(See https://cs.stanford.edu/~knuth/programs.html for date.)

1. Intro. Sequence M1805, posets with linear order $1 \ldots n$. Same as upper triangular $n \times n$ Boolean matrices B such that $I \subseteq B^2 \subseteq B$.

I'm in kind of a hurry tonight, so please excuse terseness. This prog was developed from POSET0, which does the cases up to $n = 9$ in a few seconds, but with much calculation repeated unnecessarily. Therefore I reformulated the method using dynamic programming.

We're given a list of pairs (A, w) where A is an $n \times n$ Boolean matrix and w is a positive weight. The problem is to compute the sum of w times the number of upper triangular B such that $I \subseteq B^2 \subseteq B \subseteq \overline{A}$. The solution is to go through that list and generate all first rows of each A, creating a new list with n decreased by 1.

An $n \times n$ upper triangular Boolean matrix is represented in $\binom{n}{2}$ bits, because we don't care about the diagonal. We want to be able to go up to $n = 12$ at least, on my 32-bit computer, so this program is set up to handle multiple precision. It operates in two phases: Double precision for matrices at first, then single precision when n has been reduced. But in the latter case we need double precision for the weight w.

```
#define n 12
#define memsize 36000000   /* should be a multiple of 12 */
#define thresh (1 << 7)   /* boundary between phases; maximum is 1 << 7 */
#define hashsize (1 << 21)   /* should be a power of 2 */

#include <stdio.h>
#include "gb_flip.h"
  typedef unsigned int uint;
  uint mem[memsize];   /* memory pool */
  int *hash[1 << 21];   /* heads of hash lists or direct data pointers */
  short uni[4][256];   /* random bits for universal hashing */
  uint *top, *start, *ostart;   /* key places in mem */
  int offset[1 << (n - 1)];   /* table showing how bits are mapped */
  int t, x, y, z, mask, hmask;
  int count, curn;
  main()
  {
    register int j, k, l, curbits, curaux;
    uint *p;
    ⟨Initialize 2⟩;
    ⟨Do the first phase 5⟩;
    ⟨Do the second phase 7⟩;
  }
```

2. First we initialize the *uni* table, for hashing.

```
⟨Initialize 2⟩ ≡
  gb_init_rand(0);
  for (j = 0; j < 4; j++)
    for (k = 1; k < 256; k++) uni[j][k] = gb_next_rand();
```

See also section 4.

This code is used in section 1.

3. If *thresh* is $1 \ll k$, the *bits* field will contain rows having k or fewer bits, and the *aux* field will contain the longer rows.

4. ⟨ Initialize 2 ⟩ +≡

for $(j = 0, k = 1, l = 2;\ l \le thresh;\ k{+}{+}, l \ll= 1)\ \ offset[l-1] = j, j \mathrel{+}= k;$
for $(j = 0;\ l < 1 \ll n;\ k{+}{+}, l \ll= 1)\ \ offset[l-1] = j, j \mathrel{+}= k;$

5. Data is kept sequentially in *mem*, beginning at *start*; the first available location is *top*. During the first phase the data appears in four-word packets, because we want link fields for hashing.

#**define** $wt(p)\ *p$ /* first word of packet */
#**define** $aux(p)\ *(p+1)$ /* second word of packet */
#**define** $bits(p)\ *(p+2)$ /* third word of packet */
#**define** $link(p)\ *(p+3)$ /* fourth word of packet (phase one only) */

⟨ Do the first phase 5 ⟩ ≡
$start = \&mem[0];$
$wt(start) = 1, aux(start) = bits(start) = 0, link(start) = (\mathbf{uint})\ \Lambda;$
$top = start + 4;$
for $(l = (1 \ll (n-1)) - 1, curn = n;\ l > thresh;\ l \gg= 1, curn{-}{-})$ {
  $hmask = (1 \ll offset[l]) - 1;$
  for $(j = 0;\ j < hashsize;\ j{+}{+})\ \ hash[j] = \Lambda;$
  $count = 0;$
  for $(p = start, start = top;\ p \ne start;\ p = (p \equiv \&mem[memsize - 4]\ ?\ \&mem[0] : p + 4))$ {
    $count{+}{+};$
    $mask = (aux(p) \gg offset[l])\ \&\ l;$
    for $(x = 0;\ x \le l;\ x = ((x \mid mask) + 1)\ \&\ {\sim}mask)$ {
      $curbits = bits(p);$
      $curaux = aux(p);$
      for $(y = x\ \&\ (x+1), t = x \oplus -1;\ y;\ y \mathrel{-}= z)$ {
        $z = y\ \&\ -y;$
        if $(z \le thresh)\ \ curbits \mathrel{|}= (t\ \&\ (z-1)) \ll offset[z-1];$
        else $curaux \mathrel{|}= (t\ \&\ (z-1)) \ll offset[z-1];$
      }
      ⟨ Put *curbits* and *curaux* into the new list with weight *w* 6 ⟩;
    }
  }
  printf ("␣%d␣item%s␣on␣list␣%d;\n", *count*, $count > 1$? "s" : "", *curn*);
}

This code is used in section 1.

