mathcal{O}\big(\alpha^k\big).$$

Since $k = \log_\beta(n)$ and $f(n) = a_k$ we have

$$f(n) \in \mathcal{O}\Big(\alpha^{\log_\beta(n)}\Big).$$

Next, we observe that

$$\alpha^{\log_\beta(n)} = n^{\log_\beta(\alpha)}$$

holds. This equation is easily proven by taking the logarithm with base β on both sides of the equation. Using this equation we conclude that

$$\alpha > \beta^\delta \rightarrow f(n) \in \mathcal{O}\big(n^{\log_\beta(\alpha)}\big)$$

holds. □

Example: Assume that f satisfies the recurrence relation

$$f(n) = 9 \cdot f(n\backslash 3) + n.$$

Define $\alpha := 9$, $\beta := 3$, and $\delta := 1$. Then we have

$$\alpha = 9 > 3^1 = \beta^\delta.$$

This is the last case of the master theorem and, since

$$\log_\beta(\alpha) = \log_3(9) = 2,$$

we conclude that

$$f(n) \in \mathcal{O}(n^2) \quad \text{holds.}$$ □

Example: Assume that the function $f(n)$ satisfies the recurrence relation

$$f(n) = f(n\backslash 2) + 2.$$

We want to analyze the asymptotic growth of f with the help of the master theorem. Defining $\alpha := 1$, $\beta := 2$, $\delta = 0$ and noting that $2 \in \mathcal{O}(n^0)$ we see that the recurrence relation for f can be written as

$$f(n) = \alpha \cdot f(n\backslash\beta) + \mathcal{O}(n^\delta).$$

Furthermore, we have

$$\alpha = 1 = 2^0 = \beta^\delta.$$

Therefore, the second case of the master theorem tells us that

$$f(n) \in \mathcal{O}\big(\log_\beta(n) \cdot n^\delta\big) = \mathcal{O}\big(\log_2(n) \cdot n^0\big) = \mathcal{O}\big(\log_2(n)\big).$$ □

Example: This time, f satisfies the recurrence relation

$$f(n) = 3 \cdot f(n\backslash 4) + n^2.$$

Define $\alpha := 3$, $\beta := 4$, and $\delta := 2$. Then we have

$$f(n) = \alpha \cdot f(n\backslash\beta) + \mathcal{O}(n^\delta).$$

Since this time we have

$$\alpha = 3 < 16 = \beta^\delta$$

the first case of the master theorem tells us that

$$f(n) \in \mathcal{O}\bigl(n^2\bigr). \qquad \Box$$

Example: This next example is a slight variation of the previous example. Assume f satisfies the recurrence relation

$$f(n) = 3 \cdot f(n \backslash 4) + n \cdot \log_2(n).$$

Again, define $\alpha := 3$ and $\beta := 4$. This time we define $\delta := 1 + \varepsilon$ where ε is some small positive number that will be defined later. You can think of ε being $\frac{1}{2}$ or $\frac{1}{5}$ or even $\frac{1}{42}$. Since the logarithm of n grows slower than any positive power of n we have

$$\log_2(n) \in \mathcal{O}\bigl(n^\varepsilon\bigr).$$

We conclude that

$$n \cdot \log_2(n) \in \mathcal{O}\bigl(n \cdot n^\varepsilon\bigr) = \mathcal{O}\bigl(n^{1+\varepsilon}\bigr) = \mathcal{O}\bigl(n^\delta\bigr).$$

Therefore, we have

$$f(n) = \alpha \cdot f(n \backslash \beta) + \mathcal{O}(n^\delta).$$

Furthermore, we have

$$\alpha = 3 < 4 < 4^\delta = \beta^\delta.$$

Therefore, the first case of the master theorem tells us that

$$f(n) \in \mathcal{O}\bigl(n^{1+\varepsilon}\bigr) \quad \text{holds for all } \varepsilon > 0.$$

Hence, we have shown that

$$f(n) \in \mathcal{O}\bigl(n^{1+\frac{1}{2}}\bigr), \quad f(n) \in \mathcal{O}\bigl(n^{1+\frac{1}{5}}\bigr), \quad \text{and even} \quad f(n) \in \mathcal{O}\bigl(n^{1+\frac{1}{42}}\bigr)$$

holds. Using a stronger form of the master theorem it can be shown that

$$f(n) \in \mathcal{O}\bigl(n \cdot \log_2(n)\bigr)$$

holds. This example shows that the master theorem, as given in these lecture notes, does not always produce the most precise estimation for the asymptotic growth of a function. □

Exercise 10: For each of the following recurrence relations, use the master theorem to give estimates of the growth of the function f.

1. $f(n) = 4 \cdot f(n \backslash 2) + 2 \cdot n + 3$.
2. $f(n) = 4 \cdot f(n \backslash 2) + n^2$.
3. $f(n) = 3 \cdot f(n \backslash 2) + n^3$. ◇

Exercise 11: Consider the recurrence relation

$$f(n) = 2 \cdot f(n \backslash 2) + n \cdot \log_2(n).$$

How can you bound the growth of f using the master theorem?

Optional: Assume that n has the form $n = 2^k$ for some natural number k. Furthermore, you are told that $f(1) = 1$. Solve the recurrence relation in this case. ◇

3.5 Variants of Big $\mathcal{O}$ Notation

The big $\mathcal{O}$ notation is useful if we want to express that some function f does not grow faster than another function g. Therefore, when stating the running time of

the worst case of some algorithm, big $\mathcal{O}$ notation is the right tool to use. However, sometimes we want to state a lower bound for the complexity of a problem. For example, it can be shown that every comparison based sort algorithm needs at least $n \cdot \log_2(n)$ comparisons to sort a list of length n. In order to be able to express lower bounds concisely, we introduce the big Ω notation next.

Definition 13 **($\Omega(g)$)** Assume $g \in \mathbb{R}_+^{\mathbb{N}}$ is given. Let us define the set of all functions that grow <u>at least as fast as</u> the function g as follows:

$$\Omega(g) \;:=\; \left\{ f \in \mathbb{R}_+^{\mathbb{N}} \mid \exists k \in \mathbb{N} \colon \exists c \in \mathbb{R}_+ \colon \forall n \in \mathbb{N} \colon n \geq k \rightarrow c \cdot g(n) \leq f(n) \right\}. \qquad \square$$

It is not difficult to show that

$$f \in \Omega(g) \quad \text{if and only if} \quad g \in \mathcal{O}(f).$$

Finally, we introduce big Θ notation. The idea is that $f \in \Theta(g)$ if f and g have the same asymptotic growth rate.

Definition 14 **($\Theta(g)$)** Assume $g \in \mathbb{R}_+^{\mathbb{N}}$ is given. The set of functions that have the same asymptotic growth rate as the function g is defined as

$$\Theta(g) \;:=\; \mathcal{O}(g) \cap \Omega(g). \qquad \square$$

It can be shown that $f \in \Theta(g)$ if and only if the limit

$$\lim_{n \to \infty} \frac{f(n)}{g(n)}$$

exists and is greater than 0.

Sedgewick [SW11] claims that the Θ notation is too imprecise and advocates the tilde notation instead. For two functions $f, g : \mathbb{N} \rightarrow \mathbb{R}_+$ he defines

$$f \sim g \quad \text{iff} \quad \lim_{n \to \infty} \frac{f(n)}{g(n)} = 1.$$

To see why this is more precise, let us consider the case of two algorithms A and B for sorting a list of length n. Assume that the number $\mathit{count}_A(n)$ of comparisons used by algorithm A to sort a list of length n is given as

$$\mathit{count}_A(n) = n \cdot \log_2(n) + 7 \cdot n,$$

while for algorithm B the corresponding number of comparisons is given as

$$\mathit{count}_B(n) = \tfrac{3}{2} \cdot n \cdot \log_2(n) + 4 \cdot n.$$

Clearly, if n is big then algorithm A is better than algorithm B but as we have pointed out in a previous section, the big $\mathcal{O}$ notation is not able to distinguish between the complexity of algorithm A and algorithm B. However we have that

$$\tfrac{3}{2} \cdot \mathit{count}_A(n) \sim \mathit{count}_B(n)$$

and this clearly shows that for big values of n, algorithm A is faster than algorithm B by a factor of $\frac{3}{2}$.

3.6 Further Reading

Chapter 3 of the book "*Introduction to Algorithms*" by Cormen et. al. [CLRS09] contains a detailed description of several variants of the big O notation, while chapter 4 gives a more general version of the master theorem together with a detailed proof.

Chapter 4

Hoare Logic

In this chapter we introduce *Hoare logic*. This is a formal system that is used to prove the correctness of imperative computer programs. Hoare logic has been introduced 1969 by Sir Charles Antony Richard Hoare.

4.1 Preconditions and Postconditions

Hoare logic is based on preconditions and postconditions. If `P` is a program fragment and if F and G are logical formulæ, then we call F a precondition and G a postcondition for the program fragment `P` if the following holds: If `P` is executed in a state s such that the formula F holds in s, then the execution of `P` will change the state s into a new state s' such that G holds in s', This is written as

$$\{F\} \quad \texttt{P} \quad \{G\}.$$

We will read this notation as "*executing P changes F into G*". The formula

$$\{F\} \quad \texttt{P} \quad \{G\}$$

is called a *Hoare tripel*.

Examples:

1. The assignment "`x := 1;`" satisfies the specification

   $$\{\texttt{true}\} \quad \texttt{x := 1;} \quad \{x = 1\}.$$

   Here, the precondition is the trivial condition "`true`", since the postcondition "$x = 1$" will always be satisfied after this assignment.

2. The assignment "`x = x + 1;`" satisfies the specification

   $$\{x = 1\} \quad \texttt{x := x + 1;} \quad \{x = 2\}.$$

   If the precondition is "$x = 1$", then it is obvious that the postcondition has to be "$x = 2$".

3. Let us consider the assignment "`x = x + 1;`" again. However, this time the precondition is given as "$\mathit{prime}(x)$", which is only true if x is a prime number. This time, the Hoare tripel is given as

   $$\{\mathit{prime}(x)\} \quad \texttt{x := x + 1;} \quad \{\mathit{prime}(x-1)\}.$$

   This might look strange at first. Many students think that this Hoare triple should rather be written as

$$\{prime(x)\} \quad \texttt{x := x + 1;} \quad \{prime(x+1)\}.$$

However, this can easily be refuted by taking x to have the value 2. Then, the precondition $prime(x)$ is satisfied since 2 is a prime number. After the assignment, x has the value 3 and

$$x - 1 = 3 - 1 = 2$$

still is a prime number. However, we also have

$$x + 1 = 3 + 1 = 4$$

and as $4 = 2 \cdot 2$ we see that $x + 1$ is not a prime number!

Let us proceed to show how the different parts of a program can be specified using Hoare tripels. We start with the analysis of assignments.

4.1.1 Assignments

Let us generalize the previous example. Let us therefore assume that we have an assignment of the form

```
x := h(x);
```

and we want to investigate how the postcondition G of this assignment is related to the precondition F. To simplify matters, let us assume that the function h is invertible, i. e. we assume that there is a function h^{-1} such that we have

$$h^{-1}\bigl(h(x)\bigr) = x \quad \texttt{and} \quad h\bigl(h^{-1}(x)\bigr) = x$$

for all x. In order to understand the problem at hand, let us consider an example. The assignment

```
x := x + 1;
```

can be written as

```
x := h(x);
```

where the function h is given as

$$h(x) = x + 1$$

and the inverse function h^{-1} is

$$h^{-1}(x) = x - 1.$$

Now we are able to compute the postcondition of the assignment "`x := h(x);`" from the precondition. We have

$$\{F\} \quad \texttt{x := h(x);} \quad \{F\sigma\} \quad \text{where} \quad \sigma = \bigl[x \mapsto h^{-1}(x)\bigr].$$

Here, $F\sigma$ denotes the application of the substitution σ to the formula F. The expression $F\sigma$ is computed from the expression F by replacing every occurrence of the variable x by the term $h^{-1}(x)$. In order to understand why this is the correct way to compute the postcondition, we consider the assignment "`x := x + 1`" again and choose the formula $x = 7$ as precondition. Since $h^{-1}(x) = x - 1$, the substitution σ is given as $\sigma = [x \mapsto x - 1]$. Therefore, $F\sigma$ has the form

$$(x = 7)[x \mapsto x - 1] \;\equiv\; (x - 1 = 7).$$

I have used the symbol "$\equiv$" here in order to express that these formulæ are syntactically identical. Therefore, we have

$$\{x = 7\} \quad \texttt{x := x + 1;} \quad \{x - 1 = 7\}.$$

Since the formula $x - 1 = 7$ is equivalent to the formula $x = 8$ the Hoare tripel above can be rewritten as

$$\{x = 7\} \quad \texttt{x := x + 1;} \quad \{x = 8\}$$

and this is obviously correct: If the value of x is 7 before the assignment

"`x := x + 1;`"

is executed, then after the assignment is executed, x will have the value 8.

Let us try to understand why

$$\{F\} \quad \texttt{x := h(x);} \quad \{F\sigma\} \quad \text{mit} \quad \sigma = [x \mapsto h^{-1}(x)]$$

is, indeed, correct: Before the assignment "x := h(x);" is executed, the variable x has some fixed value x_0. The precondition F is valid for x_0. Therefore, the formula $F[x \mapsto x_0]$ is valid before the assignment is executed. However, the variable x does not occur in the formula $F[x \mapsto x_0]$ because it has been replaced by the fixed value x_0. Therefore, the formula

$$F[x \mapsto x_0]$$

remains valid after the assignment "`x = h(x);`" is executed. After this assignment, the variable x is set to $h(x_0)$. Therefore, we have

$$x = h(x_0).$$

Let us solve this equation for x_0. We find

$$h^{-1}(x) = x_0.$$

Therefore, after the assignment the formula

$$F[x \mapsto x_0] \equiv F[x \mapsto h^{-1}(x)]$$

is valid and this is the formula that we had written as $F\sigma$ above.

We conclude this discussion with another example. The unary predicate *prime* checks whether its argument is a prime number. Therefore, $\mathit{prime}(x)$ is true if x is a prime number. Then we have

$$\{\mathit{prime}(x)\} \quad \texttt{x := x + 1;} \quad \{\mathit{prime}(x-1)\}.$$

The correctness of this Hoare triple should be obvious: If x is a prime and if x is then incremented by 1, then afterwards $x - 1$ is prime.

Different Forms of Assignments Not all assignments can be written in the form "`x := h(x);`" where the function h is invertible. Often, a constant c is assigned to some variable x. If x does not occur in the precondition F, then we have

$$\{F\} \quad \texttt{x := c;} \quad \{F \wedge x = c\}.$$

The formula F can be used to restrict the values of other variables occurring in the program under consideration.

General Form of the Assignment Rule In the literature the rule for specifying an assignment is given as

$$\{F[x \mapsto t]\} \quad \texttt{x := t;} \quad \{F\}.$$

Here, t is an arbitrary term that can contain the variable x. This rule can be read as follows:

> "*If the formula $F(t)$ is valid in some state and t is assigned to x, then after this assignment we have $F(x)$.*"

This rule is obviously correct. However, it is not very useful because in order to apply this rule we first have to rewrite the precondition as $F(t)$. If t is some complex term, this can be impossible.

4.1.2 The Weakening Rule

If a program fragment P satisfies the specification

$$\{F\} \quad \texttt{P} \quad \{G\}$$

and if, furthermore, the formula G implies the validity of the formula H, that is if

$$G \to H$$

holds, then the program fragment P satisfies

$$\{F\} \quad \texttt{P} \quad \{H\}.$$

The reasoning is as follows: If after executing P we know that G is valid, then, since G implies H, the formula H has to be valid, too. Therefore, the following *verification rule*, which is known as the *weakening rule*, is valid:

$$\frac{\{F\} \quad \texttt{P} \quad \{G\}, \qquad G \to H}{\{F\} \quad \texttt{P} \quad \{H\}}$$

The formulæ written over the fraction line are called the *premisses* and the formula under the fraction line is called the *conclusion*. The conclusion and the first premiss are Hoare tripels, the second premiss is a formula of first order logic. The interpretation of this rule is that the conclusion is true if the premisses are true.

