



NAME	ID
Mohamed Hossam El-Din El-Sayed	1808581
Mohamed Hesham Alsaied Albendary	1808442
Ahmed Salah Abdelhak	1803820
Ahmed Reda Ibrahim Taha Elawam	1802745
Mohamed Ashraf Farouk Ibrahim	1805188
Mohamed Ahmed Mahmoud Asran	1808040

Contents

Task 1	7
Assumptions	7
Problem Description	7
Detailed Solution	8
Complexity Analysis	11
Comparison With Another Algorithm	12
Sample Output	12
Conclusion	13
Task 2	14
Assumptions	14
Problem discription	14
Detailed solution	15
Complexity analysis	20
Comparison with another algorithm	21
Simple output of the algorithm	23
Conclusion	25

Task 3	26
Assumptions	26
Problem Description	26
Detailed Solution	27
Complexity Analysis	29
Comparison With Another Algorithm	30
Sample Output	32
Conclusion	33
Task 4	34
Assumptions	34
Problem Description	34
Detailed Solution	35
Complexity Analysis	37
Comparison With Another Algorithm	37
Sample Output	38
Conclusion	39
Task 5	40

Assumptions _____	40
Problem Descriptions _____	40
Detailed Solution _____	40
Pseudo code _____	41
Java Code _____	41
Complexity Analysis _____	42
Comparison With Another Algorithm _____	42
Sample output _____	42
Task 6 _____	43
Assumptions _____	43
Problem Description _____	43
Detailed Solution _____	43
Complexity Analysis _____	47
Comparison With Another Algorithm _____	47
Sample Output _____	48
Conclusion _____	51
References _____	52

Table of Figures

Figure 1	7
Figure 2	8
Figure 3	9
Figure 4	12
Figure 5	13
Figure 6	15
Figure 7	15
Figure 8	15
Figure 9	16
Figure 10	23
Figure 11	23
Figure 12	24
Figure 13	25
Figure 14	26
Figure 15	27

Figure 16.....	30
Figure 17.....	31
Figure 18.....	32
Figure 19.....	34
Figure 20.....	38
Figure 21.....	38
Figure 22.....	38
Figure 23.....	48
Figure 24.....	48
Figure 25.....	49
Figure 26.....	49
Figure 27.....	50
Figure 28.....	50

Task 1

Assumptions

User will input correct data type whenever they are asked for input (ex. When asked to enter number of rows, they won't enter a string for example). Also, the centers of the coins are assumed to be at the points of an equilateral triangular lattice

Problem Description

Inverting a Coin Triangle Consider an equilateral triangle formed by closely packed pennies or other identical coins like the one shown in **Error! Not a valid bookmark self-reference..** It's required to use iterative improvement method to design an algorithm to flip the triangle upside down in the minimum number of moves if on each move it's possible to slide one coin at a time to its new position.

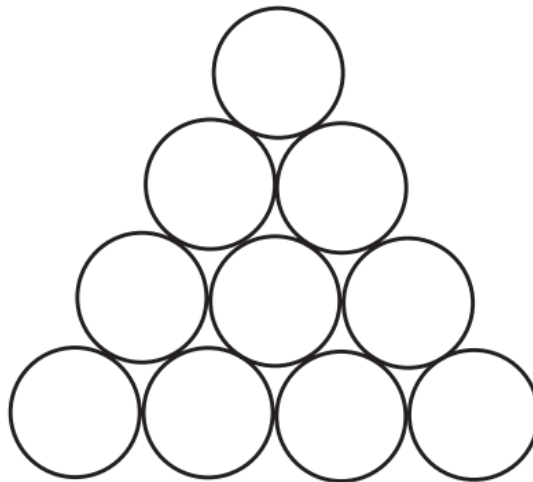


Figure 1

Detailed Solution

A trivial solution to flip a triangle with n rows is to just move all the coins to their new positions, which takes number of steps equal to number of coins in that triangle " $\frac{n*(n+1)}{2}$ " or 10 steps when applied to **Error! Not a valid bookmark self-reference.**(obviously inefficient!)

To flip the pyramid in the least number of moves, we will try to overlap both starting and ending positions together, this way the coins marked in green in Figure 2, will not be moved at all. And the ones marked in blue only will be moved to their new positions "Marked with white". So, in **Error! Not a valid bookmark self-reference.**, we will need to make only 3

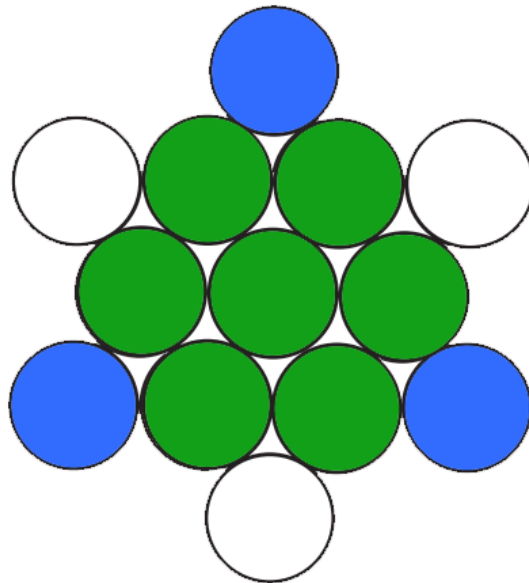


Figure 2

moves

Let's draw this triangle with 8 rows

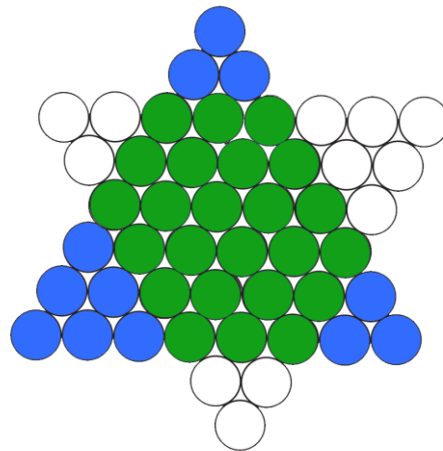


Figure 3

If we keep on drawing more and more triangles, we will find out that

1. The small blue triangle in the upper corner can always be moved to the corresponding small white triangle in the bottom
2. The small blue triangle in the bottom left corner can always be moved to the corresponding small white triangle in the upper right corner
3. The small blue triangle in the bottom right corner can always be moved to the corresponding small white triangle in the upper left corner

