1*. Intro. Given row sum, column sums, and diagonal sums on *stdin*, this program outputs clauses by which a SAT solver can determine if they are compatible with the existence of an $m \times n$ matrix x_{ij} of zeros and ones.

The row sums are $r_i = \sum_{j=1}^{n} x_{ij}$, for $1 \le i \le m$. The column sums are $c_j = \sum_{i=1}^{m} x_{ij}$, for $1 \le j \le n$. And the diagonal sums are $a_d = \sum\{x_{ij} \mid i + j = d + 1\}$ and $b_d = \sum\{x_{ij} \mid i - j = d - n\}$, for $0 < d < m + n$. They should appear one per line in the input, in a format such as '`r3=20`'. Zero sums need not be given. The program deduces m and n from the largest subscripts that appear, and it makes fairly careful syntax checks.

Well actually, the above should be amended: This version works not only with sums of ones, it also uses sums of '11's (that is, consecutive occurrences of 1s in the same line). The second-order sums are given after a comma; for example, '`r3=20,8`'.

```
#define mmax 200   /* should be at most 255 unless I use bigger radix than hex */
#define nmax 100   /* should be at most 255 unless I use bigger radix than hex */
#include <stdio.h>
#include <stdlib.h>
  int r[mmax + 1], c[mmax + 1], a[mmax + nmax], b[mmax + nmax];   /* the given data */
  int rr[mmax + 1], cc[mmax + 1], aa[mmax + nmax], bb[mmax + nmax];   /* and its extensions */
  int count[mmax + mmax + nmax + nmax];   /* leaf counts for the BB method */
  char buf[80];
  char name[mmax + nmax][9];
  ⟨ Subroutines 10 ⟩;
  main()
  {
    register int d, i, j, k, l, ll, m, n, nn, t;
    register char *p;
    ⟨ Input the data 2* ⟩;
    ⟨ Check the data 7* ⟩;
    ⟨ Output the clauses 8 ⟩;
  }
```

2*. ⟨ Input the data 2* ⟩ ≡

```
m = n = 0;
while (1) {
  if (¬fgets(buf, 80, stdin)) break;
  for (d = 0, p = buf + 1; *p ≥ '0' ∧ *p ≤ '9'; p++) d = 10 * d + *p − '0';
  if (*p++ ≠ '=') {
    fprintf(stderr, "Missing␣‘=’␣sign!\nBad␣line:␣%s", buf);
    exit(−1);
  }
  for (l = 0; *p ≥ '0' ∧ *p ≤ '9'; p++) l = 10 * l + *p − '0';
  if (*p++ ≠ ',') {
    fprintf(stderr, "Missing␣comma!\nBad␣line␣%s", buf);
    exit(−12);
  }
  for (ll = 0; *p ≥ '0' ∧ *p ≤ '9'; p++) ll = 10 * ll + *p − '0';
  if (*p ≠ '\n') {
    fprintf(stderr, "Missing␣\\n␣character!\nBad␣line␣%s", buf);
    exit(−2);
  }
  switch (buf[0]) {
    ⟨ Cases for row, column, and diagonal sums 3* ⟩
  default: fprintf(stderr, "Data␣must␣begin␣with␣r,␣c,␣a,␣or␣b!\nBad␣line␣%s", buf);
    exit(−3);
  }
}
```

This code is used in section 1*.

3*. ⟨ Cases for row, column, and diagonal sums 3* ⟩ ≡

```
case 'r':
  if (d < 1 ∨ d > mmax) {
    fprintf(stderr, "Row␣index␣out␣of␣range!\nBad␣line␣%s", buf);
    exit(−4);
  }
  if (l < 0 ∨ l > nmax) {
    fprintf(stderr, "Row␣data␣out␣of␣range!\nBad␣line␣%s", buf);
    exit(−5);
  }
  if (d > m) m = d;
  if (r[d]) {
    fprintf(stderr, "The␣value␣of␣r%d␣has␣already␣been␣given!\nBad␣line␣%s", d, buf);
    exit(−6);
  }
  r[d] = l, rr[d] = ll;
  break;
```

See also sections 4*, 5*, and 6*.

This code is used in section 2*.

