Version: 2016-10-19 21:14

EAA for International Students

Prof. Dr. Markus E. Nebel · Sebastian Wild, M. Sc.

Contents

1 Purpose of this Document

1.1 What is 'EAA'?

The acronym EAA stands for the German lecture „Entwurf und Analyse von Algorithmen“ (“Design and Analysis of Algorithms”), which is a mandatory module in the courses of studies “Bachelor in Computer Science” and “Bachelor in Computer Science in Applications”.[1] As such, it is an indirect prerequisite for the corresponding consecutive Master courses of studies, as well.

EAA covers many basic algorithms and data structures and puts some focus on their mathematical analysis. The main skills this course conveys to students are

- designing (new) algorithms, along typical design patterns,
- competence in judging a problem's computational complexity and, on the basis of that, in choosing appropriate algorithms.
- Moreover, proficiency in formally describing correctness and efficiency is required to communicate the quality of a new solution to other experts. Such descriptions are also found in today's programming libraries[2], hence understanding them is vital also for software developers.

The EAA module comprises a one semester lecture with two weekly sessions and weekly tutorials with hand-in exercise tasks for students, which are graded by teaching assistants. A minimal number of points from these assignments is necessary to participate in the exam; passing the exam is required to get credit for the module. The lecture is based on the German textbook of the same name [Neb12].

1.2 International Students

Many international Master students did not have a course equivalent to EAA in their Bachelor studies, their admission to the Master program is therefore conditional on the completion of prerequisite studies. The EAA lecture, the corresponding book and the tutorials are all offered in German only, which makes it hard for foreign students to participate and fulfill their prerequisites.

This document is intended to help international students by collecting a list of references to English textbooks that cover roughly the same material as the original German EAA course. It provides guidance through the material by breaking it up into individual lessons of manageable size.

[1] Find detailed information in the Modulhandbuch (German): `http://www.informatik.uni-kl.de/studium/lehrveranstaltungen/modulhb/#89-0006`

[2] Here are two examples from Java library: (1) $\mathcal{O}(\log n)$ time guarantees of `java.util.PriorityQueue`, (2) “expected average $\log(n)$ time cost” in `java.util.concurrent.ConcurrentSkipListMap`, the latter requires proper understanding of the combination of several concepts.

Note that a written script as this can never be a full surrogate for a whole semester of interactive tutorials. In particular, the ability to design algorithms is best learned by designing algorithms for yourself — and in failing to do so many times and then trying it again. Generations of students have made this experience that searching for solutions for yourself, but under the guidance of a good teacher, is the most effective way to learn how to find solutions.

Unfortunately, we cannot guarantee this assistance all the time. With a little sincere diligence, however, it is still possible to master the material collected herein. The referenced reading materials list many exercise problems, and you are highly recommended to work through at least two or three of those for each single lesson. This is the most effective way to gain proficiency with the material.

1.3 Theoretical Computer Science Amendment

The module "Theoretical Computer Science Amendment" can be part of the Master course of studies, in which case it is worth up to 8 ECTS credit points. It is intended for students who lack some, *but not all*, of the theoretical foundations of computer science which are covered by the German Bachelor courses EAA, „Logik" (logic) and „Formale Grundlagen der Programmierung" (mathematical foundations of programming).

This module comprises much more material than what would be usual for a 8 credit-point course since it is assumed that one can build on skills acquired in a previous Bachelor programme. No one student should have to learn all content from scratch, instead material is provided to fill individual gaps and deepen existing knowledge. Programming skills and fundamental skills in mathematics are required for this course.

It is obvious then that the module "Theoretical Computer Science Amendment" cannot be equivalent to the three German Bachelor courses per se; rather it complements other theoretical computer science courses from a Bachelor programme so that, together with the amendment, the students acquired skills equivalent to the three German Bachelor modules EAA, „Logik" and „Formale Grundlagen der Programmierung".

As for the EAA-related content, here is the corresponding excerpt from the official documents („Modulhandbuch") `89-5021`:

> From Design and Analysis of Algorithms:
>
> - Asymptotic notations (O, o, Θ, $\sim$) and their use;
> - Cost models (worst-, best-, average-case, bit complexity, elementary operations);
> - Master Theorem with applications;
> - Recursive algorithms $\rightarrow$ recurrence relations $\rightarrow$ techniques for solving recurrence relations;
> - Elementary data structures (lists, stacks, [priority] queues, sets, graphs and trees, partitions);
>   - Dictionaries

  - binary search trees and balanced trees (with analysis);
  - hashing (various schemes, uniform as well as universal hashing, some with analysis);
- Fundamental graph algorithms (traversal, shortest path, minimal spanning trees [Kruskal, Prim]);
- Design patterns for algorithms
  - Divide and conquer (with examples from sorting);
  - Dynamic programming (with examples from formal languages);
  - Greedy (with examples from approximation) and matroids;
- Complexity theory (reduction, NP-completeness, Cook's theorem, fundamental NP-complete problems with proofs).

2 EAA by Topic

EAA is taught as a one semester course with two lectures per week plus tutorials. On average, each of the following lessons covers roughly the content presented in one of these 90 minute lecture sessions, but you should expect the lessons to vary in size. Also, some of the more specialized topics are excluded for the theoretical computer science amendment.

List of Lessons

▶ Lesson 0: *How to Use This Document*

10min reading

This meta-lesson describes the parts that any lesson is composed of and how to approach self-study using this document.

Every lesson has a number and a title ("Lesson 0: How to Use This Document" for this lesson). Below the title, you find an approximate estimate of how long it takes to complete this lesson, assuming basic knowledge of math and computer science, but no specialized education in algorithms. Your mileage may vary! Often, separate durations are given for reading the material and for practicing your understanding and problem solving skills by working on accompanying recommended exercise problems.

