The towers of Hanoi

Eric Martin, CSE, UNSW

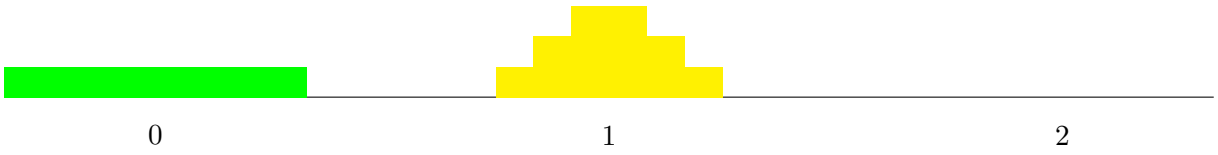
COMP9021 Principles of Programming

Let n be an integer at least equal to 1. A total of n disks all of different sizes, stacked at position 0, have to be moved to position 2, with position 1 available, 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 of the game, all disks are stacked from largest to smallest.

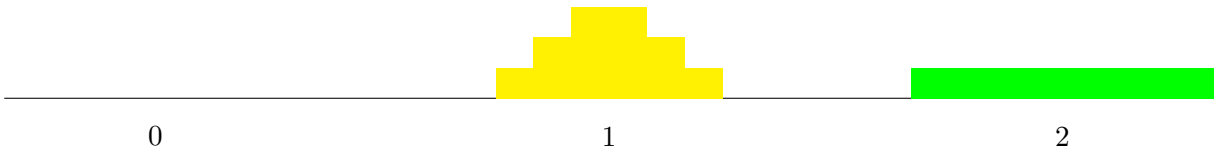
It is necessary to eventually move the largest disk from position 0 to position 2. That is possible only when the $n - 1$ smaller disks have been moved away from position 0 and stacked at position 1. Then, with the largest disk having found its final position at the base of position 2, those disks have to be moved away 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 moved 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 moved from position 1 to position 2:

The recursive implementation immediately follows from that observation:

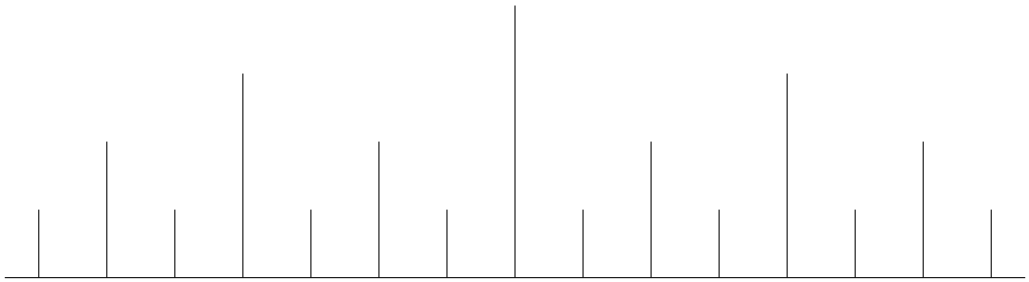
[1]:
```
def recursive_towers(n, start_pos, end_pos, extra_pos):
    if n == 1:
        print('Move smallest disk from position', start_pos, 'to position',
              end_pos
             )
    else:
        recursive_towers(n - 1, start_pos, extra_pos, end_pos)
        print('Move disk of size',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

recursive_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
```

Also, that observation establishes 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:

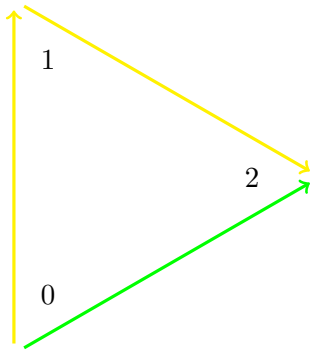
- If $n = 1$, then $2^n - 1 = 1$, and 1 move is necessary and sufficient indeed.
- If $2^n - 1$ moves are necessary and sufficient to move n disks, then $(2^n - 1) \times 2 + 1 = 2^{n+1} - 1$ moves are necessary and sufficient indeed to move $n + 1$ disks.

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, 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.

Note that the expression `1 - n % 2 * 2` evalues to 1 if `n` is even, and to -1 if `n` is odd:

[2]:
```
n = 2; 1 - n % 2 * 2
n = 3; 1 - n % 2 * 2
```

[2]:
```
1
```

[2]:
```
-1
```

So `1 - n % 2 * 2` just needs to be added to the currrent position of the smallest disk to, modulo 3, give the next position of the smallest disk.

