(See https://cs.stanford.edu/~knuth/programs.html for date.)

1. Introduction. I'm writing this program in order to gain personal experience implementing the simplex algorithm—even though I know that commercial codes do the job much, much better. My aim is to learn, to make the logic "crystal clear" if not lightning fast, and perhaps also to watch and interact with this magical process.

The computational task to be solved has been traditionally called Linear Programming, and it takes many forms. The particular case considered here is to maximize $b_1v_1+\cdots+b_nv_n$ subject to the constraints $v_j\ge0$ for $1\le j\le n$ and $a_{i1}v_1+\cdots+a_{in}v_n\le c_i$ for $1\le i\le m$, where a_{ij} is a given $m\times n$ matrix of integers and the other parameters b_j, c_i are integers with $c_i\ge0$. For example, what is the maximum of $v_1+3v_2-v_3$, if we require that

$$3v_1-v_2+5v_3\le8,\qquad -v_1+2v_2-v_3\le1,\qquad 2v_1+4v_2+v_3\le5,$$

and $v_1,v_2,v_3\ge0$? [I took this example from Muroga's book on threshold logic (1971).] The fact that the constants c_i on the right-hand side are nonnegative means that the trivial values $v_1=\cdots=v_n=0$ always satisfy the constraints; hence the maximum is always nonnegative.

The algorithm below also solves a "dual" problem as a special bonus. Namely, it tells us how to minimize the quantity $c_1u_1+\cdots+c_mu_m$ subject to $u_i\ge0$ and $a_{1j}u_1+\cdots+a_{mj}u_m\ge b_j$ for $1\le j\le n$.

There may no values $(u_1,\ldots,u_m)$ that meet those dual constraints. In such a case, the algorithm will effectively prove the impossibility, and it will also demonstrate that the maximum in the original problem is $+\infty$. (For example, suppose $m=n=1$, $b_1=1$, $c_1=0$, and $a_{11}=-1$. The problem of maximizing v_1 subject to $-v_1\le0$ and $v_1\ge0$ obviously has $+\infty$ as its answer; and $+\infty$ is also the "minimum" of the quantity 0 taken over all u_1 such that $u_1\ge0$ and $-u_1\ge1$, because no such numbers u_1 exist.)

The first line of the standard input file should contain the integers $b_1\ b_2\ \ldots\ b_n$ in decimal notation, separated by spaces. That first line should be followed by m further lines that each contain $n+1$ integer values $c_i\ a_{i1}\ a_{i2}\ \ldots\ a_{in}$, for $1\le i\le m$.

To enhance this learning experience, I'm solving the problem both with floating-point arithmetic and with an all-integer method that produces rational numbers as output. Hopefully the two answers will agree. But the all-integer method might overflow, and the floating-point method may suffer from rounding errors.

```
#define maxm 10  /* of course these limits can be raised if desired */
#define maxn 100  /* (up to a point) */
#define buf_size BUFSIZ
#include <stdio.h>
#include <ctype.h>
  ⟨ Include tricky code for zapping 2 ⟩
  typedef long intword;  /* will be long long on my other computer */
  char buf[buf_size];
  intword a[maxm+1][maxm+maxn+1];  /* integer work area */
  intword denom[maxm+maxn+1];  /* scale factors */
  double aa[maxm+1][maxm+maxn+1];  /* floating-point work area */
  double trial[maxm+maxn+1];  /* pivot testing area */
  int verbose=1;  /* can be set positive for extra output */
  int count;  /* the number of steps taken so far */
  int p[maxm+1], q[maxm+maxn+1];  /* current basis and inverse */
  main()
  {
    register intword h, i, j, k, l, m, n, s;
    register double z;
    ⟨ Check the zap trick 3 ⟩;
    ⟨ Read the input matrix 4 ⟩;
    ⟨ Solve the problem 13 ⟩;
  }
```

2. The algorithm is very sensitive to zeroness or nonzeroness of numbers. I'll try to avoid problems with floating-point roundoff by zapping anything near 0.0 to 0.0. For speed, I do this in "machine language."

```
⟨ Include tricky code for zapping 2 ⟩ ≡
#define little_endian 1      /* on less crazy machines I would define 'big_endian' instead */
#ifdef big_endian
#define bigend first
#else
#define bigend second
#endif
#define zap if ((zbuf.uint.bigend & #7fffffff) < #3e000000) zbuf.dbl = 0.0;
  union {
    struct { unsigned int first, second; } uint;
    double dbl;
  } zbuf;
```

This code is used in section 1.

