1. Intro. This program is designed to compose multiplication-skeleton puzzles of a type pioneered by Junya Take. For example, consider his puzzle for the letter `O`, in *Journal of Recreational Mathematics* **38** (2014), 132:

```
       .......
   ×    ......
  ------------
      ........
      OO.....
   ..O..O..
  ...O..O.
  ...O..O
  ------------
  ....OO......
```

Each occurrence of '`O`' should be replaced by some digit d, and each '`.`' should be replaced by a digit $\neq d$. (And no zero should be in a most significant position.) The solution is unique:

```
       2208068
   ×    357029
  ------------
      19872612
      4416136
   15456476
  11040340
  6624204
  ------------
  788344309972
```

But the purpose of this program is not to *solve* such a puzzle! The purpose of this program is to *invent* such a puzzle, namely to find integers x and y whose partial products and final product have digits that match a given binary pattern.

The pattern is given in *stdin* as a set of lines, with asterisks marking the position of the special digit. For example, the '`O`' shape in the puzzle above would be specified thus:

```
.**.
*..*
*..*
*..*
.**.
```

2. The examples above show that zeros in the multiplier will "offset" the shape in different ways. We try all possible offsets, for a given number m of nonzero multiplier digits.

A second parameter, z, specifies the maximum number of zeros in the multiplier. Both m and z are specified on the command line.

```
#define maxdigs 22   /* size of the longest numbers considered, plus 2 */
#define maxdim 8   /* maximum size of pattern */
#define bufsize maxdim + 5
#define maxm 8   /* m must be less than this */
#define o mems++
#define oo mems += 2

#include <stdio.h>
#include <stdlib.h>
  ⟨ Typedefs 6 ⟩;
  int m;   /* the number of nonzero digits in the multiplier */
  int z;   /* the maximum number of zero digits in the multiplier */
  int vbose;   /* level of verbosity */
  char buf[bufsize];   /* buffer used when inputting the shape */
  char rawpat[maxdim][maxdim];   /* pixels of the raw pattern */
  char last[maxdim];   /* positions of the rightmost asterisks */
  int count;   /* this many solutions found */
  unsigned long long nodes;   /* size of the backtrack trees, times 10 */
  int unresolved;   /* this many cases left unresolved */
  unsigned long long mems;   /* memory accesses */
  ⟨ Global variables 11 ⟩;
  ⟨ Subroutines 7 ⟩;
  main(int argc, char *argv[])
  {
    register int d, i, ii, imax, j, jj, k, kk, l, lc, lj, n, t, tt, x, pos, maxl, printed;
    ⟨ Process the command line 3 ⟩;
    ⟨ Input the pattern 4 ⟩;
    ⟨ Build the table of constants 10 ⟩;
    ⟨ Establish the minimum offsets 13 ⟩;
    while (1) {
      ⟨ Create detailed specifications from the pattern 18 ⟩;
      for (d = 0; d < 10; d++) {
        if (vbose) fprintf(stderr, "␣*=%d:\n", d);
        ⟨ Find all solutions for the current offsets and special digit d 20 ⟩;
      }
      ⟨ Advance to the next offset, or break if it needs too many zeros 14 ⟩;
    }
    fprintf(stderr, "Altogether␣%d␣solutions,␣%lld␣nodes,␣%lld␣mems.\n", count, nodes/10, mems);
    if (unresolved) fprintf(stderr, "...␣%d␣cases␣were␣unresolved!\n", unresolved);
  }
```

3. ⟨ Process the command line 3 ⟩ ≡

```
if (argc < 3 ∨ sscanf(argv[1], "%d", &m) ≠ 1 ∨ sscanf(argv[2], "%d", &z) ≠ 1) {
  fprintf(stderr, "Usage:␣%s␣m␣z␣[verbose]␣[extraverbose]␣<␣foo.dots\n", argv[0]);
  exit(−1);
}
if (m < 2 ∨ m ≥ maxm) {
  fprintf(stderr, "m␣should␣be␣between␣2␣and␣%d,␣not␣%d!\n", maxm − 1, m);
  exit(−2);
}
if (m + z > maxdigs − 2) {
  fprintf(stderr, "m+z␣should␣be␣at␣most␣%d,␣not␣%d!\n", maxdigs − 2, m + z);
  exit(−3);
}
vbose = argc − 3;
```

This code is used in section 2.