Table 1 shows the relation between number of rows n and the number of rows of each of the small triangles

Table 1

Number of rows n	Number of rows in left triangle a	Number of rows in upper triangle b	Number of rows in right triangle c
1	0	0	0
2	1	0	0
3	1	1	0
4	1	1	1
5	2	1	1
6	2	2	1
7	2	2	2
8	3	2	2

It's easy to prove that $a = \left\lfloor \frac{n+1}{3} \right\rfloor$, $b = \left\lfloor \frac{n}{3} \right\rfloor$, $c = \left\lfloor \frac{n-1}{3} \right\rfloor$ using data obtained in Table 1

Pseudo-code

```
ALGORITHM InvertTriangle(n)
  //INPUT: n - number of rows
  //OUTPUT: Inverted triangle
  //REQUIREMENTS: n > 0
  a <- [(n + 1) / 3]
  for i <- 1 to  $\frac{a*(a+1)}{2}$  do
    move one coin from bottom left triangle to upper right
    print triangle
  b <- [(n) / 3]
  for i <- 1 to  $\frac{b*(b+1)}{2}$  do
    move one coin from top triangle to bottom
    print triangle
  c <- [(n - 1) / 3]
  for i <- 1 to  $\frac{c*(c+1)}{2}$  do
    move one coin from bottom right triangle to upper left
    print triangle
  steps <- a + b + c
  return steps
```

Complexity Analysis

Complexity for each function

printTriangle: $O\left(\frac{n*(n+1)}{2}\right) \approx O(n^2)$

init: $O\left(\frac{n*(n+1)}{2}\right) \approx O(n^2)$

sum: $O(1)$

options: $O(1)$

mainmenu: $O(1)$

To calculate total complexity, recall that we make $sum(a) + sum(b) + sum(c)$ moves and in each move, we print the triangle. So, in total complexity is $(sum(a) + sum(b) + sum(c)) * O(n^2)$, since a, b, c are all functions in n , total complexity is $\approx O(n^4)$. However, the problem can be solved in $O(1)$ if we calculate only number of steps as $a + b + c$ without printing the triangle after each step

Comparison With Another Algorithm

Another way to calculate number of steps which is by using this formula $\lfloor \frac{n*(n+1)}{6} \rfloor$, this formula and my solution give the same number of steps for all values of n [\[1\]](#)

Sample Output

```
1. Start
2. Options
3. Exit
1
Enter the number of rows: 4
*
* *
* * *
* * * *

rotating the top rows

* *
* * *
* * * *
*
Steps = 1
rotating the left rows

* * *
* * *
* * *
*
Steps = 2
rotating the right rows

* * * *
* * *
* *
*
Steps = 3
Number of steps = 1 + 1 + 1 = 3
Press any key to continue . . .
```

Figure 4

Now we will try $n = 8$ but since it will produce a very long output, I'll go to options and disable step by step output so that we can see only number of steps as shown in Figure 5

```
Steps = 9
1. Start
2. Options
3. Exit
1
Enter the number of rows: 8
rotating the top rows
Steps = 1
Steps = 2
Steps = 3
rotating the left rows
Steps = 4
Steps = 5
Steps = 6
Steps = 7
Steps = 8
Steps = 9
rotating the right rows
Steps = 10
Steps = 11
Steps = 12
Number of steps = 6 + 3 + 3 = 12
Press any key to continue . . .
```

Figure 5

Conclusion

Inverting a triangle of coins using this algorithm uses iterative improvement since on each step we are getting closer to the goal result by doing the same step over and over (moving one coin between two small triangles)

Task 2

Assumptions

The input board size is positive number, the program alarms the user if he input an invalid board size (ie: odd number of board size).

Problem discription

In this problem we went to find the sequence of boards which lead to a single remaining peg in the board.

The sequence of steps to eliminate the pegs in the board is jumping with a peg over another peg to unoccupied cell then eliminating the peg which we jumped over, repeating these steps until the board became empty except for one cell occupied with a single peg.

There's linear timing approach for solving all solvable boards but if we want to make sure that the unsolvable boards are indeed unsolvable, we must use the dynamic programming approach by make a tree of all possible sequences of boards, then we check if the one or two of these sequences are the solution by checking of the length of the sequence equal the length of the board - 2.

Steps of one move

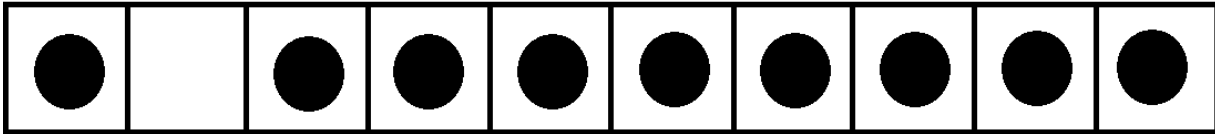


Figure 6

Moving a peg if it can move to unoccupied cell

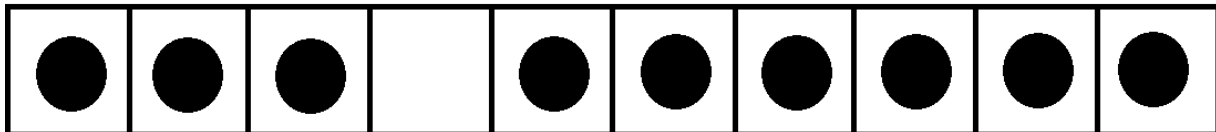


Figure 7

Removing the peg which the moved peg jumped over

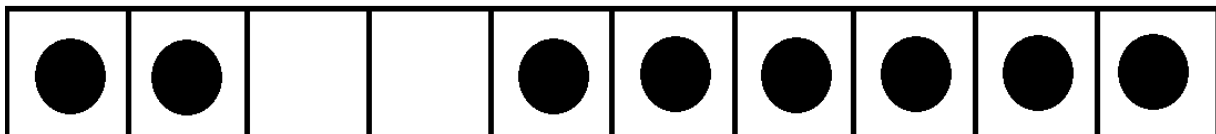


Figure 8

Detailed solution

Using the dynamic programming approach we first must construct a tree with the whole feasible moves of the initial board without repeating any sequence of boards twice this is done using memoization technique then we must traverse the tree to find if there's a solution to the board or not by checking if the sequence of moves (the depth of the tree at any point) = board_size - 2 finally we print the two complete solution if the board is solvable,

if the board is unsolvable we can print the longest sequences in the tree, which will lead to obsolete boards with no moves possible

