**1st 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 $1 \leq i \leq m$. The column sums are $c_j = \sum_{i=1}^{m} x_{ij}$, for $1 \leq j \leq n$. 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'.

```c
#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+nmax+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);
}
```
\[2^* \quad \langle \text{Input the data} \quad 2^* \rangle \equiv \]

\[m = n = 0;\]

\[\text{while (1) \{} \]
\[\text{if } (!\text{fgets}(\text{buf}, 80, \text{stdin})) \quad \text{break;} \]
\[\text{for } (d = 0, p = \text{buf} + 1; \quad \star p \geq ^{0'} \land \star p \leq ^{9'}; \quad p++) \quad d = 10 \ast d + \star p - ^{0'}; \]
\[\text{if } (\star p++ \neq ^{\ast}, ^{\ast}) \{ \]
\[\text{fprintf}(\text{stderr}, \text{"Missing }^{\ast} \text{, sign!}\text{\nBad line : }%s", \text{buf}); \]
\[\text{exit}(-1); \}
\[\text{\}} \]
\[\text{for } (l = 0; \quad \star p \geq ^{0'} \land \star p \leq ^{9'}; \quad p++) \quad l = 10 \ast l + \star p - ^{0'}; \]
\[\text{if } (\star p++ \neq ^{,}, ^{,}) \{ \]
\[\text{fprintf}(\text{stderr}, \text{"Missing, comma!}\text{\nBad line : }%s", \text{buf}); \]
\[\text{exit}(-12); \}
\[\text{\}} \]
\[\text{for } (ll = 0; \quad \star p \geq ^{0'} \land \star p \leq ^{9'}; \quad p++) \quad ll = 10 \ast ll + \star p - ^{0'}; \]
\[\text{if } (\star p \neq ^{\\text{n}}, ^{\text{n}}) \{ \]
\[\text{fprintf}(\text{stderr}, \text{"Missing \text{n}, character!}\text{\nBad line : }%s", \text{buf}); \]
\[\text{exit}(-2); \}
\[\text{\}} \]
\[\text{switch } (\text{buf}[0]) \{ \]
\[\text{(Cases for row, column, and diagonal sums } 3^* \}) \]
\[\text{default: fprintf}(\text{stderr}, \text{"Data must begin with r, c, a, or b!}\text{\nBad line : }%s", \text{buf}); \]
\[\text{exit}(-3); \}
\[\text{\}} \]
\[\text{This code is used in section } 1^* . \]

\[3^* \quad \langle \text{Cases for row, column, and diagonal sums } 3^* \rangle \equiv \]

\[\text{case } ^{\text{r}}: \]
\[\text{if } (d < 1 \lor d > \text{mmax}) \{ \]
\[\text{fprintf}(\text{stderr}, \text{"Row index out of range!}\text{\nBad line : }%s", \text{buf}); \]
\[\text{exit}(-4); \}
\[\text{\}} \]
\[\text{if } (l < 0 \lor l > \text{mmax}) \{ \]
\[\text{fprintf}(\text{stderr}, \text{"Row data out of range!}\text{\nBad line : }%s", \text{buf}); \]
\[\text{exit}(-5); \}
\[\text{\}} \]
\[\text{if } (d > m) \quad m = d; \]
\[\text{if } (\text{r}[d]) \{ \]
\[\text{fprintf}(\text{stderr}, \text{"The value of } r[d]\text{ has already been given!}\text{\nBad line : }%s", d, \text{buf}); \]
\[\text{exit}(-6); \}
\[\text{\}} \]
\[\text{r}[d] = l, \text{rr}[d] = ll; \]
\[\text{break; } \]
\[\text{See also sections } 4^*, 5^*, \text{ and } 6^*. \]
\[\text{This code is used in section } 2^*. \]
4* (Cases for row, column, and diagonal sums 3*) +

```c
if (d < 1 || d > nmax) {
    fprintf(stderr, "Column index outside of range! \nBad line \%s", buf);
    exit(-14);
}
if (l < 0 || l > mmax) {
    fprintf(stderr, "Column data outside of range! \nBad line \%s", buf);
    exit(-15);
}
if (d > n) n = d;
if (c[d]) {
    fprintf(stderr, "The value \%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*) +