4. ⟨ Input the pattern 4 ⟩ ≡

```
for (n = k = 0; ; n++) {
  if (¬fgets(buf, bufsize, stdin)) break;
  if (n ≥ maxdim) {
    fprintf(stderr, "Recompile␣me:␣I␣allow␣at␣most␣%d␣lines␣of␣input!\n", maxdim);
    exit(−3);
  }
  ⟨ Input row n of the shape 5 ⟩;
}
fprintf(stderr, "OK,␣I've␣got␣a␣pattern␣with␣%d␣rows␣and␣%d␣asterisks.\n", n, k);
if (m < n − 1) {
  fprintf(stderr, "So␣there␣must␣be␣at␣least␣%d␣multiplier␣digits,␣not␣%d!\n", n − 1, m);
  exit(−2);
}
```

This code is used in section 2.

5. ⟨ Input row n of the shape 5 ⟩ ≡

```
for (j = 0; buf[j] ∧ buf[j] ≠ '\n'; j++) {
  if (buf[j] ≡ '*') {
    if (j ≥ maxdim) {
      fprintf(stderr, "Recompile␣me:␣I␣allow␣at␣most␣%d␣columns␣per␣row!\n", maxdim);
      exit(−5);
    }
    oo, rawpat[n][j] = 1, k++, last[n] = j + 1;
  }
}
```

This code is used in section 4.

6. Bignums. We implement elementary decimal addition on nonnegative integers. Each integer is represented by an array of bytes, in which the first byte specifies the number of significant digits, and the remaining bytes specify the digits themselves (right to left).

⟨ Typedefs 6 ⟩ ≡

```
typedef char bignum [maxdigs];
```

This code is used in section 2.

7. For example, it's easy to test equality of two such bignums, or to copy one to another.

⟨ Subroutines 7 ⟩ ≡

```
int isequal(bignum a, bignum b)
{
  register int la = a[0], i;
  if (oo, la ≠ b[0]) return 0;
  for (i = 1; i ≤ la; i++)
    if (oo, a[i] ≠ b[i]) return 0;
  return 1;
}
void copy(bignum a, bignum b)
{
  register int lb = b[0], i;
  for (o, i = 0; i ≤ lb; i++) oo, a[i] = b[i];
}
```

See also sections 8 and 9.

This code is used in section 2.

8. Here's the basic routine. It's OK to have $a = b$ or $b = c$. (But beware of $a = c$.)

⟨ Subroutines 7 ⟩ +≡

```
void add(bignum a, bignum b, bignum c, int p) {     /* set a = b + 10^p c */
    register int lb = b[0], lc = c[0], i, k, d;
    if (oo, lc ≡ 0) {
      copy(a, b);
      return;
    }
    for (i = 1; i ≤ p ∧ i ≤ lb; i++) oo, a[i] = b[i];
    for (k = 0; i ≤ lb ∨ i ≤ lc + p ∨ k; i++) {
      d = k + (i ≤ lb ? o, b[i] : 0) + (i ≤ lc + p ∧ i > p ? o, c[i − p] : 0);
      if (d ≥ 10) k = 1, d −= 10; else k = 0;
      o, a[i] = d;
    }
    o, a[0] = i − 1; if ( i ≥ maxdigs )
    {
      fprintf(stderr, "Integer␣overflow,␣more␣than␣%d␣digits!\n", maxdigs −1);
      exit(−666);
    }
    if (a[a[0]] ≡ 0) fprintf(stderr, "why?\n");
    }
```

9. ⟨ Subroutines 7 ⟩ +≡

```
void print_bignum(bignum a)
{
  register int i, la = a[0];
  if (¬la) fprintf(stderr, "0");
  else
    for (i = la; i; i−−) fprintf(stderr, "%d", a[i]);
}
```

10. We might as well have a primitive multiplication table.

⟨ Build the table of constants 10 ⟩ ≡

```
o, cnst[0][0] = 0;
for (k = 1; k < 10; k++) oo, cnst[k][0] = 1, cnst[k][1] = k;
for ( ; k ≤ 81; k++) oo, o, cnst[k][0] = 2, cnst[k][2] = k/10, cnst[k][1] = k % 10;
```

This code is used in section 2.

11. ⟨ Global variables 11 ⟩ ≡

```
bignum cnst[82];
```

See also sections 17, 21, and 40.

This code is used in section 2.

12. Offsets and constraints. The kth partial product, for $0 \le k \le m$, will be shifted left by *off* $[k]$. (When $k = m$ this is the entire product, the sum of the shifted partials.) It inherits the constraints of row $k - (m + 1 - n)$ of the n-row pattern in *rawpat*.

