The towers of Hanoi

Eric Martin, CSE, UNSW

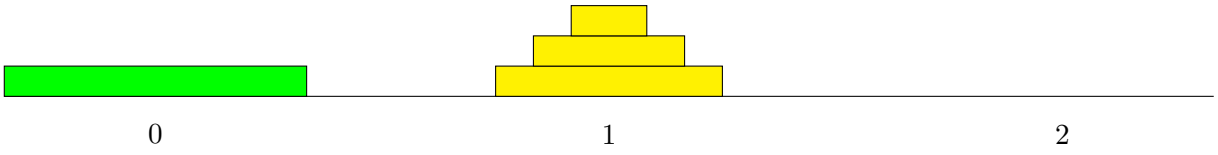
COMP9021 Principles of Programming

Let n be an integer at least equal to 1. Consider 3 positions, referred to as start position, end position and extra position, or as 0, 1 and 2. A total of n disks all of different sizes, stacked at start position, have to be displaced to end position, thanks to a succession of moves each of which consists of removing the disk at the top of a stack and bringing it to the top of another (possibly empty) stack. At any stage of the game, no disk should be above a smaller disk; in particular, at the beginning and at the end of the game, all disks are stacked from largest to smallest.

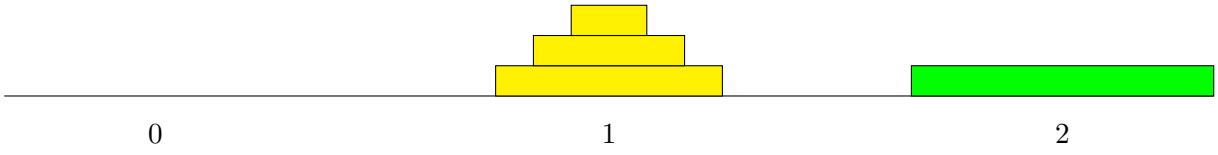
It is necessary to eventually move the largest disk from position 0 to position 2. That is possible only when the $n - 1$ smaller disks have been displaced from position 0 and stacked at position 1. Then, with the largest disk having found its final position at the base of position 2, those disks have to be displaced from position 1 and stacked at position 2. This is illustrated as follows for $n = 4$.

Initial configuration:

After the $n - 1$ smaller disks have been displaced from position 0 to position 1:



Largest disk moved from position 0 to position 2:



Final configuration, after the $n - 1$ smaller disks have been displaced from position 1 to position 2:

Formulate the problem as: displace all disks, in accordance with the rules, from some position to another position, with the help of a third position; that is, make the three positions parameters to the problem, as opposed to fixing them. Note that when moving the $n-1$ smaller disks, having a larger disk somewhere or having no larger disk anywhere makes no difference. The recursive implementation immediately follows from these observations:

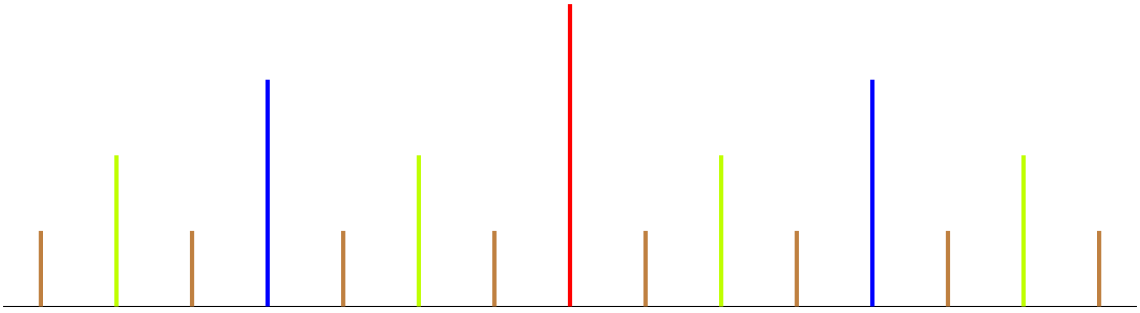
[1]:
```
def recursive_towers(n, start_pos, end_pos, extra_pos):
    if n > 1:
        recursive_towers(n - 1, start_pos, extra_pos, end_pos)
    print('Move disk of size', n, 'from position', start_pos,
          'to position', end_pos
         )
    if n > 1:
        recursive_towers(n - 1, extra_pos, end_pos, start_pos)

recursive_towers(4, 0, 2, 1)
```

```
Move disk of size 1 from position 0 to position 1
Move disk of size 2 from position 0 to position 2
Move disk of size 1 from position 1 to position 2
Move disk of size 3 from position 0 to position 1
Move disk of size 1 from position 2 to position 0
Move disk of size 2 from position 2 to position 1
Move disk of size 1 from position 0 to position 1
Move disk of size 4 from position 0 to position 2
Move disk of size 1 from position 1 to position 2
Move disk of size 2 from position 1 to position 0
Move disk of size 1 from position 2 to position 0
Move disk of size 3 from position 1 to position 2
Move disk of size 1 from position 0 to position 1
Move disk of size 2 from position 0 to position 2
Move disk of size 1 from position 1 to position 2
```

