Acta Univ. Sapientiae, Informatica, **1**, 1 (2009) 89–108

Towers of Hanoi – where programming techniques blend

Zoltán Kátai
Sapientia University
email: katai_zoltan@ms.sapientia.ro

Lehel István Kovács
Sapientia University
email: klehel@ms.sapientia.ro

Abstract. According to the literature, the Towers of Hanoi puzzle is a classical *divide et conquer* problem. This paper presents different ways to solve this puzzle. We know that the time-complexity of the puzzle is $2^n - 1$. However, the question is how much time is needed to run each implementation.

1 The legend

Once upon in time, in India, in the reign of Fo Hi, , monks in the Siva-temple of Banares (which marks the center of the world) have to move a pile of 64 sacred golden disks from one diamond peg to another. The disks are fragile; only one can be carried at a time. A disk may not be placed on top of a smaller, less valuable disk. And, there is only one other diamond peg in the temple (besides the source and destination pegs) sacred enough that a pile of disks can be placed there. So, the monks start moving disks back and forth, between the original pile, the pile at the new location, and the intermediate location, always keeping the piles in order (largest on the bottom, smallest on the top) [9]. The legend claims that once the monks are finished, the world will end. So we need to figure out how long it is going to take the monks to finish the puzzle. How many moves will it take to transfer n disks from the source peg to the destination peg?

AMS 2000 subject classifications: 68W40
CR Categories and Descriptors: D.1 [Programming Techniques]
Key words and phrases: Towers of Hanoi, programming techniques, backtracking, greedy, divide and conquer, dynamic programing, recursive, iterative

The puzzle was introduced in 1883 by N. Claus (de Siam) Professor at University of Li-Sou-Stian, an anagram pseudonym for douard Lucas (D'Ameins) Professor at Lyce Saint-Louis [1].

According to the literature – as we can see in the proof –, the Towers of Hanoi puzzle is a classical *divide et conquer* problem. The puzzle can be solved by reducing the problem to smaller, but similar sub-problems; the key-move is the transfer of the largest disc.

In this train of thought we must to transfer $n - 1$ discs from the first peg to the third, than we can move the largest disc to the second peg, finally we must to transfer $n - 1$ discs from the third peg to the second. According to this recursive move-sequence, it is easy to see, that we need $2^n - 1$ steps to solve the n-discs puzzle.

So, the monks needs to make $2^{64} - 1$ (= 18 446 744 073 709 551 615) moves to solve the puzzle. Assuming, that they are capable to move one disc per second, the end of the world comes approximately in 590 000 000 000 years (according to estimations the Universe is 13,7 billion years old!).

2 The *min/max* problem

The Tower of Hanoi problem has four parameters: $H(n, s, d, h)$

- n: the number of discs
- s: source peg
- d: destination peg
- d: h: "helping peg"

Move n discs from peg s to peg d using the peg h. Initially the source peg is a, the destination peg is b, and the helping peg is c.

The problem, as optimization task, has two versions ($H_{min}(n, a, b, c)$ / $H_{max}(n, a, b, c)$): find the shortest/longest moves-sequence. (Target function: minimize/maximize the number of the moves). In the case of the "maximum version", obviously, we are interested only in cycle free solutions.

We introduce the following notations:

'$\rightarrow$': this symbol shows a direct move (one-step move) of a given disc between to pegs ($a \rightarrow b$: move a given disc from peg a to peg b directly).

'>>': this symbol represents a move-sequence that transposes a disc-tower between to pegs ($a >> b$: transpose a given disc-tower from peg a to peg b).

The key idea behind both solutions is to focus on the largest disc. In case of the "minimum version" the largest disc is moved from peg a to peg b in one (minimum) step ($a \rightarrow b$). This action supposes that, previously, the other

$(n-1)$ discs have already been transposed from peg a to peg c $(a >> c)$. After the largest disc has made the move $(a \rightarrow b)$, the other $(n-1)$ discs have to be transposed again, but at this time from peg c to peg b $(c >> b)$. Naturally, in order to achieve the optimal solution, the two moves of the $(n-1)$ discs have to also be performed in minimum number of steps. Consequently, we get the following recursive formula (f_{min}) for the minimum version of the problem (the optimisation is present in the formula only implicitly):

• if $n = 1$, then $H_{min}(n, a, b, c) = a \rightarrow b$

• if $n > 1$, then $H_{min}(n, a, b, c) = H_{min}(n-1, a, c, b), a \rightarrow b, H_{min}(n-1, c, b, a)$

Denoting with m_n the number of moves needed to solve the minimum version of the n-size problem, we have:

$m_n = 2m_{n-1} + 1, m_1 = 1$

$m_n = 2(2m_{n-2} + 1) + 1 = 2^2 m_{n-2} + 2 + 1$

$m_n = 2(2(2m_{n-3} + 1) + 1) + 1 = 2^3 m_{n-3} + 2^2 + 2 + 1$

$\ldots$

$m_n = 2^{n-1} m_1 + 2^{n-2} + ... + 2^2 + 2 + 1 = 2^{n-1} + 2^{n-2} + ... + 2^2 + 2 + 1$

