Towers of Hanoi and Analysis of Algorithms

Paul Cull and Earl F. Ecklund, Jr.
Department of Computer Science
Oregon State University
Corvallis, Oregon 97331

84-20-1

"Towers of Hanoi and Analysis of Algorithms"

Paul Cull and E. F. Ecklund, Jr.
Department of Computer Science
Oregon State University
Corvallis, Oregon 97331

M.R. Categories: 05-04, 68C25, 68E99.

Paul Cull: I studied at Providence College and the University of Chicago, where I earned my Ph.D. in mathematical biology in 1970. Since then I have been on the faculty at Oregon State University, where I am now professor of computer science. When I am not in Oregon I like to be in Italy, where on various occasions I have been associated with the Laboratory of Cybernetics, near Naples, and the University of Salerno. My main research interests are analysis of algorithms and automata theory. I also continue my interest in mathematical biology, particularly the dynamics of population and neural nets. I believe that one's teaching and research should interact. The material in this paper grew out of my attempts to devise a good example of an analysis of a problem which could be understood by my classes and did not involve a problem, like sorting, which they had already seen in several other courses.

Earl F. Ecklund, Jr.: I received my Ph.D. in mathematics from Washington State University in 1972. Recently I joined the Computer Research Labs of Tektronix, Inc., moving from Oregon State University. My current research interests are database systems and operating systems in a distributed computing system. My avocations include number theory, particularly factoring algorithms, and squash.

1. INTRODUCTION

Mathematics has applications everywhere, but perhaps nowhere more clearly than in computer science, and within computer science perhaps nowhere more clearly than in analysis of algorithms. It would be difficult to draw a line and say that this part of analysis of algorithms is mathematics and that part is computer science.

In this paper we want to give a feeling for analysis of algorithms by investigating, in some detail, algorithms for the Towers of Hanoi problem. But first we should say a few general words about the field of analysis of algorithms. As a start one should make a bow to Donald Knuth. His "Art of Computer Programming" [5] is the standard work in the field. In the last few years, analysis of algorithms courses have become a part of many undergraduate programs in computer science. Knuth is still the standard reference, but more widely used textbooks are Aho, Hopcroft and Ullman (AHU) [1], Baase [2], and Horowitz and Sahni [4].

The task of analysis of algorithms is three-fold:

1) To produce provably correct algorithms, that is, algorithms which not only solve the problem they are designed to solve, but which also can be demonstrated to solve the problem;
2) To compare algorithms for a problem with respect to various measures of resources (e.g. time and space), so that we can say when one algorithm is better than another;
3) To find, if possible, the best algorithm for a problem with respect to a particular measure of resource usage. This involves proving "lower bounds", that is, showing that every algorithm which solves the problem must use at least so much of a particular resource. To establish a best algorithm one must have both a proof of a lower bound and an algorithm which uses no more than this lower bound.

Here a distinction should be made between bounds for an algorithm and bounds for a problem. If one establishes an upper bound on a particular resource used by an algorithm for a problem, then one has an upper bound both for the algorithm and for the problem. If one establishes a lower bound for a problem, then one also has a lower bound on all algorithms which solve this problem. But demonstrating a lower bound for one algorithm for a problem does _not_ establish a lower bound for the problem.

In this paper we will exemplify the three-fold task of analysis of algorithms using the Towers of Hanoi problem. This problem is often used as an example of a problem which can be neatly solved by a recursive algorithm, as an example of a problem which requires exponential time for its solution [5], and as an example of problem solving strategies [6]. In the Towers of Hanoi problem one is given three towers, usually called A, B and C, and n disks of different sizes. Initially the disks are stacked on tower A in order of size (disk n, the largest, on the bottom; disk 1, the smallest, on the top). The problem is to move the stack of disks to tower C, moving the disks one at a time in such a way that a disk is never stacked on top of a smaller disk. An extra constraint is that the sequence of moves should be as short as possible. An algorithm solves the Towers of Hanoi problem if, when the algorithm is given as input n the number of disks, and the names of the towers, then the algorithm produces the shortest sequence of moves which conforms to the above rules.

In this paper we will investigate a variety of algorithms which solve the Towers of Hanoi problem. We will prove the correctness of each algorithm, calculate the time and space used by each algorithm to allow a comparison among them, and prove the lower bounds on time and space required by any algorithm which solves the problem. We will show that the final algorithm which attains these bounds is the best possible for these measures.

