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

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 (well, really the zeroth) of a series of ever-more-efficient ways to do the job. My main reason for writing this group of routines was to experiment with a new (for me) style of programming, explained in 15PUZZLE-KORF1.

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];
  short tim[100];
  char dir[100], pos[100];
  int timer;
  main(int argc, char *argv[])
  {
    register int j, k, s, t, l, d, p, q, del, piece, moves;
    ⟨ Input the initial position 2 ⟩;
    ⟨ Apply Korf's procedure 4 ⟩;
    ⟨ Output the results 11 ⟩;
  }
```

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 2000 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.

4. Saving memory means that the computation lives entirely within the computer's high-speed cache. Furthermore, we can optimize the inner loop, as shown in the 15PUZZLE-KORF1, the next program in this series. But here I'm using the most straightforward implementation, so that it will be possible to test if more sophisticated ideas actually do save time on a real machine.

```
⟨ Apply Korf's procedure 4 ⟩ ≡
  ⟨ Set moves to the minimum number of happy moves 5 ⟩;
  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 6 ⟩;
    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.

5. ⟨ Set *moves* to the minimum number of happy moves 5 ⟩ ≡

```
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 4.

6. The main control routine is typical for backtracking. Directions are encoded as follows: east = 0, north = 1, west = 2, south = 3. (Think of powers of i in the complex plane.)

When a move is legal, we set $del = {}^{\#}\texttt{1}$ if the move is happy, $del = {}^{\#}\texttt{100}$ if the move is sad. Then t becomes $t - del$ after the move has been made; but we can't make the move if $t < del$.

One subtlety occurs here that is worth noting: If t is equal to del, we must have $del = {}^{\#}\texttt{1}$ and be at the goal. Reason: We can't have $t = {}^{\#}\texttt{100}$, because the right component of t never is less than the left component. For example, suppose we're trying to solve the problem with 41 happy moves and 4 sad moves. If we could get to $t = {}^{\#}\texttt{100}$, we would have taken 41 happy moves and 3 sad moves, so our distance to the goal would be negative.

⟨ Try for a solution with t more moves 6 ⟩ ≡

```
  for (j = 0; j < 16; j++) {
    board[j] = start[j];
    if (board[j] < 0) p = j;
  }
  pos[0] = 16, l = 1;
newlevel: d = 0, tim[l] = t, pos[l] = p, q = pos[l − 1];
trymove: switch (d) {
  case 0: if (col(p) ≤ 2 ∧ q ≠ p + 1) ⟨ Prepare to move east, then break 7 ⟩;
    d++;      /* fall through to next case */
  case 1: if (row(p) ≥ 1 ∧ q ≠ p − 4) ⟨ Prepare to move north, then break 8 ⟩;
    d++;      /* fall through to next case */
  case 2: if (col(p) ≥ 1 ∧ q ≠ p − 1) ⟨ Prepare to move west, then break 9 ⟩;
    d++;      /* fall through to next case */
  case 3: if (row(p) ≤ 2 ∧ q ≠ p + 4) ⟨ Prepare to move south, then break 10 ⟩;
    d++;      /* fall through to next case */
  case 4: goto backtrack;
  }      /* at this point q is the new empty cell position */
       /* and del has been set to #1 or #100 as described above */
  if (t ≤ del) {
    if (t ≡ del) goto win;
    d++; goto trymove;
  }
  dir[l] = d, board[p] = board[q], t −= del, p = q, l++;
  goto newlevel;
backtrack: if (l > 1) {
    l−−;
    q = pos[l], board[p] = board[q], p = q, q = pos[l − 1], t = tim[l], d = dir[l] + 1;
    goto trymove;
  }
```

This code is used in section 4.

7. We're going to move the *empty* cell east, by moving *piece* west.

⟨ Prepare to move east, then **break** 7 ⟩ ≡

```
{
  q = p + 1, piece = board[q];
  del = (col(piece) < col(q) ? #1 : #100);
  break;
}
```

This code is used in section 6.

8. ⟨ Prepare to move north, then **break** 8 ⟩ ≡

```
{
  q = p − 4, piece = board[q];
  del = (row(piece) > row(q) ? #1 : #100);
  break;
}
```

This code is used in section 6.

9. ⟨ Prepare to move west, then **break** 9 ⟩ ≡

```
{
  q = p − 1, piece = board[q];
  del = (col(piece) > col(q) ? #1 : #100);
  break;
}
```

This code is used in section 6.

10. ⟨ Prepare to move south, then **break** 10 ⟩ ≡

```
{
  q = p + 4, piece = board[q];
  del = (row(piece) < row(q) ? #1 : #100);
  break;
}
```

This code is used in section 6.

11. Hey, we have a winner! The *pos* 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.

⟨ Output the results 11 ⟩ ≡

```
pos[l + 1] = q;
printf("Solution␣in␣%d+%d␣moves:␣", moves & #ff, moves ≫ 8);
if (moves > 0) {
  for (j = 0; j < 16; j++) board[j] = start[j];
  for (k = 1; k ≤ l; k++) {
    printf("%x", board[pos[k + 1]] + 1);
    board[pos[k]] = board[pos[k + 1]];
  }
  printf("␣(%d␣sec)\n", time(0) − timer);
}
```

This code is used in section 1.

12. Index.

⟨ Apply Korf's procedure 4 ⟩ Used in section 1.
⟨ Input the initial position 2 ⟩ Used in section 1.
⟨ Output the results 11 ⟩ Used in section 1.
⟨ Prepare to move east, then **break** 7 ⟩ Used in section 6.
⟨ Prepare to move north, then **break** 8 ⟩ Used in section 6.
⟨ Prepare to move south, then **break** 10 ⟩ Used in section 6.
⟨ Prepare to move west, then **break** 9 ⟩ Used in section 6.
⟨ Set *moves* to the minimum number of happy moves 5 ⟩ Used in section 4.
⟨ Try for a solution with t more moves 6 ⟩ Used in section 4.

15PUZZLE-KORF0