3. Here's a test that my intentions are being fulfilled by that trickery.

```
⟨ Check the zap trick 3 ⟩ ≡
  zbuf.dbl = .000000001;      /* this value should not be zapped */
  zap;
  if (zbuf.dbl) {
    zbuf.dbl = -.0000000001;      /* but this one should */
    zap;
    if (¬zbuf.dbl) goto zap_OK;
  }
  fprintf(stderr, "Zapping␣doesn't␣work!\n");
  exit(-666);
zap_OK:
```

This code is used in section 1.

4. ⟨ Read the input matrix 4 ⟩ ≡

```
for (i = n = 0; ; i++) {
  if (¬fgets(buf, buf_size, stdin)) break;
  if (i > maxm) {
    fprintf(stderr, "Sorry, I'm set up only for m<=%d!\n", maxm);
    exit(−9);
  }
  for (k = 0, j = (i ≡ 0); buf[k]; ) {
    while (isspace(buf[k]) ∧ buf[k] ≠ '\n') k++;
    if (buf[k] ≡ '\n') break;
    if (buf[k] ≡ '-') l = 1, k++; else l = 0;
    for (s = 0; buf[k] ≥ '0' ∧ buf[k] ≤ '9'; k++) s = 10 ∗ s + buf[k] − '0';
    a[i][j++] = (l ? −s : s);
  }
  if (¬buf[k]) {      /∗ no end-of-line in the buffer ∗/
    fprintf(stderr, "Input line too long! (%s...)\n", buf);
    exit(−1);
  }
  if (i ≡ 0) {
    n = j − 1;
    if (n > maxn) {
      fprintf(stderr, "Sorry, I'm set up only for n<=%d, not n=%d!\n", maxn, n);
      exit(−2);
    }
  } else {
    if (n ≠ j − 1) {
      fprintf(stderr, "Row %d should have %d numbers!\n>%s", i, n + 1, buf);
      exit(−3);
    }
    if (a[i][j] < 0) {
      fprintf(stderr, "Row %d's constant term shouldn't be negative!\n>%s", i, buf);
      exit(−4);
    }
  }
}
m = i − 1;
```

This code is used in section 1.

5. The algorithm: An example. The famous simplex procedure is subtle yet not difficult to fathom, even when we are careful to avoid infinite loops. But I always tend to forget the details a short time after seeing them explained in a book. Therefore I will try here to present the algorithm in my own favorite way—which tends to be algebraic and combinatoric rather than geometric—in hopes that the ideas will then be forever memorable, at least in my own mind. I'm not going to explain how the algorithm was invented, but I am going to explain how and why it works.

To clarify the exposition, I'll work through the simple example in the introduction, watching each step in slow motion. That problem was to maximize the quantity $v_1 + 3v_2 - v_3$ over all nonnegative real values (v_1, v_2, v_3) for which

$$3v_1 - v_2 + 5v_3 \le 8, \qquad -v_1 + 2v_2 - v_3 \le 1, \qquad 2v_1 + 4v_2 + v_3 \le 5.$$

6. The first step in the simplex method is to introduce m additional nonnegative variables called "slack variables" $(s_1, \ldots, s_m)$, which allow us to deal with equivalities instead of inequalities. For example, the inequality $3v_1 - v_2 + 5v_3 \le 8$ becomes the equality $s_1 + 3v_1 - v_2 + 5v_3 = 8$ when we add in the nonnegative variable s_1. These slack variables lead to m further columns of the input matrix a_{ij}; we conceptually place these columns between the constant terms c_i and the other coefficients a_{i1}, $\ldots$, a_{in}.