4.1.3 Compound Statements

If the program fragments P and Q have the specifications

$$\{F_1\} \quad \texttt{P} \quad \{G_1\} \quad \text{and} \quad \{F_2\} \quad \texttt{Q} \quad \{G_2\}$$

and if, furthermore, the postcondition G_1 implies the precondition F_2, then the composition P;Q of P and Q satisfies the specification

$$\{F_1\} \quad \texttt{P;Q} \quad \{G_2\}.$$

The reasoning is as follows: If, initially, F_1 is satisfied and we execute P then we have G_1 afterwards. Therefore we also have F_2 and if we now execute Q then afterwards we will have G_2. This chain of thoughts is combined in the following verification rule:

$$\frac{\{F_1\} \quad \texttt{P} \quad \{G_1\}, \qquad G_1 \to F_2, \qquad \{F_2\} \quad \texttt{Q} \quad \{G_2\}}{\{F_1\} \quad \texttt{P;Q} \quad \{G_2\}}$$

If the formulæ G_1 and F_2 are identical, then this rule can be simplified as follows:

$$\frac{\{F_1\} \quad \texttt{P} \quad \{G_1\}, \qquad \{G_1\} \quad \texttt{Q} \quad \{G_2\}}{\{F_1\} \quad \texttt{P;Q} \quad \{G_2\}}$$

Example: Let us analyse the program fragment shown in Figure 4.1. We start our analysis by using the precondition

$$\texttt{x} = a \land \texttt{y} = b.$$

Here, a and b are two variables that we use to store the initial values of x and y. The first assignment yields the Hoare triple

$$\{\texttt{x} = a \land \texttt{y} = b\} \quad \texttt{x := x - y;} \quad \{(\texttt{x} = a \land \texttt{y} = b)\sigma\}$$

where $\sigma = [x \mapsto x + y]$. The form of σ follows from the fact that the function $x \mapsto x + y$ is the inverse of the function $x \mapsto x - y$. If we apply σ to the formula $x = a \wedge y = b$ we get

$$\big\{\mathtt{x} = a \wedge \mathtt{y} = b\big\} \quad \mathtt{x\ :=\ x\ -\ y;} \quad \big\{\mathtt{x\ +\ y} = a \wedge \mathtt{y} = b\big\}. \tag{4.1}$$

The second assignment yields the Hoare triple

$$\big\{\mathtt{x\ +\ y} = a \wedge \mathtt{y} = b\big\} \quad \mathtt{y\ :=\ y\ +\ x;} \quad \big\{(\mathtt{x\ +\ y} = a \wedge \mathtt{y} = b)\sigma\big\}$$

where $\sigma = [y \mapsto y - x]$. The reason is that the function $y \mapsto y - x$ is the inverse of the function $y \mapsto y + x$. This time, we get

$$\big\{\mathtt{x\ +\ y} = a \wedge \mathtt{y} = b\big\} \quad \mathtt{y\ :=\ y\ +\ x;} \quad \big\{\mathtt{x\ +\ y\ -\ x} = a \wedge \mathtt{y\ -\ x} = b\big\}.$$

Simplifying the postcondition yields

$$\big\{\mathtt{x\ +\ y} = a \wedge \mathtt{y} = b\big\} \quad \mathtt{y\ :=\ y\ +\ x;} \quad \big\{\mathtt{y} = a \wedge \mathtt{y\ -\ x} = b\big\}. \tag{4.2}$$

Let us consider the last assignment. We have

$$\big\{\mathtt{y} = a \wedge \mathtt{y\ -\ x} = b\big\} \quad \mathtt{x\ :=\ y\ -\ x;} \quad \big\{(\mathtt{y} = a \wedge \mathtt{y\ -\ x} = b)\sigma\big\}$$

where $\sigma = [x \mapsto y - x]$, since the function $x \mapsto y - x$ is the inverse of the function $x \mapsto y - x$. This yields

$$\big\{\mathtt{y} = a \wedge \mathtt{y\ -\ x} = b\big\} \quad \mathtt{x\ :=\ y\ -\ x;} \quad \big\{\mathtt{y} = a \wedge \mathtt{y\ -\ (y\ -\ x)} = b\big\}$$

Simplifying the postcondition gives

$$\big\{\mathtt{y} = a \wedge \mathtt{y\ -\ x} = b\big\} \quad \mathtt{x\ :=\ y\ -\ x;} \quad \big\{\mathtt{y} = a \wedge \mathtt{x} = b\big\}. \tag{4.3}$$

Combining the Hoare tripels (4.1), (4.2) and (4.3) we get

$$\big\{\mathtt{x} = a \wedge \mathtt{y} = b\big\} \quad \mathtt{x:=x+y;\ y:=y+x;\ x:=y-x;} \quad \big\{\mathtt{y} = a \wedge \mathtt{x} = b\big\}. \tag{4.4}$$

The Hoare tripel (4.4) shows that the program fragment shown in Figure 4.1 swaps the values of the variables x and y: If the value of x is a and y has the value b before the program is executed, then afterwards y has the value a and x has the value b. The trick shown in Figure 4.1 can be used to swap variables without using an auxiliary variable. This is useful because when this code is compiled into machine language, the resulting code will only use two registers.

```
1    x := x - y;
2    y := y + x;
3    x := y - x;
```

Figure 4.1: A tricky way to swap variables.

4.1.4 Conditional Statements

In order to compute the effect of a conditional of the form

`if (B) { P } else { Q }`

let us assume that before the conditional statement is executed, the precondition F is satisfied. We have to analyse the effect of the program fragments P and Q. The program fragment P is only executed when B is true. Therefore, the precondition for P is $F \wedge B$. On the other hand, the precondition for the program fragment

Q is $F \wedge \neg B$, since Q is only executed if B is false. Hence, we have the following verification rule:

$$\frac{\big\{F \wedge B\big\} \;\; \texttt{P} \;\; \big\{G\big\}, \qquad \big\{F \wedge \neg B\big\} \;\; \texttt{Q} \;\; \big\{G\big\}}{\big\{F\big\} \;\; \texttt{if (}B\texttt{) P else Q} \;\; \big\{G\big\}} \tag{4.5}$$

In this form, the rule is not always applicable. The reason is that the analysis of the program fragments `P` and `Q` yields Hoare tripel of the form

$$\big\{F \wedge B\big\} \;\; \texttt{P} \;\; \big\{G_1\big\} \qquad \text{and} \qquad \big\{F \wedge \neg B\big\} \;\; \texttt{Q} \;\; \big\{G_2\big\}, \tag{4.6}$$

and in general G_1 and G_2 will be different from each other. In order to be able to apply the rule for conditionals we have to find a formula G that is a consequence of G_1 and also a consequence of G_2, i. e. we want to have

$$G_1 \rightarrow G \qquad \text{and} \qquad G_2 \rightarrow G.$$

If we find G, then the weakening rule can be applied to conclude the validity of

$$\big\{F \wedge B\big\} \;\; \texttt{P} \;\; \big\{G\big\} \qquad \text{and} \qquad \big\{F \wedge \neg B\big\} \;\; \texttt{Q} \;\; \big\{G\big\},$$

and this gives us the premisses that are needed for the rule (4.5).

Example: Let us analyze the following program fragment:

```
if (x < y) { z := x; } else { z := y; }
```

We start with the precondition

$$F = \big(\texttt{x} = a \wedge \texttt{y} = b\big)$$

and want to show that the execution of the conditional establishes the postcondition

$$G = \big(z = min(a, b)\big).$$

The first assignment "`z := x;`" gives the Hoare tripel

$$\big\{\texttt{x} = a \wedge \texttt{y} = b \wedge \texttt{x} < \texttt{y}\big\} \;\; \texttt{z := x} \;\; \big\{\texttt{x} = a \wedge \texttt{y} = b \wedge \texttt{x} < \texttt{y} \wedge \texttt{z} = \texttt{x}\big\}.$$

In the same way, the second assignment "`z := y`" yields

$$\big\{\texttt{x} = a \wedge \texttt{y} = b \wedge \texttt{x} \geq \texttt{y}\big\} \;\; \texttt{z := y} \;\; \big\{\texttt{x} = a \wedge \texttt{y} = b \wedge \texttt{x} \geq \texttt{y} \wedge \texttt{z} = \texttt{y}\big\}.$$

Since we have

$$\texttt{x} = a \wedge \texttt{y} = b \wedge \texttt{x} < \texttt{y} \wedge \texttt{z} = \texttt{x} \rightarrow \texttt{z} = \min(a, b)$$

and also

$$\texttt{x} = a \wedge \texttt{y} = b \wedge \texttt{x} \geq \texttt{y} \wedge \texttt{z} = \texttt{y} \rightarrow \texttt{z} = \min(a, b).$$

Using the weakening rule we conclude that

$$\big\{\texttt{x} = a \wedge \texttt{y} = b \wedge \texttt{x} < \texttt{y}\big\} \;\; \texttt{z := x;} \;\; \big\{\texttt{z} = \min(a, b)\big\} \qquad \text{and}$$
$$\big\{\texttt{x} = a \wedge \texttt{y} = b \wedge \texttt{x} \geq \texttt{y}\big\} \;\; \texttt{z := y;} \;\; \big\{\texttt{z} = \min(a, b)\big\}$$

holds. Now we can apply the rule for the conditional and conclude that

$$\big\{\texttt{x} = a \wedge \texttt{y} = b\big\} \;\; \texttt{if (x < y) \{ z := x; \} else \{ z := y; \}} \;\; \big\{\texttt{z} = \min(a, b)\big\}$$

holds. Thus we have shown that the program frgament above computes the minimum of the numbers a and b.

4.1.5 Loops

Finally, let us analyze the effect of a loop of the form

`while` (B) $\{$ `P` $\}$

The important point here is that the postcondition of the n-th execution of the body of the loop P is the precondition of the $(n+1)$-th execution of P. Basically this means that the precondition and the postcondition of P have to be more or less the same. Hence, this condition is called the *loop invariant*. Therefore, the details of the verification rule for `while` loops are as follows:

$$\frac{\big\{I \wedge B\big\} \quad \texttt{P} \quad \big\{I\big\}}{\big\{I\big\} \quad \texttt{while } (B) \ \{ \ \texttt{P} \ \} \quad \big\{I \wedge \neg B\big\}}$$

The premiss of this rules expresses the fact that the invariant I remains valid on execution of P. However, since P is only executed as long as B is `true`, the precondition for P is actually the formula $I \wedge B$. The conclusion of the rule says that if the invariant I is true before the loop is executed, then I will be true after the loop has finished. This result is intuitive since every time P is executed I remains valid. Furthermore, the loop only terminates once B gets `false`. Therefore, the postcondition of the loop can be strengthened by adding $\neg B$.

4.2 The Euclidean Algorithm

In this section we show the the verification rules of the last section can be used to prove the correctness of a non-trivial program. We will show that the algorithm shown in Figure 4.2 on page 40 is correct. The procedure shown in this figure implements the *Euclidean algorithm* to compute the greatest common divisor of two natural numbers. Our proof is based on the following property of the function `ggt`:

$$\texttt{ggt}(x + y, y) = \texttt{ggt}(x, y) \quad \text{for all } x, y \in \mathbb{N}.$$

```
1   ggt := procedure(x, y) {
2       while (x != y) {
3           if (x < y) {
4               y := y - x;
5           } else {
6               x := x - y;
7           }
8       }
9       return x;
10  };
```

Figure 4.2: The Euclidean Algorithm to compute the greatest common divisor.

4.2.1 Correctness Proof of the Euclidean Algorithm

To start our correctness proof we formulate the invariant of the `while` loop. Let us define

$$I := \big(x > 0 \wedge y > 0 \wedge \texttt{ggt}(x, y) = \texttt{ggt}(a, b)\big)$$

In this formula we have defined the initial values of x and y as a and b. In order to

establish the invariant at the beginning we have to ensure that the function `ggt` is only called with positive natural numbers. If we denote these numbers as a and b, then the invariant I is valid initially. The reason is that $x = a$ and $y = b$ implies $\mathtt{ggt}(x, y) = \mathtt{ggt}(a, b)$.

In order to prove that the invariant I is maintained in the loop we formulate the Hoare triples for both alternatives of the conditional. For the first conditional we know that

$$\big\{I \wedge x \neq y \wedge x < y\big\} \quad \mathtt{y\ :=\ y\ -\ x;} \quad \big\{(I \wedge x \neq y \wedge x < y)\sigma\big\}$$

holds, where σ is defined as $\sigma = [y \mapsto y + x]$. Here, the condition $x \neq y$ is the condition controlling the execution of the `while` loop and the condition $x < y$ is the condition of the `if` conditional. We rewrite the formula $(I \wedge x \neq y \wedge x < y)\sigma$:

$$\begin{array}{ll}
 & \big(I \wedge x \neq y \wedge x < y\big)\sigma \\
\leftrightarrow & \big(I \wedge x < y\big)\sigma \qquad\qquad \text{because } x < y \text{ implies } x \neq y \\
\leftrightarrow & \big(x > 0 \wedge y > 0 \wedge \mathtt{ggt}(x, y) = \mathtt{ggt}(a, b) \wedge x < y\big)[y \mapsto y + x] \\
\leftrightarrow & x > 0 \wedge y + x > 0 \wedge \mathtt{ggt}(x, y + x) = \mathtt{ggt}(a, b) \wedge x < y + x \\
\leftrightarrow & x > 0 \wedge y + x > 0 \wedge \mathtt{ggt}(x, y) = \mathtt{ggt}(a, b) \wedge 0 < y
\end{array}$$

In the last step we have used the formula

$$\mathtt{ggt}(x, y + x) = \mathtt{ggt}(x, y)$$

and we have simplified the inequality $x < y + x$ as $0 < y$. The last formula implies

$$x > 0 \wedge y > 0 \wedge \mathtt{ggt}(x, y) = \mathtt{ggt}(a, b).$$

However, this is precisely the invariant I. Therefore we have shown that

$$\big\{I \wedge x \neq y \wedge x < y\big\} \quad \mathtt{y\ :=\ y\ -\ x;} \quad \big\{I\big\} \tag{4.7}$$

holds. Next, let us consider the second alternative of the `if` conditional. We have

$$\big\{I \wedge x \neq y \wedge x \geq y\big\} \quad \mathtt{x\ :=\ x\ -\ y;} \quad \big\{(I \wedge x \neq y \wedge x \geq y)\sigma\big\}$$

where $\sigma = [x \mapsto x + y]$. The expression $(I \wedge x \neq y \wedge x \geq y)\sigma$ is rewritten as follows:

$$\begin{array}{ll}
 & \big(I \wedge x \neq y \wedge x \geq y\big)\sigma \\
\leftrightarrow & \big(I \wedge x > y\big)\sigma \\
\leftrightarrow & \big(x > 0 \wedge y > 0 \wedge \mathtt{ggt}(x, y) = \mathtt{ggt}(a, b) \wedge x > y\big)[x \mapsto x + y] \\
\leftrightarrow & x + y > 0 \wedge y > 0 \wedge \mathtt{ggt}(x + y, y) = \mathtt{ggt}(a, b) \wedge x + y > y \\
\leftrightarrow & x + y > 0 \wedge y > 0 \wedge \mathtt{ggt}(x, y) = \mathtt{ggt}(a, b) \wedge x > 0
\end{array}$$

The last formula implies that

$$x > 0 \wedge y > 0 \wedge \mathtt{ggt}(x, y) = \mathtt{ggt}(a, b).$$

holds. Again, this is our invariant I. Therefore we have shown that

$$\big\{I \wedge x \neq y \wedge x \geq y\big\} \quad \mathtt{x\ :=\ x\ -\ y;} \quad \big\{I\big\} \tag{4.8}$$

holds. If we use the Hoare triples (4.7) and (4.8) as premisses for the rule for conditionals we have shown that

$$\big\{I \wedge x \neq y\big\} \quad \mathtt{if\ (x\ <\ y)\ \{\ x\ :=\ x\ -\ y;\ \}\ else\ \{\ y\ :=\ y\ -\ x;\}} \quad \big\{I\big\}$$

holds. Now the verification rule for `while` loops yields

$\{I\}$

```
while (x != y ) {
      if (x < y) { x := x - y; } else { y := y - x;}
}
```

$\{I \wedge x = y\}$.

