Tutorial 1

Exercise 1: Test your understanding

Answer the following questions (and justify your answers):

1. Can a DTM ever write the blank symbol $\sqcup$ on its tape?
2. Can the tape alphabet Γ be equal to the input alphabet Σ?
3. Can the head of a DTM ever stay on the same cell for two subsequent steps of a computation?
4. Can the state set of a DTM consist of only a single state?
5. Suppose that a DTM has its head at a cell with symbol a and is in a state q which is different from t and r. How many distinct states may the machine be in after a transition?
6. Is it always the case that if a language is computable then it is also computably-enumerable?

Solution:
Here are the correct answers:

1. A DTM can write a $\sqcup$, since $\sqcup \in \Gamma$ and the transition function has type $\delta : (Q \setminus \{t, r\}) \times \Gamma \to Q \times \Gamma \times \{-1, +1\}$.
2. The tape alphabet Γ can never be equal to the input alphabet Σ, since $\sqcup \in \Gamma$, whereas $\sqcup \notin \Sigma$.
3. In general, the head either moves to the left or to the right. However, there is one special case, i.e., when the head is at the left-most cell (with number 1) in the direction is -1 (i.e., the head should move left): as there is no cell to the left, the head will stay at cell 1. Hence, the answer is yes (but only due to this special case!).
4. The state set of a DTM will always contain at least two states, since t and r are different and must be in the set of states.
5. A DTM will always be in exactly one state after a transition step. This is due to the fact that the machine is deterministic, i.e., δ is a *function* and hence to every element of $Q \times \Gamma$ it assigns exactly one element of $Q \times \Gamma \times \{-1, +1\}$.
6. Yes, every computable language is also computably-enumerable. This follows directly from the definitions. Computable languages are accepted by halting DTM's and these are of course a special case of DTM's.

Exercise 2: On precise formulations

Some of the following definitions are correct and some wrong (and some are even pure nonsense). Mark the correct definitions and for the incorrect ones underline the part of the definition which is wrong and explain why.

1. A language L is computably-enumerable if the language halts in the state t whenever $w \in L$.
2. A language L is computably-enumerable if there exists a DTM M such that M, given input w, halts in the accepting state t if and only if $w \in L$.
3. A language L is computably-enumerable if it halts in the accepting state t whenever $w \in L$.
4. A language L is computably-enumerable if there exists a DTM M which has a state t and is a member of L.

5. A language L is computably-enumerable if there exists a DTM M such that for any given input w the run of the machine M on w halts in the state t if $w \in L$, and it either does not terminate or halts in r if $w \notin L$.

6. A language L is computably-enumerable if every DTM M when run on a string w halts in the state t if and only if $w \in L$.

Solution:
Here are the correct answers:

1. A language L is computably-enumerable if the language halts in the state t whenever $w \in L$.
   WRONG: *A language does not halt, Turing machines can halt.*

2. A language L is computably-enumerable if there exists a DTM M such that M, given input w, halts in the accepting state t if and only if $w \in L$.
   CORRECT.

3. A language L is computably-enumerable if it halts in the accepting state t whenever $w \in L$.
   WRONG: *It is not clear what "it halts" refers to, probably to the language L and the same argument as in Item 1. applies.*

4. A language L is computably-enumerable if there exists a DTM M which has a state t and is a member of L.
   WRONG: *A DTM cannot be a member of L, as L is a language and therefore contains words and not Turing machines. Also, by definition, every DTM has an accepting state (which we denote by t).*

5. A language L is computably-enumerable if there exists a DTM M such that for any given input w the run of the machine M on w halts in the state t if $w \in L$, and it either does not terminate or halts in r if $w \notin L$.
   CORRECT.

6. A language L is computably-enumerable if every DTM M when run on a string w halts in the state t if and only if $w \in L$.
   WRONG: *Not every DTM, but there should exist at least one such a TM.*

Exercise 3: Turing Machines and their runs

Consider the following DTM $M = (Q, \Sigma, \Gamma, s, t, r, \delta)$ from Example 2.1.3 on page 75 of "Computability and Complexity" with

- $Q = \{s, t, r, h, e, g\}$,
- $\Sigma = \{a, b\}$,
- $\Gamma = \{a, b, \sqcup\}$, and
- δ given by the following table:

δ	a	b	$\sqcup$
s	$(h, \sqcup, +1)$	$(r, b, +1)$	$(t, \sqcup, +1)$
h	$(h, a, +1)$	$(h, b, +1)$	$(e, \sqcup, -1)$
e	$(r, a, +1)$	$(g, \sqcup, -1)$	$(r, \sqcup, +1)$
g	$(g, a, -1)$	$(g, b, -1)$	$(s, \sqcup, +1)$

Have a look at Examples 2.1.3, 2.1.5, and 2.1.11 before you attempt the exercise.

1. Give the initial configurations of M on the inputs $w_0 = aaa$ and $w_1 = aab$, both using the notation from Slide 25 and the simplified notation from Slide 26.
2. Give the runs of M on w_0 and w_1. Use the simplified notation to write down the configurations.
3. Are w_0 and w_1 accepted or rejected by M?

Solution:

1. Initial configurations:
   (a) On aaa: $[s, \tau_0, 1]$ with $\tau_0(n) = \begin{cases} a & \text{if } n \leq 3, \\ \sqcup & \text{if } n > 3, \end{cases}$ or, in the simplified notation, $[s, aaa\sqcup\ldots, 1]$.
   (b) On aab: $[s, \tau_1, 1]$ with $\tau_1(n) = \begin{cases} a & \text{if } n \leq 2, \\ b & \text{if } n = 3, \\ \sqcup & \text{if } n > 3, \end{cases}$ or, in the simplified notation, $[s, aab\sqcup\ldots, 1]$.
2. Runs:
   (a) On aaa:
   $$[s, aaa\sqcup\ldots, 1] \vdash_M [h, \sqcup, aa\sqcup\ldots, 2] \vdash_M [h, \sqcup aa\sqcup\ldots, 3] \vdash_M [h, \sqcup aa\sqcup\ldots, 4] \vdash_M$$
   $$[e, \sqcup aa\sqcup\ldots, 3] \vdash_M [r, \sqcup aa\sqcup\ldots, 4].$$
   (b) On aab:
   $$[s, aab\sqcup\ldots, 1] \vdash_M [h, \sqcup ab\sqcup\ldots, 2] \vdash_M [h, \sqcup ab\ldots, 3] \vdash_M [h, \sqcup ab\sqcup\ldots, 4] \vdash_M [e, \sqcup ab\sqcup\ldots, 3] \vdash_M$$
   $$[g, \sqcup a\sqcup\ldots, 2] \vdash_M [g, \sqcup a\sqcup\ldots, 1] \vdash_M [s, \sqcup a\sqcup\ldots, 2] \vdash_M [h, \sqcup\ldots, 3] \vdash_M [e, \sqcup\ldots, 2] \vdash_M [r, \sqcup\ldots, 3]$$
3. Both words are rejected by M, as the runs on them each end in a rejecting configuration.

Exercise 4: Constructing a simple DTM

Consider the language of words over the alphabet $\{0, 1\}$ whose first letter is equal to their last letter.

1. Write a formal definition of the language informally described above. In particular, think about corner cases and how to handle them.
2. Give a DTM that accepts the language you have defined in Part 1.
3. Give the run on 101.
4. Give the run on 100.
5. Give the run on he corner cases you considered.

Solution:

1. The corner cases we need to consider are the empty word and words of length one:
   - The empty word does not have any letters (it has length zero after all), so it in particular has no first and no last letter. Hence, it is not in our language.
   - On the other hand, words of length 1 (like, say 0) have a first letter and a last letter, which happen to be at the same position. Hence, they must be equal and such words are therefore in the language.

So, the formal definition of the language is

$$\{a_1 a_2 \cdots a_n \mid a_j \in \{0,1\} \text{ for all } 1 \leq j \leq n, n \geq 1, \text{ and } a_1 = a_n\}.$$

2. The DTM we construct works intuitively as follows:
   - Store the first letter using the states q_0 and q_1. If there is no first letter, reject.
   - In the states q_0 and q_1 go to the right until the first $\sqcup$ is reached.
   - When reaching this $\sqcup$ in state q_b for $b \in \{0,1\}$, then go left and into state chck_b. The head is then on the last letter of the input.
   - Compare that to the letter stored in the state chck_b and accept if they are equal, otherwise reject. As the stored letter is the first one, this final check compares the first to the last letter.

   Formally, we define $M = (Q, \Sigma, \Gamma, s, t, r, \delta)$ with
   - $Q = \{s, t, r, q_0, q_1, \mathrm{chck}_0, \mathrm{chck}_1\}$,
   - $\Sigma = \{0, 1\}$,
   - $\Gamma = \{0, 1, \sqcup\}$,
   - and δ given by the following table:

δ	0	1	$\sqcup$
s	$(q_0, 0, +1)$	$(q_1, 1, +1)$	$(r, \sqcup, +1)$
q_0	$(q_0, 0, +1)$	$(q_0, 1, +1)$	$(\mathrm{chck}_0, \sqcup, -1)$
q_1	$(q_1, 0, +1)$	$(q_1, 1, +1)$	$(\mathrm{chck}_1, \sqcup, -1)$
check_0	$(t, 0, +1)$	$(r, 1, +1)$	$(r, \sqcup, +1)$
check_1	$(r, 0, +1)$	$(t, 1, +1)$	$(r, \sqcup, +1)$

3. Run on 101:

$$[s, 101\sqcup\cdots, 1] \vdash_M [q_1, 101\sqcup\cdots, 2] \vdash_M [q_1, 101\sqcup\cdots, 3] \vdash_M$$
$$[q_1, 101\sqcup\cdots, 4] \vdash_M [\mathrm{chck}_1, 101\sqcup\cdots, 3] \vdash_M [t, 101\sqcup\cdots, 4]$$

4. Run on 100:

$$[s, 100\sqcup\cdots, 1] \vdash_M [q_1, 100\sqcup\cdots, 2] \vdash_M [q_1, 100\sqcup\cdots, 3] \vdash_M$$
$$[q_1, 100\sqcup\cdots, 4] \vdash_M [\mathrm{chck}_1, 100\sqcup\cdots, 3] \vdash_M [r, 100\sqcup\cdots, 4]$$

5. Run on ε: $[s, \sqcup\cdots, 1] \vdash_M [r, \sqcup\cdots, 2]$.
   Run on 0: $[s, 0\sqcup\cdots, 1] \vdash_M [q_0, 0\sqcup\cdots, 2] \vdash_M [\mathrm{chck}_0, 0\sqcup\cdots, 1] \vdash_M [t, 0\sqcup\cdots, 2]$.
   Run on 1: $[s, 1\sqcup\cdots, 1] \vdash_M [q_1, 1\sqcup\cdots, 2] \vdash_M [\mathrm{chck}_1, 1\sqcup\cdots, 1] \vdash_M [t, 1\sqcup\cdots, 2]$.

Exercise 5: Constructing a (slightly more complex) DTM

Consider the language $L = \{w\#w \mid w \in \{0,1\}^*\}$.

1. Give a halting DTM for L. Explain your solution in natural language.
2. Give the accepting run on $\# \in L$.
3. Give the accepting run on $011\#011 \in L$.
4. Give the rejecting run on $01\#00 \notin L$.

Solution:
See the slides for week 2.

Exercise 6: Challenge

Show that the following problem is computable:

$$\{n \in \mathbb{N} \mid \text{the decimal expansion of } \pi \text{ contains } n \text{ consecutive } 0\text{'s}\}$$

Describe your algorithm on an intuitive level without going into technical details.

Solution:
We begin with the following observation: If n is in P, then also every smaller number. For example, if $5 \in P$ (because the decimal expansion contains the sequence 00000), then also $4 \in P$ (as 0000 is a subsequence of the previous one). Hence, by taking the contraposition: if n is not in P, then also no larger number.

The only subsets of $\mathbb{N}$ satisfying this property are

- $\mathbb{N}$ itself, and
- the finite sets $\mathrm{Pref}(n) = \{0, 1, \ldots, n\}$ for $n \in \mathbb{N}$.

Note that all these sets are trivially computable.

So, P is either equal to $\mathbb{N}$ or equal to some $\mathrm{Pref}(n)$ (note that we do not know which case it is). So, P is computable (but we do not know which halting DTM is the one that computes our language).

Tutorial 2

Exercise 1: Robustness

We want to add another "direction" for the reading head of a Turing machine, i.e., 0 for "stay". Thus, a transition of the form $\delta(q, a) = (q', b, 0)$ updates the state to q' and changes the symbol at the current cell to b, but does not move the head.

Show that every DTM with the "stay" direction can be outcome-simulated by a standard DTM (i.e., without the "stay" direction).

Solution:
The idea is to simulate a "stay" by going right once and then going left again, thereby ending at the same cell again[1]. More formally, a transition of the form $\delta(q, a) = (q', b, S)$ will be simulated as follows:

- Replace a by b and go one cell right, update state to remember that we need to go left next and reach state q' then. This is done by changing to the auxiliary state q'_L.
- When in state q'_L leave the tape unchanged, move to the left and go to state q'.

Formally given a DTM $M = (Q, \Sigma, \Gamma, s, t, r, \delta)$ where $\delta\colon (Q \setminus \{t, r\}) \times \Gamma \to Q \times \Gamma \times \{-1, 0, +1\}$ we construct the machine $M' = (Q', \Sigma, \Gamma, s, t, r, \delta')$ with

- $Q' = Q \cup \{q_L \mid q \in Q\}$ (i.e., we add a copy q_L of each state $q \in Q$), and
- for all $q \in Q$ and $a \in \Gamma$, $\delta'(q, a) = \begin{cases} \delta(q, a) & \text{if } \delta(q, a) = (q', b, -1) \text{ for some } q' \in Q, b \in \Gamma, \\ \delta(q, a) & \text{if } \delta(q, a) = (q', b, +1) \text{ for some } q' \in Q, b \in \Gamma, \\ (q'_L, b, +1) & \text{if } \delta(q, a) = (q', b, 0) \text{ for some } q' \in Q, b \in \Gamma, \end{cases}$

  and $\delta'(q_L, a) = (q, a, -1)$.

Then, one can show by an induction over $n \in \mathbb{N}$ the following correctness statement:

> Let α_w be the initial configuration of M on w (which is also the initial configuration of M' on w) and let β be an arbitrary configuration of M (which is also a configuration of M'). Then, $\alpha_w \vdash_M^n \beta$ implies $\alpha_w \vdash_{M'}^* \beta$.

The induction is straightforward (moving the reading head of M to the left and to the right can directly be simulated by M', not moving the head of M can be simulated by two moves of M'), so let us argue that M' outcome-simulates M:

- If M halts on w, then $\alpha_w \vdash_M^n \beta$ for some halting configuration β. Then, by the correctness statement above, we have $\alpha_w \vdash_{M'}^* \beta$, i.e., M' also reaches a halting configuration when started with input w. Hence, M' halts on w.
- If M accepts w, then $\alpha_w \vdash_M^n \beta$ for some accepting configuration β. Then, by the correctness statement above, we have $\alpha_w \vdash_{M'}^* \beta$, i.e., M' also reaches an accepting configuration when started with input w. Hence, M' accepts w.

This is just one example of how this problem can be solved. Your solution, though different from the above presented one, can be still correct.

[1] Bonus question: Why not left and then right?

Exercise 2: More robustness

In the lecture, we have introduced Turing machines with tapes that are infinite in one direction. However, sometimes it is useful to consider Turing machines whose tape is infinite in both directions (doubly-infinite).

1. Formally define such a machine, its configurations, and the notion of successor configuration.
2. Show that every DTM with a doubly-infinite tape can be outcome-simulated by a DTM with a singly-infinite tape.
3. Show that every DTM with a singly-infinite tape can be outcome-simulated by a DTM with a doubly-infinite tape.

Thus, DTM's are also robust with respect to changing the format of the tape.

Solution:

1. We begin by defining such a machine and its semantics formally, following the definitions for singly-infinite DTM's as introduced in the lecture and the book.

   Formally, a DTM with doubly-infinite tape has still the form $M = (Q, \Sigma, \Gamma, s, t, r, \delta)$ as a standard DTM. However, a configuration now has the form $[q, \tau, \ell]$ where

   - $q \in Q$ is the current state,
   - $\tau \colon \mathbb{Z} \to \Gamma$ is the tape content (note that we now index cells with positive *and* negative numbers including zero), and
   - $\ell \in \mathbb{Z}$ is the current position of the head (which can be an positive or negative number including zero).

   Furthermore, the initial configuration on a word $w_1 \dots w_k$ is $[s, \tau, 1]$ where $\tau(n) = \begin{cases} w_n & \text{if } n \in \{1, \dots, k\} \\ \text{␣} & \text{otherwise.} \end{cases}$

   Accepting, rejecting, and halting configurations are defined as before.

   However, the notion of successor configurations has to be adapted: Now we do not have to worry about the special case of the head being stuck at the left end of the tape. Hence, given a configuration $[q, \tau, \ell]$, let $\delta(q, \tau(\ell)) = (p, a, d)$. Then the unique successor configuration of $[q, \tau, \ell]$ is the configuration $[p, \tau', \ell + d]$ with

   $$\tau'(n) = \begin{cases} a & \text{if } n = \ell, \\ \tau(n) & \text{otherwise.} \end{cases}$$

   The definitions of runs, accepting runs, and rejecting runs are as for standard machines. This completes the definition of such a machine.

2. Now, we show how to outcome-simulate a DTM with a doubly-infinite tape by a DTM with a singly-infinite tape, thereby showing that having a doubly-infinite tape does not increase the capabilities of a DTM. So, given a DTM M with a doubly-infinite tape, we construct a DTM M' with a singly-infinite tape that outcome-simulates M, i.e., if M halts (accepts) on an input w, then so does M' halt (accept). The question we have to tackle is how to simulate M's tape, which is infinite in both directions, by a singly infinite tape. Our idea is based on the fact that even a DTM with doubly-infinite tape can only visit finitely many cells in finitely many steps, i.e., only finitely many cells are non-empty. This finite region of the tape can therefore also be stored in a singly-infinite tape. If we

need more cells to the left we can just move the finite region on the tape, thereby freeing space on the left.