The data in *rawpat* appears "left to right," but the constraints on digits are "right to left." I mean, column 0 in *rawpat* refers to the most significant digit that is constrained.

The constraints on a partial product $(\ldots p_2 p_1 p_0)_{10}$ say that $p_i = d$ for certain i, while $p_i \ne d$ for the others. We represent them as a bignum, with 1 in the "d" positions and 0 elsewhere.

For example, the opening problem in the introduction has $m = 5$, $z = 1$, offsets (0, 1, 3, 4, 5), and constraints (0, 1100000, 100100, 10010, 1001, 11000000).

We do not constrain the length of the multiplicand or the partial products; we simply require that any digits to the left of explicitly constrained positions must differ from d. This produces multiple potential puzzles, some of which won't have unique solutions.

13. ⟨ Establish the minimum offsets 13 ⟩ ≡

```
for (i = 0; i < m; i++) o, off[i] = i;
```

This code is used in section 2.

14. The offset table runs through all combinations $s_0 < s_1 < \cdots < s_{m-1}$ with $s_0 = 0$ and $s_{m-1} < m + z$, in lexicographic order.

⟨ Advance to the next offset, or **break** if it needs too many zeros 14 ⟩ ≡

```
for (i = m - 1; i > 0; i--)
  if (o, off[i] < i + z) break;
if (i ≡ 0) break;
o, off[i]++;
for (i++; i < m; i++) oo, off[i] = off[i - 1] + 1;
```

This code is used in section 2.

15. We must choose the position *pos* where column 0 of the raw pattern will appear in the final product. Then column j of the kth partial product will be in position $pos - off[k] - j$.

In the rightmost (smallest) setting of *pos*, at least one of the constraints will end with 1. A harder puzzle is obtained if *pos* exceeds this minimum. This program sets *pos* to the minimum possible, plus a compile-time parameter called *slack*. Junja Take has published several examples with $slack = 1$, and I want to explore such cases; however, the default version of this program sets $slack = 0$.

```
#define slack 0   /* amount to shift the pattern left in harder problems */
```

⟨ Choose *pos* 15 ⟩ ≡

```
for (i = pos = 0; i ≤ m; i++)
  if (oo, off[m + 1 - n + i] + last[i] > pos) pos = off[m + 1 - n + i] + last[i];
pos += slack - 1;
```

This code is used in section 18.

16. Sometimes two constraints are identical, and we'll want to know that fact. So we set up a table called *id*, where $id[j] = id[k]$ if and only if $c_j = c_k$.

⟨ Set up the constraints 16 ⟩ ≡

```
for (k = ids = 0; k ≤ m; k++) {
  o, i = k − (m + 1 − n), constr[k][0] = 0;
  if (i ≥ 0) {
    for (oo, j = pos − off[k] − last[i] + 1; j ≥ 0; j−−) o, constr[k][j] = 0;
    for (o, j = last[i] − 1; j ≥ 0; j−−) {
      if (o, rawpat[i][j]) oo, o, constr[k][pos − off[k] − j + 1] = 1, constr[k][0] = pos − off[k] − j + 1;
      else oo, constr[k][pos − off[k] − j + 1] = 0;
    }
  }
  for (j = k − 1; j ≥ 0; j−−)
    if (oo, isequal(constr[j], constr[k])) break;
  if (j ≥ 0) oo, id[k] = id[j]; else o, id[k] = ids++;
}
```

This code is used in section 18.

17. ⟨ Global variables 11 ⟩ +≡

```
char off[maxm];   /* blanks at right of partial products */
bignum constr[maxm];   /* the constraint patterns, decimalized */
char id[maxm];   /* equivalence class number for a given constraint */
char ids;   /* how many classes are there? */
```

18. ⟨ Create detailed specifications from the pattern 18 ⟩ ≡

```
{
  ⟨ Choose pos 15 ⟩;
  ⟨ Set up the constraints 16 ⟩;
  if (vbose) {
    fprintf(stderr, "Constraints␣for␣offsets");
    for (k = 0; k ≤ m; k++) fprintf(stderr, "␣%d", off[k]);
    fprintf(stderr, ":");
    for (k = 0; k ≤ m; k++) {
      fprintf(stderr, "␣");
      print_bignum(constr[k]);
    }
    fprintf(stderr, ".\n");
  }
}
```

This code is used in section 2.

