Let's first review the general situation for linear programming problems. Our problem in **standard form** is to choose a vector $x \in \mathbf{R}^n$, such that $x \geq \mathbf{0}$ and $Ax = b$, to maximize a linear function $c \cdot x$. Here A is an m by n matrix, b is an m-vector with $b \geq \mathbf{0}$, and c is an n-vector.

Last time we argued that a standard-form problem can be converted into **slack form**, where we think of x as being defined by $n - m$ of its variables. The rule $Ax = b$ defines the other m variables in terms of those variables, and changes the n constraints $x_i \geq 0$ into **linear inequalities** on those variables.

Remember that we also adjusted A so that its rows are linearly independent. This means that some sets of m columns of A are linearly independent. If we have such a set of columns, there is a unique way to form the m-vector b as a linear combination of those columns. There is thus a unique vector x satisfying $Ax = b$ and having only those m entries of x nonzero. This is called a **basic solution** to the linear program.

Geometrically, we represent vectors x by points in $\mathbf{R}^{n-m}$, and represent each linear constraint by a **half-space**, all the points on a given side of some **hyperplane**. The set of points, if any, that are on the right side of all n hyperplanes is called the **feasible region**. Our goal is to find the point within the feasible region that maximizes $c \cdot x$. The objective function $c \cdot x$ is still a linear function of the $n - m$ variables we are modeling, because each of the other variables is a linear function of those given the rule $Ax = b$.

We also observed last time that the intersection of the n half-spaces forms a **polyhedron** in $\mathbf{R}^{n-m}$, called a **polytope** if it is bounded. We argued that $c \cdot x$ must be maximized (or minimized) at some **vertex** of the polyhedron, unless it takes on arbitrarily large or arbitrarily small values within the feasible region.

Note that at any of the points x giving a basic solution, the equations of $n - m$ of the constraint planes are true, and x must therefore be on the intersection of those $n-m$ planes, at a *particular point*. A basic solution is a **basic feasible solution** or **BFS** if the other variables are non-negative, so the point is a *vertex* of the feasible region. As long as the constraint planes are in general position, each vertex of the feasible region must be a BFS.

If we could compute the value of the objective function at each vertex and find the largest value, we would thus find the maximum. But it is entirely possible for there to be exponentially many vertices in the polytope. For example, suppose that $m = n/2$ and that along with $x_i \geq 0$ for each of the $n - m$ variables considered, we get a constraint $x_i \leq 1$ from one of the other variables. The feasible region is then a **hypercube** of dimension m. Every point $(x_1, \ldots, x_{n-m})$ with $x_i = 0$ or $x_i = 1$ is a vertex of the hypercube, so there are exactly 2^{n-m} vertices.

Of course this particular problem does not have the constraint hyperplanes in **general position**, because various pairs of them are parallel. (The other condition for general position is that no three planes mutually intersect in a space of dimension $n - m - 2$ instead of a space of dimension $n - m - 3$ – think of three planes in $\mathbf{R}^3$ intersecting in a common line instead of a point. Actually it' s possible to have k hyperplance intersecting in a space of dimension greater than $n - m - k$, and we have to rule this out as well.) But the planes could be in slightly different places and still give a very similar-looking feasible region with 2^{n-m} vertices.

The basic idea of the **simplex algorithm** is very simple:

- Find a vertex inside the feasible region.
- As long as there is a neighbor of the current vertex with a larger value of $c \cdot x$, move to it.
- When you reach a local optimum, declare victory.

Here a "neighbor" is a vertex that is connected to the current vertex by a line segment. Remember that the vertex is a point on $n - m$ of the hyperplanes. If we take any $n - m - 1$ of those hyperplanes, their intersection forms a line. If we follow one of these lines, we either reach a vertex (when we cross one of the other hyperplanes) or go on forever.

This algorithm is *correct* because the feasible region is **convex** – any line segment connecting two points in the region lies entirely in the region. (This is proved by induction on the number of half-spaces.) If there were a point in the region with value greater than the local optimum, there would be a way to get to it by following line segments.

Each of the two tasks we have set ourselves, finding a feasible vertex and moving to a more desirable neighbor, presents some implementation difficulties. Once we have dealt with them, we can consider *how fast* the simplex algorithm works. Specifically, how many vertices might we have to visit before we find the optimum one?

