Note: Your TA probably will not cover all the problems. This is totally fine, the discussion worksheets are not designed to be finished in an hour. They are deliberately made long so they can serve as a resource you can use to practice, reinforce, and build upon concepts discussed in lecture, readings, and the homework.

1 LP Basics

Linear Program. A *linear program* is an optimization problem that seeks the optimal assignment for a linear objective over linear constraints. Let $x \in \mathbb{R}^n$ be the set of variables and $A \in \mathbb{R}^{m \times n}, b \in \mathbb{R}^m, c \in \mathbb{R}^n$. The canonical form of a linear program is

$$\begin{aligned} &\text{maximize } c^\top x \\ &\text{subject to } Ax \leq b \\ &\qquad\qquad x \geq 0 \end{aligned}$$

Any linear program can be written in canonical form.

Let's check this is the case:

(i) What if the objective is minimization?

(ii) What if you have a constraint $Ax \geq b$?

(iii) What about $Ax = b$?

(iv) What if the constraint is $x \leq 0$?

(v) What about unconstrained variables $x \in \mathbb{R}$?

Solution:

(i) Take the negative of the objective.

(ii) Negate both sides of the inequality.

(iii) Write both $Ax \leq$ and $Ax \geq b$ into the constraint set.

(iv) Change of variable: replace every x by $-z$, and add constraint $z \geq 0$.

(v) Replace every x by $x^+ - x^-$, add constraints $x^+, x^- \geq 0$. Note that for every solution to the original LP, there is a solution to the transformed LP (with the same objective value). Similarly, if there is a feasible solution for the transformed problem, then there is a feasible solution for the original problem with the same objective value.

2 LP Meets Linear Regression

One of the most important problems in the field of *statistics* is the *linear regression problem*. Roughly speaking, this problem involves fitting a straight line to statistical data represented by points (x_1, y_1), $(x_2, y_2), \ldots, (x_n, y_n)$ on a graph. Denoting the line by $y = a + bx$, the objective is to choose the constants a and b to provide the "best" fit according to some criterion. The criterion usually used is the *method of least squares*, but there are other interesting criteria where linear programming can be used to solve for the optimal values of a and b.

Suppose instead we wish to minimize the sum of the absolute deviations of the data from the line:

$$\min \sum_{i=1}^{n} |y_i - (a + bx_i)|$$

Write a linear program with variables a, b to solve this problem.

Hint: Create new variables z_i and new constraints to help represent $|y_i - (a+bx_i)|$ in a linear program.

Solution:

Note that the smallest value of z that satisfies $z \geq x, z \geq -x$ is $z = |x|$. Now, consider the following linear programming problem:

$$\min \sum_{i=1}^{n} z_i$$

$$\text{subject to} \begin{cases} y_i - (a + bx_i) \leq z_i & \text{for } 1 \leq i \leq n \\ (a + bx_i) - y_i \leq z_i & \text{for } 1 \leq i \leq n \end{cases}.$$

Since $\sum_i z_i$ is minimized, z_i will be set to the $\max(y_i - (a + bx_i), (a + bx_i) - y_i)$. Note that $\max(y_i - (a + bx_i), (a + bx_i) - y_i)$ is, in fact, $|y_i - (a + bx_i)|$.

If for some solution we have that $z_i > |y_i - (a + bx_i)|$, then by setting $z_i = |y_i - (a + bx_i)|$ we will get a solution with a smaller value of the objective function, therefore the initial solution was not optimal. Hence, the constraints requires that the optimal solution will set $z_i = |y_i - (a + bx_i)|$, so the new problem is indeed equivalent to the original problem. However, now it is a linear programming problem.

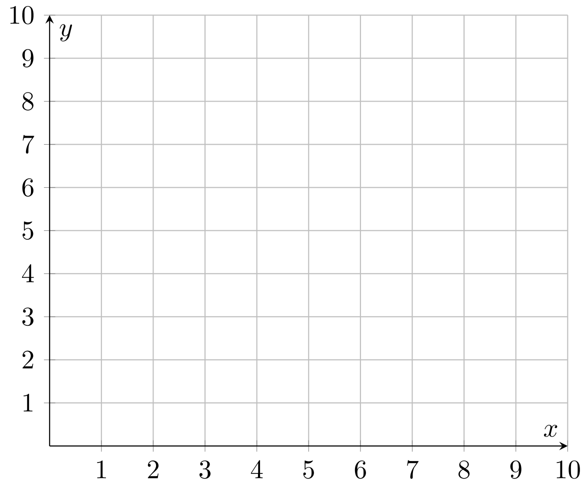
3 Simply Simplex

Consider the following linear program.