To delimit this region, M' has two additional tape symbols ℓ and r that are not in the tape alphabet of M, which we will use to mark the left end of the tape and the right end of the tape used so far. Then, whenever M wants to move left to a new cell (which M' cannot do on its left-bounded tape), M' will reach the leftmost cell, which is marked by ℓ, and then pause the simulation of M and shifts the whole tape content one position to the right, thereby freeing a new empty cell at the left. In order for the shifting to terminate, we use the right-end marker r, i.e., we only shift up to that position. Similarly, when new empty cells are required at the end of the finite region, we have to show the right-end marker r. Also, note that while M' moves the markers to create new empty cells, it has to remember the current state and head position when the simulation was interrupted. This can easily be done by either writing this information on the tape or by storing it in the states used to create the new empty cells.

Formally, on input w the DTM M' with singly-infinite tape does the following:

(a) First, replace the input w by $\ell w r$ on the tape so that ℓ is in the first cell.

(b) Move head to first position of w (cell 2).

(c) If accepting state of M is reached, accept. If rejecting state of M is reached, reject.

(d) Simulate one transition of M.

(e) If the head now points to a cell containing ℓ goto step 6, if it points to a cell containing r goto step 9, otherwise goto step 3.

(f) Shift content of tape (but for the leftmost cell marked ℓ) up to (and including) the cell marked r one cell to the right, thereby creating a new blank cell right after the position marked with ℓ.

(g) Move head to that blank cell.

(h) Goto step 3.

(i) Replace the r by a $\sqcup$, write r in next cell to the right, move head to newly created blank left of r.

(j) Goto step 3.

The resulting DTM M' does outcome-simulate M, which can be shown using a correctness statement similar to that one presented in Exercise 1.

3. Now, we show how to outcome-simulate a DTM with a singly-infinite tape by a DTM with a doubly-infinite tape. This is rather straightforward, as a doubly-infinite tape contains a singly-infinite tape. We just need to implement the special case where the head of a DTM with a singly-infinite tape stays at the first cell, if it moves left when in the first cell. To this end, we mark cell zero (the first one that is in a doubly-infinite tape, but not a singly-infinite tape) by a special symbol $\#$ that is not in the tape alphabet of M. Whenever, this cell is reached, the head is moved one cell to the right to correctly implement the special case.

   Formally, given a DTM $M = (Q, \Sigma, \Gamma, s, t, r, \delta)$ with a singly-infinite tape, we construct a DTM $M' = (Q', \Sigma, \Gamma \cup \{\#\}, s', t, r, \delta')$ with a doubly-infinite tape that outcome-simulates M as follows:

   - $Q' = Q \cup \{s', q_\#\}$ (where s' and $q_\#$ are not in Q),
   - δ' is defined as follows:
     - $\delta'(s', a) = (q_\#, a, -1)$: at the start, leave the first cell unchanged and move the head to the left.
     - $\delta'(q_\#, \sqcup) = (s, \#, +1)$: then place a $\#$ in the empty cell to the left of the input, move the head to the first letter of the input and go to the initial state of M.[2]
     - $\delta'(q, a) = \delta(q, a)$ for all $q \in Q$ and $a \in \Gamma$: simulate M.

[2] $\delta(q_\#, a)$ for $a \neq \sqcup$ can be defined arbitrarily, since will never be reached.

- $\delta'(q, \#) = (q, \#, +1)$: if the head reaches the cell marked with $\#$ (which means the special case has to be triggered) then move the head back to the right without changing the state.

The resulting DTM M' does outcome-simulate M, which can be shown using a correctness statement similar to that one presented in Exercise 1.

Note that there are again multiple ways this problem can be solved. Your solution, though different from the above presented one, can be still correct.

Exercise 3: Test your understanding

Which of the definitions of nondeterministic Turing machines given below is correct? Read them carefully before you answer.

1. An NTM M accepts input w if there exists more than one computation of M on w such that M halts in the state t.
2. An NTM M accepts input w if no computation of M on w halts in the state r.
3. An NTM M accepts input w if there exists a computation of M on w such that M halts in the state t.
4. An NTM M accepts input w if there is exactly one accepting computation of M on w.
5. An NTM M accepts input w if for every computation of M on w we have that M halts in state t.

Solution:

1. WRONG: An NTM accepts if there is at least one computation in which the machine halts in state t, but there need not be more than one such computation.
2. WRONG: "No computation of M halts in r" still allows all computations to loop. In this case, no accepting configuration is reached and the machine rejects.
3. RIGHT.
4. WRONG: There could be more than one accepting computation and our definition from the slides allows for this.
5. WRONG: Not *all* computations need to halt in the accepting state t, just at least one. It is irrelevant for acceptance how the other runs behave.

Exercise 4: Constructing NTM's

1. Give an NTM for the language of words over the alphabet $\{0, 1\}$ that contain the infix 110 by specifying all its components in detail.

2. A natural number $n > 1$ is composite if it is not prime, i.e., it is the product of two number greater than 1. So, the first composite numbers are $4 = 2 \cdot 2$, $6 = 2 \cdot 3$, , $8 = 2 \cdot 4$, , $9 = 3 \cdot 3$, etc.

   Describe (informally, but with enough detail) an NTM accepting the language $\{a^n \mid n \text{ is composite}\}$ over the alphabet $\{a\}$.

Solution:

1. Note that this language is regular, so we construct an NTM that essentially behaves like an NFA for that language: we guess nondeterministically the place where the infix 110 starts and then check whether this guess was correct.

   If the infix is in the word, then one guess will be correct. If it is not in the word, then all guesses will be wrong.

   Formally, we define $M = (Q, \Sigma, \Gamma, s, t, r, \delta)$ with

   - $Q = \{s, t, r, q_1, q_{11}\}$ where we use the states q_1 and q_{11} to remember that we have seen the infix 1 and 11, respectively.
   - $\Sigma = \{0, 1\}$,
   - $\Gamma = \{0, 1, \text{␣}\}$, and
   - δ is given by the following table:

δ	0	1	␣
s	$\{(s, 0, +1)\}$	$\{(s, 1, +1), (q_1, 1, +1)\}$	$(r, \text{␣}, +1)$
q_1	$\{(r, 0, +1)\}$	$\{(q_{11}, 1, +1)\}$	$\{(r, \text{␣}, +1)\}$
q_{11}	$\{(t, 0, +1)\}$	$\{(r, 1, +1)\}$	$\{(r, \text{␣}, +1)\}$

   Note that the machine accepts as soon as the infix 110 has been found, there is no need to read the rest of the word.

   It is instructive to construct the computation trees for some short inputs to recall how nondeterminism is used to accept this language.

2. Given an input a^n, use nondeterminism to guess two numbers $1 < n_0 \leq n$ and $1 < n_1 \leq n$ and then compute $n_0 \cdot n_1$. If this is equal to n, then accept, otherwise reject.

   If n is composite, then we can guess such n_0 and n_1 and at least one computation is accepting. Hence, the machine accepts n.

   If n is not composite, then there are no such n_0 and n_1 and all computations are rejecting. Hence, the machine rejects n.

Exercise 5: Challenge

In this exercise, we explore the origin of the name "computably-enumerable".

An *enumerator* E is a two-tape DTM with a special state q_{print} that outputs words on the second tape (by entering the state q_{print}). The language generated by an enumerator is the set of all words that are outputted on the second tape.

An enumerator is a lexicographical enumerator if it outputs words in the lexicographical order, meaning that once a word w is printed, all other words that are printed afterwards are in lexicographical order strictly larger than w, in particular implying that their length is at least $|w|$.

1. Prove that the class of languages that are generated by lexicographical enumerators is equal to the class of computable languages.
2. Prove that the class of languages that are generated by enumerators is equal to the class of computably-enumerable languages.

Solution:

1. We have to show two directions: a) if L is a language generated by a lexicographical enumerator then L is a computable language, and b) if L is a computable language then it is generated by some lexicographical enumerator.

   To prove a) we consider two cases. If L is a finite language then it is clearly a computable language (make sure you can argue why). So, let L be an infinite language generated by the lexicographical enumerator E. We construct a halting multi-tape DTM M for the language L in order to prove that L is a computable language:

   "On input w:

   1 Run the enumerator E (on two tapes) until it prints a word w'.
   2 If $w = w'$ then **accept**.
   3 If $|w| < |w'|$ then **reject**.
   4 Return to step 1 and continue running the enumerator."

   Now we first argue that M is a halting DTM, i.e. that on any input w it halts after finitely many steps. Clearly, as L is an infinite language, E keeps printing infinitely many words. However, there are only finitely many words of length at most $|w|$ and hence eventually the enumerator E must print a word of length larger than $|w|$, implying that M will reject (unless it accepted even earlier should E print the word w). The fact that the words are printed in lexicographical order guarantees that once E prints a word w' which is longer than w (and hence comes in lexicographical order after w), it is not possible that w is printed afterwards and the machine M can reject the input. Thus, M is a halting DTM with $L(M) = L$.

   To prove b), let L be a computable language that is accepted by a halting DTM M. We construct a lexicographical enumerator E generating L as follows:

   1 Let $w := \varepsilon$.
   2 Run M on w and if M accepts then print w.
   3 Let $w := w'$ where w' is the next word in the lexicographical order after w. Goto step 2.

   Clearly, the enumerator prints only words in L in lexicographical order as required. Moreover, because M is halting and does not loop on any word, all accepted words will be eventually printed.

2. Now, we explain how to generalize the constructions given above to computably-enumerable languages. Again, we have to show two directions: a) if L is a language generated by an enumerator then L is a computably-enumerable language, and b) if L is a computably-enumerable language then it is generated by some enumerator.

   To prove a), assume L is generated by the lexicographical enumerator E. We construct a multi-tape DTM M (not necessarily halting) for language L in order to prove that L is a computably-enumerable language:

   "On input w:

   1 Run the enumerator E (on two tapes) until it prints a word w'.
   2 If $w = w'$ then **accept**.

3 Return to step 1 and continue running the enumerator."

Now, if w is in L, then E will print it eventually which means M will accept it. If w is not in L, then E will never print it, which means M will not accept it (but also not reject it). Hence, we have $L(M) = L$ as required.

To prove b), let L be a computably-enumerable language, say $L = L(M)$ for a DTM M (not necessarily halting). We construct an enumerator E generating L as follows:

1 $n = 1$:
2 Run M on all w with $|w| \leq n$ for n steps. Print all w for which an accepting configuration is reached within these n steps.
3 $n = n + 1$.
4 Goto step 2.

Clearly, the enumerator prints only words in L as required. Moreover, let $w \in L$. We need to show that w is eventually printed by E. As w is in $L = L(M)$, there is some m so that M accepts w with a run with m steps. Then, w is printed by E when the variable n has value $\max\{|w|, m\}$. Altogether, E generates exactly L.

Tutorial 3

Exercise 1: Constructing an NTM

Give a halting NTM $(Q, \Sigma, \Gamma, s, t, r, \delta)$ for the language $\{ww \mid w \in \mathbb{B}^*\}$ by specifying all components of the NTM formally as done in the sample solution of Exercise 6 in week 1. Explain your solution.

Solution:

We begin by explaining the intuition: Given an input $x = x_0x_1 \cdots x_{n-1}$ our NTM nondeterministically guesses some $j \in \{0, 1, \ldots, n\}$ by splitting w into $x_0 \cdots x_{j-1}\#x_j \cdots x_{n-1}$. Now, the machine deterministically checks whether the resulting word is of the form $\{w\#w \mid w \in \mathbb{B}^*\}$. Note that we already constructed a halting DTM for that language. So, we will reuse that for the latter part.

We implement the split as follows:

1. If the tape is empty we accept immediately, as ε is in the language.
2. Otherwise, our NTM moves its head to the right through x leaving the cells unchanged (using the state p_s) until it nondeterministically puts a $\#$ on the tape and then copies the remaining part of x one cell to the right (using states p_0 and p_1).
3. If this nondeterministic choice is not made before x ends, the machine rejects.
4. Then it moves the head to the first letter of the input (now with a $\#$) and behaves like the halting DTM for $\{w\#w \mid w \in \mathbb{B}^*\}$ (state p_ℓ and the states of the DTM constructed for $\{w\#w \mid w \in \mathbb{B}^*\}$). To simplify the finding of the first letter, we assume our NTM has a doubly-infinite tape. Then, the first letter can be found by going left to the first empty cell and then one step to the right again.

   As we have seen in Exercise 2 on Exercise sheet 2, we can do so without loss of generality and thereby simplify the construction of our machine.

Formally, we define the halting NTM $(Q, \Sigma, \Gamma, s', t, r, \delta)$ with

- $Q = \{s', p_s, p_0, p_1, p_\ell, s, q_0, q_1, q_0^\#, q_1^\#, q_\ell, q_\ell^\#, q_s, t, r\}$,
- $\Sigma = \{0, 1\}$,
- $\Gamma = \{0, 1, \#, X, \textvisiblespace\}$,
- and the following $\delta\colon Q \times \Gamma \to Q \times \Gamma \times \{-1, +1\}$, with $b \in \mathbb{B}$:

$$\begin{aligned}
\delta(s',0) &= \{(p_s,0,+1),(p_0,\#,+1)\}\\
\delta(s',1) &= \{(p_s,1,+1),(p_1,\#,+1)\}\\
\delta(s',\#) &= (r,\#,+1)\\
\delta(s',\sqcup) &= (t,\sqcup,+1)\\
\delta(s',X) &= (r,X,+1)
\end{aligned}$$

$$\begin{aligned}
\delta(p_s,0) &= \{(p_s,0,+1),(p_0,\#,+1)\}\\
\delta(p_s,1) &= \{(p_s,1,+1),(p_1,\#,+1)\}\\
\delta(p_s,\#) &= (r,\#,+1)\\
\delta(p_s,\sqcup) &= (r,\sqcup,+1)\\
\delta(p_s,X) &= (r,X,+1)
\end{aligned}$$

$$\begin{aligned}
\delta(p_b,0) &= (p_0,b,+1)\\
\delta(p_b,1) &= (p_1,b,+1)\\
\delta(p_b,\#) &= (r,\#,+1)\\
\delta(p_b,\sqcup) &= (p_\ell,b,-1)\\
\delta(p_b,X) &= (r,X,+1)
\end{aligned}$$

$$\begin{aligned}
\delta(p_\ell,0) &= (p_\ell,0,-1)\\
\delta(p_\ell,1) &= (p_\ell,1,-1)\\
\delta(p_\ell,\#) &= (p_\ell,\#,-1)\\
\delta(p_\ell,\sqcup) &= (s,\sqcup,+1)\\
\delta(p_\ell,X) &= (r,X,+1)
\end{aligned}$$

$$\begin{aligned}
\delta(s,0) &= (q_0,X,+1)\\
\delta(s,1) &= (q_1,X,+1)\\
\delta(s,\#) &= (q_s,\#,+1)\\
\delta(s,\sqcup) &= (r,\sqcup,+1)\\
\delta(s,X) &= (r,X,+1)
\end{aligned}$$

$$\begin{aligned}
\delta(q_b,0) &= (q_b,0,+1)\\
\delta(q_b,1) &= (q_b,1,+1)\\
\delta(q_b,\#) &= (q_b^{\#},\#,+1)\\
\delta(q_b,\sqcup) &= (r,\sqcup,+1)\\
\delta(q_b,X) &= (r,X,+1)
\end{aligned}$$

$$\begin{aligned}
\delta(q_b^{\#},X) &= (q_b^{\#},X,+1)\\
\delta(q_b^{\#},b) &= (q_\ell,X,-1)\\
\delta(q_b^{\#},1-b) &= (r,1-b,-1)\\
\delta(q_b^{\#},\sqcup) &= (r,\sqcup,-1)\\
\delta(q_b^{\#},\#) &= (r,X,+1)
\end{aligned}$$

$$\begin{aligned}
\delta(q_\ell,X) &= (q_\ell,X,-1)\\
\delta(q_\ell,\#) &= (q_\ell^{\#},\#,-1)\\
\delta(q_\ell,0) &= (r,0,-1)\\
\delta(q_\ell,1) &= (r,1,-1)\\
\delta(q_\ell,\sqcup) &= (r,\sqcup,-1)
\end{aligned}$$

$$\begin{aligned}
\delta(q_\ell^{\#},0) &= (q_\ell^{\#},0,-1)\\
\delta(q_\ell^{\#},1) &= (q_\ell^{\#},1,-1)\\
\delta(q_\ell^{\#},X) &= (q_r,X,+1)\\
\delta(q_\ell^{\#},\sqcup) &= (r,\sqcup,-1)\\
\delta(q_\ell^{\#},\#) &= (q_\ell^{\#},\#,-1)
\end{aligned}$$

$$\begin{aligned}
\delta(q_s,0) &= (r,0,-1)\\
\delta(q_s,1) &= (r,1,-1)\\
\delta(q_s,X) &= (q_s,X,+1)\\
\delta(q_s,\sqcup) &= (t,\sqcup,-1)\\
\delta(q_s,\#) &= (r,\#,+1)
\end{aligned}$$

Note that the transition function for the states of the DTM for $\{w\#w \mid w \in \mathbb{B}^*\}$ is used unchanged here. Also note that the initial state of our NTM is s' while s (the initial state of the DTM for $\{w\#w \mid w \in \mathbb{B}^*\}$) is just a state of our NTM.

Exercise 2: Test your understanding

Consider the following claim:

> **Claim:** If L is a computable language and $L' \subseteq L$, then L' is a computable language too.

Is this claim true? If yes, prove it. If not, give a counter-example.

Solution:
This claim is wrong. Let $L = \Sigma^*$. This is clearly a computable language. However, any language L' is now a subset of L. Then any uncomputable language L' (and we know that uncomputable languages exist, e.g., HP) will satisfy the precondition of the claim (being a subset of L) but will break the conclusion as L' is not a computable language.

