The Towers of Hanoi

Kiyoshi Akima
k_akima@hotmail.com

2011.04.23

Contents

1 The Towers of Hanoi Puzzle **1**

2 Solving the Towers of Hanoi **2**
2.1 Recursion—Divide-and-Conquer 2
2.2 Iteration 3
2.2.1 Alternation 3
2.2.2 Binary 3
2.2.3 Gray Code 3

3 The Programs **4**
3.1 RPL (HP 48G/49g/50g) 5
3.2 System RPL (HP 48G/49g/50g) 6
3.3 RPN 6
3.3.1 HP 35s 6
3.3.2 HP 30b 8
3.3.3 HP-16C 9
3.3.4 HP-15C 11
3.4 HP-BASIC (HP 38g/39g/40g) 13

1 The Towers of Hanoi Puzzle

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.

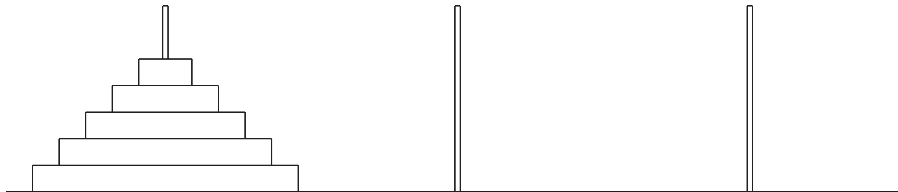

Figure 1: Initial position

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 snallest 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

How does one go about solving a puzzle such as this one? Well, there are several ways, and this document will only touch upon some of them.

2.1 Recursion—Divide-and-Conquer

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 nth 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.

2.2 Iteration

It's also possible to solve the Towers of Hanoi puzzle without using recursion.

2.2.1 Alternation

Imagine the pegs arranged in a triangle. On odd-numbered moves, move the smallest disk one peg clockwise. On even-numbered moves make the only legal move not involving the smallest disk.

The above algorithm is consise, and correct. But in contrast to the earlier divide-and-conquer algorithm, it is hard to understand how it works, and hard to invent on the spur of the moment.

2.2.2 Binary

The source and destination pegs for the mth move can also be found elegantly from the binary representation of m using bitwise operations. To use the syntax of the C programming language, the mth move is from peg `(m&m-1)%3` to peg `((m|m-1)+1)%3`, where the disks begin on peg 0 and finish on peg 1 or 2 according as whether the number of disks is even or odd.

2.2.3 Gray Code

There is yet another algorithm for solving the Towers of Hanoi. If one numbers the disks $1, 2, 3, \ldots$ up to n in the usual manner from smallest to largest, it turns out that each move in the puzzle's solution is indicated by a binary number. For example, to solve the five-disk puzzle here for illustrative purposes, we would list the five-bit binary numbers in the usual order of counting. The first nine five-bit binary numbers are:

decimal	binary	(disk)
0	00000	
1	00001	(1)
2	00010	(2)
3	00011	(1)
4	00100	(3)
5	00101	(1)
6	00110	(2)
7	00111	(1)
8	01000	(4)

Each binary number that has a predecessor in the sequence also has exactly one bit that has just changed from a 0 to a 1. The position of this bit (counting from the right) is given by the decimal number written in parenthesis beside the binary one. These numbers are also the numbers of the first eight disks moved; the correspondence holds throughout the standard solution sequence.

There is an ambiguity in this solution. When we are to move the smallest disk, there are two possible places to put it. However, this is solved in exactly the same way as in the first iterative algorithm: always move the disk in the same direction around the imagined triangle.

3 The Programs

This document presents a collection of programs for solving the Towers of Hanoi puzzle. I make no attempt to cover every programming language on Earth, instead concentrating on programming Hewlett-Packard handheld calculators. I only present a sample of programs, showing different methods. (Or maybe the programs merely represent some of the calculators I happen to have.) If you want to program the Towers of Hanoi for some other calculator, hopefully one or more of these programs will prove useful as a starting point (and if they don't, they don't).