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 between 1 and n to denote the n disks, from smallest to largest. To start with, all disks are on the first stack, stacked up from largest to smallest:

[3]:
```
n = 4
stacks = list(range(n, 0, -1)), [], []

stacks
```

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

The smallest disk is at position 0. With n set to 4, it will have to move clockwise, so to position 1. A combination of `pop()` and `append()` realises the move:

[4]:
```
small_disk_pos = 0
direction = 1 - n % 2 * 2
new_small_disk_pos = (small_disk_pos + direction) % 3

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

stacks
```

```
Move smallest disk 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
```

[5]: 2

[5]: 0

If there is an empty stack at one of these positions, then that position should be the destination position for the next move. Otherwise, if the stacks at both positions are not empty, then the position whose stack has the largest disk at the top should be the destination position. So to determine the destination position, one can sort both positions using a key which can be $n+1$ for a position where the stack is empty, and for a position where the stack S is not empty, the integer between 1 and n that represents the size of the disk at the top of S; that way, the position sorted second (and last) is guaranteed to be the destination position. The second move is again realised by a combination of `pop()` and `append()`:

[6]:
```
stacks

from_pos, to_pos = sorted(((small_disk_pos + 1) % 3, (small_disk_pos + 2) % 3),
                          key=lambda x: not stacks[x] and n + 1
                                        or stacks[x][-1]
                         )
print('Move disk from position', from_pos, 'to position', to_pos)
stacks[to_pos].append(stacks[from_pos].pop())

stacks
```

[6]: ([4, 3, 2], [1], [])

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

[6]: ([4, 3], [1], [2])

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

[7]:
```
stacks

new_small_disk_pos = (small_disk_pos + direction) % 3
print('Move smallest disk 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 = sorted(((small_disk_pos + 1) % 3, (small_disk_pos + 2) % 3),
                          key=lambda x: not stacks[x] and n + 1
                                        or stacks[x][-1]
                         )
print('Move disk from position', from_pos, 'to position', to_pos)
stacks[to_pos].append(stacks[from_pos].pop())
```

```
stacks
```

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

```
Move smallest disk to position 2
```

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

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

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

[8]:
```
stacks

new_small_disk_pos = (small_disk_pos + direction) % 3
print('Move smallest disk 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 = sorted(((small_disk_pos + 1) % 3, (small_disk_pos + 2) % 3),
                          key=lambda x: not stacks[x] and n + 1
                                        or stacks[x][-1]
                         )
print('Move disk from position', from_pos, 'to position', to_pos)
stacks[to_pos].append(stacks[from_pos].pop())

stacks
```

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

```
Move smallest disk to position 0
```

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

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

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

Putting it all together:

[9]:
```
def iterative_towers(n, start_pos, end_pos, extra_pos):
    small_disk_pos = 0
    direction = 1 - n % 2 * 2
    stacks = list(range(n, 0, -1)), [], []
    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 = sorted(((small_disk_pos + 1) % 3,
                                       (small_disk_pos + 2) % 3
                                      ), key=lambda x: not stacks[x] and n + 1
                                                       or stacks[x][-1]
                                     )
            stacks[to_pos].append(stacks[from_pos].pop())
            print('Move disk of size', stacks[to_pos][-1], 'from position',
                  from_pos, 'to position', to_pos
                 )

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 `display_move()`, which itself will call a function `display_towers()`. The function `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:

[10]:
```
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__
```

[10]:
```
([], set(), {})
```

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

[10]:
```
([], 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()`, `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:

[11]:
```
def recursive_towers_variant(n, start_pos, end_pos, extra_pos, w):
    if n == 1:
        display_move(start_pos, end_pos, w)
    else:
        recursive_towers_variant(n - 1, start_pos, extra_pos, end_pos, w)
        display_move(start_pos, end_pos, w)
        recursive_towers_variant(n - 1, extra_pos, end_pos, start_pos, w)
```

[12]:
```
def display_move(start_pos, end_pos, w):
    display_towers.__defaults__[end_pos].append(
                                    display_towers.__defaults__[start_pos].pop()
                                                )
    display_towers(w)
```

[13]:
```
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'{i < len(stack) and "-" * (stack[i] * 2 - 1) or "":^{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:

[14]:
```
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)
```

[15]:
```
recursive_towers_display_solution(4)
```

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


  ---
 -----
-------     -


 -----
-------     -       ---


 -----                -
-------              ---


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