2. COMPARING ALGORITHMS

For any (solvable) problem there will be an infinity of algorithms which solve the problem. How do we decide which is the "best" algorithm? There are a number of possible ways to compare algorithms. We will concentrate on two measures: time and space. We would like to say that one algorithm is faster, uses less time, than another algorithm if when we run the two algorithms on a computer the faster one will finish first. Unfortunately, to make this a fair test we would have to keep a number of conditions constant. For example, we would have to code the two algorithms in the same programming language, compile the two programs using the same compiler, and run the two programs under the same operating system on the same computer, and have no interference with either program while it is running. Even if we could practically satisfy all these conditions, we might be chagrined to find that algorithm A is faster under conditions C, but that algorithm B is faster under conditions D.

To avoid this unhappy situation we will only calculate time to order. We let n be some measure of the size of the problem, and give the running time as a function of n. For example, in the Towers of Hanoi we will use n for the number of disks. We do not distinguish running times of the same order. For our purposes two functions of n, $f(n)$ and $g(n)$, have the same order if for some N there are two positive constants C_1 and C_2 so that $C_1|g(n)| \leq |f(n)| \leq C_2|g(n)|$ for all $n \geq N$. We symbolize this relation by $f(n) = \theta(g(n))$, read $f(n)$ is order $g(n)$. Thus we will consider two algorithms to take the same time if their running times have the same order. In particular, we do not distinguish between algorithms whose running times are constant multiples of one another.

If we find that algorithm A has a time order which is strictly less than algorithm B, then we can be confident that for any large enough problem algorithm A will run faster than algorithm B, regardless of the actual conditions.

On the other hand if algorithms A and B have the same time order, then we will not predict which one will be faster under a given set of actual conditions.

The space used by an algorithm is the number of bits the algorithm uses to store and manipulate data. We expect the space to be an increasing function of n, the size of the problem. This space measurement ignores the number of bits used to specify the algorithm, which has a fixed constant size independent of the size of the problem. Since we have chosen bits as our unit, we can be more exact about space than we can be about time. We can distinguish an algorithm which uses 3n bits from an algorithm which uses 2n bits. But we will not distinguish an algorithm which uses $3n + 7$ bits from an algorithm which uses $3n + 1$ bits, because we can hide a constant number of bits within the algorithm itself.

So we will say that we have the "best" algorithm for a problem if we can show that the algorithm has minimal time order, and uses minimal space to within an additive constant.

It is not clear that such a best algorithm must exist. In some problems there is a time-space trade-off; a faster algorithm requires more space. We will demonstrate that this sort of trade-off does not exist in the Towers of Hanoi problem by eventually presenting an algorithm which achieves simultaneously minimal time and minimal space.

3. A RECURSIVE ALGORITHM

The road to a best algorithm starts with some algorithm which one then attempts to improve. One often uses some sort of strategy to create an algorithm. A very useful strategy is to look at the problem and see if the solution can be expressed in terms of the solutions of several problems of the same kind, but of smaller size. This strategy is usually called divide-and-conquer. If the problem yields to the divide-and-conquer approach, one can construct a recursive algorithm which solves the problem. This construction also gives almost immediately an inductive proof that the algorithm is correct. Time and space analyses of a divide-and-conquer algorithm are often straightforward, since the algorithm directly gives difference equations for time and space usage.

While these divide-and-conquer algorithms have many nice properties, they may not use minimal time and space. They may, however, serve as a starting point for constructing more efficient algorithms.

Consideration of the Towers of Hanoi problem leads to the key observation that moving the largest disk requires that all of the other disks are out of the way. Hence the n-1 smaller disks should be moved to tower B, but this is just another Towers of Hanoi problem with fewer disks. After the largest disk has been moved the n-1 smaller disks can be moved from B to C; again this is a smaller Towers of Hanoi problem. These observations lead to the following recursive algorithm [6], [7], [9]:

```
PROCEDURE HANOI(A,B,C,n)
    IF n=1 THEN move the top disk from tower A to tower C
            ELSE HANOI(A,C,B,n-1)
                 move the top disk from tower A to tower C
                 HANOI(B,A,C,n-1).
```

Is this the best algorithm for the problem? We will show that this algorithm has minimum time complexity, but does not have minimum space complexity. First though we prove that the algorithm correctly solves the problem, the first task of analysis of algorithms as outlined in the Introduction.

Proposition 1: The recursive algorithm HANOI correctly solves the Towers of Hanoi problem.