Also, these observations establish that the output of the recursive implementation is not only a solution; it is in fact the only possible solution for that optimal number of moves, equal to 2^n-1, as shown by induction on n:

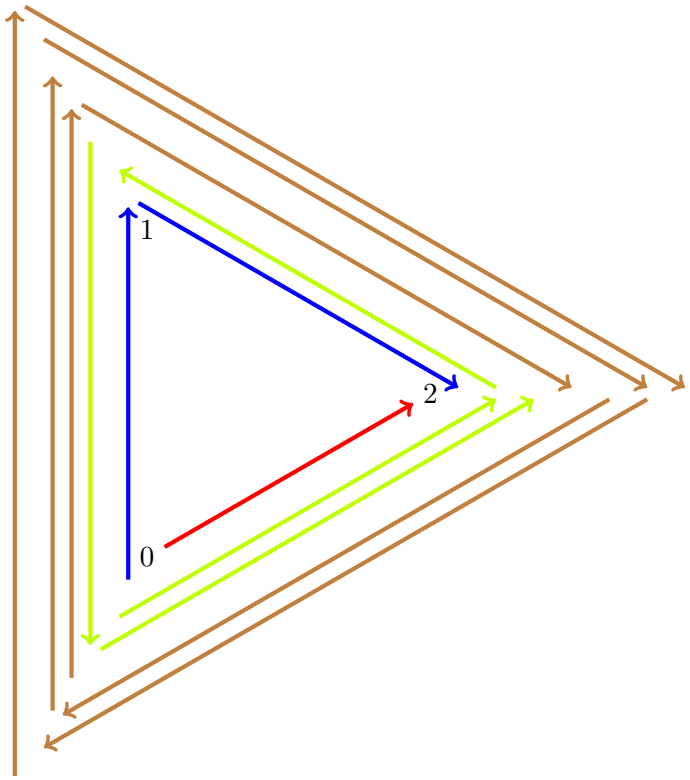
- If $n=1$, then $2^n-1=1$, and 1 move is necessary and sufficient indeed.
- If 2^n-1 moves are necessary and sufficient to move n disks, then $(2^n-1)\times 2+1=2^{n+1}-1$ moves are necessary and sufficient indeed to move $n+1$ disks.

Moving the smaller $n-1$ disks, then moving the largest disk, then moving the $n-1$ smaller disks again to move the n disks, done recursively, shows that the Hanoi of towers puzzle is equivalent to that of putting ticks of n different sizes on a ruler, as illustrated next for $n=4$:



This shows that every second move involves moving the smallest disk; the other moves are clearly imposed. So to convert the recursive implementation into an iterative implementation, it suffices to determine where to move the smallest disk every time it has to be moved.

Place the three positions, 0, 1 and 2, clockwise on a ring.



- Moving the n disks from position 0 to position 2 can be visualised as moving on the ring counterclockwise.
- Moving the $n-1$ disks from position 0 to position 1 can be visualised as moving on the ring clockwise.
- Moving the $n-1$ disks from position 1 to position 2 can be visualised as moving on the ring clockwise.

It follows that:

- the stack of n disks have to be moved from where they are to the next position counterclockwise, which if $n > 1$ involves

- twice moving the stack of the $n-1$ smaller disks from where they are to the next position clockwise, which if $n > 2$ involves
- four times moving the stack of the $n-2$ smaller disks from where they are to the next position counterclockwise, which if $n > 3$ involves
- eight times moving the stack of the $n-3$ smaller disks from where they are to the next position clockwise
- ...