$m_n = 2^n - 1$

In case of the maximum version of the problem, the largest disc is moved from peg a to peg b in two (maximum) steps $(a \rightarrow c,\ c \rightarrow b)$. As we exposed above, these moves entail that before, between and after them the other $(n-1)$ discs are transposed in optimal way from peg a to peg b $(a >> b)$, from peg b to peg a $(b >> a)$, and again, from peg a to peg b $(a >> b)$. The recursive formula (f_{max}) that describes the optimal solution for the maximum version of the problem is the following:

• if $n = 1$, then $H_{max}(n, a, b, c) = a \rightarrow c, c \rightarrow b$

• if $n > 1$, then $H_{max}(n, a, b, c) = H_{max}(n-1, a, b, c), a \rightarrow c, H_{max}(n-1, b, a, c), c \rightarrow b, H_{max}(n-1, a, b, c)$

Denoting with M_n the number of moves needed to solve the maximum version of the n-size problem, we have:

$M_n = 3M_{n-1} + 2, M_1 = 2$

$M_n = 3(3M_{n-2} + 2) + 2 = 3^2 M_{n-2} + 2 \cdot 3 + 2$

$M_n = 3(3(3M_{n-3} + 2) + 2) + 2 = 3^3 M_{n-3} + 2 \cdot 3^2 + 2 \cdot 3 + 2$

$\ldots$

$M_n = 3^{n-1} M_1 + 2 \cdot 3^{n-2} + ... + 2 \cdot 3^2 + 2 \cdot 3 + 2 = 2(3^{n-1} + 3^{n-2} + ... + 3^2 + 3 + 1)$

$M_n = 2((3^n - 1)/(3 - 1))$

$M_n = 3^n - 1$

Notice that there are only these two possibilities: the largest disc is moved from the source peg to the destination peg in one or two steps. If we always

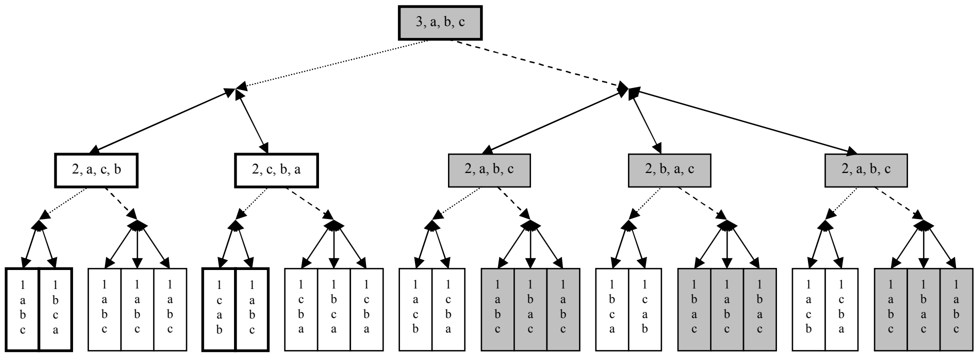


Figure 1: The n=3 case.

choose the one-step solution, we get the minimum-solution. Choosing always the second possibility we have the maximum-solution. If we combine the two possibilities we receive the other solutions. All these can be represented by a rooted tree (see Figure 1). The root-node represents the n-size problem that can be reduced in two ways ("minimum-branch"/"maximum-branch") to two or three $(n-1)$-size sub-problems, together to five sub-problems. This means, that we have a complete n-level tree with 5^{k-1} node at each level k $(k = 1, 2, \dots, n)$. This tree has $5^n - 1$ nodes. On the one hand, the complete n-level sub-tree that contains only "minimum-branches" (dotted-arcs) correspond to the optimal solution of the minimum version of the problem. On the other hand, the complete n-level sub-tree with only "maximum-branches" (dashed-arcs) represents the maximum-solution of the problem. Notice that the "minimum sub-tree" (bold-line rectangles) has $2^n - 1$ nodes, and the maximum one (filled rectangles) $3^n - 1$ nodes. Furthermore, since each complete n-level sub-tree represents a solution, the problem has as many solutions as such sub-trees exist. For $n = 3$ there are one minimum, one maximum and 1870 other solutions.

The recursive formulas can be seen as the ones that describe the structure of the optimal solutions. These structures can be represented with rooted trees, which correspond to the "minimum sub-tree" and "maximum sub-tree" of the tree shown in Figure 1. Figure 2 shows the tree attached to the minimum version of the 3-size problem $(H_{min}(3, a, b, c))$. The sub-trees represent the optimal sub-solutions. The leaves correspond to the moves. The leaf-sequence represented by dotted arcs corresponds to the optimal move-sequence. The root-node (white-filled rectangles) of each sub-tree has its "own leaf" (dashed-

