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

1. Introduction. This program solves a fairly general kind of sliding block puzzle. Indeed, it emphasizes generality over speed, although it does try to implement breadth-first search on large graphs in a reasonably efficient way. (I plan to write a program based on more advanced techniques later, with this one available for doublechecking the results.) I apologize for not taking time to prepare a fancy user interface; all you'll find here is a shortest-path-to-solution-of-a-sliding-block-puzzle engine.

2. The puzzle can have up to 15 different kinds of pieces, named in hexadecimal from `1` to `f`. These pieces are specified in the standard input file, one line per piece, by giving a rectangular pattern of 0s and 1s, where 0 means 'empty' and 1 means 'occupied'. Rows of the pattern are separated by slashes as in the examples below.

The first line of standard input is special: It should contain the overall board size in the form '*rows* `x` *columns*', followed by any desired commentary (usually the name of the puzzle). This first line is followed by piece definitions of the form '*piecename* `=` *pattern*'.

Two more lines of input should follow the piece definitions, one for the starting configuration and one for the stopping configuration. (I may extend this later to allow several ways to stop.) Each configuration is specified in a shorthand form by telling how to fill in the board, repeatedly naming the piece that occupies the topmost and leftmost yet-unspecified cell, or `0` if that cell is empty, or `x` if that cell is permanently blocked. Trailing zeros may be omitted.

For example, here's how we could specify a strange (but easy to solve) 5×5 puzzle that has four pieces of three kinds:

```
5 x 5 (a silly example)
1 = 111/01
2 = 101/111
3 = 1
1xx200000000033
000xx00033001002
```

The same puzzle can be illustrated more conventionally as follows:

Starting position

1	1	1		
2	1	2	0	0
2	2	2	0	0
0	0	0	0	0
3	3	0	0	0

Stopping position

0	0	0		
0	0	0	3	3
0	0	1	1	1
0	0	2	1	2
0	0	2	2	2

The two '`3`' pieces are indistinguishable from each other. If I had wanted to distinguish them, I would have introduced another piece name, for example by saying '`4 = 1`'.

3. Six different styles of sliding-block moves are supported by this program, and the user should specify the desired style on the command line.

Style 0. Move a single piece one step left, right, up, or down. The newly occupied cells must previously have been empty.

Style 1. Move a single piece one or more steps left, right, up, or down. (This is a sequence of style-0 moves, all applied to the same piece in the same direction, counted as a single move.)

Style 2. Move a single piece one or more steps. (This is a sequence of style-0 moves, all applied to the same piece but not necessarily in the same direction.)

Style 3. Move a subset of pieces one step left, right, up, or down. (This is like style 0, but several pieces may move as if they were a single "superpiece.")

Style 4. Move a subset of pieces one or more steps left, right, up, or down. (This is the superpiece analog of style 1.)

Style 5. Move a subset of pieces one or more steps. (The superpiece analog of style 2.)

The subsets of pieces moved in styles 3, 4, and 5 need not be connected to each other. Indeed, an astute reader will have noticed that our input conventions allow individual pieces to have disconnected components.

The silly puzzle specified above can, for example, be solved in respectively (20, 10, 4, 10, 4, 2) moves of styles (0, 1, 2, 3, 4, 5). Notice that a small change to that puzzle would make certain positions impossible without superpiece moves; thus, superpiece moves are not simply luxuries, they might be necessary when solving certain puzzles.

4. OK, here now is the general outline of the program. There are no surprises yet, except perhaps for the fact that we prepare to make a *longjmp*.

```
#define verbose Verbose   /* avoid a possible 64-bit-pointer glitch in libgb */
#include <stdio.h>
#include <stdlib.h>
#include <setjmp.h>
#include "gb_flip.h"   /* GraphBase random number generator */
  typedef unsigned int uint;
  jmp_buf success_point;
  int style;
  int verbose;
  ⟨ Global variables 6 ⟩
  ⟨ Subroutines 10 ⟩
  main(int argc, char *argv[])
  {
    register int j, k, t;
    volatile int d;
    ⟨ Process the command line 5 ⟩;
    ⟨ Read the puzzle specification; abort if it isn't right 13 ⟩;
    ⟨ Initialize 24 ⟩;
    ⟨ Solve the puzzle 34 ⟩;
  hurray: ⟨ Print the solution 39 ⟩;
  }
```

5. If the style parameter is followed by another parameter on the command line, the second parameter causes verbose output if it is positive, or suppresses the solution details if it is negative.

⟨ Process the command line 5 ⟩ ≡
```
if (¬(argc ≥ 2 ∧ sscanf(argv[1], "%d", &style) ≡ 1 ∧
        (argc ≡ 2 ∨ sscanf(argv[2], "%d", &verbose) ≡ 1))) {
  fprintf(stderr, "Usage: %s style [verbose]\n", argv[0]);
  exit(−1);
}
if (style < 0 ∨ style > 5) {
  fprintf(stderr, "Sorry, the style should be between 0 and 5, not %d!\n", style);
  exit(−2);
}
```
This code is used in section 4.

6. Representing the board. An $r \times c$ board will be represented as an array of $rc+2c+r+1$ entries. The upper left corner corresponds to position $c+1$ in this array; moving up, down, left, or right corresponds to adding $-(c+1)$, $(c+1)$, -1, or 1 to the current board position. Boundary marks appear in the first $c+1$ and last $c+1$ positions, and in positions $c+k(c+1)$ for $1 \le k < r$; these prohibit the pieces from sliding off the edges of the board.

The following code uses the fact that $rc+2c+r+1$ is at most $3m+2$ when $rc \le m$; the maximum occurs when $r=1$ and $c=m$.

```
#define bdry 999999   /* boundary mark */
#define obst 999998   /* permanent obstruction */
#define maxsize 256   /* maximum r × c; should be a multiple of 8 */
#define boardsize (maxsize * 3 + 2)
```

```
⟨ Global variables 6 ⟩ ≡
  int board[boardsize];   /* main board for analyzing configurations */
  int aboard[boardsize];   /* auxiliary board */
  int rows;   /* the number of rows in the board */
  int cols;   /* the number of columns in the board */
  int colsp;   /* cols + 1 */
  int ul, lr;   /* location of upper left and lower right corners in the board */
  int delta[4] = {1, −1};   /* offsets in board for moving right, left, down, up */
```

See also sections 8, 11, 18, 21, 25, 27, 49, and 53.