Working on exercise problems is an important part of each lesson! Take your time to go through each problems and to think it through. Then try to write down explicitly a (partial) solution for the problem. This is the best — maybe the only effective — way to prepare for the final written exam for the vast majority of students.

For some selected problems, detailed solutions are provided at the end of this document. The intention is to give you a feeling for the level of detail expected from you in a written exam. *Spoiler alert:* do resist the temptation to look at provided solutions without making a serious attempt to come up with and *write down* a solution on your own. Don't cheat on yourself!

You will sometimes find an estimate like "1h reading" and sometimes "1h careful reading". The latter indicates that the material in this section is especially demanding, or probably new to you — or both. Avoid these lesson in the coming-home-from-party-at-1am time slots.

After title and effort estimate, a short description of the topic is given. This summary is not intended to explain or even replace the detailed exposition of the topic. Rather, it briefly describes the lesson's content, so that experts might decide to skip a lesson about material they are familiar with. Don't worry if the summary contains terms/statements you do not (yet!) know or understand.

? Central Questions

- Here, we give some guidance on reading by posing basic questions you should be able to answer immediately after having read the material.
- Often, you are asked to restate definitions and algorithms in your own words.
- Sometimes questions point to assumptions that may be implicit in the text.
- These questions are the starting point, exam problems will go deeper (more like the exercise problems)!

Reading Material 0.1 Section X (pages xx–yy), [REF]

Each reading material item represents one section of a textbook. The precise reference is given in the title line (as above), where [REF] is an entry from the bibliography (page 27).

Below the reference, a short summary is given of what is covered in this reading material. Again, this summary is neither intended to replace the material, nor to be understood without knowledge on the lesson's topic; it is intended for you to recognize material you already know. We highly recommend to skim over the textbook sections anyway, as they contain some material not covered in all basic algorithms courses.

▶ Lesson 1: *Introduction and Motivation*

1 h reading

In this first lesson, some examples and applications of algorithms will be given and we motivate the analysis-driven approach used in this course.

Central Questions

- What is the advantage of mathematical, asymptotic analysis of algorithms over running time experiments?
- What is an *algorithm*? What is the difference to a program?
- Analysis of algorithms typically means to *count* something; what do we count? Give examples.

Reading Material 1.1 Chapter 1 (pages 5–15), [Cor+09]

This chapter gives some practical examples where algorithms are put to good use. It does not contain vital information for EAA and so might be skipped, but the examples are nice enough to skim over them in a few minutes of spare time.

Reading Material 1.2 Chapter 1 up to 1.3 (pages 1–16), [Raw92]

This is an introductory chapter giving motivation for our analysis-focused approach to algorithms and data structures using metaphors from real life. Let the author speak for himself:

> "It isn't sensible to run the program to see if it's too slow. If it's too slow we have to change it to speed it up. But, with no guiding principles to aid prediction, every time we change the program we have to rerun it to find out how long it takes after the change. This is frustrating, it wastes our time, and it wastes computer resources. Worse, it isn't even guaranteed to make the program fast enough. On the other hand, if the program is fast enough for the current movie we still won't know if it will be fast enough for more complex movies. In both cases we need a way to

> predict — based on the time it takes to produce simple scenes — how much time it will take to produce complex scenes. We need a *yardstick*[3] to measure the program's performance as a function of the movie's complexity." [Raw92, p. 2]

> "Instead of thinking of a particular program we should consider the idea behind the program, the *algorithm* — the language- and machine- independent strategy the program uses. Algorithms are to programs like plots are to novels." [Raw92, p. 3]

▶ Lesson 2: *Mathematical Basics and Asymptotics*

4 h careful reading; plus time for exercises as necessary

This lesson is a recap from basic math courses. We highly recommend to at least skim over the material below, even if you have seen it before. Maybe, a few parts are new to you. More importantly, though, the math covered in the given chapters is essential for understanding most later parts, so make sure you do the exercises in this lesson.

Apart from recapitulating mathematical knowledge and tools, the purpose of this lesson is also to have a least common denominator of formula-language. It is important to become fluent with the mathematical language. But:

> "don't be afraid of the symbols. [...]
> Mathematics is no more about symbols than mountain climbing is about ropes. Symbols only help us get to our goal — they can neither do the job for us nor are they indispensable — we could do without them, but the climb would be much harder." [Raw92, p. 359]

? Central Questions

- What is the definition of $\mathcal{O}(f)$?
- How can we prove that $n \in \mathcal{O}(n \log n)$?

🕮 Reading Material 2.1

Sections 1.6 and 1.7 (pages 27–49), [Raw92]

Recap of elementary (real) analysis and discrete math:

- logarithms
- analysis (limits)
- calculus (derivatives)
- l'Hôpital's rule
- inductive proofs
- Fibonacci numbers
- factorials
- binomials
- finite and infinite sums
- $\mathcal{O}$-classes and relatives

[3] Of course, we use meters to measure our algorithms in Europe ;)

Reading Material 2.2 Chapter 3 (pages 43–64), [Cor+09]

This chapter introduces $\mathcal{O}$-classes and their relatives, shows how to "compute" with them. Additionally, a list of basic concepts and special functions/function classes is given with their properties, (similar in part to Reading Material 2.4):

- floor and ceiling, div and mod
- polynomials, exponentials, logarithms, factorial
- Fibonacci numbers

Reading Material 2.3 Appendix A (pages 1145–1157), [Cor+09]

This chapter is a reference list of rules for computing with finite and infinite sums (similar in part to Reading Material 2.4):

- basic rules for sums
- telescoping sums
- approximation by integrals
- bounds by splitting summations

Reading Material 2.4 Appendices A, B and C (pages 467–499), [Raw92]

This is a collection of most mathematical preliminaries used in the course, including many examples of how to apply them. You might skip this material first and look up things on demand, whenever you stumble upon math that you are not familiar with.