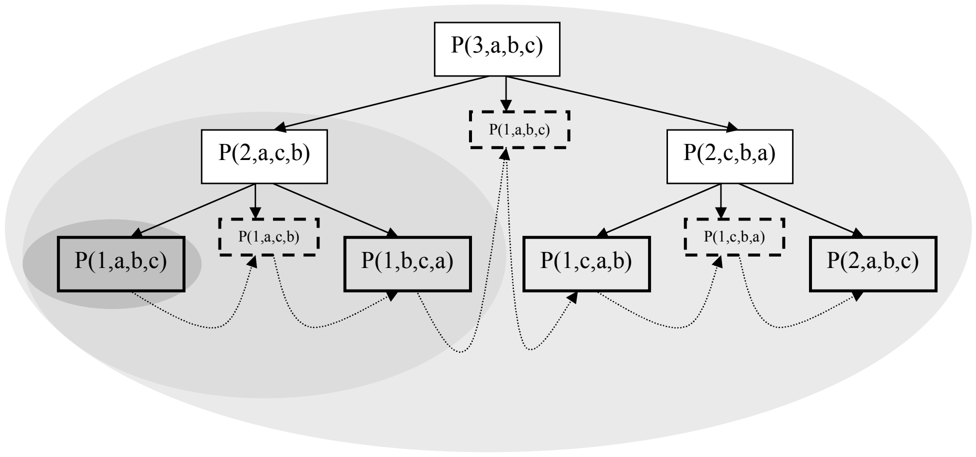


Figure 2: The $H_{min}(3, a, b, c)$ problem.

line rectangles) that represent the move of the largest disc of the corresponding "tower". We consider that these leaves are at the same level with their parent nodes (the "root-nodes" "assimilate" their "own leaves"). If we denote the discs (from bottom to top) and the tree-levels (from root to leaves) with $1, 2, 3, \ldots, n$, respectively, then it can be observed that leaves from a given level k correspond to the moves of disc k. Disc k makes 2^{k-1} moves. Since every second leaf (bolded-line rectangles) of the leaf-sequence is placed on the n^{th} level, these moves are performed by the smallest disc.

3 The recursive implementation

The below recursive procedures (P_recursive_min, P_recursive_max) are direct transcriptions of the recursive formulas.

```
void P_recursive_min(int k, char s, char d, char h)
{
  if(k==1)
    printf("%c -> %c\n", s, d);
  else
    {
      P_recursive_min(k-1, s, h, d);
      printf("%c -> %c\n", s, d);
      P_recursive_min(k-1, h, d, s);
    }
}


void P_recursive_max(int k, char s, char d, char h)
{
  if(k==1)
    printf("%c -> %c\n%c -> %c\n", s, h, h, d);
  else
    {
      P_recursive_max(k-1, s, d, h);
      printf("%c -> %c\n", s, h);
      P_recursive_max(k-1, d, s, h);
      printf("%c -> %c\n", h, d);
      P_recursive_max(k-1, s, d, h);
    }
}
```

What programming technique(s) applies these implementations?

4 Divide and conquer or greedy?

At the first sight the above implementations are pure divide and conquer algorithms: the problem to be solved is divided in two/three simpler, similar sub-problems. However, why is the problem divided in sub-problems even in these specific ways? These are direct consequences of the decision we are made with respect to the largest disc, decisions that are greedy choices. (The largest disc is moved from the source peg to the destination peg in minimum/maximum steps) Consequently, from this point of view, the recursive implementations are such divide and conquer strategies that apply greedy decisions in the dividing stage of the method.

On the other hand it can be stated that the "principle solving" of each sub-problem starts with a greedy decision (relative to the largest disc) that reduces the current sub-problem to two/three simpler, similar sub-sub-problems. Particularly in the case of this problem (Tower of Hanoi) the two (minimum version) sub-sub-problems have to be solved before, and after (respectively) the greedy decision is implemented. As a consequence, the order the greedy decisions have to be implemented follows a kind of in-order traverse of the tree representing the optimal solution (see figure 2). Accordingly, from this point of view, the recursive implementations are greedy strategy that use divide and conquer like methods (in-order DFS algorithm) in order to establish the implementation order of the greedy decisions.

The standard greedy algorithm first implements the greedy decision and after this it solves the "reduced sub-problem(s)". The pre-order DFS and the BFS traverses of the "solution-tree" allow of such a greedy strategy. In order to establish the proper move-order, these implementations need to store the moves. Unfortunately, there is a major concern relating this idea. The size of the solution-code depends exponentially on the problem-size (n). Coding the six possible moves $a \to b$, $a \to c$, $b \to c$, $c \to b$, $c \to a$, $b \to a$ with digits 0, 1, 2, 3, 4, 5 the code of the optimal solution of a 64-size problem has $2^{64} - 1$ digits. No computer capable to store that huge amount of data. Assuming that n takes moderate-values the optimal move-sequence can be stored in array moves[0..(2^n-1)-1]. Bi-dimensional array code stores the move-codes. Procedure P_greedy_min determines the elements of array moves according to the BFS traverse of the tree. Array q[0..(2^n-1)-1] implements the queue necessary for the BFS. Variables first and last indicate the beginning and the end of the queue. The items of the queue q store the source (s), destination (d) and helping (h) pegs of the corresponding sub-problem, and the left (l) and right (r) margins of the current segment in array moves. The array-segment [0..(2^n-1)-1] (the whole array moves) is attached to the original problem. At each step procedure P_greedy_min stores the greedy-move (s→d) corresponding to the sub-problem from the front of the queue in the middle element (m = (q[first].l+q[first].r)>>1) of the corresponding array-segment ([q[first].l .. q[first].r]). The array-sub-segments [q[first].l .. m-1] and [m+1 .. q[first].r] are attached to the left and right son-sub-problems.

```
void P_greedy_min(int n, char a, char b, char c, int* moves, item* q,
int(*code)[3])
{
  int min_step_nr =(1<<n)–1;
  int first = 0;
  q[0].s = a; q[0].d = b; q[0].h = c;
  q[0].l = 0; q[0].r = min_step_nr – 1;
  int last = 1;
  int m;
  while (first < last)
  {
    m = (q[first].l + q[first].r) >> 1;
    moves[m] = code[q[first].s–'a'][q[first].d–'a'];
    if (q[first].l < q[first].r)
    {
      q[last].s = q[first].s; q[last].d = q[first].h; q[last].h = q[first].d;
      q[last].l = q[first].l; q[last].r = m – 1;
      ++last;
      q[last].s = q[first].h; q[last].d = q[first].d; q[last].h = q[first].s;
      q[last].l = m + 1; q[last].r = q[first].r;
      ++last;
    }
  ++first;
  }
  for(i = 0; i < min_step_nr; ++i)
    printf("%d, ", moves[i]);
}
```