Constructing the tree with all feasible solution

We can simply iterating over the board trying to move any peg in the tree then if we find a single peg to move we check if the resultant board after moving the peg to the new location is an already stored board, if yes we terminate the move any continue trying to move another peg, if no we add the board to the memoization data structure and then we recursively try to move another peg in the board until there's no new moves can be done in the board

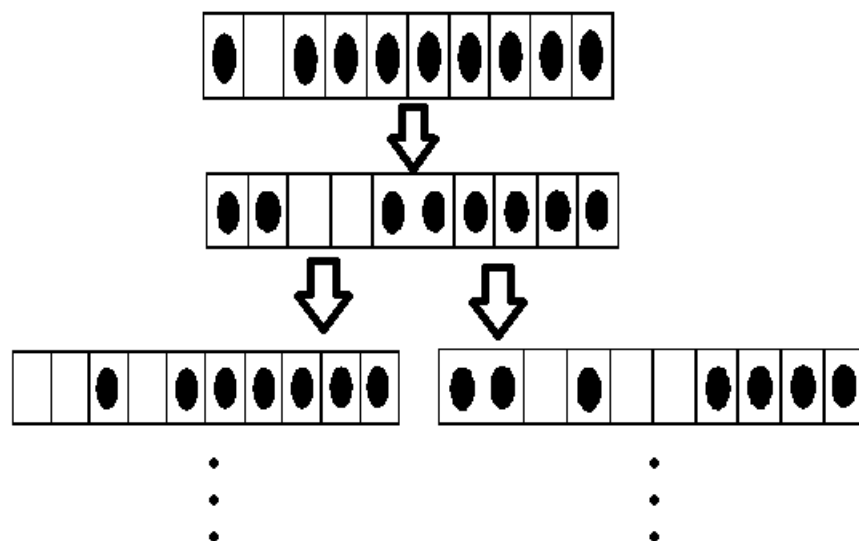


Figure 9

Memoization

We can use set structure to store all resultant boards, why set?

because using it we can add and find if the board already exist in logarithmic time, this possible because the binary search tree implementation

Traversing the solution tree

This can be done simply by in-order traverse for all nodes in the tree structure and saving the contents of every node (the board) in an array of boards to output the right solution in the next step.

Output the right solution if exist

After we get all unique possible moves from the tree as an array on sequences of boards, we can loop on this array to check if any of the sequence has the length of $\text{board_size} - 2$, if such sequence exists, we can return it as the solution of the board,

If such sequence does not exist, we know that the board has no solution (unsolvable board).

Representation of the board

we can represent the board by several data structure, this is especially easy in this problem because all cells have only two states occupied or not occupied.

optimal data structure I found for representing exceptionally large boards and compare them efficiently is by using BitSet structure then we can simply implement the comparator to be

used inside the set structure for comparing the different boards represented in different BitSets.

Pseudo code

ALGORITHM SOLVE1DPEGSOLITAIRE

// input: board_size

// output: sequence of boards

solveBoard(Board board)

Board tryBoard;

BitSet tryBS;

for i <- 3 to board_size do

 if (validateMove(board, I, RIGHT))

 tryBoard = board.clone()

 move(tryBoard, I, RIGHT)

 if (!memory.contains(tryBoard) do

 board.addChild(tryBoard)

 memory.add(tryBoard)

 Recursion: solveBoard(tryBoard)

```

for i <- 1 to board_size - 2 do

    if (validateMove(board, I, LEFT))

        tryBoard = board. clone()

        move(tryBoard, I, LEFT)

        if (!memory.contains(tryBoard) do

            board.addChild(tryBoard)

            memory.add(tryBoard)

            Recursion: solveBoard(tryBoard)

ArrayList<ArrayList<Board>> Solution = Traverse(headBoard)

Print(Solution)

```

Complexity analysis

The timing is proportional to the number of all possible boards which can we deduce from the initial board without any repetition, thanks to memoization technique

$T(n)$ = number of moves needed to solve the board * one move complexity

In average case: $T(n) = (n-2)*n \rightarrow O(N^2)$

In worst case: $T(n) = (2^{(n-1)} - 2) * n = n * 2^{(n-1)} - 2n$

Comparison with another algorithm

Another solution of this problem, is first check if the unoccupied cell is not the cell with number 2 or 5 from the right or the left, then there's no solution for this board, iterate over the board trying to move the pegs to the right of the unoccupied cell to the left, then the pegs to the left to the unoccupied cell to the right

```
private static ArrayList<ArrayList<Integer>> solveBoard(ArrayList<Integer> board) {

    ArrayList<ArrayList<Integer>> solution = new ArrayList<>();

    int noOfConversions = 0;

    boolean direction = LEFT;

    final int BOARD_SIZE = board.size() - 1;

    int nextMove = 4;

    solution.add(new ArrayList<>(board));

    while (noOfConversions < BOARD_SIZE - 2) {

        if (validateMove(board, nextMove, direction)) {

            board = move(board, nextMove, direction);
```

```

if (nextMove + 2 > BOARD_SIZE) {

    nextMove = 1;

    direction = RIGHT;

    solution.add(new ArrayList<>(board));

    noOfConversions++;

    continue;

}

nextMove = nextMove + 2;

noOfConversions++;

}

solution.add(new ArrayList<>(board));

}

return solution;

}

```

using this approach, we will find the solution in **linear time** but we cannot make sure that the unsolvable boards is indeed unsolvable and we can only find one of two solution with this specific implementation (though we can modify it to find the two solution of the board)

Simple output of the algorithm

```
The board size (have to be even number or 3 to be solved) = 6
The position of the empty slot from 1 to 6 is 2
//*****//
Solution number: 1

Step 0 : 1 0 1 1 1 1
Step 1 : 1 1 0 0 1 1
Step 2 : 1 1 0 1 0 0
Step 3 : 0 0 1 1 0 0
Step 4 : 0 1 0 0 0 0
//*****//
//*****//
Solution number: 2

Step 0 : 1 0 1 1 1 1
Step 1 : 1 1 0 0 1 1
Step 2 : 1 1 0 1 0 0
Step 3 : 0 0 1 1 0 0
Step 4 : 0 0 0 0 1 0
//*****//
```

