1. Intro. Michael Keller suggested a problem that I couldn't stop thinking about, although it is extremely special and unlikely to be mathematically useful or elegant: "Place seven 7s, ..., seven 1s into a 7×7 square so that the 14 rows and columns exhibit all 14 of the integer partitions of 7 into more than one part."

I doubt if there's a solution. But if there is, I guess I want to know. So I'm writing this as fast as I can, using brute force for simplicity wherever possible (and basically throwing efficiency out the door).

[Footnote added after debugging: To my astonishment, there are 30885 solutions! And this program needs less than half an hour to find them all, despite the inefficiencies.]

I break the problem into $\binom{13}{6} = 1716$ subproblems, where each subproblem chooses the partitions for the first six columns; the last column is always assigned to partition 1111111. The rows are, of course, assigned to the remaining seven partitions.

Given such an assignment, I proceed to place the 7s, then the 6s, etc. To place l, I choose a "hard" row or column where the partition has a large part, say p. If that row/col has m empty slots, I loop over the $\binom{m}{p}$ ways to put l's into it. And for every such placement I loop over the $\binom{7l-m}{7-p}$ ways to place the other l's.

Array a holds the current placements. At level l, the row and column partitions for unoccupied cells are specified by arrays *rparts*$[l]$ and *cparts*$[l]$. A partition is a hexadecimal integer $(p_1 \ldots p_7)_{16}$ with $p_1 \geq \cdots \geq p_7 \geq 0$.

```
#define modulus 100   /* print only solutions whose number is a multiple of this */
#define lobits(k) ((1U << (k)) - 1)   /* this works for k < 32 */
#define gosper(b)
        { register x = b, y;
          x = b & -b, y = b + x;
          b = y + (((y ⊕ b)/x) >> 2); }
#include <stdio.h>
#include <stdlib.h>
  int parts[14] = {#6100000, #5200000, #5110000, #4300000, #4210000, #4111000, #3310000, #3220000,
      #3211000, #3111100, #2221000, #2211100, #2111110, #1111111};
  int rparts[8][8], cparts[8][8];
  char a[8][8];   /* the current placements */
  unsigned long long count;
  ⟨Subroutines 3⟩;
  void main(void)
  {
    register int b, i, j, k, bits;
    cparts[7][6] = parts[13];
    for (bits = lobits(6); bits < 1 << 13; ) {
      ⟨Do subproblem bits 2⟩;
      fprintf(stderr, "finished␣subproblem␣%x;␣so␣far␣%lld␣solutions.\n", bits, count);
      gosper(bits);
    }
  }
```

2. ⟨Do subproblem *bits* 2⟩ ≡

```
  for (i = j = k = 0, b = 1 << 12; b; b >>= 1, k++) {
    if (bits & b) cparts[7][j++] = parts[k];   /* partition k goes into a column */
    else rparts[7][i++] = parts[k];   /* partition k goes into a row */
  }
  place(7);
```

This code is used in section 1.

3. The recursive subroutine *place*(*l*) decides where to put all occurrences of the digit *l*. (If $l > 0$, it calls *verify*(*l*), which calls *place*($l - 1$).)

```
⟨ Subroutines 3 ⟩ ≡
  void verify(int l);   /* defined later */
  void place(int l)
  {
    register int b, i, j, k, m, p, max, abits, bbits, thisrow, thiscol = −1;
    if (l ≡ 0) ⟨ Print a solution and return 8 ⟩;
    for (max = i = 0; i < 7; i++)
      if (rparts[l][i] > max) max = rparts[l][i], thisrow = i;
    for (j = 0; j < 7; j++)
      if (cparts[l][j] > max) max = cparts[l][j], thiscol = j;
    if (thiscol ≥ 0) ⟨ Put most of the l's in column thiscol 5 ⟩
    else ⟨ Put most of the l's in row thisrow 4 ⟩;
  }
```

See also section 6.

This code is used in section 1.

4. ⟨ Put most of the *l*'s in row *thisrow* 4 ⟩ ≡

```
  {
    p = max ≫ 24;   /* this many (the largest element of the partition) in thisrow */
    for (m = 0; max; m += max & #f, max ≫= 4) ;   /* m is number of empty cells */
    for (abits = lobits(p); abits < 1 ≪ m; ) {
      for (b = 1, j = 0; j < 7; j++)
        if (¬a[thisrow][j]) {
          if (abits & b) a[thisrow][j] = l;
          b ≪= 1;
        }
      for (bbits = lobits(7 − p); bbits < 1 ≪ (7 * l − m); ) {
        for (b = 1, i = 0; i < 7; i++)
          if (i ≠ thisrow) {
            for (j = 0; j < 7; j++)
              if (¬a[i][j]) {
                if (bbits & b) a[i][j] = l;
                b ≪= 1;
              }
          }
        verify(l);   /* if the current placement isn't invalid, recurse */
        for (i = 0; i < 7; i++)
          if (i ≠ thisrow) {
            for (j = 0; j < 7; j++)
              if (a[i][j] ≡ l) a[i][j] = 0;   /* clean up other rows */
          }
        gosper(bbits);
      }   /* end loop on bbits */
      for (j = 0; j < 7; j++)
        if (a[thisrow][j] ≡ l) a[thisrow][j] = 0;   /* clean up thisrow */
      gosper(abits);
    }   /* end loop on abits */
  }
```

This code is used in section 3.