Lesson 3: *Designing an Algorithm — Overview on Examples*

2 h careful reading

This lesson shows the complete process of designing an algorithm, as we will use it in EAA for many problems. As an example, we pick a simple algorithm, namely Insertionsort, where the process can be followed without being lost too easily in technical detail.

The process of designing an algorithm consists of the following steps:

0. a clear description of the problem to be solved, (this will usually be given to you),
1. an informal description of the algorithmic idea for your solution,
2. a pseudocode implementation of your algorithm,
3. a proof of the correctness of your algorithm, (usually by separately showing that your algorithm terminates and that it correctly solves the problem),
4. and finally, a running time analysis of your algorithm, giving the Θ-class of the number of elementary operations (RAM model). Depending on the application, we may seek best-case, worst-case and/or average-case results.

You will be expected to address each of these points in exercise and exam problems where an algorithm is to be designed.

? Central Questions

- How does Insertionsort work? Write down pseudocode for it *on your own*.
- How do best-case and worst-case inputs for Insertionsort look like? Argue why these really are the most extreme cases possible.
- What is a suitable (set of) elementary operation(s) (the *model* in the sense of Rawlins, see Reading Material 1.2) to study Insertionsort? Suitable here means, that the resulting Θ-class of running time is the same as Cormen et al. find.
- How does the algorithm Hanoi (for solving the Towers of Hanoi problem) work? Write down pseudocode for it *on your own*.
- How is the code for Hanoi analyzed?

Reading Material 3.1 Sections 2.1 and 2.2 (pages 16–29), [Cor+09]

Shows the above steps in detail on the example of Insertionsort.

Reading Material 3.2 Section 1.4 (pages 16–26), [Raw92]

Illustrates the design of algorithms on the example of The Towers of Hanoi.

▶ Lesson 4: *Elementary Data Structures*

2 h reading; plus 2 h exercises

In this lesson, we consider some elementary data structures — *stacks*, *queues* and *sets* — that are used as building blocks later on many times. For each, we devise sequential (array-based) and linked-listed based implementations.

This lesson is usually covered in programming courses, so much of the content should be familiar to you. If so, skim briefly over the reading material and go on with the exercises.

? Central Questions

- What are the characteristics of an array? (running time of access to ith element, insertion of elements, overwriting values, resizing?)
- What are the characteristics of a (doubly) linked list?
- How can you implement a stack with an array? How a queue? Draw pictures.

Reading Material 4.1 Chapter 10 up to incl. 10.2 (pages 229–241), [Cor+09]

Briefly explains how to implement stacks and queues using arrays and introduces (doubly) linked lists.

Reading Material 4.2 Section 1.3 (pages 120–171), [SW11]

More detailed description of the data structures including full Java code and very nice application examples.

▶ Lesson 5: *Basic Rules for the Analysis of Algorithms*

2 h careful reading; plus 2 h for exercises

In general, it can be very complicated to analyze the running time of an algorithm, but for many important and frequent special cases, we can get along by following simple rules. In this lesson, you will learn these rules.

Central Questions

- How can be mathematically describe the running time of a for-loop?
- How can be mathematically describe the running time of a recursive algorithm?
- In Lesson 3, you have already seen two examples of algorithm analysis. Which rules were used there?

Reading Material 5.1 Appendix B and C.1, C.2 (pages 481–496), [Raw92]

Reference list of the rules of translating pseudocode to formulas describing their running time.

▶ Lesson 6: *Solving Recurrences I — Guess and Prove & Master Theorem*

1 h careful reading; plus 3 h forexercises

As we have seen in Lesson 5, recurrence equations result from the analysis of recursive algorithms. The recurrence itself is, however, not suitable to compare algorithms, we first have to *solve* it — or at least determine its asymptotic behavior. This and the next lesson teach you methods for solving frequently occurring classes of recurrence equations.

Central Questions

- How can we formally prove a guessed solution for a recurrence?
- How can we come up with an educated guess for a solution? List and explain at least two approaches.
- What does the master theorem provide us with?
- Give an example for a recurrence for each of the three cases of the master theorem.

Reading Material 6.1 Sections 4.3 and 4.4 (pages 83–97), [Cor+09]

These sections introduce the first round of techniques for solving recurrences:

- guessing a solution and proving its validity by induction,
- using the recursion tree idea for coming up with good guesses, and
- the master theorem as shortcut for recurrences of specific forms.

Optional Reading Material 6.2 Section 4.6 (pages 97–106), [Cor+09]

This section contains the proof of the master theorem as stated in Reading Material 6.1.

Lesson 7: *Solving Recurrences II — Linear Recurrences*

90 min careful reading; plus 4 h for exercises

Apart from the master theorem that allows to determine an asymptotic solution for certain recurrences, there is a whole class of recurrences that we can even solve precisely: linear recurrences with constant coefficients. In this lesson, you learn how.

These recurrences appear naturally in the analysis of many algorithms and can also serve as basis to solve more complicated recurrences, e. g., via substitution.

Central Questions

- Are the following recurrences linear recurrences with constant coefficients?
- Explain how to solve recurrences of the form $a_n = ca_{n-1}$, $a_0 = d$ for two numbers c and d using the techniques from Lesson 6 and using generating functions.
- What is an (ordinary) generating function?
- If $A(z)$ and $B(z)$ are the generating functions for sequences $\{a_n\}$ and $\{b_n\}$, how does the sequence with generating $A(z) \cdot B(z)$ look like?