Figure 11

```
The board size (have to be even number or 3 to be solved) = 10
The position of the empty slot from 1 to 10 is 5
//*****//
Solution number: 1

Step 0 : 1 1 1 1 0 1 1 1 1 1
Step 1 : 1 1 0 0 1 1 1 1 1 1
Step 2 : 1 1 0 1 0 0 1 1 1 1
Step 3 : 1 1 0 1 0 1 0 0 1 1
Step 4 : 1 1 0 1 0 1 0 1 0 0
Step 5 : 0 0 1 1 0 1 0 1 0 0
Step 6 : 0 0 0 0 1 1 0 1 0 0
Step 7 : 0 0 0 0 0 0 1 1 0 0
Step 8 : 0 0 0 0 0 0 1 0 0 0
//*****//
//*****//
Solution number: 2

Step 0 : 1 1 1 1 0 1 1 1 1 1
Step 1 : 1 1 0 0 1 1 1 1 1 1
Step 2 : 1 1 0 1 0 0 1 1 1 1
Step 3 : 1 1 0 1 0 1 0 0 1 1
Step 4 : 1 1 0 1 0 1 0 1 0 0
Step 5 : 0 0 1 1 0 1 0 1 0 0
Step 6 : 0 0 0 0 1 1 0 1 0 0
Step 7 : 0 0 0 0 0 0 1 1 0 0
Step 8 : 0 0 0 0 0 0 0 0 1 0
//*****//
```

Figure 10

Unsolvable board

```
The board size (have to be even number or 3 to be solved) = 6
The position of the empty slot from 1 to 6 is 4
Unsolvable Board
Largest tries :
//*****//
Largest try number: 1

Step 0 :    1    1    1    0    1    1
Step 1 :    1    1    1    1    0    0
Step 2 :    1    1    0    0    1    0
Step 3 :    0    0    1    0    1    0
Unsolvable Board
//*****//
```

Figure 12

Conclusion

Question B)

For board to be solvable the cells with numbers 3 and 4 must be empty after the first move, this can be achieved if the initial board is unoccupied in cell number 2 or 5

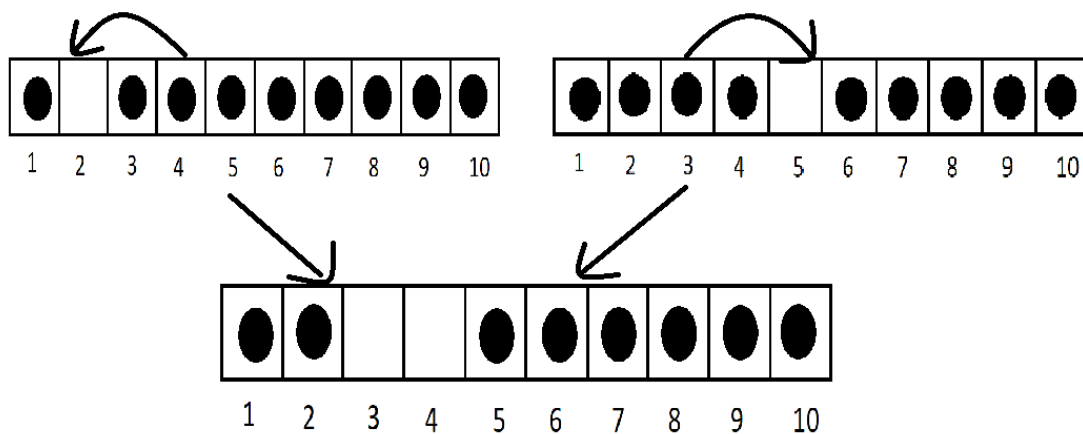


Figure 13

The final position for the only remaining peg in this case will be in the cell number $\text{board_size} - 1$ or $\text{board_size} - 4$.

And the board is symmetric: the board must be empty in cell number $\text{board_size} - 1$ or $\text{board_size} - 4$ and in this case The final position for the only remaining peg will be in the cell number 2 or 5.

Task 3

Assumptions

User will make only two knight arrays, one of them for black knights and other one for white knights

Each knight has its own type black or white, start position and ending arrival position.

Program will automatically initialize start position and arrival position for each knight to meet problem criteria

Each knight can't visit same position twice to grantee the minimum number of movements

Problem Description

There are six knights on a 3×4 chessboard: the three white knights are at the bottom row, and the three black knights are at the top row. Design a divide and conquer algorithm to exchange the knights to get the position shown on the right of the figure in the minimum number of knight moves, not allowing more than one knight on a square at any time.

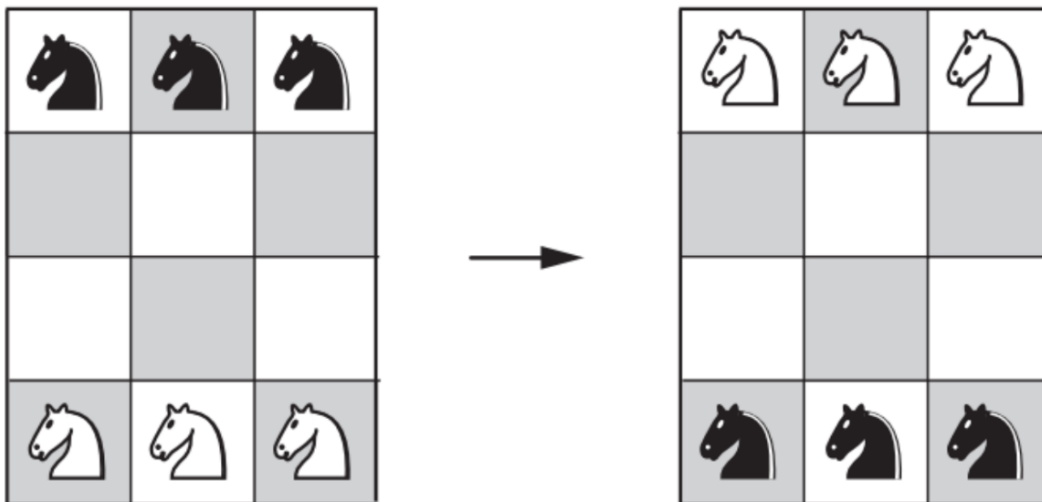


Figure 14

Detailed Solution

On calling Invade “the recursive function”, first we check if each knight has arrived at his arrival position “base case”

If not, all knights have arrived to their positions, we loop on the two arrays of knights (black and whites) and each non-arrived knight move according to the knightMoveList

