

Theory of Computation

That's all very well in practice, but how does it work in theory?

Prof. M. P. Schellekens¹
University College Cork
Department of computer science
Email: m.schellekens@cs.ucc.ie

Course website

<https://sites.google.com/site/sitecs3509fa13/home>

¹Acknowledgments: many thanks to Christian van den Bosch for preparing a latex draft of my handwritten notes from which this document grew. Thanks to the students who took this class and whose questions and critical remarks greatly helped improve earlier drafts.

Contents

0.1	Work in progress	2
1	Introduction	3
1.1	What is “Theory of Computation”?	3
1.2	The science of computing	4
1.2.1	Universal laws of nature	4
1.2.2	Computation	5
	Examples of early computation	5
	Automating computation	6
	Computation in nature	7
1.2.3	Turing’s contribution	8
1.3	Questions arising in the theory of computation	8
1.4	Computer science’s three C’s	10
2	Course overview	11
2.1	Countability	11
2.2	Computability	12
2.3	Comparison-based algorithms & basic complexity measures	12
2.4	Lower bounds	14
2.4.1	Lower bounds on algorithmic improvement	14
2.4.2	Lower bounds on compressibility	14
2.5	Computational tractability	14
2.6	Computational intractability: classification of computational difficulty	15
2.7	Course excursions	16
3	A quick revision of functions and proof techniques	17
3.1	Basic notions	17
3.2	The input-output function of a program	18
3.3	Finite sequences	20
3.4	Finite sequences as functions	20
3.5	Power-notation for sets of functions	21
3.6	Infinite sequences as functions	21
3.6.1	Infinite binary sequences	22
3.7	Injections, surjections and bijections	22

3.8	The Holmes Principle (proof by way of contradiction)	26
4	Countability	28
4.1	Countable sets	28
4.2	Examples	29
4.3	Characterization of countability	31
4.4	Cantor’s world: exponentiation and powersets	36
4.4.1	powersets in Computer Science	36
4.4.2	Lattice representation of powersets	37
4.4.3	Binary sequences encode powersets	38
	Binary subset-encoding: the finite case	39
	Binary subset-encoding: the infinite case	40
5	Computability	45
5.1	Introduction to computability	45
5.2	Distinguishing between the function to be computed and programs that compute it	47
5.3	Enumerating all programs	47
5.4	Total and partial functions	48
5.5	The “output” undefined	49
5.6	Existence of non-computable functions	49
5.7	Infinite exponentials (optional section)	51
6	The halting problem	52
6.1	Outline of the problem	53
6.2	The halting problem today	53
6.3	The model of computation	54
6.4	The difficulty with the halting problem	54
6.5	Interpreters	55
6.6	Variants of the problem	55
6.7	Reduction techniques	55
6.8	Non-computability of the Halting Problem	56
6.8.1	The halting problem is undecidable	57
6.8.2	Halting problem: solved exercises	59
6.8.3	The halting problem: practice exercises	63
6.9	Can humans solve the halting problem?	64
6.10	Twin primes	64
6.11	Implications of the halting problem for the design of Real-Time pro- gramming languages	65

7	Knuth's lab: comparison-based algorithms	67
7.1	Comparison-based algorithms	67
7.2	Examples of comparison-based sorting algorithms	72
7.3	Binary comparison-paths	78
7.4	Lists sharing the same relative order	80
7.5	Completeness	84
7.6	Faithfulness	86
7.7	Separation	87
7.8	Reduction: the zero-one principle	88
8	Basic complexity measures	94
8.1	An introduction to software timing	94
8.2	Timing methods: an evaluation	94
8.3	Size-based methods	98
8.3.1	Repeated elements	98
8.3.2	Reduction of the analysis to finitely many input cases	99
8.3.3	Main timing measures	101
	Worst-case analysis	101
	Average-case analysis	102
	Trade-offs	103
	Average-case time is bounded by worst-case time	104
	Asymptotic classification of timing measures	105
8.3.4	Overview of timing measures	106
8.3.5	Brute force time computation for comparison-based sorting	107
	Stirling's approximation	108
	Gauging the size of $52!$	108
8.3.6	The Library of Babel	110
8.3.7	Beyond the Library of Babel	110
9	Complexity analysis techniques	112
9.1	Random sampling	112
9.2	Comparison time	113
9.3	Constant time algorithms	113
9.3.1	Analysis of Bubble Sort	114
9.3.2	Analysis of Selection Sort	115
9.4	Non-constant time algorithms	117
9.4.1	Grouping inputs by running time	117
9.4.2	Analysis of Sequential Search	118
9.5	Algorithms involving while loops	122
9.5.1	Analysis of Insertion Sort	122
9.5.2	Worst-case time of Insertion Sort	124
9.5.3	Average-case time of Insertion Sort	126
9.5.4	Average-case of Insertion Sort (alternative method)	129

10 Modularity	134
10.1 Timing modularity: the real-time engineer's paradise	134
10.2 Modularity of exact time	135
10.3 Reuse	136
10.4 Semi-modularity of worst-case time	137
10.4.1 Frog-Leap example	137
10.4.2 Semi-modularity of worst-case time: output-based version . .	141
Semi-modularity of worst-case time: size-based version	141
10.5 Modularity of average-case time	143
10.5.1 Modularity of average-case time: output-based version	143
10.5.2 Average-case time "paradox"	144
10.5.3 Modularity of average-case time : failure of size-based version	144
10.6 Predictability and hardware design	145
11 Duality	146
11.0.1 Duality at work: a tale of three camels	146
11.0.2 Duality in graphs	146
12 Lower bounds	148
12.1 Lower bounds on algorithmic improvement	148
12.2 How can this result be demonstrated?	149
12.3 Decision trees: definition and example	149
12.4 Verification of $\Omega(n \log n)$ lower bound for worst-case time	157
12.5 Kraft's inequality	159
12.6 Jensen's inequality	165
12.7 Verification of $\Omega(n \log n)$ lower bound for average-case time	168
12.8 Non-comparison-based algorithms: Counting Sort	170
13 Recurrence equations and generating functions	171
14 Entropy and applications	172
14.1 Probability and entropy	172
14.2 Predicting average compression cost for Huffman codes	176
14.2.1 Exercise on Huffman compression	180
14.2.2 Lower bounds revisited	181
14.3 Binary subset-encoding revisited	183
14.3.1 Basic notions	183
14.3.2 Ordering finite sets	183
14.3.3 Subset-indexing lemma	184
14.3.4 Subset-index encoding	187
14.3.5 Subset-index decoding	187
14.4 Machine learning	187

15 Computational tractability	188
15.1 Series-parallel partial orders	188
15.2 Labeled partial orders & counting labelings	188
16 Computational intractability	189
16.1 Polynomial time computable algorithms	189
16.2 Polynomial time exercises	190
16.3 NP-complete problems	191
17 Appendix 1: Useful maths facts	193
17.0.1 The triangle equality	193
17.0.2 Pascal's triangle	194
17.0.3 Telescopic sums	195
17.0.4 Hockey stick lemma	196
18 Appendix 2: Solutions to selected exercises	198

0.1 Work in progress

The course notes are ever evolving and form the basis of a book in progress. Some sections and chapters need completing. These are clearly indicated in the text. Some of the material is covered from other sources as indicated on the course webpage (NP-completeness results).

Chapter 1

Introduction

Computer science is no more about computers than astronomy is about telescopes.

Edsger Dijkstra

1.1 What is “Theory of Computation”?



Boundaries of computation

The theory of computation explores “extreme computability” questions. The theory pushes the boundaries of exploration as far as possible focusing on “what can and can’t be done” in terms of computation.

The theory of computation focuses on questions such as:

- Can we always improve the speed of an algorithm significantly through better coding?
- What is the best file compression achievable?
- Are all general-purpose programming languages equally powerful? Can you, say, compute problems in Python that can’t be computed in Java, or the other way around?”

The theory of computation is part of the cultural baggage of every computer scientist. At times theory of computation results are used in industry to point out impossible requirements for products or to indicate that products can’t be further improved. Care needs to be taken with such arguments to ensure that they match up with what actually occurs in practice.

Knowledge of key results in the theory of computation affects practice in many ways. For instance, the famous “halting problem”, which we will study early on in the notes, affects the field of safety-critical applications and influenced the development of “real-time programming languages”—languages in which execution timing can be assured.

Many problems addressed in the course arise in such practical contexts. You will learn when and how these occur. The subject, at the very least, will make you question your basic assumptions about the field. We will visit the relation between super large numbers and the use of sampling techniques and static timing methods, the existence of uncomputable problems, the existence of entire classes of problems that are (most likely) far too hard to compute as well as nifty compression techniques.

Topics covered in this course will be useful in more applied areas (such as the use of Shannon’s entropy in machine learning) and will lend insight into the fundamental nature of computing (such as lower bounds on complexity of programs).

Aside from merely focusing on what can’t be achieved in Computer Science (such as solving the halting problem), we will also focus on what’s achievable (such as optimal means to compress files).

1.2 The science of computing

1.2.1 Universal laws of nature

To gain a better understanding of the theory of computation, it is useful to consider the *science* aspect of computer science and compare the field with other scientific areas such as physics.

Physics studies nature, where one can observe universal laws. Newton’s second law links gravitational force to mass and gravitational induced acceleration ($F = ma$). The law is considered universal, i.e. expected to hold anywhere in the universe. Computers do not occur “in nature”, at least not in any shape we are used to. Computers are man-made, where industry produces *different* machines to carry out computation. If there is so much freedom in designing computers, how can we hope to study them?

One answer is that computers are produced to *compute* problems for humans and that the *process of computation* turns out to be universal. The process of computation will satisfy laws. For instance, the running-time of comparison-based sorting algorithms must satisfy $\Omega(n \log n)$. These are theoretical bounds, similar to theoretical laws of physics such as “nothing proceeds faster than the speed of light”.

There are also practical bounds, obtained by experimentation. For instance Zipf’s law, a useful law in Information Retrieval and compression, states that the given a large corpus of files containing words, the frequency of any word is inversely proportional to its rank in the frequency table. The most frequent word will occur approximately twice as often as the second most frequent word, three times as often as the third most frequent word, etc. The rank-frequency relation is an inverse

relation. For example, in the a large corpus of English text, the word “the” occurs most frequently, and say accounts for nearly 6% of all word occurrences. According to Zipf’s Law, the second-place word of accounts for about 3% of words. This has been tested on many corpora of documents in many languages. The experimentally determined rank-frequency relation in all these cases is closely matched (and hence predicted) by Zipf’s law. Such a law is an experimentally determined law, similar to experimental laws of physics. Ohm’s law regarding electric currents was originally determined by experimentation. Yet the law plays a crucial role in Information Retrieval and predictions for efficient compression.

The lower bound on the running time of comparison based algorithms can be established both experimentally and with a theoretical argument, as we will do in this course. In fact, the law holds both for the worst-case running time and the average-case running time. This law guides the design of efficient algorithms. Since it holds, the designer knows that they need to aim for $O(n \log n)$ algorithms when designing a sorting algorithm. And they know that they cannot improve on this (bar at the level of the constant factor involved in the running time, e.g. an improvement of say $1000n \log(n)$ comparisons on a list of size n to an algorithm that only takes $3n \log(n)$ comparisons.

Theoretical laws and experimental laws typically go hand in hand. When a theoretical law is discovered, it is important to test it experimentally. And when an experimental law is discovered, it needs to be backed up with a theoretical model so the deeper reasons for its occurrence can be understood. To this day, no one really knows why Zipf’s law holds. There are proposed explanations, but the theory has not yet been fully established.

1.2.2 Computation

Examples of early computation

Computers are man-made, but the process of “computation” occurs naturally. Humans have carried it out long before the advent of computers, for instance on the abacus, developed in the period between 2700-2300 BCE.

A second example of early computation, among many others, are the calculations used by Archimedes in the 3rd century BCE. Archimedes discovered and used the **law of exponents**

$$10^x \times 10^y = 10^{x+y}$$

and used exponentiation¹ to compute the number of grains of sand in the universe (10^{63} , one thousand trillion trillion trillion trillion sand grains² published in “The sand reckoner”). This involved a recursive notation for numbers, starting with the myriad-myriad³, namely $10^4 \times 10^4 = 10^8$. Numbers up to this point (but not including

¹Known to Euclid, who introduced the terminology “power” in this context.

²Eureka Man: The Life and Legacy of Archimedes, by Alan Hirshfeld.

³From the Greek myrias, designating ten thousand.

10^8), he called the “first order” numbers. Numbers of the second order were arrived at from 10^8 up to $10^8 \times 10^8 = 10^{16} - 1$, and so on. He kept this up until “a myriad of myriads of the unit of the myriad-myriadth order” $10^{8^{10^8}}$.

Exponentiation underpins several of the results we will discuss: the existence proof of non-computable functions, Stirling’s approximation and Kraft’s inequality characterizing binary trees among others.

Large numbers such as the ones studied by Archimedes, **that can defy current computational capacity** set limits on the size of the data we can store and analyze. This in turn, as we will see, affects the choice of time analysis techniques for algorithms.

Archimedes also introduced pre-calculus techniques to compute surface areas and volumes⁴. Calculus techniques and statistics now underpin **neural networks and deep learning**.

Automating computation

Principles of computation became partly automated with the invention of the Jacquard loom (1801). Further automation occurred in various pre-modern age computation devices. Shortly after George Boole published *The Laws of Thought* in the mid nineteenth century, Jevons built a “logic machine”, the first to carry out proposition logic operations faster than a human. In the same period, Charles Babbage worked on his analytical engine and Augusta Ada King-Noel, Countess of Lovelace, published the first algorithm ever specifically tailored for implementation on a computer. A full characterization of the notion of computation however was only achieved by Alan Turing. We will return to his central contribution. This course does not introduce Turing Computability in detail, i.e. via Turing Machines or Unlimited Register Machines URM. A good reference to this topic is the book by N. J. Cutland: *Computability: An Introduction to Recursive Function Theory* from Cambridge University Press⁵.

⁴Archimedes used the mechanical principle of balance in combination with what would now be called limit-arguments to determine surface areas and volumes. The details of this method, in which Archimedes refers to a type of “infinitesimals”, an approach that only reemerged through Newton’s work, were copied on a parchment in the 10th-century. The history of the palimpsest is fascinating. The original text was nearly destroyed due to the many damaging processes it was subjected to in its history, including suspected theft, overwriting and damp storage. It was bought in the end by a Silicon Valley IT specialist. Imaging scientists discovered and painstakingly recovered the original content—a lucky find to say the least! The method consists of a pre-modern calculus technique in which Archimedes achieved his computations through limit procedures (using approximating upper and lower bounds that coincide in the limit).

⁵ISBN-10 0521294657

Computation in nature

Computation is not a purely human endeavour. Computation also occurs in nature.

Bees share food source locations, passing on information through coordinated “waggle dances” in hives⁶. Such interactions between independent “units” are linked to the topic of modern distributed computing.

Ants find their way through mazes, determining optimal routes in a coordinated way. Argentine ants foraging through a purposely designed maze that mimicked the Tower of Hanoi problem, actually solved the well-known Computer Science problem, converging on the most optimal solution⁷.

Today's computers have been developed following designs that often favour speed over predictability. Hence studying computers is intrinsically difficult. Studying computation however is largely separate from the machines supporting the process. As Dijkstra puts it, Computer science is no more about computers than astronomy is about telescopes. The study of computation will form the topic of the current course, focusing on “universal laws of computation.”

⁶Frisch, Karl von. (1967) *The Dance Language and Orientation of Bees*. Cambridge, Mass.: The Belknap Press of Harvard University Press.

⁷Reid, C. R., D. J. T. Sumpter and M. Beekman. 2011. *Optimisation in a natural system: Argentine ants solve the Towers of Hanoi*. Journal of Experimental Biology. 214 (1): 50-58

1.2.3 Turing's contribution

If one is lucky, a solitary fantasy can totally transform a million realities.

Maya Angelou

Turing

With the advent of computers, Alan Turing (1912-1945), an English mathematician, logician and cryptographer, considered what it means to be able to compute. Turing provided a formalisation of the concept of computation through the notion of a Turing machine—and in doing so laid the foundations for the field of theoretical computer science.

Several researchers aimed to capture what it means for a problem to be computable. All models turned out to be equivalent, and in particular to the Turing machine model. Rather than providing an introduction to Turing Machines we opt for an approach to computability based on “general purpose programming languages”, where we will rely on pseudo-code to specify programs. Time allowing, we will cover Unlimited Register Machines (or URMs, an equivalent computational model) in the tutorials. Unlimited Register Machines are introduced in the wonderful book by N. J. Cutland: *Computability: An Introduction to Recursive Function Theory* from Cambridge University Press⁸.

Church-Turing thesis

Turing formulated the widely accepted “Turing” version of the Church-Turing thesis, namely that any practical computing model has either the equivalent or a subset of the capabilities of a Turing machine. In doing so, Turing captured the notion delineating the limits of computation.

Exploring these limits is a main theme of this course. Since there is one accepted notion of computation, it makes sense to carry out an in depth scientific analysis of computation and to further question its nature.

We will link theory to practical considerations wherever possible. This philosophy influenced the selection of topics.

1.3 Questions arising in the theory of computation

We reiterate some of the main questions pertinent to the theory of computation and add a few new ones.

⁸ISBN-10 0521294657

- Are all general-purpose programming languages equally powerful? Could Java programs solve more problems than say Python programs?
- Say you are asked to program a certain task on your computer. Are you sure that a program exists that will compute this particular task?
- Could there be tasks that are not computable, i.e. for which no program exists that solves the problem?
- How can you be sure your program is correct?
- How can you be sure your program runs in a reasonable time?
- How will you know that when you present your novel algorithm for use, it is the most optimal version out there?
- Could you automate debugging?
- What is the best possible file compression achievable?

We will answer most of these questions, either fully or partially, in the course, bar the topic of code correctness, which we leave aside due to time considerations. The course will introduce both “large-scale” and “small-scale” results in the theory of computation, somewhat akin to physics studying astronomy scale versus atomic scale problems.

Large scale results regard general considerations on computability, including general computability results (the halting problem) as well as general complexity-related results (NP-completeness).

Smaller scale results regard bit-level considerations, such as entropy for file compression or complexity lower bounds, as well as the study of specific timing measures and their modularity properties.

The story of the study of computation is far from completed. Sciences, and especially younger ones such as computer science, beg for ever more facts to be discovered. The impressive applications of computer science, including artificial intelligence and data analytics, dominate our current world. With such impact, it is easy to forget that computer science still is in its infancy as a science. Computer science emerged from the 1920s on. The field is less than a century old, yet already making tremendous impact.

Mathematics is currently making large inroads once more into Computer Science applications through the use of statistics and calculus in deep learning approaches. The two fields are intimately linked and this link will only increase as Computer Science evolves.

The study of computation is exciting, challenging and subject to foundational work keeping up with rapid progress. Theory and practice are intricately linked. Clarifying and exploring these links will form the topic of this course.

1.4 Computer science's three C's

From the above list of problems addressed by theory of computation, it is clear that the theory broadly deals with the following three questions:



The three C's of computer science

- **Computability:** *How do we know whether a problem is computable?*
- **Complexity:** *How fast does a program run? How much memory does it use?*
- **Correctness:** *How can we tell whether a program is correct?*

Most topics in theory of computation fall under one of these three headings. Due to the concise nature of this course, we will focus on the first two C's: Computability & Complexity.

The course is not intended to be a purely “classical introduction” to the theory of computation. It is offered as a gentle introduction to the subject with the intent to help support a future independent study of additional topics in the area or as a spring board to more advanced follow-up courses.

The selection of topics is driven by a wish to clarify bridges between theory and practice, to motivate the relevance of theoretical investigations in support of practice and to gain insight in the fundamental nature of computation. To paraphrase Einstein, we will make everything as simple as possible, but not more so.

Chapter 2

Course overview

Un jour j'irai vivre en théorie parce
qu'en théorie tout se passe bien
(One day I'll live in theory, because
in theory all goes well . . .)
So let the journey begin!

Anonymous

We continue to give a more detailed outline of the course contents.

2.1 Countability

Only by counting can humans
demonstrate their independence
from computers.

Ford Prefect (Ix)

We will give an introduction to “countability.” Countability regards a classification of infinite sets that are “similarly in size” to the set of natural numbers. Countable sets—finite sets or sets that can be enumerated as an infinite sequence $(s_n)_{n \in \mathbf{N}}$ —play a key role in Computer Science. The topic forms a prerequisite to discussing computability. The set of all programs of a programming language forms a countable set. This will allow us to enumerate these programs in an infinite sequence—a handy tool in our considerations of computability.

2.2 Computability

Computing is not about computers
any more. It is about living.

Nicholas Negroponte

In this section we will focus on the question: “Is everything computable? What are the limits of computation?” It turns out there are plenty of problems that are not computable and turn up in the context of (semi-)automated debugging and in safety-critical software, aimed at determining the worst-case execution time of programs to support task scheduling.

Once we have established that there are problems that are “non-computable” and establish a sense of how the size of the computable problems scales up against the size of the non-computable ones, we move on to investigate other limits of computation, such as lower bounds on running time for comparison-based algorithms, and limits related to computable problems for which actual computations turn out to be infeasible in practice.

2.3 Comparison-based algorithms & basic complexity measures

People who analyze algorithms have double happiness. First of all they experience the sheer beauty of elegant mathematical patterns that surround elegant computational procedures. Then they receive a practical payoff when their theories make it possible to get other jobs done more quickly and more economically.

Don Knuth

The double happiness involved in the analysis of algorithms reflects a successful partnership between theory of practice.



The importance of studying ways to reduce the time necessary for completing computations and the associated need to *measure* the time involved in computations

has been realized early on.

“In almost every computation a great variety of arrangement for the succession of the processes is possible, and various considerations must influence the selections amongst them for the purpose of a calculating engine. One essential object is to choose that arrangement which shall tend to reduce to a minimum the time necessary for completing the calculation.”—Ada Lovelace.

The subject of the analysis of algorithm is a beautiful one, with many intricate results. We will focus on comparison-based algorithms as a “laboratory” in which to explore the analysis of algorithms. Comparison-based algorithms include most sorting and search algorithms.

“Sorting and searching provides an ideal framework for discussing a wide variety of important issues [design, improvement and analysis]. Virtually every important aspect of programming arises somewhere in the context of sorting and searching.” — Don Knuth

We will introduce some of fundamental time analysis techniques in the course in this context, including the main basic complexity measures of worst-case and average-case running time and links between these two measures.

We will analyze the worst-case running time and average-case running time of a variety of algorithms and learn useful techniques to simplify the analysis. These include measuring comparison-time, the identification of constant-time algorithms, a method involving “constant-time input partitions” and the important software engineering notion of modularity.

“Modularity is a property of systems, which reflects the extent to which it is decomposable in parts, from the properties of which we are able to predict the properties of the whole.” — T. Maibaum

Traditional engineering allows one to predict the weight a bridge can carry from the blue-prints of the intended construction via Newton’s calculus. In software design, no equivalent theory enabling blue-print prediction is available to determine the running time of an algorithm from source code. In the presence of modularity however, better predictions can be achieved as will be illustrated for the case of the worst-case and average-case running time measures.

2.4 Lower bounds

Nature to be commanded must be obeyed.

*Francis Bacon, Novum Organum
(1620)*

“Lower bounds” provide useful information for the algorithm designer, as lower bounds indicate ahead what the best-case scenario will be. Lower bounds help us decide whether it still makes sense to try and optimize further. Knowledge of lower bounds also lends useful insight into the nature of a problem.

2.4.1 Lower bounds on algorithmic improvement

Decision trees will be introduced to model all computation paths of comparison-based algorithms (for inputs of fixed size). Decision trees arise naturally in modelling processes with binary outcomes. This model is used to show that comparison-based algorithms satisfy complexity lower bounds. So no matter how much ingenuity you pour into their design, such algorithms will never take less computational steps than a certain lower bound. The design of efficient algorithms involves approaching this bound as closely as possible.

2.4.2 Lower bounds on compressibility

The course will introduce the notion of entropy to help determine lower bounds for file compression, dictating how far we can compress.

2.5 Computational tractability

In der Beschränkung zeigt sich erst der Meister.

(A master is revealed in their capacity to restrict)

J.W. von Goethe (1802)

This topic involves the study of computationally tractable classes—i.e. classes of problems that enable computationally feasible solutions. We will focus on one such class, consisting of so-called series-parallel data structures. The topic will be covered pending available time and serves as a contrast with computationally intractable problems (NP-complete problems).

2.6 Computational intractability: classification of computational difficulty

Experience is a hard teacher
because she gives the test first, the
lesson afterward.

Anonymous

Some problems might very well be “computable” (solvable via a program), but what if the solution is too hard to compute in practice?

Are there problems that are computable but for which no solution is known to be practical? If so, it would be good to learn to recognize them when possible. It turns out that there are a multitude of such problems. This is the realm of NP-complete problems. Progress has been made since the year 2000 on finding good heuristics to help compute such problems despite their inherent toughness (a topic beyond the material covered in this course). We will focus on learning to recognize such problems and gain insight in how they are related. The P versus NP problem is of interest in a variety of fields beyond computer science (all of which study inherently difficult computational problems). Check out the Clay Mathematics Institute Millennium Prizes for more information on this topic.

We suggest the excellent book *Algorithm Design* by Kleinberg and Tardos as a reference for the topic of polynomial time computability and the subject of NP-completeness. Topics covered may include:

Chapter 8: *NP and Computational Intractability*, Sections 8.1 - 8.4

Of these: you only need to know in detail:

- **Section 8.1:** The definition of *polynomial time reducibility* and exercises.
- **Section 8.2:** *Reduction via gadgets*
- **Section 8.3:** *Circuit satisfiability*
The 3-SAT problem, but not the reduction proof to independent set
- **Section 11.3:** *Set Cover: A General Greedy Heuristic* (Approximation Algorithm). You need to know the algorithm, but not the proof.
- **Section 8.4:** definition of an *NP-complete problem*. You need to know some *examples of NP-complete problems* (including the ones presented in 8.1 and a few extra ones).
- **Additional topic:** We may also cover a distributed algorithm to compute *Maximal Independent Set* as another type of approximation algorithm (using randomization). You need to know the algorithm and the proof.

It is important to take notes in class as the presentations may differ slightly from the material covered in the book.

2.7 Course excursions

I still think of myself as primarily a fiction writer, but I sometimes wonder what the point is now. I've already published every possible story I could write...

*Jonathan Basile, creator of the
digitized Library of Babel*

The course includes two excursions. We no longer offer a dinner at the Restaurant at the Edge of the Universe, but we do offer a close second: experiencing Czep's count-down. The course also includes a visit to the Library of Babel.

Chapter 3

A quick revision of functions and proof techniques

Functions play a special role in this course as they will represent the collection input-output pairs of programs.

We repeat some basic facts about functions.

3.1 Basic notions

We let \mathbf{N} denote the set of all natural numbers, i.e.,

$$\mathbf{N} = \{0, 1, 2, 3, 4, \dots, n, n+1, \dots\}$$

and we let \mathbf{R} denote the set of the real numbers.

Definition 1 A function f from a set A to a set B , denoted as $f: A \rightarrow B$ is set of pairs

$$f \subseteq A \times B = \{(x, y) \mid x \in A \text{ and } y \in B\}$$

where f needs to satisfy the following condition:

$$\text{if } (x, y), (x, y') \in f \text{ then } y = y'.$$

If $(x, y) \in f$, we write $f(x) = y$ and we call x an argument of f and y its image (or value), i.e. y is the image (under the function f) of the argument x .

Hence the definition can be formulated as: a function can only take one value y on its argument x .

Definition 2 We call the set A the domain of the function f , denoted by $\text{dom}(f)$ and the set B the codomain of the function f , denoted by $\text{codom}(f)$. Also, $f(A)$ denotes the set of all the images of the elements of A , i.e.

$$f(A) = \{f(x) \mid x \in A\}.$$

We refer to $f(A)$ as the image of the set A (under the function f).

Definition 3 Given a function $f: A \rightarrow B$. We define the range of f , denoted by $\text{ran}(f)$ to be the image of its domain, i.e.

$$\text{ran}(f) = f(A).$$

Consider for instance the function $f: \mathbf{R} \rightarrow \mathbf{R}: x \rightarrow x^2$ (the function that squares every real number). Then $\text{dom}(f) = \text{codom}(f) = \mathbf{R}$, but $\text{range}(f) = f(\mathbf{R}) = \{x^2 \mid x \in \mathbf{R}\} = \{x \mid x \in \mathbf{R} \text{ and } x \geq 0\}$.

Clearly for any function $f: A \rightarrow B$ we have $f(A) \subseteq B$. We can generalize this notation to subsets C of A , i.e. $C \subseteq A$. In that case, we define:

Definition 4

$$f(C) = \{f(x) \mid x \in C\},$$

i.e. the set of the images of the elements of C (under the function f). We also refer to $f(C)$ as the image of the set C (under the function f).

Definition 5 We define the inverse image of a subset C of B as the set

$$f^{-1}(C) = \{x \in A \mid f(x) = c \text{ for some } c \in C\}.$$

In other word, the inverse image of a set C (under the function f) is the set of all elements of A that are mapped to an element of C .

Definition 6 A function $f: \mathbf{N} \rightarrow \mathbf{N}$ is increasing iff for any two natural numbers x and y

$$x \leq y \Rightarrow f(x) \leq f(y)$$

3.2 The input-output function of a program

If we think about a program as a “recipe” for a computation that transform inputs to outputs then it is clear that to each program P there naturally corresponds an “input-output” function which we denote by F_P .

Some programs do not produce outputs, such as

$$[\text{While } x == x \text{ do } x := x + 1]$$

executed on input $x = 1$.

Such non-terminating programs can still be represented as functions, where in case of non-termination on an input x , the output \perp is used (to indicate non-termination).

Definition 7 (Programs computing a given function)

Given a function F , then we can interpret the function as specifying input-output pairs. The function is viewed as a a problem (or specification) to be computed. *IW* say that a program P computes the function F^1 if for every pair (x, y) in F , i.e. $y = F(x)$, the program P produces output $y = F(x)$ on input x .

¹Alternatively: computes the problem F or computes the specification F .

Example 8 (*Examples of programs computing a problem*)

1) Consider the function F that maps every strictly positive integer n to \perp (the value indicating non-termination). The program $[\text{While } x == x \text{ do } x := x + 1]$ computes the function F .

2) Consider a function F consisting of pairs of lists (L, L') where in each case L' is the sorted version of L . Hence F is the sorting problem. Every sorting program P computes the function F , i.e. $F_P = F$.

Remark 9 (*Can every function be computed by a program?*)

The question whether every function is computable has relevance for practice. For instance, consider functions that map natural numbers to natural numbers. They can be viewed as specifying an input-output relation for a program. The function $f(n) = 2n$, doubling its arguments, clearly is computable. But is the case for all functions mapping natural numbers to natural numbers, no matter how random the values taken by the function may be? We will return to this question later.

Definition 10 (*Function determined by a program*)

If the program P is deterministic and terminates on each input, then clearly every input corresponds to a single output. In function terminology: every argument (i.e. “input”) has a single value corresponding to it (i.e. “output”).

On each execution of the program P , the same output will be produced. Hence the collection of input-output pairs of P forms a function.

Thus every such program determines a function it computes (aka a problem it computes or a specification for this program), namely the input-output relation consisting of all input-output pairs. This function is called the function determined by program P and is denoted by F_P .

Combining both definitions we say that program P computes function F in case $F_P = F$.

Example: Consider a sorting program, say Bubble Sort, which we denote by B . The input-output function F_B corresponding to Bubble Sort maps each input list (L_1, \dots, L_n) to a (sorted) output list. The list $(1, 2, 2, 1, 3, 6, 5, 1)$ is mapped to the sorted list $(1, 1, 1, 2, 2, 3, 5, 6)$ by Bubble Sort.

$$F_B((1, 2, 2, 1, 3, 6, 5, 1)) = (1, 1, 1, 2, 2, 3, 5, 6).$$

Clearly, every sorting algorithm S has the function F_B as its input-output function.

For instance, the sorting algorithm Quick Sort, which we denote by Q has an input-output function F_Q that equals F_B .

Input-output functions

- To each deterministic program P , returning an output O for every input I , we can associate a unique function F_P consisting of all the input-output pairs (I, O) of P .
- Two such programs P_1 and P_2 can have the same input-output function ($F_{P_1} = F_{P_2}$). If this is the case, we say that P_1 and P_2 “compute the same problem”.

3.3 Finite sequences

Definition 11 *The set of all finite sequences consisting of natural numbers and of length k is denoted by \mathbf{N}^k . I.e.,*

$$\mathbf{N}^k = \{(n_1, \dots, n_k) \mid n_1, \dots, n_k \in \mathbf{N}\}$$

The set of all finite sequences of natural numbers is denoted by $FinSeq(\mathbf{N})$.

$$FinSeq(\mathbf{N}) = \cup_{k \in \mathbf{N}} \mathbf{N}^k = \mathbf{N}^0 \cup \mathbf{N}^1 \cup \mathbf{N}^2 \cup \dots \cup \mathbf{N}^k \cup \dots,$$

where

$$\begin{aligned}\mathbf{N}^0 &= \emptyset \text{ (the empty sequence)} \\ \mathbf{N}^1 &= \{(n) \mid n \in \mathbf{N}\} \text{ (the singleton sequences)} \\ \mathbf{N}^2 &= \{(m, n) \mid m, n \in \mathbf{N}\} \text{ (the pairs of natural numbers), etc.}\end{aligned}$$

Exercise 12 *The input set for a sorting program S , i.e. the set representing all input lists, is the set $FinSeq(\mathbf{N})$ of all finite sequences of natural numbers. The domain and the codomain for the function F_S is the set $FinSeq(\mathbf{N})$, i.e.*

$$F_S: FinSeq(\mathbf{N}) \rightarrow FinSeq(\mathbf{N})$$

- Determine the range of F_S .*
- Determine $F_S^{-1}((1, 2, 3))$.*

3.4 Finite sequences as functions

An sequence of length n , say (a_1, \dots, a_n) , can be regarded as a function, F , whose domain is the sequence’s set of element-indices, X , and whose codomain, Y , is the sequence’s set of elements.

$$X = \{1, 2, \dots, n\} \quad (3.1)$$

$$Y = \{a_1, a_2, \dots, a_n\} \quad (3.2)$$

$$F = \{(1, a_1), (2, a_2), \dots, (n, a_n)\}. \quad (3.3)$$

$$(3.4)$$

Hence:

$$(F(1), F(2), \dots, F(n)) = (a_1, \dots, a_n)$$

In Computer Science terms, we can interpret a list $L = (L[1], \dots, L[n])$ to be a function F from the array-indices $\{1, \dots, n\}$ to the array-values stored in each index location i.e. the set $\{L[1], \dots, L[n]\}$. The function F maps i to $L[i]$, i.e. $F(i) = L[i]$.

3.5 Power-notation for sets of functions

Definition 13 We call the number of elements in a (finite) set the cardinality of the set and denote the cardinality of A by $|A|$. For instance, the cardinality of the set $A = \{0, 3, 1, 5, 9\}$ is $|A| = 5$.

Given two sets X and Y . We denote the set of all functions that map X to Y by Y^X .

$$Y^X = \{f \mid f: X \rightarrow Y\}$$

The motivation behind the notation is that, for the case of finite sets X and Y , the size of the set Y^X is precisely $|Y|^{|X|}$.

We verify this as an exercise for the set $X = \{1, 2, 3\}$ and $Y = \{a, b\}$. Clearly, each function from X to Y corresponds uniquely to a selection of three values from the set Y (where these values can be repeated). In other words, the set Y^X corresponds uniquely to the set of triples $Y \times Y \times Y = \{(y_1, y_2, y_3) \mid y_1, y_2, y_3 \in Y\}$.

Hence the set Y^X has $2 \times 2 \times 2 = 2^3 = |Y|^{|X|}$ elements. Similarly, one can show for the case of finite sets X and Y that the size of the set Y^X is precisely $|Y|^{|X|}$, justifying the notation Y^X for the set of functions $\{f \mid f: X \rightarrow Y\}$. *Note that this is merely a notation, not an exponentiation operation.* The exponentiation operation only occurs in the computation of the size of the set Y^X . The notation Y^X merely serves as a reminder of this fact.

3.6 Infinite sequences as functions

As for the case of (finite) sequences, we can regard an infinite sequence $(a_n)_{n \in \mathbf{N}}$ as a function from the indices n to the values a_n .

The sequence $(a_n)_{n \in \mathbf{N}}$ is defined to be the function $F: \mathbf{N} \rightarrow A$, where $A = \{a_n \mid n \in \mathbf{N}\}$ and for all natural numbers n

$$F(n) = a_n.$$

Example 14 Consider the function $F: \mathbf{N} \rightarrow \mathbf{N}: n \rightarrow 2n$, doubling natural numbers. It maps each natural number n to the even number $2n$. The corresponding sequence $(a_n)_{n \in \mathbf{N}}$ is the infinite sequence listing the even natural numbers: $(a_n)_{n \in \mathbf{N}} = (0, 2, 4, 6, 8, 12, \dots, 2n, \dots)$, where $a_n = 2n$.

From here on we won't distinguish between infinite sequences $(a_n)_{n \in \mathbf{N}}$ and functions $F: \mathbf{N} \rightarrow A$ where $A = \{a_n | n \in \mathbf{N}\}$. One determines the other, the sequence $(a_n)_{n \in \mathbf{N}}$ determines the function

$$F: \mathbf{N} \rightarrow A: n \rightarrow a_n$$

and a function $F: \mathbf{N} \rightarrow A$ determines the sequence $(a_n)_{n \in \mathbf{N}}$ defined for all natural numbers n

$$a_n = f(n).$$

Using the notation introduced in Section 3.5, identifying sequences and functions means that in the following we will identify the set of sequences $\{(a_n)_{n \in \mathbf{N}} | a_n \in A\}$ and the set of functions $A^{\mathbf{N}}$.

$$A^{\mathbf{N}} = \{(a_n)_{n \in \mathbf{N}} | a_n \in A\}$$

3.6.1 Infinite binary sequences

In the following we will be particularly interested in the set of all binary sequences taking only the values 0 and 1. We follow the mathematical convention to denote the set $\{0, 1\}$ by the symbol 2—indicating a two element set. Again, this notation is a little sloppy as the number 2 only comes in to play when we *count* the number of elements in the set $A = \{0, 1\}$. However, the standard notation is that the set of all infinite binary sequences is denoted by $2^{\mathbf{N}}$, meaning the set of all functions from the set of natural numbers \mathbf{N} to the values 0 and 1.

3.7 Injections, surjections and bijections

Definition 15 A function $f: A \rightarrow B$ is an injection (“one-to-one”), in case for any two elements x, y in the set A

$$x \neq y \Rightarrow f(x) \neq f(y)$$

In other words, the function f never maps two different values to the same value. “Individuality is preserved.”

Equivalently

for any two elements x, y in the set A

$$f(x) = f(y) \Rightarrow x = y$$

Definition 16 A function $f: A \rightarrow B$ is a surjection (“onto”) in case for any element y in the set B there is an element x in the set A such that

$$f(x) = y$$

In other words, each element of B is reached as the image of some element in A , i.e. no elements of B are left out (are “redundant”) when all elements of A are mapped to elements in B .

Exercise 17 Consider the input-output function F_B of Bubble sort. Show that:

- a) $F_B: \text{FinSeq}(\mathbf{N}) \rightarrow \text{FinSeq}(\mathbf{N})$ is not an injection.
- b) $F_B: \text{FinSeq}(\mathbf{N}) \rightarrow \text{FinSeq}(\mathbf{N})$ is not a surjection.
- c) Define SORT to be the set of all finite sequences that are in sorted order, i.e.

$$\text{SORT} = \{(a_1, \dots, a_k) \mid a_1 \leq a_2 \leq \dots \leq a_k\} \text{ and } k \in \mathbf{N}\}$$

Show that the function $F_B: \text{FinSeq}(\mathbf{N}) \rightarrow \text{SORT}$ is a surjection.

Definition 18 A bijection is an injective function (“one-to-one”) function that is also a surjection (“onto”).

Consider a set $B = \{a, b, c, d, e\}$. Clearly, $|A| = |B| = 5$ and

$$f = \{(0, a), (1, b), (3, c), (5, d), (9, e)\}$$

is a bijection between the two sets.

This is true in general, i.e. if there exists a bijection between two finite sets, then the two sets have the same cardinality, i.e. these finite sets have the same number of elements. Our three notions, injection, surjection and bijection are handy in the context of size comparisons of finite sets (and as we will see in a moment, in size comparisons of infinite sets too).

Exercise 19 (cardinality exercises)

- 1) Given two finite sets A and B and an injective function $f: A \rightarrow B$. Verify that $|A| \leq |B|$, i.e. B must have at least as many elements as A .
- 2) Given two finite sets A and B and a surjective function $f: A \rightarrow B$. Verify that $|B| \leq |A|$, i.e. A must have at least as many elements as B .
- 3) Two finite sets A and B have the same cardinality $|A| = |B|$ if and only if there exists a bijection $f: A \rightarrow B$.

Clearly 3) follows from 1) and 2). We show that 1) and 2) hold true as part of the tutorials.

The need to consider the infinite case

To compute we need to consider finite sets as well as infinite sets. Of course, all data that we deal with have a finite representation (this is clear since our hardware is finite, i.e. we ultimately store information as say a number of bytes). However, programs are designed to compute over potentially infinite data structures (even though each part of the computation will be finite). For instance, a program that computes whether a natural number is even or odd, takes natural numbers n as inputs and returns the value 0 if the input n is even and the value 1 in case the input n is odd. Clearly, the program can take any natural number as input and hence the set of its inputs is \mathbf{N} , the set of all natural numbers. This is an infinite set—an infinite data structure².

When dealing with infinite data structures it is still useful to be able to “count” or at least to “compare” size. For finite data structures, this can be easily achieved through the notion of cardinality, counting the number of elements in a set. For infinite sets we don’t have this possibility. However, we can still *compare* sizes in a way that is similar to the finite case, relying on the notion of a bijection. This approach is an age-old method known to sheep herders.

Early sheep herders relied on bijections to keep track of the size of their flock (without knowing the actual number of sheep in their flock)

Sheep herders can keep track of the size of their flock without necessarily knowing how many sheep are exactly part of it (i.e. the method could be used pre-counting).

For this purpose herders keep a large sack of pebbles near the entrance of the sheep pen. As sheep leave the pen in the morning, for each passing sheep, the herder will take one pebble from the sack and put it in a bowl. Once the last sheep has left the pen, the pebbles in the bowl form a bijection with the flock. When the sheep return, the herder simply removes a pebble from the bowl, returning it to the sack. When the bowl is empty, the herder knows all his sheep have returned (we assume the herder can distinguish arriving sheep from their neighbour’s strays). If some pebbles are left, it is time to whistle for the dog and go check.

The point of this observation is that the herder does not need to know how many cheep are in the pen. Checking the flock’s size simply can be carried out through a bijection, assigning one stone per animal. The same holds for infinite sets, i.e. infinite sets can be shown to have equal size (“have same cardinality”) by establishing a bijection between them³.

²Some programming languages, in particular functional ones such as Haskell will actually support computations over “infinite data structures”.

³Note that George Cantor did develop a theory that allows one to compute “sizes” for infinite sets via cardinal arithmetic—a topic beyond the scope of this course.



Cardinality

Definition 20 We say that two sets A and B have the same cardinality if and only there exists a bijection $f: A \rightarrow B$. We denote this, as for the finite case, by $|A| = |B|$.

Remark 21 The notation $|A| = |B|$ is merely a formal notation. Contrary to finite sets (where $|A|$ stands for the number of elements of the set A) we do not attach a number interpretation to the notation $|A| = |B|$. This notation, for infinite sets, simply means that there exists a bijection between these sets, nothing more.⁴

We also introduce the notation $|A| \leq |B|$ and $|A| \geq |B|$ for infinite sets. Intuitively these mean that the set A is considered to be smaller (or equal to) in size than the size of set B . But this is only an intuitive interpretation. We do not measure sizes of infinite sets, we merely compare them, as in “one set is larger in size than the other” (without computing what the size for each set is, as we would for finite sets by counting the number of elements).

The following definition is inspired by Exercise 19.

Definition 22 Given two infinite sets A and B , the notation $|A| \leq |B|$ means that there exists an injection f from A into B . The notation $|A| \geq |B|$ means that there exists a surjection g from A onto B .

Remark 23 Note that the definitions makes sense. Clearly we would like to conclude from $|A| \leq |B|$ and $|B| \leq |A|$ that $|A| = |B|$. We can't do this right away as these notations are mere formal notations. Their meaning is: there exists an injection from A into B and there exists a surjection from A onto B . But these facts combined mean that there exists a bijection from A to B and thus $|A| = |B|$.

We will consider the notation $|A| \geq |B|$ and $|B| \leq |A|$ to be equivalent. The same holds for the notation $|A| \leq |B|$ and $|B| \geq |A|$.

⁴The notation is similar to for instance the notation $O(f) = O(g)$, which means that the two functions f and g are of the same big-Oh order.

3.8 The Holmes Principle (proof by way of contradiction)

The Holmes Principle

“After eliminating all possibilities that cannot occur,
whatever remains and however unlikely,
must be accepted as the truth.”
– Sherlock Holmes

The Holmes Principle embodies the main idea underlying a proof “by way of contradiction”. We give a few examples, one tongue-in-cheek, another showing that there are infinitely many prime numbers.

On boring numbers

| **Theorem:** There is no such thing as a boring number.

We prove this by way of contradiction. Assume that there exist one or more boring numbers—numbers with no special characteristics. Let n be the smallest one. However, now n is interesting because it is the smallest non-interesting number—a contradiction. Hence there are no boring numbers.

On prime numbers

| **Theorem:** There are infinitely many prime numbers.

Again, we prove the result by contradiction. Assume that there are finitely many prime numbers, say p_1, \dots, p_k . Consider the number $p = (p_1 \times \dots \times p_k) + 1$. None of the primes p_1, \dots, p_k divides it (the remainder is always 1). Hence p is only divisible by 1 and p . Thus p is prime. This is a contradiction since p is greater than any of the finitely many prime numbers p_1, \dots, p_k hence is not prime⁵.

Remark 24 *The result is significant for cryptography. Primes are commonly used to secure public-key encryption systems. A common practice is to use the result of the multiplication of two prime numbers as the number securing the encryption. The RSA encryption algorithm is used in secure commerce web sites. RSA is based on the fact that it is easy to take two (very large) prime numbers and multiply them, but it is extremely hard to do the opposite, i.e. to take a very large number that has only*

⁵A number $p = p_1 \times \dots \times p_k + 1$ for which p_1, \dots, p_k are primes is not always prime. Consider $p = (2 \times 3 \times 5 \times 7 \times 11 \times 13) + 1 = 30031 = 59 \times 509$.

two prime factors, and find these two factorrrs. The existence of infinitely many prime numbers means we can always compute larger prime numbers when needed, improving cryptographic encoding.

Chapter 4

Countability

Only by counting can humans demonstrate their independence from computers.

Ford Prefect (Ix)

In order to talk about “computability” we first introduce the notion of “countability” (we need to be able to “count” before we can “compute”).

4.1 Countable sets

Note that in Definition 20 the sets A and B are infinite. In computer science we typically deal with one type of infinite sets, the so-called “countable sets, i.e. the sets that have the same cardinality as the set of the natural number \mathbf{N} . In other words: the sets that are in bijection with the set of natural numbers \mathbf{N} ”¹.



Countable set

Definition 25 A countable set A is a set that is finite or a set that has the same cardinality as \mathbf{N} (in other words, there is a bijection from \mathbf{N} to A).

Enumeration of countable sets

A countable set A which is not finite is hence in bijection with the set of natural numbers \mathbf{N} . In other words: $|A| = |\mathbf{N}|$. Intuitively, these two infinite sets are considered to have the same “size”, i.e. they have the same cardinality. Again, this merely

¹Of course, the real numbers, which form a “non-countable” set, play a role too. As we only ever can deal with finite data in reality in a computer science context, real numbers form a special case, treated through suitable approximation by finite data.

means there exists a bijection between these two sets. As for the case of the sheep herders, this only gives us a sense that the two collections can be considered equal in size (without knowing what this size actually is, i.e. no counting occurs).

Intuitive interpretation of countability

Intuitively, a countable set is either finite or infinite, where the infinite size is “essentially the same” as the size of \mathbf{N} the set of natural numbers. “Essentially the same” is expressed through the notion of a bijection, i.e. cardinality.

Another way to view countable sets is that they essentially are sets that can be enumerated as a finite or infinite sequence of elements.