Reading Material 7.1 Section 2.4 (pages 55–59), [SF13]

This section defines the class of homogeneous linear recurrences with constant coefficients and shows a solution using ad-hoc arguments.

Reading Material 7.2 Section 3.1 (pages 92–97), [SF13]

In this section, (ordinary) generating functions are introduced.

Reading Material 7.3 Section 3.3 (pages 101–108), [SF13]

With the preliminaries from in Reading Material 7.1 and 7.2, this section shows you how to solve linear recurrences with generating functions.

▶ Lesson 8: *Union-Find Data Structure*

90 min careful reading, plus 2 h exercises

This lesson introduces the *Union-Find* data structure which represents a (dynamic) partition of a finite set: starting with singleton sets (each element of the set forms a cluster of its own), we can "union" two partitions and we can "find" the identifier of the partition a certain element currently belongs to. Comparing the result of two "finds" thus allows to decide whether two elements are in the same partition or not.

Even though this data structure might not seem very natural at first sight, it is a helpful building block for more complex algorithms. It is used, e. g., for computing minimal spanning trees of graphs (Lesson 16) and to speed up connectivity queries in graphs.

Moreover it serves as example for once more showing the complete agenda of design and analysis of a data structure, paralleling Lesson 3 on the design of algorithms.

? Central Questions

- Explain how we can use a Union-Find data structure to efficiently answer connectivity queries in a graph (are nodes v und u connected by a path?), after preprocessing the graph once.
- What are the different implementations of Union-Find given in the text? Explain them in your own words, especially the meaning of the `id` array. What are the differences in terms of efficiency?

🕮 Reading Material 8.1

Section 1.5 (pages 216–240), [SW11]

This section introduces the Union-Find data structure and successively devises several implementations for it, ever improving overall performance. Efficiency is both analyzed theoretically and evaluated on practical inputs.

▶ Lesson 9: *Dictionaries I — Binary Serach Trees*

2 h reading; plus 8 h for exercises

Binary search trees (BSTs) are arguably the most important non-elementary data structure in computer science. They are directly useful in practice and serve as building block for many more elaborate data structures.

In this lesson, we cover unbalanced BSTs, which are only efficient in the average case, but set the stage for balanced BSTs discussed in Lesson 10

You will probably have seen BSTs in a programming course, but we might not have seen the precise analysis of their efficiency and maybe also some of the less common algorithms on BSTs like rank-selection.

? Central Questions

- What is a BST? Can you give a concise inductive definition?
- Can a BST contain several nodes with the same key?
- How does a non-successful search in a BST help with inserting new keys?
- What are the performance characteristics of a search in a BST? Best case? Worst case? Average case? (For the latter, what exactly are we averaging over?)
- What is the internal path length of a BST?
- Is a BST a digraph in the sense of Lesson 14?

Reading Material 9.1 Section 3.2 (pages 396–423), [SW11]

This section gives a from-the-ground introduction to BSTs, featuring a detailed implementation of a sorted dictionary with (eager) deletion and rank-based query methods. Moreover, the expected path length is computed using recurrences and results on expected height are discussed.

▶ Lesson 10: *Dictionaries II — Balanced Search Trees*

3 h careful reading

BSTs as introduced in Lesson 9 only yield logarithmic running time in the average case, which is not sufficient for many applications. By enforcing more structure in the BSTs, i. e., only allowing certain restricted subclasses of trees that are logarithmic in height, we obtain a dictionary implementation with guaranteed logarithmic update and query time.

Historically, *AVL trees* were the first data structure to achieve this goal by enforcing that trees remain *height-balanced*: The difference between the height of the left subtree and the right subtree may be at most one, for *node* in the tree.

In practice, *red-black trees* proved more successful, which ensure that the depth of any two leaves in the tree are within a factor of 2 of each other. This property is established by coloring nodes either black or red, hence the name. We will focus on red-black trees in this lesson.

? Central Questions

- What are rotations in BSTs? What are they used for?
- What are (perfectly balanced) 2-3 trees? How can we insert into such a tree?
- How can we (conceptually) transform a 2-3 tree into a left-leaning red-black tree, and vice versa?
- Why do we not directly implement 2-3 tree dictionaries? Discuss!

Reading Material 10.1 Section 3.3 (pages 424–456), [SW11]

This section first defines 2-3 trees, which are a special case of B-trees. These are the conceptual data structure behind red-black trees that make understanding the operations on red-black trees much easier.

Then the actual (left-leaning) red-black BSTs are introduced as an efficient way to represent 2-3 trees in a computer and detailed implementations are given for the dictionary operations. Finally the authors show that the resulting trees are guaranteed to have height at most $2 \ln n$, so that all dictionary operations run in logarithmic worst-case time.

▶ Lesson 11: *Dictionaries III — Hashing*

2.5 h careful reading

In this lesson, you will see that in expectation, all dictionary operations can be supported in constant time using *hash tables*. We cover the basic versions of hash tables hashing with indirect chaining and linear probing hashing.

Hashing is one of the most important strategies in practice for efficient programs, but at the same time its characteristics are not always intuitive and there are subtle errors to make that can severely degrade performance. Therefore it is vital — for this lesson more than in any other — to understand the details of the assumptions that are required for hashing to be efficient.

Central Questions

- What is the basic idea behind hashing? Explain in your own words.
- What is the worst case for hashing (assuming deterministic hash functions)? Can we avoid it?
- What does "uniform hashing" mean?
- What happens if more entries than hash table array cells are inserted in the dictionary for (a) indirect chaining and (b) for linear probing?
- What are the pros and cons of indirect chaining and linear probing?