Exercise 3: Closure properties

Prove that the classes of computable languages and computably-enumerable languages are closed under concatenation.

Solution:
Let L_1 and L_2 be two computable languages. By definition there are halting DTM's M_1 and M_2 such that $L(M_1) = L_1$ and $L(M_2) = L_2$. We construct the following 3-tape NTM M accepting $L_1 \cdot L_2$:

1. "On input w:
2. Nondeterministically split the input string into two parts $w = w_1 w_2$ and copy w_1 on second tape and w_2 on the third tape.
3. On the second tape simulate M_1 on w_1.
4. If M_1 accepts then continue with step 5, else M rejects.
5. On the third tape simulate M_2 on w_2.
6. If M_2 accepts then M accepts else M rejects."

Now M is surely a halting NTM because both M_1 and M_2 are halting and $L(M) = L_1 \cdot L_2$. Any 3-tape halting NTM is equivalent to some single-tape halting DTM. Hence we have a halting DTM for the concatenation of L_1 and L_2.

The same construction also works if M_1 and M_2 are only computably enumerable (e.g., the M_i are not necessarily halting), yielding a NTM that accepts the concatenation $L_1 \cdot L_2$, but which is not necessarily halting.

Exercise 4: Spot the error

Consider the following two proofs that try to show that computably-enumerable languages are closed under complement.

Proof 1:

> Let L be a computably-enumerable language. By definition there is a DTM M such that $L(M) = L$. Consider the following DTM M':
>
> On input w:

1. Simulate M on w.
2. If M accepts then M' rejects. If M rejects then M' accepts.

Now we can see that $L(M') = \overline{L}$. Hence, $\overline{L}$ is computably-enumerable.

Proof 2:

Let L be a computably-enumerable language. By definition there is a DTM M such that $L(M) = L$. Consider the following DTM M':

On input w:

1. Simulate M on w.
2. If M accepts then M' rejects. If M rejects or does not halt on w then M' accepts.

Now we can see that $L(M') = \overline{L}$. Hence, $\overline{L}$ is computably-enumerable.

As we already know, computably-enumerable languages are not closed under complement and hence there must be an error in both proofs. Locate the errors by underlining them and explain (e.g. by giving a counter-example) where the problem is.

Solution:
Proof 1:

Let L be a recognizable language. By definition there is a Turing machine M such that $L(M) = L$. Consider the following machine M':

1. On input w:
2. Simulate M on w.
3. If M accepts then M' rejects. If M rejects then M' accepts.

Now we can see that $\underline{L(M') = \overline{L}}$. Hence $\overline{L}$ is recognizable.

The presented construction does not recognize the complement of the language L. Consider a TM M over $\Sigma = \{a\}$ with three states s, t and r such that $\delta(s, a) = (s, a, +1)$ and $\delta(s, \textvisiblespace) = (s, a, +1)$. On any given input w the machine M will not halt (it keeps moving the head to the right) and hence $L(M) = \emptyset$. Because M''s run on w will first try to simulate M on w, it will never halt either and hence $L(M') = \emptyset$, which is not equal to the complement of the empty language.

Proof 2:

Let L be a recognizable language. By definition there is a Turing machine M such that $L(M) = L$. Consider the following machine M':

1. On input w:
2. Simulate M on w.
3. If M accepts then M' rejects. If M rejects <u>or does not halt</u> on w then M' accepts.

Now we can see that $L(M') = \overline{L}$. Hence $\overline{L}$ is recognizable.

The problem here is that the machine M' has no way to verify whether the machine M on w does not halt or not. Hence conditions of the type "if a Turing machine M on an input w does not halt then do something" cannot be a part of any correct definition of a Turing machine because this test cannot be done <u>algorithmically.</u>

Exercise 5: Halting is CE

Show that the halting problem HP is computably-enumerable.

Solution:
We need to construct a DTM $\mathcal{H}$ that accepts the language

$$\text{HP} = \{\langle \ulcorner M \urcorner, w\rangle \mid M \text{ halts on input } w\},$$

i.e., if $w \in \text{HP}$, then $\mathcal{H}$ halts and accepts on input w, but if $w \notin \text{HP}$ then $\mathcal{H}$ either halts and rejects or does not halt on input w.

We define $\mathcal{H}$ as follows: On input w..

1. If w is not of the form $\langle \ulcorner M \urcorner, w'\rangle$ such that M is a Turing machine and w' is an input for M, then reject.
2. Otherwise simulate the run of M on w'.
3. If M halts, then accept (note that if M does not halt on w', then the simulation does not halt either, i.e., $\mathcal{H}$ does not halt on w and therefore does not accept w).

This Turing machine has the desired properties described above. In particular, it accepts $\langle \ulcorner M \urcorner, w'\rangle$ if M accepts w'. If M does not accept w' (i.e., it rejects or does not halt) then $\mathcal{H}$ rejects or does not halt either. Finally, if the input to $\mathcal{H}$ is of the wrong format (and therefore not in HP), then it rejects as well.

Tutorial 4

Exercise 1: Computability

Which of the following languages are computable? If you claim that a particular language is computable, describe a halting DTM for the language.

- $L_1 = \{\langle \ulcorner M \urcorner \rangle \mid M \text{ is a DTM and } M \text{ has more than } 5 \text{ states } \}$
- $L_2 = \{\langle \ulcorner M \urcorner, w \rangle \mid M \text{ is a DTM and } M \text{ accepts } w \text{ in less than } 1000 \text{ computational steps } \}$
- $L_3 = \{\langle \ulcorner M \urcorner, w \rangle \mid M \text{ is a DTM and } M \text{ accepts } w \text{ in a finite number of computational steps } \}$
- $L_4 = \{\langle \ulcorner M \urcorner, w \rangle \mid M \text{ is a DTM and } M \text{ on } w \text{ halts either in the accepting or rejecting state}\}$

Solution:

- The language L_1 is computable. A halting DTM M_1 for L_1 would first check whether the input encodes a DTM (this can be done by a halting DTM as argued in the lecture). If not, then it rejects. Otherwise, the input is of the form $\ulcorner M \urcorner$. In particular, it starts with a prefix $1^n\#$ where $n \in \mathbb{N}$ is the number of states of M. So, M_1 accepts if that number is greater than 5 and rejects otherwise. This can be done by checking that the first six letters of the input are all 1's.

  This DTM will surely halt and hence we have a halting DTM M_1 for L_1, i.e., L_1 is computable.

- The language L_2 is computable. Consider the following halting DTM M_2 accepting L_2:

  1. If the input is not of the form $\langle \ulcorner M \urcorner, w \rangle$, then reject.
  2. If it is, simulate M on w for 1000 steps.
  3. If M accepts during the simulation, accept.
  4. Otherwise, reject.

  Clearly the machine M_2 is halting DTM as it will halt after having simulated at most 1000 steps of M.

- Note that "M accepts w in a finite number of steps" is equivalent to "M accepts w". So, the language L_3 is equal to the language AP and hence we know that it is not computable.

- Note that a DTM only halts in the accepting or in the rejecting state. All other states are nonterminating. Hence, "M on w halts either in the accepting or rejecting state" is equivalent to "M on w halts". So, the language L_4 is equal to the language HP and hence we know that it is not computable.

Exercise 2: Computable functions

Argue that the following functions are computable by describing a DTM computing them.

1. $f(w) = X^{|w|}$.

2. $g(w) = \begin{cases} \ulcorner M' \urcorner & \text{if } w = \ulcorner M \urcorner \text{ for some DTM } M, \text{ where } M' \text{ is obtained from } M \\ & \text{by replacing every occurrence of the rejecting state in a transition by a state} \\ & \text{in which } M' \text{ never halts.} \\ \varepsilon & \text{otherwise.} \end{cases}$

Solution:

1. The first one is implemented by a DTM with five states, the initial one s, the accepting one t, and the rejecting one r and states q_r and q_ℓ going from left to right through the input replacing every letter (but the first one) by X (state q_r) and one to move the head back to the first letter of the tape. As this is still unchanged, we replace it by X and move the head left, which means it is still at the first cell.

   Formally, we define the transition relation as follows:

   - For every letter a in the input alphabet, we define $\delta(s, a) = (q_r, a, +1)$ (leaving the first letter unchanged). Also, we define $\delta(s, \textvisiblespace) = (t, \textvisiblespace, -1)$ to cover the special case of the empty input.
   - For every letter a in the input alphabet, we define $\delta(q_r, a) = (q_r, X, +1)$ (replacing each other input letter by X). Also, we define $\delta(q_r, \textvisiblespace) = (q_\ell, \textvisiblespace, -1)$ which starts the process of moving the head back to the first cell.
   - We define $\delta(q_\ell, X) = (q_\ell, X, -1)$ (to move the head left towards the first cell again) and $\delta(q_\ell, a) = (t, X, -1)$ for every letter a in the input alphabet (replacing the first letter that has been left unchanged so far).
   - For all other values, the transition function can be defined arbitrarily, as they are unreachable.

   Thus, on input w, after termination the tape only contains $X^{|w|}$ and the head is at termination on the first cell again.

2. First, let us recall the definition of $\ulcorner M \urcorner$:

$$1^{|Q|}\,0\,1^{|\Gamma|}\,0\,s\,0\,t\,0\,r\,0\,w_\delta$$

   where w_δ is the list of encodings of transitions. Each $\delta(q, b) = (p, a, d)$ is encoded by

$$q\,0\,\mathit{rep}_\Gamma(b)\,0\,p\,0\,\mathit{rep}_\Gamma(a)\,0\,\mathit{dir}(d)\,0$$

   where $\mathit{dir}(-1) = 1$ and $\mathit{dir}(+1) = 11$. Here, every state is assumed to be of the form 1^n for some $n \geq 1$.

   The DTM computing g works as follows: First it checks whether the input is an encoding of a Turing machine M by checking whether it has the format above. If not, it empties the tape and accepts.

   If it indeed encodes a DTM, let n be the number of states of the encoded DTM (i.e., the number of 1's before the first 0 in the input). Our DTM first adds another 1 in the beginning of the input (this adds another state).

   Now, the DTM moves its head to the 0 before the beginning of w_δ, i.e., to the fifth 0 in the input.

   Now, for each transition encoding

$$q\,0\,\mathit{rep}_\Gamma(b)\,0\,p\,0\,\mathit{rep}_\Gamma(a)\,0\,\mathit{dir}(d)\,0,$$

   it checks for each number encoded between the second and third 0 in such a block whether it is equal to r and, if yes, replaces it by 1^{n+1}, the encoding of the new state. Finally, it appends the encodings

$$1^{n+1}\,0\,\mathit{rep}_\Gamma(b)\,0\,1^{n+1}\,0\,\mathit{rep}_\Gamma(b)\,0\,11\,0,$$

   for each letter b of the tape alphabet, making sure that the machine encoded by the output does not terminate in the state 1^{n+1}.

   Once it has completed this, it moves the head back to the initial cell (which requires some mechanism to mark that cell in order to find it again).

Exercise 3: Computable vs. computably-enumerable

Let $L \subseteq \Sigma^*$ be a language. Show that L is computable if and only if L and its complement $\overline{L} = \Sigma^* \setminus L$ are computably-enumerable.

Solution:
To prove the equivalence of both statements, we prove the two implications separately, i.e.,

1. If L is computable, then L and its complement $\overline{L} = \Sigma^* \setminus L$ are computably-enumerable.
2. If L and its complement $\overline{L} = \Sigma^* \setminus L$ are computably-enumerable, then L is computable.

So, let us start with the first one. If L is computable, then so is $\overline{L}$ (apply closure of computable languages under complementation (see Lecture 3)). Now, we know that every computable language is also computably-enumerable. Hence, L and $\overline{L} = \Sigma^* \setminus L$ are indeed computably-enumerable, as required.

Now, consider the second implication. As L and $\overline{L}$ are computably-enumerable, there are one-tape DTM's M and $\overline{M}$ accepting L and $\overline{L}$, i.e.,

- If $w \in L$, then M on input w halts and accepts.
- If $w \notin L$, then M on input w either halts and rejects or does not halt at all.
- If $w \notin L$, then $\overline{M}$ on input w halts and accepts.
- If $w \in L$, then $\overline{M}$ on input w either halts and rejects or does not halt at all.

Consider the following two-tape DTM $\mathcal{D}$: On input w, simulate the run of M on w on the first tape and the run of $\overline{M}$ on w on the second tape. The simulations are run in parallel step-by-step.

If the simulation of M on w reaches the accepting state, then $\mathcal{D}$ acceps. If the simulation of $\overline{M}$ reaches the accepting state, then $\mathcal{D}$ rejects.

Let us first show that $\mathcal{D}$ is a halting DTM for L. If $w \in L$, then the simulation of M on w halts by reaching its accepting state (and the one of $\overline{M}$ will not reach its accepting state). Hence $\mathcal{D}$ halts and accepts w.

Dually, if $w \notin L$, then the simulation of $\overline{M}$ on w halts by reaching its accepting state (and the one of M will not reach its accepting state). Hence $\mathcal{D}$ halts and rejects w.

Altogether, we have $L(\mathcal{D}) = L$. As one of the cases "$w \in L$" and "$w \notin L$" is true for every input, we conclude that $\mathcal{D}$ halts on every input, i.e., it is a halting DTM.

Exercise 4: The complement of the halting problem

Show that $\overline{\mathrm{AP}}$ (the complement of the language AP from the lecture) is not computably-enumerable.

Solution:
Towards a contradiction, assume that $\overline{\mathrm{AP}}$ is computably-enumerable. Then, AP and $\overline{\mathrm{AP}}$ are computably-enumerable and therefore AP computable (by Exercise 3). However, we know that AP is not computable, i.e., we have derived a contradiction. Hence, $\overline{\mathrm{AP}}$ cannot be computably-enumerable.

Exercise 5: Reductions

Show that the language $\mathrm{NEP} = \{\ulcorner M \urcorner \mid L(M) \neq \emptyset\}$ is not computable by giving a reduction from a suitable non-computable problem.

Solution:
Given a DTM M and an input w we define the DTM $\mathcal{E}_{M,w}$ which behaves as follows:

1. It removes its own input from the tape.

2. It writes w on the tape.

3. It simulates M on w, i.e., if the simulation reaches the accepting state (the rejecting state) of M, then $\mathcal{E}_{M,w}$ accepts (rejects) as well.

Note that the behaviour of $\mathcal{E}_{M,w}$ does not depend on its input, as this is erased immediately.

Hence, we have the following: $\mathcal{E}_{M,w}$ accepts an input w' if and only if M accepts w. In particular, $\mathcal{E}_{M,w}$ has a nonempty language if and only if M accepts w, i.e., if $\langle \ulcorner M \urcorner, w \rangle$ is in AP.

Hence, we present a reduction from the acceptance problem AP. Let f be the computable function

$$f(x) = \begin{cases} \ulcorner \mathcal{E}_{M,w} \urcorner & \text{if } x \text{ is of the form } \langle \ulcorner M \urcorner, w \rangle \\ \varepsilon & \text{otherwise.} \end{cases}$$

Then, we have, as argued above

$$x \in \mathrm{AP} \Leftrightarrow f(x) \in \mathrm{NEP},$$

i.e., we have shown $\mathrm{AP} \leq_m \mathrm{NEP}$. Hence, NEP is not computable as well.

Tutorial 5

Exercise 1: Test your understanding

Assume that a language A is reducible to a language B. Which of the following claims are true?

1. A halting DTM for language A can be used to compute the language B.
2. If A is computable then B is computable too.
3. If A is not computable then B is not computable too.

Solution:

1. This claim is wrong, the correct claim is: a halting DTM for the language B can be used to compte the language A. See also the next item.
2. This claim is wrong. Consider, e.g., the case where A is the empty language (which is clearly computable) and B is the halting problem HP. The empty set is reducible to the halting problem HP (make sure you understand why this is the case!), but HP is not computable.

   But the claim is true the other way round: If B is computable then A is computable too.
3. This claim is true (see Lecture 4, slide 18).

Exercise 2: DTM universality

Consider the following decision problem:

> "Does a given DTM M accept all strings over its input alphabet?"

1. Define this problem as a language T.
2. Prove that T is not computable by a reduction from the acceptance problem.
3. Prove that NEP (see lecture 4) is not computable by a reduction from T.

Solution:

1. $T = \{\ulcorner M \urcorner \mid M \text{ is a DTM such that } L(M) = \{0,1\}^*\}$. Note that we restrict ourselves here to DTM's with alphabet $\{0,1\}$, as we have defined the encoding function only for such machines.
2. Given a Turing machine M and an input w for M, let $A_{M,w}$ be the Turing machine that behaves as follows:

   On input w':

   1. Remove w' from the tape and write w on the tape.
   2. Simulate M on w.
   3. Accept (reject) if M accepts (rejects) w.

   Then, we have $\langle \ulcorner M \urcorner, w\rangle \in \text{AP} \Leftrightarrow \ulcorner A_{M,w} \urcorner \in T$. Note that this statement only talks about valid encodings of Turing machines and inputs. As usual, in a reduction, we also need to take care of words that do not encode such valid encodings.

   The function

   $$f(x) = \begin{cases} \ulcorner A_{M,w} \urcorner & \text{if } x = \langle \ulcorner M \urcorner, w\rangle \text{ for some DTM } M \text{ and some input } w \\ \varepsilon & \text{otherwise} \end{cases}$$

   is computable and we have $x \in \text{AP} \Leftrightarrow f(x) \in T$. Hence, $\text{AP} \leq_m T$.

3. This is actually impossible! If you have a such a reduction, try to find the error. If you cannot find it, stop by to search together.

Exercise 3: NFA nonemptiness

Consider the following decision problem:

"Does a given NFA $\mathcal{A}$ have a nonempty language?"

1. Define this problem as a language N_{NFA}.
2. Argue that N_{NFA} is in P.

Solution:

1. To define this problem as a language, we first have to think about how to encode NFA's as words over some fixed alphabet. Let $\mathcal{A} = (Q, \Sigma, s, T, \Delta)$ with $\delta\colon Q \times \Sigma \to 2^Q$ be an NFA. Here, we proceed similarly to the encoding of DTM's.