Expanding the invariant I in the formula $I \wedge x = y$ shows that the postcondition of the `while` loop is given as

$$x > 0 \wedge y > 0 \wedge \texttt{ggt}(x, y) = \texttt{ggt}(a, b) \wedge x = y.$$

Now the correctness of the Euclidean algorithm can be established as follows:

$$\begin{array}{ll} & x > 0 \wedge y > 0 \wedge \texttt{ggt}(x, y) = \texttt{ggt}(a, b) \wedge x = y \\ \Rightarrow & \texttt{ggt}(x, y) = \texttt{ggt}(a, b) \wedge x = y \\ \Rightarrow & \texttt{ggt}(x, x) = \texttt{ggt}(a, b) \\ \Rightarrow & x = \texttt{ggt}(a, b) \qquad \text{because} \quad \texttt{ggt}(x, x) = x. \end{array}$$

All in all we have shown the following: If the `while` loop terminates, then the variable x will be set to the greatest common divisor of a and b, where a and b are the initial values of the variables x and y. In order to finish our correctness proof we have to show that the while does indeed terminate for all choices of a and b. To this end let us define the variable s as follows:

$$s := x + y.$$

The variables x and y are natural numbers. Therefore s is a natural number, too. Every iteration of the loop reduces the number s: either x is subtracted from s or y is subtracted from s and the invariant I shows that both x and y are positive. Therefore, if the `while` loop would run forever, at some point s would get negative. Since s can not be negative, the loop must terminate. Hence we have shown the correctness of the Euclidean algorithm.

Exercise 12: Show that the function power(x, y) that is defined in Figure 4.3 does compute x^y, i. e. show that power$(x, y) = x^y$ for all natural numbers x and y.

```
1   power := procedure(x, y) {
2       r := 1;
3       while (y > 0) {
4           if (y % 2 == 1) {
5               r := r * x;
6           }
7           x := x * x;
8           y := y \ 2;
9       }
10      return r;
11  };
```

Figure 4.3: A program to compute x^y iteratively.

Hints:

1. If the initial values of x and y are called a and b, then an invariant for the while loop is given as

$$I := \bigl(r \cdot x^y = a^b\bigr).$$

2. The verification rule for the conditional without else is given as

$$\frac{\bigl\{F \wedge B\bigr\} \quad \texttt{P} \quad \bigl\{G\bigr\}, \qquad F \wedge \neg B \rightarrow G}{\bigl\{F\bigr\} \quad \texttt{if } (B) \texttt{ \{ P \}} \quad \bigl\{G\bigr\}}$$

This rule is interpreted as follows:

(a) If both the precondition F and the condition B is valid, then execution of the program fragment P has to establish the validity of the postcondition G.

(b) If the precondition F is valid but we have $\neg B$, then this must imply the postcondition G.

Remark: Proving the correctness of a nontrivial program is very tedious. Therefore, various attempts have been made to automate the task. For example, *KeY Hoare* is a tool that can be used to verify the correctness of programs. It is based on Hoare calculus.

4.3 Symbolic Program Execution

The last section has shown that using Hoare logic to verify a program can be very tedious. There is another method to prove the correctness of imperative programs. This method is called *symbolic program execution*. Let us demonstrate this method. Consider the program shown in Figure 4.4.

The main difference between a mathematical formula and a program is that in a formula all occurrences of a variable refer to the same value. This is different in a program because the variables change their values dynamically. In order to deal with this property of program variables we have to be able to distinguish the

```
1  power := procedure(x_0, y_0) {
2      r_0 := 1;
3      while (y_n > 0) {
4          if (y_n % 2 == 1) {
5              r_{n+1} := r_n * x_n;
6          }
7          x_{n+1} := x_n * x_n;
8          y_{n+1} := y_n \ 2;
9      }
10     return r_N;
11 };
```

Figure 4.4: An annotated programm to compute powers.

different occurrences of a variable. To this end, we index the program variables. When doing this we have to be aware of the fact that the same occurrence of a program variable can still denote different values if the variable occurs inside a loop. In this case we have to index the variables in a way that the index includes a counter that counts the number of loop iterations. For concreteness, consider the program shown in Figure 4.4. Here, in line 5 the variable `r` has the index n on the right side of the assignment, while it has the index r_{n+1} on the left side of the assignment in line 5. Here, n denotes the number of times the `while` loop has been iterated. After the loop in line 10 the variable is indexed as $\texttt{r}_N$, where N denotes the total number of loop iterations. We show the correctness of the given program next. Let us define

$$a := x_0, \quad b := y_0.$$

We show, that the `while` loop satisfies the invariant

$$r_n \cdot x_n^{y_n} = a^b. \tag{4.9}$$

This claim is proven by induction on the number of loop iterations.

I.A. $n = 0$: Since we have $r_0 = 1$, $x_0 = a$, and $y_0 = b$ we have

$$r_n \cdot x_n^{y_n} = r_0 \cdot x_0^{y_0} = 1 \cdot a^b = a^b.$$

I.S. $n \mapsto n + 1$: We need a case distinction with respect to $y \,\%\, 2$:

(a) $y_n \,\%\, 2 = 1$. Then we have $y_n = 2 \cdot (y_n \backslash 2) + 1$. This implies

$$\begin{aligned}
& r_{n+1} \cdot x_{n+1}^{y_{n+1}} \\
= \;& (r_n \cdot x_n) \cdot (x_n \cdot x_n)^{y_n \backslash 2} \\
= \;& r_n \cdot x_n \cdot x_n^{y_n \backslash 2} \cdot x_n^{y_n \backslash 2} \\
= \;& r_n \cdot x_n^{1 + y_n \backslash 2 + y_n \backslash 2} \\
= \;& r_n \cdot x_n^{2 \cdot (y_n \backslash 2) + 1} \\
= \;& r_n \cdot x_n^{y_n} \\
\stackrel{IH}{=} \;& a^b
\end{aligned}$$

(b) $y_n \,\%\, 2 = 0$. Then we have $y_n = 2 \cdot (y_n \backslash 2)$. This implies

$$\begin{array}{cl} & r_{n+1} \cdot x_{n+1}^{y_{n+1}} \\ = & r_n \cdot (x_n \cdot x_n)^{y_n \backslash 2} \\ = & r_n \cdot x_n^{y_n \backslash 2} \cdot x_n^{y_n \backslash 2} \\ = & r_n \cdot x_n^{y_n \backslash 2 + y_n \backslash 2} \\ = & r_n \cdot x_n^{2 \cdot (y_n \backslash 2)} \\ = & r_n \cdot x_n^{y_n} \\ \stackrel{IH}{=} & a^b \end{array}$$

This shows the validity of the equation (4.9). If the `while` loop terminates, we must have $y_N = 0$. If $n = N$ equation (4.9) yields:

$$r_N \cdot x_N^{y_N} = x_0^{y_0} \leftrightarrow r_N \cdot x_N^0 = a^b \leftrightarrow r_N \cdot 1 = a^b \leftrightarrow r_N = a^b$$

This shows $r_N = a^b$ and since we already know that the `while` loop terminates we have proven that $\texttt{power}(a, b) = a^b$.

Exercise 13: Use the method of symbolic program execution to prove the correctness of the implementation of the Euclidean algorithm that is shown in Figure 4.5. During the proof you should make use of the fact that for all positive natural numbers a and b the equation

$$\texttt{ggt}(a, b) = \texttt{ggt}(a \,\%\, b, b)$$

is valid.

```
1  ggt := procedure(a, b) {
2      while (b != 0) {
3          r := a % b;
4          a := b;
5          b := r;
6      }
7      return a;
8  };
```

Figure 4.5: An efficient version of the Euclidean algorithm.

Chapter 5

Sorting

In this chapter, we assume that we have been given list l. The elements of l are members of some set s. If we want to *sort* the list l we have to be able to compare these elements to each other. Therefore, we assume that s is equipped with a binary relation $\leq$ which is *reflexive*, *anti-symmetric* and *transitive*, i. e. we have

1. $\forall x \in s \colon x \leq x$,
2. $\forall x, y \in s \colon \bigl(x \leq y \wedge y \leq x \rightarrow x = y\bigr)$,
3. $\forall x, y, z \in s \colon \bigl(x \leq y \wedge y \leq z \rightarrow x \leq z\bigr)$.

A pair $\langle s, \leq \rangle$ where s is a set and $\leq \subseteq s \times s$ is a relation on s that is *reflexive*, *anti-symmetric* and *transitive* is called a *partially ordered set*. If, furthermore

$$\forall x, y \in s \colon \bigl(x \leq y \vee y \leq x\bigr)$$

holds, then the pair $\langle m, \leq \rangle$ is called a *totally ordered set*.

Examples:

1. $\langle \mathbb{N}, \leq \rangle$ is a totally ordered set.
2. $\langle 2^{\mathbb{N}}, \subseteq \rangle$ is a partially ordered set but it is not a totally ordered set. For example, the sets $\{1\}$ and $\{2\}$ are not comparable since we have

   $$\{1\} \not\subseteq \{2\} \quad \text{and} \quad \{2\} \not\subseteq \{1\}.$$

3. If P is the set of employees of some company and if we define for given employees $a, b \in P$

   $$a < b \quad \text{iff} \quad a \text{ earns less than } b,$$

   then $\langle P, \leq \rangle$ is a partially ordered set.

Remark: In the last example, instead of defining the relation $\leq$ we have defined the relation $<$. If a relation $<$ is given, then the corresponding relation $\leq$ is defined as follows:

$$x \leq y \stackrel{\text{def}}{\Longleftrightarrow} x < y \vee x = y.$$

In the examples given above we see that it does not make sense to sort subsets of $\mathbb{N}$. However, we can sort natural numbers with respect to their size and we can also sort employees with respect to their income. This show that, in order to sort, it is not enough to have a partially ordered set but we do not necessarily need a totally ordered set either. In order to capture the requirements that are needed to be able to sort we introduce the notion of a *quasiorder*.

Definition 15 (Quasiorder)
A pair $\langle m, \preceq \rangle$ is a quasiorder *if $\preceq$ is a binary relation on m such that we have the following:*

1. $\forall x \in s \colon x \preceq x.$ *(reflexivity)*
2. $\forall x, y, z \in s \colon \bigl(x \preceq y \wedge y \preceq z \rightarrow x \preceq z\bigr).$ *(transitivity)*

If, furthermore,

$$\forall x, y \in M \colon \bigl(x \preceq y \vee y \preceq x\bigr)$$

holds, then $\langle s, \preceq \rangle$ is called a total quasiorder*. This will be abbreviated as* TQO.

Bei dem Begriff der Quasi-Ordnung wird im Unterschied zu dem Begriff der partiellen Ordnung auf die Eigenschaft der Anti-Symmetrie verzichtet. Trotzdem sind die Begriffe fast gleichwertig, denn wenn $\langle M, \preceq \rangle$ eine Quasi-Ordnung ist, so kann auf M eine Äquivalenz-Relation $\approx$ durch

$$x \approx y \stackrel{\text{def}}{\Longleftrightarrow} x \preceq y \wedge y \preceq x$$

definiert werden. Setzen wir die Ordnung $\preceq$ auf die von der Relation $\approx$ erzeugten Äquivalenz-Klassen fort, so kann gezeigt werden, dass diese Fortsetzung eine partielle Ordnung ist.

Es sei nun $\langle M, \preceq \rangle$ eine TQO. Dann ist das *Sortier-Problem* wie folgt definiert:

1. Gegeben ist eine Liste L von Elementen aus M.
2. Gesucht ist eine Liste S mit folgenden Eigenschaften:
   (a) S ist aufsteigend sortiert:

   $$\forall i \in \{1, \cdots, \#S - 1\} \colon S(i) \preceq S(i+1)$$

   Hier bezeichnen wir die Länge der Liste S mit $\#S$.
   (b) Die Elemente treten in L und S mit der selben Häufigkeit auf:

   $$\forall x \in M \colon \mathit{count}(x, L) = \mathit{count}(x, S).$$

   Dabei zählt die Funktion $\mathit{count}(x, L)$ wie oft das Element x in der Liste L auftritt:

   $$\mathit{count}(x, L) := \#\bigl\{i \in \{1, \cdots, \#L\} \mid L(i) = x\bigr\}.$$

In diesem Kapitel präsentieren wir verschiedene Algorithmen, die das Sortier-Problem lösen, die also zum Sortieren von Listen benutzt werden können. Wir stellen zunächst zwei Algorithmen vor, die sehr einfach zu implementieren sind, deren Effizienz aber zu wünschen übrig läßt. Im Anschluß daran präsentieren wir zwei effizientere Algorithmen, deren Implementierung allerdings aufwendiger ist.

5.1 Sortieren durch Einfügen

Wir stellen zunächst einen sehr einfachen Algorithmus vor, der als "*Sortieren durch Einfügen*" (engl. *insertion sort*) bezeichnet wird. Wir beschreiben den Algorithmus durch *Gleichungen*. Der Algorithmus arbeitet nach dem folgenden Schema:

1. Ist die zu sortierende Liste L leer, so wird als Ergebnis die leere Liste zurück gegeben:

   $$\texttt{sort}([]) = []$$

2. Andernfalls muß die Liste L die Form $[x] + R$ haben. Dann sortieren wir den Rest R und fügen das Element x in diese Liste so ein, dass die Liste sortiert bleibt.

$$\texttt{sort}\big([x] + R\big) = \texttt{insert}\big(x, \texttt{sort}(R)\big)$$

Das Einfügen eines Elements x in eine sortierte Liste S erfolgt nach dem folgenden Schema:

1. Falls die Liste S leer ist, ist das Ergebnis $[x]$:

$$\texttt{insert}(x, []) = [x].$$

2. Sonst hat S die Form $[y] + R$. Wir vergleichen x mit y.

   (a) Falls $x \preceq y$ ist, können wir x vorne an die Liste S anfügen:

   $$x \preceq y \rightarrow \texttt{insert}\big(x, [y] + R\big) = [x, y] + R.$$

   (b) Andernfalls fügen wir x rekursiv in die Liste R ein:

   $$\neg x \preceq y \rightarrow \texttt{insert}\big(x, [y] + R\big) = [y] + \texttt{insert}(x, R).$$

```
1   sort := procedure(l) {
2       match (l) {
3           case []   : return [];
4           case [x|r]: return insert(x, sort(r));
5       }
6   };
7
8   insert := procedure(x, l) {
9       match (l) {
10          case []   : return [ x ];
11          case [y|r]: if (x <= y) {
12                          return [x, y] + r;
13                      } else {
14                          return [y] + insert(x, r);
15                      }
16      }
17  };
18
19  l := [ rnd({1 .. 200}) : n in [1 .. 20] ];
20  print("l = $l$");
21  s := sort(l);
22  print("s = $s$");
```

Figure 5.1: Der Algorithmus "*Sortieren durch Einfügen*"

Der Algorithmus "*Sortieren durch Einfügen*" läßt sich leicht in SetlX umsetzen. Abbildung 5.1 zeigt das resultierende Programm.

1. Bei der Definition der Funktion $\texttt{sort}(l)$ benutzen wir das `match`-Konstrukt der Sprache `SetlX`. Das `match`-Konstrukt ist eine Weiterentwicklung des `switch`-Konstruktes. Zeile 3 greift, wenn die zu sortierende Liste leer ist. In Zeile 4 testen wir, ob die Liste l die Form

$$l = [x] + r$$

hat. Hier bezeichnet x das erste Element der Liste und r umfasst alle restlichen Elemente der Liste l. In SETLX dient die Syntax `[x|r]` dazu, eine List der Form $[x] + r$ zu erkennen.

