1. Introduction. This program obediently carries out the wishes of POLYNUM, which has compiled a set of one-byte and four-byte instructions for us to interpret.

But instead of producing high-precision output, it does all its arithmetic modulo a given number $m \le 256$. That trick keeps memory usage small; it allows the user to reconstruct true answers of almost unlimited size just by trying sufficiently many different values of m. And if anybody is concerned about bits getting clobbered by cosmic radiation, they can gain additional confidence in the accuracy by running the calculations for more moduli than strictly necessary.

```
#include <stdio.h>
#include <setjmp.h>
  jmp_buf restart_point;
  ⟨Type definitions 2⟩
  ⟨Global variables 5⟩
  ⟨Subroutines 3⟩
  main(int argc, char *argv[])
  {
    ⟨Local variables 6⟩;
    ⟨Scan the command line 4⟩;
    setjmp(restart_point);    /* longjmp will return here if necessary */
    ⟨Initialize 15⟩;
    ⟨Interpret the instructions in the input 17⟩;
    ⟨Print statistics 25⟩;
    exit(0);
  }
```

2. It is easy to adapt this program to work with counters that occupy either one byte (**unsigned char**), two bytes (**unsigned short**), or four bytes (**unsigned int**), depending on how much memory is available.

Even if we limit ourselves to one-byte counters, exact results of up to 362 bits can be determined. For example, the eleven moduli 256, 253, 251, 247, 245, 243, 241, 239, 233, 229, 227 will suffice to enumerate n-ominoes for $n \le 46$; and the additional modulus 223 will carry us through $n \le 50$. (If some day we have the resources to go even higher, the next moduli to try would be 211, 199, and 197.)

However, the author's experience with the case $n = 47$ showed that the memory space needed for counters in this program was not as precious as the memory space needed for configurations in POLYNUM. Therefore the four-byte moduli $2^{31} = 2147483648$, $2^{31} - 1$ (which is prime), and $2^{31} - 3$ (which equals $5 \cdot 19 \cdot 22605091$) worked out best. Together they reach nearly to 10^{28}, which would actually be large enough to count 49-ominoes.

With a little extra work I could have allowed moduli up to 2^{32}. But I didn't bother, because 2^{31} turned out to be plenty big.

```
#define maxm (1 << 31)    /* the modulus m must not exceed this */
⟨Type definitions 2⟩ ≡
  typedef unsigned int counter;    /* the main data type in our arrays */
```

See also section 11.

This code is used in section 1.

3. The program checks frequently that everything in the input file is legitimate. If not, too bad; the run is terminated (although a debugger can help diagnose nonobvious problems). Extensive checks like this have helped the author to detect errors in the program as well as errors in the input.

```
⟨Subroutines 3⟩ ≡
  void panic(char *mess)
  {
    fprintf(stderr, "%s!\n", mess);
    exit(−1);
  }
```

See also sections 8, 9, 10, and 12.

This code is used in section 1.

4. Several gigabytes of input might be needed, so the input file name will be extended by `.0`, `.1`, . . . , just as in POLYNUM.

Output data suitable for processing by *Mathematica* will be written on a file with the same name as the input but extended by the modulus and `.m`.

```
⟨Scan the command line 4⟩ ≡
  if (argc ≠ 3 ∨ sscanf(argv[2], "%u", &modulus) ≠ 1) {
    fprintf(stderr, "Usage:␣%s␣infilename␣modulus\n", argv[0]);
    exit(−2);
  }
  base_name = argv[1];
  if (modulus < 2 ∨ modulus > maxm) panic("Improper␣modulus");
  m = modulus;
  sprintf(filename, "%.90s-%u.m", base_name, modulus);
  math_file = fopen(filename, "w");
  if (¬math_file) panic("I␣can't␣open␣the␣output␣file");
```

This code is used in section 1.

5.
```
⟨Global variables 5⟩ ≡
  unsigned int modulus;   /* results will discard multiples of this number */
  char *base_name, filename[100];
  FILE *math_file;   /* the output file */
```

See also sections 7, 14, 16, and 28.

This code is used in section 1.

6.
```
⟨Local variables 6⟩ ≡
  register int k;   /* all-purpose index register */
  register unsigned int m;   /* register copy of modulus */
```

See also section 18.

This code is used in section 1.