I also make no claims regarding optimality. If you can make any of these programs faster and/or smaller, great. (If you can squeeze a Towers of Hanoi program into the HP-19C/29C, I'd really, *really* like to see it.) Furthermore, these programs are presented without any warranty or suitability for any purpose, expressed or implied. Should any of these programs lock up your calculator, cause loss of any data, or cause the world to vanish, don't blame me.

3.1 RPL (HP 48G/49g/50g)

The recursive algorithm (Section 2.1) leads to a nearly trivial implementation in RPL. Given a number on the stack representing the number of disks (which actually could be larger than 64), the program proceeds to display letter pairs representing the "from" and "to" pegs. The three pegs are labeled "S", "D", and "T" for "source", "destination", and "temporary" respectively.

```
«
 «
  → n s d t
  «
   IF n 0. > THEN
    n 1. - s t d ←h EVAL
    s d + 1. DISP
    n 1. - t d s ←h EVAL
   END
  »
 »
 → ←h
 «
  "S" "D" "T" ←h EVAL
 »
»
```

For machines like those in the HP 48S series that do not have compiled local variables, the embedded routine would have to be brought out as its own global routine.

The iterative algorithms produce bigger, slower programs on this calculator. This one is based on the Binary algorithm in Section 2.2.2. It works the same way as the previous one, except that the pegs are numbered "1," "2," and "3" instead of being named "S", "D", and "T" where the destination is "2" or "3" depending on whether the number of disks is even or odd.

```
«
  DEC 64 STWS
  #1d 1. ROT START
   DUP +
  NEXT #1d - #0d
  DO
   DUP #1d + SWAP OVER
   DUP2 AND DUP #3d / #3d × - #10d ×
   UNROT OR DUP #3d / #3d × - #1d + DUP #3d / #3d × -
   + #11d + 1. DISP
  UNTIL
   DUP2 ==
  END
  DROP2
»
```

3.2 System RPL (HP 48G/49g/50g)

A straightforward translation of the recursive UserRPL program into SysRPL produces a program that's two-thirds the size and nearly twice the speed. Running on the HP 50g, this is the fastest program presented in this document.

```
!NO CODE !RPL
::
 ' ::
  4NULLLAM{} BIND
  4GETLAM %0> IT ::
   4GETLAM %1- 3GETLAM 1GETLAM 2GETLAM LAM ←h
   3GETLAM 2GETLAM &$ DISPROW1
   4GETLAM %1- 1GETLAM 2GETLAM 3GETLAM LAM ←h
  ;
  ABND
 ;
 { LAM ←h } BIND
 "S" "D" "T" LAM ←h EVAL
 ABND
;
@
```

3.3 RPN

RPN calculators come in a wide range of capabilities and styles. We're not going to write programs for all of them, but hopefully we'll cover a representative sample.

3.3.1 HP 35s

The HP 35s is HP's latest advanced scientific RPN calculator.

The RPN language doesn't lend itself easily to recursion, lacking local variables and an adequate subroutine return stack. That does not mean that recursion is impossible, however. We just have to implement the recursion ourselves using our own data and return stacks.

This program actually combines both stacks into one. Variable I points to the top element on this stack. The two least significant decimal digits of the stack element are the destination and source pegs, respectively. The third digit is the number of the temporary peg. The fourth digit tells the subroutine where to return. The remainder of the stack element (which could be a number greater than 64 as far as the program is concerned) is the number of disks to be moved.

The simulated subroutine starts at line `H009`, while lines `H028`-`H036` handle figuring out where to return to when the subroutine finishes.

Given the number of disks in X, this program proceeds to display a sequence of two-digit numbers on the bottom line of the display (you can and should ignore the numbers on the top line). The tens digits represents the “from” peg, the units digits represents the “to” peg. An output of zero signals program termination.