The state of the computation is conveniently represented in a working array with $m+1$ rows and $m+n+1$ columns. In our example, the working array takes the following form:

Row	(-1)	(s_1)	(s_2)	(s_3)	(v_1)	(v_2)	(v_3)
0	0	0	0	0	−1	−3	1
1	8	1	0	0	3	−1	5
2	1	0	1	0	−1	2	−1
3	5	0	0	1	2	4	1

Rows 1 through m represent the given equations, with coefficients to be multiplied by the column labels. For example, the bottom row represents the equation $s_3 + 2v_1 + 4v_2 + v_3 = 5$, namely

$$5(-1) + 0(s_1) + 0(s_2) + 1(s_3) + 2(v_1) + 4(v_2) + 1(v_3) \;=\; 0.$$

The top row, row 0, is somewhat special, but it also represents an equation—in this case

$$v_1 + 3v_2 - v_3 + 0(-1) + 0(s_1) + 0(s_2) + 0(s_3) - 1(v_1) - 3(v_2) + 1(v_3) \;=\; 0.$$

Let X be the set of all nonnegative vectors $(s_1, s_2, s_3, v_1, v_2, v_3)$ that satisfy the equations in rows 1, 2, and 3. The simplex algorithm will transform the array in such a way that the rows change, but the new set of rows will still define exactly the same set X. Furthermore the special equation represented by row 0 will also remain true, as we will see below.

7. The algorithm also maintains two other invariants. First, there will always be m special columns called "basis columns," one for each index i in the range $1 \le i \le m$. All entries of basis column i are 0 except for a 1 in row i. When we begin, the initial basis columns are those for slack variables, labeled (s_1) through (s_m).

Second, all rows except row 0 will remain *lexicographically nonnegative*. In other words, the leftmost nonzero entry of row i will always be positive, for $1 \le i \le m$. (That row might begin with many zeros, but it cannot be entirely zero, because of the 1 in its basis column.)

8. Elementary row operations on this array do not change the set X. Namely, we can multiply each element of row i by a nonzero constant, when $i \neq 0$; and we can also add any multiple of row $i \neq 0$ to any other row $j \neq i$, even when $j = 0$.

Of course such row operations might mess up the basis columns. But a suitable combination of row operations, called a "pivot step," does allow us to change the position of a basis column. For example, suppose we multiply row 2 by -1, then add multiples of that row to the other rows so as to zero out the other entries in column (v_1); we get a new basis column for row 2:

Row	(-1)	(s_1)	(s_2)	(s_3)	(v_1)	(v_2)	(v_3)
0	-1	0	-1	0	0	-5	2
1	11	1	3	0	0	5	2
2	-1	0	-1	0	1	-2	1
3	7	0	2	1	0	8	-1

However, we don't really want to do this, because row 2 has now become lexicographically negative.

Going back to the former matrix, let's try pivoting in column (v_1) on row 1 instead of row 2. This time we divide row 1 by 3 and add suitable multiples to rows 0, 2, and 3, obtaining

Row	(-1)	(s_1)	(s_2)	(s_3)	(v_1)	(v_2)	(v_3)
0	8/3	1/3	0	0	0	$-10/3$	8/3
1	8/3	1/3	0	0	1	$-1/3$	5/3
2	11/3	1/3	1	0	0	5/3	2/3
3	$-2/3$	0	7	1	0	14/3	$-7/3$

Hmm; it's still no good. Now the *bottom* row is giving trouble.

If we want column (v_1) to move into the basis, our only hope is to pivot on row 3. That works:

Row	(-1)	(s_1)	(s_2)	(s_3)	(v_1)	(v_2)	(v_3)
0	5/2	0	0	1/2	0	-1	3/2
1	1/2	1	0	$-3/2$	0	-7	7/2
2	7/2	0	1	1/2	0	4	$-1/2$
3	5/2	0	0	1/2	1	2	1/2

All of the invariants mentioned above have now been preserved. And we've moved the basis, thereby obtaining a new way to look at the set X over which we wish to take a maximum.

9. But oops, we've now run into fractions instead of nice, simple integers. No problem: We can also scale the columns, by changing the labels:

Row	$(-1/2)$	(s_1)	(s_2)	$(s_3/2)$	(v_1)	(v_2)	$(v_3/2)$
0	5	0	0	1	0	-1	3
1	1	1	0	-3	0	-7	7
2	7	0	1	1	0	4	-1
3	5	0	0	1	1	2	1

10. Okay, let's continue. Another pivot operation, in column (v_2) and row 2, followed by another rescaling, yields

Row	$(-1/8)$	(s_1)	$(s_2/4)$	$(s_3/8)$	(v_1)	(v_2)	$(v_3/8)$
0	27	0	1	5	0	0	11
1	53	1	7	-5	0	0	21
2	7	0	1	1	0	1	-1
3	6	0	-2	2	1	0	6

11. And now we're done! From this tableau we can read off the answer to the maximization problem, namely that the desired maximum of $v_1 + 3v_2 - v_3$ is $27/8$, and that it is obtained when $v_1 = 6/8$, $v_2 = 7/8$, and $v_3 = 0$.