Proof: Clearly the algorithm gives the correct minimal sequence of moves for 1 disk. If there is more than 1 disk the algorithm moves n-1 disks to tower B, then moves the largest disk to tower C, and then moves the n-1 disks from tower B to tower C. This is precisely what is required in a minimum move algorithm because according to the rules the largest disk can only be moved when all the other n-1 disks are on a single tower. So the n-1 disks must be moved from tower A to some other tower. Clearly at least one move is required to move the largest disk from tower A to tower C. When the largest disk is moved to tower C, the other n-1 disks are on a single tower and still have to be moved to tower C. By inductively assuming n-1 disks are moved in the minimum number of moves, we see that the algorithm for n disks makes no more than the minimal number of moves and finishes with all the n disks moved from tower A to tower C. ▨

Here we should remark that we have not only produced a provably correct algorithm for the problem; we have also shown that the minimal sequence of moves is unique. This uniqueness makes the proof of correctness easy. The proof would be more complicated if more than one minimum sequence were possible.

We would like to calculate the running time of HANOI, but we don't know how long various operations will take. How long will it take to move a disk? How long will it take to subtract 1 from n? How long will it take to test if n = 1? How long will it take to issue a procedure call? Because we only wish to calculate time to order we don't have to answer these questions exactly, but we do have to make a distinction between operations which take a constant amount of time, independent of n, and operations whose running time depends on n.

One possibility is to assume that each operation takes constant time independent of n. AHU [1] calls this assumption the uniform cost criterion. With this uniform cost assumption and letting T(n) be the running time for n disks we have the difference equation

$$T(n) = 2T(n-1) + c$$

because there are 2 calls to the same procedure with n-1 rings and c is the sum of the constant running times for the various operations. Letting T(1) be the running time of the algorithm for 1 disk, we find

$$T(n) = (T(1) + c)2^{n-1} - c$$

which can be verified by direct substitution. This gives

$$T(n) = \theta(2^n)$$

since $\frac{T(1)}{2} 2^n \leq T(n) < (\frac{T(1) + c}{2})2^n$.

Another possibility is to assume that some of the operations have running times which are a function of n. But which function of n should we use? Each of the numbers in the algorithm is between 1 and n, and the disks can also be represented by numbers between 1 and n. Since such numbers can be represented using about log n bits, it seems reasonable to assume that each operation which manipulates numbers or disks has running time which is a constant times log n. AHU [1] calls this the logarithmic cost criterion and suggests using it when the numbers used by an algorithm do not have fixed bounds. Using the logarithmic cost criterion we have the difference equation

$$T(n) = 2T(n-1) + c \log n$$

for the running time of the algorithm. This difference equation has the solution

$$T(n) = 2^n \left[\frac{T(1)}{2} + c \sum_{i=1}^{n} \frac{\log i}{2^i} \right]$$

which can be verified by substitution. Since the summation in this solution converges, as one can demonstrate by the ratio test, and assuming that the constants are positive, we have

$$T(n) = \theta(2^n).$$

Since both cost criteria give the same running time, we conclude:

Proposition 2: The algorithm HANOI has running time $\theta(2^n)$.

Although we have established the running time for a particular algorithm which solves the Towers of Hanoi problem, we have not yet established the time complexity of the problem. We need to establish a lower bound so that every algorithm which solves the problem must have running time greater than or equal to the lower bound. We establish $\theta(2^n)$ as the lower bound in the proof of the following proposition.

Proposition 3: The Towers of Hanoi problem has time complexity $\theta(2^n)$.

Proof: Following the proof of Proposition 1, a straightforward induction shows that the minimal number of moves needed to solve the Towers of Hanoi problem is 2^n-1. Since each move requires at least constant time we have established the lower bound on time complexity.

An upper bound for the time complexity of the problem comes from Proposition 2. Since the upper bound and lower bound are equal to order, we have established the $\theta(2^n)$ time complexity of the problem. ▨

Now that we know HANOI's time complexity we would like to consider its space complexity. First we will establish a lower bound on space which follows from the lower bound on time.

Proposition 4: Any algorithm which solves the Towers of Hanoi problem must use at least n + constant bits of storage.

Proof: Since the algorithm must produce 2^n-1 moves to solve the problem, the algorithm must be able to distinguish 2^n different situations. If the algorithm did not distinguish this many situations then the algorithm would halt in the same number of moves after each of the two nondistinguished situations, which would result in an error in at least one of the cases.

The number of situations distinguished by an algorithm is equal to the number of storage situations times the number of internal situations within the algorithm. Since the algorithm has a fixed finite size it can have only a constant number of different internal situations. The number of storage situations (states) is 2 to the number of storage bits. Thus $C \cdot 2^{BITS} \geq 2^n$, and so BITS $\geq n - \log C = n +$ constant. ∎

In order to discuss the space complexity of the recursive algorithm, let us now consider the data structure used. Two possible data structures are the array and the stack. An array is a set of locations indexed by a set of consecutive integers so that the information stored at a location in the array can be referenced by indicating the integer which indexes the location. For example, the information at location I in the array ARRAY would be referenced by ARRAY[I]. A stack is a linearly ordered set of locations in which information can be inserted or deleted only at the beginning of the stack.