   We assume without loss of generality that $Q = \{1, 11, \ldots, 1^n\}$ for some $n \geq 1$ with $s = 1$ and $\Sigma = \{1, 11, \ldots, 1^s\}$ for some $s \geq 1$. Furthermore, let $\Delta = \{\delta_0, \ldots, \delta_k\} = \{(q, a, q') \in Q \times \Sigma \times \Delta \mid q' \in \delta(q, a)\}$ be the set of transitions.

   Then, we encode $\mathcal{A}$ by the word

   $$\ulcorner \mathcal{A} \urcorner = 1^n 0 1^s 0 w_T 0 w_\Delta \in \{0,1\}^*$$

   where

   - $w_T \in \{0,1\}^n$ is the bitvector of length n such that w_T is 1 at position j if and only if $1^j \in T$ (the state 1^j is an accepting one), and
   - $w_\Delta = w_{\delta_0} 0 w_{\delta_1} 0 \cdots 0 w_{\delta_k}$ encodes the transitions such that for each $\delta_j = (q, a, q')$ we have $w_{\delta_j} = q 0 a 0 q'$.

   Then, we have

   $$N_{\text{NFA}} = \{\ulcorner \mathcal{A} \urcorner \mid \mathcal{A} \text{ is an NFA with nonempty language}\} \subseteq \{0,1\}^*.$$

2. Now, we construct a multi-tape halting DTM for N_{NFA} with polynomial time complexity.

   On input $w \in \{0, 1, \#, \$\}^*$:

   1. First check whether $w = \ulcorner \mathcal{A} \urcorner$ for some NFA $\mathcal{A}$. If not, reject.
   2. Otherwise, let n be the number of states encoded in $w = \ulcorner \mathcal{A} \urcorner$.
   3. Write the adjacency matrix of $\mathcal{A}$ on the tape, i.e., the matrix of the directed graph (V, E) with $V = \{1, \ldots, 1^n\}$ and $E = \{(1^i, 1^j) \in V \times V \mid (1^i, 1^k, 1^j)$ is a transitions of $\mathcal{A}$, for some letter $1^k\}$.
   4. For each accepting state 1^j check whether 1^j is reachable from $s = 1$ in (V, E). If yes, accept.
   5. Reject.

   Each step can be done in polynomial time, so the overall running time is polynomial as well. Hence, N_{NFA} is in P.

Exercise 4: Constructing a polynomial-time DTM

Describe (in sufficient detail) a one-tape halting DTM for the language $\{0^m1^m \mid m \geq 1\}$ that runs within time $\mathcal{O}(n \log_2 n)$.

Solution:

Consider the following halting DTM:

On input w:

1. Scan the tape and reject if w is not of the form 0^*1^*.
2. Repeat lines 3 and 4 as long as at least one 0 and at least one 1 are on the tape.
3. If there is an odd number of cells that have a 0 or a 1, then reject.
4. Else, replace every second 0 by an X and every second 1 by an X.
5. If every 0 and every 1 is replaced by an X then accept, else reject.

Let us analyze the time complexity: The first line takes $\mathcal{O}(|w|)$ steps and is executed only once. Checking whether the tape contains at least one 0 and at least one 1 takes again $\mathcal{O}(|w|)$ steps. Let us say line two is executed i times. We will later bound i.

The third and fourth line also each take $\mathcal{O}(|w|)$ steps and are executed i times. Finally, the fifth line takes $\mathcal{O}(|w|)$ steps and is executed once.

So, if we show that $i \leq \log |w|$, then we are done. Note that in each iteration of line 4, half of the 0's and half of the 1's on the tape are replaced by X's. An induction shows that after j executions, only $\frac{|w|}{2^j}$ many non X symbols are left on the tape. Thus, after $\log |w|$ executions, none are left and the "loop" is left. Hence, $i \leq \log |w|$ as required.

Tutorial 6

Exercise 1: Reminder on Big-O notation (part 1)

Which of these definitions of the O-notation are correct?

1. $f = \mathcal{O}(g)$ iff
   there exist positive integers c and n_0 such that for all $n \geq n_0$ we have that $f(n) \geq c \cdot g(n)$
2. $f = \mathcal{O}(g)$ iff
   for all positive integers c and n_0 it is the case that for all $n \geq n_0$ we have that $f(n) \leq c \cdot g(n)$
3. $f = \mathcal{O}(g)$ iff
   there exist positive integers c and n_0 such that for all $n \geq n_0$ we have that $f(n) \leq c \cdot g(n)$
4. $f = \mathcal{O}(g)$ iff
   for all positive integers c and n_0 it is the case that there exists an $n \geq n_0$ such that $f(n) \leq c \cdot g(n)$

Solution:

1. incorrect: f should be bounded by g, i.e., $f(n) \leq c \cdot g(n)$.
2. incorrect: here, we require the property for all c and all n_0, which is too strong.
3. correct
4. incorrect: again, the quantifiers are flipped which yields an incorrect definition.

Exercise 2: Reminder on Big-O notation (part 2)

Which of the following claims are true? Give precise arguments (proofs) for your answers.

1. $3n^2 + 2n + 7 = \mathcal{O}(n^2)$
2. $n^2 = \mathcal{O}(n \log n)$
3. $3^n = \mathcal{O}(2^n)$ (Hint: $3 = 2^{\log_2 3}$)
4. $3^n = \mathcal{O}(2^{n^2})$ (Hint: $3 = 2^{\log_2 3}$)

Solution:

1. True. Take $c = 12$ and $n_0 = 1$. Then clearly $3n^2 + 2n + 7 \leq 3n^2 + 2n^2 + 7n^2 \leq 12n^2$ and so $3n^2 + 2n + 7 \leq c \cdot n^2$ for all $n \geq n_0 = 1$.
2. False. By contradiction. Assume that there are constants c and n_0 such that $n^2 \leq c \cdot n \log n$ for all $n \geq n_0$. This would mean that $n \leq c \cdot \log n$ and hence $\frac{n}{\log n} \leq c$ for all $n \geq n_0$. However, this cannot be the case as $\frac{n}{\log n}$ goes to ∞ as n goes to ∞ and hence the expression $\frac{n}{\log n}$ will eventually be larger than any chosen constant c. Contradiction.
3. False. By contradiction. Assume that there are constants c and n_0 such that $3^n \leq c \cdot 2^n$ for all $n \geq n_0$. This would mean that hence $\frac{3^n}{2^n} = (\frac{3}{2})^n = (1.5)^n \leq c$ for all $n \geq n_0$. However, this cannot be the case as $(1.5)^n$ goes to ∞ as n goes to ∞ and hence the expression $(1.5)^n$ will eventually be larger than any chosen constant c. Contradiction.
4. We have $3^n = (2^{\log_2 3})^n = 2^{n \log_2 3}$. As $\log_2 3 \leq 2$, we have $n \log_2 3 \leq 2n \leq n^2$ for all $n \geq 2$. Thus, we can pick $c = 1$ and $n_0 = 2$ and then have
   $$3^n = 2^{n \log_2 3} \leq c 2^{n^2}$$
   for all $n \geq n_0$.

Exercise 3: Test your understanding

1. Which of the following statements about the class P are correct?
   (a) P is the class of all languages that are computable by single-tape DTM's running in polynomial time.
   (b) P is the class of all languages such that if $w \in P$ then there is a single-tape DTM which accepts the string w in polynomial time.
   (c) P is the class of all languages that are computable by multi-tape DTM's running in polynomial time.
   (d) A language L belongs to P iff there is a constant k and a halting DTM M running within time $O(n^k)$ such that $L = L(M)$.
   (e) A language L belongs to P iff $L \in \text{TIME}(2^n)$.
2. Give a language in P that we haven't discussed in the lecture and exercises.
3. Does HP belong to P?

Solution:

1. (a) Correct.
   (b) Incorrect. The elements of P are languages, not strings! So, the expression $w \in$ P does not make any sense.
   (c) Correct. Note that any multi-tape TM running in polynomial can be simulated by a single-tape TM running also in polynomial time.
   (d) Correct.
   (e) Incorrect. The implication from left to right of course holds, but the one from right to left does not hold.
2. The following languages (for example) belong to P:
   - $\emptyset$
   - $\{a^k b^k c^k \mid k \geq 0\}$
   - $\{G \mid G \text{ is a connected graph }\}$
   - $\{\ulcorner M \urcorner \mid M \text{ is a TM which has more than } 10 \text{ states }\}$
   - every regular language
   - every context-free language
   - and many more
3. The language HP does not belong to P because the language HP is not computable and P contains only computable languages (as it is defined as a class of languages accepted by halting DTM's).

Exercise 4: DNF-SAT in P

A Boolean formula φ is in disjunctive normal form (DNF) if it is of the form $\varphi = \bigvee_{i=1}^{n} \bigwedge_{j=1}^{m_i} \ell_{i,j}$ such that each $\ell_{i,j}$ is a literal, i.e., either a variable x or a negated variable $\neg x$. For example, $(x \wedge \neg y \wedge z) \vee (\neg x) \vee (\neg y \wedge \neg z)$ is in DNF.

Show that satisfiability of formulas in DNF is in P, i.e.,

$$\text{DNF-SAT} = \{\varphi \mid \varphi \text{ is in DNF and has a satisfying assignment}\} \in \text{P}.$$

Note that the example formula above is in DNF-SAT.

Solution:
Let $\varphi = \bigvee_{i=1}^{n} \bigwedge_{j=1}^{m_i} \ell_{i,j}$ be a fomula in DNF. Then, φ is satisfiable if and only if one of the disjuncts $\bigwedge_{j=1}^{m_i} \ell_{i,j}$ is satisfiable.

We say that a variable x is contradictory in $\bigwedge_{j=1}^{m_i} \ell_{i,j}$ if $\ell_{i,j} = x$ for some j and $\ell_{i,j'} = \neg x$ for some j'. A formula of the form $\psi = \bigwedge_{j=1}^{m_i} \ell_{i,j}$ is satisfiable if and only if there is no contradictory variable x in ψ. For example, $x \wedge \neg y \wedge z \wedge \neg z'$ is satisfiable while $x \wedge \neg y \wedge z \wedge \neg x$ is not.

Thus, the following algorithm shows that DNF-SAT is in P. Given an input w:

1. If w does not encode a formula in DNF, reject (otherwise, let $\bigvee_{i=1}^{n} \bigwedge_{j=1}^{m_i} \ell_{i,j}$ be the formula encoded by w).
2. $i = 1$.
3. If there is no contradictory variable in $\bigwedge_{j=1}^{m_i} \ell_{i,j}$ then accept.
4. Increment i.
5. If $i = n + 1$ goto step 6, else goto step 3.
6. Reject.

Exercise 5: 3-coloring in NP

Show that 3-coloring is in NP.

Solution:
The following Turing machine is a halting NTM with polynomial time-complexity:

On input w:

1. If w is not of the form $\mathbb{B}^{n^2}$ for some n, then reject (if the machine does not reject, then w is interpreted as the adjacency matrix of a graph with n vertices).
2. Guess $c_1 \cdots c_n \in \{R, G, B\}^n$.
3. For each edge (i, j) encoded by w check whether $c_i = c_j$. If yes, reject.
4. Accept.

Tutorial 7

Exercise 1: Properties of NP-complete problems

1. Prove the theorem on Slide 20 of Lecture 7: Let L be NP-complete. If $L \in P$, then P = NP.
2. Complete the picture on Slide 20 of Lecture 7: Show that if P = NP then every nontrivial problem (i.e., every $L \subseteq \Sigma^*$ with $L \neq \emptyset$ and $L \neq \Sigma^*$) in NP is NP-complete.

Solution:

1. Recall the theorem from Lecture 6, slide 19 stating that $A \leq_p B$ and $B \in$ P implies $A \in$ P.

   Now, we have that L is NP-complete, so we have by definition $A \leq_p L$ for every $A \in$ NP. Also, we have that L is in P.

   Hence, by applying the theorem, we obtain $A \in$ P for every $A \in$ NP, i.e., NP $\subseteq$ P.

   The other inclusion, i.e., P $\subseteq$ NP, holds by definition (see Lecture 6, Slide 13). Hence, we have shown P = NP as required.

2. Let $L \in$ NP be an arbitrary nontrivial problem in NP. We need to prove that L is NP-complete, i.e., that L is NP-hard and in NP. The latter already holds by the choice of L, so we only need to prove the former.

   By our assumption P = NP, there is a polynomial-time halting DTM for L', call it M'. Also, as L is nontrivial, there are words $w_L^+ \in L$ and $w_L^- \notin L$.

   Now, consider the following function f:

   $$f(w) = \begin{cases} w_L^+ & \text{if } w \in L', \\ w_L^- & \text{if } w \notin L'. \end{cases}$$

   The following Turing machine computes f in polynomial time, i.e., f is polynomial-time computable:

   On input w:

   (a) Run M' on w.

   (b) If M' accepts, empty the tape and write w_L^+.

   (c) If M' rejects, empty the tape and write w_L^-.

   (d) Terminate.

   Each step takes a polynomial number of steps, so the overall running time is polynomial. Further, the Turing machine obviously computes f.

   To conclude, we show that f witnesses $L' \leq_p L$:

   - If $w \in L' = L(M')$, then M' accepts w, which implies $f(w) = w_L^+ \in L$.
   - If $w \notin L' = L(M')$, then M' rejects w, which implies $f(w) = w_L^- \notin L$.

   Altogether, we have $w \in L' \Leftrightarrow f(w) \in L$, as required.

   Hence, L is NP-complete.

Exercise 2: Vertex cover in NP

Show that VC is in NP.

Solution:

Consider the following Turing machine. On input $w \in \mathbb{B}^*$ and $k \in \mathbb{N}$:

1. Check whether $|w| = n^2$ for some $n \in \mathbb{N}$. If not, reject.
2. If $k > n$ reject.
3. Guess a $0/1$-vector $c_1 \cdots c_n$ of length n with exactly k 1's.
4. For each edge (i, j) of the graph encoded by w check whether $c_i = 1$ or $c_j = 1$.
5. If yes, accept. Otherwise reject.

The Turing machine is a halting NTM with polynomial time-complexity and accepts VC.

Exercise 3: CNF-SAT and 3SAT are NP-hard

1. Show that CNF-SAT is NP-hard by adapting the construction of $\varphi_{M,w}$, so that the resulting formula is in CNF.
2. Show that 3SAT is NP-hard by giving a polynomial-time reduction from CNF-SAT to 3SAT.
3. Are both problems NP-complete?

Solution:

1. When we proved that SAT is NP-hard, we constructed, given a polynomial-time halting NTM M and an input w, a formula $\varphi_{M,w}$ that is satisfiable if and only if M accepts w. Here, we show that we can construct a CNF formula $\varphi'_{M,w}$ with the same properties.

   Recall that $\varphi_{M,w}$ is of the form

   $$\varphi_{M,w} = \varphi_{\text{setup}} \wedge \varphi_{\text{init}} \wedge \varphi_{\text{move}} \wedge \varphi_{\text{accept}}$$

   If we can, for each of the four conjuncts, construct an equivalent formula in CNF, then we have achieved our goal, as CNF-formulas are closed under conjunction: if φ_1 and φ_2 are in CNF, then so is $\varphi_1 \wedge \varphi_2$.

   - Recall that we have

     $$\varphi_{\text{setup}} = \bigwedge_{i=0}^{p(n)+2} \bigwedge_{j=1}^{p(n)} \bigvee_{e \in E} \left(x_{i,j,e} \wedge \bigwedge_{e' \in E \setminus \{e\}} \neg x_{i,j,e'} \right)$$

     expressing that every grid cell (i, j) contains some symbol e, but no other symbol $e' \neq e$.
     This property can equivalently be expressed by the formula

     $$\left(\bigwedge_{i=0}^{p(n)+2} \bigwedge_{j=1}^{p(n)} \bigvee_{e \in E} x_{i,j,e} \right) \wedge \left(\bigwedge_{i=0}^{p(n)+2} \bigwedge_{j=1}^{p(n)} \bigwedge_{e \in E} \bigwedge_{e' \in E \setminus \{e\}} (\neg x_{i,j,e} \vee \neg x_{i,j,e'}) \right)$$

     where the first part expresses that there is at least one symbol e in every grid cell (i, j) and the second part expresses that for every grid cell (i, j) and every pair $e \neq e'$, grid cell (i, j) contains at most one of the symbols e and e'. Together, the two parts express that every grid cell contains exactly one symbol.

- Recall that we have

$$\varphi_{\text{init}} = x_{0,1,\$} \wedge x_{1,1,(w_0,s)} \wedge \left(\bigwedge_{i=2}^{n} x_{i,1,w_{i-1}} \right) \wedge \left(\bigwedge_{i=n+1}^{p(n)+1} x_{i,1,\textvisiblespace} \right) \wedge x_{p(n)+2,1,\$},$$

  which is already in CNF (where every disjunct has length 1).

- Recall that we have

$$\varphi_{\text{move}} = \bigwedge_{i=1}^{p(n)+1} \bigwedge_{j=2}^{p(n)} \psi_{i,j}$$

  with

$$\psi_{i,j} = \bigvee_{\begin{array}{|c|c|c|}\hline e_3 & e_4 & e_5 \\ \hline e_0 & e_1 & e_2 \\ \hline \end{array}} x_{i-1,j-1,e_0} \wedge x_{i,j-1,e_1} \wedge x_{i+1,j-1,e_2} \wedge x_{i-1,j,e_3} \wedge x_{i,j,e_4} \wedge x_{i+1,j,e_5}$$

  where the disjunction ranges over all legal windows.

  Here, we need to apply a trick to obtain an equivalent formula in CNF. Let W denote the set of of legal windows. We add a new type of variable: $x_{i,j,w}$ for $1 \leq i \leq p(n)+1$, $2 \leq j \leq p(n)$, and $w \in W$ with the intended meaning that window$(i,j) = w$ if and only if $x_{i,j,w}$ is true.