5 Dynamic programming

The recursive formulas can be interpreted as the ones that describe the way the optimal solution is built by optimal sub-solutions (principal of the optimality). The principle of the optimality was introduced by Richard Bellman [10], who called the corresponding recursive formulas as the functional equations of the problem. Dynamic programming follows this strategy: it starts from the optimal solutions of the trivial sub-problems and builds the optimal solutions of the more and more complex sub-problems and eventually of the original problem. In case of this problem the bottom-up way means that we solve the problem for k=1, 2, 3, ..., n. Although for a specific k there are six different k-size sub-problems ($P(k, a, b, c)$, $P(k, a, c, b)$, $P(k, b, c, a)$, $P(k, b, a, c)$, $P(k, c, a, b)$, $P(k, c, b, a)$), their solutions can be obtained from one another by simple letter-changing. For instance, the move-sequence that solves optimally the $H_{min}(k, c, a, b)$ sub-problem can be obtained from the optimal solution of the $H_{min}(k, a, b, c)$ sub-problem by changing letter a with c, letter b with a and letter c with a. Accordingly, for each k=1, 2, 3, ..., n only one “variant” of the k-size sub-problems would be enough to be solved.

The main difficulty of dynamic programming is that it is often nontrivial to establish what sub-problems in what order have to be solved. In Figure 2 the imbricated ellipses illustrate the bottom-up strategy the dynamic programming follows. Notice, that the optimal solution of the $H_{min}(3, a, b, c)$ problem implies only two 2-size and five 1-size optimal sub-solutions. The "dynamic programming order" the sub-problems (for n=3) have to be solved is the following: 1. $H_{min}(1, a, b, c)$; 2. $H_{min}(2, a, c, b)$ (built up from the optimal solutions of the $H_{min}(1, a, b, c)$ and $H_{min}(1, b, c, a)$ 1-size "variants"); 3. $H_{min}(3, a, b, c)$ (built up from the optimal solutions of the $H_{min}(2, a, c, b)$ and $H_{min}(2, c, b, a)$ 2-size "variants"). The growing sub-tree sequence included in the imbricated ellipse-sequence represents the "increasing" sub-problem sequence that, according to the dynamic programming strategy, has to be solved. In this sequence the current sub-problem succeeds its left-son-sub-problem, and precedes its father-sub-problem. For n odd/even these sub-problem sequences are the followings (in case of the minimum version of the problem):

$H_{min}(1, a, b, c)$, $H_{min}(2, a, c, b)$, $H_{min}(3, a, b, c)$, $H_{min}(4, a, c, b)$, . . . , $H_{min}(n, a, b, c)$.

$H_{min}(1, a, c, b)$, $H_{min}(2, a, b, c)$, $H_{min}(3, a, c, b)$, $H_{min}(4, a, b, c)$, . . . , $H_{min}(n, a, b, c)$.

If it is difficult to build an iterative algorithm that determines this "dynamic programming order", than, it is advisable to try to use the recursive formula dictated order. Unfortunately the direct transcription of the recursive formula into recursive procedure usually results in inefficient divide and conquer algorithm. Since dynamic programming problems are often characterized by overlapping sub-problems, the standard divide and conquer approach commonly results in repeated evaluation of the identical sub-problems. To avoid this ingredient the so-called "recursion with result caching" (memoization) technique can be applied. According to this technique ones a sub-problem has been solved its optimal solution (often the optimal value of the target function) is stored (memorized), and whenever later the recursive algorithm meets again the same sub-problem its stored solution is simply retrieved.

As specificness, in case of Towers of Hanoi problem solving a sub-problem *means* to solve its son-sub-problems *effectively.* Consequently, all sub-problems have to be solved as many times as the algorithm meets them (as in case of divide and conquer problems). Nevertheless we could do to store the code of the optimal solutions of the solved sub-problems. (This method also works only for moderate values of parameter n) Moreover, as we pointed out above, it would be enough to store the code of the optimal solution of only one variant for each k=1, 2, 3, . . . , n. Unfortunately, managing (storing, retrieving, generating from one another, printing) these exponential-size move-sequences has the same time-complexity as generating them again (The optimal solution of the k-size problem has the same number of moves as the number of nodes of the corresponding k-level binary tree). Consequently, the memoization technique, in this case, do not decrease (compared to the direct, divide and conquer like implementations of the formulas f_{min} and f_{max}) the time complexity of the algorithms.

