This explains the gist and provides context for the exercises from the problem sets. This is mildly adapted from Martin's emails, all props go to him.

Problem set 1

Only one "theory" problem, and the rest is setting up the Python environment and playing around with it.

Problem set 2

Problem 1 is another modelling exercise, where you have to come up with a linear program (i.e. linear objective, finitely many linear constraints). Be careful not to write a non-linear program or infinitely many constraints!

Problem 3 is technically and probably the most important one, as it shows that in LPs, optimal values are always attained. This is not a compactness argument, as the feasible region might not be compact. What is important is the fact that we have finitely many constraints only. Here's an example with infinitely many constraints where the optimum is not attained: let f be a piecwise linear approximation of $y = \frac{1}{x}$ that is exact precisely at integral x-coordinates; if you minimize y over the epigraph of f, the optimum is 0, but it is not attained.

Problems 4 and 5 are propositions 1.32 and 1.33 from the lecture notes (they are not proved there), and they are technically a bit involved. The steps should guide you to a proof (this is clearly stated in problem 5, but maybe less so in problem 4).

For the programming problem, there is a helper module that hides some code from the problem statement. This should be self-explanatory. Content-wise, you should implement an LP you saw in class (among the LP examples).

Problem set 3

Problem 1 is Carathéodory's theorem for polytopes — mentioned in class, but not yet proved. "Proof by picture" for the 2D-case: If you draw a 2D-polytope and do a triangulation, then any interior point will lie in one of the cells, i.e. in the convex hull of the vertices of the cell. In 2D, cells are triangles, hence the desired 3 vertices. Of course, this can be generalized to higher dimensions, but is probably nasty to write, so we propose a polyhedral proof which that exploits structure of vertices.

Link back to problem set 2: we proved that $P = \text{conv}\,(\text{vertices}\,(P))$, but this statement only guarantees that we can write points in P as convex combinations of vertices at all!

Problem 2 is Proposition 1.37 from the script.

Problem 3 is a helper problem for problems 4 and 5. This is one of the problems where intuition (maybe drawing a picture) immediately tells you that the statement should be true, but then actually defining a proper c in general might not be that easy.

Problem 4 is the forward direction of Proposition 1.38, and **Problem 5** is the backward direction. These are somewhat involved, in particular the backward direction, which is why exhaustive guidance on what to do is provided. You might have seen seemingly more direct proofs of this decomposition theorem, but be aware that textbooks with these easier proofs usually start with Fourier-Motzkin elimination and hence projections, while we use the decomposition theorem to prove the projection statement (Proposition 1.40). Thus, there is a reason why things seem to be a bit complicated here. Both approaches that we suggest use induction on the dimension: In cases where the objects we are considering are not full-dimensional or contain lines, it is easy to reduce dimension. The full-dimensional and no-line case is (in both problems) attacked directly.

The programming exercise is a small one, implementing basis changes in simplex tableaus, where the idea is to multiply the tableau with the inverse of an appropriately chosen submatrix of the tableau (the one corresponding to the new basis columns).

Problem set 4

Problem 1 is to work through a full Phase I + Phase II example using short tableaus. The more practice, the better.

Problem 2 is about information that can be read from simplex tableaus. Should be pretty self-explanatory. It is very important that you know how to read simplex tableaus, otherwise there is no point in doing simplex at all. Problems 1 and 2 are somewhat the minimal requirements to be able to apply Simplex Method to real problems.

Problem 3 treats simplex tableaus once again. If you look at a (short) tableau, you always see some free variables, which correspond to slacks of some of the constraints. If you set these free variables to zero, there is a unique assignment to the basic variables that fills up to a solution of the system. Geometrically, this is the

same as setting the constraints corresponding to the free variables to equality — this system will have a unique solution, as we just said. Hence, the (normal vectors of the) constraints we set to equality must be linearly independent (i.e., full-rank). This is essentially the intuition (actually, a proof) for the implication from left to right. The problem shows that the reverse implication holds true, too. The problem can be solved purely algebraically without thinking about intuition.