19. Backtracking. Let the multiplicand be $(a_l \ldots a_2a_1a_0)_{10}$. We proceed by trying all possibilities $\neq d$ for a_0, then all possibilities consistent with a_0 for a_1, and so on. The upper limit on l is **maxdigs** $-2-s_{m-1}$, because of our limit on the size of bignums; but I doubt if we'll often get really big solutions.

(If *slack* > 0, we forbid $a_0 = 0$, because those solutions would have been obtained with lesser *slack*.)

The basic ideas will become clear if we look more closely at the constraints and offsets of our running example, supposing for convenience that $d = 1$. The multiplier is $(b_5b_4b_30b_1b_0)_{10}$, because of the given offsets. The partial products $(p_0, p_1, p_2, p_3, p_4, p_5)$ apply respectively to b_0, b_1, b_3, b_4, b_5, and the grand total. They are supposed to satisfy the constraints (0, 1100000, 100100, 10010, 1001, 11000000), as stated earlier.

Suppose $a_0 = 3$. Then we must have $b_5 = 7$; that's the only way to have p_4 end with 1.

And $b_5 = 7$ implies that b_0, b_1, b_3, b_4 can't be 7: All five constraints are different in this problem, hence no two b's can be equal.

Moving on, if $a_0 = 3$ we cannot have $a_1 = 3$. The reason is that the candidates for multiplier digits are 2 thru 9, and the values of $33k \bmod 100$ for $2 \le k \le 9$ are respectively (66, 99, 32, 65, 98, 31, 64, 97); none of those is suitable for the constraint 10010.

If $a_0 = 3$ and $a_1 = 4$, we must have $b_5 = 7$ and $b_4 = 5$. Furthermore, $a_2 = 4$ will mess up the constraint 1001, because $443 \times 7 = 3101$. The values $a_2 \in \{3, 8, 9\}$ are also impossible, because they yield no multiplier digits for the constraint 100100. Thus a_2 must be 0, 2, or 6.

Proceeding in this way, we're able to rule out most of the potential trailing digits of the multiplicand before exploring very far. When we're choosing suitable values of a_l, we check the least significant l digits of each constraint c_k for $0 \le k < m$; at least one of the eight possible nonzero multiplier digits $\neq d$ must satisfy it. Furthermore, if *exactly* one multiplier digit is valid, we've forced one of the multiplier digits b_i to a particular value.

When sufficiently many multiplier digits are forced, we can begin to enforce the final constraint c_m (i.e., the constraint on the total product). This program does that only if the current number of ways to satisfy the other m constraints individually is less than a certain threshold. Suppose, for example, that $m = 5$ and the current "status" is 33121, meaning that constraints $(c_0, c_1, c_2, c_3, c_4)$ can be individually satisfied in $(3, 3, 1, 2, 1)$ ways. Then we test c_m only if the threshold is 18 or more.

A constraint that is satisfied to infinite precision, not just with respect to the l trailing digits, is said to be *totally* satisfied. Whenever all constraints are totally satisfied, we have a solution.

After a solution is found, we can sometimes extend it by prepending nonzero digits to the multiplicand. For example, we know that $a = 2208068$, $b = 357029$, $d = 4$ leads to a valid puzzle for the `O` pattern; so does $a = 302208068$, $b = 357029$, $d = 4$. The extra prefix '30' doesn't introduce any unwanted 4's into the partial products or the total product.

20. Such considerations lead us to a standard backtracking scheme that takes the following overall form, if we follow the recipe of Algorithm 7.2.2B:

```
⟨ Find all solutions for the current offsets and special digit d 20 ⟩ ≡
  b1: o, maxl = maxdigs −2− off[m − 1];
  l = 0;
  ⟨ Initialize the data structures 22 ⟩;
b2: nodes += 10;
  if (vbose > 1) {
    fprintf(stderr, "Level␣%d,", l);
    ⟨ Print the csize status information 23 ⟩;
  }
  if (l ≥ maxl) ⟨ Check for unusual solutions and goto b5 34 ⟩;
  ⟨ If all constraints are totally satisfied, print a solution 30 ⟩;
  x = 0;
b3: if (slack ∧ l ≡ 0 ∧ x ≡ 0) goto b4;
  if (x ≡ d) goto b4;
  if (vbose > 2) fprintf(stderr, "␣testing␣%d\n", x);
  ⟨ If some constraint can't be satisfied when a_l = x, goto b4 24 ⟩;
  o, a[l] = x;
  if (vbose > 1) fprintf(stderr, "Trying␣a[%d]=%d\n", l, x);
  ⟨ Update the data structures 28 ⟩;
  l = l + 1; goto b2;
b4: if (x ≡ 9) goto b5;
  x = x + 1; goto b3;
b5: l = l − 1;
  if (l ≥ 0) {
    if (vbose > 1) fprintf(stderr, "Back␣to␣level␣%d\n", l);
    o, x = a[l];
    ⟨ Downdate the data structures 29 ⟩;
    goto b4;
  }
```

This code is used in section 2.