If the output is too fast, the PSE in step H040 may be replaced with a STOP, in which case you’ll need to press R/S to resume the program.

```
 H001 LBL H
 H002 0
 H003 STO I
 H004 R↓
 H005 1E4
 H006 ×
 H007 213
 H008 +
▶H009 ISG I
 H010 ABS
 H011 STO(I)
 H012 1E4
 H013 x>y?
 H014 GTO H028
 H015 x<>y
 H016 XEQ H068
 H017 10
 H018 ×
 H019 RCL(I)
 H020 10
 H021 XEQ H067
 H022 +
 H023 10
 H024 ×
 H025 RCL(I)
 H026 1E2
 H027 GTO H056
▶H028 RCL(I)
 H029 DSE I
 H030 GTO H033
 H031 CLSTK
 H032 RTN
▶H033 1E3
 H034 XEQ H067
 H035 x≠0?
 H036 GTO H028
 H037 RCL(I)
 H038 1E2
 H039 RMDR
 H040 PSE
 H041 10
 H042 RCL(I)
 H043 10
 H044 XEQ H067
 H045 +
 H046 10
 H047 ×
 H048 RCL(I)
 H049 1E2
 H050 XEQ H067
 H051 +
 H052 10
 H053 ×
 H054 RCL(I)
 H055 1
▶H056 XEQ H067
 H057 +
 H058 RCL(I)
 H059 1E4
 H060 INT÷
 H061 1
 H062 -
 H063 1E4
 H064 ×
 H065 +
 H066 GTO H009
▶H067 ÷
▶H068 IP
 H069 10
 H070 RMDR
 H071 RTN

CK=D173
LEN=259
```

The ABS instruction in step H010 is a NOP. Arrows (▶) preceeding step numbers indicate destinations of branches, to aid in porting to a calculator that uses labels instead of line addressing.

We can also program the HP 35s to solve the puzzle using one of the iterative algorithms. Despite the fact that the calculator limits binary numbers to 36 bits, we can implement the binary algorithm (Section 2.2.2) without too much trouble: we simply have to split up the number into two registers. R_A and R_B hold the loop counter, R_C and R_D hold the current move number, and R_E and R_F hold the previous move number. We put the lower 34 bits into the first register of each pair, making use of the fact that $2^{34} \mod 3 = 1$ to avoid any further bit-twiddling.

This results in a program that’s slightly larger than the recursive version above, but runs about twice as fast.

Given the number of disks in X, this program proceeds to display a sequence of two-digit numbers on the bottom line of the display (you can and should ignore the numbers on the top line). The tens digits represents the “from” peg,

the units digits represents the “to” peg. An output of zero signals program termination.

If the output is too fast, the PSE in step I071 may be replaced with a STOP, in which case you’ll need to press R/S to resume the program.

I001 LBL I	I024 RCL G	I047 RCL C	I070 +
I002 CLVARS	▶I025 1	I048 RCL E	I071 PSE
I003 2	I026 -	I049 AND	I072 CF 0
I004 34	I027 STO A	I050 +	I073 RCL A
I005 y^x	▶I028 RCL D	I051 3	I074 x=0?
I006 STO G	I029 STO F	I052 RMDR	I075 SF 0
I007 R↓	I030 RCL C	I053 10	I076 FS? 0
I008 SF 0	I031 STO E	I054 ×	I077 RCL G
I009 34	I032 1	I055 RCL D	I078 1
I010 x≤y?	I033 +	I056 RCL F	I079 FS? 0
I011 CF 0	I034 STO C	I057 OR	I080 STO- B
I012 RMDR	I035 RCL G	I058 3	I081 -
I013 STO I	I036 x≠y?	I059 RMDR	I082 STO A
I014 1	I037 GTO I042	I060 RCL C	I083 RCL B
I015 x>y?	I038 1	I061 RCL E	I084 OR
I016 GTO I021	I039 STO+ D	I062 OR	I085 x≠0?
▶I017 ENTER	I040 CLx	I063 +	I086 GTO I028
I018 +	I041 STO C	I064 1	I087 CLSTK
I019 DSE I	▶I042 RCL D	I065 +	I088 RTN
I020 GTO I017	I043 RCL F	I066 3	
▶I021 FS? 0	I044 AND	I067 RMDR	
I022 GTO I025	I045 3	I068 +	CK=9E3C
I023 STO B	I046 RMDR	I069 11	LEN=283

3.3.2 HP 30b

The HP 30b is HP’s most recent programmable financial calculator.

Even though it’s a financial calculator, the HP 30b is perfectly capable of handling this puzzle. It’s not as if we’re computing transcendental functions or using complex arithmetic, after all.