This code is used in section 4.

7. Every type of piece is specified by a list of board offsets from the piece's topmost/leftmost cell, terminated by zero. For example, the offsets for the piece named `1` in the silly example are $(1, 2, 7, 0)$ because there are five columns. If there had been six columns, the same piece would have had offsets $(1, 2, 8, 0)$.

The following code is executed when a new piece is being defined.

```
#define boardover()
        {
          fprintf(stderr, "Sorry,␣I␣can't␣handle␣that␣large␣a␣board;\n");
          fprintf(stderr, "␣please␣recompile␣me␣with␣more␣maxsize.\n");
          exit(−3);
        }

⟨Compute the offsets for a piece 7⟩ ≡
  {
    register char *p;
    for (t = −1, j = k = 0, p = &buf[4]; ; p++)
      switch (*p) {
      case '1': if (t < 0) t = k; else off[curo++] = k − t;
        if (curo ≥ maxsize) boardover();
      case '0': j++, k++; break;
      case '/': k += colsp − j, j = 0; break;
      case '\n': goto offsets_done;
      default: fprintf(stderr, "Bad␣character␣'%c'␣in␣definition␣of␣piece␣%c!\n", *p, buf[0]);
        exit(−4);
      }
  offsets_done: if (t < 0) {
      fprintf(stderr, "Piece␣%c␣is␣empty!\n", buf[0]); exit(−5);
    }
    off[curo++] = 0;
    if (curo ≥ maxsize) boardover();
  }
```

This code is used in section 15.

8.
```
#define bufsize 1024    /* maximum length of input lines */

⟨Global variables 6⟩ +≡
  int off[maxsize];    /* offset lists for pieces */
  int offstart[16];    /* starting points in the off table */
  int curo;    /* the number of offsets stored so far */
  char buf[bufsize];    /* input buffer */
```

9. A board position is specified by putting block numbers in each occupied cell. The number of blocks on the board might exceed the number of piece types, since different blocks can have the same type; we assign numbers arbitrarily to the blocks.

For example, the *board* array might look like this in the "silly" starting position:

bdry	*bdry*	*bdry*	*bdry*	*bdry*	*bdry*
4	4	4	*obst*	*obst*	*bdry*
3	4	3	0	0	*bdry*
3	3	3	0	0	*bdry*
0	0	0	0	0	*bdry*
2	1	0	0	0	*bdry*
bdry	*bdry*	*bdry*	*bdry*	*bdry*	

Any permutation of the numbers $\{1, 2, 3, 4\}$ would be equally valid.

10. Here is a subroutine that fills the board from a specification in the input buffer. It returns -1 if too many cells are specified, or -2 if an illegal character is found. Otherwise it returns the number of conflicts found, namely the number of cells that were erroneously filled more than once.

```
⟨Subroutines 10⟩ ≡
  int fill_board(int board[], int piece[], int place[])
  {
    register int j, c, k, t;
    register char *p;
    for (j = 0; j < ul; j++) board[j] = bdry;
    for (j = ul; j ≤ lr; j++) board[j] = −1;
    for (j = ul + cols; j ≤ lr; j += colsp) board[j] = bdry;
    for ( ; j ≤ lr + colsp; j++) board[j] = bdry;
    for (p = &buf[0], j = ul, bcount = c = 0; *p ≠ '\n'; p++) {
      while (board[j] ≥ 0)
        if (++j > lr) return −1;
      if (*p ≡ '0') board[j] = t = 0;
      else if (*p ≥ '1' ∧ *p ≤ '9') t = *p − '0';
      else if (*p ≥ 'a' ∧ *p ≤ 'f') t = *p − ('a' − 10);
      else if (*p ≡ 'x') t = 0, board[j] = obst;
      else return −2;
      if (t) {
        bcount++;
        piece[bcount] = t;
        place[bcount] = j;
        board[j] = bcount;
        for (k = offstart[t]; off[k]; k++)
          if (j + off[k] < ul ∨ j + off[k] > lr ∨ board[j + off[k]] ≥ 0) c++;
          else board[j + off[k]] = bcount;
      }
      j++;
    }
    for ( ; j ≤ lr; j++)
      if (board[j] < 0) board[j] = 0;
    return c;
  }
```

See also sections 12, 20, 22, 23, 26, 28, 43, 50, and 55.

This code is used in section 4.

11. ⟨ Global variables 6 ⟩ +≡
int *bcount*; /* the number of blocks on the board */
int *piece*[*maxsize*], *apiece*[*maxsize*]; /* the piece names of each block */
int *place*[*maxsize*], *aplace*[*maxsize*]; /* the topmost/leftmost positions of each block */

12. The next subroutine prints a given board on standard output, in a somewhat readable format that shows connections between adjacent cells of each block. The starting position specified by the “silly” input would, for example, be rendered thus:

```
1-1-1
  |
2 1 2 0 0
|   |
2-2-2 0 0

0 0 0 0 0

3 3 0 0 0
```

#define *cell*(*j*, *k*) *board*[*ul* + (*j*) ∗ *colsp* + *k*]

⟨ Subroutines 10 ⟩ +≡

```
void print_board(int board[], int piece[])
{
  register int j, k;
  for (j = 0; j < rows; j++) {
    for (k = 0; k < cols; k++)
      printf("␣%c", cell(j, k) ≡ cell(j − 1, k) ∧ cell(j, k) ∧ cell(j, k) < obst ? '|' : '␣');
    printf("\n");
    for (k = 0; k < cols; k++)
      if (cell(j, k) < 0) printf("␣?");
      else if (cell(j, k) < obst)
        printf("%c%x", cell(j, k) ≡ cell(j, k − 1) ∧ cell(j, k) ∧ cell(j, k) < obst ? '-' : '␣',
            piece[cell(j, k)]);
      else printf("␣␣");
    printf("\n");
  }
}
```

13. Armed with those routines and subroutines, we’re ready to process the entire input file.

⟨ Read the puzzle specification; abort if it isn’t right 13 ⟩ ≡
⟨ Read the board size 14 ⟩;
⟨ Read the piece specs 15 ⟩;
⟨ Read the starting configuration into *board* 16 ⟩;
⟨ Read the stopping configuration into *aboard* 17 ⟩;

This code is used in section 4.