First, what exactly does it mean in terms of programming to "be at a vertex" or "move along a line segment to another vertex"? (The discussion here is only a general sketch – CLRS Chapter 29 has a much more detailed implementation of the simplex algorithm.) Our point is a vector x, which it is convenient to store as an n-vector (so that only some of the variables represent coordinates in the feasible region of $\mathbf{R}^{n-m}$). Since we are in the feasible region, we know that $Ax = b$ is true, and to remain in the region, we must keep this equation true.

If we are at a vertex, this means that $n - m$ of our n variables are zero (because we are on $n - m$ of the hyperplanes). There are thus m nonzero entries in x, and it is the m columns of A corresponding to these entries that are participating in the equation $Ax = b$.

Now suppose that we want to move to a different vertex. This means that we must take a variable that is currently zero and make it nonzero. If we give x_i some positive value λ, though, we will violate the constraint $Ax = b$ because we will increase Ax by λ times the i'th column of A. We need to change the other components of x to compensate.

Fortunately, we know that the m columns of A corresponding to the nonzero entries of our original x form a basis of $\mathbf{R}^m$. (Why? Since we are at a vertex, the n-vector version of x is a BFS, and the columns corresponding to the nonzero entries of a BFS are linearly independent.) This means that column i can be written as a linear combination s of those n columns, so that A times the unit vector e_i is equal to As, where s is a vector that is nonzero only in those m locations. If we subtract λAs from Ax, by subtracting s from x, at the same time we add λ times column i to Ax, we keep Ax the same and it still equals b.

As we increase λ from zero, x_i becomes nonzero, the other $n-m-1$ entries of x that were zero stay zero, and the m that were nonzero change, perhaps some up and some down. As the point moves, we remain on $n-m-1$ of the constraint planes, meaning that we are on a *line* between two vertices. Eventually, if one or more of the nonzero components is decreasing, λ will reach a value where $x - \lambda s$ has a new zero entry. The first time this happens, we have $n-m$ entries of x equal to zero again and we are at a new vertex.

This vertex must still be in the feasible region, because we could have left the feasible region only by making an entry of x negative (since we made sure that $Ax = b$ was still true). What happened to the objective function $c \cdot x$? By computing $c \cdot (e_i - s)$, we can find the change in $c \cdot x$ from every unit that λ increases. We *only* want to make variable x_i nonzero if this change is *positive*. In that case, the objective function will be larger at the new vertex than it was at the old vertex, and we will have made progress.

There remains the problem of finding an initial BFS, so we can start the whole process of the simplex algorithm. This is a nontrivial problem! There are $\binom{n}{m}$ possible sets of m columns of A, some of which represent basic solutions because the columns are linearly independent. But the basic solution for that set of columns may have negative entries, making it infeasible.

We can find a basic feasible solution to one problem in standard form by creating another problem, with an easy-to-find basic feasible solution, and solving that. Suppose we want a basic feasible solution to the linear program defined by $Ax = b$. We make a new linear program with m new variables in an m-vector y and defining equation $Ax + y = b$. Our objective function in the new problem is $y_1 + \ldots + y_n$.

We apply the simplex method to find a solution minimizing this objective function. If we minimize it at 0 (it can't get any lower because the y_i's must all be non-negative) then we have a solution to $Ax + \mathbf{0} = b$ which is just a solution to $Ax = b$ – this can be our initial BFS for the original problem. And if we minimize the objective function at some positive value, this means that there is no solution to $Ax = b$ at all – the feasible region of the original problem is empty.

Of course we still have to start the simplex algorithm on the new problem with an initial BFS. But now we can take x to be $\mathbf{0}$ and y to be b, finally using the fact that $b \geq \mathbf{0}$.

Given any vertex, a bit of arithmetic can tell us which variables we could make nonzero and increase the objective function – which **choices of pivot** are available. But if there are more than one choice, we must choose which. Some possibilities are:

- We could choose the pivot that maximizes the change in the objective function per unit of λ,
- We could calculate the total increase in the objective function for each move to an adjacent vertex, and maximize that,
- We could choose randomly from the acceptable pivots.

Unfortunately, for each of these methods there are examples of linear programs where the simplex method visits (or probably visits) exponentially many vertices and thus takes exponential time. On the other hand, simplex "usually" does well, visiting only a linear number of vertices. I say "usually" in quotes because it is not clear how to define a probability distribution on linear programming problems! Many papers have been written about that "usually".

