

1 The Towers of Hanoi

Good puzzles provide an excellent way to log in to the realm of abstract thought inhabited by mathematicians and other theorists. The best puzzles embody themes from this realm; the significance of such themes extends considerably beyond the puzzles themselves.

One such classic puzzle, the *Towers of Hanoi*, suggest two pairs of contrasting themes: recursion and iteration, unity and diversity. Apart from such serious considerations, the puzzle is fun and also provides the neophyte with a satisfying sense of confusion, hallmark of his or her slow entry into the realm of abstract thought.

The Towers of Hanoi consist of three vertical pegs set in a board. A number of disks, graded in size, are initially stacked on one of the pegs so that the smallest disk is uppermost, as shown in Figure 1. The aim of the puzzle is to transfer all the disks from the initial peg to one of the other two pegs. The disks are manipulated according to these two simple rules:

1. Move one disk at a time from one peg to another.
2. No disk may be placed on top of a smaller disk.

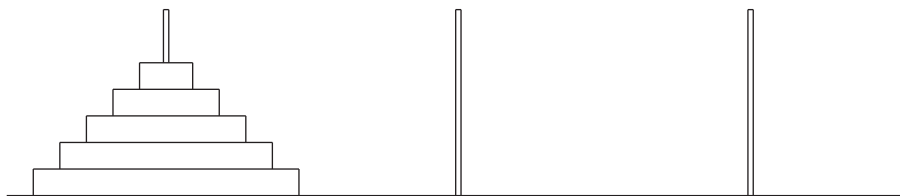


Figure 1: Initial position

The smallest disk must be moved first since it is the only one that is initially accessible. On the next turn there are two moves for the smallest disk (both pointless) and one move for the second-smallest-disk. It goes onto the unoccupied peg since it cannot be placed on top of the smallest disk (Rule 2). On the third turn it is not quite so obvious what to do: should the second disk be returned to the initial peg or should the first disk be moved again—and if so, onto what peg?

From this point on one is faced with a long succession of moves and with many opportunities for wrong choices. Even if all the right choices are made, $2^n - 1$ moves are needed (as we shall see below) to relocate a tower of n disks, one at a time, onto another peg. The surprisingly long time required to solve a puzzle made up of even a moderate number of disks is well illustrated by the following tale quoted from W. W. Rouse Ball's classic puzzle book, *Mathematical Recreations and Essays*:

In the great temple of Benares. . . beneath the dome which marks the centre of the world, rests a brass plate in which are fixed three

diamond needles, each a cubit high and as thick as the body of a bee. On one of these needles, at the creation, God placed sixty-four discs of pure gold, the largest disc resting on the brass plate and the others getting smaller and smaller up to the top one. This is the Tower of Bramah. Day and night unceasingly the priests transfer the discs from one diamond needle to another according to the fixed and immutable laws of Bramah, which require that the priest on duty must not move more than one disc at a time and that he must place this disc on a needle so that there is no smaller disc below it. When the sixty-four discs shall have been thus transferred from the needle on which at the creation God placed them to one of the other needles, tower, temple, and Brahmins alike will crumble into dust, and with a thunderclap the world will vanish.

That the world has not yet vanished attests to the extreme length of time it takes to solve the puzzle: even if the priests move one disk every second, it would take more than 500 billion years to relocate the initial tower of 64 disks!

At this point (and at no risk to the universe) the reader can involve himself or herself more directly by picking up five playing cards, for example the ace through five of hearts, and visualizing three spots on a table. Stack the cards on one of the spots, in order, so that the ace is on top. It is now possible to attempt a solution to the five-disk tower puzzle by moving one card at a time between two spots—but never place a card on one of lower value. Can you complete the relocation of the five-card tower before the end of the world? According to the formula $2^5 - 1$, the transfer should be possible in 31 moves.

Of course, if you're reading this, you're more likely to reach for an HP graphing calculator than a deck of playing cards. So, let's do just that.

2 Solving the Towers of Hanoi on the HP 50g

One technique for solving this problem is a strategy commonly called “divide-and-conquer.” It consists of breaking a problem of size n into smaller problems in such a way that from solutions to smaller problems we can easily construct a solution to the entire problem.

The problem of moving the n smallest disks from A to C can be thought of as consisting of two subproblems of size $n - 1$. First move the $n - 1$ smallest disks from A to B , exposing the n^{th} smallest disk on A . Move that disk from A to C . Then move the $n - 1$ smallest disks from B to C .

So how do we move the $n - 1$ smallest disks from A to B ?

Well, we can do that by moving the $n - 2$ smallest disks from A to C , exposing the $(n - 1)^{th}$ smallest disk on A , moving that disk to B , then moving the $n - 2$ smallest disks from C to B .

See the pattern here?

Moving all n disks is accomplished by a recursive application of the method. As the n disks involved in the moves are smaller than any other disks, we need

not concern ourselves with what lies below them on pegs *A*, *B*, or *C*. Although the actual movement of individual disks is not obvious, and hand simulation is hard because of the stacking of recursive calls, the algorithm is conceptually simple to understand, to prove correct and, we would like to think, to invent in the first place. It is probably the ease of discovery of divide-and-conquer algorithms that makes the technique so important.

2.1 Output

Solving the puzzle does us no good if we can't see the results. One way to do this would be to output a sequence of moves. This can be done by inserting the code like the following into the routine that moves one disk:

```
from "→" + to + 1 DISP
```

The problem with this approach is that the moves flash by too fast to be of much use. To get around this, we could leave the moves on the stack or accumulate them into a single string (or a list).