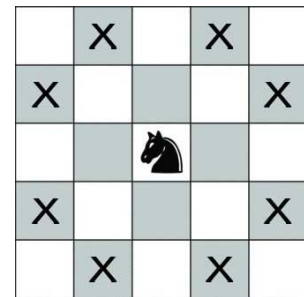


Figure 15

knightMoveList list is simply list of all available moves that each knight can move in his current position and other knights’ positions and is not visited before

So, we check the 8 moves of each knight and if there is one available it’s pushed into the knightMoveList

for each knight knightMoveList is sorted according to priority queue as black knights’ moves are prioritized toward down and white knights prioritized towards up and side moves is prioritized in both types of knights

Simulate function used to print the board pieces in form of string

Solved function iterate on each knight and check if it’s arrived or not

Each knight carries boolean state called arrived marked true on arrival

Pseudo-code

```
ALGORITHM solution (knight blackKnight[3], knight whiteKnight[3])
//INPUT: two arrays of knights, one for black and one for white
//OUTPUT: solution of the problem and output after each recursive call
for i <- 0 to 3 do
    initialization of each knight according to the problem statement

Invade (knight blackKnight[3], knight whiteKnight[3])
```

```
ALGORITHM Invade (knight blackKnight[3], knight whiteKnight[3])
//INPUT: two arrays of knights, one for black and one for white
//OUTPUT: board after switching both black and white knights
If(solved(blackKnight , whiteKnight))
    return
for i <- 0 to 3 do
    whiteMoves = whiteKnight[i].knightMoveList()
    sort moves of white knight[i]
    if(knightMoves)
        whiteknight[i].move(whiteMoves[0])
    blackMoves = blackKnight[i].knightMoveList()
    sort moves of black knight[i]
    if(knightMoves)
        blackKnight[i].move(blackMoves[0])

Invade (knight blackKnight[3], knight whiteKnight[3])
```

```
ALGORITHM solved (knight blackKnight[3], knight whiteKnight[3])
//INPUT: two arrays of knights, one for black and one for white
//OUTPUT: true if all knights has arrived otherwise false
for i <- 0 to 3 do
    if(!blackKnight[i].arrived || !whiteKnight[i].arrived)
        return true
```

Complexity Analysis

Complexity for each function

// n is number of knights which is fixed as 3, therefore all function of (n) is actually $O(1)$

// or divide and conquer technique is transform and conquer

knightMoveList: $O(1)$

ArrivalChecker: $O(1)$

Move: $O(1)$

Solved: $O(n)$

Simulate: $O(n)$

Invade: $O(n+P(n))$, where p is the new form of problem after transformation

Total complexity depends on the $p(n)$ which is the new transformation of the problem

And worst-case complexity is undefined when knights get stuck

But to avoid that we set arrival positions and sort moveList according to destination

And best Case in our problem is 5 recursive calls so best-case complexity is

$n + n + n + n + n = 5n$, which is $O(n)$

Comparison With Another Algorithm

Graph unfolding number of steps is 16

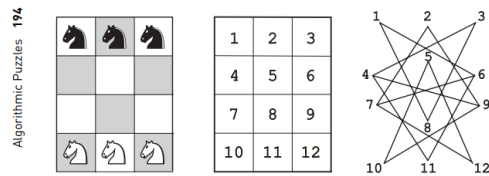


FIGURE 4.81 Graph representing possible moves in the Six Knights puzzle.

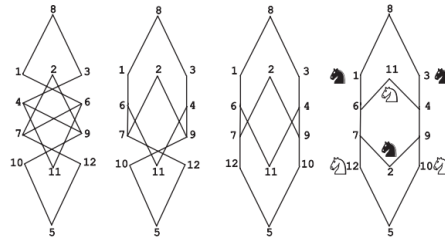


FIGURE 4.82 Unfolding of the graph in Figure 4.81.

positions at 10 and 11. The symmetry implies the same lower bound of 7 moves for the white knights to reach their goal positions. Thus, the puzzle cannot be solved in fewer than 14 moves. This cannot be done in 14 or 15 moves, however, which is easy to prove by contradiction. Assume that there is a sequence of moves that solves the puzzle in less than 16 moves. In such a sequence, the black knight at 1 will have to reach 12 in 3 moves because it cannot do this in 4 moves and if it does this in 5 or more moves the total number of moves will be at least $(5 + 4) + 7 = 16$. But before the black knight in 1 can move from 6 to 7, the white knight at 12 must go to 2, which must be preceded by the black knight at 2 move to 9, which must be preceded by the white knight at 10 move to 4, which must be preceded by the black knight at 3 move to 11, which must be preceded by the white knight at 11 move to 6, blocking the black knight at 1 and thus making the necessary number of moves exceed 15. But the puzzle can be solved in 16 moves, for example, those in the following sequence:

$B(1 - 6 - 7), W(11 - 6 - 1), B(3 - 4 - 11), W(10 - 9 - 4 - 3),$
 $B(2 - 9 - 10), B(7 - 6), W(12 - 7 - 2), B(6 - 7 - 12).$

Figure 16

Other research on solving this problem using

DFS – BFS – cost uniform – A* easy – A* same column – A* average

Source: <https://github.com/sirbuoanabianca/GuariniPuzzleGame>

Expanded nodes

	dificultate 1	dificultate 2	dificultate 3
dfs	14437	notDef	notDef
bfs	17175	898770	notDef
cost uniform	17175	898770	notDef
A* easy	1615	117	227
A* same column	162	47	29
A* average	252	177	104

Cost

	dificultate 1	dificultate 2	dificultate 3
dfs	984	notDef	notDef
bfs	16	16	notDef
cost uniform	16	16	notDef
A* easy	16	18	22
A* same column	18	18	22
A* average	18	20	22

Running time (sec)

	dificultate 1	dificultate 2	dificultate 3
dfs	5.5	notDef	notDef
bfs	4.8	484.3	notDef
cost uniform	100.8	> 10 ore	notDef
A* easy	3.6	0.19	2.38
A* same column	0.06	0.13	0.07
A* average	0.21	0.51	0.6

Figure 17

Sample Output

```
Select C:\Users\Arela\Documents\CodeBlocksDocs\Cpp\bin\Debug\Cpp.exe
B0  B1  B2
00  00  00
00  00  00
W0  W1  W2
#####
00  00  00
W1  W2  B0
B1  B2  W0
00  00  00
#####
B2  W0  W1
00  00  00
00  00  00
W2  B0  B1
#####
00  W0  00
00  B1  B2
00  W1  W2
00  B0  00
#####
W1  W0  00
W2  00  00
B2  00  00
B1  B0  00
#####
W1  W0  W2
00  00  00
00  00  00
B1  B0  B2
#####
total number of knight moves=22
Process returned 0 (0x0)   execution time : 0.025 s
Press any key to continue.
_
```

Figure 18

We return the shape of the board after each recursive call, and we increment the counter on each move to determine total number of moves

Conclusion

Exchanging the position of black and white knights using divide and conquer requires changing the positions of knights into a closer position to target positions and recursively moving until reaching the position

Task 4

Assumptions

The user will input a positive integer number of pennies.

