tower\_of\_hanoi.py (all 5 steps done)

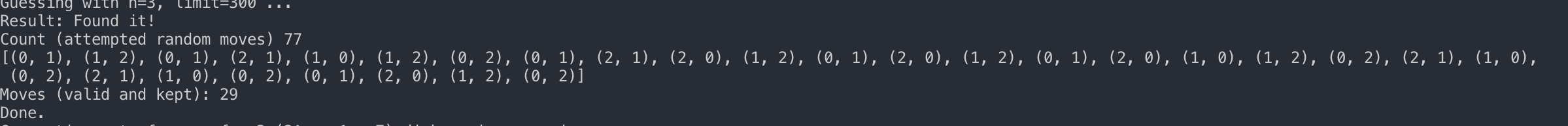
Tower of Hanoi is a puzzle created on Python. At step 2, a hard coding was done to path out the moves needed in order to complete the sequence. For step 3, the limit of 100 was not enough so it was changed to 300. The random search got upgraded in step 4. It reduced the length of the sequence, plus the chance of correct random guesses was also increased. Step 5 implements recursion, with the ‘move’ function called itself.

Before

A close up of a keyboard

Description automatically generated

After



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

*### 1 Simple test section.*

if True:

*# setup*

n = 7

s1 = init\_poles(n)

*# try each type of print*

print\_poles(s1)

print\_poles\_as\_state(s1)

print\_poles\_as\_text(s1, n)

*# test for valid first state*

print(is\_valid\_state(s1))

*# move first disk*

s2 = move\_disk(s1, 0, 2)

print\_poles\_as\_text(s2, n)

print(is\_valid\_state(s2))

*# do an invalid move*

s3 = move\_disk(s2, 0, 2)

print\_poles\_as\_text(s3, n)

print(is\_valid\_state(s3))

*# do a valid second move*

s3 = move\_disk(s2, 0, 1)

print\_poles\_as\_text(s3, n)

print(is\_valid\_state(s3))

*### 2 Perform a sequence of moves*

if True:

*# do a sequence of moves*

print("Running coded sequence ...")

n = 3

s = init\_poles(n)

print\_poles\_as\_text(s, n)

*### 3. Complete the sequence of moves to solve the game*

moves = [(0, 2), (0, 1), (2, 1), (0, 2), (1, 0), (1, 2), (0, 2)]

for src, dest in moves:

print('> Moving from', src, 'to', dest)

s = move\_disk(s, src, dest)

print\_poles\_as\_text(s, n)

*#print(is\_valid\_state(s))*

print\_poles\_as\_state(s, test\_valid=True)

print('Done.')

*### 4 Try to find the solution using random (but valid) guesses for each move*

if True:

attempt\_using\_random\_moves(n=3, limit=300)

*### 5 Generate the recursive sequence of moves*

if True:

solve\_using\_recursion(n=3)

**OUTPUT**

A close up of a logo

Description automatically generated

A close up of a logo

Description automatically generated

A screenshot of a cell phone

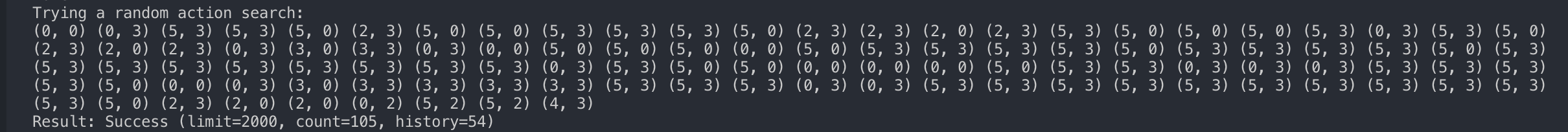
Description automatically generated

water\_jug\_problem.py (all 5 steps done)

The water jug is a puzzle created on python. For step 6, asserts were fixed so that the moves worked successfully. In step 7, sequence 1 & 2 were solved successfully. At step 8, the limit was set to 4000. At step 9, random search got an upgrade in comparison to step 8, which gave out more frequent successful results. The solution path got smaller, and the guess count decreased.

