Algorithmen und Datenstrukturen

PVW-Skript

Leonardo Del Giudice, Soel Micheletti[1], Jonas Meier[2],
François Hublet, Simone Guggiari

December 28, 2021

[1]Kursleiter. Kontakt: `msoel@ethz.ch`.
[2]Kursleiter. Kontakt: `jonmeier@ethz.ch`.

Contents

Über den Workshop und dieses Skript

Dieses Skript wird uns durch den Prüfungsvorbereitungsworkshop für "Algorithmen und Datenstrukturen" führen. Das Material, das in den ersten vier Kapiteln angeführt wird, enthält das Wichtigste aus den 14 Vorlesungen des Semesters, welches auch im Laufe des PVWs wiederholt wird. Das Motto: Alles, was in diesem Skript zu finden ist, sollte jeder bei der Prüfung wissen.

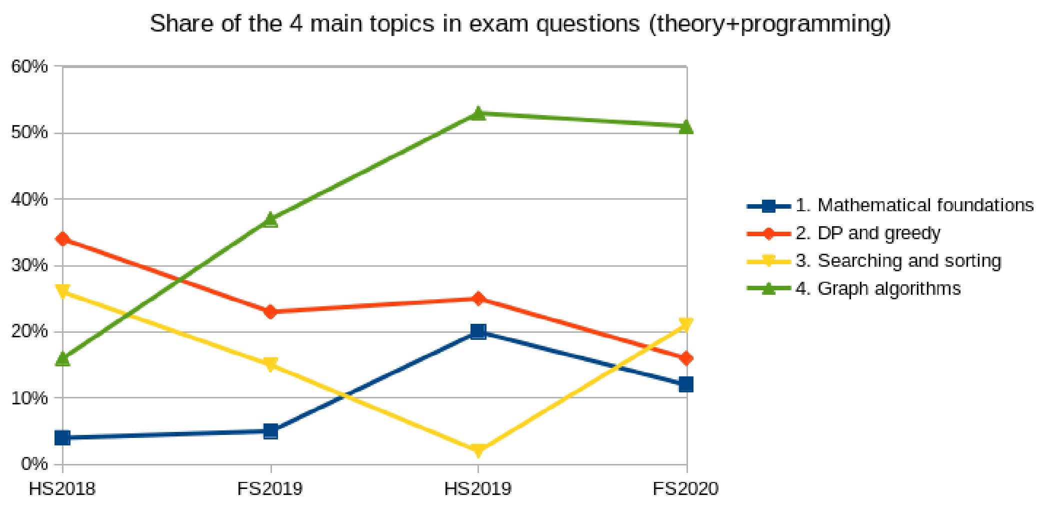
Jedoch erhebt dieses Dokument keinen Anspruch, das offizielle Skript des Kurses zu ersetzen; insbesondere werden detaillierte Beschreibungen und Beweise meist beiseitegelassen; bei Bedarf wird auf das offizielle Skript und die Vorlesungsnotizen verwiesen. Die Reihenfolge entspricht der des PVWs, nicht notwendigerweise der der Vorlesungen:

- Montag: Mathematische Grundlagen (Kap. 1) + Suchen und Sortieren (Kap. 3);
- Dienstag: DP und Greedy (Kap. 2) + Programmierung.
- Mittwoch: Fortsetzung DP + Datenstrukturen (Kap. 3);
- Donnerstag: Graphenalgorithmen (Kap. 4);
- Freitag: Fortsetzung Graphenalgorithmen und Programmierung.

Dieses Skript enthält für jedes Kapitel auch zusätzliche Übungen. Nach einer Wiederholung der wichtigsten Inhalte in der ersten Stunde des PVWs widmen wir die zweite Stunde gemeinsam zu lösenden Übungen (im folgenden als "Exercises" gekennzeichnet); in der dritten Stunde werden frühere Prüfungsaufgaben besprochen, die je für den folgenden Tag vorzubereiten sind (in diesem Skript als "Exam questions" markiert). Zusätzlich beschäftigen wir uns am Dienstag und Freitag mit dem Programmierung-Teil der Prüfung.

Im Laufe der Woche werdet die Studierenden darum gebeten, ihre Fragen zu den Programmieraufgaben, die im Laufe des Semesters zu lösen waren, dem Kursleiter per E-Mail zuzuschicken. Diese Fragen werden das Material für die Freitagsstunde liefern, in der diese besprochen werden sollen.

F.H.



Chapter 1

Mathematical foundations

1.1 Asymptotic notation (Landau's notation)

The formal definitions of the various asymptotic notations are given below. Each subsection provides both the definition of the set of functions and the relationship between two sets.

Upper bound (big-O)

$$\mathcal{O}(g) := \{f : \mathbb{N} \to \mathbb{R}^+ \mid \exists c \in \mathbb{R}^+, n_0 \in \mathbb{N}, \forall n \geq n_0 : f(n) \leq c \cdot g(n)\}$$

$$\mathcal{O}(f) \leq O(g) \Leftrightarrow \exists c, n_0. \forall n \geq n_0. f(n) \leq c \cdot g(n)$$

Lower bound (big-Omega)

$$\Omega(g) := \{f : \mathbb{N} \to \mathbb{R}^+ \mid \exists c \in \mathbb{R}^+, n_0 \in \mathbb{N}, \forall n \geq n_0 : f(n) \geq c \cdot g(n)\}$$

$$\Omega(f) \geq \Omega(g) \Leftrightarrow \exists c, n_0. \forall n \geq n_0. f(n) \geq c \cdot g(n)$$

Tight bound (big-Theta)

$$\Theta(g) := \{f : \mathbb{N} \to \mathbb{R}^+ \mid \exists c_1, c_2 \in \mathbb{R}^+, n_0 \in \mathbb{N}, \forall n \geq n_0 : 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)\}$$

$$\Theta(f) = \Theta(g) \Leftrightarrow \exists c_1, c_2 n_1, n_2. \forall n \geq n_1. f(n) \leq c \cdot g(n) \wedge \forall n \geq n_2. f(n) \geq c \cdot g(n)$$

Therefore:

$$\Theta(f) = \Theta(g) \Leftrightarrow O(f) \leq O(g) \text{ and } \Omega(f) \geq \Omega(g)$$

Limits of quotients and asymptotic relations

Let $f, g : \mathbb{R} \to \mathbb{R}^+$ such that the limit of $\frac{f}{g}$ exists. Then:

$$\lim_{x \to \infty} \frac{f}{g} = \infty \Rightarrow g \in \mathcal{O}\left(f\right) \text{and} f \in \Omega\left(g\right)$$

$$\lim_{x \to \infty} \frac{f}{g} = C \in \mathbb{R}^+ \setminus \{0\} \Rightarrow f \in \Theta\left(g\right) \text{ and } g \in \Theta\left(f\right)$$

$$\lim_{x \to \infty} \frac{f}{g} = 0 \Rightarrow f \in \mathcal{O}\left(g\right) \text{ and } g \in \Omega\left(f\right)$$

Master theorem

Let $T : \mathbb{N} \to \mathbb{R}^+$ be a non-decreasing function such that for all $k \in \mathbb{N}$ and $n = 2^k$,

$$T(n) \leq aT\left(\frac{n}{2}\right) + O\left(n^b\right)$$

for some $a \in \mathbb{R}^+$, $b \in \mathbb{R}$. Then

- If $b > \log_2(a)$, $T(n) \in O\left(n^b\right)$;
- If $b = \log_2(a)$, $T(n) \in O\left(n^{\log_2 a} \cdot \log n\right)$;
- If $b < \log_2(a)$, $T(n) \in O\left(n^{\log_2 a}\right)$.

Equivalents of sums of n^a, $a > 0$

For $a > 0$, let

$$u_n^a = \sum_{i=0}^{n} i^a.$$

We have the asymptotic tight approximation

$$u_n^a = \Theta\left(n^{a+1}\right).$$

Notion of (temporal) complexity

The time complexity (short: complexity) of an algorithm A is the number of elementary operations it takes to execute A. The complexity is generally expressed a function of some measure n of the input, often its size, sometimes its value. As A applied to different entries of the same size n can have different runtimes based on which input it is given, we need to distinguish between

- Worst-case complexity (**default**): the maximum number of elementary operations necessary for any input of size n,
- Average-case complexity: the average number of elementary operations necessary for inputs of size n,
- Best-case complexity: the minimum number of elementary operations necessary for any input of size n.

1.2 Important formulae

Following some important definitions, identities and formulas used in induction and combinatorial problems are given. It is best to know those by heart.

1.2.1 Combinatorics

Factorial

$$n! = n \cdot (n-1) = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot n \quad \forall n \in \mathbb{N} \qquad\qquad 0! = 1$$

Binomial coefficient

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

Useful identities

$$\binom{n}{0} = \binom{n}{n} = 1 \qquad \binom{n+1}{k+1} = \binom{n}{k} + \binom{n}{k+1} \qquad \binom{n}{n-k} = \binom{n}{k}$$

1.2.2 Summation - Geometric sum/series

$$\sum_{i=0}^{n} x^i = \frac{x^{n+1}-1}{x-1}$$

1.2.3 De l'Hôpital rule

Let $f, g : \mathbb{R} \to \mathbb{R}$ be differentiable functions with $f(x) \to \infty, g(x) \to \infty$ for $x \to \infty$. If $\lim_{x\to\infty} \frac{f'(x)}{g'(x)}$ exists, then

$$\lim_{x\to\infty} \frac{f(x)}{g(x)} = \lim_{x\to\infty} \frac{f'(x)}{g'(x)}$$

1.3 Logic and proofs

1.3.1 How to write good proofs

Good proofs are:

- Always provided—a result without a proof is almost like no result at all;
- Honest, readable, clear and concise—there is no practical difference between the corrector not being able to read or understand your proof and your proof being wrong;
- Full sentences, not simply a sequence of equations or computations without explanations;
- Not mixing abbreviations/math with English/German text (write "for all", "there exists" in text, not "$\forall$", "$\exists$"), and write computations on separate lines when useful;
- Equipped with an introduction sentence stating what is proven and ending with a conclusion sentence recalling what has been proved;
- Avoiding expressions that suggest that some result if evident or needs not be proven at all, except in the rare cases when it obviously is the case ("it is clear", "it is trivial", "we clearly see that", "this is simple"...)—these formulations are commonly misused by people not wanting or not able to prove the underlying statement;
- Written from top to bottom and from left to right which means:
  1. (Always!) state the hypothesis A,
  2. State the theorem, result or lemma that proves $A \Rightarrow B$,
  3. Conclude that B holds.

  In particular, sequences of equations aiming at proving some equality or inequality generally start from what is known to arrive at what is to be proven. If one side of an equation has to be developed to obtain the other side, then start from this one side to arrive at the other.
  e.g. To prove $(a+b+c)^2 = a^2+b^2+c^2+2(ab+bc+ca)$, do

$$\begin{aligned}(a+b+c)^2 &= (a+b+c)\cdot(a+b+c)\\ &= a^2+ab+ac+ba+b^2+bc+ca+cb+c^2\\ &= a^2+b^2+c^2+2ab+2bc+2ca\\ &= a^2+b^2+c^2+2(ab+bc+ca)\,;\end{aligned}$$

  NOT

$$\begin{aligned}(a+b+c)^2 &= a^2+b^2+c^2+2(ab+bc+ca)\\ a^2+ab+ac+ba+b^2+bc+ca+cb+c^2 &= a^2+b^2+c^2+2ab+2bc+2ca\\ a^2+b^2+c^2+2ab+2bc+2ca &= a^2+b^2+c^2+2ab+2bc+2ca\\ 0 &= 0.\end{aligned}$$

1.3.2 The induction principle

Proofs by induction are used to prove results of the type "for all n, $A(n)$" where $A(n)$ is some statement depending on n. Of course, n might be any positive integer, but it could also be more specifically a power of two, a multiple of some number etc. depending on the structure of the problem.

When writing inductive proofs, it is essential to keep in mind their specific structure; the following is for a statement of the type "for all $n \in \{1, 2, \dots\}$, $A(n)$":

0. **Induction statement**: define $A(n)$.

$\frac{1}{2}$. "Let us prove by induction over n..."

1. **Base case**: prove $A(1)$ or $A(0)$ (and in some particular cases more than one base case is needed).

2. **Induction step**: prove $A(k) \Rightarrow A(k+1)$
   (a) State **induction hypothesis** ("assume $A(k)$");
   (b) State **what is to be proven** ("let us prove $A(k+1)$");
   (c) Prove $A(k+1)$.

3. (recommended) **Conclusion** sentence.

Note that both part of an induction proof (base case and induction step) are equally important: leaving out one of them immediately makes the proof invalid.

1.3.3 Proofs by contradiction (indirect proofs)

In a proof by contradiction, we start by stating the opposite of our claim and deduce a logical contradiction from it. We then conclude that our claim has to be true.

The structure of a proof by contradiction is:

0. Let us prove A.

1. "Assume by contradiction that $\neg A$".

2. Derive a contradiction.

3. "As this is a contradiction with..., we have proven A".

1.4 Exercises

1. **Asymptotics**
   (a) Given the following functions

   $$n^5 + n \quad \log n^4 \quad \sqrt{n} \quad \binom{n}{3} \quad 2^{16} \quad n^n \quad n! \quad \frac{2^n}{n^2} \quad \log^8 n$$

   find an order for which it holds that if a function. f is to the left of g, then g grows strictly faster than f

   (b) True or false?
      i. $\log_2\left(n^{1000}\right) \in O\left(\log_{10}\sqrt{n}\right)$;
      ii. $n^4 \in \Omega\left(\binom{n}{4}\right)$;
      iii. $\log\log n \in \Omega\left(\log^2 n\right)$;
      iv. $e^{\sqrt{\ln n}} \in O\left(\sqrt{e^{\ln n}}\right)$;
      v. For all $a > b$, $n^b \in O\left(n^a\right)$;

vi. For all $a > b$, $e^{bn} \in O\left(e^{an}\right)$;
vii. For all $a > b$, $\log bn \in O\left(\log an\right)$;
viii. Repeat the last three questions with Θ instead of O;
ix. There exists $b > 0$ such that $n! \in \Omega\left(n^b\right)$;
x. There exists $b > 0$ such that $n! \in O\left(n^b\right)$;
xi. There exists some function f such that $f \in \Omega\left(n \log n\right)$ and $f \in O\left(n^2\right)$,but $f \neq n \log n$ and $f \neq n^2$.

(c) Give the simplest tight bound for the following formulae:
i. $P\left(n\right) = 15n^{41} + 14n^{42} + \log n$;
ii. $Q\left(n\right) = n^{41} + n^{42} + e^{5n}$;
iii. $R\left(n\right) = \frac{n^2+13}{n^3+n+8} + n^{-2}$;
iv. $S\left(n\right) = \sqrt{e^{\ln n}}$.

2. **Proofs by induction**

(a) Prove the following variant of the geometric sum above:

$$1 + a + a^2 + \cdots + a^k = \frac{a^{k+1} - 1}{a - 1}.$$

(b) Prove that for all $n \geq 1$, the following identity holds:

$$\sum_{i=1}^{n} i(i+1) = \frac{n(n+1)(n+2)}{3}.$$

(c) Prove *Bernoulli's Inequality* i.e., for all $n \in \mathbb{N}_0$ and $x \in \mathbb{R}, x > -1$,

$$(1+x)^n \geq 1 + nx$$

holds.

(d) Prove that for all $n \in \mathbb{N}^+$ following equation holds:

$$\sum_{i=1}^{n} \frac{1}{i(i+1)} = \frac{n}{n+1}.$$

(e) Prove that for all $n \in \mathbb{N}^+$following equation holds:

$$\sum_{i=1}^{n} (2i-1) = n^2.$$

(f) Prove that for any $n \in \mathbb{N}$, $n^3 + 2n$ is divisible by 3.