```c
if (d < 1 || d > mmax + nmax) {
    fprintf(stderr, "Diagonal index outside of range! \nBad line \%s", buf);
    exit(-24);
}
if (l < 0 || l > mmax || l > nmax) {
    fprintf(stderr, "Diagonal data outside of range! \nBad line \%s", buf);
    exit(-25);
}
if (a[d]) {
    fprintf(stderr, "The value \%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*) +

```c
if (d < 1 || d > mmax + nmax) {
    fprintf(stderr, "Diagonal index outside of range! \nBad line \%s", buf);
    exit(-34);
}
if (l < 0 || l > mmax || l > nmax) {
    fprintf(stderr, "Diagonal data outside of range! \nBad line \%s", buf);
    exit(-35);
}
if (b[d]) {
    fprintf(stderr, "The value \%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!
", 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!
", 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!
", nn, l);
      exit(-41);
   }
   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, r[i]);
}

This code is used in section 1*.

8. The variables 𝑥𝑖𝑗 of the unknown Boolean matrix are denoted by ‘ixj’. Auxiliary variables by which we check lower and upper bounds for row sum 𝑟𝑖 are denoted by ‘𝑖𝑅’. 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 𝑟𝑖 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 𝛼 𝑑 18);
   for (d = 1; d < m + n; d++) (Output clauses to check 𝑏 𝑑 19);

This code is used in section 8.

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 𝑟𝑖 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, r[r[i]]);
   }

This code is used in section 8.
10. \{Subroutines 10\} ≡

\begin{verbatim}
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\}
    else {
        \{Build the complete binary tree with n leaves\}
        for (i = n - 2; i; i--) {
            \{Generate the clauses for node i\}
            \{Generate additional clauses for node i\}
        }
        \{Generate the clauses at the root\}
        \{Generate additional clauses at the root\}
    }
}
\end{verbatim}

This code is used in section 1*.

11. The tree has $2^n - 1$ 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 \texttt{count} array.

\begin{verbatim}
\begin{verbatim}
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] \neq n) {
    fprintf(stderr, "I'm totally confused.\n");
    exit(-666);
}
\end{verbatim}
\end{verbatim}

This code is used in section 10.
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 \( 1 \leq j \leq k \). 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.

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

(Generate the clauses for node \( i \))

```c
{  
  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 <= t; 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%c02x%02x", buf, i + i + 2, jl);  
        if (jr)  
          printf ("n");  
          if (i + i + 2 >= n - 1) xbar(i + i + 2);  
          else printf ("%s%c02x%02x", buf, i + i + 3, jr);  
        if (jl + jr <= r) printf (",%s%c02x%02x\n", buf, i + 1, jl + jr);  
        else printf ("\n");
}
```

This code is used in section 10.

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 the right subtree, there are fewer than \( tl + tr - 1 \) in the leaves of below node \( i \).

(Generate additional clauses for node \( i \))

```c
{  
  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%c02x%02x", buf, i + i + 2, jl);  
          printf ("n");  
        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%c02x%02x", buf, i + i + 3, jr);  
          printf ("n");
  
  printf (",%s%c02x%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.

\[ tl = \text{count}[1], tr = \text{count}[2]; \]
\[ \text{for } (jl = 1; jl \leq tl; jl++) \{ \]
\[ jr = r + 1 - jl; \]
\[ \text{if } (jr > 0 \land jr \leq tr) \{ \]
\[ \text{if } (1 \geq n - 1) \ xbar(1); \]
\[ \text{else } {\text{printf}}(\text{"\%s02\%02x", buf, jl}); \]
\[ \text{printf(\"\n\"); } \]
\[ \} \]
\[ \} \]

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.

\[ tl = \text{count}[1], tr = \text{count}[2]; \]
\[ \text{for } (jl = 1; jl \leq r; jl++) \{ \]
\[ jr = r + 1 - jl; \]
\[ \text{if } (jr > 0) \{ \]
\[ \text{if } (jl \leq tl) \{ \]
\[ \text{if } (1 \geq n - 1) \ xbar(1); \]
\[ \text{else } {\text{printf}}(\text{"\%s02\%02x", buf, jl}); \]
\[ \text{printf(\"\n\"); } \]
\[ \} \]
\[ \} \]
\[ \text{if } (jr \leq tr) \{ \]
\[ \text{if } (2 \geq n - 1) \ xbar(2); \]
\[ \text{else } {\text{printf}}(\text{"\%s03\%02x", buf, jr}); \]
\[ \text{printf(\"\n\"); } \]
\[ \} \]

This code is used in section 10.