Hence:

- if n is even then the smallest disk always has to be moved to the next position clockwise, that is, from 0 to 1, from 1 to 2, and from 2 to 0;
- if n is odd then the smallest disk always has to be moved to the next position counterclockwise, that is, from 0 to 2, from 2 to 1, and from 1 to 0.

With n set to 4, the smallest disk will have to move clockwise:

[2]:
```
n = 4
direction = -1 if n % 2 else 1

direction
```

[2]:
```
1
```

We can use a tuple of 3 lists, `stacks`, to represent the stacks of disks at position 0, 1 and 2, respectively. We can use the integers from 1 to n to denote the n disks, from smallest to largest. To start with, all disks are in the first stack, stacked from largest to smallest:

[3]:
```
stacks = list(range(n, 0, -1)), [], []

stacks
```

[3]:
```
([4, 3, 2, 1], [], [])
```

To start with, the smallest disk is at position 0. It has to be moved to position 1. A combination of `pop()` and `append()` realises the move:

[4]:
```
small_disk_pos = 0
new_small_disk_pos = (small_disk_pos + direction) % 3

print('Move smallest disk from position', small_disk_pos,
      'to position', new_small_disk_pos
     )
stacks[new_small_disk_pos].append(stacks[small_disk_pos].pop())

stacks
```

```
Move smallest disk from position 0 to position 1
```

[4]:
```
([4, 3, 2], [1], [])
```

When not moving the smallest disk, one has to work with the two positions different to the new position of the smallest disk:

[5]:
```
small_disk_pos = new_small_disk_pos

small_disk_pos, (small_disk_pos + 1) % 3, (small_disk_pos + 2) % 3
```

[5]:
```
(1, 2, 0)
```

To determine which of those two positions is the one where there is a nonempty stack whose disk at the top should be moved to the other position, because there is either no disk or a larger disk at the top of the stack there, we can take advantage of the way Python evaluates conjunctions and disjunctions. The conjuncts or disjuncts that make up a conjunction or disjunction are evaluated, from left to right, only until it can be concluded that the whole conjunction or disjunction is True or False. The value of the last conjunct or disjunct that has been evaluated is then returned:

[6]:
```
# None, 0, '', {}, [], () all evaluate to False,  2, [3], 4.5 all
# evaluate to True. Only when () is processed can we conclude that the
# expression is false.
None or 0 or '' or {} or [] or ()
# When 2 is processed, and not before, we can conclude that the
# expression is true.
None or 0 or '' or {} or () or 2 or [3] or 4.5 or []
# When not 0 processed, and not before, we can conclude that the
# expression is true.
None or not 1 or '' or not {()} or not 0 or [3] or 4.5 or []

# When 4.5 is processed, and not before, we can we conclude that the
# expression is true.
2 and [3] and 4.5
# When {} is processed, and not before, we can conclude that the
# expression is false.
2 and [3] and {} and [] and None and 4.5
# When not {0} is processed, and not before, we can conclude that the
# expression is false.
2 and not [] and not {0} and [] and None and 4.5
```

[6]:
```
()
```

[6]:
```
2
```

[6]:
```
True
```

[6]:
```
4.5
```

[6]:
```
{}
```

[6]:
```
False
```

We assume that `(small_disk_pos + 1) % 3` and `(small_disk_pos + 2) % 3` are the positions where from and where to a disk has to be moved now, but both positions are to be swapped in case in should be the other way around because either there is no disk at the former position, or there are disks at both positions but the disk at the top of the stack located at the latter is smaller than the disk at the top of the stack located at the former. For the second move, both positions need to be swapped indeed:

[7]:
```
from_pos, to_pos = (small_disk_pos + 1) % 3, (small_disk_pos + 2) % 3

from_pos, to_pos
stacks

if not stacks[from_pos]\
   or stacks[to_pos] and stacks[to_pos][-1] < stacks[from_pos][-1]:
    from_pos, to_pos = to_pos, from_pos

from_pos, to_pos
```