Problem 4 is a few short statements, maybe partially tricky ones, to again build up a better feeling for the Simplex Method.

Problem 5 explains an alternative to Bland's rule for avoiding cycling. It is not too hard and actually quite nice.

The programming exercise is to implement a pivoting function. If you want, you may use it to check your calculations in Problem 1 or perhaps even perform them. However, do remember: in the exam, you should be able to do calculations by hand, so it is worth it to do a few steps by hand and it might be better to use the results from problem 1 to check the pivoting function than the other way round.

Problem set 5

Problem 1 is about reading uniqueness of a solution from simplex tabelaus. You could also do the reverse direction and think about what a zero entry in the objective function row implies.

Problem 2 is on Simplex once more, completing the picture of options that we have with a certificate of infeasibility in case Phase I fails. Indeed:

(a) If phase I does not have value 0, this problem shows how to get a certificate of infeasibility for the initial problem.

(b) Else, we have seen how to continue with phase II, where the options were treated in class already:

  (i) If unbounded, we can read an improving feasible direction from the tableau.

  (ii) If bounded, then from an optimal tableau, we can read an optimal solution; here, the dual solution obtained from dual reading is a certificate of optimality (due to strong duality).

Problem 3 is to write the dual of a general linear program, i.e., to see how $\geq$, $=$, or $\leq$-constraints in the primal correspond to ≥ 0, unconstrained, or ≤ 0-variables in the dual, etc. Moreover, you can once again train the transition from any form to canonical form and back.

Problem 4 shows that pivoting in a (primal) tableau corresponds to transformations in the systems obtained from the dual reading that lead to equivalent dual systems. In particular, from this problem we get that the dual systems obtained by dual reading from an initial simplex tableau and an optimal simplex tableau are equivalent.

Problem 5 shows equivalence of the Farkas' Lemma and strong duality. In textbooks (and maybe also in courses that some of you have seen earlier), the Farkas' Lemma is usually used to prove strong duality, but in our course we do it the other way round. Obtaining duals of P' and D' is a special case of problem 3. We want to use the pair P'/D' because these have precisely the constraints that the Farkas' Lemma has, and it helps mapping the dichotomy that we see in the Farkas' Lemma to the one that we see for primal/dual pairs.

The programming exercise has an interesting theory aspect: we use duality for obtaining a separating hyperplane for two non-intersecting polyhedra. Also here, either there is a point that belongs to both polyhedra, i.e. a common solution to both systems, or we can find a suitable separating hyperplane from dualizing the infeasible systems.

Problem set 6

Problem 1 is about complementary slackness and deriving certificates for (non-)optimality — the prime application of complementary slackness.

Problem 2 should be standard for all those who have seen Landau notation and asymptotic growth already, and should make others comfortable with it. Note that our definition of Landau notation is not via limsups/liminfs, but via existence of suitable constants c and M such that $f(n) \leq c \cdot g(n)\ \forall n \geq M$. If you have seen the limit definitions, you may apply them if you want, given everything is done properly. You can also think about the equivalence of the different definitions.

Problem 3 is about running time analysis of a sorting algorithm, should be self-explanatory. Note: what we show in (b) and (c) are two completely different statements.

The point of **Problem 4** is that a trivial algorithm takes time $O(\max\{\deg(u), \deg(v)\})$, namely just going through the incidence lists at u and v, which, course, can be parallelized.

Problem 5 is to apply BFS the right way. The immediate algorithm is to run BFS from every potential meeting point and store the maximum of the distances of the caterpillars (which is the earliest meeting time at

this particular vertex), and then find the best vertex. This is quadratic, though, and it is better to run BFS only three times, once from each of the caterpillar starting vertices, and find the best vertex then.