14. ⟨ Read the board size 14 ⟩ ≡

```
fgets(buf, bufsize, stdin);
if (sscanf(buf, "%d␣x␣%d", &rows, &cols) ≠ 2 ∨ rows ≤ 0 ∨ cols ≤ 0) {
  fprintf(stderr, "Bad␣specification␣of␣rows␣x␣cols!\n");
  exit(−6);
}
if (rows * cols > maxsize) boardover();
colsp = cols + 1;
delta[2] = colsp, delta[3] = −colsp;
ul = colsp;
lr = (rows + 1) * colsp − 2;
```

This code is used in section 13.

15. ⟨ Read the piece specs 15 ⟩ ≡

```
for (j = 1; j < 16; j++) offstart[j] = −1;
while (1) {
  if (¬fgets(buf, bufsize, stdin)) {
    buf[0] = '\n'; break;
  }
  if (buf[0] ≡ '\n') continue;
  if (buf[1] ≠ '␣' ∨ buf[2] ≠ '=' ∨ buf[3] ≠ '␣') break;
  if (buf[0] ≥ '1' ∧ buf[0] ≤ '9') t = buf[0] − '0';
  else if (buf[0] ≥ 'a' ∧ buf[0] ≤ 'f') t = buf[0] − ('a' − 10);
  else {
    printf("Bad␣piece␣name␣(%c)!\n", buf[0]);
    exit(−7);
  }
  if (offstart[t] ≥ 0) printf("Warning:␣Redefinition␣of␣piece␣%c␣is␣being␣ignored.\n", buf[0]);
  else {
    offstart[t] = curo;
    ⟨ Compute the offsets for a piece 7 ⟩;
  }
}
```

This code is used in section 13.

16. ⟨ Read the starting configuration into *board* 16 ⟩ ≡

```
t = fill_board(board, piece, place);
printf("Starting␣configuration:\n");
print_board(board, piece);
if (t) {
  if (t > 0)
    if (t ≡ 1) fprintf(stderr, "Oops,␣you␣filled␣a␣cell␣twice!\n");
    else fprintf(stderr, "Oops,␣you␣overfilled␣%d␣cells!\n", t);
  else fprintf(stderr, "Oops,␣%s!\n", t ≡ −1 ? "your␣board␣wasn't␣big␣enough" :
        "the␣configuration␣contains␣an␣illegal␣character");
  exit(−8);
}
if (bcount ≡ 0) {
  fprintf(stderr, "The␣puzzle␣doesn't␣have␣any␣pieces!\n");
  exit(−9);
}
```

This code is used in section 13.

17. ⟨ Read the stopping configuration into *aboard* 17 ⟩ ≡

```
fgets(buf, bufsize, stdin);
t = fill_board(aboard, apiece, aplace);
printf("\nStopping␣configuration:\n");
print_board(aboard, apiece);
if (t) {
  if (t > 0)
    if (t ≡ 1) fprintf(stderr, "Oops,␣you␣filled␣a␣cell␣twice!\n");
    else fprintf(stderr, "Oops,␣you␣overfilled␣%d␣cells!\n", t);
  else fprintf(stderr, "Oops,␣%s!\n", t ≡ −1 ? "your␣board␣wasn't␣big␣enough" :
        "the␣configuration␣contains␣an␣illegal␣character");
  exit(−10);
}
for (j = 0; j < 16; j++) balance[j] = 0;
for (j = ul; j ≤ lr; j++) {
  if ((board[j] < obst) ≠ (aboard[j] < obst)) {
    fprintf(stderr, "The␣dead␣cells␣(x's)␣are␣in␣different␣places!\n");
    exit(−11);
  }
  if (board[j] < obst) balance[piece[board[j]]]++, balance[apiece[aboard[j]]]−−;
}
for (j = 0; j < 16; j++)
  if (balance[j]) {
    fprintf(stderr, "Wrong␣number␣of␣pieces␣in␣the␣stopping␣configuration!\n");
    exit(−12);
  }
```

This code is used in section 13.

18. ⟨ Global variables 6 ⟩ +≡

```
int balance[16];    /* counters used to ensure that no pieces are lost */
```

19. Breadth-first search. Now we're ready for the heart of the calculation, which is conceptually very simple: If we have found all configurations reachable in fewer than d moves, those reachable in d moves are obtained by making one more move from each of those that are reachable in $d-1$. In other words, we want to proceed essentially as follows.

```
c_1 = starting position;
m_0 = 1;  k = 2;
for (d = 1; ; d++) {
   m_d = k;
   for (j = m_{d-1}; j < m_d; j++)
      for (all positions p reachable in one move from c_j)
         if (p is new) c_k = p,  k++;
   if (m_d ≡ k) break;
}
```

The main problem is to test efficiently whether a given position p is new. For this purpose we can use the fact that moves from configurations at distance $d-1$ always go to configurations at distance $d-2$, $d-1$, or d; therefore we can safely *forget* all configurations c_j for $j < m_{d-2}$ when making the test. This principle significantly reduces the memory requirements.

One convenient way to test newness and to discard stale data rapidly is to use hash chains, ignoring all entries at the end of a chain when their index j becomes less than a given cutoff. In other words, we compute a hash address for each configuration, and we store each configuration with a pointer to the previous one that had the same hash code. Whenever we come to a pointer that is less than m_{d-2}, we can stop looking further in a chain.

20. A configuration is represented internally as a sequence of nybbles that list successive piece names, just as in the shorthand form used for starting and stopping configurations in the input but omitting the `x`'s. For example, the "silly" starting configuration is the hexadecimal number `1200000000033`, which is actually stored as two 32-bit quantities #`12000000` and #`00033000`.

