1. Introduction. This program finds a minimum-move solution to the famous "15 puzzle," using a method introduced by Richard E. Korf [*Artificial Intelligence* **27** (1985), 97–109]. It's the first of a series of ever-more-efficient ways to do the job. (You might want to read the zeroth program in the series, 15PUZZLE-KORF0, as background, although much of the documentation is repeated here.) My main reason for writing this group of routines was to experiment with a new (for me) style of programming, explained below.

The initial position is specified on the command line as a permutation of the hexadecimal digits {`0`, `1`, `2`, `3`, `4`, `5`, `6`, `7`, `8`, `9`, `a`, `b`, `c`, `d`, `e`, `f`}; this permutation is used to fill the rows of a 4×4 matrix, from top to bottom and left to right. For example, '`159d26ae37bf48c0`' specifies the starting position

```
1 5 9 d
2 6 a e
3 7 b f
4 8 c 0
```

The number `0` stands for a blank cell. Each step in solving the puzzle consists of swapping `0` with one of its neighbors. The goal position is always

```
1 2 3 4
5 6 7 8
9 a b c
d e f 0
```

(Korf had a different goal position, namely `0123456789abcdef`. I agree that his convention is mathematically superior to mine; but it conflicts with a 125-year-old tradition. So I have retained the historic practice. One can of course interchange our conventions, if desired, by rotating the board by 180° and by replacing each nonzero digit x by $16 - x$.)

```
#include <stdio.h>
#include <time.h>
  char board[16];
  char start[16];
  int stack[100];
  int timer;
  main(int argc, char *argv[])
  {
    register int j, k, s, t, del, piece, moves;
    ⟨ Input the initial position 2 ⟩;
    ⟨ Apply Korf's procedure 5 ⟩;
    ⟨ Output the results 24 ⟩;
  }
```

2. Let's regard the 16 cell positions as two-digit numbers in quaternary (radix-4) notation:

$$\begin{array}{cccc} 00 & 01 & 02 & 03 \\ 10 & 11 & 12 & 13 \\ 20 & 21 & 22 & 23 \\ 30 & 31 & 32 & 33 \end{array}$$

Thus each cell is identified by its row number r and its column number c, making a two-nyp code (r, c), with $0 \le r, c < 4$.

Furthermore, it's convenient to renumber the input digits `1`, `2`, ..., `f`, so that they match their final destination positions, 00, 01, ..., 32. This conversion simply subtracts 1; so `0` gets mapped into -1. The example initial position given earlier will therefore appear as follows in the *start* array:

$$\begin{array}{cccc} 00 & 10 & 20 & 30 \\ 01 & 11 & 21 & 31 \\ 02 & 12 & 22 & 32 \\ 03 & 13 & 23 & -1 \end{array}$$

Half of the initial positions make the puzzle unsolvable, because the permutation must be odd if and only if the `0` must move an odd number of times. This solvability condition is checked when we read the input.

```
#define row(x) ((x) ≫ 2)
#define col(x) ((x) & #3)
⟨ Input the initial position 2 ⟩ ≡
  if (argc ≠ 2) {
    fprintf(stderr, "Usage: %s startposition\n", argv[0]); exit(−1);
  }
  for (j = 0; k = argv[1][j]; j++) {
    if (k ≥ '0' ∧ k ≤ '9') k −= '0';
    else if (k ≥ 'a' ∧ k ≤ 'f') k −= 'a' − 10;
    else {
      fprintf(stderr, "The start position should use only hex digits (0123456789abcdef)!\n");
      exit(−2);
    }
    if (start[k]) {
      fprintf(stderr, "Your start position uses %x twice!\n", k); exit(−3);
    }
    start[k] = 1;
  }
  for (k = 0; k < 16; k++)
    if (start[k] ≡ 0) {
      fprintf(stderr, "Your start position doesn't use %x!\n", k); exit(−4);
    }
  for (del = j = 0; k = argv[1][j]; j++) {
    if (k ≥ '0' ∧ k ≤ '9') k −= '0'; else k −= 'a' − 10;
    start[j] = k − 1;
    for (s = 0; s < j; s++)
      if (start[s] > start[j]) del++;      /* count inversions */
    if (k ≡ 0) t = j;
  }
  if (((row(t) + col(t) + del) & #1) ≡ 0) {
    printf("Sorry ... the goal is unreachable from that start position!\n");
    exit(0);
  }
```

This code is used in section 1.

3. Korf's method. If piece (r, c) is currently in board position (r', c'), it must make at least $|r - r'| + |c - c'|$ moves before it reaches home. The sum of these numbers, over all fifteen pieces, is called h; this quantity is also known as the "Manhattan metric lower bound."

We will say that a move is *happy* if it moves a piece closer to the goal; otherwise we'll call the move *sad*. Korf's key idea is to try first to find a solution that makes only happy moves. (For example, one can actually win from the starting position `bc9e80df3412a756` by making $h = 56$ moves that are entirely happy.) If that fails, we start over, but this time we try to make $h + 1$ happy moves and 1 sad move. And if that also fails, we try for $h + k$ happy moves and k sad ones, for $k = 2, 3, \ldots$, until finally we succeed.

That strategy may sound silly, because each new value of k repeats calculations already made. But it's actually brilliant, for two reasons: (1) The search can be carried out with almost no memory, for any fixed value of k — in fact, this program uses fewer than 500 bytes for all its data. (2) The total running time for $k = 0, 1, \ldots, k_0$ isn't much more than the time needed for a *single* run with $k = k_0$, because the running time increases exponentially with k.

Memory requirements are minimal because we can get away with memoryless depth-first search to explore all solutions. The fifteen puzzle is much nicer in this respect than many other problems; indeed, a blind depth-first search as used here is often a poor choice in other applications, because it might explore many subproblems repeatedly, having no inkling that it has already "been there and done that." But the search tree of the fifteen puzzle has comparatively few overlapping branches. For example, the shortest cycle of moves that restores a previously examined position occurs only when we go thrice around a 2×2 subsquare, leading to a cycle of length 12; therefore the first five levels below any node of the tree consist entirely of distinct states of the board, and only a few duplicates occur at level six.

Note: The Manhattan metric is somewhat weak, and substantially better lower bounds are known. Some day I plan to incorporate them into later programs in this series. The simple approach adopted here is sufficient to handle most random initial positions in a reasonable amount of time. But when it is applied to the example of transposition, in the introduction, it is too slow: That example has a Manhattan lower bound of 40, yet the shortest solutions have 72 moves. Thus the k value for that problem is 16; and each new value of k takes empirically about 5.7 times as long as the previous case. (So the running time was 40.4 hours on my Opteron computer, vintage 2004.)

Moreover, I could improve the present scheme in various other ways. Even with the Manhattan metric, I find that there are 88,728,779 positions from which the goal is achievable without any sad moves. (A surprisingly large number — at least, I was expecting less than a million. Incidentally, exactly 114 of those positions have the maximum distance to the goal, namely 56, as in the example stated above.) If we had a table of those positions in memory, we would know that our last-permitted sad move must be to something in the table; that might make a significant speedup. Korf and Taylor have described a series of other improvements that actually allowed them to solve random instances of the 24-puzzle in 1996 [*AAAI National Conference Proceedings* (1996), 1202–1207].

Another note: Maybe my terminology "happy" versus "sad" is too poignant. Should I have called them "downhill" and "uphill" moves instead? Readers comments are invited.