The programming exercise is about measuring running time in Python, and about efficiently implementing an algorithm that counts connected components in a graph. The obtained plots look pretty nice.

Problem set 7

Problem 1 is to apply the Ford–Fulkerson algorithm. Part (b) should be immediate.

Problem 2 has a small catch: It might seem tempting to just take an s_i-s_{i+1} flow of value k for all i, and then sum all those flows up to obtain an s_1-s_ℓ flow of value k. It is true that the resulting flow will be a flow and of value k, but it will potentially not respect capacity constraints.

Problem 3 is to prove the max-flow min-cut theorem via duality. A very nice problem connecting LP duality and the flows. Note that the probabilistic construction that is done in the problem is essentially used to construct an integral optimal solution of the dual cut LP. It is not clear that there exists an integral solution upfront, but it can actually also be seen easily by explointing TUness, which we do in one of the TUness problem sets later.

Problem 4 and **Problem 5** are applications of the max-flow min-cut theorem to combinatorial problems. In both, we construct suitable digraphs where flows/cuts can be suitably interpreted, which is an approach common to all applications using flows/cuts (also the really applied ones). The challenge is mostly to prove that the correcness of the correspondence of flows/cuts to the modelled objects.

Problem set 8

Problem 1 is a decomposition statement for flows: every flow can be decomposed into paths such that consecutively augmenting along these paths (starting from the zero flow) gives the flow. The Ford–Fulkerson algorithm implies existence of such paths (for maximum flows f, strictly speaking, but this can be extended to all flows), but there are a few caveats:

- The augmenting paths that Ford–Fulkerson takes might not be paths in the initial graph.
- The above can be repaired by uncrossing paths whenever a backward arc is used, but this comes with a blow-up in terms of efficiency (maybe there is data structures that allow doing this fast).
- There is no good guarantee on the number of paths (no strongly polynomial one). This is better with the strongly polytime variations, but will not get to $O(m)$.

The solution is to construct graphs greedily: Choose a path, take the highest capacity among all edges of the path, let this be the corresponding γ. Subtracting the path from the flow, a flow remains, but its support went down by 1. Iterating (at most m times, until support is empty), we get the result.

Problem 2 is about the Dinic's algorithm, another strongly polytime variation that can be seen as an improvement of Edmonds–Karp (and this is the way we present it). It is cut a lot into steps, with the first subproblems for the students to get a feeling of what's going on. The hard part might be (f), and maybe (j), although the idea in (j) is very similar to what you do in Problem 1.

The first programming problem is on minimum cuts, and maximal/minimal min cuts. Note that there might be confusion: Maximal/minimal always refers to maximal/minimal w.r.t. inclusion (hence, is used for sets), while the "minimum" in min cuts refers to the value of the cut (hence, is used for numbers).

The second programming problem is on the flow application with the winning possibilities of sports teams. Here you implement the algorithm and test it on a real-world example (Martin spent quite some time finding one, actually!). In the last part, you find a certificate if a team can not win, which was discussed in the lectures.

Problem set 9

Problem 1 is about the vertex cover polytope. In class you saw that the given description is correct for bipartite graphs. The point is that this is no longer true for general graphs, but there is still things one can do: taking the correct description from the bipartite case over to the non-bipartite case, we obtain a half-integral polytope. Knowing this is useful, since it is somewhat easy to obtain a 2-approximation for the minimum vertex cover problem from that.

Problem 2 is one direction of a result seen in class (but only the other direction was proved).

Problem 3 links back to duality and the max-flow min-cut theorem. Recall that in problem set 7, we already proved the max-flow min-cut theorem via LP duality, and a peculiarity of that proof was that we had to construct an integral dual solution from a randomized argument. This can be done more directly now that we know TU-ness.

Problem 4 is in a similar spirit as Problem 1. You have seen the bipartite matching polytope (not yet the general matching polytope) in class, and the question is whether the description is correct for non-bipartite graphs. (Spoiler: it is not.)