Here's a subroutine that packs a given *board* into its encoded form. It puts 32-bit codes into the *config* array, and returns the number of such codes that were stored.

```
⟨Subroutines 10⟩ +≡
  int pack(int board[], int piece[])
  {
    register int i, j, k, p, s, t;
    for (j = ul; j ≤ lr; j++) xboard[j] = 0;
    for (i = s = 0, p = 28, j = ul, t = bcount; t; j++)
      if (board[j] < obst ∧ ¬xboard[j]) {
        k = piece[board[j]];
        if (k) {
          t--, s += k ≪ p;
          for (k = offstart[k]; off[k]; k++) xboard[j + off[k]] = 1;
        }
        if (¬p) config[i++] = s, s = 0, p = 28;
        else p −= 4;
      }
    if (p ≠ 28) config[i++] = s;
    return i;
  }
```

21. ⟨ Global variables 6 ⟩ +≡

char *xboard*[*boardsize*]; /* places filled ahead of time */
uint *config*[*maxsize*/8]; /* a packed configuration */

22. ⟨ Subroutines 10 ⟩ +≡

```
void print_config(uint config[], int n)
{
  register int j, t;
  for (j = 0; j < n − 1; j++) printf("%08x", config[j]);
  for (t = config[n − 1], j = 8; (t & #f) ≡ 0; j−−) t ≫= 4;
  printf("%0*x", j, t);   /* we omit the trailing zeros */
}
```

23. Conversely, we can reconstruct a board from its packed representation.

⟨ Subroutines 10 ⟩ +≡

```
int unpack(int board[], int piece[], int place[], uint config[])
{
  register int i, j, k, p, s, t;
  for (j = ul; j ≤ lr; j++) xboard[j] = 0;
  for (p = i = 0, j = ul, t = bcount; t; j++)
    if (board[j] < obst ∧ ¬xboard[j]) {
      if (¬p) s = config[i++], p = 28;
      else p −= 4;
      k = (s ≫ p) & #f;
      if (k) {
        board[j] = t, piece[t] = k, place[t] = j;
        for (k = offstart[k]; off[k]; k++) xboard[j + off[k]] = 1, board[j + off[k]] = t;
        t−−;
      } else board[j] = 0;
    }
  for ( ; j ≤ lr; j++)
    if (board[j] < obst ∧ ¬xboard[j]) board[j] = 0;
  return i;
}
```

24. We use "universal hashing" to compute hash codes, xoring random bits based on individual bytes. These random bits appear in tables called *uni*.

⟨ Initialize 24 ⟩ ≡

```
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 30.

This code is used in section 4.

25. The number of hash chains, *hashsize*, should be a power of 2, and it deserves to be chosen somewhat carefully. If it is too large, we'll interfere with our machine's cache memory; if it is too small, we'll spend too much time going through hash chains. At present I've decided to assume that *hashsize* is at most 2^{16}, so that the *uni* table entries are **short** (16-bit) quantities.

The total number of configurations might be huge, so I allow 64 bits for the main hash table pointers. (Programmers in future years will chuckle when they read this code, having blissfully forgotten the olden days when people like me had to fuss over 32-bit numbers.)

```
#define hashsize (1 ≪ 13)    /* should be a power of two, not more than 1 ≪ 16 */
#define hashcode(x) (uni[0][x & #ff] + uni[1][(x ≫ 8) & #ff] + uni[2][(x ≫ 16) & #ff] + uni[3][x ≫ 24])
```

⟨ Global variables 6 ⟩ +≡

```
  short uni[4][256];    /* bits for universal hashing */
  uint hash[hashsize];    /* hash table pointers (low half) */
  uint hashh[hashsize];    /* hash table pointers (high half) */
```

26. ⟨ Subroutines 10 ⟩ +≡

```
  void print_big(uint hi, uint lo)
  {
    printf("%.15g", ((double) hi) * 4294967296.0 + (double) lo);
  }
  void print_bigx(uint hi, uint lo)
  {
    if (hi) printf("%x%08x", hi, lo);
    else printf("%x", lo);
  }
```

27. Of course I don't expect to keep all configurations in memory simultaneously, except on simple problems. Instead, I keep a table of *memsize* integers, containing variable-size packets that represent individual configurations. An address into this table is conceptually a 64-bit number, but we actually use the address mod *memsize* because stale data is discarded. The value of *memsize* is a power of 2 so that this reduction is efficient.

The first word of a packet is a pointer to the previous packet having the same hash code. This pointer is *relative* to the current packet, so that it needs to contain only 32 bits at most.

The second word of a packet p is a (relative) pointer to the configuration from which p was derived. This word could be omitted in the interests of space, but it is handy if we want to see an actual solution to the puzzle instead of merely knowing the optimum number of moves.

The remaining words of a packet are the packed encoding of a configuration. If the packet begins near the end of the *pos* array, it actually extends past *pos*[*memsize*]; enough extra space has been provided there to avoid any need for wrapping packets around the *memsize* boundary.

#**define** *memsize* $(1 \ll 25)$ /* space for the configurations we need to know about */
#**define** *maxmoves* 1000 /* upper bound on path length */

⟨ Global variables 6 ⟩ +≡
uint $pos[memsize + maxsize/8 + 1]$; /* currently known configurations */
uint *cutoff*; /* pointer below which we needn't search (low half) */
uint *cutoffh*; /* pointer below which we needn't search (high half) */
uint *curpos*; /* pointer to first unused configuration slot (low half) */
uint *curposh*; /* pointer to first unused configuration slot (high half) */
uint *source*; /* pointer to the configuration we're moving from (low half) */
uint *sourceh*; /* pointer to the configuration we're moving from (high half) */
uint *nextsource*, *nextsourceh*; /* next values of *source* and *sourceh* */
uint *maxpos*; /* pointer to first unusable configuration slot (low half) */
uint *maxposh*; /* pointer to first unusable configuration slot (high half) */
uint *configs*; /* total number of configurations so far (low half) */
uint *configsh*; /* total number of configurations so far (high half) */
uint *oldconfigs*; /* value of *configs* when we began working at distance d */
uint $milestone[maxmoves]$; /* value of *curpos* at various distances */
uint $milestoneh[maxmoves]$; /* value of *curposh* at various distances */
uint *shortcut*; /* $milestone[d] - cutoff$ */
int *goalhash*; /* hash code for the stopping position */
uint $goal[maxsize/8]$; /* packed version of the stopping position */
uint $start[maxsize/8]$; /* packed version of the starting position */

28. The *hashin* subroutine looks for a given *board* configuration in the master table, inserting it if it is new. The value returned is 0 unless the *trick* parameter is nonzero. In the latter case, which is used for moves of style 2 or style 5, special processing needs to be done; we'll explain it later.