Also, the HP 30b is blazing fast. This program is the fastest RPN program presented in this document, lagging behind only the two recursive programs for the HP 50g.

This program closely follows the scheme used by the simulated recursive HP 35s program given earlier in Section 3.3.1. So close that I won’t bother even attempting a line-by-line description. In fact, the HP 35s program was written from this HP 30b program, which was based on an earlier HP 35s version (which was based on. . .).

Enter the number of disks then start the program. The program then proceeds to display a sequence of two-digit numbers. The tens digits represents the

"from" peg, the units digits represents the "to" peg. An output of zero signals program termination.

If the output is too fast, the Disp0 in line 58 can be replaced by a slower display or even by a R/S. In the latter case, you'll have to press SHIFT+R/S to resume the program.

1	Input	29	2	57	EEX	85	RCL Data
2	0	30	Call04	58	2	86	EEX
3	STO 0	31	+	59	*	87	4
4	R↓	32	EEX	60	Disp0	88	/
5	EEX	33	1	61	EEX	89	Math
6	4	34	*	62	1	90	Up
7	*	35	RCL Data	63	RCL Data	91	Up
8	2	36	EEX	64	EEX	92	Input
9	1	37	3	65	2	93	1
10	3	38	Gto 03	66	Call04	94	-
11	+	39	Lbl 01	67	+	95	EEX
12	Lbl 00	40	RCL Data	68	EEX	96	4
13	ISG 0	41	DSE 0	69	1	97	*
14	CashFL	42	Gto 02	70	*	98	+
15	STO Data	43	ON	71	RCL Data	99	Gto 00
16	EEX	44	Stop	72	EEX	100	Lbl 04
17	4	45	Lbl 02	73	3	101	/
18	?<	46	EEX	74	Call04	102	Math
19	GT 01	47	4	75	+	103	Up
20	Swap	48	Call04	76	EEX	104	Input
21	EEX	49	GT 01	77	1	105	EEX
22	1	50	RCL Data	78	*	106	1
23	Call04	51	EEX	79	RCL Data	107	*
24	EEX	52	2	80	EEX	108	Math
25	1	53	/	81	1	109	Up
26	*	54	Math	82	Lbl 03	110	Up
27	RCL Data	55	Up	83	Call04	111	Input
28	EEX	56	Input	84	+	112	RTN
length/checksum = 142.056							

The CashFL instruction in line 14 is a NOP. Math Up Input is FP and Math Up Up Input is IP.

3.3.3 HP-16C

Hewlett-Packard calls their HP-16C the "Computer Scientist." While it only provides rudimentary floating-point capabilities, it really shines at working with bits. Its liquid-crystal display is capable of displaying integers in binary, octal, decimal, and hexadecimal. In addition to basic arithmetic it can perform boolean operations, shifts and rotates with and without carry, and other operations of use to the assembly language programmer. Of course, it does conversions between the aforementioned bases.

The 16C's powerful bit-twiddling capabilities allow us to produce surprisingly short programs to solve this puzzle. This program implements the Binary iterative algorithm give in Section 2.2.2. Register R_I holds the move counter, R_1 holds the current move number, and R_0 holds the previous move number.

Put the number of disks into X and start the program, which will proceed to display a series of two-digit numbers. The tens digits represents the "from" peg, the units digits represents the "to" peg. An output of zero signals program termination.

01	43,22, A	LBL A	15	42 20	AND	29	42 3	RMD
02	42 3	UNSGN	16	3	3	30	40	+
03	0	0	17	42 9	RMD	31	1	1
04	44 0	STO 0	18	1	1	32	1	1
05	42 44	WSIZE	19	0	0	33	40	+
06	24	DEC	20	20	×	34	43 34	PSE
07	42 8	MASKR	21	45 0	RCL 0	35	45 1	RCL 1
08	44 32	STO I	22	45 1	RCL 1	36	44 0	STO 0
09	43,22, 1	LBL 1	23	42 40	OR	37	43 23	DSZ
10	45 0	RCL 0	24	3	3	38	22 1	GTO 1
11	1	1	25	42 3	RMD	39	43 35	CLx
12	40	+	26	1	1	40	43 21	RTN
13	44 1	STO 1	27	40	+			
14	45 0	RCL 0	28	3	3			

An even shorter program is possible for the Gray Code algorithm given in Section 2.2.3 further illustrating the power and versatility of this calculator.