Countability and enumeration

Countable infinite sets can be “enumerated” as an infinite sequence. Indeed, consider the bijection $f: \mathbf{N} \rightarrow A$. This bijection “enumerates” the set A since A can now be written as the set $\{f(0), f(1), f(2), \dots, f(n), \dots\}^a$. This set corresponds to the infinite sequence $(f(0), f(1), f(2), \dots, f(n), \dots)$. **Hence every infinite countable set can be enumerated via an infinite sequence.**

^aRecall that functions with the natural numbers as domain are the same as infinite sequences, see Section 3.6.

We consider a few examples to make this clear.

4.2 Examples

1) The set $A = \{0, 3, 1, 5, 9\}$ is a countable set (as it is finite). So is the set $B = \{a, b, c, d, e\}$.

2) The set E of even natural numbers is a countable set since

$$f: \mathbf{N} \rightarrow E: n \rightarrow 2n$$

is a bijection.

$$f = \{(0, 0), (1, 2), (2, 4), (3, 6), \dots\}$$

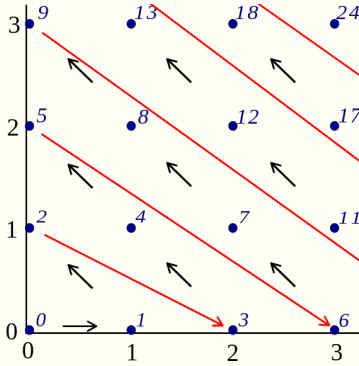
3) The set $\mathbf{N} \times \mathbf{N}$ of ordered pairs of natural numbers,

$$\mathbf{N} \times \mathbf{N} = \{(k, l) \mid k, l \in \mathbf{N}\}$$

is a countable set.

One way to verify this is to “zig-zag”-enumerate the pairs as follows²:

²Image source wikipedia



We define the corresponding bijection f :

a) Value of f at the origin

$$f(0,0) = 0$$

b) The values of f on the x-axis

for $n > 0$ we define

$$f(n,0) = f(n-1,0) + n$$

c) The values f for any given n , corresponding to following the black arrows up the diagonal, starting from $f(n,0)$ on the x -axis and working your way up to the y -axis. For any $k \in \{1, \dots, n\}$

$$f(n-k, k) = f(n,0) + k$$

Exercise 26 Show that:

$$f(n,0) = \frac{n(n+1)}{2}$$

and hence for $k \in \{1, \dots, n\}$

$$f(n,k) = \frac{n(n+1)}{2} + k$$

4) The set of integers \mathbf{Z} is countable. Consider the bijection $h: \mathbf{N} \rightarrow \mathbf{Z}$ such that for n even

$$h(n) = \frac{n}{2}$$

and for n odd

$$h(n) = -\lceil \frac{n}{2} \rceil$$

Ranking elements

The notion of a “rank function” r is useful to determine the order of natural numbers in a finite set A .

Rank

The *rank* of a natural number n in a set A , denoted by $\text{rank}(n)$ is defined to be 0 if no natural numbers smaller than n occur in A . Otherwise, the rank of a natural number n in A is defined to be the number of elements smaller than n in A .

For instance, consider the set $A = \{6, 3, 9\}$. The rank of 3 is 0, the rank of 6 is 1 and the rank of 9 is 2.

“Ranking” is tightly related to sorting elements in a list. Sorting elements essentially determines the rank of each element in an unordered list. For instance, for the list $(6, 3, 9)$, the sorted list produced by a sorting algorithm is $(3, 6, 9)$, where the element 3 with rank 0 is placed in position 0, 6 with rank 1 is placed in position 1 and 9 with rank 2 is placed in the final position.

The following result provides a useful alternative to show that sets are countable.

4.3 Characterization of countability

Characterization of countability

Theorem 27 *Given a set S . The following statements are equivalent:*

- 1) S is countable, i.e. either S is finite or there exists a bijection $h: \mathbf{N} \rightarrow S$.
- 2) there exists an injection $f: S \rightarrow \mathbf{N}$.

Remark 28 *Recall that for an injection $f: S \rightarrow \mathbf{N}$, $|S| \leq |\mathbf{N}|$. In other words the “size” of S is bounded by the “size” of \mathbf{N} . Hence, loosely stated, S must be of the same size as \mathbf{N} or finite and “hence” S is countable. This constitutes a proof for the case of finite sets for which the notion of size corresponds to the number of elements in the set. For infinite sets, a proof needs to use the definition of cardinality, i.e. the notion of a bijection, which is the approach followed below.*

Proof: We first show that 1) implies 2). Let S be a countable set. First assume that S is finite, of size k . Clearly there exists an injection from S into \mathbf{N} mapping the k elements of S to the first k natural numbers. If there exists a bijection $h: \mathbf{N} \rightarrow S$

as in 1), then define f to be $h^{-1}: S \rightarrow \mathbf{N}$, which is a bijection and hence an injection. Hence 2) follows.

To prove the converse, assume that 2) holds, i.e. there is an injection f from S to \mathbf{N} . If S is finite, we are done (i.e. 1) holds). Otherwise, S is infinite. In this case we can “shove” the non-consecutive images of elements of S to the left until they are consecutive. Indeed, consider the image of S under f

$$f(S) = \{f(x) \mid x \in S\} \subseteq \mathbf{N}$$

Compose the injective function f with the rank function r . In this composition $r \circ f$ the rank function r “shoves” the numbers to the left so they occur in consecutive order.

Finally note that $r \circ f$ is a bijection, as required. Indeed, since the set S is infinite, clearly the ranks obtained consist exactly of all elements in \mathbf{N} (so $r \circ f$ is surjective). No two elements from A obtain the same rank since f is injective and so is the rank function r hence $r \circ f$ is injective.

To obtain the bijection required for 2), simply take the inverse of the bijection $r \circ f$, i.e. $(r \circ f)^{-1}$ which maps \mathbf{N} to S , as required.

□

We can apply the previous results to derive some useful facts on countable sets.

1) We indicate a second way to show that **the set $\mathbf{N} \times \mathbf{N}$ of ordered pairs of natural numbers is countable**. Use a trick based on prime number factorisation (which will come in handy in example 5 as well). Use the injection

$$g: \mathbf{N} \times \mathbf{N} \rightarrow \mathbf{N}: (m, n) \rightarrow 2^m 3^n.$$

2) **The set \mathbf{N}^k of all k -tuples (n_1, \dots, n_k) of natural numbers is countable**. Proof (using “Gödel encoding”): apply 1) repeatedly to a k -tuple or generalise the injection g to

$$g: \mathbf{N}^k \rightarrow \mathbf{N}: (m_1, \dots, m_k) \rightarrow p_1^{m_1} p_2^{m_2} p_3^{m_3} \dots p_k^{m_k}$$

where we use the first k primes $p_1 = 2, p_2 = 3, p_3 = 5, \dots, p_k = k$ -th prime.

3) **The cartesian product $A \times B$ of two countable sets A and B is countable**. (Recall that $A \times B = \{(a, b) \mid a \in A \text{ and } b \in B\}$.)

Since A and B are countable, there are injections $f: A \rightarrow \mathbf{N}$ and $g: B \rightarrow \mathbf{N}$. Define

$$f \times g: A \times B \rightarrow \mathbf{N} \times \mathbf{N}$$

to be the function that maps the pair (a, b) to the pair $(f(a), g(b))$.

This is clearly an injection, as f and g are.

By Example 4.2 part 3, $\mathbf{N} \times \mathbf{N}$ is countable, hence there exists a injection h from $\mathbf{N} \times \mathbf{N}$ to \mathbf{N} . The composition $h \circ (f \times g)$ maps $A \times B$ to \mathbf{N} , hence $A \times B$ is countable.

4) From repeated application of 3) we obtain that **the cartesian product $A_1 \times \dots \times A_k$ of k countable sets is countable.**

5) **The set $PROG(\mathcal{L})$ of all programs of a general purpose programming language³ \mathcal{L} is countable.**

We need to find an injection from the set of all programs (written in your favourite general purpose programming language) to the set of natural numbers \mathbf{N} . This is achievable by an encoding known as “*Gödel numbering*” similar to the encoding presented in part 2 above. First observe that all programs in your language are simply finite collections of symbols. The trick to “enumerate” all programs, is to replace each symbol in your program by a natural number, then encode all these numbers together again into a single natural number.

Say we assign a natural number to each character in your programming language. For this we can for instance rely on ASCII encoding of characters. Each program hence corresponds to a sequence $(x_1, x_2, x_3, \dots, x_n)$ of positive integers (the integers replacing each character in your finite program). The Gödel encoding of the sequence is the product of the first n primes p_i raised to their corresponding character-value x_i in the sequence.



Gödel encoding

$$\text{Göd-Enc}(x_1, x_2, x_3, \dots, x_n) = 2^{x_1} \cdot 3^{x_2} \cdot 5^{x_3} \dots p_n^{x_n}$$

According to the fundamental theorem of arithmetic, any number can be uniquely factored into prime factors. So it is possible to recover the original sequence from its Gödel number.

This encoding allows us to construct an injection, namely Göd-Enc, from the set of all programs of a given language into the set of natural numbers, demonstrating that the set of all programs of a language is countable. We will return to this useful observation below.

For now, we observe the following corollary.

³A general purpose programming a language is a language capable of computing all Turing computable functions.



Computer Science's prime example of a countable set

The set $PROG(\mathcal{L})$ of all programs of a programming language \mathcal{L} is a countable infinite set and hence enumerable via an infinite sequence $(P_n)_{n \in \mathbb{N}}$.

Note 1: The Göd-Enc function encodes all programs in your chosen general-purpose programming language (Java, Python, C, ...) by a unique natural number. This injection of the set of all programs into the natural numbers can be replaced by a bijection by “shoving” indices to the left to remove gaps. Compose the rank function with the injection (see the proof of part 2 of Theorem 27).

Note 2: Our enumeration of programs may seem a little unsatisfactory at first. Indeed, it assumes that we simply “shove” the indices of programs “to the left” to obtain our final enumeration. No program is given to achieve this. In fact, in general computability theory, it is perfectly possible to give an explicit encoding of programs that assigns to each program exactly one natural number such that there are no gaps in the enumeration. The encoding constitutes of an explicit bijection between the set of natural numbers and the set of all programs for which the values can be computed. The treatment is a little more technical and omitted here. This so-called “effective enumeration” is an important part of computability theory⁴. For our current purposes it is sufficient to know that an enumeration of programs exists (on the back of our “shove-to-the-left” ranking argument).

Note 3: The Gödel encoding of the programs of a programming language is an example of encryption. Each program is encrypted to a single natural number. From this natural number we can decode the original program. Hence if all programs of a programming language are enumerated via an infinite list: $P_0, P_1, \dots, P_n, \dots$ then each program P_i in this list can be decoded from its index i . I.e. given a natural number i , we can always find back the program that it encodes (assuming we follow the approach described under Note 2, i.e. we do not apply ranking, but an effective enumeration.).

6) **The set \mathbb{Q} of the rational numbers is countable.** Note that $\mathbb{Z} \times \mathbb{N}$ is countable (where we use 1). Solve exercise 29 part 5. Then define the surjection $g: \mathbb{Z} \times \mathbb{N} \rightarrow \mathbb{Q}$ as follows: $f(k, l) = \frac{k}{l+1}$.

Cantor has shown that the set of real numbers \mathbb{R} is *not* countable. We won't show this in this course, but we will show a related result which will be of use in our study of computability: the set of all infinite binary sequences is not countable.

⁴See Cutland's book following the completion of the course for more information about this topic. N. J. Cutland, *Computability: An Introduction to Recursive Function Theory*.

Exercise 29 (on countable sets and Gödel encoding)

1) Show that every subset of a countable set must be countable.

2) Given two infinite sets A and B , where A is countable. Show that if there exists a bijection from A to B then B must be countable.

3) Display (but do not compute) the Gödel encoding for the final line in the pseudo-code of the algorithm Bubblesort (where L is the input list and $|L|$ denotes the length of L). You may assume that each symbol s in the code is encoded by a positive integer $c(s) \in \mathbb{N}$.

- Each lowercase letter of the alphabet receives its number in the alphabetical order, i.e. $c(a) = 1, \dots, c(z) = 26$
- Upper case letters receive the numbers $27, \dots, 52$ respectively, i.e. $c(A) = 27, \dots, c(Z) = 52$
- We use 0 as a separation symbol (for blanks)
- 53 denotes $|$, 54 denotes $>$, 55 denotes $[$, 56 denotes $]$, 57 denotes $($, 58 denotes $)$, 59 denotes $+$, 60 denotes $-$ and 61 denotes the equality sign $=$
- Finally, 62 denotes 1 and 63 denotes the comma $,$

This is sufficient to encode Bubble Sort. Obviously, for a general language (including the one specifying pseudo code) we would need a code for each symbol of the language).

To solve this exercise, use the table below listing the first 63 prime numbers.

2	3	5	7	11	13	17	19	23	29
31	37	41	43	47	53	59	61	67	71
73	79	83	89	97	101	103	107	109	113
127	131	137	139	149	151	157	163	167	173
179	181	191	193	197	199	211	223	227	229
233	239	241	251	257	263	269	271	277	281
283	293	241							

Display the Gödel encoding for the final line of the pseudo-code of Bubblesort, namely: **swap(L[j],L[j+1])**

Bubblesort(L)

```
for i = |L| - 1 downto 1 do
  for j = 1 to i do
    if L[j] > L[j + 1] then
      swap(L[j], L[j + 1])
```


4) The programs of any programming language can be assumed to take natural numbers as inputs and yield natural numbers as outputs. Why is this the case? Consider for instance a program that takes binary trees as inputs and produces binary trees as outputs. How could you replace it with an equivalent program that computes over natural numbers?

5) Let A be a countable set and suppose that there exists a surjective function $f: A \rightarrow B$. Prove that B is also countable.

6) Show that the set of \mathbf{N} of natural numbers can be represented as a union of an infinite number of disjoint infinite sets.

4.4 Cantor's world: exponentiation and powersets

The difference between the poet and the mathematician is that the poet tries to get his head into the heavens while the mathematician tries to get the heavens into his head.

G.K. Chesterton

4.4.1 powersets in Computer Science

The powerset of a set A is the set of all subsets of the given set. Consider the set $A = \{1, 2, 3\}$. The powerset of A , denoted by $P(A)$ is given by:

$$P(A) = \{\emptyset, \{1\}, \{2\}, \{3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\}\}$$

The powerset in this example counts 8 elements. In general:

The powerset of a finite set A with n elements has 2^n elements.

The notion of a powerset hence is tightly linked to exponentiation. Exponentials correspond to fast growth. This is illustrated by the classical chess-board problem: if a chessboard were to have wheat placed upon each square such that one grain were placed on the first square, two on the second, four on the third, and so on (doubling the number of grains on each subsequent square), how many grains of wheat would be on the chessboard at the finish?⁵ Other examples of exponential growth include for

⁵Answer: 18,446,744,073,709,551,615. See “Telescopic Sums” in this course, which yields the answer: $2^{64} - 1$ —a rice heap larger than Mount Everest, around 1,000 times the global production of rice in 2010 (464,000,000 metric tons). Reference: “*World rice output in 2011 estimated at 476 million tonnes: FAO*”. The Economic Times. New Delhi. 2011-06-24.

instance bacterial growth or nuclear chain reactions. Exponential processes typically grow “too fast to be contained.”

This also holds in computer science. Exponential growth in the size of a data set means we run out of storage. Exponential growth in the running time of a program (in function of the size of its inputs) means that we run out of time to complete the execution, even for relatively small input sizes.

powersets $P(A)$, growing exponentially in the size of the underlying set A , are at the root of computational obstacles. Writing a program that will enumerate all subsets of a given set is a futile task in general as any such program will produce outputs (the elements of the set $P(A)$) that grow exponentially in terms of the size of the input set A .

Our later treatment of NP-complete problems will show that the search for subsets (with certain properties) of a given set will typically lead to a computationally intractable problem. Many graph problems require the search of subgraphs with certain properties (for instance independent subsets of the graph—subsets for which no pair of nodes is linked by an edge). Such problems typically are “NP-complete”—computationally infeasible at this point in time. The root problem for many such problems is that they “skate too close to search over exponential sized spaces.”

In the following sections we will show that powersets also lie at the root of the existence of non-computable problems. The root cause here is, once again, that powersets are linked to exponentiation⁶.

4.4.2 Lattice representation of powersets

We recall the definition of a powerset.

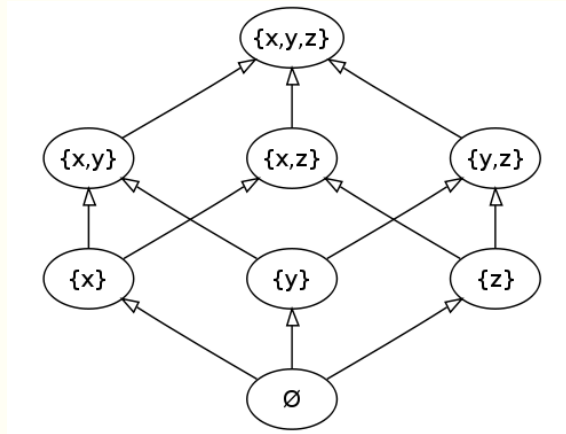
⁶Revise our discussion of Archimedes’ discoveries on power laws and exponentiation at the outset of the course

powerset

Definition 30 The powerset $P(A)$ of a set A is the set of all subsets of A , i.e.

$$P(A) = \{B \mid B \subseteq A\}.$$

powersets of finite sets can be neatly displayed by a lattice structure. The powerset of the three element set $\{x, y, z\}$ is displayed⁷ below:



Remark 31 The number of subsets with k elements in the powerset of a set with n elements is given by the number of combinations, $C(n, k) = \frac{n!}{k!(n-k)!}$, i.e. the binomial coefficients. For example, the powerset of a set $A = \{x, y, z\}$ has: $C(3, 0) = 1$ subset with 0 elements (the empty subset), $C(3, 1) = 3$ subsets with 1 element (singleton subsets), $C(3, 2) = 3$ subsets with 2 elements (the complements of the singleton subsets) and $C(3, 3) = 1$ subset with 3 elements (the set A).

The set $A = \{x, y, z\}$ has $8 = 2^3$ subsets. In general, the powerset $P(A)$ of a finite set A has $2^{|A|}$ subsets.

4.4.3 Binary sequences encode powersets

In the following we will show that **powersets and binary sequences are intimately related**. There are 2^n powersets of a set A of size n . There are 2^n binary sequences of size n . This is not a coincidence since binary sequences encode powersets.

⁷Image source: wikipedia

Binary subset-encoding: the finite case

Binary subset encoding

The binary subset encoding establishes a bijection between subsets of A and binary sequences of length $|A|$.

For a finite set $A = \{x_1, x_2, \dots, x_n\}$, encode each subset B of A as a binary number b_1b_2, \dots, b_nb_n where for each bit b_i ($i \in \{1, \dots, n\}$)

$$b_i = 1 \text{ exactly when } x_i \in B \text{ and } 0 \text{ otherwise.}$$

Example 32 For instance, the subset $B = \{x, z\}$ of the set $A = \{x, y, z\}$ is encoded by 101. The subset $\{y\}$ is encoded by 010 and the empty set is encoded by 000.

The bijective binary subset encoding implies the following result.

Exponential cardinality of powersets

Theorem 33

$$\text{For any finite set } A : |P(A)| = 2^{|A|}$$

Exercise 34 Consider the set $A = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17\}$. Let b be the binary sequence $b = (0, 1, 1, 0, 1, 0, 1, 0, 0, 0, 1, 0, 1, 0, 0, 0, 1)$. Which subset does the binary sequence b encode? What property do the elements of this subset share?

Recursive computation of the powerset of a finite set

The exponential cardinality of a powerset means that any algorithm generating the subsets of a given set will take exponential time (in terms of the size of this set). Such algorithms take too long to execute when the size of the set grows large. The bijection between the powerset of a finite set A (of size n) and the set of all binary sequences of size n means that we can generate the powerset from the given input set, or first generate the binary sequences of size n instead and then decode these to the subsets of A they represent.

We present a recursive algorithm computing the powerset of a set.

Define the element-addition operation *Add* which adds the element x to each set in a set Y (all of whose elements are sets):

$$\text{Add}(x, Y) = \{A \cup \{x\} \mid A \in Y\}$$

Add extends every set belonging to Y with the element x .

For a finite set A the following recursive algorithm $\text{Pow}(A)$, taking as input the finite set A , calculates the powerset $P(A)$.

powerset: recursive computation

Pow(A)

If $A = \emptyset$ then return $Pow(A) = \{\emptyset\}$
 otherwise let x be any element of A and let $A' = A - \{x\}$
 return $Pow(A) = Pow(A') \cup Add(x, Pow(A'))$

The powerset of a non-empty set is formed by taking all the subsets of the set containing some specific element and all the subsets of the set not containing that specific element.

Example (recursive computation of the powerset)

$$\begin{aligned}
 Pow(\{1, 2, 3\}) &= Pow(\{2, 3\}) \cup Add(1, Pow(\{2, 3\})) \\
 &= Pow(\{3\}) \cup Add(2, Pow\{3\}) \cup Add(1, Pow(\{3\}) \cup Add(2, Pow\{3\})) \\
 &= Pow(\emptyset) \cup Add(3, Pow(\emptyset)) \cup Add(2, Pow(\emptyset) \cup Add(3, Pow(\emptyset))) \\
 &\quad \cup Add(1, Pow(\emptyset) \cup Add(3, Pow(\emptyset)) \cup Add(2, Pow(\emptyset) \cup Add(3, Pow(\emptyset)))) \\
 &= \{\emptyset, \{3\}\} \cup \{\{2\}, \{2, 3\}\} \cup Add(1, \{\emptyset, \{3\}\} \cup \{\{2\}, \{2, 3\}\}) \\
 &= \{\emptyset, \{3\}, \{2\}, \{2, 3\}, \{1\}, \{1, 3\}, \{1, 2\}, \{1, 2, 3\}\}
 \end{aligned}$$

Thus far we have considered finite sets. We will extend the binary subset-encoding to the case of the powerset $P(\mathbf{N})$.

Binary subset-encoding: the infinite case

Remark that, by generalizing the binary subset-encoding for the case of finite sets, we establish a bijection between $P(\mathbf{N})$, the set of all subsets of natural numbers, and the set of all the set *infinite*⁸ binary sequences.

Binary subset encoding: the infinite case

A subset A of \mathbf{N} is encoded via the infinite binary sequence $(b_0, b_1, b_2, \dots, b_n, \dots)$ where for each $b_i = 1$ if and only if $i \in A$, and $b_i = 0$ otherwise.

Example 35 Consider the subset O of \mathbf{N} consisting of the odd natural numbers. The subset O can be encoding by the infinite binary sequence: $(0, 1, 0, 1, 0, 1, \dots)$. Similarly, the sub set E , consisting of the even natural numbers, can be encoded via the complemented⁹ binary sequence $(1, 0, 1, 0, 0, 0, \dots)$.

⁸By “infinite” we mean countably infinite.

⁹Complementation in this context refers to “flipping all bits b ” via the complementation operation $1 - b$.

The set \mathbf{N} of all natural numbers is encoded by the constant sequence $(1, 1, 1, \dots)$ while the empty set is encoded by the complement sequence $(0, 0, 0, \dots)$.

As we have seen, the set of all infinite binary sequences can be interpreted as the set of all subsets of the set of natural numbers.

The set of binary sequences can also be interpreted as the set of all bit-valued functions over the natural numbers.

We recall the definition of the set $2^{\mathbf{N}}$ from Section 3.6.



The set $2^{\mathbf{N}}$

Definition 36 The set $2^{\mathbf{N}}$ is defined as the set of functions mapping natural numbers to the values 0 and 1:

$$2^{\mathbf{N}} = \{f: \mathbf{N} \rightarrow \{0, 1\}\}.$$

Bit-valued functions are called decision functions. For each natural number n the binary function value $f(n)$ decides whether the number n is in a given set (function value 1) or not (function value 0).

$$A_f = \{n \mid f(n) = 1\}$$

is the set corresponding to a decision function f .

We have interpreted decision functions as subsets of the natural numbers. Decision functions also determine infinite binary sequences.



The set $2^{\mathbf{N}}$ of infinite binary sequences

As discussed in Section 3.6, we interpret the set $2^{\mathbf{N}}$ as the set B of the infinite binary sequences

$$B = \{(s_n)_{n \in \mathbf{N}} \mid s_n = 0 \text{ or } s_n = 1\}$$

There is a bijection between $2^{\mathbf{N}}$ and the set of all infinite binary sequences, denoted by $\text{Seq}_2(\mathbf{N})$, mapping a function f in $2^{\mathbf{N}}$ to the binary sequence

$$f(n)_{n \in \mathbf{N}} = (f(0), f(1), f(2), \dots, f(n), \dots)$$

Remark 37 In the following we won't distinguish between the sequence set $\text{Seq}_2(\mathbf{N})$ and the function set $2^{\mathbf{N}}$ since infinite sequences typically are defined as functions. Hence we will refer to the decision functions contained in $2^{\mathbf{N}}$ as “infinite binary sequences” and to the set $2^{\mathbf{N}}$ as “the set of infinite binary sequences”. Equally so, we

will refer to decision functions as “subsets of the natural numbers”. Hence we identify the elements of the function set $2^{\mathbf{N}}$, the sequence set $\text{Seq}_2(\mathbf{N})$ and the powerset $P(\mathbf{N})$ without explicitly referring to the (trivial) bijections between these sets.

Example 38 Consider the function $f \in 2^{\mathbf{N}}$ which maps the even numbers to 0 and the odd numbers to 1, i.e. $f(2n) = 0, f(2n + 1) = 1$ for any natural number n . The function f thus consists of the pairs: $(0, 0), (1, 1), (2, 0), (3, 1), (4, 0), (5, 1), \dots$. This set of pairs uniquely determines a sequence $(s_n)_{n \in \mathbf{N}}$, where $s_n = f(n)$. For our example, the sequence $(s_n)_{n \in \mathbf{N}}$ corresponding to the function f is the infinite alternating sequence consisting of all the values f reaches:

$$(f(0), f(1), f(2), f(3), f(4), f(5), \dots) = (0, 1, 0, 1, 0, 1, \dots)$$

Note that this binary sequence encodes exactly the infinite subset O consisting of the odd numbers.



An uncountable set

Theorem 39 The powerset $P(\mathbf{N})$ is uncountable.

Since the infinite binary sequences of $2^{\mathbf{N}}$ are in bijection with the elements of the powerset $P(\mathbf{N})$ (through our usual 0-1 encoding of subsets), it suffices to show the following result.



Theorem 40 The set $2^{\mathbf{N}}$ of infinite binary sequences is uncountable.

Proof: The proof relies on a “diagonalization argument”¹⁰. We show the result by way of contradiction, where we assume that the set $2^{\mathbf{N}}$ is countable¹¹. We will derive a contradiction from this fact.

Since $2^{\mathbf{N}}$ is countable, there exists a bijection s from \mathbf{N} to the set $2^{\mathbf{N}}$. The function s hence “enumerates” all the infinite binary sequences of $2^{\mathbf{N}}$, where $s(0)$ is the first binary sequence in the enumeration, $s(1)$ is the second binary sequence and so on. The enumeration might look as follows:

$$\begin{aligned} s(0) &= (0, 1, 1, 1, 0, 0, 1, 1, \dots) \\ s(1) &= (0, 1, 0, 1, 1, 1, 0, 1, \dots) \\ s(2) &= (1, 0, 1, 1, 0, 0, 0, 0, \dots) \\ s(3) &= (1, 1, 1, 0, 1, 0, 1, 1, \dots) \\ s(4) &= (0, 0, 0, 1, 1, 1, 0, 0, \dots) \end{aligned}$$

¹⁰A similar diagonalization argument will be used later on as part of the proof of the halting problem’s undecidability.

¹¹See Exercise 2, part 2

$$\begin{aligned}
s(5) &= (0, 0, 0, 1, 1, 0, 1, 0, \dots) \\
&\quad \dots \\
s(n) &= (1, 1, 0, 0, 1, 1, 1, 1, \dots) \\
&\quad \dots
\end{aligned}$$

Of course we don't know what the binary sequences truly look like (which is why we will shortly introduce more general notation). Assuming that the sequences are as above, $s(0), s(1), \dots, s(n), \dots$ enumerate exactly all possible binary sequences (i.e. the entire set $2^{\mathbb{N}}$) by our assumption. However, from the above list it is easy to construct a binary sequence that “escapes” all the binary sequences in the given list. To achieve this, consider the bold bits in the sequences (which are the bits positioned on the diagonal of this huge matrix):

$$\begin{aligned}
s(0) &= (\mathbf{0}, 1, 1, 1, 0, 0, 1, 1, \dots) \\
s(1) &= (0, \mathbf{1}, 0, 1, 1, 1, 0, 1, \dots) \\
s(2) &= (1, 0, \mathbf{1}, 1, 0, 0, 0, 0, \dots) \\
s(3) &= (1, 1, 1, \mathbf{0}, 1, 0, 1, 1, \dots) \\
s(4) &= (0, 0, 0, 1, \mathbf{1}, 1, 0, 0, \dots) \\
s(5) &= (0, 0, 0, 1, 1, \mathbf{0}, 1, 0, \dots) \\
&\quad \dots \\
s(n) &= (1, 1, 0, 0, 1, 1, 1, 1, \dots, \mathbf{0}, \dots) \\
&\text{(where the bold zero is in position } n) \\
&\quad \dots
\end{aligned}$$

To construct a binary sequence that does not occur in this enumeration, simply flip all the bits on the diagonal (replace each bit b by $1 - b$). This changes the diagonal sequence $(0, 1, 1, 0, 1, 0, \dots, 0, \dots)$ into the “complement” binary sequence $(1, 0, 0, 1, 0, 1, \dots, 1, \dots)$. Clearly this new sequence does not occur in the above list (as it differs from each of the given sequences $s(n)$ by at least one bit, namely the bit in position n). This is a contradiction (since we assumed our enumeration included *all* possible binary sequences). Hence we have shown that the set $2^{\mathbb{N}}$ is *not* countable.

As pointed out, we don't know what the sequences look like in practice. Of course, it is easy to generalise the above argument.

Consider the following enumeration of binary sequences:

$$\begin{aligned}
s(0) &= (\mathbf{b}_0^0, b_1^0, b_2^0, b_3^0, b_4^0, b_5^0, \dots, b_n^0, \dots) \\
s(1) &= (b_0^1, \mathbf{b}_1^1, b_2^1, b_3^1, b_4^1, b_5^1, \dots, b_n^1, \dots) \\
s(2) &= (b_0^2, b_1^2, \mathbf{b}_2^2, b_3^2, b_4^2, b_5^2, \dots, b_n^2, \dots) \\
s(3) &= (b_0^3, b_1^3, b_2^3, \mathbf{b}_3^3, b_4^3, b_5^3, \dots, b_n^3, \dots) \\
s(4) &= (b_0^4, b_1^4, b_2^4, b_3^4, \mathbf{b}_4^4, b_5^4, \dots, b_n^4, \dots) \\
s(5) &= (b_0^5, b_1^5, b_2^5, b_3^5, b_4^5, \mathbf{b}_5^5, \dots, b_n^5, \dots) \\
&\quad \dots \\
s(n) &= (b_0^n, b_1^n, b_2^n, b_3^n, b_4^n, b_5^n, \dots, \mathbf{b}_n^n, \dots) \\
&\quad \dots
\end{aligned}$$

Again, we indicate the bits on the diagonal in bold.



Diagonalization

The sequence escaping the enumeration is the binary sequence

$$d = (\overline{b_0^0}, \overline{b_1^1}, \dots, \overline{b_n^n}, \dots)$$

where for any bit $b \in \{0, 1\}$ we define

$$\overline{b} = 1 - b \text{ (the flipped bit)}$$

The diagonal sequence d differs from each sequence $s(n)$ in at least one bit.

The bit $s(n)[n] = b_n^n$, i.e. the bit in the n -th position of the sequence $s(n)$, differs from the bit $d[n]$, i.e. the bit in the n -th position of the sequence d (they are opposite bits, i.e. $s(n)[n] = 1 - d[n]$ and hence $s(n)[n] \neq d[n]$). Since $s(n)[n] \neq d[n]$ holds for all $n \in \mathbf{N}$, we know that d differs from *every* sequence $s(n)$ in the above enumeration. Hence the enumeration is not a bijection between \mathbf{N} and the set of infinite binary sequences $2^{\mathbf{N}}$.

□

Remark 41 *Though we only have shown the result for the case of the set of the natural numbers, we observe that the result holds for all infinite countable sets: the powerset of an infinite countable set is not countable.*

We now focus on the problem of computability. We will show that there are a huge number of non-computable problems—a surprising result at first sight. In fact, we will show that the cardinality of the set of decision functions far exceeds the cardinality of the set of *computable* decision functions.

Chapter 5

Computability

Computing is not about computers
any more. It is about living.

Nicholas Negroponte

5.1 Introduction to computability

We will simplify the presentation of computability by using a model of computation that you are already familiar with. Just keep in mind your favourite programming language. We say that “a problem is computable” in case you can write a program for it that solves the problem (a program written in your favourite programming language). The only restriction to this approach is that the language you choose must be a “general-purpose” programming language¹, such as Java or Python.

To make the notion of “computability” more precise, recall that programs are considered to be a “recipe” for a computation that transform inputs to outputs. So to each program there naturally corresponds an “input-output” function. For a program P , we have denoted this input-output function by F_P , where we recall the following example:

Example: Consider a sorting program, say Bubble Sort, which we denote by B . The input-output function F_B corresponding to Bubble Sort is the function that maps an input list to a (sorted) output list. So the input-output function F_B corresponding to Bubble Sort, maps every list (L_1, \dots, L_n) to its sorted version. The list $(1, 2, 2, 1, 3, 6, 5, 1)$ is mapped to the sorted list $(1, 1, 1, 2, 2, 3, 5, 6)$ by Bubble Sort. In function notation:

$$F_B((1, 2, 2, 1, 3, 6, 5, 1)) = (1, 1, 1, 2, 2, 3, 5, 6).$$

¹A programming language capable of computing all “Turing computable” functions.

Clearly, every sorting algorithm S has the function F_B as its input-output function. For instance, the sorting algorithm Quick Sort, which we denote by Q has an input-output function F_Q that equals F_B .

Input-output functions

- To each program P , we can associate a unique function F_P consisting of all the input-output pairs (I,O) of P .
- Two programs P_1 and P_2 can have the same input-output function ($F_{P_1} = F_{P_2}$). If this is the case, we say that P_1 and P_2 “compute the same problem”.

We introduce the definition of a computable function next.

Computable functions

Definition 42 *We say that a function F is computable in case there exists a program P in a general purpose programming language^a such that $F_P = F$. In other words, we call a function computable in case we can write a program P that will “compute the values for F ”.*

^aSuch as Python or Java, each of which is equivalent in terms of computability to the so-called Turing machines. As pointed out at the outset of the course, we will refrain from formally introducing Turing machines in favour of working with “general purpose” programming languages.

Notation 43 *The set of all computable functions is denoted by “COMP”.*

Of course every program P determines a computable (input-output) function F_P . The question we consider is whether *all* functions are computable, i.e. given a function F , can we always find a program P such that $F = F_P$? The answer is relevant to practice. The function F can be considered as a specification for the program P . Indeed, it specifies which output we would like to obtain, for each given input. In other words, we specified the desired result of the computation. Can we always find a program that computes this given specification?

- If all functions are computable then any task a programmer is assigned (in the form of a function to be computed) can be solved.
- If not all functions are computable, a programmer could be set a problem to compute a function for which no solution (program) exists!

We will show in Section 5.6 that the second case indeed can occur.

5.2 Distinguishing between the function to be computed and programs that compute it

It is important to keep a clear distinction between the two notions:

- The function to be computed
- A program that computes the function

Programs and functions

- a) *The function to be computed* is a specification of the problem to be computed, i.e., the inputs and the outputs specified for these inputs).
- b) *A program computing such a function f* is any program that for each input i returns the output $f(i)$.

Consider the sorting problem once more. *The function to be computed* is the collection of pairs consisting of the input-lists, paired up with their corresponding sorted output-lists. This sorting function contains pairs:

$$[(2, 3, 1), (1, 2, 3)], [(1, 3, 2), (1, 2, 3)], [(5, 6, 1, 9, 7, 2), (1, 2, 5, 6, 7, 9)]$$

and so on. Any program that computes this particular function is a sorting program. Bubble Sort, Insertion Sort, Quick Sort, ... all compute this same function.

5.3 Enumerating all programs

Imagine someone starts writing out all possible programs one can code in Python (or your favourite general-purpose programming language). There are infinitely many possible such programs. For instance, a program which for any input returns the number 1 as output, another which returns the number 2 as output and so on.

Enumerating all programs

- Each program is a finite sequence of symbols. Using Gödel encoding, each program P can be assigned a unique Gödel number e .
- Through (explicitly computed) ranking, we can ensure that the unique numbers correspond exactly to the natural numbers.
- Hence we can enumerate all programs in an infinite list:

$$(P_0, P_1, \dots, P_{n-1}, P_n, \dots)$$

Consider the example of Bubble Sort. This program will be part of the sequence and could, say, have number 347, i.e. Bubble Sort is listed in our enumeration as the program P_{347} ²

We have shown that the set of all programs can be enumerated as an infinite sequence $(P_n)_n$. The sequence $(P_n)_n$ is exhaustive. If we encode all Python programs, then every Python program belongs to the sequence. The same argument holds for other programming languages, e.g. Java.

Any program you write hence equals some program P_e in this infinite sequence for some natural number e . This infinite enumeration will allow us to discuss *all programs of a programming language* at the same time and prove properties about them.

5.4 Total and partial functions

Input-output functions can be of two kinds: they can be *total* or *partial*.



Total and partial functions

A computable function is *total*, in case a program that computes the function returns an output for every input. Otherwise the function is said to be *partial*.

If the program P computing the function f (i.e. $f = F_P$) does not terminate on input i , then the function f won't contain a pair of the form (i, j) (i is not mapped to any value).

Each program that computes the function f must be undefined on i , i.e. the program “runs” forever or simply does not produce an output.

Sorting programs determine total computable functions. These programs produce an output for every input.

An example of a program P that determines a partial function is the program P with pseudo-code: [while $(x = x)$ do continue]. This is an extreme example. None of the inputs yield an output and hence $f_P = \emptyset$.

Note: We will show in Section 75 that there exists non-computable functions. In our argument, we only consider total computable functions from the natural numbers to the natural numbers. We will demonstrate the existence of non-computable functions among these *total* functions. The same argument could be applied (with minor tweaks) to show that there are non-computable functions among the partial functions from the natural numbers to the natural numbers³. In fact, the halting problem, a

²Exercise 29 illustrates that the exact number will be vastly larger, pre-ranking).

³Of course the existence of non-computable functions among the smaller class of total functions is sufficient to establish that non-computable functions exist, and in particular total non-computable functions.

concrete example of a non-computable function, will be dealt with in this general context. We restrict our presentation to total functions to make the introduction to non-computability a little more straightforward.

5.5 The “output” undefined



$\perp = \text{undefined}$

- In computer science it is customary to denote an input I that does not yield an output (for a given program) as a pair (I, \perp) . The symbol \perp indicates non-termination.
- This artificially introduced notation allows us to not only consider *total* functions as sets of pairs, but also to consider *partial* functions as a set of pairs, incorporating inputs that do not yield an output.
- For such inputs, the “output” is denoted by \perp —a value indicating that the program does not terminate on this input.

Under this convention, the program P specified by the pseudo-code

$$[\text{while } (x = x) \text{ do skip}]$$

computes a partial function F_P that maps every input to \perp , i.e.

$$F_P = \{(n, \perp) \mid n \in \mathbf{N}\} = \{(0, \perp), (1, \perp), (2, \perp), \dots, (n, \perp), \dots\}$$

In other words, F_P is the constant function mapping all natural numbers to \perp .

5.6 Existence of non-computable functions

Here we address the question whether there exist non-computable functions, i.e. functions f for which there does not exist a program P computing the function. A related question is whether these functions could arise in practice.

The first question is an interesting academic one. The second affects programming practice. It turns out that the answer to both questions is “yes”. We will deal with the first question in this section and show that there exists non-computable decision functions⁴, i.e. decision functions f in the $2^{\mathbf{N}}$ for which no program P exists such that $F_P = f$.

Remark 44 Intuition for the existence of non-computable decision functions

⁴And hence non-computable infinite binary sequences and non-computable sets of natural number. Cf. Remark 37.

We interpret decision functions in this context as infinite binary sequences. Many such sequences are computable. The constant sequences consisting entirely of zeros or consisting entirely of ones clearly are computable. So is any sequence varying in values up to a fixed index, following which the sequence remains constant⁵.

However, we will show that there are “insufficiently many” programs to compute all infinite binary sequences. Requiring that every binary sequence—no matter how random in nature—is computable, is a “big” ask—“too much” to ask as the following result shows.

Non-computable functions

Theorem 45 *Non-computable decision functions f exist for which there is no program P that computes f , i.e., for every program P , $F_P \neq f$.*

Proof: Recall that the set of all programs of a programming language is countable. Hence the set of all programs that compute decision functions form a countable set⁶. There is a surjection from the set of programs computing decision functions to the set of all computable decision functions⁷. Hence the set of all computable decision functions is countable⁸. The countable set of all computable decision functions is included in the uncountable set $2^{\mathbb{N}}$. Hence these two sets can’t be equal, i.e., the inclusion is strict, implying the existence of non-computable decision functions.

□

In fact, there exist far more non-computable functions than computable ones—“exponentially more.”

But what do non-computable decision functions look like in practice?

Are these decision functions esoteric exceptions that never arise?

Or are there non-computable decision functions that could occur

and make our computational life difficult?

*This brings us to our next chapter on
the “halting problem.”*

⁵A suitable series of if-then-else cases can treat the bits up to and including position n , after which the program only needs to produce a constant output.

⁶A subset of a countable set is countable. Use Exercise 29, part 1.

⁷Clearly, different programs could compute the same decision function.

⁸Exercise 29, part 5.

5.7 Infinite exponentials (optional section)

We will expand a little on the fact that there exist far more non-computable functions than computable ones—“exponentially more”. This section is optional reading for the mathematically inclined reader and can be skipped without affecting the remainder of the notes.

Strictly speaking, “exponentially more” makes no sense in the infinite case: the set of infinite binary sequences $2^{\mathbf{N}}$ corresponds to all subsets of the set \mathbf{N} , i.e. $2^{\mathbf{N}}$ is in bijection with the powerset $P(\mathbf{N})$ (and hence both sets have the same cardinality).

We know from Theorem 33 that for finite sets A , the powerset $P(A)$ contains exponentially more elements than A , $2^{|A|}$ to be precise. If this result holds for infinite sets, we could conclude that the powerset $P(\mathbf{N})$ contains “exponentially more” elements than \mathbf{N} , namely $2^{|\mathbf{N}|}$ to be precise.

However the set \mathbf{N} is infinite, i.e. $|\mathbf{N}| = \infty$. Hence $2^{|\mathbf{N}|} = 2^{\infty}$ makes little sense as this “value”, again, would be infinite. It is possible, through Cantor’s work on set theory, to make sense of this infinite “number”⁹ $2^{|\mathbf{N}|}$, which indeed is “exponentially” larger than $|\mathbf{N}|$, the size of the set \mathbf{N} . For our purposes it suffices to note that:

The size of the set of non-computable decision functions dwarfs that of the computable decision functions (even though both are infinite sets).

Equally so:

**The size of the set of non-computable functions dwarfs
the size of the set of the computable functions**

Thus there must be vastly more non-computable functions than there are computable functions in the set of functions $\mathbf{N}^{\mathbf{N}}$.

⁹For the mathematical reader, we remark that the exact classification of this exponential among cardinals is impossible to pin down due to the independence of the continuum hypothesis from the axioms of set theory.

Chapter 6

The halting problem

Solving a problem for which you know there's an answer is like climbing a mountain with a guide, along a trail someone else has laid. In mathematics, the truth is somewhere out there in a place no one knows, beyond all the beaten paths.

Yoko Ogawa, The Housekeeper and the Professor

We will outline the halting problem and prove that the problem is non-computable. The result is central in Computer Science. It is useful to spend sufficient time climbing this small mountain to gain insight in the core of the argument.

6.1 Outline of the problem

The halting problem is a decision problem yielding a bit-valued outcome.



Halting problem

- Given the code for program P and input i for this program, decide whether P terminates when executed on input i .
- The decision needs to be made by a program H that would, for any given program P and input i , determine whether the program P terminates when executed on input i .
- The program H must return 1 in case P terminates on i and 0 otherwise.
- Alan Turing proved in 1936 that a program solving the halting problem for all possible program-input pairs cannot exist, i.e., the halting problem is *undecidable*.

6.2 The halting problem today

Turing's discovery still affects practice. Real-Time Languages are programming languages for which every program is timeable with respect to the worst-case time measure. This affects all safety-critical areas, including chemical plant control, navigation (aircraft) and robotics for which precise scheduling (and hence timing) is a necessity.

The halting problem implies that no programming language can guarantee that *all* its programs can be timed with respect to the worst-case measure (explain this as an exercise).



Halting problem affects safety-critical applications

- Because of the halting problem, Real-Time Languages such as Real-Time Java, for which all programs *are* guaranteed to terminate, can only be obtained by restricting existing languages.
- Such restrictions ensure that all programs written in the restricted language are guaranteed to terminate. This is typically achieved by eliminating while loops from the language—the main source of non-termination—and by restricting for-loops so the number of loop-body executions is predictable.

6.3 The model of computation

The halting problem is a decision problem about properties of computer programs. For the sake of illustration, our programs will be assumed to be written in a general-purpose language such as Python or Java, or, indeed, any general-purpose programming language with “full computational power”¹. We will generally only make use of pseudo-code in this course.



Infinite resources

In the context of the halting problem, one assumes that there are no resource limitations of memory or time on the program’s execution. Programs can take arbitrarily long and/or use arbitrarily much storage space, before halting. The question is simply whether the given program will ever halt on a particular input.

6.4 The difficulty with the halting problem

The program `[while (x = x) continue;]` doesn’t halt, it runs forever in an infinite loop. On the other hand, the program `[print(“Hello, world”);]` halts quickly, regardless of its input. Anyone can determine the halting properties of these examples. A program either halts on a given input or it doesn’t.

Consider a program H_1 that takes as inputs a program P and input i and always returns the answer 1 (i.e. “ P halts on i .”)

Consider another program H_0 that takes as inputs a program P and input i and always returns the answer 0. (i.e. “ P doesn’t halt on i .”)

For any specific program P and input i , one of these two programs H_0 or H_1 will answer correctly, though we may not know which one.



The problem with the halting problem

The difficulty with the halting problem is that the decision procedure must work for *all* possible programs and inputs.

¹General purpose languages such as Java, Python and C compute the class of Turing-computable functions. No language can compute more than this class.

6.5 Interpreters

Interpreters are programs that simulate the execution of source code. Such programs can demonstrate that a program does halt if this is the case: the interpreter itself will eventually halt its simulation, which shows that the original program has halted. However, an interpreter will not halt if its input program does not halt so this approach cannot solve the halting problem.

6.6 Variants of the problem

There are many variants of the halting problem:

- A program eventually halts when run on input 0
- There is any input x such that a program eventually halts when run on input x

We will address some of these variants in the exercises. Such examples show that the halting problem has a way of “sneaking up” in practice. This happens in debugging contexts where tests could well include harvesting information on the behaviour of programs on certain inputs. Such a context should make programmers immediately aware of the possible pitfalls due to the halting problem.

6.7 Reduction techniques

The halting problem was one of the first problems to be proved undecidable². Subsequently, many such problems have been discovered.

The typical method of proving a problem to be undecidable is by “reduction”. The argument proceeds by way of contradiction. If a solution to the new problem were to be found, a “black-box” solution, then an argument by reduction shows that the black-box solution could be used to decide a known undecidable problem such as the halting problem. Instances of the new problem are transformed into equivalent instances of the known undecidable problem, for instance via duality as discussed in chapter 11. The black-box solution helps solve the known undecidable problem—an impossibility demonstrating that the black-box solution cannot exist. Since we already know that no program can decide the old problem, no program can decide the new problem. Rice’s theorem exploits this fact.