Returning to the "recursion with result caching" technique, notice that in the sub-problem sequence to be solved there are only to type of problems: $H_{min}(k, a, b, c)$ and $H_{min}(k, a, c, b)$. In case of first-type sub-problems ($H_{min}(k, a, b, c)$) the move of the largest disc is $a \to b$ (coded with 0). Furthermore, after the optimal solution of the left-son-sub-problem ($H_{min}(k-1, a, c, b)$) has been stored, the optimal move-sequence for the right-son-sub-problem ($H_{min}(k-1, c, b, a)$) is generated by the following letter-changes: a is changed with c, b is changed with a and c is changed with b. According to these letter-changes we have the following move-changes: change move 0 with move 4, move 1 with move 3, move 2 with move 0, move 3 with move 5, move 4 with move 2, move 5 with move 1. For the second type problems ($H_{min}(k, a, c, b)$) the largest-disc-move is $a \to c$ (coded with 1). The corresponding move-changes are the followings: change move 0 with move 2, move 1 with move 5, move 2 with move 4, move 3 with move 1, move 4 with move 0, move 5 with move 3.

Procedure P_memoization_min applies the memoization technique. Rows 0 and 1 of array move_changing[0..1][0..5] store the two move-changing patterns to be applied (alternatively). (Binary variable pattern indicates the largest-disc-move and the move-changing-pattern to be applied)

```
void P_memozation_min(int k, char s, char d, char h, int n, int *moves)
{
  int i, p, pattern;
  pattern = ((n+k)&1);
  if(k==1)
    moves[0]=pattern;
  else
  {
    P_memozation_min(k-1, s, h, d, n, moves);
    p=(1<<(k-1))-1; //the number of moves in the son-sub-problems
    moves[p]=pattern; //the move of the largest disc
    for(i=0;i<p;++i) //generating the right-son-solution
      moves[p+1+i] = move_changing[pattern][moves[i]];
  }
}
```

What can we say as a conclusion for this ? Since the recursive formulas implemented by procedures P_recursive_min (P_recursive_max) and P_memoization_min are direct materializations of the principle of the optimality, and what is more, they accomplish (along the back-way of the recursion) the bottom-up building process prescribed by this principle, the strategy these procedures implement can be considered dynamic programming. More exactly: recursive dynamic programming. Most exactly (especially in case of procedures P_recursive_min and P_recursive_max): divide and conquer like recursive dynamic programming algorithms.

Remark: If we analyse the recursive formulas in top-down direction, then they describe the way the greedy decisions reduce the problem to similar, simpler sub-problems. The bottom-up analysis of the same formulas shows the way the optimal solution (of the problem) is built on the score of the optimal sub-solutions (of the sub-problems). This is why the algorithm can be seen both greedy and dynamic programming strategy.

6 Backtracking

According to the backtracking strategy, we try to find the optimal solution as a move-sequence. At each stage of the problem-solving process there are three possible moves. Considering the three pegs "large", "medium" and "small" according to the size of their top-disc, the three moves are: "small"→"medium", "small"→"large", "medium"→"large". The solution-space of the problem can also be represented by a rooted-tree. The nodes correspond to the stages of the problem. Each stage can be characterised by a set-triplet. The sets of a give triplet contain the discs from the corresponding pegs. Since at each stage there are three possible moves, all nodes (expecting the leaves) have three son-nodes. The root-node represents the initial stage of the problem when all discs are on peg a: $\{(1, 2, \ldots, n);();()\}$. In the final stage, corresponding to the solution-leaves, all discs are on peg b: $\{();(1, 2, \ldots, n);()\}$. The shortest/longest root – ,,solution-leaf" path, represents the optimal solution.

The backtracking algorithms apply depth-first search (usually implemented recursively), and choose the optimal solution by the standard minimum-/maximum-search method. In case of the Towers of Hanoi problem the backtracking strategy, in its primitive form, avoids only the loops and it is very inefficient. For example, the tree representing the structure of the optimal solution of the minimum version of the problem (Figure 2) has $2^n - 1$ nodes (supposing that "root-nodes" assimilate their "own leaves"), equal with the numbers of the moves along the optimal move-sequence. In case of the solution-space-tree, only its shortest root – "solution-leaf" path contains so many nodes. Furthermore, since the backtracking algorithm also needs to store the code of the current move-sequence, this algorithm also works only for moderate-values of parameter n.

The elements of array pegs[0..2] store the number of discs on the corresponding peg (pegs[i].nr), and the discs themselves (pegs[i].discs[0..(n-1)]). The source and destination pegs corresponding to the six possible moves are memorized in array move_types[0..5]. Arrays moves and states store the current moves- and state-sequence. (In a given stage of the solution building process the state of the problem can be described by the peg-sequence corresponding to the positions the discs occupy) Function valid verifies if move i is valide as next step (k). Procedures move_forward and move_backward move and remove the current disc.

```
void P_backtrack_min(int k, ITEM_P*pegs, ITEM_MT*move_types,
int*moves, ITEM_ST*states)
{
  if (states[k]==ENDSTATE)
  {if(k<kmin){kmin=k; copy(opt_solution, moves, kmin);}}
  else
  {
    for(i=0; i<6;++i)
    {
      if(valid(i, k, pegs, move_types, states))
      {
        move_forward(i, k, pegs, move_types, states, moves);
        P_backtrack_min(k+1, pegs, move_types, moves, states);
        move_backward(i, k, pegs, move_types, states);
      }
    }
  }
}
```

To optimise a backtracking algorithm means to reduce the traversed part of the solution-space-tree. An utterly optimised backtracking algorithm traverses only the optimal root – "solution-leaf" path. This means that we are able to establish at each stage the optimal move. However such an algorithm would be rather greedy or dynamic programming than backtracking. (See "The iterative implementation" section.)