The towers could each be represented by an array with n locations, and each location would need at most log n bits. So an array data structure with $\theta(n \log n)$ bits would suffice. Alternately, each tower could be represented by a stack. Each stack location would need log n bits, so again this is an $\theta(n \log n)$ bit structure. Actually a savings would be made. Since only n disks have to be represented, the stack structure needs only n locations versus the 3n locations used by the array structure. Another possible structure is an array in which the i^{th} element holds the name of the tower on which the i^{th} disk is located. This structure uses only $\theta(n)$ bits. Yet another possibility is to not represent the towers, but to output the moves in the form FROM __ TO __. Thus we could use no storage for the towers.

The recursive algorithm still requires space for its recursive stack. When a recursive algorithm calls itself, the parameters for this new call will take the places of the previous parameters, so these previous parameters are placed on a stack from which they can be recalled when the new call is completed. Also placed on the stack is the return address, the position in the algorithm at which execution of the old call should be resumed. All of this information, the parameters and the return address, for a single call are referred to as a stack frame. At most n stack frames will be active at any time and each frame will use a constant number of bits for the names of the towers and log n bits for the number of disks. So the recursive algorithm will use $\theta(n \log n)$ bits whether or not the towers are actually represented. We summarize these considerations by the following proposition.

Proposition 5: The recursive algorithm HANOI correctly solves the Towers of Hanoi problem and uses $\theta(2^n)$ time and $\theta(n \log n)$ space.

The recursive algorithm uses more than minimal space. We are faced with several possibilities:

1) Minimal space is only a lower bound and is not attainable by any algorithm;
2) Minimal space can only be achieved by an algorithm which uses more than minimal time;
3) Some other algorithm attains both minimal time and minimal space.

By developing a series of iterative algorithms, we will arrive at an algorithm which uses both minimal time and minimal space.

4. SOME ITERATIVE ALGORITHMS

As a first step in obtaining a better algorithm, we will consider an iterative algorithm which simulates the recursive algorithm for $n \geq 2$. This algorithm RECURSIVE SIM is similar to an algorithm given by Tenenbaum and Augenstein [7], but we have chosen to explicitly keep track of the stack counter because this will aid us in finding an algorithm using even less space.

```
PROCEDURE RECURSIVE SIM (A,B,C,n)
   I:= 1
   L1[1]:= A; L2[1]:= C; L3[1]:= B
   NUM[1]:= n-1 ; PAR[1]:= 1 ; PAR[0]:= 1
   WHILE I ≥ 1 DO
     IF NUM[I] > 1
        THEN L1[I+1]:= L1[I]
             L2[I+1]:= L3[I]
             L3[I+1]:= L2[I]
             NUM[I+1]:= NUM[I] - 1
             PAR[I+1]:= 1
             I:= I+1
        ELSE MOVE FROM L1[I] TO L3[I]
             WHILE PAR[I] = 2 DO
                   I:= I-1
             IF I ≥ 1 THEN MOVE FROM L1[I] TO L2[I]
                           PAR[I]:= 2
                           TEMP:= L1[I]
                           L1[I]:= L3[I]
                           L3[I]:= L2[I]
                           L2[I]:= TEMP
```

The names of the towers are stored in the three arrays L1, L2, L3; the number of disks in a recursive call is stored in NUM; and the value of PAR indicates whether a call is the first or second of a pair of recursive calls.

RECURSIVE SIM sets up the parameters for the call HANOI (A,C,B,n-1). When the last move for this call is made, the arrays will contain the parameters for calls with 1 through n-2 disks, where each of these calls will have PAR=2. The arrays will still contain the parameters for the (A,C,B,n-1) call with PAR=1. The inner WHILE loop will pop each of the calls with PAR=2, leaving the array counter pointing at the (A,C,B,n-1) call. Since I will be 1 at this point the IF condition is satisfied and the MOVE FROM L1[I] TO L2[I] accomplishes the MOVE FROM A TO C of the recursive algorithm HANOI. The following assignment statements set up the call (B,A,C,n-1) with PAR=2. So when the moves for this call are completed all of the calls in the array will have PAR=2, and the inner WHILE loop will pop all of these calls setting I to 0. Then the IF condition will be false, so no operations are carried out, and the outer WHILE condition will be false so the algorithm will terminate.

Proposition 6: The RECURSIVE SIM algorithm correctly solves the Towers of Hanoi problem, and uses $\theta(2^n)$ time and $\theta(n \log n)$ space.