Steps 01 − 08 initialize the machine, setting unsigned mode so we don't see negative disk numbers, placing the number of disks in R_0, the first binary number (0) in R_1, and the move counter in R_I. Steps 10 − 17 find the one bit that changed from 0 to 1 and steps 18 − 21 determine its position, which is then displayed in step 22 with a 'd' for "disk." Step 23 decrements the loop counter and step 24 loops back if necessary. When the solution is completed, steps 25−27 place a 0 in the display with an 'o' for "over" and halts the program. Unlike the other programs in this collection, this one only shows the disk number, not the "from" and "to" pegs.

01	43,22, b	LBL B	10	45 1	RCL 1	19	45 0	RCL 0
02	42 3	UNSGN	11	45 1	RCL 1	20	34	x⇌y
03	44 0	STO 0	12	1	1	21	30	−
04	42 44	WSIZE	13	40	+	22	42 24	showDEC
05	43 35	CLx	14	44 1	STO 1	23	43 23	DSZ
06	44 1	STO 1	15	42 10	XOR	24	22 6	GTO 6
07	42 30	NOT	16	45 1	RCL 1	25	43 35	CLx
08	44 32	STO I	17	42 20	AND	26	25	OCT
09	43,22, 6	LBL 6	18	43 A	LJ	27	43 21	RTN

3.3.4 HP-15C

The HP-15C was a truly impressive calculator when it was introduced nearly thirty years ago, and it's still a perfectly competent machine. Its built-in complex arithmetic, matrix operations, numeric solver, and numeric integrator were all pioneering capabilities. And all in a truly pocketable device with almost unlimited battery life. (There are users who now are totally baffled by a blinking symbol on the screen, never before having seen the low-battery indicator.)

Unfortunately, the HP-15C lacks the vast memory spaces of more recent calculators. We could conceivably fit the entire data stack into the registers, but then that wouldn't leave any memory for the program, and we're definitely not going to set out to solve the problem by hand.

One solution is a form of data compression. We can get around the memory problem by putting two stack elements into each register. We can't fit two six-digit decimal numbers into a register (not on this ten-digit calculator, though this could be done on the more recent twelve-digit calculators). Instead of treating the stack element as a decimal number, we'll consider it as a base-four number. This allows us to represent the entire element in five decimal digits, and we can pack two of them into a single ten-digit number.

If you look closely at the program listing on the next page, you might see the DNA inherited from its HP 35s and HP 30b predecessors. The simulated subroutine starts at line 010. The next twenty steps determine whether to push the current stack element into the top half or the bottom half of the register indicated by the stack pointer in R_I. The inverse process starts at line 054.

The routine beginning at line 119 fetches the top element of the stack (without popping it). For future convenience, this value is placed into the RAN# register so that the routine beginning at line 135 can retrieve it. (What? You didn't know the 15C had a RAN# register?)

Like the HP 30b, the HP-15C lacks a remainder instruction. The routine beginning at line 141 rectifies that shortcoming.

The constant 256 appearing five times in the program is merely 10000_4, and 4, 16, and 64 are other integer powers of 4, while the 39 near the beginning of the program is 213_4, the same "number" used in two preceeding programs.