7 The iterative implementation

In the literature [6], [7], [8] and others debates on non-recursive dynamic programming algorithms, as solutions of the Towers of Hanoi puzzle. The ideas analysed in previous sections raise the following question: Is it possible to generate the move-sequence that represents the optimal solution iteratively? It is not hard to realize that procedure P_memozation_min can be easily transcribed to an iterative dynamic programming algorithm.

According to the "dynamic programming order" presented above, the algorithm advances from father to father. The optimal solution of any father-sub-problem can be determined on the score of the optimal solution of its left-son-sub-problem. If the current sub-problem in the bottom-up building process is $H_{min}(k, x, y, z)$, then $H_{min}(k, y, z, x)$ is its right-brother-sub-problem, $H_{min}(k + 1, x, z, y)$ is its father-sub-problem, and the move the largest disc of the father-sub-problems has to perform is $x \to z$. As we previously mentioned, the optimal solution of the right-brother-sub-problem can be found from the solution of the left-brother-sub-problem by simply letter changing. To solve the $H_{min}(k + 1, x, z, y)$ father-sub-problem means, that, after the current left-son-sub-problem ($H_{min}(k, x, y, z)$) has been solved, we move disc $(k + 1)$ from peg x to peg z, and than we generate the solution of the right-son-sub-problem ($H_{min}(k, y, z, x)$). The starting problem is $H_{min}(1, a, b, c)$ or $H_{min}(1, a, c, b)$ depending if n is odd or even. Procedure P_iterative_DP_min follows this iterative dynamic programming strategy.

```
void P_iterative_DP_min(int n, int *moves)
{
  int i, k, pattern, p;
  moves[0] = pattern = !(n&1); //the first move
  for(k=2;k<=n;++k)
  {
    p=(1<<(k–1))–1;
    pattern ^ = 1; //we change the pattern
    moves[p]=pattern; //the move of the largest disc
```

```
                    //corresponding to the father problem
    for(i=0;i<p;++i) //generating the righ-brother-solution
      moves[p+1+i] = move_changing[pattern][moves[i]];
  }
}
```

The main concern about procedure P_iterative_DP_min is related (as we described above) with the limited memory capacity of the computers. How can we eliminate this ingredient? The solution is based on the following observations (We have considered the minimum version of the problem; We consider the three pegs "large", "medium" and "small" according to the size of their top-disc):

• In any intermediate state of the problem solving process there are three possible moves: "small"$\rightarrow$"medium", "small"$\rightarrow$"large", "medium"$\rightarrow$"large".

• The top-disc of the "small-peg" is always the smallest disc. It is not allowed two consecutive moves with the smallest disc. (To avoid the loops, and to maintain the minimal character of the move-sequence). So, the smallest and the medium-size top-discs move alternatively.

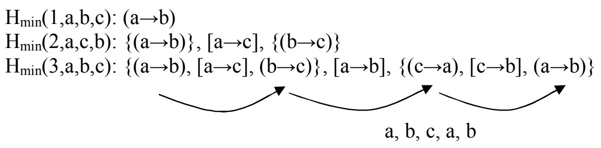
• If the next to move is the medium-size top-disc, than it is clear that the "medium"$\rightarrow$"large" move has to be performed. The ,,own leaves" of the root-nodes represent these moves. (Figure 2)

• The smallest disc (disc n) moves according to the patterns (a, b, c, a, b, c, ...) or (a, c, b, a, c, b, ...) depending on the odd or even character of parameter n. Consequently, the moves of the smallest-disc are also unambiguously. (There is only one optimal-size solution)

What reasoning lies behind the last remark? These move-patterns can be established by a careful analysis of the bottom-up building process, and the correctness of them is proved by mathematical induction. We assume that the problem to be solved is $H_{min}(n, a, b, c)$, and n is odd. We will prove that in this case the smallest disc follows the (a, b, c, a, b, c, ..., c, a, b) 2^{n-1} long move-sequence.

For n=1 we have only the smallest disc and its move is $a \rightarrow b$. Assume that for a given odd $n > 1$ the 2^{n-1} long move-sequence of the smallest disc is: a, b, c, a, b, c, ..., c, a, b. The right-brother sub-problem of problem $H_{min}(n, a, b, c)$ is $H_{min}(n, b, c, a)$. The corresponding move-sequence for the smallest disc is: b, c, a, b, c, a, ..., a, b, c. Concating these patterns, we get the 2^n long move-sequence corresponding to the father-sub-problem $H_{min}(n + 1, a, c, b)$: a, b, c, a, b, c, ..., a, b, c. Repeating this procedure we get for the $H_{min}(n + 2, a, b, c)$ grandfather-sub-problem the 2^{n+1} long a, b,

c, a, b, c, ..., c, a, b move-sequence-pattern. We can use the same train of thought for n even.