Proof: Correctness follows since this algorithm simulates the recursive algorithm which we have proved correct. The major space usage is in the arrays. Since each time I is incremented the corresponding NUM[I] is decremented and since NUM[I] never falls below 1, there are at most n-1 locations ever used in an array. The four arrays L1, L2, L3, and PAR use only a constant amount of space for each element, but NUM must store a number as large as n-1 so it uses $\theta(\log n)$ bits for an element. Thus the arrays use $\theta(n \log n)$ bits.

Now we have to argue about time usage. Most of the operations deal with constant-sized operands so these operations will take constant time. The exceptional operations are incrementing, decrementing, assigning, and comparing numbers which may have $\theta(\log n)$ bits. A difference equation for the time is

$$T(n) = 2T(n-1) + C \log n$$

where $T(n)$ is the time to solve a problem with n disks and $C \log n$ is the time for manipulating the numbers with $\theta(\log n)$ bits. As in the proof of Proposition 1 we have $T(n) = \theta(2^n)$. ▯

Notice that this algorithm does not improve on the recursive algorithm, but study of this form can lead to a saving of space. Storing the array NUM causes the use of $\theta(n \log n)$ space. If we did not have to store NUM, the algorithm would use only $\theta(n)$ space. Do we need to save NUM? NUM is used as a control variable so it seems necessary. But if we look at NUM[1] + 1 we get n. When NUM[I+1] is set, it is set equal to NUM[I] - 1, but then

$$\begin{aligned} \text{NUM[I+1]} + I + 1 &= \text{NUM[I]} - 1 + I + 1 \\ &= \text{NUM[I]} + I = n. \end{aligned}$$

Thus the information we need about NUM is stored in I and n. So if we replace the test on NUM[I] = 1 with a test on I = n-1, we can dispense with storing NUM and improve the space complexity from $\theta(n \log n)$ to $\theta(n)$. This replacement does not increase the time complexity of any step in the algorithm, so the time complexity remains $\theta(2^n)$.

Our new procedure is

```
PROCEDURE NEW SIM (A,B,C,n)
   I: = 1
   L1[1]:= A ; L2[1]:= C ; L3[1]:= B
   PAR[1]:= 1 ; PAR[0]:= 1
   WHILE I ≥ 1 DO
     IF I ≠ n-1
       THEN L1[I+1]:= L1[I]
            L2[I+1]:= L3[I]
            L3[I+1]:= L2[I]
            PAR[I+1]:= 1
            I:= I+1
       ELSE MOVE FROM L1[I] TO L3[I]
            WHILE PAR[I] = 2 DO
                  I:= I-1
            IF I ≥ 1 THEN MOVE FROM L1[I] TO L2[I]
                          PAR[I]:= 2
                          TEMP:= L1[I]
                          L1[I]:= L3[I]
                          L3[I]:= L2[I]
                          L2[I]:= TEMP
```

From the above observation we have:

Proposition 7: NEW SIM correctly solves the Towers of Hanoi problem and uses $\theta(2^n)$ time and $\theta(n)$ space.

Although we have reached $\theta(n)$ space we would like to decrease the space even further, hopefully to n + constant bits. If we look at the array PAR, we find that the algorithm scans PAR to find the first element not equal to 2, replaces that element by 2 and then replaces all the previous 2's by 1's. This is analogous to the familiar operation of adding 1 to a binary number, in which we find the first 0, replace it by a 1, and replace all the previous 1's by 0's. So it seems that we can replace the array PAR by a simple counter. The number of bits in the counter will, of course, depend on n.

So far this has not resulted in any saving of space. Will there be enough information in the counter to determine from which tower we should move a disk? The affirmative answer will enable us to achieve a minimal space algorithm. To motivate the design of our minimal space algorithm we will examine the sequence of 31 moves needed to solve the problem with 5 disks. This sequence is shown in Table 1.