001	42,21,11	LBL A	050	32 5	GSB 5	099	20	×
002	42 34	CL REG	051	6	6	100	32 5	GSB 5
003	2	2	052	4	4	101	4	4
004	5	5	053	22 2	GTO 2	102	42,21, 2	LBL 2
005	6	6	054	42,21, 1	LBL 1	103	32 4	GSB 4
006	20	×	055	32 3	GSB 3	104	40	+
007	3	3	056	45 25	RCL I	105	32 5	GSB 5
008	9	9	057	48	.	106	2	2
009	40	+	058	5	5	107	5	5
010	42,21, 0	LBL 0	059	30	−	108	6	6
011	48	.	060	44 25	STO I	109	10	÷
012	5	5	061	43,30, 0	x≠0?	110	43 44	INT
013	45,40,25	RCL+ I	062	22 1	GTO 1	111	26	EEX
014	44 25	STO I	063	43 35	CLx	112	30	−
015	42 44	FRAC	064	43 32	RTN	113	2	2
016	43 20	x=0?	065	42,21, 1	LBL 1	114	5	5
017	22 0	GTO 0	066	33	R↓	115	6	6
018	33	R↓	067	2	2	116	20	×
019	45 24	RCL (i)	068	5	5	117	40	+
020	42 44	FRAC	069	6	6	118	22 0	GTO 0
021	22 6	GTO 6	070	32 4	GSB 4	119	42,21, 3	LBL 3
022	42,21, 0	LBL 0	071	43,30, 0	x≠0?	120	45 25	RCL I
023	33	R↓	072	22 1	GTO 1	121	42 44	FRAC
024	26	EEX	073	32 3	GSB 3	122	43 20	x=0?
025	5	5	074	1	1	123	22 3	GTO 3
026	10	÷	075	6	6	124	45 24	RCL (i)
027	45 24	RCL (i)	076	32 4	GSB 4	125	43 44	INT
028	43 44	INT	077	26	EEX	126	26	EEX
029	42,21, 6	LBL 6	078	1	1	127	5	5
030	40	+	079	20	×	128	10	÷
031	44 24	STO (i)	080	32 5	GSB 5	129	22 6	GTO 6
032	32 3	GSB 3	081	4	4	130	42,21, 3	LBL 3
033	2	2	082	32 4	GSB 4	131	45 24	RCL (i)
034	5	5	083	40	+	132	42 44	FRAC
035	6	6	084	42 31	PSE	133	42,21, 6	LBL 6
036	43,30, 7	x>y?	085	4	4	134	44 36	STO RAN#
037	22 1	GTO 1	086	32 5	GSB 5	135	42,21, 5	LBL 5
038	32 5	GSB 5	087	1	1	136	45 36	RCL RAN#
039	4	4	088	6	6	137	26	EEX
040	32 4	GSB 4	089	32 4	GSB 4	138	5	5
041	4	4	090	40	+	139	20	×
042	20	×	091	4	4	140	43 32	RTN
043	32 5	GSB 5	092	20	×	141	42,21, 4	LBL 4
044	1	1	093	32 5	GSB 5	142	10	÷
045	6	6	094	6	6	143	42 44	FRAC
046	32 4	GSB 4	095	4	4	144	4	4
047	40	+	096	32 4	GSB 4	145	20	×
048	4	4	097	40	+	146	43 44	INT
049	20	×	098	4	4	147	43 32	RTN

3.4 HP-BASIC (HP 38g/39g/40g)

Because HP-BASIC provide for neither recursion nor subroutines, we end up with a helper program to move one disk from peg `A` to peg `B`.

.HANOI.MOVE

```
M9(B,1)+1▶M9(B,1):
M9(A,M9(A,1))▶M9(B,M9(B,1)):
M9(A,1)-1▶M9(A,1)
DISP 1;A "->" B:
```

This subprogram is used by the main program, which prompts for the number of disks, sets up the initial configuration, then proceeds to solve the puzzle by displaying the “from” and “to” pegs for each move.

HANOI

```
INPUT N;"TOWERS OF HANOI";"DISKS";"ENTER NUMBER";5:
INT(MAX(2,MIN(64,N)))▶N:
REDIM M9;3,N+2
N+1▶M9(1,1):
1▶M9(2,1):
1▶M9(3,1):
FOR I=1 TO N;
 I▶M9(1, I+2):
END:
1+N MOD 2▶D:
1▶A:1+D▶B:
RUN ".HANOI.MOVE":B▶S:
DO
 1+(1==S)▶A:
 3-(3==S)▶B:
 IF M9(A,M9(A,1))<M9(B,M9(B,1))
 THEN A▶I:B▶A:I▶B:
 END:
 RUN ".HANOI.MOVE":
 S▶A:1+(D+S-1)MOD 3▶B:
 RUN ".HANOI.MOVE":B▶S:
UNTIL
 M9(3,1)==N+2:
END:
REDIM M9;1,1
```