6. ⟨ Put *curbits* and *curaux* into the new list with weight w 6 ⟩ ≡

$\{$
  register int h;
  register uint $*q$;
  $curaux$ &= $hmask$;
  $h = uni[0][curbits \,\&\, \text{\#ff}] + uni[1][(curbits \gg 8) \,\&\, \text{\#ff}] + uni[2][(curbits \gg 16) \,\&\, \text{\#ff}] + uni[3][curbits \gg 24]$;
  $h \mathrel{+}= uni[0][curaux \,\&\, \text{\#ff}] + uni[1][(curaux \gg 8) \,\&\, \text{\#ff}] + uni[2][(curaux \gg 16) \,\&\, \text{\#ff}] + uni[3][curaux \gg 24]$;
  h &= $hashsize - 1$;
  for $(q = hash[h];\ q;\ q = (\textbf{uint}\ *)\ link(q))$
    if $(bits(q) \equiv curbits \wedge aux(q) \equiv curaux)$ **goto** *found*;
  $q = top$;
  if $(q \equiv p)$ $\{$
    fprintf (*stderr*, "Sorry,␣I␣need␣more␣memory!\n");
    exit (-1);
  $\}$
  $bits(q) = curbits, aux(q) = curaux, wt(q) = 0$;
  $link(q) = (\textbf{uint})\ hash[h], hash[h] = q$;
  $top = q + 4$;
  if $(top \equiv \&mem[memsize])$ $top = \&mem[0]$;
found: $wt(q) \mathrel{+}= wt(p)$;
$\}$

This code is used in section 5.

7. In the second phase we use the *hash* table as a direct pointer to the data.

⟨ Do the second phase 7 ⟩ ≡
  ⟨ Repack the data into shorter packets 9 ⟩;
  for $(\ ;\ l;\ l \gg= 1, curn\,{-}{-})$ $\{$
    $hmask = (1 \ll offset[l]) - 1$;
    for $(j = 0;\ j \le hmask;\ j{+}{+})$ $hash[j] = \Lambda$;
    $count = 0$;
    for $(p = start, start = top;\ p \ne start;\ p = (p \equiv \&mem[memsize - 3]\ ?\ \&mem[0] : p + 3))$ $\{$
      $count{+}{+}$;
      $mask = (bits(p) \gg offset[l]) \,\&\, l$;
      for $(x = 0;\ x \le l;\ x = ((x \mid mask) + 1) \,\&\, {\sim}mask)$ $\{$
        $curbits = bits(p)$;
        for $(y = x \,\&\, (x + 1), t = x \oplus -1;\ y;\ y \mathrel{-}= z)$ $\{$
          $z = y \,\&\, -y$;
          $curbits \mathrel{|}= (t \,\&\, (z - 1)) \ll offset[z - 1]$;
        $\}$
        ⟨ Put *curbits* into the new list with weight w 8 ⟩;
      $\}$
    $\}$
    printf ("␣%d␣items␣on␣list␣%d;\n", *count*, *curn*);
  $\}$
  printf ("...and␣the␣solution␣for␣%d␣is␣%d%09d.\n", n, $aux(start)$, $wt(start)$);

This code is used in section 1.

8. ⟨Put *curbits* into the new list with weight *w* 8⟩ ≡

```
{
  register uint *q;
  y = curbits & hmask;
  q = hash[y];
  if (¬q) {
    q = top;
    if (q ≡ p) {
      fprintf(stderr, "Sorry,␣I␣need␣more␣memory!\n");
      exit(−2);
    }
    bits(q) = curbits, wt(q) = aux(q) = 0;
    hash[y] = q;
    top = q + 3;
    if (top ≡ &mem[memsize]) top = &mem[0];
  }
  wt(q) += wt(p), aux(q) += aux(p);
  if (wt(q) ≥ 1000000000) aux(q) += 1, wt(q) −= 1000000000;
}
```

This code is used in section 7.

9. ⟨Repack the data into shorter packets 9⟩ ≡

```
ostart = top;
x = (top − mem) % 3;
if (x) top += 3 − x;
for (p = start, start = top; p ≠ ostart; p = (p ≡ &mem[memsize − 4] ? &mem[0] : p + 4)) {
  wt(top) = wt(p), aux(top) = 0, bits(top) = bits(p);
  top += 3;
  if (top ≡ &mem[memsize]) top = &mem[0];
}
```

This code is used in section 7.

10. Index.

POSETS