H_min(1,a,b,c): (a→b)
H_min(2,a,c,b): {(a→b)}, [a→c], {(b→c)}
H_min(3,a,b,c): {(a→b), [a→c], (b→c)}, [a→b], {(c→a), [c→b], (a→b)}
a, b, c, a, b

Figure 3: We used round brackets for the moves of the smallest disc and square brackets for the moves of the largest disc of the corresponding sub-problem $(k > 1)$. The curly brackets-pairs represent the brother sub-problems.

Consequently: the strategy applied by this iterative algorithm is mainly dynamic programming due two the following reasons:

• The algorithm generates the move-sequence that implements the bottom-up solution-building process of the dynamic programming strategy.

• The way the move-sequence is established is rooted in optimalisations included in the principal of the optimality.

• The move-pattern the smallest disc follows can be determined by bottom-up analyses of the recursive formulas.

Interestingly, the move-patters the smallest disc has to follows can also be determined by a greedy approach of the problem. Solving the $H_{min}(n, a, b, c)$ problem we have the following moving-patterns:

• all n discs get from peg a to peg b: $a >> b$; (between stages a and b may also be other stages)

• disc 1 moves: $a \rightarrow b$ (disc 1 moves directly from peg a to peg b; greedy-move)

• top (n-1) discs: $a >> c >> b$ (top (n-1) discs pass through stages a, c, b)

• disc 2 moves: $a \rightarrow c \rightarrow b$, greedy-moves

• top (n-2) discs: $a >> b >> c >> a >> b$;

• disc 3 moves: $a \rightarrow b \rightarrow c \rightarrow a \rightarrow b$, greedy-moves

• top (n-3) discs: $a >> c >> b >> a >> c >> b >> a >> c >> b$;

• disc 4 moves: $a \rightarrow c \rightarrow b \rightarrow a \rightarrow c \rightarrow b \rightarrow a \rightarrow c \rightarrow b$, greedy-moves

• . . .

Since for all $k > 1$ sub-problem $H_{min}(k, x, y, z)$ is reduced to sub-problems $H_{min}(k-1, x, z, y)$ and $H_{min}(k-1, z, y, x)$ the next moving-pattern is generated from the current one by intercalating between all consecutive stages the "third stage". Notice that there are only two patters, one for odd-discs (a, b, c, a, b, c, ...) and one for even-discs (a, c, b, a, c, b, ...).

The fact that the $p_1 >> p_2 >> \ldots >> p_m$ stage-sequence of the k-size tower in case of its largest disc means $p_1 \rightarrow p_2 \rightarrow \ldots \rightarrow p_m$ direct move-sequence is based on greedy decisions. Furthermore, the above-presented "pattern generating process" is hand-by-hand with the top-down greedy strategy described previously. Accordingly, we can state, that (form this point of view) the move-patterns the iterative algorithm follows, can also be established in greedy way.

Procedure P_iterative_min implements the above-presented iterative algorithm. The current values in array state[1..n] represent the current state of the problem. Element state[i] store the peg corresponding to the position of disc i. Variable i represents the current disc that moves. During odd steps (k is odd) disc n moves. At even steps (k is even) the smallest disc, that is on different peg than disc n, moves. For odd i disc i follows the "increasing circular pattern": a, b, c, a, b, c,... In cases when i is even disc i has to move according to the "decreasing circular pattern": c, b, a, c, b, a, ...

```
void P_iterative_min(int n, char a, char b, char c)
{
  char p = (char*)calloc(n+1, sizeof(char));
  for(i=1;i<=n;++i) p[i] = a;
  int nr=(1<<n)–1;
  int k=1;
  while(k<=nr)
  {
    if(k&1) i=n; //the smalest disc moves
    else
      for(i=n-1;p[i]==p[n];––i);
      //the smallest disc, that is on different peg than disc n, moves
    printf("%c – > ",p[i]); //move from peg p[i]
    p[i]+=1-((i&1)<<1);
    // according to the odd/even character of i,
    // the ,,increasing/decreasing" pattern is followed
    if(p[i]==c+1) p[i]=a;
    elseif(p[i]==a-1) p[i]=c; // circular pattern are followed
    printf(" %c\n",p[i]); //move to peg p[i]
    ++k;
  }
}
```

8 Analysis of programs

The Towers of Hanoi puzzle was analyzed in the literature in many ways, such [2], [3], [4], [5]. In this paper we solved the problem according to the principles of the four major programming techniques. Including the iterative solution we have got 5 programs (P_recursive_min – divide and conquer solution, P_greedy_min – greedy solution, P_memozation_min – dynamic programming, P_backtrack_min – backtracking solution, and P_iterative_min – the iterative solution).

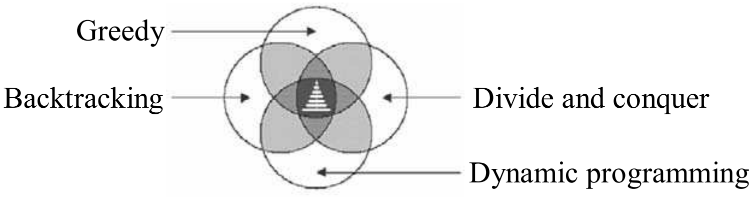


Figure 4: Towers of Hanoi – where programming techniques blend.