$$\max\ 2x + 3y$$

$$\text{subject to} \begin{cases} x + 3y \geq 6 \\ 2x - y \leq 12 \\ x + y \leq 9 \\ x \geq 0, y \geq 0 \end{cases}$$

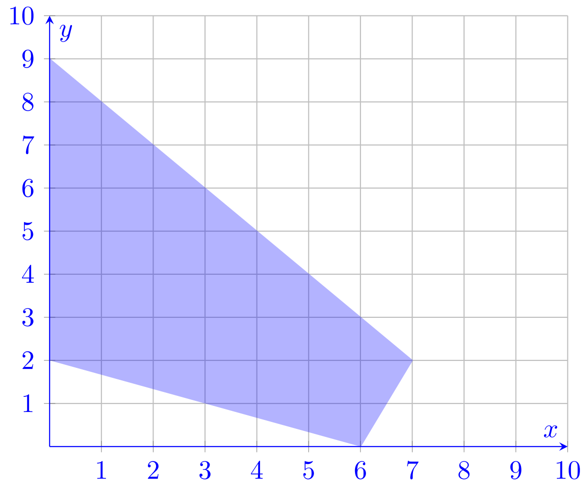
(a) Sketch the feasible region below.



(b) Run the Simplex algorithm on this LP starting at $(6, 0)$. What are the vertices visited? **Please show your work.**

Solution:

(a) We sketch the feasible region below:



(b) The vertices of the feasible region are $(0, 2), (0, 9), (7, 2), (6, 0)$. Depending on the starting vertex, Simplex will visit the vertices in a greedy fashion that maximizes the objective $2x + 3y$. For instance, suppose we started at $(6, 0)$. Here are the steps that Simplex would take:

  (a) We note that the objective value at $(6, 0)$ is $2 \cdot 6 + 3 \cdot 0 = 12$. Its neighbors are $(0, 2)$ and $(7, 2)$, with objective values $2 \cdot 0 + 3 \cdot 2 = 6$ and $2 \cdot 7 + 3 \cdot 2 = 20$. Since we're trying to maximize the objective function, we go to the neighbor with the highest objective value. Thus, we go to $(7, 2)$.

  (b) From before, we know that the objective value of $(7, 2)$ is 20. Now, we compute its neighbors' objective values: we already know that the objective value of $(6, 0)$ is 12, so now just need to find the objective value of $(0, 9)$, which is $2 \cdot 0 + 3 \cdot 9 = 27$. We see that $27 > 20$, and so we move to $(0, 9)$.

  (c) The objective value of $(0, 9)$ is 27, which is higher than the objective values of any of its neighbors. Thus, we have found the optimum point in the LP!

  Throughout this run of Simplex, we visited $(6, 0)$, $(7, 2)$, and $(0, 9)$ in that order.

4 Huffman and LP

Consider the following Huffman code for characters a, b, c, d: $a = 0, b = 10, c = 110, d = 111$.

Let f_a, f_b, f_c, f_d denote the fraction of characters in a file (only containing these characters) that are a, b, c, d respectively. Write a linear program with variables f_a, f_b, f_c, f_d to solve the following problem: What values of f_a, f_b, f_c, f_d (that can generate this Huffman code) result in the Huffman code using the most bits per character?

Solution:

Our objective is to maximize the bits per character used:

$$\max f_a + 2f_b + 3f_c + 3f_d$$

We know the fractions must add to 1 and be non-negative:

$$f_a + f_b + f_c + f_d = 1, f_a, f_b, f_c, f_d \geq 0$$

We know the frequencies of the characters must satisfy $f_a \geq f_b \geq f_c, f_d$. We also know that $f_c + f_d \leq f_a$, since we chose to merge (c, d) with b instead of merging a. So we get the following constraints:

$$f_c \leq f_b, f_d \leq f_b, f_b \leq f_a, f_c + f_d \leq f_a$$

We can write this LP in canonical form as follows:

$$\max\ f_a + 2f_b + 3f_c + 3f_d$$
$$\text{subject to} \begin{cases} f_a + f_b + f_c + f_d \leq 1 \\ -f_a - f_b - f_c - f_d \leq -1 \\ f_c - f_b \leq 0 \\ f_d - f_b \leq 0 \\ f_b - f_a \leq 0 \\ f_c + f_d - f_a \leq 0 \\ f_a, f_b, f_c, f_d \geq 0 \end{cases}$$

Note that for the second-to-last constraint, we write $f_c + f_d - f_a \leq 0$ instead of just $f_c + f_d \leq f_a$ because canonical form requires all the variables to be on the left, and only constants on the right.

