**ARTIFICIAL INTELLIGENCE**

**Prepared by : GROUP-1**

**Section : K18XC**

**Roll no : 01,02,03,04**

**Group mates:**

1. **Hritik Sinha**
2. **Shalin Karavadia**
3. **Shaik Altaf**
4. **Mohammed Jamal.**

**PART-I: Tower of Hanoi**

**INTRODUCTION**

The tower of Hanoi (also called the tower of Brahma or the Lucas tower) was invented by a French mathematician Édouard Lucas in the 19th century. It is associated with a legend of a Hindu temple where the puzzle was supposedly used to increase the mental discipline of young priests. In the legend the young priests were given 64 gold disks stacked neatly on one of three posts. Each disk rested on a slightly larger disk. The priests' goal was to re-create the stack on a different post by moving disks, one at a time, to another post with the rule that a larger disk could never be placed on top of a smaller disk. Using mathematics, you can calculate that even when the priests found the most efficient way to solve the problem, and moved the disks at a rate of one per second, it would take almost 585 billion years to finish the job. That is more than 40 times the age of the universe!

You might wonder how mathematics is involved in playing this game. As you play the game with more and more disks, you will notice you start to look for patterns. If you try to explain how you solve the puzzle, you might realize you use one of the following mathematical concepts:  
—Iterative solutions, where the same sequence of instructions is repeated over and over  
—Recursive solutions, where you use information from one step to find the next step  
—Patterns and translating these in mathematical formulas.  
These terms might seem hard to understand. Don’t worry! Play the game, and you might realize that you develop these “solutions” on your own.

**OBJECTIVE & METHODOLOGY**

Start the game with your two smallest disks. Stack them on the leftmost square of your cardboard, with the smaller disk on top of the larger disk.

The starting position of the game is a tower on the leftmost square of the board (like the two-disk tower you have now). The goal of the game is to move the tower to the rightmost square of the board while following these rules:

You can only move one disk at a time.You can only move the topmost disk from any square.Disks can be placed on an empty square on the board or on a larger disk, but disks cannot be placed next to each other on top of a larger disk.

* Looking at the rules, can you see it is not allowed to place a disk off the board? Where does it state you cannot start two towers on one square?
* Try the game with your two-disk tower. Can you move the tower to the rightmost square, following the rules above? How many moves did you need? If you get confused or cannot finish the game, ask a friend to help you.
* Replay the game with the two-disk tower. Do you think you can find the “best” solution, with the smallest possible number of moves?
* Once you master the two-disk tower, try with three disks. Stack three disks (from largest to smallest) on the leftmost square and start over. How many moves did you need this time?
* Try the game again with three disks. Can you finish the game with fewer moves? Do you think you can finish the game with the smallest possible number of moves?
* Now try with four disks. Can you find a way to do it in as few moves as possible? Does anything you learned while solving the three-disk tower help you solve the four-disk one?
* Play the game a few more times and observe closely. Do you have a strategy? Could you explain to a friend how you finish the game?
* Challenge yourself and try the five-disk tower. If you had a strategy, can you extend it to the five-disk tower?
* If you did not have a strategy yet, do not give up! See if you can find a pattern now. Can you find the smallest number of moves needed to finish the five-disk game? How many more moves would you need compared with the four-disk one? How many more moves are needed to move the four-disk tower compared with the three-disk one?

**RESULTS AND CONCLUSION**

Were you able to move the two-disk stack in three moves? Three is the minimal number of moves needed to move this tower. Maybe you also found in the games three-disks can be finished in seven moves, four-disks in 15 and five-disks in 31.

To play the two-disk game with only three moves, place the small disk on the middle square, the bigger one on the rightmost square and finish by placing the smaller disk on the bigger one. A way to play the three-disk game is to move the topmost two-disk stack to the middle square. This is a slight variation of the two-disk game and can be done in three moves. Then move the biggest disk from the leftmost to the rightmost square (one move). Next move the two-disk stack that is now in middle square on top of the biggest disk. Again a slight variation of the two-disk game, which can be done with three moves. This yields 3 + 1 + 3, or seven moves. A similar approach helps play the four-disk game. First move the topmost three-disk stack to the middle square using seven moves. Then move the biggest disk from the leftmost to the rightmost square (one move). Now move the three-disk stack that is in middle square on top of the biggest disk (seven moves). This yields exactly 7 + 1 + 7, or 15 moves. Mathematicians call this is a recursive procedure. To solve the puzzle with more disks, you start out with the solution for a smaller number of disks.