  So, as before, we express that at each position there is one window and then connect variables $x_{i,j,w}$ to the variables $x_{i,j,e}$ with $e \in E$ (which express the actual grid). Formally, we define

$$\varphi'_{\text{move}} = \left(\bigwedge_{i=1}^{p(n)+1} \bigwedge_{j=2}^{p(n)} \bigvee_{w \in W} x_{i,j,w} \right) \wedge \left(\bigwedge_{i=1}^{p(n)+1} \bigwedge_{j=2}^{p(n)} \bigwedge_{w \in W} (\neg x_{i,j,w} \vee \psi_{i,j,w}) \right)$$

  where

$$\psi_{i,j,w} = x_{i-1,j-1,e_0} \wedge x_{i,j-1,e_1} \wedge x_{i+1,j-1,e_2} \wedge x_{i-1,j,e_3} \wedge x_{i,j,e_4} \wedge x_{i+1,j,e_5}$$

  for $w = \begin{array}{|c|c|c|}\hline e_3 & e_4 & e_5 \\ \hline e_0 & e_1 & e_2 \\ \hline \end{array}$. The first part expresses the selection of some legal window for (i,j) and the second part expresses that the grid corresponds to that selected window (note that $(\neg x_{i,j,w} \vee \psi_{i,j,w})$ is equivalent to the implication $(x_{i,j,w} \to \psi_{i,j,w})$).

  However, the formula is still not in CNF, as $\psi_{i,j,w}$ contains conjunctions. But, an application of the distributive law yields

$$\begin{aligned} \neg x_{i,j,w} \vee \psi_{i,j,w} =& \neg x_{i,j,w} \vee (x_{i-1,j-1,e_0} \wedge x_{i,j-1,e_1} \wedge x_{i+1,j-1,e_2} \wedge x_{i-1,j,e_3} \wedge x_{i,j,e_4} \wedge x_{i+1,j,e_5}) \\ =& (\neg x_{i,j,w} \vee x_{i-1,j-1,e_0}) \wedge (\neg x_{i,j,w} \vee x_{i,j-1,e_1}) \wedge (\neg x_{i,j,w} \vee x_{i+1,j-1,e_2}) \wedge \\ & (\neg x_{i,j,w} \vee x_{i-1,j,e_3}) \wedge (\neg x_{i,j,w} \vee x_{i,j,e_4}) \wedge (\neg x_{i,j,w} \vee x_{i+1,j,e_5}) \end{aligned}$$

  The resulting formula is thus in CNF.

- Finally, recall that

$$\bigvee_{i=1}^{p(n)+1} \bigvee_{j=1}^{p(n)} \bigvee_{a \in \Gamma} x_{i,j,(a,t)},$$

  which is already in CNF.

All formulas are of polynomial size in $|\ulcorner M \urcorner| + |w|$ and therefore satisfy all our requirements. Thus, we can define

$$\varphi'_{M,w} = \varphi'_{\text{setup}} \wedge \varphi_{\text{init}} \wedge \varphi'_{\text{move}} \wedge \varphi_{\text{accept}}.$$

2. We present a reduction from CNF formulas φ into 3CNF formulas $f(\varphi)$ that preserves satisfiability, i.e., φ is satisfiable if and only if $f(\varphi)$ is satisfiable.

   So, let $\varphi = \bigwedge_{i=1}^{n} \bigvee_{j=1}^{m_i} \ell_{i,j}$ be an arbitrary formula in CNF. We show how to turn each $\bigvee_{j=1}^{m_i} \ell_{i,j}$ into a 3CNF formula ψ_i. Then, the desired 3CNF formula is $f(\varphi) = \bigwedge_{i=1}^{n} \psi_i$, which is indeed in 3CNF.

   So, consider some $\bigvee_{j=1}^{m_i} \ell_{i,j}$. We distinguish several cases for m_i:

   - If $m_i = 3$, i.e., the disjunction has already the right length, then we set $\psi_i = \bigvee_{j=1}^{m_i} \ell_{i,j}$.
   - If $m_i = 1$, then we define $\psi_i = \ell_{i,1} \vee \ell_{i,1} \vee \ell_{i,1}$ (i.e., we repeat the literal three times). This is logically redundant, but does not influence satisfiability.
   - If $m_i = 2$, then we define $\psi_i = \ell_{i,1} \vee \ell_{i,2} \vee \ell_{i,2}$ (i.e., we repeat the second literal). Again, this is logically redundant, but does not influence satisfiability.
   - If $m_i > 3$, then we define $\psi_i = (\ell_{i,1} \vee \ell_{i,2} \vee p_1) \wedge (\neg p_1 \vee \ell_{i,3} \vee p_2) \cdots (\neg p_{m_i-3} \vee \ell_{i,m_i-1} \vee \ell_{i,m_i})$ where the p_j are new variables not occurring in φ. This formula is satifiable if and only if $\bigvee_{j=1}^{m_i} \ell_{i,j}$ is satisfiable (make sure you understand why this is true).

3. Yes, we have shown both to be NP-hard and they are both in NP: For example, the following polynomial-time halting NTM accepts CNF-SAT (cp. Lecture 6, Slide 10).

   On input w:

   (a) Check whether w encodes a formula φ of propositional logic in CNF. If not, reject.
   (b) Determine the set of propositions in φ.
   (c) For each proposition guess a truth value.
   (d) Evaluate the formula w.r.t. these truth values.
   (e) If it evaluates to 1 accept. Otherwise reject.

   The polynomial-time halting NTM for 3SAT is analogous.

Exercise 4: Spot the error

Every week someone claims to haven proven that P = NP or that P $\neq$ NP. An incomplete list of "proofs" can be found here: `https://www.win.tue.nl/~wscor/woeginger/P-versus-NP.htm`

Last week a very famous professor published the following proof that P $\neq$ NP:

Proof: Consider the following halting DTM for CLIQUE:

On input $(\langle G \rangle, k)$:

1. Generate all possible subsets of vertices from G.
2. If one of these subsets is a k-clique, then accept.
3. Otherwise reject."

Because there are 2^n different subsets of nodes to examine, the algorithm clearly does not run in polynomial time. Therefore we have proved that CLIQUE has exponential time complexity and this means CLIQUE $\notin$ P. Because we know that CLIQUE $\in$ NP, we conclude that P $\neq$ NP.

Describe the error in the above proof.

Solution:
It is true that the suggested *algorithm* for CLIQUE does not run in polynomial time, but from this fact we cannot conclude that the *language* CLIQUE does not belong to the class P. There can still be other (faster) algorithms for CLIQUE that run in polynomial time; we simply did not exclude this possibility by presenting one particular algorithm with an exponential running time. To conclude that CLIQUE $\notin$ P we would have to show *no* halting DTM for CLIQUE has a polynomial time complexity.

Exercise 5: Challenge

Consider the decision problem

Given an undirected graph G, is it the case that G has a clique of size 4?

1. Express this problem as the language 4CLIQUE.
2. Prove 4CLIQUE $\in$ NP.
3. Prove 4CLIQUE $\in$ P.
4. Why is CLIQUE not known to be in P if 4CLIQUE is? Justify your answer.

Solution:

1. The language is
$$\text{4CLIQUE} = \{\ulcorner G \urcorner \mid G \text{ is an undirected graph with a clique of size } 4\}$$
2. We can show that 4CLIQUE $\in$ NP in two different ways:

   By constructing a polynomial time verifier: The certificate is here a clique of size 4. The verifier is

   On input $\ulcorner G \urcorner, C$:
   (a) If C is not a set of 4 nodes from G then reject
   (b) Else check that every pair of nodes is connected by an edge.
   (c) If this is the case, then accept, else reject.

   This verifier has polynomial time complexity and the certificate C has polynomial length in the encoding of G.

   By constructing a polynomial time halting NTM: A halting NTM for 4CLIQUE is

   On input $\ulcorner G \urcorner$:
   (a) Guess C, a set of 4 nodes from G.
   (b) Check that each pair of nodes from C is connected by an edge.
   (c) If this is the case, then accept else reject

   The running time of this halting NTM is polynomial.
3. In a graph with n nodes there are
$$\left(\begin{array}{c} n \\ 4 \end{array} \right) = \frac{n!}{(n-4)!4!} \leq n(n-1)(n-2)(n-3) = \mathcal{O}(n^4)$$
sets of nodes with exactly 4 elements. We can therefore create the following polynomial-time algorithm for 4CLIQUE:

   On input $\langle G \rangle$:
   1. For every set C of 4 nodes from G:
      - Check that every pair of nodes from C is connected by an edge.
      - If this is the case, then accept.
   2. If no set of 4 nodes is a clique, then reject.

   Given a graph with n vertices, the algorithm will perform at most $O(n^4)$ traversals of its main loop. Every traversal will require at most n^2 steps, since each edge must be examined at most once. The algorithm therefore has polynomial time complexity.
4. We get no information about CLIQUE from our knowledge that 4CLIQUE $\in$ P, since CLIQUE is a different problem, i.e., CLIQUE has two parameters, while 4CLIQUE only has one. Furthermore, for 4CLIQUE there is a polynomial number of potential certificates to check (for a fixed polynomial) while for CLIQUE there is no such fixed polynomial that bounds the number of potential certificates.

Tutorial 8

Exercise 1: Backtracking for knapsack

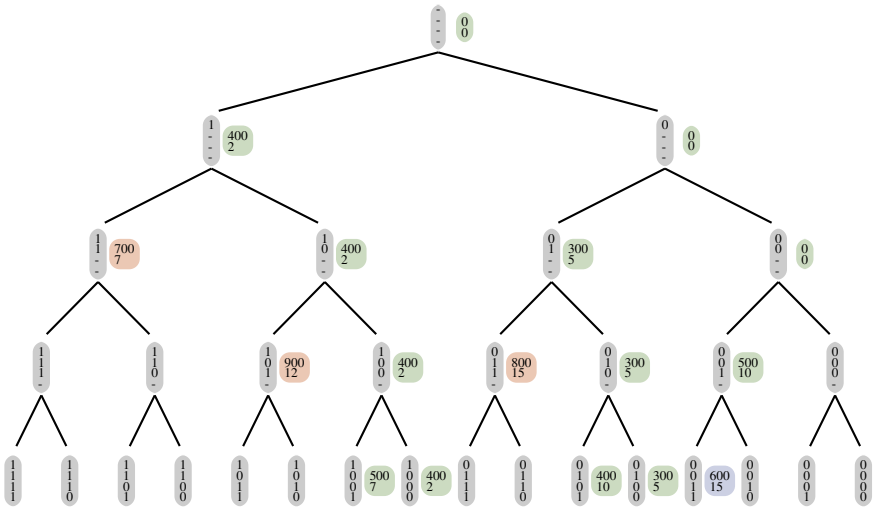
1. Apply the backtracking algorithm for Knapsack to the following instance: $W = 600$, $T = 14$, and

j	1	2	3	4
w_j	400	300	500	100
c_j	2	5	10	5

2. How can the algorithm as presented during the lecture be improved?

Solution:

1.



2. Here are four straightforward improvements:

   - Items whose weight exceeds the capacity of the knapsack (e.g., item 3 above) can be removed from the list of items, as they will never be part of a valid solution.
   - At the last level, the right successor does not have to be explored, as it does not yield a new solution.
   - The algorithm as presented during the lecture considers the items in the order they are given in the arrays. So, sorting the items by their average value, i.e., the ratio $\frac{c_j}{w_j}$, ensures that we consider the more promising items first.
   - Whenever the value of the remaining items is smaller than $T -$ `cur_v`, then the remaining subtree does have need to be explored, as not valid solution can be found there.

Exercise 2: The subset sum problem

In the *subset sum problem*, we are given a target $t \in \mathbb{N}$ and a sequence $n_1, n_2, \ldots, n_k$ of natural numbers, and the question is whether a subset of the n_j sums up to exactly t. For example, given $1, 2, 3, 4, 5$ and $t = 10$ the answer is yes ($1 + 4 + 5 = 10$), given $1, 3, 3, 5, 7$ and $t = 2$ the answer is no (note that the 1 in the sequence can only be used once in the sum).

1. Formulate the problem as a decision problem (Hint: recall the definition of Knapsack).
2. Give a backtracking algorithm for the subset sum problem.
3. Apply your algorithm to the instance with $t = 307$ and the sequence $160, 56, 158, 123, 67, 26$.

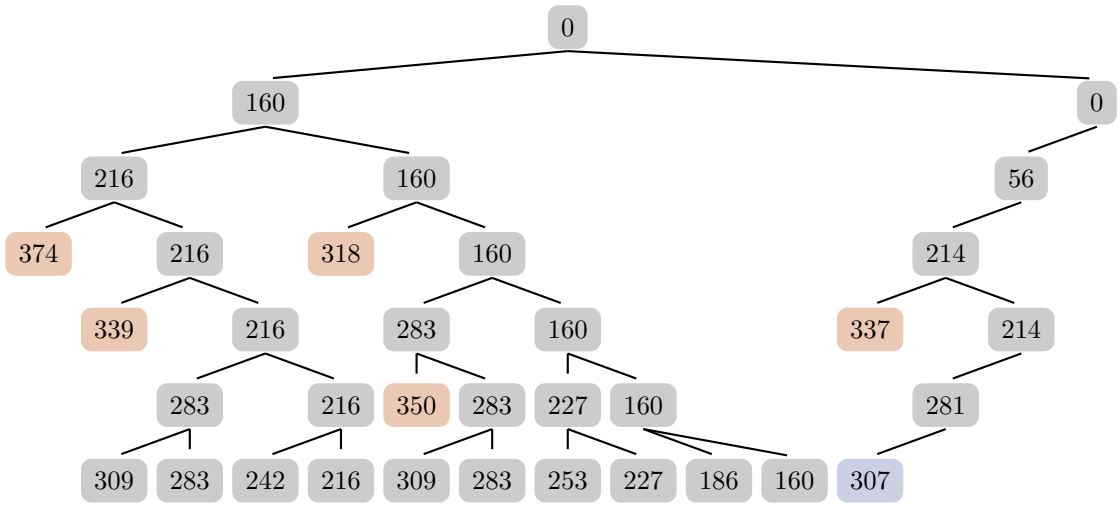
Solution:

1. SUBSETSUM $= \{(t, n_1, \ldots, n_k) \mid \exists b_1, \ldots, b_k \text{ s.t. } \sum_{j=1}^{k} b_j \cdot n_j = t\}$.
2.

```
# n is an integer array
def subsetsum(t, n):
  return rec_subsetsum(t, n, 0, 0)

def rec_subsetsum(t, n, cur_sum, j):
  # valid solution found
  if cur_sum = t:
    return true
  # target exceeded
  if cur_sum > t:
    return false
  # all items checked
  if j ≥ length(n):
    return false
  # add j-th number and recursively continue
  if rec_subsetsum(t, n, cur_sum+n[j], j+1):
    return true
  # do not add j-th number and recursively continue
  if rec_subsetsum(t, n, cur_sum, j+1):
    return true
  return false
```

3. We depict the current sum in each node of the search tree, the corresponding partial solutions can be reconstructed from the branching of the tree: a left successor on the j-th level means n_j is added to the sum ($b_j = 1$), a right successor on the j-th level means n_j is not added to the sum ($b_j = 0$).



Exercise 3: Sudoku

Describe how to encode a standard 9×9 Sudoku by an instance of SAT, i.e., given an incomplete Sudoku generate a formula so that a satisfying assignment to the formula encodes a valid completion of the Sudoku.

First, think about which variables to use, then about how to express the required constraints.

Solution:

We use variables of the form s_{xyz} for $1 \leq x, y, z \leq 9$ with the meaning that s_{xyz} is true if and only the entry in row x and column y is the number z.

Now, we express the following constraints:

- There is at least one entry in each cell:

$$\bigwedge_{x=1}^{9} \bigwedge_{y=1}^{9} \bigvee_{z=1}^{9} s_{xyz}$$

- Each number appears at most once in every row:

$$\bigwedge_{y=1}^{9} \bigwedge_{z=1}^{9} \bigwedge_{x=1}^{8} \bigwedge_{i=x+1}^{9} (\neg s_{xyz} \vee \neg s_{iyz})$$

  Note that we restrict ourselves here to $x < i$ to eliminate some redundancy.

- Each number appears at most once in every column:

$$\bigwedge_{x=1}^{9} \bigwedge_{z=1}^{9} \bigwedge_{y=1}^{8} \bigwedge_{i=y+1}^{9} (\neg s_{xyz} \vee \neg s_{xiz})$$

  Note that we restrict ourselves here again to $y < i$ to eliminate some redundancy.

- Every number appears at most once in every 3×3-subgrid:

$$\bigwedge_{z=1}^{9} \bigwedge_{i=0}^{2} \bigwedge_{j=0}^{2} \bigwedge_{x=1}^{3} \bigwedge_{y=1}^{3} \bigwedge_{k=y+1}^{3} (\neg s_{(3i+x)(3j+y)z} \vee \neg s_{(3i+x)(3j+k)z})$$

  and

$$\bigwedge_{z=1}^{9} \bigwedge_{i=0}^{2} \bigwedge_{j=0}^{2} \bigwedge_{x=1}^{3} \bigwedge_{y=1}^{3} \bigwedge_{k=x+1}^{3} \bigwedge_{\ell=1}^{3} (\neg s_{(3i+x)(3j+y)z} \vee \neg s_{(3i+k)(3j+\ell)z})$$

Here, the first subformula is concerned with cells $y < k$ in the same column of a subgrid while the second one is concerned with cells (x, y) and (k, ℓ) with $x < k$, i.e., all columns to the right. This is again done to avoid redundancy.

- For every clue "in row x_j and column y_j there is number z_j" there is a unit clause expressing exactly that fact:

$$s_{x_j y_j z_j}$$

A satisfying assignment of the conjunction of these formulas yields a valid sudoku solution and vice versa. This is because if there is at least one number in each cell and no number appears twice in a row, then there is exactly one number in each cell (which is required by the rules, but not explicitly stated in our formula).

