Tower of Hanoi: Transforming the Recursion

Nitin Verma
mathsanew.com

September 28, 2021

The well-known *Tower of Hanoi* problem can be described as follows. There are 3 vertical pegs numbered 1, 2 and 3, and N disks of radius $1, 2, 3, \ldots, N$ units each having a hole in the center. Initially, all the N disks are stacked onto peg 1 in decreasing order of their radius with the largest one (radius N) at the bottom. Our goal is to shift this "tower" of N disks to peg 3 using a sequence of moves where each "move" moves one disk from top of one peg to top of another peg. A move cannot place a larger disk over a smaller one. Peg 2 can be used as an auxiliary peg.

Following is a well-known recursive algorithm to solve this problem, implemented in C. The subproblem instance $recur(n, s, d, a)$ shifts the tower of top n disks from peg s (source) to peg d (destination) with peg a (auxiliary) serving as the auxiliary peg. The initial problem is $recur(N, 1, 3, 2)$.

```
#define MOVE(x, y) { printf("%d->%d\n", x, y); }

void recur(int n, int s, int d, int a)
{
  if(n == 1)
  {
    MOVE(s, d);
    return;
  }

  recur(n-1, s, a, d);

  MOVE(s, d);

  recur(n-1, a, d, s);
}
```

We will now try to write an equivalent of this recursive algorithm without using recursive calls. The recursive calls internally happen over a call-stack

in which each stack-frame represents one call and stores the local state and arguments of that call. In our non-recursive equivalent, we will maintain our own stack to simulate the recursive calls, similar to how they would happen on the internal call-stack. (This kind of approach to write non-recursive equivalent of a recursive method was discussed in an earlier article titled *Maintaining the Stack for Recursion* [1]).

Notice that each call of *recur*() requires storing its 4 arguments on the call-stack; there can be upto N such calls at any point. Now we will see how *recur*() can be transformed into a method without parameters, hence avoiding the need to store their values.

Recursion without Parameters

Consider the 4-tuple (n, s, d, a) formed by the arguments of *recur*(), which will be referred as "args-tuple". Notice that the args-tuple for its two recursive ("child") calls, $(n-1, s, a, d)$ and $(n-1, a, d, s)$, are simple transformations of the args-tuple (n, s, d, a) of the current ("parent") call. Specifically, the transformation would consist of decrementing n and swapping (d, a) or (s, a) (for the first and second recursive calls respectively). Thus, args-tuple for the child calls are easily obtainable from that of the parent's call.

Interestingly, these transformations are reversible and hence allow obtaining back the parent's args-tuple from the corresponding child's args-tuple.

This suggests that we can maintain a single global 4-tuple of arguments, g, which would correspond to the arguments of the current *recur*() call and will be modified by appropriate transformations (as mentioned above) during recursive calls and their return. Then, method *recur*() would access its arguments from this global 4-tuple g and so can be made parameter-less.

Following equivalent of *recur*() is based on the above idea:

```
typedef struct args_tuple
{
  int n;
  int s;
  int d;
  int a;
} args_tuple;

args_tuple g;
```

```
#define SWAP_g(x, y) { int t; t = g.x; g.x = g.y; g.y = t; }

#define MOVE_g(x, y) { printf("%d->%d\n", g.x, g.y); }

void recur2()
{
  /* We will denote the three members s, d and a in struct g as
     g(s, d, a). Say, at the start of this method,
     g(s, d, a) = (S, D, A) */

  if(g.n == 1)
  {
    MOVE_g(s, d);
    return;
  }

  /* decrement n */
  g.n = g.n - 1;

  /* update g(s, d, a) to (S, A, D) */
  SWAP_g(d, a);
  recur2();
  /* revert above update */
  SWAP_g(d, a);

  MOVE_g(s, d);

  /* update g(s, d, a) to (A, D, S) */
  SWAP_g(s, a);
  recur2();
  /* revert above update */
  SWAP_g(s, a);

  /* revert above decrement */
  g.n = g.n + 1;
}

/* initial call */
g.n = N;
g.s = 1;
g.d = 3;
g.a = 2;
recur2();
```

Non-Recursive Equivalent

We will use the parameter-less method *recur2*() to write a non-recursive equivalent of the recursive algorithm. As said earlier, we will maintain our

own stack to simulate the recursive calls.

Method *recur2*() has no arguments and no local state to be stored on the stack for each call. We only need to store the equivalent of *instruction-pointer* (as happens for the call-stack) to indicate where to resume execution after a recursive call returns. There are two recursive calls in this method and hence only two possible *resume-points* (the very next statement to execute after a call returns). We will identify these resume-points by integers 1 and 2. A special value 0 of resume-point will be used to indicate a fresh call (not an actual resume); this will make our implementation simpler.

So, our stack-frame for each call is a very small one, consisting of only the resume-point. As recursive calls are made and return, these frames are pushed-on and popped-off the stack.

The global 4-tuple g can now be made local in the method; we will unwrap its members such that they also serve as the parameters of this method.

Following non-recursive equivalent of *recur*() is based on the above ideas:

```
#define SWAP(x, y) { int t; t = x; x = y; y = t; }

void nonrecur(int n, int s, int d, int a)
{
  /* each frame of this stack actually requires only 2 bits, for
     storing one of the resume-points: 0, 1, 2 */
  char stack[n];
  int top;

  stack[0] = 0;
  top = 0;

  while(top >= 0)
  {
    /* execute a code-block of recur2() according to the current
       resume-point */
    switch(stack[top])
    {
      case 0:
        if(n == 1)
        {
          MOVE(s, d);
          /* return from this call */
          top--;
          continue;
        }
```

```
        n = n - 1;
        SWAP(d, a);
        /* make the first recursive call */
        stack[top] = 1;
        stack[++top] = 0;
        break;

      case 1:
        SWAP(d, a);
        MOVE(s, d);
        SWAP(s, a);
        /* make the second recursive call */
        stack[top] = 2;
        stack[++top] = 0;
        break;

      case 2:
        SWAP(s, a);
        n = n + 1;
        /* return from this call */
        top--;
        break;
    }
  }
}
```

Recursion as Tree Traversal

The recursive execution of *recur*() corresponds to a binary tree where each node represents a call (with its args-tuple) and its two child nodes represent the two recursive calls. The root node is the initial call. Consider the *inorder traversal* of this tree in which, upon a node's *visit* (processing), we perform the move step of its call. We can reason that, the move steps thus performed will be same as those performed by the complete execution of the initial call.

In *recur2*() and *nonrecur*() also, the same inorder traversal of the tree takes place. There, the nodes (args-tuples) are generated on-the-fly by using transformations between parent and child nodes. But once we *visit* a node (do its move step), we will often end-up performing multiple such transformations before we obtain and *visit* its *successor* node (do we see why?). (In inorder traversal, successor of a node is the node visited just after it).

B. Meyer makes some interesting observations about how a node and its successor are related, and how the successor can be obtained directly from a node by a simple transformation [2]. Thus, the same inorder traversal can be

performed but while directly obtaining and visiting the successor node after each node. This saves the multiple intermediate transformations (between parent-child nodes) we did above.

■

References

[1] Nitin Verma. *Maintaining the Stack for Recursion.* `https://mathsanew.com/articles/own_stack_for_recursion.pdf`.

[2] B. Meyer. *A Note on Iterative Hanoi.* ACM SIGPLAN Notices, Vol 19 (12) (1984), 38–40.