TOWER 0	TOWER 1	TOWER 2	DECIMAL COUNT	COUNT	DISK	FROM	TO
12345	-	-	0	00000	1	0	2
2345	-	1	1	00001	2	0	1
345	2	1	2	00010	1	2	1
345	12	-	3	00011	3	0	2
45	12	3	4	00100	1	1	0
145	2	3	5	00101	2	1	2
145	-	23	6	00110	1	0	2
45	-	123	7	00111	4	0	1
5	4	123	8	01000	1	2	1
5	14	23	9	01001	2	2	0
25	14	3	10	01010	1	1	0
125	4	3	11	01011	3	2	1
125	34	-	12	01100	1	0	2
25	34	1	13	01101	2	0	1
5	234	1	14	01110	1	2	1
5	1234	-	15	01111	5	0	2
-	1234	5	16	10000	1	1	0
1	234	5	17	10001	2	1	2
1	34	25	18	10010	1	0	2
-	34	125	19	10011	3	1	0
3	4	125	20	10100	1	2	1
3	14	25	21	10101	2	2	0
23	14	5	22	10110	1	1	0
123	4	5	23	10111	4	1	2
123	-	45	24	11000	1	0	2
23	-	145	25	11001	2	0	1
3	2	145	26	11010	1	2	1
3	12	45	27	11011	3	0	2
-	12	345	28	11100	1	1	0
1	2	345	29	11101	2	1	2
1	-	2345	30	11110	1	0	2
-	-	12345	31	11111			

Table 1. Towers of Hanoi Solution for 5 disks.

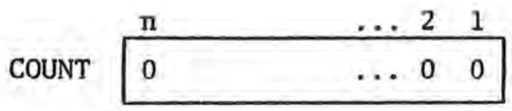
Every other move in the solution involves moving disk 1. So if we know which tower contains disk 1 we would know from which tower to move, in alternate moves, but we might not know which tower to move to. When we consider the three towers to be arranged in circle we see from table 1 that disk 1 always moves in a counterclockwise direction when we have an odd number of disks. Similarly disk 1 always moves in a clockwise direction when we have an even number of disks. Thus by keeping track of the tower which contains disk 1 and whether n is odd or even we would know how to make every other move.

For the moves which do not involve disk 1, we know that the move involves the two towers which do not contain disk 1. Looking again at table 1 we see that the odd numbered disks always move in the same direction as disk 1 and the even numbered disks always move in the opposite direction. So knowing the towers involved and whether the disk to be moved is odd or even would allow us to decide which way to move.

Can we determine from a counter whether the disk being moved is odd or even? If we look at the COUNT column of table 1 we see that the position of the rightmost 0 tells us the number of the disk to be moved. Thus a single counter with n bits is sufficient to solve the Towers of Hanoi problem.

We use these facts to construct the algorithm which follows.

```
PROCEDURE TOWERS (n)
   T:= 0  (*TOWER NUMBER COMPUTED MODULO 3*)
  COUNT:= 0  (*COUNT HAS n BITS*)
   P:= $\begin{cases} 1 \text{ if n is even} \\ -1 \text{ if n is odd} \end{cases}$
   WHILE TRUE DO
     MOVE DISK 1 FROM T TO T+P
     T:= T+P
     COUNT:= COUNT + 1
     IF COUNT = ALL 1's THEN RETURN
     IF RIGHTMOST 0 IN COUNT IS IN EVEN POSITION
       THEN MOVE DISK FROM T-P TO T+P
       ELSE MOVE DISK FROM T+P TO T-P
     COUNT:= COUNT + 1
   ENDWHILE
```



A picture of the storage used for COUNT.

Notice that it has n bits, and that we have called the rightmost bit position 1. The positions from right to left are then odd, even, odd, even....

Remarks: We can still improve this algorithm by removing the first COUNT := COUNT + 1 statement and deleting the rightmost bit of COUNT. This would also require changing the numbering of the bits in COUNT so that the rightmost bit is bit 0. An algorithm similar to our TOWERS has recently been published by T. R. Walsh [8].

We have to show that TOWERS correctly solves the Towers of Hanoi problem. We do this by proving that a certain sequence of moves has been accomplished when COUNT contains a number of the form 2^k-1, so that when $k = n$, the sequence of moves for HANOI (A,B,C,n) has been completed and the procedure will terminate since COUNT contains all 1's.

Proposition 8: When COUNT = 2^k-1, that is COUNT = $\boxed{00\ldots01\ldots1}$ with k 1's, then if $k \not\equiv n \pmod 2$ the correct moves for HANOI (A,C,B,k) have been completed and T contains 1 (which represents B),
if $k \equiv n \pmod 2$ the correct moves for HANOI (A,B,C,k) have been completed and T contains 2 (which represents C).

Proof: If $k = 1$ the single move T to T + P has been completed, which is A to C if n is odd, and is A to B if n is even, and T contains T + P which is 2 if n is odd and is 1 if n is even. This agrees with our claim.