Problem Description

A “machine” consists of a row of boxes. To start, one places n pennies in the leftmost box. The machine then redistributes the pennies as follows. On each iteration, it replaces a pair of pennies in one box with a single penny in the next box to the right. The iterations stop when there is no box with more than one coin as shown in Figure 19 that shows the work of the machine in distributing six pennies by always selecting a pair of pennies in the leftmost box with at least two coins. It’s required to design an algorithm using greedy method automate the machine.

6				
4	1			
2	2			
0	3			
0	1	1		

Figure 19

Detailed Solution

Pseudo-code

```
ALGORITHM pennyMachine(n)
//INPUT: n - number of pennies
//OUTPUT: machine's distribution of n pennies
result[i] <- n
Do
    result[i+1] <- (result[i] / 2)
    result[i] <- (result[i] % 2)
    i++
While result[i] > 1
return result
```

C++ Code with detailed steps' explanation

```
#include <iostream>
#include <vector>
using namespace std;
vector<int> pennyMachine(int n)
{/* n represents the number of pennies */
    int index=0;
    vector<int> result;
    /* Put the number of in the first box */
    result.push_back(n);
    /* Iterate on each box if it's greater than 1
     * and Divide the value in the box by 2.
     * Put the remainder in the same box
     * and the division result in the next box */
```

```

do{
    result.push_back(result.at(index) / 2);
    result.at(index) = result.at(index) % 2;
    index++;
}while(result.at(index) > 1);
/* Return the result boxes */
return result;
}

int main()
{
    /* Ask the user for the number of pennies */
    int pennies=0;
    cout << "Enter number of pennies: ";
    cin >> pennies;
    vector<int> boxes = pennyMachine(pennies);
    /* Print the final boxes distribution */
    int length = boxes.size();
    for(int j=0; j<length; j++)
    {
        cout << boxes.at(j) << " ";
    }
}

```

Complexity Analysis

- Time Complexity of the algorithm is **$O(\log(n))$**
which is shown from the division of the number of pennies by 2 each time to get the final result.
- Space Complexity of the algorithm is also **$O(\log(n))$**
which is shown from the number of boxes used to store the final values of the distribution of pennies.

Comparison With Another Algorithm

Another way to solve this problem can be reached using a brute force algorithm as follows:

```
ALGORITHM pennyMachine(n)
//INPUT: n - number of pennies
//OUTPUT: machine's distribution of n pennies
result[i] <- n
While result[i] > 1
    result[i] <- (result[i] - 2)
    nextIndex <- (nextIndex + 1)
    if result[i] <= 1
        result[i+1] <- (nextIndex)
        nextIndex <- 0
        i <- (i + 1)
return result
```

But this algorithm is slower than the greedy one as its time complexity is **$O(n)$** .

Sample Output

Sample output of randomly selected input:

```
<terminated> (exit value: 0) Task4.exe [C/C++ Application]  
Enter number of pennies: 6  
0 1 1
```

Figure 20

```
<terminated> (exit value: 0) Task4.exe [C/C++ Application]  
Enter number of pennies: 15  
1 1 1 1
```

Figure 21

```
<terminated> (exit value: 0) Task4.exe [C/C++ Application]  
Enter number of pennies: 30  
0 1 1 1 1
```

Figure 22

Conclusion

- a) The final distribution of pennies does not depend on the order in which the machine processes the coin pairs as it divides the number in the first box by 2 and puts the remainder in the same box then puts the result in the next box. These steps are repeated till each box has a 0 or 1.
- b) The minimum number of boxes needed to distribute n pennies is the minimum number of bits needed to represent a decimal number which is **$\text{ceil}(\text{Log}_2(n+1))$** as n is the number of pennies.
- c) Using the greedy algorithm, the machine needs **$\text{ceil}(\text{Log}_2(n+1)) - 1$** iterations to distribute the pennies before stopping.

Task 5

Assumptions

The result of the algorithm is less than $2^{64}-1$

Problem Descriptions

There is a row of n security switches protecting a military installation entrance. The switches can be manipulated as follows:

- (i) The rightmost switch may be turned on or off at will.
- (ii) Any other switch may be turned on or off only if the switch to its immediate right is on and all the other switches to its right, if any, are off.
- (iii) Only one switch may be toggled at a time.

The goal is to design a divide and conquer solution to get the minimum number of moves to turn all the lamps off.

Detailed Solution

To turn all lamps off we need to turn them off one by one from the left, in order to turn the lamp at index 0 from the left off, we need to turn off all the lamps starting from index 2 through n (that is $n-2$ lamps) and then turn the lamp at index 0, after that we need to turn them back on, and then we recurse on $n-1$, until we reach the base case that's the case of 1 which equals to 1

Pseudo code

Func solve (int n):

If n == 1 then return 1

Else return 1 + 2*solve(n-2) + solve(n-1)

Java Code

```
import java.io.*;
import java.util.*;

public class problem5{
    static Scanner in = new Scanner(System.in);
    static PrintWriter out = new PrintWriter(System.out);

    static final int MAXN = 100;
    static long[] mem = new long[MAXN];

    public static void main(String[] args){
        System.out.print("Enter the value of n: ");
        int n = in.nextInt();

        System.out.println();
        long answer = solve(n);
        System.out.println("The minimum number of moves is: "+answer);
    }
}
```

```

public static long solve(int n){
    if(n<=2) return n;
    if(mem[n] != 0)
        return mem[n];

    mem[n] = 1 + 2*solve(n-2) + solve(n-1);
    return mem[n];
}
}

```

Complexity Analysis

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

$$T(n) = \frac{2}{3} 2^n - \frac{1}{6} (-1)^n - \frac{1}{2}$$

$$O(2^n)$$

Comparison With Another Algorithm

The problem could be solved with dynamic programming technique which will make the algorithm run for a little wider range of input,

The algorithm is almost the same but we add another array to remember the value of the previous calculated results in this way we can get better results

Sample output

```

Enter the value of n: 5
The minimum number of moves is: 21
|

```

Task 6

Assumptions

- The user will input a positive integer number from 0 to 63.
- The source peg is A, the destination peg is D, and the other two auxiliary pegs are B and C.