21. What data structures will support this computation nicely? First, there's an array of bignums: *ja*$[l][j]$ contains j times the partial multiplier $(a_l \ldots a_0)_{10}$ at a given level. Clearly *ja*$[l][j]$ is *ja*$[l-1][j]$ plus $j \cdot 10^l a_l$. These entries are computed only for values of j that are necessary; *stamp*$[l][j]$ contains the node number at which they were most recently computed (actually it contains *nodes* $+\,x$).

We also maintain arrays called *choice*$[k]$, which list the all nonzero multiplier digits that haven't been ruled out for constraint k. Their sizes at level l are *csize*$[l][k]$. Actually *choice*$[k]$ is a permutation of $\{0, 1, \ldots, 9\}$, and *where*$[k]$ is the inverse permutation; the viable elements at level l are those j with *where*$[k][j] <$ *csize*$[l][k]$. This setup permits easy deletion from the lists while backtracking.

```
⟨ Global variables 11 ⟩ +≡
  bignum ja[maxdigs][10];   /* multiples of the multiplicand */
  unsigned long long stamp[maxdigs][10];   /* when they were computed */
  char choice[maxm][10], where[maxm][10];   /* available multipliers, ranked */
  char csize[maxdigs][maxm];   /* current degree of viability */
  char stack[maxm];   /* constraints that have become uniquely satisfied */
  char stackptr;   /* current size of stack */
  char a[maxdigs];   /* the multiplicand */
  bignum total;   /* grand total when checking for a solution */
```

22. ⟨Initialize the data structures 22⟩ ≡
```
if (d ≡ 0 ∧ off[m − 1] ≥ m) goto b5;      /* forbid zeros in multiplier if d = 0 */
for (i = 0, j = 1; j < 10; j++)
  if (j ≠ d) {
    for (k = 0; k < m; k++) oo, choice[k][i] = j, where[k][j] = i;
    i++;
  }
for (k = 0; k < m; k++) oo, oo, csize[0][k] = i, choice[k][i] = d, where[k][d] = i, where[k][0] = 9;
    /* note that i = 9 if d = 0, otherwise 8 */
```
This code is used in section 20.

23. ⟨Print the *csize* status information 23⟩ ≡
```
for (k = 0; k < m; k++) fprintf(stderr, "%d", csize[l][k]);
fprintf(stderr, "\n");
```
This code is used in section 20.

24. **#define** *thresh* 25

⟨If some constraint can't be satisfied when $a_l = x$, **goto** *b4* 24⟩ ≡
```
for (stackptr = 0, k = m − 1; k ≥ 0; k−−) ⟨If constraint k can't be satisfied when a_l = x, goto b4 25⟩;
while (stackptr) {
  o, k = stack[−−stackptr];
  if (vbose > 2) fprintf(stderr, "␣b%d␣must␣be␣%d\n", off[k], choice[k][0]);
  ⟨Delete choice[k][0] from all constraints ≠ c_k 27⟩;
}
for (o, t = csize[l + 1][0], k = 1; k < m ∧ t ≤ thresh; k++) o, t *= csize[l + 1][k];
if (t ≤ thresh) {
  ⟨Test the overall product constraint c_m 35⟩;
  while (stackptr) {
    o, k = stack[−−stackptr];
    if (vbose > 2) fprintf(stderr, "␣b%d␣has␣to␣be␣%d\n", off[k], choice[k][0]);
    ⟨Delete choice[k][0] from all constraints ≠ c_k 27⟩;
  }
}
```
This code is used in section 20.

25. Now we've come to the heart and soul of the program. As we test each constraint, we also store some data that will be needed on level $l+1$ if we get there.