16. Handle the trivial case directly

\[ tl = \text{count}[1], tr = \text{count}[2]; \]
\[ \text{for } (i = 1; i \leq n; i++) \{ \]
\[ xbar(n - 2 + i); \]
\[ {\text{printf}}(\"\n\"); } \]

This code is used in section 10.
17. \( \langle \text{Output clauses to check } c_j \rangle \equiv \)
   \[
   \begin{align*}
   &\text{\texttt{sprintf}}(\texttt{buf}, \texttt{"%'d\texttt{C}"}, j); \\
   &\text{\texttt{for}} (i = 1; i \leq m; i++) \texttt{sprintf(name[i], \texttt{"%'dx%'d\texttt{d}"}, i, j));} \\
   &\text{\texttt{gen\_clauses(m, c[j])};}
   \end{align*}
   \]
   This code is used in section 8.

18. \( \langle \text{Output clauses to check } a_d \rangle \equiv \)
   \[
   \begin{align*}
   &\text{\texttt{sprintf(buf, \texttt{"%'d\texttt{A}"}, d));} \\
   &\text{\texttt{if}} (m \leq n) \{ \\
   &\quad \text{\texttt{if}} (d \leq m) \{ \\
   &\quad \quad \texttt{for} (i = 1; i \leq d; i++) \texttt{sprintf(name[i], \texttt{"%'dx%'d"}, i, d + 1 - i));} \\
   &\quad \}\text{\texttt{gen\_clauses(m, a[d])};} \\
   &\text{\texttt{else}} \{ \\
   &\quad \texttt{for} (i = 1; i \leq d; i++) \texttt{sprintf(name[i], \texttt{"%'dx%'d"}, i, d + 1 - i));} \\
   &\quad \texttt{gen\_clauses(m, a[d])}; \\
   &\text{\texttt{else}} \{ \\
   &\quad \texttt{for} (t = 1; t \leq m + n - d; t++) \texttt{sprintf(name[t], \texttt{"%'dx%'d"}, d + t - n, n + 1 - t));} \\
   &\quad \texttt{gen\_clauses(m + n - d, a[d])};
   &\}\} \text{\texttt{else}} \{ \\
   &\quad \texttt{if} (d \leq n) \{ \\
   &\quad \quad \texttt{for} (i = 1; i \leq d; i++) \texttt{sprintf(name[i], \texttt{"%'dx%'d"}, i, d + 1 - i));} \\
   &\quad \}\text{\texttt{gen\_clauses(d, a[d])};} \\
   &\text{\texttt{else}} \text{\texttt{if}} (d \leq m) \{ \\
   &\quad \texttt{for} (j = 1; j \leq n; j++) \texttt{sprintf(name[j], \texttt{"%'dx%'d"}, d + 1 - j, j));} \\
   &\quad \text{\texttt{gen\_clauses(n, a[d])};} \\
   &\text{\texttt{else}} \{ \\
   &\quad \texttt{for} (t = 1; t \leq m + n - d; t++) \texttt{sprintf(name[t], \texttt{"%'dx%'d"}, d + t - n, n + 1 - t));} \\
   &\quad \texttt{gen\_clauses(m + n - d, a[d])};
   &\}\}
   &\}
   \]
   This code is used in section 8.
\[\text{Output clauses to check } b_d \quad 19\]

