1*. Intro. Given an input file that contains a partial specification of a Boolean function of N variables, this program generates clauses that are satisfiable if and only if the function has a disjunctive normal form with at most K terms. Parameters N and K are given on the command line.

The main variables are i+j (meaning that term i contains x_j) and i−j (meaning that term i contains $\bar{x}_j$), for $1 \le i \le K$ and $1 \le j \le N$. There also are subsidiary variables i.k for $1 \le i \le K$ and $1 \le k \le T$, if T of the specified function values are true.

For example, the input file

```
101:1
001:0
100:1
111:0
011:1
```

informs us that $f(1,0,1) = 1$, $f(0,0,1) = 0$, …, $f(0,1,1) = 1$; here $N = 3$ and $T = 3$. If we specify $K = 2$, the satisfiability problem will be satisfied, for example, by `1+1`, `1-2`, `2-1`, `2+2`; that is, $f(x_1, x_2, x_3) = x_1\bar{x}_2 \vee \bar{x}_1 x_2$ agrees with the given specifications. [This example is taken from a paper by Kamath, Karmarker, Ramakrishnan, and Resende, *Mathematical Programming* **57** (1992), 215–238, where the problem is introduced and many examples are given.]

The first line of input in the example above generates seven clauses:

`1.1 2.1`	(term 1 or term 2 must be true at 101)
`~1.1 ~1-1`	(if term 1 is true at 101, it doesn't contain $\bar{x}_1$)
`~1.1 ~1+2`	(if term 1 is true at 101, it doesn't contain x_2)
`~1.1 ~1-3`	(if term 1 is true at 101, it doesn't contain $\bar{x}_3$)
`~2.1 ~1-1`	(if term 2 is true at 101, it doesn't contain $\bar{x}_1$)
`~2.1 ~1+2`	(if term 2 is true at 101, it doesn't contain x_2)
`~2.1 ~1-3`	(if term 2 is true at 101, it doesn't contain $\bar{x}_3$)

And the second line generates two:

`1+1 1+2 1-3`	(term 1 is false at 001, so it contains x_1, x_2, or $\bar{x}_3$)
`2+1 2+2 2-3`	(term 2 is false at 001, so it contains x_1, x_2, or $\bar{x}_3$)

In general, a 'true' line in the input generates one clause of size K and NK clauses of size 2; a 'false' line generates K clauses of size N.

```
#define maxn 100    /* we assume that N doesn't exceed this */
#define O "%"    /* used for percent signs in format strings */
#include <stdio.h>
#include <stdlib.h>
  char buf[maxn + 4];
  int K, N, cutoff;    /* command-line parameters */
  int perm_swap[] = {0, 1, 2, 0, 2, 1, 0, 2, 0, 1, 2, 0, 2, 1, 0, 2, 0, 1, 2, 0, 2, 1, 0};
  int perm[] = {1, 2, 3, 4};
  int dat[4][21];
  main(int argc, char *argv[])
  {
    register int i, j, k, t, count;
    ⟨ Process the command line 2* ⟩;
    printf("~␣sat-synth-trunc-kluj␣%d␣%d␣%d\n", N, K, cutoff);
    ⟨ Print 24 solution-excluding lines 6* ⟩;
    t = 0;    /* this many 'true' lines so far */
    for (count = 0; count < cutoff; count++) {
      if (¬fgets(buf, N + 4, stdin)) break;
      ⟨ Generate clauses based on buf 3 ⟩;
    }
  }
```

2*. ⟨ Process the command line 2* ⟩ ≡

```
if (argc ≠ 4 ∨ sscanf(argv[1], ""O"d", &N) ≠ 1 ∨ sscanf(argv[2], ""O"d", &K) ≠ 1 ∨ sscanf(argv[3],
      ""O"d", &cutoff) ≠ 1) {
  fprintf(stderr, "Usage:␣"O"s␣N␣K␣cutoff\n", argv[0]);
  exit(−1);
}
if (N > maxn) {
  fprintf(stderr, "That␣N␣("O"d)␣is␣too␣big␣for␣me,␣I'm␣set␣up␣for␣at␣most␣"O"d!\n", N,
      maxn);
  exit(−2);
}
```

This code is used in section 1*.