4. Saving memory means that the computation lives entirely within the computer's high-speed cache. Of course we do need to examine billions of positions, in tough cases; so we want the inner loop of depth-first search to be short.

Let's take a look at the operations that are involved when, say, we've just moved the empty cell into position (r, c), coming from the south. (In other words, we will imagine that the empty cell was previously at $(r + 1, c)$. We'll think of the *empty* cell as moving north, although the board has actually changed by moving a *piece* to the south.) From (r, c) we'll try to move the empty cell either west to $(r, c - 1)$, or north to $(r - 1, c)$, or east to $(r, c + 1)$; up to three moves are possible, depending on whether (r, c) is a middle cell, an edge cell, or a corner cell. After having tried all those possibilities, we'll backtrack and make the next move from the cell to the south that got us started.

A straightforward implementation appears in the program 15PUZZLE-KORF0, where the inner loop contains many tests: (1) Choosing a direction. (2) Testing if this direction is feasible from the current (r, c). (3) If feasible, is the move happy or sad? (4) If it's sad, have we exceeded our quota of sad moves? (5) Have we reached the goal? (6) Have we exhausted all possibilities?

Such conditional branches play havoc with the pipeline organization of a modern computer. Therefore the present program avoids most of them by straight-line coding, using essentially a finite-state automaton to control the actions. The program consists conceptually of $4 \times 4 \times 4 \times 4 = 256$ small parts, one for each combination of (r, c, d, p), where we're at empty cell position (r, c) trying to move the empty cell in direction d, having entered this cell from direction p.

(In fact, 256 is a generous upper estimate, because many combinations are impossible at the edges and corners. We won't, for example, try to move east when the empty cell is in column 3. The true number of cases is 152, of which 48 simply undo a move that was made.)

I could have generated the individual pieces of code with the macro capability of C's preprocessor. But my debugging tools don't know how to deal with macros satisfactorily. So I did the repetitive steps with macros at another level, namely with `emacs` as I typed this program into the computer. The C preprocessor did, however, come in handy to make the code more readable than it would have been if I had fully expanded everything by text editing.

Instead of writing this program as a collection of 152 little modules, I could have made it a collection of 152 little procedures. But that wouldn't have bought me anything; as a full professor with tenure, I don't have to worry about being fired when I use **goto** statements. The overhead of subroutine calling, and the fact that "tail recursion" is extremely useful in the situations met here, both convince me that no state-of-the-art compiler would come up with anything near as efficient if I wrote procedures instead of this code that purposely looks like spaghetti. Nor would a procedure-oriented version be easier for me to write.

The only drawback to the style adopted here, as far as I can see, is that I had many chances to make mistakes; a program of this kind will seem to work even though it contains typographic errors. Therefore I had to do a lot of careful desk checking.

Does all this replication help? The program now has shifted the computer's work from the instruction cache to the data cache. One natural benchmark is the case `ca6098dfb73254e1`, which proved to be the toughest of the 100 random examples in Korf's original paper. (He reported that his implementation searched more than 6 billion nodes in that case.) In my first tests, on an Athlon computer purchased in the year 2000, the running time for this toughie was 8 minutes and 58 seconds. By comparison, 15PUZZLE-KORF0 requires 24 minutes and 26 seconds on the same machine. Korf's original experiment, using a Pascal compiler on a DEC 2060 in 1984, took about 4000 minutes; so the present program represents a more than 400-fold speedup, of which we can attribute a factor of roughly 160 to improvements in hardware, and another factor of about 2.7 to the implementation technique adopted here.

5. Enough of this introduction. Let's get on with the program.

```
⟨ Apply Korf's procedure 5 ⟩ ≡
  ⟨ Set moves to the minimum number of happy moves 6 ⟩;
  if (moves ≡ 0) goto win;      /* otherwise our solution will take 6 + 6 moves! */
  while (1) {
    timer = time(0);
    t = moves;      /* desired number of ((sad moves) ≪ 8) + (happy moves) */
    ⟨ Try for a solution with t more moves 23 ⟩;
    printf("␣...␣no␣solution␣with␣%d+%d␣moves␣(%d␣sec)\n", moves & #ff, moves ≫ 8,
        time(0) − timer);
    moves += #101;      /* add a sad move and a happy move to the current quota */
  }
win:
```

This code is used in section 1.

6.
```
⟨ Set moves to the minimum number of happy moves 6 ⟩ ≡
  for (j = moves = 0; j < 16; j++)
    if (start[j] ≥ 0) {
      del = row(start[j]) − row(j);
      moves += (del < 0 ? −del : del);
      del = col(start[j]) − col(j);
      moves += (del < 0 ? −del : del);
    }
```

This code is used in section 5.

7. The main control routine is a stack, which records two things: In the left 16 bits is the number of (sad, happy) moves remaining, called t; this number will be restored when backtracking. And in the right 16 bits is a code number of the routine to execute next, after finishing every task that's higher on the stack.

8.
```
⟨ Switch into action 8 ⟩ ≡
switcher: t = stack[−−s] ≫ 16;
  switch (stack[s] & #ffff) {
    ⟨ Cases that move east from column 0 10 ⟩
    ⟨ Cases that move east from column 1 11 ⟩
    ⟨ Cases that move east from column 2 12 ⟩
    ⟨ Cases that move west from column 1 13 ⟩
    ⟨ Cases that move west from column 2 14 ⟩
    ⟨ Cases that move west from column 3 15 ⟩
    ⟨ Cases that move north from row 1 16 ⟩
    ⟨ Cases that move north from row 2 17 ⟩
    ⟨ Cases that move north from row 3 18 ⟩
    ⟨ Cases that move south from row 0 19 ⟩
    ⟨ Cases that move south from row 1 20 ⟩
    ⟨ Cases that move south from row 2 21 ⟩
    ⟨ Cases that move back 22 ⟩
  case bottom: break;
  default: fprintf(stderr, "Oops,␣I'm␣confused␣about␣case␣%x!\n", stack[s]);
  }
```

This code is used in section 23.

9. Directions are encoded as follows: east = 0, north = 1, west = 2, south = 3. (Think of powers of i in the complex plane.)

Each feasible combination of (r, c, d, p) is encoded as an 8-bit quantity called $code(r, c, d, p)$, so that we can easily switch to it. There's also $tailcode(r, c, d)$; this stands for a case that will try only a single direction d at cell (r, c) before returning.

To avoid complicated testing for happiness and sadness, the arithmetical calculation of *del* shown here will produce `#100` if a move in direction d is currently sad, `#001` if that move is currently happy.

```
#define code(r,c,d,p) (((((r << 2) + c) << 2) + d) << 2) + p
#define tailcode(r,c,d) #100 + (code(0,r,c,d))
#define bottom code(0,0,1,1)      /* an otherwise unused code */
#define bord(r,c) board[((r) << 2) + (c)]
#define east(r,c) piece = bord(r,c+1); del = (((c - col(piece)) >> 2) & #ff) + 1
#define west(r,c) piece = bord(r,c-1); del = (((col(piece) - c) >> 2) & #ff) + 1
#define north(r,c) piece = bord(r-1,c); del = (((row(piece) - r) >> 2) & #ff) + 1
#define south(r,c) piece = bord(r+1,c); del = (((r - row(piece)) >> 2) & #ff) + 1
```