Reading Material 11.1 Section 3.4 (pages 458–477), [SW11]

This section introduces the idea of hashing, how to implement reasonable (deterministic) hash functions and it discusses the two most elementary organizations for hash tables: chaining keys with the same hash in an additional list and linear probing to use existing free slots in the hash table itself.

For both, detailed implementations are discussed, including dynamic array resizing. Indirect chaining is also analyzed under the uniform hashing assumption, for linear probing missing analytical results are cited.

▶ Lesson 12: *Dictionaries IV — Universal Hashing*

45 min careful reading

Hashing as introduced in Lesson 11 has linear worst-case time for a single operation, which is undesirable if a malicious adversary might choose such a bad input. *Universal hashing* provides a probabilistic defense against such attacks.

? Central Questions

- What is a universal collection of hash functions?
- Classical hashing and universal hashing have the same expected performance — where is the difference then?! (Hint: what is taken expectations over?)

🕮 Reading Material 12.1

Section 11.3.3 (pages 265–269), [Cor+09]

The section defines the concept of universal hashing and shows that linear functions modulo primes are universal.

▶ Lesson 13: *Heaps and Priority Queues*

2 h careful reading; plus 3 h for exercises

Priority Queues are data structures that store a set of elements inserted into the queue and provide direct access to the current largest element in the queue.

? Central Questions

- What are the two main operations of priority queues?
- How can priority queues be used for sorting? Give the abstract idea.
- What is the (complete binary) tree representation of binary heaps? Draw at least two different heaps representing the numbers $1, \ldots, 9$.
- How are heaps stored as arrays? For your examples from above, give the corresponding sequential representations as arrays.

🕮 Reading Material 13.1

Section 2.4 (pages 308–322), [SW11]

This section introduces the abstract data type *priority queue* and gives a few examples where it is used. After observing that elementary implementations require linear time for at least one of the main operations "insert" or "delMin", the authors discuss a detailed implementation based on binary heaps.

Additionally, they introduce *index priority queues*, which allow index-based access to elements to change the key value of elements already in the queue. This data structure is an important building block for further algorithms.

▶ Lesson 14: *Graphs I — Basics*

2 h reading; plus time for exercises

This lesson recaps basic definitions of *graphs, digraphs (directed graphs)*, and *trees*. It also covers elementary graph traversal algorithms *depth-first search (DFS)* and *breadth-first search (BFS)*, which serve as building blocks for more complicated algorithms. In this lesson, we only consider unweighted graphs.

? Central Questions

- What are the typical options for storing graphs in a computer? How do they differ (w. r. t. efficiency)?
- How do DFS and BFS work? Draw an example graph and determine the order nodes are visited in.
- What is the relation between topologically sortable digraph and DAGs (directed acyclic graphs)?

🕮 Reading Material 14.1 Section 4.1 up to "Breadth-First Search" (pages 514–542), [SW11]

This section gives the basic definitions for undirected graphs, paths and trees. It discusses different representations for graphs and gives a detailed implementation of a graph class. Then DFS and BFS are introduced including detailed code.

🕮 Reading Material 14.2 Section 4.2 up to "Cycles and DAGs" (pages 566–583), [SW11]

This section gives the definitions of digraphs and traversals on digraphs. As an application cycle detection and topological sort is discussed.

▶ Lesson 15: *Graphs II — Shortest Paths*

1 h careful reading

In this lesson, you will learn about the famous algorithm by *Dijkstra* for computing shortest paths in weighted digraphs.

The algorithm builds on *priority queues* introduced in Lesson 13.

? Central Questions

- How are weights of paths defined?
- Is there always a shortest path between nodes s and t? And if there is one, is it unique? Give examples!
- What is the definition of a shortest path tree (SPT)?

- How does Dijkstra's algorithm work?
  Explain in your own words.
- What is the running time of Dijkstra's algorithm in terms of running time for the operations on the used priority queue? With the binary-heap based implementation of Lesson 13, what is the overall running time in terms of $n = |V|$ and $m = |E|$?

Reading Material 15.1 Section 4.4 up to incl. "Dijkstra's algorithm" (pages 638–657), [SW11]

This section defines the problem, gives a detailed implementation of Dijkstra's algorithm and proves the correctness via generalizing Dijkstra to a generic edge relaxation algorithm.

▶ Lesson 16: *Graphs III — Minimum Spanning Trees*

90 min reading; plus time for exercises

In this lesson, we consider the problem to compute a *minimum spanning tree (MSTs)* of a weighted (undirected) graph, which serves as important abstraction of real-world scenarios where connectivity of a graph is to be maintained at minimal cost.

The MST algorithms discussed in this lesson build on the priority queue data structure and the union-find data structure introduced in Lesson 13 and 8, respectively.

? Central Questions

- What is the definition of a spanning tree? What is *minimal* in MSTs?
- Which algorithm design paradigm is used in all considered MST algorithms?
- How does Prim's algorithm work, how does Kruskal's algorithm work? Explain in your own words. What is similar and what is different in the two?

Reading Material 16.1 Section 4.3 (pages 604–629), [SW11]

This section defines MSTs, discusses the cut property in trees that forms the basis of MST algorithms and it finally gives detailed implementations of Prim's algorithm and Kruskal's algorithm.

The section "Eager version of Prim's algorithm" (pages 620–623) can be skipped.

▶ Lesson 17: *Sorting I — Quicksort and Quickselect*

2 h careful reading; plus 5 h for exercises

In this lesson you will consider in detail the arguably most important sorting algorithm in practice: *Quicksort*. You will probably already know how Quicksort works, but many algorithms courses skip its detailed analysis and important optimizations used in practice. This lesson is therefore highly recommended to all students.