⟨ If constraint k can't be satisfied when $a_l = x$, **goto** *b4* 25 ⟩ ≡

```
{
  o, imax = csize[l][k];     /* how many multipliers worked in the previous level? */
  for (i = 0; i < imax; i++) {
    o, j = choice[k][i];
    ⟨ If j remains satisfactory when a_l = x, goto jok 26 ⟩;
    if (vbose > 2) fprintf(stderr, "␣c%d␣loses␣option␣%d\n", k, j);
    if (--imax ≡ 0) goto b4;     /* we've lost the last option */
    if (i ≠ imax) oo, oo, oo, choice[k][i] = choice[k][imax], where[k][choice[k][imax]] = i--,
          choice[k][imax] = j, where[k][j] = imax;
        /* swap j into last position (for easy backtracking) */
  jok: continue;
  }
  o, csize[l + 1][k] = imax;
  if (imax ≡ 1 ∧ (o, csize[l][k] ≠ 1)) o, stack[stackptr++] = k;
}
```

This code is used in section 24.

26. We've previously verified constraint k in the least significant l digits, and those digits don't depend on a_l. Thus it suffices to do an "incremental" test, looking only at digit l of the constraint.

⟨ If j remains satisfactory when $a_l = x$, **goto** *jok* 26 ⟩ ≡

```
if (o, stamp[l][j] ≠ nodes + x) {     /* have we already updated ja[l]? */
  o, stamp[l][j] = nodes + x;
  if (l ≡ 0) oo, copy(ja[0][j], cnst[x * j]);
  else oo, add(ja[l][j], ja[l − 1][j], cnst[x * j], l);
}
oo, t = (ja[l][j][0] ≤ l ? 0 : ja[l][j][l + 1]);
o, tt = (constr[k][0] ≤ l ? 0 : o, constr[k][l + 1]);
if ((tt ≡ 1 ∧ t ≡ d) ∨ (tt ≠ 1 ∧ t ≠ d)) goto jok;
```

This code is used in section 25.

27. ⟨ Delete $choice[k][0]$ from all constraints $\neq c_k$ 27 ⟩ ≡

```
for (o, kk = 0, j = choice[k][0]; kk < m; kk++)
  if (oo, id[kk] ≠ id[k]) {
    oo, i = csize[l + 1][kk] − 1, ii = where[kk][j];
    if (ii ≤ i) {
      if (i ≡ 0) goto b4;
      o, csize[l + 1][kk] = i;
      if (i ≡ 1) o, stack[stackptr++] = kk;
      if (ii ≠ i) oo, oo, oo, choice[kk][ii] = choice[kk][i], where[kk][choice[kk][i]] = ii, choice[kk][i] = j,
            where[kk][j] = i;
    }
  }
```

This code is used in section 24.

28. The data structures that I've got don't seem to need any updating (other than what has already been done during the tests), except in one respect: When a zero digit is prepended to the multiplicand, we may have already printed the current solution. Otherwise we haven't.

⟨ Update the data structures 28 ⟩ ≡
  if (x) *printed* $= 0$;

This code is used in section 20.

29. Downdating seems to be completely unnecessary, thanks largely to the *choice* and *csize* mechanism, and the fact that other data is recomputed at each level.

⟨ Downdate the data structures 29 ⟩ ≡

This code is used in section 20.

30. ⟨ If all constraints are totally satisfied, print a solution 30 ⟩ ≡
  if (*printed*) **goto** *nope*; /* we've already printed this guy */
  for $(k = 0;\ k < m;\ k{+}{+})$
    if $(o, \mathit{csize}[l][k] > 1)$ **goto** *nope*;
  for $(k = m - 1;\ k \ge 0;\ k{-}{-})$ ⟨ If constraint c_k isn't totally satisfied, **goto** *nope* 31 ⟩;
  ⟨ If constraint c_m isn't totally satisfied, **goto** *nope* 32 ⟩;
  ⟨ Print a solution 33 ⟩;
  nope:

This code is used in section 20.

31. ⟨ If constraint c_k isn't totally satisfied, **goto** *nope* 31 ⟩ ≡
  {
    $oo, o, j = \mathit{choice}[k][0], lj = ja[l-1][j][0], lc = \mathit{constr}[k][0]$;
    if $(lc > lj)$ **goto** *nope*; /* this is correct even if $d = 0$ */
    for $(i = 1;\ i \le lj;\ i{+}{+})$ {
      $o, t = ja[l-1][j][i], tt = (i \le lc\ ?\ o, \mathit{constr}[k][i] : 0)$;
      if $((t \equiv d \land tt \equiv 0) \lor (t \ne d \land tt \ne 0))$ **goto** *nope*;
    }
  }

This code is used in section 30.