5 Job Assignment

There are I people available to work J jobs. The value of person i working 1 day at job j is a_{ij} for $i = 1, \dots, I$ and $j = 1, \dots, J$. Each job is completed after the sum of the time of all workers spend on it add up to be 1 day, though partial completion still has value (i.e. person i working c portion of a day on job j is worth $a_{ij}c$). The problem is to find an optimal assignment of jobs for each person for one day such that the total value created by everyone working is optimized. No additional value comes from working on a job after it has been completed.

(a) What variables should we optimize over? In other words, in the canonical linear programming definition, what is x?

Solution: An assignment x is a choice of numbers x_{ij} where x_{ij} is the portion of person i's time spent on job j.

(b) What are the constraints we need to consider? Hint: there are three major types.

Solution: First, no person i can work more than 1 day's worth of time.

$$\sum_{j=1}^{J} x_{ij} \leq 1 \qquad \text{for } i = 1, \dots, I.$$

Second, no job j can be worked past completion:

$$\sum_{i=1}^{I} x_{ij} \leq 1 \qquad \text{for } j = 1, \dots, J.$$

Third, we require positivity.

$$x_{ij} \geq 0 \qquad \text{for } i = 1, \dots, I, j = 1, \dots, J.$$

(c) What is the maximization function we are seeking?

Solution: By person i working job j for x_{ij}, they contribute value $a_{ij}x_{ij}$. Therefore, the net value is

$$\sum_{i=1,j=1}^{I,J} a_{ij}x_{ij} = A \bullet x.$$

6 Taking a Dual

Consider the following linear program:

$$\begin{aligned}\max\ & 4x_1 + 7x_2\\ & x_1 + 2x_2 \leq 10\\ & 3x_1 + x_2 \leq 14\\ & 2x_1 + 3x_2 \leq 11\\ & x_1, x_2 \geq 0\end{aligned}$$

Construct the dual of the above linear program.

Solution: If we scale the first constraint by $y_1 \geq 0$, the second by $y_2 \geq 0$, the third by $y_3 \geq 0$, and we add them up, we get an upperbound of $(y_1 + 3y_2 + 2y_3)x_1 + (2y_1 + y_2 + 3y_3)x_2 \leq (10y_1 + 14y_2 + 11y_3)$. We need y_1, y_2, y_3 to be non-negative, otherwise the signs in the inequalities flip. Minimizing for a bound for $4x_1 + 7x_2$, we get the tightest possible upperbound by

$$\begin{aligned}\min\ & 10y_1 + 14y_2 + 11y_3\\ & y_1 + 3y_2 + 2y_3 \geq 4\\ & 2y_1 + y_2 + 3y_3 \geq 7\\ & y_1, y_2, y_3 \geq 0\end{aligned}$$

7 Egg Drop Revisited

Recall the Egg Drop problem from Homework 7:

You are given m identical eggs and an n story building. You need to figure out the highest floor $b \in \{0, 1, 2, \ldots n\}$ that you can drop an egg from without breaking it. Each egg will never break when dropped from floor b or lower, and always breaks if dropped from floor $b + 1$ or higher. ($b = 0$ means the egg always breaks). Once an egg breaks, you cannot use it any more. However, if an egg does not break, you can reuse it.

Let $f(n, m)$ be the minimum number of egg drops that are needed to find b (regardless of the value of b).

Instead of solving for $f(n, m)$ directly, we define a new subproblem $M(x, m)$ to be the maximum number of floors for which we can always find b in at most x drops using m eggs.

For example, $M(2, 2) = 3$ because a 3-story building is the tallest building such that we can always find b in at most 2 egg drops using 2 eggs.

(a) Find a recurrence relation for $M(x, m)$ that can be computed in constant time given the previous subproblems. Briefly justify your recurrence.