? Central Questions

- What does "partitioning" in the context of Quicksort mean? What is a "pivot element"?
- What is the abstract idea behind Quicksort? Describe it in your own words.
- What is the worst-case running time (Θ-class) of Quicksort? What is the average case?
- What are the reasons to use Quicksort in practice according to Sedgewick and Wayne?
- What is the rank selection problem? Explain with an example.
- How does Quickselect work? Describe it in your own words.

🕮 Reading Material 17.1

Section 2.3 (pages 288–307), [SW11]

This section introduces Quicksort from scratch and provides an implementation around Hoare's crossing-pointer partitioning method. A correctness proof is sketched and the expected and worst-case number of comparisons are analyzed in detail. Finally, some improvements to the core algorithm are presented and discussed:

- Insertionsort on small subarrays
- pivot as median-of-three
- three-way partitioning to handle duplicate keys more efficiently

🕮 Reading Material 17.2

Section 2.5 (pages 345–347), [SW11]

This paragraph briefly explains the rank selection problem and presents Quickselect as an expected linear-time algorithm.

▶ Lesson 18: *Sorting II — Mergesort & the Lower Bound for Sorting*

1 h reading; plus time for exercises

This lesson covers Mergesort, the second famous divide-and-conquer sorting algorithm. Even though in practice, usually Quicksort is preferred to Mergesort, the latter has theoretical qualities that are important to know for computer scientists: This lesson

coveres the information-theoretic lower bound for comparison-based sorting algorithms and shows that Mergesort is asymptotically optimal with respect to the number of needed comparisons.

Even though you will probably have heard of Mergesort already, skim over the material of this lesson, especially for the theory part.

? Central Questions

- What does it mean to "merge" two lists/arrays? What do we have to assume about the two lists so that we can merge them?
- Can we (easily) merge to halves of an array in place? How is merging done in 18.1?
- What is the difference between top-down and bottom-up Mergesort? Describe the workings of both in your own words.
- What is the information-theoretic lower bound on comparison-based sorting? Explain the idea of the proof.
- What is meant if one says "Mergesort is optimal" (in the context of this lesson)? Give a (mathematically) precise statement that justifies this claim. What are the (implicit) assumptions behind this statement?

🕮 Reading Material 18.1 Section 2.2 (pages 270–287), [SW11]

The section introduces Mergesort from the ground, giving top-down and bottom-up variants including worst- and best-case analyses. Then, the lower bound for comparison-based sorting is proven, which matches the worst-case bound for Mergesort. Practical implications of this theory are discussed briefly, as well.

🕮* Optional Reading Material 18.2 Section 4.7 (pages 259–268), [Raw92]

This section gives some more background on information theory and details on how to obtain the asymptotic estimate $\mathrm{ld}(N!) \sim N \,\mathrm{ld}\, N$.

▶ Lesson 19: *Sorting III — Distribution Counting & Radixsort*

◔ 1 h careful reading; plus time for exercises

Lesson 18 taught us that comparison-based sorting always needs $\Omega(n \log n)$ time; in this lesson, you will learn how to circumvent this lower bound. The trick will be to make further assumptions on the elements to sort so that we need not use binary comparisons for sorting.

The resulting algorithm, *Radixsort*, is among the most efficient methods in practice to sort lists of numbers. Its parameters need to be tuned so to match the used hardware, so it is not usually used as default sorting method. Radixsort implementations in high performance libraries, however, typically outperform any general purpose sorting method by far.

? Central Questions

- How does Countingsort work? Describe the algorithm in your own words. First consider the case that all elements are distinct.
- What is the assumption on the input in Countingsort?
- In Reading Material 19.1 the authors show that Countingsort needs $\Theta(n+k)$ time; where does the n come from?
- How does Radixsort work? Describe the algorithm in your own words.
- Why is it important for Radixsort that Countingsort is stable?
- What is the assumption on the input in Radixsort?
- Is the assumption on the input for Radixsort a limitation in practice? Discuss.

🕮 Reading Material 19.1

Sections 8.2 and 8.3 (pages 194–200), [Cor+09]

The sections first introduce Countingsort, which is then used as building block in Radixsort. Both algorithms as analyzed. The authors briefly discuss the choice of the radix and practical implications.

▶ Lesson 20: *Sorting IV — Sorting Networks*

30 min reading

In this last lesson on sorting we have a brief look at sorting networks, which are data-oblivious sorting algorithms. Apart from the direct application in designing hardware sorting modules, data-oblivious methods are also important for sorting vector components in parallel, especially on modern GPUs and processors with long pipelines.

? Central Questions

- What do the drawings with the wires mean? Explain at an example, how a sorting network is "executed".
- How does the sorting network for Insertionsort look like? Draw the network for $n = 5$ and explain the connection to the code (as given, e. g., in Reading Material 3.1).

🕮 Reading Material 20.1

Section 5.3.4 (pages 219–225), [Knu98]

This section introduces sorting networks, including sample constructions for Insertionsort and Selectionsort. It also covers the zero-one-principle and uses this to study the odd-even merge network.

Optional Reading Material 20.2 Section 5.3.4 (pages 225–229), [Knu98]

This paragraph discusses the problem of finding minimum-comparison sorting networks and also lists the best known networks for small sizes.

Lesson 21: *Algorithm Design Patterns I — Dynamic Programming*

2 h careful reading; plus time for exercises

Many problems appear very hard to solve at first sight, but sometimes a clever use of some scratch memory allows to solve them efficiently. In particular for optimization problems with many solution candidates, a brute force search is prohibitively expensive, but we can often exploit structure in the solution space to reuse solved subproblems. This is the core idea of *dynamic programming (DP)*, a very powerful technique that every computer scientist should have in his toolbox.