```
⟨Subroutines 10⟩ +≡
  int hashin(int trick)
  {
    register int h, j, k, n, bound;
    n = pack(board, piece);
    for (h = hashcode(config[0]), j = 1; j < n; j++) h ^= hashcode(config[j]);
    h &= hashsize − 1;
    if (hashh[h] ≡ cutoffh) {
      if (hash[h] < cutoff) goto newguy;
    } else if (hashh[h] < cutoffh) goto newguy;
    bound = hash[h] − cutoff;
    for (j = hash[h] & (memsize − 1); ; j = (j − pos[j]) & (memsize − 1)) {
      for (k = 0; k < n; k++)
        if (config[k] ≠ pos[j + 2 + k]) goto nope;
      if (trick) ⟨Handle the tricky case and return 44⟩;
      return 0;
    nope: bound −= pos[j];
      if (bound < 0) break;
    }
  newguy: ⟨Insert config into the pos table 31⟩;
    if (h ≡ goalhash) ⟨Test if config equals the goal 29⟩;
    return trick;
  }
```

29. If the current configuration achieves the goal, *hashin* happily terminates the search process, and sends control immediately to the external label called '*hurray*'.

```
⟨Test if config equals the goal 29⟩ ≡
  {
    for (k = 0; k < n; k++)
      if (config[k] ≠ goal[k]) goto not_yet;
    longjmp(success_point, 1);
  not_yet: ;
  }
```

This code is used in section 28.

30. ⟨Initialize 24⟩ +≡

```
  if (setjmp(success_point)) goto hurray;    /* get ready for longjmp */
```

31. ⟨ Insert *config* into the *pos* table 31 ⟩ ≡

$j = curpos \mathbin{\&} (memsize - 1);$
$pos[j] = curpos - hash[h];$
if $(pos[j] > memsize \lor curposh > hashh[h] + (pos[j] > curpos))$ $pos[j] = memsize;$
/* relative link that exceeds all cutoffs */
$pos[j+1] = curpos - source;$ /* relative link to previous position */
for $(k = 0;\ k < n;\ k{+}{+})$ $pos[j+2+k] = config[k];$
$hash[h] = curpos, hashh[h] = curposh;$
⟨ Update *configs* 32 ⟩;
⟨ Update *curpos* 33 ⟩;

This code is used in section 28.

32. When we encounter a new configuration, we print it if it's the first to be found at the current distance, or if *verbose* is set.

⟨ Update *configs* 32 ⟩ ≡

```
if (configs ≡ oldconfigs ∨ verbose > 0) {
  print_config(config, n);
  if (verbose > 0) {
    printf("␣(");
    print_big(configsh, configs);
    printf("=#");
    print_bigx(curposh, curpos);
    printf(",␣from␣#");
    print_bigx(sourceh, source);
    printf(")\n");
  }
}
configs++;
if (configs ≡ 0) configsh++;
```

This code is used in section 31.

33. ⟨ Update *curpos* 33 ⟩ ≡

```
curpos += n + 2;
if (curpos < n + 2) curposh++;
if ((curpos & (memsize − 1)) < n + 2) curpos &= −memsize;
if (curposh ≡ maxposh) {
  if (curpos ≤ maxpos) goto okay;
} else if (curposh < maxposh) goto okay;
fprintf(stderr, "Sorry,␣my␣memsize␣isn't␣big␣enough␣for␣this␣puzzle.\n");
exit(−13);
okay:
```

This code is used in section 31.

34. So now we know how to deal with configurations, and we're ready to carry out our overall search plan.

```
⟨ Solve the puzzle 34 ⟩ ≡
  printf("\n(using␣moves␣of␣style␣%d)\n", style);
  ⟨ Remember the starting configuration 36 ⟩;
restart: ⟨ Remember the stopping configuration 35 ⟩;
  ⟨ Put the starting configuration into pos 37 ⟩;
  for (d = 1; d < maxmoves; d++) {
    printf("***␣Distance␣%d:\n", d);
    milestone[d] = curpos, milestoneh[d] = curposh;
    oldconfigs = configs;
    ⟨ Generate all positions at distance d 38 ⟩;
    if (configs ≡ oldconfigs) exit(0);    /* no solution */
    if (verbose ≤ 0) printf("␣and␣%d␣more.\n", configs − oldconfigs − 1);
  }
  printf("No␣solution␣found␣yet␣(maxmoves=%d)!\n", maxmoves);
  exit(0);
```

This code is used in section 4.

35. ⟨ Remember the stopping configuration 35 ⟩ ≡

```
  t = pack(aboard, apiece);
  for (k = goalhash = 0; k < t; k++) goal[k] = config[k], goalhash ⊕= hashcode(config[k]);
  goalhash &= hashsize − 1;
```

This code is used in section 34.

36. We might need to return to the starting position when reconstructing a solution.

