General Problems - 2

Analysis of solutions

António Rui Borges

Summary

Towers of Hanoi

History

Type of solutions
- *Recursive algorithm*
- *Iterative algorithm*

Using inheritance mechanism

Introducing polimorphism

History - 1

In a room of a certain budist temple, three poles were spiked into the ground. The poles could hold up to 64 gold disks of different diameters. At each pole, the disks overlapped one another following an increased diameter rule, top to bottom. Every day, one of the monks would make a movement, taking the top disk from one of the poles and placing it on one of the others. The only allowed movements were the ones that lead to the placement of a smaller diameter disk on top of a larger one.

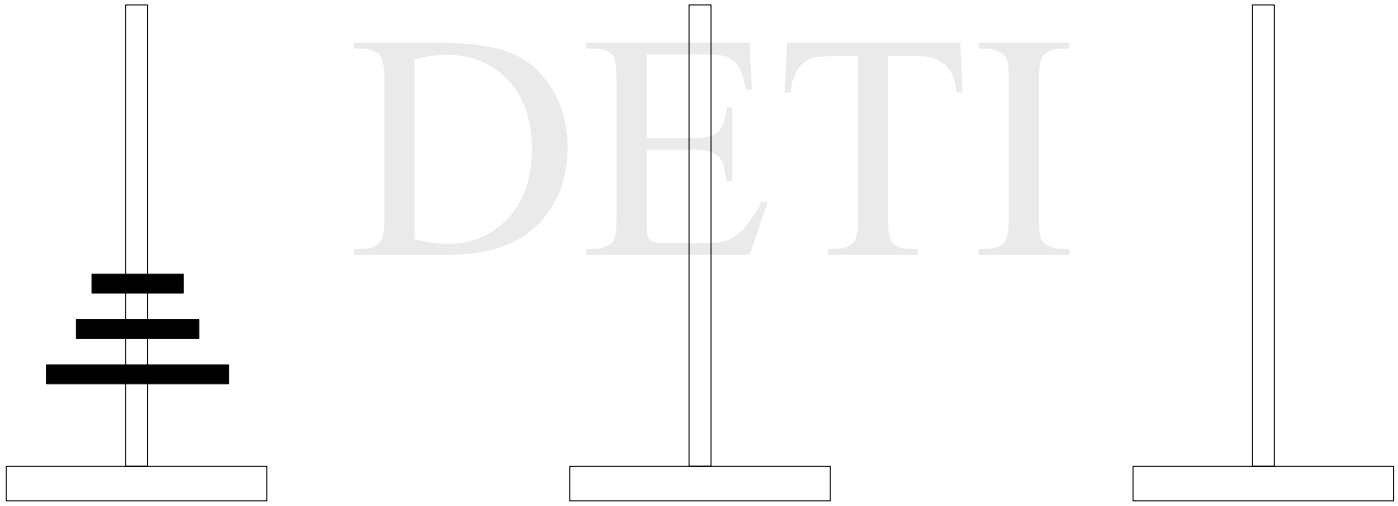
The aim was to move all the 64 disks, initially placed on the left pole, to the center one, using the right pole as auxiliary. According to tradition, once the task was completed, the world would come to an end ...

Believing the legend, should we worry about the event?

For a detailed description of the problem, see

Spitznagel, E. L., *Selected topics in mathematics*. Holt, Rinehart & Winston, p. 137, 1971.

History - 2



tower A ***tower B*** ***tower C***

Recursive algorithm - 1

```
void moveDisks (int n, Tower a, Tower b, Tower c)
{
   if (n == 1)
      moveOneDisk (a,b);
      else { moveDisks (n-1, a, c, b);
             moveOneDisk (a,b);
             moveDisks (n-1, c, b, a);
           }
}
```

Recursive algorithm - 2

Movements that take place assuming 3 disks at start

nIter	*tower A*	*tower B*	*tower C*
0	0-1-2	-	-
1	1-2	0	-
2	2	0	1
3	2	-	0-1
4	-	2	0-1
5	0	2	1
6	0	1-2	-
7	-	0-1-2	-

Recursive algorithm - 3

Movements that take place assuming 4 disks at start

nIter	*tower A*	*tower B*	*tower C*
0	0-1-2-3	-	-
1	1-2-3	-	0
2	2-3	1	0
3	2-3	0-1	-
4	3	0-1	2
5	0-3	1	2
6	0-3	-	1-2
7	3	-	0-1-2
8	-	3	0-1-2
9	-	0-3	1-2
10	1	0-3	2
11	0-1	3	2
12	0-1	2-3	-
13	1	2-3	0
14	-	1-2-3	0
15	-	0-1-2-3	-