3. **Recursion and induction**

(a) Let $k \in \mathbb{Z}$. We define the following integer sequence u:

$$\begin{aligned} u_0 &= 1 \\ u_1 &= k \\ u_n &= 2u_{n-1} - u_{n-2} && n \geq 2. \end{aligned}$$

Prove that for $n \geq 1$,

$$u_n = nk - (n-1).$$

(b) Given is the following conditional recursive function:

$$T(n) = \begin{cases} 5 \cdot T\left(\frac{n}{7}\right) + 8, & \text{if } x > 1 \\ 3, & x = 1 \end{cases}$$

Find a closed formula for $T(n)$ and prove it using complete induction. You may assume that n is a power of 7.
Now replace 7 by 2 in the definition of T. Could we use a theorem from the course to obtain an upper bound without induction?

1.5 Exam questions

- FS 2020: **T1.a), T2.a)-c)**;
- HS 2019: T1.a), T1.g), T2.a)-b), T2.d)
- FS 2019: T1.e)-f);
- HS 2018: T1.h)-i);
- HS 2017: **T1.i)-l)**;
- HS 2016: T1.j)-m).

Chapter 2

Algorithmic methods: DP and Greedy

2.1 Dynamic Programming

Dynamic programming is a technique that allows to solve certain problems with exponential-time naive solutions in usually polynomial (order of $O(n^a)$ for some $a \in \mathbb{N}$) or pseudopolynomial (order of $O(N^a)$ for some $a \in \mathbb{N}$, where N is the value of an input parameter) time. It achieves this by avoiding to recompute subproblems multiple times, instead saving those intermediate results in either a table T or a memo map M, depending on the approach.

DP approaches with *tables* vs. *memoization* are generally equivalent:

- With a table, you generally solve problems bottom-up: all subproblems have to be solved and complexity analysis is simpler;
- With memoization, you generally solve problems top-down; only useful subproblems must be computed and writing the solution is often easier, but recursion can lead to some time overhead[1].

Below, we present two pseudo-codes for computing the Fibonacci number F_n for $n \geq 0$ with tables and memoization.

Algorithm 1 Fibonacci numbers – bottom-up/imperative/table

```
function F($n$)
    if $n = 0$ then
        return $0$
    $v \leftarrow$ new int $[n+1]$
    $v[0] \leftarrow 0$
    $v[1] \leftarrow 1$
    for $i \in \{2, \ldots, n\}$ do
        $v[i] \leftarrow v[i-1] + v[i-2]$
    return $v[n]$
```

2.1.1 Structure

In an exam, it is important to follow this structure when solving a DP problem. The structure allows to correctly describe the solution to a DP problem and makes it easier to understand and implement.

- Definition of table:
  - Dimensions and index range (starting at 0 or 1?),
  - Meaning of entry and type;

[1]The idea that recursive implementations are necessarily less efficient than non-recursive ones is often exaggerated, especially given the capabilities of modern compilers and/or of functional programming languages. Still, this might be a real issue in some cases.

Algorithm 2 Fibonacci numbers – top-down/recursive/memoization

```
$m \leftarrow$ new map()
function F($n$)
    if $n \leq 1$ then
        return $n$
    if $n \in m$.keys() then
        return $m[n]$
    $r \leftarrow$ F$(n-1)$ + F$(n-2)$
    $m[n] \leftarrow r$
    return $r$
```

- Computation of entry:
  - Initialization,
  - Recursive formula: how to compute an entry from previous ones;
- Calculation order:
  - Dependencies: on which previous entries does a new entry depend?
  - Which global order ensures this?
- Extract solution:
  - How to extract solution value once table has been filled,
  - How to extract complete value (sequence, subset...);
- Running time

An easy way to remember all the points is the abbreviation SMIROST, each letter of which corresponds to one point above: <u>S</u>ize, <u>M</u>eaning, <u>I</u>nitialization, <u>R</u>ecursive formula, <u>O</u>rder, <u>S</u>olution, <u>T</u>ime.

Note that 'size' (sometimes also called 'dimension') includes how many vector-space dimensions the table has (e.g. 2D) but also the size of each dimension (e.g. $n \times m$) and the range of the index in each dimension (e.g. $i \in \{0, \dots, n-1\}$, $j \in \{1, \dots, m\}$) whereas 'meaning' includes both the informal meaning of one entry relative to its index (e.g. $T[i, j] =$ how many combinations... as well as the entry type (e.g. $T[i, j]$: `int`).

2.1.2 How to solve DP exercises?

There are a few steps that you may want to follow:

1. Read the task at least twice: What is given? What is to be computed?
2. In a typical DP task, we must first generalize the problem and identify subproblems.
   (a) What are potential subproblems? Typical cases are: all left or right subarrays instead of the whole subarray; all $k \leq n$ instead of only n; all nodes except only start or target node...
   (b) Describe the subproblems. How many are there? (n) Is there a subproblem that exactly corresponds to the original problem?
   (c) How do the subproblems depend upon one another? What must be computed first, what must be computed later? How much does an update cost? (k)
   (d) Is backtracking useful?
3. Follow the 6 steps from the previous subsection to describe your algorithm, including its correctness and complexity (in general, the complexity ist $k \cdot n$).

2.1.3 Worst-case asymptotic runtime

It is equal to the size of the DP table (in 2D: $r \cdot c$ where r is the number of rows and c the number of columns) multiplied by how long it takes to compute one entry in the worst case. You also have to add the time to extract the solution. In 2D, this is:

$$O(r \cdot c \cdot \text{entry} + \text{extraction})$$

2.1.4 Complexity: caveats

The phrase 'the complexity of algorithm X' is in general a shortcut for 'the worst-case asymptotic runtime complexity of algorithm X': other cases (average complexity, space complexity etc.) are generally marked as such.

Moreover, it is essential to distinguish clearly between polynomial and pseudopolynomial runtimes. An algorithm is polynomial iff it computes its output in time polynomial in the **size of its input**. An algorithm is pseudopolynomial iff it computes its output in time polynomial in the **value of one of its input parameters**.

Consider the following naive algorithm:

Algorithm 3 Find smallest prime divisor of n

```
$i \leftarrow 2$
while $i \leq n$ do
    if $n$ is a multiple of $i$ then
        return $i$
    else
        $i \leftarrow i + 1$
```

Does this algorithm have polynomial runtime? This algorithm takes as input an integer n. This means that the size of its input, coded as a binary number, is $\lfloor \log_2 n \rfloor$ bits. Now, in the worst case (if n is prime), it needs n iterations of the while-loop, each of which has constant time complexity. As a consequence, the overall time complexity is $\Theta(n)$. But n is not polynomial in $\lfloor \log_2 n \rfloor$ (actually, it is exponential in $\lfloor \log_2 n \rfloor$), so the algorithm is not polynomial. Nevertheless, it is pseudopolynomial, because n is polynomial (linear) in the value n of the input.

2.1.5 Types of tables

You can have 1D, 2D or even nD tables, depending on the problem. In the exam, DP problems often tend to follow some classic schemes; once you know these, all other can be seen as reformulations of the same problem.

To know how to set up your table, try to see how you can use the solution of a subset of what you are trying to solve to derive your solution.

One good approach is to proceed incrementally, i.e. consider solving the problem under the assumption you can only use one single element, then extend it to two, and so on, each time using the previous result to avoid unnecessary calculations.

The trick lies in seeing which previous entries you need to use. Depending on the problem, you might want to move *left* and ev. *up* from your current cell to retrieve the needed values. In some problems the amount of cells to move has to be computed, as it might depend on the entry.

Usually, the DP table either contains elements of type `int` or `bool`; the former when we want to determine some form of cost, for example when we have to select a series of elements to take; the latter usually indicates if it is possible to create a desired sequence with a subset of the elements available.

2.1.6 Backtracking

It is the technique of extracting the solution from the DP table by following the cells in reverse order from the final computed cell. This is useful in problems of the type "find a path that minimizes some cost". Once you find the minimal total cost, you backtrack from there, saving in reverse order the path to follow, which you will output when you reach the beginning.

2.1.7 Keep in mind

In DP problems it is normally asked to *describe* an algorithm to compute what is required. Therefore, the emphasis is on *correctness, clarity and precision*, and **not** on actual implementation. Keep the answer length to a minimum! Focus on how to compute an entry once you have explained the meaning and the base cases, so use concise mathematical notation and eventually case distinction. Something like

$$T[i,j] = \begin{cases} T[i-1,x] & \text{if} \ldots \\ T[i-1,j-x_i] & \text{if} \ldots \end{cases}$$

Also, remember to specify the ranges your indexes can take and the meaning of such a definition.

2.2 Greedy algorithm

Greedy is an algorithmic paradigm that follows the idea of making the optimal choice locally at each stage. A greedy algorithm is nothing more than a DP algorithm where each subproblem depends only on one other subproblem!

For many problems the greedy algorithm does *not* provide an optimal solution; however, when it does, its ease of implementation and short execution time offer a very powerful approach to combinatorial problems. In other cases, greedy algorithms can serve as cost-efficient approximation algorithms of complexer problems. The proposed solution might be very close or even the same of the optimal, based on the problem and the specific values.

2.3 Exercises

1. **Theory questions**

   (a) Name algorithms seen in the course that are DP or greedy algorithms.

   (b) Is the above algorithm for computing the Fibonacci sequence polynomial? pseudopolynomial?

2. **DP and greedy problems**

   For each of the following problems, design a DP and/or a greedy algorithm that solves it and discuss its time and space complexity. For greedy algorithms, also provide a correctness proof.

   (a) **Knapsack I**
   A set of n items is given, each one with a weight $w_i > 0$ and a value v_i. We are looking for a subset $S \subseteq \{1, ..., n\}$ of all elements such that the combined weight $\sum_{i\in S} w_i$ does not exceed a given max weight W and the total value $\sum_{i\in S} v_i$ of the selected items is maximized.

   (b) **Knapsack II**
   A set of n liquids is given, each one with a density $d_i > 0$ (in kg by liter), a value v_i (in CHF by liter) and a finite supply $s_i > 0$. We are looking for a choice of volumes $q_1 \leq s_1, \ldots, q_n \leq s_n$ for each liquid such that the combined weight $\sum_{i=1}^{n} d_i q_i$ does not exceed a given max weight W and the total value $\sum_{i=1}^{n} v_i q_i$ of the selected items is maximized.

   (c) **Coin Exchange**
   We are given a set $S = \{s_1, s_2, ..., s_n\}$ of coin values (e.g. in Swiss francs that would be $\{1, 2, 5\}$ if we don't consider cents) and we are also given an amount N. The problem asks in how many ways a cashier could give change for N CHF if he has as infinite disposal of coins of values in S.

   (d) **Lights**
   We are given a sequence of n bits $B = (b_1, b_2, ..., b_n)$, each of which encode the state of the i-th light l_i in a sequence of n lights (1 = on, 0 = off). We know that we can control the lights in two ways: either by performing one operation and changing the state of light l_i (flipping its bit), or to perform one *collective* operation up to light l_k (with the cost of 1 operation) and change the state of all the lights in $(l_1, ..., l_k)$. The goal of this problem is to find the minimum number of operations we have to perform to turn off all the lights.

(e) **Stairs**

Imagine having a stair with n steps, and a cute bunny that, starting from step 1, can run up the stair hopping either 1, 2 or 3 steps at a time. Count in how many different ways the bunny can run up the stairs.

(f) **Robot**

Given are two indices x and y of an $x \times y$ grid. You have a robot starting at position $(0, 0)$. This robot can only move along the positive axes, either one step right or one step up. Count how many different ways the robot has to reach position (x, y).

(g) **Line wrapping I**

Given a sequence of words $w_1, \dots, w_n$ of lengths $\ell_1, \dots, \ell_n$ separated by spaces and a maximal number of characters per line $L > \max_i \ell_i$, determine the positions at which to insert line breaks in order to print the text (without breaking any words) using as few lines as possible.

(h) **Line wrapping II**

Given the same $w_1, \dots, w_n$ and L, determine the positions at which to insert line breaks to minimize the sum of the squares of the number of remaining spaces at the end of each line.

(i) **Grammar parsing with Cocke-Younger-Kasami**

A context-free grammar in Chomsky Normal Form (CNF) is a sequence of rules that can have one of two forms:

i. $C \rightarrow \alpha$ where C is a 'category symbol' and α is a word, meaning that word α is matched by category C. For example, $Name \rightarrow \texttt{Alice}$ encodes the fact that the word `Alice` is a $Name$.

ii. $C \rightarrow AB$ where A, B and C are categories, meaning that a phrase of category C can be obtained by concatenating two phrases of categories A and B. For example $FullName \rightarrow Name\ Surname$ encodes the fact that a full name is a name followed by a surname.

The language of a category of the grammar is the set of all phrases matched by this category. For instance, in grammar

$$G = \{Name \rightarrow \texttt{Alice}, Surname \rightarrow \texttt{Mustermann}, FullName \rightarrow Name\ Surname\},$$

the language of category $FullName$ is $\{\texttt{Alice Mustermann}\}$. Given a grammar G with R categories, a category S in G and a phrase $p = w_1, \dots, w_n$ of length n, determine in time $\mathcal{O}(n^3 R)$ whether p is in the language of S.

Hint: Consider all subsequences of p.

2.4 Exam questions

- FS 2020: **P2**;
- HS 2019: T4.c), P2;
- FS 2019: **T3.a)-c)**, P1;
- HS 2018: **T2**, P2;
- HS 2016: T3.

2.5 Programming exercises

- HS 2018 P2

  Given an $M \times N$ matrix A filled with 0s and 1s, find the size of the largest square submatrix of zeros in A:

  – In time $\mathcal{O}(M^2 \cdot N^2)$ for 10/20 points,

  - In time $\mathcal{O}(M \cdot N \cdot \min(M, N))$ for 15/20 points,
  - In time $\mathcal{O}(M \cdot N)$ for 20/20 points.

- HS 2019 P2

  Given a string s and a list of words D, return the length of the longest initial sequence of s that can be split into words from D, possibly with repetitions:

  - Only single-letter words in D for 2/16 points,
  - In time $\mathcal{O}(|s| \cdot |D| \cdot \ell)$, where ℓ is the maximal length of a word in D, for 5/16 points,
  - In time $\mathcal{O}(|s| \cdot |D|)$, for 14/16 points[2],
  - In time $\mathcal{O}((|s| \cdot \ell \cdot \log |D| + |D| \log |D|)$ or $\mathcal{O}((|s| + |D|) \cdot \ell)$ for 16/16 points.

[2]This is in fact an average complexity which you are not required to prove here. For now, just focus on how to improve your algorithm from the previous question.

Chapter 3

Searching and sorting

3.1 Searching

In this section, we recall Abstract Data Structures (ADS) that allow for easy insertion, deletion and search of elements, as well as the corresponding algorithms.

The most common ADS and the corresponding runtime complexities are as follows:

Data structure	**Search**	**Insert**	**Delete**
Unsorted Array	$O(n)$	$O(1)$	$O(n)$
Sorted Array	$O(\log n)$	$O(n)$	$O(n)$
Unsorted List	$O(n)$	$O(1)$	$O(n)$
Sorted List	$O(n)$	$O(n)$	$O(n)$
Unbalanced Tree	$O(n)$	$O(n)$	$O(n)$
AVL tree	$O(\log n)$	$O(\log n)$	$O(\log n)$

We observe that there is generally a trade-off between simple data structures that allow for easy insertion and deletion but do not preprocess the entries in order to facilitate subsequent searches (unordered array) and more complex data structures with higher insertion and deletion costs that can be more efficiently queried.

3.1.1 Binary search

Binary search is the standard search algorithm for sorted arrays. It has an efficient (logarithmic) runtime complexity.