2. Bei der Definition der Funktion $\texttt{insert}(x, l)$ verwenden wir ebenfalls ein `match`-Konstrukt. Die drei bedingten Gleichungen, welche die Funktion $\texttt{insert}()$ spezifizieren, lassen sich dadurch 1-zu-1 umsetzen.

3. In Zeile 19 definieren wir eine Liste der Länge 20, deren Elemente mit Hilfe der Funktion $\texttt{rnd}()$ zufällig aus der Menge

   $$\{n \in \mathbb{N} \mid 1 \leq n \wedge n \leq 200\}$$

   ausgewählt werden. Diese Liste wird dann ausgegeben und sortiert und die sortierte Liste wird zur Kontrolle ebenfalls ausgegeben.

5.1.1 Komplexität

Wir berechnen nun die Anzahl der Vergleichs-Operationen, die bei einem Aufruf von "`Sortieren durch Einfügen`" in Zeile 26 von Abbildung 5.1 auf Seite 48 durchgeführt werden. Dazu berechnen wir zunächst die Anzahl der Aufrufe von "`<=`", die bei einem Aufruf von $\texttt{insert}(x, l)$ im schlimmsten Fall bei einer Liste der Länge n durchgeführt werden. Wir bezeichnen diese Anzahl mit a_n. Der schlimmste Fall liegt dann vor, wenn x größer als jedes Element aus x ist, denn dann ist der Test "`x <= y`" in Zeile 12 immer falsch und wir haben solange einen rekursiven Aufruf der Funktion `insert`, solange die Restliste r nicht leer ist. Insgesamt haben wir dann

$$a_0 = 0 \quad \text{und} \quad a_{n+1} = a_n + 1.$$

Durch eine einfache Induktion läßt sich sofort sehen, dass diese Rekurrenz-Gleichung die Lösung

$$a_n = n$$

hat. Im schlimmsten Falle führt der Aufruf von $\texttt{insert}(x, l)$ bei einer Liste l mit n Elementen also n Vergleichs-Operationen durch, denn wir müssen dann x mit jedem der n Elemente aus l vergleichen.

Wir berechnen nun die Anzahl der Vergleichs-Operationen, die im schlimmsten Fall beim Aufruf von $\texttt{sort}(l)$ für eine Liste l der Länge n durchgeführt werden. Wir bezeichnen dieses Anzahl mit b_n. Offenbar gilt

$$b_0 = 0 \quad \text{und} \quad b_{n+1} = b_n + n, \tag{1}$$

denn für eine Liste $l = [x] + r$ der Länge $n + 1$ wird zunächst für die Liste r rekursiv die Funktion $\texttt{sort}(r)$ aufgerufen. Das liefert den Summanden b_n. Anschließend wird mit $\texttt{insert}(x, \texttt{sort}(\texttt{r}))$ das erste Element in diese Liste eingefügt. Wir hatten oben gefunden, dass dazu schlimmstenfalls n Vergleichs-Operationen notwendig sind, was den Summanden n erklärt.

Ersetzen wir in der Rekurrenz-Gleichung (1) n durch $n - 1$, so erhalten wir

$$b_n = b_{n-1} + (n - 1).$$

Diese Rekurrenz-Gleichung können wir lösen, wenn wir die rechte Seite mit dem *Teleskop-Verfahren* expandieren:

$$
\begin{aligned}
b_n &= b_{n-1} + (n-1) \\
&= b_{n-2} + (n-2) + (n-1) \\
&\vdots \\
&= b_{n-k} + (n-k) + \cdots + (n-1) \\
&\vdots \\
&= b_0 + 0 + 1 + \cdots + (n-1) \\
&= b_0 + \sum_{i=0}^{n-1} i \\
&= \frac{1}{2} \cdot n \cdot (n-1),
\end{aligned}
$$

denn $b_0 = 0$ und für die Summe der Zahlen von 0 bis $n-1$ läßt sich die Gleichung

$$\sum_{i=0}^{n-1} i = \frac{1}{2} \cdot n \cdot (n-1)$$

durch eine einfache Induktion nachweisen. Wir halten fest, dass für die Anzahl der Vergleiche im schlimmsten Fall folgendes gilt:

$$b_n = \frac{1}{2} \cdot n^2 - \frac{1}{2} \cdot n = \frac{1}{2} \cdot n^2 + \mathcal{O}(n)$$

Im schlimmsten Fall werden also $\mathcal{O}(n^2)$ Vergleiche durchgeführt, der Algorithmus "*Sortieren durch Einfügen*" erfordert einen quadratischen Aufwand. Sie können sich überlegen, dass der schlimmste Fall genau dann eintritt, wenn die zu sortierende Liste l absteigend sortiert ist, so dass die größten Elemente gerade am Anfang der Liste stehen.

Der günstigste Fall für den Algorithmus "*Sortieren durch Einfügen*" liegt dann vor, wenn die zu sortierende Liste bereits aufsteigend sortiert ist. Dann wird beim Aufruf von $\texttt{insert}(x, \texttt{sort}(r))$ nur ein einziger Vergleich durchgeführt. Die Rekurrenz-Gleichungen für die Anzahl der Vergleiche in $\texttt{sort}(L)$ lautet dann

$$b_0 = 0 \quad \text{und} \quad b_{n+1} = b_n + 1. \tag{1}$$

Die Lösung dieser Rekurrenz-Gleichung haben wir oben berechnet, sie lautet $b_n = n$. Im günstigsten Falle ist der Algorithmus "*Sortieren durch Einfügen*" also linear.

5.2 Sortieren durch Auswahl

Wir stellen als nächstes den Algorithmus "*Sortieren durch Auswahl*" (engl. *selection sort*) vor. Der Algorithmus kann wie folgt beschrieben werden:

1. Ist die zu sortierende Liste l leer, so wird als Ergebnis die leere Liste zurück gegeben:

   $$\texttt{sort}([]) = []$$

2. Andernfalls suchen wir in der Liste l das kleinste Element und entfernen dieses Element aus l. Wir sortieren rekursiv die resultierende Liste, die ja ein Element weniger enthält. Zum Schluß fügen wir das kleinste Element vorne an die sortierte Liste an:

   $$l \neq [] \rightarrow \texttt{sort}\big(l\big) = \big[\texttt{min}(l)\big] + \texttt{sort}\big(\texttt{delete}(\texttt{min}(l), l)\big).$$

Der Algorithmus um ein Auftreten eines Elements x aus einer Liste l zu entfernen, kann ebenfalls leicht rekursiv formuliert werden. Wir unterscheiden drei Fälle:

1. Falls l leer ist, gilt

   $$\texttt{delete}(x, []) = [].$$

2. Falls x gleich dem ersten Element der Liste l ist, gibt die Funktion den Rest r zurück:

   $$\texttt{delete}(x, [x] + r) = r.$$

3. Andernfalls wird das Element x rekursiv aus r entfernt:

   $$x \neq y \rightarrow \texttt{delete}(x, [y] + r) = [y] + \texttt{delete}(x, r).$$

Schließlich geben wir noch rekursive Gleichungen an um das Minimum einer Liste zu berechnen:

1. Das Minimum der leeren Liste ist größer als irgendein Element. Wir schreiben daher

   $$\texttt{min}([]) = \infty.$$

2. Um das Minimum der Liste $[x] + r$ zu berechnen, berechnen wir rekursiv das Minimum von r und benutzen die zweistellige Minimums-Funktion:

   $$\texttt{min}([x] + r) = \texttt{min}\big(x, \texttt{min}(r)\big).$$

   Dabei ist die zweistellige Minimums-Funktion wie folgt definiert:

   $$\texttt{min}(x, y) = \begin{cases} x & \text{falls } x \preceq y\,; \\ y & \text{sonst.} \end{cases}$$

Die Implementierung dieses Algorithmus in SetlX sehen Sie in Abbildung 5.2 auf Seite 51. Es war nicht notwendig, die Funktion *min*() zu implementieren, denn diese Funktion ist bereits vordefiniert. Die Funktion $\texttt{delete}(x, l)$ ist *defensiv* programmiert: Normalerweise sollte diese Funktion nur dann aufgerufen werden, wenn das zu entfernende Element x auch tatsächlich in der Liste l auftritt. Daher liegt in dem Fall, dass am Ende der Kette von rekursiven Aufrufen der Funktion $\texttt{delete}(x, l)$ die Liste l leer ist, ein Fehler vor, der ausgegeben wird.

```
1   sort := procedure(l) {
2       if (l == []) {
3           return [];
4       }
5       x := min(l);
6       return [x] + sort(delete(x,l));
7   };
8
9   delete := procedure(x, l) {
10      match (l) {
11          case []    : assert(false, "element $x$ not in list $l$");
12          case [y|r] : if (x == y) {
13                           return r;
14                       }
15                       return [y] + delete(x,r);
16      }
17  };
```

Figure 5.2: Der Algorithmus “*Sortieren durch Auswahl*”

5.2.1 Komplexität

Um die Komplexität von "*Sortieren durch Auswahl*" analysieren zu können, müssen wir zunächst die Anzahl der Vergleiche, die bei der Berechnung von $\mathtt{min}(l)$ durchgeführt werden, bestimmen. Es gilt

$$\mathtt{min}([x_1, x_2, x_3, \cdots, x_n]) = \mathtt{min}(x_1, \mathtt{min}(x_2, \mathtt{min}(x_3, \cdots \mathtt{min}(x_{n-1}, x_n) \cdots))).$$

Also wird bei der Berechnung von $\mathtt{min}(l)$ für eine Liste l der Länge n der Operator `min` insgesamt $(n-1)$-mal aufgerufen. Jeder Aufruf von `min` bedingt dann einen Aufruf des Vergleichs-Operators "$\preceq$". Bezeichnen wir die Anzahl der Vergleiche bein Aufruf von $\mathtt{sort}(l)$ für eine Liste der Länge l mit b_n, so finden wir also

$$b_0 = 0 \quad \text{und} \quad b_{n+1} = b_n + n,$$

denn um eine Liste mit $n+1$ Elementen zu sortieren, muss zunächst das Minimum dieser Liste berechnet werden. Dazu sind n Vergleiche notwendig. Dann wird das Minimum aus der Liste entfernt und die Rest-Liste, die ja nur noch n Elemente enthält, wird rekursiv sortiert. Das liefert den Beitrag b_n in der obigen Summe.

Bei der Berechnung der Komplexität von "*Sortieren durch Einfügen*" hatten wir die selbe Rekurrenz-Gleichung erhalten. Die Lösung dieser Rekurrenz-Gleichung lautet also

$$b_n = \tfrac{1}{2} \cdot n^2 - \tfrac{1}{2} \cdot n = \tfrac{1}{2} \cdot n^2 + \mathcal{O}(n).$$

Das sieht so aus, als ob die Anzahl der Vergleiche beim "*Sortieren durch Einfügen*" genau so wäre wie beim "*Sortieren durch Auswahl*". Das stimmt aber nicht. Bei "*Sortieren durch Einfügen*" ist die Anzahl der durchgeführten Vergleiche im schlimmsten Fall $\frac{1}{2} \cdot n \cdot (n-1)$, beim "*Sortieren durch Auswahl*" ist Anzahl der Vergleiche <u>immer</u> $\frac{1}{2} \cdot n \cdot (n-1)$. Der Grund ist, dass zur Berechnung des Minimums einer Liste mit n Elementen immer $n-1$ Vergleiche erforderlich sind. Um aber ein Element in eine Liste mit n Elementen einzufügen, sind im <u>Durchschnitt</u> nur etwa $\frac{1}{2}n$ Vergleiche erforderlich, denn im Schnitt sind etwa die Hälfte der Elemente kleiner als das einzufügende Element und daher müssen beim Einfügen in eine sortierte Liste der Länge n im Durchschnitt nur die ersten $\frac{n}{2}$ Elemente betrachtet werden. Daher ist die durchschnittliche Anzahl von Vergleichen beim "*Sortieren durch Einfügen*" $\frac{1}{4}n^2 + \mathcal{O}(n)$, also halb so groß wie beim "*Sortieren durch Auswahl*".

5.3 Sortieren durch Mischen

Wir stellen nun einen Algorithmus zum Sortieren vor, der für große Listen erheblich effizienter ist als die beiden bisher betrachteten Algorithmen. Der Algorithmus wird als "*Sortieren durch Mischen*" (engl. *merge sort*) bezeichnet und verläuft nach dem folgenden Schema:

1. Hat die zu sortierende Liste l weniger als zwei Elemente, so ist l bereits sortiert und es wird l zurück gegeben:

   $$\#l < 2 \rightarrow \mathtt{sort}(l) = l.$$

2. Ansonsten wird die Liste in zwei etwa gleich große Listen zerlegt. Diese Listen werden rekursiv sortiert und anschließend so gemischt, dass das Ergebnis sortiert ist:

   $$\#l \geq 2 \rightarrow \mathtt{sort}(l) = \mathtt{merge}\Big(\mathtt{sort}\big(\mathtt{split}_1(l)\big), \mathtt{sort}\big(\mathtt{split}_2(l)\big)\Big)$$

   Hier bezeichnen $\mathtt{split}_1$ und $\mathtt{split}_2$ die Funktionen, die eine Liste in zwei Teil-Listen zerlegen und `merge` ist eine Funktion, die zwei sortierte Listen so mischt, dass das Ergebnis wieder sortiert ist.

Abbildung 5.3 zeigt die Umsetzung dieser sortierten Gleichungen in ein SetlX-Programm, das eine verkettete Liste sortiert. Die beiden Funktionen $\texttt{split}_1$ und $\texttt{split}_2$ haben wir dabei zu einer Funktion `split` zusammen gefaßt, die zwei Listen zurück liefert. Ein Aufruf der Form

$\texttt{split}(l)$

liefert als Ergebnis ein Paar von Listen

$[l_1, l_2]$,

wobei die Elemente der Liste l gleichmäßig auf die beiden Listen l_1 und l_2 verteilt werden. Damit können wir die Details des SetlX-Programms in Abbildung 5.3 verstehen:

```
1   sort := procedure(l) {
2       if (#l < 2) {
3           return l;
4       }
5       [ l1, l2 ] := split(l);
6       return merge(sort(l1), sort(l2));
7   };
8
9   split := procedure(l) {
10      match (l) {
11          case []      : return [ [], [] ];
12          case [x]     : return [ [x], [] ];
13          case [x,y|r] : [r1,r2] := split(r);
14                         return [ [x] + r1, [y] + r2 ];
15      }
16  };
17
18  merge := procedure(l1, l2) {
19      if (l1 == []) {
20          return l2;
21      }
22      if (l2 == []) {
23          return l1;
24      }
25      x := l1[1];
26      y := l2[1];
27      if (x < y) {
28          return [x] + merge(l1[2..], l2);
29      } else {
30          return [y] + merge(l1, l2[2..]);
31      }
32  };
```

Figure 5.3: Implementierung des Algorithmus "*Sortieren durch Mischen*".

1. Wenn die zu sortierende Liste aus weniger als zwei Elementen besteht, dann ist diese Liste bereits sortiert und wir können diese Liste unverändert zurück geben.

2. Der Aufruf von `split` verteilt die Elemente der zu sortierenden Liste auf die

beiden Listen `l1` und `l2`.

3. Diese Listen werden rekursiv sortiert und anschließend durch den Aufruf von `merge` zu einer sortierten Liste zusammen gefaßt.

Als nächstes überlegen wir uns, wie wir die Funktion `split` durch Gleichungen spezifizieren können.