A screenshot of a cell phone

Description automatically generated



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

*### 1 Basic testing of methods and operations*

if True:

JUG\_CFG = [5,3] *# (Die Hard movie version)*

s = setup\_jugs()

print(s)

*# test fillling*

s = fill(s, 0)

print(s)

assert s == (5, 0)

s = pour(s, 0, 1)

print(s)

assert s == (2, 3)

print(s)

*# test emtpy*

s = empty(s, 1)

print(s)

assert s == (2, 0)

*# test pour / leftover actions*

s = pour(s, 0, 1)

assert s == (0, 2)

print (s)

s = fill(s, 0)

print(s)

assert s == (5, 2)

s = pour(s, 0, 1)

print(s)

assert s == (4, 3)

s = empty(s, 1)

print(s)

assert s == (4, 0)

*### 2 Solve using a pre-defined sequence of actions*

if True:

action\_calls = {

'fill': fill,

'empty': empty,

'pour': pour

}

print('Doing sequence 1 ...')

*### 3 Sequence 1 of moves*

actions = [

*# tuples, string of method to call then arguments to call*

('fill', (0,)),

('pour', (0, 1)), *# (5,0) poor 1 into 2*

('empty', (1,)), *# (2,3) empty 2*

('pour', (0, 1)), *# (2,0) tranfer from 1 to 2*

('fill', (0,)), *# (0,2) fill jug 1*

('pour', (0, 1)),

('empty', (1,)),

*# TODO: missing move - see header for sequence.*

*# result should be (4,0)*

]

*# execute the sequence of actions*

JUG\_CFG = [5,3] *# (Die Hard movie version)*

s = setup\_jugs()

for fn, args in actions:

*#print('Calling...', fn, 'with', args, 'on', s)*

s = action\_calls[fn](s, \*args)

print(s)

print('Done')

*### 4 Solve using sequence 2*

if True:

action\_calls = {

'fill': fill,

'empty': empty,

'pour': pour

}

print('Doing sequence 2 ...')

actions = [

*# tuples, string of method to call then arguments to call*

('fill', [1]), *# fill jug 2 => (0,3)*

('pour', [1, 0]), *# transfer 2 to jug 1 => (3,0)*

('fill', [1]),

('pour', [1, 0]),

('empty', [0]),

('pour', [1, 0]),

('fill', [1]),

('pour', [1, 0]),

*###TODO: complete the sequence*

*# result should be (4,0)*

]

*# run sequence of actions*

JUG\_CFG = [5,3] *# (Die Hard movie version)*

s = setup\_jugs()

for fn, args in actions:

*#print('Calling...', fn, 'with', args, 'on', s)*

s = action\_calls[fn](s, \*args)

print(s)

print('Done')

*# Random choice from all possible actions for a fixed problem*

if True:

*# There is a set of six unique actions to choose from*

actions = [

*# all possible fill's*

(fill, [0]),

(fill, [1]),

*# all possible pour's*

(pour, [0, 1]),

(pour, [1, 0]),

*# all possble empty's*

(empty, [0]),

(empty, [1]),

]

*# Notes:*

*# - We exclude pour 0->0 and 1->1 as they pointless*

*# - Some actions might have no effect (empty if already empty)*

*# but we are not making conditional actions (only naive ones)*

from random import choice, seed

*#seed(1234)*

*# For the Die Hard 3 movie two-jug problem ...*

JUG\_CFG = [5, 3]

s = setup\_jugs()

s\_end = (4, 0)

*###TODO: use a list of valid end\_states, not just one*

end\_states = [(4,0), (4,1), (4,2), (4,3)]

status = 'searching'

count = 0

limit =4000

history = [] *# history of moves taken*

*# Search loop*

print('Trying a random action search:')

while status == 'searching':

*# select a random action to try*

fn, args = choice(actions)

new\_s = fn(s, \*args)

*# print(str(fn.\_\_name\_\_), args, 'on', s, '=>', new\_s) # details*

*# print('.', end='') # progress dots ...*

print(new\_s, end=' ') *# verbose*

*# if move outcome state is valid (not None) keep it*

if new\_s and new\_s != s:

s=new\_s

history.append((fn, args))

if new\_s in end\_states:

status = 'Success'

*# count and stop test*

count += 1

if count >= limit:

status = 'Hit limit'

print()

print('Result: %s (limit=%d, count=%d, history=%d)' % (status, limit, count, len(history)))