Notice that COUNT can only take on the value 2^k-1 immediately before the IF ... RETURN statement. Assume the moves for either HANOI (A,B,C,k) or HANOI (A,C,B,k) have been completed. If $k \not\equiv n \pmod 2$ the next move will be A to C since by assumption T now contains 1; if k is odd the move is T - P to T + P which is 1 - (1) to 1 + 1 which represents A to C, and if k is even the move is T + P to T - P which is 1 + (-1) to 1 - (-1) which represents A to C. If $k \equiv n \pmod 2$ the next move will be A to B since by assumption T now contains 2; if k is odd the move is T - P to T + P which is 2 - (-1) to 2 + (-1) which represents A to B, and if k is even the move is T + P to T - P which is 2 + 1 to 2 - 1 which represents A to B.

Next COUNT will be incremented to $\boxed{0\ldots 010\ldots 0}$, i.e., k trailing 0's. When COUNT = $2^{k+1}-1$ the algorithm will have repeated the same sequence of moves as before since it only "sees" the rightmost information in COUNT, with the difference that T will have started with a different value. The different starting value of T will result in a cyclic permutation of the labels.
If $k \not\equiv n \pmod 2$ then the completed moves will be

HANOI (A,C,B,k)

A to C

HANOI (B,A,C,k)

giving HANOI (A,B,C,k+1) with $k+1 \equiv n \pmod 2$ and T will contain 2 (i.e., 1 + 1).
If $k \equiv n \pmod 2$ then the completed moves will be

HANOI (A,B,C,k)

A to B

HANOI (C,A,B,k)

giving HANOI (A,C,B,k+1) with $k+1 \not\equiv n \pmod 2$ and T will contain 1 (i.e., 2 + 2). ∎

Proposition 9: The algorithm TOWERS uses $\theta(2^n)$ time and n + constant bits of space.

Proof: For space usage, there are n bits in COUNT, and a constant number of bits are used for T and P.

For time, the initialization takes $\theta(n)$ and the WHILE loop is iterated 2^n-1 times. If each iteration took a constant amount of time we would have $\theta(2^n)$, but the test and increment instruction on count could take time $\theta(n)$ giving $\theta(n2^n)$. So we have to show that only $\theta(2^n)$ time is used.

If the value in COUNT is even then incrementing and testing will only require looking at one bit. If the value in COUNT is odd and (COUNT - 1)/2 is even, then the algorithm only looks at 2 bits. In fact, the algorithm will look at k bits in COUNT in 2^{n-k} cases. Thus the time used will be $\theta(\sum_{k=1}^{n} k\cdot 2^{n-k}) = \theta(2^n)$ since $\sum_{k=1}^{\infty} k\cdot 2^{-k}$ converges. ∎

We summarize these results in the following theorem.

Theorem: Any algorithm which solves the Towers of Hanoi problem for n disks must use at least $\theta(2^n)$ time and n + constant bits of storage. The algorithm TOWERS solves the problem and simultaneously uses minimum time and minimum space.

5. SUMMARY AND CONCLUSION

The goal of best algorithm has been attained. To attain the goal we started by analyzing the problem in a divide-and-conquer fashion and deriving from this analysis a recursive algorithm which we could prove solved the problem. Next we analyzed the time used by this recursive algorithm and argued that to order this time was best possible since any solution of the problem requires 2^n-1 moves.

From the lower bound on time we derived a lower bound of n bits on the space used by any algorithm which solves the problem. A space analysis of the recursive algorithm showed that it used more space than our lower bound, and that the space usage was required for the recursive stack.

To decrease the space usage we built an iterative algorithm which directly simulated the recursive algorithm. Since this was a direct simulation it used the same amount of space, but we now could investigate whether all the information being stored was necessary. We found that storing the number of disks on each simulated recursive call was unnecessary. This led to a new iterative algorithm which used only $\theta(n)$ space.

We then noticed that one of the arrays was functioning as a counter, but replacing it by a counter did not decrease the space usage. Next we investigated whether there was sufficient information in the counter to tell us which disk to move and where to move it. We found that we could tell which disk to move, but where to move the disk depended on whether the total number of disks was odd or even.

When we added a variable to keep track of the parity of the number of disks and another variable to keep track of the tower which contained the smallest disk, we found that we had sufficient information to solve the problem, and that we could dispense with the arrays which kept track of the tower names for each simulated recursive call.

At this point we had an algorithm which used n + constant number of bits, which was equal to our lower bound on space. We then showed that the algorithm still used minimal time order, and hence we had obtained a best algorithm.

We may remark that there could be other quite different looking algorithms which solve the problem and use minimal time and space. What we have tried to exemplify is a design methodology which is frequently used in deriving good algorithms. We have chosen the Towers of Hanoi problem because for this problem we could arrive at the goal of best algorithm. For other problems the process may get stuck. We might find a provable algorithm and lower bounds for the problem, but find that there is a gap between the running time of the algorithm and the lower bound, or find a gap between the space usage of the algorithm and the lower bound. Often the next crucial insight, like a disk always moves clockwise or counterclockwise, might not be discovered for many years after an algorithm is created. Alternately a discovered algorithm may be a best algorithm, but an insight is needed to raise the lower bounds for the problem.