When the number of dimensions of the problem (which we have been calling $n - m$ and will now call d) is small, there are ways to get better performance such as $O(n2^d)$. Some of these are randomized algorithms and are presented in Chapter 9 of Motwani-Raghavan.

They note a potential lower bound on the number of vertices visited by the simplex algorithm. Thought of as just a graph, the feasible region has a **diameter**, the longest distance between two points if each edge has length 1. If the initial BFS and the optimum were at these two points, then the algorithm would have to visit at least as many vertices as the diameter. How big could the diameter be? A theorem of Kalai and Kleitman says that it is at most $n^{2+\log d}$, but a better bound might hold. Even a randomized version of simplex could only run in polynomial time on all problems if this diameter bound can be improved to polynomial.

(An aside about MIT combinatorist Dan Kleitman, apropos of the social network talk yesterday – since he appeared briefly in and was the mathematical advisor for the film *Good Will Hunting*, Kleitman has a Bacon number of two to go with his Erdös number of one.)

There are non-simplex algorithms that are proved to always run in polynomial time. The first discovered were **ellipsoid** algorithms (Khachian, 1979), but these are not generally competitive with the simplex method. Karmarkar in 1984 developed the first **interior-point** method, which considers points that are not on the boundary of the feasible region. This was controversial at the time because he at first kept his code proprietrary, which meant that it could not be evaluated by the research community. His and other interior-point methods are now more widely used.

We'll conclude today by looking at two more applications of linear programming.

We saw in Lecture #23 that the network flow problem can be reformulated as a linear program. Since we already have good polynomial-time deterministic algorithms for network flow, this may seem to be a curiosity. But linear programming is a very general problem. There are two interesting variations of network flow for which the best known method is linear programming:

- In MINIMUM-COST NETWORK FLOW (see CLRS p. 787), each edge has a cost per unit transported over it. Given a flow target, our problem is to find a flow (if any) that meets the flow target and has the minimum possible cost. It is easy to adjust our linear program for network flow to force the total flow to be some number t, and then to minimize the cost, which is just a linear function of the individual edge flows.

- In MULTICOMMODITY FLOW (CLRS p. 788), we have a constant number of different commodities to move across our network, each with its own source, destination, and desired total flow. The total amount of flow of all commodities across any edge may not exceed the edge capacity. In effect we have a constant number of flow problems which must be solved simultaneously, so that the sum of their flows must meet the edge capacities. (While flows of one commodity in opposite directions cancel, flows of different capacities do not – this poses some complications.) But we can make a linear program just as before, with more variables and more constraints – CLRS say that this is the best known way to solve the problem.

We can use linear programming to get another approximation algorithm for MIN-VERTEX-COVER with approximation factor 2. It is easy to formulate VERTEX-COVER as an *integer* programming problem – variable x_v is 0 if vertex v is not in the cover and 1 if it is, each edge (u, v) gives a constraint $x_u + x_v \geq 1$, and we want to minimize $\Sigma_v\, x_v$. The solution to this in integers is just the minimum vertex cover, so it is **NP**-hard to find it.

But suppose we view these variables and constraints as defining a *linear* program, and solve that. We get a real number for each vertex, saying "how much" it is a member of the vertex cover. So an edge (u, v) might be "covered" because $x_u = 0.4$ and $x_v = 0.6$.

The approximation problem is to **round** this real-valued vector to an integer vector, setting y_1 to be 1 if $x_i \geq 1/2$ and $y_i = 0$ otherwise. Then the vector y defines a set of vertices. This set is a vertex cover, because for any edge (u, v) we had $x_u + x_v \geq 1$ and so at least one of u or v must have had x value at least $1/2$ and thus have y value 1.

How does this approximation compare to the optimal vertex cover? The total weight of y can be no more than double that of x, because we only spent a unit of y on a vertex when x had already spent at least a half-unit on v. And the total weight of x must be no more than the total weight of the optimal vertex cover, because x is the optimal real-valued vector meeting the constraints and the optimum meets the constraints.

This general approach is called **relaxing** the integer programming problem to a corresponding linear programming problem – relaxing and **rounding** like this is a fruitful source of approximation algorithms.