10. ⟨ Cases that move east from column 0 10 ⟩ ≡

```
case tailcode(0,0,0): case code(0,0,0,3): r0c0d0p3: east(0,0);
  if (t ≤ del) { if (t ≡ del) goto win; else goto r0c0d3p3; }
  bord(0,0) = piece, stack[s++] = (t << 16) + code(0,0,3,3), t -= del;
  goto r0c1d0p2;
case code(1,0,0,1): r1c0d0p1: east(1,0);
  if (t ≤ del) { if (t ≡ del) goto win; else goto r1c0d3p1; }
  bord(1,0) = piece, stack[s++] = (t << 16) + code(1,0,3,1), t -= del;
  goto r1c1d1p2;
case tailcode(1,0,0): case code(1,0,0,3): r1c0d0p3: east(1,0);
  if (t ≤ del) { if (t ≡ del) goto win; else goto r1c0d3p3; }
  bord(1,0) = piece, stack[s++] = (t << 16) + code(1,0,3,3), t -= del;
  goto r1c1d1p2;
case code(2,0,0,1): r2c0d0p1: east(2,0);
  if (t ≤ del) { if (t ≡ del) goto win; else goto r2c0d3p1; }
  bord(2,0) = piece, stack[s++] = (t << 16) + code(2,0,3,1), t -= del;
  goto r2c1d1p2;
case tailcode(2,0,0): case code(2,0,0,3): r2c0d0p3: east(2,0);
  if (t ≤ del) { if (t ≡ del) goto win; else goto r2c0d3p3; }
  bord(2,0) = piece, stack[s++] = (t << 16) + code(2,0,3,3), t -= del;
  goto r2c1d1p2;
case tailcode(3,0,0): case code(3,0,0,1): r3c0d0p1: east(3,0);
  if (t ≤ del) { if (t ≡ del) goto win; else goto r3c0d1p1; }
  bord(3,0) = piece, stack[s++] = (t << 16) + code(3,0,1,1), t -= del;
  goto r3c1d1p2;
```

This code is used in section 8.

11. ⟨ Cases that move east from column 1 11 ⟩ ≡

```
case code(0,1,0,2): r0c1d0p2: east(0,1);
  if (t ≤ del) { if (t ≡ del) goto win; else goto r0c1d3p2; }
  bord(0,1) = piece, stack[s++] = (t ≪ 16) + code(0,1,3,2), t −= del;
  goto r0c2d0p2;
case tailcode(0,1,0): case code(0,1,0,3): r0c1d0p3: east(0,1);
  if (t ≤ del) { if (t ≡ del) goto win; else goto r0c1d3p3; }
  bord(0,1) = piece, stack[s++] = (t ≪ 16) + code(0,1,3,3), t −= del;
  goto r0c2d0p2;
case code(1,1,0,1): r1c1d0p1: east(1,1);
  if (t ≤ del) { if (t ≡ del) goto win; else goto r1c1d3p1; }
  bord(1,1) = piece, stack[s++] = (t ≪ 16) + code(1,1,3,1), t −= del;
  goto r1c2d1p2;
case code(1,1,0,2): r1c1d0p2: east(1,1);
  if (t ≤ del) { if (t ≡ del) goto win; else goto r1c1d3p2; }
  bord(1,1) = piece, stack[s++] = (t ≪ 16) + code(1,1,3,2), t −= del;
  goto r1c2d1p2;
case tailcode(1,1,0): case code(1,1,0,3): r1c1d0p3: east(1,1);
  if (t ≤ del) { if (t ≡ del) goto win; else goto r1c1d3p3; }
  bord(1,1) = piece, stack[s++] = (t ≪ 16) + code(1,1,3,3), t −= del;
  goto r1c2d1p2;
case code(2,1,0,1): r2c1d0p1: east(2,1);
  if (t ≤ del) { if (t ≡ del) goto win; else goto r2c1d3p1; }
  bord(2,1) = piece, stack[s++] = (t ≪ 16) + code(2,1,3,1), t −= del;
  goto r2c2d1p2;
case code(2,1,0,2): r2c1d0p2: east(2,1);
  if (t ≤ del) { if (t ≡ del) goto win; else goto r2c1d3p2; }
  bord(2,1) = piece, stack[s++] = (t ≪ 16) + code(2,1,3,2), t −= del;
  goto r2c2d1p2;
case tailcode(2,1,0): case code(2,1,0,3): r2c1d0p3: east(2,1);
  if (t ≤ del) { if (t ≡ del) goto win; else goto r2c1d3p3; }
  bord(2,1) = piece, stack[s++] = (t ≪ 16) + code(2,1,3,3), t −= del;
  goto r2c2d1p2;
case code(3,1,0,1): r3c1d0p1: east(3,1);
  if (t ≤ del) { if (t ≡ del) goto win; else goto r3c1d2p1; }
  bord(3,1) = piece, stack[s++] = (t ≪ 16) + code(3,1,2,1), t −= del;
  goto r3c2d1p2;
case tailcode(3,1,0): case code(3,1,0,2): r3c1d0p2: east(3,1);
  if (t ≤ del) { if (t ≡ del) goto win; else goto r3c1d2p2; }
  bord(3,1) = piece, stack[s++] = (t ≪ 16) + code(3,1,2,2), t −= del;
  goto r3c2d1p2;
```

This code is used in section 8.

12. ⟨ Cases that move east from column 2 12 ⟩ ≡

```
case code(0,2,0,2): r0c2d0p2: east(0,2);
  if (t ≤ del) { if (t ≡ del) goto win; else goto r0c2d3p2; }
  bord(0,2) = piece, stack[s++] = (t ≪ 16) + code(0,2,3,2), t −= del;
  goto r0c3d3p2;
case tailcode(0,2,0): case code(0,2,0,3): r0c2d0p3: east(0,2);
  if (t ≤ del) { if (t ≡ del) goto win; else goto r0c2d3p3; }
  bord(0,2) = piece, stack[s++] = (t ≪ 16) + code(0,2,3,3), t −= del;
  goto r0c3d3p2;
case code(1,2,0,1): r1c2d0p1: east(1,2);
  if (t ≤ del) { if (t ≡ del) goto win; else goto r1c2d3p1; }
  bord(1,2) = piece, stack[s++] = (t ≪ 16) + code(1,2,3,1), t −= del;
  goto r1c3d1p2;
case code(1,2,0,2): r1c2d0p2: east(1,2);
  if (t ≤ del) { if (t ≡ del) goto win; else goto r1c2d3p2; }
  bord(1,2) = piece, stack[s++] = (t ≪ 16) + code(1,2,3,2), t −= del;
  goto r1c3d1p2;
case tailcode(1,2,0): case code(1,2,0,3): r1c2d0p3: east(1,2);
  if (t ≤ del) { if (t ≡ del) goto win; else goto r1c2d3p3; }
  bord(1,2) = piece, stack[s++] = (t ≪ 16) + code(1,2,3,3), t −= del;
  goto r1c3d1p2;
case code(2,2,0,1): r2c2d0p1: east(2,2);
  if (t ≤ del) { if (t ≡ del) goto win; else goto r2c2d3p1; }
  bord(2,2) = piece, stack[s++] = (t ≪ 16) + code(2,2,3,1), t −= del;
  goto r2c3d1p2;
case code(2,2,0,2): r2c2d0p2: east(2,2);
  if (t ≤ del) { if (t ≡ del) goto win; else goto r2c2d3p2; }
  bord(2,2) = piece, stack[s++] = (t ≪ 16) + code(2,2,3,2), t −= del;
  goto r2c3d1p2;
case tailcode(2,2,0): case code(2,2,0,3): r2c2d0p3: east(2,2);
  if (t ≤ del) { if (t ≡ del) goto win; else goto r2c2d3p3; }
  bord(2,2) = piece, stack[s++] = (t ≪ 16) + code(2,2,3,3), t −= del;
  goto r2c3d1p2;
case code(3,2,0,1): r3c2d0p1: east(3,2);
  if (t ≤ del) { if (t ≡ del) goto win; else goto r3c2d2p1; }
  bord(3,2) = piece, stack[s++] = (t ≪ 16) + code(3,2,2,1), t −= del;
  goto r3c3d1p2;
case tailcode(3,2,0): case code(3,2,0,2): r3c2d0p2: east(3,2);
  if (t ≤ del) { if (t ≡ del) goto win; else goto r3c2d2p2; }
  bord(3,2) = piece, stack[s++] = (t ≪ 16) + code(3,2,2,2), t −= del;
  goto r3c3d1p2;
```

This code is used in section 8.