Problem Description

There are eight disks of different sizes and four pegs. Initially, all the disks are on the first peg in order of size, the largest on the bottom and the smallest on the top. Use dynamic programming method to transfer all the disks to another peg by a sequence of moves. Only one disk can be moved at a time, and it is forbidden to place a larger disk on top of a smaller one. Does the dynamic programming method can solve the puzzle in 33 moves? If not then design an algorithm that solves the puzzle in 33 moves.

Detailed Solution

Since this problem is a deviation from the Towers of Hanoi, it can be solved using the same recursive concept, but with the aid of dynamic programming to save our answers and greatly reduce the recursive calls made.

The trivial cases are moving 0, 1, or 2 disks from A to D and the answer for these cases are 0, 1, and 3 respectively.

In the case of moving n disks, where $n > 2$ and $n < 64$, k disks should be moved from A to one of the 2 auxiliary pegs, B or C, using the 4 pegs in order to achieve this. Then, the remaining $n - k$ disks are moved to the destination peg D using the remaining 3 pegs, which is exactly the Towers of Hanoi problem. Finally, the k disks are moved to D using all four pegs.

So, we have to identify which value to k would result in moving all disks from A to D using the minimum number of steps.

Remember that: The number of steps to move n disks from the source to the destination peg using only 3 pegs = $2^n - 1$ (Towers of Hanoi relation). Note that moving more than 63 disks from A to D results in an overflow because of this relation.

Thus, the recurrence relation for our problem is:

$$R(n) = \min_{1 \leq k < n} [2R(k) + 2n - k - 1] \text{ for } n > 2, R(1) = 1, R(2) = 3.$$

Dynamic programming can be used to store the values calculated by this recursive function to greatly reduce the number of recursive calls, such that `reveDp[i].first` represents the minimum number of steps to move i disks from A to D, and `reveDp[i].second` represents the number of disks to be moved using the 4 pegs (k).

After obtaining the minimum number of steps and the k value, messages are printed recursively to show the exact steps of moving the disks. Note that printing the messages cannot be achieved in any other way than recursively as each message specifies a certain disk number to be moved, a certain source peg, and a certain destination which is unique for this case and cannot be known using any previously stored values.

Pseudo-code

```
int reveDP[n];
/*The dp array used to store the minimum number of steps to move the disks from
source to destination (s)
*and the number of disks to be moved using all 4 pegs (k)
*/
ALGORITHM getRevePair(n)
//INPUT: n - number of disks to be moved.
//OUTPUT: minimum number of steps to move the disks from source to destination (s)
* and the number of disks to be moved using all 4 pegs (k).
*/
int s < -infinity;
int k < --1; //a flag to indicate a base case
if n = 0
return s < -0 and k < --1;
if n = 1
return s < -1 and k < --1;
if n = 2
return s < -3 and k < --1;
for i = 1 to n - 1 do
int temp < -(2 * getRevePair(i).first + pow(2, n - i) - 1);
if temp < s
    s = temp;
k = i;
return s and k;

ALGORITHM hanoiTransfer(n, from, to, aux, offset)
/*INPUT: n - number of disks to be moved using 3 pegs.
* from: the name of the source peg.
* to: the name of the destination peg.
* aux: the name of the third peg
* offset: an offset to be added to the number of disk to be moved
    as some of the disks are already moved using the four pegs before calling this
    algorithm
*/
//OUTPUT: prints the steps of moving the disks from the source to the destination
using three pegs.
if n = 0
return;
// move all the disks above this disk from the source to the aux peg
hanoiTransfer(n - 1, from, aux, to, offset);
print the message of moving disk n + offset from the source to the destination;
// move the previously moved disks from the aux to the destination peg
hanoiTransfer(n - 1, aux, to, from, offset);
```

```

Algorithm reveTransfer(n, from, to, aux1, aux2)
/*INPUT: n - number of disks to be moved using 4 pegs.
* from: the name of the source peg.
* to: the name of the destination peg.
* aux1: the name of the third peg
* aux2: the name of the fourth peg
*/
//OUTPUT: prints the steps of moving the disks from the source to the destination
using four pegs.
if n = 0
return;
if n = 1
move the only disk from source to destination.
if n = 2
move the top disk from source to aux
move the bottom disk from source to destination
move the first disk from aux to destination
//transition:
// transfer k disks from the source to aux2
reveTransfer(reveDp[n]->k, from, aux2, aux1, to);
//transfer the remaining disks from source to destination using all pegs except aux2
hanoiTransfer(n - reveDp[n]->k, from, to, aux1, reveDp[n].second);
//move the previously transferred disks from aux2 to the destination

```

For simplicity, the algorithm is divided into three sub algorithms that interact with each other.

Complexity Analysis

- Time complexity for hanoiTransfer = $O(2^n)$, since the number of recursion calls per function call = 2, the number of disks = n, and is decreased by 1 at every recursive level.
- Time complexity for getRevePair = $O\left(\frac{n*(n+1)}{2}\right) \approx O(n^2)$, since getRevePair (n) calls getRevePair (1) : getRevePair (n - 1) and so on, and each one of these calls gets evaluated in $O(1)$ using dynamic programming.