However, we can add more constraints that are logically redundant (as they are implied by the above ones), but speed up the solution process.[1]

- There is at most one entry in each cell:

$$\bigwedge_{x=1}^{9} \bigwedge_{y=1}^{9} \bigwedge_{z=1}^{8} \bigwedge_{i=z+1}^{9} (\neg s_{xyz} \vee \neg s_{xyi})$$

- Each number appears at least once in each row:

$$\bigwedge_{y=1}^{9} \bigwedge_{z=1}^{9} \bigvee_{x=1}^{9} s_{xyz}$$

- Each number appears at least once in each column:

$$\bigwedge_{x=1}^{9} \bigwedge_{z=1}^{9} \bigvee_{y=1}^{9} s_{xyz}$$

- Each number appears at least once in each 3×3-subgrid:

$$\bigwedge_{z=1}^{9} \bigwedge_{i=0}^{2} \bigwedge_{j=0}^{2} \bigvee_{x=1}^{3} \bigvee_{y=1}^{3} s_{(3i+x)(3j+y)z}$$

Exercise 4: DPLL

Apply the DPLL algorithm to the following formulas to determine whether they are satisfiable or unsatisfiable.

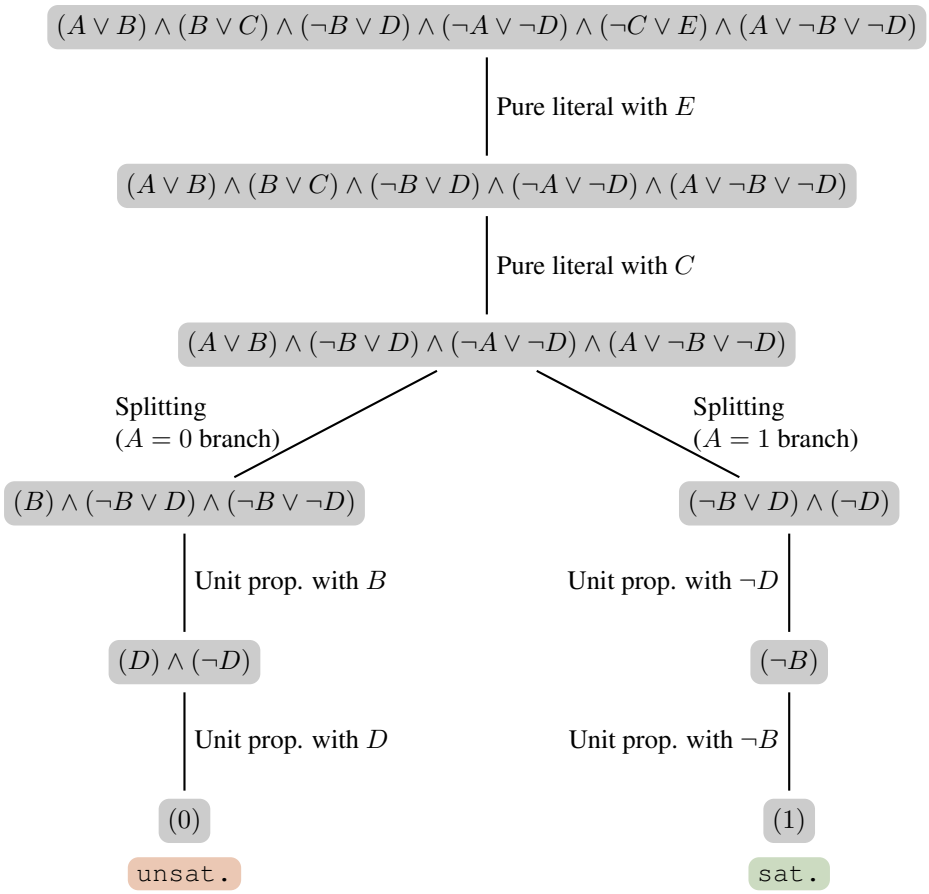
Illustrate the execution of the algorithm is we did in the lecture using a tree structure, i.e., showing the simplified formula for each recursive call and the rule applied. Assume that DPLL selects variables in alphabetical order (i.e., $A, B, C, D, E, \ldots$) if a rule is applicable to more than one variable (e.g., if there is more than one unit clause).

1. $\varphi_1 = (A \vee B) \wedge (B \vee C) \wedge (\neg B \vee D) \wedge (\neg A \vee \neg D) \wedge (\neg C \vee E) \wedge (A \vee \neg B \vee \neg D)$
2. $\varphi_2 = (\neg A \vee \neg B \vee \neg C) \wedge (A \vee \neg B) \wedge (A \vee \neg D) \wedge (B \vee \neg E) \wedge (\neg C \vee D) \wedge (C \vee E)$ $\wedge (C \vee \neg E) \wedge (\neg D \vee E)$

Solution:

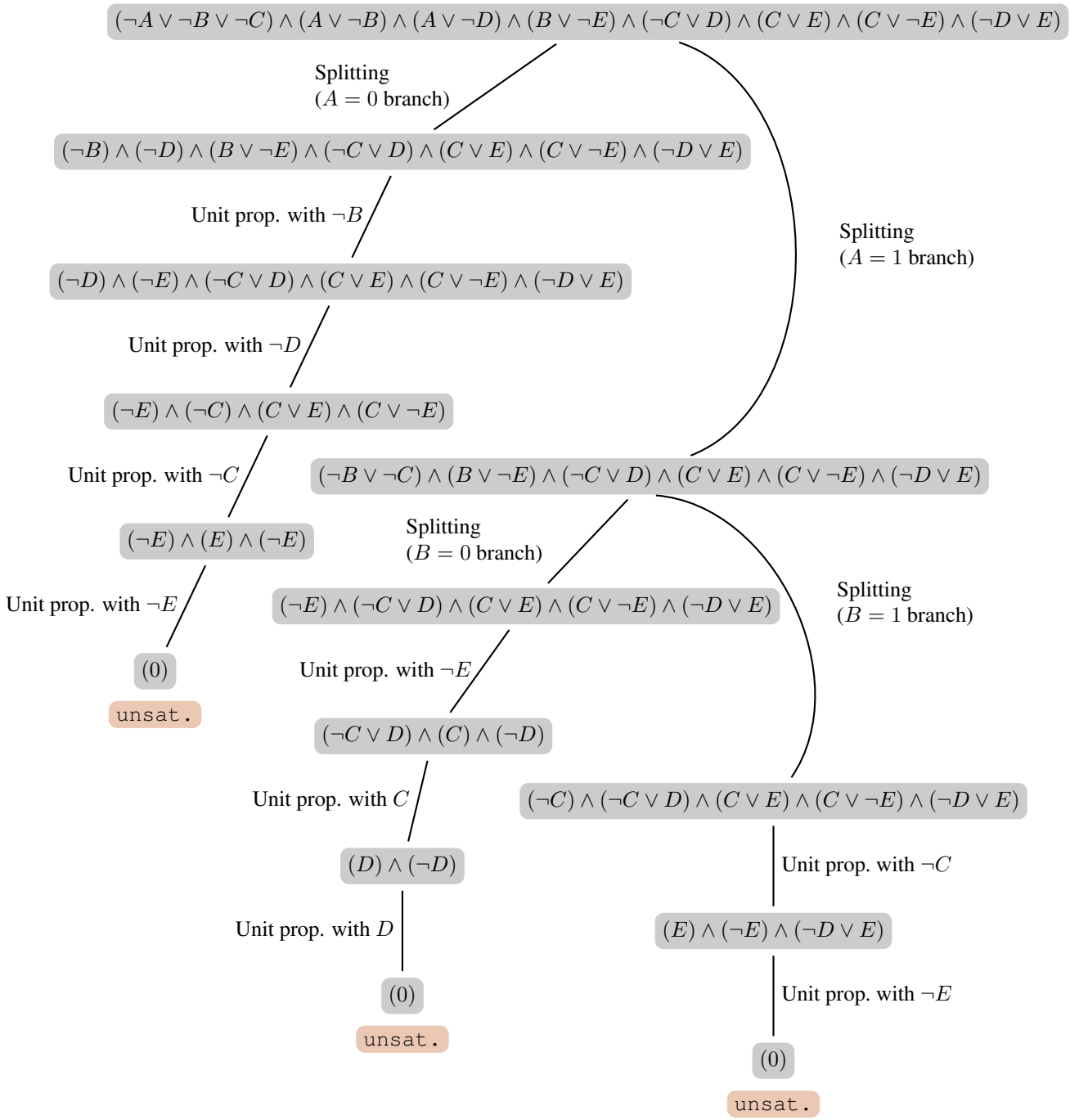
[1] See "Inês Lynce and Joël Ouaknine. Sudoku as a SAT Problem." at `https://people.mpi-sws.org/ joel/publications/sudoku05.pdf` for details.

1.



So, φ_1 is satisfiable and we can read of a satisfying assignment by collecting the truth values along the path from the satisfiable leaf to the root, e.g., $B = 0$, $D = 0$, $A = 1$, $C = 1$, and $E = 1$.

2.


$(\neg A \vee \neg B \vee \neg C) \wedge (A \vee \neg B) \wedge (A \vee \neg D) \wedge (B \vee \neg E) \wedge (\neg C \vee D) \wedge (C \vee E) \wedge (C \vee \neg E) \wedge (\neg D \vee E)$
Splitting ($A = 0$ branch)
$(\neg B) \wedge (\neg D) \wedge (B \vee \neg E) \wedge (\neg C \vee D) \wedge (C \vee E) \wedge (C \vee \neg E) \wedge (\neg D \vee E)$
Unit prop. with $\neg B$
$(\neg D) \wedge (\neg E) \wedge (\neg C \vee D) \wedge (C \vee E) \wedge (C \vee \neg E) \wedge (\neg D \vee E)$
Unit prop. with $\neg D$
$(\neg E) \wedge (\neg C) \wedge (C \vee E) \wedge (C \vee \neg E)$
Unit prop. with $\neg C$
$(\neg E) \wedge (E) \wedge (\neg E)$
Unit prop. with $\neg E$
(0)
unsat.
Splitting ($A = 1$ branch)
$(\neg B \vee \neg C) \wedge (B \vee \neg E) \wedge (\neg C \vee D) \wedge (C \vee E) \wedge (C \vee \neg E) \wedge (\neg D \vee E)$
Splitting ($B = 0$ branch)
$(\neg E) \wedge (\neg C \vee D) \wedge (C \vee E) \wedge (C \vee \neg E) \wedge (\neg D \vee E)$
Unit prop. with $\neg E$
$(\neg C \vee D) \wedge (C) \wedge (\neg D)$
Unit prop. with C
$(D) \wedge (\neg D)$
Unit prop. with D
(0)
unsat.
Splitting ($B = 1$ branch)
$(\neg C) \wedge (\neg C \vee D) \wedge (C \vee E) \wedge (C \vee \neg E) \wedge (\neg D \vee E)$
Unit prop. with $\neg C$
$(E) \wedge (\neg E) \wedge (\neg D \vee E)$
Unit prop. with $\neg E$
(0)
unsat.

Tutorial 9

Exercise 1: Modelling

1. Model knapsack as an optimization problem according to the definition on Slide 6 of Lecture 9.
2. Model vertex cover as an optimization problem according to the definition on Slide 6 of Lecture 9.

Solution:

1. Recall that the decision variant of Knapsack is defined as

$$\{W, T, w_1, \ldots, w_n, c_1, \ldots, c_n \in \mathbb{N} \mid \exists b_1, \ldots, b_n \in \{0,1\} \text{ s.t. } \sum_{j=1}^{n} b_j \cdot w_j \leq W \text{ and } \sum_{j=1}^{n} b_j \cdot c_j \geq T\}$$

   In the optimization variant, we want to maximize (why?) the sum of the values of the items we put in the knapsack, while still satisfying the weight constraint.

   Hence, we define:

   - The set of instances as

$$I = \{(W, w_1, \ldots, w_n, c_1, \ldots, c_n) \mid W \in \mathbb{N}, w_j \in \mathbb{N} \text{ for all } j, c_j \in \mathbb{N} \text{ for all } j, \text{ and } n \in \mathbb{N}\}.$$

     Note that this definition is equivalent to $I = \bigcup_{n \in \mathbb{N}} \mathbb{N}^{2n+1}$, i.e., the set of instances is the set of tuples of natural numbers of odd length.
   - Given an instance $(W, w_1, \ldots, w_n, c_1, \ldots, c_n) \in I$, a possible solution represents a choice of items to put in the knapsack. In the definition of the decision variant, these are represented by bit vectors $(b_1, \ldots, b_n)$ of length n.
     But, not every such vector represents a *feasible* solution, as it might violate the weight constraint. Hence, the set $f(i)$ of feasible solutions for i is

$$\{(b_1, \ldots, b_n) \in \mathbb{B}^n \mid \sum_{j=1}^{n} b_j \cdot w_j \leq W\}.$$

   - Let $i = (W, w_1, \ldots, w_n, c_1, \ldots, c_n)$ be an instance. We measure the *value* of a feasible solution $s = (b_1, \ldots, b_n) \in f(i)$ by the value of the items chosen by the solution s, i.e.,

$$m(i, s) = \sum_{j=1}^{n} b_j \cdot c_j.$$

   - Finally, we want to maximize the value of the solution, i.e., we pick $g = \max$.

2. Recall that the decision variant of vertex cover is defined as

$$\{(G, k) \mid G \text{ is an undirected graph with } k\text{-vertex cover}\}.$$

   In the optimization variant, we are interested in finding the size of a minimal vertex cover.

   Hence, we define:

   - The set I of instances is the set of undirected graphs.
   - The set of feasible solutions for an instance $G = (V, E)$ is

$$\{C \subseteq V \mid C \text{ is a } k\text{-vertex cover in } G \text{ for some } k\}.$$

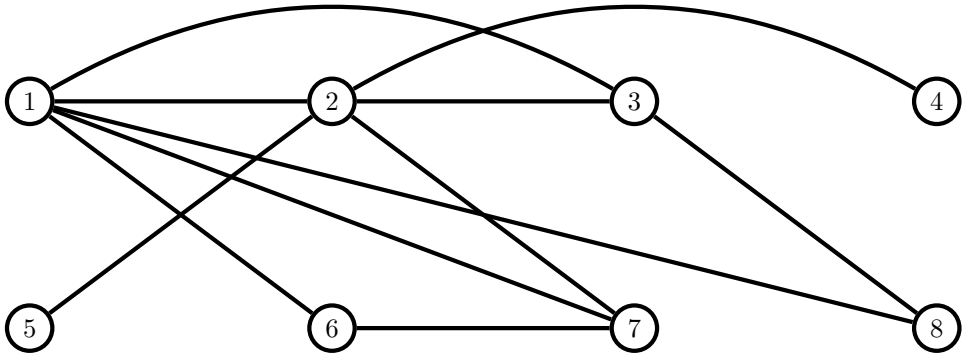
   - We measure the value of a vertex-cover by its size, i.e.,

$$m(G, C) = |C|.$$

   - We want to minimize, so $g = \min$.

Exercise 2: Vertex cover

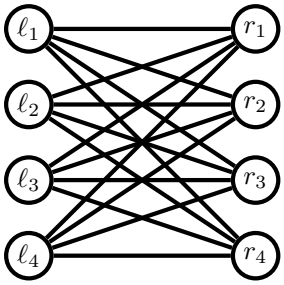
1. Apply the vertex cover approximation algorithm to the following graph.



2. The complete bipartite graph $K_{m,m}$ is defined as $K_{m,m} = (V_m, E_m)$ with

   - $V_m = \{\ell_j, r_j \mid 1 \leq j \leq m\}$ and
   - $E_m = \{(\ell_j, r_{j'}), (r_j, \ell_{j'}) \mid 1 \leq j, j' \leq m\}$.

   Intuitively, $K_{m,m}$ consists of two disjoint sets of m vertices, and all edges between these sets, but no edges in the two sets. $K_{4,4}$ is depicted below.



   (a) What is the size a smallest vertex cover in $K_{m,m}$?

   (b) What is the size of a vertex cover computed by the approximation algorithm presented during the lecture?

   (c) What does this tell us about the approximation ratio of the algorithm?

Solution:

1. We use the same heuristic as on the slides to select an edge to cover, i.e., we prioritize vertices with many edges.

   (a) We cover the edge $(1, 7)$ by adding both 1 and 7 to C. This covers the edges $(1, 2), (1, 3), (1, 6), (1, 7), (1, 8), (6, 7), (2, 7)$.
   So, after the first iteration, we have $C = \{1, 7\}$ and $E' = \{(2, 3), (2, 4), (2, 5), (3, 8)\}$.

   (b) We cover the edge $(2, 3)$ by adding both 2 and 3 to C. This covers the edges $(2, 3), (2, 4), (2, 5), (3, 8)$.
   So, after the second iteration, we have $C = \{1, 2, 3, 7\}$ and E' is empty, i.e., the algorithm terminates.

2. (a) The size of a smallest vertex cover is m, e.g., the one containing all m vertices ℓ_j on the left side: every edge is incident to such a vertex, so this is indeed a vertex cover.

   Now, consider an arbitrary set C with strictly less than m vertices. We show that C is not a vertex cover, showing that m is indeed the minimum.

   As $|C| < m$, there is a j such that ℓ_j and r_j are not in C. Hence, the edge $(\ell_j, r_j) \in E_m$ is not covered by C, i.e., it is indeed not a vertex cover.

   (b) We show that the algorithm selects all $2m$ vertices.

   We begin by noting that after every iteration of the algorithm, the value of the variable C satisfies

   $$|C \cap \{\ell_j \mid 1 \leq j \leq m\}| = |C \cap \{r_j \mid 1 \leq j \leq m\}|,$$

   i.e., at every time it has picked as many vertices on the left side as it has on the right side. This is true, as every edge of the graph has one vertex in $\{\ell_j \mid 1 \leq j \leq m\}$ and on vertex in $\{r_j \mid 1 \leq j \leq m\}$ and, once a vertex is added to C, no edge incident to it is selected later.