Buy why are we done, you ask? Good question. Here's why: First, by looking at the three basis columns, we see that the point $(s_1, s_2, s_3, v_1, v_2, v_3) = (53/8, 0, 0, 6/8, 7/8, 0)$ is in X, since rows 1, 2, and 3 represent equations that hold everywhere in that set.

Second, row 0 also represents a valid equation, namely

$$v_1 + 3v_2 - 3v_3 + 27(-1/8) + 0(s_1) + 1/(s_2/4) + 5(s_3/8) + 0(v_1) + 0(v_2) + 11(v_3/8) \;=\; 0.$$

(Think about it: Pivot steps don't change the validity of the equation represented by row 0, which always is prefaced by '$v_1 + 3v_2 - v_3$', since they simply add or subtract multiples of zero.)

Thus, at all points $(s_1, s_2, s_3, v_1, v_2, v_3)$ of X, the value of $v_1 + 3v_2 - 3v_3$ is equal to $\frac{27}{8} - \frac{1}{4}s_2 - \frac{5}{8}s_3 - \frac{11}{8}v_3$; it can't possibly get any lower than $27/8$.

12. There's more good news, too: The solution to the corresponding *minimization* problem, namely to minimize $8u_1 + u_2 + 5u_3$ subject to $u_1, u_2, u_3 \ge 0$ and

$$3u_1 - u_2 + 2u_3 \ge 1, \qquad -u_1 + 2u_2 + 4u_3 \ge 3, \qquad 5u_1 - u_2 + u_3 \ge -1,$$

can *also* be read from our final tableau, by looking at the slack-variable columns of row 0. The minimum value $27/8$ is attained when $u_1 = 0$, $u_2 = 1/4$, and $u_3 = 5/8$.

Again you ask, why? Again it's a good question. In the first place we can use the v's from the maximization problem to prove that $27/8$ is indeed unbeatable, because the inequalities

$$\frac{6}{8}\big(3u_1 - u_2 + 2u_3\big) \ge \frac{6}{8}, \qquad \frac{7}{8}\big(-u_1 + 2u_2 + 4u_3\big) \ge \frac{21}{8}, \qquad 0\big(5u_1 - u_2 + u_3\big) \ge 0$$

can be added to give $\frac{11}{8}u_1 + u_2 + 5u_3 \ge \frac{27}{8}$; increasing $\frac{11}{8}u_1$ to $8u_1$ can't make the value any smaller. In general, if we multiply the inequalities that affect (u_1, u_2, u_3) by any values $(v_1, v_2, v_3) \in X$, we obtain a lower bound $8u_1 + u_2 + 5u_3 \ge t_1u_1 + t_2u_2 + t_3u_3 \ge v_1 + 3v_2 - v_3$, because $t_1 \le 8$, $t_2 \le 1$, and $t_3 \le 5$.

Why, however, do the values (u_1, u_2, u_3) from the slack-variable columns of row 0 actually attain the best lower bound? To understand the answer, let's look at the final tableau *without* suppressing the scale factors that were used to avoid fractions:

Row	(-1)	(s_1)	(s_2)	(s_3)	(v_1)	(v_2)	(v_3)
0	27/8	0	1/4	5/8	0	0	11/8
1	53/8	1	7/4	−5/8	0	0	21/8
2	7/8	0	1/4	1	0	1	−1/8
3	3/4	0	−1/2	1/4	1	0	3/4

Consider especially the $(m+1) \times m$ submatrix in the slack columns; these entries encapsulate the effects of all the pivot steps that brought us to the present state. Namely, we replaced row 0 by $r_0 + 0r_1 + \frac{1}{4}r_2 + \frac{5}{8}r_3$, where r_0, r_1, r_2, and r_3 are the original contents of those rows. (We also replaced row 1 by $\frac{7}{4}r_1 - \frac{5}{8}r_2$; we replaced row 2 by $\frac{1}{4}r_1 + \frac{1}{8}r_2$; and we replaced row 3 by $-\frac{1}{2}r_1 + \frac{1}{4}r_2 + r_3$. These coefficients should suffice to convince a skeptic that our final tableau does follow from the original constraints, without forcing him or her to replay the actual pivot steps.)