Rice’s theorem

Rice’s theorem essentially states that the truth of any non-trivial statement about the *function* that is defined by an algorithm is undecidable. The proof relies on a

²Turing’s proof went to press in May 1936, whereas Church’s proof of the undecidability of a problem in the lambda calculus had already been published in April 1936.

reduction technique similar to the reduction technique applied in the exercises. A reference on Rice's theorem is Cutland's excellent book.

Reduction techniques lie at the heart of this course. We will encounter such techniques again in the context of NP-complete problems.

6.8 Non-computability of the Halting Problem

Before embarking on the proof that the Halting Problem is not computable, we will cover a similar proof for a related problem that will help clarify the main idea.

We apply the “Holmes principle” (proof by contradiction) to show that there is no all-powerful³ wizard such as Voldemort.

Later on we introduce a similar proof by a computer scientist wizard (Alan Turing) that there is no “all-powerful program” that predicts whether programs terminate on their inputs.



The All-Powerful Wizard and the Stone

We show by way of contradiction that an all-powerful wizard cannot exist.

If there is an all-powerful wizard, then he should be able to make a stone he cannot lift.

Either such a wizard can make such a stone or he cannot.

- *If he can make a stone he cannot lift then he cannot lift this stone, hence is not all-powerful—a contradiction.*
- *If he cannot make a stone he cannot lift then he is not all-powerful—a contradiction.*

The existence of an all-powerful wizard leads to contradictions whichever logical route we take. By the Holmes principle such a wizard cannot exist since we eliminated all possibilities. What remains must be accepted as the truth. In this case, the existence of Voldemort has been eliminated—not surprisingly and certainly not unlikely.

We will use a similar approach to show that the halting problem is not decidable—a conclusion which may seem more unlikely, depending on one's initial intuitions regarding the solvability of this problem. The argument is conducted along similar lines. We assume that there is an “all-powerful” program H that solves the Halting Problem—i.e. we assume that there exists a black box solution H consisting of the program that computes the halting problem. H decides for every program-input pair

³All-powerful but with the caveat that the wizard cannot change the nature of logic. Otherwise, our reasoning may well go up in a puff of smoke.

(P, i) whether the program P halts on the input i . We obtain a contradiction from this assumption.

6.8.1 The halting problem is undecidable

The proof proceeds by way of contradiction.

Using Gödel encoding, consider an enumeration of all programs, $P_1, P_2, \dots, P_n, \dots$ denoted as the infinite sequence $(P_n)_n$. We assume that inputs are natural numbers. This assumption is not essential as one could encode any input via a natural number (Gödel Encoding). We omit the technical details.

We will show that there cannot exist a program which decides for each program P_i and input i whether P_i halts on input x . We assume (by way of contradiction) that there exists a program H that decides whether arbitrary program P_i halts on arbitrary input x .

The function computing this decision is called the Halting Function h :

$$\begin{aligned} h(i, x) &= 1 \text{ if program } P_i \text{ halts on input } x \\ &= 0 \text{ otherwise.} \end{aligned}$$

Program P_i refers to the i th program in the enumeration of all programs.

We show, via a proof by contradiction, that there cannot exist a program H that computes the halting function h .

Assuming that the program H exists, define a new program G with the following properties:

For any program P_i in the enumeration of all programs, G acts as follows:

- If P_i terminates on i (decided via H), then G does not terminate
- If P_i does not terminate on i (decided via H), then G does terminate



Definition of G

G can be defined from the black box solution H as follows:

If $H(P_i, i) = 1$ then while $[x = x]$ do skip else output 0

Note that G is one of the programs in the enumeration of all programs, say $G = P_e$. Thus:

- If $H(P_e, e) = 1$, i.e., P_e terminates on e , then $G = P_e$ does not terminate on e
- If $H(P_e, e) = 0$, i.e., P_e does not terminate on e , then $G = P_e$ does terminate on e ⁴

⁴With output 0

In each case, we obtain a contradiction, showing that program H cannot exist. The core idea is that program P_e is designed to work “opposite to its own (predicted) behaviour.” G is proving to be an “unliftable stone” for the program H . H cannot properly decide the halting behavior of G as each case leads to the opposite result of the predicted outcome—a contradiction.

6.8.2 Halting problem: solved exercises

The Halting Problem is the problem of determining whether a given program P running on a given input i will run forever, or terminate in some finite number of steps. This problem is known to be undecidable, in the sense that a general algorithm to determine for all programs and inputs whether the program will halt on the input (known as a Halting Computer) cannot be created. We discuss some problems related to the Halting Problem, which we can show to be undecidable based on the Halting Problem result. The method used is a proof by contradiction. We show that if we had an algorithm which could decide the given problem, we could use this to construct a Halting Computer. Since this is impossible, we conclude that the algorithm being sought also cannot be created.

Question 1: Can we decide whether a given program will halt when called on the input 0?

Answer: No

We display the **5-step proof** on the next page. Note that a similar argument can produce to show that it is impossible to decide whether a given program terminates on other inputs, say on input 42. In that case, simply replace 0 by 42 in a consistent way in the following argument.

5-step proof demonstrating that there is no program that decides whether input-programs terminate on input 0

a) **Proceed by way of contradiction: assume that the program *does* exist**
Suppose, by way of contradiction, that we could write the code for a program that checks whether a given program will halt when called on input 0. Let's call this program H_0 . We will show that, using H_0 , we can solve the halting problem. (Since we know this cannot be done, we will conclude that H_0 cannot exist in the first place.).

b) **Setting the stage to check that we can solve the Halting Problem**

To show that we can solve the Halting Problem, we assume that we are given an arbitrary program P and an input i for P . We will show (using H_0) that we can decide whether P halts on i .

c) **From P and i , we define a new program Q on which we will call H_0 .**

Assume that we start with input pair (P, i) from which we construct a new program Q . Q is identical to P except that it replaces every input j on which it is called by i .

Program $Q(j)$

1: Run program P on input i

Q , on receiving an input j , always executes P on i . Now we use H_0 to check whether Q terminates on input 0.

d) **Using H_0 to check that Q halts on input 0 is equivalent to checking whether P terminates on i**

We execute program H_0 on input Q . Remember that Q on any input j will execute P on i . Hence, for input 0, Q will execute P on i . We distinguish two cases:

- H_0 , when run on Q , returns 0 (i.e., Q does not terminate on input 0).
This case happens if and only if P does not terminate on i .
- H_0 when run on Q , returns 1 (i.e., Q terminates on input 0).
This case happens if and only if P terminates on i .

e) **Concluding the argument by contradiction**

For any pair (P, i) , we can (in the above way) determine whether P terminates on i by constructing the program Q from P and i and running H_0 on Q . By the above two cases, executing H_0 on Q decides whether P terminates on i . The program P and its input i are arbitrary. For each such pair we can construct program Q and execute H_0 on Q . Hence we can always decide whether P terminates on i and we have solved the halting problem. Solving the halting problem is impossible. So (since the only assumption we made in this argument is that H_0 exists), H_0 cannot exist.

In this way, we can see that the apparently much simpler problem of deciding termination on a given fixed input is equivalent to the halting problem, and thus cannot be solved.

Note: The halting program H which solves the halting problem, as constructed in the previous argument to determine whether P terminates on i , is specified as follows:

Program $H(P, i)$
1: $H_0(Q)$

Here Q is the program constructed from P and i (the inputs of H), where $Q(j)$ executes P on i .

Question 2: Can we determine whether there is any input for which a given program will halt?

Answer: No

The same construction as in the previous example would give us a program which would execute $P(i)$ on all inputs j , and hence halts on j if and only if P would halt on i . So again, solving the original problem is equivalent to solving the halting problem. As an exercise, write out the complete argument (as we did for Question 1).

Question 3: Can we determine whether a program will ever output a 5? Whether it will ever execute line 2 of its code?

Answer: No

In general, the answer to any question of the form “Can we determine whether a program will do X ?” will be “no”. This very informally stated result is known as Rice’s Theorem. To see how we could use such a decision to decide the Halting Problem, consider the following (again, very informal) pseudo-code for the program Q :

Program Q
1: Run program P on input i
2: Do X

It is easy to see that program Q will execute X if and only if program P terminates on input i (otherwise, part 1 will run forever, and part 2 will never be executed). So deciding whether program Q does X (whether this be outputting a 5 or simply executing the second line in the code, or any similar step) will be equivalent to deciding whether program P terminates on input i , and is therefore undecidable.

We illustrate this technique in more detail on a particular problem.

Show that there does not exist a program that determines whether a program will ever output 33.

We highlight the steps involved. Note that the argument differs from the one showing that we can't decide termination on a given *input*. The parts highlighted in red (the choice for Q) indicate the main difference.

5-step proof demonstrating that there does not exist a program that decides whether a program will ever output 33.

a) Proceed by way of contradiction: assume that the program does exist
Let's call this program H_{33} . H_{33} takes the code of programs P as input and returns 1 if P outputs 33 and 0 otherwise. We show that the existence of H_{33} allows us to solve the halting problem (this yields a contradiction and hence we will conclude that H_{33} cannot exist).

b) Setting the stage to check that we can solve the Halting Problem
Given a program P and input i , we will show that we can decide whether P terminates when executed on i . For this purpose, construct a program Q as in part c) below.

c) From P and i , we define a new program Q (on which we will call H_{33})

Program Q

- 1: Run program P on input i
- 2: Return 33

Note that program Q when executed on input j , will ignore j and simply execute P on input i , then return 33.

d) Checking, via H_{33} that Q returns 33 is equivalent to checking whether P terminates on i

We call H_{33} on Q to determine whether Q ever returns 33. We distinguish two cases:

- $H_{33}(Q)$ returns 1 and hence Q returns 33.
This is possible if and only if P terminates on i .
- $H_{33}(Q)$ returns 0 and hence Q does not return 33.
This is possible if and only if P does not terminate on i .

Hence H_{33} , for input Q , returns 1 if and only if P terminates on i and returns 0 if and only if P does not terminate on i . So H_{33} decides whether P terminates on i .

e) Concluding the argument by contradiction

Since this argument can be repeated for any program P and input i , where we call H_{33} on the newly constructed program Q (defined from P and i), it is clear that we

can solve the halting problem. As this is impossible, our original assumption—the existence of H_{33} —must be wrong and hence is eliminated from our reasoning. In other words, H_{33} cannot exist.

Note: The halting program H constructed in the previous argument, is specified as follows (where once again, Q is defined from P and i)

Program $H(P, i)$
1: $H_{33}(Q)$

Question 4: Can we determine whether a given program called on a given input will halt within 100 steps? 3 steps? n steps?

Answer: Yes! (Important to understand why this one is different!)

The undecidability of the Halting Problem only applies to algorithms with unlimited resources of time, memory, etc. For any finite n , we can decide if P halts within n steps when called on input i by running the first n steps of the execution of P on i , and checking if any of these is a halting instruction.

6.8.3 The halting problem: practice exercises

Problem 1: A programmer writes programs P_i for $i = 1, 2, 3, \dots$ which generate finite sequences of integers as their output in case they terminate, and intends to call a program Q on each integer of the output. The programmer knows that the running time of Q is very small except on the digit 33, where it is very large.

So, to estimate the average efficiency of his program Q , he decides to write another program T , which will take a program P as an input, and output the total number of 33s outputted by P . Is it possible to write such a program T ? (You may assume that the programs P_i do not take any input.)

Problem 2: Suppose that the programmer is worried about the program Q running for too long, and decides to prevent this by automatically terminating each program P after a finite running time t on some given machine (if it has not terminated already), and feeding whatever output has already been produced to Q .

Problem 3: Suppose instead that the programmer decides to prevent overly long execution times for Q by automatically terminating each program P when it has output ten 33s (if it has not terminated already).

Problem 4: Can you write a program that checks for any given Python program P containing a conditional statement (i.e. an if-then-else statement), whether the Python program P will ever execute the “then” branch (i.e. the first branch) of the conditional statement?

6.9 Can humans solve the halting problem?

It might seem like humans could solve the halting problem. After all, a programmer can often look at a program and tell whether it will halt. It is useful to understand why this cannot be true. To “solve” the halting problem means to be able to look at any program and tell whether it halts. It is not enough to be able to look at some programs and decide. Humans may also not be able to solve the halting problem, due to the sheer size of the input (a program with millions of lines of code). Even for short programs, it isn’t clear that humans can always tell whether they halt.

6.10 Twin primes

It is usually easy to see how to write a simple brute-force search program that looks for counterexamples to any particular conjecture in number theory. If the program finds a counter example, it stops and prints out the counter example, and otherwise it keeps searching forever. For example, consider the famous (and still unsolved) twin prime conjecture. This asks whether there are arbitrarily large prime numbers p and q with $p + 2 = q$. Now consider the following program, which accepts an input N :

```
findTwinPrimeAbove(int N)
  int  $p := N$ 
  loop
    if  $p$  is prime and  $p + 2$  is prime then return, else  $p := p + 1$ 
```

This program searches for twin primes p and $p + 2$ both at least as large as N . If there are arbitrarily large twin primes, it will halt for all possible inputs. But if there is a largest pair of twin primes p and $p + 2$, then the program will never halt if it is given an input N larger than p . Thus if we could answer the question of whether this program halts on all inputs, we would have the long-sought answer to the twin prime conjecture. It’s similarly straightforward to write programs which halt depending on the truth or falsehood for many other conjectures of number theory.

Since some of these problems have been open for many many decades, with the best mathematical minds trying to solve them, it is clear that a human observing code is unlikely to be able to resolve the question whether a particular program actually halts. In the case of the twin prime conjecture, the conjecture is open since 2310 years! (Euclid formulated it 300 BC). Number theory pops up many interesting problems such as these, easy to read and understand, yet extremely hard to resolve. For instance Fermat’s theorem (which was only resolved quite recently). Another example is the Collatz conjecture.⁵

⁵The Collatz conjecture is a conjecture in mathematics named after Lothar Collatz, who first proposed it in 1937. The conjecture is the following: take any positive integer n . If n is even, divide it by 2 to get $\frac{n}{2}$. If n is odd, multiply it by 3 and add 1 to obtain $3n + 1$. Repeat the process indefinitely. The conjecture is that no matter what number you start with, you will always eventually reach 1.

6.11 Implications of the halting problem for the design of Real-Time programming languages

Timing software is crucial for many applications. “Real-Time Software” in particular is a type of software designed to make timing of programs possible. Real-Time Software can guarantee that deadlines are met in safety critical situations. This is widely applied in telephone exchanges, satellite communications, medical equipment, chemical plant control, stock market analysis and robotics.

For Real-Time applications, time analysis needs to be linked with the type of hardware one considers. Issues such as pipelining, cache memory and garbage collection (for Real-Time Java applications) come into play when one needs to predict the execution time of software. A fully detailed treatment of timing is well beyond the scope of this course⁶.

Once the timing of software can be achieved, the various processes taking place can be scheduled and accurate predictions can be made to ensure that the system performs in a timely and safe manner.

One example is the Mars Rover, a robot for space exploration. Its functioning relies heavily on real-time programming and proper scheduling⁷.

The halting problem directly impacts the field of Real-Time programming languages. Due to the halting problem, we know that it is impossible to compute whether, for any program-input pair, the program will terminate on the input. In particular it is not possible to develop software that for any program-input pair times how long the program will run on the input (giving an infinite value in case the program runs for ever on the input). If such software could be developed, we would in particular be able to determine whether a program halts or not—a contradiction.

Hence the problem of timing needs to be applied in a restricted context. In order to guarantee termination, Real-Time languages typically ban while-loops (since this type of loop is a cause of non-termination). Real-Time languages also may pose restrictions on for-loops to ensure manageable timing. The resulting “Real-Time programs”, which do not contain while-loops (and have tightly controlled for-loops) must be guaranteed to terminate (can you see why such restrictions can ensure termination?).

The famous mathematician Paul Erdős said about the Collatz conjecture: “Mathematics may not be ready for such problems”, as discussed in: Guy, Richard K, “*Unsolved problems in number theory* (3rd ed.). Springer-Verlag, “Permutation Sequences”, pp. 336-337, 2004.

⁶We will focus on the basic timing measures of worst-case running time analysis (the main time measure used in practice) and average-case running time analysis of algorithms (a useful measure to classify algorithms according to their expected time).

⁷Theory and practice intersect regularly: an error in the Mars Rover’s original programming, called priority-inversion, led to a malfunction during a mission and could only be resolved through an analysis of the programs.

A second advantage of eliminating while-loops, is that while-loops, even when they do happen to terminate, are unpredictable regarding the number of times such a loop will exit. Indeed, the number of times that a while-loop executes on a given input depends on the boolean expression in the while loop. This boolean condition governs when the loop will execute. Predicting when this boolean condition will exit for a given input is highly non trivial (even assuming that the while loop will actually terminate). For-loops on the other hand, under suitable restrictions, are very predictable regarding the number of times they will execute. Indeed, for an ordinary for-loop, [for $i = A_1$ to A_2 do P], this number is clearly indicated by the difference $A_2 - A_1 + 1$ and for a down-to loop, [for $i = A_2$ downto A_1 do P] this is indicated by $A_1 - A_2 + 1$ assuming the values A_1 and A_2 hold specify natural numbers.

Learning to program without the use of while-loops and under various other restrictions on for-loops and possibly with tight memory control (to enable garbage collection timing) means that Real-Time programmers need to develop a set of unique skills—an immediate consequence of the halting problem's undecidability.

Chapter 7

Knuth's lab: comparison-based algorithms

Sorting and searching provides an ideal framework for discussing a wide variety of important issues [design, improvement and analysis]. Virtually every important aspect of programming arises somewhere in the context of sorting and searching.

Don Knuth

7.1 Comparison-based algorithms

We will introduce “comparison-based algorithms”, a class of algorithms that includes most sorting and search algorithms. Comparison-based algorithms provide a basic “laboratory” in which we can investigate important features of general programming.

Comparison-based algorithms create order in the data on the basis of comparisons. The computation is assumed to be entirely “driven by comparisons.” Essentially, every branch the algorithm computation follows is based on prior comparisons between elements. Every swap placing elements in “the right order” is based on prior comparisons. We will clarify this below.

Examples of comparison algorithms abound in the algorithms literature. Most sorting and search algorithms are comparison based. This includes Insertion Sort, Bubble Sort, Merge Sort and Heap Sort. We will revisit comparison-based algorithms in Section 12.3.

Relevance of comparison-based algorithms

- The class includes a large number of data restructuring algorithms including most sorting and search algorithms. Such algorithms can take up to 25% of the total computation time of applications.
- Their asymptotic time (worst-case and average-case) is guaranteed to satisfy a lower bound, namely: $\Omega(n \log_2 n)$.
- Their running time is in general directly linked to the number of comparisons carried out during their execution. This enables one to approximate their running time via a static determination of the number of comparisons carried out during their execution.

We restrict the discussion to comparison-based algorithms taking lists as inputs. The definitions can be generalized in a straightforward way to algorithms taking other data structures as inputs (e.g. heaps).

Consider a comparison-based algorithms that takes a list

$$L = (L[1], \dots, L[n])$$

as input. Each decision made by the algorithm is determined by a series of comparisons. These comparisons are evaluated as part of a boolean expression involving these comparisons determining the outcome for the decision.

Each swap performed during the execution of the algorithm is carried out only in the wake of such a decision. The execution of the swap between two list elements $L[i]$ and $L[j]$ must be based on prior comparisons between elements of the input list L , leading to the conclusion that the two elements $L[i]$ and $L[j]$ are out of order.

For a sorting algorithm such as Bubble Sort, each swap between list elements is the direct result of a comparison between these two elements. For the Quick Sort algorithm, each swap is the result of comparisons involving two elements, each of which is compared to a pivot element. When these two elements are found to be out of order (not by direct comparison to each other, but through comparison of each element with the pivot element), the swap is executed.

We will return to these examples below.



Clarification

We have used a rather vague terminology: “a swap is based on (prior) comparisons between list elements, *leading to the conclusion that* the two elements (to be swapped) are out of order.” This terminology means that the determination that these two elements are out of order is either made

- *directly* as the result of a single comparison between these elements
- or
- *indirectly* based a number of prior comparisons.

Conclusion that a pair is out of order, based on a *direct* comparison

For basic algorithms, such as Bubble Sort and Insertion Sort, all swaps will be based on an immediate comparison between list elements, i.e. in each case these algorithms perform a swap between elements $L[i]$ and $L[j]$, for which $i \leq j$, after an immediate comparison between the elements $L[i]$ and $L[j]$, where the conclusion is that the elements are out of order, i.e. $L[i] \geq L[j]$.

Conclusion that a pair is out of order, based on *indirect* comparisons

Sometimes the conclusion that a pair of elements $L[i]$ and $L[j]$ is out of order is not reached via a *direct* comparison between these elements. For instance, for the case of Quicksort, elements are compared to a fixed pivot element p . We assume in the following that the pivot element p is the first element of the list L , i.e., $p = L[1]$. In this case, the determination that two elements $L[i]$ and $L[j]$ (other than the pivot element) are out of order can only be reached “indirectly” since, throughout Quick Sort’s computation, the only comparisons between list elements made are comparisons involving a list element $L[i]$ and the pivot p . Swaps in this situation are only ever permissible (for the case of comparison-based algorithms) in case such prior comparisons with the pivot element p imply that two elements $L[i]$ and $L[j]$ are out of order.

Consider for instance the case where prior comparisons have lead to the information that $L[i] > p$ and $L[j] < p$, where $i < j$. In this case $L[j] < p < L[i]$ and hence (using transitivity of this order) $L[j] < L[i]$. These two elements are out of order—an outcome based on *prior* indirect comparisons. Thus the swap is justified.

“Transitive closure” is a formal way to derive logical conclusions of the following type:

“If Jack is smaller than Eileen
and Eileen is smaller than Louise
then Jack is smaller than Louise.”

The same holds true if more people are involved:

“If Jack is smaller than Ashling
and Ashling is smaller than Gilbert
and Gilbert is smaller than Louise
then Jack is smaller than Louise.”

In the first case, the conclusion that Jack is smaller than Louise, is the result of the “transitive closure” of the relations “Jack is smaller than Eileen” and “Eileen is smaller than Louise.”

In general, the derivation of information of the order between two elements is established based on prior comparisons through the notion of “transitive closure.”

We have already seen one example of such a transitive closure, for the case where $L[i] > p$ and $L[j] < p$. In this case, we have that $L[j] < p < L[i]$ and hence (using transitivity of this order) we can conclude that $L[j] < L[i]$.

This can be generalized to the case where we have more than two comparisons.

Transitive closure (formal)

Let $k \geq 2$. Transitive closure of k inequalities between list elements means that if we know that the following inequalities hold:

$$L[i_1] \leq L[i_2], L[i_2] \leq \dots, L[i_{k-2}] \leq L[i_{k-1}], L[i_{k-1}] \leq L[i_k]$$

then we can derive the inequality

$$L[i_1] \leq L[i_k]$$

as a “transitive closure” of the previous inequalities.

The notion of transitive closure serves to provide a formal definition of comparison-based algorithms.

Algorithms take different computational paths depending on the outcome of boolean conditions. Such “branching-decisions” made by algorithms are determined by

- a) boolean conditions in if-then-else statements (branching conditions)

or

- b) boolean conditions in while-loops (exit conditions).

Definition 46 *A comparison-based algorithm is an algorithm satisfying rules A), B) and C) as specified below.*

A) Comparison-based algorithm: rules on conditions

- Boolean conditions are entirely based on comparisons between list elements only.

Definition 47 A positional assignment is the assignment of a new value for a variable j determining the position of a list element $L[j]$. For example: $j := j + 1$ is a positional assignment in case it is followed at any stage in the computation by a comparison between list elements, one of which is $L[j]$. In other words, a positional assignment is an assignment governing the position of a list element. We refer to the variable j as a positional variable.

B) Comparison-based algorithm: rules on positional assignments

- Positional assignments to a (positional) variable j , other than its initialization, must occur in the body of a while loop or as part of one of the branches of an if-then-else statement. Moreover, one of the prior list element comparisons must have involved $L[j]$. Hence every positional assignment is based on prior comparisons between list elements.
- Comparisons between list elements are the only justification for positional assignments.

C) Comparison-based algorithm: rules on swaps

- Swaps must occur in the body of a while loop or as part of one of the branches of an if-then-else statement. Hence every *swap* is the outcome of a decision based on prior comparisons between list elements.
- Comparisons between list elements are the only justification for swaps.
- For the case of a swap between list elements, the prior comparisons between list elements must lead to the conclusion that the two elements (to be swapped) are out of order, where this conclusion is based on the transitive closure of the relations established by the prior comparisons.
- We allow for a swap to occur between equal list elements. In case the list elements differ, the elements need to be out of order for a swap to occur.

Remark 48 *From the definition it is clear that a comparison-based algorithm can only change its input (through swaps, possibly determined by prior positional assignments) by comparing pairs of elements first (and by no other method). Comparison-based algorithms can only gain information about an input and rearrange an input's elements on the basis of comparisons between elements. Hence the term comparison-based (or comparison-driven) algorithm.*

7.2 Examples of comparison-based sorting algorithms

Convention: A for-loop of the form “for $i = A_1$ to A_2 do P ” will only be executed when the value of A_1 is less than or equal to the value of A_2 . Similarly, a “downto” for loop will only be executed when the value of A_1 is greater than or equal to the value of A_2 .

Indentation: to clearly indicate where loops and conditionals begin and end, indentation is used. That is, each such program part starts more to the right than the previous part.

Example 1: Bubble Sort

```
for  $i = |L| - 1$  downto 1 do
  for  $j = 1$  to  $i$  do
    if  $L[j] > L[j + 1]$  then
       $k := L[j]$ 
       $L[j] := L[j + 1]$ 
       $L[j + 1] := k$ 
```

This swapping of the values of elements of a list (i.e. interchanging of values) occurs so often that we introduce an extra symbol for it in pseudocode:

$$\text{swap}(L[j], L[j + 1]) = \begin{cases} k := L[j] \\ L[j] := L[j + 1] \\ L[j + 1] := k \end{cases}$$

So that Bubble Sort can now be rewritten as follows:



Bubble Sort

```
for  $i = |L| - 1$  downto 1 do
  for  $j = 1$  to  $i$  do
    if  $L[j] > L[j + 1]$  then
      swap ( $L[j], L[j + 1]$ )
```

Note that for a list L of length 1 ($|L| = 1$) we have that the for loop is *not* executed, since $i = |L| - 1 = 1 - 1 = 0 < 1$, while for the downto for loop we need $|L| - 1 > 1$ to have the for loop executed.

This is fine, however, since a list of length 1 is *always* sorted, and thus no program needs to be run on this input in order to sort it. Our convention is that *every* sorting or search algorithm will be assumed to simply return a list of size 1, even if this case is not explicitly treated in the pseudo-code. Note that we also assume this for the empty list of size 0, which again is considered to be a sorted list.

Example of Bubble Sort's execution on the list $L = (1, 4, 3, 2)$

$$|L| - 1 = 4 - 1 = 3$$

Case 1: $i = 3$

$\begin{array}{cccc} 1 & 4 & 3 & 2 \\ \uparrow & & & \\ j & & & \end{array} \Rightarrow \begin{array}{cccc} 1 & 4 & 3 & 2 \\ \uparrow & & & \\ j & & & \end{array} \Rightarrow \begin{array}{cccc} 1 & 3 & 4 & 2 \\ \uparrow & & & \\ j & & & \end{array} \Rightarrow \begin{array}{cccc} 1 & 3 & 2 & 4 \\ \uparrow & & & \\ j & & & \end{array}$ (Note: 4 “bubbled up”).

Case 2: $i = 2$

$\begin{array}{cccc} 1 & 3 & 2 & 4 \\ \uparrow & & & \\ j & & & \end{array} \Rightarrow \begin{array}{cccc} 1 & 3 & 2 & 4 \\ \uparrow & & & \\ j & & & \end{array} \Rightarrow \begin{array}{cccc} 1 & 2 & 3 & 4 \\ \uparrow & & & \\ j & & & \end{array}$ (Note: 3 “bubbled up”).

Case 3: $i = 1$

$\begin{array}{cccc} 1 & 2 & 3 & 4 \\ \uparrow & & & \\ j & & & \end{array} \Rightarrow \begin{array}{cccc} 1 & 2 & 3 & 4 \\ \uparrow & & & \\ j & & & \end{array}$ (Note: 2 “bubbled up” and output = the sorted list)

Bubble Sort “bubbles down” the “lighter” (smaller) elements of a list to the front of the list while “heavier” (larger) elements “bubble up” to the end of the list.

We present one more example of a sorting algorithm written in pseudocode:

Example 2: Insertion Sort

```

for  $i = 2$  to  $|L|$  do
   $j := i$ 
  while ( $j > 1$  and  $L[j - 1] > L[j]$ ) do
    swap ( $L[j - 1], L[j]$ )
     $j := j - 1$ 
  
```

We continue to provide a detailed example of Insertion Sort's execution on a particular list.

Example of how Insertion Sort runs on the list: $L = (1, 4, 3, 2)$; $|L| = 4$

$i = 2$: $j := 2$

1 4 3 2
 \uparrow \Rightarrow 1432
 j

Next insert 3 in priorly sorted list 14.

$i = 3$: $j := 3$

1 4 3 2 1 3 4 2
 \uparrow \Rightarrow \uparrow \Rightarrow 1342
 j j

Next 2 will be inserted in the priorly sorted list 134

$i := 3$, $j := 3$

1 3 4 2 1 3 2 4 1 2 3 4
 \uparrow \Rightarrow \uparrow \Rightarrow \uparrow \Rightarrow 1234
 j j j

Finally we obtained the sorted list as output.

Sentinel values to streamline the code

We remark that we need the use of the logical operator “and” in our code of to make sure that $j > 1$.

Since at each run of the while loop, j is decreased by 1, one does indeed need to make sure that j does not become less than 1 (why?...).)

Similarly, if in a loop the value of a loop variable j is increased at each run of the loop, we would need to verify that j does not become larger than $|L|$ (why?...). To avoid this we take the following approach.



Sentinel values

We change the structure of lists a little to have a “sentinel value” at each end of the list, which will prevent a variable-setting from “flying off the end of a list”.

- “ $-\infty$ ” is defined to be a “sentinel value” which is strictly less than any

element of a list.

- “ $+\infty$ ” is defined to be a “sentinel value” which is strictly greater than any element of a list.

To each list $L = (L[1], L[2], \dots, L[n])$ we add the sentinel value “ $-\infty$ ” in front and a sentinel value “ $+\infty$ ” at the end.

From here on (unless specified otherwise), each list will be of the following form:

$$L = (-\infty, L[1], L[2], \dots, L[n], +\infty).$$

In practice we will not indicate this extra sentinel element ∞ and we simply write: $L = (L[1], L[2], \dots, L[n])$. The elements $-\infty$ and $+\infty$ however are still assumed to be there. Hence, if the algorithm compares any other element with one of these infinity elements, **an additional comparison needs to be counted!**

On the other hand, we will be more loose with the meaning of the “length of a list”. Strictly speaking we should say that the length of a list L now is $|L| + 2$ where the addition of the number 2 takes into account the two infinity elements. However, since we agreed that typically we do not display these elements, we will still regard the length of a list to be $|L|$ where we count all the elements *except* the elements $-\infty$ and $+\infty$. So the length of the list $(1, 4, 2)$ (which of course is the list $(-\infty, 1, 4, 2, +\infty)$) is 3 and not 5.

Let’s see how Insertion Sort can be rewritten using sentinel values:

Insertion Sort with sentinel values

```
for  $i = 2$  to  $|L|$  do
   $j := i$ 
  while  $L[j - 1] > L[j]$  do
    swap ( $L[j - 1], L[j]$ )
     $j := j - 1$ 
```

Our convention is that for a given list L of length n ,

$$L[0] = -\infty$$

$$L[n + 1] = +\infty$$

So, for the above algorithm, when j is decreased from 1 to 0 via the assignment $j := j - 1$, the while loop is automatically terminated since $L[0] < L[1]$ is always true (so the “front bumper” sentinel value has done its job).

Exercise 49 a) *Run Bubble Sort on the following lists:*

$(1, 3, 2, 7)$, $(3, 9, 5, 15)$ and $(20, 40, 30, 100)$

How many comparisons does Bubble Sort take on each of these lists?

How about Insertion Sort?

Can you generalize your conclusion for each algorithm?

Bubble Sort and Insertion Sort are comparison-based algorithms.

For Bubble Sort, the only swap occurs in the following line of the code:

```
if  $L[j] > L[j + 1]$  then  
    swap ( $L[j]$ ,  $L[j + 1]$ )
```

Clearly this swap between list elements $L[j]$ and $L[j + 1]$ is the direct result of a comparison between the elements $L[j]$ and $L[j + 1]$, where the swap only occurs if these elements are found to be out of order.

For Insertion Sort (sentinel version), the only swap occurs in the following line:

```
while  $L[j - 1] > L[j]$  do  
    swap ( $L[j - 1]$ ,  $L[j]$ )
```

This swap between list elements $L[j - 1]$ and $L[j]$ is the direct result of a comparison between the elements $L[j - 1]$ and $L[j]$, where the swap only occurs if these elements are found to be out of order.

Hence both algorithms are comparison-based sorting algorithms.

We consider one more example of a sorting algorithm, the well-known algorithm Quick Sort, which compares two list elements to a chosen pivot and places each of the elements in the correct position in relation to the pivot. We use Sir Tony Hoare's original version of Quick Sort, which we assume is familiar. We recall the partitioning part of the code:

Quick Sort's Partition

```
Partition( $L, low, high$ )
```

```
    pivot :=  $L[low]$ 
```

```
     $i := low - 1$ 
```

```
     $j := high + 1$ 
```

```
    while  $1 \leq 1$ 
```

```
        do
```

```

         $i := i + 1$ 
    while  $L[i] < pivot$ 
    do
         $j := j - 1$ 
    while  $L[j] > pivot$ 
    if  $i \geq j$  then
        return  $j$ 
    swap( $L[i], L[j]$ )

```

Quick Sort then is defined as follows:



Quick Sort

```

Quick Sort( $L, low, high$ )
    if  $low < high$  then
         $p := Partition(L, low, high)$ 
        Quick Sort( $L, low, p$ )
        Quick Sort( $L, p + 1, high$ )

```

Sorting the entire array is accomplished by Quick Sort($L, 1, |L|$).

The only swap in the code occurs in the partitioning part. Clearly the swap is only executed in case the elements $L[i]$ and $L[j]$ are out of order. Indeed, the partition part of the algorithm only executes the swap when two elements $L[i]$ and $L[j]$ are reached for which $L[i] \geq pivot$ and $L[j] \leq pivot$. Hence the swap occurs after it has been determined that $L[i] \geq L[j]$, i.e. the two elements are out of order (in case they actually differ).

Note that the determination that $L[i]$ and $L[j]$ are out of order is not based on a direct comparison between $L[i]$ and $L[j]$. This is why, in the definition of a comparison-based algorithm, Definition 46, the following requirement is made:

every *swap* between list elements $L[i]$ and $L[j]$ is based on prior comparisons between list elements and these comparisons lead to the conclusion that the two elements $L[i]$ and $L[j]$ are out of order, i.e. $i \leq j$ and $L[i] \geq L[j]$.

We recall that the phrase “lead to the conclusion” in this context means that $L[i] \geq L[j]$ is either concluded as the result of a direct comparison between these elements, or derived (via transitive closure) from a number of prior comparisons. The latter case is illustrated by the comparisons $L[i] \geq pivot$ and $pivot \geq L[j]$ for the case of Quick Sort, leading to the (implicit) conclusion that $L[i] \geq L[j]$, hence justifying the subsequent swap, i.e., Quick Sort is a comparison-based algorithm.

Exercise 50 a) *Verify that Selection Sort is a comparison-based sorting algorithm.*
b) *Verify that Merge Sort is a comparison-based sorting algorithm.*

Next, we introduce the notion of a comparison-path, which will be useful to help represent the execution of a comparison-based sorting algorithm on a given input list.

7.3 Binary comparison-paths

The notion of a binary comparison path serves to capture a snapshot of the computation of a comparison-based algorithm. We recall that for comparison-based algorithms the computation is entirely “driven by comparisons”. All decisions the algorithms make during the computation are performed in an evaluation of conditional statements. Such boolean conditional statements control the outcome of if-then-else statements and of while-loops. For comparison-based algorithms, each such conditional is based on one or more comparisons between input list elements.



Comparison-paths record information flow

- The outcomes of comparisons made during the computation on a given input-list reflect exactly which branch of the if-then-else statement is executed or whether while-loops exit.
- Think of the execution of the algorithm as “information traffic along branching highways.” The outcomes of comparisons reflect the “turns” or “exits” taken by the traffic. The traffic is controlled by higher level constructs such as if-then-else statements and while-loops that govern these turns and exits in function of the outcomes of boolean decisions.
- The binary comparison-path for a given input list records the outcomes of these decisions made during the execution.

The notion of a binary comparison-path is a first step towards the notion of “decision trees” that will be introduced in Section 12.3. Decision trees are binary trees that model the *entire* computation during the execution of the algorithm over *all* inputs. Binary comparison-paths in such trees represent the information flow that occurs during the computation on a *single* input.



Binary comparison paths

Definition 51 *A binary comparison-path records the consecutive binary outcomes of the comparisons made during the computation of a comparison-based algorithm on an input list.*

Notation 52 *The binary comparison path of a comparison-based algorithm A for an input list L is denoted by $BCP_A(L)$*

Exercise 53 *Determine the binary comparison-path for Insertion Sort on the input list $L = (3, 6, 1, 5, 7)$ and do the same for the input list $L' = (1, 3, 6, 7, 5)$ (where we do not assume sentinel values as part of our list data structure).*

Solution for the list $L = (3, 6, 1, 5, 7)$: when we track the execution, we conclude that the following pairs are compared in the given order, where we mark for each pair the outcome of the comparison: $((3, 6), 1), ((6, 1), 0), ((3, 1), 0), ((6, 5), 0), ((3, 5), 1), ((5, 7), 1)$. The binary comparison path for this input list is the binary sequence $(1, 0, 0, 0, 1, 1)$.

Remark 54 *If we only have a finite number of input lists to consider, all of fixed size n , then we can represent the collection of all binary comparison paths for these input lists via a binary tree. The root node of the tree is labelled with the finite collection of inputs. For each binary comparison-path, draw a left branch in case the first element of the path is 0 and a right branch otherwise. At the endpoint of the branch you drew, once again, draw a left branch or a right branch depending on whether the next bit in the binary comparison path is zero or one. Repeat this for each of the binary-comparison-paths. The end result is a binary tree “representing” all the binary comparison-paths. (It is clear that from the tree you can reconstruct all binary comparison paths you started out with. Explain why.)*

Exercise 55 *a) Draw the binary tree representation for the binary comparison paths of Insertion Sort, corresponding to the following inputs of size 3:*

$(1, 2, 3), (1, 3, 2), (2, 1, 3), (2, 3, 1), (3, 1, 2), (3, 2, 1)$.

Now do the same, but this time for the following inputs of size 3:

$(1, 5, 13), (1, 13, 5), (5, 1, 13), (5, 13, 1), (13, 1, 5), (13, 5, 1)$.

What is your conclusion?

b) Write a program which takes as inputs the code for a comparison-based sorting algorithm and a size n of the input lists.

The program should output the binary tree representation for the binary comparison-paths of the algorithm given as input, for input lists of the given size n .

Recall that the execution of the algorithm as “information traffic along branching highways.” The outcomes of comparisons reflect the “turns” or “exits” taken by the traffic. The traffic is controlled by higher level constructs such as if-then-else statements and while-loops governing these turns and exits depending on the outcomes of boolean decisions. This means that if the outcomes of comparisons during the execution of a comparison-based algorithm on two input lists is exactly the same, these algorithms compute in exactly the same way. They take the same turns and the same exits during the computation. This is captured by the following notion.

Inputs sharing the same computation path

Definition 56 Given two input lists L_1 and L_2 of the same length. We say that a comparison-based algorithm A carries out the same computation for each of these lists (or computes in the same way) in case these lists share the same binary comparison paths:

$$BCP_A(L_1) = BCP_A(L_2)$$

When these binary comparison paths differ, we say that the computation of the algorithm diverges on these input lists.

7.4 Lists sharing the same relative order

Twin pairs

Definition 57 Let L_1 and L_2 be lists of the same length. Two pairs of elements $(L_1[i], L_1[j])$ and $(L_2[k], L_2[l])$ (for which $i \leq j$ and $k \leq l$) form twin pairs in case their respective elements occur in the same positions, i.e., $i = k$ and $j = l$.

Example 58 Consider the lists $L_1 = (1, 3, 23, 4, 0, 9)$ and $L_2 = (1, 1, 1, 1, 4, 2)$. The pairs $(L_1[2], L_1[4])$ and $(L_2[2], L_2[4])$ are twin pairs of elements, where

$$(L_1[2], L_1[4]) = (3, 4) \text{ and } (L_2[2], L_2[4]) = (1, 1).$$

Note that when considering twin pairs, we implicitly assume that the positions of the elements in the pairs are known. It does not make sense to consider whether $(3, 23)$ and $(1, 1)$ are twin pairs without specifying the location of these elements in the respective lists. Not specifying these locations leads to ambiguity since the element 1 occurs in positions 1 through 4 in the list L_2 .

Relative order

Definition 59 Two lists L_1 and L_2 for which $|L_1| = |L_2|$ share the same relative order in case all twin pairs occur in the same order. For any pair of indices i and j the following holds:

$$L_1[i] \leq L_1[j] \text{ if and only if } L_2[i] \leq L_2[j].$$

Example 60 a) The sorted lists $(2, 3, 4)$ and $(3, 4, 5)$ share the same relative order as the sorted list $(1, 7, 100)$.
b) The lists $(2, 4, 3)$ and $(7, 100, 9)$ have the same relative order.

Proposition 61 Two lists L and L' share the same relative order if and only if the ranks of their respective elements are the same. In other words, L and L' have the same relative order if and only if for each $i \in \{1, \dots, n\}$, $\text{rank}(L[i]) = \text{rank}(L'[i])$.

Proof: Recall that the rank of an element is the number of elements strictly smaller than the element. Clearly, if two lists share the same relative order, then the number of elements smaller than or equal to a given element must be the same.

To show the converse, assume, by way of contradiction, that the corresponding elements $L[k]$ and $L'[k]$ share the same rank for each index k and that there exists indices i and j such that $L[i] \leq L[j]$ and $L'[i] \not\leq L'[j]$, i.e., $L'[i] > L'[j]$. Note that $L[i] \leq L[j]$ implies that $\text{rank}(L[i]) \leq \text{rank}(L[j])$. Likewise, $L'[i] > L'[j]$ implies that $\text{rank}(L'[i]) > \text{rank}(L'[j])$. This results in the contradiction $\text{rank}(L[i]) \neq \text{rank}(L'[i])$.

□

Example 62 a) The sorted lists $(2, 3, 4)$, $(3, 4, 5)$ and $(1, 7, 100)$ possess elements of the same rank. Represented in a list, these ranks are given by: $(0, 1, 2)$.
b) The lists $(1, 3, 2)$ and $(7, 100, 9)$ possess elements of the same rank. Represented in a list, these ranks are given by: $(0, 2, 1)$.

We proceed to discuss four main characteristics of comparison-based algorithms.

Four main properties of comparison-based algorithms

- 1) **Completeness**

In order for a comparison-based algorithm to properly sort a list, the algorithm needs to determine the outcome of the relative order of each pair of elements in the list at least once. Note that this can be through a direct comparison or as a result of the transitive closure of prior comparisons.

- 2) **Faithfulness**

Comparison-based algorithms execute in the same way on input lists that share the same relative order.

- 3) **Separation**

Comparison-based algorithms executed on inputs that do not share the same relative order must diverge in their computation.

- 4) **Reduction: the 0-1 principle**

In order to check that a comparison-based algorithm is a sorting algorithm, it suffices to verify that it sorts all *binary* sequences.

We give an intuitive explanation for each of the four properties on the next page, followed by formal proofs that each of the properties hold.

Intuitive motivation for each of the four main properties

- **1) Completeness**

In order for a comparison-based algorithm to properly sort a list, the algorithm needs to determine the outcome of the relative order of each pair of elements in the list at least once. (Again: note that this can be through a direct comparison or as a result of the transitive closure of prior comparisons.)

Informal motivation: For comparison-based algorithms the entire computation is comparison-driven. The only way in which these algorithms gather information to sort a list is by making comparisons between list elements. In order to properly do this job, algorithms need to determine the relative order between each pair of elements from the list at least once. Otherwise they lack sufficient information to carry out the sorting.

- **2) Faithfulness**

Comparison-based algorithms execute in exactly the same way on input lists that share the same relative order.

Informal motivation: The entire computation is controlled only by means of comparisons between elements. Two lists sharing the same relative order have the same binary comparison paths. For two such lists L_1 and L_2 , the outcome of a comparison between every twin pairs $L_1[i], L_1[j]$ and $L_2[i], L_2[j]$ always yields the same outcome. Hence the computation will proceed in exactly the same way on each of these lists. The outcomes of a comparison in an if-then-else statement will be the same (i.e. the same branch of the if-then-else statement will be executed) and while loops while terminate at the exact same point. No sudden changes arise during computations on order-similar lists. The execution of the algorithm remains “faithful” to the (relative) order.

- **3) Separation**

Comparison-based algorithms executed on inputs that do not share the same relative order must diverge in their computation on these lists.

Informal motivation: Lists with distinct binary comparison paths must differ in relative order as somewhere along the line a different decision is made based on a comparison. We say that the binary comparison paths “separate” lists according to relative order. Equivalently, two lists sharing the same comparison paths must share the same relative order.

- **4) Reduction: the 0-1 principle**

In order to check that a comparison-based algorithm is a sorting algorithm, it suffices to verify that it sorts all *binary sequences*.

Informal motivation: Since for comparison-based algorithms the core of the computation is the determination of the outcome of comparisons, we can test the execution of the algorithm for binary lists only. A pair of list elements that is in the correct order can be represented

by the pair $(0, 1)$. A pair of list elements that is out of order can be represented by the pair $(1, 0)$. And a pair of list elements that happens to consist of identical elements is represented by $(0, 0)$, or, alternatively $(1, 1)$.

This fact helps to simplify the verification of correctness of comparison-based sorting algorithms. To check that such algorithms are guaranteed to sort arbitrary lists, it suffices to check that they sort all *binary* sequences.

7.5 Completeness

The notion of completeness captures the fact that in order for an algorithm to properly sort a list, the algorithm needs to compare each pair of elements in the list at least once, to determine the order between them.

This is the case for comparison-based algorithms for which the entire computation is comparison-driven. In other words, the only way in which these algorithms gather information to sort a list is to make comparisons between list elements. Hence it is clear that in order to properly do this job, algorithms need to compare each pair of elements from the list at least once. Otherwise they lack sufficient information to carry out the sorting. This is shown more formally in the following.



Completeness

Theorem 63 *A comparison-based sorting algorithm must determine the order between each pair of elements in its input list at least once, i.e., given a list $L = (L[1], \dots, L[n])$ then for each $i, j \in \{1, \dots, n\}$ the algorithm must at least determine the order between $L[i]$ and $L[j]$ once through prior comparisons in its execution.*

Read the following argument. Is it correct?

First attempt at a proof

If a sorting algorithm did not obtain sufficient information through a direct comparison or from its prior indirect comparisons to determine the order between two list elements $L[i]$ and $L[j]$ of some list L , then the algorithm would not be able to sort the list. This follows from the fact that a comparison-based sorting algorithm can only gain information about an input by comparing pairs of list elements. If the order between a particular pair of list elements is never determined, then no swap between these elements will ever occur.

Say the list elements for which no conclusion is reached on their relative order by the algorithm are the elements $L[i]$ and $L[j]$. Say, without loss of generality, that $L[i] < L[j]$ (The case where $L[i] \geq L[j]$, can be settled through similar reasoning.)

Consider the list

$$L = (L[1], \dots, \mathbf{L[j]}, \dots, \mathbf{L[i]}, \dots, L[n]),$$

where the out-of-order pair $L[j], L[i]$ has been highlighted in bold.

A comparison-based sorting algorithm that never determines the order between $L[i]$ and $L[j]$ through prior comparisons (whether these be direct or indirect) will never swap these two elements and thus will produce an output for L in which $L[j]$ occurs before $L[i]$. This is a contradiction since the algorithm is assumed to sort all inputs, where the output for L has not been sorted.

The proof is wrong! Can you see why? Note that the elements, say $L[i]$ may well be compared with elements other than $L[j]$ and swapped on that basis. The same holds for $L[j]$. Comparisons with elements other than $L[i]$ may well move $L[j]$ to different positions in the list. So how do we know that $L[i]$ is not swapped to a position after $L[j]$, hence placing them in the right order? We don't.