3. The buffer should now hold N digits, then colon, digit, `'\n'`, and `'\0'`.

⟨ Generate clauses based on *buf* 3 ⟩ ≡

```
if (buf[N] ≠ ':' ∨ buf[N + 1] < '0' ∨ buf[N + 1] > '1' ∨ buf[N + 2] ≠ '\n' ∨ buf[N + 3])
  fprintf(stderr, "bad␣input␣line␣`"O"s'␣is␣ignored!\n", buf);
else {
  for (k = 0; k < N; k++)
    if (buf[k] < '0' ∨ buf[k] > '1') break;
  if (k < N) fprintf(stderr, "nonbinary␣data␣`"O"s'␣is␣ignored!\n", buf);
  else if (buf[N + 1] ≡ '0') ⟨ Generate clauses for a 'false' line 4 ⟩
  else ⟨ Generate clauses for a 'true' line 5 ⟩;
}
```

This code is used in section 1*.

4. ⟨ Generate clauses for a 'false' line 4 ⟩ ≡

```
{
  for (i = 1; i ≤ K; i++) {
    for (j = 1; j ≤ N; j++) printf("␣"O"d"O"c"O"d", i, buf[j − 1] ≡ '0' ? '+' : '-', j);
    printf("\n");
  }
}
```

This code is used in section 3.

5. ⟨ Generate clauses for a 'true' line 5 ⟩ ≡

```
{
  t++;
  for (i = 1; i ≤ K; i++) printf("␣"O"d."O"d", i, t);
  printf("\n");
  for (i = 1; i ≤ K; i++)
    for (j = 1; j ≤ N; j++)
      printf("~"O"d."O"d␣~"O"d"O"c"O"d\n", i, t, i, buf[j − 1] ≡ '0' ? '+' : '-', j);
}
```

This code is used in section 3.

6* ⟨ Print 24 solution-excluding lines 6* ⟩ ≡

```
dat[0][2] = dat[0][3] = dat[0][10] = −1;      /* $\bar{x}_2\bar{x}_3\bar{x}_{10}$ */
dat[1][6] = dat[1][10] = dat[1][12] = −1;      /* $\bar{x}_6\bar{x}_{10}\bar{x}_{12}$ */
dat[2][8] = 1, dat[2][13] = dat[2][15] = −1;      /* $x_8\bar{x}_{13}\bar{x}_{15}$ */
dat[3][10] = 1, dat[3][8] = dat[3][12] = −1;      /* $\bar{x}_8x_{10}\bar{x}_{12}$ */
for (i = 0; ; i++) {
  for (j = 0; j < 4; j++)
    for (k = 1; k ≤ 20; k++) {
      if (dat[j][k] > 0) printf("␣~"O"d+"O"d␣"O"d-"O"d", perm[j], k, perm[j], k);
      else if (dat[j][k] < 0) printf("␣"O"d+"O"d␣~"O"d-"O"d", perm[j], k, perm[j], k);
      else printf("␣"O"d+"O"d␣"O"d-"O"d", perm[j], k, perm[j], k);
    }
  printf("\n");
  if (i ≡ 23) break;
  j = perm_swap[i];
  k = perm[j], perm[j] = perm[j + 1], perm[j + 1] = k;
}
```

This code is used in section 1*.

7* Index.

The following sections were changed by the change file: 1, 2, 6, 7.

⟨ Generate clauses based on *buf* 3 ⟩ Used in section 1*.
⟨ Generate clauses for a 'false' line 4 ⟩ Used in section 3.
⟨ Generate clauses for a 'true' line 5 ⟩ Used in section 3.
⟨ Print 24 solution-excluding lines 6* ⟩ Used in section 1*.
⟨ Process the command line 2* ⟩ Used in section 1*.

SAT-SYNTH-TRUNC-KLUJ