32. ⟨ If constraint c_m isn't totally satisfied, **goto** *nope* 32 ⟩ ≡
  $oo, oo, \mathit{add}(\mathit{total}, ja[l-1][\mathit{choice}[0][0]], ja[l-1][\mathit{choice}[1][0]], \mathit{off}[1])$;
  for $(k = 2;\ k < m;\ k{+}{+})$ $oo, o, \mathit{add}(\mathit{total}, \mathit{total}, ja[l-1][\mathit{choice}[k][0]], \mathit{off}[k])$;
  $o, lj = \mathit{total}[0], lc = \mathit{constr}[m][0]$;
  if $(lc > lj)$ **goto** *nope*; /* this is correct even if $d = 0$ */
  for $(i = 1;\ i \le lj;\ i{+}{+})$ {
    $o, t = \mathit{total}[i], tt = (i \le lc\ ?\ o, \mathit{constr}[m][i] : 0)$;
    if $((t \equiv d \land tt \equiv 0) \lor (t \ne d \land tt \ne 0))$ **goto** *nope*;
  }

This code is used in section 30.

33. When a solution is found, I first print out the lengths of the multiplicand, multiplier, partial products, and total product. (By sorting these lines later, I can distinguish unique solutions.) Then I print the multiplicand, multiplier, d, and the solution number.

⟨ Print a solution 33 ⟩ ≡

```
count++;
for (i = l - 1; a[i] ≡ 0; i--) ;   /* bypass leading zeros of multiplicand */
printf("%d,%d;", i + 1, off[m - 1] + 1);
for (k = 0; k < m; k++) printf("%d|%d,", ja[l - 1][choice[k][0]][0], off[k]);
printf("%d,␣", total[0]);
for ( ; i ≥ 0; i--) printf("%d", a[i]);
printf("␣x␣");
for (k = m - 1, i = off[k]; k ≥ 0; k--, i--) {
  while (i > off[k]) printf("0"), i--;
  printf("%d", choice[k][0]);
}
printf(",d=%d␣(#%d)\n", d, count);
printed = 1;
```

This code is used in section 30.

34. It's conceivable that we've constructed a max-length multiplicand without finding enough obstructions to force all digits of the multiplier. In such cases constraint m (the constraint on the entire product) has probably not yet been fully tested. We should therefore backtrack over all choices of multipliers, in order to be sure that no solutions have been overlooked.

Pathological patterns can make this happen, but I don't think it will occur in the cases that interest me. So I am simply reporting the unusual case here. Then I can follow up later if additional investigations are called for.

(If $a_{l-1}! = 0$, there might exist very long solutions that cannot be tested without exceeding our *maxdigits* precision.)

#define *show_unresolved* 0

⟨ Check for unusual solutions and **goto** *b5* 34 ⟩ ≡

```
{
  for (k = 0; k < m; k++)
    if (o, csize[l][k] > 1) break;
  if (k < m) {
    unresolved++;
    if (o, a[l - 1] ≡ 0 ∨ show_unresolved) {
      fprintf(stderr, "Unresolved␣case␣with␣d=%d␣and␣offsets", d);
      for (k = 0; k < m; k++) fprintf(stderr, "␣%d", off[k]);
      fprintf(stderr, ":\n␣a=...");
      for (k = l - 1; k ≥ 0; k--) fprintf(stderr, "%d", a[k]);
      fprintf(stderr, ",␣status␣");
      for (k = 0; k < m; k++) fprintf(stderr, "%d", csize[l][k]);
      fprintf(stderr, "!\n");
    }
  }
  goto b5;
}
```

This code is used in section 20.

35. An inner loop. When we're testing the "bottom line" constraint c_m, we might need to vary several of the multiplier digits independently. The process is a bit tedious, but straightforward: It's just a loop over all m-tuples that haven't yet been filtered out, and we know that the total number of such m-tuples is *thresh* or less.

The multiplier digit that is subject to constraint c_k is one of the $csize[l+1][k]$ possibilities that appear at the beginning of the list $choice[k]$. So we represent it by an index $g[k]$, meaning that the digit we're trying is $choice[k][g[k]]$.

For every such m-tuple $g_0 g_1 \ldots g_{m-1}$, we check if constraint c_m holds in its rightmost $l+1$ digits. If so, we set bit g_k to 1 in $shadow[k]$, for $0 \le k < m$, thereby indicating that g_k is valid in at least one solution.