Algorithm 4 Binary search

```
function FindIndex($A$, $e$)                    ▷ Search item $e$ in sorted array $A$
    $l, r \leftarrow 0, A.\texttt{length} - 1$
    while $r > l$ do
        $m \leftarrow \left\lfloor \frac{l+r}{2} \right\rfloor$
        if $A[m] = e$ then
            return $m$
        else if $A[m] > e$ then
            $r \leftarrow m - 1$
        else
            $l \leftarrow m + 1$
    return "not found"
```

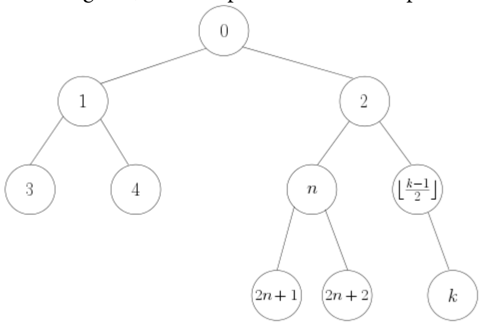
Though, with binary search, the cost of searching in ordered arrays becomes logarithmic, insertion and deletion are still linear: in the worst case, i.e. when the element to be inserted or deleted is the first element of the array, we have to shift all items one step to the right/to the left.

3.1.2 Heaps

Even more efficient than ordered arrays are therefore tree-based structures, for which the cost of insertion and deleting can also be made logarithmic. Heaps are a specific class of tree-based structures which provide an efficient (constant-time) way to extract the smallest or largest element.

More precisely, min (resp. max) heaps are tree-based data structures that satisfy the following heap invariant: if A is the parent of B, then the value of node A is smaller (resp. larger) than the value of node B. Here we will consider binary min-heaps, which means that a parent has value smaller than the value of its (at most two) children. In binary min-heaps, we additionally impose the following shape invariant: the heap we consider is always a complete binary tree, i.e. all layers of the tree are filled top-down and left-to-right.

Figure 3.1: Example of 0-based heap



Implementation

The most common implementation involves an array (of fixed or dynamic size). Assuming a 0-based array (cf. figure above), the *children* of node n are $2n + 1$ and $2n + 2$ and the *parent* of node k is node $\lfloor \frac{k-1}{2} \rfloor$.

Common operations

Here are the most common operations that a min-heap must support:

- **Basic operations**
  - Find min,
  - Delete min,
  - Insert,
  - Find min and delete it (pop);
- **Initialization**
  - Create empty heap,
  - Heapify (transform array into heap);
- **Inspection**
  - Return size,
  - Test if empty;
- **Other**
  - Increase/decrease,
  - Delete,
  - Restore heap invariant,
  - Merge/union.

Preserving the heap invariant

When updating a heap, it is essential to maintain the invariant described above. After a deletion or an insertion, it may happen that some (single) element becomes smaller that one of its children; we can then restore the heap invariant in logarithmic time by percolating this element down.

Algorithm 5 Restore heap invariant by percolating element down

```
function PercolateDown($H, i$)                ▷ Percolate element $i$ in $H$
    $e \leftarrow H[i]$
    if $2i + 2 = H.\texttt{length}$ then
        if $e > H[2i + 1]$ then
            Swap $H[i]$ and $H[2i + 1]$
    else if $2i + 2 < H.\texttt{length}$ then
        $l, r \leftarrow H[2i + 1], H[2i + 2]$
        if $l < r$ then
            if $l < e$ then
                Swap $H[i]$ and $H[2i + 1]$
                PercolateDown($H, 2i + 1$)
        else
            if $r < e$ then
                Swap $H[i]$ and $H[2i + 2]$
                PercolateDown($H, 2i + 2$)
```

Costs

The following table summarizes the costs of standard heap operations for three types of heaps: binary heaps, binomial heaps and Fibonacci heaps.

Operation	Binary	Binomial	Fibonacci
search min	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
delete min	$\Theta(\log n)$	$\Theta(\log n)$	$O(\log n)$
insert	$\Theta(\log n)$	$\Theta(1)$	$\Theta(1)$
increase key	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(1)$
union	$\Theta(m \log n)$	$O(\log n)$	$\Theta(1)$

3.1.3 AVL Trees

AVL trees are a special kind of binary search trees that guarantee that the tree is balanced and therefore has a worst-case access cost of $O(\log n)$. Keep in mind that **this does not hold for search trees** in general: the complexity of searching is proportional to the depth of the tree, which can be linear in degenerate cases. Therefore, an additional property has to be mantained to ensure a good worse-time complexity; this is achieved using what is called the *balance factor*.

Balance factor

We define the following quantities:

- $\mathsf{depth}(v) =$ distance of v to root (along unique path)
- $\mathsf{height}(T) = \max_{v \in V} \mathsf{depth}(v) + 1$

Then, to each node v, we can assign a number b called the balance factor:

$$b(v) := \mathsf{height}(v.R) - \mathsf{height}(v.L)$$

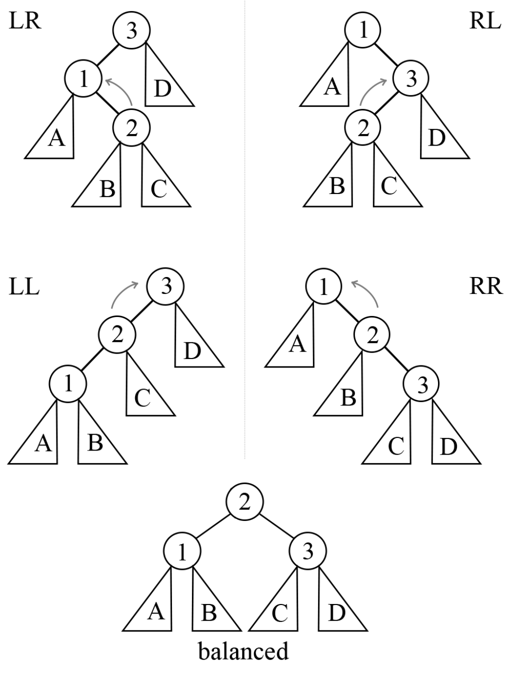
In AVL trees, we impose that this value be in the range $\{-1, 0, +1\}$; this implies that the tree is 'almost' balanced, yielding a logarithmic access time. If, after insertion or deletion, one of the factors is not in this range (i.e. it is ± 2), rotations have to be performed to restore the invariant.

Rotations

The following schemas should help you execute rotations on AVL trees to restore their invariant. We distinguish 4 cases: LR, RL, LL, RR. In the first 2 cases, we need to perform 2 rotations, the first of which brings us to LL or RR respectively. The second rotation then brings us to the final balanced configuration.

The arrows indicate which node "rotates" and how to connect subtrees A, B, C, D to keep tree properties. A key observation is that only one of these sequences of rotations has to be performed along the path from the inserted node to the root to restore the heap property.

Figure 3.2: All possible rotations and their next state



Operations and complexity

The tree utilizes exactly $\Theta(n)$ space. It supports the standard tree operations *search*, *insert* and *delete*, all of which have an average and worst case time complexity of $O(\log n)$.

3.2 Sorting

The following sorting algorithms should be considered standard; you should be able to execute them on paper, code them in Java or in pseudo-code and analyze them.

3.2.1 Bogo sort

This is a fairly inefficient sorting algorithm that generates a random permutation until the elements are sorted. An analogy with a deck of card would be to shuffle the deck, then check if it is sorted and loop if it is not.

This algorithm doesn't have a worst case asymptotic time as it is not guaranteed to terminate within a given time. It has an average runtime of $O(n \cdot n!)$.

Algorithm 6 Bogo Sort

```
**while** $\neg$isSorted $(A)$ **do**
    $A \leftarrow$ randomPermutation $(A)$
```

3.2.2 Bubble sort

Iteratively go over the array, always considering two neighbor cells. If they are not sorted, swap them, then continue with the next pair on the right. If you go over the whole array without performing a single swap, then you are done. The variable *swapped* is for this purpose.

Algorithm 7 Bubble sort

```
$swapped \leftarrow$ true
while $swapped$ do
    $swapped \leftarrow$ false
    for $i \in \{0, \ldots, A.\mathtt{length} - 1\}$ do
        if $A[i] > A[i+1]$ then
            Swap $A[i]$ and $A[i+1]$
            $swapped \leftarrow$ true
```

3.2.3 Insertion sort

This algorithm is very similar to how a human sorts a deck of cards. It proceeds by keeping a sequence of already sorted elements, and always considering the next element in the sequence. It then inserts this element in the right place within the sorted sequence while shifting all larger elements right.

Algorithm 8 Insertion sort

```
for $i \in \{0, \ldots, A.\mathtt{length} - 1\}$ do
    $v \leftarrow A[i]$
    $j \leftarrow i - 1$
    while $j \geq 0$ and $A[j] > v$ do
        $A[j+1] \leftarrow A[j]$
        $j \leftarrow j - 1$
    $A[j+1] \leftarrow v$
```

3.2.4 Selection sort

This algorithm also keeps a sequence of sorted elements, iteratively expanding it with the largest of the remaining unsorted elements. It then swaps this element with the one closest to the sorted elements. This effectively selects the next element, hence the name.

3.2.5 Quick sort

This algorithm works by recursively choosing a pivot (one element, usually taken at the first or last position in the array) and splitting all other elements with respect to it. It first creates sets S^- and S^+, in which all elements smaller/larger than the pivot are stored. It then calls itself recursively on those two sets, and after they are returned sorted, it simply concatenates them while adding the pivot in the middle. The recursion stops in the base case in which the array contains just one element. Note that an in-place implementation is also possible, which reduces the memory overhead. This algorithm has a worse upper bound in time of operations, but it is often used in practice, since the average time performs better that both merge and heap sort. Note that the $\cdot$ operator here means concatenation.

Algorithm 9 Selection sort

```
for $i \in \{0, A.\texttt{length} - 2\}$ do
    $m, v \leftarrow i, A[i]$
    for $j \in \{i+1, \ldots, A.\texttt{length} - 1\}$ do
        if $A[j] < v$ then
            $m, v \leftarrow j, A[j]$
    if $m \neq i$ then
        Swap $A[i]$ and $A[m]$
```

Algorithm 10 Quicksort

```
function QuickSort($A$)
    if $A.\texttt{length} \leq 1$ then
        return $A$
    Pick pivot $p \leftarrow A[0]$
    for $i \in \{1, \ldots, A.\texttt{length} - 1\}$ do
        if $A[i] \leq p$ then
            $S^{-}.\texttt{add}(A[i])$
        else
            $S^{+}.\texttt{add}(A[i])$
    $S^{+} \leftarrow$ QuickSort$(S^{+})$
    $S^{-} \leftarrow$ QuickSort$(S^{-})$
    return $S^{-} \cdot \{p\} \cdot S^{+}$
```

3.2.6 Merge sort

This algorithm also works recursively. It splits the input into two sets, then calls itself recursively on them. After the two sets are returned sorted, it merges them in linear time. The name derives from this last part of the algorithm.

3.2.7 Heap sort

This sorting algorithm works by inserting all keys into a min heap, and iteratively extracting the minimum (the root) in constant time and attaching it to the list of sorted nodes. It does this n times; the overall runtime depends on the complexity of standard heap operations.

3.2.8 Other sorting algorithms (not part of the exam)

Additionally, if you are interested in algorithms that achieve asymptotic runtime better than $O(n \log n)$ *on specific inputs*, namely linear time $O(n)$, you can check out the following:

- Radix sort;
- Bucket sort.

Note: These algorithms achieve better asymptotic time because they *do not* compare elements and know the range of the keys used. In general, the lower bound for sorting $\Omega(n \log n)$ still holds.

3.2.9 Costs of sorting algorithms

Classical algorithms

Here are reported the min and max costs for a few sorting algorithms, based on their input size n, as well the particular type of input that leads to that particular complexity.

Algorithm 11 Merge sort

```
function MergeSort($A$)
    if $A$.length $\leq 1$ then
        return $A$
    $n \leftarrow A$.length
    $n_1 \leftarrow \frac{n}{2}$
    $n_2 \leftarrow n - n_1$
    for $i \in \{0, \ldots, A$.length $- 1\}$ do
        if $i < n_1$ then
            $S^-$.add $(A[i])$
        else
            $S^+$.add $(A[i])$
    $S^+ \leftarrow$ MergeSort $(S^+)$
    $S^- \leftarrow$ MergeSort $(S^-)$
    $i \leftarrow 0$                                        ▷ Merge $S^+$ and $S^-$
    $j \leftarrow 0$
    while $i < n_1$ and $j < n_2$ do
        if $S^-[i] \leq S^+[j]$ then
            $R$.add $(S^-[i])$
            $i \leftarrow i + 1$
        else
            $R$.add $(S^+[j])$
            $j \leftarrow j + 1$
    Concatenate remaining elements to $R$
    return $R$
```

Algorithm 12 Heap sort

```
$H \leftarrow$ CreateMinHeap ()
for $i \in \{0, \ldots, A$.length $- 1\}$ do
    $H$.insert $(A[i])$
for $i \in \{0, \ldots, A$.length $- 1\}$ do
    $v \leftarrow H$.pop ()
    $R$.add $(v)$
return $R$
```

Table 3.1: Best and worst case costs for sorting algorithms

	bubblesort		**insertion sort**		**selection sort**		**quicksort**	
	b.c.	w.c.	b.c.	w.c.	b.c.	w.c.	b.c.	w.c.
# comparisons	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n \log n)$	$\Theta(n^2)$
# permutations	0	$\Theta(n^2)$	0	$\Theta(n^2)$	0	$\Theta(n)$	$\Theta(n)$	$\Theta(n \log n)$
corresponding order	A	B	A	B	A	C	C	C

- A = already ordered
- B = inverse order
- C = special order

Lower bound for sorting

Any general comparison-based sorting algorithm has a worst-case runtime complexity of at least $\Omega(n \log n)$.

3.2.10 Properties of sorting algorithms

Stability Two objects with equal keys appear in the same order in the (sorted) output as they appeared in the input.

In-place or in-situ describe those algorithms which transform the input using no auxiliary data structure, and therefore no additional memory space. In some cases, additional space to store a few variables is allowed.

Table 3.2: Properties of sorting algorithms

	bubble	**insert**	**select**	**quick**	**merge**	**heap**
stable	Y	Y	N	N	Y	N
in-situ	Y	Y	Y	Y	N	Y

3.3 Exercises

1. **Searching**

   (a) Prove that the complexity of binary search is $O(\log n)$.

   (b) Suppose we are given a 2D table T of size $n \times n$ whose elements are integers such that

   $$\forall 0 \leq i < j < n, \forall 0 \leq k < n, T[i,k] \leq T[j,k] \text{ and } T[k,i] \leq T[k,j],$$

   i.e. elements are sorted both vertically and horizontally. Design an algorithm to efficiently search an element in this table. What is its complexity?

   (c) Consider a sorted doubly-linked list (SDLL), that is an ADS in which each element keeps a pointer to the previous and next elements (or `NULL`, if there is no previous or next element) and all elements are smaller than their predecessors and larger than their successors. You can assume that an already constructed SDLL is provided to you. How to extract (read + delete) the min element of a SDLL? How to insert a new element while preserving the invariants? How to increase a key? Provide pseudo-code and discuss the complexity of these operations compared to the binary tree approach.

   (d) Give an example of a sequence of operations that results in a worse than logarithmic runtime complexity for insertion and retrieval with a simple binary search tree but is still logarithmic with an AVL tree.

2. **Sorting**

   (a) Run each of the above algorithms on the input

   $$[3, 1415, 926, 535, 897, 932, 384, 626, 433].$$

(b) Analyze the worst-case runtime complexity of bubble sort, insertion sort and quicksort in detail. Indicate the number of iterations of each loop and the cost of every instruction in the given pseudocode. Deduce the upper bounds above.

(c) Show that the halting condition in the while-loop of bubble sort is correct, i.e., show that whenever a sequence of $n - 1$ tests does not result in any swap, the array is completely sorted.