7. Input. Let's start with the basic routines that are needed to read instructions from the input file(s). As soon as 2^{30} bytes of data have been read from file `foo.0`, we'll turn to file `foo.1`, etc.

```
#define filelength_threshold (1 ≪ 30)   /* should match the corresponding number in POLYNUM */
#define buf_size (1 ≪ 16)   /* should be a divisor of filelength_threshold */

⟨ Global variables 5 ⟩ +≡
  FILE *in_file;   /* the input file */
  union {
    unsigned char buf[buf_size + 10];   /* place for binary input */
    unsigned int foo;   /* force in.buf to be aligned somewhat sensibly */
  } in;
  unsigned char *buf_ptr;   /* our current place in the buffer */
  int bytes_in;   /* the number of bytes seen so far in the current input file */
  unsigned int checksum;   /* a way to help identify bad I/O */
  FILE *ck_file;   /* the checksum file */
  unsigned int checkbuf;   /* a check sum for comparison */
  int file_extension;   /* the number of GGbytes input */
```

8. ⟨ Subroutines 3 ⟩ +≡

```
  void open_it()
  {
    sprintf(filename, "%.90s.%d", base_name, file_extension);
    in_file = fopen(filename, "rb");
    if (¬in_file) {
      fprintf(stderr, "I␣can't␣open␣file␣%s", filename);
      panic("␣for␣input");
    }
    bytes_in = checksum = 0;
  }
```

9. If the check sum is bad, we go back to the beginning. Some incorrect definitions may have been output to *math_file*, but we'll append new definitions that override them.

```
⟨ Subroutines 3 ⟩ +≡
  void close_it()
  {
    if (fread(&checkbuf, sizeof(unsigned int), 1, ck_file) ≠ 1)
      panic("I␣couldn't␣read␣the␣check␣sum");
    if (fclose(in_file) ≠ 0) panic("I␣couldn't␣close␣the␣input␣file");
    printf("[%d␣bytes␣read␣from␣file␣%s,␣checksum␣%u.]\n", bytes_in, filename, checksum);
    if (checkbuf ≠ checksum) {
      printf("Checksum␣mismatch!␣Restarting...\n");
      longjmp(restart_point, 1);
    }
    fflush(stdout);
  }
```

10. My first draft of this program simply used *fread* to input one or three bytes at a time. But that turned out to be incredibly slow on my system, so now I'm doing my own buffering.

The program here uses the fact that six consecutive zero bytes cannot be present in a valid input; thus we need not make a special check for premature end-of-file.

```
#define end_of_buffer &in.buf[buf_size + 4]

⟨ Subroutines 3 ⟩ +≡
  void read_it()
  {
    register int t, k;
    register unsigned int s;
    if (bytes_in ≥ filelength_threshold) {
      if (bytes_in ≠ filelength_threshold) panic("Improper␣buffer␣size");
      close_it();
      file_extension++;
      open_it();
    }
    t = fread(in.buf + 4, sizeof(unsigned char), buf_size, in_file);
    if (t < buf_size) in.buf[t + 4] = in.buf[t + 5] = in.buf[t + 6] = in.buf[t + 7] = in.buf[t + 8] = #81;
        /* will cause sync 1 error if read */
    bytes_in += t;
    for (k = s = 0; k < t; k++) s = (s ≪ 1) + in.buf[k + 4];
    checksum += s;
  }
```

11. A four-byte instruction has the binary form $(0xaaaaaaa)_2$, $(bbbbbbbb)_2$, $(cccccccc)_2$, $(dddddddd)_2$, where $(aaaaaabbbbbbbbccccccccdddddddd)_2$ is a 30-bit address specified in big-endian fashion. If $x = 0$ it means, "This is the new source address s." If $x = 1$ it means, "This is the new target address t."

A one-byte instruction has the binary form $(1ooopppp)_2$, with a 3-bit opcode $(ooo)_2$ and a 4-bit parameter $(pppp)_2$. If the parameter is zero, the following byte is regarded as an 8-bit parameter $(pppppppp)_2$, and it should not be zero. (In that case the "one-byte instruction" actually occupies two bytes.)

In the instruction definitions below, p stands for the parameter, s stands for the current source address, and t stands for the current target address. The slave processor operates on a large array called *count*.