After running through all the m-tuples, we can backtrack if no solutions were found. Otherwise the shadows will tell us whether any of the *csize* entries can be lowered.

I could do this step in a fancier way, by working only "incrementally" after having gotten l-digit compliance instead of always working to higher and higher precision. (In such a case I'd have to save the sum of carries from the lower l digits, for use in testing the $(l+1)$st digit incrementally.)

I could also avoid many of the m-tuples by backtracking during this process, because c_m can be tested digit-by-digit as those digits become known.

But I don't think this step will be a bottleneck, so I've opted for simplicity.

```
⟨Test the overall product constraint c_m 35⟩ ≡
  {
    for (k = 0; k < m; k++) o, shadow[k] = 0;
    ⟨Run through all m-tuples g_0 ... g_{m-1} 36⟩;
    if (o, shadow[0] ≡ 0) goto b4;      /* there were no solutions */
    for (k = 0; k < m; k++) {
      if (oo, shadow[k] + 1 ≠ 1 ≪ csize[l + 1][k]) ⟨Remove items from choice[k] 39⟩;
    }
  }
```

This code is used in section 24.

36. ⟨Run through all m-tuples $g_0 \ldots g_{m-1}$ 36⟩ ≡

```
bb1: k = 0;
bb2: if (k ≡ m) ⟨Test compliance with c_m and goto bb5 38⟩;
  g[k] = 0;
bb3: ⟨Set acc[k] to the least significant digits of the kth partial sum 37⟩;
  k++;
  goto bb2;
bb4: oo, g[k]++;
  if (o, g[k] < csize[l + 1][k]) goto bb3;
bb5: k--;
  if (k ≥ 0) goto bb4;
```

This code is used in section 35.

37. ⟨Set $acc[k]$ to the least significant digits of the kth partial sum 37⟩ ≡

```
  oo, o, j = choice[k][g[k]], lj = ja[l][j][0];
  for (i = 0; o, i < off[k]; i++) oo, acc[k][i] = acc[k - 1][i];
  for (ii = 1, kk = 0; i ≤ l; i++, ii++) {
    t = (k > 0 ? o, acc[k - 1][i] + kk : kk);
    if (ii ≤ lj) o, t += ja[l][j][ii];
    if (t ≥ 10) o, acc[k][i] = t - 10, kk = 1; else o, acc[k][i] = t, kk = 0;
  }
```

This code is used in section 36.

38. ⟨ Test compliance with c_m and **goto** *bb5* 38 ⟩ ≡

```
{
  for (o, i = 0, lc = constr[m][0]; i ≤ l; i++) {
    o, t = acc[m − 1][i];
    if (i < lc) o, tt = constr[m][i + 1]; else tt = 0;
    if ((t ≡ d ∧ tt ≡ 0) ∨ (t ≠ d ∧ tt ≠ 0)) goto noncomp;
  }
  if (vbose > 2) {
    fprintf(stderr, "␣ok␣");
    for (k = m − 1; k ≥ 0; k−−) fprintf(stderr, "%d", choice[k][g[k]]);
    fprintf(stderr, "\n");
  }
  for (k = 0; k < m; k++) oo, shadow[k] |= 1 ≪ g[k];
noncomp: goto bb5;
}
```

This code is used in section 36.

39. ⟨ Remove items from *choice*[*k*] 39 ⟩ ≡

```
{
  o, imax = csize[l + 1][k];
  for (i = imax − 1; i ≥ 0; i−−)
    if (o, (shadow[k] & (1 ≪ i)) ≡ 0) {
      o, j = choice[k][i];
      if (vbose > 2) fprintf(stderr, "␣b%d␣ain't␣%d\n", k, j);
      imax−−;
      if (i ≠ imax) oo, oo, oo, choice[k][i] = choice[k][imax], where[k][choice[k][imax]] = i,
          choice[k][imax] = j, where[k][j] = imax;
    }
  o, csize[l + 1][k] = imax;
  if (imax ≡ 1) o, stack[stackptr++] = k;
}
```

This code is used in section 35.

40. ⟨ Global variables 11 ⟩ +≡

```
char acc[maxm][maxdigs];   /* partial sums */
char g[maxm];   /* indices for inner loop */
int shadow[maxm];   /* bits where solutions were found */
```

41. Index.

BACK-SKELETON