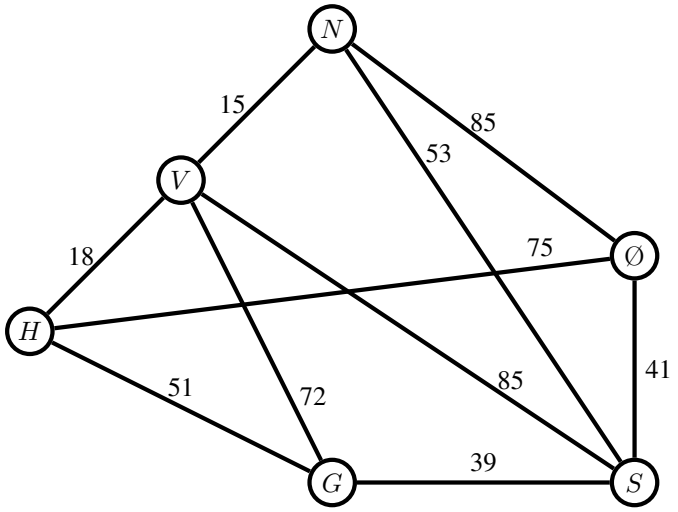
   Now, consider the output C of the algorithm, and, towards a contradiction, assume that $C \neq V_m$, i.e., there is a vertex not in C. Then, by the property above, we conclude that at least two vertices are not in C, one of the form ℓ_j for some $1 \leq j \leq m$ and one of the form $r_{j'}$ for some $1 \leq j' \leq m$.

   But then the edge $(\ell_j, r_{j'})$ is not covered by C, i.e., C is not a vertex cover. This is a contradiction, as the algorithm outputs a vertex cover.

   (c) It is at least 2, i.e., the claimed upper bound of 2 presented in the lecture is tight.

Exercise 3: Minimum spanning trees

Compute an MST for the following graph using Prim's algorithm.



Solution:

1. Initially, all vertices are in Q and have value ∞, and all successors are undefined.
2. All vertices have value ∞, so we start at vertex H by removing it from Q. This updates the values/-successors of the following vertices:

   V: value 18, successor H.

   G: value 51, successor H.

   Ø: value 75, successor H.

3. The vertex with the lowest value is V, so we remove it from Q. This updates the values/successors of the following vertices:

   N: value 15, successor V.

   S: value 85, successor V.

   The following values/successors are unchanged:

   G: value 51, successor H.

   $\emptyset$: value 75, successor H.

4. The vertex with the lowest value is N, so we remove it from Q. This updates the values/successors of the following vertices:

   S: value 53, successor N.

   The following values/successors are unchanged:

   G: value 51, successor H.

   $\emptyset$: value 75, successor H.

5. The vertex with the lowest value is G, so we remove it from Q. This updates the values/successors of the following vertices:

   S: value 39, successor G.

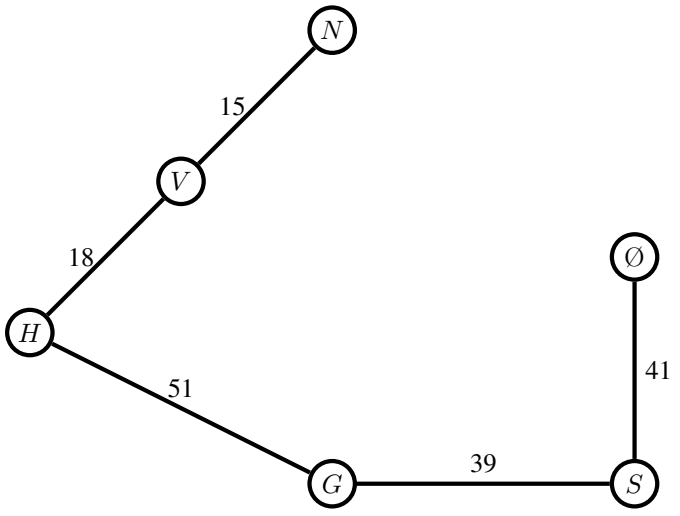
   The following values/successors are unchanged:

   $\emptyset$: value 75, successor H.

6. The vertex with the lowest value is S, so we remove it from Q. This updates the values/successors of the following vertices:

   $\emptyset$: value 41, successor S.

7. The vertex with the lowest value is $\emptyset$, so we remove it from Q. This was the last vertex, so the algorithm terminates and yields the following minimal spanning tree:

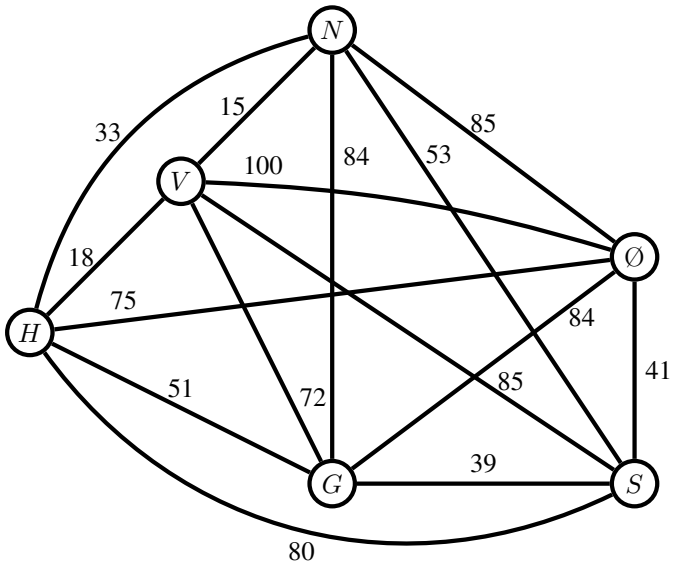


Exercise 4: Travelling Salesperson

Apply the approximation algorithm for TSP to the graph from Exercise 3.

Solution:

We begin by completing the graph (using shortest paths between the missing vertices) yielding the following graph:



The MST computed in Exercise 3 (with weight 164) is still an MST for the completed graph, so we use it.

We start the DFS at vertex H, yielding the sequence

$$HGS\emptyset SGHVNVH.$$

Removing duplicates yields the cycle

$$HGS\emptyset VH$$

with weight 269.

Exercise 5: Decision vs. optimization problems

In the lecture, we have claimed that "Decision and optimization are interreducible with polynomial overhead".

1. Formalize the statement for the decision and the optimization variant of the clique problem.
2. Prove your statement.
3. Can you think of a situation where the interreducibility fails?

Solution:

1. CLIQUE $\in$ P if and only if the optimization variant of CLIQUE can be solved in polynomial time.

2. We show both directions:

   $\Rightarrow$: Assume CLIQUE $\in P$, i.e., there is a halting DTM M with time-complexity $p(n)$ for CLIQUE, where $p(n)$ is some polynomial.

   The following Turing machine solves the optimization variant in polynomial time.

   On input G:

   1. compute the number n of vertices of G
   2. $k = 1$
   3. Run M on (G, k).
   4. If M accepts and $k \leq n$, increment k and goto step 3.
   5. If M accepts and $k = n$, return n.
   6. If M rejects, return $k - 1$.

   In the worst, case, we run M n times, where n is the number of vertices of G. Therefore, the algorithm is polynomial in the size of G. Furthermore, it returns the largest k such that G contains a k-clique: if M rejects (G, k) in line 6, then it has accepted for $k - 1$ and will accept for no $k' > k$.

   $\Leftarrow$: Assume the optimization variant of CLIQUE can be solved in polynomial time, i.e., there is a Turing machine M with polynomial time complexity that, given a graph G returns the maximal k such that G has a k-clique.

   Then, the following halting DTM accepts CLIQUE.

   Given input (G, k):

   1 Run M on G.

   2 If the output of M is strictly smaller than k reject, otherwise accept.

   This halting DTM runs in polynomial-time, as it calls M only once, and is obviously correct.

3. The $\Rightarrow$-direction depends on the fact that the maximal size of cliques we need to check for is bounded by the size of the graph. In problems where such a bound is too large, this approach does not work.

Tutorial 10

Exercise 1: Matchings

Let (V, E) be an undirected graph. A matching is a subset $M \subseteq E$ of edges such that for all vertices $v \in V$, at most one edge in M is incident to v (i.e., (V, M) does not contain a path containing more than two vertices). A maximum matching is a matching M such that we have $|M| \geq |M'|$ for every matching M'.

Furthermore, we say that (V, E) is bipartite if $V = L \cup R$ such that

- L and R are disjoint ($L \cap R = \emptyset$) and
- all edges in E go between L and R ($(u, v) \in E$ implies $\{u, v\} \cap L \neq \emptyset$ and $\{u, v\} \cap R \neq \emptyset$).

1. Describe a task that can be solved by modelling it as an maximum matching problem in a bipartite undirected graph. Bonus points if it involves pizza.
2. Show how to reduce the maximum matching problem in bipartite undirected graphs to the maximum flow problem.
3. Show how to express the maximum matching problem in bipartite undirected graphs as a linear program.

Solution:

1. You are in charge of matching donated kidneys to patients. Every kidney is modelled by a vertex in L, every patient needing a kidney is modelled by a vertex in R, and an edge (u, v) between a kidney $u \in L$ and a patient $v \in R$ signifies that the kidney u is compatible with patient v (the blood group matches, etc). The graph is obviously bipartite as edges only connect kidneys and patients, but not kidneys with kidneys or patients with patients.

   A matching in this graph corresponds to kidney-patient pairs so that every kidney is assigned to at most one patient and every patient receives at most one kidney (both obviously desirable properties). Also obviously, we try to maximize the size of the matching, i,e., try to assign as many kidneys as possible.

   After you have modelled and solved the problem (see below how you did that), you order a pizza for dinner.

2. Given a bipartite undirected graph (V, E) with $V = L \cup R$ we construct a flow network such that from a flow we can construct a matching and vice versa.

   Intuitively, we add two new states s and t, connect s to all vertices $\ell \in L$ with an edge (s, ℓ), direct all edges in E to go from L to R and connect all vertices $r \in R$ to t with an edge (r, t). Furthermore, each edge gets capacity 1.

   Then, the inflow of every vertex $\ell \in L$ is at most 1 and the outflow of every vertex $r \in R$ is at most 1.

   Now we apply the integrality theorem: if all edges have an integer capacity (like the one we construct), then the network has a maximal flow f such that $f(u, v)$ is an integer for every edge (u, v). As we only use the capacity 1, this means we can identify a flow with a matching and vice versa.

   Let us describe the reduction in more detail: We begin by defining the flow network $(V \cup \{s, t\}, E', c, s, t)$ where s, t are fresh vertices not in V,

   $$E' = \{(s, \ell) \mid \ell \in L\} \cup \{(\ell, r) \in E \mid \ell \in L \text{ and } r \in R\} \cup \{(r, t) \mid r \in R\},$$

   and $c(u, v) = 1$ for all $(u, v) \in E'$.

   Now, we show that (V, E) has a matching of size k if and only if $(V \cup \{s, t\}, E', c, s, t)$ has a flow of value k. Then, the size of a maximal matching corresponds to the value of a maximal flow.

First, let M be a matching of size k. We define the flow f_M as follows: For each edge $(\ell, r) \in M$, define $f(s, \ell) = f(\ell, r) = f(r, t) = 1$. All other edges have flow 0 under f. As the edges in the matching do not share vertices, this mapping satisfies indeed flow conservation. Finally, the capacity constraints are trivially satisfied, i.e., f is indeed a flow. Furthermore, its value is k.

Now, let f be a maximal flow in $(V \cup \{s, t\}, E', c, s, t)$. Due to the integrality theorem (cf. Theorem 26.10 in CLRS) and the capacity constraints we can assume $f(u, v) \in \{0, 1\}$ for each edge $(u, v) \in E'$.

We define $M = \{(\ell, r) \in E \mid f(\ell, r) = 1\}$ and claim it is a matching whose size is equal to the value of f. The latter claim follows from flow conservation: at most one unit of flow can reach a vertex of $\ell \in L$. Assume it is one. Then, due to the integrality theorem and conservation of flow, that unit must flow to a unique vertex $r \in R$, i.e., the edge has flow $f(\ell, r) = 1$. Hence, there is at most one edge in M that is incident to ℓ. For vertices in r, the outflow is at most one. Hence, the inflow is also at most one, which means at most one unit of flow can reach r, which means that there is at most one edge in M that is incident to r.

Hence, we have shown the claim and the proof is completed.

3. Given a bipartite undirected graph (V, E) with $V = L \cup R$ we construct an LP capturing the matching problem. We use one decision variable $x_{(u,v)}$ for every edge $(u, v) \in E$ expressing whether (u, v) is in the matching ($x_{(u,v)} = 1$) or not ($x_{(u,v)} = 0$). Note that the decision variables range over the reals, but our constraints are such that in every optimal solution the variables can only have the values 0 or 1.

   As we want to maximize the size of the matching, we use the objective function

$$\sum_{(u,v) \in E} x_{(u,v)}.$$

   Now, we need to express the requirements on a matching, i.e., that for each vertex at most one edge in the matching is incident to that vertex:

$$\sum_{v \in V} x_{(u,v)} \leq 1 \text{ for all } u \in V.$$

   Finally, we express that all variables are indeed in $[0, 1]$:

$$x_{(u,v)} \geq 0 \text{ for all } (u, v) \in E$$

   and

$$x_{(u,v)} \leq 1 \text{ for all } (u, v) \in E.$$

Exercise 2: Disjoint paths

Let (V, E) be a directed graph with two dedicated vertices $s \neq t \in V$. A path from s to t is a sequence $v_0 \cdots v_n$ of vertices such that $v_0 = s$, $(v_i, v_{i+1}) \in E$ for all $0 \leq i < n$, and $v_n = t$. We say that the path contains the vertices $v_0, v_1, \ldots, v_n$ and that it contains the edges $(v_0, v_1), (v_1, v_2), \ldots, (v_{n-1}, v_n)$.

- We say that two paths from s to t are vertex disjoint if the only vertices that are contained in both paths are s and t.
- We say that two paths from s to t are edge disjoint if there is no edge that is contained in both paths.

The maximum vertex-disjoint (egde-disjoint) path problem asks, given a directed graph (V, E) and vertices $s \neq t \in V$, to compute the maximal number of vertex disjoint (edge disjoint) paths from s to t.

1. Describe a problem that can be solved by modelling it as an maximum vertex disjoint path problem in a directed graph.

2. Describe a problem that can be solved by modelling it as an maximum edge disjoint path problem in a directed graph.
3. Show how to reduce the maximum vertex disjoint path problem in directed graphs to the maximum flow problem.
4. Show how to express the maximum vertex disjoint path problem in directed graphs as a linear program.
5. Show how to reduce the maximum edge disjoint path problem in directed graphs to the maximum flow problem.
6. Show how to express the maximum edge disjoint path problem in directed graphs as a linear program.

Solution:

1. Model the internet as a graph consisting of vertices (routers) and edges (links between them). You need to have a reliable connection between two routers s and t. By computing the maximal number of vertex disjoint paths gives the maximal number of independent routes from s to t that do not share a router.

   Bonus fact: By Menger's theorem, this number is equal to the number of *routers* you need to remove from the internet to disable all routes between s and t.

2. We again consider the internet example. By computing the maximal number of edge disjoint paths gives the maximal number of independent routes from s to t that do not share a link, but may share routers.

   Bonus fact: By Menger's theorem, this number is equal to the number of *links* you need to remove from the internet to disable all routes between s and t.

3. Let (V, E) and $s, t \in V$ be an instance of the maximum vertex disjoint path problem in directed graphs.

   We define the network (V', E', c, s', t') with

   - $V' = V \times \{i, o\}$,
   - $E' = \{((v, i), (v, o)) \mid v \in V\} \cup \{((u, o), (v, i)) \mid (u, v) \in E\}$,
   - $c(u, v) = 1$ for all $(u, v) \in E'$,
   - $s' = (s, o)$, and
   - $t' = (t, i)$.

   Draw the resulting network for a small graph to get an intuition for the construction before you continue reading.

   Intuitively, we split each vertex in two and add an edge with capacity one in between. This ensures that at most one unit of flow goes through each vertex. This ensures that each flow through the network corresponds to a union of vertex-disjoint paths from (s, o) to (t, i) and vice versa.

   Thus, the value of a maximal flow is equal to the maximal number of vertex-disjoint paths from s to t in (V, E). The formal proof of correctness of this construction is very similar to that of Exercise 1 and relies on the integrality theorem.

4. Instead of developing a bespoke LP encoding of the maximum vertex disjoint path problem in directed graphs, we apply reductions: The maximum vertex disjoint path problem in directed graphs can be reduced to the maximum flow problem (as shown above), which in turn can be reduced to LP (as seen in the lecture).

5. Let (V, E) and $s, t \in V$ be an instance of the maximum edge disjoint path problem in directed graphs. We proceed as in Item 3, but do not need to split the vertices.

   Instead we define the network (V, E, c, s, t) with capacity $c(u, v) = 1$ for all $(u, v) \in E$. Then, the value of a maximal flow is equal to the maximal number of vertex-disjoint paths from s to t in (V, E). The formal proof of correctness of this construction is very similar to that of Exercise 1 and relies on the integrality theorem.

6. Instead of developing a bespoke LP encoding of the maximum edge disjoint path problem in directed graphs, we apply reductions: The maximum edge disjoint path problem in directed graphs can be reduced to the maximum flow problem (as shown above), which in turn can be reduced to LP (as seen in the lecture).

Exercise 3: The mimimum-cost flow problem

In the minimum-cost flow problem, each edge (u, v) in a flow network comes with a cost-coefficient $a(u, v)$ and a throughput $d \in \mathbb{R}$. One is interested in determining a flow f of value d (if one exists) that minimizes the cost of the flow, which is obtained by summing up the cost of f for each edge (u, v), defined as $a(u, v) \cdot f(u, v)$.

1. Define the minimum-cost flow problem formally.

2. Show how to express the minimum-cost flow problem as a linear program.

Solution:

1. A flow network with costs $((V, E, c, s, t), a)$ consists of a flow network (V, E, c, s, t) (as defined in the lecture) and a cost function $a \colon E \to \mathbb{R}$.