Hint: As a starting point, what is the highest floor that we can drop the first egg from and still be guaranteed to solve the problem with the remaining $x - 1$ drops and $m - 1$ eggs if the egg breaks?

(b) Give an algorithm to compute $M(x, m)$ given x and m and analyze its runtime.

(c) Briefly justify the correctness of your algorithm via an inductive proof.

(d) Modify your algorithm from (b) to compute $f(n, m)$ given n and m.

Hint: If we can find b when there are more than n floors, we can also find b when there are n floors.

(e) Show that the runtime of the algorithm from part (c) is $O(nm)$. Compare this to the runtime you found in last week's homework.

(f) Show that we can implement the algorithm from part (c) to use only $O(m)$ space.

Solution:

(a)
$$M(x,m) = M(x-1,m-1) + M(x-1,m) + 1$$

We will first deduce i, the optimal floor from which we will drop the first egg given we have x drops and m eggs. If the egg breaks after being dropped from floor i, we have reduced the problem to floors 0 through $i-1$, and the strategy can solve this subproblem using $x-1$ drops and $m-1$ eggs. The optimal strategy for $x-1$ drops and $m-1$ eggs can distinguish between $M(x-1,m-1)+1$ floors, so we should choose $i = M(x-1,m-1)+1$.

On the other hand, if the egg doesn't break, we reduce the problem to floors i to n with $x-1$ drops and m eggs. The maximum number of floors we can distinguish using this many drops and eggs is $M(x-1,m)+1$, so we can solve this subproblem for n as large as $i+M(x-1,m) = M(x-1,m-1)+M(x-1,m)+1$.

(b) For base cases, we'll take $M(0,m) = 0$ for any m and $M(x,0) = 0$ for any x. Starting with $x = 1$, we compute $M(x,m)$ for all, $1 \leq x \leq m$, and do so again for increasing values of x, up until we compute $M(x,m)$ for all $1 \leq x \leq m$. We return $M(x,m)$.

We compute xm subproblems, each of which takes constant time, so the overall runtime if $\Theta(xm)$.

(c) We proceed via induction on both x, m as follows:

Base Case: for any x or m, we have $M(x,0) = M(0,m) = 0$, which are correct.

Inductive Hypothesis: for any $i < x$ and $j < m$, assume that $M(i,j)$ all correctly represent the max number of floors for which we can always find b in at most i drops using m eggs.

Inductive Step: consider the recurrence relation from part (a).

$$M(x,m) = M(x-1,m-1) + M(x-1,m) + 1.$$

We re-use our justification from part (a):

> We will first deduce i, the optimal floor from which we will drop the first egg given we have x drops and m eggs. If the egg breaks after being dropped from floor i, we have reduced the problem to floors 0 through $i-1$, and the strategy can solve this subproblem using $x-1$ drops and $m-1$ eggs. The optimal strategy for $x-1$ drops and $m-1$ eggs can distinguish between $M(x-1,m-1)+1$ floors, so we should choose $i = M(x-1,m-1)+1$.
>
> On the other hand, if the egg doesn't break, we reduce the problem to floors i to n with $x-1$ drops and m eggs. The maximum number of floors we can distinguish

using this many drops and eggs is $M(x-1,m)+1$, so we can solve this subproblem for n as large as $i + M(x-1,m) = M(x-1,m-1) + M(x-1,m) + 1$.

Hence, we've proven that our algorithm correctly computes the $M(\cdot,\cdot)$ values.

(d) Again, starting with $x = 1$, we compute $M(x,m)$ for all, $1 \leq x \leq m$, and do so for increasing values of x. This time, we stop the first time we find that $M(x,m) \geq n$, and return this value of x.

(e) Because there are only n floors, the optimal number of drops, x will always be at most n. From part (b), we know the runtime is $\Theta(xm)$, so if $x \leq n$, we know the runtime must be $O(nm)$.

(This is a very loose bound, but it is still much better than the naive $O(n^2 m)$-time algorithm we'd get from using the recurrence on $f(n,m)$ directly.)

(f) While we're computing $M(x,m)$ for all $1 \leq x \leq m$, we only need to store $M(x-1,m)$ and $M(x,m)$ for all x, i.e. we only ever need to store $O(m)$ values. In particular, after computing $M(x,m)$ for all x, we can delete our stored values of $M(x-1,m)$.