Opcode 0 (*sync*) means, "We have just finished row p." A report is given to the user.
Opcode 1 (*clear*) means, "Set $count[t + j] = 0$ for $0 \le j < p$."
Opcode 2 (*copy*) means, "Set $count[t + j] = count[s + j]$ for $0 \le j < p$."
Opcode 3 (*add*) means, "Set $count[t + j] \mathrel{+}= count[s + j]$ for $0 \le j < p$."
Opcode 4 (*inc_src*) means, "Set $s \mathrel{+}= p$."
Opcode 5 (*dec_src*) means, "Set $s \mathrel{-}= p$."
Opcode 6 (*inc_trg*) means, "Set $t \mathrel{+}= p$."
Opcode 7 (*dec_trg*) means, "Set $t \mathrel{-}= p$."

```
#define targ_bit #40000000    /* specifies t in a four-byte instruction */

⟨ Type definitions 2 ⟩ +≡
  typedef enum {
    sync, clear, copy, add, inc_src, dec_src, inc_trg, dec_trg
  } opcode;
```

12. The *get_inst* routine reads the next instruction from the input and returns the value of its parameter, also storing the opcode in the global variable *op*. Changes to s and t are taken care of automatically, so that *op* is reduced to either *sync*, *clear*, *copy*, or *add*.

```
#define advance_b
        if (++b ≡ end_of_buffer) { read_it( ); b = &in.buf[4]; }

⟨ Subroutines 3 ⟩ +≡
  opcode get_inst( )
  {
    register unsigned char *b = buf_ptr;
    register opcode o;
    register int p;
  restart: advance_b;
    if (¬(*b & #80)) ⟨ Change the source or target address and goto restart 13 ⟩;
    o = (*b ≫ 4) & 7;
    p = *b & #f;
    if (¬p) {
      advance_b;
      p = *b;
      if (¬p) panic("Parameter␣is␣zero");
    }
    switch (o) {
    case inc_src: cur_src += p; goto restart;
    case dec_src: cur_src −= p; goto restart;
    case inc_trg: cur_trg += p; goto restart;
    case dec_trg: cur_trg −= p; goto restart;
    default: op = o;
    }
    if (verbose) {
      if (op ≡ clear) printf("{clear␣%d␣->%d}\n", p, cur_trg);
      else if (op > clear) printf("{%s␣%d␣%d->%d}\n", sym[op], p, cur_src, cur_trg);
    }
    buf_ptr = b;
    return p;
  }
```

13. ⟨ Change the source or target address and **goto** *restart* 13 ⟩ ≡

```
  {
    if (b + 3 ≥ end_of_buffer) {
      *(b − buf_size) = *b, *(b + 1 − buf_size) = *(b + 1), *(b + 2 − buf_size) = *(b + 2);
      read_it( );
      b −= buf_size;
    }
    p = ((*b & #3f) ≪ 24) + (*(b + 1) ≪ 16) + (*(b + 2) ≪ 8) + *(b + 3);
    if (*b & #40) cur_trg = p;
    else cur_src = p;
    b += 3;
    goto restart;
  }
```

This code is used in section 12.

14. ⟨ Global variables 5 ⟩ +≡

```
opcode op;   /* operation code found by get_inst */
int verbose = 0;   /* set nonzero when debugging */
char *sym[4] = {"sync", "clear", "copy", "add"};
int cur_src, cur_trg;   /* current source and target addresses, s and t */
```

15. The first six bytes of the instruction file are, however, special. Byte 0 is the number n of cells in the largest polyominoes being enumerated. When a *sync* is interpreted, POLYSLAVE outputs the current values of $count[j]$ for $1 \le j \le n$.

Byte 1 is the number of the final row. If this number is r, POLYSLAVE will terminate after interpreting the instruction *sync* r.

Bytes 2–5 specify the (big-endian) number of elements in the *count* array.

Initially $s = t = 0$, $count[0] = 1$, and $count[j]$ is assumed to be zero for $1 \le j \le n$.

⟨ Initialize 15 ⟩ ≡

```
sprintf(filename, "%.90s.ck", base_name);
ck_file = fopen(filename, "rb");
if (¬ck_file) panic("I␣can't␣open␣the␣checksum␣file");
open_it();
read_it();
n = in.buf[4];
last_row = in.buf[5];
prev_row = 0;
slave_size = (in.buf[6] ≪ 24) + (in.buf[7] ≪ 16) + (in.buf[8] ≪ 8) + in.buf[9];
buf_ptr = &in.buf[9];
w = n + 2 − last_row;
if (w < 2 ∨ n < w + w − 1 ∨ n > w + w + 126) panic("Bad␣bytes␣at␣the␣beginning");
count = (counter *) calloc(slave_size, sizeof(counter));
if (¬count) panic("I␣couldn't␣allocate␣the␣counter␣array");
count[0] = 1;   /* prime the pump */
cur_src = cur_trg = 0;
scount = (counter *) calloc(n + 1, sizeof(counter));
if (¬scount) panic("I␣couldn't␣allocate␣the␣array␣of␣totals");
```

This code is used in section 1.