13. ⟨ Cases that move west from column 1 13 ⟩ ≡

```
case tailcode(0,1,2): case code(0,1,2,0): r0c1d2p0: west(0,1);
  if (t ≤ del) { if (t ≡ del) goto win; else goto r0c1d0p0; }
  bord(0,1) = piece, stack[s++] = (t ≪ 16) + code(0,1,0,0), t −= del;
  goto r0c0d3p0;
case code(0,1,2,3): r0c1d2p3: west(0,1);
  if (t ≤ del) { if (t ≡ del) goto win; else goto r0c1d0p3; }
  bord(0,1) = piece, stack[s++] = (t ≪ 16) + code(0,1,0,3), t −= del;
  goto r0c0d3p0;
case tailcode(1,1,2): case code(1,1,2,1): r1c1d2p1: west(1,1);
  if (t ≤ del) { if (t ≡ del) goto win; else goto r1c1d1p1; }
  bord(1,1) = piece, stack[s++] = (t ≪ 16) + code(1,1,1,1), t −= del;
  goto r1c0d3p0;
case code(1,1,2,0): r1c1d2p0: west(1,1);
  if (t ≤ del) { if (t ≡ del) goto win; else goto r1c1d1p0; }
  bord(1,1) = piece, stack[s++] = (t ≪ 16) + code(1,1,1,0), t −= del;
  goto r1c0d3p0;
case code(1,1,2,3): r1c1d2p3: west(1,1);
  if (t ≤ del) { if (t ≡ del) goto win; else goto r1c1d1p3; }
  bord(1,1) = piece, stack[s++] = (t ≪ 16) + code(1,1,1,3), t −= del;
  goto r1c0d3p0;
case tailcode(2,1,2): case code(2,1,2,1): r2c1d2p1: west(2,1);
  if (t ≤ del) { if (t ≡ del) goto win; else goto r2c1d1p1; }
  bord(2,1) = piece, stack[s++] = (t ≪ 16) + code(2,1,1,1), t −= del;
  goto r2c0d3p0;
case code(2,1,2,0): r2c1d2p0: west(2,1);
  if (t ≤ del) { if (t ≡ del) goto win; else goto r2c1d1p0; }
  bord(2,1) = piece, stack[s++] = (t ≪ 16) + code(2,1,1,0), t −= del;
  goto r2c0d3p0;
case code(2,1,2,3): r2c1d2p3: west(2,1);
  if (t ≤ del) { if (t ≡ del) goto win; else goto r2c1d1p3; }
  bord(2,1) = piece, stack[s++] = (t ≪ 16) + code(2,1,1,3), t −= del;
  goto r2c0d3p0;
case tailcode(3,1,2): case code(3,1,2,1): r3c1d2p1: west(3,1);
  if (t ≤ del) { if (t ≡ del) goto win; else goto r3c1d1p1; }
  bord(3,1) = piece, stack[s++] = (t ≪ 16) + code(3,1,1,1), t −= del;
  goto r3c0d1p0;
case code(3,1,2,0): r3c1d2p0: west(3,1);
  if (t ≤ del) { if (t ≡ del) goto win; else goto r3c1d1p0; }
  bord(3,1) = piece, stack[s++] = (t ≪ 16) + code(3,1,1,0), t −= del;
  goto r3c0d1p0;
```

This code is used in section 8.

14. ⟨ Cases that move west from column 2 14 ⟩ ≡

```
case tailcode(0,2,2): case code(0,2,2,0): r0c2d2p0: west(0,2);
  if (t ≤ del) { if (t ≡ del) goto win; else goto r0c2d0p0; }
  bord(0,2) = piece, stack[s++] = (t ≪ 16) + code(0,2,0,0), t −= del;
  goto r0c1d3p0;
case code(0,2,2,3): r0c2d2p3: west(0,2);
  if (t ≤ del) { if (t ≡ del) goto win; else goto r0c2d0p3; }
  bord(0,2) = piece, stack[s++] = (t ≪ 16) + code(0,2,0,3), t −= del;
  goto r0c1d3p0;
case tailcode(1,2,2): case code(1,2,2,1): r1c2d2p1: west(1,2);
  if (t ≤ del) { if (t ≡ del) goto win; else goto r1c2d1p1; }
  bord(1,2) = piece, stack[s++] = (t ≪ 16) + code(1,2,1,1), t −= del;
  goto r1c1d3p0;
case code(1,2,2,0): r1c2d2p0: west(1,2);
  if (t ≤ del) { if (t ≡ del) goto win; else goto r1c2d1p0; }
  bord(1,2) = piece, stack[s++] = (t ≪ 16) + code(1,2,1,0), t −= del;
  goto r1c1d3p0;
case code(1,2,2,3): r1c2d2p3: west(1,2);
  if (t ≤ del) { if (t ≡ del) goto win; else goto r1c2d1p3; }
  bord(1,2) = piece, stack[s++] = (t ≪ 16) + code(1,2,1,3), t −= del;
  goto r1c1d3p0;
case tailcode(2,2,2): case code(2,2,2,1): r2c2d2p1: west(2,2);
  if (t ≤ del) { if (t ≡ del) goto win; else goto r2c2d1p1; }
  bord(2,2) = piece, stack[s++] = (t ≪ 16) + code(2,2,1,1), t −= del;
  goto r2c1d3p0;
case code(2,2,2,0): r2c2d2p0: west(2,2);
  if (t ≤ del) { if (t ≡ del) goto win; else goto r2c2d1p0; }
  bord(2,2) = piece, stack[s++] = (t ≪ 16) + code(2,2,1,0), t −= del;
  goto r2c1d3p0;
case code(2,2,2,3): r2c2d2p3: west(2,2);
  if (t ≤ del) { if (t ≡ del) goto win; else goto r2c2d1p3; }
  bord(2,2) = piece, stack[s++] = (t ≪ 16) + code(2,2,1,3), t −= del;
  goto r2c1d3p0;
case tailcode(3,2,2): case code(3,2,2,1): r3c2d2p1: west(3,2);
  if (t ≤ del) { if (t ≡ del) goto win; else goto r3c2d1p1; }
  bord(3,2) = piece, stack[s++] = (t ≪ 16) + code(3,2,1,1), t −= del;
  goto r3c1d2p0;
case code(3,2,2,0): r3c2d2p0: west(3,2);
  if (t ≤ del) { if (t ≡ del) goto win; else goto r3c2d1p0; }
  bord(3,2) = piece, stack[s++] = (t ≪ 16) + code(3,2,1,0), t −= del;
  goto r3c1d2p0;
```

This code is used in section 8.