4*. ⟨ Cases for row, column, and diagonal sums 3* ⟩ +≡

```
case 'c':
  if (d < 1 ∨ d > nmax) {
    fprintf(stderr, "Column␣index␣out␣of␣range!\nBad␣line␣%s", buf);
    exit(−14);
  }
  if (l < 0 ∨ l > mmax) {
    fprintf(stderr, "Column␣data␣out␣of␣range!\nBad␣line␣%s", buf);
    exit(−15);
  }
  if (d > n) n = d;
  if (c[d]) {
    fprintf(stderr, "The␣value␣of␣c%d␣has␣already␣been␣given!\nBad␣line␣%s", d, buf);
    exit(−16);
  }
  c[d] = l, cc[d] = ll;
  break;
```

5*. ⟨ Cases for row, column, and diagonal sums 3* ⟩ +≡

```
case 'a':
  if (d < 1 ∨ d ≥ mmax + nmax) {
    fprintf(stderr, "Diagonal␣index␣out␣of␣range!\nBad␣line␣%s", buf);
    exit(−24);
  }
  if (l < 0 ∨ l > mmax ∨ l > nmax) {
    fprintf(stderr, "Diagonal␣data␣out␣of␣range!\nBad␣line␣%s", buf);
    exit(−25);
  }
  if (a[d]) {
    fprintf(stderr, "The␣value␣of␣a%d␣has␣already␣been␣given!\nBad␣line␣%s", d, buf);
    exit(−26);
  }
  a[d] = l, aa[d] = ll;
  break;
```

6*. ⟨ Cases for row, column, and diagonal sums 3* ⟩ +≡

```
case 'b':
  if (d < 1 ∨ d ≥ mmax + nmax) {
    fprintf(stderr, "Diagonal␣index␣out␣of␣range!\nBad␣line␣%s", buf);
    exit(−34);
  }
  if (l < 0 ∨ l > mmax ∨ l > nmax) {
    fprintf(stderr, "Diagonal␣data␣out␣of␣range!\nBad␣line␣%s", buf);
    exit(−35);
  }
  if (b[d]) {
    fprintf(stderr, "The␣value␣of␣b%d␣has␣already␣been␣given!\nBad␣line␣%s", d, buf);
    exit(−36);
  }
  b[d] = l, bb[d] = ll;
  break;
```

7*. ⟨ Check the data 7* ⟩ ≡

```
for (i = 1, l = 0; i ≤ m; i++) l += r[i];
nn = l;
for (j = 1, l = 0; j ≤ n; j++) l += c[j];
if (l ≠ nn) {
  fprintf(stderr, "The total of the r's is %d, but the total of the c's is %d!\n", nn, l);
  exit(−40);
}
for (d = 1, l = 0; d < m + n; d++) l += a[d];
if (l ≠ nn) {
  fprintf(stderr, "The total of the r's is %d, but the total of the a's is %d!\n", nn, l);
  exit(−41);
}
for (d = 1, l = 0; d < m + n; d++) l += b[d];
if (l ≠ nn) {
  fprintf(stderr, "The total of the r's is %d, but the total of the b's is %d!\n", nn, l);
  exit(−41);
}
fprintf(stderr, "Input for %d rows and %d columns successfully read", m, n);
fprintf(stderr, " (total %d)\n", nn);
printf("~ sat-tomography-2nd (%dx%d, %d)\n", m, n, nn);
```

This code is used in section 1*.

8. The variables x_{ij} of the unknown Boolean matrix are denoted by 'ixj'. Auxiliary variables by which we check lower and upper bounds for row sum r_i are denoted by 'iRl'. And similar conventions hold for the column sums and the diagonal sums.

⟨ Output the clauses 8 ⟩ ≡

```
for (i = 1; i ≤ m; i++) ⟨ Output clauses to check r_i 9* ⟩;
for (j = 1; j ≤ n; j++) ⟨ Output clauses to check c_j 17 ⟩;
for (d = 1; d < m + n; d++) ⟨ Output clauses to check a_d 18 ⟩;
for (d = 1; d < m + n; d++) ⟨ Output clauses to check b_d 19 ⟩;
```

This code is used in section 1*.

9*. We use the methods of Bailleux and Boufkhad (see SAT-THRESHOLD-BB-EQUAL). Indeed, Bailleux and Boufkhad introduced those methods because they wanted to study digital tomography problems.

⟨ Output clauses to check r_i 9* ⟩ ≡

```
{
  sprintf(buf, "%dR", i);
  for (j = 1; j ≤ n; j++) sprintf(name[j], "%dx%d", i, j);
  gen_clauses(n, r[i]);
  gen_clauses1(n − 1, rr[i]);
}
```

This code is used in section 8.