To fix the argument, consider the following proof.

Proof: The argument above has taught us something. To ensure that the elements are not swapped around based on comparisons with other elements, we consider an input list that is quite “stable”, i.e., an input list for which the two elements we consider will not be swapped around much when they are compared with other elements.

To make matters concrete, let's consider lists of size 12 and say the algorithm never determines the relative order between the fourth element and the eighth list element during the execution. Consider the binary list

$$(0, 0, 0, \mathbf{1}, 1, 1, 1, \mathbf{0}, 1, 1, 1, 1)$$

obtained from the sorted binary list

$$(0, 0, 0, \mathbf{0}, 1, 1, 1, \mathbf{1}, 1, 1, 1, 1)$$

by interchanging the bits in positions four and eight.

Now, when we execute the algorithm, the bold faced bits are never compared directly nor is their relative order ever determined through a transitive closure of other comparison outcomes. All we need to observe is that when the relative order between any bold-faced bit and a non-bold-faced bit is determined, this relative order is always the correct order and hence no swaps occur between bold-faced and non-bold-faced bits—except for the bold-faced zero following the three non-bold-faced ones. Hence this zero may well be swapped closer to the bold-faced one. But it will never be swapped to the left of it.

There is one exception. Since we allow swaps between equal elements, even if the bold-faced zero is swapped with a non-bold faced zero that forms part of the prefix (0,0,0) of the list, the list does not change, i.s. the bold-faced zero is again replaced by a zero to the right of the bold-faced one and the list is still not in the right order. Purely non-bold-faced bit pairs also are never swapped either since all such pairs are in the correct order. So the situation is never altered.

The bold-faced bits stay out of order during the computation and thus the algorithm is not a sorting algorithm. This argument (by way of contradiction) shows that a comparison-based algorithm needs to determine the order between all pairs of a list elements.

□

Not all comparison-based algorithms are complete in this sense of course. Consider for instance the algorithm that compares the first two elements of a list and swaps these in case the elements are out of order and then terminates. The algorithm is comparison-based but not complete.

Definition 64 *We say that a comparison-based algorithm which takes lists as inputs is complete iff it determines the relative order between each pair of elements of its input list at least once.*

Exercise 65 *Is a complete comparison-based algorithm necessarily a sorting algorithm? If your answer is “yes”, then prove that this result holds. If your answer is “no”, then provide the pseudo-code for such a complete comparison-based algorithm that is not guaranteed to sort and provide an example of an input list that is not sorted by the algorithm.*

7.6 Faithfulness

Faithfulness means that comparison-based algorithms execute “in exactly the same way” on input lists that share the “same relative order.” In other words, the execution of the algorithm remains faithful to the (relative) order. For lists sharing the same relative order, no sudden changes arise during the computation. Faithfulness once again is quite an obvious property of comparison-based algorithms. For such algorithms the entire computation is controlled only by means of comparisons between list elements. For two lists sharing the same relative order, i.e. for two lists L_1 and L_2 for which the outcome of a comparison between every twin pairs $L_1[i], L_1[j]$ and $L_2[i], L_2[j]$ always yields the same outcome. Hence the outcome of each boolean condition in an if-then-else statement will be the same (i.e. the same branch of the statement will be executed) and while loops while exit at the exact same point.



Faithfulness

Theorem 66 *Comparison-based sorting algorithms execute in exactly the same way on input-lists that share the same relative order. The binary comparison-paths for such lists are identical. We say that comparison-based algorithms execute in an “order-faithful way.”*

Proof: Recall that a comparison-based sorting algorithm is an algorithm for which every *swap* between list elements $L[i]$ and $L[j]$ is based on prior comparisons between list elements where these comparisons lead to the conclusion that the two elements $L[i]$ and $L[j]$ are out of order, i.e. $i \leq j$ and $L[i] \geq L[j]$. Clearly, if two lists share the same relative order, each comparison between list elements, say between a pair $L[i]$ and $L[j]$ in the first list L and a twin pair of elements $L'[i]$ and $L'[j]$ in the second list L' will lead to the exact same outcome. Hence, their binary comparison-paths are identical. In all cases, the swaps carried out on such twin pairs of list elements are based on prior comparisons between these list elements, which led to the exact same conclusion for each each list. Hence the algorithm executes the same comparisons and swaps for both lists. The computations thus proceed in exactly the same way. □

7.7 Separation

By faithfulness, we know that two lists sharing the same relative order have the same binary comparison paths. The converse is also true. Two lists sharing the same binary comparison paths share the same relative order. Equivalently, lists with distinct binary comparison paths must differ in relative order. Hence the binary comparison paths “separate” lists according to relative order.



Faithfulness and Separation combined

Theorem 67 *If S is a comparison-based sorting algorithm and L_1 and L_2 input lists of the same size then:*

L_1 and L_2 share the same relative order if and only if L_1 and L_2 share the same binary comparison path.

Next, we consider the particular case where we restrict the inputs to the permutations of size n of which there are $n!$ —each case of which represents *different* relative order.



Bijection between inputs and binary comparison paths

Corollary 68 *Consider the $n!$ inputs lists representing the possibly orderings an input list of size n can occur in. The function which maps each input list L to its binary comparison path $BCP_S(L)$ is a bijection.*

7.8 Reduction: the zero-one principle

The zero-one principle states that in order to check that a comparison-based algorithm is a sorting algorithm, it suffices to verify that it sorts all *binary sequences*. In other words our verification space can be reduced from general lists to binary sequences.

A (non-exhaustive) test to check whether a given algorithm sorts data properly, could constitute of running the algorithm on lists of size n , where we consider, say, the $n!$ permutations on the numbers $1 \dots n$. The zero-one principle implies that to carry out such a test, it suffices to check whether the algorithm sorts the 2^n binary sequences of size n , reducing the number of cases from $n! \sim \sqrt{2\pi n} \left(\frac{n}{e}\right)^n$ to 2^n . In other words, the size of the number of cases drops from $O(n^n)$ to $O(2^n)$. Since $n^n = 2^{\log_2(n)n}$ we obtain a reduction of $O(2^{\log_2(n)n})$ to $O(2^n)$. To put this in perspective, consider the example of $52!$ considered in Section 8.3.5. Recall that:

$$52! = 80658175170943878571660636856403766975289505440883277824000000000000$$

while

$$2^{52} = 4503599627370496$$

Of course, in practice, 2^n is still too fast-growing in general to carry out tests over meaningfully large sizes. We can reduce the size of this collection slightly more by eliminating all sorted binary sequences from this list¹.

The zero-one principle plays a key role in the context of “distributed sorting” where the process of sorting is hardwired through so-called “sorting networks.” Such sorting networks consist of “comparators” that sort pairs of values in the correct order. These comparators are organised in series and parallel. For such sorting networks, taking e.g. 8 inputs only, the number of binary sequences 2^8 on which the network needs to be tested is of feasible size and the zero-one principle helps to reduce the number of test cases from general sequences of inputs to binary input sequences.

To demonstrate that the 0-1 principle holds, we first show that if you apply an increasing function to all values of an input list L to produce a new input list L' , a comparison-based algorithm will “compute in exactly the same way” (i.e., follow the same computation path along all decision-branchings) on the new input list L' as on the original input list L . For instance, if you square all values of an input list $L = (a, b, c, d)$ containing only positive numbers², the result is the list $L' = (a^2, b^2, c^2, d^2)$. A comparison-based algorithm will follow the same computational path on both lists, i.e., if the algorithm happens to sort the original list L then it must sort the list L' as well.

¹See Exercise 74. For the case of binary lists of size 4, this reduces the number of cases from 16 to 11

²This guarantees that “squaring” is an increasing function.



Monotonicity

Lemma 69 (*Monotonicity Lemma*) *If a comparison-based algorithm transforms*

$$(a_1, a_2, \dots, a_n) \text{ into } (b_1, b_2, \dots, b_n),$$

*where (b_1, \dots, b_n) is a permutation of (a_1, a_2, \dots, a_n) ,
then for any increasing function f , the algorithm transforms*

$$(f(a_1), f(a_2), \dots, f(a_n)) \text{ into } (f(b_1), f(b_2), \dots, f(b_n)).$$

Proof: For comparison-based algorithms, the only actions that change the lists are swaps between two list elements, say a_i and a_j , based on a prior comparison between these elements. Note that by definition of an increasing function f^3 :

$$a_i \leq a_j \Rightarrow f(a_i) \leq f(a_j).$$

So, the pairs (a_i, a_j) and $(f(a_i), f(a_j))$ always are in the same relative order. Indeed, we both have $a_i \leq a_j$ and $f(a_i) \leq f(a_j)$ or we both have $a_i \geq a_j$ and $f(a_i) \geq f(a_j)$.

Hence for any comparison made by the algorithm between a_i and a_j , the algorithm will conclude the same outcome (0 or 1) as for a comparison between $f(a_i)$ and $f(a_j)$. Since our algorithm is comparison-based, i.e. all swaps are based on a prior comparison, it is clear that if the algorithm determines that a_i and a_j occur out of order (as based on prior comparisons carried out during its execution), and the algorithm hence executes a swap on the pair a_i and a_j , then the algorithm must reach the same conclusion on the *images* of those prior comparisons under the function f , and hence also execute a swap on $f(a_i), f(a_j)$. Thus, if a comparison-based algorithm transforms

$$(a_1, a_2, \dots, a_n) \text{ into } (b_1, b_2, \dots, b_n)$$

then the algorithm transforms

$$(f(a_1), f(a_2), \dots, f(a_n)) \text{ into } (f(b_1), f(b_2), \dots, f(b_n)).$$

□

Remark 70 *The functions $f(x) = x^2, f(x) = \log(x), f(x) = e^x$ are examples of increasing functions for which Lemma 69 applies.*

Exercise 71 *Consider the sequence $(3, 2, 6, 1)$. Check that the lemma holds for the case of the Insertion Sort and the increasing function $f(x) = x^2$.*

³Also referred to as a “monotone” function, hence the name of this lemma.

We prove that the zero-one principle holds for comparison-based algorithms, using Lemma 69. We will assume, by way of contradiction, that a comparison-based algorithm sorts all *binary* sequences but not all arbitrary sequences. So we assume, by way of contradiction, that on some non-binary input-list L the algorithm produces a non-sorted output.

We will define a bit-valued increasing function f and we will apply this function f to all values of the input list L . This will transform the original input-list L into a binary input sequence, say B . Lemma 69 then implies that this binary input list B is transformed into a non-sorted binary output list. This gives a contradiction with the fact that we assumed that the algorithm sorted all binary sequences. We make this argument precise in the proof given below.



Reduction: 0-1 principle

Theorem 72 *If a comparison-based algorithm sorts each of the 2^n binary sequences of length n then it sorts all sequences of length n .*

Proof: Suppose that the zero-one principle is not true, i.e., assume that there exists a comparison-based algorithm that sorts all binary sequences, but for which there is a non-binary input sequence (a_1, a_2, \dots, a_n) that does not get sorted. Hence in the output sequence produced for this input sequence, say $(a_{i_1}, \dots, a_{i_n})$ (a permutation of the list (a_1, \dots, a_n)), there exists a pair of values a_{i_k}, a_{i_l} with $k \leq l$ but $a_{i_k} > a_{i_l}$, i.e. a_{i_l} comes after a_{i_k} in the output sequence and these two values are out of order, i.e., $a_{i_k} > a_{i_l}$.



Binary witness-generator

Define the increasing function f as follows:

$$\text{if } x \leq a_{i_l} \text{ then } f(x) = 0 \text{ otherwise } f(x) = 1$$

Hence f indicates whether its input value x is strictly greater than a_{i_l} (by assigning the value 1 in that case).

We check that f is increasing. Indeed, assume that $y < z$. We must verify that $f(y) \leq f(z)$. Distinguish three cases:

$$1) a_{i_l} < y < z \quad 2) y \leq a_{i_l} < z \text{ and } 3) y < z \leq a_{i_l}$$

In the first case $f(y) = f(z) = 1$, in the second case $f(y) = 0$ and $f(z) = 1$ and in the third case $f(y) = f(z) = 0$. So in all three cases (the only cases that can occur) we have $f(y) \leq f(z)$. Thus f is increasing.

Recall that we assume that a_{i_l} comes after a_{i_k} in the output, where $a_{i_k} < a_{i_l}$. Hence, since f is increasing, we can use Lemma 69 to conclude that if we run the

comparison-based algorithm on the input $(f(a_1), f(a_2), \dots, f(a_n))$ then $f(a_{i_l})$ must come after $f(a_{i_k})$ in the binary output sequence produced by the algorithm. But $f(a_{i_l}) = 0$ and $f(a_{i_k}) = 1$ (since $a_{i_l} < a_{i_k}$). So a “one” precedes a “zero” in our output sequence, which means the binary output sequence is not sorted—a contradiction with our assumption that the algorithm sorts all binary sequences.

□



Binary witnesses

We call the newly produced binary list (in the above proof) a “binary-witness” of the original list.

Given a list (a_1, \dots, a_n) which is transformed by a comparison-based algorithm into a non-sorted list $(a_{i_1}, \dots, a_{i_n})$, which is a permutation of the list (a_1, \dots, a_n) . We assume that $(a_{i_1}, \dots, a_{i_n})$ is not sorted, hence for some k, l with $k \leq l$ we have $a_{i_k} > a_{i_l}$. Pick the smallest of these, i.e. a_{i_l} and define the function f as follows:

$$\text{if } x \leq a_{i_l} \text{ then } f(x) = 0 \text{ otherwise } f(x) = 1$$

The binary-witness of the sequence (a_1, \dots, a_n) is the list $(f(a_{i_1}), \dots, f(a_{i_n}))$.

The function f on a given argument x will output 1 in case $x > a_{i_l}$ and 0 otherwise. In other words, f “identifies” the values x that are strictly greater than a_{i_l} .

Exercise 73 *Given the comparison-based algorithm consisting of an execution of the inner loop transversal of the input list as produced by Bubble Sort. I.e., consider the comparison-based algorithm consisting of the following pseudo-code:*

Single-Bubble-Sweep

```
for  $j = 1$  to  $|L| - 1$  do
  if  $L[j] > L[j + 1]$  then
    swap ( $L[j], L[j + 1]$ )
```

Determine the output for the input list $(1, 3, 7, 2, 4, 5)$. Check whether any two elements in the output list are still out of order. Compute the binary-witness list for this pair. Run the algorithm on this binary-witness list and check that this algorithm does not sort this list.

Solution: Note that Single-Bubble-Sweep transforms $(1, 3, 7, 2, 4, 5)$ into the list $(1, 3, 2, 4, 5, 7)$. This list is not sorted. The elements 3 and 2 are out of order. 2 is the smaller one, so define the function f as follows:

$$\text{if } x \leq 2 \text{ then } f(x) = 0 \text{ otherwise } f(x) = 1$$

Now, apply f to each element of the output list $(1, 3, 7, 2, 4, 5)$. We obtain

$$(f(1), f(3), f(7), f(2), f(4), f(5)) = (0, 1, 1, 0, 1, 1).$$

This binary-witness list does not get sorted by Single-Bubble-Sweep. The output when running Single-Bubble-Sweep on $(0, 1, 1, 0, 1, 1)$ is $(0, 1, 0, 1, 1, 1)$, which is not sorted.

Exercise 74 Say you want to test whether a comparison-based sorting algorithm sorts all binary sequences of size n . Show that it suffices to check whether the algorithm sorts all non-sorted binary sequences of size n . Illustrate the reduction achieved by considering the binary sequences of length 4. Show that the number of lists to be tested drops from 16 to 11.

Exercise 75 Does the zero-one-principle hold for non-comparison-based algorithms? If so, prove that the principle holds for such algorithms. If not, produce an algorithm that is 1) non-comparison-based, 2) sorts all binary sequences and 3) for which there exists a sequence of integers that it does not sort.

Exercise 76 (Binary witness-lists form a lossy encoding)
Consider a list $L = (a_1, \dots, a_n)$. For each list element a_i of this list we define a function f_{a_i} in the same spirit as the function used to produce binary-witnesses:

$$\text{if } x \leq a_i \text{ then } f_{a_i}(x) = 0 \text{ otherwise } f_{a_i}(x) = 1$$

This function f_{a_i} can be applied to each element of L to produce the binary list

$$(f_{a_i}(a_1), \dots, f_{a_i}(a_n)).$$

In general you cannot reverse this process, i.e. it is not possible in general to recover the list $L = (a_1, \dots, a_n)$ from a single binary list $(f_{a_i}(a_1), \dots, f_{a_i}(a_n))$. In other words, such a binary list forms a “lossy encoding.”

Consider the list of binary lists produced by the functions $f_{a_1}, f_{a_2}, \dots, f_{a_n}$ when applied to the list $L = (a_1, \dots, a_n)$:

$$(**) [(f_{a_1}(a_1), \dots, f_{a_1}(a_n)), \dots, (f_{a_n}(a_1), \dots, f_{a_n}(a_n))]$$

Show that this list of binary lists together with the set $\{a_1, \dots, a_n\}$ of all elements contained in the list L , is sufficient to recover the original list $L = (a_1, \dots, a_n)$. Specify a decoding that computes L from this list of binary lists and the set $\{a_1, \dots, a_n\}$.

Hint: Consider a list $L' = (7, 1, 5, 3)$. Check how you can recover this list from the list of binary lists (displayed below) and where you can make use of the set of values $\{7, 1, 5, 3\}$ to achieve the decoding.

$$[(f_7(7), f_7(1), f_7(5), f_7(3)), (f_1(7), f_1(1), f_1(5), f_1(3)), \\ (f_5(7), f_5(1), f_5(5), f_5(3)), (f_3(7), f_3(1), f_3(5), f_3(3))]$$

Compute the actual bit-values for each element of these binary lists and explain how you can decode the original list from the given list of four binary lists and from the set of values $\{7, 1, 5, 3\}$.

Now, generalize your approach to arbitrary lists L and explain how to decode such a list from its set of values and the list of binary lists displayed in $(**)$ above.

Chapter 8

Basic complexity measures

In almost every computation a great variety of arrangement for the succession of the processes is possible, and various considerations must influence the selections amongst them for the purpose of a calculating engine. One essential object is to choose that arrangement which shall tend to reduce to a minimum the time necessary for completing the calculation.

Ada Lovelace

Complexity can be approached in the general context of computable functions, through the theory of Blum axiomatisation. We take a different approach here, where we look at lower level examples of complexity, i.e. traditional complexity measures such as worst-case and average-case time for basic algorithms, in the spirit that we need to learn to walk before we can run.

8.1 An introduction to software timing

8.2 Timing methods: an evaluation

We will tacitly assume that all our programs terminate, i.e. timing, in principle, is possible.

The simplest way to time a program when executed on a given input is to use a clock.

Method 1: Clock-based timing

- a) Use a clock. For each given input, measure how long it takes to produce the output (again: we consider cases where an output is produced for each input, i.e. the program terminates on each input!).
- b) Record the running times on a “large” number of inputs.
- c) Plot a graph of the results obtained in b).

Method 1 is not suitable in general. We outline some of the difficulties involved.

Disadvantages of Method 1

- 1) the running time measured in this way is *machine dependent* (for different machines we may get different times)
- 2) method 1 is not only machine-dependent but also *input dependent*.

Regarding 1), note that on differently designed machines, method 1) is bound to give different timing results. Hence our results are machine-dependent. So in order to record our results, we need to note which machine was used and the results will be useless for someone using a different machine, unless there is an explicit and very detailed comparison between the architectures of both as well as a formal way to translate timing results obtained on the first machine to the second. Typically this is not available in practice. Technology advances too rapidly to make this feasible in general.

Regarding 2), note that method 1 provides limited information. Using method 1, we can only record and graph a *finite* number of running times, since we can only measure the time on a finite number of inputs. Some algorithms, such as sorting and search algorithms, can take an infinite amount of inputs.

Of course we can always decide to record some more running times for additional inputs, but we can never know ahead which inputs we will want to consider in practice. This depends on the context in which the algorithm will be applied.

Moreover, the computer under consideration may actually enable one to handle such a large amount of inputs that it is infeasible to measure the time for each possible input and to plot a graph and study the results. This means that improved timing needs to aim for two goals:

Goals to improve timing

- 1: eliminate machine-dependency
- 2: eliminate input-dependency

The following method provides a solution which reaches goal 1.

Method 2: static time analysis

- a) assign a time-unit to each “basic step” occurring in the pseudocode. Basic steps can include swaps between two values, assignments of a value to a variable or comparisons between values.
- b) the running time of the algorithm on an input is obtained by adding up all time-units of basic steps carried out during the execution of the algorithm.

Static timing typically involves the following process in assigning time units:

Time units

For an algorithm A we can assign unit times to the following steps:

- each assignment takes a time-unit c_1
- each comparison takes a time-unit c_2
- each swap between two values takes a time-unit c_3

We have been silent about potential differences in the cost of basic steps. Indeed, our approach has been to consider the cost of a comparison as a fixed unit (and similarly for the costs of assignments and swaps). Is this a reasonable assumption? Why can we regard the cost of comparing say the comparison between 1 and 2 to be “similar” to the cost of comparing 1,000,000 with 40,000,000,000?

Upper bounds on time units

For a given computer, one can determine upper bounds (in microseconds) on the values of each assignment, comparison and swap.

Exercise 77 *Explain why such upper bounds can be determined.*

Machine independent timing

Counting time units by analyzing the source code allows us to compare two algorithms in a precise machine-independent way.

Once upper bounds are known for the time units, we can set c_1, c_2 and c_3 to these respective upper bounds and hence estimate the running time of executing the algorithm on a particular input on a given computer.

Hence we achieved machine-independency (where our result can be estimated for any given computer for which we obtain upper bounds on the values of c_1, c_2 and c_3).

Runtime

Definition 78 *The running time of an algorithm A on an input I is denoted by $T_A(I)$. $T_A(I)$ expresses the total number of time-units for all the basic steps taken by A during its execution on input I .*

For the case of comparison-based algorithms, we will typically only count the number of comparisons carried out in the execution as these give a very good indication of the time in which the algorithm will actually run. Indeed, recall that comparison-based algorithms carry out swaps and assignments on the basis of priorly executed comparisons. Hence comparisons precede other operations, and, indeed, drive the entire computation. If we want to fine tune the analysis by counting other basic steps, such as swaps and assignments, this can typically be done by determining an upper bound K on the number of swaps and assignments following each comparison. An upper bound on the running time of the algorithm is then obtained by multiplying K with the comparison time.

Comparison-time

Definition 79 *The comparison-time of a comparison-based algorithm A on an input list L is denoted by $T_A(L)$. $T_A(L)$ expresses the total number of comparisons taken by A during its execution on input L .*

There still are some issues to be addressed in relation to Method 2.

Disadvantages of method 2

Method 2 still is input-dependent. Indeed, it is still the case that we can only obtain the exact number of computation steps for *each* input via Method 2. Once again, we could plot a graph after making a determination of the time on a large number of inputs, but we cannot get the complete picture for *all* inputs

, (as this is a potentially infinite amount).

8.3 Size-based methods

One way to make the running time analysis less input-dependent is to focus on input collections sharing the same input size. The goal is to obtain information on all inputs of a given size n . The notion of size is well-defined for a variety of data structures. For instance: the size of a list is its length, the size of a graph (including of course trees) is the number of nodes in the graph etc.

The advantage of focusing on the collection of data structures sharing the same size is that the potentially infinite amount of such data can be reduced to a finite number of cases for the case of comparison-based algorithms.

8.3.1 Repeated elements

When we carry out the analysis of algorithms, we will typically only consider lists that have no repeated elements. Our approach is to assume that our comparison-based algorithms will carry out some pre-processing on each input list L .

Assigning tie-breakers

For each input list L of length $|L| = n$, we assign distinct numbers in the range from 1 to n in an independent and uniform way to each list element.

In other words, to each list element $L[i]$ one assigns a random value from $\{1, \dots, n\}$, where no list element is assigned the same value¹. Each list element $L[i]$ is paired with a natural number value k in the range of 1 to n , i.e. lists will consist of pairs $(L[i], k)$.

¹This assumption constitutes only a linear cost, which is not an issue for this type of algorithms satisfying a $\Omega(n \log n)$ lower bound. Note that if we do not work under this assumption of assigning tie-breakers from the start to all elements (whether they are repeated or not), typically, tie-breakers are introduced on the fly during the execution of the algorithm, whenever equal list elements are encountered. This is a more frugal approach. However, if the same pair is encountered later on, randomly generated tie-breakers may impose the opposite order between them. This approach requires, for some theoretical considerations, that no redundant comparisons are made in the execution of the algorithm, i.e. tie-breakers are only invoked once. This requirement is for instance made in the proof of the zero-one principle presented in Knuth, Vol 1, Section 5.3.1 Exercise 12 [M25].

Breaking ties

Upon execution of our comparison-based algorithm, we use these randomly assigned numbers as tie-breakers whenever two elements are compared which happen to be the same. In other words, if we compare $L[i]$ with $L[j]$ and these list elements happen to be the same, then we still decide on $L[i]$ and $L[j]$'s order by comparing their assigned numbers, say k and l , and we consider $L[i] \leq L[j]$ in case $k \leq l$ and $L[i] > L[j]$ in case $k > l$.

In this way, our algorithms will execute as if all elements are distinct, and will carry out swaps even for the case where $L[i]$ and $L[j]$ are the same, but their assigned numbers, say k and l are out of order, i.e. $k > l$.

The analysis remains the same

The tie-breaking convention implies that our analysis of the number of comparisons carried out during the computation can be done under the assumption that we only need to consider lists that have no repeated elements.

8.3.2 Reduction of the analysis to finitely many input cases

By our tie-breaker convention, we can assume, when carrying out the time analysis, that the elements of the lists are not repeated. This will simplify the analysis. Moreover, we can assume that we only need to consider finitely many cases.

Finitely many cases to consider

For comparison-based algorithms and a given input size, there are essentially only finitely many cases of different input lists to consider for that size.

Consider for instance the case of lists of length n , where we do not allow repetition of elements in the list. For such lists, we need only consider $n!$ different lists of this fixed size n .

We illustrate this on the lists of length 3 where we assume that the lists store natural number values. Clearly there are an infinite number of such lists. For instance, consider the following infinitely many cases of a sorted list:

$$(1, 2, 3), (2, 3, 4), (3, 4, 5), \dots, (1000, 1001, 1002), \dots \quad (*)$$

These lists can all be represented by the single list (a, b, c) where $a < b < c$ are three distinct numbers.

In general, any of the infinite number of possible lists of size 3 (without repeated elements) can be represented via $3! = 6$ cases.

Again, pick three values, a, b and c , where $a < b < c$. This leads to the following six possible cases:

$$(a, b, c), (a, c, b), (b, a, c), (b, c, a), (c, a, b), (c, b, a) \quad (**)$$

where we recall that, by Section 8.3.1, we only need to consider lists that have pairwise distinct elements, i.e. no two elements are repeated.

Each of the infinite possible lists of size 3 corresponds, if we only take into account the relative order between the list elements into account, uniquely to one of the six possible lists displayed in (**) above.

For instance, the list $(3, 1, 2)$ as well as the list $(100, 3, 19)$ both correspond uniquely to the list (c, a, b) .

In general we will consider data structures for which there are only finitely many possible “cases” for any given size n . Another example of such a data structure satisfying our finiteness condition is the “heap” data structure, consisting of a binary tree in which every node is labelled with a natural number, such that parent nodes always have labels greater than or equal to their children (cf. Section 11.1).

As we will see, this finiteness-condition is a crucial aspect of the definition of the worst-case time and the average-case time (which may not make sense for the infinite case).

Note that Theorem 66 immediately implies the following result:



Invariance of timing

Theorem 80 *Comparison-based algorithms take the exact same comparison time on inputs for which the relative order between elements is the same.*

Consider for instance Insertion Sort. Insertion Sort will take exactly the same number of comparisons on each of the lists from the infinite collection (**) displayed above, namely 2 comparisons. The computation is entirely determined by the order between the elements of the list, but not by their actual values.

Hence, to determine, say, the longest running time of Insertion Sort on all lists of size 3, we only need to analyze how long Insertion Sort will take on each of the six lists from (**).

In general, when we carry out an analysis of the number of comparisons carried out during the execution of a comparison-based algorithm, we can restrict the analysis to the $n!$ representative input-cases consisting of all permutations of the numbers from 1 to n , i.e. the lists of size n for which elements take values in $\{1, \dots, n\}$ and for which none of the lists have repeated elements.

Hence in the analysis of comparison-based algorithms, we can reduce the need to check the time on an infinite number of inputs to a finite number of cases.

8.3.3 Main timing measures

We will focus on two main timing measures. The first is the worst-case running time, the second is the average-case running time. The worst-case time allows one to determine the longest time an algorithm will run on any of its inputs.

Worst-case time is the main measure used in practice, e.g. in Real-Time Programming where scheduling tasks is an essential part of designing safety-critical systems. Once we know the worst-case time of say an algorithm A , we can schedule a second algorithm B to safely run after A is completed. For this, we simply need to ensure that the system waits for A 's worst-case time to pass before running B .

Consider for instance an algorithm that runs slow on a particular input, but much faster on all other inputs, such as the algorithm Quick Sort. Quick Sort can be shown to have $O(n^2)$ on its worst-case inputs, namely the sorted or reverse sorted list, or any lists “near” these cases². However, Quick Sort's average-case running time is $O(n \log n)$.

For Real-Time applications, where scheduling depends purely on the worst-case information, it makes sense to replace Quick Sort with an algorithm that may run a bit slower on average, but with a better worst-case—a decision that requires knowledge of both the worst-case and the average-case time information.

Worst-case time provides limited information. If we only consider the worst-case time, we may very well reject say an $O(n^2)$ worst-case time algorithm such as Quick Sort for an algorithm A that runs say in $O(n^{3/2}(\log(n))^2)$ on average and in worst-case time. Clearly Quick Sort, running in $O(n \log n)$ in the average case, beats A in average time. Hence, in general, both measures need to be considered to make a considered decision³.

For applications other than Real-Time programming, one typically prefers the faster Quick Sort algorithm, where, if needed, fast execution can be guaranteed by randomizing the input lists first (to avoid a situation where the worst-case would occur too frequently due to the data distribution being unfavourable for fast execution, e.g. a collection of input data consisting of lists, half of which are sorted).

Worst-case analysis

Worst-case analysis is a size-based method. We consider an input data structure for which the number of cases of inputs of size n is $i(n)$, where these inputs are given by: $I_1, I_2, \dots, I_{i(n)}$. We denote the set of all inputs of size n by \mathcal{I}_n .

For the case of lists of size n , $i(n) = n!$

²This can be made precise by Spielman's notion of smoothed complexity. For our purposes it suffices it to note that the worst-case input for Quick Sort takes place on a neglectably small portion of inputs.

³For average-case time it also makes sense to also consider standard deviation information (a topic beyond the scope of this course).

The worst-case running time of a given algorithm A on inputs of size n is the greatest (= worst) running time which the algorithm takes when executed on any input of size n .



Worst-case running time of an algorithm A

$$T_A^W(n) = \max\{T_A(I_1), T_A(I_2), \dots, T_A(I_{i(n)})\}$$

Note that this definition makes sense, since we assume that there are only finitely many inputs of size n .

This worst-case time for an algorithm A and input size n is denoted by: $T_A^W(n)$.

For each given size n , this worst-case time $T_A^W(n)$ is “input-independent” since it gives information on all inputs of size n at once; namely: *each* input of size n will be such that its output is computed by the algorithm A in a running time bounded by $T_A^W(n)$.

Average-case analysis

Worst-case analysis is a second example of a size-based method. Again, we consider an input data structure for which the number of cases of inputs of size n is $i(n)$, where these inputs are given by: $I_1, I_2, \dots, I_{i(n)}$.

We denote the set of all inputs of size n by \mathcal{I}_n .

The average running time for a given algorithm A on all inputs of size n is computed as follows: $\bar{T}_A(n)$ = sum of all running times on inputs of size n , divided by the number of inputs of size n .



Average-case running time of an algorithm A

$$\bar{T}_A(n) = \frac{T_A(I_1) + T_A(I_2) + \dots + T_A(I_{i(n)})}{i(n)}$$

Example: lists of size n , where the number of input list cases of size n is $i(n) = n!$

Say these lists are $L_1, L_2, \dots, L_{n!}$

$$\bar{T}_A(n) = \frac{T_A(L_1) + T_A(L_2) + \dots + T_A(L_{n!})}{n!}$$

For each given size n , this average time $\bar{T}_A(n)$ gives more input-independent information since it provides results on “all inputs of size n ” at once.

Remark 81 *The average-case running time concept tacitly assumes that the $n!$ input lists under consideration occur with equal probability. Hence the distribution on the inputs is assumed to be the uniform distribution and the average-case time reflects the expected average running time for this distribution.*

In practice distributions typically are not guaranteed to be uniform. If the distribution is quite skewed then algorithms may well have an expected time that may differ substantially from the average-case time determined under the uniform distribution assumption.

Consider an extreme case where the data consist mostly of sorted lists. An algorithm such as Quick Sort will, for this distribution, have the same order $O(n^2)$ for its average-case time as for the worst-case time. This could be remedied by randomizing the data prior to running the algorithm.

The average-case time is typically determined to obtain a first impression of how the algorithm will run “all things being equal”, i.e. under a uniform input distribution. Determining expected running times for arbitrary distributions remains quite an open area of research.

Trade-offs

We traded input-dependency for size-dependency—trading an infinite number of cases per input size for a finite number of cases. What have we gained? Clearly the reduction enabled us to define the worst-case and average-case timing measures. (As an exercise, check what issues might arise in the above definitions in the case the number of cases to be considered is infinite.)

Method 2 (summing costs of basic operations per input) is nice since (in theory) since it allows one to compute an algorithm’s running time per input.

The problem with method 2 is that in general it is not possible to write: $T_A(L)$ = “some formula” involving L . As we will see, there are however formulas that allow one to write:

$$T_A^W(n) = \text{“some formula” involving } n,$$

or

$$\overline{T_A}(n) = \text{“some formula” involving } n.$$

Remarks

1) Though this is clear from the above discussion, we will analyze the running time under a number of assumptions, including the assumption of a uniform distribution of inputs for average-case analysis.

For sorting algorithms for instance, we will compute the time based on the finitely

many cases identified above: the $n!$ different lists of size n . This tacitly assumes that the lists have equal probability to occur (i.e. have uniform distribution). Indeed, it is clear from the definition of the average-case time that this measure is computed based on the assumption that each list is given equal weight in the average.


Uniform distribution is of course not guaranteed to occur in practice (where lists could be produced by another algorithm that might, say, produce far more cases of sorted lists than any other type of lists). The uniform distribution however can be achieved through randomizing the data if needed (for sorting this can occur by “shuffling” (randomizing) the input list prior to sorting).

If in practice we suspect that randomizing data is not a good solution, e.g. for the case where we would suspect that many of the lists are already pre-sorted in part and we wish to take advantage of this to speed up the computation, machine learning can be used to analyze the data distribution and machine learning can guide the selection of algorithms known to execute better over certain data distributions.

2) Trade-offs arise between the worst-case and the average-case time measure. Worst-case time focuses on a very particular property, the worst-case time that can arise on a given input. In practice it will be easier to determine worst-case times than to determine average-case times, since the latter regards a statement about all inputs (their average-case time) as opposed to a single input (causing the worst-case). This will be reflected in the analysis techniques. Worst-case analysis will typically be obtained through simpler arguments than average-case analysis.

Average-case time is bounded by worst-case time

We remark that the worst-case and the average-case time satisfy the following relation:

 **Binding average-case by worst-case**
| Theorem 82 $\forall n. \bar{T}_A(n) \leq T_A^W(n)$

Theorem 82 states that the average-case time is always bounded by the worst-case time (as one might expect). We demonstrate this below.

Proof: For a given algorithm A and for lists of size n , we know that the worst case time, $T_A^W(n)$, is the greatest time which A takes on *any* input I of size n . So we know that for any input I of size n the following holds:

$$(*) T_A(I) \leq T_A^W(n).$$

Recall that the average time for A on inputs of size n is given by:

$$\bar{T}_A(n) = \frac{T_A(I_1) + \dots T_A(I_{i(n)})}{i(n)}$$

where the inputs $I_1, \dots, I_{i(n)}$ represent the inputs of size n .

For each such input I_i of size n , where i ranges between 1 and $i(n)$, we have, using equation (*) above, that $T_A(I_i) \leq T_A^W(n)$.

So we obtain:

$$\bar{T}_A(n) \leq \frac{T_A^W(n) + \dots + T_A^W(n)}{i(n)} = \frac{i(n)T_A^W(n)}{i(n)} = T_A^W(n)$$

Hence:

$$\bar{T}_A(n) \leq T_A^W(n)$$

□

Asymptotic classification of timing measures

Even though the size based methods, by reducing focus on a finite number of cases per input size, reduce input dependency, the problem of infinitely many input cases is not fully addressed. After all, we still have an infinite amount of sizes to consider. Hence, plotting say the worst-case time of an algorithm, where we record the worst-case values per input-size still runs into the problem that we can't produce the entire graph.

To achieve a form of input-independency we can consider Asymptotic Analysis of time ("Big-Oh analysis"). We will return to this approach in Section 9.7. Essentially, asymptotic analysis focuses on capturing the running time behaviour for inputs of "large size". In other words, asymptotic time analysis, as the name indicates, focuses on running time behaviour that occurs from a given point on, or, indeed, by taking the limit as sizes approaches infinity. Asymptotic analysis forms the subject of basic algorithms courses or introductions to mathematics for computer science. We assume familiarity with this topic, but for the sake of completeness we recall the main definitions below.

8.3.4 Overview of timing measures

We will formulate the overview in terms of programs rather than algorithms, but the ideas of course are the same.

The *exact time* $T_P(I)$ on an input I to be the number of basic steps made by the program P during the computation of the output $P(I)$. The notation $T_P(n)$ indicates the restriction of the function T_P to the set \mathcal{I}_n . We will consider subsets \mathcal{I} of \mathcal{I}_n and consider the following time measures with respect to \mathcal{I} :

Overview of timing measures

The *Total Time* of P for inputs from \mathcal{I} , denoted by $T_P^t(\mathcal{I})$ is defined by:

$$T_P^t(\mathcal{I}) = \sum_{I \in \mathcal{I}} T_P(I).$$

The *Worst-Case Time* of P for inputs from \mathcal{I} , denoted by $T_P^W(\mathcal{I})$ is defined by:

$$T_P^W(\mathcal{I}) = \max\{T_P(I) \mid I \in \mathcal{I}\}.$$

The *Best-Case Time* of P for inputs from \mathcal{I} , denoted by $T_P^B(\mathcal{I})$ is defined by:

$$T_P^B(\mathcal{I}) = \min\{T_P(I) \mid I \in \mathcal{I}\}.$$

The *Average-Case Time* of P for inputs from \mathcal{I} , denoted by $\bar{T}_P(\mathcal{I})$ is defined by:

$$\bar{T}_P(\mathcal{I}) = \frac{T_P^t(\mathcal{I})}{|\mathcal{I}|} = \frac{\sum_{I \in \mathcal{I}} T_P(I)}{|\mathcal{I}|}.$$

In order to denote an arbitrary measure, which can include any of the above, we use the notation \mathcal{T}_P and the usual corresponding notations $\mathcal{T}_P(\mathcal{I})$ and $\mathcal{T}_P(n)$.

We observe that:

Relations between timing measures

$$\forall \mathcal{I}. T_P^B(\mathcal{I}) \leq \bar{T}_P(\mathcal{I}) \leq T_P^W(\mathcal{I}).$$

For the special case of constant-time algorithms, the following equalities hold:

If the exact time of P is a constant C on the inputs from \mathcal{I} then:



Constant time algorithms

If the exact time of P on each input of \mathcal{I} is a constant C then:

$$\bar{T}_P(\mathcal{I}) = T_P^B(\mathcal{I}) = T_P^W(\mathcal{I}) = C.$$

Of course, in case $\mathcal{I} = \mathcal{I}_n$, we will for the Total, Worst-Case, Best-Case and Average-Case Time respectively use the following standard notation based on size indication only: $T_P^t(n)$, $T_P^W(n)$, $T_P^B(n)$ and $\bar{T}_P(n)$.

We recall the asymptotic classification of running times. Given two functions $f, g: \mathcal{N} \rightarrow \mathcal{R}^+$. Then



Asymptotic time

$$f \in O(g) \iff \exists c > 0 \exists n_0 \forall n \geq n_0. f(n) \leq cg(n).$$

$$f \in \Omega(g) \iff \exists c > 0 \exists n_0 \forall n \geq n_0. f(n) \geq cg(n).$$

For comparison-based algorithms one can show that in the asymptotic hierarchy the Worst-Case Time and the Average-Case Time satisfy the following lower bound: $T_P^W(n) \in \Omega(\log(N_n))$ and $\bar{T}_P(n) \in \Omega(\log(N_n))$ where N_n is the number of leaves in the decision tree of the algorithm P for inputs of size n (cf. Section 11).

This gives us a sense of how far we can improve comparison-based algorithms. No matter how much thought we put into their design, we can never do better than the lower bounds given above.

8.3.5 Brute force time computation for comparison-based sorting

To compute the worst-case time, we need to compute the running time of all inputs and select the worst-case among these. Similarly, for the average-case time, we need to take the average of all of these results (i.e. divide the sum of these times by the number of inputs). How many inputs might we need to consider? This depends on the size of the inputs.

Let's say we consider comparison-based sorting algorithms and say that the inputs are lists of size n . In that case we need to consider $n!$ inputs.

To compute the the worst-case time, we need to determine an input among $n!$ choices, that results in the longest comparison-time.

To compute the average-case time, we need to compute the comparison-time on each of the $n!$ inputs, sum the results and divide the total by $n!$. In each case, we need to compute over a collection of size $n!$. Hence it is important to gain a sense of how fast $n!$ grows in order to determine whether these computations would be feasible in practice.

Stirling's approximation

Stirling's approximation gives a sense of how $n!$ grows:



Stirling's approximation

$$n! \sim \sqrt{2\pi n} \left(\frac{n}{e}\right)^n.$$

For $n = 5$ we have $5! = 120$ permutations. Stirling's formula gives us: $5! \sim \sqrt{2\pi 5} \left(\frac{5}{e}\right)^5$ or roughly 118 (compute it as an exercise).

The exponentiation used in Stirling's approximation, clearly indicates that $n!$ grows fast. We consider a concrete example in the following, focusing on the number of ways one can shuffle a deck of cards.

Gauging the size of $52!$

To get a sense of this, consider shuffling a deck of 52 cards. The result of shuffling a deck of cards is one of $52!$ possibilities. When we compute $52!$ on a “large number calculator”⁴, we need to type in the product of the 52 first natural numbers⁵

The outcome is:

$$52! = 80658175170943878571660636856403766975289505440883277824000000000000$$

The number (a factorial of a relatively small number 52) is astronomically large. To get a sense of the size of this number, watch the film by Michael Stevens at:

<https://curiosity.com/paths/math-magic-vsauce/#math-magic-vsauce>

(start the video at the 14:16 mark and watch it until the end).

Next, read the related blog (which supplies the computational details):

<http://czep.net/weblog/52cards.html>

Czepiel's calculation gives a sense of the size of $52!$ Clearly, computing the average case time and worst-case time by brute force is not feasible as we would face the same situation for lists of size 52: there are $52!$ different input permutations to be

⁴such as for instance the one provided at: <http://web2.0calc.com>

⁵Try it out. It's a useful exercise to gain an appreciation for Stirling's approximation formula. Using Stirling's approximation we can estimate $52!$ (where we approximate e by 2.7182818284590452353602874713527): $52! \sim \sqrt{2\pi 52} \left(\frac{52}{e}\right)^{52}$ which (rounded) yields:

$$52! \sim 80529020383886611100085980003054963255286086559413182231073552585000$$

considered. As an exercise, write a program that computes the $n!$ permutations of size n . At which point (i.e. for which size of n) can you no longer store the $n!$ lists? (Typically this runs out around the 11 to 13 list-size mark).

Since computing the worst-case time or the average-case time for input lists of size 52 (a relatively small size for lists) is not feasible, we need to look into alternative techniques to determine the running time of algorithms. This is the topic of the next section. Before moving on, take a look at the next exercise, which presents different views of the size involved in $52!$

Exercises

1) Given that the average thickness of Bridge size cards is (roughly) 0.3 mm, the average deck size of bridge sized cards can be taken to be $52 \times 0.3mm = 1.56cm$. Say you purchase $52!$ deck of cards (yeah, good luck with that, but imagine you do buy them). Now order each deck into one of the $52!$ possible card orderings (again, good luck with that, but, strictly speaking, there is no need for this. We simply want to get an idea of how high our stack will be if we were to stack all $52!$ permutations (represented by the decks) on top of one another. To find this out, we don't need to put them in that order as any reshuffle has the same thickness). We get $52! \times 0.0156$ meter in height. How far is this in light years? A light year is 9.461×10^{15} metres. So the outcome is: $(52! \times 0.0156) / 9.461 \times 10^{15} = 1.3299519423599242 \times 10^{50}$ lightyears. Check the answer on an online "big number calculator". The star Deneb is the farthest star that can be easily seen with the naked eye. It is thought to be between 1,400 and 3,000 light years from Earth. Relate your answer to the following information: the Milky Way is probably between 100,000 and 150,000 light years across. The observable Universe is, of course, much larger. According to current thinking it is about 93 billion light years in diameter. The whole Universe is at least 250 times as large in diameter as the observable Universe.

2) Consider the same problem but this time compute the weight of $52!$ decks of cards, given that the weight of one deck is 94 grams. Compare this to the mass of the observable universe (10^{53} kg).

3) Not quite an exercise, but for this course involving exponential size considerations, the final "marble galaxy" scene of the first Men In Black movie may evoke some interesting considerations about relative size.

8.3.6 The Library of Babel

I still think of myself as primarily a fiction writer, but I sometimes wonder what the point is now. I've already published every possible story I could write...

*Jonathan Basile, creator of the
digitised Library of Babel*

The number $52!$ may seem huge. A visit to the library of Babel puts this in perspective⁶. “The Library of Babel” is a short story by Jorge Luis Borges (1899-1986), describing a vast library of hexagonal connected rooms, containing all 410-page books for which each page has 40 lines and each line about 80 letters. That makes $410 \times 40 \times 80 = 1,312,000$ characters. The story also mentions that there are 25 orthographic symbols (including spaces and punctuation). The Library contains at least $25^{1,312,000} \approx 1.956 \times 10^{1,834,097}$ books.

Every book conceivable, within this given size is contained in the library. The library of Babel hence offers accurate and inaccurate predictions of the future, biographies of all people future or present, translations of every book in all languages, books in non-existent languages satisfying obscure rules of grammar, all shuffled versions of books—including a massive amount of jabberwocky, gobbledygook and gibberish. Conversely, for many of the texts some language could be devised that would make it readable with any of a vast number of different contents. One of the books, the index of the library's contents, is a sought after treasure in the story.

One of the excursions of this course includes a visit to the Library of Babel (at <https://libraryofbabel.info>).

8.3.7 Beyond the Library of Babel

The size of the collection of books in the library of Babel is pretty much unfathomable. We have however encountered a far larger collection than this set: the infinite collection of the natural numbers. The set \mathbf{N} dwarfs the (finite) collection contained in the library of Babel. This infinite set of natural numbers is dwarfed in turn by the collection of all functions $\mathbf{N} \rightarrow \mathbf{N}$ from the natural numbers to the natural numbers encountered in Section 5.6. Indeed, this set of functions contains the powerset $2^{\mathbf{N}}$, which has “2 to the power *infinity*” elements—an exponential jump of the size of the natural numbers. To make this “number” “2 to the power *infinity*” precise, we

⁶Borges' short story “The Library of Babel” is a recommended read for this course. See also: Bloch, William Goldbloom. *The Unimaginable Mathematics of Borges' Library of Babel*. Oxford University Press: Oxford, 2008.

would need to introduce Cantor’s theory of infinite sets, and in particular the notions of ordinals and cardinals—a topic beyond the scope of this course.

The construction of the powerset however is clear (and has been discussed in Section 5.6). So we can play the same game again: just as Archimedes introduced a myriad-myriad times the number 10^8 , i.e. $(10^8)^{10^8}$, as part of his discussion of the number of grains of sand in the universe, we can repeat exponentiation on infinite sets over and over (by iterating the powerset operation)—creating ever larger infinite sets.

The first exponent in this sequence, the powerset $2^{\mathbb{N}}$, has the same cardinality as the set of the real numbers⁷. Interesting questions have been raised about the size of this set⁸.