(d) Consider the following algorithm:

Algorithm 13 Gnome sort

```
$i \leftarrow 0$
**while** $i < A.\mathtt{length} - 1$ **do**
    **if** $A[i] \leq A[i+1]$ **then**
        $i \leftarrow i + 1$
    **else**
        Swap $A[i]$ and $A[i+1]$
        **if** $i > 0$ **then**
            $i \leftarrow i - 1$
```

i. Run this algorithm on the input from question (a). Mark 'phases' in which the pointer i is first decremented $O(n)$ times to position an element at its correct position in the left subarray by a series of swaps, and then incremented again to reach its original position plus one.

ii. Using your observations from the last subquestion, formally prove that this algorithm is correct and sorts the given array in time $O(n^2)$. Is this upper bound tight? If yes, for which instances is it realized?

(e) Consider the following procedure:

```
**function** ComputeRanks($A$)
    $r \leftarrow \mathtt{new\ int}[A.\mathtt{length}]$
    **for** $i \in \{0, \dots, A.\mathtt{length} - 1\}$ **do**
        $r[i] \leftarrow 0$
        **for** $j \in \{0, \dots, A.\mathtt{length} - 1\} \setminus \{i\}$ **do**
            **if** $A[j] < A[i]$ **or** $(A[j] = A[i]$ **and** $j < i)$ **then**
                $r[i] \leftarrow r[i] + 1$
    **return** $r$
```

i. What is the running time of ComputeRanks?

ii. Design a sorting algorithm based on ComputeRanks and prove that it is correct. What is its complexity?

iii. Would your algorithm still work correctly if we replace the condition in the `if` block by $A[j] < A[i]$? If it is the case, explain why; if it is not, give an instance in which your algorithm does not behave as expected.

3. **Another tree-based data structure: red-black trees**

A red-black tree is a binary search tree with every node colored black or red such that:

- The root is black,
- The children of a red node are black nodes,
- Every path from the root to the leaves contains the same number of black nodes.

(a) Prove that the height of a red black trees with n nodes is at most $2\log_2(n+1)$.

(b) As in AVL trees, rotations can be used to restore the red-black properties above after a node has been inserted or deleted. Here, we will only consider the case of insertions. In red-black trees, new nodes (except the first one) are always inserted red. A necessity to restore the property only arises when the parent of the inserted node is itself red, since red nodes cannot have red children. Describe how rotations can be used to restore the red-black property when a red node z is inserted as the child of another red node x.

Hint: Distinguish four cases according to whether the uncle y of z (i.e. the brother of x) is red or black, and z is inserted to the left or to the right of x.

3.4 Exam questions

- FS 2020: **T1.c), T1.e), T2.e), T3**;
- HS 2019: T1.c);
- FS 2019: T1.g)-j), T3.b)-c);
- HS 2018: T1.a), **T1.c)-f)**, P1;
- HS 2017: T1.d)-e), **T1.g)-h)**;
- HS 2016: T1.a), T1.g)-i).

Chapter 4

Graph algorithms

In this chapter, technical terms are given in both English and Deutsch. Always make sure that your vocabulary is consistent.

4.1 Definitions

We usually note $G = (V, E)$ with

- $V = \{v_1, ..., v_n\}$, $|V| = n$ the set of nodes (Knotenmenge);
- $E = \{e_1, ..., e_m\}$, $|E| = m$ the set of edges (Kantenmenge).

4.1.1 Undirected graph (ungerichteter Graph)

Here $E \subseteq \{\{u, v\} \mid u, v \in V\}$ contains undirected edges which are denoted as $e_k = \{v_i, v_j\}$. Note that here v_i and v_j are in one **set**, which in particular implies $\{v_i, v_j\} = \{v_j, v_i\}$. Vertices v_i and v_j are called adjacent (benachbart or adjazent) iff $\{v_i, v_j\} \in E$ and a vertex v_i and an edge e_k are incident (inzident) iff $v_i \in e_k$.

4.1.2 Directed graph (gerichteter Graph)

Here $E \subseteq V \times V$ are directed edges which are denoted as ordered pairs $e_k = (v_i, v_j)$. Now, $(v_i, v_j) \neq (v_j, v_i)$. Vertices v_i and v_j are called adjacent (benachbart or adjazent) iff $(v_i, v_j) \in E$ and a vertex v_i and an edge e_k are incident (inzident) iff $v_i \in e_k$.

4.1.3 Bipartite graph (bipartiter Graph)

A graph is bipartite iff we can decompose its set of vertices into $V = U \sqcup W$ two disjoint sets of nodes ($U \cap W = \emptyset$) such that

- Either $E \subseteq \{\{u, w\} \mid u \in U, w \in W\}$ (undirected bipartite graph); or
- $E \subseteq (U \times W) \cup (W \times U)$ (directed bipartite graph).

4.1.4 Tree (Baum) and forest (Wald)

A tree is a graph that is both connected (zusammenhÃ¤ngend, i.e. there exists a path between all pairs of points) and acyclic (azyklisch, i.e. contains no cycle, see below). A connected graph is a tree iff it has exactly $m = n - 1$ edges.

A forest is a graph that is acyclic, but not necessarily connected. Hence, a forest can have several connected components (Zusammenhangskomponenten), each of which is a tree. In short, a forest is a collection of trees.

4.1.5 Adjacency lists (Adjazenzlisten)

The adjacency lists L associate to each $v_i \in V$ the list $L[i]$ of its neighbors (Nachbarn) in G, i.e.

$$L[i] = \{v_j \in V \mid v_i \text{ and } v_j \text{ are adjacent}\}.$$

4.1.6 Adjacency matrix (Adjazenzmatrix)

Let $n = |V|$. The adjacency matrix A is defined as:

$$A \in \{0, 1\}^{n \times n}$$

where

$$a_{i,j} = \begin{cases} 1 & \text{if } v_i \text{ and } v_j \text{ are adjacent} \\ 0 & \text{otherwise} \end{cases}$$

In particular, this definition implies that, for undirected graphs, A is symmetric. The particular case $a_{i,i} = 1$ means that there is a loop on node v_i. This is possible on both directed and undirected graphs, and depends on the definition of graph given and the allowed cases.

4.1.7 What can or cannot we have in a graph?

In general, 'standard' graphs (=what will be called graphs in the exam) **do not**

- Contain several parallel edges between the same pair of nodes—allowing this to happen would make our graph a **multigraph** (Multigraph);
- Contain hyperedges, i.e. edges with more than two endpoints—allowing this to happen would make our graph a **hypergraph** (Hypergraph).

'Standard' graphs may or may not contain (self-)loops (Schleifen). The A&D Skript considers that 'standard' graphs may contain loops.

4.1.8 Sequences

Consider the sequence of vertices

$$\langle v_1, v_2, ..., v_k \rangle \in V^k.$$

This sequence is

- A **walk** (Weg) iff there are edges between v_i and v_{i+1} for all $i \in \{1, \dots, k-1\}$; in this case, the length of the walk is $k - 1$;
- A **path** (Pfad) iff it is a walk whose vertices are all distinct;
- A **tour** (Reise) iff it is a walk whose edges are all distinct;
- A **circuit** (Zyklus) iff it is a walk with $v_1 = v_k$ (starts and ends at the same vertex);
- A **cycle** (Kreis) iff it is a circuit for which no vertex, except the first/last one, is visited more than once;
- A **loop** (Schleife) iff it is a cycle $\langle v_i, v_i \rangle$ of length 1.

When the considered objects cover all vertices or all objects, this gives rise to the following definitions:

- A **Eulerian walk** (Eulerweg) is a walk that visits all edges exactly once, i.e. a walk of length $m = |E|$.
- A **Eulerian circuit** (Eulerkreis[1]) is a circuit that visits all edges exactly once, i.e. a walk of length $m = |E|$.
- A **Hamiltonian path** (Hamiltonpfad) is a path that visits all vertices exactly one, i.e. a path of length $n - 1$.
- A **Hamiltonian circuit** (Hamiltonkreis[2]) is a cycle that visits all vertices, i.e. a cycle of length n.

[1]Note that this terminology is not consistent with the above definition of Kreis; the German translation of the following definition is "Ein Eulerkreis ist ein **Zyklus**, der alle Kanten des Graphen genau einmal durchlÃ¤uft, d. h. ein Zyklus der LÃ¤nge m".

[2]No consistency problem here; the German definition is "Ein Hamiltonkreis ist ein Kreis, der alle Knoten durchlÃ¤uft, d. h. ein Kreis der LÃ¤nge n".

4.1.9 Degree

The degree (or valency) of a vertex v of a graph is the number of edges incident to this vertex, with loops counted twice. It is denoted by $\deg(v)$.

In directed graphs, we distinguish between the *in-degree* $\deg^-(v)$ and the *out-degree* $\deg^+(v)$.

4.1.10 Neighborhood

The neighborhood of some vertex v (set of vertices adjacent to v) is denoted by $N(v)$. We have $\deg(v) = |N(v)|$.

In directed graphs, we again distinguish between $N^-(v)$ and $N^+(v)$. We have $\deg^-(v) = |N^-(v)|$ and $\deg^+(v) = |N^+(v)|$.

4.1.11 Degree-sum formula (handshaking lemma)

$$\sum_{v \in V} \deg(v) = 2|E| = 2m$$

4.2 BFS & DFS

Breadth- and depth-first search are two very similar algorithms most commonly used to explore graphs. More than single algorithms, they are algorithmic templates in which the action to be performed on each node (independantly from the exploration itself) must be adapted to every context.

The algorithms below are presented for the case of connected graphs; for graphs with several connected components, several runs, one by connected component, are necessary.

4.2.1 Stacks and Queues

Stacks and queue are two elementary ADS that can be implemented with (doubly-)linked lists.

Stack

LIFO (last in, first out) data structure: inserted elements are retrieved by inverse order of insertion.

- PUSH (x) : push object x on top of the stack
- POP : removes **last** object added to stack. Throws error if stack empty.

Queue

FIFO (first in, first out) data structure: inserted elements are retrieved by order of insertion.

- ENQUEUE (x): add object x at the end of the queue
- DEQUEUE : removes **first** object added to the queue. Throws error if queue empty.

Additional methods and complexity

Methods ENQUEUE and DEQUEUE might be also called PUSH and POP. In addition, stacks and queues might share several methods which allow for testing emptyness (ISEMPTY), returning length (LENGTH), returning the head element (top or end) without deleting it (HEAD) or deleting all elements (CLEAR).

An essential observation is that all operations above (except CLEAR) have constant-time runtime complexity.

4.2.2 Breadth-first search (BFS)

In BFS, nodes are explored from a root r by order of increasing distance to the root.

Algorithm 14 Breadth-first search

```
$Q \leftarrow$ new queue ()
$Q$.PUSH $(r)$
$D \leftarrow \{r\}$                          ▷ Stores nodes that are done (in $Q$ or already processed)
while $\neg Q$.ISEMPTY () do
    $v \leftarrow Q$.POP ()
    /*do something with $v$*/
    for $w$ s.t. $v$ and $w$ are adjacent in $G$ do
        if $w \notin D$ then
            $Q$.PUSH $(w)$
            $D \leftarrow D \cup \{w\}$
```

The complexity of BFS is $O(n + m)$: each node is considered exactly once and each edge exactly twice.

The time at which an element v is first added to the queue and the time at which all elements in the subtree of root v, including v, have been processed, are called pre- and post-time of v respectively and denoted by pre (v) and post (v). The same terminology can also be used for BFS.

4.2.3 Depth-first search (DFS)

In DFS, nodes are explored top-down from a root r, always exploring all descendants of a node before exploring its non-explored peers. There are two standard implementations of DFS.

The first one is imperative; it is similar to the implementation of BFS above, where we replace the queue by a stack.

Algorithm 15 Depth-first search, imperative style

```
$S \leftarrow$ new stack ()
$S$.PUSH $(r)$
$D \leftarrow \{r\}$
while $\neg S$.ISEMPTY () do
    $v \leftarrow S$.POP ()
    /*do something with $v$*/
    for $w$ s.t. $v$ and $w$ are adjacent in $G$ do
        if $w \notin D$ then
            $S$.PUSH $(w)$
            $D \leftarrow D \cup \{w\}$
```

The second one is recursive, more compact and in some sense more natural:

Algorithm 16 Depth-first search, recursive style

```
function DFS($v, G, D = \emptyset$)
    /*do something with $v$*/
    for $w$ s.t. $v$ and $w$ are adjacent in $G$ do
        if $w \notin D$ then
            $D \leftarrow D \cup \{w\}$
            DFS $(w, G, D)$
DFS $(s, G)$
```

The complexity of DFS is also $O(n + m)$.

4.2.4 Topological sort

A topological sorting of a graph G is an ordering $\prec$ of V such that for all u, v s.t. u and v are adjacent, $u \prec v$. If a topological sorting exists, it can be produced by running DFS on the graph and adding each node to the ordering after handling its children. The multiple runs of DFS in the algorithm below handle possible non-connectedness.

Algorithm 17 Topological sort

```
function DFS($s, G, T, D$)
    for $w$ s.t. $v$ and $w$ are adjacent in $G$ do
        if $w \notin D$ then
            $D \leftarrow D \cup \{w\}$
            DFS $(w, G, T, D)$
    $T \leftarrow T \cup \{s\}$
$T \leftarrow []$
$D \leftarrow \emptyset$
for $v \in V$ do
    if $v \notin D$ then
        DFS $(v, G, T, D)$
```

The complexity of topological sort is $O(n + m)$.

4.3 Shortest path algorithms

Shortest path algorithms in graphs solve the problem of finding a path between two vertices, minimizing the sum of edge costs along the path. The cost (or weight) for a given edge $e = (v_i, v_j)$ is given as w_{ij} or $w(i, j)$. Recall that $n = |V|, m = |E|$.

There are essentially three types of shortest path algorithms: one-to-one (computes distance between a pair of nodes), one-to-all (computes distance between one node and all other nodes), all-to-all (computes distances between all pairs of nodes).

For any nodes u and v in a graph, whenever v is accessible from u and there exists a negative weight cycle which is also accessible for u, parts of arbitrarily low cost can be found between u and v. Therefore, most shortest path algorithm require that there are no negative cycles in the graph, which can be tested using e.g. Bellman-Ford's algorithm.

4.3.1 BFS

Type One-to-one or one-to-all

Complexity $\Theta(n + m)$

Restriction on input All edges weights are equal

Idea One-to-one: run DFS from source and stop as soon as you reach the destination; one-to-all: run DFS from source and choose first path you find to any other node.

4.3.2 Floyd-Warshall's algorithm

Type All-to-all

Complexity $\Theta\left(n^3\right)$

Restriction on input No negative cycles

Idea A DP algorithm, finds shortest path between all pairs in an incremental fashion, using previously stored results. More precisely, for all $1 \leq i, j, k \leq n$, it computes $sp(i, j, k)$, which is the shortest path from v_i to v_j using only vertices from the set $\{v_1, ..., v_k\}$. To compute the next intermediate result $sp(i, j, k+1)$, we can use the previous results: the new shortest path can still be computed either using only vertices from that set or combining a path from v_i to v_{k+1} with a path from v_{k+1} to v_j:

$$sp(i, j, 0) = w(i, j)$$
$$sp(i, j, k+1) = \min\{sp(i, j, k), sp(i, k+1, k) + sp(k+1, j, k)\}$$

4.3.3 Dijkstra's algorithm

Type One-to-all

Complexity $O(m + n \log n)$

Restriction on input No negative edge weights

Idea For each node, stores the best path from source and previous node along this path. Iterates along unvisited nodes, always picking the previously unvisited node with least distance to the root. If neighbor v of current visited node u has higher cost from source than the sum of cost of u and path $w(u, v)$, sets its new cost to this sum and previous node to u. Mark u as visited and repeat.

The actual cost of the algorithm is $O(m \cdot T_{dk} + n \cdot T_{em})$, with T_{dk} the time to decrease a key in the ADS and E_{em} the time to extract the minimum of the DS. If we use adjacency lists and a binary heap, we reach our time above.

Dijsktra's algorithm is one of the most important—if not the most important—algorithm in CS. You should know its structure, its complexity, and be able to implement in pseudo-code and in Java when an implementation of the required ADS is provided.