\[
\begin{array}{l}
\text{\texttt{\{ printf(buf, "\%dB", d);}
\text{\enspace if} \ (m \leq n) \quad \text{\{}
\text{\enspace \texttt{if} \ (d \leq m) \quad \text{\{}
\text{\quad \texttt{for} \ (i = 1; \ i \leq d; \ i++) \quad \texttt{sprintf(name[i], "\%dx\%d", i, n + i - d);}
\text{\quad \texttt{gen\_clauses(d, b[d]);}}
\text{\}}
\text{\enspace \texttt{else if} \ (d \leq n) \quad \text{\{}
\text{\quad \texttt{for} \ (i = 1; \ i \leq m; \ i++) \quad \texttt{sprintf(name[i], "\%dx\%d", i, n + i - d);}
\text{\quad \texttt{gen\_clauses(m, b[d]);}}
\text{\}}
\text{\enspace \texttt{else} \quad \text{\{}
\text{\quad \texttt{for} \ (j = 1; \ j \leq m + n - d; \ j++) \quad \texttt{sprintf(name[j], "\%dx\%d", j + d - n, j);}
\text{\quad \texttt{gen\_clauses(m + n - d, b[d]);}}
\text{\}}
\text{\\texttt{else if} \ (d \leq n) \quad \text{\{}
\text{\quad \texttt{for} \ (i = 1; \ i \leq d; \ i++) \quad \texttt{sprintf(name[i], "\%dx\%d", i, n + i - d);}
\text{\quad \texttt{gen\_clauses(d, b[d]);}}
\text{\}}
\text{\enspace \texttt{else if} \ (d \leq m) \quad \text{\{}
\text{\quad \texttt{for} \ (j = 1; \ j \leq n; \ j++) \quad \texttt{sprintf(name[j], "\%dx\%d", j + d - n, j);}
\text{\quad \texttt{gen\_clauses(n, b[d]);}}
\text{\}}
\text{\enspace \texttt{else} \quad \text{\{}
\text{\quad \texttt{for} \ (j = 1; \ j \leq m + n - d; \ j++) \quad \texttt{sprintf(name[j], "\%dx\%d", j + d - n, j);}
\text{\quad \texttt{gen\_clauses(m + n - d, b[d]);}}
\text{\}}
\text{\}\texttt{}}
\end{array}
\]

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.
a: 1*
   "a": 1* 5*
b: 1*
   "bb": 1* 6*
buf: 1* 2* 3* 4* 5* 6* 9* 10, 12, 13, 14, 15, 17, 18, 19.
c: 1*
   "cc": 1* 4*
count: 1* 11, 12, 14.
d: 1*
   "exit": 2* 3* 4* 5* 6* 7* 10, 11.
   "fgets": 2*
   "fprintf": 2* 3* 4* 5* 6* 7* 10, 11.
gen_clauses: 9* 10, 17, 18, 19.
gen_clauses1: 9*
i: 1* 10
   "j": 1* 10
   "jl": 10, 12, 13, 14, 15.
   "jr": 10, 12, 13, 14, 15.
k: 1* 10.
l: 1*
   "ll": 1* 2* 3* 4* 5* 6*
m: 1*
   "main": 1*
   "mmax": 1* 3* 4* 5* 6*
   "name": 1* 9* 12, 17, 18, 19.
   "nmax": 1* 3* 4* 5* 6*
   "nn": 1* 7*
p: 1*
   "printf": 7* 12, 13, 14, 15, 16.
r: 1* 10
   "rr": 1* 3* 9*
sprintf: 9* 17, 18, 19.
stderr: 2* 3* 4* 5* 6* 7* 10, 11.
stdin: 1* 2* 
swap: 10, 12.
t: 1* 10.
   "tl": 10, 12, 13, 14, 15.
tr: 10, 12, 13, 14, 15.
x: 12
   "xbar": 12, 14, 16.
(Build the complete binary tree with $n$ leaves 11) Used in section 10.
(Cases for row, column, and diagonal sums 3*, 4*, 5*, 6*) Used in section 2*.
(Check the data 7*) Used in section 1*.
(Cases for row, column, and diagonal sums 3*, 4*, 5*, 6*) Used in section 2*.
(Generate additional clauses at the root 15) Used in section 10.
(Generate additional clauses for node $i$ 13) Used in section 10.
(Generate the clauses at the root 14) Used in section 10.
(Generate the clauses for node $i$ 12) Used in section 10.
(Handle the trivial case directly 16) Used in section 10.
(Input the data 2*) Used in section 1*.
(Output clauses to check $a_d$ 18) Used in section 8.
(Output clauses to check $b_d$ 19) Used in section 8.
(Output clauses to check $c_j$ 17) Used in section 8.
(Output clauses to check $r_i$ 9*) Used in section 8.
(Output the clauses 8) Used in section 1*.
(Subroutines 10) Used in section 1*.