For our purposes, exponentiation is useful to demonstrate the existence of non-computable functions, as has been achieved in Section 4.6. Exponentiation also links to the topic of primitive recursive functions (a sub class of the Turing Computable functions, where each function is guaranteed to halt) and the existence of non-primitive recursive functions via the (halting) Ackerman function⁹. The topic of primitive recursive functions and its elegant hierarchy is a beautiful one, but regards a topic we won’t address in this course. Note however the possibility to complete a final year project on this topic.

⁷A proof of this fact falls outside the scope of the course.

⁸In relation to the so-called continuum hypothesis.

⁹A halting Turing Computable recursively defined function of enormously fast growth.

Chapter 9

Complexity analysis techniques

People who analyze algorithms have double happiness. First of all they experience the sheer beauty of elegant mathematical patterns that surround elegant computational procedures. Then they receive a practical payoff when their theories make it possible to get other jobs done more quickly and more economically.

Don Knuth

The analysis of algorithms exemplifies a “marriage between theory and practice”.



9.1 Random sampling

In practice, the following approach can be taken to deal with the combinatorial explosion in the number of input cases. Regarding our prior example of card shuffling, consider a random sample of lists of size 52. Say you compute 100,000 of these. Compute the running time for all of these inputs. Compute the worst-case time and the average case time. This gives an estimate of the real worst-case and average-case running time. To succeed, the random sample needs to be representative of the entire collection of $52!$ lists. In view of the scale of that collection, you can see there cases can occur for which random sampling results could be off. Still, random sampling, with judicious use of statistical methods, can provide useful information in practice.

Different techniques to brute force and random sampling can be applied. We discuss some of these “exact-time” techniques below, which will be based on a static analysis of the code.

9.2 Comparison time

To simplify matters for the analysis we will focus on counting *some* steps carried out by an algorithm, rather than all of them. One of the main steps which a sorting algorithm carries out is that of *comparing* two elements. Many algorithms are entirely *comparison-based*—each basic operation (swap or assignment) is typically the outcome of a prior comparison between two elements.

This is not surprising since sorting algorithms are designed to order a list, so in particular the computation needs to determine the order of between the elements by comparing them. When the elements are out of order, typically, a swap is performed. The outcome of the comparison for instance the determination of the largest element, or the smallest) may be stored in a variable, leading to an assignment.

In the analysis, we will assume that each comparison takes a constant time c . We recall the notion of comparison time.



Comparison time

The comparison time $T_A(L)$ which an algorithm A takes on an input L is defined to be the total number of comparisons A makes in computing the output for L .

To extend this approach to counting other basic steps (swaps and assignments) we can consider an upper bound on the number of swaps and assignments following each comparison in a given algorithm. Running time estimations can be updated by taking into account these upper bounds. For instance, if there are at most 4 swaps and 3 assignments following each comparison, then an algorithm taking 10 comparisons on list L , will be assumed to take at most time $T_A(L) + 4s + 3a$ on the list L , where s is the time per swap and a is the time per assignment. Moreover, if all basic assignments are assumed to take the same upper bound time denoted by say u , we can bind the time which the algorithm takes on a list L by $8u$ in this case.

9.3 Constant time algorithms

By constant time algorithms, we mean algorithms A that take a constant time C_n on all inputs of size n . For such algorithms, the best-case time, worst-case time and average-case time coincide with the constant involved.



Constant time algorithms

If the algorithm A takes a constant time C_n on all inputs of size n then

$$\overline{T}_A(n) = T_A^W(n) = T_A^B(n) = C_n.$$

We leave the straightforward verifications as an exercise.

We discuss two constant time algorithms below, Bubble Sort and Selection Sort, for which we carry out a comparison-time analysis.

9.3.1 Analysis of Bubble Sort

Bubble Sort(L)

```

for  $i = |L| - 1$  downto 1 do           (loop 1)
  for  $j = 1$  to  $i$  do                   (loop 2)
    if  $L[j] > L[j + 1]$  then
      swap ( $L[j], L[j + 1]$ )

```

Comparison-time analysis:

For each run of loop 2, i comparisons are made; hence loop 2 depends on i .

Loop 1 (which has loop 2 as subloop) lets i range from $i = |L| - 1$ to 1.

So the total number of comparisons for a given list L is $T_B(L) = \sum_{i=n-1}^1 i$.

Remark: in this case a nice formula *is* found for the running time of each given list.

To obtain the value of $\sum_{i=n-1}^1 i$, we apply the following triangle equality:

$$\sum_{i=n-1}^1 i = 1 + 2 + 3 + \dots + n - 1 = \frac{((n-1) + 1)(n-1)}{2} = \frac{n(n-1)}{2}.$$

(Check out Section 17.0.1 if you don't know or recall this equality and its proof.)

Hence we obtain that for each L of length n , $T_B(L) = \frac{n(n-1)}{2}$.

Bubble Sort is an example of a “constant-time” algorithm. Its exact time is expressed as follows: for each L of length n , $T_B(L) = \frac{n(n-1)}{2}$.

Hence, it is clear that for every list of size n , the Bubble Sort algorithm takes the constant time $\frac{n(n-1)}{2}$. In other words, no matter what list of size n you provide as input, Bubble Sort will take exactly the same number of comparisons to compute the output on each such input.

This simplifies the analysis considerably since as mentioned above, **any algorithm A that has constant time, say C_n on all inputs of size n , must satisfy:**

$$\overline{T}_A(n) = C_n \text{ and } T_A^W(n) = C_n.$$

So, we obtained the worst-case and the average-case time of Bubble Sort.

Bubble Sort

$$T_B^W(n) = \overline{T}_B(n) = \frac{n(n-1)}{2}$$

A moment of reflection

The analysis of Bubble Sort is quite straightforward. We reflect on the gains made by introducing the simple analysis technique for constant time algorithms. Determining the worst-case time or the average-case time of Bubble Sort on lists of size 52 by brute force computation, requires one to compute the comparison-times of the algorithm on each of the $52!$ lists—an impossibility. The above argument however allows us to conclude that:

$$T_B^W(52) = \overline{T}_B(52) = \frac{52(52-1)}{2} = 1326 \text{ comparisons}$$

Hence *without* determining the $52!$ input lists first, we obtained the worst-case comparison time and the average-case comparison time of Bubble Sort in relation to the huge (and unmanageable) collection of $52!$ inputs. Reread the notes at this stage to understand how we arrived at this result. Note that constant time algorithms are rare of course¹, which is why we will introduce another technique in Section 9.4.1.

9.3.2 Analysis of Selection Sort

Consider the following sorting algorithm “Selection Sort”, where we picked the version which selects the least element (it is easy to produce a similar version which selects the greatest element).

The underlying intuition is the following:

If Selection Sort is given as input a list $L = (L[1], \dots, L[n])$ then, via a for-loop, the algorithm considers first the entire list $(L[1], \dots, L[n])$ and selects the smallest element from this list. This selected element is then swapped (if necessary) with $L[1]$ (to put it in the right place).

Next the algorithm considers the list $(L[2], \dots, L[n])$. It again selects the smallest element from this list and (if necessary) swaps it with $L[2]$.

Next it considers the list $(L[3], \dots, L[n])$, selects again the smallest element and (if necessary) swaps it with $L[3]$ etc.

¹Rare but useful: in a Real-Time context, constant time algorithms have a running time fully determined by their worst-case time. Hence the usual problems of inefficacy arising due to a big discrepancy between the worst-case and average-case time do not arise for such algorithms.

Until the last list of 2 elements ($L[n-1], L[n]$) is considered. Again the least element is selected and (if necessary) is swapped with $L[n-1]$.

This clearly results in a sorted list.

Of course, in case the smallest element of one of these lists already sits in the correct position (that is the first position of the list) then no swap is carried out, as this is not necessary.

Selection Sort(L)

```

for  $i = 1$  to  $|L| - 1$  do
     $k \leftarrow i$ 
     $l \leftarrow L[i]$ 
    for  $j = i + 1$  to  $|L|$  do
        if  $L[j] < l$  then  $k \leftarrow j$ 
         $l \leftarrow L[j]$ 
    SWAP( $L[i], L[k]$ )

```

We determine the *worst-case* comparison-time for this algorithm and the *average-case* comparison-time for this algorithm.

Selection Sort is a constant time algorithm on lists of size n

We remark that the outer for-loop is run through $n - 1$ times. So we need to sum up the number of comparisons made in the inner for-loop, $n - 1$ times. The inner for-loop, each time it is carried out for i ranging from 1 to $n - 1$, will make exactly one comparison for each j where j ranges from $i + 1$ until n . Hence $n - i$ comparisons are made. So the comparison-time for any input-list L of length n is exactly:

$$T_S(L) = \sum_{i=1}^{n-1} (n - i).$$

Working this out gives:

$$T_S(L) = \sum_{i=1}^{n-1} (n - i) = (n-1) + (n-2) + \dots + (n - (n-1)) = (n-1) + (n-2) + \dots + 1 = 1 + \dots + (n-2) + (n-1) = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}.$$

So, exactly as for Bubble Sort, we obtain that the comparison-time of Selection-sort does not depend on the particular list under consideration, only on the *size* (= length) of the list. I.e. Selectionsort is a constant time algorithm (on lists of fixed size n).

Selection Sort

$$T_S^W(n) = \overline{T}_S(n) = \frac{n(n-1)}{2}$$

9.4 Non-constant time algorithms

For algorithms which do not have constant time on inputs of size n , such as Sequential Search, we will identify groups of inputs on which the algorithm does have constant time. This will simplify the average-case analysis of such algorithms quite a bit in general.

9.4.1 Grouping inputs by running time

Constant time groups

Say an algorithm A is not constant time and can take the following different comparison times on inputs of size n : c_1, \dots, c_k . Then we can identify the following input-groups:

G_1 = the inputs on which takes c_1 comparisons.

G_2 = the inputs on which A takes c_2 comparisons.

...

G_k = the inputs on which A takes c_k comparisons.

If we can determine these groups, then we can express the total comparison time taken by A on inputs of size n as follows:

Total time expressed in terms of groups

$$T_A^{total}(n) = |G_1| \times c_1 + \dots + |G_k| \times c_k,$$

where $|G_i|$ is the size of group G_i , i.e. the number of inputs on which A takes time c_i .

Since the average time is the total time divided by the number $i(n)$ of inputs of size n , we get:

Average time expressed in terms of groups

$$\bar{T}_A(n) = \frac{|G_1| \times c_1 + \dots + |G_k| \times c_k}{i(n)}$$

Now comes the interesting part.

Simplification for groups of equal size

If we are lucky (and in a good many cases this will happen), then all groups have equal size: i.e. $|G_1| = |G_2| = \dots = |G_k|$. Let's denote this fixed size by $|G|$.

In this case the above average-time expression simplifies to:

$$\bar{T}_A(n) = \frac{|G| \times (c_1 + \dots + c_k)}{i(n)}$$

We provide an example of an algorithm for which all groups have equal size in the following section.

9.4.2 Analysis of Sequential Search

The pseudo-code for Sequential Search (notation SS) is given below:

Sequential search

Sequential Search(L, target, n)

```
for  $i = 1$  to  $n$  do  
  if ( $target = L[i]$ )  
    return  $i$ 
```

For simplicity we assume for our analysis that the target occurs always in the input list L

Optimization

Note that we could optimize the code where if we have looked through the first $n - 1$ elements of a list of size n and did not find the target, we can immediately return position n as the position of the target without making the final comparison (with the element in position n). We do not do so however and will analyze the above (non-optimized) code.

Can we reduce matters to $n!$ cases as for sorting algorithms?

We have seen that by the faithfulness property for comparison-based algorithms, in

particular for comparison-based sorting algorithms, we can limit the analysis to $n!$ lists of size n where the list elements consist of pairwise distinct elements of $\{1, \dots, n\}$. This followed because comparison-based algorithms execute in exactly the same way on lists sharing the same relative order. Sequential search involves a fixed target. Does this not complicate matters? Clearly the number of comparisons made by sequential search is entirely determined by the position of the target (not by the relative order between the elements). If the target resides in position i then i comparisons must be made to discover the target.

Hence to determine the worst-case time or the average-case time for all inputs of size n containing the target (and consisting of pairwise distinct elements) it suffices to consider the lists of size n consisting of the $n!$ permutations of the values $\{1, \dots, n\}$ where we arbitrarily pick one of these values as the target. The *value* of the target is irrelevant (as it does not influence the number of comparisons made) and so is the relative order between the list elements (unlike for the case of sorting). The only factor determining the number of comparisons during an execution is the position that the target resides in. Hence the worst-case and the average-case time of the algorithm when analyzed over this restricted case is identical to the worst-case and average-case time for the case of arbitrary inputs with pairwise distinct elements and containing the target.

Worst-case comparison time of Sequential Search

$$T_{SS}^W(n) = n$$

This is clear, since the for-loop proceeds in left-to-right fashion, starting at the first element of the list and comparing each element of the list with the target, until the end of the list is reached. When the target sits in the last position of a list of size n , the largest number of comparisons needs to be made in order to detect the target, namely n comparisons.

Similarly, the best-case comparison time of Sequential Search is 1, for the case where the target sits in first position of the list. Clearly, the average-case will be somewhere in the middle of the two extremes, i.e. halfway between 1 and n , which would lead to an estimate of $\frac{n}{2}$. This is pretty close to the correct answer.

Average-case comparison time of Sequential Search

$$\bar{T}_{SS}(n) = \frac{n+1}{2}$$

Sequential Search is not a constant time algorithm. For two lists of size n it is possible that Sequential Search takes a different number of comparisons in order to locate the target.

As observed, in case the target sits in first position, only one comparison will be made. In case the target sits in last position, then the algorithm will need to make n comparisons with to detect the target.

For Bubble Sort, we could easily find worst-case and average-case time since the algorithm has constant time on lists of size n .

For algorithms which do not have constant time on lists of size n , such as Sequential Search, we will identify groups of inputs on which the algorithm does take a constant number of comparisons.

We already found one such group when we carried out the worst-case analysis, namely the lists for which the target sits in the last position, i.e. in position n . All such lists result in a computation which takes n comparisons.

Generalizing this approach, we define the following groups for the case of inputs of size n :

G_1 , consisting of the lists for which the target sits in position 1

G_2 , consisting of the lists for which the target sits in position 2

...

G_n , consisting of the lists for which the target sits in position n

Sequential Search takes constant comparison time on each input from the group G_i , namely i comparisons.

Hence, applying the formula obtained above for the average-case time of non-constant time algorithms, we obtain the following equality:

$$\overline{T}_{SS}(n) = \frac{|G_1| \times 1 + \dots + |G_n| \times n}{n!}$$

To proceed, we need to determine the size of each group. Since group G_i consists of all lists in which we keep one element fixed, namely the target which is fixed in position i , the only elements that can be freely assigned are the $n - 1$ remaining elements. This can be done in exactly $(n - 1)!$ ways, so

$$|G_i| = (n - 1)!$$

I.e. all our groups have the same size. Hence:

$$\begin{aligned}
\overline{T}_{SS}(n) &= \frac{(n-1)! \times 1 + \dots + (n-1)! \times n}{n!} \\
&= \frac{(n-1)!(1+2+\dots+n)}{n!} \\
&= \frac{\sum_{i=1}^n i}{n} \\
&= \frac{n(n+1)}{2n} \\
&= \frac{n+1}{2}
\end{aligned}$$

9.5 Algorithms involving while loops

Thus far we considered the analysis of algorithms that did not involve while loops. As observed, while loops are more tricky to analyze than for loops. The main reason is that a while loop's termination depends on a conditional, the truth or falsity of which may be hard to analyze. While loops also are at the root of non-termination. Nevertheless it is possible in some cases to carry out the comparison-time analysis for algorithms involving while loops. We illustrate this on the basic example of the Insertion Sort algorithm.

9.5.1 Analysis of Insertion Sort

We recall Insertion Sort's pseudo-code:

```
Insertion Sort(L)  
  for  $i = 2$  to  $|L|$  do  
     $j \leftarrow i$   
    while  $L[j - 1] > L[j]$  do  
      swap ( $L[j - 1], L[j]$ )  
       $j \leftarrow j - 1$ 
```

Intuition: Insertion Sort systematically inserts an element into a priorly sorted list. Initially the sorted list is a list of length one consisting of the first element of the input-list. This creates a sorted list of length two, after which the insertion process is repeated on the third list element, etc.

We introduce some additional notation to facilitate representing the Insertion Sort's execution

1) If L is a list then L^S is the sorted version of L

$$L = (3, 1, 5, 4, 2)$$

$$L^S = (1, 2, 3, 4, 5)$$

2) If L is a list, say $L = (L[1], \dots, L[n])$ then for any i ranging from 1 to n :

$$L_i = (L[1], \dots, L[i])$$

$$L = (3, 1, 5, 4, 2)$$

$$L_1 = (3)$$

$$L_2 = (3, 1)$$

$$L_3 = (3, 1, 5)$$

$$L_4 = (3, 1, 5, 4)$$

$$L_5 = (3, 1, 5, 4, 2)$$

3) Given a list L , say $L = (L[1], \dots, L[n])$, then for j ranging from 2 to n :

$$\tilde{L}_j = \text{Append}(L_{j-1}^S, L[j])$$

where the “append” operation concatenates the list L_{j-1}^S and the element $L[j]$.

$$L = (3, 1, 5, 4, 2)$$

$$\tilde{L}_2 = \text{append}(L_1^S, L[2]) = \text{append}((3), 1) = (3, 1)$$

$$\tilde{L}_3 = \text{append}(L_2^S, L[3]) = \text{append}((1, 3), 5) = (1, 3, 5)$$

$$\tilde{L}_4 = \text{append}(L_3^S, L[4]) = \text{append}((1, 3, 5), 4) = (1, 3, 5, 4)$$

$$\tilde{L}_5 = \text{append}(L_4^S, L[5]) = \text{append}((1, 3, 4, 5), 2) = (1, 3, 4, 5, 2)$$

4) We introduce one more notation to indicate the while-loop part of Insertion’s Sort pseudo-code.

$$J_j = \left[\begin{array}{l} \text{while } L[j-1] > L[j] \text{ do} \\ \quad \text{swap } (L[j-1], L[j]) \\ \quad j \leftarrow j-1 \end{array} \right.$$

Insertion Sort’s pseudo-code can now be re-written as:

$$I = \left[\begin{array}{l} \text{for } i = 2 \text{ to } |L| \text{ do} \\ \quad j \leftarrow i \\ \quad J_j \end{array} \right.$$

Where is all this notation leading? Let’s have a look at how insertion sort executes on the input list $(3, 1, 5, 4, 2)$. We display the results obtained after executing the outer for-loop:

$$L = (\underline{3, 1}, 5, 4, 2) (\tilde{L}_2 \text{underlined})$$

$$i = 2 \rightarrow (\underline{1, 3}, 5, 4, 2) (\tilde{L}_3 \text{underlined})$$

$$i = 3 \rightarrow (\underline{1, 3, 5}, 4, 2) (\tilde{L}_4 \text{underlined})$$

$$i = 4 \rightarrow (\underline{1, 3, 4, 5}, 2) (\tilde{L}_5 \text{underlined})$$

$$i = 5 \rightarrow (\underline{1, 2, 3, 4, 5}) (\text{sorted list})$$

So... now we know:

$$T_I(L) = T_{J_2}(\tilde{L}_2) + T_{J_3}(\tilde{L}_3) + \dots + T_{J_n}(\tilde{L}_n)$$

for lists of length $n \geq 2$.

So, do we now have a nice formula for the comparison time T_I , which I takes when run on an input L ?

Not really... ideally we'd like

$$T_I(L) = \text{some formula involving } L,$$

or, better,

$$T_I(L) = \text{some number of comparisons.}$$

So what's the problem?

We don't know what for instance $T_{J_3}(\tilde{L}_3)$ is. That depends on the form of L , and on when the while loop exits. Let's take two examples to illustrate this fact:

$$L = (3, 1, 5, 4, 2) \rightarrow \tilde{L}_3 = (1, 3, 5) \rightarrow T_{J_3}(\tilde{L}_3) = 1$$

$$L' = (3, 1, 2, 4, 5) \rightarrow \tilde{L}'_3 = (1, 3, 2) \rightarrow T_{J_3}(\tilde{L}'_3) = 2 \text{ (why?)}$$

So the formula we obtained is dependent on the actual list we consider and it not clear how to write out the values such as $T_{J_3}(\tilde{L}_3)$ in a formula which enables us to compute the outcome. We knew already that determining $T_I(L)$ for all lists L is asking a lot as you will remember from the trade-off discussion in Section 8.6. Let's make life easy and focus on worst-case comparison time - the previous formula will help us here:

9.5.2 Worst-case time of Insertion Sort

Recall that

$$T_I(L) = T_{J_2}(\tilde{L}_2) + T_{J_3}(\tilde{L}_3) + \dots + T_{J_n}(\tilde{L}_n).$$

To find the worst-case time, we focus on lists of length $n \geq 2$.

$$T_I^W(n) = ?$$

Clearly the worst case is arrived at when each of the comparison times

$$T_{J_2}(\tilde{L}_2), T_{J_3}(\tilde{L}_3), \dots, T_{J_n}(\tilde{L}_n)$$

is as large as possible.

This happens when each of the while loops J_i only exits at the last possible moment...

...which only happens when each \tilde{L}_j is such that its last element is smaller than all other elements to its left - which is when L is in reverse order.

For instance, for $|L| = 5$, the list in reversed order is $L = (5, 4, 3, 2, 1)$. Here,

$$\begin{aligned}\tilde{L}_2 &= (5, 4) \\ \tilde{L}_3 &= (4, 5, 3) \\ \tilde{L}_4 &= (3, 4, 5, 2) \\ \tilde{L}_5 &= (2, 3, 4, 5, 1)\end{aligned}$$

$$\text{and } T_I(L) = T_{J_2}(\tilde{L}_2) + T_{J_3}(\tilde{L}_3) + T_{J_4}(\tilde{L}_4) + T_{J_5}(\tilde{L}_5) = 2 + 3 + 4 + 5 = 14.$$

In general, for a reversed list L of length n , $L = (n, n-1, n-2, \dots, 3, 2, 1)$, we have:

$$T_I(L) = T_{J_2}(\tilde{L}_2) + T_{J_3}(\tilde{L}_3) + \dots + T_{J_n}(\tilde{L}_n) = 2 + 3 + \dots + n = \left(\sum_{i=1}^n i\right) - 1 = \frac{n(n+1)}{2} - 1$$

Hence we obtain the following result on Insertion Sort's worst-case time.

Worst-case time of Insertion Sort

$$T_I^W(n) = \frac{n(n+1)}{2} - 1.$$

Exercise 83 (*Algorithm analysis*)

- a) Determine Insertion Sort's best-case time.*
- b) Consider the standard algorithm to merge two sorted lists of equal size.*
 - b-1) What is the worst-case time of the standard Merge algorithm?*
 - b-2) What is the best-case time for the standard Merge algorithm?*

9.5.3 Average-case time of Insertion Sort

Rather than presenting a general argument, we focus on an example and generalise our findings.

We focus on lists of length 4, so there are $4! = 4 \cdot 3 \cdot 2 \cdot 1 = 24$ cases, listed below.

We indicate computation steps corresponding to J_2 and J_3 and J_4 (the last of which of course has as result always the sorted list).

The symbol $|$ marks how much of the list is known to be sorted, as the sorting process proceeds from left to right. We will divide lists \tilde{L}_i ($i : 2, 3, 4$) into groups according to the number of comparisons which are needed to insert $L[i]$ in the list L_{i-1}^S ; then we count the elements in each group, and generalise this to lists of length n .

	\nearrow	J_2	\searrow	\nearrow	J_3	\searrow	\nearrow	J_4	\searrow	
<i>I</i>	1 234		<i>I</i>	12 34		<i>I</i>	123 4	—		1234
<i>I</i>	1 243		<i>I</i>	12 43		<i>II</i>	124 3	—		1234
<i>I</i>	1 324		<i>II</i>	13 24		<i>I</i>	123 4	—		1234
<i>I</i>	1 342		<i>I</i>	13 42		<i>III</i>	134 2	—		1234
<i>I</i>	1 423		<i>II</i>	14 23		<i>III</i>	124 3	—		1234
<i>I</i>	1 432		<i>II</i>	14 32		<i>II</i>	134 2	—		1234
<i>II</i>	2 134		<i>I</i>	12 34		<i>I</i>	123 4	—		1234
<i>II</i>	2 143		<i>I</i>	12 43		<i>II</i>	124 3	—		1234
<i>I</i>	2 314		<i>III</i>	23 14		<i>I</i>	123 4	—		1234
<i>I</i>	2 341		<i>I</i>	23 41		<i>IV</i>	234 1	—		1234
<i>I</i>	2 413		<i>III</i>	24 13		<i>II</i>	124 3	—		1234
<i>I</i>	2 431		<i>II</i>	24 31		<i>IV</i>	234 1	—		1234
<i>II</i>	3 124		<i>II</i>	13 24		<i>I</i>	123 4	—		1234
<i>II</i>	3 142		<i>I</i>	13 42		<i>III</i>	134 2	—		1234
<i>II</i>	3 214		<i>III</i>	23 14		<i>I</i>	123 4	—		1234
<i>II</i>	3 241		<i>I</i>	23 41		<i>IV</i>	234 1	—		1234
<i>I</i>	3 412		<i>III</i>	34 12		<i>III</i>	134 2	—		1234
<i>I</i>	3 421		<i>III</i>	34 21		<i>IV</i>	234 1	—		1234
<i>II</i>	4 123		<i>II</i>	14 23		<i>II</i>	124 3	—		1234
<i>II</i>	4 132		<i>II</i>	14 32		<i>III</i>	134 2	—		1234
<i>II</i>	4 213		<i>III</i>	24 13		<i>II</i>	124 3	—		1234
<i>II</i>	4 231		<i>II</i>	24 31		<i>IV</i>	234 1	—		1234
<i>II</i>	4 312		<i>III</i>	34 12		<i>III</i>	134 2	—		1234
<i>II</i>	4 321		<i>III</i>	34 21		<i>IV</i>	234 1	—		1234

Summary:

At start of J_2 :

12 lists of group *I*

12 lists of group *II*

At start of J_3 :

8 lists of group *I*

8 lists of group *II*

8 lists of group *III*

At start of J_4 :

6 lists of group *I*

6 lists of group *II*
6 lists of group *III*
6 lists of group *IV*

Generalisation:

For lists of length n and at the start of J_j (where $2 \leq j \leq n$), we have j groups, each of $\frac{n!}{j}$ lists.

For each of these j groups, we have that for group i (where $1 \leq i \leq j$) J_j uses exactly i comparisons (recall the buffer $-\infty$ which requires an extra comparison). So J_j will use in total $\frac{n!}{j}(1 + 2 + \dots + j)$ comparisons, where $\frac{n!}{j}$ is the number of lists per group, and $1, 2, \dots, j$ are the number of comparisons per list of group j .

For each J_j , exactly $\frac{n!}{j}(\sum_{i=1}^j i)$ comparisons are made. We know that $(\sum_{i=1}^j i) = \frac{j(j+1)}{2}$, so each J_j uses exactly $\frac{n!}{j}(\sum_{i=1}^j i) = \frac{n!}{j} \cdot \frac{j(j+1)}{2} = \frac{n!}{2}(j+1)$ comparisons.

Let's check this for the case where $n = 4$ and for J_3 .

J_3 uses 1×8 comparisons in group *I*, 2×8 comparisons in group *II*, 3×8 comparisons in group *III* - i.e. $8 + 16 + 24 = 48$ comparisons used in total by J_3 .

Our formula for J_3 and $n = 4$ gives $\frac{4!}{2}(3+1) = \frac{4!}{2}(3+1) = \frac{24}{2}(4) = 48$, the same result.

We can now conclude as follows:

$\bar{T}_I(n)$ is the total number of comparisons made by I on all lists of length n , divided by $n!$

So,

$$\begin{aligned}
\bar{T}_I(n) &= \sum_{j=2}^n \frac{\frac{n!}{2}(j+1)}{n!} \\
&= \frac{1}{2} \sum_{j=2}^n (j+1) \\
&= \frac{1}{2}(3 + 4 + 5 + \dots + (n+1)) \\
&= \frac{1}{2}(1 + 2 + 3 + \dots + (n+1) - 1 - 2) \\
&= \frac{1}{2} \left(\frac{(n+1)(n+2)}{2} - 1 - 2 \right).
\end{aligned}$$

So

$$\begin{aligned}
 \overline{T}_I(n) &= \frac{1}{2} \left(\frac{(n+1)(n+2)}{2} - 3 \right) \\
 &= \frac{1}{2} \left(\frac{(n+1)(n+2) - 6}{2} \right) \\
 &= \frac{1}{2} \left(\frac{n^2 + 3n + 2 - 6}{2} \right) \\
 &= \frac{1}{4} (n^2 + 3n - 4).
 \end{aligned}$$



Insertion Sort's average-case time

$$\overline{T}_I(n) = \frac{n^2 + 3n - 4}{4}$$

Let's check this: $n = 4$,

$$\begin{aligned}
 \frac{\text{total number of comparisons}}{4!} &= \frac{12 \times (1+2) + 8 \times (1+2+3) + 6 \times (1+2+3+4)}{24} \\
 &= \frac{36 + 48 + 60}{24} \\
 &= \frac{144}{24} \\
 &= 6.
 \end{aligned}$$

The formula gives

$$\frac{1}{4} (4^2 + 3 \cdot 4 - 4) = \frac{24}{4} = 6$$

(the same).

9.5.4 Average-case of Insertion Sort (alternative method)

We now proceed to give an alternative formal proof of the fact that

$$\overline{T}_I(n) = \frac{1}{4} (n^2 + 3n - 4).$$

For this we analyse the case of $n = 4$ once more, but this time round we focus on the average time of each of the insertion loops $J_j (2 \leq j \leq n)$.

$$\overline{T}_{J_2} = ? ; \overline{T}_{J_3} = ? ; \dots ; \overline{T}_{J_j} = ?$$

We determine \overline{T}_{J_4} as an example:

$J_4()$	\overline{T}_{J_4}	$J_4()$	\overline{T}_{J_4}
(*) 123 4	1	(*) 123 4	1
(A) 124 3	$1 + T_{J_3}(12 3)$	(B) 134 2	$1 + T_{J_3}(13 2)$
(*) 123 4	1	(*) 123 4	1
(B) 134 2	$1 + T_{J_3}(13 2)$	(C) 234 1	$1 + T_{J_3}(23 1)$
(B) 124 3	$1 + T_{J_3}(13 2)$	(B) 134 2	$1 + T_{J_3}(13 2)$
(A) 134 2	$1 + T_{J_3}(12 3)$	(C) 234 1	$1 + T_{J_3}(23 1)$
(*) 123 4	1	(A) 124 3	$1 + T_{J_3}(12 3)$
(A) 124 3	$1 + T_{J_3}(12 3)$	(B) 134 2	$1 + T_{J_3}(13 2)$
(*) 123 4	1	(A) 124 3	$1 + T_{J_3}(12 3)$
(C) 234 1	$1 + T_{J_3}(23 1)$	(C) 234 1	$1 + T_{J_3}(23 1)$
(A) 124 3	$1 + T_{J_3}(12 3)$	(B) 134 2	$1 + T_{J_3}(13 2)$
(C) 234 1	$1 + T_{J_3}(23 1)$	(C) 234 1	$1 + T_{J_3}(23 1)$

Summary:

- 6 cases of (*) = case of list ending with 4 (largest number)
- 6 cases of (A) = case of list of type 12|3
- 6 cases of (B) = case of list of type 13|2
- 6 cases of (C) = case of list of type 23|1

So $T_{J_4} = 6 + 6(T_{J_3} + 3)$ (T_{J_3} being the total time spent by J_3 on 12|3, 13|2 and 21|3).

Hence

$$\begin{aligned}
 \overline{T}_{J_4} &= \frac{1}{24}(6 + 6(T_{J_3} + 3)) \\
 &= \frac{1}{4} + \frac{6}{24}(T_{J_3} + 3) \\
 &= \frac{1}{4} + \frac{6}{8}\left(\frac{T_{J_3}}{3} + 1\right)
 \end{aligned}$$

and

$$\overline{T}_{J_4} = \frac{1}{4} + \frac{3}{4}(\overline{T}_{J_3} + 1).$$

This is clear from our previous summary; $\frac{1}{4}$ of the lists (marked with (*)) take 1 comparison; $\frac{3}{4}$ of the lists (marked with (A), (B), (C)) take $1 + \overline{T}_{J_3}$ on average.

In general we obtain that $\overline{T}_{J_j} = \frac{1}{j} \cdot 1 + \frac{j-1}{j}(\overline{T}_{J_{j-1}} + 1)$ (formula *) - verify this for \overline{T}_{J_2} .

Note: we allow $j = 1$ which corresponds to the average number of comparisons made when comparing the bumper ∞ with the unique list of length 1, so $\overline{T}_{J_1} = 1$ (check that the formula * yields this for $j = 1$).

We can now determine the exact value of \overline{T}_{J_j} in an easy way:

To calculate \overline{T}_{J_j} we simply expand this as follows:

Note: $\overline{T}_{J_j} = \frac{1}{j} + \frac{j-1}{j}\overline{T}_{J_{j-1}} + \frac{j-1}{j}$ (sum of first and last terms is 1).

$$\overline{T}_{J_j} = 1 + \frac{j-1}{j}\overline{T}_{J_{j-1}} \text{ (formula **).}$$

Hence (expanding **) we get:

$$\begin{aligned} \overline{T}_{J_j} &= 1 + \frac{j-1}{j}\overline{T}_{J_{j-1}} \\ &= 1 + \frac{j-1}{j}\left(1 + \frac{j-2}{j-1}\overline{T}_{J_{j-2}}\right) \\ &= 1 + \frac{j-1}{j}\left(1 + \frac{j-2}{j-1}\left(1 + \frac{j-3}{j-2}\overline{T}_{J_{j-3}}\right)\right) \\ &= 1 + \frac{j-1}{j} + \frac{j-2}{j} + \frac{j-3}{j}\overline{T}_{J_{j-3}} \end{aligned}$$

Generalising this line of reasoning, we obtain:

$$\begin{aligned} \overline{T}_{J_j} &= 1 + \frac{j-1}{j} + \frac{j-2}{j} + \frac{j-3}{j} + \dots + \frac{j-(j-2)}{j} + \frac{j-(j-1)}{j}\overline{T}_{J_1} \\ &= 1 + \frac{1+2+\dots+(j-3)+(j-2)+(j-1)}{j} \\ &= 1 + \frac{\frac{(j-1) \cdot j}{2}}{j} = 1 + \frac{(j-1)}{2} \end{aligned}$$

So,

$$\overline{T}_{J_j} = \left(1 + \frac{j-1}{2}\right), \forall j \geq 1.$$

We will use this formula to determine $\overline{T}_I(n)$.

We show that: $\overline{T}_I(n) = \overline{T}_{J_2} + \overline{T}_{J_3} + \dots + \overline{T}_{J_n}$ (formula \otimes)

Note that:

$$\bar{T}_I(n) = \frac{1}{n!} \cdot (\text{total number of comparisons made by } I \text{ on lists of size } n)$$

$$= \frac{1}{n!} \left[(\bar{T}_{J_2} \binom{2}{1}) \cdot \frac{n!}{\binom{2}{1}} + (\bar{T}_{J_3} \binom{3}{2}) \cdot \frac{n!}{\binom{3}{2}} + \dots + (\bar{T}_{J_n} \binom{n}{n-1}) \cdot \frac{n!}{\binom{n}{n-1}} \right]$$

where $(\bar{T}_{J_n} \binom{n}{n-1})$ is the total number of comparisons J_n makes per group, and $\frac{n!}{\binom{n}{n-1}}$ is the number of groups involved.

Cancelling out all the terms in the above equality we get:

$$\bar{T}_J(n) = \bar{T}_{J_2} + \dots + \bar{T}_{J_n}$$

We can check this on an example:

We show for $n = 3$; $\bar{T}_I(n) = \bar{T}_{J_2} + \bar{T}_{J_3}$,

	J_2		J_3	
1 23	$^1 \rightarrow$	12 3	$^1 \rightarrow$	123
2 13	$^2 \rightarrow$	12 3	$^1 \rightarrow$	123
1 32	$^1 \rightarrow$	13 2	$^2 \rightarrow$	123
3 12	$^2 \rightarrow$	13 2	$^2 \rightarrow$	123
2 31	$^1 \rightarrow$	23 1	$^3 \rightarrow$	123
3 21	$^2 \rightarrow$	23 1	$^3 \rightarrow$	123

So

$$\begin{aligned} \bar{T}_I(3) &= \frac{(1+2) + (1+2) + (1+2) + (1+2+3) + (1+2+3)}{3!} \\ &= \frac{3.T_{J_2}^*(3) + 2.T_{J_3}^*(3)}{6} \\ &= \frac{1}{6} \left([\bar{T}_{J_2} \cdot \binom{2}{1}] \cdot \frac{3!}{\binom{2}{1}} + [\bar{T}_{J_3} \cdot \binom{3}{2}] \cdot \frac{3!}{\binom{3}{2}} \right). \end{aligned}$$

(where $T_{J_2}^*(3)$ = total number of comparisons made by J_2 on input lists of length 2.)

Finally we combine formula ** with formula \otimes , and obtain:

$$\begin{aligned}
\overline{T}_I(n) &= \overline{T}_{J_2} + \dots + \overline{T}_{J_n} \\
&= \left(1 + \frac{1}{2}\right) + \dots + \left(1 + \frac{n-1}{2}\right) \\
&= (n-1) \cdot 1 + \frac{1}{2}(1 + \dots + (n-1)) \\
&= (n-1) + \frac{1}{2} \cdot \frac{(n-1)n}{2} \\
&= \frac{4(n-1) + (n-1)n}{4} \\
&= \frac{4n - 4 + n^2 - n}{4}
\end{aligned}$$

So,

$$\boxed{\overline{T}_I(n) = \frac{n^2+3n-4}{4}}$$

which is exactly what we obtained before.

Chapter 10

Modularity

10.1 Timing modularity: the real-time engineer's paradise

Modularity is a highly desirable property in computer science. We quote from Tom Maibaum's article, "*Mathematical Foundations of Software Engineering: a roadmap.*"

"The only effective method for dealing with the complexity of software based systems is decomposition. Modularity is a property of systems, which reflects the extent to which it is decomposable into parts, from the properties of which we are able to predict the properties of the whole. Languages that do not have sufficiently strong modularity properties are doomed to failure, in so far as predictable design is concerned."

To understand the crucial nature of the concept of modularity in Computer Science, it is useful to compare Software Engineering to "traditional" engineering disciplines, say civil engineering. In civil engineering applications, we can analyze the properties of the "modular" parts of a building, say the building's bricks and the mortar that holds the bricks together, and use this information to derive a property about the building *before it is ever constructed*. Consider the example of a bridge. If we know the properties of the materials involved in building a bridge, we can predict the load that the bridge will be able to carry or the stresses it can sustain under high speed winds. This is because from the properties of the building materials, we can predict the property of the entire bridge once the materials have been put together. The method by which we can derive this information is based on computations relying on Newton's Calculus.

For the case of software engineering, we can consider, say, a program P to constitute of sequential components, say Q_1, \dots, Q_n , i.e.,

$$P = Q_1, \dots, Q_n.$$

Ideally, we would like to predict properties of P from the properties of its "modular building blocks", namely the sequential parts Q_1, \dots, Q_n .

Unfortunately, modularity is not a given in Computer Science. Computer Science's building blocks (programs) are far less well-behaved in general than traditional engineering's building blocks (say bricks). Indeed, the behaviour of a program intricately depends on which inputs it is provided with and this behaviour can vary wildly per input. A brick may suffer some variation under temperature differences but, in general, a brick's shape will be quite stable. A brick's properties remain pretty much the same. This is not the case for a piece of code, the properties/behaviour of which vary considerable with the inputs it is provided with—inputs that in turn may be produced as outputs of a different piece of code. Civil Engineers may very well wonder why Software Engineering lags in its capacity to predict the behaviour of code in industry-sized applications, as opposed to smaller scale academic applications. Traditional engineers do not, in general, have to deal with, say “go-to bricks” or “bricks” that have properties depending on the type of bricks that will surround it. Software engineers do need to deal with this situation.

In general, modularity does not hold in Computer Science, i.e., in general we cannot predict the behaviour of a system (including say a piece of software) from the behaviour of its parts. However, if modularity holds then the Software Engineer can carry out predictions similar to the Civil Engineer. This occurs for instance in safety-critical systems design, where software engineers can predict the worst-case time of code from the worst-case time of the sequential components thanks to a “semi-modularity” property which will be discussed in Section 10.4.

Aside from real-time programming, currently, many general purpose programming languages have some kind of “modularity” built in. Object-orientation or typing go some way to introducing a degree of modularity to ensure that larger programs can be built in a better way, i.e. with more chance that they will be correct and satisfy the requirements if their components satisfy the requirements.

The ideal situation for any programming language is to be able to predict the properties of software (correctness, running time, power-use, security, ...) from the properties of its parts; i.e. in a modular way.

Most results, e.g. for the case of predicting power-use (of software or hardware), are achieved by requiring that components never interact at all, e.g. for the case of parallel execution. This situation is similar to the classical Civil Engineering situation. Bricks do not influence each others behaviour. For the case of running-time analysis however, more can be said regarding modularity of components that *do* interact.

In this chapter we will focus entirely on modularity properties of timing measures, including exact time, worst-case time and average-case time.

10.2 Modularity of exact time

Assume that you have two programs, P_1 and P_2 , for which the sequential execution $P_1; P_2$ is carried out. Modularity in this context means that in order to determine information about the sequential composition $P_1; P_2$ we aim to determine information

about the program P_1 and information about the program P_2 and combine these two pieces of information to determine the information about $P_1; P_2$.

Let's be more concrete. Say we wish to know "how long" the program $P_1; P_2$ will run, where we assume that the execution of both components of the composition terminates. Clearly, for any input I we can write the following expression for the exact time.

Exact time modularity

$$(*) \quad T_{P_1; P_2}(I) = T_{P_1}(I) + T_{P_2}(P_1(I))$$

where the expression $P_1(I)$ indicates the output produced by the program P_1 on the input I .

Hence we have shown that modularity holds for the exact time on any given input, since in order to determine the exact time of the sequential composition $T_{P_1; P_2}(I)$, it suffices to determine $T_{P_1}(I)$ and $T_{P_2}(P_1(I))$ separately, after which we obtain the desired outcome $T_{P_1; P_2}(I)$, by adding the results.

This is a first example of the software engineer's paradise: in the case of modularity, we only need to focus on timing the *sequential components* of a program, say P , in order to determine the time of the entire program.

10.3 Reuse

Note that, in case the modularity property holds, we can always reuse the information on the time of the sequential components in case these same components would occur in another program, say Q . In that case we would only need to time the novel components of Q , namely those which were not timed earlier on (in the analysis of the components of P) and determine the time of Q in a modular fashion, by adding the relevant times.

Moreover, if someone were to replace a sequential component of P by a new component, resulting in a program P' , all we need to do is time the new component and use the previously obtained timings for the other components of P in order to re-evaluate the time and obtain the time of P' , by adding the relevant times.

Clearly, modularity of software implies useful properties, in particular the capacity to reuse/replace pieces of software while capitalising on the work done before.

Of course, our first example involving exact time is a trivial one and hardly demonstrates the use of modularity. The main timing measure used in so-called safety critical systems and real-time engineering applications is worst-case time. Indeed, worst-case time is particularly useful in the context of scheduling, where worst-case time of a

given process indicates the time we need to wait before the process is guaranteed to terminate—a prerequisite to safe scheduling of processes.

Real-Time programming languages, i.e. language designed to enable the timing of their programs, are generally obtained by “clipping the wings” of traditional languages, forcing all programs of a given language to terminate. Examples are Ada and Real-Time Java. Operations are restricted, e.g. by eliminating while loops (even bounded ones) in favour of for-loops, to facilitate worst-case execution time predictability. Worst-case time prediction for such languages relies heavily on a crucial property of the worst-case time measure, which we will discuss next.

10.4 Semi-modularity of worst-case time

Unfortunately, as the following exercise shows, worst-case time is not modular. Luckily we can still demonstrate that it worst-case time is “semi-modular.” Once we have established this fact, we will discuss the implications for Real-Time Languages, which make heavy use of the worst-case time and its semi-modular property.

10.4.1 Frog-Leap example

We will consider the “Conditional Frog-Leap” algorithm. This algorithm systematically compares two consecutive elements of a list L and, in case the first is smaller than the second, “frog-leaps” the first element over all the others to land it in the final list-position.

We recall two basic operations on lists, “head” and “tail”, which each return a list. If

$$L = (L[1], \dots, L[n])$$

is a list of length $n \geq 2$ then

$$Head(L) = (L[1])$$

and

$$Tail(L) = (L[2], \dots, L[n]).$$

The “Frog-Leaping” effect occurs via the operation “Frog-Leap,” which we denote by FL . Frog-Leap places the head of a list L of length n in the final position of the list.

Note also that we will use sentinel values for our lists, which simplifies our while-loop pseudo-code.

Frog-Leap (FL)

$L := Append(Tail(L), Head(L)).$

Conditional-Frog-Leap (CFL)

while $L[1] < L[2]$ *do* $FL(L)$

We also consider the following version of Conditional-Frog-Leap, which is obtained from CFL by inverting the comparison sign.

Conditional-Frog-Leap* (CFL*)

while $L[1] > L[2]$ *do* $FL(L)$

To illustrate that modularity breaks down for the worst-case time measure we first compute $T_{CFL;CFL^*}^W(3)$ and then compare this information to the worst-case times of the components, i.e. T_{CFL}^W and $T_{CFL^*}^W$.

We illustrate the execution of $CFL;CFL^*$ on inputs of size 3:

$123 \rightarrow_{CFL} 231 \rightarrow_{CFL} 312 \rightarrow_{CFL} 312 \rightarrow_{CFL^*} 123 \rightarrow_{CFL^*} 123$

$132 \rightarrow_{CFL} 321 \rightarrow_{CFL} 321 \rightarrow_{CFL^*} 213 \rightarrow_{CFL^*} 132 \rightarrow_{CFL^*} 132$

$213 \rightarrow_{CFL} 213 \rightarrow_{CFL^*} 132 \rightarrow_{CFL^*} 132$

$231 \rightarrow_{CFL} 312 \rightarrow_{CFL} 312 \rightarrow_{CFL^*} 123 \rightarrow_{CFL^*} 123$

$312 \rightarrow_{CFL} 312 \rightarrow_{CFL^*} 123 \rightarrow_{CFL^*} 123$

$321 \rightarrow_{CFL} 321 \rightarrow_{CFL^*} 213 \rightarrow_{CFL^*} 132 \rightarrow_{CFL^*} 132$

The respective comparison times for these inputs are: 5,5,3,4,3,4 and hence the worst-case time is:

$$T_{CFL;CFL^*}^W(3) = 5.$$

Our intuition may tell us that the worst-case time of the composition of two programs should be equal to the sum of the worst-case times of each component. After all, we showed that a similar summation property holds for the exact time (i.e. the computation time on a fixed input) as illustrated by equality (*) above.

However, as we will show, in general:

$$T_{P_1;P_2}^W(n) \neq T_{P_1}^W(3) + T_{P_2}^W(3).$$

Indeed, note (from the above diagram where we only focus on the computation steps which are determined by CFL): $T_{CFL}^W(3) = 3$.

It is easy to see (by making a similar diagram for CFL^* 's execution on all input lists of size 3 that $T_{CFL^*}^W(3) = 3$. Hence

$$T_{CFL;CFL^*}^W(3) = 5 < T_{CFL}^W(3) + T_{CFL^*}^W(3) = 6.$$

One could argue that the problem may lie in the fact that once CFL has terminated, CFL^* will continue the computation on the *outputs* of CFL . The collection of

outputs of CFL does not necessarily coincide with the collection of lists of size 3. As it happens, for our example above, the outputs form a subset $\{(3, 1, 2), (3, 2, 1), (2, 1, 3)\}$ of the original six lists. (In general the outputs could be completely different from the list data structure and e.g. be heaps instead of lists!).

So to make the analysis fair, we really should check whether the following holds:

$$T_{CFL;CFL^*}^W(3) = T_{CFL}^W(3) + T_{CFL^*}^W(Outputs(CFL, 3)),$$

where $Outputs(CFL, 3)$ are the outputs of program CFL on the inputs of size 3.