1. Falls l leer ist, produziert $\texttt{split}(l)$ zwei leere Listen:

   $$\texttt{split}(\texttt{[]}) = [\texttt{[]}, \texttt{[]}].$$

2. Falls l genau ein Element besitzt, stecken wir dieses in die erste Liste:

   $$\texttt{split}([x]) = [\texttt{[}x\texttt{]}, \texttt{[]}].$$

3. Sonst hat l die Form $[x, y] + r$. Dann spalten wir rekursiv r in zwei Listen auf. Vor die erste Liste fügen wir x an, vor die zweite Liste fügen wir y an:

   $$\texttt{split}(r) = [r_1, r_2] \rightarrow \texttt{split}\big([x, y] + r\big) = \big[[x] + r_1, [y] + r_2\big].$$

Als letztes spezifizieren wir, wie zwei sortierte Listen l_1 und l_2 so gemischt werden können, dass das Ergebnis anschließend wieder sortiert ist.

1. Falls die Liste l_1 leer ist, ist das Ergebnis l_2:

   $$\texttt{merge}([], l_2) = l_2.$$

2. Falls die Liste l_2 leer ist, ist das Ergebnis l_1:

   $$\texttt{merge}(l_1, []) = l_1.$$

3. Andernfalls hat l_1 die Form $[x]+r_1$ und l_2 hat die Gestalt $[y]+r_2$. Dann führen wir eine Fallunterscheidung nach der relativen Größe von x und y durch:

   (a) $x \preceq y$.
   In diesem Fall mischen wir r_1 und l_2 und setzen x an den Anfang dieser Liste:

   $$x \preceq y \rightarrow \texttt{merge}\big([x] + r_1, [y] + r_2\big) = [x] + \texttt{merge}\big(r_1, [y] + r_2\big).$$

   (b) $\neg x \preceq y$.
   In diesem Fall mischen wir l_1 und r_2 und setzen y an den Anfang dieser Liste:

   $$\neg x \preceq y \rightarrow \texttt{merge}\big([x] + r_1, [y] + r_2\big) = [y] + \texttt{merge}\big([x] + r_1, r_2\big).$$

1. Falls eine der beiden Listen leer ist, so geben wir als Ergebnis die andere Liste zurück.

2. Wenn der Kontrollfluß in Zeile 7 ankommt, dann wissen wir, dass beide Listen nicht leer sind. Wir holen uns jeweils das erste Element der beiden Listen und speichern diese in den Variablen x und y ab. Da wir diese Elemente aber mit Hilfe der Methode `getFirst` bekommen, bleiben diese Elemente zunächst Bestandteil der beiden Listen.

3. Anschließend prüfen wir, welche der beiden Variablen die kleinere ist.

   (a) Falls x kleiner als y ist, so entfernen wir x aus der ersten Liste und mischen rekursiv die verkürzte erste Liste mit der zweiten Liste. Anschließen fügen wir x an den Anfang der Ergebnis-Liste ein.

   (b) Andernfalls entfernen wir y aus der zweiten Liste und mischen rekursiv die erste Liste mit der verkürzten zweiten Liste und fügen dann y am Anfang der beim rekursiven Aufruf erhaltenen Liste ein.

5.3.1 Komplexität

Wir wollen wieder berechnen, wieviele Vergleiche beim Sortieren einer Liste mit n Elementen durchgeführt werden. Dazu analysieren wir zunächst, wieviele Vergleiche zum Mischen zweier Listen l_1 und l_2 benötigt werden. Wir definieren eine Funktion

$$\texttt{cmpCount} : \mathit{List}(M) \times \mathit{List}(M) \rightarrow \mathbb{N}$$

so dass für zwei Listen l_1 und l_2 der Term $\texttt{cmpCount}(l_1, l_2)$ die Anzahl Vergleiche angibt, die bei Berechnung von $\texttt{merge}(l_1, l_2)$ erforderlich sind. Wir behaupten, dass für beliebige Listen l_1 und l_2

$$\texttt{cmpCount}(l_1, l_2) \leq \#l_1 + \#l_2$$

gilt. Für eine Liste l bezeichnet dabei $\#l$ die Anzahl der Elemente der Liste. Wir führen den Beweis durch Induktion nach der Summe $\#l_1 + \#l_2$.

I.A.: $\#l_1 + \#l_2 = 0$.

Dann müssen l_1 und l_2 leer sein und somit ist beim Aufruf von $\texttt{merge}(l_1, l_2)$ kein Vergleich erforderlich. Also gilt

$$\texttt{cmpCount}(l_1, l_2) = 0 \leq 0 = \#l_1 + \#l_2.$$

I.S.: $\#l_1 + \#l_2 = n + 1$.

Falls entweder l_1 oder l_2 leer ist, so ist kein Vergleich erforderlich und wir haben

$$\texttt{cmpCount}(l_1, l_2) = 0 \leq \#l_1 + \#l_2.$$

Wir nehmen also an, dass gilt:

$$l_1 = [x] + r_1 \quad \text{und} \quad l_2 = [y] + r_2.$$

Wir führen eine Fallunterscheidung bezüglich der relativen Größe von x und y durch.

(a) $x \preceq y$. Dann gilt

$$\texttt{merge}\big([x] + r_1, [y] + r_2\big) = [x] + \texttt{merge}\big(r_1, [y] + r_2\big).$$

Also haben wir:

$$\texttt{cmpCount}(l_1, l_2) = 1 + \texttt{cmpCount}(r_1, l_2) \stackrel{IV}{\leq} 1 + \#r_1 + \#l_2 = \#l_1 + \#l_2.$$

(b) $\neg x \preceq y$. Dieser Fall ist völlig analog zum 1. Fall. $\square$

Exercise 14: Überlegen Sie, wie die Listen l_1 und l_2 aussehen müssen, damit der Wert von

$$\texttt{cmpCount}(l_1, l_2)$$

maximal wird und geben Sie an, welchen Wert $\texttt{cmpCount}(l_1, l_2)$ in diesem Fal annimmt.

Wir wollen nun die Komplexität des Merge-Sort-Algorithmus im schlechtesten Fall berechnen und bezeichnen dazu die Anzahl der Vergleiche, die beim Aufruf von $\texttt{sort}(l)$ für eine Liste l der Länge n schlimmstenfalls durchgeführt werden müssen mit a_n. Zur Vereinfachung nehmen wir an, dass n die Form

$$n = 2^k \qquad \text{für ein } k \in \mathbb{N}$$

hat und definieren $b_k = a_n = a_{2^k}$. Zunächst berechnen wir den Anfangs-Wert, es

gilt

$$b_0 = a_{2^0} = a_1 = 0,$$

denn bei einer Liste der Länge 1 findet noch kein Vergleich statt. Im rekursiven Fall wird zur Berechnung von `sort`(l) die Liste l zunächst durch `split` in zwei gleich große Listen geteilt, die dann rekursiv sortiert werden. Anschließend werden die sortierten Listen gemischt. Das liefert für das Sortieren einer Liste der Länge 2^{k+1} die Rekurrenz-Gleichung

$$b_{k+1} = 2 \cdot b_k + 2^{k+1}, \tag{1}$$

denn das Mischen der beiden halb so großen Listen kostet schlimmstenfalls $2^k + 2^k = 2^{k+1}$ Vergleiche und das rekursive Sortieren der beiden Teil-Listen kostet insgesamt $2 \cdot b_k$ Vergleiche.

Um diese Rekurrenz-Gleichung zu lösen, führen wir in (1) die Substitution $k \mapsto k + 1$ durch und erhalten

$$b_{k+2} = 2 \cdot b_{k+1} + 2^{k+2}. \tag{2}$$

Wir multiplizieren Gleichung (1) mit dem Faktor 2 und subtrahieren die erhaltene Gleichung von Gleichung (2). Dann erhalten wir

$$b_{k+2} - 2 \cdot b_{k+1} = 2 \cdot b_{k+1} - 4 \cdot b_k. \tag{3}$$

Diese Gleichung vereinfachen wir zu

$$b_{k+2} = 4 \cdot b_{k+1} - 4 \cdot b_k. \tag{4}$$

Diese Gleichung ist eine homogene lineare Rekurrenz-Gleichung 2. Ordnung. Das charakteristische Polynom der zugehörigen homogenen Rekurrenz-Gleichung lautet

$$\chi(x) = x^2 - 4 * x + 4 = (x - 2)^2.$$

Weil das charakteristische Polynom an der Stelle $x = 2$ eine doppelte Null-Stelle hat, lautet die allgemeine Lösung

$$b_k = \alpha \cdot 2^k + \beta \cdot k \cdot 2^k. \tag{5}$$

Setzen wir hier den Wert für $k = 0$ ein, so erhalten wir

$$0 = \alpha.$$

Aus (1) erhalten wir den Wert $b_1 = 2 \cdot b_0 + 2^1 = 2$. Setzen wir also in Gleichung (5) für k den Wert 1 ein, so finden wir

$$2 = 0 \cdot 2^1 + \beta \cdot 1 \cdot 2^1,$$

also muß $\beta = 1$ gelten. Damit lautet die Lösung

$$b_k = k \cdot 2^k.$$

Aus $n = 2^k$ folgt $k = \log_2(n)$ und daher gilt für a_n

$$a_n = n \cdot \log_2(n).$$

Wir sehen also, dass beim "*Sortieren durch Mischen*" für große Listen wesentlich weniger Vergleiche durchgeführt werden müssen, als dies bei den beiden anderen Algorithmen der Fall ist.

Zur Vereinfachung haben wir bei der obigen Rechnung nur eine obere Abschätzung der Anzahl der Vergleiche durchgeführt. Eine exakte Rechnung zeigt, dass im schlimmsten Fall

$$n \cdot \log_2(n) - n + 1$$

Vergleiche beim "*Sortieren durch Mischen*" einer nicht-leeren Liste der Länge n

durchgeführt werden müssen.

5.3.2 Eine feldbasierte Implementierung

Unsere bisherigen Implementierungen der Sortier-Algorithmen waren bezüglich des Speicherverbrauchs sehr ineffizient, denn wir haben dort nach Belieben neue Listen erzeugt. In der Realität ist es so, dass die zu sortierende Liste als Feld gegeben ist. Dann ist es das Ziel, möglichst mit diesem Feld auszukommen und beim Sortieren keinen weiteren Speicher zu allokieren. Bei dem Algorithmen "*Sortieren durch Mischen*" ist diese Ziel nur sehr schwer erreichbar, aber wenn wir uns die Benutzung eines Hilfsfeldes gestatten, das die gleiche Länge wie das zu sortierende Feld hat, dann läßt sich der Algorithmus so implementieren, dass bis auf das Hilfsfeld nur noch etwas Speicher auf dem Stack benötigt wird. Abbildung 5.4 auf Seite 58 zeigt eine feldbasierte Implementierung des Algorithmus "*Sortieren durch Mischen*".

In der Methode `sort`(l) haben wir mit dem Schlüsselwort "`rw`" den Parameter l als einen *read-write Parameter* spezifiziert. Dies bewirkt, dass Änderungen des Parameters l auch außerhalb der Methode `sort`() sichtbar sind. Daher gibt der Aufruf `sort`(l) auch kein Ergebnis zurück. Statt dessen wird die als Argument übergebene Liste l sortiert.

Der Algorithmus "*Sortieren durch Mischen*" wird in der Methode `mergeSort` implementiert. Diese Methode erhält die Argumente `start` und `end`, die den Bereich der Liste eingrenzen, der zu sortieren ist: Der Aufruf `mergeSort`(l, *start*, *end*, a) sortiert nur den Bereich

$$l\,[\mathit{start}, \cdots, \mathit{end} - 1].$$

Der Parameter a bezeichnet dabei eine Liste, die als Hilfsfeld benutzt wird. Diese Liste muss die selbe Länge haben wie die Liste l. Die feldbasierte Implementierung weicht von der listenbasierten Implementierung ab, da wir keine Funktion `split` mehr benötigen, um die Liste aufzuspalten. Statt dessen berechnen wir die Mitte des zu sortierenden Teils der Liste l mit der Formel

```
middle = (start + end) \ 2;
```

und spalten das Feld dann an dem Index `middle` in zwei etwa gleich große Teile, die wir in den Zeilen 11 und 12 rekursiv sortieren. Beachten Sie, dass wir hier bei der Division den Operator "\" benutzen, der eine ganzzahlige Division durchführt.

Nachdem wir die Liste in zwei Teile aufgeteilt haben, sortieren wir die beiden Teile rekursiv und rufen dann in Zeile 13 die Methode `merge` aus, welche die beiden sortierten Felder zu einem sortierten Feld zusammenfaßt. Diese Methode ist in den Zeilen 16 — 36 implementiert. Die Methode erhält 5 Argumente:

1. Der erste Parameter l spezifiziert die zu sortierende Liste.
2. Die Parameter `start`, `middle` und `end` spezifizieren die beiden Teilfelder, die zu mischen sind. Das erste Teilfeld besteht aus den Elementen

   $$l\,[\mathit{start}, \cdots, \mathit{middle} - 1],$$

   das zweite Teilfeld besteht aus den Elementen

   $$l\,[\mathit{middle}, \cdots, \mathit{end} - 1].$$

3. Der letzte Parameter a bezeichnet ein Hilfsfeld, das die selbe Größe hat wie die Liste l.

Der Aufruf setzt voraus, dass die beiden Teilfelder bereits sortiert sind. Das Mischen funktioniert dann wie folgt.

```
sort := procedure(rw l) {
    a := l;
    mergeSort(l, 1, #l + 1, a);
};

mergeSort := procedure(rw l, start, end, rw a) {
    if (end - start < 2) {
        return;   // nothing to do if there is at most one element
    }
    middle := (start + end) \ 2;
    mergeSort(l, start,  middle, a);
    mergeSort(l, middle, end   , a);
    merge(l, start, middle, end, a);
};

merge := procedure(rw l, start, middle, end, rw a) {
    for (i in [start .. end-1]) {
        a[i] := l[i];
    }
    idx1 := start;
    idx2 := middle;
    i    := start;
    while (idx1 < middle && idx2 < end) {
        if (a[idx1] <= a[idx2]) {
            l[i] := a[idx1]; i += 1; idx1 += 1;
        } else {
            l[i] := a[idx2]; i += 1; idx2 += 1;
        }
    }
    while (idx1 < middle) {
        l[i] := a[idx1]; i += 1; idx1 += 1;
    }
    while (idx2 < end) {
        l[i] := a[idx2]; i += 1; idx2 += 1;
    }
};
```

Figure 5.4: Eine feldbasierte Implementierung des Algorithmus "*Sortieren durch Mischen*".

1. Zunächst werden die Daten aus den beiden zu sortierenden Teilfelder in Zeile 14 in das Hilfs-Feld a kopiert.
2. Anschließend definieren wir drei Indizes:
   (a) `idx1` zeigt auf das nächste Element im ersten Teilfeld von a.
   (b) `idx2` zeigt auf das nächste Element im zweiten Teilfeld von a.
   (c) `i` gibt die Position im Ergebnis-Feld l an, an die das nächste Element geschrieben wird.
3. Solange weder das erste noch das zweite Teilfeld des Feldes a vollständig abgearbeitet ist, vergleichen wir in Zeile 30 die Elemente aus den beiden Teil-

feldern und schreiben das kleinere von beiden an die Stelle, auf die der Index i zeigt.

4. Falls in der `while`-Schleife eines der beiden Felder vor dem anderen erschöpft ist, müssen wir anschließend die restlichen Elemente des verbleibenden Teilfeldes in das Ergebnis-Feld kopieren. Die Schleife in Zeile 30 — 32 wird aktiv, wenn das zweite Teilfeld zuerst erschöpft wird. Dann werden die verbleibenden Elemente des ersten Teilfeldes in das Feld l kopiert. Ist umgekehrt das erste Teilfeld zuerst erschöpft, dann werden in Zeile 33 — 35 die verbleibenden Elemente des zweiten Teilfeldes in das Feld l kopiert.