15. ⟨ Cases that move west from column 3 15 ⟩ ≡

```
case tailcode(0,3,2): case code(0,3,2,3): r0c3d2p3: west(0,3);
  if (t ≤ del) { if (t ≡ del) goto win; else goto r0c3d3p3; }
  bord(0,3) = piece, stack[s++] = (t ≪ 16) + code(0,3,3,3), t −= del;
  goto r0c2d3p0;
case tailcode(1,3,2): case code(1,3,2,1): r1c3d2p1: west(1,3);
  if (t ≤ del) { if (t ≡ del) goto win; else goto r1c3d1p1; }
  bord(1,3) = piece, stack[s++] = (t ≪ 16) + code(1,3,1,1), t −= del;
  goto r1c2d3p0;
case code(1,3,2,3): r1c3d2p3: west(1,3);
  if (t ≤ del) { if (t ≡ del) goto win; else goto r1c3d1p3; }
  bord(1,3) = piece, stack[s++] = (t ≪ 16) + code(1,3,1,3), t −= del;
  goto r1c2d3p0;
case tailcode(2,3,2): case code(2,3,2,1): r2c3d2p1: west(2,3);
  if (t ≤ del) { if (t ≡ del) goto win; else goto r2c3d1p1; }
  bord(2,3) = piece, stack[s++] = (t ≪ 16) + code(2,3,1,1), t −= del;
  goto r2c2d3p0;
case code(2,3,2,3): r2c3d2p3: west(2,3);
  if (t ≤ del) { if (t ≡ del) goto win; else goto r2c3d1p3; }
  bord(2,3) = piece, stack[s++] = (t ≪ 16) + code(2,3,1,3), t −= del;
  goto r2c2d3p0;
case tailcode(3,3,2): case code(3,3,2,1): r3c3d2p1: west(3,3);
  if (t ≤ del) { if (t ≡ del) goto win; else goto r3c3d1p1; }
  bord(3,3) = piece, stack[s++] = (t ≪ 16) + code(3,3,1,1), t −= del;
  goto r3c2d2p0;
```

This code is used in section 8.

16. ⟨ Cases that move north from row 1 16 ⟩ ≡

case *tailcode*$(1,0,1)$: **case** *code*$(1,0,1,0)$: *r1c0d1p0*: *north*$(1,0)$;
  if $(t \le del)$ { **if** $(t \equiv del)$ **goto** *win*; **else goto** *r1c0d0p0*; }
  bord$(1,0) = piece, stack[s{+}{+}] = (t \ll 16) + code(1,0,0,0), t \mathrel{-}= del$;
  goto *r0c0d0p3*;
case *code*$(1,0,1,3)$: *r1c0d1p3*: *north*$(1,0)$;
  if $(t \le del)$ { **if** $(t \equiv del)$ **goto** *win*; **else goto** *r1c0d0p3*; }
  bord$(1,0) = piece, stack[s{+}{+}] = (t \ll 16) + code(1,0,0,3), t \mathrel{-}= del$;
  goto *r0c0d0p3*;
case *tailcode*$(1,1,1)$: **case** *code*$(1,1,1,0)$: *r1c1d1p0*: *north*$(1,1)$;
  if $(t \le del)$ { **if** $(t \equiv del)$ **goto** *win*; **else goto** *r1c1d0p0*; }
  bord$(1,1) = piece, stack[s{+}{+}] = (t \ll 16) + code(1,1,0,0), t \mathrel{-}= del$;
  goto *r0c1d2p3*;
case *code*$(1,1,1,2)$: *r1c1d1p2*: *north*$(1,1)$;
  if $(t \le del)$ { **if** $(t \equiv del)$ **goto** *win*; **else goto** *r1c1d0p2*; }
  bord$(1,1) = piece, stack[s{+}{+}] = (t \ll 16) + code(1,1,0,2), t \mathrel{-}= del$;
  goto *r0c1d2p3*;
case *code*$(1,1,1,3)$: *r1c1d1p3*: *north*$(1,1)$;
  if $(t \le del)$ { **if** $(t \equiv del)$ **goto** *win*; **else goto** *r1c1d0p3*; }
  bord$(1,1) = piece, stack[s{+}{+}] = (t \ll 16) + code(1,1,0,3), t \mathrel{-}= del$;
  goto *r0c1d2p3*;
case *tailcode*$(1,2,1)$: **case** *code*$(1,2,1,0)$: *r1c2d1p0*: *north*$(1,2)$;
  if $(t \le del)$ { **if** $(t \equiv del)$ **goto** *win*; **else goto** *r1c2d0p0*; }
  bord$(1,2) = piece, stack[s{+}{+}] = (t \ll 16) + code(1,2,0,0), t \mathrel{-}= del$;
  goto *r0c2d2p3*;
case *code*$(1,2,1,2)$: *r1c2d1p2*: *north*$(1,2)$;
  if $(t \le del)$ { **if** $(t \equiv del)$ **goto** *win*; **else goto** *r1c2d0p2*; }
  bord$(1,2) = piece, stack[s{+}{+}] = (t \ll 16) + code(1,2,0,2), t \mathrel{-}= del$;
  goto *r0c2d2p3*;
case *code*$(1,2,1,3)$: *r1c2d1p3*: *north*$(1,2)$;
  if $(t \le del)$ { **if** $(t \equiv del)$ **goto** *win*; **else goto** *r1c2d0p3*; }
  bord$(1,2) = piece, stack[s{+}{+}] = (t \ll 16) + code(1,2,0,3), t \mathrel{-}= del$;
  goto *r0c2d2p3*;
case *code*$(1,3,1,2)$: *r1c3d1p2*: *north*$(1,3)$;
  if $(t \le del)$ { **if** $(t \equiv del)$ **goto** *win*; **else goto** *r1c3d3p2*; }
  bord$(1,3) = piece, stack[s{+}{+}] = (t \ll 16) + code(1,3,3,2), t \mathrel{-}= del$;
  goto *r0c3d2p3*;
case *tailcode*$(1,3,1)$: **case** *code*$(1,3,1,3)$: *r1c3d1p3*: *north*$(1,3)$;
  if $(t \le del)$ { **if** $(t \equiv del)$ **goto** *win*; **else goto** *r1c3d3p3*; }
  bord$(1,3) = piece, stack[s{+}{+}] = (t \ll 16) + code(1,3,3,3), t \mathrel{-}= del$;
  goto *r0c3d2p3*;

This code is used in section 8.