Problems 5 and 6 are more applications of TUness and Theorem 5.8. Note that the title of Problem 5 is "Low discrepancy coloring", because the set R can be interpreted as a coloring of $[n]$: color the elements in R red, and those not in R blue. Then, the problem asks for a coloring such that if we look at the two permutations σ_1 and σ_2, every interval has about the same number of red and blue elements; and discrepancy k means that the difference of the number of blue and red balls in every interval is at most k. To get some intuition, think about the following questions:

- What would discrepancy 0 mean? Can this be achieved?
- What does discrepancy 1 mean? Are there permutations that allow a coloring of discrepancy 1?
- Can you give a lower bound on the minimum discrepancy of a coloring if there are no assumptions on the permutations?

The above might put Problem 5 into a better context and show that we are not proving something fully trivial here.

Problem 6 is an application of the consecutive-ones property to se that the constraint matrix is TU.

Problem set 10

Problem 1 is about adopting the description of the bipartite matching polytope to the case where you are interested in finding matchings of a given fixed size. Two proofs are proposed in (b) and (c): one using TUness and the other using more elementary arguments. Both approaches were shown in class, so it's just a matter of applying them. In general, this problem once again follows the recipe "1. Find a candidate description (provided). 2. Prove that the right integral points are contained (part a). 3. Prove integrality (parts b or c)".

Problem 2 is a property of perfect matching polytopes. This is needed in the proof of Theorem 5.21: note that in the third paragraph, it says "In particular, G is connected". This problem shows that if G is not connected, then there indeed is a smaller counterexample.

Problem 3 is a classical application of polyhedral descriptions: if you have the right polyhedral description of a discrete set (here the set of perfect matchings), then in order to show that the set is non-empty, it is enough to show that the polytope is non-empty, i.e., that it contains any (potentially fractional) point. This is often much easier than finding an actual element of the set. This technique was applied for perfect matchings in d-regular bipartite graphs (Theorem 5.15), and is applied again here.

Problem 4 and **Problem 5** are about properties of the spanning tree polytope, namely its facets and degeneracy. You have only seen the description of the spanning tree polytope (Theorem 5.17), but not that the description is actually the right one. Still, it is good to play around with the polytope a bit before going into the technical uncrossing part. It's useful to recall the definitions of facets, facet-defining inequalities, redundancy, and degeneracy.

Problem 6 is about the dominant of the r-arborescence polytope. You have already seen the description of the spanning tree polytope in Theorem 5.20, but not its proof. In the uncrossing chapter, integrality of the description will be discussed, but not step 2 of the recipe (showing that the description contains the right integral points), which is done in this problem.

The programming exercise is to implement a linear programming problem that has a consecutive-ones matrix, to see in practice that TU descriptions give integral vertex solutions. There is a "bouns" problem to find a combinatorial algorithm for the corresponding unweighted problem.

Problem set 11

Problem 1 is about an alternative description of the perfect matching polytope for general graphs. The idea is that perfect matchings are just matchings with an additional cardinality constraint. Translating this to polytopes, the equivalent statement would be that the perfect matching polytope is simply the matching polytope with a (continuous) cardinality constraint. It turns out that the latter is indeed true, and the Problem 1 guides you to two different proofs of this fact.

Problem 2 settles two properties that are needed in the proof of integrality of the spanning tree polytope. (a) is about deletion of arcs on which the fixed vertex has value 0 (and the restriction to the rest), and (b) is crucial when stating that one system of tight constraints implies the other: Note that in the script, we only show that coefficient vectors are linear combinations of coefficient vectors of subsystems, but we never show that the right-hand sides match, too. (b) shows that we actually don't have to, given a common solution of the systems.

Problem 3 is a property that is needed at the very end of the proof of integrality of the spanning tree polytope, to reach the contradiction on the assumption that $\mathcal{H}_S$ is of minimum cardinality.