5.3.3 Eine iterative Implementierung von *Sortieren durch Mischen*

```
1   sort := procedure(rw l) {
2       a := l;
3       mergeSort(l, a);
4   };
5
6   mergeSort := procedure(rw l, rw a) {
7       n := 1;
8       while (n < #l) {
9           k := 0;
10          while (n * (k + 1) + 1 <= #l) {
11              merge(l, n*k + 1, n*(k+1) + 1, min([n*(k+2), #l]) + 1, a);
12              k += 2;
13          }
14          n *= 2;
15      }
16  };
```

Figure 5.5: Eine nicht-rekursive Implementierung des Merge-Sort-Algorithmus

Die in Abbildung 5.4 gezeigte Implementierung des Merge-Sort-Algorithmus ist rekursiv. Die Effizienz einer rekursiven Implementierung ist in der Regel schlechter als die Effizienz einer sequentiellen Implementierung. Der Grund ist, dass der Aufruf einer rekursiven Funktion relativ viel Zeit kostet, denn beim Aufruf einer rekursiven Funktion müssen die lokalen Variablen der Funktion zusammen mit den Argumenten auf den Stack gelegt werden. Wir zeigen daher, wie sich der Merge-Sort-Algorithmus iterativ implementieren läßt. Abbildung 5.5 auf Seite 59 zeigt eine solche Implementierung. Statt der rekursiven Aufrufe haben wir hier zwei ineinander geschachtelte `while`-Schleifen. Die Idee ist, dass wir zunächst die ursprüngliche Liste l in Listen der Länge 1 aufspalten, die offenbar bereits sortiert sind, denn jede Liste der Länge 1 ist sortiert. Anschließend mischen wir immer zwei benachbarte Listen der Länge 1 mit Hilfe der Funktion `merge`() und erhalten dann sortierte Listen der Länge 2. Diesen Schritt wiederholen wir nun für die Listen der Länge 2: Wir mischen jeweils zwei benachbarte Listen der Länge 2, die ja nun sortiert sind, zu einer sortierten Liste der Länge 4. Aus diesen Listen erzeugen wir in analoger Weise sortierte Listen der Länge 8, 16, $\cdots$. Das Verfahren bricht ab, wenn die gesamte Liste sortiert ist.

Die genaue Arbeitsweise der Implementierung wird deutlich, wenn wir die Invarianten der beiden `while`-Schleifen formulieren. Die Invariante der äußeren Schleife

besagt, dass alle Teil-Felder der Liste l, welche eine Länge von n haben und deren erstes Element einen Index der Form $n \cdot k + 1$ hat, bereits sortiert sind. Diese Teilfelder haben also die Form

$$l[n \cdot k + 1, \cdots, n \cdot (k + 1)].$$

Aufgabe der Schleife ist es dann, mit Hilfe der Funktion `merge()` jeweils zwei aufeinander folgende Teilfelder dieser Form zu einem sortierten Teilfeld der doppelten Länge zusammenzufassen.

In dem Ausdruck $l[n \cdot k + 1, \cdots, n \cdot (k + 1)]$ ist k eine natürliche Zahl, mit der die einzelnen Teilfelder durchnummeriert werden. Ein solches Teilfeld hat also die Länge n, das erste Element ist

$$l[n \cdot k + 1]$$

und das letzte Element eines solchen Teilfeldes ist

$$l[n \cdot (k + 1)].$$

Für das erste Teilfeld hat der Index k natürlich den Wert 0, so dass dieses Teilfeld die Form

$$l[1, \cdots, n]$$

hat, es besteht als aus den ersten n Elementen der Liste l. Das zweite Teilfeld ist dann

$$l[n + 1, \cdots, 2 \cdot n]$$

und so geht es bis zum Ende der Liste l weiter. Möglicherweise hat das letzte Teilfeld eine Länge, die kleiner als n ist. Dieser Fall tritt dann auf, wenn ein Feld der Länge n nicht mehr in die Liste l hinein paßt. Daher nehmen wir für das dritte Argument der Methode *merge* das Minimum der beiden Zahlen $n \cdot (k + 2)$ und $\#l$.

Exercise 15: Geben Sie die Invariaten der beiden `while`-Schleifen explizit als prädikatenlogische Formeln an!

5.4 Der *Quick-Sort*-Algorithmus

Der von Tony Hoare entdeckte "*Quick-Sort*-Algorithmus" [Hoa61] funktioniert nach folgendem Schema:

1. Ist die zu sortierende Liste L leer, so wird L zurück gegeben:

   $$\texttt{sort}([]) = [].$$

2. Sonst hat L die Form $L = [x] + R$. Dann verteilen wir die Elemente von R so auf zwei Listen S und B, dass S alle Elemente von R enthält, die kleiner-gleich x sind, während B die restlichen Elemente von R enthält. Wir implementieren die Berechnung der beiden Listen über eine Funktion `partition`, die das Paar von Listen S und B erzeugt:

   $$\mathit{partition}(x, R) = \langle S, B \rangle.$$

   Hierbei gilt dann

   (a) Die Listen S und B enthalten zusammen genau die Elemente der Liste R

   $$\forall y \in R : \mathit{count}(y, S) + \mathit{count}(y, B) = \mathit{count}(y, R)$$

(b) Alle Elemente aus S sind kleiner-gleich x, die Elemente aus B sind größer als x:

$$\forall y \in S : y \preceq x \quad \text{und} \quad \forall y \in B : x \prec y.$$

Formal können wir die Funktion *partition*() durch die folgenden Gleichungen beschreiben:

(a) $\mathit{partition}(x, []) = \big\langle [], [] \big\rangle$,

(b) $\quad y \preceq x \wedge \mathit{partition}(x, R) = \big\langle S, B \big\rangle \rightarrow \mathit{partition}(x, [y] {+} R) = \big\langle [y] {+} S,\ B \big\rangle$

(c) $\neg\,(y \preceq x) \wedge \mathit{partition}(x, R) = \big\langle S, B \big\rangle \rightarrow \mathit{partition}(x, [y] + R) = \big\langle S,\ [y] + B \big\rangle$,

Anschließend sortieren wir die Listen S und B rekursiv. An die sortierte Liste S hängen wir dann das Element x und darauf folgt die sortierte Liste B. Insgesamt wird dieser Algorithmus durch die folgende Gleichung beschrieben:

$$\mathit{partition}(x, R) = \langle S, B \rangle \rightarrow \mathit{sort}([x] + R) = \mathit{sort}(S) + [x] + \mathit{sort}(B).$$

```
1   sort := procedure(l) {
2       match (l) {
3           case []   : return [];
4           case [x|r]: [s,b] := partition(x, r);
5                       return sort(s) + [x] + sort(b);
6       }
7   };
8
9   partition := procedure(p, l) {
10      match (l) {
11          case []   : return [ [], [] ];
12          case [x|r]: [ r1, r2 ] := partition(p, r);
13                      if (x <= p) {
14                          return [ [x] + r1, r2 ];
15                      }
16                      return [ r1, [x] + r2 ];
17      }
18  };
```

Figure 5.6: Der *Quick-Sort*-Algorithmus.

Abbildung 5.6 zeigt die Umsetzung dieser Überlegung in einem SetlX-Programm. Die Funktion *sort*(l) ist für eine Liste l durch eine Fall-Unterscheidung implementiert.

1. Falls die zu sortierende Liste l leer ist, wird als Ergebnis die leere Liste zurück gegeben.

2. Andernfalls können wir die Liste in der Form $[x|r]$ aufspalten. Hier ist x das erste Element der Liste l, während die Liste r alle restlichen Elemente der Liste l enthält. Mit Hilfe der Funktion *partition*() wird nun die Restliste r in zwei Listen s und b aufgespalten. Die Liste s enthält alle die Elemente aus der Liste r, die kleiner oder gleich x sind, während b die Liste alle der Elemente der Liste r ist, die größer als x sind. Anschließend werden die beiden Teillisten s und b rekursiv sortiert und in der richtigen Reihenfolge zum Ergebnis zusammengesetzt.

Die Funktion *partition*(p,l) bekommt als erstes Argument eine Zahl p und als zweites Argument eine Liste von Zahlen l. Die Funktion berechnet dann ein Paar $[s,b]$, das aus zwei Listen s und b besteht. Die Zahlen in der Liste s sind kleiner oder gleich dem *Pivot-Element* p, während die Elemente in der Liste b alle größer als p sind. Auch diese Funktion ist durch eine Fallunterscheidung definiert:

1. Falls die Liste l, deren Elemente auf zwei Listen zu verteilen sind, leer ist, dann werden als Ergebnis zwei leere Listen zurück gegeben.

2. Anderfalls hat l die Form $[x|r]$. Dann wird zunächst die Liste r rekursiv in zwei Teillisten r_1 und r_2 aufgeteilt. Jetzt muss nur noch durch einen Vergleich von x mit dem Pivot-Element p entschieden werden, in welche der beiden Listen r_1 oder r_2 das Element x einzuordnen ist.

5.4.1 Komplexität

Als nächstes analysieren wir die Komplexität von *Quick-Sort*. Dazu untersuchen wir wieder die Zahl der Vergleiche, die beim Aufruf von *sort*(L) für eine Liste L mit n Elementen durchgeführt werden. Wir betrachten zunächst die Anzahl der Vergleiche, die wir bei einem Aufruf der Form

$$\textit{partition}(x, L, S, B)$$

für eine Liste L mit n Elementen durchführen müssen. Da wir jedes der n Elemente der Liste L mit x vergleichen müssen, ist klar, dass dafür insgesamt n Vergleiche erforderlich sind.

Komplexität im schlechtesten Fall

Wir berechnen als nächstes die Anzahl a_n von Vergleichen, die wir im schlimmsten Fall beim Aufruf von *sort*(L) für eine Liste L der Länge n durchführen müssen. Der schlimmste Fall tritt beispielsweise dann ein, wenn die Liste `small` leer ist und die Liste `big` folglich die Länge $n-1$ hat. Für die Anzahl a_n der Vergleiche gilt in diesem Fall

$$a_n = a_{n-1} + n - 1.$$

Der Term $n-1$ rührt von den $n-1$ Vergleichen, die beim Aufruf von *partition*(x,R) in Zeile 6 von Abbildung 5.6 durchgeführt werden und der Term a_{n-1} erfaßt die Anzahl der Vergleiche, die beim rekursiven Aufruf von *sort*(L_2) benötigt werden.

Die Anfangs-Bedingung für die Rekurrenz-Gleichung lautet $a_0 = 0$, denn beim Sortieren einer leeren Liste sind keine Vergleiche notwendig. Damit läßt sich die obige Rekurrenz-Gleichung mit dem *Teleskop-Verfahren* lösen:

$$\begin{array}{lcl}
a_n & = & a_{n-1} + (n-1) \\
& = & a_{n-2} + (n-2) + (n-1) \\
& = & a_{n-3} + (n-3) + (n-2) + (n-1) \\
& = & \vdots \\
& = & a_0 + 0 + 1 + 2 + \cdots + (n-2) + (n-1) \\
& = & 0 + 0 + 1 + 2 + \cdots + (n-2) + (n-1) \\
& = & \sum\limits_{i=0}^{n-1} i = \frac{1}{2}n \cdot (n-1) = \frac{1}{2}n^2 - \frac{1}{2}n.
\end{array}$$

Damit ist a_n in diesem Fall genauso groß wie im schlimmsten Fall des Algorithmus' *Sortieren durch Einfügen*. Es ist leicht zu sehen, dass der schlechteste Fall dann eintritt, wenn die zu sortierende Liste L bereits sortiert ist. Es existieren

Verbesserungen des *Quick-Sort*-Algorithmus, für die der schlechteste Fall sehr unwahrscheinlich ist und insbesondere nicht bei sortierten Listen eintritt. Wir gehen später näher darauf ein.

Durchschnittliche Komplexität

Der Algorithmus *Quick-Sort* würde seinen Namen zu Unrecht tragen, wenn er im *Durchschnitt* ein Komplexität der Form $\mathcal{O}(n^2)$ hätte. Wir analysieren nun die *durchschnittliche* Anzahl von Vergleichen d_n, die wir beim Sortieren einer Liste L mit n Elementen erwarten müssen. Im Allgemeinen gilt: Ist L eine Liste mit $n+1$ Elementen, so ist die Zahl der Elemente der Liste `small`, die in Zeile 19 von Abbildung 5.6 berechnet wird, ein Element der Menge $\{0, 1, 2, \cdots, n\}$. Hat die Liste `small` insgesamt i Elemente, so enthält die Liste `big` die restlichen $n-i$ Elemente. Gilt $\#\texttt{small} = i$, so werden zum rekursiven Sortieren von `small` und `big` durchschnittlich

$$d_i + d_{n-i}$$

Vergleiche durchgeführt. Bilden wir den Durchschnitt dieses Wertes für alle $i \in \{0, 1, \cdots, n\}$, so erhalten wir für d_{n+1} die Rekurrenz-Gleichung

$$d_{n+1} = n + \frac{1}{n+1}\sum_{i=0}^{n}(d_i + d_{n-i}) \tag{5.1}$$

Der Term n stellt die Vergleiche in Rechnung, die beim Aufruf von

```
partition(pivot, list, small, big)
```

durchgeführt werden. Um die Rekurrenz-Gleichung (1) zu vereinfachen, bemerken wir zunächst, dass für beliebige Funktionen $f : \mathbb{N} \to \mathbb{N}$ folgendes gilt:

$$\begin{aligned}
\sum_{i=0}^{n} f(n-i) &= f(n) + f(n-1) + \cdots + f(1) + f(0) && (5.2)\\
&= f(0) + f(1) + \cdots + f(n-1) + f(n) && (5.3)\\
&= \sum_{i=0}^{n} f(i) && (5.4)
\end{aligned}$$

Damit vereinfacht sich die Rekurrenz-Gleichung (5.1) zu

$$d_{n+1} = n + \frac{2}{n+1} \cdot \sum_{i=0}^{n} d_i. \tag{5.5}$$

Um diese Rekurrenz-Gleichung lösen zu können, substituieren wir $n \mapsto n+1$ und erhalten

$$d_{n+2} = n + 1 + \frac{2}{n+2} \cdot \sum_{i=0}^{n+1} d_i. \tag{5.6}$$

Wir multiplizieren nun Gleichung (5.6) mit $n+2$ und Gleichung (5.5) mit $n+1$ und erhalten

$$\begin{aligned}
(n+2) \cdot d_{n+2} &= (n+2) \cdot (n+1) + 2 \cdot \sum_{i=0}^{n+1} d_i \qquad \text{und} && (5.7)\\
(n+1) \cdot d_{n+1} &= (n+1) \cdot n + 2 \cdot \sum_{i=0}^{n} d_i. && (5.8)
\end{aligned}$$

Wir bilden die Differenz der Gleichungen (5.7) und (5.8) und beachten, dass sich die Summationen bis auf den Term $2 \cdot d_{n+1}$ gerade gegenseitig aufheben. Dann erhalten wir

$$(n+2) \cdot d_{n+2} - (n+1) \cdot d_{n+1} = (n+2) \cdot (n+1) - (n+1) \cdot n + 2 \cdot d_{n+1} \tag{5.9}$$

Diese Gleichung vereinfachen wir zu

$$(n+2) \cdot d_{n+2} = (n+3) \cdot d_{n+1} + 2 \cdot (n+1). \tag{5.10}$$

Einer genialen Eingebung folgend teilen wir diese Gleichung durch $(n+2) \cdot (n+3)$ und erhalten

$$\frac{1}{n+3} \cdot d_{n+2} = \frac{1}{n+2} \cdot d_{n+1} + \frac{2 \cdot (n+1)}{(n+2) \cdot (n+3)}. \tag{5.11}$$

Als nächstes bilden wir die Partialbruch-Zerlegung von dem Bruch

$$\frac{2 \cdot (n+1)}{(n+2) \cdot (n+3)}.$$

Dazu machen wir den Ansatz

$$\frac{2 \cdot (n+1)}{(n+2) \cdot (n+3)} = \frac{\alpha}{n+2} + \frac{\beta}{n+3}.$$

Wir multiplizieren diese Gleichung mit dem Hauptnenner und erhalten

$$2 \cdot n + 2 = \alpha \cdot (n+3) + \beta \cdot (n+2),$$

was sich zu

$$2 \cdot n + 2 = (\alpha + \beta) \cdot n + 3 \cdot \alpha + 2 \cdot \beta$$

vereinfacht. Ein Koeffizientenvergleich liefert dann das lineare Gleichungs-System:

$$\begin{aligned} 2 &= \alpha + \beta \\ 2 &= 3 \cdot \alpha + 2 \cdot \beta \end{aligned}$$