   The minimum-cost flow problem asks, given a flow network with costs $((V, E, c, s, t), a)$ and a throughput $d \in \mathbb{R}$, to compute a flow f in (V, E, c, s, t) of value d that minimizes the cost

   $$\sum_{(u,v)\in E} a(u, v) \cdot f(u, v).$$

2. We adapt the LP formulation presented in the lecture. In particular, we replace the original objective function (which aimed to maximize the flow) by an objective function aiming to minimize the cost of the flow and add a constraint expressing that the value of the flow is d.

   As before, we use decision variables $x_{(u,v)}$ for each $(u, v) \in E$, representing the flow through edge (u, v).

   As said above, we are interested in minimizing the cost of the flow, so we use the objective function

   $$\sum_{(u,v)\in E} a(u, v) \cdot x_{(u,v)}.$$

   The first four constraints are as in the lecture, expressing the capacity constraint and the flow constraint.

   (a) $x_{(u,v)} \geq 0$ for all $(u, v) \in E$ (lower bound of capacity constraint)

   (b) $x_{(u,v)} \leq c(u, v)$ for all $(u, v) \in E$ (upper bound of capacity constraint)

   (c) $\sum_{(u,v)\in E} f(u, v) - \sum_{(v,u)\in E} f(v, u) \leq 0$ for all $v \in V \setminus \{s, t\}$ (one "half" of flow constraint)

   (d) $\sum_{(v,u)\in E} f(v, u) - \sum_{(u,v)\in E} f(u, v) \leq 0$ for all $v \in V \setminus \{s, t\}$ (other "half" of flow constraint)

The new constraints we add express that the value of the flow is exactly d (compare them to the original objective function):

$$\sum_{(s,v)\in E} x_{(s,v)} - \sum_{(v,s)\in E} x_{(v,s)} \leq d$$

and

$$\sum_{(v,s)\in E} x_{(v,s)} - \sum_{(s,v)\in E} x_{(s,v)} \leq -d$$

are equivalent to $\sum_{(s,v)\in E} x_{(s,v)} - \sum_{(v,s)\in E} x_{(v,s)} = d$, which expresses that the value of the flow is d.

Tutorial 11

Exercise 1: Modelling the knapsack problem

Model the optimization variant of knapsack as a 0-1 ILP.

Solution:
Given an instance $W, w_1, \ldots, w_n, c_1, \ldots, c_n$ we construct a 0-1 ILP as follows.

We have one decision variable b_j for each item signifying whether item j is put into the backpack or not. We want to maximize the value of the items in the backpack, hence we use the objective

$$\max \sum_{j=1}^{n} c_j \cdot b_j.$$

The only constraint is that the selected items fit in the backpack, i.e.,

$$\sum_{j=1}^{n} w_j \cdot b_j \leq W.$$

Finally, we require all variables to be binary:

$$b_j \in \{0, 1\} \text{ for all } 1 \leq j \leq n.$$

Exercise 2: Modelling the subset sum problem

Model the decision variant of subset sum as a 0-1 ILP.

Solution:
Given an instance $t, n_1, \ldots, n_k$ we construct a 0-1 ILP as follows.

We have one decision variable b_j for each n_j signifying whether we put it in the sum or not. Note that subset sum is not an optimization problem, we are only interested in satisfying the constraint that a subset of the n_j sums up to t. Hence, we use the constant objective function

$$f(b_1, \ldots, b_j) = 0.$$

Thus, every solution has the same value.

We express the only constraint, that the selected n_j sum up to t, as

$$\sum_{j=1}^{k} n_j \cdot b_j \leq t$$

and

$$\sum_{j=1}^{k} -n_j \cdot b_j \leq -t$$

which is equivalent to $\sum_{j=1}^{k} n_j \cdot b_j = t$.

Finally, we require all variables to be binary:

$$b_j \in \{0, 1\} \text{ for all } 1 \leq j \leq n.$$

Exercise 3: Modelling 3-coloring

Model the decision variant of 3-coloring (see Lecture 6, Slide 8) as a 0-1 ILP.

Solution:
Given an undirected graph (V, E) we construct a 0-1 ILP as follows.

For every vertex $v \in V$ we have three decision variables, R_v, G_v, and B_v signifying whether the vertex v is colored red, green, or blue.

As we are again concerned with a decision problem, we use a constant objective function mapping every variable assignment to 0 (as in Exercise 2).

Now, we express that every vertex has exactly one color:

$$R_v + G_v + B_v = 1 \text{ for all } v \in V.$$

Next, we express for each edge $(u, v) \in E$, u and v are not both red, not both green, and not both blue:

$$R_u + R_v \leq 1 \text{ for all } (u, v) \in E$$

$$G_u + G_v \leq 1 \text{ for all } (u, v) \in E$$

$$B_u + B_v \leq 1 \text{ for all } (u, v) \in E$$

Finally, we require all variables to be binary:

$$R_v, G_v, B_v \in \{0, 1\} \text{ for all } v \in V.$$

Exercise 4: ILP in NP

Show that 0-1 ILP is in NP.

Solution:
To show that the problem is in NP, we consider the decision variant in standard form with an additional threshold t. As the standard form is a maximization problem, we want to decide whether there is a solution whose objective is at least t.

Now, consider the following halting NTM:

Given input $c_1, \ldots, c_n, a_{1,1}, \ldots, a_{k,n}, b_1, \ldots, b_k, t$:

1. Guess a bit vector of length n, and refer to the i-th bit as x_i.
2. Compute $\sum_{i=1}^{n} c_i \cdot x_i$. If this is strictly smaller than t, reject.
3. For $j = 1, \ldots, k$ compute $\sum_{i=1}^{n} a_{j,i} \cdot x_i$. If this is strictly larger than b_j, reject.
4. Accept.

Note that the coefficients c_j and $a_{i,j}$ as well as the bounds b_j are all rationals, which implies that the computations in step 2 and 3 can be done in polynomial time. Thus, we have shown that 0-1 ILP is in NP.

Tutorial 12

Exercise 1: Decrementing counters

1. Let us add a decrement operation `DEC` to the k-bit counter example from the lecture, which is implemented analogously to the increment operation. In particular, the cost of one decrement is the number of bits inspected and/or updated.

   Show that a sequence of n operations on an initially zero counter can cost at least $n \cdot k$ time.

2. Let us add a reset operation `RESET` to the k-bit counter example from the lecture (i.e., without decrement). Again, the cost of one reset is the number of bits inspected and/or updated.

   Implement a counter as an array of bits so that any sequence of n increments and resets takes $\mathcal{O}(n)$ time on an initially zero counter. Show your implementation of the operations and the analysis.

   Hint: Keep a pointer to the highest-order 1.

Solution:

1. Decrementing a counter holding the value zero (all bits are zero) yields a counter with all bits being one. Dually, incrementing a counter with all bits being one yields a counter with all bits being zero. Hence, when starting at the zero counter, a sequence of n alternating decrements and increments (starting with a decrement) flips every bit during each operations, i.e., $n \cdot k$ bit flips. As each bitflip takes constant time, the sequence takes overall time of at least $(n \cdot k)$.

2. We add a new variable `p` which keeps that of the maximal `i` such that $A[i] = 1$, initialized with zero. Then, we implement increment and reset as follows:

```
def inc2(A, p):
  i = 0
  while(i < length(A) and A[i] == 1):
    A[i] = 0
    i = i + 1
  if i < length(A):
    A[i] = 1
    p = max(p,i)
  else:
    p = -1

def reset(A, p):
  i = 0
  while(i <= p):
    A[i] = 0
  p = -1
```

   As in the lecture/book, we assume that the actual cost of a bitflip is 1. In addition, we assume the cost of updating `p` is 1 as well.

   Setting and resetting of bits by `INC` will work exactly as for the original counter in the lecture (also, read the corresponding section in the book): Setting a bit to 1, the credit is increased by 1, and that credit is used to reset the bit during incrementing. In addition, we increase the credit to later update `p` (if `p` doesn't increase, we can just waste that credit, it won't be needed.)

Since `RESET` manipulates bits at positions only up to `p`, and since each bit up to there must have become the highest-order 1 at some time before the highest-order 1 got up to `p`, every bit seen by `RESET` has some nonzero credit left. So the zeroing of bits of `A` by `RESET` can be completely paid for by the credit stored on the bits.

We just need cost 1 to pay for resetting max. Thus, a charge of 4 for each `INC` and 1 for each `RESET` is sufficient, so that the sequence of n `INC` and `RESET` operations takes $\mathcal{O}(n)$ time.

Exercise 2: Applying amortized analysis

Consider a sequence of n operations on some data structure in which the i-th operation costs i if i is an exact power of two (i.e, $i \in \{1, 2, 4, 8, 16, \ldots\}$) and 1 otherwise.

1. Use aggregate analysis to determine the amortized cost per operation of such a sequence (for an arbitrary n).
2. Use the accounting method to determine the amortized cost per operation of such a sequence (for an arbitrary n).
3. Use the potential method to determine the amortized cost per operation of such a sequence (for an arbitrary n).

Solution:

1. The actual cost c_i of operation i is

$$c_i = \begin{cases} i & \text{if } i \text{ is a power of 2} \\ 1 & \text{otherwise.} \end{cases}$$

Hence, we have

$$\sum_{i=1}^{n} c_i \leq n + \sum_{i=1}^{\log_2(n)} 2^i \leq n + 2n = 3n.$$

Here, we used the facts $\sum_{i=1}^{n} 2^i = 2^{n+1} - 2$ and $2^{\log_2(n)+1} = 2n$.

Hence, the cost of a sequence of n operations is at most $3n$, i.e., the amortized cost of each operation is at most 3.

2. We charge cost 3 for each operation. Then, the credit after n operations is $3n - (\sum_{i=1}^{n} c_i)$, which is always nonnegative as we have seen in Item 1. that $\sum_{i=1}^{n} c_i \leq 3n$.

Hence, since the amortized cost of each operation is constant and the credit is never negative, the total cost of n operations is $\mathcal{O}(n)$.

3. Let D_i denote the data structure before the i-th operation.

Consider $i \geq 1$ and let p_i be the largest power of 2 that is smaller or equal to i and let $\Delta_i = i - p_i$, which is always nonnegative. In particular, Δ_i is zero if and only if i is a power of 2.

We define the function Φ as $\Phi_0 = 0$ and $\Phi(D_i) = 2\Delta_i$. This satisfies the requirements of a potential function, as argued above.

We now compute the amortized cost of each operation by considering two cases:

- If $\Delta_i = 0$, then i is a power of two (say $i = 2^j$), and we have $i - 1 = 2^j - 1$. Hence, $p_i = 2^{j-1}$ and

$$\Delta_{i-1} = (i-1) - 2^{j-1} = 2^j - 1 - 2^{j-1} = 2 \cdot 2^{j-1} - 2^{j-1} - 1 = 2^{j-1} - 1.$$

Hence, the amortized cost of the i-th operation is

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = i - 2(2^{j-1} - 1) = i - 2^j + 2,$$

which is equal to 2, as $i = 2^j$.

- If $\Delta_i \neq 0$, then
$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1 + 2\Delta_i - 2\Delta_{i-1}.$$
Now, as i is not a power of 2, we have $p_i = p_{i-1}$, which implies $\Delta_i = \Delta_{i-1} + 1$. Hence,
$$1 + 2\Delta_i - 2\Delta_{i-1} = 1 + 2(\Delta_{i-1} + 1) - 2\Delta_{i-1} + 1 = 3.$$
Hence, the amortized cost of each operation is at most 3. Thus, the amortized cost of a sequence of n operations is at most $3n$, which bounds the actual cost of the sequence.

Exercise 3: Bounded stacks

You perform a sequence of `PUSH` and `POP` operations on a stack whose size never exceeds k. After every k operations, a copy of the entire stack is made automatically, for backup purposes.

Show that the cost of n stack operations, including the copying of the stack, is $\mathcal{O}(n)$ by assigning suitable amortized costs to the various stack operations.

Solution:
The cost of a copy operations is proportional to the stack heigh of the stack to be copied, which is bounded by k.

We assign amortized cost 3 to the `PUSH` operation, 1 to the `POP` operation, and 0 to the copy operation.

Then, one can show by induction that the credit after n operations is at least the stack height of the current stack plus the number of operations since the last copy. Hence, when a copy is executed there have been k operations since the last copy, which gained enough credit to pay for the copy.

Exercise 4: Potential functions

Suppose you have a potential function Φ such that $\Phi(D_i) \geq \Phi(D_0)$ for all i, but $\Phi(D_0) \neq 0$. Show that there is a potential function Φ' such that $\Phi'(D_0) = 0$, $\Phi'(D_i) \geq 0$ for all i, and the amortized costs using Φ' are the same as the amortized costs using Φ.

Solution:
Let $\Delta = \Phi(D_0)$. We define the function Φ' by $\Phi'(D_i) = \Phi(D_i) - \Delta$ and claim that Φ' has the desired properties, i.e., it is a potential function such that $\Phi'(D_0) = 0$, $\Phi'(D_i) \geq 0$ for all i, and the amortized costs using Φ' are the same as the amortized costs using Φ.

- We have $\Phi'(D_0) = \Phi(D_0) - \Phi(D_0) = 0$ as required.
- We have $\Phi'(D_i) = \Phi(D_i) - \Phi(D_0) \geq 0$, as we have $\Phi(D_i) \geq \Phi(D_0)$ for all i by assumption on Φ.
- Let c_i be an operation. As Φ is a potential function, the amortized cost $\hat{c}_i$ of c_i w.r.t. Φ is
$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}).$$
Now, the amortized cost $\hat{c}'_i$ of c_i w.r.t. Φ' is defined as
$$\hat{c}_i = c_i + \Phi'(D_i) - \Phi'(D_{i-1}).$$
By definition of Φ', we have
$$c_i + \Phi'(D_i) - \Phi'(D_{i-1}) = c_i + (\Phi(D_i) - \Phi(D_0)) - (\Phi(D_{i-1}) - \Phi(D_0)) = c_i + \Phi(D_i) - \Phi(D_{i-1}) = \hat{c}_i,$$
i.e., the amortized costs of c_i w.r.t. Φ and Φ' are equal as required.

Tutorial 13

Exercise 1: Subset sum

Give a dynamic programming algorithm for Subset Sum. To this end, specify the recurrence relation (including initial values).

Solution:
Given an instance $n_1, \ldots, n_k \in \mathbb{N}$ and $t \in \mathbb{N}$, define

$$F(j,t') \begin{cases} 1 & \text{there is a subset of } n_1, \ldots, n_j \text{ that sums up to } t' \\ 0 & \text{otherwise.} \end{cases}$$

for $0 \leq j \leq k$ and $0 \leq t' \leq t$.

This is captured by the recurrence relation

$$F(j,t') = \begin{cases} F(j-1, t'-n_j) \vee F(j-1,t') & \text{if } t' \geq n_j \\ F(j-1,t') & \text{otherwise} \end{cases}$$

with

$$F(j,0) = 1 \text{ for all } j \geq 0$$

and

$$F(0,t) = 0 \text{ for all } t > 0$$

stating that one can sum up to t' with a subset of $\{n_1, \ldots, n_j\}$ by either

- taking n_j into the sum and summing up to $t' - n_j$ with a subset of $\{n_1, \ldots, n_{j-1}\}$, or
- by not taking n_j into the sum and summing up to t' with a subset of $\{n_1, \ldots, n_{j-1}\}$.

The algorithm computes all values $F(j,t')$ and returns $F(k,t)$.

Exercise 2: Matrix multiplication

Multiplying an $n \times m$-matrix with an $m \times o$-matrix takes nmo multiplications of numbers using the naive matrix multiplication algorithm. Furthermore, matrix multiplication is associative, i.e., $A(BC) = (AB)C$ for matrices A, B, C of the right sizes. While the results of the different ways of multiplying the matrices are the same, the number of multiplications is not necessarily the same. For example, if A is a 10×30-matrix, B is a 30×5-matrix, and C is a 5×60-matrix, then

- computing $(AB)C$ takes 4500 multiplications while
- computing $A(BC)$ takes 27.000 multiplications.

The matrix multiplication problem is, given a sequence of n matrices $A_1, A_2, \ldots, A_n$ that can be multiplied as $A_1 A_2 \cdots A_n$, to determine the minimal number of number multiplications necessary to compute the product (but not to compute the product).

Give a dynamic programming algorithm for the matrix multiplication problem. To this end, specify the recurrence relation (including initial values).

Solution:
Given an instance $A_1, \ldots, A_n$ matrices, let $m_0, m_1, \ldots, m_n$ such that A_j is an $m_{j-1} \times m_j$-matrix (such m_j exist, as we assume that the matrices can be multiplied).

Now, define $F(i,j)$ for $1 \leq i \leq j \leq n$ to be the minimal number of multiplications necessary to compute $A_i \cdots A_j$. Then, we have $F(i,i) = 0$ for all i and

$$F(i,j) = \min\{F(i,k) + F(k+1,j) + m_{k-1} m_k m_{k+1} \mid i \leq k < j\}$$

for $i < j$ signifying that to compute $A_i \cdots A_j$ for $i < k$ we need to first compute $A_i \cdots A_k$ (with minimal number of multiplications $F(i, k)$) and $A_{k+1} \cdots A_j$ (with minimal number of multiplications $F(k + 1, j)$) and then multiply the results, which takes $m_{k-1}m_k m_{k+1}$ multiplications.

The algorithm computes all values $F(i, j)$ and returns $F(1, n)$.

Exercise 3: Challenge

Solve Day 16 of the Advent of Code 2022: `https://adventofcode.com/2022/day/16`. Use the input given on Moodle.

Solution:

The dynamic programming idea is to compute $F(v, t, O)$, which is defined as the maximal amount of pressure that can be released when starting at valve v with time $t \leq 30$ left and the valves in O already open. Note that many valves have flow rate 0, so they are useless and can be removed! Thus, O cannot get too large.

$F(v, t, O)$ can be expressed by a recurrence relation taking into account the time it takes to move to a valve and the time it takes to open the valve (note that not every valve has a direct tunnel to every other valve). So, we compute all these values and return $F(v_0, 30, \emptyset)$ where v_0 is the initial valve.