Note that $Outputs(CFL, 3) = \{(3, 1, 2), (3, 2, 1), (2, 1, 3)\}$, i.e. the collection of outputs produced by CFL when executed on all lists of size 3.

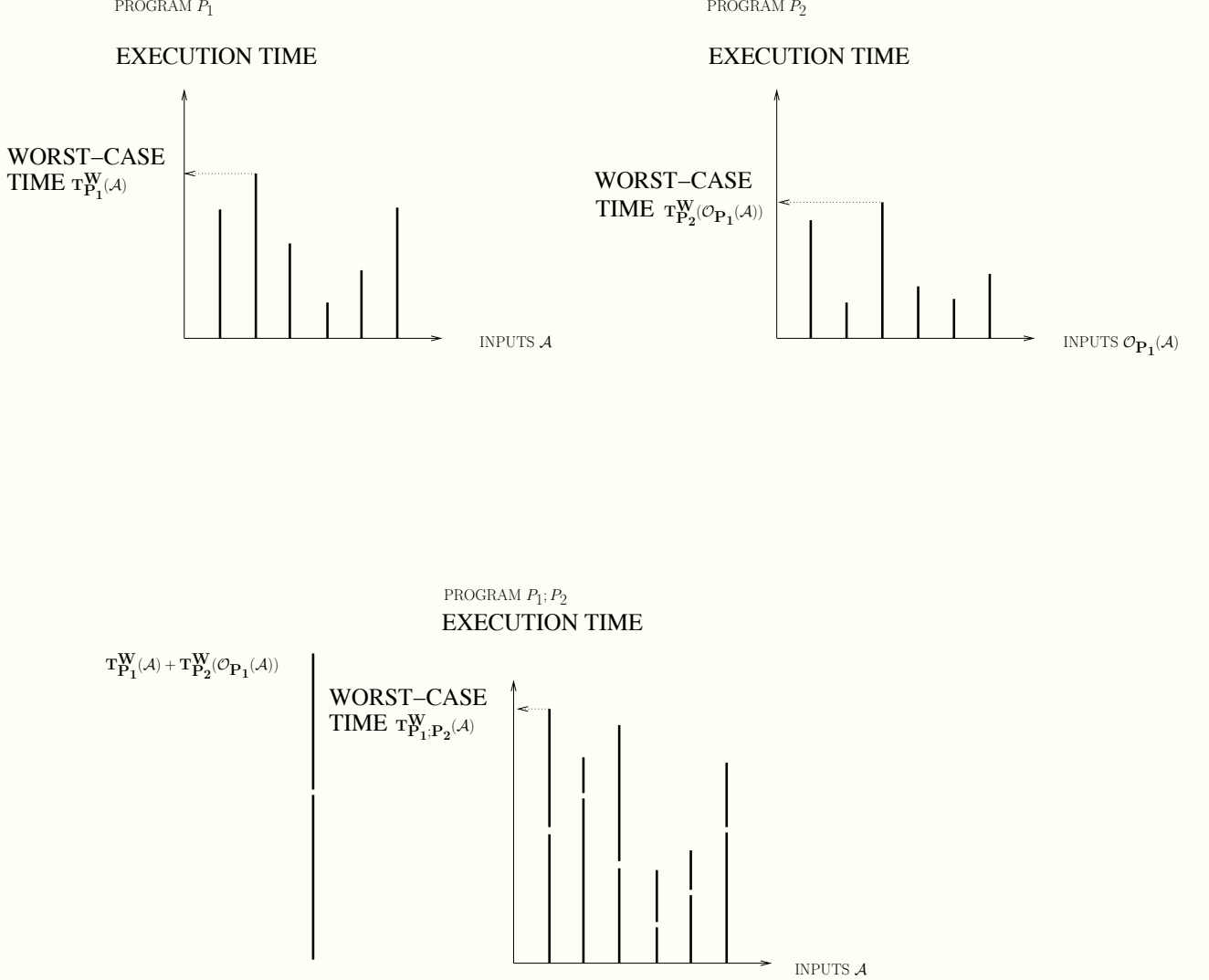
From the above diagram it is clear that, even for this fine-tuned analysis, we still get: $T_{CFL^*}^W(Outputs(P_1, 3)) = 3$ and hence once again:

$$T_{CFL;CFL^*}^W(3) = 5 < T_{CFL}^W(3) + T_{CFL^*}^W(Outputs(CFL, 3)) = 6.$$

The root of the problem can be illustrated by the following counter-example which shows that worst-case time is not modular in general.

In this example $O_{P_1}(\mathcal{A})$ represents the multi-set of outputs obtained by executing P_1 on the input set \mathcal{A} .

Counter-Example 84 (Worst-Case Time)



The non-modularity arises because of the fact that when P_1 reaches a worst-case time (on the second input), P_2 will execute quite fast on the output generated by P_1 .

Similarly, when P_2 reaches worst case time on one of the outputs of P_1 (namely the output of P_1 on the third input); once again, the time which P_1 takes on the third input is actually not very long.

So in general we have:

$$T_{P_1;P_2}^W(\mathcal{A}) \neq T_{P_1}^W(\mathcal{A}) + T_{P_2(\mathcal{O}_{P_1})}^W(\mathcal{A}).$$

In order for the equality to hold, we would need that when P_1 reaches a worst-case on a particular input, say I , then P_2 also reaches its worst-case on the output $P_1(I)$. Of course, as illustrated by the above picture, this is not the case in general!

However, the following fact can be shown to hold in general, i.e. we have a type of “semi-modularity”, where the worst-case time of the composition of two programs can be bounded by the sum of the worst-case times of the components:

10.4.2 Semi-modularity of worst-case time: output-based version

We start by considering an “*output-based*” version of *semi-modularity* for worst-case time, i.e., an expression involving the outputs of the first program P_1 in the sequential composition $P_1; P_2$.

Semi-modularity of worst-case time: output-based version

$$T_{P_1;P_2}^W(\mathcal{A}) \leq T_{P_1}^W(\mathcal{A}) + T_{P_2(O_{P_1}(\mathcal{A}))}^W$$

Proof: Note that for all inputs $I \in \mathcal{A}$ the following holds:

$$T_{P_1;P_2}(I) = T_{P_1}(I) + T_{P_2}(P_1(I)) \leq T_{P_1}^W(\mathcal{A}) + T_{P_2}^W(O_{P_1}(\mathcal{A}))$$

Since this holds for all inputs $I \in \mathcal{A}$, we obtain that

$$T_{P_1;P_2}^W(\mathcal{A}) = \max_{I \in \mathcal{A}} T_{P_1;P_2}(I) \leq T_{P_1}^W(\mathcal{A}) + T_{P_2}^W(O_{P_1}(\mathcal{A}))$$

□

The right-hand side sum only gives an upper bound estimate of the left-hand side, i.e. loosely stated:

“the sum of the worst-case times is an upper bound of the worst-case time of the composition”.

Exercise 85 Show that if P_1 is a constant time algorithm then we the modularity equality holds, i.e., $T_{P_1;P_2}^W(\mathcal{A}) = T_{P_1}^W(\mathcal{A}) + T_{P_2}^W(O_{P_1}(\mathcal{A}))$. Show that the same holds when P_2 is a constant time algorithm.

Semi-modularity of worst-case time: size-based version

Next, we consider a “*size-based*” version of *semi-modularity*, i.e., an expression that only involves the input sizes of both programs in the sequential composition $P_1; P_2$. Given two programs P_1 and P_2 , where P_1 takes inputs of size m and P_2 takes inputs of size n (the size of the outputs of P_1). Then the following holds:



Semi-modularity of worst-case time: size-based version

$$T_{P_1;P_2}^W(m) \leq T_{P_1}^W(m) + T_{P_2}^W(n)$$

Proof: It suffices to observe that if \mathcal{A} is the collection of inputs of P_1 of size m , then $T_{P_2(O_{P_1}(\mathcal{A}))}^W \leq T_{P_2}^W(n)$. The result follows from the output-based version of semi-modularity for the worst-case time.

□

In the Real-Time Languages area, the inequality $T_{P_1;P_2}^W(m) \leq T_{P_1}^W(m) + T_{P_2}^W(n)$ is precisely the technique used to predict the worst-case time in a (loosely stated) “modular fashion”¹. Worst-case time prediction is crucial to help schedule processes. The semi-modularity of worst-case time is used to estimate the worst-case of large programs, consisting of the sequential composition of components.

Consider for instance a program

$$P = Q_1, \dots, Q_n.$$

The real-time engineer analyses the worst-case of the program’s components, i.e. obtains

$$T_{Q_1}^W(m_1), \dots, T_{Q_n}^W(m_n).$$

The program P , by semi-modularity, has a worst-case time

$$T_P^W(m_1)$$

which is guaranteed to be less than or equal to the sum of the worst-case times of these components, namely

$$T_{Q_1}^W(m_1) + \dots + T_{Q_n}^W(m_n).$$

Hence, in practice, this estimate is used as a prediction of the worst-case time execution of P .

In the Real-Time literature, the finer points of the above discussion are sometimes missed and it is occasionally claimed that worst-case time is “compositional” i.e. modular, without really pinpointing the fact that one only obtains an upper-bound, not a true estimate. The loose terminology hides the fact that upper-bounds introduce errors in estimation and that in general the worst-case time prediction over-shoots the real worst-case time of the program. This is not a major issue in real-time programming, where the name of the game is to tailor programs to enable timing

¹Strictly speaking we should refer to “semi-modularity”, but several textbooks simply refer to this property as “modularity”.

of software in the first place. This type of tailoring can cause the creation of slower programs.

Real-Time programming languages, i.e. language designed to enable the timing of their programs, are generally obtained by “clipping the wings” of traditional languages, forcing all programs of a given language to terminate. Examples are Ada and Real-Time Java. Moreover, operations are restricted (e.g. by eliminating while loops (even bounded ones) in favour of for-loops) to facilitate worst-case execution time predictability. Such restrictions ensure timing, but can lead to slower execution time.

This is tolerable in a context where the focus is on timing. As long as the software allows one to time its execution (i.e. predict the worst-case) and scheduling can occur within the required safety-limits (e.g. the automated adjusting the balance of an aeroplane in a reasonably short time in case of turbulence) the safety constraints are satisfied. Hence real-time programming, by its nature, tolerates paying a price, paid in slowing down software, to gain improved predictability. Similarly, upper bound estimates for worst-case time, based on the semi-modularity property, and hence an accumulation of overshoot errors in time estimation, are tolerated in practice, as long as these predicted overshoots still result in reasonable scheduling²

So what about modularity of average-case time measure? As it happens, average-case time satisfies a wonderfully strong modularity property, but nevertheless poses inherent difficulties.

10.5 Modularity of average-case time

10.5.1 Modularity of average-case time: output-based version

As mentioned above, average-case time satisfies a strong modularity property.



Modularity of average-case time: output-based version

Proposition 86 *The average-time measure is modular, i.e. the following equality holds for any two programs P_1, P_2 , where P_1 operates on an input multi-set \mathcal{I} and produces the output multi-set $\mathcal{O}_{P_1}(\mathcal{I})$:*

$$\overline{T}_{P_1;P_2}(\mathcal{I}) = \overline{T}_{P_1}(\mathcal{I}) + \overline{T}_{P_2}(\mathcal{O}_{P_1}(\mathcal{I}))$$

²Note that the upper bounds will affect practice: processes are required to wait for the estimated worst-case time to ensure completion of the computation before the next process kicks in.

Proof:

$$\begin{aligned}
\bar{T}_{P_1;P_2}(\mathcal{I}) &= \frac{\sum_{I \in \mathcal{I}} T_{P_1;P_2}(I)}{|\mathcal{I}|} \\
&= \frac{\sum_{I \in \mathcal{I}} T_{P_1}(I) + \sum_{J \in \mathcal{O}_{P_1}(\mathcal{I})} T_{P_2}(J)}{|\mathcal{I}|} \\
&= \bar{T}_{P_1}(\mathcal{I}) + \bar{T}_{P_2}(\mathcal{O}_{P_1}(\mathcal{I})),
\end{aligned}$$

where the last equality follows from the fact that $|\mathcal{I}| = |\mathcal{O}_{P_1}(\mathcal{I})|$.

□

10.5.2 Average-case time “paradox”

The (output-based) modularity of average-case time is rather surprising since it is well known that average-case time analysis is much harder to determine in practice than worst-case time. This is due to the fact that the first measure needs to take into account the computation times for *all* inputs, while for the second measure it suffices to focus on determining a *specific* extreme input-case.

On the other hand, the modularity of the average-case time, as opposed to the worst-case time, opens the way to a compositional, and hence simplified, determination of average-time recurrences. The problem is to determine the set $\mathcal{O}_{P_1}(\mathcal{I})$ in practice. We will do so for the example of Insertion Sort. In general however the characterization of the output set poses an inherent difficulty. This explains the so-called average-case time “paradox”—not a true paradox, yet a startling observation nevertheless:

Average-case time analysis is relatively hard in general, despite the handy modularity equation for this timing measure.

10.5.3 Modularity of average-case time : failure of size-based version

The average-case time measure does *not* satisfy the following size-based modularity equality.

Failure of the modularity of average-case time: size-based version

$$\bar{T}_{P_1;P_2}(m) \neq \bar{T}_{P_1}(m) + \bar{T}_{P_2}(n),$$

where m is the input-size for program P_1 and n is the input size for program

, P_2 , or, in other words, the size of the outputs of P_1 .

Counter-Example 87 Consider the case where P_1 is the Merge Sort algorithm M and P_2 is the Quicksort algorithm Q , where both algorithms take input lists of size n and the timing measure is the average-case comparison time. Note that Quicksort has the sorted list as a worst-case input, causing $O(n^2)$ comparison time. The sorting algorithm $P_1 = M$, on each of its inputs, returns the sorted output to $P_2 = Q$ as input. Hence,

$$\overline{T}_{M;Q}(n) \in O(n^2)$$

(explain why this is the case). However,

$$\overline{T}_M(m) + \overline{T}_Q(n) = O(n \log n) + O(n \log n) = O(n \log n).$$

Hence

$$\overline{T}_{P_1;P_2}(m) \neq \overline{T}_{P_1}(m) + \overline{T}_{P_2}(n).$$

10.6 Predictability and hardware design

It is clear that (semi-)modularity can provide considerable gain in terms of improved time-predictability. Despite this helpful software property, safety-critical timing remains fraught with problems, providing challenges the real-time engineer needs to address.

Hardware design has largely been focused on producing fast computation devices, where predictability typically was sacrificed at the expense of speed. Pipelining, caching and multi-core contribute speed yet provide the real-time engineer with extreme difficulties on the predictability front. None of these approaches currently support time-prediction due to their inherent complexity.

The safety critical area could benefit from the design of hardware designed to support better predictability of execution behaviour. Currently, the market share of safety-critical applications (including chemical and nuclear plant applications, aeroplane applications, medical devices etc.) is too small to incentivise hardware developers to develop more predictable alternatives.

The situation has somewhat improved with the advent of the automobile industry, expanding the market of safety-critical applications. It is possible that this will drive the development of more predictable hardware to support real-time engineers in their safety-critical work—a future development that can only benefit the users of safety-critical devices.

Chapter 11

Duality

Duality allows one to transform algorithmic problems into other algorithmic problems in a one-to-one fashion. The resulting problems may look quite different from the original ones yet essentially constitute an equivalent problem. Duality plays a crucial role in NP-completeness—a topic we will visit in the final chapter.

11.0.1 Duality at work: a tale of three camels

Once upon a time three nomads purchased a new camel each. They soon found out that the animals were slow. Bickering ensued and each rider claimed that their camel was even slower than the others. They decided on a contest to determine the slowest camel. Each rode on as slowly as possible till the animals traipsed through the sand at snail pace. After half an hour during which their complaining intensified they came upon a woman. She shouted two words at the troupe and they rode off at fast pace. What did the woman shout?

Answer: The woman realized that they could hold a proper race from which the fastest but also the slowest camel would emerge. However, seated on their own camels each rider would gain by sabotaging the new race by holding back their camel as they had before. Hence the woman shouted: “switch camels”. From there on the race proceeded as a race-to-the-top rather than a race-to-the-bottom. The two problems, “race-to-the-bottom” (aimed at the slowest tempo) and “race-to-the-top” (aimed at the fastest tempo) are equivalent provided that the contest aim is inversed (from bottom to top), the rider-camel pairing is altered, ensuring no rider is mounted on their own camel, and the winner of the first race is identified with the loser of the second. This transforms the first problem into the second—a solution for one yields a solution for the other. The transformation illustrates that the selection of a winner for the race-to-the-bottom is dual to selecting a loser for a race-to-the-top.

11.0.2 Duality in graphs

We consider some basic dualities in graph theory.

Definition 88 (*Independent subsets, vertex covers and cliques*)

- 1) An independent subset of a graph is a subset of vertices containing no pairs that are connected by an edge.
- 2) A vertex cover of a graph is a subset of vertices such that each edge has one of its vertices in the subset.
- 3) A clique of a graph is a subset of vertices for which every pair is connected by an edge.

Exercise 89 (*Duality exercises*)

- 1) Show that finding the largest independent subset of a graph is dual to finding the smallest vertex cover of a graph.
- 2) Show that finding the largest independent subset of a graph is dual to finding the smallest clique of a graph.

Chapter 12

Lower bounds

Nature to be commanded must be obeyed.

*Francis Bacon, Novum Organum
(1620)*

“Lower bounds” provide useful information for the algorithm designer, as lower bounds indicate ahead what the best-case scenario will be. lower bounds help us decide whether it still makes sense to try and optimize further. Knowledge of lower bounds also lends useful insight into the nature of a problem.

12.1 Lower bounds on algorithmic improvement

We will show that comparison-based algorithms satisfy certain complexity lower bounds. Both the worst-case time and the average case time of comparison-based algorithms are bounded via $\Omega(n \log_2(n))$. Hence no matter how much ingenuity you pour into their design, such algorithms will never take less computational steps than the given bound. In particular, it is not possible to design linear time comparison-based algorithms.

The design of efficient algorithms involves approaching the obtained bound as closely as possible. The two main complexity lower bounds for comparison-based sorting algorithms are discussed in the next theorem.



lower bounds

Theorem 90 *Every comparison-based sorting algorithm A satisfies:*

$$T_A^W(n) \in \Omega(n \log n) \text{ and } \bar{T}_A(n) \in \Omega(n \log n)$$

12.2 How can this result be demonstrated?

Demonstrating that $T_A^W(n) \in \Omega(n \log n)$ seems like a daunting task. The claim holds for *any* comparison-based algorithm A . How do we show that it holds for all such algorithms? And how do we verify that for each such algorithm A and on *each* of its inputs the execution time takes at least the given bound?

We will introduce the notion of decision tree. A decision tree $DT_A(n)$ for a comparison-based algorithm A is a binary tree which represents *all* possible computation paths on inputs of size n in a finitary way. For size n , the tree will have exactly $n!$ leaves corresponding to the number of inputs of size n .

Given an input of size n , every edge on a branch of the tree corresponds to a comparison carried out by the algorithm between list elements. Every branch of the tree leading from the root to a node records (through annotation on its edges) the comparisons made during the computation made up to that point during the computation on a particular input list. The length of a branch leading from the root of the tree to a leaf corresponds to the number of comparisons made by the algorithm when run on the input giving rise to the computation path to the “output-leaf”.

The branches correspond to the comparison-paths introduced in Section 7.3. A decision tree hence provides information on the comparison-time of the algorithm A . A longest branch in the tree represents the computation path for a worst-case input. Its length is the worst-case time. The average path length of all branches in the tree corresponds to the average-case time.

We will show that the length of the longest path in a binary tree with $n!$ leaves must satisfy the $\Omega(n \log n)$. The result then follows for the worst-case time of *all* comparison-based algorithms.

We will treat the average-case time in a later section.

We specify the construction of decision trees more formally in the next section.

12.3 Decision trees: definition and example



Decision trees

We will use decision trees to represent all possible comparison-paths of a comparison-based sorting algorithm—a gateway to worst-case and average-case *comparison time* considerations.

The path lengths in our tree will correspond to comparison-paths, which in turn represent the comparisons made per input.

Decision trees essentially will be binary trees representing all binary comparison-paths of a comparison-based algorithm (cf. Definition 51), where the binary tree will be annotated with more information (the outcomes of the comparisons during the

execution of the algorithm and the inputs that gave rise to this particular computation).

For each of the $n!$ possible inputs of a comparison-based sorting algorithm, one obtains exactly one comparison-path per input.



Unique comparison-paths

Proposition 91 *Each input among the possible $n!$ inputs of a comparison-based sorting algorithm gives rise to a unique comparison-path.*

Proof: Recall that comparison-based algorithms can only gain information about an input by comparing pairs of elements. Recall also that by Lemma 63 a comparison-based sorting algorithms must compare every pair of elements of a given input list. So if two *different* inputs give rise to the same comparison-path, then the algorithm would perform exactly the same comparisons between pairs of lists, in the same execution order and with the same outcome (“yes” or “no”) for both input lists. Hence, these two lists must have the same relative order (for all pairs, we obtain that they have the same relative order). This implies that these two input lists must be the same—a contradiction. Hence the result follows.

□

Given an comparison-based sorting algorithm A . The $n!$ input lists of size n are represented at the root of the decision tree via the input list (a, b, c, \dots) , which can represent any of the $n!$ input lists of size n . For instance, inputs of size 3 are represented as (a, b, c) . For any comparison-based sorting algorithm A and for any input size n , the decision tree $DT_A(n)$ is created as follows:



First step in creating a decision tree

At the root of the tree $DT_A(n)$ we record the possible inputs of the given size n . These $n!$ inputs are the permutations of the list $(1, \dots, n)$.

For the next step, consider the first comparison between list elements $L[i]$ and $L[j]$ carried out by the algorithm A on an input list L of size n (as specified by the pseudo-code). Say, this is a comparison between the elements $L[i]$ and $L[j]$.

For such a comparison, we generate two edges leading from the root to two new nodes.

- The left edge is labeled with $L[i] < L[j]$ indicating that the result of

comparing $L[i]$ and $L[j]$ yields that $L[i]$ is smaller than $L[j]$.

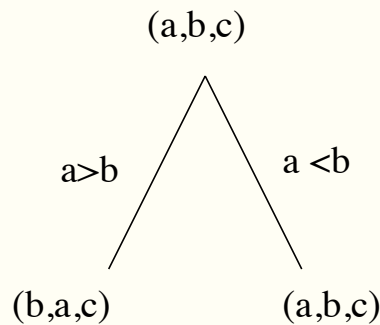
- The right edge is labeled with $L[i] > L[j]$ indicating that the result of comparing $L[i]$ and $L[j]$ yields that $L[i]$ is greater than $L[j]$.

This creates a partial tree, which forms part of the *decision tree* and records the “decisions” made by the algorithm during the computation up to that stage.

For comparison-based sorting algorithms, all decisions are governed by the comparisons made between elements. Depending on the outcome of a comparison, one of two actions will be taken by the algorithm, i.e. a decision will be made.

For comparison-based algorithms, the actions taken by the comparisons are “swaps”. We record these swaps in the decision tree.

For instance, assume that the algorithm has an input list of size 3, $L = (a, b, c)$, and assume that the first comparison is between $L[1]$ and $L[2]$, i.e. between a and b . Assume that the algorithm swaps a and b in case $a > b$, but the algorithm does not take any action in case $a < b$. The decision tree records this information as follows:



Note that at the left leaf, we recorded the swap-result on a and b , namely (b, a, c) . Since no swap occurs in case $a < b$, the right leaf simply records the original input list without change, i.e. (a, b, c) .



Splitting the inputs

After carrying out the first comparison, the original collection of $n!$ lists is split into two groups of lists: the lists of size n for which $a > b$ and the lists of size n for which $a < b$.

We keep track of the inputs in a decision tree.

Inputs at the root and its two leaves

At the root of a decision tree for lists of size 3, in addition to the notation (a, b, c) , we record the $3!$ possible kinds of input lists that the input (a, b, c) could actually be. These are represented by:

$(1, 2, 3), (1, 3, 2), (2, 1, 3), (2, 3, 1), (3, 1, 2), (3, 2, 1)$.

- The left branch corresponds to the computation on input lists (a, b, c) for which the first element a is greater than the second element b , namely the lists:

$(2, 1, 3), (3, 1, 2), (3, 2, 1)$.

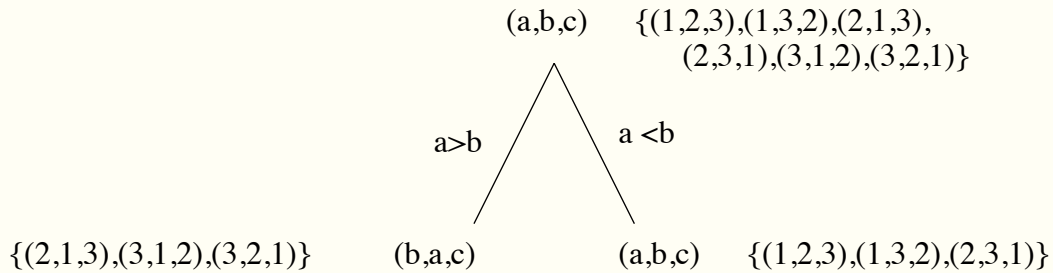
These lists are recorded at the left-child of the root.

- The remaining inputs, i.e. the inputs for which $a < b$, namely the lists:

$(1, 2, 3), (1, 3, 2), (2, 3, 1)$

These are recorded at the right-child of the root.

The part of the decision tree representing the computation up to this point hence is updated as follows:



Producing the entire decision tree

The decision tree is subsequently updated in a similar way for *each* comparison carried out by the algorithm (in the order that the comparisons are made during this execution).

The leaves of the tree are updated as follows:



The leaves of the decision tree

Note that a decision tree for inputs of size n has exactly $n!$ leaves.

The leaves are marked with:

- a single input list, which represents the input that gave rise to this particular computation path.
- a permutation of the input list (a, b, c) , which records the outcome of the computation path, i.e. the effect of the swaps that occurred along the path on the input list (a, b, c) .

Decision trees have exactly $n!$ leaves since each branch of the tree from the root to a leaf corresponds to a binary comparison path for one of the $n!$ inputs. Each input represent a different relative order. By corollary 68 there are exactly $n!$ binary comparison paths and hence the tree has $n!$ leaves.

Each permutation of (a, b, c) recorded at a leaf represents the swaps that have occurred on elements along the path leading to the leaf. As the algorithm is a sorting algorithm, these swaps must sort the original input.

Distinct inputs hence must be paired with different permutations of (a, b, c) at each leaf (as permutations are injections which will map different inputs to different values. Hence no two distinct inputs can be sorted by the same permutation).

The leaves of the decision tree play a special role in verifying the correctness of the tree.



Correctness Test

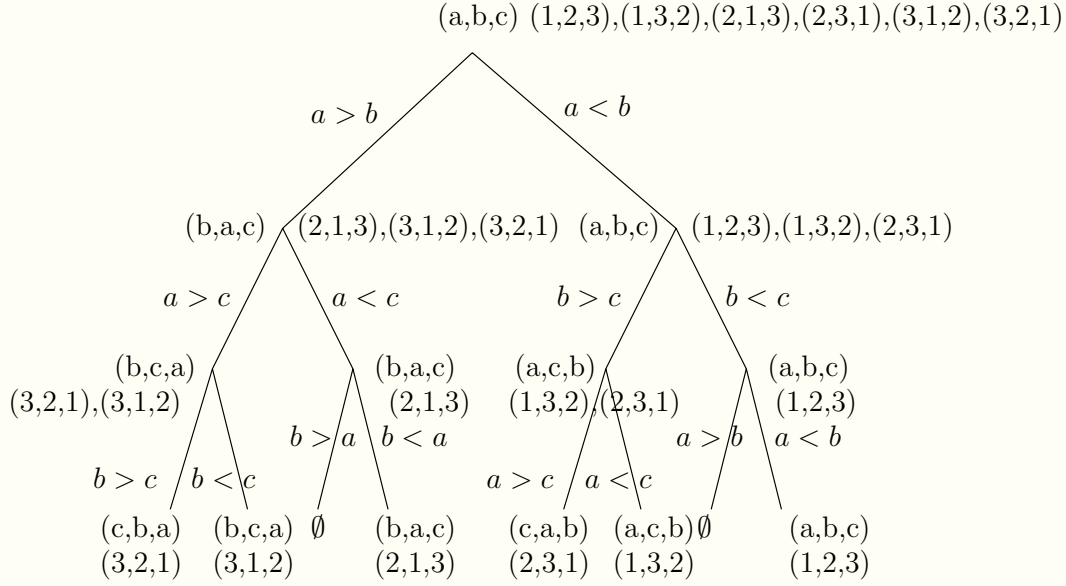
The leaves and their annotations help one test whether the algorithm sorts properly.

Say the input list recorded at a leaf is the list L . Execute the permutation of the list (a, b, c) , as recorded at the leaf, on the input list L . The result should be the sorted list.

This should hold for each of the $n!$ leaves of the decision tree.

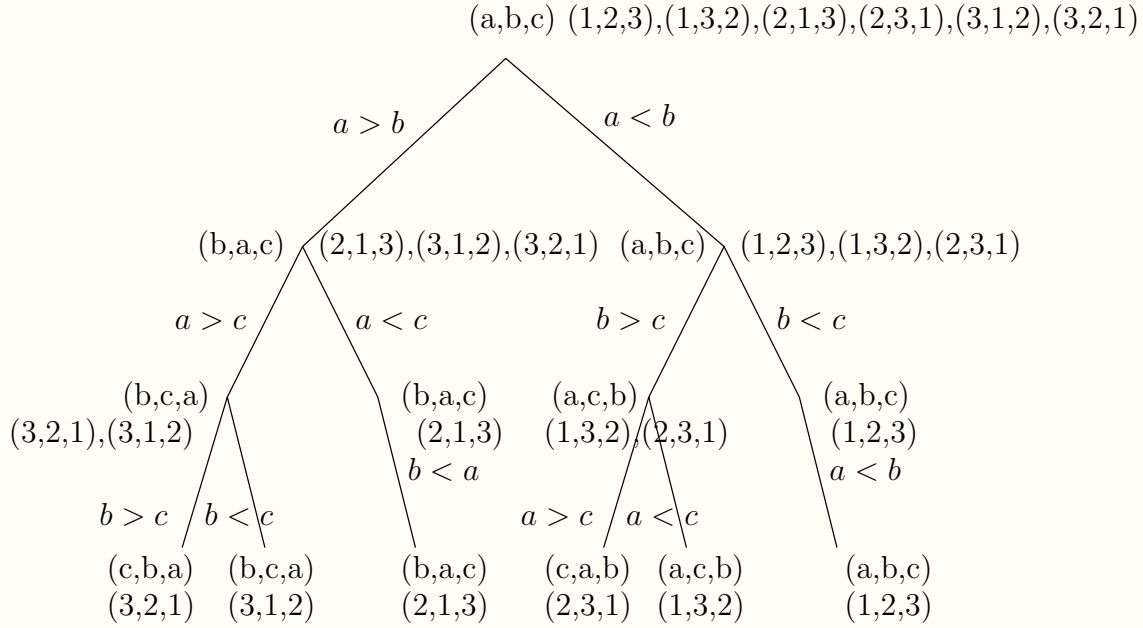
We illustrate this on the decision tree $DT_B(3)$ for the sorting algorithm Bubble Sort for lists of size 3, constructed via the process described above. The tree has $n! = 3! = 6$ leaves.

DECISION TREE FOR BUBBLE SORT ON LIST OF SIZE 3



Typically one prunes the leaves which have the empty set of inputs attached to them, since none of the inputs gives rise to a computation leading to such a leaf. This is illustrated in the picture below.

DECISION TREE FOR BUBBLE SORT ON LIST OF SIZE 3



The path from the root to a given leaf, corresponds to the comparisons which Bubble Sort would carry out on the input attached to the leaf. For instance, the left-most branch from the root to the (left-most) leaf (c,b,a), originates from the computation on the input (3,2,1).

We carry out the correctness test on the decision tree.

Correctness Test: The input (a,b,c) is systematically transformed along any of the 6 branches to an output, recorded at the corresponding leaf, which is a permutation of the original three values a,b and c.

For instance, the first leaf (in left to right order on the above tree) is (c,b,a). The input which gives rise to this computation path is also indicated at this leaf: (3,2,1). To test that this branch has been constructed correctly, note that (a,b,c) is transformed along this computation path into (c,b,a). Hence the input (a,b,c) = (3,2,1) will be transformed to the output (c,b,a), which is (1,2,3), i.e. the sorted output.

Since the algorithm is a sorting algorithm, *every* input will be transformed in this way to the sorted output.

We verify this on the fourth leaf encountered in left-to-right order, labeled with the input-permutation (c,a,b) and the input list (2,3,1). Note that for (a,b,c) = (2,3,1) we obtain that (c,a,b) = (1,2,3), i.e. the sorted output.

Verify as an exercise that this holds true for the four remaining leaves in the decision tree, and hence that the algorithm sorts each of its 6 input lists.

Exercise 92 a) Determine the decision tree for the sorting algorithm Selection Sort and for lists of size 3. We recall Selection Sort's pseudo-code from Section 9.3.2.

Selection Sort(L)

```
for i = 1 to |L| - 1 do
  k ← i
  l ← L[i]
  for j = i + 1 to |L| do
    if L[j] < l then k ← j
    l ← L[j]
  SWAP(L[i], L[k])
```

b) Determine the decision tree for Insertion Sort. We recall the pseudo-code.

Insertion Sort(L)

```
for i = 2 to |L| do
  j := i
  while (j > 1 and L[j - 1] > L[j]) do
    swap (L[j - 1], L[j])
  j := j - 1
```


c) Determine the decision tree for the following simplified (and less efficient) version of Quick Sort and for lists of size 3.

Lomuto-Partition(A, lo, hi)

```

    pivot := A[hi]
    i := lo
    for j := lo to hi - 1 do
        if A[j] ≤ pivot then
            swap(A[i], A[j])
            i := i + 1
    swap(A[i], A[hi])
    return i

```

Lomuto's version of Quick Sort is defined by:

Lomuto-Quick-Sort(A, lo, hi)

```

    if lo < hi then
        p := partition(A, lo, hi)
        Lumoto-Quick-Sort(A, lo, p - 1)
        Lumoto-Quick-Sort(A, p + 1, hi)

```

In the following section we will demonstrate that the worst-case time of a comparison-based sorting algorithm must satisfy the $\Omega(n \log n)$ lower bound.

The following two intermediate results will be useful in the proof of this fact.



Asymptotic lower bound for $\log_2(n!)$

Lemma 93 For any positive integer n , the following holds:

$$\log_2(n!) \in \Omega(n \log_2 n) \quad (12.1)$$

We prove a series of lemmas leading to the above result.

Lemma 94 $n! \geq \left\lceil \frac{n}{2} \right\rceil^{\left\lceil \frac{n}{2} \right\rceil}$

Proof:

$$n! = n \times (n-1) \times (n-2) \times \cdots \times \left\lceil \frac{n}{2} \right\rceil \times \cdots \times 2 \times 1$$

$$\Rightarrow n! \geq n \times (n-1) \times (n-2) \times \cdots \times \left\lceil \frac{n}{2} \right\rceil$$

$$\Rightarrow n! \geq \left\lceil \frac{n}{2} \right\rceil^{\left\lceil \frac{n}{2} \right\rceil}$$

□

Remark 95 We verify the above statement on examples for even values of n and odd values of n , say $n = 6$ and $n = 7$.

For $n = 6$, $6! = 6 \times 5 \times 4 \times 3 \times 2 \times 1$, $\lceil \frac{n}{2} \rceil = \frac{n}{2} = 3$ and $6! \geq 3^3$ since

$$6 \times 5 \times 4 \geq 3 \times 3 \times 3$$

For $n = 7$, $7! = 7 \times 6 \times 5 \times 4 \times 3 \times 2 \times 1$, $\lceil \frac{n}{2} \rceil = 4$ and $7! \geq 4^4$ since

$$7 \times 6 \times 5 \times 4 \geq 4 \times 4 \times 4 \times 4$$

Lemma 96 $\log_2(\lceil \frac{n}{2} \rceil^{\lceil \frac{n}{2} \rceil}) \geq \frac{1}{4}n \log_2 n$ for values of n of 4 or higher.

Proof: We prove the fact for the case where n is odd and leave the case where n is even as a (similar) exercise. Note that when n is odd, $\lceil \frac{n}{2} \rceil = \frac{n+1}{2}$.

$$\text{So } \log_2(\lceil \frac{n}{2} \rceil^{\lceil \frac{n}{2} \rceil}) = \log_2((\frac{n+1}{2})^{(\frac{n+1}{2})}) = \frac{n+1}{2} \log_2(\frac{n+1}{2}) \geq \frac{n}{2} \log_2(\frac{n}{2}) = \frac{n}{2}(\log_2(n) - 1).$$

It suffices to show that $\log_2(n) - 1 \geq \frac{1}{2} \log_2(n)$ for $n \geq 4$.

$(\log_2(n) - 1 \geq \frac{1}{2} \log_2(n)) \Leftrightarrow (\log_2(n) - \log_2(\sqrt{n}) \geq 1) \Leftrightarrow (\log_2(\sqrt{n}) \geq 1)$ which holds for $n \geq 4$.

□

Combining Lemma 94 and Lemma 96 yields Lemma 93

$$\log_2(n!) \in \Omega(n \log_2 n)$$

12.4 Verification of $\Omega(n \log n)$ lower bound for worst-case time

We verify that the worst-case comparison time of a comparison-based sorting algorithm which takes as inputs lists of size n , is $\Omega(n \log n)$.

Note that for any comparison-based sorting algorithm A , which takes as inputs lists of size n , the decision tree $DT_A(n)$ has $n!$ leafs (one leaf per input). Recall that the longest path in this tree has a length¹ which is the worst-case comparison time of A on lists of size n .

We show the following result which essentially states that “the more leaves a binary tree has, the longer the length of its longest path needs to be.”

¹Length of a path = number of edges on path = number of nodes on the path - 1 = number of comparisons on the branch.

“Densely crowned trees have long branches”



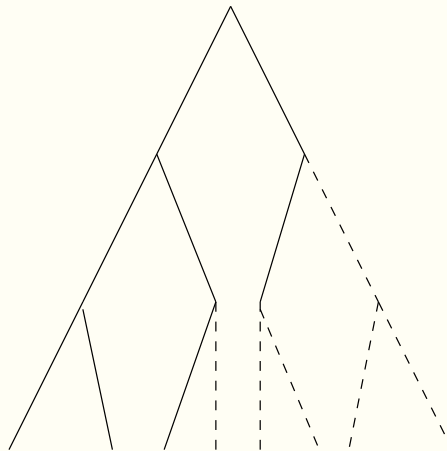
Proposition 97 *In any binary tree T the following holds:*

*The length of the longest path in a binary tree T is at least $\log_2(N)$,
where N is the number of leaves in T .*

“The more leaves a tree has, the longer its longest path (“branch”) needs to be.”

Proof: To show this, note that any binary tree T can be embedded in a complete binary tree T^* , which has the same depth as T . We recall that a complete binary tree is a binary tree which, at each level, has the maximum number of nodes possible at that level. I.e. at level 0 there is $2^0 = 1$ node, at level 1 there are $2^1 = 2$ nodes, ..., at level k there are 2^k nodes. And hence the tree has a “triangular” shape.

To embed a binary tree in a complete binary tree, simply follow the process indicated on the picture below, where the full lines indicate the branches of a given binary tree T and where the dashed lines indicate additional edges “grown” on T to create the complete binary tree T^* which has same depth as T .



Note that obviously the number N of leaves of T is less than or equal to the number of leaves of T^* . Since the number of leaves of T^* is 2^k where k is the maximum level in the tree, we obtain that

$$N \leq 2^k.$$

Note that k is the length of the longest path in the tree T .

From $N \leq 2^k$, we obtain that $\log_2(N) \leq k$.

□

Exercise 98 Consider a binary tree T which has 1 million leaves. What can you say about the length of its longest path?

Answer: T has 1 million leaves. Hence $N = 10^6$. The length of the tree's longest path is at least $\log_2(10^6) = 19.9 \approx 20$.



lower bound for worst-case time

Corollary 99 The worst-case time for comparison-based algorithms on lists of size n is $\Omega(n \log n)$.

Proof: Note that for any comparison-based algorithm A which takes input lists of size n , the decision tree $DT_A(n)$ is a binary tree with $n!$ leaves. From the previous proposition, we obtain that the length of the longest path k in this tree is at least $\log_2(n!)$. The length of the longest path in the decision tree equals the number of comparisons the algorithm makes during the computation along that path, i.e. the algorithm's worst-case time. The result follows by Lemma 93.

□

12.5 Kraft's inequality

The Kraft inequality captures an interesting property of binary trees.

Definition 100 Given a tree T with N leaves.

a) The path length sequence of the binary tree is the sequence (l_1, \dots, l_N) consisting of the lengths of the paths from the root to the N leaves of T .

b) The Kraft number of a binary tree T is defined to be $K(T) = \sum_{i=1}^N 2^{-l_i}$, where (l_1, \dots, l_N) is the path length sequence of T .



Kraft inequality

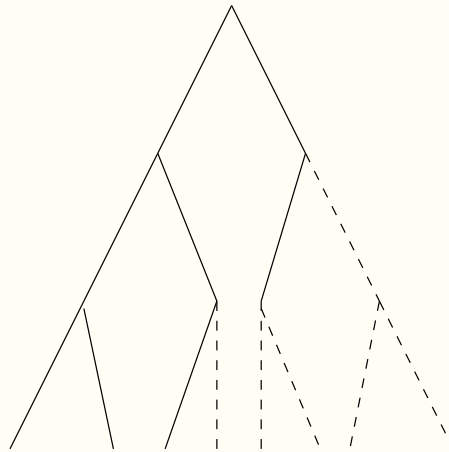
Proposition 101 *The Kraft inequality states that for each binary tree with path length sequence (l_1, \dots, l_N) the following inequality holds:*

$$K(T) \leq 1.$$

In case the binary tree is a full binary tree, i.e. each node has 0 or 2 children, then the inequality is actually an equality:

$$K(T) = 1.$$

We consider the following example, where T denotes the binary tree displayed below with the “full-line” edges and T^* denotes the binary tree consisting of T and the additional “dashed-line” edges.



The path-length sequence for T (in left-to-right leaf order) is $(3, 3, 3, 2)$

$$K(T) = 2^{-3} + 2^{-3} + 2^{-3} + 2^{-2} = \frac{3}{8} + \frac{1}{4} = \frac{5}{8} \leq 1$$

The path-length sequence for T^* in left-to-right leaf order is: $(3, 3, 3, 3, 3, 3, 3, 3)$

$$K(T^*) = 8 \times 2^{-3} = 1$$

The Kraft inequality involves negative powers of 2. Why would this be the case?

For a binary tree having only a root and two children the Kraft inequality clearly holds:

$$2^{-1} + 2^{-1} = 1$$

This is also the case for complete binary trees. Complete binary trees (such as the tree T^* displayed above) are full binary trees for which all path-lengths have the same

size, i.e. the tree is displayed as a “perfect triangle.” Complete binary trees, at every depth level (starting from level 0 at the root, level one for the root’s children and so on) have exactly 2^k leaves at every depth-level. For such a tree, Kraft’s inequality amounts to:

$$\sum_{i=1}^{2^k} 2^{-k} = 1.$$

One intuition on why the Kraft inequality involves negative powers of two could be linked to the fact that complete binary trees double in number of nodes at each level. However, binary trees come in many shapes. The Kraft inequality involves path lengths and it is clear that paths overlap. So the Kraft inequality cannot be linked to tree size (number of nodes) in a direct way.

Instead we will look at the number of ways in which binary trees can be extended to form larger binary trees. The argument is due to Mortada Mehyar as outlined on his blog “Random Points².” The proof clearly shows that the Kraft inequality exactly captures the condition that prefix codes (and hence binary trees) must satisfy the prefix-exclusion principle (which for binary trees amounts to the typical cycle-freeness condition of trees). The last line in the proof in particular removes any puzzlement the reader may have as to why negative powers of two are involved in the inequality. They represent the part that is banned in the prefix-exclusion (represented by a negative sign).

Mehyar’s argument is formulated for prefix codes. We present the proof for the case of prefix codes that are special types of collections of binary numbers. We will return to such codes later on in the context of file compression and Huffman encoding.

Definition 102 *A prefix code is a finite collection of binary numbers, none of which is a prefix of the other.*

Example 103 $\{0, 10, 110, 111\}$ is a prefix code. $\{0, 10, 100, 101\}$ is not a prefix code since 10 is a prefix of the binary number 100 (and of 101).

Definition 104 *A 01-labelled binary tree is a binary tree in which left branches are labelled with 0 and right branches labelled with 1. Binary sequences obtained by recording the zeros and ones occurring on the paths from the root to a leaf³ in a 01-labelled binary tree are referred to as binary-paths.*

Example 105 *The complete 01-labelled binary tree T of size 7 is determined by the binary paths: 00, 01, 10 and 11. Note that these binary sequences form a prefix code. Can you explain why?*

Lemma 106 *Prefix codes are in bijection with 01-labelled binary trees.*

²mortada.net, “Simple proof of Kraft’s inequality”

³Akin to the binary comparison paths we have encountered before, but not entirely the same concept. Binary comparison paths are obtained from decision trees, where zeros can occur on left or right branches and likewise for the case of ones.

Proof: (prefix codes and 01-labelled binary trees equivalence)

a) The binary paths of a 01-labelled binary tree form a prefix code. This is clear, since otherwise a leaf of the binary tree would also be an internal node of some path (i.e. a non-leaf), contradicting the cycle-free condition of a tree.

b) Prefix codes can be used to define a binary tree in the obvious way: read each binary number in left to right order. Starting from a newly introduced root node, from each created node, form a branch to the left in case of a 0 and a branch to the right in case of a 1 for each entry in the binary number under consideration. This determines a branch in the tree for each binary number in the prefix code. No cycles are created as none of the binary numbers is a prefix of another binary number.

□

We will discuss prefix codes later in the course in relation to file compression and Huffman encodings. For now we use such codes as a handy notion to represent 01-labelled binary trees. We state the following theorem in terms of 01-labelled binary trees. The use of prefix codes in the proof simplifies matter a little, where it is easier to discuss binary number extensions than tree extensions.

Theorem 107 (*Kraft inequality*) *A sequence of positive integers (l_1, \dots, l_k) forms the path length sequence of a binary tree if and only if $\sum_{i=1}^k 2^{-l_i} \leq 1$*

Proof: (Proof of Kraft inequality adapted from an argument given by M. Mortada) We show that a sequence of positive integers (l_1, \dots, l_k) form the lengths of the binary numbers in a prefix code if and only if $\sum_{i=1}^k 2^{-l_i} \leq 1$

Without loss of generality, we can assume that the binary number lengths are listed in increasing order, i.e. $l_1 \leq \dots \leq l_k$ ⁴. We distinguish cases.

Case 1) $k = 1$ In this case the prefix code consists of a single binary number of length l_1 . No prefix condition applies.

Case 2) $k \geq 2$ We will analyze conditions for the existence of a binary number of length j to be added to a priorly created prefix code (for which the binary numbers satisfy the given lengths l_1, \dots, l_{j-1}). The existence of such a binary number is formulated in relation to the first $j - 1$ binary numbers already obtained (for some value of $j - 1$ where $1 \leq j - 1 < k$, i.e., $2 \leq j \leq k$). Clearly a first such binary number is obtainable as for case 1).

Given $j - 1$ binary numbers that constitute our current prefix code. Any new binary number to be added to the prefix code must exclude all the other binary numbers as prefix.

Without this condition there would be 2^{l_j} binary numbers to choose from.

⁴Exercise: consider what this means for binary trees corresponding to the prefix codes.

With the condition we note that any binary number that has one of the $j - 1$ chosen binary numbers as prefix is excluded. If we focus on ones such binary number of length l_i (where $i \in \{1, \dots, j - 1\}$) then there are $(*)2^{l_j - l_i}$ binary numbers that have the chosen binary number of length l_i as prefix and hence need to be excluded.