Comparison With Another Algorithm

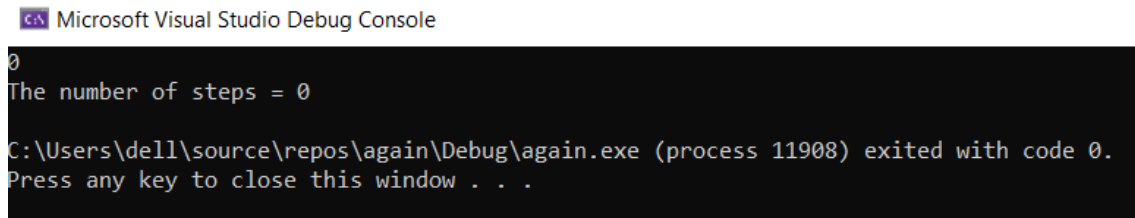
An alternative solution to moving the disks from the source to the destination pegs is using the exact same approach as the Towers of Hanoi with slight modifications

- Using two auxiliary pegs instead of one.
- Instead of moving all the disks above the bottom one (with number n) to the auxiliary peg, then moving the bottom disk to the destination, and finally moving the previously transferred disks from the auxiliary peg to the destination, we use a different approach. All of the disks above the disk number n - 1 (the one above the bottom disk) are moved to one of the auxiliary pegs, then the two bottom disks are moved from source to destination with the help of the second auxiliary peg, and finally the previously transferred disks are moved again from the first auxiliary peg to the destination.

This algorithm uses pure recursion. Thus, it is less efficient than the dp solution, it also does not guarantee the minimum number of steps to transfer the disks from source to destination, but it is much more straight forward to implement.

Sample Output

- Output in case of zero disks (base case).

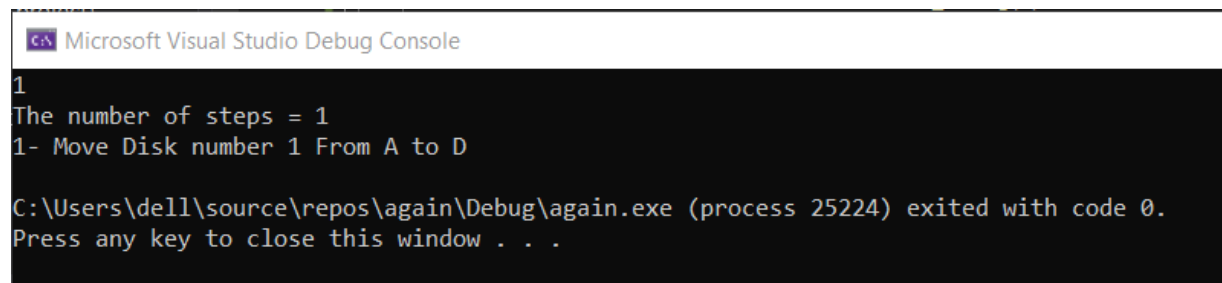


```
0
The number of steps = 0

C:\Users\dell\source\repos\again\Debug\again.exe (process 11908) exited with code 0.
Press any key to close this window . . .
```

Figure 23

- Output in case of one disk (base case).

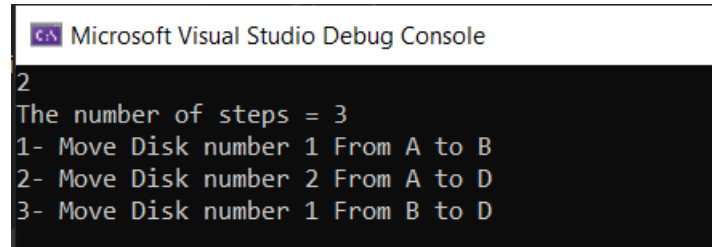


```
1
The number of steps = 1
1- Move Disk number 1 From A to D

C:\Users\dell\source\repos\again\Debug\again.exe (process 25224) exited with code 0.
Press any key to close this window . . .
```

Figure 24

- Output in case of 2 disks (base case).



```
Microsoft Visual Studio Debug Console
2
The number of steps = 3
1- Move Disk number 1 From A to B
2- Move Disk number 2 From A to D
3- Move Disk number 1 From B to D
```

Figure 25

- Output in case of 5 disks (randomly selected number).



```
Microsoft Visual Studio Debug Console
5
The number of steps = 13
1- Move Disk number 1 From A to B
2- Move Disk number 2 From A to C
3- Move Disk number 1 From B to C
4- Move Disk number 3 From A to D
5- Move Disk number 4 From A to B
6- Move Disk number 3 From D to B
7- Move Disk number 5 From A to D
8- Move Disk number 3 From B to A
9- Move Disk number 4 From B to D
10- Move Disk number 3 From A to D
11- Move Disk number 1 From C to B
12- Move Disk number 2 From C to D
13- Move Disk number 1 From B to D

C:\Users\dell\source\repos\again\Debug\again.exe (process 6204) exited with code 0.
Press any key to close this window . . .
```

Figure 26

- Output in case of 8 disks (the description given number).

```

Microsoft Visual Studio Debug Console
8
The number of steps = 33
1- Move Disk number 1 From A to D
2- Move Disk number 2 From A to C
3- Move Disk number 3 From A to B
4- Move Disk number 2 From C to B
5- Move Disk number 4 From A to C
6- Move Disk number 2 From B to A
7- Move Disk number 3 From B to C
8- Move Disk number 2 From A to C
9- Move Disk number 1 From D to C
10- Move Disk number 5 From A to B
11- Move Disk number 6 From A to D
12- Move Disk number 5 From B to D
13- Move Disk number 7 From A to B
14- Move Disk number 5 From D to A
15- Move Disk number 6 From D to B
16- Move Disk number 5 From A to B
17- Move Disk number 8 From A to D
18- Move Disk number 5 From B to D
19- Move Disk number 6 From B to A
20- Move Disk number 5 From D to A
21- Move Disk number 7 From B to D
22- Move Disk number 5 From A to B
23- Move Disk number 6 From A to D
24- Move Disk number 5 From B to D
25- Move Disk number 1 From C to A
26- Move Disk number 2 From C to D
27- Move Disk number 3 From C to B
28- Move Disk number 2 From D to B

```

Figure 27

```

29- Move Disk number 4 From C to D
30- Move Disk number 2 From B to C
31- Move Disk number 3 From B to D
32- Move Disk number 2 From C to D
33- Move Disk number 1 From A to D

C:\Users\dell\source\repos\again\Debug\again.exe (process 23696) exited with code 0.
Press any key to close this window . . .

```

Figure 28

Conclusion

- The Reve puzzle can be solved using dynamic programming recursively by moving a subset of disks to an auxiliary peg using 4 pegs. Then, moving the remaining disks to the destination using the Towers of Hanoi algorithm. Finally, retransferring the previously moved pegs from the auxiliary peg the destination.
- The puzzle can be solved using pure recursion with easier implementation, but the optimum solution and the dp algorithm efficiency are to be sacrificed.

References

1. Anany Levitin, M. L. (2011). Algorithmic Puzzles. Oxford University Press.