In particular, we got the number $\frac{27}{8}$ in the constant column by adding $0 \cdot 5 + \frac{1}{4} \cdot 1 + \frac{5}{8} \cdot 5$.

13. The algorithm: An implementation. We've been looking at a small example, but our reasoning has been perfectly general. The main point is that we were able to find a sequence of pivot steps that preserved the desired invariants and that also led to a tableau in which row 0 had no negative entries. Whenever such a tableau is found, we have solved the maximization problem for $(v_1, \ldots, v_n)$ as well as the minimization problem for $(u_1, \ldots, u_m)$.

```
⟨ Solve the problem 13 ⟩ ≡
  ⟨ Set up the initial tableaux 14 ⟩;
loop: if (verbose) ⟨ Print out the current state 22 ⟩;
  for (j = m + n; j > 0; j−−)
    if (a[0][j] < 0) {
      ⟨ Try to pivot in column j 15 ⟩;
      count++;
      goto loop;
    }
  ⟨ Report the answers 23 ⟩;
```

This code is used in section 1.

14. Instead of distinguishing slack variables s_i from ordinary variables v_j, we will henceforth call the variables $w_1, \ldots, w_{m+n}$, with $w_i = s_i$ for $1 \le i \le m$ and $w_{m+j} = v_j$ for $1 \le j \le n$.

Here we set up an $(m+1) \times (m+n+1)$ working tableau a of integers, as well as a table of scale factors. A column label like $(s_2/4)$ in the example above will be represented by $denom[2] = 4$ in this program.

A floating-point tableau aa is also inaugurated here, since we will compute everything in two ways.

The current basis is represented by arrays p and q. If the basis column for row i is column j, we have $p[i] = j$ and $q[j] = i$. Other entries of the q array are zero.

```
⟨ Set up the initial tableaux 14 ⟩ ≡
  for (i = 0; i ≤ m; i++) {
    for (j = n; j > 0; j−−)
      if (i ≡ 0) a[0][j + m] = −a[0][j]; else a[i][j + m] = a[i][j];
    for (j = m; j > 0; j−−) a[i][j] = (i ≡ j);
    p[i] = q[i] = i;
  }
  for (j = m + n + 1; j ≥ 0; j−−) denom[j] = 1;
  for (i = 0; i ≤ m; i++)
    for (j = m + n + 1; j ≥ 0; j−−) aa[i][j] = a[i][j];
```

This code is used in section 13.

15. At this point we have reached a tableau with a negative entry in row 0 and column (w_j). Two cases arise: The corresponding column might contain at least one positive entry; or it might not.

In the latter case, we can stop. Our tableau proves that the maximization problem has $+\infty$ as its answer, because arbitrarily large values of w_j lie in X. These values increase $c_1v_1+\cdots+c_nv_n$ without limit, because of the negative coefficient in row 0. Moreover, there cannot be any values $(u_1,\ldots,u_m)$ that satisfy the dual inequalities; if they did, $b_1u_1+\cdots+b_mu_m$ would be an upper bound on $c_1v_1+\cdots+c_nv_n$.

```
⟨ Try to pivot in column j 15 ⟩ ≡
  l = 0;
  for (i = 1; i ≤ m; i++)
    if (a[i][j] > 0) ⟨ Consider pivoting at (i, j) 16 ⟩;
  if (l ≡ 0) {
    printf("The␣maximum␣is␣infinite;␣the␣dual␣has␣no␣solution!\n");
    ⟨ Print out the current state 22 ⟩;
    exit(0);
  }
  ⟨ Pivot at (l, j) 17 ⟩;
```

This code is used in section 13.

16. When $a_{0j}<0$ and $a_{ij}>0$, a pivot step in row i and column j always increases the lexicographic value of row 0, because it adds a positive multiple of the (lexicographically positive) row i. Therefore Pivoting is a Good Thing: It leads to continual progress toward larger and larger top rows.