[7]:
```
(2, 0)
```

[7]:
```
([4, 3, 2], [1], [])
```

[7]:
```
(0, 2)
```

The second move is again realised by a combination of `pop()` and `append()`:

[8]:
```
stacks

print('Move disk of size', stacks[from_pos][-1],
      'from position', from_pos, 'to position', to_pos
     )
stacks[to_pos].append(stacks[from_pos].pop())

stacks
```

[8]:
```
([4, 3, 2], [1], [])
```

```
Move disk of size 2 from position 0 to position 2
```

[8]:
```
([4, 3], [1], [2])
```

It suffices to perform those operations again and again. Illustrating by doing them twice again:

[9]:
```
stacks

new_small_disk_pos = (small_disk_pos + direction) % 3
print('Move smallest disk from position', small_disk_pos,
      'to position', new_small_disk_pos
     )
```

```
stacks[new_small_disk_pos].append(stacks[small_disk_pos].pop())

stacks

small_disk_pos = new_small_disk_pos
from_pos, to_pos = (small_disk_pos + 1) % 3, (small_disk_pos + 2) % 3
if not stacks[from_pos]\
   or stacks[to_pos] and stacks[to_pos][-1] < stacks[from_pos][-1]:
    from_pos, to_pos = to_pos, from_pos
print('Move disk of size', stacks[from_pos][-1],
      'from position', from_pos, 'to position', to_pos
     )
stacks[to_pos].append(stacks[from_pos].pop())

stacks
```

[9]: ([4, 3], [1], [2])

```
Move smallest disk from position 1 to position 2
```

[9]: ([4, 3], [], [2, 1])

```
Move disk of size 3 from position 0 to position 1
```

[9]: ([4], [3], [2, 1])

[10]:
```
stacks

new_small_disk_pos = (small_disk_pos + direction) % 3
print('Move smallest disk from position', small_disk_pos,
      'to position', new_small_disk_pos
     )
stacks[new_small_disk_pos].append(stacks[small_disk_pos].pop())

stacks

small_disk_pos = new_small_disk_pos
from_pos, to_pos = (small_disk_pos + 1) % 3, (small_disk_pos + 2) % 3
if not stacks[from_pos]\
   or stacks[to_pos] and stacks[to_pos][-1] < stacks[from_pos][-1]:
    from_pos, to_pos = to_pos, from_pos
print('Move disk of size', stacks[from_pos][-1],
      'from position', from_pos, 'to position', to_pos
     )
stacks[to_pos].append(stacks[from_pos].pop())

stacks
```

[10]: ([4], [3], [2, 1])

```
Move smallest disk from position 2 to position 0
```

[10]: ([4, 1], [3], [2])

```
Move disk of size 2 from position 2 to position 1
```

[10]: ([4, 1], [3, 2], [])

Putting it all together:

[11]:
```
def iterative_towers(n, start_pos, end_pos, extra_pos):
    direction = -1 if n % 2 else 1
    stacks = list(range(n, 0, -1)), [], []
    small_disk_pos = 0
    for i in range(2 ** n - 1):
        if i % 2 == 0:
            new_small_disk_pos = (small_disk_pos + direction) % 3
            print('Move smallest disk from position', small_disk_pos,
                  'to position', new_small_disk_pos
                 )
            stacks[new_small_disk_pos].append(stacks[small_disk_pos].pop())
            small_disk_pos = new_small_disk_pos
        else:
            from_pos, to_pos =\
                      (small_disk_pos + 1) % 3, (small_disk_pos + 2) % 3
            if not stacks[from_pos] or\
               stacks[to_pos] and stacks[to_pos][-1] < stacks[from_pos][-1]:
                from_pos, to_pos = to_pos, from_pos
            print('Move disk of size', stacks[from_pos][-1],
                  'from position', from_pos, 'to position', to_pos
                 )
            stacks[to_pos].append(stacks[from_pos].pop())

iterative_towers(4, 0, 2, 1)
```

```
Move smallest disk from position 0 to position 1
Move disk of size 2 from position 0 to position 2
Move smallest disk from position 1 to position 2
Move disk of size 3 from position 0 to position 1
Move smallest disk from position 2 to position 0
Move disk of size 2 from position 2 to position 1
Move smallest disk from position 0 to position 1
Move disk of size 4 from position 0 to position 2
Move smallest disk from position 1 to position 2
Move disk of size 2 from position 1 to position 0
Move smallest disk from position 2 to position 0
```

```
Move disk of size 3 from position 1 to position 2
Move smallest disk from position 0 to position 1
Move disk of size 2 from position 0 to position 2
Move smallest disk from position 1 to position 2
```