```
⟨ Remember the starting configuration 36 ⟩ ≡
  t = pack(board, piece);
  for (k = 0; k < t; k++) start[k] = config[k];
```

This code is used in section 34.

37. ⟨ Put the starting configuration into *pos* 37 ⟩ ≡

```
  curpos = cutoff = milestone[0] = 1, curposh = cutoffh = milestoneh[0] = 0;
  source = sourceh = configs = configsh = oldconfigs = d = 0;
  maxposh = 1;
  printf("***␣Distance␣0:\n");
  hashin(0);
  if (verbose ≤ 0) printf(".\n");
```

This code is used in section 34.

38. ⟨ Generate all positions at distance *d* 38 ⟩ ≡

```
  if (d > 1) cutoff = milestone[d − 2], cutoffh = milestoneh[d − 2];
  shortcut = curpos − cutoff;
  maxpos = cutoff + memsize, maxposh = cutoffh + (maxpos < memsize);
  for (source = milestone[d−1], sourceh = milestoneh[d−1]; source ≠ milestone[d] ∨ sourceh ≠ milestoneh[d];
        source = nextsource, sourceh = nextsourceh) {
    j = unpack(board, piece, place, &pos[(source & (memsize − 1)) + 2]) + 2;
    nextsource = source + j, nextsourceh = sourceh + (nextsource < j);
    if ((nextsource & (memsize − 1)) < j) nextsource &= −memsize;
    ⟨ Hash in every move from board 42 ⟩;
  }
```

This code is used in section 34.

39. The answer. We've found a solution in d moves.

⟨ Print the solution 39 ⟩ ≡
```
if (d ≡ 0) {
  printf("\nYou're␣joking:␣That␣puzzle␣is␣solved␣in␣zero␣moves!\n");
  exit(0);
}
printf("...␣Solution!\n");
if (verbose < 0) exit(0);
⟨ Print all of the key moves that survive in pos; exit if done 40 ⟩;
⟨ Apologize for lack of memory and go back to square one with reduced problem 41 ⟩;
```
This code is used in section 4.

40. Going backward, we can reconstruct the winning line, as long as the data appears in the top *memsize* positions of our configuration list.

⟨ Print all of the key moves that survive in *pos*; *exit* if done 40 ⟩ ≡
```
if (curposh ∨ curpos > memsize) {
  maxpos = curpos − memsize;
  maxposh = curposh − (maxpos > curpos);
} else maxpos = maxposh = 0;
for (j = 0; j ≤ lr + colsp; j++) aboard[j] = board[j];
while (sourceh > maxposh ∨ (sourceh ≡ maxposh ∧ source ≥ maxpos)) {
  d−−;
  if (d ≡ 0) exit(0);
  printf("\n%d:\n", d);
  k = source & (memsize − 1);
  unpack(aboard, apiece, aplace, &pos[k + 2]);
  print_board(aboard, apiece);
  if (source < pos[k + 1]) sourceh−−;
  source = source − pos[k + 1];
}
```
This code is used in section 39.

41. ⟨ Apologize for lack of memory and go back to square one with reduced problem 41 ⟩ ≡
```
printf("(Unfortunately␣I've␣forgotten␣how␣to␣get␣to␣level␣%d,\n", d);
printf("␣so␣I'll␣have␣to␣reconstruct␣that␣part.␣Please␣bear␣with␣me.)\n");
for (j = 0; j < hashsize; j++) hash[j] = hashh[j] = 0;
unpack(board, piece, place, start);
goto restart;
```
This code is used in section 39.

42. Moving. The last thing we need to do is actually slide the blocks. It seems simple, but the task can be tricky when we get into moves of high-order styles.

⟨Hash in every move from *board* 42⟩ ≡
```
if (style < 3)
  for (j = 0; j < 4; j++)
    for (k = 1; k ≤ bcount; k++) move(k, delta[j], delta[j]);
else ⟨Try all supermoves 57⟩;
```
This code is used in section 38.

43. In the *move* subroutine, parameter k is a block number, parameter *del* is a displacement, and parameter *delo* is such that we've recently considered a board with displacement $del - delo$.

⟨Subroutines 10⟩ +≡
```
void move(int k, int del, int delo)
{
  register int j, s, t;
  s = place[k], t = piece[k];
  for (j = offstart[t]; ; j++) {    /* we remove the piece */
    board[s + off[j]] = 0;
    if (¬off[j]) break;
  }
  for (j = offstart[t]; ; j++) {    /* we test if it fits in new position */
    if (board[s + del + off[j]]) goto illegal;
    if (¬off[j]) break;
  }
  for (j = offstart[t]; ; j++) {    /* if so, we move it */
    board[s + del + off[j]] = k;
    if (¬off[j]) break;
  }
  if (hashin(style ≡ 2) ∨ style ≡ 1) ⟨Unmove the piece and recurse 45⟩
  else {
    for (j = offstart[t]; ; j++) {    /* remove the shifted piece */
      board[s + del + off[j]] = 0;
      if (¬off[j]) break;
    }
  illegal:
    for (j = offstart[t]; ; j++) {    /* replace the unshifted piece */
      board[s + off[j]] = k;
      if (¬off[j]) break;
    }
  }
}
```

44. Style 1 is straightforward: We keep moving in direction *delo* until we bump into an obstacle. But style 2 is more subtle, because we need to explore all reachable possibilities. I thank Gary McDonald for pointing out a serious blunder in my first attempt to find all of the style-2 moves.

The basic idea we use, to find all configurations that are reachable by moving a single piece any number of times, is the well-known technique of depth-first search. But there's a twist, because such a sequence of moves might go through configurations that already exist in the hash table; we can't simply stop searching when we encounter an old configuration. For example, consider the starting board `0102`, from which we can reach `0120` or `0012` or `1002` in a single move. A second move, from `0120`, leads to `1020`. And then when we're considering possible second moves from `1002`, we dare not stop at the "already seen" `1020`, lest we fail to discover the move to `1200`.

We can, however, argue that every valid style-2 move at distance d can be reached by a path that begins at distance $d-1$ and stays entirely at distance d after the first step. (The shortest path to that move clearly has this property.)

Suppose we're exploring the style-2 moves at distance d that are successors of configuration α at distance $d-1$. If we encounter some configuration β that has already been seen, there are two cases: The predecessor of β might be α, or it might be some other configuration, α'. In the former case, we needn't explore any further past β, because the depth-first search procedure will already have been there and done that. (Only one piece has moved, when changing from α to β, so it must be the same as the piece we're currently trying to move.) On the other hand if $\alpha \neq \alpha'$, the example above shows that we need to look past β into potentially unknown territory, or we might miss some legal moves from α. In this second case we need a way to avoid encountering β again and again, endlessly.

To resolve this dilemma without adding additional "mark bits" to the data structure, we will *rename* the predecessor of β, by changing it from α' to α. This change is legitimate, since β is reachable in one move from both α' and α, which both are at distance $d-1$. Then if we encounter β again, we won't have to reconsider it; infinite looping will be impossible.

This strategy tells us how to implement the unfinished "tricky" part of the *hashin* routine. When the following code is encountered, we've just found a known configuration β that begins at j in the *pos* array.

⟨Handle the tricky case and **return** 44⟩ ≡