16. ⟨ Global variables 5 ⟩ +≡

```
int n;   /* the maximum polyomino size of interest */
int last_row;   /* the row whose end will complete our mission */
int prev_row;   /* the row whose end we've most recently seen */
int w;   /* width of polynominoes being counted (deduced from n and last_row) */
int slave_size;   /* the number of counters in memory */
counter *count;   /* base address of The Big Table */
counter *scount;   /* base address of totals captured at sync commands */
```

17. Servitude. This program is so easy to write, I could even have done it without the use of literate programming. (But of course it wouldn't be nearly as much fun without CWEB.)

⟨ Interpret the instructions in the input 17 ⟩ ≡

```
while (1) {
  p = get_inst();
  if (cur_trg + p > slave_size ∧ op ≥ clear) panic("Target␣address␣out␣of␣range");
  if (cur_src + p > slave_size ∧ op ≥ copy) panic("Source␣address␣out␣of␣range");
  switch (op) {
  case sync: ⟨ Finish a row; goto done if it was the last 22 ⟩; break;
  case clear: ⟨ Clear p counters 19 ⟩; break;
  case copy: ⟨ Copy p counters 20 ⟩; break;
  case add: ⟨ Add p counters 21 ⟩; break;
  }
}
done:
```

This code is used in section 1.

18. ⟨ Local variables 6 ⟩ +≡

```
register int p;   /* parameter of the current instruction */
register unsigned int a;   /* an accumulator for arithmetic */
```

19. ⟨ Clear p counters 19 ⟩ ≡

```
for (k = 0; k < p; k++) count[cur_trg + k] = 0;
```

This code is used in section 17.

20. ⟨ Copy p counters 20 ⟩ ≡

```
for (k = 0; k < p; k++) count[cur_trg + k] = count[cur_src + k];
```

This code is used in section 17.

21. I wonder what kind of machine language code my C compiler is giving me here, but I'm afraid to look.

⟨ Add p counters 21 ⟩ ≡

```
for (k = 0; k < p; k++) {
  a = count[cur_trg + k] + count[cur_src + k];
  if (a ≥ m) a −= m;
  count[cur_trg + k] = a;
}
```

This code is used in section 17.

22. The *sync* instruction, at least, gives me a little chance to be creative, especially with respect to checking the sanity of the source file.

⟨ Finish a row; goto done if it was the last 22 ⟩ ≡

```
⟨ Check that p has the correct value 23 ⟩;
⟨ Output the relevant counters for completed polyominoes 24 ⟩;
for (k = 1; k ≤ n; k++) scount[k] = count[k];
if (p ≡ last_row) goto done;
```

This code is used in section 17.

23. ⟨ Check that p has the correct value 23 ⟩ ≡

```
if (p ≡ 255) ⟨ Go into special shutdown mode 26 ⟩;
if (p ≡ 1) panic("File␣read␣error");   /* see read_it */
if (¬prev_row) {
  if (p ≠ w + 1) panic("Bad␣first␣sync");
} else if (p ≠ prev_row + 1) panic("Out␣of␣sync");
prev_row = p;
```

This code is used in section 22.

24. ⟨ Output the relevant counters for completed polyominoes 24 ⟩ ≡

```
printf("Polyominoes␣that␣span␣%dx%d␣rectangles␣(mod␣%u):\n", p − 1, w, m);
fprintf(math_file, "p[%d,%d,%u]={0", p − 1, w, m);
for (k = 2; k < w + p − 2; k++) fprintf(math_file, ",0");
for ( ; k ≤ n; k++) {
  if (count[k] ≥ scount[k]) a = count[k] − scount[k];
  else a = count[k] + m − scount[k];
  printf("␣%d:%d", a, k);
  fprintf(math_file, ",%d", a);
}
printf("\n");
fflush(stdout);
fprintf(math_file, "}\n");
```

This code is used in section 22.