4.3.4 Bellman-Ford's algorithm

Type One-to-all

Complexity $O(n \cdot m)$

Restriction on input No negative cycles

Idea Very similar to Dijkstra, but instead of repeatedly picking the node with minimal cost, processes them all $n - 1$ times. This allows the shortest path to propagate in the graph correctly. Worse time than Dijkstra's but negative edges are accepted.

4.3.5 Johnson's algorithm

Type One-to-all

Complexity $O(n^2 \log n + nm)$

Restriction on input No negative cycles

Idea Starts with a transformation applied on the original graph that removes all negative edges. For this, adds a new node q connected to all existing ones with weight 0. Finds the cost of the shortest path from q to v, $h(v)$, using Bellman-Ford. The costs of the edges of the original graph are updated to $w_n(u, v) := w(u, v) + h(u) - h(v)$. At the end, node q is removed and Dijkstra's algorithm used on the new graph without negative weights w_n. Then uses Dijkstra's algorithm on this new graph.

4.4 Minimal spanning trees

A minimal spanning tree (MST, minimaler Spannbaum) of a graph $G = (V, E)$ is a tree T with vertices $V(T) = V$ and edges $E(T) \subset E$ such that $\sum_{e \in E} w(e)$ is minimal among all such trees.

Note When analyzing the complexity of algorithms that involve the number of vertices n as well as the number of edges m, be aware that as $m \leq n^2$, we also have $\log m \leq \log\left(n^2\right) = 2\log(n) = O(\log n)$.

4.4.1 Borůvka's algorithm

Complexity $O(m \log n)$

Idea Constructs a spanning forest iteratively until it becomes a spanning tree. Starts with each vertex in a distinct component (a trivial tree with one vertex and zero edge). All vertices select their nearest neighbor simultaneously, and all corresponding edges are added. This results in a number of trees being formed. Then, all of these trees select again their smallest outgoing edge, which are added to the forest, dividing the number of trees by two at each iteration. The process goes on for at most $\Theta(\log n)$ rounds until only one connected component remains.

4.4.2 Prim's algorithm

Complexity $O(m \log n)$ (binary heap and adjacency lists), can be improved to $O(m + n \log n)$ with Fibonacci heaps

Idea A variant of Dijkstra's algorithm, where for each vertex v not already processed, we keep the cost of the shortest edge connecting v to the tree under construction. Just as in Dijkstra, extends the current tree with the edge $e = (w, v)$ and vertex v of minimal cost, updating the costs of neighbors of v. Repeats until all vertices are covered.

4.4.3 Kruskal's algorithm

Complexity $O(m \log n)$

Idea Sorts edges by increasing order of weight and tries to add them in this order, abstaining from it whenever adding the new edge would result in a cycle. The final set of edges is a spanning tree.

4.4.4 Union-find

Both Borůvka's and Kruskal's algorithms require an ADS that allow us to retrieve the connected component of a vertex and unify two connected components efficiently. One such ADS is union-find, which typically provides the following interface:

- CREATE (n) initializes a union-find structure with n objects $\{0, \ldots, n-1\}$ that all have their own connected component (n of them in total), each indexed by some $i \in \{0, \ldots, n-1\}$;
- FIND (i) returns the index of the connected component of object i;
- UNION (i, j) finds the indices of the connected components of objects i and j and merges these two connected components into one.

Reasonably simple implementations have an amortized $O(\log n)$ running time for both UNION and FIND; more elaborated ones yield an amortized $O(\alpha(n))$ time, where α is the extremely slowly growing (quasi-constant) inverse Ackermann function.

4.5 Exercises

1. **Theory questions**

   (a) True or false?

   i. In an undirected graph, a tight bound for the number of edges is n^2.
   ii. In an undirected graph, a tight asymptotic bound for the number of edges is $\Theta\left(n^2\right)$.
   iii. If the maximum degree of any node in an undirected graph G is 1, then this graph is a tree.
   iv. The complexity of computing the out-degree of a vertex v in an adjacency matrix is $\Theta\left(\deg v\right)$.
   v. The complexity of computing the sum of all out-degrees of vertices in an adjacency matrix is $\Theta\left(n^2\right)$.
   vi. The list of vertices accessible from a vertex v in a graph G can be computed by using at most $n-1$ multiplications of $n \times n$ matrices.
   vii. A connected graph is Eulerian (i.e. contains a Eulerian circuit) iff all its vertices have even degree.
   viii. A graph contains a Eulerian walk iff all its vertices have even degree.
   ix. Testing if a graph is Eulerian is NP-complete.
   x. The post-order of BFS always gives a valid topological ordering.

   (b) Is the pseudo-code of the second DFS implementation still correct if we swap the two lines in the `if` block? If yes, explain way. Otherwise, provide an input that leads to an incorrect output if the modified algorithm is run on it.

   (c) Give an example of a graph that has no topological ordering. What is the result of running on this graph the topological sorting algorithm given above?

   (d) Give examples of graphs on $n = 2^k - 1$ vertices that have exactly $1, n, 2^n$ and $n!$ topological orderings respectively.

2. **Shortest paths**

   (a) Implement Dijkstra's algorithm in pseudo-code. You can assume that you are provided with an already implemented class `Table` that has the following interface:

```
class Table {
  Table(int n); //creates empty table with n uninitialized fields
  int get(int i) throws NotInitialized; //returns the value of
                                        //field i if initialized
  int set(int i, int v); //sets the value of field i to v,
                         //initializing it if necessary
  int smallest(); //returns i such that table.get(i) is minimal
}
```

   (b) Design an algorithm to compute the costs of one-to-all shortest paths in a graph where all edge costs are 0 or 1. The complexity of your algorithm should be $\mathcal{O}\left(\alpha(n)m + n\right)$, where α is quasi-constant. You might want to check out the section on union-find!

   (c) Explain how we can retrieve the shortest paths between all pairs of vertices after running Floyd-Warshall's algorithm. What is the runtime of doing so (in tight asymptotic notation)?

   (d) In medieval Switzerland, there were n cities connected by m one way roads. Each city i charged merchants that traversed it a toll t_i. You are a merchant and want to travel from your boss's farm in Visp VS to your hometown of Mumpf (city 1). Design an algorithm to find the path that minimizes your travel cost. You can assume that you will not be asked to pay a toll when you will leave Visp (only merchants entering the city are charged).

   (e) Consider the same setting as in the previous question. Now you are in Mumpf and your boss in Visp. You want to choose some city in Switzerland where you could meet to discuss about your next raise in salary. You both have to travel and want (i) first, to minimize the sum of your costs (ii) then, among all pairs of paths of same cost, to share the costs as fairly as possible. Solve the problem algorithmically.

3. **Minimal spanning trees**

   (a) Prove that there exists only one path between any pair of nodes in a tree.

   (b) Prove or disprove: For all vertices u, v of a graph G, the only path between u and v in an MST T of G is a shortest path between u and v in G.

   (c) Give an example of a graph on which Prim's and Kruskal's algorithms return different correct MSTs.

4.6 Exam questions

- FS 2020: T1.b), T1.d), T2.d), **T4**, P1;
- HS 2019: T1.b), T1.d)-f), T2.c), T3, T4.a)-b), P1;
- FS 2019: **T1.d), T2**, T3.a), **P2**;
- HS 2018: T1.b), T1.g), **T3**;
- HS 2017: T1.b)-c), T1.f), **T2, T3**;
- HS 2016: T1.d)-f), T2.

Chapter 5

List of common linguistic mistakes

Important: Do not mix languages in your exam submissions! Choose the language you are most comfortable with!

5.1 Deutsch

Das Wort *Graph* heißt im Genitiv/Dativ/Akkusativ Singular *Graphen*, also: *betrachten wir einen Graphen*, nicht **betrachten wir einen Graph*.

5.2 English

The singular of *vertices* is *vertex*, not **vertice* (the Italian word for 'summit').

German *Kreis* = English *cycle*; German *Zyklus* = English *circuit*.

Forms of the relative pronoun *who*: *who* (Nom.), *whom* (Dat./Akk.), *whose* (Gen.). The relative pronoun *who* is only used for people, not for objects, for which you should use *that* or *which*.

Distinguish between the noun *half* (pl. *halves*) (Hälfte, -n) and the verb *to halve* (halbieren). Distinguish between the noun *proof* and the verb *to prove*.

Do not put commas before relative pronouns! Write *every number that is* and not *every number, that is*.

Another word with an irregular plural is *leaf* (pl. *leaves*).

Avoid colloquial contractions (*gonna* etc.).

Do not translate German *also* by English *also*. The German word *also* means *therefore*, *hence*.

Use *once*, *twice* and not *one time*, *two times*.

Chapter 6

Solutions of exercises

Chapter 1

1. **Asymptotics**

   (a) We propose following solution:

   $$2^{16} \quad \log n^4 \quad \log^8 n \quad \sqrt{n} \quad \binom{n}{3} \quad n^5+n \quad \frac{2^n}{n^2} \quad n! \quad n^n$$

   In simplest asymptotic notation

   $$\mathcal{O}(1) \quad \mathcal{O}(\log n) \quad \mathcal{O}\left(\log^8 n\right) \quad \mathcal{O}(\sqrt{n}) \quad \mathcal{O}\left(n^3\right) \quad \mathcal{O}\left(n^5\right) \quad \mathcal{O}\left(\frac{2^n}{n^2}\right) \quad \mathcal{O}(n!) \quad \mathcal{O}(n^n)$$

   (b) True or false?

   i. True
   ii. True
   iii. False
   iv. True, note that $\lim_{n\to\infty} \dfrac{e^{\sqrt{\ln n}}}{\sqrt{e^{\ln n}}} = 0$
   v. True
   vi. True
   vii. True
   viii. False, False, True
   ix. True, actually for all $b \in \mathbb{N}$
   x. False
   xi. True, $f(x) = x^{1.5}$ for example

   (c) Give the simplest tight bound for the following formulae:

   i. $P(n) = 15n^{41} + 14n^{42} + \log n \in \mathcal{O}\left(n^{42}\right)$;
   ii. $Q(n) = n^{41} + n^{42} + e^{5n} \in \mathcal{O}\left(e^{5n}\right)$;
   iii. $R(n) = \frac{n^2+13}{n^3+n+8} + n^{-2} \in \mathcal{O}\left(n^{-1}\right)$;
   iv. $S(n) = \sqrt{e^{\ln n}} \in \mathcal{O}(\sqrt{n})$.

2. **Proofs by induction**

   (a) Proof by induction over n of $P(n) \equiv \sum_{i=0}^{n} a^i = \frac{a^{n+1}-1}{a-1}$.

   - **Base Case** $n = 0$

   $$\sum_{i=0}^{0} a^i = a^0 = 1 = \frac{a-1}{a-1}$$

- **Induction Hypothesis**
  We assume that for some $n \in \mathbb{N}$, $P(n)$ holds true.
- **Induction step** $n \to n+1$

$$\sum_{i=0}^{n+1} a^i = \sum_{i=0}^{n} a^i + a^{n+1} \overset{\text{I.H.}}{=} \frac{a^{n+1}-1}{a-1} + a^{n+1} = \frac{a^{n+1}-1+(a-1)a^{n+1}}{a-1} = \frac{a^{n+2}-1}{a-1}$$

  Therefore by mathematical induction $P(n)$ holds for all $n \in \mathbb{N}$

(b) Proof of $P(n) \equiv \sum_{i=1}^{n} i(i+1) = \frac{n(n+1)(n+2)}{3}$ by induction over n.

- **Base Case** $n = 1$

$$\sum_{i=1}^{1} k(k+1) = 1(1+1) = 1(1+1)\frac{3}{3} = \frac{1(1+1)(1+2)}{3}.$$

- **Induction Hypothesis**
  We assume that for some $n \in \mathbb{N}$, $P(n)$ holds true.
- **Induction step** $n \to n+1$

$$\begin{aligned}\sum_{i=1}^{n+1} i(i+1) &= (n+1)(n+2) + \sum_{i=1}^{n} i(i+1)\\ &\overset{\text{I.H.}}{=} (n+1)(n+2) + \frac{n(n+1)(n+2)}{3}\\ &= \frac{(n+1)(n+2)(n+3)}{3}\\ &= \frac{(n+1)((n+1)+1)((n+1)+2)}{3}.\end{aligned}$$

(c) Proof of $P(n) \equiv \forall x > -1,\ (1+x)^n \geq 1 + nx$ by induction over n.

- **Base Case** $n = 0$

$$\forall x > -1,\ (1+x)^0 = 1 \geq 1 + 0 \cdot x.$$

- **Induction Hypothesis**
  Let us assume that for some $n \in \mathbb{N}$ it $P(n)$ holds.
- **Inductive Step:**
  Let $x > -1$.

$$\begin{aligned}(1+x)^{n+1} &= (1+x)^n(1+x)\\ &\overset{IH}{\geq} (1+nx)(1+x)\\ &= 1 + (n+1)x + nx^2\\ &\geq 1 + (n+1)x.\end{aligned}$$

(d) Proof of $P(n) \equiv \sum_{i=1}^{n} \frac{1}{i(i+1)} = \frac{n}{n+1}$ by induction over n.

- **Base Case**

$$\sum_{i=1}^{1} \frac{1}{i(i+1)} = \frac{1}{1+1}.$$

- **Induction Hypothesis**
  Let us assume that for some $n \in \mathbb{N}$, $P(n)$ holds

- **Induction Step**

$$\begin{aligned}\sum_{i=1}^{n+1}\frac{1}{i(i+1)} &= \frac{1}{(n+1)(n+2)}+\sum_{i=1}^{n}\frac{1}{i(i+1)}\\ &\overset{IH}{=} \frac{1}{(n+1)(n+2)}+\frac{n}{n+1}\\ &= \frac{1+n(n+2)}{(n+1)(n+2)}\\ &= \frac{n^2+2n+1}{(n+1)(n+2)}\\ &= \frac{(n+1)^2}{(n+1)(n+2)}\\ &= \frac{n+1}{n+2}\\ &= \frac{n+1}{(n+1)+1}.\end{aligned}$$

(e) Proof by induction over n of $P(n) \equiv \sum_{i=1}^{n}(2i-1) = n^2$.

- **Base Step:** $n = 1$

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

- **Induction Hypothesis**
Let us assume for some $n \in \mathbb{N}$ that $P(n)$ holds.
- **Inductive Step:**

$$\begin{aligned}\sum_{i=1}^{n+1}(2i-1) &= (2(n+1)-1)+\sum_{i=1}^{n}(2i-1)\\ &= (2n+1)+\sum_{i=1}^{n}(2i-1)\\ &\overset{\text{IH}}{=} (2n+1)+n^2\\ &= n^2+2n+1\\ &= (n+1)^2\end{aligned}$$

(f) Proof of $P(n) \equiv 3 \mid (n^3+2n)$ by induction over n.

- **Base Case** $n = 1$ we prove $P(1)$

$$1^3 + 2(1) = 3.$$

3 is divisible by 3.
- **Induction Hypothesis**
Let us assume that for some $n \in \mathbb{N}$ that $P(n)$ holds.
- **Inductive Step**

$$\begin{aligned}(n+1)^3+2(n+1) &= n^3+3n^2+5n+3\\ &= (n^3+2n)+(3n^2+3n+3)\\ &\overset{\text{IH}}{=} 3u+3(n^2+n+1)\\ &= 3(u+n^2+n+1).\end{aligned}$$

3. **Recursion and induction**

(a) Let $k \in \mathbb{Z}$. We define the following integer sequence u:

$$\begin{aligned} u_0 &= 1 \\ u_1 &= k \\ u_n &= 2u_{n-1} - u_{n-2} && n \geq 2. \end{aligned}$$

We'll prove that for $n \geq 1$

$$u_n = nk - (n-1).$$

by total induction over n.