17. ⟨ Cases that move north from row 2 17 ⟩ ≡

```
case tailcode(2,0,1): case code(2,0,1,0): r2c0d1p0: north(2,0);
  if (t ≤ del) { if (t ≡ del) goto win; else goto r2c0d0p0; }
  bord(2,0) = piece, stack[s++] = (t ≪ 16) + code(2,0,0,0), t −= del;
  goto r1c0d1p3;
case code(2,0,1,3): r2c0d1p3: north(2,0);
  if (t ≤ del) { if (t ≡ del) goto win; else goto r2c0d0p3; }
  bord(2,0) = piece, stack[s++] = (t ≪ 16) + code(2,0,0,3), t −= del;
  goto r1c0d1p3;
case tailcode(2,1,1): case code(2,1,1,0): r2c1d1p0: north(2,1);
  if (t ≤ del) { if (t ≡ del) goto win; else goto r2c1d0p0; }
  bord(2,1) = piece, stack[s++] = (t ≪ 16) + code(2,1,0,0), t −= del;
  goto r1c1d2p3;
case code(2,1,1,2): r2c1d1p2: north(2,1);
  if (t ≤ del) { if (t ≡ del) goto win; else goto r2c1d0p2; }
  bord(2,1) = piece, stack[s++] = (t ≪ 16) + code(2,1,0,2), t −= del;
  goto r1c1d2p3;
case code(2,1,1,3): r2c1d1p3: north(2,1);
  if (t ≤ del) { if (t ≡ del) goto win; else goto r2c1d0p3; }
  bord(2,1) = piece, stack[s++] = (t ≪ 16) + code(2,1,0,3), t −= del;
  goto r1c1d2p3;
case tailcode(2,2,1): case code(2,2,1,0): r2c2d1p0: north(2,2);
  if (t ≤ del) { if (t ≡ del) goto win; else goto r2c2d0p0; }
  bord(2,2) = piece, stack[s++] = (t ≪ 16) + code(2,2,0,0), t −= del;
  goto r1c2d2p3;
case code(2,2,1,2): r2c2d1p2: north(2,2);
  if (t ≤ del) { if (t ≡ del) goto win; else goto r2c2d0p2; }
  bord(2,2) = piece, stack[s++] = (t ≪ 16) + code(2,2,0,2), t −= del;
  goto r1c2d2p3;
case code(2,2,1,3): r2c2d1p3: north(2,2);
  if (t ≤ del) { if (t ≡ del) goto win; else goto r2c2d0p3; }
  bord(2,2) = piece, stack[s++] = (t ≪ 16) + code(2,2,0,3), t −= del;
  goto r1c2d2p3;
case code(2,3,1,2): r2c3d1p2: north(2,3);
  if (t ≤ del) { if (t ≡ del) goto win; else goto r2c3d3p2; }
  bord(2,3) = piece, stack[s++] = (t ≪ 16) + code(2,3,3,2), t −= del;
  goto r1c3d2p3;
case tailcode(2,3,1): case code(2,3,1,3): r2c3d1p3: north(2,3);
  if (t ≤ del) { if (t ≡ del) goto win; else goto r2c3d3p3; }
  bord(2,3) = piece, stack[s++] = (t ≪ 16) + code(2,3,3,3), t −= del;
  goto r1c3d2p3;
```

This code is used in section 8.

18. ⟨ Cases that move north from row 3 18 ⟩ ≡

```
case tailcode(3,0,1): case code(3,0,1,0): r3c0d1p0: north(3,0);
  if (t ≤ del) { if (t ≡ del) goto win; else goto r3c0d0p0; }
  bord(3,0) = piece, stack[s++] = (t ≪ 16) + code(3,0,0,0), t −= del;
  goto r2c0d1p3;
case tailcode(3,1,1): case code(3,1,1,0): r3c1d1p0: north(3,1);
  if (t ≤ del) { if (t ≡ del) goto win; else goto r3c1d0p0; }
  bord(3,1) = piece, stack[s++] = (t ≪ 16) + code(3,1,0,0), t −= del;
  goto r2c1d2p3;
case code(3,1,1,2): r3c1d1p2: north(3,1);
  if (t ≤ del) { if (t ≡ del) goto win; else goto r3c1d0p2; }
  bord(3,1) = piece, stack[s++] = (t ≪ 16) + code(3,1,0,2), t −= del;
  goto r2c1d2p3;
case tailcode(3,2,1): case code(3,2,1,0): r3c2d1p0: north(3,2);
  if (t ≤ del) { if (t ≡ del) goto win; else goto r3c2d0p0; }
  bord(3,2) = piece, stack[s++] = (t ≪ 16) + code(3,2,0,0), t −= del;
  goto r2c2d2p3;
case code(3,2,1,2): r3c2d1p2: north(3,2);
  if (t ≤ del) { if (t ≡ del) goto win; else goto r3c2d0p2; }
  bord(3,2) = piece, stack[s++] = (t ≪ 16) + code(3,2,0,2), t −= del;
  goto r2c2d2p3;
case tailcode(3,3,1): case code(3,3,1,2): r3c3d1p2: north(3,3);
  if (t ≤ del) { if (t ≡ del) goto win; else goto r3c3d2p2; }
  bord(3,3) = piece, stack[s++] = (t ≪ 16) + code(3,3,2,2), t −= del;
  goto r2c3d2p3;
```

This code is used in section 8.

19. ⟨ Cases that move south from row 0 19 ⟩ ≡

```
case tailcode(0,0,3): case code(0,0,3,0): r0c0d3p0: south(0,0);
  if (t ≤ del) { if (t ≡ del) goto win; else goto r0c0d0p0; }
  bord(0,0) = piece, stack[s++] = (t ≪ 16) + code(0,0,0,0), t −= del;
  goto r1c0d0p1;
case code(0,1,3,0): r0c1d3p0: south(0,1);
  if (t ≤ del) { if (t ≡ del) goto win; else goto r0c1d2p0; }
  bord(0,1) = piece, stack[s++] = (t ≪ 16) + code(0,1,2,0), t −= del;
  goto r1c1d0p1;
case tailcode(0,1,3): case code(0,1,3,2): r0c1d3p2: south(0,1);
  if (t ≤ del) { if (t ≡ del) goto win; else goto r0c1d2p2; }
  bord(0,1) = piece, stack[s++] = (t ≪ 16) + code(0,1,2,2), t −= del;
  goto r1c1d0p1;
case code(0,2,3,0): r0c2d3p0: south(0,2);
  if (t ≤ del) { if (t ≡ del) goto win; else goto r0c2d2p0; }
  bord(0,2) = piece, stack[s++] = (t ≪ 16) + code(0,2,2,0), t −= del;
  goto r1c2d0p1;
case tailcode(0,2,3): case code(0,2,3,2): r0c2d3p2: south(0,2);
  if (t ≤ del) { if (t ≡ del) goto win; else goto r0c2d2p2; }
  bord(0,2) = piece, stack[s++] = (t ≪ 16) + code(0,2,2,2), t −= del;
  goto r1c2d0p1;
case tailcode(0,3,3): case code(0,3,3,2): r0c3d3p2: south(0,3);
  if (t ≤ del) { if (t ≡ del) goto win; else goto r0c3d2p2; }
  bord(0,3) = piece, stack[s++] = (t ≪ 16) + code(0,3,2,2), t −= del;
  goto r1c3d3p1;
```

This code is used in section 8.