Central Questions

- What is the matrix-chain multiplication problem? Explain at a small example.
- How would a brute force search for this problem look like?
- How does the dynamic programming solution avoid solving exponentially many subproblems? Formulate the crucial property that the matrix-chain multiplication problem has.
- The DP recurrence only gives the value of the optimal solution; how can we get the solution itself? Explain on the matrix example, but try to highlight the general pattern.

Optional Reading Material 21.1 Section 15.1 (pages 360–370), [Cor+09]

In this section, the authors devise in great detail a DP solution for the problem of cutting rods to maximize profit. If you are new to dynamic programming, consider reading this section first; if you are already familiar with the main concept, Reading Material 21.2 is sufficient.

Reading Material 21.2 Sections 15.2 and 15.3 (pages 370–390), [Cor+09]

The problem of optimal parenthesization of an (associative) matrix product is introduced and solved by dynamic programming. Then along this example, general sufficient conditions are discussed for a problem to be amenable to a DP solution. Moreover the authors compare the two alternative algorithm schemes for DP solutions: filling a table bottom-up or using a top-down recursion with *memoization*.

▶ Lesson 22: *Algorithm Design Patterns II — Greedy Algorithms*

3 h careful reading; plus time for exercises

Whereas the dynamic programming method, covered in Lesson 21, allows to devise (polynomial-time) solutions for a large class of problems, the resulting algorithms are often too expensive in practice, often both in terms of time and space requirements. For certain problems, the simpler, and more efficient, *greedy strategy* works, as well: Choose a locally optimal part of the solution, keep this part fixed for good, and continue on the remaining subproblem.

The greedy method is another important technique for the toolbox of every computer scientist. In this lesson, you learn when exactly this myopic strategy yields optimal solutions.

? Central Questions

- What is the activity-selection problem?
  Give an example and draw a time-line plot for the activities.
- How does the greedy algorithm fo activity selection work?
- Why is it sufficient to always choose the compatible activity with earliest end time?
  Argue in your own words.
- Does greedy always work?
  Give an example of a problem that can be solved using dynamic programming, but not greedily.
- What is the definition of a matroid?
- What are two examples of matroid structures?
- How does the generic greedy algorithm on weighted matroids work?
- What is the matroid underlying the generic MST-growing algorithm?
- Which algorithm (Kruskal or Prim) corresponds to the generic greedy on the MST matroid?

🕮 Reading Material 22.1

Sections 16.1 and 16.2 (pages 414–428), [Cor+09]

The authors describe a greedy algorithm for the activity-selection problem, starting from a DP solution, which is then refined by noting that the locally optimal "greedy" choice suffices. Then, the general paradigm of designing greedy algorithms is presented.

🕮 Reading Material 22.2

Section 16.4 (pages 437–443), [Cor+09]

This section gives an introduction to matroid theory and shows that a generic greedy algorithm can be used to compute maximal independent sets in any weighted matroid.

Reading Material 22.3 Sections 23.1 and 23.2 (pages 624–638), [Cor+09]

This section covers the two methods, Kruskal's algorithm and Prim's algorithm, for computing minimal spanning trees that you already encountered in Lesson 16. The reason for this second visit to MSTs is that Kruskal's and Prim's algorithms are the most classic applications of matroid theory introduced in Reading Material 22.2.

You can skip the paragraphs explaining the MST problem and the detailed example executions if you carefully worked through Lesson 16, but try to formulate the Prim's and Kruskal's algorithms in matroid jargon.

▶ Lesson 23: *Complexity Theory I — Motivation and Introduction to NP*

30 min reading

This lesson motivates the study of complexity theory and informally explains what "hard" problems are. It also sketches approaches to deal with hardness.

You may skip this lesson, if you already had a formal course on complexity theory, but it is highly recommended for everyone to skim the reading material.

? Central Questions

- What are conceptually hard problems, what are computationally hard problems according to Rawlins? Give examples.
- What does "fast algorithm" or "efficient algorithm" in the context of $\mathcal{NP}$ mean? Be precise.
- What exactly does it mean if a problem is $\mathcal{NP}$-complete? Why is it plausible then that no fast algorithm exists for the problem? Has it been proven that no fast algorithm can possibly exist?
- What is the advantage for practitioners to know about $\mathcal{NP}$-hard problems?

Reading Material 23.1 Section 1.9 (pages 52–58), [Raw92]

This section gives a brief and informal introduction to $\mathcal{NP}$-completeness and also sketches ways to treat hard problems in practice: approximation and probabilistic algorithms.

▶ Lesson 24: *Complexity Theory II — Definitions*

2.5 h careful reading; plus time for exercises

The notions informally discussed in Lesson 23 have to be made precise before we can properly deal with $\mathcal{NP}$-completeness. That will be done in this lesson. Unless you have already had a mathematically precise, formal treatment of complexity theory, it is highly recommended you go carefully through the material of this lesson.

The theory of $\mathcal{NP}$-completeness is very sensitive to details in the definitions, which might not be obvious at first sight. The content of this lesson is thus an important achievement of theoretical computer science, even though it is mainly definitions, and hardly any results. (Those will follow, of course, in the coming lessons.)

The material in this lesson is split into small units to make it easy to come back an rehearse some of the definitions during the following lessons on $\mathcal{NP}$-completeness.