Ziehen wir die erste Gleichung zweimal von der zweiten Gleichung ab, so erhalten wir $\alpha = -2$ und Einsetzen in die erste Gleichung liefert $\beta = 4$. Damit können wir die Gleichung (5.11) als

$$\frac{1}{n+3} \cdot d_{n+2} = \frac{1}{n+2} \cdot d_{n+1} - \frac{2}{n+2} + \frac{4}{n+3} \tag{5.12}$$

schreiben. Wir definieren $a_n = \dfrac{d_n}{n+1}$ und erhalten dann aus der letzten Gleichung

$$a_{n+2} = a_{n+1} - \frac{2}{n+2} + \frac{4}{n+3}$$

Die Substitution $n \mapsto n-2$ vereinfacht diese Gleichung zu

$$a_n = a_{n-1} - \frac{2}{n} + \frac{4}{n+1} \tag{5.13}$$

Diese Gleichung können wir mit dem Teleskop-Verfahren lösen. Wegen $a_0 = \frac{d_0}{1} = 0$ gilt

$$a_n = 4 \cdot \sum_{i=1}^{n} \frac{1}{i+1} - 2 \cdot \sum_{i=1}^{n} \frac{1}{i}. \tag{5.14}$$

Wir vereinfachen diese Summe:

$$\begin{aligned}
a_n &= 4 \cdot \sum_{i=1}^{n} \frac{1}{i+1} - 2 \cdot \sum_{i=1}^{n} \frac{1}{i} \\
&= 4 \cdot \sum_{i=2}^{n+1} \frac{1}{i} - 2 \cdot \sum_{i=1}^{n} \frac{1}{i} \\
&= 4 \cdot \frac{1}{n+1} - 4 \cdot \frac{1}{1} + 4 \cdot \sum_{i=1}^{n} \frac{1}{i} - 2 \cdot \sum_{i=1}^{n} \frac{1}{i} \\
&= 4 \cdot \frac{1}{n+1} - 4 \cdot \frac{1}{1} + 2 \cdot \sum_{i=1}^{n} \frac{1}{i} \\
&= -\frac{4 \cdot n}{n+1} + 2 \cdot \sum_{i=1}^{n} \frac{1}{i}
\end{aligned}$$

Um unsere Rechnung abzuschließen, berechnen wir eine Näherung für die Summe

$$H_n = \sum_{i=1}^{n} \frac{1}{i}.$$

Der Wert H_n wird in der Mathematik als die n-te *harmonische Zahl* bezeichnet. Dieser Wert hängt mit dem Wert $\ln(n)$ zusammen, Leonhard Euler (1707 – 1783) hat gezeigt, dass für große n die Approximation

$$\sum_{i=1}^{n} \frac{1}{i} \approx \ln(n)$$

benutzt werden kann. Genauer hat er folgendes gezeigt:

$$H_n = \ln(n) + \mathcal{O}(1).$$

Wir haben bei dieser Gleichung eine Schreibweise benutzt, die wir bisher noch nicht eingeführt haben. Sind f, g, h Funktionen aus $\mathbb{R}^{\mathbb{N}}$, so schreiben wir

$$f(n) = g(n) + \mathcal{O}\big(h(n)\big) \quad \text{g.d.w.} \quad f(n) - g(n) \in \mathcal{O}\big(h(n)\big).$$

Also haben wir insgesamt für a_n nun die Näherung

$$a_n = -\frac{4 \cdot n}{n+1} + 2 \cdot \ln(n) + \mathcal{O}(1)$$

gefunden. Wegen $d_n = (n+1) \cdot a_n$ gilt jetzt:

$$\begin{aligned}
d_n &= -4 \cdot n + 2 \cdot (n+1) \cdot H_n \\
&= -4 \cdot n + 2 \cdot (n+1) \cdot \big(\ln(n) + \mathcal{O}(1)\big) \\
&= 2 \cdot n \cdot \ln(n) + \mathcal{O}(n).
\end{aligned}$$

Wir vergleichen dieses Ergebnis mit dem Ergebnis, das wir bei der Analyse von "*Sortieren durch Mischen*" erhalten haben. Dort hatte sich die Anzahl der Vergleiche, die zum Sortieren eine Liste mit n Elementen durchgeführt werden mußte, als

$$n \cdot \log_2(n) + \mathcal{O}(n)$$

ergeben. Wegen $\ln(n) = \ln(2) \cdot \log_2(n)$ benötigen wir bei Quick-Sort im Durchschnitt

$$2 \cdot \ln(2) \cdot n \cdot \log_2(n)$$

Vergleiche, also etwa $2 \cdot \ln(2) \approx 1.39$ mehr Vergleiche als beim "*Sortieren durch Mischen*".

5.4.2 Eine feldbasierte Implementierung von *Quick-Sort*

Zum Abschluß geben wir eine feldbasierte Implementierung des *Quick-Sort*-Algorithmus an. Abbildung 5.7 zeigt diese Implementierung.

```
1   sort := procedure(rw l) {
2       quickSort(1, #l, l);
3   };
4   quickSort := procedure(s, e, rw l) {
5       if (e <= s) {
6           return; // at most one element, nothing to do
7       }
8       m := partition(s, e, l);  // split index
9       quickSort(s, m - 1, l);
10      quickSort(m + 1, e, l);
11  };
12  partition := procedure(start, end, rw l) {
13      pivot := l[start];
14      left  := start + 1;
15      right := end;
16      while (true) {
17          while (left <= end && l[left] <= pivot) {
18              left += 1;
19          }
20          while (l[right] > pivot) {
21              right -= 1;
22          }
23          if (left >= right) {
24              break;
25          }
26          swap(left, right, l);
27      }
28      swap(start, right, l);
29      return right;
30  };
```

Figure 5.7: Implementierung des *Quick-Sort*-Algorithmus in SetlX.

1. Im Gegensatz zu der feldbasierten Implementierung des *Merge-Sort-Algorithmus* benötigen wir diesmal kein zusätzliches Hilfsfeld. Dies ist ein wesentlicher Vorteil des *Quick-Sort-Algorithmus*.
2. Die Funktion `sort` wird auf die Implementierung der Funktion `quickSort` zurück geführt. Diese Funktion bekommt zunächst die beiden Parameter `s` und `e`.
   (a) `s` gibt den Index des ersten Elementes des Teilfeldes an, das zu sortieren ist.
   (b) `e` gibt den Index des letzten Elementes des Teilfeldes an, das zu sortieren ist.

   Außerdem bekommt diese Funktion als letzten Parameter das zu sortierende Feld. Der Aufruf `quickSort`(s, e) sortiert die Elemente

$$l\,[s]\,,\,l\,[s+1]\,,\,\cdots,\,l\,[e]$$

des Feldes l, d. h. nach dem Aufruf gilt:

$$l\,[s] \preceq l\,[s+1] \preceq \cdots \preceq l\,[e].$$

Die Implementierung der Funktion `quickSort` entspricht weitgehend der listenbasierten Implementierung. Der wesentliche Unterschied besteht darin, dass die Funktion `partition`, die in Zeile 8 aufgerufen wird, die Elemente des Feldes l so umverteilt, dass hinterher alle Elemente, die kleiner oder gleich dem *Pivot-Element* sind, links vor dem Index m stehen, während die restlichen Elemente rechts von m stehen. Das Pivot-Element selbst steht hinterher an der durch m definierten Stelle.

3. Die Schwierigkeit bei der Implementierung von *Quick-Sort* liegt in der Codierung der Funktion `partition`, die in Zeile 17 beginnt. Die Funktion `partition` erhält zwei Argumente:

   (a) `start` ist der Index des ersten Elementes in dem aufzuspaltenden Teilbereich.

   (b) `end` ist der Index des letzten Elementes in dem aufzuspaltenden Teilbereich.

   Die Funktion `partition` liefert als Resultat einen Index m aus der Menge

   $$\texttt{splitIdx} \in \{\texttt{start},\ \texttt{start}+1,\ \cdots,\ \texttt{end}\}.$$

   Außerdem wird der Teil des Feldes zwischen `start` und `end` so umsortiert, dass nach dem Aufruf der Funktion gilt:

   (a) Alle Elemente mit Index aus der Menge $\{\texttt{start}, \cdots, m-1\}$ kommen in der Ordnung "$\preceq$" vor dem Element an der Stelle `splitIdx`:

   $$\forall i \in \{\texttt{start}, \cdots, m-1\}\colon\ l[i] \preceq l[m].$$

   (b) Alle Elemente mit Index aus der Menge $\{m+1, \cdots, \texttt{end}\}$ kommen in der Ordnung "$\preceq$" hinter dem Element an der Stelle m:

   $$\forall i \in \{m+1, \cdots, \texttt{end}\}\colon\ l[m] \prec l[i].$$

   Der Algorithmus, um diese Bedingungen zu erreichen, wählt zunächst das Element

   $$l\,[\texttt{start}]$$

   als sogenanntes *Pivot-Element* aus. Anschließend läuft der Index `left` ausgehend von dem Index `start`$+1$ von links nach rechts bis ein Element gefunden wird, das größer als das Pivot-Element ist. Analog läuft der Index `right` ausgehend von dem Index `end` von rechts nach links, bis ein Element gefunden wird, dass kleiner oder gleich dem Pivot-Element ist. Falls nun `left` kleiner als `right` ist, werden die entsprechenden Elemente des Feldes ausgetauscht. In dem Moment, wo `left` größer oder gleich `right` wird, wird dieser Prozeß abgebrochen. Jetzt wird noch das Pivot-Element in die Mitte gestellt, anschließend wird `right` zurück gegeben.

4. Der Aufruf `swap`(i, j, l) vertauscht die Elemente des Feldes l, die die Indizes i und j haben.

Der einfachste Weg um zu verstehen, wie die Funktion `partition` funktioniert, besteht darin, diese Funktion an Hand eines Beispiels auszuprobieren. Wir betrachten dazu einen Ausschnitt aus einem Feld, der die Form

$$\cdots,\ 7,\ 2,\ 9,\ 1,\ 8,\ 5,\ 11,\ \cdots$$

hat. Wir nehmen an, dass der Index `start` die Position der Zahl 7 angibt und das der Index `end` auf die 11 zeigt.

1. Dann zeigt der Index `left` zunächst auf die Zahl 2 und der Index `right` zeigt auf die die Zahl 11.

2. Die erste `while`-Schleife vergleicht zunächst die Zahl 2 mit der Zahl 7. Da $2 \leq 7$ ist, wird der Index `left` inkrementiert, so dass er jetzt auf die Zahl 9 zeigt.

3. Anschließend vergleicht die erste `while`-Schleife die Zahlen 9 und 7. Da $\neg(9 \leq 7)$ ist, wird die erste `while`-Schleife abgebrochen.

4. Nun startet die zweite `while`-Schleife und vergleicht die Zahlen 7 und 11. Da $7 < 11$ ist, wird der Index `right` dekrementiert und zeigt nun auf die Zahl 5.

5. Da $\neg(7 < 5)$ ist, wird auch die zweite `while`-Schleife abgebrochen.

6. Anschließend wir geprüft, ob der Index `right` bereits über den Index `left` hinüber gelaufen ist und somit `left` $\geq$ `right` gilt. Dies ist aber nicht der Fall, denn `left` zeigt auf die 9, während `right` auf die 5 zeigt, die rechts von der 9 liegt. Daher wird die äußere `while`-Schleife noch nicht abgebrochen.

7. Jetzt werden die Elemente, auf die die Indizes `left` und `right` zeigen, vertauscht. In diesem Fall werden also die Zahlen 9 und 5 vertauscht. Damit hat der Ausschnitt aus dem Feld die Form

   $$\cdots,\ 7,\ 2,\ 5,\ 1,\ 8,\ 9,\ 11,\ \cdots$$

8. Jetzt geht es in die zweite Runde der äußeren `while`-Schleife. Zunächst vergleichen wir in der inneren `while`-Schleife die Elemente 5 und 7. Da $5 \leq 7$ ist, wird der Index `left` inkrementiert.

9. Dann vergleichen wir die Zahlen 1 und 7. Da $1 \leq 7$ ist, wird der Index `left` ein weiteres Mal inkrementiert und zeigt nun auf die 8.

10. Der Vergleich $8 \leq 7$ fällt negativ aus, daher wird die erste `while`-Schleife jetzt abgebrochen.

11. Die zweite `while`-Schleife vergleicht nun 7 und 9. Da $7 < 9$ ist, wird der Index `right` dekrementiert und zeigt jetzt auf die 8.

12. Anschließend werden die Zahlen 7 und 8 verglichen. Da auch $7 < 8$ gilt, wird der Index `right` ein weiteres Mal dekrementiert und zeigt nun auf die 1.

13. Jetzt werden die Zahlen 7 und 1 verglichen. Wegen $\neg(7 < 1)$ bricht nun die zweite `while`-Schleife ab.

14. Nun wird geprüft, ob der Index `right` über den Index `left` hinüber gelaufen ist und somit `left` $\geq$ `right` gilt. Diesmal ist der Test positiv, denn `left` zeigt auf die 8, während `right` auf die 1 zeigt, die links von der 8 steht. Also wird die äußere Schleife durch den `break`-Befehl in Zeile 28 abgebrochen.

15. Zum Abschluß wird das Pivot-Element, das durch den Index `start` identifiziert wird, mit dem Element vertauscht, auf das der Index `right` zeigt, wir vertauschen also die Elemente 7 und 1. Damit hat das Feld die Form

    $$\cdots,\ 1,\ 2,\ 5,\ 7,\ 8,\ 9,\ 11,\ \cdots$$

    Als Ergebnis wird nun der Index `right`, der jetzt auf das Pivot-Element zeigt, zurück gegeben.

5.4.3 Korrektheit

Die Implementierung der Funktion `partition` ist trickreich. Daher untersuchen wir die Korrektheit der Funktion jetzt im Detail. Zunächst formulieren wir Invarianten, die für die äußere `while`-Schleife, die sich von Zeile 23 bis Zeile 35 erstreckt, gelten. Zur Abkürzun bezeichnen wir das Pivot-Element mit x, wir setzen also

$$x := \texttt{pivot} = l[\texttt{start}].$$

Dann gelten die folgenden Invarianten:

(I1) $\forall i \in \{\texttt{start} + 1, \cdots, \texttt{left} - 1\}\colon\ l[i] \preceq x$

(I2) $\forall j \in \{\texttt{right} + 1, \cdots, \texttt{end}\}\colon\ x \prec l[j]$

(I3) $\texttt{start} + 1 \leq \texttt{left}$

(I4) $\texttt{right} \leq \texttt{end}$

(I5) $\texttt{left} \leq \texttt{right} + 1$

Wir weisen die Gültigkeit dieser Invarianten durch Induktion nach. Dazu ist zunächst zu zeigen, dass diese Invarianten dann erfüllt sind, bevor die Schleife zum ersten Mal durchlaufen wird. Zu Beginn gilt

$$\texttt{left} = \texttt{start} + 1.$$

Daraus folgt sofort, dass die dritte Invariante anfangs erfüllt ist. Außerdem gilt dann

$$\{\texttt{start} + 1, \cdots, \texttt{left} - 1\} = \{\texttt{start} + 1, \cdots, \texttt{start}\} = \{\}$$

und damit ist auch klar, dass die erste Invariante gilt, denn für $\texttt{left} = \texttt{start} + 1$ ist die erste Invariante eine leere Aussage. Weiter gilt zu Beginn

$$\texttt{right} = \texttt{end},$$

woraus unmittelbar die Gültigkeit der vierten Invariante folgt. Außerdem gilt dann

$$\{\texttt{right} + 1, \cdots, \texttt{end}\} = \{\texttt{end} + 1, \cdots, \texttt{end}\} = \{\},$$

so dass auch die zweite Invariante trivialerweise erfüllt ist. Für die fünfte Invariante gilt anfangs

$$\texttt{left} \leq \texttt{right} + 1 \ \Leftrightarrow\ \texttt{start} + 1 \leq \texttt{end} + 1 \ \Leftrightarrow\ \texttt{start} \leq \texttt{end} \ \Leftrightarrow\ \texttt{true},$$

denn die Funktion $\texttt{partition}(\texttt{start}, \texttt{end})$ wird nur aufgerufen, falls $\texttt{start} < \texttt{end}$ ist.