In the above mentioned programs, we used a time measuring sequence, as follows:

```
#include <windows.h>
__int64 freq, tStart, tStop;
unsigned long TimeDiff;
QueryPerformanceFrequency((LARGE_INTEGER*)&freq);
QueryPerformanceCounter((LARGE_INTEGER*)&tStart);
```

With this sequence we started the timer, then we called the program solving sequence. At the end, we calculate the time difference:

```
QueryPerformanceCounter((LARGE_INTEGER*)&tStop);
TimeDiff = (unsigned long)(((tStop − tStart) ∗ 1000000) / freq);
```

Using the 1 000 000 multiplier, we got the time in microsecond – a very precise chronometer. Because of the time sharing algorithm of the operating system, we did execute 20 times each program, and calculate an average of time-need.

We tested the programs for 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 15, and 20 discs. During the tests we disabled any I/O operation (ex. printf).

Table 1 concludes the tests, and the time-needs.

Technical information: We measured the time in microsecond. The computer: Intel Pentium 4; 2,40 GHz CPU; 1,50 GB RAM; the programs: Mi-

n	*steps* 2^n-1	*recursive*	*greedy*	*backtracking*	*dynamic*	*iterative*
1	1	1	1	5	1	1
2	3	1	1	15	1	1
3	7	1	1	143	1	1
4	15	2	2	29 022	2	2
5	31	3	3	1 791 167 126	2	2
6	63	5	5	„∞”	2	2
7	127	8	8	„∞”	3	3
8	255	14	15	„∞”	4	4
9	511	27	30	„∞”	6	6
10	1023	51	63	„∞”	10	11
15	32 767	1599	2071	„∞”	250	320
20	1 048 575	52 080	70 112	„∞”	10 511	10 849

Table 1: Comparative analysis of programs.

crosoft Visual C++ 6.0, Win32 Console Application.

According to the comparative analysis, we can conclude that the dynamic programming and the iterative solutions are the fastest (aprox. 5-times faster than the recursive, divide and conquer solution), the classic backtracking is the slowest. There is a very small difference between the iterative solution and the recursive dynamic programming solution.

9 Conclusions

As we pointed out in this paper the principle solving process of Towers of Hanoi problem mainly follows a greedy strategy. Then again, the way the optimal solution is built mostly follows the dynamic programming "way of thinking". The greedy algorithms produce only one decision-sequence, a sequence of greedy decisions. The dynamic programming strategies usually generate several optimal sub-sequences. The fact, that both the recursive and iterative algorithms generate only one decision-sequence is another reason why these solutions can be considered as greedy strategies implemented in dynamic programming way.

The recursive formulas are born in top-down greedy way, but they are implemented in bottom-up way as dynamic programming strategies. Furthermore, considering them in top-down / bottom-up way, they are materializations of the principal of greedy-chooses / optimality. Although the current sub-problem at each step is reduced to two/three sub-sub-problems (that is characteristic to divide and conquer strategies) the problem solving processes (both the principle and the implementation) are directed by greedy and dynamic programming optimalisations included in the recursive formulas. The fact that the problem solving process pretends the repeated "evaluation" of the overlapping sub-problems is also a divide and conquer feature of the problem.

With respect to the time complexity of the implementations, they are exponential, since the size of the optimal solutions depends exponentially on the size of the input. All algorithms traverse the same optimal-solution-tree with $(2^n - 1)$ or $(3^n - 1)$ nodes (expecting the backtracking algorithm that traverses the whole solution-space-tree). The running time differences between the recursive and iterative implementations can mainly be explained by the loss of time that arises due to the recursive calls. Whereas procedure P_recursive_min implements the bottom-up building process on the back-way of the recursion, procedure P_iterative_min performs this directly. Since the optimal solution has exponential size, even minor variances between implementations are reflected in exponential way with respect to the running time of the algorithms.

References

[1] N. Claus (pseudonym for Édouard Lucas), *La Tour d'Hanoï: V'eritable Cassetête Annamite*, Original instruction sheet printed by Paul Bousrez, Tours, 1883.

[2] Z. Kátai, *Algoritmusok felülnézetből*, Stientia Kiadó, Kolozsvár, 2006.

[3] D.T. Barnard, The Towers of Hanoi: An Exercise in Non-Recursive Algorithm Development, *Tech. Report 80-103*, Queen's University, 1980.

[4] J.P. Bode, A.M. Hinz, Results and open problems on the Tower of Hanoi, *Congr. Numer.*, **139** (1999) 113–122.

[5] P. Cull, E.F. Ecklund, Jr., Towers of Hanoi and analysis of algorithms, *Amer. Math. Monthly*, **92** (1985) 407–420.

[6] B. Eggers, The Towers of Hanoi: Yet another nonrecursive solution, *SIGPLAN Notices*, **20**, 9 (1985) 32–42.

[7] F.O. Ikpotokin, S.C. Chiemeke, E.O. Osaghae, Alternative approach to the optimal solution of tower of Hanoi, *J. Inst. Math. Comput. Sci.*, **15** (2004) 229–244.

[8] M. Sniedovich, OR/MS Games: 2. Towers of Hanoi Puzzle, *INFORMS Transactions on Education*, **3**, 1 (2002) 34–51.

[9] Hanoi tornyai, http://mattort.fvt.hu/, *Abacus Matematikai Lapok*, 2001-2002. .

[10] R.E. Bellman, *Dynamic Programming*, Princeton, New Jersey, 1957.

Received: November 18, 2008.