Problem 4 is similar to 5.23, and you have seen in the lecture that such equalities can be proved by going over different edge types (based on where their endpoints lie w.r.t. the sets A, B). (b) can be derived from (a) as hinted at in the problem set, or it can be proved using the same method.

Problem 5 is interesting on its own, should improve your understanding of laminar families, and will be needed (at least twice) in the near future: In problem 6, and in the lecture next week when bounding the number of edges in a minimally k-edge-connected graph.

Problem 6 is about a constrained spanning tree problem, where we have degree bounds that should be satisfied at all vertices. We are proving that extreme point solutions have small support, and this is also useful for algorithms: Goemans (http://www-math.mit.edu/∼goemans/PAPERS/bmst-focs06.pdf) proved that one can efficiently find a spanning tree of cost no more than the cost of an optimal solution, but with all degree bounds violated by at most two units (additive violation), and his result relies on the small support observation. The state of the art is a result similar to the mentioned one by Singh and Lau, where the additive violation is only one unit at every vertex (https://cs.uwaterloo.ca/∼lapchi/papers/mbdst.pdf). The latter extends ideas along the following lines: If you have an LP for your problem, solve it. If there are zeroes or ones in the solution, fix them. If there is a "large" value in the solution (for example, $x(e) > 0.5$, round it up to 1, fix this one, resolve the LP, and iterate. This is called iterative rounding. Part (b) shows that there always exists a large coordinate in our problem. This is not enough to get a good approximation algorithm for minimum bounded degree spanning trees, hence Singh and Lau used a different approach: They do iterative relaxation, where they resolve the LP in rounds, every time relaxing some of the degree constraint. This is just to give you some context for the problem.

Problem set 12

Problem 1 and **Problem 2** are there for you to go through a full example using the uncrossing technique themselves. Problem 2 does the actual uncrossing, Problem 1 is to externally prove that (a part of) the resulting system is TU.

Problem 3 (in particular, part (b)), is used in section 6.3 to see that the intersection of the unit hypercube with the dominant of a combinatorial polytope is a $\{0,1\}$-polytope.

Problem 4 adresses a fact that is also mentioned in the lecture, namely proving that the ellipsoids that we are using are actually minimum-volume ellipsoids in each step. The problem shows this for the transformed case where we look for a small ellipsoid containing a half ball. The proof uses KKT conditions and convex programs. You have most likely never seen this before; the goal of this problem is not that you understand KKT conditions after doing it, but that you get to see the ingredients needed to prove the statement. Please note that for what we are doing in the lecture, it is not required that we choose a minimum volume ellipsoid, we only need one with volume small enough (but still containing the half ball) so that the inequality in Lemma 6.4 is satisfied. Hence, just guessing the right ellipsoid (as done in the lecture) is enough.

The programming problem is about a separation oracle for the forest polytope. This is an interesting theory problem on its own, but we decided to make it a programming exercise, too.

Problem set 13

Problem 1 is about another separation oracle, thus giving another polytope to apply the ellipsoid method to. In particular, we look at the perfect matching polytope, where the challenging part is to separate over the odd cut constraints. Separation is easily reduced to the minimum T-odd cut problem, for which the problem porposes an algorithm; the main task is to prove correctness of this algorithm and a strongly polynomial running time bound. Proving correctness involves an uncrossing argument (in the else-case of step 2, we need to see that there is a minimum T-odd cut that still exists in one of the two instances that we look at, and this can be shown by uncrossing an optimal T-odd cut and the cut C found in step 1).

Problem 2 is about the minimum volume ellipsoid containing a rotated half-ball, which is essentially a combination of last weeks result and an appropriate rotation, plus calculating the resulting matrix and vector defining the ellipsoid explicitly.

Problem 3 calculates the volume of a standard simplex. This is mentioned in the script (page 201), where we need a lower bound on the volume of the polytope for which we want to find a feasible point (which we get by taking the volume of a simplex contained in the polytope).