10. ⟨ Subroutines 10 ⟩ ≡

```
void gen_clauses(int n, int r)
{
  register int i, j, k, jl, jr, t, tl, tr, swap = 0;
  if (r > n − r) swap = 1, r = n − r;
  if (r < 0) {
    fprintf(stderr, "Negative␣parameter␣for␣case␣%s!\n", buf);
    exit(−99);
  }
  if (r ≡ 0) ⟨ Handle the trivial case directly 16 ⟩
  else {
    ⟨ Build the complete binary tree with n leaves 11 ⟩;
    for (i = n − 2; i; i−−) {
      ⟨ Generate the clauses for node i 12 ⟩;
      ⟨ Generate additional clauses for node i 13 ⟩;
    }
    ⟨ Generate the clauses at the root 14 ⟩;
    ⟨ Generate additional clauses at the root 15 ⟩;
  }
}
```

This code is used in section 1*.

11. The tree has $2n-1$ nodes, with 0 as the root; the leaves start at node $n-1$. Nonleaf node k has left child $2k+1$ and right child $2k+2$. Here we simply fill the *count* array.

⟨ Build the complete binary tree with n leaves 11 ⟩ ≡

```
for (k = n + n − 2; k ≥ n − 1; k−−) count[k] = 1;
for ( ; k ≥ 0; k−−) count[k] = count[k + k + 1] + count[k + k + 2];
if (count[0] ≠ n) {
  fprintf(stderr, "I'm␣totally␣confused.\n");
  exit(−666);
}
```

This code is used in section 10.

12. If there are t leaves below node i, we introduce $k = \min(r, t)$ auxiliary variables, beginning with the symbolic name in *buf* and ending with two hex digits of $i + 1$ and two hex digits of j, for $1 \le j \le k$. This variable will be 1 if and only if at least j of those leaf variables are true. If $t > r$, we also assert that no $r + 1$ of those variables are true.

```
#define x(k) printf("%s%s", swap ? "~" : "", name[(k) − n + 2])
#define xbar(k) printf("%s%s", swap ? "" : "~", name[(k) − n + 2])
```

⟨ Generate the clauses for node i 12 ⟩ ≡

```
{
  t = count[i], tl = count[i + i + 1], tr = count[i + i + 2];
  if (t > r + 1) t = r + 1;
  if (tl > r) tl = r;
  if (tr > r) tr = r;
  for (jl = 0; jl ≤ tl; jl++)
    for (jr = 0; jr ≤ tr; jr++)
      if ((jl + jr ≤ t) ∧ (jl + jr) > 0) {
        if (jl) {
          if (i + i + 1 ≥ n − 1) xbar(i + i + 1);
          else printf("~%s%02x%02x", buf, i + i + 2, jl);
        }
        if (jr) {
          printf("␣");
          if (i + i + 2 ≥ n − 1) xbar(i + i + 2);
          else printf("~%s%02x%02x", buf, i + i + 3, jr);
        }
        if (jl + jr ≤ r) printf("␣%s%02x%02x\n", buf, i + 1, jl + jr);
        else printf("\n");
      }
}
```

This code is used in section 10.

13. So far we've only propagated the effects of the known 1s among the x's. Now we propagate the effects of the 0s: If there are fewer than tl 1s in the leaves of the left subtree and fewer than tr 1s in those of the right subtree, there are fewer than $tl + tr - 1$ in the leaves of below node i.

⟨ Generate additional clauses for node i 13 ⟩ ≡

```
if (t > r) t = r;
for (jl = 1; jl ≤ tl + 1; jl++)
  for (jr = 1; jr ≤ tr + 1; jr++)
    if (jl + jr ≤ t + 1) {
      if (jl ≤ tl) {
        if (i + i + 1 ≥ n − 1) x(i + i + 1);
        else printf("%s%02x%02x", buf, i + i + 2, jl);
        printf("␣");
      }
      if (jr ≤ tr) {    /* note that we can't have both jl > tl and jr > tr */
        if (i + i + 2 ≥ n − 1) x(i + i + 2);
        else printf("%s%02x%02x", buf, i + i + 3, jr);
        printf("␣");
      }
      printf("~%s%02x%02x\n", buf, i + 1, jl + jr − 1);
    }
```

This code is used in section 10.

14. Finally, we assert that at most r of the x's are true, by implicitly asserting that the (nonexistent) variable for $i = 0$ and $j = r + 1$ is false.