Iterative algorithm - 1

```
void moveDisks (int n, Tower a, Tower b, Tower c)
{
  Tower ax = a,
        bx = ((n % 2) == 0) ? c : b,
        cx = ((n % 2) == 0) ? b : c;
  int nMov = (1 << n) - 1;

  for (int i = 1; i <= nMov; i++)
  { moveOneDisk (ax, bx);
    permutation (i, ax, bx, cx);
  }
}
```

Iterative algorithm - 2

Movements that take place assuming 3 disks at start

nIter	*tower A*	*tower B*	*tower C*	*tower S*	*tower D*	*tower X*	*permut*
0	0-1-2	-	-				
1	1-2	0	-	A	B	C	D<–>X
2	2	0	1	A	C	B	S<–>X
3	2	-	0-1	B	C	A	S–>D–>X–>S
4	-	2	0-1	A	B	C	S–>D–>X–>S
5	0	2	1	C	A	B	D<–>X
6	0	1-2	-	C	B	A	S<–>X
7	-	0-1-2	-	A	B	C	

Iterative algorithm - 3

Movements that take place assuming 4 disks at start

nIter	*tower A*	*tower B*	*tower C*	*tower S*	*tower D*	*tower X*	*permut*
0	0-1-2-3	-	-				
1	1-2-3	-	0	A	C	B	D<−>X
2	2-3	1	0	A	B	C	S<−>X
3	2-3	0-1	-	C	B	A	S−>D−>X−>S
4	3	0-1	2	A	C	B	S−>D−>X−>S
5	0-3	1	2	B	A	C	D<−>X
6	0-3	-	1-2	B	C	A	S<−>X
7	3	-	0-1-2	A	C	B	D<−>X
8	-	3	0-1-2	A	B	C	S<−>X
9	-	0-3	1-2	C	B	A	D<−>X
10	1	0-3	2	C	A	B	S<−>X
11	0-1	3	2	B	A	C	S−>D−>X−>S
12	0-1	2-3	-	C	B	A	S−>D−>X−>S
13	1	2-3	0	A	C	B	D<−>X
14	-	1-2-3	0	A	B	C	S<−>X
15	-	0-1-2-3	-	C	B	A	

Iterative algorithm - 4

Movements that take place assuming 5 disks at start

nIter	*tower A*	*tower B*	*tower C*	*tower S*	*tower D*	*tower X*	*permut*
15	4	-	0-1-2-3	B	C	A	S−>D−>X−>S
16	-	4	0-1-2-3	A	B	C	S−>D−>X−>S
17	0	4	1-2-3	C	A	B	

Movements that take place assuming 6 disks at start

nIter	*tower A*	*tower B*	*tower C*	*tower S*	*tower D*	*tower X*	*permut*
31	5	-	0-1-2-3-4	A	C	B	D<−>X
32	-	5	0-1-2-3-4	A	B	C	S<−>X
33	-	0-5	1-2-3-4	C	B	A	

Iterative algorithm - 5

Type of permutation in function of iteration number

nIter mod 4 = 1	D<–>X
nIter mod 4 = 2	S<–>X
(nIter mod 4 = 3) and [(nIter+1) mod $2^k \neq 2^{k-1}$], with odd k	D<–>X
(nIter mod 4 = 0) and (nIter mod $2^k \neq 2^{k-1}$)	S<–>X
(nIter mod 4 = 3) and [(nIter+1) mod $2^k = 2^{k-1}$], with odd k	S–>D–>X–>S
(nIter mod 4 = 0) and (nIter mod $2^k = 2^{k-1}$)	S–>D–>X–>S

Applying the inheritance mechanism - 1

Problem simulation entails modeling the poles with the inserted disks through *stack* memories. One will be dealing with modified *stacks* since state description after each movement will require a new operation to enable the visualization of its contents without altering the stored values.

This means according to the object oriented paradigm that

- one applies the inheritance mechanism to specialize the *stack* into a new data type that defines the operation.

Applying the inheritance mechanism - 2

```
public class Tower<R> extends MemStack<R>
{
   public Tower (R [] storage) throws MemException
   {
     super (storage);
   }

   public R peek (int pos) throws MemException
   {
     if ((pos < 0) || (pos >= mem.length))
        throw new MemException ("Illegal position!");
        else if (pos < stackPnt)
                return (mem[pos]);
                else return (null);
   }
}
```

Introducing polymorphism - 1

Aiming for two different solutions to the problem, it would be operationally convenient that, after a given solution is chosen, one gets a complete uniformity on its execution.