But which rows can we pivot on, without making another row lexicographically negative? Our example above showed that random pivoting doesn't always work. Perhaps we were just lucky to find a good pivot in that problem; it's conceivable that another example might run into a state from which no decent pivot is possible.

Fortunately there *is* always a row on which to pivot, in fact a *unique* row, in any given column j that has at least one positive entry a_{ij}. The reason is that the operation of pivoting on a_{ij} causes row k (call it r_k) to be replaced by $r_k-a_{kj}r_i/a_{ij}$ for each $k\neq i$; and it is easy to see that $r_k-a_{kj}r_i/a_{ij}$ is lexicographically positive if and only if $(r_k/a_{kj})-(r_i/a_{ij})$ is lexicographically positive. Hence we must pivot on the row for which r_i/a_{ij} is *lexicographically smallest*, among all rows i with $a_{ij}>0$.

We cannot have r_k/a_{kj} exactly equal to r_i/a_{ij} when $k\neq i$, because those rows differ in basis columns k and i.

Notice that this choice of pivot row does not depend on the scale factors in *denom*. We can safely use floating-point arithmetic when making the choice, because such rounding errors are tightly controlled.

```
⟨ Consider pivoting at (i, j) 16 ⟩ ≡
  if (l ≡ 0) l = i, s = 0;
  else {
    for (h = 0; ; h++) {
      if (h ≡ s) trial[s++] = (double) a[l][h]/(double) a[l][j];
      z = (double) a[i][h]/(double) a[i][j];
      if (trial[h] ≠ z) break;
    }
    if (trial[h] > z) l = i, trial[h] = z, s = h + 1;      /* trial[h] is best so far, for 0 ≤ h < s */
  }
```

This code is used in section 15.

17.

```
⟨ Pivot at (l, j) 17 ⟩ ≡
  ⟨ Do floating-point pivoting 18 ⟩;
  ⟨ Do integer pivoting 20 ⟩;
  q[p[l]] = 0, p[l] = j, q[j] = l;
```

This code is used in section 15.

18. Before we do any integer pivoting, we'd like to be sure that an all-floating-point method would make the same decision. So this step repeats some of the work we've already done, but it uses the aa tableau instead of a.

```
⟨ Do floating-point pivoting 18 ⟩ ≡
  {
    register int ii, jj, kk, ll;
    for (ll = 0, jj = m + n; jj > 0; jj −−)
      if (aa[0][jj] < 0) {
        if (jj ≠ j) goto mismatch;
        for (ii = 1; ii ≤ m; ii ++)
          if (aa[ii][j] > 0) {
            if (ll ≡ 0) ll = ii, s = 0;
            else {
              for (h = 0; ; h++) {
                if (h ≡ s) trial[s++] = aa[ll][h]/aa[ll][j];
                z = aa[ii][h]/aa[ii][j];
                zbuf.dbl = trial[h] − z; zap;
                if (zbuf.dbl) break;
              }
              if (zbuf.dbl > 0.0) ll = ii, trial[h] = z, s = h + 1;
            }
          }
        if (ll ≠ l) goto mismatch;
        ⟨ Really do floating-point pivoting 19 ⟩;
        goto fp_pivot_done;
      }
  mismatch: printf("The␣floating-point␣and␣fixed-point␣implementations␣disagree!\n");
    printf("(Floating-point␣pivoting␣at␣(%d,%d),␣not␣(%d,%d).)\n", ll, jj, l, j);
    ⟨ Print out the current state 22 ⟩;
    exit(−99);
  }
fp_pivot_done:
```

This code is used in section 17.

19. Pivoting. We're ready at last to update the tableaux: Arithmetic happens here, as we pivot on column j of row l.

```
⟨ Really do floating-point pivoting 19 ⟩ ≡
  for (k = 0, z = aa[l][j]; k ≤ m + n; k++)
    if (aa[l][k]) aa[l][k] = aa[l][k]/z;      /* no zap needed */
  for (i = 0; i ≤ m; i++)
    if (i ≠ l) {
      for (k = 0, z = aa[i][j]; k ≤ m + n; k++)
        if (k ≡ j) aa[i][k] = 0.0;
        else {
          zbuf.dbl = aa[i][k] − z ∗ aa[l][k]; zap;
          aa[i][k] = zbuf.dbl;
        }
    }
```

This code is used in section 18.