```
{
  if (bound < shortcut) return 0;    /* return if β not at distance d */
  n = (j − source) & (memsize − 1);    /* find the distance from β to α */
  if (pos[j + 1] ≡ n) return 0;    /* return if α preceded β */
  pos[j + 1] = n;    /* otherwise make α precede β */
  return 1;    /* and continue the depth-first search */
}
```

This code is used in section 28.

45. Local variables s and t need not be preserved across the recursive call in this part of the *move* routine. (I don't expect a typical compiler to recognize that fact; but maybe I underestimate the current state of compiler technology.)

⟨ Unmove the piece and recurse 45 ⟩ ≡

```
{
  for (j = offstart[t]; ; j++) {      /* remove the shifted piece */
    board[s + del + off[j]] = 0;
    if (¬off[j]) break;
  }
  for (j = offstart[t]; ; j++) {      /* replace the unshifted piece */
    board[s + off[j]] = k;
    if (¬off[j]) break;
  }
  if (style ≡ 1) move(k, del + delo, delo);
  else
    for (j = 0; j < 4; j++)
      if (delta[j] ≠ −delo) move(k, del + delta[j], delta[j]);
}
```

This code is used in section 43.

46. Supermoving. The remaining job is the most interesting one: How should we deal with the possibility of sliding several blocks simultaneously?

A puzzle with m blocks has $2^m - 1$ potential superpieces, and one can easily construct examples in which that upper limit is achieved. Fortunately, however, reasonable puzzles have only a reasonable number of superpiece moves; our job is to avoid examining unnecessary cases. The following algorithm is sort of a cute way to do that.

First, we prepare for future calculations by making *aboard* an edited copy of *board*. In the process, we change *bdry* and *obst* items to zero, considering the zeros now to be a special kind of "stuck" block, and we link together all cells belonging to each block. This linking will be more efficient than the offset-oriented method used before.

⟨ Copy and link the *board* 46 ⟩ ≡

```
for (j = 0; j ≤ bcount; j++) head[j] = −1;
for (j = 0; j ≤ lr + colsp; j++) {
  k = board[j];
  if (k) {
    if (k ≥ obst) k = 0;
    aboard[j] = k;
    link[j] = head[k];
    head[k] = j;
  } else aboard[j] = −1;
}
```

This code is used in section 57.

47. Elementary graph theory helps now.

Consider the digraph whose vertices are blocks, with arcs $u \to v$ whenever u would bump into v when block u is shifted by a given amount. The superpieces are *ideals* of this graph, namely they have the property that if u is in the superpiece and $u \to v$ then v is also in the superpiece. Indeed, every ideal that is nonempty and does not contain the stuck block is a superpiece, and conversely. So the problem that faces us is equivalent to generating all such ideals in a given digraph.

The complement of an ideal is an ideal of the dual digraph (the digraph in which arcs are reversed). And the digraph for sliding left is the dual of the digraph for sliding right. So the problem of generating all superpieces for left/right slides is equivalent to generating all ideals of the digraph that corresponds to moving from $k-1$ to k. If such an ideal doesn't contain the stuck block, it defines a superpiece for sliding right; otherwise its complement defines a superpiece for sliding left.

We can construct that digraph by running through the links just made: After the following code has been executed, the arcs leading from u will be to $aboard[l]$, $aboard[l']$, $aboard[l'']$, etc., where $l = out[u]$, $l' = olink[l]$, $l'' = olink[l']$, etc.; the arcs leading into u will be similar, with *in* and *ilink* instead of *out* and *olink*.

⟨ Construct the digraph for $del = 1$ 47 ⟩ ≡

```
for (j = 0; j ≤ bcount; j++) out[j] = in[j] = −1;
for (j = 0; j ≤ bcount; j++)
  for (k = head[j]; k ≥ ul; k = link[k]) {      /* aboard[k] = j */
    t = aboard[k − 1];
    if (t ≠ j ∧ t ≥ 0 ∧ (out[t] < 0 ∨ aboard[out[t]] ≠ j)) {
      olink[k] = out[t], out[t] = k;
      ilink[k − 1] = in[j], in[j] = k − 1;
    }
  }
```

This code is used in section 57.

48. And the problem of generating all superpieces for up/down slides is equivalent to generating all ideals of a very similar digraph.

⟨ Construct the digraph for *del* = *colsp* 48 ⟩ ≡

```
for (j = 0; j ≤ bcount; j++) out[j] = in[j] = −1;
for (j = 0; j ≤ bcount; j++)
  for (k = head[j]; k ≥ ul; k = link[k]) {    /* aboard[k] = j */
    t = aboard[k − colsp];
    if (t ≠ j ∧ t ≥ 0 ∧ (out[t] < 0 ∨ aboard[out[t]] ≠ j)) {
      olink[k] = out[t], out[t] = k;
      ilink[k − colsp] = in[j], in[j] = k − colsp;
    }
  }
```

This code is used in section 57.

49. ⟨ Global variables 6 ⟩ +≡

```
int head[maxsize + 1], out[maxsize + 1], in[maxsize + 1];    /* list heads */
int link[boardsize], olink[boardsize], ilink[boardsize];    /* links */
```

50. The following subroutine for ideals of a digraph maintains a permutation of the vertices in an array *perm*, with the inverse permutation in *iperm*. Elements $inx[l]$ through $inx[l+1]-1$ of this array are known to be simultaneously either in or out of the ideal, according as $decision[l]=1$ or $decision[l]=0$, based on the decision made on level l of a backtrack tree.

The basic invariant relation is that we could obtain an ideal by either excluding or including all elements of index $\ge inx[l]$ in *perm*. This property holds when $l=0$ because $inx[0]=0$. To raise the level, we decide first to exclude vertex $perm[inx[l]]$; this also excludes all vertices that lead to it, and we rearrange *perm* in order to bring those elements into their proper place. Afterwards, we decide to include vertex $perm[inx[l]]$; this also includes all vertices that lead from it, in a similar way.

Vertex 0 corresponds to an artificial piece that is "stuck," as explained above. If this vertex is excluded from the ideal, we create a list of all board positions for vertices that are included; this will define a superpiece for shifts by *del*. But if the stuck vertex is included in the ideal, we create a list of all board positions for vertices that are excluded; the list in that case will define a superpiece for shifts by $-del$. The list contains $lstart[l]$ entries at the beginning of level l.