- **Base Case** $n = 1$

$$u_1 = k = k \cdot 1 - 0 = k$$

- **Induction Hypothesis**
Assume that for some $n \in \mathbb{N}$ it holds that for any $n' < n$

$$u_{n'} = n'k - (n-1)$$

- **Induction Hypothesis** $n \to n+1$

$$\begin{aligned} u_{n+1} &\overset{\text{def}}{=} 2u_n - u_{n-1} \\ &\overset{\text{IH}}{=} 2(nk - (n-1)) - ((n-1)k - (n-2)) \\ &= 2nk - 2n - 2 - nk + k + n - 2 \end{aligned}$$

$$= nk + k - n = (n+1)k - n$$

(b) Given the following conditional recursive function:

$$T(n) = \begin{cases} 5 \cdot T\left(\frac{n}{7}\right) + 8, & \text{if } x > 1 \\ 3, & x = 1 \end{cases}$$

We use telescoping to arrive to an hypothesis.
Let $n > 1$ we also assume n is a power of 7, we write $n = 7^k$.

$$\begin{aligned} T(n) &= 5 \cdot T\left(\frac{n}{7}\right) + 8 \\ &= 5 \cdot \left(5 \cdot T\left(\frac{n}{7^2}\right) + 8\right) + 8 \\ &= 5 \cdot \left(5 \cdot \left(5 \cdot T\left(\frac{n}{7^3}\right) + 8\right) + 8\right) + 8 = \dots \\ &= 5^k \cdot 3 + 8 \cdot \sum_{i=0}^{k-1} 5^i \\ &= 5^k \cdot 3 + 8 \cdot \frac{5^k - 1}{4} \\ &= 5^k \cdot 3 + 2 \cdot 5^k - 2 \\ &= 5^{k+1} - 2. \end{aligned}$$

We now have our hypothesis, $\forall k \in \mathbb{N},\ P(k) \equiv T(7^k) = 5^{k+1} - 2$
We prove $\forall k \in \mathbb{N},\ P(k)$ by complete induction over k
Base Step: we prove $P(0)$:

$$T(7^0) = T(1) = 3 = 5^1 - 2.$$

Inductive Step

The Inductive Hypothesis (IH) allows us to use $P(n)$ to prove $P(n+1)$

$$\begin{aligned} T(7^{k+1}) &= 5 \cdot T(7^k) + 8 \\ &\overset{\text{IH}}{=} 5 \cdot (5^{k+1} - 2) + 8 \\ &= 5 \cdot 5^{k+1} - 10 + 8 \\ &= 5^{k+2} - 2. \end{aligned}$$

If we replace 7 by 2 in the definition of 2, we can use the Master theorem with $a = 5$ and $b = 0$ (as 8 is constant) to show $T(n) \in \mathcal{O}\left(n^{\log_2 5}\right) \simeq \mathcal{O}\left(n^{2.32}\right)$.

Chapter 2

1. **Theory questions**

   (a) Longest increasing subsequence, longest common subsequence, edit distance, matrix chain multiplication, subset sum, knapsack, but also many standard algorithms: mergesort, quicksort, Floyd-Warshall...

   (b) The above algorithm for computing the Fibonacci sequence has complexity $\mathcal{O}(n)$. The bottom-up variant first initializes an array of size $n+1$ ($\mathcal{O}(n)$) and then performs $n-1$ iterations of a loop with constant-time body (one addition, two array reads, one array write) before returning the last element of the array, also in constant time. The top-down variant computes every field of m only once and uses it at most twice, which results in the same time complexity. As n is the *value* and not the *size* of the input (the size is $s = \log n$, which is the number of bits needed to encode n), this algorithm is pseudopolynomial, but not polynomial.

2. **DP and greedy problems**

   (a) DP. Define the following table T of size $(W+1) \times (n+1)$:

   $$\forall 0 \le w \le W, 0 \le j \le n,\ T[w,j] = \max_{S \subseteq \{1,\dots,j\}, \sum_{i \in S} w_i \le w} \sum_{i \in S} v_i.$$

   The result can be read in entry $T[W,n]$. For w, j with $w = 0$ or $j = 0$, since all w_i are positive, we have $T[w,j] = 0$.
   For $w > 0, j > 0$, we compute

   $$T[w,j] = \begin{cases} \max\left(T[w,j-1], T[w-w_j,j] + v_j\right) & \text{if } v_j > 0, w_j \le w \\ T[w,j-1] & \text{otherwise} \end{cases}$$

   for increasing w and j.
   We have $\mathcal{O}(nW)$ entries in our table, each of which can be computed in constant time. The overall complexity is $\mathcal{O}(nW)$.

   (b) Greedy. Consider the following algorithm: This algorithm first sorts the liquids by order of decreasing "worth density" (in CHF per kg). It then tries to add as much of the first liquid as it can (until the knapsack is full and/or the supply is exhausted); if the volume left in the knapsack (represented by r) is positive, it proceeds with adding from the second liquid, then from the third, fourth etc. The sorting costs $\mathcal{O}(n \log n)$ (including the n divisions), while the main loop costs $\mathcal{O}(n)$ (n constant-time iterations). Let us prove that this algorithm is correct.

Algorithm 18 Fractional knapsack

```
Sort $(d_i, v_i)$ such that $\frac{v_1}{d_1} \geq \frac{v_2}{d_2} \geq \cdots \geq \frac{v_n}{d_n}$
$q \leftarrow \texttt{int}[n]$
$r \leftarrow W$
for $i \in \{1, \ldots, n\}$ do
    $q[i] \leftarrow \min\left(s_i, \frac{r}{d_i}\right)$
    $r \leftarrow r - q[i]d_i$
```

Assume w.l.o.g. that $\frac{v_1}{d_1} > \frac{v_2}{d_2} > \cdots \geq \frac{v_n}{d_n}$ and that $v_i > 0$ for all i (nonpositive values can be safely ignored). We can safely suppose that all $\frac{v_i}{d_i}$ are different, since weight can be shared arbitrarily between liquids of same worth density without affecting the result.

First, we observe that the choice of q returned by the algorithm is always of the form

$$q^* = \left[s_1, \ldots, s_k, \frac{W - \sum_{j=1}^k s_j d_j}{d_j}, 0, \ldots, 0\right].$$

for some k. In other words, the algorithm always takes the whole supply of the most valuable liquids until it eventually reaches the end of the list or a liquid $k+1$ which it can not exhaust without reaching the capacity limit of the knapsack. Only a fraction of this liquid (corresponding to the space left in the knapsack) is chosen, and the following liquids are not considered.

If the full supply of all liquids can be taken without exceeding the capacity of the knapsack, then it is clear that this choice is optimal, since we have assumed that all v_i are positive. Otherwise, let q^*, k as above, and consider an optimal choice q' of q. We now show that $q[i] \geq q^*[i]$ for all $i \in \{1, \ldots, k+1\}$, which proves $q' = q^*$ since q^* already fills up the knapsack. By contradiction, let i_0 be the smallest $i \in \{1, \ldots, k+1\}$ such that $q'[i_0] < q^*[i_0]$. By definition of i_0, we have $q'[i] \geq q^*[i]$ and therefore $q'[i] = q^*[i]$ for all $0 \leq i < i_0$. Hence, the total spare volume available to store $q'[i_0+1], \ldots, q'[n]$ is at most $W - \sum_{j=1}^{i_0-1} q^*[j]d_j - q'[i_0]d_{i_0}$. The constraint is

$$\sum_{j=i_0+1}^{n} q'[j]d_j \leq W - \sum_{j=1}^{i_0-1} q^*[j]d_j - q'[i_0]d_{i_0}.$$

Moreover, since our algorithm exhausts the knapsack capacity, we get

$$\sum_{j=i_0+1}^{n} q^*[j]d_j = W - \sum_{j=1}^{i_0} q^*[j]d_j$$

so

$$\sum_{j=i_0+1}^{n} \left(q'[j] - q^*[j]\right) d_j \leq \left(q^*[i_0] - q'[i_0]\right) d_{i_0}. \ (*)$$

We have

$$\begin{aligned}\sum_{j=1}^{n} q'[j]v_j - \sum_{j=1}^{n} q^*[j]v_j &= (q'[i_0]-q^*[i_0])\,v_{i_0} + \sum_{j=i_0+1}^{n}(q'[j]-q^*[j])\,v_j \\ &= (q'[i_0]-q^*[i_0])\,v_{i_0} + \sum_{j=i_0+1}^{n}(q'[j]-q^*[j])\,\frac{v_j}{d_j}d_j \\ &\le (q'[i_0]-q^*[i_0])\,v_{i_0} + \frac{v_{i_0+1}}{d_{i_0+1}}\sum_{j=i_0+1}^{n}(q'[j]-q^*[j])\,d_j \\ &\le (q'[i_0]-q^*[i_0])\left(v_{i_0} - \frac{v_{i_0+1}}{d_{i_0+1}}d_{i_0}\right) \\ &= \underbrace{(q'[i_0]-q^*[i_0])}_{<0}\,\underbrace{v_{i_0}}_{>0}\underbrace{\left(1-\frac{v_{i_0+1}}{d_{i_0+1}}\frac{d_{i_0}}{v_{i_0}}\right)}_{>0} \\ &< 0\end{aligned}$$

(c) DP. Define the following table T of size $(N+1)\times(n+1)$:

$$\forall 0\le x\le N, 0\le j\le n,\ T[x,j] = \min_{c\in\mathbb{N}^j,\sum_{i=1}^{j} c_i s_i = x}\ \sum_{i=1}^{j} c_i$$

with the convention that $\min\emptyset = +\infty$. We find the final result in $T[N,n]$. For all j, we have $T[0,j]=0$ and for all $x>0$, we have $T[x,0]=+\infty$.
For $x>0, j>0$, we compute

$$T[x,j] = \begin{cases} \min\left(T[x,j-1], T[x-s_j,j]+1\right) & \text{if } s_j \le x \\ T[x,j-1] & \text{otherwise}\end{cases}$$

for increasing x and j.
As for knapsack I, the overall complexity is $\mathcal{O}(nN)$.

(d) DP. Define the following tables N and F of size $n+1$ each.

$$\begin{aligned} N[i] &= \text{min. number of operations to turn } b_1,\dots,b_i \text{ on} && \forall 0\le i\le n \\ F[i] &= \text{min. number of operations to turn } b_1,\dots,b_i \text{ off} && \forall 0\le i\le n.\end{aligned}$$

When the tables are filled in, the result can be read in $F[n]$. Clearly, $N[0]=F[0]=0$.
For the recurrence relation, we first observe that the order of operations does not matter. The two types of operations (single or collective switch) are simply translations, either by $[0\dots010\dots0]$ or by $[1\dots10\dots0]$, in the vector space $\mathbb{F}_2^n$, and therefore commutative.
For $0<i\le n$, assume that $N[i-1]$ and $F[i-1]$ are already computed. By the previous remark, to obtain an optimal sequence turning $b_1,\dots,b_i$ on or off, we can first perform all operations that do not involve light i and then all operations that involve light i. Possible operations that impact b_i are of two sorts: either a single switch s_i of b_i or a collective switch c_j of b_1 to b_j for some $j\ge i$. Performing both is never optimal w.r.t. $b_1,\dots,b_i$, since, for all $k\in\{1,\dots,i\}$, $s_i\circ c_j(k) = c_{i-1}(k)$ which spares one operation. Hence, the last operation performed can be chosen to be either s_i, c_i or none (in case b_i is already set to its final value). If the last operation performed in an optimal sequence to turn $b_1,\dots,b_i$ off is s_i, the optimal moves to reach the previous configuration ($b_1,\dots,b_{i-1}$ off) can be found in $N[i-1]$; if the last operation is c_i, the optimal number of moves to turn $b_1,\dots,b_{i-1}$ on is $F[i-1]$. We get:

$$N[i] = \begin{cases}\max(F[i-1],N[i-1])+1 & \text{if } b_i=0 \\ N[i-1] & \text{if } b_i = 1\end{cases}$$

$$F[i] = \begin{cases}\max(F[i-1],N[i-1])+1 & \text{if } b_i=1 \\ F[i-1] & \text{if } b_i = 0\end{cases}.$$

The complexity of each update is $\mathcal{O}(1)$, resulting in an overall time complexity of $\mathcal{O}(n)$.

(e) DP. Define the following table T of size n:

$$\forall 1 \leq j \leq n,\ T[j] = \text{number of sequences of hops to reach } j$$

The result can be read in entry $T[n]$.

Since the bunny starts on step 1, there is only one (empty) sequence of hops leading to step 1, so $T[1] = 1$. For $j < 4$, we easily compute $T[2] = 1, T[3] = 2$. For $j \geq 4$, the last hop may be of height 1, 2 or 3, so that

$$T[j] = T[j-1] + T[j-2] + T[j-3].$$

We have $\mathcal{O}(n)$ entries in our table, each of which can be computed in constant time. The overall complexity is $\mathcal{O}(n)$.

Alternatively, the previous recurrence relation can be used to compute the closed-form formula $T[n] = \frac{n^2-3n}{2} + 2$ and return the result in time $\mathcal{O}(1)$.

(f) DP. Define the following table T of size $(x+1) \times (y+1)$:

$$\forall 0 \leq x' \leq x, 0 \leq y' \leq y,\ T[x', y'] = \text{number of sequences of moves to reach } (x', y')$$

The result can be read in entry $T[x, y]$.

Since the robot starts at $(0,0)$, $T[0,0] = 1$. For $x' > 0, T[x', 0] = 1$ and for $y' > 0, T[0, y'] = 1$. For $x' > 0, y' > 0$, we have

$$T[x', y'] = T[x', y'-1] + T[x'-1, y']$$

as the robot comes either from the left or from the bottom.

We have $\mathcal{O}(xy)$ entries in our table, each of which can be computed in constant time. The overall complexity is $\mathcal{O}(xy)$.

Alternatively, the previous recurrence relation can be used to compute the closed-form formula $T[x, y] = \binom{x+y}{x} = \binom{x+y}{y}$ and return the result in time $\mathcal{O}(1)$.

(g) Greedy. Add words (and spaces between words) to the current line as long as it is still possible. Then insert a line break, print the next word on the next line, and repeat.

Let us prove that this is optimal. More precisely, for any $k \geq 1$ and for any sequence of words that can be wrapped on k lines, let us prove by induction on k that the greedy algorithm returns a placement on k lines. For $k = 1$, the result is obvious. Assume now that the property holds for all k' up to a certain rank $k \geq 1$. Consider an optimal placement P (not necessarily obtained by the greedy strategy) on $k+1$ lines. If P can be obtained by the greedy strategy, we are done. Otherwise, there is a first line $1 \leq i \leq k$ such that there is enough space at the end of line i to add the next word w', which in the optimal placement P has been added to line $i+1$. Now, move as many words as possible from $P_{i+1}, P_{i+2} \ldots$ up into P_i to fill line i as in the greedy algorithm. Let w_r be the first word that cannot be added into i. Looking at P, we see that the subsequence $w_r, \ldots, w_n$ can be wrapped on at most $k - i < k$ lines, so by our induction hypothesis the greedy algorithm is optimal on this subsequence. Therefore, the full greedy algorithm wraps the text on at most $i - 1 + 1 + k - i = k$ lines, which completes the proof.

(h) DP. Define the following table T of size n:

$$\forall 1 \leq i \leq n,\ T[i] = \text{cost of a square-sum optimal wrapping of } (w_1, \ldots, w_i)$$

The result can be read in entry $T[n]$.

Clearly, $T[1] = (L - \ell_1)^2$. For $i > 1$, we have the recurrence relation

$$T[i] = \min_{1 \leq j \leq i, \left(\sum_{p=j}^{i} \ell_p\right) + i - j \leq L} \left[T[j-1] + \left(L - \left(\sum_{p=j}^{i} \ell_p \right) - i + j \right)^2 \right]$$

which tries all possible sequences of words fitting into the last line.
We have $\mathcal{O}(n)$ entries in our table, each of which can be computed in time $\mathcal{O}(n)$. The overall complexity is $\mathcal{O}(n^2)$.
The positions at which to break lines can easily be found by backtracking, remembering which choice of j minimizes the objective function. This only results in an additional linear cost.