There are other ways to play the game. There is a pattern used to start the game when played with a minimal number of moves: The smallest disk goes to the rightmost square when playing with an even number of disks and to the middle square when playing with an odd number of disks. Maybe you also noticed you alternate between moving the smallest and the next-smallest movable disk while playing the game. If you pay close attention, you might note that you repeat the same set of instructions over and over until the tower appears on the other side of your board. Mathematicians call this is an iterative solution.

**Code Snippet:**

**import random, pdb, time**

**from Tkinter import \***

**global time\_of\_sleep;**

**time\_of\_sleep = 1; # time in seconds for next Animation... ;)**

**'''time in seconds for next Animation can be given in global variable "time\_of\_sleep";)'''**

**class visual\_tower:**

**def \_\_init\_\_(self,n,p0,p1,p2,p3):**

**self.myGui=Tk();**

**self.myGui.title("Tower Of Hanoi - By Group1");**

**self.p=[];**

**self.p.append(p0);**

**self.p.append(p1);**

**self.p.append(p2);**

**self.p.append(p3);**

**self.Frames=[];**

**#self.\_n = self.p[0].\_size+self.p[1].\_size + self.p[2].\_size + self.p[3].\_size;**

**self.\_n=n;**

**for i in range(4):**

**self.Frames.append(Frame(self.myGui,height=self.\_n\*100,width=(self.\_n+1)\*25));**

**self.Frames[i].grid(row=0,column=i);**

**self.showit();**

**self.myGui.update();**

**self.myGui.quit();**

**def showit(self):**

**w=range(self.\_n);**

**for f in range(4):**

**w\_tmp = Canvas(self.Frames[f], width=(self.\_n)\*25, height=26);**

**w\_tmp.grid(row=self.\_n,column=0,columnspan=(self.\_n+1));**

**w\_tmp.create\_rectangle(0,0, self.\_n\*25 ,10,fill="blue");**

**ff=[0 for i in range(self.\_n)];**

**for i in range(0,self.p[f].\_size):**

**w[i] = Canvas(self.Frames[f], width=(self.p[f].stk[i])\*25, height=26);**

**w[i].grid(row=self.\_n-i-1,column=0,columnspan=(self.\_n-self.p[f].stk[i]+1));**

**w[i].create\_rectangle(0,0, self.p[f].stk[i]\*25 ,25,fill="black");**

**for i in range(self.p[f].\_size,self.\_n):**

**w[i] = Canvas(self.Frames[f], width=(self.\_n)\*25, height=26);**

**w[i].grid(row=self.\_n-i-1,column=0,columnspan=(self.\_n+1));**

**w[i].create\_rectangle(0,0, self.\_n\*25 ,25,fill="white");**

**self.myGui.update();**

**if(self.p[3].\_size == self.\_n and self.p[0].\_size == 0 ):**

**self.myGui.mainloop();**

**class stack:**

**def \_\_init\_\_(self,size,name):**

**self.name=name;**

**self.\_size=size;**

**self.stk=[];**

**for i in range(self.\_size,0,-1):**

**self.stk.append(i);**

**def pop(self):**

**if self.\_size > 0:**

**\_tmp=self.stk[ self.\_size-1 ]; #element to be popped**

**self.stk=self.stk[:self.\_size-1];**

**self.\_size=self.\_size-1;**

**return \_tmp;**

**else:**

**return -1;**

**def push(self,elem):**

**if(self.\_size==0 or elem < self.stk[ self.\_size-1 ]):**

**self.stk.append(elem);**

**self.\_size=self.\_size+1;**

**else :**

**print "This Operation is Invalid in Tower of Hanoi ;)"**

**def printit(self):**

**for i in range(self.