Rather than displaying instructions on which disk to move from where to where, let us display the successive states of the three stacks starting with the initial configuration of the game, ending with the final configuration of the game, representing a disk of size n as $2n - 1$ successive hyphens. For this purpose, we could immediately adapt `iterative_towers()`, but let us rather adapt `recursive_towers()` to a similar function, `recursive_towers_variant()`, replacing the `print()` statements in `recursive_towers()` with calls to a function `make_and_display_move()`, which itself will call a function `display_towers()`. The function `make_and_display_move()` can know from `recursive_towers_variant()` from which stack the disk at the top should be popped, and to (the top of) which stack that disk should be moved at this stage of the game. Once the pop and move operations have been performed, `display_towers()` can display the three stacks. Working with 3 stacks, moving a disk from the top of one stack to the top of another stack, is what `iterative_towers()` implements, and we can borrow from that implementation. To let the stacks "survive" between successive calls to the functions, we could use global variables, but we can also make use of function parameters with lists as default values. The values of the default parameters can be manipuled thanks to the function's `__defaults__` attribute:

[12]:
```
def f(arg_1, arg_2, arg_3=[], arg_4=set(), arg_5={}):
    pass

f.__defaults__

f.__defaults__[0].extend((10, 11, 12))
f.__defaults__[1].add(20)
f.__defaults__[2][0] = 'A'
f.__defaults__

for default in f.__defaults__:
    default.clear()
f.__defaults__
```

[12]:
```
([], set(), {})
```

[12]:
```
([10, 11, 12], {20}, {0: 'A'})
```

[12]:
```
([], set(), {})
```

If we let `display_towers()` have parameters with default values that represent the three stacks, so `display_towers()` can keep track of the various states of the stacks as the game is being played, then we can let `display_move()` perform the pop and move operations on `display_towers()`'s parameters. The functions `recursive_towers_variant()`, `make_and_display_move()` and `display_towers()` can then be implemented as follows; they all take an extra argument, `w`, meant to denote the width of the largest disk, so that `display_towers()` knows how to center the disks that make up the stacks:

[13]:
```
def recursive_towers_variant(n, start_pos, end_pos, extra_pos, w):
    if n == 1:
        make_and_display_move(start_pos, end_pos, w)
    else:
        recursive_towers_variant(n - 1, start_pos, extra_pos, end_pos, w)
        make_and_display_move(start_pos, end_pos, w)
        recursive_towers_variant(n - 1, extra_pos, end_pos, start_pos, w)
```

[14]:
```
def make_and_display_move(start_pos, end_pos, w):
    display_towers.__defaults__[end_pos].append(
                                    display_towers.__defaults__[start_pos].pop()
                                                )
    display_towers(w)
```

[15]:
```
def display_towers(w, stack_1=[], stack_2=[], stack_3=[]):
    stacks = stack_1, stack_2, stack_3
    print()
    for i in range(max(len(stack) for stack in stacks) - 1, -1, -1):
        print('  '.join(
                f'{"-" * (stack[i] * 2 - 1) if i < len(stack) else "":^{w}}'
                            for stack in stacks
                        )
              )
    print()
```

Finally, we define a function, `recursive_towers_display_solution()`, meant to take as argument the number of disks, that sets `display_towers()`'s three default arguments to what they should be before the game starts (the preliminary calls to `clear()` are necessary if the function is summoned more than once), displays the initial configuration of the game, and calls `recursive_towers_variant()` to play the game and display each new configuration all the way to the end of the game:

[16]:
```
def recursive_towers_display_solution(n):
    for stack in display_towers.__defaults__:
        stack.clear()
    display_towers.__defaults__[0].extend(range(n, 0, -1))
    w = 2 * n - 1
    display_towers(w)
    recursive_towers_variant(n, 0, 2, 1, w)
```

[17]:
```
recursive_towers_display_solution(4)
```

```
   -
  ---
 -----
-------
```