According to the object oriented paradigm, this means

- to define a common *interface* to the data types that implement the solutions
- use the resulting polymorphism on calling the solving methods on variables of the `interface` type.

Introducing polymorphism - 2


```
public interface TowersOfHanoi
{
   public void problemSolving ();
}
```

Introducing polymorphism - 3

```
public class RecursiveSolution implements TowersOfHanoi
{
   private int nDisks;
   private Tower<Integer> a;
   private Tower<Integer> b;
   private Tower<Integer> c;
   private int nIter;
   public RecursiveSolution (int nDisks)
   {
     try
     { if (nDisks <= 0) throw new MemException ("Illegal number of disks!");
       this.nDisks = nDisks;
       a = new Tower<> (new Integer [nDisks]);
       b = new Tower<> (new Integer [nDisks]);
       c = new Tower<> (new Integer [nDisks]);
       for (int i = nDisks-1; i >= 0; i--)
       { try
         { a.write (i);
         }
         catch (MemException e)
         { GenericIO.writelnString ("Error: ", e.getMessage (),
                                    " in iteration " + (i+1));
           e.printStackTrace ();
           System.exit (1);
         }
       }
     }
     catch (MemException e)
     { GenericIO.writelnString ("Error: ", e.getMessage ());
       e.printStackTrace ();
       System.exit (1);
     }
   }
```

Introducing polymorphism - 4

```
@Override
public void problemSolving ()
{
  GenericIO.writelnString ("           Listing of movements");
  GenericIO.writelnString ();
  showState ();
  moveDisks (nDisks, a, b, c);
}
private void moveDisks (int n, Tower<Integer> a, Tower<Integer> b, Tower<Integer> c)
{
   if (n == 1)
      moveOneDisk (a,b);
      else { moveDisks (n-1, a, c, b);
             moveOneDisk (a,b);
             moveDisks (n-1, c, b, a);
           }
}
private void moveOneDisk (Tower<Integer> a, Tower<Integer> b)
{
  try
  { b.write (a.read ());
  }
  catch (MemException e)
  { GenericIO.writelnString ("Error: ", e.getMessage ());
    e.printStackTrace ();
    System.exit (1);
  }
  showState ();
}
```

Introducing polymorphism - 5

```
  private void showState ()
  {
    Integer [] val = new Integer[3];
    boolean isValue;
    Tower<Integer> w;

    GenericIO.writelnString ("Iteration " + nIter);
    GenericIO.writelnString ();
    for (int i = nDisks-1; i >= 0; i--)
    { isValue = false;
      for (int j = 0; j < 3; j++)
      { try
        { w = (j == 0) ? a : (j == 1) ? b : c;
          val[j] = w.peek (i);
          isValue = isValue || (val[j] != null);
        }
        catch (MemException e)
        { GenericIO.writelnString ("Error: ", e.getMessage (),
                                   " in iteration ", "(" + i + "," + j + ")" );
          e.printStackTrace ();
          System.exit (1);
        }
      }
      if (isValue)
         { for (int j = 0; j < 3; j++)
             if (val[j] != null)
                GenericIO.writeFormInt (6, val[j]);
                else GenericIO.writeFormChar (6, ' ');
           GenericIO.writelnString ();
         }
    }
    GenericIO.writelnString ("\n     A     B     C\n");
    nIter += 1;
  }
}
```

Introducing polymorphism - 6

```
public class IterativeSolution implements TowersOfHanoi
{
   private int nDisks;
   private Tower<Integer> a;
   private Tower<Integer> b;
   private Tower<Integer> c;
   private int nIter;
   public RecursiveSolution (int nDisks)
   {
     try
     { if (nDisks <= 0) throw new MemException ("Illegal number of disks!");
       this.nDisks = nDisks;
       a = new Tower<> (new Integer [nDisks]);
       b = new Tower<> (new Integer [nDisks]);
       c = new Tower<> (new Integer [nDisks]);
       for (int i = nDisks-1; i >= 0; i--)
       { try
         { a.write (i);
         }
         catch (MemException e)
         { GenericIO.writelnString ("Error: ", e.getMessage (),
                                    " in iteration " + (i+1));
           e.printStackTrace ();
           System.exit (1);
         }
       }
     }
     catch (MemException e)
     { GenericIO.writelnString ("Error: ", e.getMessage ());
       e.printStackTrace ();
       System.exit (1);
     }
   }
```