(i) DP. First, observe that only categories that appear on the left-hand-side of at least one rule are relevant, since those that never do are always empty. Hence, w.l.o.g., we can assume that all categories appear on the left-hand-side of a rule, and that there are at most R categories.
Define the following table T of size $\mathcal{O}(n^2R)$:

$$\forall 1 \leq i \leq j \leq n, C \text{ category in } G,\ T[i,j,C] = \begin{cases} \texttt{True} & \text{if } (w_i, \dots, w_j) \text{ is matched by } C \\ \texttt{False} & \text{otherwise.} \end{cases}$$

The result can be read in entry $T[1,n,S]$.
We can fill in the table by order of increasing $j - i$ (the order of categories does not matter). Subsequences of length 1 can only be matched by a rule of the form $C \to \alpha$:

$$\forall 1 \leq i \leq n, C \text{ category in } G,\ T[i,i,C] = (\exists C \to w_i \in G).$$

Longer subsequences can only be obtained by concatenation of two shorter subsequences using a rule of the form $C \to AB$, provided that we can split the sequence $w_i, \dots, w_j$ into two subsequences $w_i, \dots, w_{k-1}$ and $w_k, \dots, w_j$ matched by A and B respectively:

$$\forall 1 \leq i < j \leq n, C \text{ category in } G,$$
$$T[i,j,C] = (\exists k \in \{i+1, \dots, j\}, \exists C \to AB \in G.\, T[i,k-1,A] \wedge T[k,j,B]).$$

Each entry of our table checks only those rules that have the corresponding category in their left-hand side. Therefore every category needs to be checked only once for each subsequence (i,j). The overall cost of filling in the table is therefore $\mathcal{O}(n^2R)$.

Programming exercises

- HS 2018 P2
  - $\mathcal{O}(M^2 \cdot N^2)$: Test all possible squares.
  - $\mathcal{O}(M \cdot N)$:
    Define the following table T of size $\mathcal{O}(M \cdot N)$:

$$\forall 1 \leq i \leq M, 1 \leq j \leq N,\ T[i,j] = \text{size of largest square of 0s with south-east corner } (i,j).$$

The result can be extracted as $\max_{1 \leq i \leq M, 1 \leq j \leq N} T[i,j]^2$.
We fill in the table by order of increasing i and j. Squares located along the northern or western border of the grid can be of size at most 1, hence

$$\forall 1 \leq j \leq N,\ T[0,j] = \begin{cases} 1 & \text{if } A[0,j] = 0 \\ 0 & \text{otherwise} \end{cases}$$
$$\forall 1 \leq i \leq M,\ T[i,0] = \begin{cases} 1 & \text{if } A[i,0] = 0 \\ 0 & \text{otherwise} \end{cases}.$$

Let $i > 0, j > 0$. We observe that for all ℓ, there is a square of 0s of size ℓ with its south-east corner located at (i,j) iff there are squares of 0s of size $\ell - 1$ with their south-east corners at $(i-1,j)$, $(i,j-1)$, $(i-1,j-1)$ and we have $A[i,j] = 0$. This yields the following formula:

$$\forall 1 \leq i \leq M, 1 \leq j \leq N,\ T[i,j] = \begin{cases} \min(T[i-1,j], T[i,j-1], T[i-1,j-1]) + 1 & \text{if } A[0,j] = 0 \\ 0 & \text{otherwise} \end{cases}.$$

Each entry can be computed in constant time. The overall complexity is therefore $\mathcal{O}(M \cdot N)$.
Code:

```
int maxArea = -1;
if(M == 0 || N == 0) {
    out.println(0);
    continue;
}
int S[][] = new int[M][N];
for(int j = 0; j < N; j++) {
    S[0][j] = B[0][j];
}
for(int i = 1; i < M; i++) {
    S[i][0] = B[i][0];
    for(int j = 1; j < N; j++) {
        S[i][j] = B[i][j];
        if(B[i][j] == 1) {
            int minOfThree = S[i-1][j];
            if(S[i][j-1] < minOfThree)
                minOfThree = S[i][j-1];
            if(S[i-1][j-1] < minOfThree)
                minOfThree = S[i-1][j-1];
            if(minOfThree+1 > S[i][j])
                S[i][j] = minOfThree+1;
        }
    }
}
for(int i = 0; i < M; i++)
    for(int j = 0; j < N; j++)
        if(S[i][j] * S[i][j] > maxArea)
             maxArea = S[i][j] * S[i][j];
out.println(maxArea);
```

  - $\mathcal{O}(M \cdot N \cdot \min(M, N))$: Either use a boolean table

$$\forall 1 \leq i \leq M, 1 \leq j \leq N, 1 \leq k \leq \min(M, N)$$
$$T[i, j, k] = \text{the square of size } k \text{ with south-east corner } (i, j) \text{ is filled with } 0s$$

    and use a recurrence relation similar to the one above, or use the same table as in the $\mathcal{O}(M \cdot N)$ solution and fill it in by considering only $T[i-1, j-1]$ and inspecting the values of the $T[i-1, y]$ and $T[x, j-1]$ every time in time $\mathcal{O}(\min(M, N))$.

- HS 2019 P2

  - $\mathcal{O}(|s| \cdot |D| \cdot \ell)$:
    Define the following table boolean T of size $\mathcal{O}(|s|)$:

$$\forall 1 \leq i \leq |s|,\ T[i] = s_1, \dots, s_i \text{ can be split into words from } D.$$

    The result can be extracted as $\max \{1 \leq i \leq |s| \mid T[i] \text{ is } \texttt{true}\}$.
    We fill in the table by order of increasing i. We have:

$$\forall 1 \leq i \leq |s|,\ T[i] = \exists j < i, \exists w \in D, (T[i - |w|] \vee i = |w|) \wedge \left(s_{i-|w|}, \dots, s_{i-1} = w\right).$$

    The first conjunct in the "exists" expresses that the string $s_1, \dots, s_{i-|w|}$ can be split in words of D (which in particular holds when it is empty) while the second conjunct checks that w terminates $s_1, \dots, s_i$. In other words, our recurrence formula simply expresses that in order to be splittable into

words from D, a string has to be obtainable by concatenating a (shorter) string which is itself splittable into words from D and a word from D.
We obtain a solution in time $\mathcal{O}(|s| \cdot |D| \cdot \ell)$ by naïvely iterating over all words from D for each position and testing whether all $|w| \leq \ell$ characters at the end of $s_1, \dots, s_i$ match those of w.

– $\mathcal{O}(|s| \cdot |D|)$:
We can easily spare on the cost of each comparison by stopping as soon as two non-matching characters are detected. In practice, this is sufficient to ensure the desired complexity.
The following computation is only for your understanding, and would not be required at the exam. Assuming that characters are uniformly distributed in both D and s over an alphabet of size $N \geq 2$, two strings of length k diverge on average at position

$$\begin{aligned}\sum_{i=1}^{k} i \left(\frac{1}{N^{i-1}} - \frac{1}{N^i} \right) &= \sum_{i=0}^{k-1} \frac{i+1}{N^i} - \sum_{i=1}^{k} \frac{i}{N^i} \\ &= 1 + \sum_{i=1}^{k-1} N^{-i} - \frac{k}{N^k} \\ &= \frac{1 - N^{-k}}{1 - \frac{1}{N}} - \frac{k}{N^k} \\ &\leq \frac{N}{N-1} \leq 2\end{aligned}$$

which is sufficient to ensure that we stop on average after $\mathcal{O}(1)$ steps when comparing two strings. In this case, the resulting complexity is $\mathcal{O}(|s| \cdot |D|)$.
Code:

```
int solve(String str) {
    int n = str.length();
    boolean T[] = new boolean[n];
    for (int i = 0; i < n; i++)
        for (String w: this.dictionary)
            if(w.length() <= i+1)
                T[i] |= (str.regionMatches(i-w.length()+1, w, 0, w.length())
                     && (i+1 == w.length() || T[i-w.length()]));
    for (int i = n-1; i >= 0; i--)
        if (T[i]) return i+1;
    return 0;
}
```

– $\mathcal{O}(|s| \cdot \ell \cdot \log |D| + |D| \log |D|)$:
Instead of testing all words in D, we can test whether the ℓ different suffixes of length 1 to ℓ of $s_1, \dots, s_i$ are in the dictionary. If we sort the dictionary first (in time $\mathcal{O}(|D| \log |D|)$), each of the ℓ prefixes can be searched for in time $\mathcal{O}(\log |D|)$, which yields the desired result.

– $\mathcal{O}((|s| + |D|) \cdot \ell)$:
Note: This solution is very advanced and not required for the exam.
We can further improve the efficiency of the search described in the previous solution by using a Trie, a simple data structure which efficiently stores a dictionary in a tree. We store the reverse of each word $w \in D$ in a Trie T. Building the tree costs $\mathcal{O}(\ell \cdot |D|)$, while all suffixes of s of length $\leq \ell$ can be tested in time $\mathcal{O}(\ell)$. This results in an algorithm with overall complexity $\mathcal{O}((|s| + |D|) \cdot \ell)$.

Chapter 3

1. (a) In each step of binary search, we check the difference between two integers (to detect base cases), compute a mean of two integers, read one entry in the array, perform one comparison and do at most

one recursive call. Each of these operations costs constant time. Therefore, the complexity of the algorithm is proportional to the number of steps needed. Since the size of the search area is halved in each iteration, there are $\log_2 n = \mathcal{O}(\log n)$ such steps, and the overall complexity is $\mathcal{O}(\log n)$.

(b) Consider algorithm 19. The pseudocode is similar in spirit to traditional 1D binary search. Depending

Algorithm 19 Two-dimensional binary sort

```
function Search($T, s, i = 0, j = n, k = 0, l = n$)
    if $i = j \wedge k = l$ then
        return $s = T[i, k]$
    $p \leftarrow \left\lfloor \frac{i+j}{2} \right\rfloor$
    $q \leftarrow \left\lfloor \frac{k+l}{2} \right\rfloor$
    $m \leftarrow T[p, q]$
    if $m < s$ then
        return Search$(T, s, i, p, q+1, l) \vee$ Search$(T, s, p+1, j, q+1, l) \vee$ Search$(T, s, p+1, j, k, q)$
    else
        return Search$(T, s, i, p, q+1, l) \vee$ Search$(T, s, i, p, k, q) \vee$ Search$(T, s, p+1, j, k, q$
```

on the result of a comparison between the searched element s and the entry m located in the "middle" of the considered subtable, we can safely reduce the search space to three of the four quadrants. The correctness argument is simple: if $T[p, q] = m < s$, then for all $i \leq p, j \leq q$, we have

$$T[i, j] \leq T[i, q] \leq T[p, q] = m < s$$

and thus we know that s is not contained in the top-left quadrant. The same holds for the bottom-right quadrant if $m \geq s$.

Regarding complexity, we have the following recurrence relation

$$T(n) = 3T(n/2) + \mathcal{O}(1) = 3T(n/2) + \mathcal{O}(n^0).$$

As $\log_2 3 > 0$, the Master theorem yields $T(n) = \mathcal{O}(n^{\log_3 2}) = \mathcal{O}(n^{1.58})$.

(c) Assume an SDLL object of the following type (here with integers):

```
class SDLL {
    SDLL prev; //may be null
    SDLL next; //may be null
    int value;
};
```

The following functions implement extraction, insertion and key increase.

```
int extract(SDLL sdll) {
    int value  = sdll.value;
    sdll.value = sdll.next.value;
    sdll.next  = sdll.next.next;
    return value; // here O(1), vs. binary tree O(h) or AVL O(log n)
}
int insert(SDLL sdll, int value) {
    while (sdll.value < value && sdll.next != null)
        sdll = sdll.next;
    new_sdll = SDLL();
    new_sdll.prev = sdll;
    new_sdll.next = sdll.next;
    new_sdll.value = value;
    if (sdll.next != null)
```

```
        sdll.next.prev = new_sdll;
    sdll.next = new_sdll;
} // here O(n), vs. binary tree O(h) or AVL O(log n)
int move_right(SDLL sdll) {
    if (sdll.next != null && sdll.next.value < sdll.value) {
        int value = sdll.next.value;
        sdll.next.value = sdll.value;
        sdll.value = value;
        move_right(sdll.next);
    }
}
int increase(SDLL sdll, int new_value) {
    assert new_value >= sdll.value;
    sdll.value = new_value
    move_right(sdll);
} // here O(n), vs. binary tree O(h) or AVL O(log n)
```

(d) Insertion of pre-sorted integers $1, 2, \dots, n$ (or $n, n-1, \dots, 1$) in a simple binary search tree produces a path graph with average insertion cost $\frac{1+2+\cdots+n}{n} = \mathcal{O}(n)$, while the average runtime is still $\mathcal{O}(\log n)$ with AVL trees.

2. **Sorting**

(a) See Vorlesungsnotizen.

(b) See Vorlesungsnotizen.

(c) When a sequence of $n-1$ tests does not result in any swap, this means (with the same notations as in the algorithm) that $A[i] \leq A[i+1]$ for all $i \in \{0, \dots, n-1\}$, and therefore

$$A[0] \leq A[1] \leq \cdots \leq A[n-1],$$

meaning that the array is already completely sorted.

(d) Consider the following algorithm:

Algorithm 20 Gnome sort

```
$i \leftarrow 0$
**while** $i < A.\texttt{length} - 1$ **do**
    **if** $A[i] \leq A[i+1]$ **then**
        $i \leftarrow i + 1$
    **else**
        Swap $A[i]$ and $A[i+1]$
        **if** $i > 0$ **then**
            $i \leftarrow i - 1$
```

i. We run gnome sort on $A = [3, 1415, 926, 535, 897, 932, 384, 626, 433]$. First, $i = 0$ is incremented until $i = 1$, and 1415 and 926 are swapped, resulting in $A = [3, 926, 1415, \dots]$, with subarray $A[: 3]$ sorted and $i = 0$. Then i is incremented again until $i = 1$. This is the end of the first phase. Then, i is incremented to 2, after which 1415 and 535 are swapped and i is decremented to 1 with $A = [3, 926, 535, 1415]$. Again, as $A[i] = A[1] = 926 > 535 = A[2] = A[i+1]$, we swap 926 and 535 and get $A = [3, 535, 926, 1415]$ and $i = 0$. Then i is incremented until $i = 2$, ending the second phase. The rest of the execution is similar. We observe a succession of phases (starting at phase 0), with phase j starting at position $i = j$ with $A[0..j]$ pre-sorted, decrementing j and swapping elements to put $A[j]$ in place as in insertion sort, and then re-incrementing i to reach $j + 1$.

ii. Let us make the above argument formal. For all $j \in \{0, \dots, n\}$, we will prove the following property by induction on j. $P(j)$: there exists a state of the algorithm such that $i = j$, $A[0..j]$ is sorted and at most $5j^2$ comparisons, arithmetic and memory operations have been performed since execution started.
The base case $P(0)$ is trivial, since the required property holds in the initial state. Now, let $0 \leq j < n$ such that $P(j)$ holds. After reaching the state described by $P(j)$, gnome sort compares $A[j]$ and $A[j + 1]$ (two memory reads, one comparison). If $A[j] \leq A[j + 1]$, then after incrementing i the subarray $A[0..j + 1]$ is already sorted, $i = j + 1$ and at most $5j^2 + 3 \leq 5(j+1)^2$ operations have been performed, which proves $P(j+1)$. Otherwise, $A[j]$ and $A[j+1]$ are swapped and i decremented. Let A^- denote the set of A before this swap. The sequence of a comparison, a swap, two tests and a decrement is repeated until $A[i] \leq A[i + 1]$ or $i = 0$. Then, we have

$$A[i] = A^-[i] \leq A[i + 1] = A[j + 1] \leq A[i + 2] = A^-[i + 1] \leq \cdots \leq A[j + 1] = A^-[j]$$

i.e. $A^-[j+1]$ has been successfully inserted into $A[0..j+1]$ within at most $5j$ operations. In the following $\leq j+1$ steps, which taken together cost at most $5(j+1)$ operations, i is incremented until $i = j + 1$. After $5j^2 + 5j + 5(j + 1) = 5j^2 + 10j + 5 = 5(j + 1)^2$ operations, we have $i = j + 1$ and $A[0..j + 1]$ is sorted, which is exactly $P(j + 1)$.
Property $P(n)$ shows that in time $\mathcal{O}(n^2)$, gnome sort correctly sorts the provided array. This upper bound is asymptotically tight, realized on decreasingly sorted inputs.