The lengths of the chosen binary numbers are l_1, \dots, l_{j-1} . Hence the total number of “forbidden” binary numbers of length l_j that have one of these $j - 1$ binary numbers as prefix is:

$$\sum_{i=1}^{j-1} 2^{l_j - l_i}$$

In other words, for a prefix code binary number to exist, there must be enough binary numbers to chose from once the forbidden ones have been excluded, i.e. we must have:

$$2^{l_j} > \sum_{i=1}^{j-1} 2^{l_j - l_i}$$

Since all numbers involved are integers, this is equivalent to:

$$2^{l_j} \geq \sum_{i=1}^{j-1} 2^{l_j - l_i} + 1 = \sum_{i=1}^j 2^{l_j - l_i}$$

Hence, for all values of $j \in \{2, \dots, k\}$ the following must hold:

$$2^{l_j} \geq \sum_{i=1}^j 2^{l_j - l_i}$$

Dividing by 2^{l_j} , we obtain that for all values of $j \in \{2, \dots, k\}$:

$$1 \geq \sum_{i=1}^j 2^{-l_i}$$

Every term in each summation of this collection of $k - 1$ inequalities is positive, thus

$$\sum_{i=1}^k 2^{-l_i} \geq \sum_{i=1}^{k-1} 2^{-l_i} \geq \dots \geq \sum_{i=1}^2 2^{-l_i}$$

(**) Hence the collection of $k - 1$ inequalities is equivalent to the single inequality

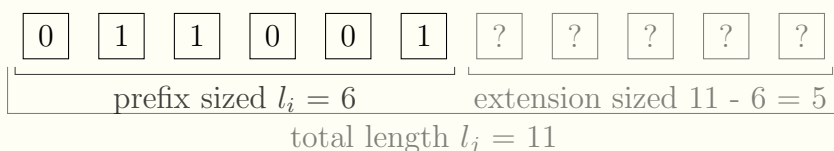
$$1 \geq \sum_{i=1}^k 2^{-l_i}$$

i.e., the Kraft inequality.

☐

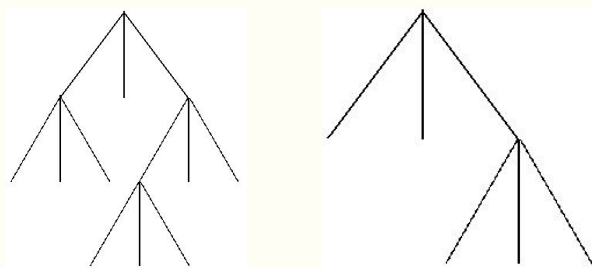
Exercise 108 *Did you fully understand the proof? Verify fact (*), i.e., verify that if we focus on one binary number of length l_i and exclude all binary numbers of length l_j that have the given binary number of length l_i as prefix then $2^{l_j-l_i}$ such binary numbers need to be excluded.*

Answer: Consider a binary sequence of length $l_j \geq 1$ and a positive integer i where $1 \leq i < j$. Verify that there are $2^{l_j-l_i}$ binary numbers of length l_j that have the binary sequence of length l_i as prefix. Start with the following example and generalize. Say we have chosen the binary sequence 011001 as part of the prefix code and say we are considering the choice of a binary number of length $l_j = 11$. We need to avoid all binary numbers of length 11 starting with prefix 011001. These binary numbers take the following shape:



There are $2^{l_j-l_i} = 2^{11-6} = 2^5 = 32$ binary numbers of length $l_j = 11$ that have the binary sequence 011001 of length $l_i = 6$ as prefix.

Exercise 109 Consider the following two full ternary trees (i.e. trees where every node has zero or three children):



1. For each of the trees, write down the path lengths to each leaf from the root of the tree. For each tree T , calculate

$$f(T) = \sum 3^{-p}$$

where p ranges over all the path lengths of T (with repetition).

2. Based on this information, and on your knowledge of the Kraft Equality for complete binary trees, what would you expect to be the behaviour of $f(T)$ where T is a complete ternary tree?

3. What would you expect to be the generalisation of the Kraft Inequality for ternary trees in general (not necessarily complete)?

Exercise 110 Use the Kraft inequality to give an alternative proof for Corollary 99.

12.6 Jensen's inequality

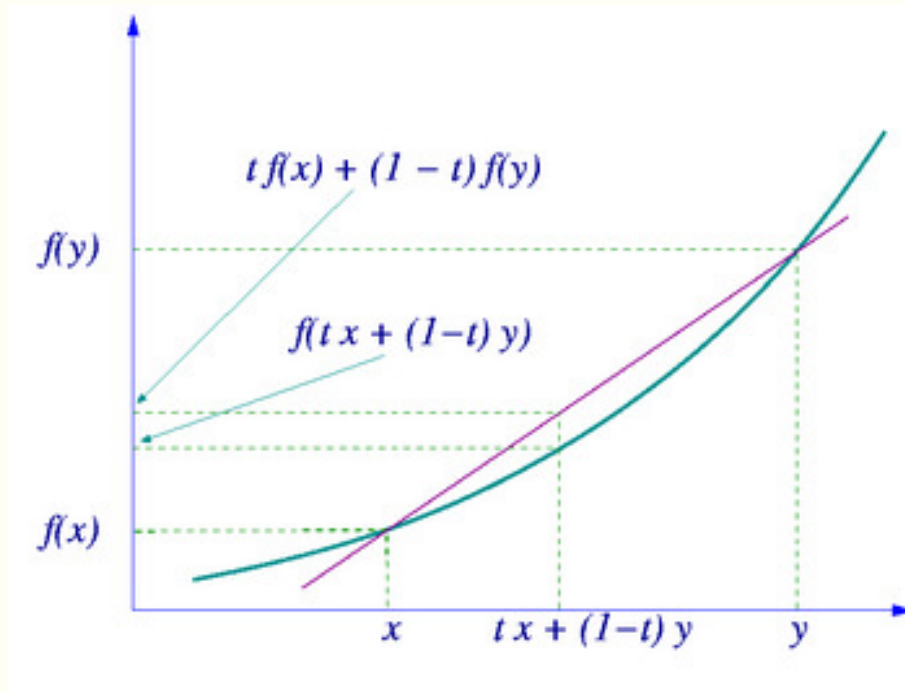
Jensen's inequality regards convex real-valued functions f , for which the function values over an interval $[a, b]$ are always bounded by the line segment values for the segment connecting $(a, f(a))$ and $(b, f(b))$.



Convex functions

Definition 111 A real-valued function f defined on an interval is called convex, if for any two points x and y in its domain and any t in $[0, 1]$, we have

$$(*) \quad f(tx + (1 - t)y) \leq tf(x) + (1 - t)f(y).$$



If we consider the convexity inequality $(*)$ for the case where $t = \frac{1}{2}$, we obtain that

$$f\left(\frac{x+y}{2}\right) \leq \frac{f(x)}{2} + \frac{f(y)}{2}$$

. In other words, “the value of the mean is bounded by the mean of the values”.

Jensen’s inequality, named after the Danish mathematician Johan Jensen, generalizes the above statement. The inequality generalizes the above case from two values to $n \geq 2$ values and establishes that “the value of a convex function on an average value is bounded by the average value of the convex function”. This result was obtained by Jensen in 1906.

Jensen’s inequality

Lemma 112 *If φ is a convex function, $\lambda_1, \lambda_2, \dots, \lambda_n$ are positive real numbers such that $\lambda_1 + \lambda_2 + \dots + \lambda_n = 1$ and x_1, x_2, \dots, x_n belong to the domain of φ then:*

$$\varphi(\lambda_1 x_1 + \lambda_2 x_2 + \dots + \lambda_n x_n) \leq \lambda_1 \varphi(x_1) + \lambda_2 \varphi(x_2) + \dots + \lambda_n \varphi(x_n)$$

Remark 113 *Note that for the case where $\lambda_1 = \dots = \lambda_n = \frac{1}{n}$, Jensen’s inequality simply states that a convex function value of the mean value of a selection of numbers is bounded by the mean value of the function values over these numbers. Hence Jensen’s result is applicable in the context of average-case time, which is based on the notion of a mean. The proof of Jensen’s inequality proceeds by induction.*

Proof: Consider two arbitrary positive real numbers λ_1, λ_2 , such that $\lambda_1 + \lambda_2 = 1$ and x_1, x_2 elements in the domain of φ , then the convexity of φ implies that:

$$\varphi(\lambda_1 x_1 + \lambda_2 x_2) \leq \lambda_1 \varphi(x_1) + \lambda_2 \varphi(x_2).$$

Hence the inequality holds for $n = 2$.

Assume that the inequality holds for $n \geq 2$, where $\lambda_1, \lambda_2, \dots, \lambda_n$ are n positive real numbers such that $\lambda_1 + \lambda_2 + \dots + \lambda_n = 1$ and x_1, \dots, x_n are elements in the domain of φ , then:

$$\varphi(\lambda_1 x_1 + \lambda_2 x_2 + \dots + \lambda_n x_n) \leq \lambda_1 \varphi(x_1) + \lambda_2 \varphi(x_2) + \dots + \lambda_n \varphi(x_n)$$

We show that the inequality holds for $n + 1$.

At least one of the λ_i is strictly positive, say λ_1 . Therefore, by the convexity inequality:

$$\begin{aligned} \varphi\left(\sum_{i=1}^{n+1} \lambda_i x_i\right) &= \varphi\left(\lambda_1 x_1 + (1 - \lambda_1) \sum_{i=2}^{n+1} \frac{\lambda_i}{1 - \lambda_1} x_i\right) \\ &\leq \lambda_1 \varphi(x_1) + (1 - \lambda_1) \varphi\left(\sum_{i=2}^{n+1} \frac{\lambda_i}{1 - \lambda_1} x_i\right). \end{aligned}$$

Since $\sum_{i=2}^{n+1} \frac{\lambda_i}{1-\lambda_1} = 1$, one can apply the induction hypotheses to the last term in the previous formula:

$$\begin{aligned} \varphi \left(\sum_{i=1}^{n+1} \lambda_i x_i \right) &\leq \lambda_1 \varphi(x_1) + (1 - \lambda_1) \left(\sum_{i=2}^{n+1} \frac{\lambda_i}{1 - \lambda_1} \varphi(x_i) \right) \\ &= \lambda_i \sum_{i=1}^{n+1} \varphi(x_i) \end{aligned}$$

□

Similar to the notion we can define concave functions by switching around the inequality in Definition 114.



Concave functions

Definition 114 A real-valued function f defined on an interval is called concave, if for any two points x and y in its domain and any t in $[0, 1]$, we have

$$(**) f(tx + (1 - t)y) \geq tf(x) + (1 - t)f(y).$$

If we consider the concave inequality $(**)$ for the case where $t = \frac{1}{2}$, we obtain that

$$f\left(\frac{x + y}{2}\right) \geq \frac{f(x)}{2} + \frac{f(y)}{2}$$

- . In other words, “the value of the mean is bounded below by the mean of the values”. Jensen’s inequality holds for the opposite inequality sign.



Jensen’s inequality

Lemma 115 If φ is a concave function, $\lambda_1, \lambda_2, \dots, \lambda_n$ are positive real numbers such that $\lambda_1 + \lambda_2 + \dots + \lambda_n = 1$ and x_1, x_2, \dots, x_n belong to the domain of φ then:

$$\varphi(\lambda_1 x_1 + \lambda_2 x_2 + \dots + \lambda_n x_n) \geq \lambda_1 \varphi(x_1) + \lambda_2 \varphi(x_2) + \dots + \lambda_n \varphi(x_n)$$

Remark 116 Note that for the case where $\lambda_1 = \dots = \lambda_n = \frac{1}{n}$, Jensen’s inequality simply states that a convex function value of the mean value of a selection of numbers is bounded below by the mean value of the function values over these numbers. Hence Jensen’s result for concave functions is applicable in the context of average-case time, which is based on the notion of a mean.

12.7 Verification of $\Omega(n \log n)$ lower bound for average-case time

Lower bound for average-case time

Proposition 117 *The average-case time for comparison-based sorting algorithms on lists of size n is $\Omega(n \log n)$.*

Proof: Note that for any comparison-based sorting algorithm A which takes input lists of size n , the decision tree $DT_A(n)$ is a binary tree with $n!$ leaves and, say, a path length sequence $(l_1, \dots, l_{n!})$. Recall that the average-time of the algorithm A can be expressed as follows:

$$\bar{T}_A(n) = \frac{l_1 + \dots + l_{n!}}{n!}$$

Note that $f(x) = 2^{-x}$ is a convex function. Hence Jensen's inequality implies that

$$2^{-\bar{T}_A(n)} = 2^{-\left(\frac{l_1 + \dots + l_{n!}}{n!}\right)} \leq \frac{2^{-l_1} + \dots + 2^{-l_{n!}}}{n!}$$

By the Kraft inequality

$$2^{-\bar{T}_A(n)} \leq \frac{2^{-l_1} + \dots + 2^{-l_{n!}}}{n!} \leq \frac{1}{n!}$$

Hence

$$2^{-\bar{T}_A(n)} \leq \frac{1}{n!}$$

and

$$\bar{T}_A(n) \geq \log_2(n!)$$

By lemma 93

$$\bar{T}_A(n) \in \Omega(n \log_2 n)$$

□

Exercise 118 *a) Is it possible for a comparison-based sorting algorithm to sort at least half of its inputs of size n in linear worst-case comparison-time? Here, we assume that on the other half of the inputs the algorithm is allowed to run arbitrarily slower. In other words, could we speed up sorting to ensure that at least on half of the inputs (of size n) will take linear worst-case time, while possibly paying the price of raising running times on the other half of the inputs (of size n)? Justify your answer.*

b) Same question for average-case comparison-time.

c) Is it possible for a comparison-based sorting algorithm to sort at least $\frac{1}{n}$ of its inputs of size n in linear worst-case comparison-time? Again, on the other inputs, the algorithm is allowed to run arbitrarily slower. Justify your answer.

d) Same question as in c) but for average-case comparison-time.

e) Is it possible for a comparison-based sorting algorithm to sort $\Omega(2^n)$ of its inputs of size n in linear worst-case comparison-time? On the other inputs, the algorithm is allowed to run arbitrarily slower.

Justify your answer. If you state that such a comparison-based algorithm cannot exist, then prove why it can't exist. If you state that such an algorithm can exist, then provide the pseudo-code for such a sorting algorithm and demonstrate that it runs in linear time on 2^n of its input lists.

Sample solution:

a) The answer is “no”. We know that for a decision tree with K leaves, the longest path length is greater than or equal to $\log_2 K$. Let A be a comparison-based algorithm that takes linear time on half of its inputs of size n . For this algorithm, consider a decision tree representation of all the execution paths of A on each of the $\frac{n!}{2}$ inputs (on which A takes linear time). This tree has $\frac{n!}{2}$ leaves. So the longest path is greater than or equal to $c \log_2(\frac{n!}{2})$ for some constant c (and from a certain value of n onward). Hence, on this collection of inputs (for which A supposedly runs in linear time), the algorithm A must have worst case time $T_A^W(n) \geq c \log_2(\frac{n!}{2}) = c \log_2(n!) - c \log_2(2) = c \log_2(n!) - c \in \Omega(n \log_2 n)$. So the algorithm cannot run in linear time on half of its inputs.

Definition 119 *If a comparison-based algorithm has $O(n \log n)$ worst-case time, we say that the algorithm is optimal in worst-case time and similarly for average-case time.*

The definition is justified by Corollary 99 and Proposition 117.

Exercise 120 *Show that if a comparison-based sorting algorithm is optimal in worst-case time then it must be optimal in average-case time.*

12.8 Non-comparison-based algorithms: Counting Sort

Non-comparison based algorithms can break the lower bounds

If we have additional information on the range of our input values then we can potentially exploit this knowledge to make our algorithm faster and break the $\Omega(n \log n)$ lower bound on the number of comparisons.

Counting sort is an example of an algorithm that is not comparison-based. For this algorithm, we assume that the input lists of size n have elements in a pre-determined range, for instance $\{1, \dots, n\}$.

Counting Sort

Counting Sort(L)

For $i = 1$ to $|L|$ do $L[L[i]] := L[i]$

This algorithm runs in $O(n)$ *assignment-time*⁵, both in worst-case and average-case, which breaks the $\Omega(n \log n)$ lower bound for both measures. No comparisons between list elements are made during the execution of the algorithm. It is not a comparison-based algorithm and exploits a particular property of the input range. General comparison-based algorithms, executed on *all* lists of a given size, cannot rely on input specific knowledge and hence must respect the lower bounds.

Exercise 121 (*Counting sort*)

Adapt the code for counting sort so it can handle the following situation: the input to counting sort consists of a collection of n items, each of which has a non-negative integer key whose maximum value is at most k . Analyze the running time for this version of counting sort.

⁵Assignment-time is a measure that counts the number of assignments carried out during the execution of the algorithm

Chapter 13

Recurrence equations and generating functions

UNDER REVISION

Chapter 14

Entropy and applications

14.1 Probability and entropy

Entropy is a key notion applied in information retrieval and data compression.

- 1) Entropy = a measure of randomness
- 2) Entropy = a measure of compressibility

These two intuitions, as we will motivate in a later section, are linked as follows:

More random = Less compressible

Hence:

High entropy = high randomness/low compressibility
Low entropy = low randomness/high compressibility

Entropy enables one to compute a lower bound on the compressibility of data.

We will illustrate this with a well-known file compression method: the Huffman algorithm. We will compute the Huffman code and measure the compression of the file. This is compared to the “entropy”, a measure of file compressibility obtained from the file (without the need to actually compress).

Entropy is a concept used to predict the average number of bits needed to represent a collection of data, where we know the probability distribution of the data. We make this clear on a basic example. For the case of an 8-sided “dungeons and dragons” fair dice, the “probability events” are the eight outcomes of a throw: the numbers 1 up to 8. Each of these is equally likely, having probability $p = \frac{1}{8}$. To encode n distinct numbers in binary notation we need to use binary numbers of length $\log_2(n)$. Hence to represent the eight possible outcomes for a fair 8-sided dice in binary notation, we need to use binary numbers of length $\log_2 8 = 3$ bits.

Note that the number of outcomes, 8, is also the inverse of the probability $p = \frac{1}{8}$. Hence the number of outcomes can be determined from the probability in this simple

case (where all events have the same probability of occurrence). In other words, the number of bits needed to encode the outcomes can be obtained from the probability p by observing that $\log_2 \frac{1}{p} = 3$.

We are in a similar situation as for the case of constant-time algorithms, where the average-case running time was simply equal to the constant time exhibit by the algorithms. In this case, the average encoding length required for the outcomes of an 8-sided dice is:

$$\begin{aligned} p_1 \log_2 \left(\frac{1}{p_1} \right) + p_2 \log_2 \left(\frac{1}{p_2} \right) + \dots + p_8 \log_2 \left(\frac{1}{p_8} \right) &= 8 \frac{1}{8} \log_2 \left(\frac{1}{8} \right) \\ &= \log_2(8) \\ &= 3 \end{aligned}$$

where $p_i = \frac{1}{8}$ = the probability of the outcome with value $i \in \{1, \dots, 8\}$.

Note that from here on “log” will denote the logarithm in base 2 (in the context of the present section) since we are interested in binary compression only.

In general, we don’t obtain a natural number outcome for the average number of bits needed to encode a number of outcomes. We check this on the fair dice for which p = probability of an outcome = $\frac{1}{6}$.

$$\frac{1}{p} = 1 / \left(\frac{1}{6} \right) = 6, \text{ so } \frac{1}{p} = \text{number of outcomes} = 6$$

$$\log \left(\frac{1}{p} \right) = \log(6) = 2.59$$

$\log \left(\frac{1}{p} \right) = \log(\text{number of outcomes})$ = average number of bits needed to represent the $\frac{1}{p} = 6$ outcomes.

We now generalize our approach to a situation where not all probabilities are equal. For instance, for the case of a biased dice.

Example 122 *Fair and biased dice*

6-sided fair dice

$p(i) = [\text{Probability of outcome} = i] = 1/6$ where i is any number from 1 to 6.

6-sided biased dice

$p(6) = \frac{3}{12} = \frac{1}{4}$ (*6 is more likely*)

$p(1) = \frac{1}{12}$ (*1 is less likely, say someone placed a piece of led in the single “dot”*)

$p(2) = p(3) = p(4) = p(5) = \frac{2}{12} = \frac{1}{6}$

where:

$$p(1) + p(2) + p(3) + p(4) + p(5) + p(6) = 1.$$

For the case of a biased dice, we would still like to measure the average number of bits needed to encode the outcomes. As some outcomes are more likely than others, we can exploit this information to improve our compression to bit-encoding (see Huffman code example in Section 14.2).

Definition 123 *Given probabilities, p_1, \dots, p_n , with sum 1, then we define the entropy H of this distribution as:*

$$H(p_1, \dots, p_n) = p_1 \log\left(\frac{1}{p_1}\right) + p_2 \log\left(\frac{1}{p_2}\right) + \dots + p_n \log\left(\frac{1}{p_n}\right)$$

Note: \log has base 2 in this notation and $\log_2(k) = \frac{\ln(k)}{\ln(2)}$ (where “ln” is “log in base e”)

Here

$$p_i \log\left(\frac{1}{p_i}\right) = (\text{probability of occurrence}) \times (\text{the encoding length})$$

Thus:

$$H(p_1, \dots, p_n) = \text{average encoding length}$$

Exercise 124 *Entropy of dice*

a) *Compute the entropy for the probability distribution of the fair dice in example 122*
Compute the entropy for the probability distribution of the biased dice in example 122

Recall that $p \log\left(\frac{1}{p}\right) = -p \log(p)$. So our entropy can be written as:

$$H(p_1, \dots, p_n) = -p_1 \log(p_1) - p_2 \log(p_2) - \dots - p_n \log(p_n)$$

Solution

a) For a fair dice: $p_1 = p_2 = \dots = p_6 = \frac{1}{6}$. So

$$H(p_1, \dots, p_6) = -\frac{1}{6} \log\left(\frac{1}{6}\right) \times 6 = -\log\left(\frac{1}{6}\right) = -(-\log(6)) = \log(6) = 2.59$$

Interpretation: Entropy measures the amount of randomness. In the case of a fair dice, the randomness is “maximum”. All 6 outcomes are equally likely. This means that to represent the outcomes we will “roughly” need $\log(6) = 2.59$ bits to represent them in binary form (the form compression will take). To encode 6 numbers, we need to use binary numbers of “length” $\log(6)$ (in fact, we need to take the nearest integer above this value, i.e. 3). Binary numbers of length 2 will not suffice (there are only 4 which is not suitable to encode 6 numbers). We keep matters as an approximation and talk about binary numbers of “length” $\log(6)$, even though this is not an integer value.

b) The entropy for the biased dice is:

$$-1/4 \log(\frac{1}{4}) \times 3/20 \log(\frac{3}{20}) \times 5 = \frac{1}{4} \log(4) + \frac{3}{20} \log(\frac{20}{3}) 5 = \frac{1}{4} 2 + \frac{3}{4} \log(\frac{20}{3})$$

Note that:

$$\frac{1}{4} 2 + \frac{3}{4} \log(\frac{20}{3}) = \frac{1}{2} + \frac{3}{4} \log(\frac{20}{3}) = 0.5 + 2.055 = 2.555$$

(Lower value than our previous result)

Exercise 125 *Entropy of 8-sided dice*

Determine the entropy for an 8-sided dice which is a) Fair and b) Totally biased, with $\text{prob}(8) = 1$ (and thus $\text{prob}(1) = \dots = \text{prob}(7) = 0$).

Answers

a) Entropy is $\log(8) = 3$, we need 3 bits to represent the 8 outcomes (maximum randomness)

b) Entropy is $1 \log(1) = 0$, we need a bit of length 0 to represent the outcome. Justify! (Note: bit of length 1 has 2 values. Bit of length 0 has ? value(s)).

Compression for outcomes of fair dice

No compression (we still need 8 values to encode) (Maximum randomness, outcome of entropy/(total number of values) = $8/8 = 1$)

Compression for outcomes of biased dice

Total compression (we only need 1 bit to encode) (No randomness, outcome of entropy/(total number of values) = $0/8 = 0$)

Exercise 126 *Compute the entropy of the collection of random lists of size n , i.e. compute the entropy of the $n!$ lists of size n consisting of non-repeated elements from the set $\{1, \dots, n\}$. Relate the outcome to the lower bound result for the average-case time of comparison-based algorithms.*

Solution:

There are $n!$ such lists of size n . The lists are assumed to be random, occurring with uniform distribution, i.e. the probability for each list is $\frac{1}{n!}$. The entropy of this collection of lists hence is:

$$\sum_{i=1}^{n!} p_i \log_2\left(\frac{1}{p_i}\right) = \sum_{i=1}^{n!} \frac{1}{n!} \log_2(n!) = \log_2(n!)$$

Hence the entropy coincides with the lower bound for the worst-case time and the average-case time of comparison-based algorithms. In fact, for the average-case,

entropy provides a lower bound for the average path length. Because binary trees and prefix codes are essentially the same, we obtain that entropy provides a lower bound for the average-path length of the prefix code associate with binary decision trees, and hence a lower bound for the average code-length. This holds even for the case of a non-uniform distribution as we will see in Section 14.2.2.

14.2 Predicting average compression cost for Huffman codes

Huffman codes are used to compress data of a file. Huffman's greedy algorithm uses a table of frequencies. The frequency denotes how many times a character occurs in the file. Each character will be represented as a binary string.

Goal: Represents file as a binary string of shortest length.

Example: 100,000 characters data-file. Only six different characters occur in the file with frequencies (in thousands)

	a	b	c	d	e	f
	45	13	12	16	9	5

Many encoding exist, e.g.:

A) Fixed length code:

	a	b	c	d	e	f
	000	001	010	011	100	101

B) Variable-length code:

	a	b	c	d	e	f
	0	101	100	111	1101	1100

Case A) Encoding size: 300,000 bits

Case B) Encoding size: 224,000 bits ($45*1 + 13*3 + 12*3 + 16*3 + 9*4 + 5*4 = 224$)

Prefix Codes: Are codes in which no code word is a prefix of another code word.

Prefix codes simplify encoding and decoding (Compression and decompression)

One can show that the optimal compression can always be achieved via a prefix codes.

Example: B) forms a prefix code. *abc* can be represented as:

$$0.101.100 = 0101100$$

(where “.” is the concatenation)

Decoding is unique: (using prefix code table B))

$$0101100 \longrightarrow 0, 101, 100$$

Example of a non-prefix code:

Encoding:	a	b	c
	1	00	001

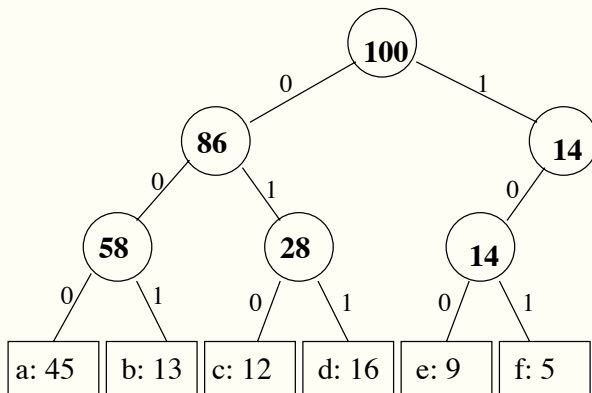
Decoding is not unique as 100001 can be interpreted either as 1,00,001 or as 1,00,00,1 .

Representation of prefix code via binary trees:

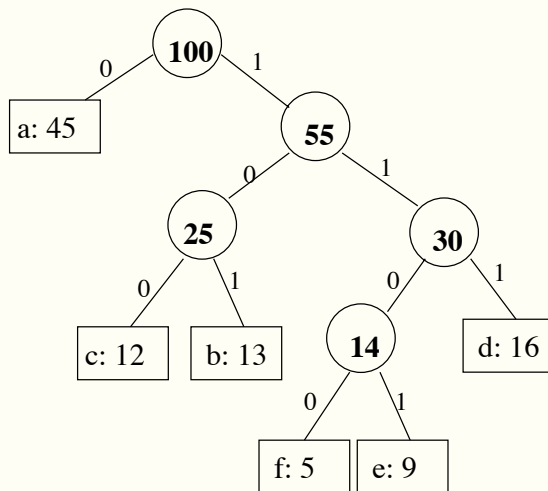
we interpret: $\begin{cases} 0 \longrightarrow \text{go to left child} \\ 1 \longrightarrow \text{go to right child} \end{cases}$

in the tree: Label of node = a number
 Label of leaf = frequency of character

Example A:



Example B:



Let A be a set of characters. The cost of encoding = cost of binary tree representation.

$$\mathcal{C}(T) = \sum_{a \in A} f(a) * d_T(a)$$

where f = frequency, and $d_T(a)$ = depth of leaf at which a occurs (i.e. = number of bits on path from root to leaf a).

Pseudo-Code for Huffman algorithm:

We keep a priority queue Q (keyed on f) to identify the two least-frequent objects to merge together.

The result of merger is a new object whose frequency is the sum of the frequencies of the two merged objects.

HUFFMAN(A)

$n := |A|$

$Q := A$

For $i=1$ To $n-1$ Do

$Z := \text{Allocate-Node}$

$\text{Left}[Z] := \text{Extract-Min}(Q)$

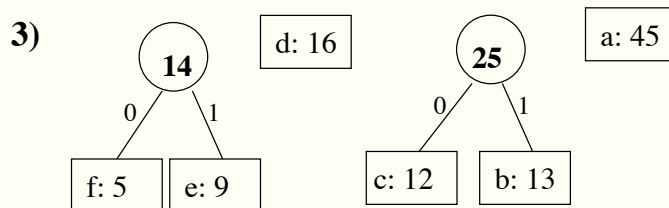
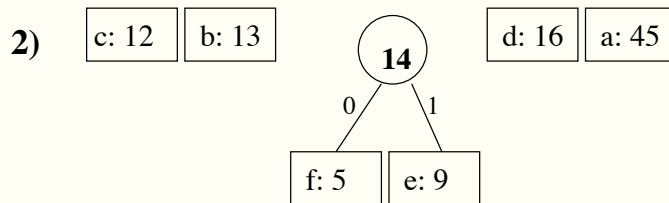
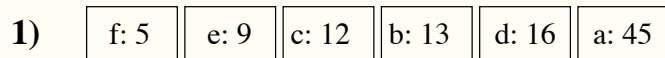
$\text{Right}[Z] := \text{Extract-Min}(Q)$

$f(Z) := f(\text{Left}[Z]) + f(\text{Right}[Z])$

$\text{Insert}(Q, Z)$

FACT: The Huffman algorithm produces optimal prefix codes.

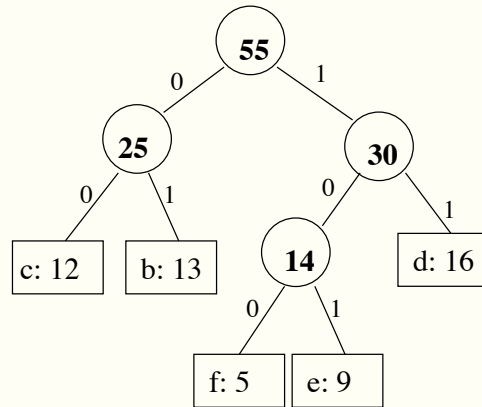
Example:



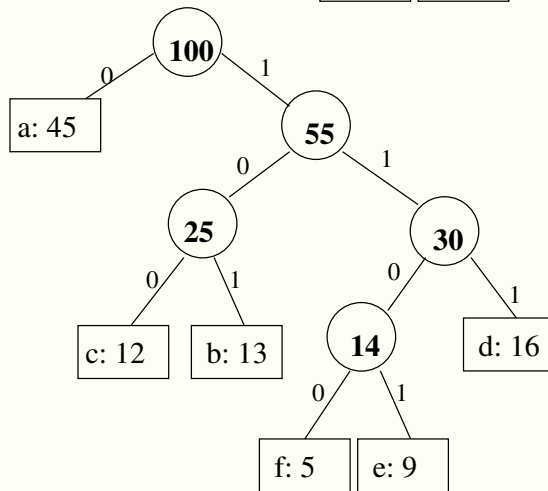
4) We leave part 4 as an exercise.

5)

a: 45



6)



14.2.1 Exercise on Huffman compression

Consider a file with the following properties:

Characters in file: a,b,c,d,e and f

Number of characters: 100,000

Frequencies of characters (in multiples of 1,000):

$$\text{freq}(a) = 45$$

$$\text{freq}(b) = 13$$

$$\text{freq}(c) = 12$$

$$\text{freq}(d) = 16$$

$$\text{freq}(e) = 9$$

$$\text{freq}(f) = 5$$

So the character “a” occurs 45,000 times and similar for the others.

a) Compute the Huffman encoding

b) Compute the cost of the encoding

- c) Compute the average length of the encoding
- d) Express the probability of encountering a character in the file (for each character)
- e) Compute the Entropy
- f) Compare the Entropy to the compression percentage. What is your conclusion?

Solution:

- a) (prefix) codes for characters: $a : 0, b : 101, c : 100, d : 111, e : 1101, f : 1100$.
- b) Cost of encoding = number of bits in encoding =
 $45 \times 1 + 13 \times 3 + 12 \times 3 + 16 \times 3 + 9 \times 4 + 5 \times 4 = 224,000$ bits
- c) $224,000/100,000 = 2.24$ average encoding length
- d) $\text{Prob}(\text{char} = a) = 45/100, \dots, \text{Prob}(\text{char} = f) = 5/100$
 (Note: check that the sum of the probabilities = $100/100 = 1$)
- e) Entropy = $H(45/100, 13/100, 12/100, 16/100, 9/100, 5/100) =$
 $-45/100 \log(45/100) - 13/100 \log(13/100) - 12/100 \log(12/100) - 16/100 \log(16/100) -$
 $9/100 \log(9/100) - 5/100 \log(5/100) = 2.23$.
- f) Conclusion: Entropy is an excellent prediction of average binary encoding length (some minor round-off errors). It predicted the average code length to be 2.23, very close to 2.24. It also predicts total size of compressed file: $2.23 \times 100,000 = 223,000$ which is very close to actual compressed size: 224,000.

14.2.2 Lower bounds revisited

The Huffman algorithm produces prefix-free codes for a given alphabet and frequencies. One can show that the Huffman encoding is close to optimal. We won't do so here, but we will look into what "optimal" means in this context. The above discussion implied that entropy is an excellent prediction of the average binary encoding length. In fact, *entropy is a lower bound for the average encoding length*.

Definition 127 *Given a prefix code corresponding to an alphabet of characters occurring in a given file to be encoded. Each character has a probability of occurrence, say p , namely its frequency in the file divided by the total number of characters in the file. After replacing this character in the file by its binary code number, the binary number also occurs with probability p . Hence, we refer to these as the probabilities of occurrence of the binary numbers of the Huffman encoding. These are once again the frequencies of the code numbers divided by the total number of code numbers in the encoded file.*

Theorem 128 *Given a prefix encoding of a file for which the k binary code numbers have probabilities of occurrence of p_1 up to p_k . The average length of a binary number in the prefix code is bounded below by the entropy. In other words:*

$$\sum_{i=1}^k p_i l_i \geq \sum_{i=1}^k p_i \log_2\left(\frac{1}{p_i}\right)$$

Proof:

$$\begin{aligned} \sum_{i=1}^k p_i \log_2\left(\frac{1}{p_i}\right) - \sum_{i=1}^k p_i l_i &= \sum_{i=1}^k p_i \log_2\left(\frac{1}{p_i}\right) - \sum_{i=1}^k p_i \log_2(2^{l_i}) \\ &= \sum_{i=1}^k p_i \log_2\left(\frac{2^{-l_i}}{p_i}\right) \\ &\leq \log_2\left(\sum_{i=1}^k p_i 2^{-l_i} p_i\right) \\ &= \log_2\left(\sum_{i=1}^k 2^{-l_i}\right) \\ &\leq \log_2(1) \\ &= 0 \end{aligned}$$

The first inequality is an application of Jensen's inequality for the concave function $\log_2(n)$. The second inequality is an application of Kraft's inequality.

□

Remark 129 *Note that binary decision trees reflect the computation paths on all inputs for comparison-based algorithms. Comparison-based algorithms satisfy the separation property, i.e. different input lists in the decision tree result in different paths leading to different leaves. Hence each of the $n!$ inputs corresponds to a unique comparison-path. Each comparison-path can be represented by a binary list (recording zero in case the comparison in question leads to a positive outcome and one otherwise). Hence inputs are in bijection with the binary lists reflecting the decisions made along the computation paths to the output. In other words these paths form an encoding of the $n!$ input lists. Such an encoding must have an average path length bounded below by $\log_2(n!)$, the entropy of the input collection (assuming uniform distribution)—an alternative proof of the fact that comparison-based algorithms take $\Omega(n \log n)$ comparisons on average.*

14.3 Binary subset-encoding revisited

14.3.1 Basic notions

Exercise 130 (*On binary subset-encoding*)

a) Consider the binary subset-encoding encountered in the proof of Theorem 33. Is this coding of the subsets of a finite set the most efficient possible? Use entropy to determine a lower bound on compressibility of all subsets of a finite set. Assume that each subset is equally likely to occur, what is the entropy in this case?

b) Given a finite set A of size $|A| = n$. Assume that we only consider subsets of a given size $k \leq n$. We denote the set of all subsets of size k by $P_k(A)$, the restriction of the powerset to subsets of size k .

Consider the one-element subsets, i.e. the case where $k = 1$. Our binary subset-encoding will use n bits per one element subset, which would be rather wasteful. There are $\binom{n}{1} = n$ one-element subsets. The entropy of the subsets is $\log_2 n$ bits are needed on average to achieve such an encoding, indicating we could improve the encoding. In general, for subsets of size k , the entropy lower bound is $\log_2 \binom{n}{k}$.

Design an encoding that is more tuned to the size of the subsets in the following sense: for each size k find an encoding that will use less than n bits per subset and, ideally, approaching $\log_2 \binom{n}{k}$.

Hint: read the next two sections before solving the exercise.

14.3.2 Ordering finite sets

We will order finite sets based on the following intuition: say you have a pool of soccer players from which you can select teams. Assume teams have 11 members and there are 50 players to choose from, i.e. $\binom{50}{11}$ possible selections. We have a ranking of players, i.e. we know who the best player is, the second best etc. (where we assume there are no ties).

In the absence of knowledge about how the teams will perform, we adopt the following strategy: the “best” team, will be the team that contains “the most top ranked players”. There is a clear winner in that case.

We adopt this idea to rank the teams. Say you have two teams of 11 players, team A and team B . We regard A as better than B in case the highest ranked player which is not shared by both teams belongs to A . In other words, either A has a player that is higher ranked than all of the players of B or A and B share x highest ranked players but the first highest ranked player not in both, belongs to A .

Clearly this is a very crude measure that does not take into account how players will perform when working together in a team context. Also, we could have a situation where A has the highest ranked player while all of its other players are of lowest possible rankings. In that case B could very well have the second best player up to the 11-th best player and hence is likely to beat A . So let's drop the analogy here, and simply use the idea to rank sets of natural numbers.

Given a finite set of numbers, we can rank the numbers through the ranking function discussed in remark 4.2. For instance, consider the set

$$X = \{2, 4, 6, 8, 10, 12, 14, 16, 18, 20\}.$$

Its elements have rank 1 up to rank 10 respectively. The elements in the subset $A = \{2, 4, 14, 18\}$ have ranks 1, 2, 7 and 9 respectively. We order the finite sets according to the same idea as the one adopted for the soccer teams: a finite set A is defined to be larger than B in case the highest ranked number not common to both sets belongs to A . We refer to this order as the *higher-rank-preference order*.

For $A = \{2, 4, 14, 18\}$ and $B = \{2, 6, 18\}$ we obtain that $A > B$ since the highest ranked number not common to both sets is 14 (of rank 7). This ordering on subsets of a finite set is a total order, i.e. for any two subsets A, B of a finite set X , we always have $A \leq B$ or $B \leq A$.

It is useful to consider this order from a binary notation point of view. Recall that our binary subset-encoding can be used to encode subsets of a given (finite) set. So a trivial solution is to encode a subset of k elements from a set of n elements (in order) as a binary string with 1 in position i if the i^{th} element in the set is included, and a 0 otherwise—an inefficient encoding as we have observed.

Instead we define an order on the binary strings as follows:

Binary version of higher-rank-preference order

Definition 131 *string $A >$ string B if, in the **last** position in which they differ, string A has a 1 and string B has a 0.*

Example 132 $1010 > 1100$, because the last position where they differ is the third position and the first string has a 1.

Exercise: a) Consider the powerset of the set $X = \{1, 2, 3, 4\}$. $P(X)$ has $2^4 = 16$ elements. Consider the 3-element subsets among these: $A_1 = \{1, 2, 3\}$, $A_2 = \{1, 2, 4\}$, $A_3 = \{2, 3, 4\}$, $A_4 = \{1, 3, 4\}$. Determine the higher-rank-preference order among these subsets.

b) Use the binary version of the order and derive the correct ordering based on definition 131.

We will assign an index to each subset (of a given size k) through a neat combinatorial trick.

14.3.3 Subset-indexing lemma

Given a set X of size n , the subset-indexing lemma is a useful result that will allow us to assign an index value to each subset of size $k \leq n$ such that the smallest subset in

the higher-rank-preference order receives value 0, the second smallest subset in this order receives index 1, up to the largest subset in the higher-rank-subset-order, which receives index $\binom{n}{k} - 1$. In this way, each subset in the given order is ranked by an index.

Lemma 133 *Given two natural numbers k and n , $k \in [0, n]$, the function S defined by:*

$$S(\{x_i\}_{i=1}^k) = \sum_{i=1}^k \binom{x_i - 1}{i},$$

takes on all values between 0 and $\binom{n}{k} - 1$ as $\{x_i\}_{i=1}^k$ ranges over all k -element subsets of the set $\{1, \dots, n\}$ of the first n natural numbers (where the x_i are listed in increasing order).

Note that the lemma implies that the function S is an injection.

Before proving the lemma, let's have a closer look at the expressions $\binom{x_i - 1}{i}$ in the subset-indexing lemma. The expression $\binom{x_i - 1}{i}$ counts all subsets smaller than a given set (in the higher-rank-subset-order).

Say we consider a set X of n elements, with respective ranks $1, \dots, n$. Consider a subset A of X containing k elements with respective ranks: x_1, \dots, x_k (listed in increasing order). Now, consider a subset B smaller than the subset A (smaller in the higher-rank-subset-order). Since B is smaller than A , we know that A contains an element of rank say x_i that is not in B and all elements of rank greater than x_i belong to both A and B . There is no restriction on the elements of B that are smaller than x_i . There are $x_i - 1$ of such elements to pick from. Also, the set B must have size k . We know that B does not contain element of rank x_i but it contains the elements of ranks $x_i + 1, \dots, x_k$, all of which it shares with A . Hence there are p more elements to pick to ensure B has k elements. This means there are $\binom{x_i - 1}{i}$ possible choices for a set B smaller than A . That means that the total number of subsets B smaller than A is $\sum_{i=1}^k \binom{x_i - 1}{i}$. This sum serves as an index for the set A . It uniquely specifies the set A as A is determined by all subsets strictly smaller than it.

Let's look at the same argument in our binary encoding interpretation:

The intuition for indexing of subsets is as follows:

- The index of any subset is equal to the number of k -element subsets with binary codes less than it in this order - we show that this number can be calculated with the formula in Lemma 133.
- In order to be less than the given string, another string must have a 0 where the given string has a 1. Suppose the last occurrence of this is at the p^{th} 1 in the given string, in position x_p . Then the strings must match for all positions $> x_p$, and the smaller string must have p 1s in the first $x_p - 1$ positions, for which there are $\binom{x_p - 1}{p}$ possibilities.

- But since every smaller string must match this *pattern* for some p , and since these patterns are mutually exclusive (as they each match a different number of 1s at the end of the given string), it follows that the number of smaller strings, and hence the index, is given by $\sum_{i=1}^k \binom{x_i-1}{i}$ as required.

Exercise 134 Consider the set $X = \{1, 2, 3, 4, 5\}$. Display the $\binom{5}{3} = 10$ subsets of size 3 listed in a column according to the higher-ranking-preference order, starting with the set $\{1, 2, 3\}$ at the bottom and $\{3, 4, 5\}$ at the top. For each subset, provide its binary subset-encoding next to it. This creates a second column. For this column, group the binary numbers according to “the pattern” discussed above. Write the correct corresponding binomial coefficient $\binom{x_p-1}{p}$ next to each group in this pattern.

The fact that $\sum_{i=1}^k \binom{x_i-1}{i}$ enumerates precisely the values between 0 and $\binom{n}{k} - 1$ (as $\{x_i\}_{i=1}^k$ ranges over all k -element subsets of the first n positive integers, and the x_i are in increasing order) is shown via the following proof.

Proof: (of Lemma 133)

Note that if $k = n$, then the result holds. If $k = n$, then $\{x_1, \dots, x_k\} = \{x_1, \dots, x_n\} = \{1, \dots, n\}$, i.e. $x_i = i$ for each $i \in \{1, \dots, n\}$. Hence there is only one choice for a subset of size n and $\sum_{i=1}^n \binom{x_i-1}{i} = \sum_{i=1}^n \binom{i-1}{i} = 0$. This coincides with the value between 0 and $\binom{n}{n} - 1 = 0$, namely 0.

So we can assume that $k < n$ in the following.

Note that if $n = 1$, the result also holds. Since $k < n$, we must have $k = 0$. There is only one 0-element subset, the empty set, which gives $S = 0$ (there is nothing to sum over, resulting in zero), which again coincides with the value between 0 and $\binom{1}{0} - 1 = 0$.

So we can assume that $k < n$, where $k, n \geq 1$. We prove the result by induction on the value of n .

If $n = 2$, then either $x_1 = 1$ or $x_1 = 2$, so that $S = 0$ or $S = 1$, and the result is true.

Now, suppose that the result is true whenever $n \geq 2$, and consider a set of size $n + 1$. We partition the k -element subsets of the first $n + 1$ integers into two groups according to whether or not they contain $n + 1$. If a subset does not contain $n + 1$, then it is an k -element subset of the first n integers, and by the inductive hypothesis the range of corresponding values of S is $[0, \binom{n}{k} - 1]$

If the set does include $n + 1$ then clearly $x_k = n + 1$, so that $\binom{x_k-1}{k} = \binom{n}{k}$. But now, the remainder of the set is a $k - 1$ -element subset of the first n positive integers, so by the inductive hypothesis $\sum_{i=1}^{k-1} \binom{x_i-1}{i}$ must take on all integer values in $[0, \binom{n}{k-1} - 1]$. Adding the last term to the sum, we find that the range of S for the subsets in this group is $[\binom{n}{k}, \binom{n}{k-1} + \binom{n}{k} - 1]$.

Combining the two ranges, we see that S takes on all integer values in $[0, \binom{n+1}{k} - 1]$ as required. By induction, the result holds for all values of n .

□

14.3.4 Subset-index encoding

Entropy guided us to conclude that we could design a more efficient encoding for subsets of fixed size. The above approach provides one such encoding, allowing us to index all subset of size k with values between 0 and $\binom{n}{k}$. For a given subset A of a set X , where A has size $k \leq |X| = n$, we can encode A by applying the function S specified in Lemma 133 to the *ranks* x_1, \dots, x_k of the elements of A (with respect to X and listed in increasing order).

Of course, the potential for such an indexing was clear from the outset, since there are $\binom{n}{k}$ subsets of size k . Hence, assuming we have a uniform distribution of subsets of size k , i.e. all subsets are equally likely to occur, entropy informs us that we only need $\log_2 \binom{n}{k}$ bits to encode the k -element subsets of a given set to size n .

The nice part of the subset-indexing encoding is that, as for the case of the binary subset-encoding, we have the ability to decode.

14.3.5 Subset-index decoding

Exercise 135 (*Subset index decoding*)

Specify a subset-indexing decoding. To start, you might want to consider an example. For instance, consider the set $X = \{1, 2, 3, 4\}$. The subset-indexing encoding grants each subset of size 3 an index value between 0 and $\binom{4}{3} = 4$. Say you are given the information that a subset of size 3 has index 2. How can you determine the subset from the index value 2? A solution is provided in Appendix 2. Of course, don't look at it until you've given it a serious try.

14.4 Machine learning

Time allowing, we will discuss a basic application of entropy in machine learning, where we will refer to the additional course documentation on the class website and where you need to take notes in class.

Chapter 15

Computational tractability

In der Beschränkung zeigt sich erst
der Meister.

J.W. von Goethe (1802))

This section will be covered pending available time. We will introduce a class of “computationally tractable data structures” consisting of the so-called “labelled series-parallel partial orders”. We will show that the computations over such structures are computationally very feasible (i.e. efficient algorithms typically exists). Class notes need to be taken for this section. Topics will include:

15.1 Series-parallel partial orders

We will introduce these inductively defined structures, formed from two basic operations: the series and the parallel operator. A structural characterisation can be achieved in terms of N-freeness.