\_size-1,-1,-1):**

**print(self.stk[i]);**

**class hanoi:**

**def \_\_init\_\_(self,n):**

**self.n=n;**

**self.p1=stack(self.n,name="peg1");**

**self.p2=stack(self.n,name="peg2");**

**self.p3=stack(self.n,name="peg3");**

**self.p4=stack(self.n,name="peg4");**

**self.vt=visual\_tower(self.n,self.p1,self.p2,self.p3,self.p4);**

**while(self.p2.pop() !=-1):**

**pass;**

**while(self.p3.pop() !=-1):**

**pass;**

**while(self.p4.pop() !=-1):**

**pass;**

**self.displayit();**

**def displayit(self):**

**self.vt.showit();**

**time.sleep(time\_of\_sleep);**

**def tower\_of\_hanoi(num\_of\_Disks, src, inter1, inter2, dest):**

**#pdb.set\_trace();**

**global H,time\_of\_sleep;**

**time\_of\_sleep=1; # change it as per your convenience**

**if(num\_of\_Disks %2 == 1 and num\_of\_Disks < 2):**

**print "\nTransfer Disc from\n",src.name," >> TO >> ",dest.name;**

**dest.push(src.pop());**

**H.displayit();**

**elif(num\_of\_Disks % 2==0 and num\_of\_Disks < 3):**

**print "\nTransfer Disc from\n",src.name," >> TO >> ",inter1.name;**

**inter1.push(src.pop());**

**H.displayit();**

**print "\nTransfer Disc from\n",src.name," >> TO >> ",dest.name;**

**dest.push(src.pop());**

**H.displayit();**

**print "\nTransfer Disc from\n",inter1.name," >> TO >> ", dest.name;**

**dest.push(inter1.pop());**

**H.displayit();**

**else:**

**tower\_of\_hanoi(num\_of\_Disks-2,src,inter2,dest,inter1);**

**print "\nTransfer Disc from\n",src.name, " >> TO >> ", inter2.name;**

**inter2.push(src.pop());**

**H.displayit();**

**print "\nTransfer Disc from\n",src.name, " >> TO >> ", dest.name;**

**dest.push(src.pop());**

**H.displayit();**

**print "\nTransfer Disc from\n",inter2.name, " >> TO >> ", dest.name;**

**dest.push(inter2.pop());**

**H.displayit();**

**tower\_of\_hanoi(num\_of\_Disks-2,inter1,src,inter2,dest);**

**def main():**

**n=input("No. Of Disks : "); # no. of disks**

**global H;**

**H = hanoi(n);**

**tower\_of\_hanoi(n,H.p1,H.p2,H.p3,H.p4);**

**def TOI(N):**

**global H;**

**H = hanoi(N); # creating object of hanoi class :)**

**tower\_of\_hanoi(N,H.p1,H.p2,H.p3,H.p4);**

**if \_\_name\_\_=="\_\_main\_\_":main();**

**PART-II: Missionaries & Cannibals**

**INTRODUCTION**

In this problem, three missionaries and three cannibals must cross a river using a boat which can carry at most two people, under the constraint that, for both banks, that the missionaries present on the bank cannot be outnumbered by cannibals. The boat cannot cross the river by itself with no people on board.

**SOLUTION & METHODOLOGY**

First let us consider that both the missionaries (M) and cannibals(C) are on the same side of the river.  
Initially the positions are: 0M , 0C and 3M , 3C (B)

Now let’s send 2 Cannibals to left of bank: 0M , 2C (B) and 3M , 1C

Send one cannibal from left to right : 0M , 1C and 3M , 2C (B)

Now send the 2 remaining Cannibals to left : 0M , 3C (B) and 3M , 0C  
Send 1 cannibal to the right : 0M , 2C and 3M , 1C (B)

Now send 2 missionaries to the left : 2M , 2C (B) and 1M . 1C

Send 1 missionary and 1 cannibal to right : 1M , 1C and 2M , 2C (B)