Als nächstes zeigen wir, dass die Invarianten bei einem Schleifen-Durchlauf erhalten bleiben.

1. Die erste Invariante gilt, weil `left` nur dann inkrementiert wird, wenn vorher

   $$l[\texttt{left}] \preceq x$$

   gilt. Wenn die Menge $\{\texttt{start} + 1, \cdots, \texttt{left} - 1\}$ also um $i = \texttt{left}$ vergrößert wird, so ist sichergestellt, dass für dieses i gilt:

   $$l[i] \preceq x.$$

2. Die zweite Invariante gilt, weil `right` nur dann dekrementiert wird, wenn vorher

   $$x \prec l[\texttt{right}]$$

gilt. Wenn die Menge $\{\texttt{right}+1, \cdots, \texttt{end}\}$ also um $i = \texttt{right}$ vergrößert wird, so ist sichergestellt, dass für dieses i gilt

$$x \prec l[i].$$

3. Die Gültigkeit der dritten Invariante folgt aus der Tatsache, dass `left` in der ganzen Schleife höchstens inkrementiert wird. Wenn also zu Beginn $\texttt{start}+1 \leq \texttt{left}$ gilt, so wird dies immer gelten.

4. Analog ist die vierten Invariante gültig, weil zu Beginn $\texttt{right} \leq \texttt{end}$ gilt und `right` immer nur dekrementiert wird.

5. Aus den ersten beiden Invarianten (I1) und (I2) folgt:

   $$\{\texttt{start}+1, \cdots, \texttt{left}-1\} \cap \{\texttt{right}+1, \cdots, \texttt{end}\} = \{\},$$

   denn ein Element des Feldes kann nicht gleichzeitig kleiner-gleich x und größer x sein. Wenn $\texttt{right}+1 \leq \texttt{end}$ ist, dann ist die zweite Menge nicht-leer und es folgt

   $$\texttt{left}-1 < \texttt{right}+1 \quad \text{und das impliziert} \quad \texttt{left} \leq \texttt{right}+1.$$

   Andernfalls gilt $\texttt{right} = \texttt{end}$. Dann haben wir

   $$\texttt{left} \leq \texttt{right}+1 \;\Leftrightarrow\; \texttt{left} \leq \texttt{end}+1 \;\Leftrightarrow\; \texttt{true},$$

   denn wenn $\texttt{left} > \texttt{end}$ ist, wird `left` in der ersten Schleife nicht mehr erhöht. `left` wird nur dann und auch nur um 1 inkrementiert, solange $\texttt{left} \leq \texttt{end}$ gilt. Also kann `left` maximal den Wert $\texttt{end}+1$ annehmen.

Um den Beweis der Korrektheit abzuschließen, muß noch gezeigt werden, dass alle `while`-Schleifen terminieren. Für die erste innere `while`-Schleife folgt das daraus, dass bei jedem Schleifen-Durchlauf die Variable `left` inkrementiert wird. Da die Schleife andererseits die Bedingung

$$\texttt{left} \leq \texttt{end}$$

enthält, kann `left` nicht beliebig oft inkrementiert werden und die Schleife muß irgendwann abbrechen.

Die zweite innere `while`-Schleife terminiert, weil einerseits `right` in jedem Schleifen-Durchlauf dekrementiert wird und andererseits aus der dritten und der fünften Invariante folgt:

$$\texttt{right}+1 \geq \texttt{left} \geq \texttt{start}+1, \quad \text{also} \quad \texttt{right} \geq \texttt{start}.$$

Die äußere `while`-Schleife terminiert, weil die Menge

$$M := \{\texttt{left}, \cdots, \texttt{right}\}$$

ständig verkleinert wird. Um das zu sehen, führen wir eine Fall-Unterscheidung durch:

1. Fall: Nach dem Ende der Schleife in Zeile 17 – 19 gilt

   $$\texttt{left} > \texttt{end}.$$

   Diese Schleife bricht also ab, weil die Bedingung $\texttt{left} \leq \texttt{end}$ verletzt ist. Wir haben oben schon gesehen, dass dann

   $$\texttt{left} = \texttt{end}+1 \quad \text{und} \quad \texttt{right} = \texttt{end}$$

   gelten muß. Daraus folgt aber sofort

   $$\texttt{left} > \texttt{right}$$

und folglich wird die äußere Schleife dann durch den Befehl `break` in Zeile 24 abgebrochen.

2. Fall: Nach dem Ende der Schleife in Zeile 17 – 19 gilt

   $$\texttt{left} \leq \texttt{end}.$$

   Die Schleife bricht also ab, weil die Bedingung $l[\texttt{left}] \preceq x$ verletzt ist, es gilt also

   $$x \prec l[\texttt{left}].$$

   Analog gilt nach dem Abbruch der zweiten inneren `while`-Schleife

   $$l[\texttt{right}] \preceq x.$$

   Wenn die äußere Schleife nun nicht abbricht weil $\texttt{left} < \texttt{right}$ ist, dann werden die Elemente $l[\texttt{left}]$ und $l[\texttt{right}]$ vertauscht. Nach dieser Vertauschung gilt offenbar

   $$x \prec l[\texttt{right}] \quad \text{und} \quad l[\texttt{left}] \preceq x.$$

   Wenn nun also die äußere Schleife erneut durchlaufen wird, dann wird die zweite innere Schleife mindestens einmal durchlaufen, so dass also `right` dekrementiert wird und folglich die Menge $M = \{\texttt{left}, \cdots, \texttt{right}\}$ um ein Element verkleinert wird. Das geht aber nur endlich oft, denn spätestens wenn die Menge leer ist, gilt $\texttt{left} = \texttt{right} + 1$ und die Schleife wird durch den Befehl `break` in Zeile 24 abgebrochen.

Jetzt haben wir alles Material zusammen, um die Korrektheit unserer Implementierung zu zeigen. Wenn die Schleife abbricht, gilt $\texttt{left} > \texttt{right}$. Wegen der fünften Invariante gilt $\texttt{left} \leq \texttt{right} + 1$. Also gibt es nur noch die Möglichkeit

$$\texttt{left} = \texttt{right} + 1.$$

Wegen den ersten beiden Invarianten wissen wir also

$$\forall i \in \{\texttt{start} + 1, \cdots, \texttt{right}\}\colon\ l[i] \preceq x$$

$$\forall j \in \{\texttt{right} + 1, \cdots, \texttt{end}\}\colon\ x \prec l[j].$$

Durch das `swap` in Zeile 28 wird nun x mit dem Element an der Position `right` vertauscht. Dann sind anschließend alle Elemente links von x kleiner-gleich x und alle Elemente rechts von x sind größer. Damit ist die Korrektheit von `partition`() nachgewiesen.

5.4.4 Mögliche Verbesserungen

In der Praxis gibt es noch eine Reihe Tricks, um die Implementierung von *Quick-Sort* effizienter zu machen:

1. Anstatt immer das erste Element als Pivot-Element zu wählen, werden drei Elemente aus der zu sortierenden Liste ausgewählt, z. B. das erste, das letzte und ein Element aus der Mitte des Feldes. Als Pivot-Element wird dann das Element gewählt, was der Größe nach zwischen den anderen Elementen liegt.

   Der Vorteil dieser Strategie liegt darin, dass der schlechteste Fall, bei dem die Laufzeit von *Quick-Sort* quadratisch ist, wesentlich unwahrscheinlicher wird. Insbesondere kann der schlechteste Fall nicht mehr bei Listen auftreten, die bereits sortiert sind.

2. Falls weniger als 10 Elemente zu sortieren sind, wird auf "*Sortieren durch Einfügen*" zurück gegriffen.

Der Artikel von Bentley and M. Douglas McIlroy "*Engineering a Sort Function*" [BM93] beschreibt diese und weitere Verbesserungen des Quick-Sort Algorithmus.

Exercise 16: Implementieren Sie die oben aufgeführten Verbesserungen.

5.5 Eine untere Schranke für die Anzahl der Vergleiche

Wir wollen in diesem Abschnitt zeigen, dass jeder Sortier-Algorithmus, der in der Lage ist, eine beliebige Liste von Elementen zu sortieren, mindesten die Komplexität $\mathcal{O}\big(n \cdot \ln(n)\big)$ haben muss. Dabei setzen wir voraus, dass die einzelnen Elemente nur mit Hilfe des Operators $<$ verglichen werden können und wir setzen weiter voraus, dass wir eine Liste von n verschiedenen Elementen haben, die wir sortieren wollen. Wir betrachten zunächst eine Liste von zwei Elementen: $[a_1, a_2]$. Um diese Liste zu sortieren, reicht ein Vergleich aus, denn es gibt nur zwei Möglichkeiten, wie diese beiden Elemente sortiert sein können:

1. Falls $a_1 < a_2$ ist, dann ist $[a_1, a_2]$ aufsteigend sortiert.
2. Falls $a_2 < a_1$ ist, dann ist $[a_2, a_1]$ aufsteigend sortiert.

Falls wir eine Liste von drei Elementen $[a_1, a_2, a_3]$ haben, so gibt es bereits 6 Möglichkeiten, diese anzuordnen:

$$[a_1, a_2, a_3], \quad [a_1, a_3, a_2], \quad [a_2, a_1, a_3], \quad [a_2, a_3, a_1], \quad [a_3, a_1, a_2], \quad [a_3, a_2, a_1].$$

Hier benötigen wir bereits drei Vergleiche zum Sortieren, denn mit zwei Vergleichen können wir maximal aus vier verschiedenen Möglichkeiten auswählen.

Im allgemeinen gibt es $n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot (n-1) \cdot n = \prod\limits_{i=1}^{n} i$ verschiedene Möglichkeiten, die Elemente einer n-elementigen Liste $[a_1, a_2, \cdots, a_n]$ anzuordnen. Dies läßt sich am einfachsten durch Induktion nachweisen:

1. Offenbar gibt es genau eine Möglichkeit, eine Liste von einem Element anzuordnen.
2. Um eine Liste von $n+1$ Elementen anzuordnen haben wir $n+1$ Möglichkeiten, das erste Element der Liste auszusuchen. In jedem dieser Fälle haben wir dann nach Induktions-Voraussetzung $n!$ Möglichkeiten, die restlichen Elemente anzuordnen, so dass wir insgesamt auf $(n+1) \cdot n! = (n+1)!$ verschiedene Anordnungsmöglichkeiten kommen.

Wir überlegen jetzt umgekehrt, aus wievielen Möglichkeiten wir mit k verschiedenen Tests auswählen können.

1. Offenbar können wir mit einem Test aus zwei Möglichkeiten auswählen.
2. Mit zwei Tests können wir aus vier Möglichkeiten wählen.
3. Im Allgemeinen können wir mit k Tests aus 2^k Möglichkeiten auswählen.

Die letzte Aussage ist einfach zu begründen: Stellen wir die Ergebnisse der Tests durch 0 und 1 dar, so entsprechen k verschiedene Tests einem binären String der Länge k. Die binären Strings der Länge k kodieren aber genau die Zahlen von 0 bis $2^k - 1$ im Zweiersystem. Nun gilt

$$\mathit{card}\big(\{0, 1, 2, \cdots, 2^k - 1\}\big) = 2^k.$$

Daher gibt es genau 2^k solcher Strings.

Wenn wir eine beliebig angeordnete Liste der Länge n haben, dann gibt es insgesamt $n!$ Möglichkeiten, diese anzuordnen. Um nun herauszufinden, welche spezielle Anordnung unter den insgesamt möglichen $n!$ Anordnungen vorliegt, müssen wir k Vergleiche durchführen, wobei für k die Abschätzung

$$2^k \geq n!$$

gelten muss. Daraus folgt sofort

$$k \geq \log_2(n!).$$

An dieser Stelle benötigen wir eine Näherungsformel zur Berechnung von $\log_2(n!)$. Die einfachste solche Formel ist

$$\log_2(n!) = n \cdot \log_2(n) + \mathcal{O}(n).$$

Damit erhalten wir dann die Abschätzung

$$k \geq n \cdot \log_2(n) + \mathcal{O}(n)$$

und da der Merge-Sort-Algorithmus tatsächlich mit dieser Anzahl Vergleichen auskommt, ist dieser Algorithmus bezüglich der Anzahl der Vergleichs-Operationen optimal.

Bibliography

[AHU87] Alfred V. Aho, John E. Hopcraft, and Jeffrey D. Ullman. *Data Structures and Algorithms*. Addison-Wesley, 1987.

[AP10] Adnan Aziz and Amit Prakash. *Algorithms for Interviews*. CreateSpace Independent Publishing Platform, 2010.

[AVL62] Georgii M. Adel'son-Vel'skiĭ and Evgenii M. Landis. An algorithm for the organization of information. *Soviet Mathematics Doklady*, 3:1259–1263, 1962.

[BM93] Jon Louis Bentley and M. Douglas McIlroy. Engineering a sort function. *Software - Practice and Experience*, 23(11):1249–1265, 1993.

[CLRS01] Thomas H. Cormen, Charles E. Leiserson, Ronal L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, second edition, 2001.

[CLRS09] Thomas H. Cormen, Charles E. Leiserson, Ronal L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, third edition, 2009.

[CPR11] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Proving program termination. *Communications of the ACM*, 54(5):88–98, 2011.

[Dij59] Edsger W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[dlB59] René de la Briandais. File Searching Using Variable Length Keys. In *Proceedings of the Western Joint Computer Conference*, pages 195–298, 1959.

[DPV08] Sanjoy Dasgupta, Christos H. Papadimitriou, and Umesh V. Vazirani. *Algorithms*. McGraw-Hill, 2008.

[GS08] Hans-Peter Gumm and Manfred Sommer. *Einführung in die Informatik*. Oldenbourg-Verlag, 8th edition, 2008.

[HMU06] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 3rd edition, 2006.

[Hoa61] C. Antony R. Hoare. Algorithm 64: Quicksort. *Communications of the ACM*, 4:321, 1961.

[Hoa69] C. Antony R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[Huf52] David A. Huffman. A Method for the Construction of Minimum-Redundancy Codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.

[IdFF96] Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes Filho. Lua — an extensible extension language. *Software: Practice and Experience*, 26(6):635–652, 1996.

[Ier06] Roberto Ierusalimschy. *Programming in Lua*. Lua.Org, 2nd edition, 2006.

[MGS96] Martin Müller, Thomas Glaß, and Karl Stroetmann. Automated modular termination proofs for real prolog programs. In Radhia Cousot and David A. Schmidt, editors, *SAS*, volume 1145 of *Lecture Notes in Computer Science*, pages 220–237. Springer, 1996.

[Moo59] Edward F. Moore. The shortest path through a maze. In *Proceedings of the International Symposium on the Theory of Switching*, pages 285–292. Harvard University Press, 1959.

[MS08] Kurt Mehlhorn and Peter Sanders. *Algorithms and Data Structures: The Basic Toolbox*. Springer, 2008.

[Sip96] Michael Sipser. *Introduction to the Theory of Computation*. PWS Publishing Company, 1996.

[SW11] Robert Sedgewick and Kevin Wayne. *Algorithms*. Addison-Wesley, 4th edition, 2011.

[Tur36] Alan M. Turing. On computable numbers, with an application to the "Entscheidungsproblem". *Proceedings of the London Mathematical Society*, 42(2):230–265, 1936.

[WS92] Larry Wall and Randal L. Schwartz. *Programming Perl*. O'Reilly and Assoc., 1992.