20. ⟨ Cases that move south from row 1 20 ⟩ ≡

```
case code(1,0,3,0): r1c0d3p0: south(1,0);
  if (t ≤ del) { if (t ≡ del) goto win; else goto r1c0d1p0; }
  bord(1,0) = piece, stack[s++] = (t ≪ 16) + code(1,0,1,0), t −= del;
  goto r2c0d0p1;
case tailcode(1,0,3): case code(1,0,3,1): r1c0d3p1: south(1,0);
  if (t ≤ del) { if (t ≡ del) goto win; else goto r1c0d1p1; }
  bord(1,0) = piece, stack[s++] = (t ≪ 16) + code(1,0,1,1), t −= del;
  goto r2c0d0p1;
case code(1,1,3,0): r1c1d3p0: south(1,1);
  if (t ≤ del) { if (t ≡ del) goto win; else goto r1c1d2p0; }
  bord(1,1) = piece, stack[s++] = (t ≪ 16) + code(1,1,2,0), t −= del;
  goto r2c1d0p1;
case code(1,1,3,1): r1c1d3p1: south(1,1);
  if (t ≤ del) { if (t ≡ del) goto win; else goto r1c1d2p1; }
  bord(1,1) = piece, stack[s++] = (t ≪ 16) + code(1,1,2,1), t −= del;
  goto r2c1d0p1;
case tailcode(1,1,3): case code(1,1,3,2): r1c1d3p2: south(1,1);
  if (t ≤ del) { if (t ≡ del) goto win; else goto r1c1d2p2; }
  bord(1,1) = piece, stack[s++] = (t ≪ 16) + code(1,1,2,2), t −= del;
  goto r2c1d0p1;
case code(1,2,3,0): r1c2d3p0: south(1,2);
  if (t ≤ del) { if (t ≡ del) goto win; else goto r1c2d2p0; }
  bord(1,2) = piece, stack[s++] = (t ≪ 16) + code(1,2,2,0), t −= del;
  goto r2c2d0p1;
case code(1,2,3,1): r1c2d3p1: south(1,2);
  if (t ≤ del) { if (t ≡ del) goto win; else goto r1c2d2p1; }
  bord(1,2) = piece, stack[s++] = (t ≪ 16) + code(1,2,2,1), t −= del;
  goto r2c2d0p1;
case tailcode(1,2,3): case code(1,2,3,2): r1c2d3p2: south(1,2);
  if (t ≤ del) { if (t ≡ del) goto win; else goto r1c2d2p2; }
  bord(1,2) = piece, stack[s++] = (t ≪ 16) + code(1,2,2,2), t −= del;
  goto r2c2d0p1;
case code(1,3,3,1): r1c3d3p1: south(1,3);
  if (t ≤ del) { if (t ≡ del) goto win; else goto r1c3d2p1; }
  bord(1,3) = piece, stack[s++] = (t ≪ 16) + code(1,3,2,1), t −= del;
  goto r2c3d3p1;
case tailcode(1,3,3): case code(1,3,3,2): r1c3d3p2: south(1,3);
  if (t ≤ del) { if (t ≡ del) goto win; else goto r1c3d2p2; }
  bord(1,3) = piece, stack[s++] = (t ≪ 16) + code(1,3,2,2), t −= del;
  goto r2c3d3p1;
```

This code is used in section 8.

21. Yes, this has been boring. But now we're in the last such section.

⟨ Cases that move south from row 2 21 ⟩ ≡

```
case code(2,0,3,0): r2c0d3p0: south(2,0);
  if (t ≤ del) { if (t ≡ del) goto win; else goto r2c0d1p0; }
  bord(2,0) = piece, stack[s++] = (t ≪ 16) + code(2,0,1,0), t −= del;
  goto r3c0d0p1;
case tailcode(2,0,3): case code(2,0,3,1): r2c0d3p1: south(2,0);
  if (t ≤ del) { if (t ≡ del) goto win; else goto r2c0d1p1; }
  bord(2,0) = piece, stack[s++] = (t ≪ 16) + code(2,0,1,1), t −= del;
  goto r3c0d0p1;
case code(2,1,3,0): r2c1d3p0: south(2,1);
  if (t ≤ del) { if (t ≡ del) goto win; else goto r2c1d2p0; }
  bord(2,1) = piece, stack[s++] = (t ≪ 16) + code(2,1,2,0), t −= del;
  goto r3c1d0p1;
case code(2,1,3,1): r2c1d3p1: south(2,1);
  if (t ≤ del) { if (t ≡ del) goto win; else goto r2c1d2p1; }
  bord(2,1) = piece, stack[s++] = (t ≪ 16) + code(2,1,2,1), t −= del;
  goto r3c1d0p1;
case tailcode(2,1,3): case code(2,1,3,2): r2c1d3p2: south(2,1);
  if (t ≤ del) { if (t ≡ del) goto win; else goto r2c1d2p2; }
  bord(2,1) = piece, stack[s++] = (t ≪ 16) + code(2,1,2,2), t −= del;
  goto r3c1d0p1;
case code(2,2,3,0): r2c2d3p0: south(2,2);
  if (t ≤ del) { if (t ≡ del) goto win; else goto r2c2d2p0; }
  bord(2,2) = piece, stack[s++] = (t ≪ 16) + code(2,2,2,0), t −= del;
  goto r3c2d0p1;
case code(2,2,3,1): r2c2d3p1: south(2,2);
  if (t ≤ del) { if (t ≡ del) goto win; else goto r2c2d2p1; }
  bord(2,2) = piece, stack[s++] = (t ≪ 16) + code(2,2,2,1), t −= del;
  goto r3c2d0p1;
case tailcode(2,2,3): case code(2,2,3,2): r2c2d3p2: south(2,2);
  if (t ≤ del) { if (t ≡ del) goto win; else goto r2c2d2p2; }
  bord(2,2) = piece, stack[s++] = (t ≪ 16) + code(2,2,2,2), t −= del;
  goto r3c2d0p1;
case code(2,3,3,1): r2c3d3p1: south(2,3);
  if (t ≤ del) { if (t ≡ del) goto win; else goto r2c3d2p1; }
  bord(2,3) = piece, stack[s++] = (t ≪ 16) + code(2,3,2,1), t −= del;
  goto r3c3d2p1;
case tailcode(2,3,3): case code(2,3,3,2): r2c3d3p2: south(2,3);
  if (t ≤ del) { if (t ≡ del) goto win; else goto r2c3d2p2; }
  bord(2,3) = piece, stack[s++] = (t ≪ 16) + code(2,3,2,2), t −= del;
  goto r3c3d2p1;
```

This code is used in section 8.

22. The cases (r, c, d, p) with $d = p$ represent the times when we are backtracking and must restore the previous board position.