Send 2 missionaries to left : 3M , 1C (B) and 0M , 2C

Send 1 cannibal to right : 3M , 0C and 0M , 3C (B)

Send 2 cannibals to left : 3M , 2C (B) and 0M , 1C

Send 1 cannibal to right : 3M , 1C and 0M , 2C (B)’

Send 2 cannibals to left: 3M , 3C (B) and 0M , 0C

**CODE SNIPPET**

**This is a solution to the Missionaries and Cannibals problem. Uses breadth first search**

**from copy import deepcopy**

**from collections import deque**

**import sys**

**import time**

**class State(object):**

**def \_\_init\_\_(self, missionaries, cannibals, boats):**

**self.missionaries = missionaries**

**self.cannibals = cannibals**

**self.boats = boats**

**def successors(self):**

**if self.boats == 1:**

**sgn = -1**

**direction = "from the original shore to the new shore"**

**else:**

**sgn = 1**

**direction = "back from the new shore to the original shore"**

**for m in range(3):**

**for c in range(3):**

**newState = State(self.missionaries+sgn\*m, self.cannibals+sgn\*c, self.boats+sgn\*1);**

**if m+c >= 1 and m+c <= 2 and newState.isValid(): # check whether action and resulting state are valid**

**action = "take %d missionaries and %d cannibals %s. %r" % ( m, c, direction, newState)**

**yield action, newState**

**def isValid(self):**

**if self.missionaries < 0 or self.cannibals < 0 or self.missionaries > 3 or self.cannibals > 3 or (self.boats != 0 and self.boats != 1):**

**return False**

**# then check whether missionaries outnumbered by cannibals**

**if self.cannibals > self.missionaries and self.missionaries > 0: # more cannibals then missionaries on original shore**

**return False**

**if self.cannibals < self.missionaries and self.missionaries < 3: # more cannibals then missionaries on other shore**

**return False**

**return True**

**def is\_goal\_state(self):**

**return self.cannibals == 0 and self.missionaries == 0 and self.boats == 0**

**def \_\_repr\_\_(self):**

**return "< State (%d, %d, %d) >" % (self.missionaries, self.cannibals, self.boats)**

**class Node(object):**

**def \_\_init\_\_(self, parent\_node, state, action, depth):**

**self.parent\_node = parent\_node**

**self.state = state**

**self.action = action**

**self.depth = depth**

**def expand(self):**

**for (action, succ\_state) in self.state.successors():**

**succ\_node = Node(**

**parent\_node=self,**

**state=succ\_state,**

**action=action,**

**depth=self.depth + 1)**

**yield succ\_node**

**def extract\_solution(self):**

**solution = []**

**node = self**

**while node.parent\_node is not None:**

**solution.append(node.action)**

**node = node.parent\_node**

**solution.reverse()**

**return solution**

**def breadth\_first\_tree\_search(initial\_state):**

**initial\_node = Node(**

**parent\_node=None,**

**state=initial\_state,**

**action=None,**

**depth=0)**

**fifo = deque([initial\_node])**

**num\_expansions = 0**

**max\_depth = -1**

**while True:**

**if not fifo:**

**print "%d expansions" % num\_expansions**

**return None**

**node = fifo.popleft()**

**if node.depth > max\_depth:**

**max\_depth = node.depth**

**print "[depth = %d] %.2fs" % (max\_depth, time.clock())**

**if node.state.is\_goal\_state():**

**print "%d expansions" % num\_expansions**

**solution = node.extract\_solution()**

**return solution**

**num\_expansions += 1**

**fifo.extend(node.expand())**

**def usage():**

**print >> sys.stderr, "usage:"**

**print >> sys.stderr, " %s" % sys.argv[0]**

**raise SystemExit(2)**

**def main():**

**initial\_state = State(3,3,1)**

**solution = breadth\_first\_tree\_search(initial\_state)**

**if solution is None:**

**print "no solution"**

**else:**

**print "solution (%d steps):" % len(solution)**

**for step in solution:**

**print "%s" % step**

**print "elapsed time: %.2fs" % time.clock()**

**if \_\_name\_\_ == "\_\_main\_\_":**

**main();**