⟨ Generate the clauses at the root 14 ⟩ ≡

```
tl = count[1], tr = count[2];
for (jl = 1; jl ≤ tl; jl++) {
  jr = r + 1 − jl;
  if (jr > 0 ∧ jr ≤ tr) {
    if (1 ≥ n − 1) xbar(1);
    else printf("~%s02%02x", buf, jl);
    printf("␣");
    if (2 ≥ n − 1) xbar(2);
    else printf("~%s03%02x", buf, jr);
    printf("\n");
  }
}
```

This code is used in section 10.

15. To make *exactly* r of the x's true, we also assert that the (nonexistent) variable for $i = 1$ and $j = r$ is true.

⟨ Generate additional clauses at the root 15 ⟩ ≡

```
for (jl = 1; jl ≤ r; jl++) {
  jr = r + 1 − jl;
  if (jr > 0) {
    if (jl ≤ tl) {
      if (1 ≥ n − 1) x(1);
      else printf("%s02%02x", buf, jl);
      printf("␣");
    }
    if (jr ≤ tr) {
      if (2 ≥ n − 1) x(2);
      else printf("%s03%02x", buf, jr);
    }
    printf("\n");
  }
}
```

This code is used in section 10.

16. ⟨ Handle the trivial case directly 16 ⟩ ≡

```
{
  for (i = 1; i ≤ n; i++) {
    xbar(n − 2 + i);
    printf("\n");
  }
}
```

This code is used in section 10.

17. ⟨ Output clauses to check c_j 17 ⟩ ≡

```
{
  sprintf(buf, "%dC", j);
  for (i = 1; i <= m; i++) sprintf(name[i], "%dx%d", i, j);
  gen_clauses(m, c[j]);
}
```

This code is used in section 8.

18. ⟨ Output clauses to check a_d 18 ⟩ ≡

```
{
  sprintf(buf, "%dA", d);
  if (m <= n) {
    if (d <= m) {
      for (i = 1; i <= d; i++) sprintf(name[i], "%dx%d", i, d + 1 - i);
      gen_clauses(d, a[d]);
    } else if (d <= n) {
      for (i = 1; i <= m; i++) sprintf(name[i], "%dx%d", i, d + 1 - i);
      gen_clauses(m, a[d]);
    } else {
      for (t = 1; t <= m + n - d; t++) sprintf(name[t], "%dx%d", d + t - n, n + 1 - t);
      gen_clauses(m + n - d, a[d]);
    }
  } else {
    if (d <= n) {
      for (i = 1; i <= d; i++) sprintf(name[i], "%dx%d", i, d + 1 - i);
      gen_clauses(d, a[d]);
    } else if (d <= m) {
      for (j = 1; j <= n; j++) sprintf(name[j], "%dx%d", d + 1 - j, j);
      gen_clauses(n, a[d]);
    } else {
      for (t = 1; t <= m + n - d; t++) sprintf(name[t], "%dx%d", d + t - n, n + 1 - t);
      gen_clauses(m + n - d, a[d]);
    }
  }
}
```

This code is used in section 8.

19. ⟨ Output clauses to check b_d 19 ⟩ ≡

```
{
  sprintf(buf, "%dB", d);
  if (m ≤ n) {
    if (d ≤ m) {
      for (i = 1; i ≤ d; i++) sprintf(name[i], "%dx%d", i, n + i − d);
      gen_clauses(d, b[d]);
    } else if (d ≤ n) {
      for (i = 1; i ≤ m; i++) sprintf(name[i], "%dx%d", i, n + i − d);
      gen_clauses(m, b[d]);
    } else {
      for (j = 1; j ≤ m + n − d; j++) sprintf(name[j], "%dx%d", j + d − n, j);
      gen_clauses(m + n − d, b[d]);
    }
  } else {
    if (d ≤ n) {
      for (i = 1; i ≤ d; i++) sprintf(name[i], "%dx%d", i, n + i − d);
      gen_clauses(d, b[d]);
    } else if (d ≤ m) {
      for (j = 1; j ≤ n; j++) sprintf(name[j], "%dx%d", j + d − n, j);
      gen_clauses(n, b[d]);
    } else {
      for (j = 1; j ≤ m + n − d; j++) sprintf(name[j], "%dx%d", j + d − n, j);
      gen_clauses(m + n − d, b[d]);
    }
  }
}
```

This code is used in section 8.

20*. Index.

The following sections were changed by the change file: 1, 2, 3, 4, 5, 6, 7, 9, 20.

SAT-TOMOGRAPHY-2ND