```
⟨Subroutines 10⟩ +≡
  void supermove(int, int);   /* see below */
  void ideals(int del)
  {
    register int j, k, l, p, u, v, t;
    for (j = 0; j ≤ bcount; j++) perm[j] = iperm[j] = j;
    l = p = 0;
  excl: decision[l] = 0, lstart[l] = p;
    for (j = inx[l], t = j + 1; j < t; j++) ⟨Put all vertices that lead to perm[j] into positions near j 51⟩;
    if (t > bcount) {
      ⟨Process an ideal 54⟩; goto incl;
    }
    inx[++l] = t; goto excl;
  incl: decision[l] = 1, p = lstart[l];
    for (j = inx[l], t = j + 1; j < t; j++) ⟨Put all vertices that lead from perm[j] into positions near j 52⟩;
    if (t > bcount) {
      ⟨Process an ideal 54⟩; goto backup;
    }
    inx[++l] = t; goto excl;
  backup: if (l) {
      l--;
      if (decision[l]) goto backup;
      goto incl;
    }
  }
```

51. ⟨ Put all vertices that lead to *perm*[*j*] into positions near *j* 51 ⟩ ≡

```
{
  v = perm[j];
  for (k = in[v]; k ≥ 0; k = ilink[k]) {
    u = aboard[k];
    if (iperm[u] ≥ t) {
      register int uu = perm[t], tt = iperm[u];
      perm[t] = u, perm[tt] = uu, iperm[u] = t, iperm[uu] = tt;
      t++;
    }
  }
  if (decision[0] ≡ 1)
    for (v = head[v]; v ≥ 0; v = link[v]) super[p++] = v;
}
```

This code is used in section 50.

52. ⟨ Put all vertices that lead from *perm*[*j*] into positions near *j* 52 ⟩ ≡

```
{
  u = perm[j];
  for (k = out[u]; k ≥ 0; k = olink[k]) {
    v = aboard[k];
    if (iperm[v] ≥ t) {
      register int vv = perm[t], tt = iperm[v];
      perm[t] = v, perm[tt] = vv, iperm[v] = t, iperm[vv] = tt;
      t++;
    }
  }
  if (decision[0] ≡ 0)
    for (u = head[u]; u ≥ 0; u = link[u]) super[p++] = u;
}
```

This code is used in section 50.

53. ⟨ Global variables 6 ⟩ +≡

```
int perm[maxsize + 1], iperm[maxsize + 1];   /* basic permutation and its inverse */
char decision[maxsize];   /* decisions */
int inx[maxsize], lstart[maxsize];   /* backup values at decision points */
int super[maxsize];   /* offsets for the current superpiece */
```

54. ⟨ Process an ideal 54 ⟩ ≡

```
if (p) {
  super[p] = 0;   /* sentinel at end of the superpiece */
  if (decision[0] ≡ 0) supermove(del, del);
  else supermove(−del, −del);
}
```

This code is used in section 50.

55. The *supermove* routine is like *move*, but it uses the superpiece defined in *super* instead of using block k.

⟨ Subroutines 10 ⟩ +≡

```
void supermove(int del, int delo)
{
  register int j, s, t;
  for (j = 0; super[j]; j++) {      /* we remove the superpiece */
    board[super[j]] = 0;
  }
  for (j = 0; super[j]; j++) {      /* we test if it fits in new position */
    if (board[del + super[j]]) goto illegal;
  }
  for (j = 0; super[j]; j++) {      /* if so, we move it */
    board[del + super[j]] = aboard[super[j]];
  }
  if (hashin(style ≡ 5) ∨ style ≡ 4) ⟨ Unmove the superpiece and recurse 56 ⟩
  else {
    for (j = 0; super[j]; j++) {      /* remove the shifted superpiece */
      board[del + super[j]] = 0;
    }
  illegal:
    for (j = 0; super[j]; j++) {      /* replace the unshifted superpiece */
      board[super[j]] = aboard[super[j]];
    }
  }
}
```

56. After we've moved a superpiece once, the digraph changes and so do the ideals. But that's OK; the *supermove* routine checks that we aren't blocked at any step of the way.

⟨ Unmove the superpiece and recurse 56 ⟩ ≡

```
{
  for (j = 0; super[j]; j++) {      /* remove the shifted superpiece */
    board[del + super[j]] = 0;
  }
  for (j = 0; super[j]; j++) {      /* replace the unshifted superpiece */
    board[super[j]] = aboard[super[j]];
  }
  if (style ≡ 4) supermove(del + delo, delo);
  else
    for (j = 0; j < 4; j++)
      if (delta[j] ≠ −delo) supermove(del + delta[j], delta[j]);
}
```

This code is used in section 55.

57. The program now comes to a glorious conclusion as we put the remaining pieces of code together.

⟨ Try all supermoves 57 ⟩ ≡

```
{
  ⟨ Copy and link the board 46 ⟩;
  ⟨ Construct the digraph for del = colsp 48 ⟩;
  ideals(colsp);
  head[0] = lr + 1;   /* I apologize for this tricky optimization */
  ⟨ Construct the digraph for del = 1 47 ⟩;
  ideals(1);
}
```

This code is used in section 42.

58. Index.

SLIDING