20. In the all-integer version I'm hoping with fingers crossed that the numerators and denominators will stay small, at least in the simple cases that are greatest interest to me at the moment.

```
⟨ Do integer pivoting 20 ⟩ ≡
  if (verbose) printf("Pivoting␣on␣(%d,%d).\n", l, j);
  for (k = 0; k ≤ m + n; k++)
    if (a[l][k] ∧ k ≠ j) {
      register intword t, u = a[l][k], v = a[l][j];
      if (u < 0) u = −u;
      if (v < 0) v = −v;
      while (v) t = u, u = v, v = t % v;      /* Euclid's algorithm, sets u ← gcd(u, v) */
      if (u ≡ a[l][j]) a[l][k] = a[l][k]/u;
      else {
        v = a[l][j]/u, denom[k] ∗= v;      /* scale factor goes up in column k */
        for (i = 0; i ≤ m; i++) a[i][k] = (i ≡ l ? a[l][k]/u : a[i][k] ∗ v);
      }
    }
  for (i = 0; i ≤ m; i++)
    if (i ≠ l) {
      for (k = 0, h = a[i][j]; k ≤ m + n; k++) a[i][k] = (k ≡ j ? 0 : a[i][k] − h ∗ a[l][k]);
    }
  a[l][j] = 1;
  if (denom[j] ≠ 1) {
    for (h = denom[j], denom[j] = 1, k = 0; k ≤ m + n; k++)
      if (k ≠ j) a[l][k] ∗= h;
  }
```

This code is used in section 17.

21. Final touches. A few last-minute odds and ends remain.

22. ⟨ Print out the current state 22 ⟩ ≡

```
{
  printf("Step␣%d:\n", count);
  for (i = 0; i ≤ m; i++) {
    for (j = 0; j ≤ m + n; j++) printf("␣%d", a[i][j]);
    printf("\n");
  }
  printf("denom");
  for (j = 0; j ≤ m + n; j++) printf("␣%d", denom[j]);
  printf("\n");
  for (i = 0; i ≤ m; i++) {
    for (j = 0; j ≤ m + n; j++) printf("␣%.15g", aa[i][j]);
    printf("\n");
  }
}
```

This code is used in sections 13, 15, and 18.

23. ⟨ Report the answers 23 ⟩ ≡

```
printf("Optimal␣value␣%.15g=%d/%d␣found␣after␣%d␣pivots.\n", aa[0][0], a[0][0], denom[0], count);
printf("␣Optimal␣v's:");
for (j = m + 1; j ≤ m + n; j++)
  if (q[j]) printf("␣%.15g=%d/%d", aa[q[j]][0], a[q[j]][0], denom[0]);
  else printf("␣0");
printf("\n␣Optimal␣u's:");
for (j = 1; j ≤ m; j++) printf("␣%.15g=%d/%d", aa[0][j], a[0][j], denom[j]);
printf("\n");
```

This code is used in section 13.

24. Well, our little program is done. But an attentive reader may well have noticed that an important point has not yet been considered: We haven't proved that the algorithm must terminate.

An elementary knowledge of matrix theory suffices to close this final gap. We shall prove the following lemma: *Given any ordered choice of m columns, there is at most one achievable tableau for which those columns are the basis.*

Proof. Let A_0 be rows 1 to m of the original tableau. At every stage of the algorithm, rows 1 to m of the current tableau are equal to BA_0, for some nonsingular matrix B determined by the pivot operations. Furthermore, if we are told which columns are the basis columns, the matrix B is fully determined; only one B can yield the correct values in those columns. Therefore the choice of basis columns also tells us the entire contents of rows 1 to m. And row 0 is also known, because it is the original row 0 plus $\sum_{i=1}^{m} c'_i r_i$, where $c'_i = 0$ if basis column i corresponds to a slack variable, $c'_i = c_j$ if basis column i corresponds to v_j. QED.

But we have shown that row 0 continually increases, lexicographically. So the algorithm cannot get to the same basis twice; and there are only finitely many bases. Termination is inevitable.

25. Index.

LP