5. ⟨Put most of the l's in column *thiscol* 5⟩ ≡

```
{
  p = max >> 24;   /* this many (the largest element of the partition) in thiscol */
  for (m = 0; max; m += max & #f, max >>= 4) ;   /* m is number of empty cells */
  for (abits = lobits(p); abits < 1 << m; ) {
    for (b = 1, i = 0; i < 7; i++)
      if (¬a[i][thiscol]) {
        if (abits & b) a[i][thiscol] = l;
        b <<= 1;
      }
    for (bbits = lobits(7 - p); bbits < 1 << (7 * l - m); ) {
      for (b = 1, j = 0; j < 7; j++)
        if (j ≠ thiscol) {
          for (i = 0; i < 7; i++)
            if (¬a[i][j]) {
              if (bbits & b) a[i][j] = l;
              b <<= 1;
            }
        }
      verify(l);   /* if the current placement isn't invalid, recurse */
      for (j = 0; j < 7; j++)
        if (j ≠ thiscol) {
          for (i = 0; i < 7; i++)
            if (a[i][j] ≡ l) a[i][j] = 0;   /* clean up other cols */
        }
      gosper(bbits);
    }   /* end loop on bbits */
    for (i = 0; i < 7; i++)
      if (a[i][thiscol] ≡ l) a[i][thiscol] = 0;   /* clean up thiscol */
    gosper(abits);
  }   /* end loop on abits */
}
```

This code is used in section 3.

6. ⟨ Subroutines 3 ⟩ +≡

```
void verify(int l)
{
  register i, j, k, m, q;
  for (i = 0; i < 7; i++) {      /* we will check row i for inconsistency */
    for (j = k = 0; j < 7; j++)
      if (a[i][j] ≡ l) k++;      /* k occurrences of l */
    m = rparts[l][i];
    if (k > 0) {
      for ( ; m; m >>= 4)
        if ((m & #f) ≡ k) goto rowgotk;
      return;      /* invalid: k isn't one of the parts */
    } else {
      if (m & (lobits(4 * (8 − l)))) return;      /* l parts remain */
    }
  rowgotk: continue;
  }
  for (j = 0; j < 7; j++) {      /* we will check column j for inconsistency */
    for (i = k = 0; i < 7; i++)
      if (a[i][j] ≡ l) k++;      /* k occurrences of l */
    m = cparts[l][j];
    if (k > 0) {
      for ( ; m; m >>= 4)
        if ((m & #f) ≡ k) goto colgotk;
      return;      /* invalid: k isn't one of the parts */
    } else {
      if (m & (lobits(4 * (8 − l)))) return;      /* l parts remain */
    }
  colgotk: continue;
  }
  ⟨ Call place recursively 7 ⟩;
}
```

7. OK, we've verified the placement of the l's, so we can proceed to $l-1$.

⟨ Call *place* recursively 7 ⟩ ≡

```
for (i = 0; i < 7; i++) {      /* we will update row i for the residual partition */
  for (j = k = 0; j < 7; j++)
    if (a[i][j] ≡ l) k++;      /* k occurrences of l */
  if (k > 0) {      /* we must remove part k, which exists */
    for (m = rparts[l][i], q = 24; ((m ≫ q) & #f) ≠ k; q −= 4) ;
    rparts[l − 1][i] = (m & −(1 ≪ (q + 4))) + ((m & lobits(q)) ≪ 4);
  } else rparts[l − 1][i] = rparts[l][i];
}
for (j = 0; j < 7; j++) {      /* we will update column j for the residual partition */
  for (i = k = 0; i < 7; i++)
    if (a[i][j] ≡ l) k++;      /* k occurrences of l */
  if (k > 0) {      /* we must remove part k, which exists */
    for (m = cparts[l][j], q = 24; ((m ≫ q) & #f) ≠ k; q −= 4) ;
    cparts[l − 1][j] = (m & −(1 ≪ (q + 4))) + ((m & lobits(q)) ≪ 4);
  } else cparts[l − 1][j] = cparts[l][j];
}
place(l − 1);
```

This code is used in section 6.

8. ⟨ Print a solution and **return** 8 ⟩ ≡

```
{
  count++;
  if ((count % modulus) ≡ 0) {
    printf("%lld:␣", count);
    for (i = 0; i < 7; i++) {
      for (j = 0; j < 7; j++) printf("%d", a[i][j]);
      printf("%c", i < 6 ? '␣' : '\n');
    }
  }
  return;
}
```

This code is used in section 3.

9. Index.

PERFECT-PARTITION-SQUARE