15.2 Labeled partial orders & counting labelings

Labeled series parallel orders serve to represent well-known data structures. We will demonstrate some basic algorithms over these structures, illustrating their computation tractability.

Chapter 16

Computational intractability

Experience is a hard teacher
because she gives the test first, the
lesson afterward.

Anonymous

We have shown that some problems are non-computable, i.e. there is no program which computes them, as illustrated by the Halting Problem. Is this the end of the story? Is it sufficient to know that some problems are computable and others are not?

If we think back to the problems we considered so far, mainly sorting and search algorithms, most of the algorithms ran in $O(n^2)$ or $O(n \log n)$ time. Some algorithms, such as the algorithm computing a powerset, run in exponential time.

This brings us to the consideration of the computable problems in a more refined way. Indeed, once we know a problem is computable, the next question is whether a program which computes it is actually efficient.

One way of capturing efficiency is to focus on algorithms which are polynomial time computable.

16.1 Polynomial time computable algorithms



Polynomial time computability

A polynomial time computable algorithm is an algorithm for which there exist constants c and d such that their “running time” is bounded on every input of size n by cn^d .

Polynomial time algorithms are considered to be “reasonably efficient” algorithms since they satisfy a better scaling problems than say exponential time algorithms.

For instance, in case the input size n grows by a factor of 2 to $2n$, a polynomial time algorithm, say with upper bound cn^d for inputs of size n , will slow down only

by a constant factor: $c(2n)^d = c2^d n^d$. Hence this algorithm now runs again in $O(n^d)$ time, but with different constant factor, i.e. $c2^d$ instead of c .

For exponential time algorithms, this does not hold. Consider for instance an algorithm which runs in $c3^n$ time for inputs of size n . If the input size doubles, then we obtain $c3^{2n} = c9^n = c3^n 3^n$. Hence the new constant factor is $c3^n$ which depends on n and regards an exponential increase.

To obtain some intuition on the difference between the two cases, consider the following table which gives the running times (rounded) for different algorithms on inputs of increasing size. We consider a processor performing a million high-level instructions per second. In cases where the running time exceeds 10^{25} years, the algorithm is recorded as Out of Bounds (OoB).

Table 16.1:

input size	n	$n \log_2 n$	n^2	n^3	1.5^n	2^n	$n!$
n = 10	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	4 sec
n = 30	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	18 min	10^{25} years
n = 50	< 1 sec	< 1 sec	< 1 sec	< 1 sec	11 min	36 years	OoB
n = 100	< 1 sec	< 1 sec	< 1 sec	1 sec	12,892 years	10^{17} years	OoB
n = 1,000	< 1 sec	< 1 sec	1 sec	18 min	OoB	OoB	OoB
n = 10,000	1 sec	< 1 sec	2	12 days	OoB	OoB	OoB
n = 100,000	< 1 sec	2 sec	3 hours	32 years	OoB	OoB	OoB
n = 1,000,000	1 sec	20 sec	12 days	37,710 years	OoB	OoB	OoB

16.2 Polynomial time exercises

We discussed polynomial running time. A program with execution time $T(n)$ on an input of size n is said to run in polynomial time if we have $T(n) \leq cn^d$ for some positive integers c and d but for all values of n . The function $T(n)$ is said to have polynomial growth.

Example 1: $T_1(n) = 3n^2$

Clearly T_1 has polynomial growth. Just let $c = 3$ and $d = 2$.

Example 2: $T_2(n) = \alpha n^2 + \beta n + \gamma$

Since the size of an input will always be at least 1, we have $1 \leq n \leq n^2 \leq n^3 \leq \dots$. So we have $T_2(n) = \alpha n^2 + \beta n + \gamma \leq |\alpha|n^2 + |\beta|n^2 + |\gamma|n^2 = (|\alpha| + |\beta| + |\gamma|)n^2$, and so T_2 clearly has polynomial growth, with $c = |\alpha| + |\beta| + |\gamma|$ and $d = 2$. More generally, any polynomial $p(n)$ (i.e. a sum of integer powers of n) can be shown in the same way to have polynomial growth if we let c be the sum of the absolute values of the coefficients and d be the highest power of n in the polynomial. This is where the name “polynomial growth” derives from. Can you see why the absolute value (i.e.

$|\alpha|$ instead of α) is necessary? Otherwise, we might have a value which was bigger in size, but negative. For example, if $n = 3$ and $\beta = -4$, then we do not have $\beta n \leq \beta n^2$, since $\beta n = -12$, but $\beta n^2 = -36$. On the other hand, we do have $\beta n \leq |\beta|n^2$.

Example 3: $T_3(n) = \sqrt{n} \log_2 n$

Again, because we have $1 \leq n$, we have $1 \leq \sqrt{n}$ and hence $\sqrt{n} \leq n$. You also know that $\log_2 n \leq n$. Combining the two of these, we have $T_3(n) = \sqrt{n} \log_2 n \leq n \cdot n = n^2$, and so T_3 also has polynomial growth (even though it is not itself a polynomial).

Example 4: $T_4(n) = 2^n$

T_4 does not have polynomial growth—in fact, for any choices of c and d , we will have $2^n > cn^d$ for n sufficiently large.

16.3 NP-complete problems

The P versus NP problem is a main unsolved problem in computer science. It asks whether every problem whose solution can be efficiently checked by a computer can also be efficiently solved by a computer. The problem was introduced in 1971 by Stephen Cook in his paper “*The complexity of theorem proving procedures*”. The P versus NP problem is one of the Millennium Prize Problems selected by the Clay Mathematics Institute and offers a US\$ 1,000,000 prize for a correct solution. We will study the problem in detail, relying on the excellent introduction in the book *Algorithm Design* by Kleinberg and Tardos.

We will study NP-complete problems—computationally intractable problems. Such problems form a string of problems arranged in a “daisy chain”. Finding a polynomial time algorithm that computes one of these problems means that each of the problems can be solved by a polynomial time algorithm derived from the original solution—“one ring to bind them all”. No one at the time of this writing knows of a solution. NP-complete problems “skate too close” to exponential time (cf. Section 4.4.1).

Solving one of these problems in polynomial time will, like a domino-effect, lead to a polynomial time solution for each problem in the collection. The idea underlying this domino-effect is the “reduction of one problem to another”. Key to many of such reductions is the capacity to transform a problem into another through a *duality principle* as covered in Chapter 11.

We will cover the following topics:

Chapter 8: NP and Computational Intractability, Sections 8.1 - 8.4

Of these: you only need to know in detail:

- **Section 8.1:** The definition of *polynomial time reducibility* and exercises.

- **Section 8.2:** *Reduction via gadgets*
- **Section 8.3:** *Circuit satisfiability*
The 3-SAT problem, but not the reduction proof to independent set
- **Section 11.3:** *Set Cover: A General Greedy Heuristic* (Approximation Algorithm). You need to know the algorithm, but not the proof.
- **Section 8.4:** definition of an *NP-complete problem*. You need to know some *examples of NP-complete problems* (including the ones presented in 8.1 and a few extra ones).
- **Additional topic:** We may also cover a distributed algorithm to compute *Maximal Independent Set* as another type of approximation algorithm (using randomization). You need to know the algorithm and the proof.

Chapter 17

Appendix 1: Useful maths facts

17.0.1 The triangle equality

Here we show that:

$$\sum_{i=n-1}^1 i = 1 + 2 + 3 + \dots + n - 1 = \frac{((n-1) + 1)(n-1)}{2} = \frac{n(n-1)}{2}.$$

Adding up consecutive numbers can be represented by a triangle - for instance, $1 + 2 + 3 + 4 + 5$ (case: $n = 6$):

.
.
.
.
.
.

The addition $1 + 2 + 3 + 4 + 5$ corresponds to the number of dots in the above triangle. Adding up a sequence of numbers (or dots in a triangle) is tedious. So we apply a simple trick to speed up the work, by adding another (rotated) copy of the same triangle to get a rectangle:

.
.
.
.
.
.

Note that the number of dots in rectangle is twice number of dots in triangle. So to obtain the number of dots in the triangle, we only need to half the number of dots in the rectangle.

The number of dots in a rectangle is “base \times height”, i.e. $= 5(5 + 1)$

So number of dots in triangle $= \frac{5(5+1)}{2} = \frac{6.5}{2} = \frac{30}{2} = 15$.

In general (same trick): $1 + 2 + 3 + \dots + k = \frac{(k+1)k}{2}$,
so (taking $k = n - 1$) we have:

$$1 + 2 + 3 + \dots + n - 1 = \frac{((n - 1) + 1)(n - 1)}{2} = \frac{n(n - 1)}{2}.$$

17.0.2 Pascal’s triangle

Pascal’s triangle is a triangular arrangement of the binomial coefficients, illustrating the recursive computation of these coefficients. The triangle display is named after French mathematician Blaise Pascal. Other mathematicians studied it centuries before him in India, Persia (Iran), China, Germany, and Italy.

We recall Pascal’s triangle:

$$\begin{array}{ccccccc}
 & & & & & & 1 \\
 & & & & & 1 & 1 \\
 & & & 1 & 2 & 1 \\
 & & 1 & 3 & 3 & 1 \\
 & 1 & 4 & 6 & 4 & 1 \\
 1 & 5 & 10 & 10 & 5 & 1 \\
 1 & 6 & 15 & 20 & 15 & 6 & 1 \\
 1 & 7 & 21 & 35 & 35 & 21 & 7 & 1
 \end{array}$$

The triangle displays the binomial coefficients:

$$\begin{array}{ccccccc}
& & & & \binom{0}{0} & & \\
& & & & & & \\
& & & \binom{1}{0} & \binom{1}{1} & & \\
& & & & & & \\
& & \binom{2}{0} & \binom{2}{1} & \binom{2}{2} & & \\
& & & & & & \\
& \binom{3}{0} & \binom{3}{1} & \binom{3}{2} & \binom{3}{3} & & \\
& & & & & & \\
& \binom{4}{0} & \binom{4}{1} & \binom{4}{2} & \binom{4}{3} & \binom{4}{4} & \\
& & & & & & \\
& \binom{5}{0} & \binom{5}{1} & \binom{5}{2} & \binom{5}{3} & \binom{5}{4} & \binom{5}{5} \\
& & & & & & \\
& \binom{6}{0} & \binom{6}{1} & \binom{6}{2} & \binom{6}{3} & \binom{6}{4} & \binom{6}{5} & \binom{6}{6} \\
& & & & & & \\
& \binom{7}{0} & \binom{7}{1} & \binom{7}{2} & \binom{7}{3} & \binom{7}{4} & \binom{7}{5} & \binom{7}{6} & \binom{7}{7}
\end{array}$$

17.0.3 Telescopic sums

We already have seen that the triangle equality arises in analyzing the time of basic sorting algorithms, such as e.g. Bubble Sort. Such equalities are useful to help us obtain the running time. Another useful equality is the telescopic sum equation.

Ancient types of telescope consisted of parts that could be “shoved into one another” to yield a shorter piece of equipment. The sum is called telescopic since its terms cancel out to yield a shorter expression.

Telescopic sum equation:

$$\sum_{i=0}^{k-1} p^i = \frac{p^k - 1}{p - 1}, \text{ on condition } p \neq 1.$$

Note that p cannot be 1 in this equation to avoid division by $p - 1 = 0$. In case $p = 1$

$$\sum_{i=0}^{k-1} p^i = \sum_{i=0}^{k-1} 1^i = k$$

To verify the telescopic sum equation, note that:

$$\begin{aligned}
\left(\sum_{i=0}^{k-1} p^i\right)(p - 1) &= p + p^2 + p^3 + p^4 + \dots + p^{k-1} + p^k - 1 - p - p^2 - p^3 - p^4 - \dots - p^{k-1} \\
&= p^k - 1
\end{aligned}$$

Division by $(p - 1)$, where $p \neq 1$, yields

$$\sum_{i=0}^{k-1} p^i = \frac{p^k - 1}{p - 1}$$

Exercise Reduce

$$\sum_{i=0}^{18} 5^i$$

to a fraction.

Answer: $\sum_{i=0}^{18} 5^i = \frac{5^{19}-1}{5-1} = \frac{5^{19}-1}{4}$.

17.0.4 Hockey stick lemma

The Hockey stick lemma is a well-known mathematical identity¹, for which we give the standard inductive proof. This lemma is known as the hockey stick (or Christmas stocking) lemma/theorem/identity because of the shape created by highlighting all the terms in the identity in Pascal's triangle — the sum on the left-hand side forms a line parallel to one of the sides of the triangle, and the term on the right-hand side is below and to the right of the last of these, completing a 'hockey stick' shape.

Lemma 136 $\sum_{k=i}^n \binom{k}{i} = \binom{n+1}{i+1}$

We display the Hockey stick lemma as an explicit sum and provide an obvious equivalent formulation:

$$\begin{aligned} \binom{i}{i} + \binom{i+1}{i} + \binom{i+2}{i} + \cdots \binom{n}{i} &= \binom{n+1}{i+1} \\ \binom{i}{0} + \binom{i+1}{1} + \binom{i+2}{2} + \cdots \binom{n}{n} &= \binom{n+1}{i+1} \end{aligned}$$

Why does the Hockey Stick lemma hold? Imagine you order six pencils from small to large (where none have equal size). Say you have six pencils and you pick three of these in a random selection. There are $\binom{6}{3} = 20$ ways to do this. This is the number displayed in the right hand side of the hockey stick equality (for $n+1 = 6, i+1 = 3$). Now say we form the selections in a different way. This time, for each group of three pencils, we start by selecting a smallest pencil of that group. As for each smallest pencil, we need to pick two larger ones, we can only pick the smallest pencil of the six up to the third largest pencil of the six (i.e. the last two largest pencils can't be chosen). For the smallest pencil among the six, we can pick $\binom{6}{2}$ sets of two larger ones, for the second smallest pencil, we can pick $\binom{5}{2}$ larger ones, and so on, until the third largest pencil for which we only have one remaining choice, pick the two largest ones (among the two largest ones), i.e. $\binom{2}{2}$. As each selection of three pencils must

¹For instance: Paul Zeitz, *The Art and Craft of Problem Solving*, Wiley, 2006

have a smallest pencil, we generate all possible collections of 3 pencils in this way (specifying the smallest first, then picking two larger ones). Hence:

$$\binom{6}{2} + \binom{5}{2} + \binom{4}{2} + \binom{3}{2} + \binom{2}{2} = \binom{7}{3},$$

which is precisely the hockey stick identity.

To see why this lemma is called the hockey stick identity, highlight the numbers displayed on both sides of the identity in Pascale's triangle. The result has a hockey stick shape.

The expression has an obvious equivalent statement:

$$\binom{6}{4} + \binom{5}{3} + \binom{4}{2} + \binom{3}{1} + \binom{2}{0} = \binom{7}{3}$$

Once more, highlighting all terms in this equation, reveals a mirror-image of a hockey stick in Pascal's triangle.

We can prove the hockey stick identity by generalising the argument given above (do this as an exercise) or by showing the result via a proof by induction as below.

Proof: If $n < i$, then all of the terms are zero and the result is trivial. We prove the result for $n \geq i$ by induction. For $n = i$, we have $\binom{i}{i} = \binom{i+1}{i+1}$, which is true since $\binom{m}{m} = 1$ for all m . So, we assume that the result is true for $n = m \geq i$. Then we have

$$\sum_{k=i}^{m+1} \binom{k}{i} = \binom{m+1}{i} + \sum_{k=i}^m \binom{k}{i} = \binom{m+1}{i} + \binom{m+1}{i+1} = \binom{m+2}{i+2},$$

so that the result holds for $n = m + 1$, and hence by induction for all integers $n \geq i$.

□

Chapter 18

Appendix 2: Solutions to selected exercises

Cardinality exercises

1) Given two finite sets A and B and an injective function $f: A \rightarrow B$. Verify that $|A| \leq |B|$, i.e. B must have at least as many elements as A .

2) Two finite sets A and B have the same cardinality $|A| = |B|$ if and only if there exists a bijection $f: A \rightarrow B$.

Answers:

1) Say f is an injective function $f: A \rightarrow B$. Note that since f is an injection, it is a function satisfying: $\forall x, y \in A. x \neq y \Rightarrow f(x) \neq f(y)$. Hence if A has k distinct elements, the image $f(A) = \{f(a) | a \in A\}$ has k elements. Since $f(A) \subseteq B$, we obtain that: $|A| = |f(A)| \leq |B|$.

2) Note that a bijection $f: A \rightarrow B$ is both an injection and a surjection. As f is an injection, we know that $|A| \leq |B|$. Hence it suffices to show that for any surjection $g: A \rightarrow B$, the following holds: $|A| \geq |B|$ (hence: $|A| = |B|$).

Assume that $f: A \rightarrow B$ is a surjection. In that case each element of B is the image of some element in A . Say B has l elements, b_1, \dots, b_l , then there exist elements $a_1, \dots, a_l \in A$ such that $\forall i \in \{1, \dots, l\}. f(a_i) = b_i$. Note that the elements a_1, \dots, a_l are pairwise distinct, i.e. $i \neq j \Rightarrow a_i \neq a_j$ (since f is a function it is impossible for a_i and a_j to coincide when $i \neq j$, otherwise $f(a_i) = b_i$ and $f(a_i) = b_j$, where $b_i \neq b_j$, which is impossible as f is a function (one value cannot have two images). Hence there are at least l elements in A , i.e. $|A| \geq |B|$.

Countability exercises

Answer for exercise 29

1) Show that every subset of a countable set must be countable.

Answer: Say A is a countable set and $B \subseteq A$. A is countable, hence there exists an injection $f: A \rightarrow \mathbf{N}$. B is a subset of A . Let the function f_B be the restriction of the function f to the set B (i.e. f_B is the function consisting of all pairs (x, y) of f for which $x \in B$). The restricted function f_B is still an injection and maps the set B to the set \mathbf{N} . Hence B is countable.

2) Given two infinite sets A and B , where A is countable. Show that if there exists a bijection from A to B then B must be countable.

Answer: A is infinite and countable, so there exists a bijection $f: \mathbf{N} \rightarrow A$. By assumption, there exists a bijection from $h: A \rightarrow B$, so the composition function $h \circ f$ maps \mathbf{N} to B . This composition function is a bijection (as both f and h are bijections) and we know that B is an infinite set. Hence B is countable.

3) Display (but do not compute) the Gödel encoding for the final line in the pseudo-code of the algorithm Bubblesort (where L is the input list and $|L|$ denotes the length of L). You may assume that each symbol s in the code is encoded by a positive integer $c(s) \in \mathbf{N}$.

- Each lowercase letter of the alphabet receives its number in the alphabetical order, i.e. $c(a) = 1, \dots, c(z) = 26$
- Upper case letters receive the numbers $27, \dots, 52$ respectively, i.e. $c(A) = 27, \dots, c(Z) = 52$
- We use 0 as a separation symbol (for blanks)
- 53 denotes $|$, 54 denotes $>$, 55 denotes $[$, 56 denotes $]$, 57 denotes $($, 58 denotes $)$, 59 denotes $+$, 60 denotes $-$ and 61 denotes the equality sign $=$
- Finally, 62 denotes 1 and 63 denotes the comma $,$

This is sufficient to encode Bubble Sort. Obviously, for a general language (including the one specifying pseudo code) we would need a code for *each* symbol of the language).

To solve this exercise, use the table below listing the first 63 prime numbers.

2	3	5	7	11	13	17	19	23	29
31	37	41	43	47	53	59	61	67	71
73	79	83	89	97	101	103	107	109	113
127	131	137	139	149	151	157	163	167	173
179	181	191	193	197	199	211	223	227	229
233	239	241	251	257	263	269	271	277	281
283	293	241							

Display the Gödel encoding for the final line of the pseudo-code of Bubblesort, namely: **swap(L[j],L[j+1])**

Bubblesort(L)

```

for i = |L| - 1 downto 1 do
  for j = 1 to i do
    if L[j] > L[j + 1] then
      swap(L[j], L[j + 1])

```

The last line starts with the word “swap”. The encoding c for each of these letters is: $c(s) = 19, c(w) = 23, c(a) = 1, c(p) = 16$. The next symbols are: $(L[j], L[j + 1])$, which are encoded as follows: $c(()) = 57, c(L) = 38, c([]) = 55, c(j) = 10, c([]) = 56, c(,) = 63, c(L) = 38, c([]) = 55, c(j) = 10, c(+) = 59, c(1) = 62, c([]) = 56, c()) = 58$

Hence the Gödel encoding of **swap(L[j], L[j + 1])** is

$$2^{19}3^{23}5^17^{16}11^{57}13^{38}17^{55}19^{10}23^{56}29^{63}31^{38}37^{55}41^{10}43^{59}47^{62}53^{56}59^{58}.$$

5) Let A be a countable set and suppose that there exists a surjective function $f: A \rightarrow B$. Prove that B is also countable.

Answer: For each element b in the set B , note that $f^{-1}(b) \subseteq A$ (where $f^{-1}(b)$ is the set of all elements of A that are mapped to the element b by the function f). Since A is countable, there exists an injection $g: A \rightarrow \mathbf{N}$. Apply this injection g to the elements of the set $f^{-1}(b)$. This yields a subset of \mathbf{N} , namely $g(f^{-1}(b)) \subseteq \mathbf{N}$. Let x_b be the smallest natural number in this set $g(f^{-1}(b))$. Define the function $h: B \rightarrow \mathbf{N}$ to be the function that maps each element b of B to x_b .

h is an injection. Indeed, we show that if $h(b_1) = h(b_2)$ then $b_1 = b_2$ (this is enough to show that h is an injection, since it shows that if $b_1 \neq b_2$, we must have that $h(b_1) \neq h(b_2)$).

So, say $h(b_1) = h(b_2)$, then, by definition of h , we have: $h(b_1) = x_{b_1} = x_{b_2} = h(b_2)$. This means that $g(f^{-1}(b_1))$ and $g(f^{-1}(b_2))$ share a common element, namely $x_{b_1} = x_{b_2}$. Since g is an injection, we know that $g^{-1}(x_{b_1})$ is a single element of the set $f^{-1}(b_1)$, say the element a_1 . The same holds for $g^{-1}(x_{b_2})$. It is a single element, say a_2 of $f^{-1}(b_2)$. But $x_{b_1} = x_{b_2}$, hence $a_1 = a_2$. Since $a_1 \in f^{-1}(b_1)$ and $a_2 \in f^{-1}(b_2)$, we

know that $f(a_1) = b_1$ and $f(a_2) = b_2$. Since $a_1 = a_2$ we must have $b_1 = b_2$.

6) Show that the set of \mathbf{N} of natural numbers can be represented as a union of an infinite number of disjoint *infinite* sets.

Answer: There are several ways to prove this result. One way is to consider the binary representations of the natural numbers. Each natural number n can be written as a binary number $b(n)$. Now, define the set A_k to be the set of all binary numbers that have k ones. Clearly $A_0 = \{0\}$, the set containing the zero bit. The set A_1 consists of all binary numbers that have a single one, i.e. $A_1 = \{1, 10, 100, 1000, 10000, \dots\}$. The set A_2 consist of the binary numbers containing exactly two ones, i.e. $A_2 = \{11, 110, 101, 1100, 1010, 1001, 11000, \dots\}$ and so on. Clearly, every set A_k is infinite. Also, for each k, l where $k \neq l$, we have $A_k \cap A_l = \emptyset$, i.e. the sets A_k are pairwise disjoint. And clearly their (infinite) union $\cup_{k \in \mathbf{N}} A_k$ forms all the binary representations of the set \mathbf{N} . Converting all the binary representations to the natural number ones in each of these sets, gives a representation of the set \mathbf{N} as an infinite union of disjoint infinite sets.

Halting problem exercises

Problem 1: A programmer writes programs P_i for $i = 1, 2, 3, \dots$ which generate finite sequences of integers as their output in case they terminate, and intends to call a program Q on each integer of the output. The programmer knows that the running time of Q is very small except on the digit 33, where it is very large.

So, to estimate the average efficiency of his program Q , he decides to write another program \mathcal{T} , which will take a program P as an input, and output the total number of 33s outputted by P .

Is it possible to write such a program \mathcal{T} ? (You may assume that the programs P_i do not take any input.)

Solution: No.

If such a program \mathcal{T} could be constructed, then it could be used as a Halting Problem solver which could determine whether a given program halted on a given input or not. This is known to be impossible.

In fact, we have seen in the exercises of Chapter 6 that if a program can even determine whether a given program outputs one 33 (not to mention the exact number), this is sufficient to solve the Halting Problem.

Problem 2: Suppose that the programmer is worried about the program Q running

for too long, and decides to prevent this by automatically terminating each program P after a finite running time t on some given machine (if it has not terminated already), and feeding whatever output has already been produced to Q .

Under this restricted condition, is it possible to write the program \mathcal{T} ?

Solution: Yes.

When a program can only run for a limited time, the Halting Problem undecidability does not apply. If we simply write a program \mathcal{T} which simulates the running of its input program for a time t and counts each 33 outputted in that time, this will give us the total required.

Problem 3: Suppose instead that the programmer decides to prevent overly long execution times for Q by automatically terminating each program P when it has output ten 33s (if it has not terminated already).

Under this restricted condition, is it possible to write the program \mathcal{T} ?

Solution: No.

Although this also restricts the program P , it does not limit it to finite resources in a sense that would exclude an equivalence to the Halting Problem. Deciding whether a given program outputs less than ten 33s, and if so how many, also involves deciding whether or not an arbitrary program outputs any 33 or not, which we have seen to be equivalent to the Halting Problem.

Problem 4: Can you write a program that checks for any given Python program P containing a conditional statement (i.e. an if-then-else statement), whether the Python program P will ever execute the “then” branch (i.e. the first branch) of the conditional statement?

Solution: No. If such a program were to exist, say a program $COND$ that checks for any program P (containing a conditional “if-then-else” statement) whether P executes the first branch (the “then”-branch) of this conditional, then, given any program P^1 and input I we could write the following program P' :

if $[(P(i) = P(i))]$ then A else B

If $P(i)$ terminates (returning a natural number value) then P' executes A . If P' executes A then this can only happen when $P(i)$ has terminated. So P' executes A iff

¹Recall that for the halting problem we considered programs that take natural number inputs and produce natural number outputs or do not terminate.

$P(i)$ terminates. Hence, if $COND$ can determine whether P' executes the conditional “then”-part A then we can solve the halting problem—a contradiction.

Comparison-based algorithms exercises

Exercise 75: Does the zero-one-principle hold for non comparison-based algorithms? If so, prove that the principle holds for such algorithms. If not, produce an algorithm that is 1) non-comparison-based, 2) sorts all binary sequences and 3) for which there exists a sequence of integers that it does not sort.

Solution: The zero-one principle fails to hold for non-comparison based algorithms.

A quick solution is to design an algorithm making one pass through the list from left to right, and, keeping two counters “zero-count” and “one-count”, counts the number of zeros and ones in the list. Then, assuming that the final zero-count has value k and the final one-count has value l , the algorithm makes a second pass from left to right and fills in k zeros and l ones (where the final one-count is l). This algorithm is $O(n)$ in the size n of the list.

This algorithm is not comparison-based (it relies on comparisons between a list element and a constant value (0 or 1), rather than on comparisons between list elements only). The algorithm sorts all binary sequences. Clearly it does not sort arbitrary sequences. Any sequence not containing 0 and 1 is left unchanged by the algorithm.

Exercise 76: (Binary witness-lists form a lossy encoding)

Consider a list $L = (a_1, \dots, a_n)$. For each list element a_i of this list we define a function f_{a_i} in the same spirit as the function used to produce binary-witnesses:

$$\text{if } x \leq a_i \text{ then } f_{a_i}(x) = 0 \text{ otherwise } f_{a_i}(x) = 1$$

This function f_{a_i} can be applied to each element of L to produce the binary list

$$(f_{a_i}(a_1), \dots, f_{a_i}(a_n)).$$

In general you cannot reverse this process, i.e. it is not possible in general to recover the list $L = (a_1, \dots, a_n)$ from a *single* binary list $(f_{a_i}(a_1), \dots, f_{a_i}(a_n))$. In other words, such a binary list forms a “lossy encoding.”

Consider the *list* of binary lists produced by the functions $f_{a_1}, f_{a_2}, \dots, f_{a_n}$ when applied to the list $L = (a_1, \dots, a_n)$:

$$(**) [(f_{a_1}(a_1), \dots, f_{a_1}(a_n)), \dots, (f_{a_n}(a_1), \dots, f_{a_n}(a_n))]$$

Show that this list of binary lists together with the *set* $\{a_1, \dots, a_n\}$ of all elements contained in the list L , is sufficient to recover the original list $L = (a_1, \dots, a_n)$. Specify a decoding that computes L from this list of binary lists and the set $\{a_1, \dots, a_n\}$.

Hint:

Consider a list $L' = (7, 1, 5, 3)$. Check how you can recover this list from the *list* of binary lists (displayed below) and where you can make use of the set of values $\{7, 1, 5, 3\}$ to achieve the decoding.

$$[(f_7(7), f_7(1), f_7(5), f_7(3)), (f_1(7), f_1(1), f_1(5), f_1(3)), \\ (f_5(7), f_5(1), f_5(5), f_5(3)), (f_3(7), f_3(1), f_3(5), f_3(3))]$$

Compute the actual bit-values for each element of these binary lists and explain how you can decode the original list from the given list of four binary lists *and* from the set of values $\{7, 1, 5, 3\}$.

Now, generalize your approach to arbitrary lists L and explain how to decode such a list from its set of values and the *list* of binary lists displayed in (**) above.

Solution: First, let's look what happens on the suggested example. Say $L = (a_1, a_2, a_3, a_4) = (7, 1, 5, 3)$. Consider the four functions $f_{a_1} = f_7$, $f_{a_2} = f_1$, $f_{a_3} = f_5$ and $f_{a_4} = f_3$, and apply these to the list $L = (7, 1, 5, 3)$:

$$\begin{aligned} (f_7(7), f_7(1), f_7(5), f_7(3)) &= (0, 0, 0, 0) \\ (f_1(7), f_1(1), f_1(5), f_1(3)) &= (1, 0, 1, 1) \\ (f_5(7), f_5(1), f_5(5), f_5(3)) &= (1, 0, 0, 0) \\ (f_3(7), f_3(1), f_3(5), f_3(3)) &= (1, 0, 1, 0) \end{aligned}$$

In the first case, we obtain the binary list $(0, 0, 0, 0)$. It has four zeros, which means there are four elements less than or equal to the encoded list element. This means that the encoded element regards the largest element of the list, namely 7 (which we can obtain from our set of list values: $\{7, 1, 5, 3\}$). Each element of the list, i.e. **4** list elements, are less than or equal to 7. This is encoded by **4** zeros.

Similarly, in the second case, we obtain the binary list $(1, 0, 1, 1)$. It has only **1** zero, indicating that it encodes an element which only has **1** element less than or equal to it. This is the smallest element in our list, which we can obtain from our set of list values $\{7, 1, 5, 3\}$ to be the element 1.

The third binary list is the list $(1, 0, 0, 0)$. It counts **3** zeros, hence encodes a list element that has **3** elements less than or equal to it. Again, using the set $\{7, 1, 5, 3\}$, this must be the list element 5.

Finally, the fourth binary list $(1, 0, 1, 0)$ has **2** zeros, indicating it encodes a list element with **2** elements smaller than or equal to it, which, using the set $\{7, 1, 5, 3\}$ means it must be the element 3.

Hence the decoded list, obtained from the list

$$[(0, 0, 0, 0), (1, 0, 1, 1), (1, 0, 0, 0), (1, 0, 1, 0)]$$

and the set

$$\{7, 1, 5, 3\}$$

is the list

$$(7, 1, 4, 3)$$

as expected.

Basically, each binary list $(f_{a_i}(a_1), \dots, f_{a_i}(a_n))$ captures how many list elements of L are less than or equal to a_i . This is expressed by the number of zeros in the binary list $(f_{a_i}(a_1), \dots, f_{a_i}(a_n))$. The number of zeros represents the rank that a_i takes amidst the elements of the list L . We defined the rank of an element as the number of elements strictly smaller than this element, so, really, the number of zeros in the binary list $(f_{a_i}(a_1), \dots, f_{a_i}(a_n))$ is $\text{rank}(a_i) + 1$.

We can decode a general list $L = (a_1, \dots, a_n)$ from the binary lists (displayed below) and its set of values $\{a_1, \dots, a_n\}$, using the following algorithm:

For each of the binary lists in the list

$$[(f_{a_1}(a_1), \dots, f_{a_1}(a_n)), \dots, (f_{a_n}(a_1), \dots, f_{a_n}(a_n))]$$

count the number of zeros. Say this yields the list of numbers is $[k_1, \dots, k_n]$. Replace each number k_i with the k_i -th element in the sorted version of the set $\{a_1, \dots, a_n\}$. This yields the required list $L = (a_1, \dots, a_n)$.

Alternative solution: count the number of “ones” in each *column* of the matrix created by the binary witness lists. In the above example, the first column has three “ones”. This indicates that the element recorded by that column is larger than three of the elements, hence must be the element 7. The number of “ones” per column indicate the rank of each element. Generalize this approach to arbitrary lists.

Lower bounds and decision tree exercises

Exercise 110 Use Kraft’s inequality to give an alternative proof of Corollary 99.

Solution: Consider a comparison-based sorting algorithm A and let $DT_A(n)$ be its decision tree for inputs of size n . This is a binary tree and hence the Kraft inequality applies (where $N = n!$):

$$K(DT_A(n)) = \sum_{i=1}^{n!} 2^{-l_i} \leq 1,$$

where $(l_1, \dots, l_{n!})$ is the pathlength sequence of the tree $DT_A(n)$.

Consider a path length, say l_j , where $j \in \{1, \dots, n!\}$, which has maximum value among all path lengths in the set $\{l_1, \dots, l_{n!}\}$. Clearly $T_A^W(n) = l_j$. Since $\forall i \in \{1, \dots, n!\}. l_j \geq l_i$, we obtain that $\forall i \in \{1, \dots, n!\}. 2^{-l_j} \leq 2^{-l_i}$ and thus

$$2^{-l_j} n! \leq \sum_{i=1}^{n!} 2^{-l_i} \leq 1.$$

Hence $2^{-l_j} \leq \frac{1}{n!}$ and thus $2^{l_j} \geq n!$. We conclude that $l_j \geq \log_2(n!)$ and hence

$$T_A^W(n) \geq \log_2(n!).$$

Further, using Lemma 93.

$$T_A^W(n) \in \Omega(n \log_2 n)$$

Exercise 137 a) *Is it possible for a comparison-based sorting algorithm to sort half of its inputs of size n in linear worst-case comparison-time? Here, we assume that on the other half of the inputs the algorithm is allowed to run arbitrarily slower. In other words, could we speed up sorting to ensure that at least on half of the inputs (of size n) will take linear worst-case time, while paying the possible price of increasing running times on the other half of the inputs (of size n)? Justify your answer.*

b) *Same question for average-case comparison-time.*

c) *Is it possible for a comparison-based sorting algorithm to sort $\frac{1}{n}$ of its inputs of size n in linear worst-case comparison-time? Again, on the other inputs, the algorithm is allowed to run arbitrarily slower. Justify your answer.*

d) *Same question as in c) but for average-case comparison-time.*

e) *Is it possible for a comparison-based sorting algorithm to sort $\Omega(2^n)$ of its inputs of size n in linear worst-case comparison-time? On the other inputs, the algorithm is allowed to run arbitrarily slower.*

Justify your answer. If you state that such a comparison-based algorithm cannot exist, then prove why it can't exist. If you state that such an algorithm can exist, then provide the pseudo-code for such a sorting algorithm and demonstrate that it runs in linear time on 2^n of its input lists.

Solution:

a) The answer is “no”. We know that for a decision tree with K leaves, the longest path length is greater than or equal to $\log_2 K$. Let A be a comparison-based algorithm that takes linear time on half of its inputs of size n . For this algorithm, consider a decision tree representation of all the execution paths of A on each of the $\frac{n!}{2}$ inputs (on which A takes linear time). This tree has $\frac{n!}{2}$ leaves. So the longest path is greater than or equal to $c \log_2(\frac{n!}{2})$ for some constant c (and from a certain value of n onward). Hence, on this collection of inputs (for which A supposedly runs in linear time), the algorithm A must have worst case time $T_A^W(n) \geq c \log_2(\frac{n!}{2}) = c \log_2(n!) - c \log_2(2) = c \log_2(n!) - c \in \Omega(n \log_2 n)$. So the algorithm cannot run in linear time on half of its inputs.

b) The solution method is the same as for a) since both average-case and worst-case satisfy the same lower bound.

c) The answer is once again “no”. The argument is similar to the argument given under a), where we use the following facts: $\log(\frac{n!}{n}) = \log(n!) - \log(n) \geq 1/4n\log(n) - \log(n)$ by Lemma 93. We can conclude that $\log(\frac{n!}{n}) \in \Omega(n\log n)$ since $1/4n\log(n) - \log(n) \in \Omega(n\log n)$. “Subtracting a logarithmic term from an $n\log n$ term leaves the $n\log n$ growth unchanged”. Hence as for case a) the answer is “no”. More formally: $1/4n\log(n) - \log(n) = \frac{1}{2}\log(n!) + \frac{1}{2}\log(n!) - \log(n) = \frac{1}{2}\log(n!) + \log(\frac{(n!)^{1/2}}{n})$. The last term is positive for n sufficiently large so $\log(\frac{n!}{n}) \geq \frac{1}{2}\log(n!)$ and we obtain $\log(\frac{n!}{n}) \in \Omega(n\log n)$ as before.

d) The solution method is the same as for c) since both average-case and worst-case satisfy the same lower bound.

e) If the algorithm runs in linear time on $\Omega(2^n)$ of its input lists, then the lower bound for worst-case time of comparison-based algorithms A states that $T_A^W \geq \log_2(2^n) = n$ (where 2^n is the number of leaves in our decision tree, when restricted to the 2^n inputs). Hence the longest path in the decision tree can be linear in the size of n . Instead of requiring $\Omega(n\log n)$ comparisons in the worst-case, the option exists to compute in $O(n)$ in the worst-case on a collection of 2^n inputs, i.e., the existence of the algorithm is not automatically excluded by the requirement that it should take linear worst-case time on 2^n of its input lists. There could exist multiple comparison-based sorting algorithms that take linear time on 2^n input lists. We sketch one such algorithm.

A stab at a solution

We need to select $\Omega(2^n)$ input lists on which to sort in linear time. An obvious collection of 2^n lists is the collection of all binary sequences of size n . For this collection it is easy to find a sorting algorithm that sorts in linear time.

As for the case of counting sort (see Section 12.8) we are in a situation where the input lists have elements with a certain range (in this case $\{0, 1\}$ rather than $\{1, 2, \dots, n\}$ as for counting sort).

An obvious linear time algorithm (on this input collection) makes a single pass from left to right through the list and count the number of zeros. Increment counter k , originally initialized to zero, each time you encounter a zero. Count the number of ones in a similar way, incrementing a counter l , originally initialized to zero. In left to right order, fill in k zeros in the list, followed by l ones to produce the sorted output. The sorting algorithm runs in linear time.

This is the solution presented in Exercise 75. Two problems remain with adopting this answer as a “solution” for the current exercise: similarly to counting sort, the algorithm is not comparison-based and adapting it to work on arbitrary input lists is not obvious.

Back to the drawing board. Recall that we needed to produce a comparison-based sorting algorithm that works on arbitrary lists of size n and runs in linear time on $\Omega(2^n)$ of these. Below we present such an algorithm. The argument is more intricate, but, learning from the above, we reach the solution by focusing on a particular subset of inputs of size $\Omega(2^n)$.

A solution

We will select $\frac{1}{2}2^n$ such lists, i.e. 2^{n-1} lists on which the algorithm needs to run in linear time. This suffices since $\frac{1}{2}2^n \in \Omega(2^n)$.

Recall that the number 2^n is related to the binary subset-encoding. There are 2^n subsets of a set of size n . Note that there are 2^n possible sublists of the sorted list $L = (1, \dots, n)$ of size n . This is shown similarly to the binary subset encoding. The selection of a sublist of the sorted list is determined by assigning the value 1 to each selected element and 0 to the non-selected elements. Hence sublists of a sorted list of size n can be encoded via 2^n binary sequences. Each selection of a sorted sublist L_1 determines a unique complement-list L_2 , once again sorted, consisting of the elements of L not in L_1 .

For the given sorted list L , we obtain 2^n such pairs (L_1, L_2) . Merging these sorted sublists L_1 and L_2 takes linear time, when using the standard merge algorithm.

Of the 2^n we retain exactly half, i.e. the pairs (L_1, L_2) for which L_1 forms the maximal sorted prefix of the list L_3 obtained by appending L_1 and L_2 . These are the pairs of lists (L_1, L_2) for which the last element of L_1 is strictly greater than the first element of the list L_2 . We refer to the collection of these pairs as the collection \mathcal{A} .

The other half of the possible choices (L_1, L_2) , which after appending form a list L_4 have a maximal sorted prefix that is longer than L_1 because the least element of L_1 is strictly less than the first element of L_2 .

Example: $L = (1, 2, 3)$

The eight pairs (L_1, L_2) consist of the following lists, where for each list we record the binary sequence that led to the choice of L_1 .

- 1) $(\emptyset, (1, 2, 3))$ (0,0,0)
- 2) $((1), (2, 3))$ (1,0,0)
- 3) $((2), (1, 3))$ (0,1,0)
- 4) $((3), (1, 2))$ (0,0,1)
- 5) $((1, 2), (3))$ (1,1,0)
- 6) $((1, 3), (2))$ (1,0,1)
- 7) $((2, 3), (1))$ (0,1,1)
- 8) $((1, 2, 3), \emptyset)$ (1,1,1)

Of these we only consider the four pairs (L_1, L_2) for which L_1 is the longest sorted prefix of their concatenation.

- 3) ((2), (1, 3)) (0,1,0)
- 4) ((3), (1, 2)) (0,0,1)
- 6) ((1, 3), (2)) (1,0,1)
- 7) ((2, 3), (1)) (0,1,1)

The last element of L_1 is strictly greater than the first element of the list L_2 exactly in case the last 1 in the binary encoding b determining the last element of L_1 comes after the first 0 determining the first element of L_2 . The restriction that the last 1 in the binary encoding b (determining the last element of L_1) must come after the first 0 (determining the first element of L_2) amounts to the fact that the rightmost 1 in the binary sequence must come after the leftmost 0. Since the binary lists determining the choices for L_1 and L_2 are exactly all (random) binary sequences of length n , the binary lists satisfying that the rightmost 1 in the binary sequence must come after the leftmost 0 form exactly half of the original 2^n cases.

Next, we make sure this (linear-time) computation on our selection of 2^n inputs forms part of the execution of a general sorting algorithm. Consider the following pseudo-code to compute the index that determines the longest-sorted-prefix of a list.

Longest-sorted-prefix-index(L)

```

i := 1;
while (i ≤ n − 1 and L[i] ≤ L[i + 1]) do i := i + 1;
return i

```

This algorithm computes the index determining the longest sorted prefix list $(L[1], \dots, L[i])$ of the list L .

Use the value stored in i to split the list L in two parts $L_1 = (L[1], \dots, L[i])$ and $L_2 = (L[i + 1], \dots, L[n])$ (if the list L is sorted, then L_2 is the empty list).

Recall that L_1 is sorted by design. It is the longest sorted prefix-list of the list L .

Run insertion sort on the sublist L_2 . This yields a new sorted list L'_2 . Merge the sorted lists L_1 and L'_2 (a linear process).

Our algorithm now is specified by the pseudo-code:

Mock Sort

```

j := Longest-sorted-prefix-index(L);
L1 := (L[1], ..., L[j]);
L2 := (L[j + 1], ..., L[n]);
Merge[L1, Insertion Sort(L2)]

```

In general, the call to Insertion Sort on list L_2 of length k will use $O(k^2)$ comparisons. However, on the 2^n pairs (L_1, L_2) that happen to consist of two *sorted* lists, Insertion Sort takes linear time on L_2 . Hence, on the 2^{n-1} lists of the collection \mathcal{A} ,

the algorithm Mock Sort takes linear time, as required.

Exercise 138 *Say you want to test whether a comparison-based sorting algorithm sorts all binary sequences of size n . Show that it suffices to check whether the algorithm sorts all non-sorted binary sequences of size n . Illustrate the reduction achieved by considering the binary sequences of length 4. Show that the number of lists to be tested drops from 16 to 11.*

Solution:

Note that a comparison-based algorithm will never destroy the order of a pair $(L[i], L[j])$ in case $L[i] \leq L[j]$. Hence the algorithm leaves all sorted lists sorted. There are exactly 5 binary sequences of length 4 that are already sorted (and hence will remain sorted, i.e., they do not need to be part of the test to check whether the sequences will be sorted, the outcome is known):

0000
0001
0011
0111
1111

Answer for exercise 135 (Subset-index decoding)

We outline an algorithm² for extracting the sequence $\{x_i\}_{i=1}^k$ given $f(\{x_i\}_{i=1}^k)$ (and also the value of n and k)

Algorithm 1 Extracting the sequence $\{x_i\}_{i=1}^k$ given $f(\{x_i\}_{i=1}^k)$

Input: $N = f(\{x_i\}_{i=1}^k)$, n – size of original list, k – sublist size.

Output: S (a set $\{x_i\}_{i=1}^k$)

Extract(N) :

$j \leftarrow k$

$S \leftarrow \emptyset$

for $i \leftarrow n$ to 1 **do**

if $N \geq \binom{i-1}{j}$ **then**

$S \leftarrow \{i\} \cup S$

$N \leftarrow N - \binom{i}{j}$

$j \leftarrow j - 1$

end if

end for

return S

²Decoding algorithm specified by D. Early: Diarmuid Early, Ang Gao, Michel Schellekens, Frugal encoding in reversible MOQA: A case study for Quick Sort, Reversible Computation Volume 7581 of the series Lecture Notes in Computer Science pp 85-96, 2013.

Example 139 Consider the set $X = \{2, 5, 7, 13, 16, 17, 19, 20, 26\}$ and the subset of size 5: $\{2, 7, 13, 20, 26\}$. Then $(x_1, x_2, x_3, x_4, x_5) = (1, 3, 4, 8, 9)$.

$$\begin{aligned} f(\{x_i\}_{i=1}^5) &= \binom{9-1}{5} + \binom{8-1}{4} + \binom{4-1}{3} + \binom{3-1}{2} + \binom{1-1}{1} \\ &= 56 + 35 + 1 + 1 + 0 = 93 \end{aligned}$$

Now, given $N = 93$, $n = 9$, $k = 5$, we set $j = 5$ and run the algorithm:

$$\begin{aligned} i = 9, j = 5, 93 &> \binom{9-1}{5} \text{ so } S = \{9\}, j = 4, N = 93 - \binom{9-1}{5} = 37 \\ i = 8, j = 4, 37 &> \binom{8-1}{4} \text{ so } S = \{8, 9\}, j = 3, N = 37 - \binom{8-1}{4} = 2 \\ i = 7, j = 3, 2 &< \binom{7-1}{3} \text{ so skip} \\ i = 6, j = 3, 2 &< \binom{6-1}{3} \text{ so skip} \\ i = 5, j = 3, 2 &< \binom{5-1}{5} \text{ so skip} \\ i = 4, j = 3, 2 &> \binom{4-1}{3} \text{ so } S = \{4, 8, 9\}, j = 2, N = 2 - \binom{4-1}{3} = 1 \\ i = 3, j = 2, 1 &= \binom{3-1}{2} \text{ so } S = \{3, 4, 8, 9\}, j = 1, N = 1 - \binom{3-1}{2} = 0 \\ i = 2, j = 1, 0 &< \binom{2-1}{1} \text{ so skip} \\ i = 1, j = 1, 0 &= \binom{1-1}{1} \text{ so } S = \{1, 3, 4, 8, 9\}, j = 0, N = 0 - \binom{1-1}{1} = 0 \end{aligned}$$

Correctness of the algorithm

To show that the algorithm is correct, it suffices to remark that for any sequence $\{x_i\}_{i=1}^k$:

$$\sum_{i=1}^{k-1} \binom{x_i-1}{i} \leq \binom{n}{k-1} - 1$$

which follows from the proof of Lemma 133.

But $\binom{n}{k-1} - 1 < \binom{n}{k}$. Hence, the only way a multiple of $\binom{n}{k}$ can occur in the summation $S(\{x_i\}_{i=1}^k) = \sum_{i=1}^k \binom{x_i-1}{i}$ is through the last term x_k , for which $\binom{x_k-1}{k} = \binom{n}{k}$. (Again by the proof of Lemma 133.)

The end

OMG, this course did halt ;)