25. ⟨ Print statistics 25 ⟩ ≡

```
printf("All␣done!␣Final␣totals␣(mod␣%u):\n", m);
for (k = w + w − 1; k ≤ n; k++) {
  printf("␣%d:%d", count[k], k);
}
printf("\n");
close_it();
```

This code is used in section 1.

26. Checkpointing. POLYNUM issues the special command *sync* 255 when it wants to pause for breath and shore up its knowledge. Therefore, if we see that instruction, we must immediately dump all the counters into a temporary file. A special variant of this program is able to read that file and reconstitute all the data, as if there had been no break in the action. (See the change file `polyslave-restart.ch` for details.)

⟨ Go into special shutdown mode 26 ⟩ ≡

```
{
  close_it();
  printf("Checkpoint␣stop:␣After␣processing␣with␣all␣desired␣moduli,\n");
  printf("␣please␣resume␣with␣polynum-restart␣and␣polyslave-restart.\n");
  sprintf(filename, "%.90s-%u.dump", base_name, m);
  out_file = fopen(filename, "wb");
  if (¬out_file) panic("I␣can't␣open␣the␣dump␣file");
  ⟨ Dump all information needed to restart 27 ⟩;
  exit(1);
}
```

This code is used in section 23.

27. ⟨ Dump all information needed to restart 27 ⟩ ≡

```
dump_data[0] = n;
dump_data[1] = w;
dump_data[2] = m;
dump_data[3] = slave_size;
dump_data[4] = prev_row;
if (fwrite(dump_data, sizeof(unsigned int), 5, out_file) ≠ 5)
  panic("Bad␣write␣at␣beginning␣of␣dump");
if (fwrite(scount, sizeof(counter), n + 1, out_file) ≠ n + 1) panic("Couldn't␣dump␣the␣subtotals");
if (fwrite(count, sizeof(counter), slave_size, out_file) ≠ slave_size)
  panic("Couldn't␣dump␣the␣counters");
printf("[%u␣bytes␣written␣on␣file␣%s.]\n", ftell(out_file), filename);
```

This code is used in section 26.

28. ⟨ Global variables 5 ⟩ +≡

```
unsigned int dump_data[5];    /* parameters needed to restart */
FILE *out_file;
```

29. For the record, here are three shell scripts called `nums`, `slaves`, and `slaves-restart`, which were used to run POLYNUM and POLYSLAVE when $n = 47$:

`nums`

```
#!/bin/sh
if [ $# -ne 3 ]; then
  echo "Usage: nums width configs counts"
  exit 255
fi
time polynum 47 $1 $2 $3 /home/tmp/poly47-$1
slaves $1
while [ $? = 1 ]; do
  mv /home/tmp/poly47-$1.dump /home/tmp/poly47-$1.dump~
  time polynum-restart 47 $1 $2 $3 /home/tmp/poly47-$1
  slaves-restart $1
done
```

`slaves`

```
#!/bin/sh
for m in 2147483648 2147483647 2147483645; do
  time polyslave /home/tmp/poly47-$1 $m
done
```

`slaves-restart`

```
#!/bin/sh
for m in 2147483648 2147483647 2147483645; do
  cp /home/tmp/poly47-$1-$m.dump /home/tmp/poly47-$1-$m.dump~
  time polyslave-restart /home/tmp/poly47-$1 $m
done
```

And here is the *Mathematica* script used to convert modular numbers to multiprecise integers:

```
(* for Chinese Remainders, say for example
  chinese[{13,17,19}]
  x=cdecode[{1,2,3}]
and x (= 4031) will satisfy Mod[x,13]=1, Mod[x,17]=2, Mod[x,19]=3 *)
chinese[l_]:=Block[{},chinmod=Apply[Times,l];
  chinlist=Table[(chinmod/l[[k]])PowerMod[chinmod/l[[k]],-1,l[[k]]],
    {k,Length[l]}]]
cdecode[l_]:=Mod[chinlist.l,chinmod]
m=2^31
chinese[{m,m-1,m-3}]
fn[a_,b_]:="poly47-"<>a<>"-"<>ToString[m-b]<>".m"
squash[a_,w_]:=Block[{},Get[fn[a,0]];Get[fn[a,1]];Get[fn[a,3]];
  Do[q[h,w]= cdecode[{p[h,w,m],p[h,w,m-1],p[h,w,m-3]}],{h,w,48-w}];
  Save["poly47-"<>a<>".m",q];
  Clear[q]]
```

30. Index.

POLYSLAVE