⟨ Cases that move back 22 ⟩ ≡

case $code(0,0,0,0)$: *r0c0d0p0*: $bord(0,0) = bord(0,1)$; **goto** *switcher*;
case $code(0,0,3,3)$: *r0c0d3p3*: $bord(0,0) = bord(1,0)$; **goto** *switcher*;
case $code(0,1,0,0)$: *r0c1d0p0*: $bord(0,1) = bord(0,2)$; **goto** *switcher*;
case $code(0,1,2,2)$: *r0c1d2p2*: $bord(0,1) = bord(0,0)$; **goto** *switcher*;
case $code(0,1,3,3)$: *r0c1d3p3*: $bord(0,1) = bord(1,1)$; **goto** *switcher*;
case $code(0,2,0,0)$: *r0c2d0p0*: $bord(0,2) = bord(0,3)$; **goto** *switcher*;
case $code(0,2,2,2)$: *r0c2d2p2*: $bord(0,2) = bord(0,1)$; **goto** *switcher*;
case $code(0,2,3,3)$: *r0c2d3p3*: $bord(0,2) = bord(1,2)$; **goto** *switcher*;
case $code(0,3,2,2)$: *r0c3d2p2*: $bord(0,3) = bord(0,2)$; **goto** *switcher*;
case $code(0,3,3,3)$: *r0c3d3p3*: $bord(0,3) = bord(1,3)$; **goto** *switcher*;
case $code(1,0,0,0)$: *r1c0d0p0*: $bord(1,0) = bord(1,1)$; **goto** *switcher*;
case $code(1,0,1,1)$: *r1c0d1p1*: $bord(1,0) = bord(0,0)$; **goto** *switcher*;
case $code(1,0,3,3)$: *r1c0d3p3*: $bord(1,0) = bord(2,0)$; **goto** *switcher*;
case $code(1,1,0,0)$: *r1c1d0p0*: $bord(1,1) = bord(1,2)$; **goto** *switcher*;
case $code(1,1,1,1)$: *r1c1d1p1*: $bord(1,1) = bord(0,1)$; **goto** *switcher*;
case $code(1,1,2,2)$: *r1c1d2p2*: $bord(1,1) = bord(1,0)$; **goto** *switcher*;
case $code(1,1,3,3)$: *r1c1d3p3*: $bord(1,1) = bord(2,1)$; **goto** *switcher*;
case $code(1,2,0,0)$: *r1c2d0p0*: $bord(1,2) = bord(1,3)$; **goto** *switcher*;
case $code(1,2,1,1)$: *r1c2d1p1*: $bord(1,2) = bord(0,2)$; **goto** *switcher*;
case $code(1,2,2,2)$: *r1c2d2p2*: $bord(1,2) = bord(1,1)$; **goto** *switcher*;
case $code(1,2,3,3)$: *r1c2d3p3*: $bord(1,2) = bord(2,2)$; **goto** *switcher*;
case $code(1,3,1,1)$: *r1c3d1p1*: $bord(1,3) = bord(0,3)$; **goto** *switcher*;
case $code(1,3,2,2)$: *r1c3d2p2*: $bord(1,3) = bord(1,2)$; **goto** *switcher*;
case $code(1,3,3,3)$: *r1c3d3p3*: $bord(1,3) = bord(2,3)$; **goto** *switcher*;
case $code(2,0,0,0)$: *r2c0d0p0*: $bord(2,0) = bord(2,1)$; **goto** *switcher*;
case $code(2,0,1,1)$: *r2c0d1p1*: $bord(2,0) = bord(1,0)$; **goto** *switcher*;
case $code(2,0,3,3)$: *r2c0d3p3*: $bord(2,0) = bord(3,0)$; **goto** *switcher*;
case $code(2,1,0,0)$: *r2c1d0p0*: $bord(2,1) = bord(2,2)$; **goto** *switcher*;
case $code(2,1,1,1)$: *r2c1d1p1*: $bord(2,1) = bord(1,1)$; **goto** *switcher*;
case $code(2,1,2,2)$: *r2c1d2p2*: $bord(2,1) = bord(2,0)$; **goto** *switcher*;
case $code(2,1,3,3)$: *r2c1d3p3*: $bord(2,1) = bord(3,1)$; **goto** *switcher*;
case $code(2,2,0,0)$: *r2c2d0p0*: $bord(2,2) = bord(2,3)$; **goto** *switcher*;
case $code(2,2,1,1)$: *r2c2d1p1*: $bord(2,2) = bord(1,2)$; **goto** *switcher*;
case $code(2,2,2,2)$: *r2c2d2p2*: $bord(2,2) = bord(2,1)$; **goto** *switcher*;
case $code(2,2,3,3)$: *r2c2d3p3*: $bord(2,2) = bord(3,2)$; **goto** *switcher*;
case $code(2,3,1,1)$: *r2c3d1p1*: $bord(2,3) = bord(1,3)$; **goto** *switcher*;
case $code(2,3,2,2)$: *r2c3d2p2*: $bord(2,3) = bord(2,2)$; **goto** *switcher*;
case $code(2,3,3,3)$: *r2c3d3p3*: $bord(2,3) = bord(3,3)$; **goto** *switcher*;
case $code(3,0,0,0)$: *r3c0d0p0*: $bord(3,0) = bord(3,1)$; **goto** *switcher*;
case $code(3,0,1,1)$: *r3c0d1p1*: $bord(3,0) = bord(2,0)$; **goto** *switcher*;
case $code(3,1,0,0)$: *r3c1d0p0*: $bord(3,1) = bord(3,2)$; **goto** *switcher*;
case $code(3,1,1,1)$: *r3c1d1p1*: $bord(3,1) = bord(2,1)$; **goto** *switcher*;
case $code(3,1,2,2)$: *r3c1d2p2*: $bord(3,1) = bord(3,0)$; **goto** *switcher*;
case $code(3,2,0,0)$: *r3c2d0p0*: $bord(3,2) = bord(3,3)$; **goto** *switcher*;
case $code(3,2,1,1)$: *r3c2d1p1*: $bord(3,2) = bord(2,2)$; **goto** *switcher*;
case $code(3,2,2,2)$: *r3c2d2p2*: $bord(3,2) = bord(3,1)$; **goto** *switcher*;
case $code(3,3,1,1)$: *r3c3d1p1*: $bord(3,3) = bord(2,3)$; **goto** *switcher*;
case $code(3,3,2,2)$: *r3c3d2p2*: $bord(3,3) = bord(3,2)$; **goto** *switcher*;

This code is used in section 8.

23. Great — all the code has now been written for the inner loop. Only two things remain: (1) Getting the whole process rolling, and (2) eventually stopping the darn thing.

To get started, we need 2, 3, or 4 top-level (or should I say bottom-of-stack level) decisions about what direction to choose for the first move. The *tailcode*s were invented for precisely that purpose.

The process stops in one of two ways: Either there's no solution (and we have try again with a higher quota), or a winning path was found. It's easy to handle the first case without slowing anything down, by simply putting the special code '*bottom*' at the bottom of the stack.

And I'll save the other case (the best case) for last.

```
⟨ Try for a solution with t more moves 23 ⟩ ≡
  for (j = 0; j < 16; j++) {
    board[j] = start[j];
    if (board[j] < 0) k = j;
  }
  stack[0] = (t ≪ 16) + bottom, s = 1;
  if (col(k) ≠ 3) stack[s++] = (t ≪ 16) + tailcode(row(k), col(k), 0);      /* first move east */
  if (row(k) ≠ 0) stack[s++] = (t ≪ 16) + tailcode(row(k), col(k), 1);      /* first move north */
  if (col(k) ≠ 0) stack[s++] = (t ≪ 16) + tailcode(row(k), col(k), 2);      /* first move west */
  if (row(k) ≠ 3) stack[s++] = (t ≪ 16) + tailcode(row(k), col(k), 3);      /* first move south */
  ⟨ Switch into action 8 ⟩;
```

This code is used in section 5.

24. Hey, we have a winner! The stack entries tell us how we got here, so that we can easily give the user instructions about which piece should be pushed at each step — except that the last two steps haven't been placed on the stack, because of my tricky optimizations.

At first I was going to leave those two steps as an exercise for the user. Ha, ha, ha, a hilarious joke. But, well, it looked a little too silly. So I broke down and figured out how to deduce them from scratch.

```
⟨ Output the results 24 ⟩ ≡
  printf("Solution␣in␣%d+%d␣moves:␣", moves & #ff, moves ≫ 8);
  if (moves > 1) {
    for (k = 1; (stack[k] & #ffff) ≥ tailcode(0, 0, 0); k++) ;
    for (j = 0; j < 16; j++) board[j] = start[j];
    j = (stack[k] & #ffff) ≫ 4;
    for (k++; k < s; k++) {
      del = (stack[k] & #ffff) ≫ 4;
      printf("%x", board[del] + 1);
      board[j] = board[del], j = del;
    }
    if (j ≡ #d) printf("ef\n");
    else if (j ≡ #7) printf("8c\n");
    else printf("b%x\n", bord(3, 3) + 1);
  } else {
    if (moves ≡ 1) printf("%x", start[#f] + 1);
    printf("␣(%d␣sec)\n", time(0) − timer);
  }
```

This code is used in section 1.

25. Index.

15PUZZLE-KORF1