However, it would be more entertaining to have the program actually display the three pegs and the disks, showing the disks being shuffled from one peg to another. Even with the limited graphics capabilities of the HP 50g, this is not very difficult to implement. For each move we need to erase the top disk on the “from” peg before it is moved, and then after the move draw it on top of the “to” peg.

The HP 50g screen is 131 pixels wide by 80 pixels high. This gives us about 44 pixels per peg. At 3 pixels per disk, we can stack up to 27 disks on each peg. This in turn means we can make each disk 1 pixels wider than the one above it. If we identify each disk by its width, the code to draw one disk looks like this:

```
Draw @Draw/erase a disk width w at location (x, y)@
@<<
+ x w y
<<
y DUP 1 + FOR j
j R→B x R→B OVER 2 →LIST x w + R→B ROT 2 →LIST TLINE
NEXT
»
»
```

The use of the TLINE instruction ensures that the disk is erased if it is already there and drawn otherwise. Note also that the disks thus drawn are only two pixels thick; this lets us leave one row of blank pixels between disks.

2.2 Data Structures

Before we can write a program to solve the Towers of Hanoi puzzle, we need to figure out our data structures.

If we were to simply output a list of moves, then there isn't much data to keep track of: everything can be handled by the arguments of the recursive calls to a routine like this:

```
MoveN @ Move n disks from peg from to peg to using peg using @
«
  + n from using to
  «
    IF n 1 == THEN
      from to MoveOne
    ELSE
      n 1 - from using to MoveN
      from to MoveOne
      n 1 - using to from MoveN
    END
  »
»
```

However, we want to do more than just list the moves, we want to display them on the screen. In order to do this we need to keep track of the set of disks on each of the three pegs. For this program we'll keep it simple and merely maintain a list containing three sublists, each sublist containing the disks on one peg, the individual disks identified by size. This data structure can be built with RPL code like this:

```
'I' 'I' 1 n 1 SEQ ( ) DUP 3 →LIST
```

But as you'll see later on, we'll actually build the stack one disk at a time. For now, just be aware that we have a list of three lists.

2.3 Algorithms

Now that we have our sole data structure defined, we need some algorithms for operating on it.

2.3.1 Move One Disk

First let's consider the case of moving a single disk from one peg to another.

```
MoveOne @ Move one disk from peg from to peg to @
«
+ from to
«
  Pegs from GET DUP HEAD SWAP TAIL 'Pegs' from ROT PUT
  Pegs to GET + 'Pegs' to ROT PUT
»
»
```

Pegs is our list of three lists, mentioned earlier.

2.3.2 Move n Disks

We've already seen this recursive routine, the heart of the program, in Section 2.2, so we won't bother duplicating it here.

2.3.3 Display Management

Before drawing anything on the screen, we want to make sure it's blank. We also want to make sure it's being displayed, so the calculator's hard work doesn't go to waste.

```
ERASE ( # 0d # 0d ) PVIEW
```

In order to make the display work properly, we need to make sure that the disks are drawn in their initial positions. We also want to identify each disk by its width. Both are accomplished in a single loop:

```
1.  n FOR j
    30.  j - 3. * Pegs HEAD + 'Pegs' 1. ROT PUT 1. Draw
NEXT
```

Once the program has finished solving the puzzle, it would be a shame to have the display revert to the usual stack display before the user could admire the results of the calculator's efforts. We can keep the solved puzzle on the display until the user presses a key with the following code:

```
7.  FREEZE
```

2.4 Putting It All Together

Putting together all the pieces we have the complete program:

```
HANOI1
« @ Move n disks from 1 to 3 using 2 @
IP 27. MIN 1. MAX
( @ Move n disks from f to t using u @
  → n t f u
  «
    IF n 0. > THEN
      n 1. - u f t +h EVAL
      f +d EVAL
      +1 f GET DUP HEAD '+1' f 4. ROLL TAIL PUT
      +1 t GET + '+1' t ROT PUT t +d EVAL
      n 1. - t u f +h EVAL
    END
  »
)
( @ Draw/erase top disk on peg @
DUP 1. - 44. * +1 ROT GET DUP SIZE SWAP HEAD PICK3 +
→ x y w
«
  81. y 3. * - DUP 1. + FOR j
  j R→B x R→B OVER 2. →LIST w R→B ROT 2. →LIST TLINE
NEXT
»
)
@ Main program: move n disks from 1 to 3 using 2 @
( ( ) ( ) ( ) )
→ n +h +m +d +1
«
  ERASE ( # 0d # 0d ) PVIEW
  1 n. FOR j
    30. j - 3. * +1 HEAD + '+1' 1. ROT PUT 1. +d EVAL
  NEXT
  n 3. 1. 2. +h EVAL
  7. FREEZE
»
»
```

Note some final touches have been added. The program argument is converted to an integer if necessary and coerced into the range [1...13]. All of the subroutines are converted to lists and put into local variables within the main program, and all the names have been shortened to save space. Additional optimizations are also carried out.

3 Running the Program

Two programs are included in the archive. `HANOI1` is the UserRPL program shown above in Section 2.4. `HANOI2` is a SysRPL implementation of the same algorithm.

Both programs are run identically. Simply place a real number representing the desired number of disks and then press the soft key corresponding to the stored program. The program will first stack the disks on the left of the screen and then proceed to solve the puzzle. Once the puzzle is completed, the program will halt with the disks stacked on the right of the screen. Simply press any key to return to the stack display.

The SysRPL version is less than half the size of the UserRPL version and runs in about a third of the time, amply demonstrating the advantages of SysRPL.