(e) Consider the following procedure:

i. The running time of ComputeRanks is $\mathcal{O}(n^2)$: the constant-time code within the inner loop is run $n(n - 1)$ times, and initialization only adds a linear extra cost.

ii. Consider algorithm 21. To show that this algorithm is correct, we need to prove two things: (i)

Algorithm 21 Rank sort

```
$r \leftarrow$ ComputeRanks$(A)$
$B \leftarrow A$.copy()
for $i \in \{0, \dots, A$.length $- 1\}$ do
    $A[r[i]] = B[i]$
```

that all fields of A are updated (ii) that A is eventually sorted.
Concerning (i), it is enough to prove that $0 \leq r[i] \neq r[j] < n$ for all $0 \leq i \neq j < n$, for then all n values of $R[i]$ must be different and thus cover the whole range from 0 to $n - 1$. The inequalities $0 \leq r[i] < n$ are trivial, as the $r[i]$ are initialized to 0 and updated at most $n - 1$ times. Now, assume by contradiction that $r[i] = r[j]$ for some $i < j$. If $A[j] < A[i]$, then for all $0 \leq k < n$ such that $A[k] < A[j]$ or $A[k] = A[j] \wedge k < j$, we also have $A[k] < A[i]$: hence $r[j] \leq r[i]$. But $r[i]$ is also implemented once for $A[j] < A[i]$, so $r[i] \geq r[j]+1 > r[j]$, a contradiction. If $A[i] > A[j]$, we obtain a similar contradiction. If $A[i] = A[j]$, we also have $r[j] \leq r[i]$, but now $r[i]$ needs to be implemented once for $A[j] = A[i] \wedge i < j$, again leading to $r[i] > r[j]$. Hence $r[i] \neq r[j]$.
Now that we know that A is eventually updated according to the ranks r (i.e. for all $0 \leq j < n$, $A[j] = A[r[i]] = B[i]$ for some i), we can prove (ii). Let $0 \leq i < j < n$. We show that $A[i] \leq A[j]$. By (i), we have i', j' such that $i = r[i']$, $j = r[j']$ and $A[i] = A^-[i']$, $A[j] = A^-[j']$, where A^- denotes the initial (unsorted) state of A. Assume that $A[j] = A^-[j'] < A[i] = A^-[i']$. Then, by the same reasoning as in (i), we have $r[j'] < r[i']$, which is exactly $j < i$, a contradiction. Hence, we have $A[i] \leq A[j]$ for all $0 \leq i < j < n$, and our algorithm is correct.
Besides the quadratic call to ComputeRanks, the rest of our algorithm is clearly linear. The overall complexity is therefore $\mathcal{O}(n^2)$.

iii. No. If you replace the condition by $A[j] < A[i]$, equal values are not treated properly. E.g. the array $[1, 1]$ would get ranks $[0, 0]$ instead of $[0, 1]$: rank 1 would not be assigned and lemma (i) above would fail.

3. (a) We will prove by induction over $h \in \mathbb{N}$ that every red-black tree of height h has at least $2^{\frac{h}{2}} - 1$ nodes. For $h = 0$, a red-black tree of height 0 has exactly $1 = 2^{\frac{0}{2}} - 1$ node. For $h = 1$. it has at least $2 \geq 2^{\frac{1}{2}} - 1$ nodes. Now, let $h > 1$ and assume that the property holds for all $h' < h$. Consider a red-black tree of height h.

   As $h > 1$, the children x and y of its root are necessarily both non-empty. Otherwise, one child (say x) would be empty, meaning that the number of black nodes on all paths from root to leaves in the tree would be 1. As $h > 1$, y has children. But, as the number of black nodes on all paths from root to leaves would be 1, neither y nor its children could be black, meaning that both y and its children would be red, which is not allowed (children of red nodes must be black). Hence, x and y are both non-empty.

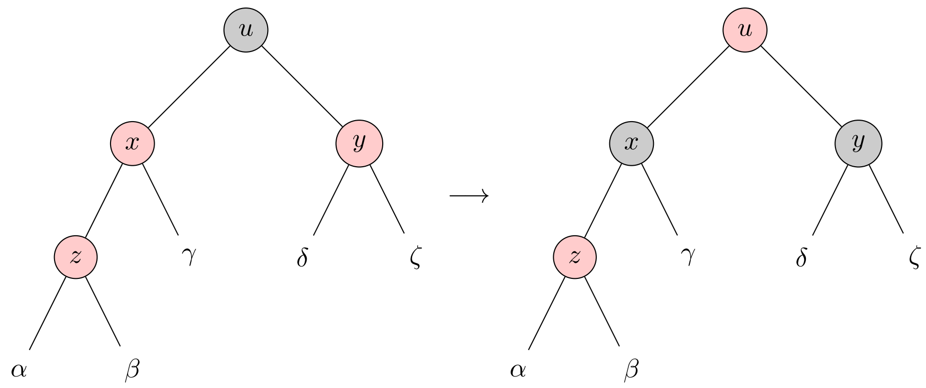
   If x is black, then its subtree is clearly a red-black tree of height $h - 1$. By our induction hypothesis, x has at least $2^{\frac{h-1}{2}} - 1$ nodes. If x is red, it has at least one child (since $h > 1$), which is necessarily black (by the red-black property). The subtree corresponding to this child contains at least $2^{\frac{h-2}{2}} - 1$ nodes by the induction hypothesis. In any case, the subtree rooted at x contains at least $2^{\frac{h-2}{2}} - 1$ nodes. As the same holds for y, we get at least

$$2 \cdot \left(2^{\frac{h-2}{2}} - 1\right) + 1 = 2^{\frac{h}{2}-1+1} - 2 + 1 = 2^{\frac{h}{2}} - 1$$

   nodes in total, which concludes the induction.

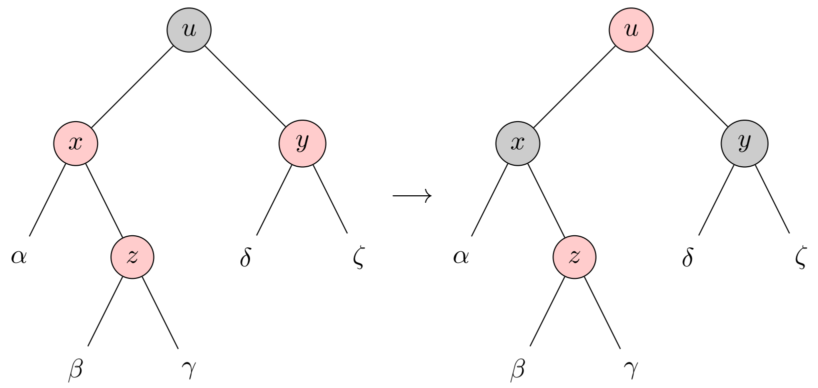
   We now have $n \geq 2^{\frac{h}{2}} - 1$, which is exactly $h \leq 2\log_2(n+1)$.

   (b) When z is inserted to the left of x and y is red, do



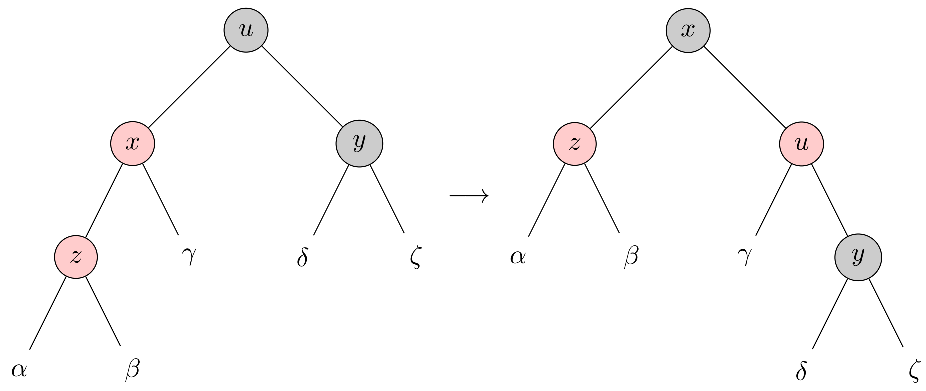
   and repeat upwards on u (u turning red might have broken the red-black property one level higher). Observe that the number of black nodes on a path from root to leaves is unchanged.

   When z is inserted to the right of x and y is red, do



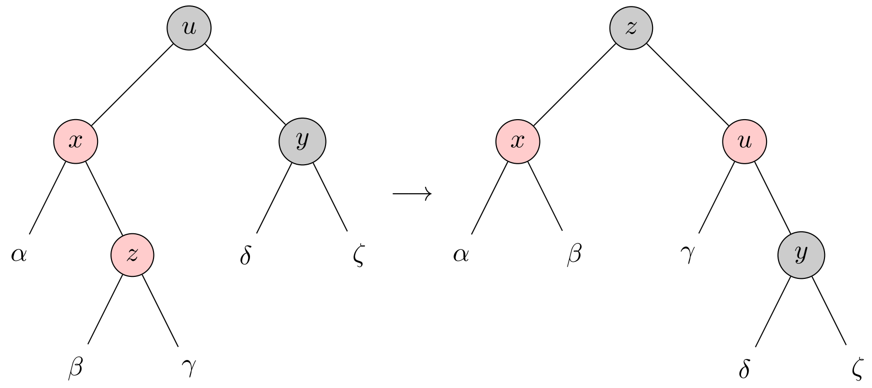
   and repeat upwards on u.

   When z is inserted to the left of x and y is black, perform the rotation



and stop: since the root of the subtree is black, we cannot have introduced any violation at a higher level.

Finally, when z is inserted to the right of x and y is black, perform the rotation



and stop.

Chapter 4

1. **Theory questions**

   (a) True or false?

   i. In an undirected graph, a tight bound for the number of edges is n^2.
   False, a better bound is $\binom{n}{2}$.

   ii. In an undirected graph, a tight asymptotic bound for the number of edges is $\Theta\left(n^2\right)$.
   True, as $\binom{n}{2} \in \Theta\left(n^2\right)$.

   iii. If the maximum degree of any node in an undirected graph G is 1, then this graph is a tree.
   False, the graph could be disconnected.

   iv. The complexity of computing the out-degree of a vertex v in an adjacency matrix is $\Theta\left(\deg v\right)$.
   False it's in $\mathcal{O}\left(n\right)$.

   v. The complexity of computing the sum of all out-degrees of vertices in an adjacency matrix is $\Theta\left(n^2\right)$.
   True.

   vi. The list of vertices accessible from a vertex v in a graph G can be computed by using at most $n-1$ multiplications of $n \times n$ matrices.
   True.

   vii. A connected graph is Eulerian (i.e. contains a Eulerian circuit) iff all its vertices have even degree.
   True.

   viii. A graph contains a Eulerian walk iff all its vertices have even degree.
   False, it can also exist if exactly two vertices have odd degree.

ix. Testing if a graph is Eulerian is NP-complete.
False, it can be done in polynomial time.

x. The post-order of BFS always gives a valid topological ordering.
False, the **reversed** post-order provides a valid topological sorting only if the graph contains no cycles.

(b) It will lead to undesirable bahaviour, whenever the graph contains a cycle e.g. consider K_3, in this case the algorithm will never stop.

(c) Any graph containing a cycle has no topological order. The algorithm will terminate but it will visit the nodes in an invalid order, as no order would be a valid one.

(d) A line has exactly 1 topological order. A cycle has exactly n orderings. A perfect binary tree has exaclty 2^n orderings. An empty graph with n nodes has $n!$ possible orderings.

2. **Shortest paths**

(a) Implement Dijkstra's algorithm in pseudo-code. You can assume that you are provided with an already implemented class `Table` that has the following interface:

```
class Table {
  Table(int n); //creates empty table with n uninitialized fields
  int get(int i) throws NotInitialized; //returns the value of
                                        //field i if initialized
  int set(int i, int v); //sets the value of field i to v,
                         //initializing it if necessary
  int smallest(); //returns i such that table.get(i) is minimal
}
```

(b) First initialize a union-find structure with n disjoint sets, each one representing a node in the graph. This takes time $\mathcal{O}(n)$
Then loop once over all edges in $\mathcal{O}(m)$ and for each edge, check in time $\mathcal{O}(\alpha(n))$ if the edge has weight 0; if it does, join (*union*) the sets connected by the edge. This takes time $\mathcal{O}(\alpha(n)m)$.
Then start a modified recursive BFS from source node. The modifications are the following: each time we take an edge we increment a counter, previously initialized at 0, and once inside a node we store that counter's value in our output array. This modifications clearly do not affect the asymptotic runtime of the BFS algorithm; ergo we consider the runtime of this step to be $\mathcal{O}(n+m)$.
Finally for every disjoint set in our union-find DS we find the minimum distance from the source to a node in that set and set every other node's distance to that. This takes time $\mathcal{O}(n)$.
Since every step is independent from the other, the total runtime is in $\mathcal{O}(\alpha(n)m+n)$

(c) The output of the Floyd-Warshall's algorihm is a 3 dimensional table. Each entry (i, j, k) represents the shortest path from i to j using only the vertices until k. Therefore the shortest path between two nodes i and j for a graph with n nodes is stored at the entry (i, j, n) which we can read in $\mathcal{O}(1)$, repeating this for $\mathcal{O}(n^2)$ pairs of nodes will cost $\mathcal{O}(n^2)$.

(d) We represent the map as a directed weighted graph over n nodes, each node representing a city in Switzerland. For two nodes $i, j \in V$ the edge (i, j) is created iff there is a road going from city i to city j on the map. The weight of such an edge is the cost of entering city j (the one we would be entering). Finally we run a simple shortest path algorithm (e.g. Dijkstra in $\mathcal{O}(m+n\log n)$) to find the shortest path from Visp to Mumpf, this is the cost of travelling there paying the tolls.

(e) We use the graph from the previous exercise but reverse all the edges (without changing the weights). Then on the new graph we run Dijkstra's algorithm twice, once from Visp, the other one from Mumpf. This way we obtain the shortest paths from all nodes to Visp and Mumpf without having to run Dijkstra n times. Once we have the cost of these shortest paths, for every node in the graph we add the cost of going to Visp and to Mumpf. If there are multiple cheapest paths we choose the one where the cost are divided better between the two sections. The overall runtime cost is $\mathcal{O}(2(m+n\log n)+n) = \mathcal{O}(m+n\log n)$.

3. **Minimal spanning trees**

   (a) Consider a tree $T = (V, E)$ defined as a connected graph with no cycles. Per definition there cannot be less than one path between any two vertices, otherwise it wouldn't be connected. It remains to show that such a path is unique. We'll prove uniqueness by contradiction (as it's often done), we'll therefore assume that there exist two different paths P_1 and P_2 between two vertices u and v. As P_1 and P_2 are different there has to be a well defined node x where P_1 and P_2 diverge at first. Similarly, as they have the same endpoint (i.e. v) but they diverged before, there has to be a well defined point y where they reconnect for the first time. Finally, if we consider the sub-paths of P_1 and P_2 from x to y, they are disjoint on every node except at their respective endpoints therefore they'd form a cycle. This contradicts the acyclicity assumption, therefore T can only contain one path for every pair of nodes.

   (b) Prove or disprove: For all vertices u, v of a graph G, the only path between u and v in an MST T of G is a shortest path between u and v in G.
   Solution
   This claim is false, and it can be easily disproved with a counterexample e.g.
   TODO: Insert Graph

   (c) Give an example of a graph on which Prim's and Kruskal's algorithms return different correct MSTs.
   Solution
   TODO: Insert example