? Central Questions

- What is an (input) encoding and why is it needed?
  Does it matter which encoding is used for the question whether a problem is in $\mathcal{P}$?
- What is the formal definition of the class $\mathcal{P}$ (as given in Cormen et al.)?
- In the language-theoretic framework, what is the difference between *accepting* a language L and *deciding* L?
  Does the difference matter for $\mathcal{P}$?
- What does it mean to verify a language L?
  What is the relation to accepting and deciding L?
- What is the formal definition of the class $\mathcal{NP}$ (as given by Cormen et al.)?
- What do the abbreviations $\mathcal{P}$ and $\mathcal{NP}$ stand for?
- What does it mean that one problem or language is polynomial-time reducible to another? Give the formal definition and the intuition behind it.
- What is the definition of $\mathcal{NP}$-complete?
  Is every $\mathcal{NP}$-complete problem also $\mathcal{NP}$-hard? How about the converse?
- Which of the following statements are true?
  - $\mathcal{NP}$-complete problems are not polynomial-time solvable.
  - Any $\mathcal{NP}$-complete problem is polynomial-time solvable if, and only if, $\mathcal{P} = \mathcal{NP}$.
  - Any problem in $\mathcal{NP}$ is polynomial-time solvable if, and only if, $\mathcal{P} = \mathcal{NP}$.

🕮 Reading Material 24.1

Section 34.1 (pages 1048–1061), [Cor+09]

The chapter starts with an overview of the theory of $\mathcal{NP}$-completeness, omitting formal definitions initially; those are then provided in the following sections. The first section defines what precisely a polynomial-time computable problem is and defines the complexity class $\mathcal{P}$.

🕮 Reading Material 24.2

Section 34.2 (pages 1061–1066), [Cor+09]

This section defines the concept of acceptance with a certificate, also called verification of a language, and the complexity class $\mathcal{NP}$ based on that.

Reading Material 24.3 Section 34.3 (pages 1067–1070), [Cor+09]

The third section defines the notion of polynomial-time reductions as a means to compare the complexity of two problems, which serves as basis to define the class of $\mathcal{NP}$-complete problems.

▶ Lesson 25: *Complexity Theory III — Cook's Theorem*

30 min careful reading

In this lesson, we will meet a first $\mathcal{NP}$-complete problem, which is subsequently used as reduction partner in $\mathcal{NP}$-hardness proofs.

Central Questions

- What is circuit satisfiability?
- Why is circuit satisfiability in $\mathcal{NP}$?
- Why is it $\mathcal{NP}$-hard? Describe the high-level idea of the proof.

Reading Material 25.1 Section 34.3 (pages 1070–1078), [Cor+09]

The section "Circuit satisfiability" introduces the satisfiability problem for boolean circuits and (informally) argues that this problem is $\mathcal{NP}$-complete.

▶ Lesson 26: *Complexity Theory IV — Famous NP-complete problems*

3h careful reading; plus 6h for exercises

In this last lesson on complexity theory, we will prove that several natural, practically relevant problems are in fact $\mathcal{NP}$-complete Knowing a handful of famous $\mathcal{NP}$-complete problems is important for (at least) two reasons: First, for the problems themselves, since any attempt to solve them by a simple polynomial-time algorithm is futile, a programmer had better recognize them, when facing them. Second, for showing that new problems are hard, the choice of a suitable reduction partner can simplify the arguments a lot.

Understanding, and checking, $\mathcal{NP}$-completeness when you see them is one thing; coming up with the idea of a hardness reduction on your own is totally different undertaking. Therefore it is indispensable to work on some exercises on your own.

Central Questions

- How do we prove that a new problem B is $\mathcal{NP}$-complete?
  Sketch the steps of the proof.
- What are the options for showing that B is in $\mathcal{NP}$?
- What are the options for showing that B is $\mathcal{NP}$-hard?

- In polynomial-time reductions, what exactly has to be polynomial?
  (It is several things; the answer makes a helpful checklist for $\mathcal{NP}$-hardness proofs.)
- What is the input, and what is the question asked, in each of the problems studied in Reading Material 26.1, i. e., the problems
  - SAT
  - 3CNF-SAT
  - Clique
  - Vertex Cover
  - Hamiltonian Cycles
  - TSP

Reading Material 26.1 Sections 34.4 and 35.5 (pages 1078–1097), [Cor+09]

In these pages, the authors prove $\mathcal{NP}$-completeness of a handful of the most famous hard problems: boolean formula satisfiability (SAT), satisfiability of formulas in conjunctive normal form with three literals per clause (3CNF-SAT), and for the graph problems Clique, Vertex Cover, Hamiltonian cycles and the traveling salesman problem (TSP).

The material is quite tough to grasp on first encounter, and it is a lot. As each of the completeness proofs can be read independent of the others, it is advisable to read this section in several sessions.

References

[Cor+09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. 3rd. MIT Press, 2009. ISBN: 978-0-262-03384-8.

[Knu98] Donald E. Knuth. *The Art Of Computer Programming: Searching and Sorting*. 2nd. Addison Wesley, 1998, p. 780. ISBN: 978-0-20-189685-5.

[Neb12] Markus E. Nebel. *Entwurf und Analyse von Algorithmen*. Wiesbaden: Springer-Vieweg, 2012. ISBN: 978-3-8348-1949-9. DOI: 10.1007/978-3-8348-2339-7.

[Raw92] Gregory J. E. Rawlins. *Compared to What? An Introduction to the Analysis of Algorithms*. Computer Science Press, 1992. ISBN: 0-7167-8243-X.

[SF13] Robert Sedgewick and Philippe Flajolet. *An Introduction to the Analysis of Algorithms*. 2nd. Addison-Wesley Professional, 2013. ISBN: 0-321-90575-X.

[SW11] Robert Sedgewick and Kevin Wayne. *Algorithms*. 4th. Addison-Wesley, 2011. ISBN: 978-0-321-57351-3.