In any case, we hope that we have given the reader some feel for analysis of algorithms.

6. EXERCISES

To see that you have understood a technique, it is useful to try to use the technique on similar problems. We give here two more algorithms for the Towers of Hanoi problem. Your task, if you decide to accept it, is to show that these algorithms do in fact solve the problem (i.e., prove that they are correct) and to determine the time and space usage of these algorithms.

Exercise 1:

```
PROCEDURE HANOI ITERATIVE (A,B,C,n)
   IF n mod 2 = 0 THEN MOVE[1]:= A TO B
                  ELSE MOVE[1]:= A TO C
   K:= 1
   WHILE n > 1 DO
      n:= n-1; K:= 2*K
      IF n mod 2 = 0 THEN MOVE[K]:= A TO B
                          L1:= C; L2:= A; L3:= B
                     ELSE MOVE[K]:= A TO C
                          L1:= B; L2:= C; L3:= A
      FOR I:= 1 TO K-1 DO
         CASE MOVE[I] OF
               A TO B : MOVE[K+I]:= L1 TO L2
               A TO C : MOVE[K+I]:= L1 TO L3
               B TO A : MOVE[K+I]:= L2 TO L1
               B TO C : MOVE[K+I]:= L2 TO L3
               C TO A : MOVE[K+I]:= L3 TO L1
               C TO B : MOVE[K+I]:= L3 TO L2
```

Hints 1: For correctness you may want to introduce a new variable and prove a statement which says that on each iteration of the WHILE loop the new variable increases (or if you want decreases) and that at the end of each iteration a Hanoi problem whose size depends on the new variable has been solved. You will need to give the tower names for the problem which has been solved. You will also need to specify the value of the new variable.

For space, you should know that the algorithm is storing each move in the array MOVE.

For time, you may want to consider both the uniform and the logarithmic cost measures.

Exercise 2: (Buneman and Levy [3])

```
MOVE SMALLEST DISK ONE TOWER CLOCKWISE
WHILE A DISK (OTHER THAN THE SMALLEST) CAN BE MOVED DO
      MOVE THAT DISK
      MOVE THE SMALLEST DISK ONE TOWER CLOCKWISE
ENDWHILE
```

Hints 2: For correctness, you should be careful since this algorithm only solves the original Towers of Hanoi problem when the number of disks is even. You will probably want to introduce a new variable and prove a statement about the configuration of the disks when the number of moves completed is a specific function of your new variable.

For time and space, the above algorithm is incomplete since it doesn't specify the data structure used to determine if a disk can be moved. You might consider representing each tower by a stack of integers with the integers representing the disks on the tower. Alternately you might consider representing the information by an array DISK, so that DISK[I] contains the name of the tower which contains the I^{th} largest disk. You may also find it useful to show that the i^{th} disk is moved 2^{n-i} times.

REFERENCES

[1] A. Aho, J. Hopcroft, and J. Ullman, The Design and Analysis of Computer Algorithms, Addison-Wesley, Reading, Massachusetts, 1974.

[2] S. Baase, Computer Algorithms, Addison-Wesley, Reading, Massachusetts, 1978.

[3] P. Buneman and L. Levy, The Towers of Hanoi Problem, Information Processing Letters 10, 1980, pp. 243-244.

[4] E. Horowitz and S. Sahni, Fundamentals of Computer Algorithms, Computer Science Press, Rockville, Maryland, 1978.

[5] D. Knuth, The Art of Computer Programming,
Vol. 1 Fundamental Algorithms (2nd edition, 1973)
Vol. 2 Seminumerical Algorithms (2nd edition, 1981)
Vol. 3 Search and Sorting (1973)
Addison-Wesley, Reading, Massachusetts.

[6] H. Simon, The Functional Equivalence of Problem Solving Skills, Cognitive Psychology, 1975, pp. 268-288.

[7] A. Tenenbaum and M. Augenstein, Data Structures Using PASCAL, Prentice-Hall, Englewood Cliffs, New Jersey, 1981, pp. 149-154.

[8] T. R. Walsh, The Towers of Hanoi Revisited: Moving the Rings by Counting the Moves, Information Processing Letters 15, 1982, pp. 64-67.

[9] D. Wood, The Towers of Brahma and Hanoi Revisited, Computer Science Technical Report No. 80-CS-23, McMaster University, 1980.