Introducing polymorphism - 7

```
@Override
public void problemSolving ()
{
  GenericIO.writelnString ("           Listing of movements");
  GenericIO.writelnString ();
  showState ();
  moveDisks (nDisks, a, b, c);
}
private void moveOneDisk (Tower<Integer> a, Tower<Integer> b)
{
  try
  { b.write (a.read ());
  }
  catch (MemException e)
  { GenericIO.writelnString ("Error: ", e.getMessage ());
    e.printStackTrace ();
    System.exit (1);
  }
private void showState ()
{
      ...
}
```

Introducing polymorphism - 8

```
private void moveDisks (int n, Tower<Integer> a, Tower<Integer> b, Tower<Integer> c)
{
  Tower<Integer> sx = a,                          // source pole during the iteration process
                 dx = ((n % 2) == 0) ? c : b,     // destination pole during the iteration process
                 xx = ((n % 2) == 0) ? b : c;     // auxiliary pole during the iteration process
  int nMov;                                       // total number of movements
  int dim = (n + (n % 2)) / 2 - 1;                // number of points of rotation
  int [] turn = new int [dim];                    // reference values for rotation
  boolean rot;                                    // signaling a rotation should take place

  nMov = (1 << n) - 1;
  for (int j = 0; j < dim; j++)
    if (j == 0)
       turn[0] = 8;
       else turn[j] = 4 * turn[j-1];
```

Introducing polymorphism - 9

```
      Tower<Integer> tx;                                          // temporary pole

      for (int i = 1; i <= nMov; i++)
      { /* base movement */
        moveOneDisk (sx, dx);
        /* pole permutation */
        rot = false;
        for (int j = dim-1; j >= 0; j--)
        { rot = (((i + 1) % turn[j]) == (turn[j] / 2) || (i % turn[j]) == (turn[j] / 2));
          if (rot) break;
        }
        if (!rot && ((i % 2) == 0) || ((i % 4) == 2))             // sx <-> xx
           { tx = sx;
             sx = xx;
             xx = tx;
           }
           else if (!rot && ((i % 2) == 1) || ((i % 4) == 1))     // dx <-> xx
                { tx = dx;
                  dx = xx;
                  xx = tx;
                }
                else { tx = sx;                                   // sx -> dx -> xx -> sx
                       sx = xx;
                       xx = dx;
                       dx = tx;
                     }
      }
    }
}
```

Introducing polymorphism - 10

```
public class ProbTowersOfHanoi
{
   private static enum MenuOpt {RECUR, REP, END};

   public static void main (String [] args)
   {
     GenericIO.writelnString ();
     GenericIO.writelnString ("            Problem of Towers of Hanoi");
     GenericIO.writelnString ();

     String [] menu = {"1 - Recursive solution",          // menu
                       "2 - Iterative solution",
                       "0 - Program exit"
                      };
     int opt;                                            // selected option
     MenuOpt [] choice = {MenuOpt.END,                   // listing of the options
                          MenuOpt.RECUR,
                          MenuOpt.REP
                         };
     boolean end = false;                                // signaling end of operations

     do
     {
       do
       { GenericIO.writelnString ();
         for (int i = 0; i < menu.length; i++)
         { GenericIO.writeFormChar (10, ' ');
           GenericIO.writelnString (menu[i]);
         }
         GenericIO.writelnString ();
         GenericIO.writeString ("What is your choice? ");
         opt = GenericIO.readlnInt ();
       } while ((opt < 0) || (opt >= menu.length));
```

enumeration of the alternatives

presentation of the alternatives

Introducing polymorphism - 11

```
      switch (choice [opt])
      { case RECUR:
        case REP:   int val;                              // number of disks in the pile
                    TowersOfHanoi tH;                     // solution configuration

                    GenericIO.writelnString ();
                    do
                    { GenericIO.writeString ("Number of disks in the pile? ");
                      val = GenericIO.readlnInt ();
                    } while ((val <= 0) || (val > 20));   // keeping number of disks within
                                                          // reasonable limits
                    switch (choice [opt])
                    { case RECUR: tH = new RecursiveSolution (val);
                                  break;
                      case REP:   tH = new IterativeSolution (val);
                                  break;
                      default:    tH = null;
                    }
                    GenericIO.writelnString ();
                    tH.problemSolving ();
                    break;
        case END:   end = true;
      }
    } while (!end);
  }
}
```

selective instantiation

polymorphic calling